EFFICIENT HARDWARE IMPLEMENTATIONS FOR LATTICE-BASED CRYPTOGRAPHY PRIMITIVES

by
AHMET CAN MERT

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Doctor of Philosophy

Sabancı University
May 2021

EFFICIENT HARDWARE IMPLEMENTATIONS FOR LATTICE-BASED CRYPTOGRAPHY PRIMITIVES

Approved by:

Date of Approval: May 21, 2021

ABSTRACT

EFFICIENT HARDWARE IMPLEMENTATIONS FOR LATTICE-BASED CRYPTOGRAPHY PRIMITIVES

AHMET CAN MERT

ELECTRONICS ENGINEERING Ph.D DISSERTATION, MAY 2021

Dissertation Supervisor: Asst. Prof. Erdinç Öztürk

Keywords: Lattice-based Cryptography, Homomorphic Encryption, Post-quantum Cryptography, Hardware Accelerator, FPGA

Lattice-based cryptography has gained a tremendous amount of attention in the last decade due to two main reasons: (*i*) being projected to be resistant against the attacks by quantum computers and (*ii*) enabling homomorphic encryption (HE) which allows arithmetic operations on the encrypted data. Despite its theoretical advantages, it lacks efficient and practical implementations due to its high computational complexity, especially in the context of HE. In this dissertation, our main objective is to design and implement high-performance and efficient hardware solutions for lattice-based cryptosystems.

To that end, we propose a collection of efficient and flexible hardware accelerators for lattice-based HE and post-quantum cryptography (PQC) schemes. Firstly, we present two different hardware architectures for Number Theoretic Transform (NTT) which is one of the most fundamental building blocks of lattice-based cryptography with several optimizations. The proposed architectures are used in a CPU-FPGA framework providing fast communication via PCI Express link to accelerate the encryption and decryption operations of the Brakerski/Fan-Vercauteren (BFV) HE scheme. Secondly, we present a run-time configurable NTT-based polynomial multiplication architecture that supports a set of algorithm parameters frequently used in lattice-based cryptosystems. Thirdly, we design and implement a high-performance hardware architecture that performs the homomorphic multiplication and relinearization operations for the full RNS variant of the BFV HE scheme on FPGA. The proposed architecture outperforms the highly-optimized Microsoft SEAL HE

library by more than an order of magnitude. Fourthly, we design and implement one of the earliest polynomial multiplication architectures of the CRYSTALS-Kyber PQC scheme, which is one of the finalists in NIST's PQC standardization process, for the FPGA platform in the literature. Finally, we investigate two different design methodologies for generating flexible NTT hardware along with a comprehensive analysis. The first method uses a compile-time configurable parametric NTT hardware generator while the second method presents the high-level synthesis approach.

ÖZET

KAFES-TABANLI KRİPTOGRAFİ ÖĞELERİ İÇİN VERİMLİ DONANIM UYGULAMALARI

AHMET CAN MERT

ELEKTRONİK MÜHENDİSLİĞİ DOKTORA TEZİ, MAYIS 2021

Tez Danışmanı: Dr. Öğr. Üyesi Erdinç Öztürk

Anahtar Kelimeler: Kafes-tabanlı Kriptogragi, Homomorfik Şifreleme, Kuantum-sonrası Kriptografi, Hızlandırıcı Donanım, FPGA

Kafes-tabanlı kriptografi, iki ana nedenden dolayı son yıllarda büyük ilgi görmektedir: (*i*) kuantum bilgisayarlar tarafından gerçekleştirilecek saldırılara karşı dirençli olduğunun öngörülmesi ve (*ii*) şifrelenmiş veriler üzerinde aritmetik operasyonlar yapılmasına izin veren homomorfik şifrelemeyi mümkün kılması. Sunduğu teorik avantajlara rağmen, özellikle homomorfik şifreleme bağlamında sahip olduğu yüksek hesaplama karmaşıklığı nedeniyle kafes-tabanlı kriptografik şemaların verimli ve pratik uygulamalarının eksikliği görülmektedir. Bu tezde temel amacımız, kafes-tabanlı kriptografi sistemleri için yüksek performanslı donanım çözümleri tasarlama ve gerçeklemektir.

Bu amaçla, kafes-tabanlı homomorfik şifreleme ve kuantum sonrası kriptografi şemaları için verimli ve esnek donanım hızlandırıcılarını içeren çalışmaları sunmaktayız. İlk olarak, kafes-tabanlı kriptografinin temel yapı taşlarından biri olan Sayılar Teorisi Dönüşümü (NTT) için çeşitli optimizasyonlar içeren iki farklı donanım mimarisi sunuyoruz. Önerilen mimariler, Brakerski/Fan-Vercauteren (BFV) homomorfik şifreleme şemasının şifreleme ve şifre çözme operasyonlarını hızlandırmak için PCI Express bağlantısı ile hızlı iletişim sağlayan bir CPU-FPGA çerçevesinde kullanılmıştır. İkinci olarak, kafes-tabanlı kriptografi sistemlerinde sıklıkla kullanılan bir dizi algoritma parametresini destekleyen ve çalışma-zamanı yapılandırılabilir NTT tabanlı bir polinom çarpıcısı mimarisi sunulmuştur. Üçüncü çalışma olarak, BFV şemasının tam kalıntı sayı sistemi varyantındaki homomorfik çarpma ve yeniden doğrusallaştırma işlemlerini gerçekleştiren yüksek performanslı bir do-

nanım mimarisin tasarımı ve gerçeklemesi sunulmuştur. Önerilen donanım mimarisi, son derece optimize edilmiş Microsoft SEAL homomorfik şifreleme kütüphanesi ile kıyaslandığında on kattan daha fazla performans iyileştirmesi göstermektedir. Dördüncü olarak, NIST tarafından başlatılan kuantum sonrası kriptografi standartlaştırma sürecinin finalistlerinden olan CRYSTALS-Kyber kuantum sonrası kriptografi şemasının polinom çarpıcısı mimarisinin FPGA platformu için ilk örneklerinden biri sunulmuştur. Son olarak, kapsamlı bir analizle birlikte esnek NTT donanımı oluşturmak için iki farklı tasarım yöntemini inceledik. İlk yöntem, derlemezamanında yapılandırılabilir parametrik NTT donanım üretecini kullanırken, ikinci yöntem yüksek düzey sentez tabanlı tasarım yaklaşımını kullanmaktadır.

ACKNOWLEDGEMENTS

I am indebted to many great people for their valuable help and support throughout the years of study for this dissertation. First of all, I would like to thank my supervisor Dr. Erdinç Öztürk for all his guidance, support, and patience throughout my studies. I am grateful to him for providing me the freedom and necessary resources to conduct research. This work would not be complete without his expertise and suggestions. I also want to thank him for always providing guidance and inspiration for my research and future career opportunities.

I also would like to express my sincere gratitude to Dr. Erkay Savaş. I appreciate very much his knowledge and expertise which significantly contributed to my research. I want to thank him for the many great discussions and brainstorming sessions we had on the board, and the countless cups of coffee he bought for me. I always feel privileged as I had change working with him.

I would like to thank Dr. Aydin Aysu for giving me an opportunity to work with him at North Carolina State University (NCSU) during the summer of 2019. He provided me with their unique research environment and helped me broaden my vision. I also want to thank Emre, Furkan, and other members of the HECTOR Lab at NCSU for their support, friendship, and collaboration.

I would like to thank the member of my dissertation committee, Dr. Yaşar Gürbüz, Dr. Sıddıka Berna Örs Yalçın, and Dr. Erdem Alkım for their precious time, effort, and valuable comments on my dissertation.

I want to thank all members of the Cryptography and Information Security Group at Sabanci University for their friendship and collaboration during my studies. I especially want to thank Utku for being a memorable friend and colleague for me with the many conversations and coffee breaks we had.

I want to thank my long-time friends Ozan, Abdurrahman, Ali Eren, Ercan, and many others for their valuable friendship. They made this journey easier for me with their support.

My deepest gratitude goes to my wife Gülizar. This dissertation is dedicated with love to her for her constant support and encouragement for going through my tough periods with me. She was always there for me when I found myself in darkness. I want to thank her for her endless support, encouragement, and patience.

I want to express my gratitude to my parents, my mother Emine and my father Şaban. I would not be the person I am today without their unconditional love and support. I want to thank my parents with all my heart for their love and support.

Finally, I would like to acknowledge Sabanci University and Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting me with scholarships throughout my studies. This dissertation was supported by TÜBİTAK BIDEB 2211-A program.

To my beloved wife Gülizar

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

AES Advanced Encryption Standard

ASIC Application Specific Integrated Circuit

BFV Brakerski/Fan-Vercauteren

BGV Brakerski-Gentry-Vaikuntanathan

BRAM Block RAM

CPU Central Processing Unit

CRT Chinese Remainder Theorem

CVP Closest Vector Problem

DFF D-Flip Flop

DFT Discrete Fourier Transform

DGHV Dijk-Gentry-Halevi-Vaikuntanathan

DS Digital Signature

DSP Digital Signal Processor

ECC Elliptic Curve Cryptography

FHE Fully Homomorphic Encryption

FPGA Field Programmable Gate Array

FT Fourier Transform

GSW Gentry-Sahai-Waters

HE Homomorphic Encryption

HLS High Level Synthesis

IoT Internet of Things

ITU International Communication Union

KEM Key Encapsulation Mechanism

LUT Lookup Table

LWE Learning With Errors

MAC Multiply and Accumulate

NIST National Institute of Standards and Technology

NTT Number Theoretic Transform

NWC Negative Wrapped Convolution

PCIe Peripheral Component Interconnect Express

PHE Partially Homomorphic Encryption

PQC Post-Quantum Cryptography

PWC Positive Wrapped Convolution

RIFFA Reusable Integration Framework for FPGA Accelerators

RNS Residue Number System

RSA Rivest-Shamir-Adleman

SEAL Simple Encrypted Arithmetic Library

SIS Shortest Integer Solution

SoC System on a Chip

SVP Shortest Vector Problem

SWHE Somewhat Homomorphic Encryption

YASHE Yet Another Somewhat Homomorphic Encryption

1. INTRODUCTION

The Internet has a very important place in today's world. According to International Communication Union (ITU), more than half of the world population has access to the internet and actively using it (ITU, 2020). With the advances in the Internet of Things (IoT) devices such as personal computers and smart digital devices, their portion in our daily routine is expected to increase gradually every day. In addition to personal usage, state agencies and private companies greatly benefit from the internet and internet technologies for reaching out to more people. For example, states promote the digital transformation to increase efficiency while the companies advertise online commerce and banking applications for their customers (Kurfalı, Arifoğlu, Tokdemir & Paçin, 2017).

All of these applications have two main requirements: (i) secure communication of the data, (ii) secure storage and process of the data. These applications need to communicate with each other or central authority. The data sent and received over the communication channels are vulnerable to malicious attackers. Therefore, it has great importance for these applications to establish ways for sending and receiving data securely such that the privacy-sensitive information of their users will be protected. Most of the time, these applications also store and process their user data on the cloud services, which need to operate fast and securely on the stored information for real-time applications. Therefore, the design and implementation of fast cryptographic building blocks should be considered as a fundamental requirement for the applications targeting secure and practical cloud applications.

Public key cryptography schemes such as Rivest–Shamir–Adleman (RSA) (Rivest, Shamir & Adleman, 1978) and elliptic curve cryptography (ECC) are based on the hard mathematical problems which are not solvable by modern devices with limited computing power in polynomial time (Bernstein & Lange, 2017). Therefore, these current cryptographic schemes ensure practical security for internet applications. However, quantum computers are conjectured to be powerful enough to break these schemes using Shor's algorithm (Shor, 1994). There are already a tremendous amount of efforts for creating public-key cryptographic schemes secure against the

attacks by the quantum computers, which is referred to as post-quantum cryptography (PQC), initiated by the National Institute of Standards and Technology (NIST) in late 2016 (Chen, Chen, Jordan, Liu, Moody, Peralta, Perlner & Smith-Tone, 2016). Among various mathematical constructions, hard lattice problems have emerged as one of the most promising construction, which is also referred to as lattice-based cryptography. Besides being resistant to attacks by quantum computers, lattice-based cryptography provides a mathematical basis for homomorphic encryption (HE) which allows secure computation by allowing arithmetic computation on the encrypted data. It enables cloud servers to process any privacy-sensitive data without leaking any sensitive information. HE and PQC are two major applications enabled by lattice-based cryptography.

Although lattice-based cryptography provides solutions for the aforementioned problems and enables useful applications, its high computational complexity is the main factor preventing its massive deployment in real-time internet applications. Therefore, hardware accelerators are emerged as perfect candidates to be employed in lattice-based HE and PQC applications. Lattice-based cryptography operates over polynomials. Although arithmetic operations in lattice-based cryptography, especially in the context of HE, have high computational complexity, they involve intrinsically parallelizable parts. However, software implementations cannot take full advantage of this parallelism due to the limitations of single-core and multi-core CPU architectures (Brodtkorb, Dyken, Hagen, Hjelmervik & Storaasli, 2010), which will perform better on FPGA platforms.

Our motivation in this dissertation is to propose efficient hardware solutions which can be utilized as practical accelerators for performance-deprived real-time lattice-based HE and PQC applications. To that end, we focus on one of the most fundamental and time-consuming operations in lattice-based cryptography, multiplication of very large degree polynomials which uses Number Theoretic Transform (NTT). We also focus on the design and implementation of larger arithmetic blocks utilizing polynomial multiplication as a sub-routine such as homomorphic encryption, decryption and multiplication. In hardware/software based heterogeneous systems, the communication cost between the hardware and software is as important as an efficient accelerator design for the employment of hardware accelerators in practical applications. Thus, the communication cost between the accelerator platform and the software should be taken into account as a crucial design consideration. To that end, we also focus on the design and implementation of heterogeneous frameworks which enable efficient communication between the hardware and software.

Lattice-based cryptographic schemes targeting different applications share similar

arithmetic blocks. Yet, they work with a wide range of parameter sets (Nejatollahi, Dutt, Ray, Regazzoni, Banerjee & Cammarota, 2019). Therefore, flexibility is one of the most desired aspects of lattice-based cryptography design. In addition to the arithmetic variations, there are also performance and area requirement variations for different platforms and applications. For example, area-constrained devices favor implementations with low area cost while cloud applications need high-performance implementations at the expense of extra area cost. Similarly, PQC schemes work with small algorithmic parameters and value low-cost designs while HE applications work with larger parameter sets and need high-performance for practical real-time applications (Acar, Aksu, Uluagac & Conti, 2018), (Nejatollahi et al., 2019). Therefore, we also focus on the flexible design methodologies for the main arithmetic blocks of lattice-based cryptography, namely NTT.

1.1 Contribution of the Dissertation

The main objective of this dissertation was to design and implement high-performance and efficient hardware accelerators for lattice-based cryptography primitives. To that end, we propose efficient hardware implementations for the fundamental arithmetic blocks of lattice-based HE and PQC schemes such as multiplication of very large degree polynomials. There are various lattice-based cryptosystems in development nearing massive deployment which will demand fast and flexible methodologies for their building blocks. Thus, we also investigate flexible design methodologies for NTT which is one of the main building blocks utilized in the polynomial multiplication operations of lattice-based cryptosystems. Our contributions in this dissertation are as follows.

- In our first study, we present two fast and scalable NTT-based polynomial multiplier architectures which employ a novel word-level Montgomery reduction algorithm and its hardware realization. We also present a CPU-FPGA framework employing PCI Express (PCIe) link for communication between the FPGA board and the host CPU. The proposed architectures are also utilized to design and implement a unified encryption/decryption architecture for the Brakerski/Fan-Vercauteren (BFV) HE scheme (Fan & Vercauteren, 2012). The proposed encryption/decryption architecture is employed with the proposed framework to accelerate the encryption and decryption operations of the BFV scheme implemented in Microsoft's Simple Encrypted Arithmetic

Library (SEAL) library (Microsoft, 2019). Overall, the proposed work shows up to one order of magnitude performance improvement compared to the pure software implementation in SEAL for the encryption and decryption operations of the BFV scheme. The result of this study is published in (Mert, Öztürk & Savaş, 2019) and (Mert, Öztürk & Savaş, 2020).

- In our second study, we focus on the flexibility requirements of lattice-based cryptosystems and present a *run-time* configurable and highly parallelized NTT-based polynomial multiplier architecture. The proposed architecture supports six different parameter sets (n and $\lceil \log_2(q) \rceil$), which are widely utilized in the lattice-based cryptography. For proof of concept, the proposed architecture is employed in a CPU-FPGA framework utilizing PCIe link to accelerate polynomial multiplication operation performed during the decryption operation of the BFV scheme, showing that it can be used as an actual accelerator in lattice-based cryptosystems. The result of this study is published in (Mert, Öztürk & Savaş, 2020).

- In the third study, we focus on one of the main and most time-consuming homomorphic operations, multiplication of two ciphertexts. We present a high-performance FPGA-based hardware accelerator architecture that performs the homomorphic multiplication and relinearization operations for full residue number system (RNS) variant of BFV HE scheme (Bajard, Eynard, Hasan & Zucca, 2017) for a fixed parameter set with a multiplicative depth of one. The proposed architecture employs an efficient NTT core architecture and an optimized operation scheduling for the homomorphic multiplication and relinearization operations. The proposed architecture shows more than one order of magnitude performance improvement for the homomorphic multiplication and relinearization operations compared to the SEAL library (Microsoft, 2020). Finally, we discuss the scalability of the accelerator for different parameter settings.

- In our fourth study, we focus on accelerating the polynomial multiplication operation of CRYSTALS-Kyber PQC scheme (Bos, Ducas, Kiltz, Lepoint, Lyubashevsky, Schanck, Schwabe, Seiler & Stehlé, 2018) which adopted a variant of NTT-based polynomial multiplication operation by changing its initial parameter set. To that end, we first introduce the CRYSTALS-Kyber scheme and we propose three polynomial multiplier architectures with one, four, and sixteen processing elements with several optimizations for the CRYSTALS-Kyber scheme with a new parameter set. This chapter presents one of the earliest hardware implementations of NTT-based polynomial multiplication for

the CRYSTALS-Kyber's new parameter set. The proposed polynomial multiplier with sixteen processing elements shows up to two orders of magnitude performance improvement compared to the high-speed software implementation on a Cortex-M4 (Alkim, Bilgin, Cenk & Gérard, 2020). The result of this study is published in (Yaman, Mert, Öztürk & Savaş, 2021).

- In our fifth study, we focus on investigating flexible design methodologies of NTT operation for hardware platforms, FPGA devices in particular. We first propose a *compile-time* configurable NTT hardware generator which takes ring size (n), coefficient modulus size ($\lceil \log_2(q) \rceil$) and the number of processing elements as input, and generates the corresponding NTT hardware. The proposed work generates hardware that shows similar or better performance compared to most of the hand-written NTT hardware in the literature. In the second method, we investigate the high-level synthesis (HLS)-based design approach for NTT and we conclude this chapter with a comprehensive analysis. The result of this study is published in (Mert, Karabulut, Öztürk, Savaş, Becchi & Aysu, 2020) and (Mert, Karabulut, Öztürk, Savaş & Aysu, 2020).

1.2 Organization of the Dissertation

In Chapter 2, we present the notation which we follow throughout the dissertation and the preliminary information necessary for a better understanding of this dissertation. The contribution of the dissertation is presented in five chapters. In Chapter 3, we present two fast and scalable NTT-based polynomial multiplier architectures utilized within a CPU-FPGA framework to accelerate the Microsoft SEAL library. In Chapter 4, we present run-time flexible and highly parallelized NTT-based polynomial multiplier architecture. In Chapter 5, we present a high-performance hardware architecture that performs the homomorphic multiplication operation of the BFV scheme. In Chapter 6, we present one of the earliest polynomial multiplication hardware architectures for the CRYSTALS-Kyber PQC scheme. In Chapter 7, we investigate flexible design methods for NTT and present our analysis. Finally, in Chapter 8, we conclude the dissertation and discuss the future work.

2. BACKGROUND

In this section, we first introduce the mathematical notation used throughout this dissertation. Then, technical and arithmetic preliminaries necessary for understanding this dissertation are presented. This dissertation focuses on efficient hardware implementations of lattice-based cryptographic applications. Therefore, a brief introduction to the history and theory of lattice-based cryptography is first presented. Then, two main applications of lattice-based cryptography are explained: (i) PQC and (ii) HE, which are excessively studied in this dissertation. Finally, one of the most fundamental arithmetic operations in lattice-based cryptography, polynomial multiplication, is explained in detail. Then, NTT operation, which is used for efficient implementation of polynomial multiplication operation, is presented and its various implementations are discussed.

2.1 Notation

Let the ring $\mathbb{Z}_q$ represent the set of integers $\{0, 1, \ldots, q-1\}$ where q is a positive integer. The polynomial ring $\boldsymbol{R}_q = \mathbb{Z}_q[x]/\phi(x)$ represents all polynomials reduced with the polynomial $\phi(x)$ where the coefficients of $\phi(x)$ are in $\mathbb{Z}_q$. The polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$ is represented as $\boldsymbol{R}_{q,n}$ when $\phi(x)$ has the form of (x^n+1). In other words, the ring $\boldsymbol{R}_{q,n}$ consists of polynomials of degree at most $(n-1)$ with polynomial coefficients in $\mathbb{Z}_q$. For the rest of the dissertation, q and n represent the coefficient modulus and the degree of the polynomial ring, respectively (which is also referred to as ring size throughout the dissertation). Also, n is assumed to be a power of two if otherwise is not stated.

Throughout the dissertation, we represent an integer, a polynomial, a vector or matrix of polynomials with regular lowercase letter (e.g. $a \in \mathbb{Z}_q$), boldface lowercase letter (e.g. $\boldsymbol{a} \in \boldsymbol{R}_{q,n}$) and boldface uppercase letter (e.g. $\boldsymbol{A} \in \boldsymbol{R}_{q,n}^{m \times k}$), respectively.

A polynomial $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$ in $\boldsymbol{R}_{q,n}$ is also represented as a vector of integers in $\mathbb{Z}_q$, $\boldsymbol{a} = [a_0, a_1, \ldots, a_{n-1}]$, where a_i (equivalently $\boldsymbol{a}_i$ or $\boldsymbol{a}[i]$) represents the integer at i-th position. We use $\boldsymbol{A}[i]$ (or $\boldsymbol{A}_i$) and $\boldsymbol{A}[i][j]$ (or $\boldsymbol{A}_{i,j}$) to represent the polynomial in a vector of polynomials at position i and in a matrix of polynomials at i-th row and j-th column, respectively. Similarly, we use $a[i]$ to represent the bit of integer a at position i.

Vectors (or polynomials) in the NTT domain are represented with a bar over their names. For example, $\boldsymbol{a}$ and $\overline{\boldsymbol{a}}$ represent the polynomial and NTT domain representations of the same vector. Let $\cdot$, $\times$, and $\odot$ represent integer, polynomial, and coefficient-wise vector multiplication, respectively. Let $(\boldsymbol{a} \cdot b)$, $(\boldsymbol{a} + b)$ and $(\boldsymbol{a} - b)$ represent that coefficients of $\boldsymbol{a}$ are multiplied, added and subtracted with integer b, respectively. Let $\lfloor . \rceil$, $\lceil . \rceil$ and $\lfloor . \rfloor$ represent round to nearest integer, rounding up and rounding down operations, respectively. When these operations are performed on a polynomial, they are applied to the coefficients of the polynomial. Let $|.|_q$ (or $[.]_q$) represent modular reduction operation by modulo q for the polynomials and integers.

Let $\boldsymbol{a} \leftarrow \mathbf{S}$ represent that polynomial $\boldsymbol{a}$ is uniformly sampled from the set $\mathbf{S}$. Similarly, if $\mathbf{S}$ is a distribution, $\boldsymbol{a} \leftarrow \mathbf{S}$ represents that polynomial $\boldsymbol{a}$ is sampled from the distribution $\mathbf{S}$. Let $\mathbf{D}_{\mu,\sigma}$ represents the discrete Gaussian distribution where μ and σ represents the center of the distribution and standard deviation, respectively.

2.2 Lattice-based Cryptography

Lattice-based cryptography is the term used to represent any cryptographic construction that is based on the hardness of the lattice problems. Given a set of n linearly-independent m-element vectors $\boldsymbol{B} = \{\boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-1}\}$ where each element of $\boldsymbol{B}$ is an m-element vector of real numbers (i.e. $\boldsymbol{B}_i \in \mathbb{R}^{1 \times m}$), a lattice $\mathcal{L}(\boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-1})$ is defined as shown in Eqn. 2.1, where vectors $\{\boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-1}\}$ are called *basis* of the lattice (Micciancio, 2011).

$$\mathcal{L}(\boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-1}) = \left\{ \sum_{i=0}^{n} c_i \cdot \boldsymbol{B}_i | c_i \in \mathbb{Z} \right\} \tag{2.1}$$

In other words, every lattice point can be considered as a linear combination of basis vectors $\{\boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_{n-1}\}$ with integer coefficients $c_i \in \mathbb{Z}$. As an example, Fig. 2.1

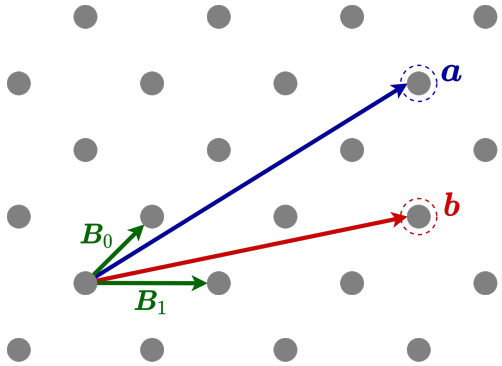


Figure 2.1 Visualization of a Two-Dimensional Lattice Space

depicts a two-dimensional lattice space with two lattice points $\boldsymbol{a}$ and $\boldsymbol{b}$, where green, blue and red lines represent bases ($\boldsymbol{B}_0$, $\boldsymbol{B}_1$), point $\boldsymbol{a}$ and point $\boldsymbol{b}$, respectively.

Although lattices are around for a very long time, they made their appearance in the field of cryptography with Ajtai's work (Ajtai, 1996). The *Shortest Vector Problem* (SVP) and the *Closest Vector Problem* (CVP) are two main standard hard lattice problems with worst-case complexity. The former problem tries to find the shortest non-zero vector in a lattice space while the CVP tries to find the vector which is closest to a given vector $\boldsymbol{r}$ in a lattice space. The *Shortest Integer Solution* (SIS) (Ajtai, 1996) and the *Learning With Errors* (LWE) (Regev, 2009) are two most popular lattice problems in lattice-based cryptography. These problems and their ring variants (R-SIS and R-LWE) enable many lattice-based schemes utilized in HE or PQC applications.

2.3 Post-quantum Cryptography

The security of a digital system relies on the cryptographic protocol it uses. Public key cryptography such as RSA (Rivest et al., 1978) is one of the most widely-used cryptographic protocols in today's digital systems for establishing secure communication. Its security relies on the factorization of a very large composite integer which is a hard task for modern computing devices. Although today's computers and systems are no near breaking the RSA, quantum computers are conjectured to solve these hard mathematical problems, on which RSA relies, in polynomial time, thanks to Shor's quantum algorithm (Shor, 1994), (Proos & Zalka, 2003).

The current state of quantum computers is not mature enough for threatening the

Table 2.1 KEM and DS Schemes in the Final Round of NIST's Post-quantum Standardization

Op.	Const.	Schemes
KEM	L-B	CRYSTALS-Kyber (Bos et al., 2018), Saber (D'Anvers et al., 2018), NTRU (Chen et al., 2019)
	Others	Classic McEliece (Bernstein et al., 2017)
KEM*	L-B	NTRU Prime (Bernstein et al., 2017), FrodoKEM (Alkim et al., 2018)
	Others	BIKE (Aragon et al., 2017), SIKE (Azarderakhsh et al., 2017), HQC (Chen et al., 2016)
DS	L-B	CRYSTALS-Dilithium (Ducas et al., 2018), Falcon (Fouque et al., 2018)
	Others	Rainbow (Beullens, 2020)
DS*	L-B	–
	Others	SPHINCS+ (Bernstein et al., 2019), Picnic (Chase et al., 2017), GeMSS (Chen et al., 2016)

* Alternative schemes

solidity of the current cryptosystems; however, there are various efforts for improving the efficiency of quantum computing. For example, a recent study showed that it is possible to break RSA-2048 in 177 days with 13436 physical qubits (Gouzien & Sangouard, 2021). Besides, future quantum computers can be used to break previously encrypted data which are collected by third parties over communication channels. This urges the immediate use and deployment of PQC in current digital systems.

To be prepared for the time when quantum computers will become powerful and widespread, NIST has started a PQC standardization process in 2016 (Chen et al., 2016). The process aims to publish the post-quantum key encapsulation mechanism (KEM) and digital signature (DS) standards by 2024. The process has started with a total of 69 submissions and there have been three rounds of evaluation. After three rounds, there are seven finalists (four KEMs and three DSs) and eight alternatives (three KEMs and five DSs) candidates, which are listed in Table 2.1. As shown in Table 2.1, lattice-based cryptography is used as the main mathematical construction for the majority of PQC schemes. To be specific, five out of seven finalist candidates and two out of eight alternative candidates are lattice-based cryptosystems.

2.4 Homomorphic Encryption

Encryption is used to preserve sensitive data. Although encryption enables privacy, it comes with certain limitations such that encrypted information needs to be decrypted with a secret key for being used for any operation. This limitation gains more importance for scenarios where the sensitive data needs to be outsourced for storage or computation to third parties. HE allows operation on the data encrypted

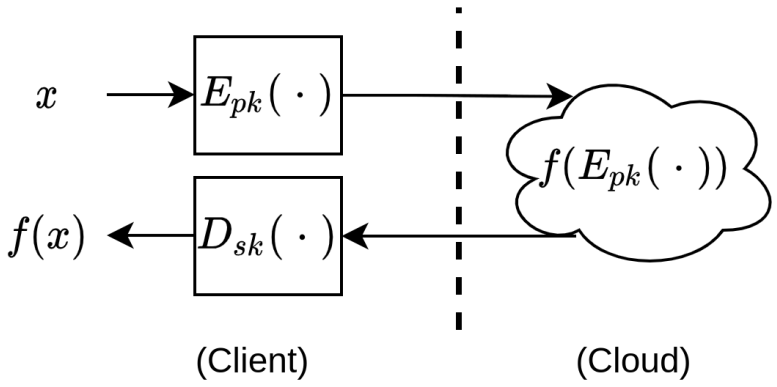


Figure 2.2 A Simple Scenario Utilizing HE

using a HE scheme without needing the decryption operation. Therefore, it enables retrieving useful information from homomorphically encrypted data without revealing any privacy-sensitive information. The idea of HE was first introduced by Rivest *et al.* in the late 1970s (Rivest, Adleman, Dertouzos & others, 1978). An example scenario utilizing HE is illustrated in Fig. 2.2. A client has sensitive data x and wants his data to be processed on a cloud. The client first encrypts his data with homomorphic encryption function $E_{pk}(\cdot)$ which needs a public key and then sends the encrypted data to the cloud. The cloud evaluates the encrypted data homomorphically with $f(\cdot)$ and sends the evaluated data back to the client. Finally, the client uses the homomorphic decryption function $D_{sk}(\cdot)$ which needs the secret key for decryption and obtains the evaluated data $f(x)$ without revealing any information about the data to the cloud.

There are two types of homomorphism, *additive* and *multiplicative*, which are described in Eqn. 2.2 and Eqn. 2.3, respectively. The former and latter supports homomorphic addition and multiplication, respectively.

$$D_{sk}(E_{pk}(a) + E_{pk}(b)) = a + b \tag{2.2}$$

$$D_{sk}(E_{pk}(a) \cdot E_{pk}(b)) = a \cdot b \tag{2.3}$$

There are mainly three types of HE definitions: (*i*) partially HE (PHE), (*ii*) somewhat HE (SWHE) and (*iii*) fully HE (FHE). PHE schemes can perform only one type of homomorphic operation (either *addition* or *multiplication*) on the encrypted data. Some of the well-known PHE schemes are El-Gamal, Paillier, and RSA (Acar et al., 2018). For example, the RSA cryptosystem allows an unlimited number of homomorphic multiplication operations, thus it has the property of multiplicative

homomorphism. SWHE schemes can perform both homomorphic addition and multiplication for a limited number of times (also referred to as *depth*) or a set of circuits. Boneh *et al.* proposed one of the earliest SWHE scheme (Boneh, Goh & Nissim, 2005). FHE schemes allow both homomorphic addition and multiplication for an unlimited number of times and any type of circuit. FHE was an open problem until Craig Gentry proposed the first FHE scheme in 2009 in his Ph.D. dissertation (Gentry, 2009). Gentry's work enables FHE by introducing a computationally complex technique called *bootstrapping* which is a procedure reducing the noise in the ciphertext so more homomorphic operations can be performed without needing a decryption operation.

Gentry's work also motivates researchers and many HE schemes are proposed in a very short time (DGHV (Van Dijk, Gentry, Halevi & Vaikuntanathan, 2010), GSW (Gentry, Sahai & Waters, 2013), YASHE (Bos, Lauter, Loftus & Naehrig, 2013), BGV (Brakerski, Gentry & Vaikuntanathan, 2014)). Some of the works propose SWHE versions of FHE schemes (BFV (Fan & Vercauteren, 2012)) for avoiding complex bootstrapping operation by utilizing a pre-defined circuit depth. Currently, BFV (Fan & Vercauteren, 2012), BGV (Brakerski et al., 2014) and CKKS (Cheon, Kim, Kim & Song, 2017) are leading and most promising HE schemes in the literature. There are also various efforts for providing efficient and practical implementations of various HE schemes (SEAL (Microsoft, 2020), PALISADE (Polyakov, Rohloff & Ryan, 2017), HElib (Halevi & Shoup, 2014), NFLlib (Aguilar-Melchor, Barrier, Guelton, Guinet, Killijian & Lepoint, 2016)).

2.4.1 Brakerski/Fan-Vercauteren Homomorphic Encryption Scheme

BFV (or FV as referred to in a part of the literature) scheme extends and adopts Brakerski's LWE-based fully homomorphic scheme (Brakerski, 2012) to R-LWE setting (Fan & Vercauteren, 2012). It requires multi-precision polynomial arithmetic and sampling of random polynomials. Let the plaintext and ciphertext spaces be $\boldsymbol{R}_{t,n}$ and $\boldsymbol{R}_{q,n}$, respectively, for some integer $t > 1$, where neither q nor t has to be prime. The operations of the BFV scheme are shown below where $\Delta = \lfloor q/t \rfloor$, χ represents a discrete Gaussian distribution with proper parameters based on the security level of the scheme, T represents decomposition base used in relinarization, and ℓ represents the number of relinearization keys.

- `BFV.KeyGen()`: $\boldsymbol{s} \leftarrow \boldsymbol{R}_{2,n}$, $\boldsymbol{a} \leftarrow \boldsymbol{R}_{q,n}$ and $\boldsymbol{e} \leftarrow \chi$,

$$\texttt{sk} = \boldsymbol{s},\ \texttt{pk} = (\boldsymbol{p}_0, \boldsymbol{p}_1) = ([-(\boldsymbol{a} \times \boldsymbol{s} + \boldsymbol{e})]_q, \boldsymbol{a}).$$

- $\texttt{BFV.RelinKeyGen}(\texttt{sk}, T, \ell)$: $\texttt{sk} = \boldsymbol{s} \in \boldsymbol{R}_{2,n}$, $\boldsymbol{a}_i \leftarrow \boldsymbol{R}_{q,n}$, $\boldsymbol{e_i} \leftarrow \chi$ for $i = 1, \ldots, \ell$,

$$\texttt{rlk} = ([-(\boldsymbol{a}_i \times \boldsymbol{s} + \boldsymbol{e}_i)]_q, \boldsymbol{a}_i) \text{ for } i = 1, \ldots, \ell.$$

- $\texttt{BFV.Enc}(\texttt{pk}, \boldsymbol{m})$: $\boldsymbol{m} \in \boldsymbol{R}_{t,n}$, $\texttt{pk} = (\boldsymbol{p}_0, \boldsymbol{p}_1) \in \boldsymbol{R}_{q,n}^{1\times 2}$, $\boldsymbol{u} \leftarrow \boldsymbol{R}_{2,n}$ and $\boldsymbol{e}_1, \boldsymbol{e}_2 \leftarrow \chi$,

$$\texttt{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) = ([\boldsymbol{m} \cdot \Delta + \boldsymbol{p}_0 \times \boldsymbol{u} + \boldsymbol{e}_1]_q, [\boldsymbol{p}_1 \times \boldsymbol{u} + \boldsymbol{e}_2]_q).$$

- $\texttt{BFV.Dec(sk,ct)}$: $\texttt{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1) \in \boldsymbol{R}_{q,n}^{1\times 2}$ and $\texttt{sk} = \boldsymbol{s} \in \boldsymbol{R}_{2,n}$,

$$\boldsymbol{m} = [\lfloor \tfrac{t}{q} [\boldsymbol{c}_0 + \boldsymbol{c}_1 \times \boldsymbol{s}]_q \rceil]_t.$$

- $\texttt{BFV.Add(ct0,ct1)}$: $\texttt{ct0} = (\boldsymbol{c}_{00}, \boldsymbol{c}_{01}) \in \boldsymbol{R}_{q,n}^{1\times 2}$ and $\texttt{ct1} = (\boldsymbol{c}_{10}, \boldsymbol{c}_{11}) \in \boldsymbol{R}_{q,n}^{1\times 2}$,

$$\texttt{ct} = ([\boldsymbol{c}_{00} + \boldsymbol{c}_{10}]_q, [\boldsymbol{c}_{01} + \boldsymbol{c}_{11}]_q).$$

- $\texttt{BFV.Mul(ct0,ct1)}$: $\texttt{ct0} = (\boldsymbol{c}_{00}, \boldsymbol{c}_{01}) \in \boldsymbol{R}_{q,n}^{1\times 2}$ and $\texttt{ct1} = (\boldsymbol{c}_{10}, \boldsymbol{c}_{11}) \in \boldsymbol{R}_{q,n}^{1\times 2}$,

$$\texttt{ct} = ([\lfloor \tfrac{t\cdot(\boldsymbol{c}_{00}\times\boldsymbol{c}_{10})}{q} \rceil]_q, [\lfloor \tfrac{t\cdot(\boldsymbol{c}_{00}\times\boldsymbol{c}_{11}+\boldsymbol{c}_{01}\times\boldsymbol{c}_{10})}{q} \rceil]_q, [\lfloor \tfrac{t\cdot(\boldsymbol{c}_{01}\times\boldsymbol{c}_{11})}{q} \rceil]_q).$$

- $\texttt{BFV.Relin(ct,rlk)}$: $\texttt{ct} = (\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2) \in \boldsymbol{R}_{q,n}^{1\times 3}$ and $\texttt{rlk} \in \boldsymbol{R}_{q,n}^{2\times \ell}$,

$$\text{rewrite } \boldsymbol{c}_2 \text{ as } \textstyle\sum_{i=0}^{\ell-1} \widetilde{\boldsymbol{c}}_2 \cdot T^i \text{ where } \widetilde{\boldsymbol{c}}_2 \in \boldsymbol{R}_{T,n},$$

$$\texttt{ct} = ([\boldsymbol{c}_0 + \textstyle\sum_{i=0}^{\ell-1} \texttt{rlk}[i][0] \times \widetilde{\boldsymbol{c}}_2]_q, [\boldsymbol{c}_1 + \textstyle\sum_{i=0}^{\ell-1} \texttt{rlk}[i][1] \times \widetilde{\boldsymbol{c}}_2]_q).$$

Textbook BFV scheme requires high precision integer arithmetic and computationally expensive divide-and-round operation which creates a bottleneck for efficient implementations when the coefficient modulus q is very large. Therefore, there have been various efforts in the literature (Bajard et al., 2017), (Halevi, Polyakov & Shoup, 2019), (Bajard, Eynard, Martins, Sousa & Zucca, 2019), (Takeshita, Schoenbauer, Karl & Jung, 2020) for improving the practicality of the BFV scheme by adopting RNS into the BFV scheme.

2.4.2 Microsoft SEAL Homomorphic Encryption Library

Microsoft SEAL (Microsoft, 2020) is a highly optimized HE library developed by the Cryptography Research Group at Microsoft Research. It enables fast and efficient homomorphic applications ranging from private information retrieval (Angel, Chen, Laine & Setty, 2018) to secure neural network inference (Brutzkus, Gilad-Bachrach & Elisha, 2019). Recently, the SEAL library is utilized for implementing

a secure password monitoring system in Microsoft's web browser Edge (Kannepalli, Laine & Moreno, 2021). The SEAL library provides support for two widely-known HE schemes, the BFV and the CKKS, for implementing homomorphic operations. BFV works with integers while CKKS enables homomorphic arithmetic using real numbers.

SEAL library uses the full RNS variant of the BFV scheme proposed by Bajard *et al.* (Bajard et al., 2017) and implements homomorphic operations slightly different than the textbook BFV. In RNS implementation, high precision polynomial arithmetic can be divided into smaller low precision integer arithmetic and can be performed in parallel. Also, division and rounding operations are eliminated. As Microsoft SEAL library is in an ongoing development phase, we used different versions of the library throughout the dissertation (namely `v3.2.0` (Microsoft, 2019) and `v3.5.1` (Microsoft, 2020)).

2.4.3 Residue Number System

The multiplicative depth of a HE system is determined by its parameters, n and q. For example, in (Sinha Roy, Turan, Jarvinen, Vercauteren & Verbauwhede, 2019), the authors implement a private information retrieval application with a multiplicative depth of four using parameters $n = 4096$ and $\lceil \log_2(q) \rceil = 180$. A larger multiplicative depth will result in larger parameters (n and q) and more complex arithmetic. To avoid costly multi-precision integer arithmetic when the coefficient modulus q is very large, RNS is frequently employed in the efficient implementations of HE schemes. It enables algorithmic parallelisms and improves the performance of the implementation (Ozerk, Elgezen, Mert, Öztürk & Savas, 2021). RNS approach employs a set of coprime moduli q_i such that

$$q = \prod_{i=0}^{r-1} q_i,$$

where r is the number of small moduli used. The RNS maps a large arithmetic operation in $\mathbb{Z}_q$ to many smaller operations in $\mathbb{Z}_{q_i}$, which can be performed in parallel. For example, an arithmetic operation with 93-bit modulus q can be performed using three 31-bit smaller moduli q_0, q_1 and q_2, In RNS arithmetic, a large integer (i.e. a) in modulus q needs to be converted into smaller integers in moduli q_i (i.e. $a_i = a \bmod q_i$). Similarly, the integers in moduli q_i are used to construct the integer in modulus q using the Chinese Remainder Theorem (CRT) (Boneh & others, 1999)

shown in Eqn. 2.4, where $M_i = (q/q_i)$ and $m_i = M_i^{-1} \pmod{q_i}$ for $i = 0, \ldots, r-1$.

$$a = \sum_{i=0}^{r-1} a_i \cdot M_i \cdot m_i \pmod{q} \tag{2.4}$$

2.5 Polynomial Multiplication

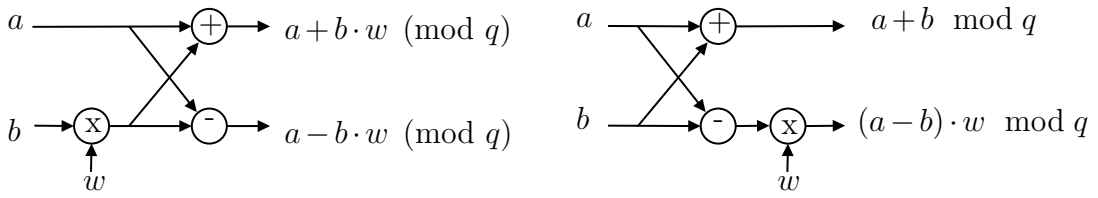
Lattice-based cryptography operates with polynomial rings and polynomial multiplication is one of the core operations. Polynomial multiplication is a well-known bottleneck for creating efficient lattice-based cryptosystems. Efficient implementation of polynomial multiplication has been widely studied and there are different arithmetic tools utilized for a long time such as schoolbook, Toom-Cook (Toom, 1963) or Karatsuba (Karatsuba & Ofman, 1962). NTT-based polynomial multiplication has emerged as one of the most powerful tools for implementing efficient polynomial multiplication operations in lattice-based cryptosystems (Göttert, Feller, Schneider, Buchmann & Huss, 2012). In the following subsections, we will explain NTT and NTT-based polynomial multiplication.

2.5.1 Number Theoretic Transform

The NTT reduces the $\mathcal{O}(n^2)$ complexity of the schoolbook polynomial multiplication to the quasi-linear complexity of $\mathcal{O}(n \cdot \log n)$. NTT is thus a major building block of most lattice-based cryptography implementations. NTT is defined as Discrete Fourier Transform (DFT) over the ring $\mathbb{Z}_q$. An n-point (pt) NTT operation takes an n-element vector $\boldsymbol{a}$ as input and generates another n-element vector $\overline{\boldsymbol{a}}$ using the operation shown in Eqn. 2.5.

$$\bar{\boldsymbol{a}}_i = \sum_{j=0}^{n-1} a_j \cdot \omega^{ij} \pmod{q} \text{ for } i = 0, 1, \ldots, n-1. \tag{2.5}$$

The NTT calculations use the constant, $\omega \in \mathbb{Z}_q$, called n-th root of unity, which is also defined as *primitive root* or *twiddle factor*. The twiddle factor should satisfy the conditions $\omega^n \equiv 1 \pmod{q}$ and $\omega^i \neq 1 \pmod{q}$ $\forall i < n$, where $q \equiv 1 \pmod{n}$, for ensuring the existence of NTT operation. The INTT operation uses a similar



(a) CT Butterfly (b) GS Butterfly

Figure 2.3 Butterfly Structures

formulation as the NTT operation as shown in Eqn. 2.6 except that $\omega^{-1} \pmod q$, which is the modular inverse of ω in $\mathbb{Z}_q$, is used instead of ω and the resulting coefficients of the INTT operation are multiplied with $n^{-1} \pmod q$ in $\mathbb{Z}_q$.

$$\boldsymbol{a}_i = \frac{1}{n} \sum_{j=0}^{n-1} \bar{a}_j \cdot \omega^{-ij} \pmod q \text{ for } i = 0, 1, \ldots, n-1. \tag{2.6}$$

Applying NTT and INTT operations as defined in Eqn. 2.5 and Eqn. 2.6 leads to inefficient implementations. The impact of NTT for lattice-based cryptosystems can be similar to that of the Fourier Transform (FT) for signal processing. There is a family of algorithms, called *Fast Fourier Transform*, for efficiently implementing FT and DFT operations. Therefore, any efficient algorithm for implementing FT and DFT operations can be adapted to perform an NTT operation (Ozerk et al., 2021). The most of the efficient FFT algorithms are constructed around two very well-known approaches: radix-2 iterative *decimation-in-time* (DIT) and *decimation-in-frequency* (DIF) FFTs, which use Cooley-Tukey (CT) and Gentleman-Sande (GS) butterfly structures, respectively (Chu & George, 1999). CT butterfly structure takes three inputs a, b and w, then it produces two outputs $a - b \cdot w \pmod q$ and $a + b \cdot w \pmod q$ which require one modular addition, one modular subtraction and one modular multiplication. Similarly, GS butterfly takes three inputs and produces $a + b \pmod q$ and $(a - b) \cdot w \pmod q$. CT and GS butterfly operations are visualized in Fig. 2.3 (a) and Fig. 2.3 (b), respectively.

In-place FFT operations may changes the order of input vector coefficients. For example, DIF FFT operation can transform an input vector with naturally ordered coefficients $[a_0, a_1, \ldots, a_{n-1}]$ into an output vector with bit-reversed coefficients $[a_{br(0,\log_2 n)}, a_{br(1,\log_2 n)}, \ldots, a_{br(n-1,\log_2 n)}]$ where $br(a, \ell)$ represents bit-reversal operation of an ℓ-bit integer a. It is possible to derive different versions of DIT and DIF FFT algorithms as explained in (Chu & George, 1999): (i) naturally ordered input and bit-reversed output (NR), (ii) naturally ordered input and naturally ordered output (NN) and (iii) bit-reversed input and naturally ordered output (RN) (namely, DIT_{RN}, DIT_{NN}, DIT_{NR}, DIF_{RN}, DIF_{NN} and DIF_{NR}). Aforementioned

Algorithm 1 Iterative Version of In-Place DIF NTT Algorithm (Chu & George, 1999)

Input: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in natural order
Input: primitive n-th root of unity $\omega \in \mathbb{Z}_q$, $n = 2^l$
Output: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order

for i from 1 by 1 to l **do**
  $m = 2^{l-i}$
  for j from 0 by 1 to $2^{i-1} - 1$ **do**
    for k from 0 by 1 to $m - 1$ **do**
      $U = \boldsymbol{a}[2 \cdot j \cdot m + k]$
      $V = \boldsymbol{a}[2 \cdot j \cdot m + k + m]$
      $N0 = (U + V) \pmod{q}$
      $N1 = (U - V) \cdot \omega^{2^{i-1}k} \cdot \pmod{q}$
      $\boldsymbol{a}[2 \cdot j \cdot m + k] = N0$
      $\boldsymbol{a}[2 \cdot j \cdot m + k + m] = N1$
    end for
  end for
end for
return $\boldsymbol{a}$

FFT algorithms can also be utilized to perform inverse FFT (IFFT) algorithms. DIF_{NR} FFT and DIF_{RN} IFFT algorithms are shown in Algorithm 1 and 2, respectively. For the rest of the dissertation, we will use NTT instead of FFT.

Besides, there are also various efficient NTT algorithms in the literature using the same core operations (CT or GS butterfly); but adapted for different platforms and applications. For example, GPU implementations use Four-step NTT algorithm (Dai & Sunar, 2015) for memory advantages while ASIC implementations favour Pease FFT (Pease, 1968) approach which enables an NTT algorithm with constant dataflow. This approach leads to the utilization of single-port SRAM memories due to its regular memory access pattern (Banerjee, Ukyab & Chandrakasan, 2019). Motivated readers can further look into the works (Feng, Li & Xu, 2019), (Lyubashevsky & Seiler, 2019), (Chung, Hwang, Kannwischer, Seiler, Shih & Yang, 2021).

2.5.2 NTT-based Polynomial Multiplication

The schoolbook polynomial multiplication operation computes the multiplication of polynomials $\boldsymbol{a}(x)$ and $\boldsymbol{b}(x)$ as shown in Eqn. 2.7. When the polynomial multiplication operation is performed in $\boldsymbol{R}_q$, the resulting polynomial $\boldsymbol{d}(x)$ should be reduced

Algorithm 2 Iterative Version of In-Place DIF INTT Algorithm (Chu & George, 1999)

Input: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order
Input: modular inverse of primitive n-th root of unity $\omega^{-1} \in \mathbb{Z}_q$, $n = 2^l$
Output: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in natural order

$m, v = 1, n$
while $v > 1$ **do**
  for i from 0 by 1 to $(m-1)$ **do**
    $k = 0$
    for j from i by $2 \cdot m$ to $(n-2)$ **do**
      $U = \overline{\boldsymbol{a}}[j]$
      $V = \overline{\boldsymbol{a}}[j+m]$
      $N0 = (U+V) \pmod q$
      $N1 = (U-V) \cdot \omega^{-\mathtt{br}(k,l-1)} \pmod q$
      $\overline{\boldsymbol{a}}[j] = N0$
      $\overline{\boldsymbol{a}}[j+m] = N1$
      $k = k+1$
    end for
  end for
  $m, v = 2 \cdot m, v/2$
end while
for i from 0 by 1 to $(n-1)$ **do**
  $\overline{\boldsymbol{a}}[i] \leftarrow \overline{\boldsymbol{a}}[i] \cdot n^{-1} \pmod q$
end for
return $\overline{\boldsymbol{a}}$

by $\phi(x)$ after the multiplication operation as shown in Eqn. 2.8.

$$\boldsymbol{d}(x) = \boldsymbol{a}(x) \times \boldsymbol{b}(x) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} a_i \cdot b_j \cdot x^{i+j} \tag{2.7}$$

$$\boldsymbol{c}(x) = \boldsymbol{d}(x) \bmod \phi(x) \tag{2.8}$$

According to convolution theory (Winkler, 2012), NTT and INTT operations can be used to convert schoolbook polynomial multiplication operation into much simpler coefficient-wise multiplication operation of two vectors as shown in Eqn. 2.9. The $\mathtt{NTT}_{2n}$, $\mathtt{INTT}_{2n}$ and $\mathtt{zp}_{2n}$ represent $2n$-pt NTT, $2n$-pt INTT and zero-padding of an n-element vector into $2n$-element, respectively. This tweak reduces the number of modular multiplication from n^2 to $2n \cdot \log_2(2n) + 2n$. However, this approach requires n element input polynomials to be zero-padded to $2n$ elements, $2n$-pt NTT/INTT operations and a separate polynomial reduction by $\phi(x)$ after the INTT operation.

$$\boldsymbol{c}(x) = \mathtt{INTT}_{2n}(\mathtt{NTT}_{2n}(\mathtt{zp}_{2n}(\boldsymbol{a}(x))) \odot \mathtt{NTT}_{2n}(\mathtt{zp}_{2n}(\boldsymbol{b}(x)))) \bmod \phi(x) \tag{2.9}$$

When the irreducible polynomial $\phi(x)$ has the form of $x^n - 1$, a technique called *positive wrapped convolution* (PWC) can be utilized to further reduce the computational complexity of polynomial multiplication over polynomial rings as shown in Eqn. 2.10.

$$\hat{\boldsymbol{a}}(x) = [a_0, a_1, \ldots, a_{n-1}] \odot [\psi^0, \psi^1, \ldots, \psi^{(n-1)}] \tag{2.10}$$

When the irreducible polynomial $\phi(x)$ has the form of $x^n + 1$ (corresponding to the polynomial ring $\boldsymbol{R}_{q,n}$) and $q \equiv 1 \pmod{2n}$, a technique called *negative wrapped convolution* (NWC) can be utilized as shown in Eqns. 2.11-2.14.

$$\hat{\boldsymbol{a}}(x) = [a_0, a_1, \ldots, a_{n-1}] \odot [\psi^0, \psi^1, \ldots, \psi^{(n-1)}] \tag{2.11}$$

$$\hat{\boldsymbol{b}}(x) = [b_0, b_1, \ldots, b_{n-1}] \odot [\psi^0, \psi^1, \ldots, \psi^{(n-1)}] \tag{2.12}$$

$$\hat{\boldsymbol{c}}(x) = \texttt{INTT}_n(\texttt{NTT}_n(\hat{\boldsymbol{a}}(x)) \odot \texttt{NTT}_n(\hat{\boldsymbol{b}}(x))) \tag{2.13}$$

$$\boldsymbol{c}(x) = [\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}] \odot [\psi^0, \psi^{-1}, \ldots, \psi^{-(n-1)}] \tag{2.14}$$

This technique eliminates the necessity of doubling input polynomials and the polynomial reduction after INTT operation. However, it requires additional pre-processing and post-processing operations which are referred to as the multiplication of the coefficients of input and output polynomials with $[\psi^0, \psi^1, \ldots, \psi^{(n-1)}]$ and $[\psi^0, \psi^{-1}, \ldots, \psi^{-(n-1)}]$, respectively. The constant ψ is called $2n$-th root of unity and it should satisfy the conditions $\psi^{2n} \equiv 1 \pmod{q}$ and $\psi^i \neq 1 \pmod{q}$ $\forall i < 2n$, where $q \equiv 1 \pmod{2n}$. The polynomial multiplication over $\boldsymbol{R}_{q,n}$ with NWC requires $n \cdot \log_2(n) + 3n$ modular multiplication operations. It should be noted that the multiplication with $n^{-1} \pmod{q}$ after INTT can be merged with post-processing operation (Pöppelmann, Oder & Güneysu, 2015).

Roy *et al.* (Sinha Roy, Vercauteren, Mentens, Chen & Verbauwhede, 2014) further improved the polynomial multiplication with NWC by merging pre-processing and NTT operations, which is achieved by employing DIT NTT operation utilizing CT butterfly structure. However, this technique does not work for INTT operation. This approach requires $n \cdot \log_2(n) + 2n$ modular multiplication operations and we

refer to this NTT technique as merged NTT (MNTT) for the rest of the dissertation. Similarly, Pöppelmann *et al.* (Pöppelmann et al., 2015) merged INTT and post-processing operations for the polynomial multiplication with NWC by employing DIF NTT operation utilizing the GS butterfly. However, this technique does not work for NTT operation. Although this approach eliminates post-processing operation, it still requires $n \cdot \log_2(n) + 3n$ modular multiplication operations due to the final multiplication with $n^{-1} \pmod{q}$ after INTT. We refer to this INTT technique as merged INTT (MINTT) for the rest of the dissertation.

Employing both techniques and using two different butterfly structures, pre-processing and post-processing operations during the polynomial multiplication operation can be eliminated as shown in Eqn. 2.15, where $\texttt{MNTT}_n$ and $\texttt{MINTT}_n$ represent n-pt MNTT and MINTT operations, respectively. This approach requires $n \cdot \log_2(n) + 2n$ modular multiplication operations.

$$\boldsymbol{c} = \texttt{MINTT}_n(\texttt{MNTT}_n(\boldsymbol{a}(x)) \odot \texttt{MNTT}_n(\boldsymbol{b}(x))) \tag{2.15}$$

MNTT and MINTT techniques eliminate the post-processing and pre-processing operations. However, the multiplication with $n^{-1} \pmod{q}$ after INTT operation, which requires n modular multiplication, cannot be eliminated. Zhang *et al.* (Zhang, Yang, Chen, Yin, Wei & Liu, 2020) proposed a technique to eliminate the multiplication with $n^{-1} \pmod{q}$ while employing MNTT and MINTT operations. This approach multiplies the resulting coefficients with $2^{-1} \pmod{q}$ after each INTT stage, which can be performed using only addition and shift operations, instead of performing the multiplication with $n^{-1} \pmod{q}$ after INTT.

3. FAST AND SCALABLE NTT-BASED POLYNOMIAL MULTIPLICATION ARCHITECTURES

In this chapter, we first present two different highly-parallelized and scalable NTT-based multiplier architectures realizing two different NTT algorithms, namely DIF_{NR}/DIF_{RN} NTT/INTT and Four-step NTT/INTT Algorithms, respectively, for the parameter set $n = 1024$ and $\lceil \log_2(q) \rceil = 32$. Then, we present two hardware architectures optimized for accelerating the encryption and decryption operations of the BFV HE scheme, which are used in the client-side of the HE applications. The hardware architectures employ the aforementioned high-performance polynomial multipliers. For proof of concept, we utilize our architectures in a CPU-FPGA accelerator framework, in which encryption and decryption operations are offloaded to an FPGA device while the rest of operations in the BFV scheme are executed in software running on an off-the-shelf desktop computer. Specifically, our accelerator framework is optimized to accelerate the SEAL library. The hardware part of the proposed framework targets the Xilinx Virtex-7 FPGA device, which communicates with its software part via a PCIe connection. For proof of concept, we implemented our designs targeting parameters $n = 1024$, $\lceil \log_2(q) \rceil = 32$ and $t = 256$ with 128-bit security level. The proposed framework achieves almost 12× and 7× latency speedups including input/output (I/O) operations for the offloaded encryption and decryption operations, respectively, compared to their pure software implementations in the SEAL library. [1]

3.1 Introduction

Although theoretically sound, FHE schemes are not quite ready to be deployed for practical applications due to the performance limitations of computer architec-

[1] This chapter presents the works in (Mert et al., 2019) and (Mert et al., 2020).

tures. Applications based on current FHE schemes, which require efficient implementations of computationally expensive mathematical operations, can be orders of magnitude slower than conventional software applications that operate on plaintext data. Most FHE schemes involve a combination of intrinsically serial and highly parallelizable algorithms that will ultimately perform best on heterogeneous architectures (Brodtkorb et al., 2010), which refers to the use of different processing cores to maximize performance. In this work, we propose such a heterogeneous accelerator framework featuring an FPGA core and a CPU to improve the performance of FHE schemes on a system level.

There is still ongoing research and race to improve the performance of arithmetic building blocks of the working FHE schemes. With similar motivation, in this chapter, we aim to obtain a framework to accelerate the BFV scheme. We focus on improving the FPGA performance of the most time-consuming arithmetic building block of many FHE schemes in the literature: large degree polynomial multiplication. The framework running on our heterogeneous architecture offloads not only polynomial multiplications but also entire encryption and decryption operations onto the FPGA core in order to minimize the communication cost between the FPGA core and the CPU. Our accelerator framework, while offloading highly parallelizable encryption and decryption operations entirely on the FPGA core, leaves the rest of the operations of SEAL intact in software. By deploying our framework, any cloud architecture utilizing SEAL for FHE applications can improve its performance by utilizing an FPGA device next to the CPU, without having to implement the entire FHE library in the FPGA.

The rest of the chapter is organized as follows. Section 3.2 presents two different NTT and NTT-based polynomial multiplier architectures. Section 3.3 presents two different hardware architectures that perform encryption and decryption operations of the BFV scheme. Section 3.4 presents the proposed framework. Finally, Section 3.5 presents the implementation results and the comparison with the literature, and Section 3.6 concludes the chapter.

3.2 NTT Architectures

Here, we present our NTT architectures implementing the radix-2 DIF_{NR}/DIF_{RN} (Chu & George, 1999) and the four-step Cooley-Tukey NTT/INTT

algorithms (Cooley & Tukey, 1965), (Dai & Sunar, 2015). They are shortly referred to as the iterative NTT hardware and the four-step NTT hardware, respectively. We first introduce common modular arithmetic units used by both architectures. Then, we present the remaining arithmetic units and overall design for the architectures.

For NTT-based polynomial multiplication operation, we do not employ the merged NTT/INTT approach since we utilize only a single type of butterfly structure, GS. Instead, we use NWC technique with pre- and post-processing operations as shown in Eqn. 2.11- 2.14. In our implementation, it should be noted that the multiplication operation with n^{-1} at the end of the INTT operation is merged with the post-processing operation.

3.2.1 Modular Adder and Modular Subtractor Units

NTT arithmetic involves a large amount of modular addition and subtraction operations. There are various approaches for efficient implementation of modular arithmetic operations such as employing *lazy reduction* technique discussed in (Yanik, Savas & Koc, 2002). This approach, for a k-bit modulus q, operates on the numbers in the range $[0, 2^k)$. However, in this work, our modular arithmetic units compute numbers in the range $[0, q)$. Constant-time modular addition and modular subtraction operations are shown in Algorithm 3 and Algorithm 4, respectively.

Algorithm 3 Modular Addition Algorithm

Input: $A, B \in \mathbb{Z}_q$
Input: q (k-bit modulus)
Output: $C \equiv A + B \pmod{q} \in \mathbb{Z}_q$
1: $T1 = A + B$
2: $T2 = T1 - q$
3: **case**$(T2[k])$
4: 0: $C = T2$
5: 1: $C = T1$
6: **endcase**

Algorithm 4 Modular Subtraction Algorithm

Input: $A, B \in \mathbb{Z}_q$
Input: q (k-bit modulus)
Output: $C \equiv A - B \pmod{q} \in \mathbb{Z}_q$
1: $T1 = A - B$
2: $T2 = T1 + q$
3: **case**$(T1[k])$
4: 0: $C = T1$
5: 1: $C = T2$
6: **endcase**

3.2.2 Modular Multiplier Unit

For an optimized polynomial multiplier architecture, a fast and efficient modular multiplier needs to be designed and utilized. A modular multiplication operation consists of two parts: (*i*) integer multiplication and (*ii*) modular reduction. The former is straightforward and easy to implement; on the other hand, the latter has high computational complexity. There are mainly two approaches in the literature for efficient modular reduction implementation: (*i*) Barrett reduction (Barrett, 1986) and (*ii*) Montgomery reduction (Montgomery, 1985). Besides, there are also modular reduction approaches for pre-determined special moduli leveraging special forms of modulus (Dai & Sunar, 2015). In this work, we propose and design a novel modular multiplier utilizing word-level Montgomery reduction technique with a lazy reduction approach as explained in (Yanik et al., 2002). Our modular multiplier architecture works for any modulus with a bit length between 22 and 32.

3.2.2.1 Integer Multiplier Unit

We design a 32-bit multiplier with four DSP blocks and an adder tree. Since we are targeting FPGA architecture, we used 16-bit core multipliers, because of the DSP size limitations of Xilinx Series-6 and Series-7 FPGAs. On Spartan-6 Architectures, DSP slices include 18-bit signed multipliers and on Virtex-7 Architectures, DSP slices include 18-bit $\times$ 25-bit signed multipliers. To follow literature, we chose to implement our multiplier for both architectures, therefore we picked a core multiplier length of 16 bits. Each input of the multiplier is divided into 16-bit pieces and one DSP block is used for each 16-bit$\times$16-bit multiplication operation. The resulting intermediate values are then added up using an adder tree. Our integer multiplier is fully pipelined, therefore the 32-bit multiplier has pipeline registers between DSP blocks and adder tree. These pipeline registers do not affect the throughput of the overall architecture in terms of clock cycles, improving the overall performance in terms of execution time significantly. The proposed integer multiplier architecture is depicted in Fig. 3.1, where each dot represents one bit.

Adder tree sums up two 32-bit and one 48-bit integers as shown in step#3 and step#4 of Fig. 3.1. In our design, we used a carry-save adder (shown with a red box in Fig. 3.1) for reducing three 32-bit integers to two 32-bit integers. Finally, we used one 48-bit adder (shown with a green box in Fig. 3.1) to calculate the final result. The proposed integer multiplier has 2 clock cycles latency, it can produce

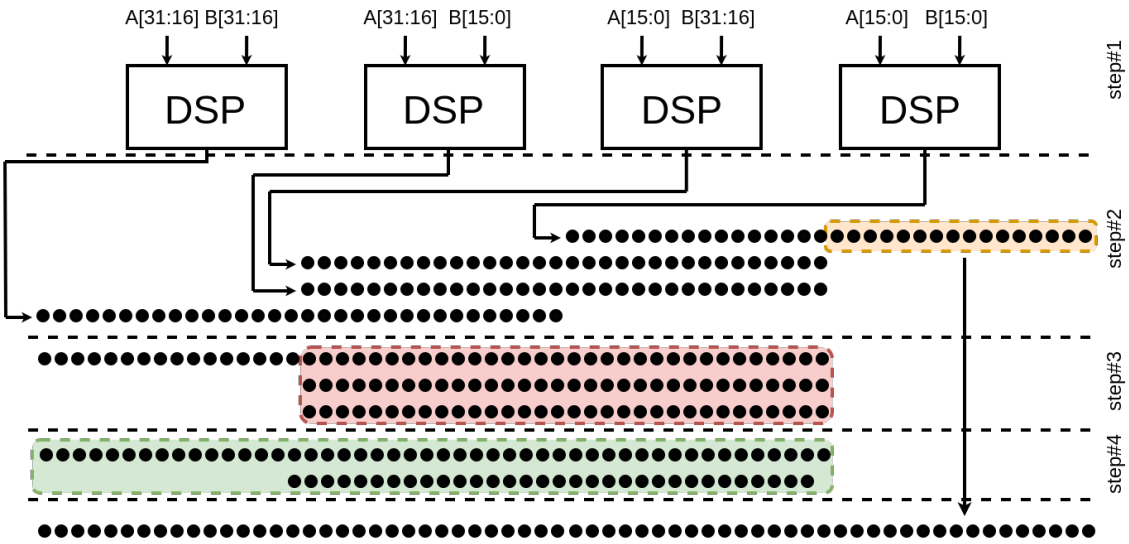


Figure 3.1 32-bit Integer Multiplier Unit

one multiplication result per clock cycle after filling the pipeline and it works for input operands with bit length between 1 and 16.

3.2.2.2 Word-level Montgomery Modular Reduction Unit

Regular Montgomery reduction algorithm is shown in Algorithm 5. It requires two multiplications, one addition and a final correction operation. Montgomery reduction algorithm takes $A \cdot B$ as input and calculates the output $A \cdot B \cdot R^{-1} \pmod{q}$, where $R = 2^k$ is defined as Montgomery reduction residual. The residual has to be corrected using an extra multiplication with R to obtain $A \cdot B \pmod{q}$. It also utilizes μ, the Montgomery reduction variable.

Montgomery reduction algorithm can also be performed by dividing the reduction operation into smaller parts, namely pre-defined word-sized pieces, instead of performing it all at once. We modified the regular Montgomery reduction algorithm in that way to propose a fast and efficient modular reduction algorithm. Besides, we utilized the property of $q \equiv 1 \pmod{2n}$, which should be satisfied by any NTT-friendly prime using *negative wrapped convolution* technique. Any NTT-friendly prime q can be written as shown in Eqn. 3.1. If we select the word size, w, for the reduction operation as $\log_2(2n)$, then the modulus q can be written as shown in Eqn. 3.2.

$$q = q_H \cdot 2^{\log_2(2n)} + 1 \tag{3.1}$$

Algorithm 5 Montgomery Reduction Algorithm

Input: $D = A \cdot B$ (a $2k$-bit positive integer)
Input: q (modulus, a k-bit positive integer)
Input: $\mu = -q^{-1} \pmod{R}$ where $R = 2^k \pmod{q}$
Output: $C = D \cdot R^{-1} \pmod{q}$

1: $T1 = D \cdot \mu \pmod{R}$
2: $T2 = D + T1 \cdot q$
3: $T3 = T2/R$ ▷ $T3 = T2 \gg k$
4: $T4 = T3 - q$
5: **if** $(T4 < 0)$ **then**
6: $\quad C = T3$
7: **else**
8: $\quad C = T4$
9: **end if**

$$q = q_H \cdot 2^w + 1 \tag{3.2}$$

Since we use word-level operations for Montgomery reduction, $R = 2^w$ and $\mu = -q^{-1} \pmod{R}$ should be redefined as $R' = 2^w$ and $\mu' = -q^{-1} \pmod{R'}$, respectively, for word-level operations. If we substitute $q = q_H \cdot 2^w + 1$ into μ', the value of μ' will become -1 and the multiplication operation with μ' will be simplified to 2's complement operation as shown in Eqn. 3.3.

$$\mu' = -q^{-1} \pmod{2^w} = -(q_H \cdot 2^w + 1)^{-1} \pmod{2^w} = -1 \pmod{2^w} \tag{3.3}$$

If we rewrite $T3$ in Algorithm 5 using R' and μ' for word-level operations, it can be simplified as shown in Eqn. 3.4-3.6. As shown in Eqn. 3.6, there is an extra one-bit *carry* produced by the term $\frac{D}{2^w} + \frac{-D(\bmod 2^w)}{2^w}$. The final form of $T3'$ with term *carry* is shown in Eqn. 3.7.

$$T3' = \frac{T2}{R'} = \frac{D + (D \cdot \mu'(\bmod 2^w))) \cdot q}{2^w} \tag{3.4}$$

$$T3' = \frac{D + (-D(\bmod 2^w)) \cdot (q_H \cdot 2^w + 1)}{2^w} \tag{3.5}$$

$$T3' = \frac{D}{2^w} + \frac{-D(\bmod 2^w)}{2^w} + (-D(\bmod 2^w)) \cdot q_H \tag{3.6}$$

Algorithm 6 Word-Level Montgomery Reduction Algorithm for NTT-friendly Primes

Input: $D = A \cdot B$ (a k-bit positive integer, $w \cdot (L-1) \leq k < w \cdot L$)
Input: $w = \log_2(2n)$ (word size)
Input: $L = \lceil k/w \rceil$ (repeat count)
Input: q (a k-bit positive integer, $q = q_H \cdot 2^w + 1$)
Input: $\mu = -q^{-1} \pmod{R}$ where $R = 2^{w \cdot L} \pmod{q}$
Output: $C = D \cdot R^{-1} \pmod{q}$

```
T3 = D
for (i = 0; i < L; i++) do
    T1_H = T3 ≫ w
    T1_L = T3 (mod 2^w)
    T2 = −T1_L (mod 2^w)
    carry = T2[w − 1] ∨ T1_L[w − 1]
    T3 = T1_H + (q_H · T2) + carry
end for
T4 = T3 − q
if (T4 < 0) then
    C = T3
else
    C = T4
end if
```

$$T3' = (D \gg w) + (-D(\bmod 2^w)) \cdot q_H + carry \tag{3.7}$$

After the first word-level operation defined in Eqn. 3.7, D is updated as $T3'$ and the word-level operation is repeated as necessary. The word-level operation defined in Eqn. 3.7 should be repeated for sufficient number of times to (at least $\lceil \frac{k}{w} \rceil$ times) reduce $2k$-bit input D to k-bit. The proposed word-level Montgomery reduction algorithm for NTT-friendly primes is shown in Algorithm 6.

For our design working with $n = 1024$, we select a word size $w = \log_2(2 \cdot 1024) = 11$. Since we work with a 32-bit q, the word-level reduction operation should be repeated $\lceil \frac{32}{11} \rceil = 3$ times. The proposed word-level Montgomery reduction algorithm for NTT-friendly primes for $n = 1024$ and $\log_2(2n)$ is shown in Algorithm 7. Our algorithm can easily be modified to scale for other n and q values. For example, for $n = 2048$, $w = 12$ and for a modulus of length $(4 \times 12) \leq k < (5 \times 12)$, 5 iterations are required. For $n = 4096$, $w = 13$ and for a modulus of length $(4 \times 13) \leq k < (5 \times 13)$, 5 iterations are required.

The proposed word-level Montgomery modular reduction algorithm divides reduction operation into a set of multiply and accumulate (MAC) operations. Namely,

Algorithm 7 Word-Level Montgomery Reduction Algorithm for NTT-friendly primes with $n = 1024$ and $w = 11$

Input: $D = A \cdot B$ (a k-bit positive integer, $22 \leq K \leq 32$)
Input: q (a k-bit positive integer, $q = q_H \cdot 2^{11} + 1$)
Input: $\mu = -q^{-1} \pmod{R}$ where $R = 2^{33} \pmod{q}$
Output: $C = D \cdot R^{-1} \pmod{q}$
1: $T1 = D$
2: **for** ($i = 0$; i<3; $i++$) **do**
3: $\quad T1_H = T1 \gg 11$
4: $\quad T1_L = T1 \pmod{2^{11}}$
5: $\quad T2 = -T1_L \pmod{2^{11}}$
6: $\quad carry = T2[10] \vee T1_L[10]$
7: $\quad T1 = T1_H + (q_H \cdot T2) + carry$
8: **end for**
9: $T4 = T1 - q$
10: **if** ($T4 < 0$) **then**
11: $\quad C = T1$
12: **else**
13: $\quad C = T4$
14: **end if**

it performs $X \cdot Y + Z + cin$ operation, which can be implemented using DSP blocks in Xilinx FPGAs. Each DSP slice has an optional output register, which can be utilized as the pipeline register, eliminating the need to utilize FPGA fabric registers for pipelining. Hardware design for Algorithm 7 is shown in Figure 3.2. The proposed modular reduction hardware is fully pipelined and has four clock cycle latency. It uses three DSP blocks and can produce one result per clock cycle after filling the pipeline.

Montgomery reduction algorithm takes $A \cdot B$ as input and calculates the output $A \cdot B \cdot R^{-1} \pmod{q}$, where $R = 2^{w \cdot \lceil \frac{k}{w} \rceil}$ is defined as Montgomery reduction residual. The residual has to be corrected using an extra multiplication with R to obtain $A \cdot B \pmod{q}$. This extra multiplication can be moved to the input by multiplying one of the inputs by R.

3.2.3 Iterative NTT Hardware

The iterative NTT hardware implements DIF_{NR} NTT and DIF_{RN} INTT algorithms shown in Algorithm 1 and Algorithm 2, respectively, which utilizes GS butterfly structure. The proposed hardware architecture utilizes 64 GS butterfly units as we aim for a high-performance design. The NTT and INTT are realized in the same

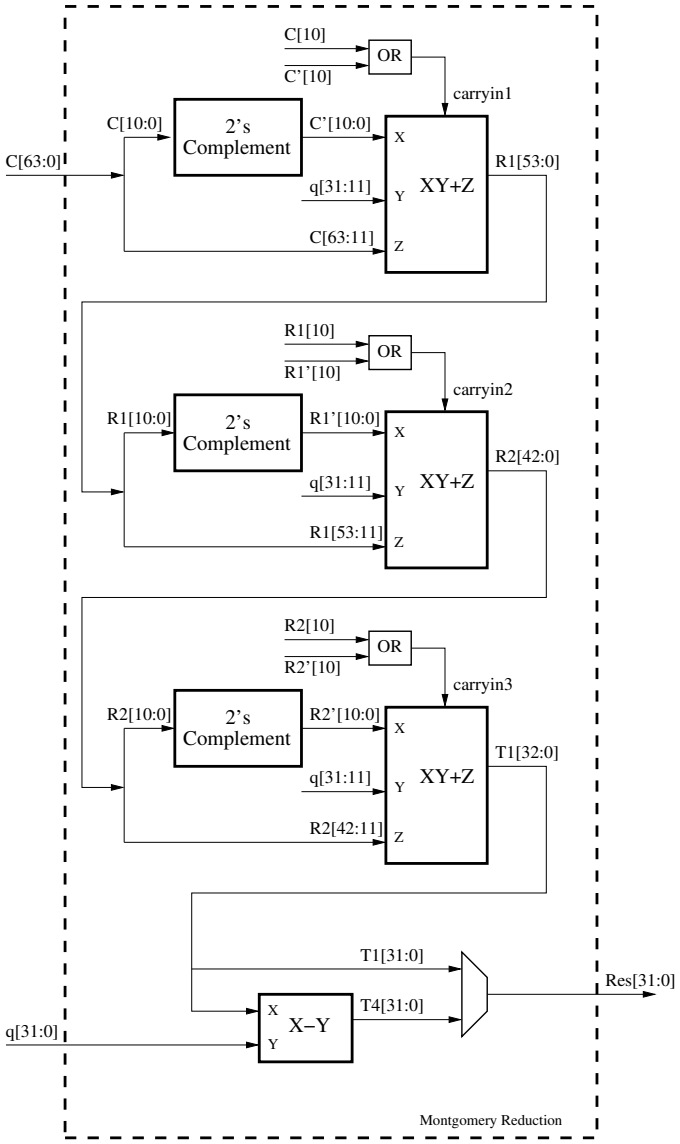


Figure 3.2 Word-Level Montgomery Modular Reduction Unit for NTT-friendly Primes

hardware, by just changing the precomputed twiddle (ω) factors.

3.2.3.1 GS Butterfly Unit

To realize the GS butterfly operation as shown in the steps 7-8 of Algorithm 1 and the steps of 8-9 of Algorithm 2, we designed a GS butterfly unit, which is shown in Fig. 3.3. The GS butterfly unit is fully pipelined and its latency is six clock cycles. The GS butterfly unit uses one modular addition, subtraction and multiplication units.

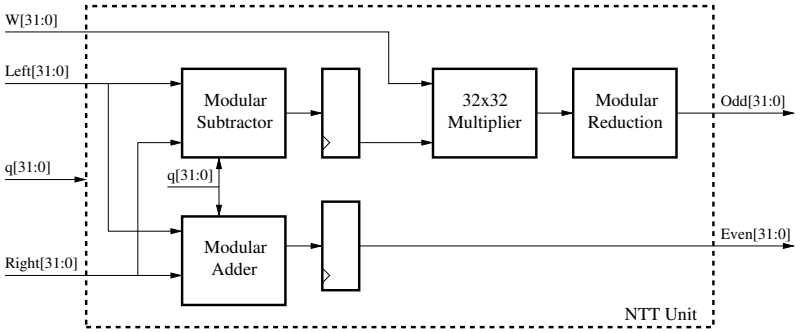


Figure 3.3 GS Butterfly Unit

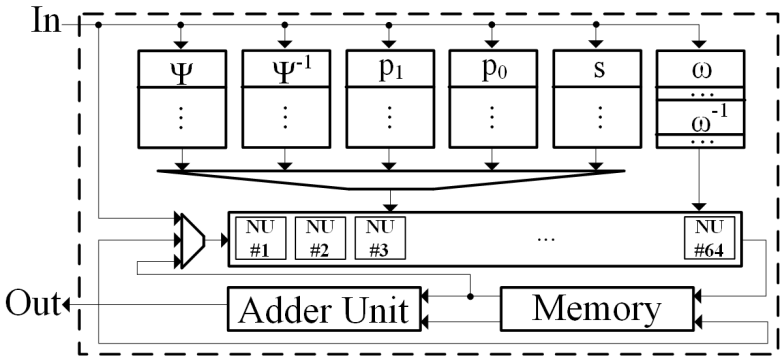


Figure 3.4 Iterative NTT Hardware

3.2.3.2 Overall Design

The overall design of our hardware is shown in Figure 3.4. This hardware employs 64 GS butterfly units and 128 separate block RAMs (BRAMs) to hold the coefficients of the input polynomial. Since each polynomial has 1024 coefficients, each BRAM holds 8 of the coefficients. Since we are utilizing an in-place NTT algorithm, after reading a coefficient from a BRAM, we only have $1024/128 = 8$ clock cycles to write back the computed result to its corresponding place. This requirement forced us to design a datapath with at most 6 clock cycle latency.

This hardware also employs 64 separate BRAMs for storing precomputed parameters (ω, ω^{-1}, modulus q). The precomputed powers of ω and ω^{-1} are multiplied with the Montgomery residual R prior to being sent to the FPGA, which eliminates extra multiplication with R after modular reduction operation. The control logic adjusts the most significant bit of BRAM address for alternating between NTT or INTT operations. NTT and INTT are realized in the same hardware, by just changing the precomputed twiddle (ω) factors. The iterative NTT hardware performs one NTT/INTT operation in 80 clock cycles in a pipelined manner.

It should be noted that DIF_{NR} NTT operation takes polynomial with coefficients in

Algorithm 8 Four-Step Cooley-Tukey NTT Algorithm (Cooley & Tukey, 1965)

Input: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in natural order
Input: primitive n-th root of unity $\omega \in \mathbb{Z}_q$, $n = n_1 \cdot n_2$
Output: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in natural order

1:
$$\boldsymbol{b} \leftarrow \begin{pmatrix} a_0 & a_1 & \dots & a_{n_2-1} \\ a_{n_2} & a_{n_2+1} & \dots & a_{2n_2-1} \\ \dots & \dots & \dots & \dots \\ a_{(n_1-1)n_2} & a_{(n_1-1)n_2+1} & \dots & a_{n_1n_2-1} \end{pmatrix}$$
2: **for** $(i = 0;\ 0 < n_2;\ i{+}{+})$ **do** ▷ applying n_1-pt NTT to the columns of $\boldsymbol{b}$
3: $\quad \boldsymbol{b}^T[i] \leftarrow \texttt{NTT}_{n_1}(\boldsymbol{b}^T[i])$
4: **end for**
5: **for** $(i = 0;\ 0 < n_1;\ i{+}{+})$ **do** ▷ the multiplication of $\boldsymbol{b}$ with the powers of ω
6: $\quad$ **for** $(j = 0;\ 0 < n_2;\ j{+}{+})$ **do**
7: $\quad\quad \boldsymbol{b}[i][j] = \boldsymbol{b}[i][j] \cdot \omega^{ij} \pmod q$
8: $\quad$ **end for**
9: **end for**
10: $\boldsymbol{b} = \boldsymbol{b}^T$
11: **for** $(i = 0;\ 0 < n_1;\ i{+}{+})$ **do** ▷ applying n_2-pt NTT to the columns of $\boldsymbol{b}$
12: $\quad \boldsymbol{b^T}[i] \leftarrow \texttt{NTT}_{n_2}(\boldsymbol{b^T}[i])$
13: **end for**
14: $\boldsymbol{a} \leftarrow \boldsymbol{b}$ ▷ converting matrix to vector
15: **return** $\boldsymbol{a}$

standard order as input and generates a polynomial with coefficients in bit-reversed order. However, since every polynomial in the NTT domain will have the same scrambled order, we can leave the result of the NTT operation as it is without doing any permutation. For polynomial multiplication, two polynomials will be converted to the NTT domain and their inner multiplication will be computed. This operation will yield a result that is still in the same scrambled order.

3.2.4 Four-step NTT Hardware

The four-step NTT hardware architecture implements Four-step NTT algorithm shown in Algorithm 8, which divides NTT/INTT operation into smaller-size NTT/INTT operations. As shown in Algorithm 8, Four-step NTT algorithm treats its input as an $n_1 \times n_2$ matrix and it requires n_1-pt NTT, n_2-pt NTT and coefficient-wise multiplication of a matrix with the powers of ω. Therefore, we designed two arithmetic units: an NTT unit for performing n_1-pt and n_2-pt NTT operations and a coefficient-wise modular multiplication unit.

Modular multiplication and NTT operations require similar hardware logic, and a reconfigurable hardware could be designed for performing both operations. However, this would require extra control logic routing throughout the device and reduce the performance. Besides, consecutive modular multiplication and NTT operations would not be pipelined efficiently. Therefore, for performance reasons, we use two separate units for modular multiplication and NTT operations. In order to make the NTT hardware efficient and reusable, n_1 and n_2 shown in Algorithm 8 are chosen as 32 for $n = n_1 \cdot n_2 = 1024$. Therefore, we can use the same 32-pt NTT unit for both n_1-pt and n_2-pt NTT operations.

3.2.4.1 32-pt NTT Unit

The 32-pt NTT unit uses Cooley-Tukey NTT algorithm (Cooley & Tukey, 1965) for implementing NTT as proposed in (Öztürk, Doroz, Savaş & Sunar, 2017). The algorithm takes the input, splits it into two halves and performs the half-sized NTT operation on the halves, and finally performs a reconstruction operation to combine the result of the two half-sized NTT operations into the result of the full-sized NTT operation. The reconstruction operation consists of a set of additions and subtractions in series with a set of multiplications. This is known as the divide-and-conquer approach that can be applied recursively to smaller parts.

The 32-pt NTT unit has 16 2-pt NTT units and four reconstruction stages. A 2-pt NTT unit takes A and B as inputs and calculates $A + B \pmod q$ and $A - B \pmod q$. The NTT unit is pipelined and its latency is 28 clock cycles. The four reconstruction stages have 8, 12, 14 and 15 modular multipliers, respectively.

3.2.4.2 Overall Design

The overall design of the four-step hardware architecture is shown in Fig. 3.5, which consists of a 32-point NTT unit and a 32-pt coefficient-wise modular multiplier unit with 32 modular multipliers. In addition to the arithmetic units, 32 separate BRAMs are used for storing precomputed powers of the twiddle factor. As in the iterative NTT hardware, the precomputed powers of ω are multiplied with the Montgomery residual R prior to being sent to the FPGA. The hardware also uses two memory blocks, each consisting of 32 BRAMs, for storing intermediate values during compu-

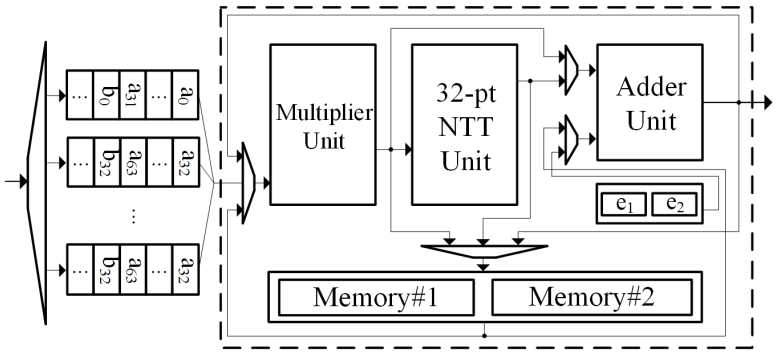


Figure 3.5 Four-Step NTT Hardware

tations. Each memory block can perform transpose operation as proposed in (Kalali, Mert & Hamzaoglu, 2016) besides read/write operations. Since the Four-Step NTT algorithm treats its input as a matrix and applies NTT to the columns of the matrix, the four-step hardware uses 32 input FIFOs for retrieving the inputs in the correct order. Thus, the four-step hardware can take 32 input coefficients per clock as in Algorithm 8. A similar structure utilizing 32 output FIFOs is also used at the output of the hardware. However, it is not shown for simplicity. The proposed four-step NTT hardware finishes one NTT/INTT operation in 140 clock cycles.

3.3 BFV Encryption/Decryption Architectures

In this section, we first present the encryption and decryption operation implementations in the SEAL library and their performance on the CPU. Then, we explain the two proposed architectures implementing encryption/decryption operations of the BFV scheme and briefly explain our optimizations. The first architecture utilizes iterative NTT hardware and it is referred to as iterative BFV hardware while the second architecture utilizes four-step NTT hardware and it is referred to as four-step BFV hardware. The proposed architectures target 128-bit security level, using degree-1024 polynomials ($n = 1024$) with 8-bit plaintext ($t = 256$) and 32-bit coefficients for ciphertext ($\lceil \log_2(q) \rceil = 32$).

Algorithm 9 Encryption Implementation in SEAL (Microsoft, 2019)

Input: $\boldsymbol{m} \in \boldsymbol{R}_{t,n},\ \overline{\boldsymbol{p}}_0, \overline{\boldsymbol{p}}_1 \in \boldsymbol{R}_{q,n}$
Output: $\boldsymbol{c}_0, \boldsymbol{c}_1 \in \boldsymbol{R}_{q,n}$
1: $\boldsymbol{u} \leftarrow R_2$
2: $\boldsymbol{p}_0\boldsymbol{u}, \boldsymbol{p}_1\boldsymbol{u} =$ NTT_DOUBLE_MULTIPLY $(\boldsymbol{u}, \overline{\boldsymbol{p}}_0, \overline{\boldsymbol{p}}_1)$
3: $\boldsymbol{e}_1, \boldsymbol{e}_2 \leftarrow \chi$
4: $\boldsymbol{c}_0 = [\boldsymbol{p}_0\boldsymbol{u} + \boldsymbol{e}_1 + \Delta \cdot \boldsymbol{m}]_q$
5: $\boldsymbol{c}_1 = [\boldsymbol{p}_1\boldsymbol{u} + \boldsymbol{e}_2]_q$
6: **return** $\boldsymbol{c}_0, \boldsymbol{c}_1$
7: **function** NTT_DOUBLE_MULTIPLY$(\boldsymbol{u},\ \overline{\boldsymbol{p}}_0,\ \overline{\boldsymbol{p}}_1)$
8: $\quad \overline{\boldsymbol{u}} = \texttt{MNTT}_n(\boldsymbol{u})$
9: $\quad \boldsymbol{p}_0\boldsymbol{u} = \texttt{MINTT}_n(\overline{\boldsymbol{p}}_0 \odot \overline{\boldsymbol{u}})$
10: $\quad \boldsymbol{p}_1\boldsymbol{u} = \texttt{MINTT}_n(\overline{\boldsymbol{p}}_1 \odot \overline{\boldsymbol{u}})$
11: $\quad$ **return** $\boldsymbol{p}_0\boldsymbol{u}, \boldsymbol{p}_1\boldsymbol{u}$
12: **end function**

3.3.1 Encryption/Decryption Implementations in SEAL Library

Encryption operation of the BFV in the SEAL (`v3.2`) is implemented the same way as the encryption operation in textbook-BFV as shown in Algorithm 9. In SEAL, public keys, $\overline{\boldsymbol{p}}_0$ and $\overline{\boldsymbol{p}}_1$, are stored in NTT domain and other ring elements used in the encryption, $\boldsymbol{u}$, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{m}$, are stored in polynomial domain. The ciphertext pair, $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$, are also stored in polynomial domain after encryption operation. In SEAL, ring elements $\boldsymbol{u}$, $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are randomly generated for each encryption operation and the SEAL uses hardware-based AES in counter mode for pseudo-randomness by default. SEAL employs encoding schemes to convert plaintexts from its integer representation to polynomial representation which is needed for the encryption operation. Therefore, the plaintext input $\boldsymbol{m}$ in Algorithm 9 is encoded as an element of $\boldsymbol{R}_{n,t}$ and stored in polynomial domain.

To improve its performance, the decryption operation of the BFV scheme in SEAL, shown in Algorithm 10, is implemented slightly different from the textbook-BFV, which requires division and rounding operations. In order to avoid these costly operations, SEAL uses the full RNS variant of textbook-BFV for decryption operation (Bajard et al., 2017) which requires base conversion as shown in Step 4 of Algorithm 10. This optimization is also used in our hardware realization. Decryption operation in SEAL uses ciphertexts, secret key and a redundant modulus $\gamma \in \mathbb{Z}$. In SEAL, secret key, $\overline{\boldsymbol{s}}$, is stored in NTT domain. Timing breakdowns of the encryption and decryption implementations in SEAL for $n = 1024$, $\lceil \log_2(q) \rceil = 27$ and $t = 256$ are shown in Table 3.1. The average time for one encryption and decryption in SEAL running on an Intel i9-7900X CPU is $151\mu s$ and $65.7\mu s$, respectively.

Algorithm 10 Decryption Implementation in SEAL (Microsoft, 2019)

Input: $\boldsymbol{c}_0, \boldsymbol{c}_1, \overline{\boldsymbol{s}} \in \boldsymbol{R}_{q,n}, \gamma \in \mathbb{Z}, \gamma > q$ s.t. $\gcd(\gamma, q) = 1$
Output: $\boldsymbol{m} \in \boldsymbol{R}_{t,n}$

$\boldsymbol{c_1 s} = \text{NTT\_MULTIPLY}\ (\boldsymbol{c}_1, \overline{\boldsymbol{s}})$
$\boldsymbol{c}_t = [(\boldsymbol{c_1 s} + \boldsymbol{c}_0) \cdot (\gamma \cdot t \pmod q)]_q$
for $m \in \{t, \gamma\}$ **do**
    $\boldsymbol{s}^{(m)} = [\text{FASTBCONV}(\boldsymbol{c_t}, q, \{t, \gamma\}) \cdot (-q^{-1} \pmod m)]_m$
end for
for $(i = 0;\ m < n;\ m{+}{+})$ **do**
    if $(\boldsymbol{s}^{(\gamma)}[i] > (\gamma/2))$ **then**
        $\boldsymbol{m}[i] = (\boldsymbol{s}^{(\gamma)}[i] - \boldsymbol{s}^{(\gamma)}[i] + \gamma) \pmod t$
    else
        $\boldsymbol{m}[i] = (\boldsymbol{s}^{(\gamma)}[i] - \boldsymbol{s}^{(\gamma)}[i]) \pmod t$
    end if
end for
$\boldsymbol{m} = [\boldsymbol{m} \cdot (\gamma^{-1} \pmod t)]_t$
return $\boldsymbol{m}$
function NTT_MULTIPLY$(c_1, \overline{s})$
    $\overline{\boldsymbol{c}}_1 = \texttt{MNTT}_n(\boldsymbol{c}_1)$
    $\boldsymbol{c_1 s} = \texttt{MINTT}_n(\overline{\boldsymbol{c}}_1 \odot \overline{\boldsymbol{s}})$
    return $\boldsymbol{c_1 s}$
end function
function FASTBCONV$(\boldsymbol{c}, q, \beta)$
    return $(\sum_{i=1}^{k} [[\boldsymbol{c}[i] . \frac{q_i}{q}]_{q_i} \cdot \frac{q}{q_i}]_m)_{m \in \beta}$
end function

Table 3.1 Timing of Encryption and Decryption Implementations in SEAL

Operation	Time (μs)	Percentage (%)
Encryption		
$\boldsymbol{u} \leftarrow R_2$	11.2	7.4 %
NTT_DOUBLE_MULTIPLY	45.6	30.1 %
$\boldsymbol{e}_1, \boldsymbol{e}_2 \leftarrow \chi$	91.1	60.2 %
Others	3.1	2.3 %
Decryption		
NTT_MULTIPLY	28.8	43.2 %
FASTBCONV	19.5	29.2 %
Others	17.4	27.6 %

3.3.2 Iterative BFV Hardware

In this work, iterative BFV hardware is constructed with slight modifications to the iterative NTT hardware in a similar way as explained in Section 3.2.3. Since the hardware architectures required to realize NTT and coefficient-wise modular multiplication operations are similar, we decide to utilize the same NTT hardware to

perform both operations. The overall design of the iterative BFV hardware architecture is shown in Fig. 3.4. There is also an adder unit performing modular additions and memory for storing intermediate operands. The iterative BFV hardware uses additional 64 modular multipliers and comparators for implementing modular multiplication and comparison operations in $\mathbb{Z}_\gamma$ shown in the step 4 and step 7 of the Algorithm 10, respectively. It also utilizes additional hardware blocks for modular addition and modular multiplication in $\mathbb{Z}_t$ used for decryption as shown in Algorithm 10. These additional hardware blocks are not shown in Fig. 3.4. In addition to 64 BRAMs for storing twiddle factors, the iterative BFV hardware employs extra BRAMs for storing public key ($\overline{\boldsymbol{p}}_0$, $\overline{\boldsymbol{p}}_1$), secret key ($\overline{\boldsymbol{s}}$) and the powers of precomputed ψ and ψ^{-1} as shown in Fig. 3.4.

The iterative hardware performs one NTT and 64 coefficient-wise modular multiplication operations in 80 clock cycles and 8 clock cycles, respectively. Therefore, one polynomial multiplication operation is performed in 192 clock cycles. Also, encryption and decryption operations are completed in 280 clock cycles and 248 clock cycles, respectively.

3.3.3 Four-step BFV Hardware

Encryption and decryption implementations in SEAL use three different arithmetic operations; namely NTT-based polynomial multiplication, coefficient-wise modular multiplication and addition. Therefore, we need three arithmetic units: an NTT unit for predefined ring degree ($n_1 = n_2 = 32$), a coefficient-wise modular multiplication unit and an adder unit in our architecture.

The proposed four-step NTT hardware is adapted with slight modifications for performing the encryption and decryption operations. The overall design of the four-step BFV hardware architecture is shown in Fig. 3.5, which consists of a 32-point NTT unit, a 32-point coefficient-wise modular multiplier and modular addition units. In addition to the arithmetic units, 32 separate BRAMs are used for storing precomputed, $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ (see Algorithm 9).

The four-step BFV hardware employs 32 separate BRAMs within the multiplier unit for storing precomputed parameters, $\overline{\boldsymbol{p}}_0$, $\overline{\boldsymbol{p}}_1$, $\overline{\boldsymbol{s}}$ and the powers of ω, ψ, ψ^{-1} as shown in Fig. 3.6. Each precomputed parameter has $n = 1024$ elements. The first 32 elements of each parameter are stored in the first BRAM. Similarly, the second 32 elements of each parameter are stored in the second BRAM and so on as shown

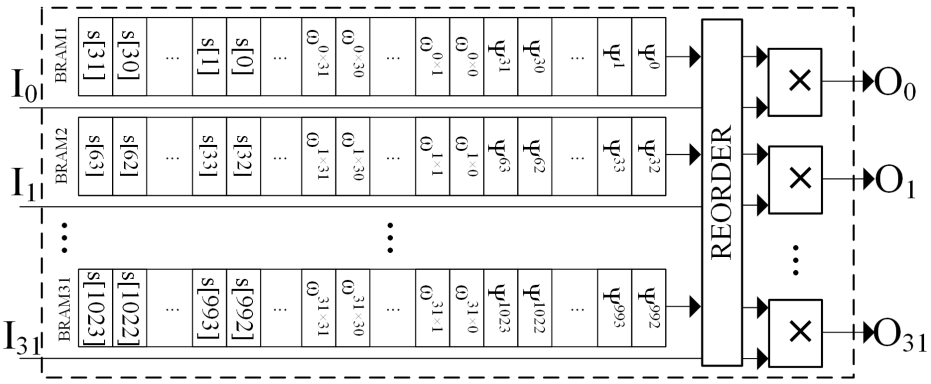


Figure 3.6 Multiplier Unit of Four-Step BFV Hardware

in Fig. 3.6. These parameters are stored in the same order as the order the four-step BFV hardware takes its inputs, which makes address generation easier. The outputs of 32 BRAMs are connected to the inputs of 32 modular multipliers in the multiplier unit. When a polynomial needs to be multiplied with one of the precomputed values, necessary addresses are generated to read the precomputed data from 32 BRAMs to the inputs of modular multipliers. Since INTT is also performed in the same unit with a different input order as explained in (Dai & Sunar, 2015), there is a reordering unit in the multiplier unit.

The polynomial multiplications with public and secret keys are performed in a slightly different way from its description in Section 2.5.2. Since the public key in the encryption operation and the secret key in the decryption operation are already in the NTT domain, none of them requires NTT. Therefore, the proposed hardware assumes one of the operands in polynomial multiplication is already in the NTT domain, which is a valid assumption for encryption/decryption operations for homomorphic applications, and it performs only one NTT and one INTT for polynomial multiplications.

The hardware starts the encryption operation by multiplying input $\boldsymbol{u}$ with the powers of ψ in the multiplier unit, which takes $32+6=38$ clock cycles. The multiplier unit takes 32 coefficients as inputs per clock cycle, and produces 32 outputs per cycle with six clock cycles latency. The resulting polynomial is sent to the NTT unit. In parallel to the $\texttt{NTT}_{1024}(\boldsymbol{u})$ operation, $\boldsymbol{m}\cdot\Delta+\boldsymbol{e}_1$ is computed using the multiplier and the adder units. Then, the result of $\boldsymbol{m}\cdot\Delta+\boldsymbol{e}_1$ is stored in the first memory block. It should be noted that since the proposed hardware is pipelined, the results of the multiplier unit are directly sent to the NTT unit as soon as the first 32 outputs are calculated. The pipeline overlaps consecutive coefficient-wise multiplication and NTT operations, and reduces the overall latency.

The NTT unit performs 32 32-point NTT operations in $28+32=60$ clock cycles and

the resulting coefficients are stored in the second memory block for the subsequent transpose operation. After the results of the last 32-point NTT are written into the memory block, 32 coefficients are read per cycle from the memory block, and sent to the multiplier unit for multiplication with the twiddle factors. The multiplier unit performs multiplication operations and the resulting coefficients are directly sent to the NTT unit, which completes in 60 clock cycles. In total, the proposed hardware finishes $\texttt{NTT}_{1024}(\boldsymbol{u})$ in 140 clock cycles.

Then, $\overline{\boldsymbol{u}}$ is sent to the multiplier unit that performs the multiplications of $\overline{\boldsymbol{u}}$ with $\overline{\boldsymbol{p}}_0$ and $\overline{\boldsymbol{p}}_1$ in $64+6=70$ clock cycles. The resulting polynomials, $\overline{\boldsymbol{p_0 u}}$ and $\overline{\boldsymbol{p_1 u}}$, are sent to the NTT unit for INTT operation. Since INTT requires different input ordering, polynomials, $\overline{\boldsymbol{p_0 u}}$ and $\overline{\boldsymbol{p_1 u}}$, are stored in the second memory block after the multiplication for the input reordering. INTT of $\overline{\boldsymbol{p_0 u}}$ and $\overline{\boldsymbol{p_1 u}}$ are performed in $140+32=172$ clock cycles, and the resulting polynomials, $\boldsymbol{p_0 u}$ and $\boldsymbol{p_1 u}$, are directly sent to the multiplier unit for multiplication with the powers of ψ^{-1}. Finally, $\boldsymbol{p_0 u}$ and $\boldsymbol{p_1 u}$ are directly sent to the adder unit for addition with $\boldsymbol{m}\cdot\Delta+\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, respectively. In total, the proposed hardware performs the encryption in 360 clock cycles.

For the decryption operation, the hardware computes $\texttt{NTT}_{1024}(\boldsymbol{c}_1)$ in the same manner as in the encryption. Then, it computes the multiplication $\overline{\boldsymbol{c_1 s}}$, performs $\texttt{INTT}(\overline{\boldsymbol{c_1 s}})$ and multiplies the result with the powers of ψ^{-1}. This polynomial multiplication operation is performed in 280 clock cycles. Since decryption operation requires comparison in $\mathbb{Z}_\gamma$, modular addition and modular multiplication in $\mathbb{Z}_t$ as shown in Algorithm 10, the proposed hardware uses additional hardware blocks for these operations. These blocks are not shown in Fig. 3.5 for simplicity. It should be noted that coefficient-wise modular multiplication operations in $\mathbb{Z}_\gamma$ (shown in Step 4 of Algorithm 10) are performed in the multiplier unit by changing modulus from q to γ and require no extra hardware. Finally, the necessary operations are performed as shown in Algorithm 10. The hardware completes one decryption operation in 360 clock cycles.

3.4 CPU-FPGA Framework

In order to demonstrate that homomorphic encryption/decryption operations of the SEAL library can be accelerated considerably, we designed a proof of concept ac-

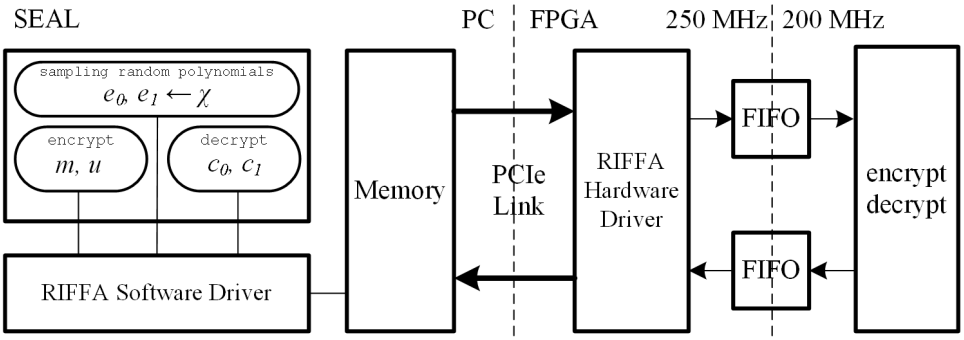


Figure 3.7 CPU-FPGA Framework

celerator framework that includes SEAL software and an FPGA accelerator that implements our architectures. For communication between the software stack and FPGA, we utilized Reusable Integration Framework for FPGA Accelerators (RIFFA) driver (Jacobsen, Freund & Kastner, 2012), which employs a PCIe connection between CPU and FPGA. The resulting framework is shown in Fig. 3.7.

In SEAL (`v3.2`), there are `encrypt` and `decrypt` functions, which work as described in Algorithm 9 and Algorithm 10. In our modified version of SEAL, `encrypt` and `decrypt` functions send their inputs $\boldsymbol{m}$, $\boldsymbol{u}$ and $\boldsymbol{c}_0$, $\boldsymbol{c}_1$, respectively, to FPGA and once FPGA returns the results to CPU, these functions return them to their caller functions. Precomputed constants such as keys are sent to FPGA only once prior to any invocation of `encrypt` and `decrypt` functions. In summary, all arithmetic operations in encryption and decryption are performed in FPGA except for sampling of random polynomials and encoding of the plaintext, which are performed in the host CPU and sent to FPGA prior to any operation.

One important aspect of the communication between CPU and FPGA is the utilization of Direct Memory Access (DMA). Instead of bringing data into the CPU first, prior to sending it to FPGA, the data is directly sent to FPGA from memory. This way, cache memory is never trashed, and running `encrypt` or `decrypt` function does not affect the performance of other operations running on CPU.

To realize our framework, we use Xilinx VC707 Evaluation Board, which includes a PCIe x8 Gen 2 Connector. Xilinx IP Core 7-Series Integrated Block for PCIe provides a 128-bit interface with a 250 MHz clock, which has a 32 Gbps theoretical maximum bandwidth. As shown in Fig. 3.7, separate FIFO structures are utilized for data input from the RIFFA driver and data output to the RIFFA driver. This approach is utilized to enable a pipelined architecture and maximize performance. In (Jacobsen et al., 2012), it is shown that RIFFA is able to achieve only 76% of the maximum theoretical bandwidth. Therefore, the bandwidth of the PCIe module is assumed to be $\sim$24 Gbps.

SEAL library uses 64-bit integer type for storing the coefficients regardless of the size of q. Since we work with 32-bit coefficients, we can pack and send $128/32 = 4$ coefficients per cycle. However, packing four 32-bit coefficients complicates the memory access in SEAL. Therefore, we pack and send $128/64 = 2$ coefficients per cycle. Both encryption and decryption operations take $2 \cdot 1024 = 2048$ coefficients as inputs and CPU can send $(8 \cdot 3 \cdot 10^9)/(8 \cdot 2048) = 183105$ encryption or decryption inputs per second with 24 Gbps bandwidth. In order not to be I/O bounded, the implementation on FPGA must finish its operations in less than $1\ sec/183105 = 5.46\ \mu s$. Since the proposed hardware implementations finish the encryption or decryption less than 5.46 μs as demonstrated in the next section, they are not I/O bounded.

3.5 Results and Comparison

We developed two architectures into Verilog modules and realized them using Xilinx Vivado 2018.1 tool for the Xilinx VC707 Evaluation Board utilizing a Virtex-7 FPGA (XC7VX485T-2FFG1761C), which has 303600 LUTs, 607200 DFFs, 2800 DSP48E1s and 1030 BRAM36E1s. The iterative and four-step BFV hardware use 25.63% and 22.36% of LUTs, 31.6% and 12.52% of RAMB36E1s, 34% and 21.39% of DSP48E1s in FPGA, respectively.

Many works were reported in the literature proposing hardware accelerators for HE schemes (Migliore, Real, Lapotre, Tisserand, Fontaine & Gogniat, 2018), (Cathébras, Carbon, Milder, Sirdey & Ventroux, 2018), (Öztürk et al., 2017), (Chen, Mentens, Vercauteren, Sinha Roy, Cheung, Pao & Verbauwhede, 2015), (Aysu, Patterson & Schaumont, 2013), (Pöppelmann & Güneysu, 2012), (Sinha Roy et al., 2019), (Sinha Roy, Järvinen, Vliegen, Vercauteren & Verbauwhede, 2018), (Pöppelmann, Naehrig, Putnam & Macias, 2015), (Feng et al., 2019), (Liu, Fan, Khalid, Rafferty & O'Neill, 2019), (Banerjee et al., 2019), (Song, Tang, Chen & Zhang, 2018), (Fritzmann & Sepúlveda, 2019). Some of these works focus on accelerating the multiplication of two large degree polynomials using NTT-based multiplication schemes (Aysu et al., 2013), (Chen et al., 2015), (Öztürk et al., 2017), (Pöppelmann & Güneysu, 2012), (Cathébras et al., 2018), (Feng et al., 2019), (Liu et al., 2019). Other works target accelerating different operations such as full encryption/decryption and homomorphic multiplication operations (Sinha Roy et al., 2019), (Migliore et al., 2018), (Sinha Roy et al., 2018), (Pöppelmann et al., 2015),

(Banerjee et al., 2019). Also, the works in (Fritzmann & Sepúlveda, 2019) and (Song et al., 2018) target fast NTT hardware for lattice-based cryptography, which can also be used for HE schemes. Although our hardware architectures accelerate encryption and decryption operations of the BFV scheme in SEAL, the core part of our architectures is the hardware implementation of a fast polynomial multiplier. For a fair comparison, therefore, we report and compare the hardware and performance results for the polynomial multiplier part of our works and the works in the literature in Table 3.2. We also include the performance results of the NTT operation of the works in the literature, if available, in Table 3.2. The proposed iterative and four-step hardware implementations have the lowest latency for both NTT and polynomial multiplication operations compared to works in the literature.

Also in Table 3.2, we include the implementation results of the iterative hardware on a low-cost Spartan-6 FPGA board (Mert et al., 2019). The results show that the timing result is comparable to the one in (Chen et al., 2015). Note that we achieve a comparable timing result using a general ciphertext modulus q while (Chen et al., 2015) uses a special modulus. In terms of the area, our design uses much less distributed logic at the expense of ten additional DSPs. Although there are other accelerators (Seiler, 2018) in the literature performing R-LWE encryption and decryption, these works use small parameters and are not designed for homomorphic operations. Thus, they are not included in the comparison.

Although the proposed work has relatively small parameters for homomorphic operations and has a low multiplicative depth, it can be extended to a new design with a larger ring degree and ciphertext modulus using exactly the same arithmetic units in this work. For example, for a design with ring degree of 4096 and 180-bit ciphertext modulus, we just need to update the control unit of NTT hardware so that it can work for ring degree of 4096 instead of 1024 using exactly the same NTT units. Also, the ciphertext modulus can be increased to 180-bit by using exactly the same polynomial multipliers in this work with additional CRT (Bonehet al. , 1999) operations employing CRT. In such a setting, the proposed hardware needs a CRT unit that transforms each 180-bit coefficient into six coefficients in six 32-bit primes, performs operations separately for each 32-bit prime using the desired number of hardware units in parallel and converts coefficients in six 32-bit primes into 180-bit coefficients. Therefore, the arithmetic blocks proposed in this work with a small parameter set can be used to design a high-performance hardware for larger parameter sets with minor modifications. We present two different scaled version of the proposed architectures for $n = 4096$ with 32-bit q and $n = 4096$ with 180-bit q, and reported estimated timing and area results, showing timing and area results are linearly proportional to n and q, in Table 3.2.

Table 3.2 Comparative Table

Work	**Platform**	($\log_2(n)$,$\log_2(q)$)	**LUT/DSP/BRAM**	**Clock (MHz)**	**Latency (μs)**	
					NTT	**PM**
(Sinha Roy et al., 2018)	Virtex-6	(16,30)	72K / 250 / 106	100	–	3376
(Pöppelmann et al., 2015)	Virtex-7	(12,125)	69K / 144 / –	100	–	1960
(Migliore et al., 2018)	Stratix-V	(11,125)	30K / 100 / –	331	–	583
(Sinha Roy et al., 2019)	Zynq US	(12,30)	64K / 200 / 400	225	73	171
(Öztürk et al., 2017)	Virtex-7	(15,32)	219K / 768 / 193	250	51	152
(Pöppelmann & Güneysu, 2012)[a]	Spartan-6	(10,30)	1644 / 1 / 6.5	200	–	110
(Aysu et al., 2013)[a]	Spartan-6	(10,17)	– / 3 / 2	–	–	100
(Chen et al., 2015)[a]	Spartan-6	(8,21)	2829 / 4 / 4	247	–	6
	Spartan-6	(10,31)	6689 / 4 / 8	241	–	33
(Cathébras et al., 2018)	Virtex-7	(12,30)	54K / 517 / 208	200	–	10
(Feng et al., 2019)[a]	Spartan-6	(8,21)	14K / 128 / 1	233	–	0.94
		(9,23)	18K / 128 / 2.5	200	–	1.77
(Liu et al., 2019)[a]	Kintex-7	(8,17)	317 / 1 / –	333	102	–
(Banerjee et al., 2019)[b]	40nm CMOS	(8,24)	106K / – / –	72	17	–
(Song et al., 2018)[b]	40nm CMOS	(9,18)	– / – / –	300	1.6	–
(Fritzmann & Sepúlveda, 2019)[b]	UMC 65nm	(10,17)	14K / – / –	25	41	–
I (Mert et al., 2019)	Spartan-6	(10,32)	1208 / 14 / 14	212	–	37
I	Virtex-7	(10,32)	77K / 952 / 325.5	200	0.4	0.96
FS			67K / 599 / 129		0.7	1.40
I[c]	Virtex-7	(12,32)	∼80K / 952 / 325.5	∼200	∼ 1.75	∼ 4.20
FS[c]			∼70K / 599 / 129		∼ 2.3	∼ 4.75
I[d]	Virtex-7	(12,180)	∼160K / 1904 / 651	∼200	∼ 5.25	∼ 12.60
FS[d]			∼140K / 1198 / 258		∼ 6.9	∼ 14.25

[a]:Fixed q.
[b]:Multiple n and q.
[c]:Scaled for n=4096.
[d]:Scaled for n=4096 and q=180-bit (assuming two 32-bit hardware are instantiated, excluding CRT).

Software implementation using only SEAL completes encryption, decryption and one polynomial multiplication in 151μs, 65.7μs and 28.8μs, respectively. Our FPGA implementation of the iterative BFV hardware, excluding I/O operations, performs encryption, decryption and polynomial multiplication in 1.4μs, 1.24μs and 0.96μs, respectively; resulting in 108×, 53× and 30× speedup values for those operations when compared with the pure software implementation. Similarly, the FPGA implementation of the four-step BFV hardware performs both encryption and decryption operations in 1.8μs and polynomial multiplication in 1.4μs; resulting in 84×, 37× and 21× speedup values for encryption, decryption and one polynomial multiplication, respectively. The iterative BFV hardware performs both encryption and decryption faster than the four-step BFV hardware at the expense of more resources.

Transmission of a polynomial of degree 1024 with 32-bit coefficients between CPU and FPGA via DMA takes 2.73μs by packing two coefficients per cycle. For iterative BFV hardware, for encryption operation, without pipelining of the transmission and the FPGA computation and with half-duplex PCIe communication, we achieve $5.46+1.4+5.46=12.32\mu s$ latency, where 5.46μs is spent for sending the input, 1.4μs for the encryption operation and another 5.46μs is spent for receiving the output. In comparison with pure software implementation, this indicates a 12× speedup for encryption. Similarly, for decryption operation, we obtain $5.46+1.24+2.73=9.42\mu s$

Table 3.3 Pipelining of I/O Operations over PCIe

Time (μs)	**Input**	**Operation**	**Output**
0	Enc1	–	–
5.46	Dec1	Enc1	–
10.92	Enc2	Dec1	Enc1
16.38	...	Enc2	Dec1
21.84	...	...	Enc2
...	...	...	...

latency, which is a 7× speedup over the software implementation. Also, we achieve a throughput of almost 81K and 106K for encryption and decryption operations, respectively, per second without pipelining. Performance results for the four-step hardware can be calculated similarly.

In the current implementation, PCIe works in half-duplex mode, where the host CPU can either send or receive one encryption/decryption operation at a time over PCIe. If we use PCIe in a full-duplex mode where PCIe can send and receive data at the same time and overlap I/O operations over PCIe with actual encryption and decryption operations as shown in Table 3.3, the proposed hardware can send the result of one encryption or decryption operation back to the host CPU in $\texttt{max}(5.46, 1.4, 1.24, 2.73) = 5.46\mu s$ after filling the pipeline. In this setting, the proposed framework can perform $1/5.46\mu s = 183150$ encryption or decryption operations per second. Compared to $1/151\mu s = 6622$ encryption and $1/65.7\mu s = 15220$ decryption operations per second, we can achieve 27× and 12× speedup over pure software encryption and decryption implementations, respectively.

3.6 Summary

We presented FPGA implementations of two fast and highly parallelized hardware architectures for the encryption and decryption operations of the BFV HE scheme. We utilized our architectures in an accelerator framework for the encryption and decryption operations of the BFV HE scheme implemented in the SEAL. We adopt a hardware/software co-design approach, in which encryption and decryption operations are offloaded to an FPGA while the rest of operations in the BFV scheme of SEAL are executed in software running on a desktop computer. We realized the framework on an FPGA connected to the PCIe bus of an off-the-shelf desktop

computer. We used the Xilinx VC707 Evaluation Board for our implementation. We improved the latency of the encryption and decryption by almost 12× and 7×, respectively, compared to their pure software implementations in SEAL.

4. AN FPGA-BASED RUN-TIME CONFIGURABLE NTT-BASED POLYNOMIAL MULTIPLICATION ARCHITECTURE

In this chapter, we propose a *run-time* configurable and highly parallelized NTT-based polynomial multiplier architecture supporting six different parameter sets, which are mostly utilized in the lattice-based HE and PQC applications. For proof of concept, we also utilize our NTT-based polynomial multiplier architecture in a CPU-FPGA framework, which provides high-speed communication between the SEAL library running on CPU and the proposed hardware on FPGA by utilizing the RIFFA driver (Jacobsen et al., 2012) employing a PCIe standard interface. Compared to the SEAL library, the proposed hardware improves the latency of polynomial multiplication operation by up to 7× and 4.2×, excluding and including I/O overhead, respectively. [1]

4.1 Introduction

The design of the NTT-based polynomial multiplication operation is mainly based on two parameters: the ring size, n, and the bit length of the coefficient modulus, $k = \lceil \log_2(q) \rceil$, where q is the coefficient modulus. For cryptographic applications utilizing such different parameters, separate polynomial multipliers need to be designed and implemented. For example, post-quantum KEM protocol CRYSTALS-Kyber (v1) (Bos et al., 2018) uses parameters $n = 256$ and $k = 13$, where $q = 7681$ is a 13-bit prime, while SEAL HE library (Microsoft, 2019) uses parameters n ranging from 1024 to 32768 and k ranging from 14-bit to 60-bit. Therefore, this is the motivation for a configurable NTT-based polynomial multiplier architecture, which can support multiple parameter sets and applications instead of separate architectures for each application with fixed parameters.

[1] This chapter presents the work in (Mert et al., 2020).

To this end, in this chapter, we propose a *run-time* configurable and highly parallelized NTT-based polynomial multiplier architecture for hardware realizations. Our motivation in our study is two fold: *i*) our architecture effectively supports different parameter sets with high efficiency and *ii*) thus aims to improve the performance of a wide range of lattice-based cryptosystems. The proposed architecture supports six parameter sets $(n,k) = \{(256,16), (512,16), (1024,16), (1024,32), (2048,32), (4096,32)\}$, which are utilized in the lattice-based cryptosystems, with a flexible memory addressing scheme. Here, for proof of concept, we also utilize our NTT-based polynomial multiplier architecture in a CPU-FPGA framework, which provides high-speed communication between the CPU and the FPGA by utilizing the RIFFA driver (Jacobsen et al., 2012) employing a PCIe standard interface.

The proposed architecture accelerates the decryption operation of the BFV scheme implemented in the SEAL (Microsoft, 2019). In the proposed framework, the polynomial multiplication operation in the decryption of the BFV scheme is offloaded to the accelerator in the FPGA via PCIe bus while the rest of the operations in the decryption of the BFV scheme are executed in software running on an off-the-shelf desktop computer. Offloading the computation to the accelerator results in overhead due to the time spent in the network stack at both ends of the communication and actual transfer of data, which we refer to as the I/O overhead. This overhead can be prohibitively high if the nature and cost of the offloading are not factored in the accelerator design.

To address the speed and configurability requirements, three crucial design goals are considered in this work: *i*) hardware accelerator architecture should be designed to provide significant levels of speedup over software implementations, *ii*) the overhead due to communication between hardware and software components should be taken into account as a design parameter or constraint and *iii*) a balanced implementation in terms of area and throughput should be designed as the proposed architecture supports different parameter sets and aims to provide acceleration for a variety of applications. Most works in the literature focus solely on the first goal and report no accurate speedup values subsuming the I/O overhead. In this chapter, we aim to address this problem by providing a fully working prototype of a framework consisting of an FPGA-based an accelerator and SEAL library running on a CPU.

The rest of the chapter is organized as follows. Section 4.2 introduces the proposed polynomial multiplication hardware. Section 4.3 presents the utilization of the proposed hardware for the SEAL library as a proof-of-concept. Finally, Section 4.4 presents the implementation results and the comparison with the literature, and Section 4.5 concludes the chapter.

4.2 Polynomial Multiplicationr Architecture

In this section, the proposed run-time configurable polynomial multiplier architecture, its main building blocks, the design techniques we used for our entire framework and our optimizations are explained. The proposed architecture employs DIF_{NR} NTT and DIF_{RN} INTT algorithms for NTT and INTT operations respectively. This design choice enables employing only GS-based butterfly structure, which reduces hardware complexity at the expense of extra computations (pre-processing and post-processing operations) during polynomial multiplication.

4.2.1 Run-time Configurable Word-Level Montgomery Modular Multiplier Unit

The proposed configurable modular multiplier hardware consists of two blocks, an integer multiplier hardware and a configurable word-level Montgomery modular reduction hardware. It supports modular multiplication operation only with NTT-friendly modulus, which satisfies $q \equiv 1 \pmod{2n}$, for $k = \lceil \log_2(q) \rceil$ ranging from 10 to 32.

First, we design a 32-bit integer multiplier. The proposed integer multiplier hardware utilizes four DSP blocks in FPGAs and an adder tree structure in a similar way as shown in Fig. 3.1. The proposed 32-bit integer multiplier hardware performs a multiplication operation in two clock cycles. It is pipelined and uses optional output registers of DSP blocks as pipeline registers, which eliminates the need to utilize FPGA fabric registers for pipelining. Therefore, the proposed integer multiplier can produce one multiplication result per clock cycle after filling the pipeline. The proposed integer multiplier works for any integer inputs with 32 or fewer bits.

After the multiplication operation, the result needs to be reduced back to the bit-length of the modulus. For a configurable architecture, we modified the word-level Montgomery modular reduction algorithm presented in Chapter 3. In Chapter 3, the word-level Montgomery reduction algorithm is implemented for a fixed parameter set, namely $n = 1024$ and $k = 32$, while we, in this work, implement a configurable architecture, which supports multiple parameter sets.

Recalling from Chapter 3, the proposed word-level Montgomery modular reduction

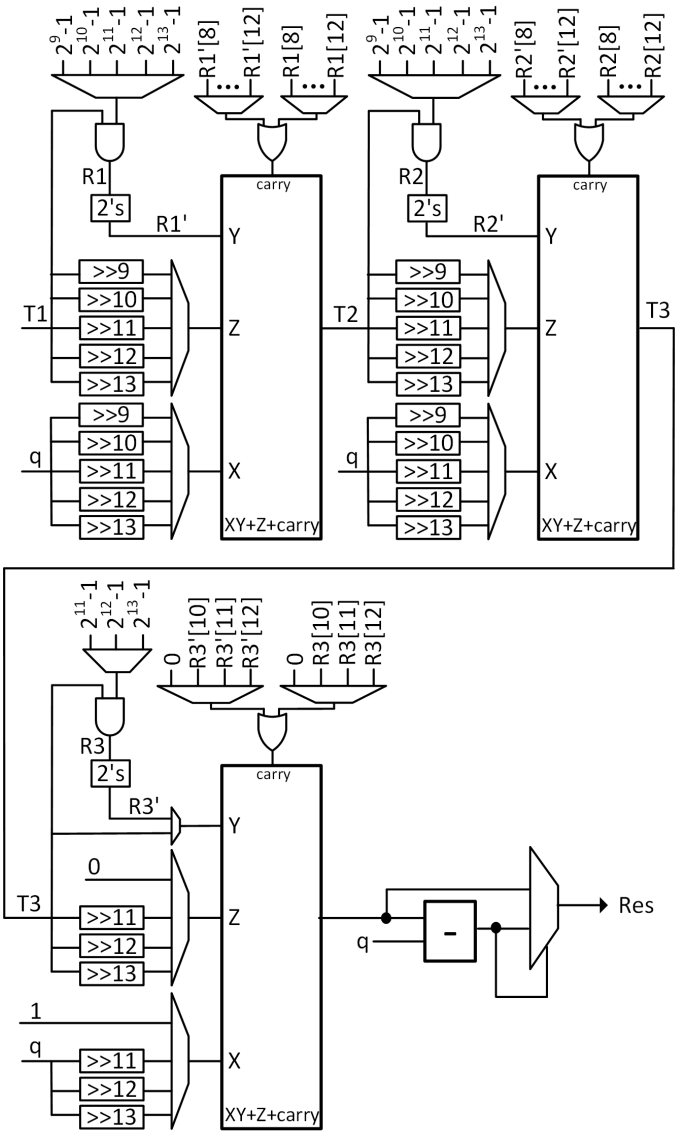


Figure 4.1 Run-time Configurable Word-level Montgomery Modular Reduction Unit

algorithm uses the property of NTT-friendly modulus with *negative wrapped convolution* technique, $q \equiv 1 \pmod{2n}$, and it divides Montgomery reduction operation into smaller steps. The modular multiplier architecture in this work can be configured to perform modular multiplication operation for six different parameter sets. The proposed run-time configurable word-level Montgomery modular multiplier architecture is shown in Fig. 4.1. Since maximum iteration count, L, is $\lceil \frac{32}{13} \rceil$=3 (for the parameter set $n = 4096$ and $k = 32$), the proposed modular multiplier uses three units performing $X \cdot Y + Z + carry$ operation. For parameters $(256, 16)$, $(512, 16)$ and $(1024, 16)$, which require L=2 iterations, the last $X \cdot Y + Z + carry$ operation is eliminated by selecting X, Y, Z and $carry$ inputs of the third unit as 1, $T3$, 0 and 0, respectively. Each $X \cdot Y + Z + carry$ operation is realized using one DSP block inside the FPGAs.

Table 4.1 Actual Supported k Range for Each Parameter Set

(n, k)	Range
$(256, 16)$	$9 < k \leq 18$
$(512, 16)$	$10 < k \leq 20$
$(1024, 16)$	$11 < k \leq 22$
$(1024, 32)$	$22 < k \leq 33$
$(2048, 32)$	$24 < k \leq 36$
$(4096, 32)$	$26 < k \leq 39$

For a given parameter set, the proposed modular multiplier selects proper inputs for DSP blocks and performs the modular multiplication operation in constant-time. For example, for parameters $n = 256$ and $k = 16$, the proposed architecture selects $T1$, $T2$ and q shifted by 9 for the first two DSP blocks and eliminates the last $X \cdot Y + Z + carry$ operation as explained before. For given w and L, the proposed modular reduction architecture supports modulus with $w \cdot (L-1) < k \leq w \cdot L$. Therefore, for a parameter set, the proposed architecture supports a range of k instead of a single k value. For example, for $(1024, 32)$ with $w = 11$ and $L = 3$, the proposed architecture supports modulus with $22 < k \leq 33$. Supported k range for each parameter set is listed in Table 4.1. As shown in the table, the proposed architecture can support modulus up to 39-bit long for the parameter set $(4096, 32)$. Therefore, although the proposed polynomial multiplier architecture supports modulus up to 32-bit, it can be easily extended to 39-bit with a slight modification of integer multiplier.

The word-level Montgomery modular reduction algorithm takes $A \cdot B$ as input and produces $A \cdot B \cdot R^{-1} \pmod{q}$, where $R = 2^{w \cdot L}$. Therefore, the output of the word-level Montgomery modular reduction algorithm should be multiplied with R for eliminating the extra R^{-1} in the result. In this work, this extra multiplication operation is avoided by multiplying one of the multiplication inputs with R or the power of R. The proposed modular multiplier architecture performs a modular multiplication operation in six clock cycles for all parameter sets. It is pipelined and uses internal registers of DSP blocks as pipeline registers. Therefore, the proposed configurable modular multiplier can produce one multiplication result per clock cycle after filling the pipeline.

Compared to the regular Montgomery and Barrett algorithms, the proposed word-level Montgomery algorithm reduces the size of multiplication operations and provides better utilization for FPGA implementations. For example, for the parameter set $(1024, 32)$, the regular Montgomery algorithm uses two 32×32 multipliers, which require seven DSP blocks for its FPGA implementation. The regular Barrett algorithm uses 32×64 and 32×32 multipliers, which require nine DSP blocks for its

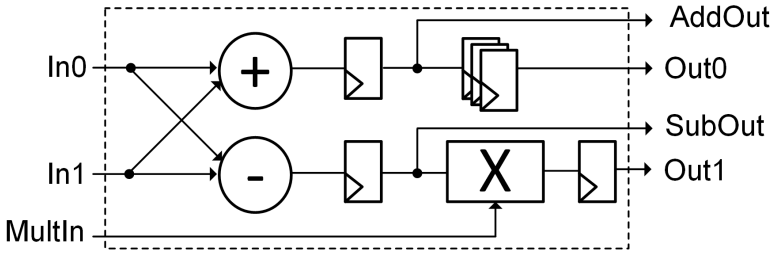


Figure 4.2 NTT Unit

implementation for the same parameter set. The proposed word-level Montgomery algorithm, on the other hand, uses three 11×21 multipliers, which require three DSP blocks for its implementation for the same parameter set. Therefore, the proposed word-level Montgomery algorithm uses 57% and 66% less DSP blocks than the regular Montgomery and Barrett algorithms, respectively, for the parameter set $(1024, 32)$. This shows the superiority of our approach.

4.2.2 NTT Unit

The proposed design uses NTT units, which implement the GS butterfly configuration shown in steps 7-8 of the Algorithm 1 and steps 8-9 of the Algorithm 2. The proposed NTT unit is shown in Fig. 4.2 and it consists of one modular adder, one modular subtractor, and one modular multiplier hardware. The first output, *Out0*, comes from the modular adder while the second output, *Out1*, comes from modular subtractor and multiplier. Due to the six clock cycles latency of the modular multiplier, there are six clock cycles differences between the two output coefficients. In order to synchronize both output coefficients, extra six flip-flops are placed at the output of modular adder hardware. The proposed NTT unit has seven clock cycles latency and it is pipelined. It takes three coefficients as inputs and produces two coefficients as outputs per clock cycle after filling the pipeline.

The proposed NTT unit can also be configured to perform a single modular multiplication operation by providing 0, first multiplicand and second multiplicand to the inputs *In0*, *In1* and *MultIn*, respectively. This configuration is used for performing coefficient-wise multiplication of polynomials in the NTT domain and multiplying the coefficients of the polynomial with the powers of Ψ, Ψ^{-1} and n^{-1} in $\mathbf{Z}_q$. This configurability and re-use of the NTT unit eliminate the need for extra modular multiplier unit. The proposed NTT unit can also be configured to perform modular addition and subtraction operations by reading *AddOut* and *SubOut* outputs.

4.2.3 Overall Design

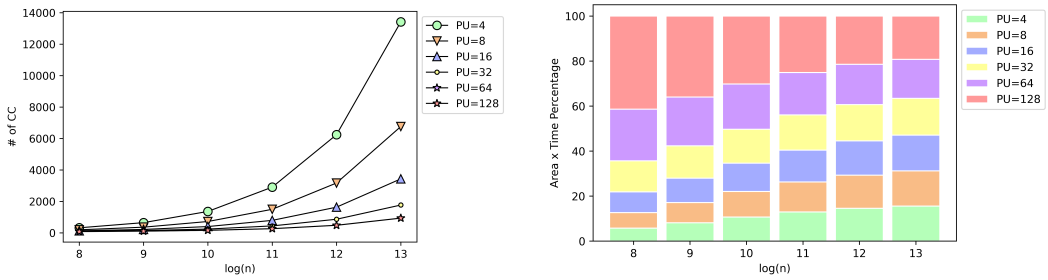
Determining the degree of parallelization in architecture is not an easy task and it depends on the area and throughput requirements of the application. The NTT and INTT schemes shown in Algorithm 1 and Algorithm 2, respectively, allow the parallelization of the NTT and INTT operations by performing multiple GS butterfly operations in parallel in one stage. An n-pt NTT consists of $\log_2(n)$ stages, where each stage has $(n/2)$ butterfly operations. Therefore, an NTT operation can be parallelized by performing multiple butterfly operations in one stage in parallel. In this work, hardware block performing one butterfly operation is referred to as processing unit (PU).

The proposed architecture aims at high performance with reasonable resource usage. Since the main building block of a polynomial multiplication operation is NTT, we analyze the performance of NTT operation with different number of PUs in order to decide the optimal number of PUs for a balanced design in terms of both area and throughput. We model the number of latency in terms of clock cycle as shown in Eqn. 4.1.

$$\log_2(n) \times (\frac{n}{2 \times \text{PU number}} + 6) \tag{4.1}$$

Then, we plot the latency vs. $\log_2(n)$ graph for PU numbers ranging from 4 to 128 for the n values in our parameter set as shown in Fig. 4.3a. We also plot the area×time vs. $\log_2(n)$ graph for PU numbers ranging from 4 to 128 for the n values used in our parameter set as shown in Fig. 4.3b, where PU number and the latency in terms of clock cycles are used for area and time parameters, respectively. When large n is used as required in homomorphic applications, the latency of NTT operation increases significantly for designs with 4, 8 and 16 PUs. Also, when n is large, the designs with 4, 8 and 16 PUs show similar area×time performance with other designs as shown in Fig. 4.3b. When small n is used as required in most lattice-based post-quantum cryptosystems, the designs with 64 and 128 PUs show worse area×time performance than other designs with similar latency performance as shown in Fig. 4.3b. As we aim for a balanced architecture in terms of area and performance, we select the PU number as 32 in our design.

The overall architecture of the proposed run-time configurable NTT-based polynomial multiplier is shown in Fig. 4.4. The proposed architecture uses 32 PUs, where a PU performs steps 5-10 and steps 7-12 in Algorithm 1 and Algorithm 2, respectively. A PU consists of one NTT unit, four BRAMs for storing the coefficients of polynomials (*POL0* and *POL1* in Fig. 4.4), one BRAM for storing the powers of ω and ω^{-1} to



(a) Number of Clock Cycles vs. $\log_2(n)$ for Different PU Numbers

(b) Area×Time Percentage vs. $\log_2(n)$ for Different PU Numbers

Figure 4.3 Number of Clock Cycles and Area×Time Percentage Estimations for Different n and PU Numbers

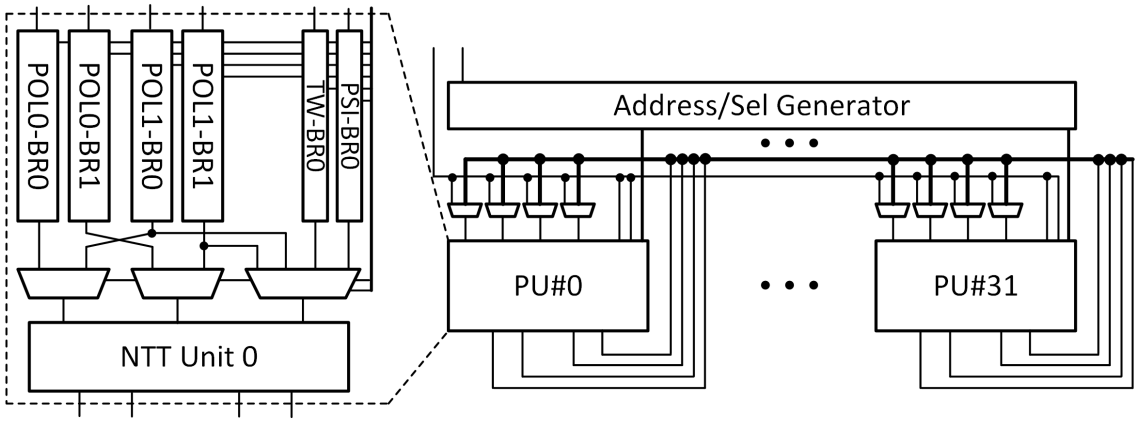


Figure 4.4 Overall Design

be used for NTT/INTT operations (TW in Fig. 4.4) and one BRAM for storing the powers of Ψ and Ψ^{-1} to be used for pre-processing and post-processing operations (PSI in Fig. 4.4). The proposed architecture also has an address/selection signal generator unit, which produces all necessary control signals for the NTT, INTT and polynomial multiplication operations.

The proposed hardware architecture can be configured to perform three different operations: NTT, INTT and NTT-based polynomial multiplication. The NTT and INTT operations are performed as described in Algorithm 1 and Algorithm 2, respectively. The polynomial multiplication, on the other hand, can be performed in two different ways: i) both input polynomials are in the polynomial domain and ii) one of the input polynomials is in the polynomial domain while the other input polynomial is already in the NTT domain. The first and second polynomial multiplication methods will be referred to as full polynomial multiplication (FPM) and half polynomial multiplication (HPM) for the rest of the chapter. The proposed architecture can perform both FPM and HPM operations.

Managing complex memory access schedule is one of the most challenging parts of

Table 4.2 Number of Clock Cycles Required for Each Operation and Parameter Set

Operations	(n,k)					
	(256,16)	(512,16)	(1024,16)	(1024,32)	(2048,32)	(4096,32)
NTT	104	153	250	250	451	876
INTT	121	178	291	291	524	1013
HPM	259	381	623	623	1121	2163
FPM	299	469	815	815	1537	3059

NTT-based polynomial multiplier architecture design. When a configurable architecture is aimed, this problem becomes more challenging since a flexible memory access schedule is required. Polynomials with different degrees (n) require different data alignment in memory and data access patterns. Therefore, the proposed memory access scheme in this work generates necessary control signals for storing and accessing polynomial coefficients as required for each parameter set.

An n-pt NTT operation can be implemented using two $(n/2)$-pt NTT operations after the first stage of the n-pt NTT operation (Chu & George, 1999). In this work, we exploit this property to design a configurable polynomial multiplier architecture in terms of n. Therefore, in this work which supports 256-pt to 4096-pt NTT/INTT, large NTT operations are performed using smaller NTT operations. For example, the proposed architecture with 32 PUs can perform one stage of 64-pt NTT with 7 clock cycles latency, where each PU performs one butterfly operation. Therefore, one 64-pt NTT with 6 stages will have a latency of $6 \cdot 7$=42 clock cycles. One 128-pt NTT operation then will perform its first stage in $(64/32) + 7$=9 clock cycles and two 64-pt NTT operations will be performed in $6 \cdot (2 + 7)$=54 clock cycles as pipelined. In total, 128-pt NTT will be performed in $9 + 54$=63 clock cycles. Similarly, 256-pt NTT operation will be performed in $8 \cdot (4 + 7)$=88 clock cycles. It should be noted that these calculations ignore delays and stalls due to control of the NTT operation in the implementation. INTT operation is also performed similarly.

Full polynomial multiplication operation uses two NTT, one INTT and four coefficient-wise multiplication of polynomials as shown in Eqn. 2.11- 2.14. Similarly, half polynomial multiplication operation uses one NTT, one INTT and three coefficient-wise multiplication of polynomials. The number of clock cycles required to perform each operation by the proposed architecture for supported parameter sets is shown in Table 4.2.

The powers of ω and ω^{-1} are stored in 32 BRAMs as shown in Fig. 4.5. Since the proposed architecture divides an NTT operation into smaller NTT operations, only the twiddle factors necessary for performing the first stage of NTT operation for

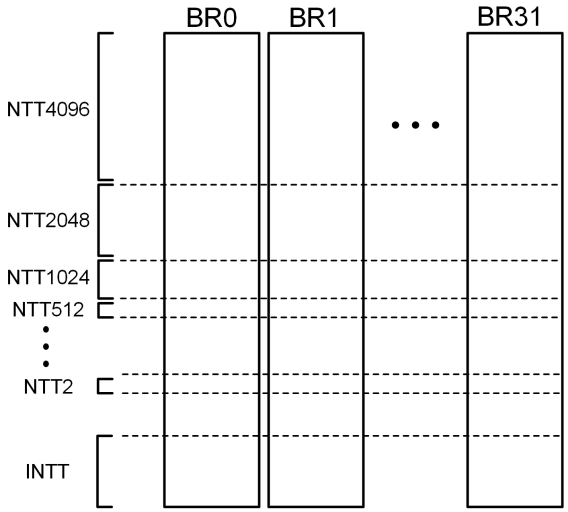


Figure 4.5 BRAMs Storing Twiddle Factors

sizes from 2 to 4096 are stored. In total, $32 \cdot (64+32+16+8+4+2+1+1+1+1+1+1)$=4224 twiddle factors are stored in 32 BRAMs. Besides, for INTT operation, 4224 inverse twiddle factors are stored in the other half of the same 32 BRAMs. Similarly, the powers of Ψ/Ψ^{-1} values are stored in 32 BRAMs. Since the proposed work uses Montgomery modular reduction algorithm, the powers of ω, ω^{-1}, Ψ, Ψ^{-1} and $n^{-1} \pmod q$ are multiplied with the necessary powers of Montgomery constant, R, prior to the FPGA in order to eliminate extra modular multiplications in run-time. The powers of ω, ω^{-1}, Ψ, Ψ^{-1} and $n^{-1} \pmod q$ are loaded into the FPGA prior to any operation for a parameter set. If the parameter set is changed, new ω, ω^{-1}, Ψ, Ψ^{-1} and $n^{-1} \pmod q$ values should be loaded.

For NTT/INTT operations, *POL1* and *PSI* BRAMs are not used while the full and half polynomial multiplication operations use all BRAMs. Polynomial multiplication operation requires the coefficients of resulting polynomial from INTT operation to be multiplied with the powers of Ψ^{-1} for post-processing operation as shown in Eqn. 2.11-2.12. INTT operation requires the coefficients of output polynomial to be multiplied with $n^{-1} \pmod q$ as shown in steps 17-19 in Algorithm 2. Therefore, we merged these two operations by multiplying the powers of Ψ^{-1} with $n^{-1} \pmod q$ prior to loading precomputed coefficients into FPGA.

Since the proposed architecture uses 32 PUs and one PU takes two polynomial coefficients as inputs, 64 BRAMs are used to store one polynomial in the proposed architecture. The memory access patterns of the first two stages of 1024-pt NTT operation are shown in Fig. 4.6 as an example. In Fig. 4.6, the numbers in the BRAMs represent the indices of the polynomial coefficients. The NTT operation in Algorithm 1 starts butterfly operation with 0^{th} and $(n/2)^{th}$ coefficients and continues with 1^{st} and $((n/2)+1)^{th}$ coefficients. Since the coefficient pairs (0, 512) to (31,

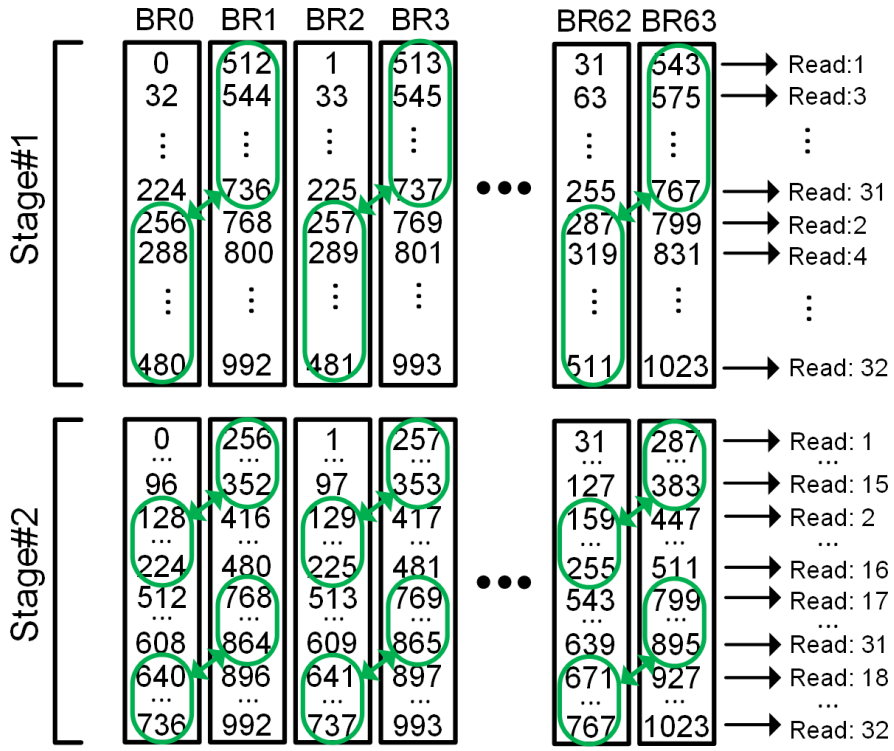


Figure 4.6 Memory Access Pattern for 1024-pt NTT Operation

543) need to be read in the same clock cycles, they are stored in different BRAMs in the proposed architecture. Due to the read/write pattern of the NTT algorithm, the coefficients read in the same stage should be stored in the same BRAMs for the next stage. For example, the coefficient pair (0, 512) read from *BR0* and *BR1* in the first stage should be stored in the *BR0* for the next stage as shown in Fig. 4.6. Since only one coefficient can be stored into one BRAM in a clock cycle, an extra register is placed at the output of modular multiplier hardware in the NTT unit as shown in Fig. 4.2. Therefore, both 0^{th} and 512^{th} coefficients can be stored into the same BRAM in two clock cycles. Since the proposed NTT unit is pipelined, this extra register does not affect the throughput of the proposed architecture.

Similarly, the coefficient pairs (32, 544) to (63, 575) need to be read and stored similar to the coefficient pairs (0, 512) to (31, 543) as shown in Fig. 4.6. This access pattern requires the swap of some coefficients in different memory blocks as shown with green boxes in Fig. 4.6. Therefore, we used an alternating memory access scheme to avoid collisions during the memory store operations.

In this scheme, for an n-pt NTT operation, coefficients stored in the first and second halves of the memory blocks should be read in an alternating way. For example, in the first read operation of the first stage of NTT, the coefficients at address 0 should be read. Then, in the second read operation, coefficients at address $(n/128)$ should

be read instead of address 1. Then, coefficients at addresses 1 and $(n/128)+1$ should be read consecutively and so on. Finally, coefficients at addresses $(n/128)-1$ and $(n/64)-1$ should be read consecutively to finish the first stage of the NTT operation. For the next stage, the same memory pattern should be used for two $(n/2)$-pt NTT operations separately. All NTT operations with different sizes use the same alternating memory access scheme. The proposed flexible memory access scheme handles memory access operations for different n values by generating necessary memory read/write signals. NTT operation takes input polynomial in standard order and produces output polynomial in bit-reversed order while INTT takes input polynomial in bit-reversed order and produces output polynomial in standard order. Therefore, INTT operation uses the same memory access pattern in reverse order.

4.2.4 CPU-FPGA Framework

In order to show the use of the proposed polynomial multiplication architecture as an accelerator in lattice-based homomorphic applications utilizing polynomial multiplication operation, we design a CPU-FPGA framework similar to the framework presented in Chapter 3. For communication between the CPU and the FPGA, we utilize RIFFA driver (Jacobsen et al., 2012), which employs a PCIe connection between CPU and FPGA.

As a case study, the proposed design is used to accelerate the decryption operation of the BFV scheme implemented in the SEAL HE library in a proof of concept accelerator framework. Although we utilize our proposed polynomial multiplier in the decryption operation of the BFV scheme, it can be utilized for accelerating other homomorphic operations of the BFV scheme, which use polynomial multiplication. The accelerator framework includes the SEAL software and an FPGA accelerator that implements our proposed polynomial multiplier architecture. The resulting framework is shown in Fig. 4.7.

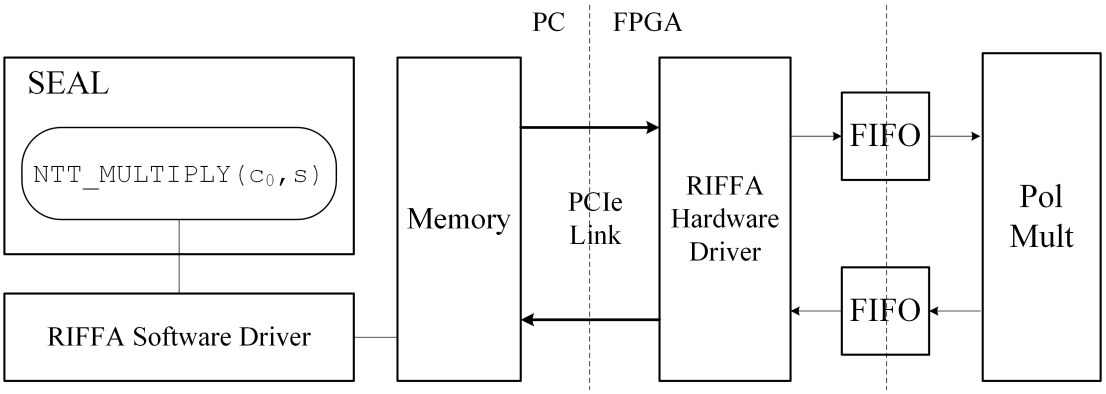


Figure 4.7 CPU-FPGA Framework

4.3 A Case Study: SEAL Library

The proposed hardware architecture, for proof of concept, is utilized to accelerate the polynomial multiplication operation in the decryption operation of the SEAL library. The decryption operation of the BFV scheme in the SEAL is shown in Algorithm 10, which utilizes an NTT-based polynomial multiplication operation. Timing breakdowns of the decryption implementation in the SEAL for the parameter sets (1024,14), (1024,27) and (2048,29) are shown in Table 4.3. The average times for one NTT-based polynomial multiplication operation in the decryption operation of the SEAL are 26.2μs, 28.8μs and 54.6μs for parameter sets (1024,14), (1024,27) and (2048,29), respectively. The timing results are an average of 1000 executions. The timing results are obtained on an Intel i9-7900X CPU @ 3.30 GHz $\times$ 20 with 32 GB RAM using GCC version 7.5.0 in Ubuntu 16.04.6 LTS. As shown in Table 4.3, NTT-based polynomial multiplication operation forms 41.4% to 45.1% of execution time of one decryption operation.

In order to demonstrate the acceleration performance of the proposed polynomial multiplier architecture, we aim to accelerate the decryption function of the SEAL for parameter sets (1024,14), (1024,27) and (2048,29) by offloading polynomial multiplication operation, $\boldsymbol{sc}_1$, into the FPGA that implements the proposed configurable NTT-based polynomial multiplier architecture. Since secret key, $\overline{\boldsymbol{s}}$, is already in the NTT domain, only the polynomial $\boldsymbol{c}_1$ is transformed into the NTT domain using NTT operation. Then, $\overline{\boldsymbol{c}_1}$ is coefficient-wise multiplied with the secret key and INTT operation is applied to transform the resulting polynomial, $\overline{\boldsymbol{sc}}_1$, from NTT domain to polynomial domain. Therefore, this operation is an HPM.

We modified `decrypt` function of the SEAL to integrate our polynomial multiplier hardware as an accelerator into the SEAL. In the modified version, the `decrypt`

Table 4.3 Timing of Decryption Implementation in the SEAL

Operation	**Time (μs)**	**Percentage (%)**
n=1024, k=14, t=256, 256-bit security		
NTT_MULTIPLY	26.2	41.4 %
FASTBCONV	17.7	27.9 %
Others	19.43	30.7 %
n=1024, k=27, t=256, 128-bit security		
NTT_MULTIPLY	28.8	43.2 %
FASTBCONV	19.5	29.2 %
Others	17.4	27.6 %
n=2048, k=29, t=256, 256-bit security		
NTT_MULTIPLY	54.6	45.1 %
FASTBCONV	34.5	28.5 %
Others	31.8	26.4 %

function sends input polynomial $\boldsymbol{c}_1$ to the FPGA, FPGA performs the polynomial multiplication $\boldsymbol{sc}_1$ and returns the resulting polynomial to the CPU. The rest of the operations in the `decrypt` function are performed in the CPU. Precomputed constants such as secret key, $\overline{s}$, and the powers of ω, ω^{-1}, Ψ, Ψ^{-1} are sent to FPGA only once prior to any invocation of `decrypt` function.

To realize our framework, we use Xilinx VC707 Evaluation Board, which includes a PCIe x8 Gen 2 Connector, with XC7VX485T-2FFG1761 FPGA. Xilinx IP Core 7-Series Integrated Block for PCIe provides a 128-bit interface with a 250 MHz clock, which has a 32 Gbps theoretical maximum bandwidth. As shown in Fig. 4.7, we utilize separate FIFO structures for data from the RIFFA driver and data to the RIFFA driver. This approach enables a pipelined architecture for maximizing performance.

SEAL library uses 64-bit integer type for storing the coefficients regardless of the actual bit-size of modulus, q. Since we can work with 16-bit or 32-bit coefficients, we can theoretically pack and send $128/16 = 8$ or $128/32 = 4$ coefficients per cycle, respectively. In order not to complicate memory access in the CPU part, we pack and send $128/32 = 4$ coefficients per cycle instead. In (Jacobsen et al., 2012), it is shown that RIFFA is able to achieve only 76% of the maximum theoretical bandwidth. Therefore, the bandwidth of the PCIe module is assumed to be $\sim$24 Gbps. For the selected parameter sets in the SEAL, the polynomial multiplication in the decryption operation takes 1024 or 2048 coefficients as inputs and CPU can send $(3 \cdot 10^9)/(4 \cdot 1024) = 732420$ or $(3 \cdot 10^9)/(4 \cdot 2048) = 366210$ polynomial multiplication inputs per second, respectively, with 24 Gbps bandwidth.

Although the proposed work uses the RIFFA driver utilizing PCIe connection for establishing communication between the host CPU and FPGA, system on a chip (SoC) platforms with less communication cost can also be used based on the requirements of the target application.

4.4 Results and Comparison

We developed the architecture described in this work into Verilog modules and realized it using Xilinx Vivado 2018.1 tool for the Xilinx VC707 Evaluation Board, which has a Virtex-7 FPGA (XC7VX485T-2FFG1761). The core part of our proposed work uses 39.6K LUTs (9.2%), 21.1K DFFs (2.5%), 96 BRAMs (6.5%) and 224 DSPs (6.2%).

There are many works reported in the literature about the efficient implementation of NTT-based polynomial multiplication operation (Sinha Roy et al., 2018), (Pöppelmann et al., 2015), (Migliore et al., 2018), (Sinha Roy et al., 2019), (Cathébras et al., 2018), (Öztürk et al., 2017), (Chen et al., 2015), (Aysu et al., 2013), (Pöppelmann & Güneysu, 2012), (Feng et al., 2019), (Mert et al., 2020), (Mert et al., 2019). In (Sinha Roy et al., 2018) and (Sinha Roy et al., 2019), the authors propose an accelerator framework for homomorphic operations. They optimize their architecture for single q and n. They also utilize a memory-based algorithm for integer modular reduction operation. Our proposed polynomial multiplier, on the other hand, supports multiple parameter sets and can be utilized as an accelerator in different applications. The works in (Mert et al., 2019) and (Mert et al., 2020) present similar architectures with our proposed polynomial multiplier architecture. However, they support a single parameter set, which has a very low multiplicative depth for HE applications. The architecture in (Pöppelmann et al., 2015) works with very large k, namely $k = 512$. However, compared to our work, it lacks parallelism due to large integer arithmetic. In (Pöppelmann & Güneysu, 2012), the authors propose an architecture for fixed q and n, which enable a highly-optimized integer modular reduction architecture. However, their parameter set has low multiplicative depth and their architecture is not reconfigurable. The work in (Migliore et al., 2018) utilizes the Karatsuba algorithm instead of NTT for polynomial multiplication. It works for a single parameter set, which has a multiplicative depth of four. Öztürk *et al.* proposes a highly parallelized NTT-based polynomial multiplier architecture with single parameter set (Öztürk et al., 2017). They utilize the conventional Barrett algorithm

for integer modular reduction and perform separate reduction operation for polynomial reduction. In (Aysu et al., 2013), the authors implement a memory-efficient NTT algorithm, which computes twiddle factors on-the-fly. However, it uses a single processing unit and supports one parameter set. In (Chen et al., 2015), the authors implement the same polynomial multiplier architecture for four different parameter sets, where n is ranging from 256 to 2048. However, they use a single processing unit and our proposed polynomial multiplier architecture shows better performance than their work. In (Cathébras et al., 2018), the authors adopt a hardware generator tool for generating NTT hardware for a given parameter set. However, their generated NTT hardware does not support run-time configurability. Although some of these works perform slightly different operations than polynomial multiplication, we only report the implementation results for the NTT and polynomial multiplication parts of these works. The implementation results of the works in the literature and the work proposed in this chapter are reported in Table 4.4 and Table 4.5, respectively.

The proposed configurable architecture in this chapter is the only work in the literature supporting multiple k and n for FPGA platforms at the time this work is presented. Other works in the literature either are designed for fixed q or do not support multiple n values. The proposed work shows better area and timing performance than most of the works in the literature. Although the works in (Cathébras et al., 2018), (Mert et al., 2020), (Mert et al., 2019) show better timing performance than the proposed work in this chapter, they do not support multiple q and n values and the proposed work in this chapter uses less FPGA resources. The polynomial multiplier in (Feng et al., 2019) shows both better area and timing performance; however, it is designed and optimized for fixed q. Although there are other implementations (Banerjee et al., 2019), (Song et al., 2018), (Fritzmann & Sepúlveda, 2019) supporting multiple n and q values, these works target ASIC platforms. Therefore, they are not included in the comparison.

Our proposed architecture is deployed into a framework that aims to accelerate the decryption operation of the BFV scheme in the SEAL. For proof of concept, we select three parameter sets (1024,14), (1024,27), (2048,29) of the SEAL and offload polynomial multiplication operation in the `decrypt` function of the SEAL into the FPGA that implements our configurable NTT-based polynomial multiplier architecture. Then, we obtained performance numbers on a real CPU-FPGA heterogeneous application setting. The polynomial multiplication operation in the `decrypt` function of the SEAL with parameter sets (1024,14), (1024,27), (2048,29) yields 26.2μs, 28.8μs, 54.6μs, respectively, in pure software implementation with host computer as specified in Section 4.3. Compared to the software, the proposed configurable polynomial multiplier architecture performs the same polynomial multiplication op-

Table 4.4 Comparative Table (FPGA Resources)

Work	**Platform**	n	$\lceil \log_2(q) \rceil$	**LUTs/DSPs/BRAMs**	**Clock (MHz)**
(Sinha Roy et al., 2018)	Virtex-6	65536	30	72K / 250 / 106	100
(Pöppelmann et al., 2015)	Virtex-7	4096	125	69K / 144 / –	100
(Migliore et al., 2018)	Stratix-V	2560	125	30K / 100 / –	331
(Sinha Roy et al., 2019)	Zynq	4096	30	64K / 200 / 400	225
(Öztürk et al., 2017)	Virtex-7	32768	32	219K / 768 / 193	250
(Pöppelmann & Güneysu, 2012)[a]	Spartan-6	1024	30	1644 / 1 / 6.5	200
(Aysu et al., 2013)[a]	Spartan-6	1024	17	250 / 3 / 2 240 / 3 / 2 250 / 3 / 2	–
(Chen et al., 2015)[a]	Spartan-6	256 1024	21 31	2829 / 4 / 4 6689 / 4 / 8	247 241
(Cathébras et al., 2018)	Virtex-7	4096	30	54K / 517 / 208	200
(Feng et al., 2019)[a]	Spartan-6	256 512	21 23	14K / 128 / 1 18K / 128 / 2.5	233 200
(Mert et al., 2020)	Virtex-7	1024	32	77K / 952 / 325.5 67K / 599 / 129	200
(Mert et al., 2019)	Spartan-6 Virtex-7	1024	32	1.2K / 14 / 14 33.8K / 476 / 227.5	212 200
Ours[*]	Virtex-7	256	16	39.6K / 224 / 96	150
		512			
		1024			
		1024	32		
		2048			
		4096			

[a]: Uses fixed q.

[*]: Excluding RIFFA Hardware Driver and input/output FIFOs shown in Fig. 4.7).

erations in 4.15μs, 4.15μs, 7.5μs, respectively, excluding I/O operations. Compared to the software, we achieved up to 7× speedup, excluding I/O operations for the polynomial multiplication operation. With our setting, sending or receiving one polynomial of degree 1024 and 2048 from the CPU to FPGA via DMA takes 1.3μs and 2.67μs, respectively, on average. Therefore, our accelerator-based implementation, including I/O overhead, yields 6.75μs, 6.75μs, 12.96μs, respectively, for selected parameter sets. Compared to the software, we achieved up to 4.2× speedup, including I/O overhead. The `decrypt` function of the SEAL with parameter sets (1024,14), (1024,27), (2048,29) yields executions times of 63.33μs, 65.7μs, 120.9μs, respectively, in pure software implementation as shown in Table 4.3. Compared to the software, the proposed framework with configurable polynomial multiplier architecture performs the same decryption operations in 43.93μs, 43.7μs, 79.26μs, respectively, including I/O overhead. Compared to the software, we still achieve up to 1.52× speedup, including I/O overhead for decryption operation.

The proposed polynomial multiplier architecture is shown to be effective as an accelerator for homomorphic applications. Since our proposed work aims to support a range of applications with a focus on accelerating homomorphic operations, it is

Table 4.5 Comparative Table (Performance)

Work	**Platform**	n	K	**Latency (μs)**	
				NTT	**Pol.Mul.**
(Sinha Roy et al., 2018)	Virtex-6	65536	30	–	3376
(Pöppelmann et al., 2015)	Virtex-7	4096	125	–	1960
(Migliore et al., 2018)	Stratix-V	2560	125	–	583
(Sinha Roy et al., 2019)	UltraScale	4096	30	73	171
(Öztürk et al., 2017)	Virtex-7	32768	32	51	152
(Pöppelmann & Güneysu, 2012)[a]	Spartan-6	1024	30	–	110
(Aysu et al., 2013)[a]	Spartan-6	1024	17	–	25 50 100
(Chen et al., 2015)[a]	Spartan-6	256 1024	21 31	– –	6 33
(Cathébras et al., 2018)	Virtex-7	4096	30	–	10
(Feng et al., 2019)[a]	Spartan-6	256 512	21 23	– –	0.94 1.77
(Mert et al., 2020)	Virtex-7	1024	32	0.4 0.7	0.96 1.40
(Mert et al., 2019)	Spartan-6 Virtex-7	1024	32	– –	37.67 1.25
Ours	Virtex-7	256	16	0.69	1.99
		512		1.02	3.12
		1024		1.66	5.43
		1024	32	1.66	5.43
		2048		3.01	10.25
		4096		5.84	20.39

[a]: Uses fixed q.

not meaningful to compare our architecture with highly-optimized hardware and/or software implementations of specific NTT-friendly lattice-based post-quantum cryptosystems such as CRYSTALS-Kyber (v1) ((Botros, Kannwischer & Schwabe, 2019), (Seiler, 2018)), NewHope ((Seiler, 2018), (Alkim, Jakubeit & Schwabe, 2016)) and qTESLA ((Alkim, Barreto, Bindel, Kramer, Longa & Ricardini, 2019)).

The proposed NTT-based polynomial multiplier architecture supports lattice-based post-quantum cryptosystems with NTT-friendly parameter set (Bos et al., 2018), (Alkim et al., 2019), (Alkim et al., 2016). However, not all lattice-based post-quantum cryptosystems such as SABER (D'Anvers et al., 2018) supports NTT operation due to their parameter sets. The proposed architecture in this chapter can be extended to perform polynomial multiplication for cryptosystems without NTT-friendly coefficient modulus with slight modifications as shown in (Dai & Sunar, 2015).

Although the proposed work supports relatively small parameter sets for homomorphic operations, homomorphic multiplication in particular, and has a low multiplicative depth, our polynomial multiplier can be easily utilized in designs with larger degrees and depths. Many works (Microsoft, 2019), (Sinha Roy et al., 2019), (Sinha Roy et al., 2018) use one large modulus (Q), which is mapped into smaller coprime moduli (q_0, q_1, $q_2 \ldots$). In order to increase the parallelism, they employ CRT (Bonehet al. , 1999), which transforms each coefficient in Q into multiple smaller coefficients in $\{q_0, q_1, q_2 \ldots\}$, and perform operations in small moduli separately in parallel instead of in Q. For example, the work in (Sinha Roy et al., 2019) with a multiplicative depth of four uses six small 30-bit coprime moduli instead of one large 180-bit modulus with $n = 4096$. Similarly, SEAL uses the same approach for implementing homomorphic operations. Therefore, the polynomial multiplier proposed in this work with a small modulus size can be utilized for accelerating operations with larger parameter sets.

Our run-time configurable architecture is optimized for the entire proof of concept framework. Therefore, the frequency that we are using does not need to be the maximum possible frequency. Our datapath speed is optimized according to the I/O bandwidth of the PCIe connection. Increasing the frequency any further will also make our datapath faster than I/O, which will still add stall cycles to our pipeline.

4.5 Summary

In this chapter, we present the design and FPGA implementation of a run-time configurable and highly parallelized NTT-based polynomial multiplier architecture, which is shown to be effective as an accelerator for lattice-based homomorphic schemes. The proposed architecture supports six different parameter sets.

For proof of concept, we utilize our architecture in a CPU-FPGA framework for the BFV HE scheme, adopting a hardware/software co-design approach for accelerating the polynomial multiplication operation in the decryption operation of the BFV scheme implemented in the SEAL library. We used the Xilinx VC707 Evaluation Board utilizing a Virtex-7 FPGA for our implementation. We improved the latency of NTT-based polynomial multiplications in decryption operation by up to 7× and 4.2× compared to the implementation in SEAL, excluding and including I/O

overhead, respectively. In addition to that, we improved the latency of decryption operation by up to 1.52× compared to its pure software implementation in SEAL including I/O overhead.

5. A HIGH PERFORMANCE HOMOMORPHIC MULTIPLICATION ARCHITECTURE FOR THE BFV SCHEME

In this chapter, we present a high performance and scalable hardware architecture that performs homomorphic multiplication and relinearization operations of the full RNS variant of the BFV scheme (Bajard et al., 2017) for parameters $n = 4096$, $\lceil \log_2(q) \rceil = 93$ and $\lceil \log_2(t) \rceil = 17$ with a multiplicative depth of one. The proposed architecture employs three NTT cores where each core consists of 16 unified butterfly units. The proposed architecture performs one homomorphic multiplication and one relinearization operation in 0.338 ms and 0.079 ms, respectively, excluding I/O overhead. Compared to the SEAL library (`v3.5`) (Microsoft, 2020) running on a laptop with Intel Core i7-9750H @ 2.60GHz x 12 using a single thread, the proposed architecture shows up to 18.4× and 16.1× performance improvements for the homomorphic multiplication and relinearization operations, respectively.

5.1 Introduction

The BFV scheme has emerged as one of the most promising and popular HE schemes for the applications working with integers (Viand, Jattke & Hithnawi, 2021). The original BFV scheme, which is also referred to as textbook-BFV scheme (Fan & Vercauteren, 2012), has high computational complexity. It requires high precision integer arithmetic for very large coefficient modulus q which needs to be employed on practical HE applications. It also uses costly division and rounding operations. To reduce its complexity and make it fast, there have been various efforts in the literature (Bajard et al., 2017), (Halevi et al., 2019), (Bajard et al., 2019), (Takeshita et al., 2020) proposing algorithmic improvements for the BFV scheme. These works adopt RNS or similar arithmetic tools into the textbook-BFV scheme for improv-

ing the practicality and the performance of the BFV scheme. In addition to the works focusing on the algorithmic-level improvements, there are several efforts in the literature for presenting efficient hardware/software implementations targeting acceleration of various SWHE and FHE schemes. Among these works, the implementations aiming at the BFV scheme are limited.

To the best of our knowledge, there are three, two, and two works targeting FPGA, GPU, and CPU platforms for efficient implementation of the BFV scheme in the literature. Roy *et al.* (Sinha Roy et al., 2018) proposes an FPGA implementation of homomorphic multiplication and addition operations of the textbook-BFV scheme. Their implementation targets a very large multiplicative depth with a security level of 85-bit. Their parameter set is $n = 32768$, $\lceil \log_2(q) \rceil = 1228$ and $t = 2$. They employ CRT for eliminating multi-precision integer arithmetic in polynomial arithmetic. Roy *et al.* (Sinha Roy et al., 2019) proposes another implementation of homomorphic multiplication and addition operations for the RNS variant of the BFV scheme proposed by Halevi *et. al* (Halevi et al., 2019) on FPGA. They target parameter set $n = 4096$, $\lceil \log_2(q) \rceil = 180$ and $t = 2$ with 80-bit security and multiplicative depth of 4. Turan *et al.* (Turan, Roy & Verbauwhede, 2020) implements the work presented in (Sinha Roy et al., 2019) on the AWS F1 server using the same parameter and security settings. Badawi *et al.* (Badawi, Veeravalli, Mun & Aung, 2018) proposes a GPU implementation of various homomorphic operations of the full RNS variant of the BFV scheme proposed by Bajard *et al.* (Bajard et al., 2017). Their implementation supports the parameter set $n = 4096$ and $\lceil \log_2(q) \rceil = 180$. In (Badawi, Polyakov, Aung, Veeravalli & Rohloff, 2018), Badawi *et al.* provides a comparison of the RNS methods for the BFV scheme proposed by Bajard (Bajard et al., 2017) and Halevi (Halevi et al., 2019) on CPU and GPU platforms for various parameter sets. Takeshita *et al.* (Takeshita, Reis, Gong, Niemier, Hu & Jung, 2020) takes advantage of compute-enabled RAM and proposes various optimizations for the Bajard's full RNS BFV scheme (Bajard et al., 2017). In (Reis, Takeshita, Jung, Niemier & Hu, 2020), Reis *et al.* uses computing-in-memory approach for implementing the BFV scheme. This method uses no RNS approach since it employs a modulus in the form of power of two.

With similar motivations to existing works in the literature, in this chapter, we present the first FPGA implementation of homomorphic multiplication operation of the full RNS variant of the BFV scheme proposed by Bajard *et al.* (Bajard et al., 2017). Our proposed architecture targets a multiplicative depth of one with parameters $n = 4096$, $\lceil \log_2(q) \rceil = 93$, $\lceil \log_2(t) \rceil = 17$ and it provides a security level more than 128-bit. The proposed architecture in this chapter follows the implementation in the SEAL library (`v 3.5.1`) (Microsoft, 2020), which uses Bajard's technique (Ba-

jard et al., 2017). The proposed FPGA implementation shows more than one order of magnitude performance improvement compared to the SEAL library while using 34% of the resources in the Xilinx VC709 Evaluation Board.

The rest of the chapter is organized as follows. Section 5.2 introduces the full RNS variant of the BFV scheme as implemented in the SEAL library. Section 5.3 presents the proposed hardware architecture. Finally, Section 5.4 presents the implementation results and the comparison with the literature, and Section 5.5 concludes the chapter.

5.2 Full RNS Variant of the BFV Scheme

In this section, we explain the homomorphic multiplication and relinearization operations of the full RNS variant of the BFV scheme implemented in SEAL library (Microsoft, 2020).

5.2.1 Homomorphic Multiplication

Bajard *et al.* (Bajard et al., 2017) proposes a variant of the BFV scheme which uses RNS for avoiding multi-precision integer arithmetic and complex divide-and-round operation. In the proposed work, instead of using one large coefficient moduli q, a set of smaller modulus q_i, which are also referred to as RNS bases, are used. This eliminates multi-precision arithmetic and enables performing operations in RNS bases in parallel. In addition to that, their work eliminates the rounding operation by introducing an approximate rounding method that employs a flooring function with some approximation error. Since the BFV scheme is using randomly generated errors for its operations, the error introduced by the approximate rounding operation does not affect the validity of the scheme. The high-level diagram of the homomorphic multiplication operation implemented in the SEAL library is shown in Fig. 5.1. It should be noted that the implementation of the BFV scheme in the SEAL library drops the last RNS base during the encryption operation. For example, a ciphertext encrypted using the BFV scheme using three RNS bases q_0, q_1, q_2 will not have any component in base q_2. The homomorphic multiplication operation takes two

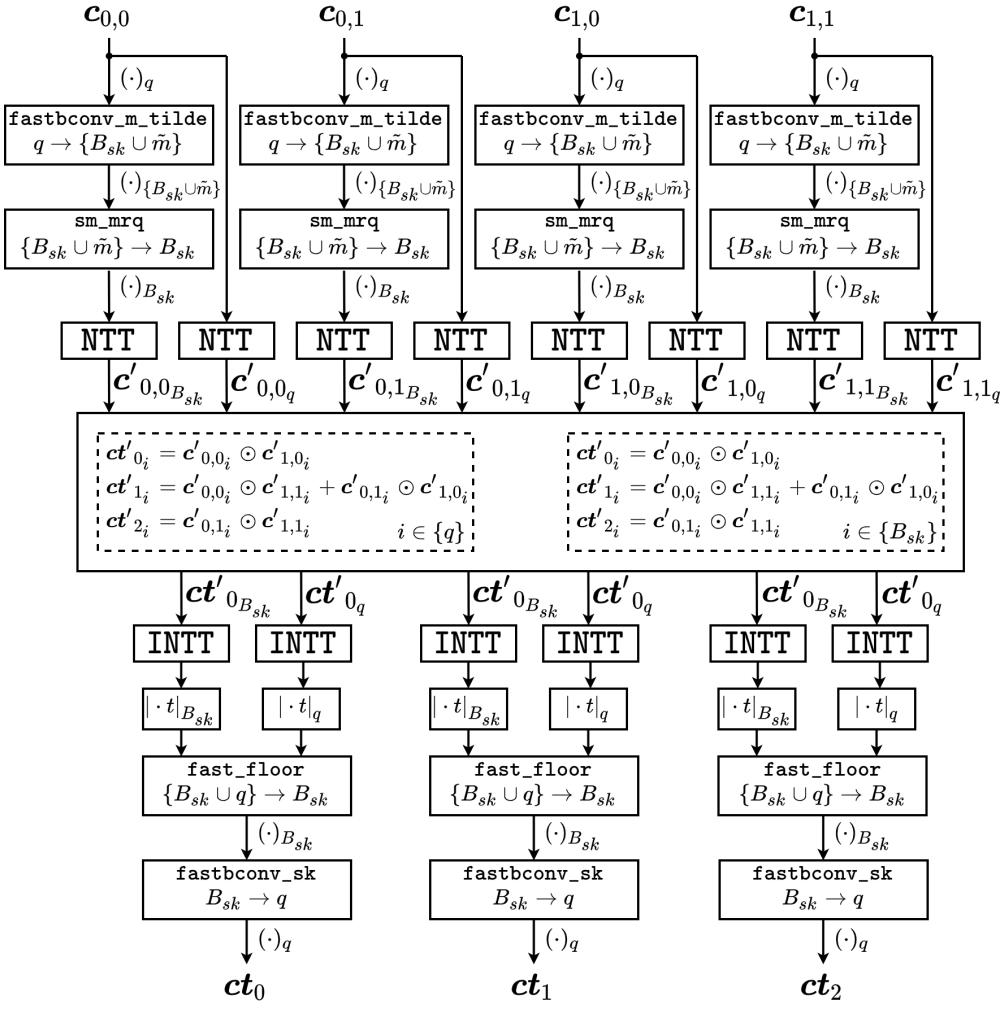


Figure 5.1 The Flow of Homomorphic Multiplication Operation in the BFV Scheme

ciphertexts as inputs where each ciphertext consisting of two components in RNS bases and produces one ciphertext as output with three components in RNS bases.

Although input and output ciphertexts of the homomorphic multiplication operation have components in RNS bases, Bajard's technique introduces additional *auxiliary bases* B and m_{sk} for using during the homomorphic multiplication operation. The RNS bases (coefficient modulus q), auxiliary bases B and m_{sk} consist of r ($\{q_0, q_1, \ldots, q_{r-1}\}$), l ($\{B_0, B_1, \ldots, B_{l-1}\}$) and one prime modulus, respectively. Auxiliary bases B and m_{sk} together are also referred as B_{sk} base. The RNS and auxiliary bases should be pairwise co-prime and NTT-friendly primes. Bajard's technique requires converting an integer from a base to another base (i.e. from q to B_{sk}), which is performed using an operation called *fast base conversion* shown in Eqn. 5.1. This conversion introduces some approximation errors in the form of extra multiples of input base (i.e. q) being in the resulting integers with output base (i.e.

B_{sk}). Therefore, a correction operation is required after the fast base conversion.

$$\texttt{fastbconv}(\{a_0, a_1, \ldots, a_{r-1}\}, q, B) = \{\sum_{i=0}^{r} |a_i \cdot \frac{q_i}{q}|_{q_i} \cdot \frac{q}{q_i} \pmod{B_i}\} \tag{5.1}$$

The full RNS variant of the BFV scheme starts homomorphic multiplication operation by converting the components of the input ciphertexts ($\boldsymbol{c}_{0,0}$, $\boldsymbol{c}_{0,1}$, $\boldsymbol{c}_{1,0}$, $\boldsymbol{c}_{1,1}$ in Fig. 5.1) in RNS base (q) to $B_{sk} \cup \tilde{m}$ base, where $\tilde{m}$ is an additional base used to remove conversion error introduced by the fast base conversion operation. This conversion is performed for enabling the multiplication of ciphertext components with each other using the RNS approach without causing any overflow (Bajard et al., 2017). The first base conversion operation is followed by a small Montgomery reduction operation which cancels out the approximation error from the resulting ciphertext components. This operation eliminates the error and converts the components in base $B_{sk} \cup \tilde{m}$ to base B_{sk}. Before the multiplication of ciphertext components with each other, the NTT operation is performed for all ciphertext components. Then, the ciphertext multiplication is performed as shown in Fig. 5.1 and the resulting ciphertext components are transformed into the polynomial domain using INTT operation. The NTT, ciphertext multiplication, and INTT operations are performed for each RNS base and auxiliary base, which can be performed in parallel.

The resulting ciphertext components are then multiplied with the plaintext modulus t. For division operation, a fast flooring function is used which converts ciphertext components from bases $q \cup B_{sk}$ to B_{sk} base. Finally, Shenoy-Kumaresan conversion (Shenoy & Kumaresan, 1989) is used to remove the approximate rounding error and convert the ciphertext components from bases $q \cup B_{sk}$ to the RNS base. The homomorphic multiplication implementation in SEAL with the BFV scheme using r RNS bases and $l+1$ auxiliary bases performs $4 \cdot (r + l - 1)$ NTT and $3 \cdot (r + l - 1)$ INTT operations, where each NTT and INTT operations can be performed in parallel. In addition to these, it also performs modular addition, subtraction and multiplication operations over $\boldsymbol{R}_{n,q_i}$, $\boldsymbol{R}_{n,B_{sk}}$, $\boldsymbol{R}_{n,\tilde{m}}$ rings.

5.2.2 Relinearization

Although SEAL library follows Bajard's relinearization scheme for the relinearization operation implemented in its earlier versions (`v3.1` and earlier), it uses a combination of Bajard's technique (Bajard et al., 2017), Halevi's technique (Halevi et al.,

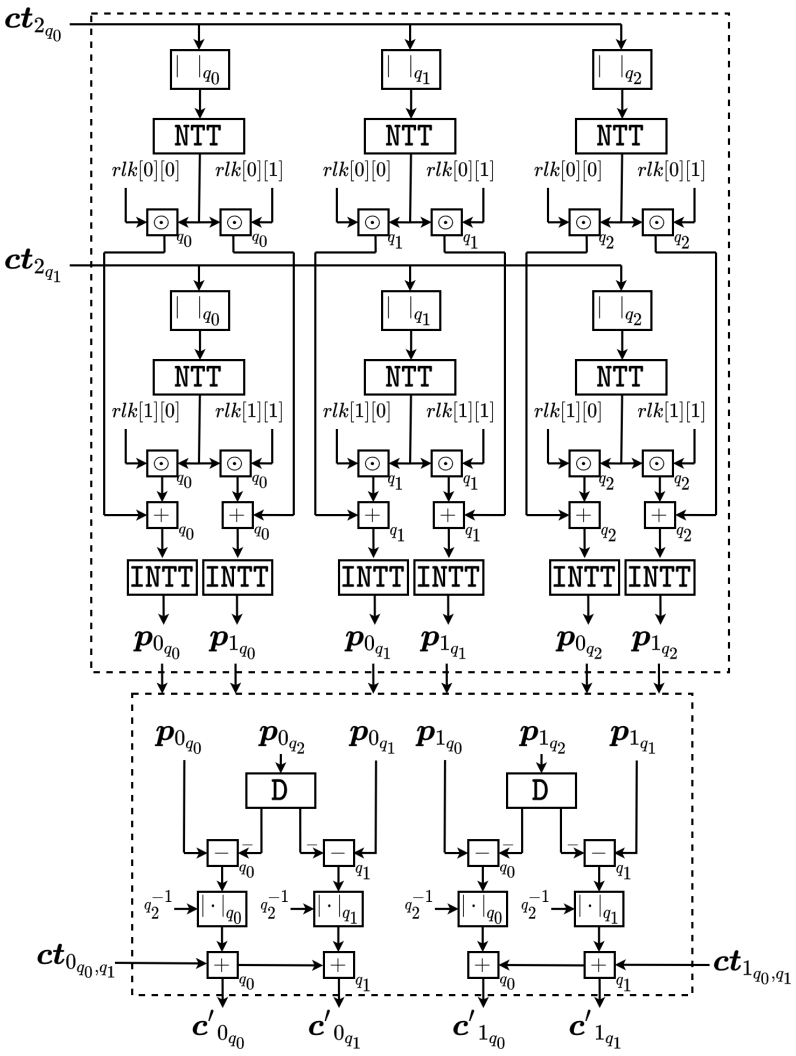


Figure 5.2 The Flow of Relinearization Operation in the BFV Scheme

2019) and special moduli technique proposed in (Chen, Dai, Kim & Song, 2019) in its latest version. Bajard's technique performs decomposition operation differently than the procedure presented in Section 2.4.1. Instead of decomposing the last ciphertext element into a base w, it uses RNS bases, q_i (Bajard et al., 2017). Halevi's technique further improves Bajard's work and eliminates some redundant operations during the relinearization operation (Halevi et al., 2019). The special moduli technique improves the noise performance of the relinearization operation; however, it requires the last RNS moduli to be the largest RNS moduli (Chen et al., 2019). The high-level diagram of the relinearization operation implemented in SEAL library for coefficient modulus q with three RNS bases (q_0, q_1, q_2) is shown in Fig. 5.2, where $|\ |_{q_i}$ represents the modular reduction operation by moduli q_i. The relinearization operation takes one ciphertext with three components in RNS bases as inputs and it produces one ciphertext with two components in RNS bases as output.

For a coefficient modulus q consisting of three RNS bases (q_0, q_1, q_2), the relinearization operation works with all three RNS bases while the homomorphic encryption operation works with only the first two RNS bases (q_0, q_1). The relinearization key is generated slightly different than the relinearization key generation procedure presented in Section 2.4.1. For a q with r RNS bases, it consists of $2 \cdot (r-1)$ components where each component has r RNS bases. For $r = 3$ as shown in Fig. 5.2, it consists of $2 \cdot 2 = 4$ components where each component consists three polynomials in $\boldsymbol{R}_{n,q_0}$, $\boldsymbol{R}_{n,q_1}$, $\boldsymbol{R}_{n,q_2}$ rings. The relinearization operation has two parts: (i) RNS decomposition and (ii) modulus switching. In the RNS decomposition part, the last component of the ciphertext produced by the homomorphic multiplication operation ($\boldsymbol{ct}_2$ in Fig. 5.2) in $(r-1)$ RNS bases (q_0, q_1) is decomposed into r RNS bases (q_0, q_1, q_2) and multiplied with the components of relinearization key. Since the relinearization key is generated and stored in the NTT domain, only the decomposed ciphertext components are transformed into the NTT domain before the multiplication operation. After the multiplication operation, the resulting polynomials are transformed into the polynomial domain using the INTT operation.

In the second part, the decomposed elements are combined together as shown in Fig. 5.2.2, where `D` represents the conversion of a polynomial from the last RNS base (q_2) to other RNS bases (q_0, q_1). Finally, the resulting polynomials are added with the first two components of the ciphertext produced by the homomorphic multiplication operation ($\boldsymbol{ct}_0$ and $\boldsymbol{ct}_1$ in Fig. 5.2). The relinearization implementation in SEAL with a coefficient modulus q using r RNS bases uses $r \cdot (r-1)$ NTT and $r \cdot (r-1)$ INTT operations, where each NTT and INTT operations can be performed in parallel. In addition to these, it performs modular reduction, addition, subtraction and multiplication operations over $\boldsymbol{R}_{n,q_i}$ rings, where $i = 0, 1, \ldots, l-1$.

5.3 Homomorphic Multiplication Architecture

In this section, the proposed hardware architecture and its main arithmetic blocks are explained starting from the NTT core, which employs 16 unified butterfly units. Then, we present the overall design and the efficient scheduling scheme used for the homomorphic multiplication and relinearization operations.

5.3.1 Parameter Set

In our design, we target a proof-of-concept hardware architecture with a multiplicative depth of one. Therefore, we select a proper parameter set with $n = 4096$, $\lceil \log_2(q) \rceil = 93$ and $\lceil \log_2(t) \rceil = 17$, where coefficient modulus q consisting of three 31-bit NTT-friendly primes q_i, $i \in \{0, 1, 2\}$. The selected parameter set is verified to have a multiplicative depth of one using SEAL library and it provides a security level of at least 128-bit (Albrecht, Player & Scott, 2015). The SEAL library generates and uses 60-bit primes for the auxiliary bases, which are used during the homomorphic multiplication as explained in Section 5.2.1. Since our RNS bases (q_0, q_1, q_2) are 31-bit primes, we modify the SEAL library accordingly such that it generates and uses 32-bit primes for the auxiliary bases (B and m_{sk}) as well. This eliminates the need for an arithmetic unit supporting both 31-bit and 60-bit integer arithmetic. Therefore, in our hardware architecture, all arithmetic units are working with 32-bit integers. For auxiliary bases, we use four 32-bit NTT-friendly primes B_0, B_1, B_2, m_{sk} and $\tilde{m}$, which is constant and equal to 2^{32}.

5.3.2 NTT Core

The NTT core is designed to perform all necessary modular arithmetic operations used during the homomorphic multiplication and relinearization operations detailed in Section 5.2, which require NTT, INTT, modular addition, modular subtraction, and modular multiplication operations. Since we target a high-performance hardware architecture, we decided to use MNTT and MINTT algorithms which require no pre-processing and post-processing operations at the expense of utilizing a unified butterfly unit as explained in Section 2.5.2. The MNTT and MINTT algorithms are shown in Algorithm 11 and Algorithm 12, respectively. The MNTT and MINTT algorithms perform CT and GS butterfly operations, which are shown in the steps 8-13 of Algorithm 11 and Algorithm 12, respectively.

For the reasons already explained in Section 4.2.3, NTT and INTT operations can be implemented using multiple butterfly units working in parallel. In our design with a ring size of 4096, we can employ multiple butterfly units in parallel up to 2048. Although we target a high-performance hardware architecture, we also want to keep the area cost reasonable. Therefore, the number of unified butterfly units in one NTT core is selected as 16.

Algorithm 11 Merged Forward NTT Algorithm Longa & Naehrig (2016)

Input: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in natural order
Input: primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$, $n = 2^l$
Output: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order

1: $t = n$
2: **for** $(m = 1;\ m < n;\ m = 2 \cdot m)$ **do**
3: $\quad t = t/2$
4: $\quad$ **for** $(i = 0;\ i < m;\ i = i + 1)$ **do**
5: $\quad\quad j_1, j_2 = 2 \cdot i \cdot t, j_1 + t - 1$
6: $\quad\quad W = \psi^{\mathtt{br}(m+i,l)} \pmod{q}$
7: $\quad\quad$ **for** $(j = j_1;\ i \leq j_2;\ j = j + 1)$ **do**
8: $\quad\quad\quad U = \boldsymbol{a}[j]$
9: $\quad\quad\quad V = \boldsymbol{a}[j + t] \cdot W \pmod{q}$
10: $\quad\quad\quad N0 = (U + V) \pmod{q}$
11: $\quad\quad\quad N1 = (U - V) \pmod{q}$
12: $\quad\quad\quad \boldsymbol{a}[j] = N0$
13: $\quad\quad\quad \boldsymbol{a}[j + t] = N1$
14: $\quad\quad$ **end for**
15: $\quad$ **end for**
16: **end for**
17: **return** $\boldsymbol{a}$

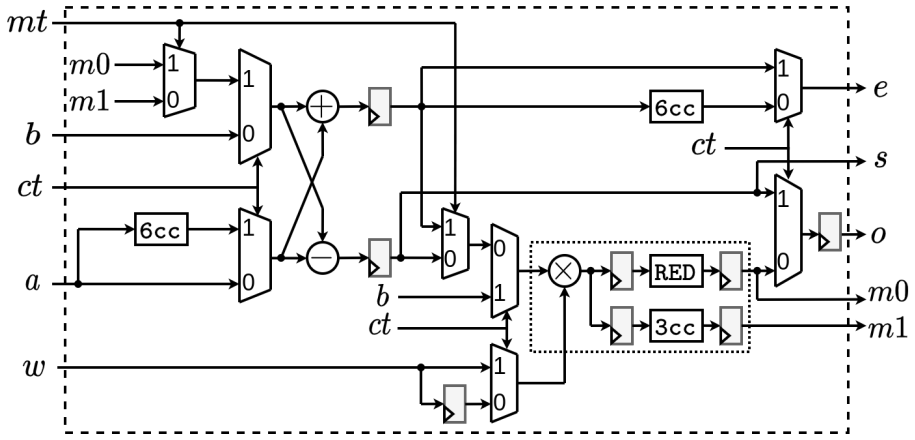


Figure 5.3 Unified Butterfly Unit

The proposed unified butterfly unit performs both CT and GS butterfly operations and it is shown in Fig. 5.3. It uses one modular multiplier, one modular adder, one modular subtractor, and extra logic units for providing reconfigurability. The modular multiplier unit, which is shown with a dashed box in Fig. 5.3, uses a 32-bit integer multiplier and the word-level Montgomery modular multiplication algorithm presented in Section 3.2.2.1 and Section 3.2.2.2, respectively. Since we work with a ring size of 4096, we select word size, w, as 13 with a repeat count, L, as 3. Thus, our word-level Montgomery modular multiplier unit supports NTT-friendly primes with bit sizes ranging from 27 to 39. The proposed modular multiplier uses

Algorithm 12 Merged Inverse NTT Algorithm Longa & Naehrig (2016)

Input: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order
Input: modular inverse of primitive $2n$-th root of unity $\psi^{-1} \in \mathbb{Z}_q$, $n = 2^l$
Output: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in natural order

$t = 1$
for $(m = n;\ m > 1;\ m = m/2)$ **do**
    $j_1, h = 0, m/2$
    for $(i = 0;\ i < h;\ i = i + 1)$ **do**
        $j_2 = j_1 + t - 1$
        $W = \psi^{\mathtt{br}(h+i,l)} \pmod{q}$
        for $(j = j_1;\ i \leq j_2;\ j = j + 1)$ **do**
            $U = \overline{\boldsymbol{a}}[j]$
            $V = \overline{\boldsymbol{a}}[j + t]$
            $N0 = (U + V) \pmod{q}$
            $N1 = (U - V) \cdot W \pmod{q}$
            $\overline{\boldsymbol{a}}[j] = N0$
            $\overline{\boldsymbol{a}}[j + t] = N1$
        end for
        $j_1 = j_1 + 2 \cdot t$
    end for
    $t = 2 \cdot t$
end for
for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $\overline{\boldsymbol{a}}[i] \leftarrow \overline{\boldsymbol{a}}[i] \cdot n^{-1} \pmod{q}$
end for
return $\overline{\boldsymbol{a}}$

four and three DSP units for integer multiplier unit and word-level Montgomery modular reduction unit, respectively. Overall, it uses seven DSP units and it has six clock cycles latency. Since it is pipelined, it can perform one modular multiplication operation per clock cycle after filling the pipeline.

In addition to the CT and GS butterfly operations, the proposed unified butterfly unit can be configured to perform modular addition, modular subtraction, and modular multiplication operations in RNS bases (mod q_i), auxiliary bases ((mod B_i) and (mod m_{sk})) and (mod 2^{32}), which are used during the homomorphic multiplication operation. The butterfly unit takes three 32-bit coefficients (a, b, w) and two control signals (ct, mt) as inputs, and it produces five 32-bit coefficients (e, o, s, $m0$, $m1$) as outputs. According to the desired functionality, the control signals ct and mt should be set as shown in Table 5.1, where unused configurations are shown with a dash (–). The outputs e, o, s with configuration $(ct, mt) = (0, 0)$ and $(ct, mt) = (1, 0)$ are used for NTT and INTT operations respectively. The other configurations are used for the rest of the arithmetic operations performed in RNS bases, auxiliary bases, and (mod 2^{32}). For example, modular multiplication operation in (mod q_i)

Table 5.1 Unified Butterfly Unit Configuration

Out.	$mt = 0$		$mt = 1$	
	$ct = 0$	$ct = 1$	$ct = 0$	$ct = 1$
e	$a + b \pmod{q_i}$	$a + b \cdot w \pmod{q_i}$	$a + b \pmod{q_i}$	$a + b \cdot w \pmod{2^{32}}$
o	$(a - b) \cdot w \pmod{q_i}$	$a - b \cdot w \pmod{q_i}$	$(a + b) \cdot w \pmod{q_i}$	$a - b \cdot w \pmod{2^{32}}$
s	–	$a - b \cdot w \pmod{q_i}^*$	–	$a - b \cdot w \pmod{2^{32}}^*$
$m0$	$(a - b) \cdot w \pmod{q_i}^*$	–	$(a + b) \cdot w \pmod{q_i}^*$	–
$m1$	–	–	$(a + b) \cdot w \pmod{2^{32}}$	–

*: Outputs one clock cycle before s

or $\pmod{B_i}$ can be performed using $(ct, mt) = (0, 0)$ configuration, giving the first operand, 0 and the second operand to a, b and w inputs respectively, and reading o output.

The MNTT and MINTT operations are performed using the same alternating memory access pattern explained in Section 4.2.3 and Section 7.3.3. Every butterfly unit takes two coefficients and one constant for MNTT and MINTT operations. Therefore, one NTT core with 16 butterfly units needs 32 BRAMs for storing input and output coefficients, and 16 BRAMs for storing the constants which are the precomputed powers of $2n$-th root of unity. Due to the alternating memory access pattern, the BRAMs storing input and output coefficients should have at least a depth of 256 while the BRAMs storing the $2n$-th root of unity powers should have a depth of 1535. All BRAMs have a 32-bit data length. One NTT core can perform both MNTT and MINTT operations in 1536 clock cycles in a pipelined manner.

5.3.3 Overall Design and Scheduling

The proposed hardware architecture works with two RNS bases (q_0, q_1), three auxiliary bases (B_0, B_1, m_{sk}) and additional $\tilde{m}$ base which is only used for small Montgomery reduction operation. As explained in Section 5.2.1, NTT, INTT, and ciphertext multiplication operations need to be performed for both RNS and auxiliary bases, and they can be performed in parallel. Since there are five bases in total, employing five NTT cores would be the best option for the performance. However, this will lead to high area cost and low utilization of NTT cores for the remaining arithmetic operations. For example, base conversion operations are not parallelizable and some NTT cores need to be idle during conversion operations. Therefore, the proposed hardware architecture uses three NTT cores. The high-level diagram of the overall design is shown in Fig. 5.4.

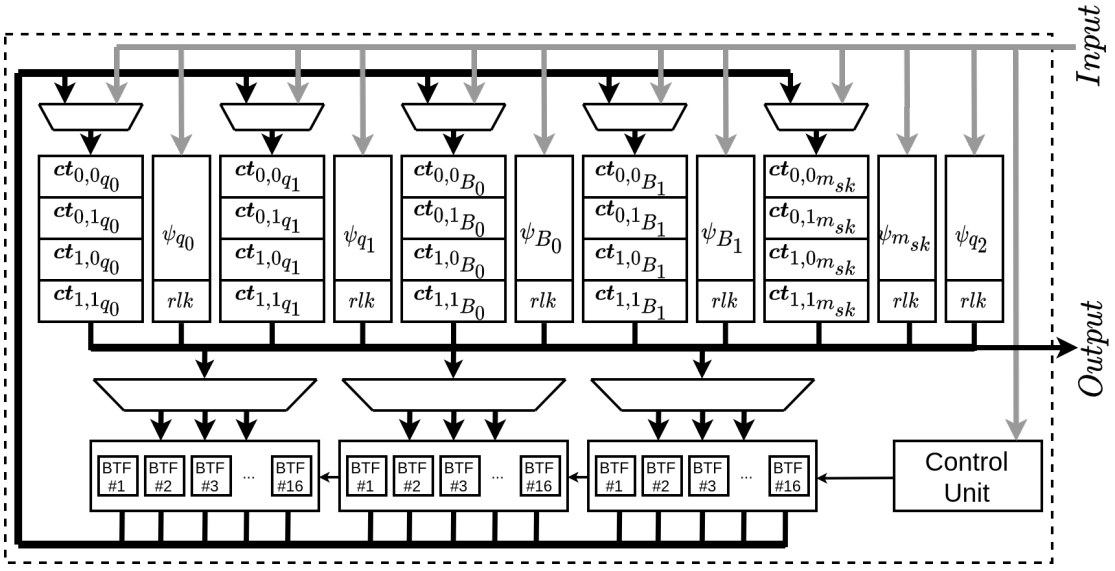


Figure 5.4 The Proposed Hardware Architecture

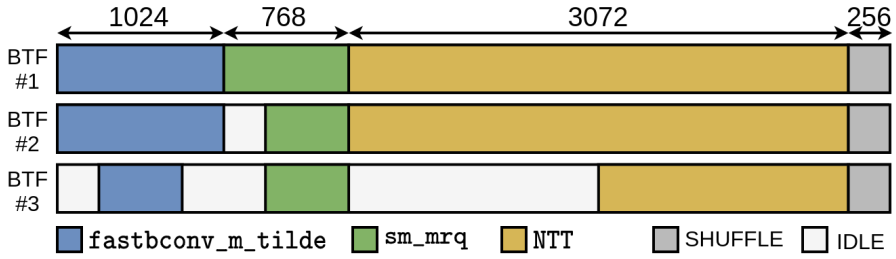
Since there are five bases, we employ $5 \cdot 32 = 160$ BRAMs for storing the input, intermediate, and output ciphertext components. Each 32 BRAMs store four ciphertext components for one base as shown in Fig. 5.4. For example, the first 32 BRAMs store all four ciphertext components ($\boldsymbol{ct}_{0,0}$, $\boldsymbol{ct}_{0,1}$, $\boldsymbol{ct}_{1,0}$, $\boldsymbol{ct}_{1,1}$) in base q_0. As explained in Section 5.2.1, BRAMs storing one ciphertext component in one base use memory depth of 256 for MNTT and MINTT operations. Therefore, in the overall design, the depth of each BRAM is set as 1024 since each 32 BRAMs store four ciphertext components in one base. The first 256 addresses of each BRAM store the first component of the first ciphertext. Similarly, the next three 256 addresses of each BRAM store other components of the ciphertexts.

For the homomorphic multiplication operation, MNTT, MINTT, and ciphertext multiplication operations need to be performed for two RNS bases (q_0, q_1) and three auxiliary bases (B_0, B_1, m_{sk}). On the other hand, for relinearization operation, MNTT, MINTT, and relinearization key multiplication operations need to be performed for three RNS bases (q_0, q_1, q_2). Therefore, the precomputed powers of $2n$-th root of unity for six different bases need to be stored in BRAMs. As explained in Section 5.2.1, the precomputed powers of $2n$-th root of unity for one base need 16 BRAMs where each BRAM has a depth of 1535. Therefore, we employ $6 \cdot 16 = 96$ BRAMs for storing the necessary powers of $2n$-th root of unity for MNTT and MINTT operations. Similarly, the precomputed relinearization keys need to be stored in the hardware. Since our parameter set uses three RNS bases, the proposed hardware should store a relinearization key with 4 components where each component is in three RNS bases (q_0, q_1, q_2) as detailed in Section 5.2.2. In overall, $4 \cdot 3 \cdot 4096 = 49152$ coefficients need to be stored for relinearization operation. In the

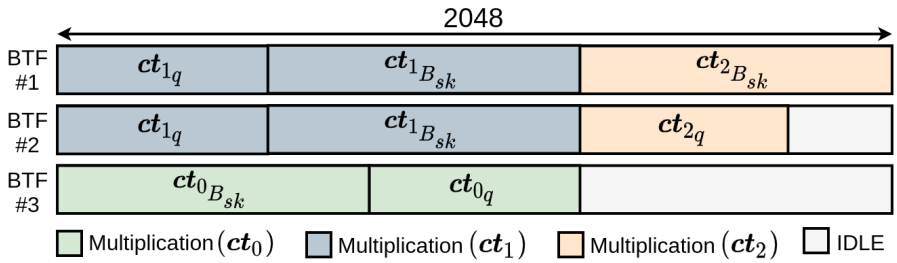
proposed architecture, instead of employing separate BRAMs for storing the relinearization keys, we use the last 512 addresses of 96 BRAMs storing the powers of $2n$-th root of unity to store the relinearization keys. Since NTT or INTT operations and the multiplication operations with relinearization keys are not performed at the same time, storing both precomputed constants in the same BRAMs will not cause any memory read conflict. It should be noted that since the proposed work uses the word-level Montgomery reduction algorithm, precomputed powers of $2n$-th root of unity and the relinearization keys are multiplied with the Montgomery residue R before being sent to the FPGA. In addition to the NTT cores and BRAMs, the proposed hardware architecture also utilizes large multiplexers for data routing between NTT cores and BRAMs.

Before any operation, the proposed hardware first reads RNS bases, auxiliary bases, precomputed powers of $2n$-th root of unity, the relinearization keys and other necessary precomputed data into the BRAMs and register files. The proposed hardware then reads input ciphertexts (in RNS bases q_0 and q_1) into the first 64 BRAMs and waits for the start signal. The homomorphic multiplication operation has three main parts: (i) base conversion, small Montgomery reduction, and NTT operations, (ii) ciphertext multiplication, and (iii) INTT, flooring, and Shenoy-Kumaresan conversion operations. The scheduling of the homomorphic multiplication operation in the proposed hardware architecture is shown in Fig. 5.5.

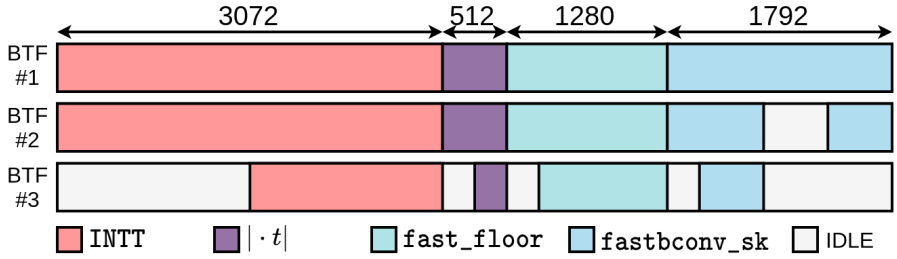
In the first part, each ciphertext component, which is stored in 256 addresses of BRAMS, is processed as shown in Fig. 5.5 (a). For one ciphertext component in RNS and auxiliary bases, the fast base conversion and small Montgomery reduction operations take 1024 and 768 clock cycles, respectively. Since one NTT operation is performed in 1536 clock cycles by one NTT core and five NTT operations need to be performed for five bases, NTT operations take $\lceil \frac{5}{3} \rceil \cdot 1536 = 3072$ clock cycles. These operations are repeated for each ciphertext component by just changing the read/write addresses during the operations. For example, the second component of the first ciphertext performs the same operations as the first component of the first ciphertext by only adding 256 to the addresses used by the first ciphertext component. At the end of the first part, four ciphertext components in five bases are stored in 160 BRAMs. For the ciphertext multiplication part, ciphertext components in the same base need to be multiplied or accumulated. This requires reading two ciphertext components in the same base at the same time. Therefore, the proposed work performs a shuffling operation by rotating the components of the second ciphertext to the next memory block (32 BRAMs), which takes 256 clock cycles, before the second part of the homomorphic multiplication operation. In total, the first part of the homomorphic multiplication operation takes 20480 clock cycles.



(a) Fast Base Conversion, Small Montgomery Reduction and NTT



(b) Ciphertext Multiplication



(c) INTT, Flooring and Shenoy-Kumaresan Conversion

Figure 5.5 Scheduling of Homomorphic Multiplication Operation

During the second part of the homomorphic multiplication operation, ciphertext multiplication is performed for five bases. It takes 2048 clock cycles in total and its scheduling is depicted in Fig. 5.5. (b). After the second step, the number of ciphertext components is reduced from four to three. In the last step, INTT operation is first performed for each ciphertext component in five bases. Then, the multiplication with plaintext modulus t is performed. In the proposed hardware, the multiplication of the resulting polynomials with n^{-1} after INTT operation is merged with the multiplication operation with plaintext modulus t. Finally, flooring and Shenoy-Kumaresan conversion operations are performed. For one ciphertext component, INTT, multiplication with $t \cdot n^{-1}$, flooring and Shenoy-Kumaresan conversion operations are performed in 3072, 512, 1280, 1792, respectively. In total, the last part of the homomorphic multiplication operation takes 6656 clock cycles for each ciphertext component. Overall, the homomorphic multiplication operation is performed in 42313 clock cycles including pipelining latency.

The relinearization operation has two parts: (i) decomposition, NTT and relin-

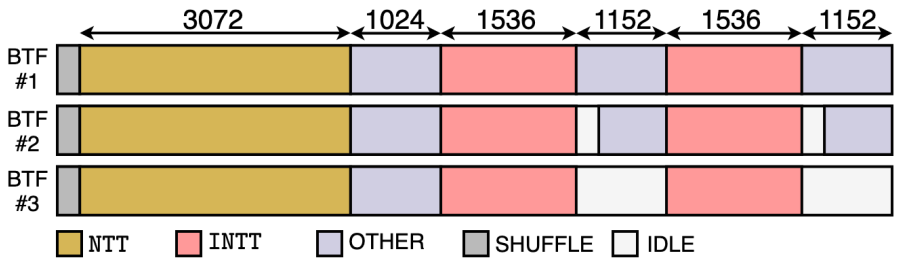


Figure 5.6 Scheduling of Relinearization Operation

earization key multiplication, and (*ii*) INTT and modulus switching. The scheduling of the relinearization operation in the proposed hardware architecture is shown in Fig. 5.6. The relinearization operation starts with the NTT operation of the third component of the ciphertext ($\boldsymbol{ct}_0$ in Fig. 5.2) generated by the homomorphic multiplication operation for three RNS bases. Since it requires NTT operation for three RNS bases, the proposed hardware re-organizes the memory structure for enabling parallel computation. The third component of the ciphertext is read from the first 64 BRAMs and read to the next 96 BRAMs before NTT operations in 256 clock cycles. Then, six NTT and the relinearization key multiplication operations are performed in 3072 and 1024 clock cycles, respectively. In the second step, ciphertext components generated after the relinearization key multiplication are transformed to the polynomial domain using INTT operation. The INTT and the following modulus switching operations take 1536 and 1152 clock cycles, respectively for one ciphertext component. In total, the proposed hardware performs relinearization operation in 9881 clock cycles including pipelining latency.

In addition to the core hardware performing homomorphic multiplication and relinearization, we design and implement a proof-of-concept wrapper for the proposed hardware architecture. The wrapper is designed to control the communication between the CPU and the FPGA. It employs 16 input FIFOs and 16 output FIFOs for receiving input data from the CPU and sending output data to the CPU, respectively.

5.4 Results and Comparison

To verify the correctness of the proposed hardware architecture, the SEAL library is modified accordingly as explained in Section 5.3.1. Then, we verify the proposed hardware architecture by comparing our hardware results with the results

Table 5.2 Our Hardware Implementation Results

Module	LUTs	(%)	DFFs	(%)	DSPs	(%)	BRAMs	(%)	Memory (KB)	Freq. (MHz)
top [a]	136416		54120		336		368		1263	
⌊ wrapper	249		1746		–		32		128	125
⌊ core	136167	(32%)	52374	(6%)	336	(9%)	336	(25%)	1135	
top [b]	147005		55232		336		400		1664	
⌊ wrapper	252		1746		–		32		128	125
⌊ core	146753	(34%)	53486	(6%)	336	(9%)	368	(27%)	1536	

[a]:Performs only multiplication.

[b]:Performs both multiplication and relinearization.

we obtained from the SEAL library for the same parameter set and test inputs. The proposed hardware architecture is developed into Verilog modules. Then, it is synthesized and placed & routed for Xilinx Virtex VC709 Evaluation Board with Virtex-7 FPGA (xc7vx690t-2ffg1761c) using performance-optimized and default settings for synthesis and place & route, respectively. The implementation results of our architecture are shown in Table 5.2, where we present the results for two different architectures. The first architecture performs only homomorphic multiplication while the second architecture can perform both homomorphic multiplication and relinearization operations. In Table 5.2, the wrapper and core represent the wrapper module designed for the communication and the proposed hardware architecture, respectively. The hardware architecture performing both operations uses 34% of LUTs in Xilinx VC709 Evaluation Board with 1.5 MB memory requirement. It can perform one homomorphic multiplication and relinearization operations in 42313 and 9881 clock cycles, respectively. With 125 MHz operating frequency, the same operations are performed in 0.338 μs and 0.079 μs, respectively, excluding I/O overhead.

Although there are different implementations of the BFV scheme targeting various platforms, it should be noted that it is not easy to present a fair comparison due to the use of different parameters and target platforms. The performance of our architecture is compared with SEAL library (Microsoft, 2020) and other related works in the literature (Sinha Roy et al., 2018), (Sinha Roy et al., 2019), (Turan et al., 2020), (Badawi et al., 2018), (Badawi et al., 2018), (Takeshita et al., 2020), (Reis et al., 2020) in Table 5.3. The timing results presented in Table 5.3 include the performance of the core operation excluding the I/O overhead for all works. The timing results of homomorphic multiplication and relinearization operations in SEAL library are obtained on a laptop with Intel Core i7-9750H @ 2.60GHz × 12 with 16 GB RAM using GCC version 7.5.0 in Ubuntu 18.04 with single thread. The proposed architecture in this work improves the performance of homomorphic multiplication and relinearization operations by 18.4× and 16.1× respectively, compared to the SEAL

Table 5.3 Comparative Table

Work	n	$\lceil \log_2(q) \rceil$	**Platform**	**Latency**	
				M*	**R***
(Sinha Roy et al., 2018)	32768	1228	Virtex-6	3360	
(Sinha Roy et al., 2019)	4096	180	Zynq US+	4.458	
(Turan et al., 2020)	4096	180	AWS F1	4.340	
(Badawi et al., 2018)	4096	180	Tesla P100	1.833	
(Badawi et al., 2018)	4096	60	Tesla V100	0.997	
(Takeshita et al., 2020)	16384	438	Intel Core i7	0.115	
(Reis et al., 2020)	8192	152	Intel Core i5-5300U	6.600	
SEAL (Microsoft, 2020)	4096	93	Intel Core i7-9750H	6.233	1.269
Ours	4096	93	Virtex-7	0.338	0.079

*:Unit is msec.

library with the same parameter set. Since the proposed hardware architecture uses only 34% of the FPGA resources, two of the proposed hardware can be instantiated to further improve the performance. In that case, the performance of homomorphic multiplication and relinearization operations will be improved by 36.8× and 32.2× respectively, compared to the SEAL library.

There is no work in the literature using our parameter set. The GPU implementation in (Badawi et al., 2018) uses a parameter set that is very similar to our parameters. It performs one homomorphic multiplication operation (including relinearization) in 1 ms. Our hardware architecture shows more than 2× better performance than the work in (Badawi et al., 2018). The works in (Badawi et al., 2018; Sinha Roy et al., 2019; Turan et al., 2020) use the ring size of 4096 with 180-bit coefficient modulus which is twice our coefficient modulus in size. Our work shows 10.6×, 10.4× and 4.4× better performance than the works in (Sinha Roy et al., 2019), (Turan et al., 2020) and (Badawi et al., 2018), respectively. The work in (Sinha Roy et al., 2018) uses a very large ring size and coefficient modulus. Thus, it is not feasible to make a fair comparison using the performance figures. The only implementation showing better performance than our architecture is the work in (Takeshita et al., 2020), which shows almost 4× better performance than our work even though it uses a larger parameter set. However, this implementation works with and is optimized for constant coefficient modulus.

Although the proposed hardware architecture does not present a fully working hardware/software co-design framework, we present a brief analysis for the I/O overhead estimate of the proposed hardware architecture. As explained in Section 3.4 and Section 4.2.4, the hardware/software co-design framework utilizing PCIe connection using RIFFA (Jacobsen et al., 2012) shows that RIFFA can achieve 76% of the 32

Table 5.4 Hardware Resource Estimates

Design	**LUTs**	(%)	**DFFs**	(%)	**DSPs**	(%)	**BRAMs**	(%)	**Mult. Depth**
RNS-3[a]	146753	(34%)	53486	(6%)	336	(9%)	368	(27%)	1
RNS-4[b]	195670	(45%)	71314	(8%)	448	(12%)	496	(36%)	2
RNS-5[c]	244587	(57%)	89143	(10%)	560	(15%)	624	(46%)	3
RNS-6[d]	293504	(68%)	106972	(12%)	672	(18%)	752	(55%)	4

[a]:Our architecture.

[b]:Architecture with four NTT cores.

[c]:Architecture with five NTT cores.

[d]:Architecture with six NTT cores.

Gbps maximum theoretical bandwidth which is around 24 Gbps. The proposed architecture takes two ciphertexts as inputs and generates one ciphertext as output, where each ciphertext has two components that consist of $2 \cdot 4096$ 32-bits coefficients. In overall, $2 \cdot 2 \cdot 2 \cdot 4096 \cdot 32 = 1048576$ bits and $2 \cdot 2 \cdot 4096 \cdot 32 = 524288$ bits should be sent to the FPGA as input and received from the FPGA as output, respectively. Thus, the I/O overhead for the communication between CPU and FPGA is calculated as $(1048576 + 524288)/(24 \cdot 10^6) = 65.55\mu s$ for one homomorphic multiplication (including relinearization) operation. Considering the I/O overhead estimate, the proposed architecture with a hardware/software framework can perform one homomorphic multiplication (including relinearization) operation in $0.482ms$, which is $15.5\times$ faster compared to the SEAL library. In addition to the input ciphertexts, the proposed architecture in the FPGA should receive and store the precomputed powers of $2n$-th root of unity for q_0, q_1, B_0, B_1, m_{sk} bases and the precomputed relinearization keys from the CPU. Since the precomputed data is sent to the FPGA only once before any operation for a parameter set, the time required to complete this data transfer is not included in the I/O overhead estimate.

Although the proposed work supports the multiplicative depth of one, it can be scaled easily for larger parameter sets by employing extra NTT cores and memory blocks. For example, a hardware architecture with the parameter set employing four RNS bases will show a similar timing performance with our architecture (employing four RNS bases) for performing the homomorphic multiplication operation. Also, it will have a larger multiplicative depth at the expense of using more hardware resources. To that end, we present the estimated hardware resources used for the scaled versions of our hardware architecture in Table 5.4.

5.5 Summary

In this chapter, we present a high-performance hardware architecture for one of the most complex and time-consuming homomorphic operations: homomorphic multiplication of two ciphertexts. The proposed hardware architecture performs homomorphic multiplication and relinearization operations of the RNS variant of the BFV scheme (Fan & Vercauteren, 2012), (Bajard et al., 2017), and supports parameters $n = 4096$, $\lceil \log_2(q) \rceil = 93$ and $\lceil \log_2(t) \rceil = 17$ with a multiplicative depth of one. The proposed architecture employs three NTT cores where each core uses 16 unified butterfly units. It uses 34% of LUTs on the Xilinx VC709 Evaluation Board. The proposed architecture finishes homomorphic multiplication and relinearization operations in 0.338 ms and 0.079 ms, respectively. Compared to highly-optimized SEAL library (`v3.5`) (Microsoft, 2020), it shows 18.4× and 16.1× performance improvements for the homomorphic multiplication and relinearization operations, respectively, excluding I/O operations.

6. A HARDWARE ACCELERATOR FOR POLYNOMIAL MULTIPLICATION OPERATION OF CRYSTALS-KYBER PQC SCHEME

In this chapter, we present three different hardware architectures (lightweight, balanced, high-performance) that implement the NTT, INTT and polynomial multiplication operations for the CRYSTALS-Kyber scheme. The proposed architectures include a unified butterfly structure for optimizing polynomial multiplication and can be utilized for accelerating the key generation, encryption and decryption operations of the CRYSTALS-Kyber scheme (Bos et al., 2018). [1]

6.1 Introduction

NIST has started a PQC standardization process in 2016 and many lattice-based post-quantum schemes are proposed since then (Chen et al., 2016). NIST recently announced the post-quantum KEM finalists at round three of the standardization process and the lattice-based KEM CRYSTALS-Kyber (Bos et al., 2018) has been selected as one of the four finalists.

Lattice-based cryptosystems work with polynomial rings and perform polynomial arithmetic; multiplication of two large-degree polynomials, in particular, which is one of the most time-consuming operations utilized in lattice-based PQC schemes. Schoolbook polynomial multiplication method is inefficient for implementing polynomial multiplication operations and it has $\mathcal{O}(n^2)$ complexity. NTT reduces $\mathcal{O}(n^2)$ complexity to quasi-linear complexity and, therefore, it is utilized in many lattice-based cryptosystems suffering from high complexity of polynomial arithmetic (Ducas et al., 2018), (Bos et al., 2018), (Alkim, Ducas, Pöppelmann & Schwabe, 2016),

[1] This chapter presents the work in (Yaman et al., 2021).

(Alkim et al., 2019), (Fouque et al., 2018). There are many works in the literature targeting efficient implementations of main arithmetic blocks of the post-quantum cryptosystems for software (Seiler, 2018), (Botros et al., 2019), (Lyubashevsky & Seiler, 2019), (Kannwischer, Rijneveld, Schwabe & Stoffelen, 2019), (Alkim et al., 2020) and hardware (Mert et al., 2020), (Fritzmann, Sigl & Sepúlveda, 2020), (Pöppelmann & Güneysu, 2013), (Sinha Roy et al., 2014), (Huang, Huang, Lei & Wu, 2020), (Xin, Han, Yin, Zhou, Yang, Cheng & Zeng, 2020), (Banerjee et al., 2019), (Xing & Li, 2021), (Alkim, Evkan, Lahr, Niederhagen & Petri, 2020) platforms.

Key generation, encryption and decryption operations of the CRYSTALS-Kyber scheme also use polynomial multiplication operations and NTT-based polynomial multiplication are utilized for efficient implementation of these operations. The team of CRYSTALS-Kyber adopted the technique proposed by Seiler *et al.* (Lyubashevsky & Seiler, 2019) and reduced the parameter q of CRYSTALS-Kyber from 7681 to 3329. This changed the definitions of NTT, INTT and coefficient-wise multiplication (CWM) operations. To the best of our knowledge, there are five hardware (Huang et al., 2020), (Xin et al., 2020), (Fritzmann et al., 2020), (Xing & Li, 2021), (Alkim et al., 2020) and two software (Botros et al., 2019), (Alkim et al., 2020) implementations proposed for the NTT/INTT and polynomial multiplication operations of the CRYSTALS-Kyber with the new parameters and operation definitions. Our proposed hardware architecture outperforms most of the works in the literature.

In this chapter, we propose three different hardware architectures (lightweight, balanced, high-performance) performing NTT/INTT and polynomial multiplication operations for the new parameter set of CRYSTALS-Kyber [2]. The proposed architectures utilize a unified butterfly structure, which can be used for both NTT and INTT operations. The lightweight, balanced and high-performance hardware architectures use one, four and sixteen butterfly units, respectively. They can be used to accelerate key generation, encryption and decryption operations of CRYSTALS-Kyber. Our high-performance hardware with 16 butterfly units shows up to 99×, 98× and 89× improved performance for NTT, INTT and polynomial multiplication, respectively, compared to the high-speed software implementations on Cortex-M4 (Alkim et al., 2020).

The rest of the chapter is organized as follows. Section 6.2 introduces preliminaries. Section 6.3 presents the hardware architectures with proposed optimizations. Section 6.4 presents the implementation results and compares the results with prior work, and Section 6.5 summarizes the chapter.

[2]Code is available at `https://github.com/acmert/kyber-polmul-hw`

6.2 Preliminaries

In this section, we present a brief definition of the CRYSTALS-Kyber scheme and its arithmetic operations.

6.2.1 A New Variant of NTT-based Polynomial Multiplication

In (Lyubashevsky & Seiler, 2019), Seiler *et al.* proposes a variant of NTT operation which enables efficient polynomial multiplication in $\boldsymbol{R}_{q,n}$ with satisfying only $q \equiv 1 \pmod{n}$ and without requiring pre-processing and post-processing operations. This technique is adopted by CRYSTALS-Kyber and their parameter q is reduced from 7681 to 3329, which is satisfying $3329 \equiv 1 \pmod{256}$. The NTT, INTT and CWM operations of CRYSTALS-Kyber are also changed accordingly (Bos et al., 2018). This new variant of NTT operation generates 128 degree-1 polynomials, different from the original NTT operation, which generates 256 degree-0 polynomials. Similarly, the new INTT operation takes 128 degree-1 polynomials as input. Since the outputs of NTT operation are 128 degree-1 polynomials, CWM operation is performed as the multiplications of two degree-1 polynomials in $\mathbb{Z}_q[x]/(x^2 - \omega^i)$ where i changes according to index of coefficients. Algorithms for NTT, INTT and CWM operations of CRYSTALS-Kyber are shown in Algorithm 13, 14 and 15, respectively. Similar to the MNTT and MINTT operations detailed in Section 2.5.2, new NTT and INTT operations utilize CT and GS butterfly structures, respectively.

This new variant of NTT and INTT operations are represented as $\mathcal{NTT}$ and $\mathcal{INTT}$ respectively. Similarly, the new CWM operation is represented with $\circ$. The polynomial multiplication operation for CRYSTALS-Kyber with new NTT definition is shown in Eqn. 6.1.

$$\boldsymbol{c} = \mathcal{INTT}_n((\mathcal{NTT}_n(\boldsymbol{a}) \circ \mathcal{NTT}_n(\boldsymbol{b}))) \tag{6.1}$$

Algorithm 13 In-place Forward NTT Algorithm for Kyber Scheme

Input: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in standard-order
Input: $\omega \in \mathbb{Z}_q$ (primitive n-th root of unity)
Input: $n = 2^l$, q (s.t. $q \equiv 1 \pmod{n}$)
Output: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order

```
k = 1
for (i = 1; i < l; i = i + 1) do
    m = 2^(l−i)
    for (s = 0; i ≤ n; s = s + m) do
        for (j = s; i ≤ s + m; j = j + 1) do
            A, B, W = a[j], a[m + j], ω^br(k,l−1) (mod q)
            T = (W · B) (mod q)
            E, O = (A + T) (mod q), (A − T) (mod q)
            a[j], a[j + m] = E, O
        end for
        k = k + 1
    end for
end for
return a
```

6.2.2 CRYSTALS-Kyber

Kyber is a KEM transformed from a public-key encryption scheme and it is proposed for NIST's post-quantum standardization process (Bos et al., 2018). CRYSTALS-Kyber algorithm works with polynomial ring $\boldsymbol{R}_{q,n}$ where q and n are 3329 and 256, respectively. The simplified version of key generation, encryption and decryption operations are described as follow:

- **Key Generation:** For $\overline{\boldsymbol{A}} \leftarrow \boldsymbol{R}_{q,n}^{k \times k}$ and $\boldsymbol{s}, \boldsymbol{e} \leftarrow \boldsymbol{R}_{q,n}^{1 \times k}$,

$$(pk, sk) = (\overline{\boldsymbol{A}} \circ \mathcal{NTT}(\boldsymbol{s}) + \mathcal{NTT}(\boldsymbol{e}), \mathcal{NTT}(\boldsymbol{s})).$$

- **Encryption:** For $\overline{\boldsymbol{A}} \leftarrow \boldsymbol{R}_{q,n}^{k \times k}$, $\boldsymbol{r}, \boldsymbol{e_1} \leftarrow \boldsymbol{R}_{q,n}^{1 \times k}$, $\boldsymbol{e_2} \leftarrow \boldsymbol{R}_{q,n}$, pk and $\boldsymbol{m} \in \boldsymbol{R}_{q,n}$,

$$ct = (\boldsymbol{u}, \boldsymbol{v}) = (\mathcal{INTT}(\overline{\boldsymbol{A}}^T \circ \mathcal{NTT}(\boldsymbol{r})) + \boldsymbol{e_1}, \mathcal{INTT}(pk^T \circ \mathcal{NTT}(\boldsymbol{r})) + \boldsymbol{e_2} + \boldsymbol{m}).$$

- **Decryption:** For $sk = \overline{\boldsymbol{s}}$ and $ct = (\boldsymbol{u}, \boldsymbol{v})$,

$$\boldsymbol{m} = \boldsymbol{v} - \mathcal{INTT}(sk^T \circ \mathcal{NTT}(\boldsymbol{u})).$$

CRYSTALS-Kyber adjusts its security level by changing the parameter k which can take any of $\{2, 3, 4\}$. Key generation operation requires $2k$ NTT operations and k^2 CWM operations. Encryption operation requires k NTT, $k^2 + k$ CWM and $k + 1$ INTT operations. Decryption operation requires k NTT, k CWM and 1 INTT operations.

Algorithm 14 In-place Inverse NTT Algorithm for Kyber Scheme

Input: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order
Input: $\omega^{-1} \in \mathbb{Z}_q$ (modular inverse of primitive n-th root of unity)
Input: $n = 2^l$, q (s.t. $q \equiv 1 \pmod{n}$)
Output: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$ in standard-order

$k = 0$
for $(i = l-1;\ i \leq 1;\ i = i-1)$ **do**
  $m = 2^{l-i}$
  for $(s = 0;\ i \leq 2^l;\ s = s+m)$ **do**
    for $(j = s;\ i \leq s+m;\ j = j+1)$ **do**
      $A, B, W = \overline{\boldsymbol{a}}[j], \overline{\boldsymbol{a}}[j+m], \omega^{br(k,l-1)+1} \pmod{q}$
      $E, O = (A+B) \pmod{q}, (A-B) \cdot W \pmod{q}$
      $\overline{\boldsymbol{a}}[j], \overline{\boldsymbol{a}}[j+m] = \texttt{DIVby2}(E), \texttt{DIVby2}(O)$
    end for
    $k = k+1$
  end for
end for
return $\overline{\boldsymbol{a}}$

Algorithm 15 Coefficient-wise Multiplication Algorithm for Kyber Scheme

Input: $\overline{\boldsymbol{a}}(x), \overline{\boldsymbol{b}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order
Input: $\omega \in \mathbb{Z}_q$ (primitive n-th root of unity)
Output: $\overline{\boldsymbol{c}}(x) \in \boldsymbol{R}_{q,n}$ in bit-reversed order

for $(i = 0;\ i \leq 2^{l-1};\ i = i+1)$ **do**
  $W = \omega^{br(i,l-1)+1} \pmod{q}$
  $a_0, a_1 = \overline{\boldsymbol{a}}[2i], \overline{\boldsymbol{a}}[2i+1]$
  $b_0, b_1 = \overline{\boldsymbol{b}}[2i], \overline{\boldsymbol{b}}[2i+1]$
  $\overline{\boldsymbol{c}}[2i] = (a_0 \cdot b_1 + a_1 \cdot b_0) \pmod{q}$
  $\overline{\boldsymbol{c}}[2i+1] = (a_1 \cdot b_1 \cdot W + a_0 \cdot b_0) \pmod{q}$
end for
return $\overline{\boldsymbol{c}}$

6.3 Polynomial Multiplication Architecture

In this section, the proposed polynomial multiplier architectures and their main arithmetic blocks are explained in a bottom-up fashion, starting from the implementation of the modular reduction unit. Then, we present our unified butterfly unit and finally overall design is presented.

6.3.1 Modular Reduction Unit

In this section, we present a constant time modular reduction hardware for $q = 3329 = 2^{12} - 2^9 - 2^8 + 1$. Modular reduction unit takes a 24-bit integer c as input from a DSP block performing 12-bit×12-bit unsigned integer multiplication with one clock cycle latency. The proposed modular reduction hardware utilizes the property $2^{12} \equiv 2^9 + 2^8 - 1 \pmod{3329}$ recursively in a similar approach with Zhang *et al.* (Zhang et al., 2020) as shown in Eqn. 6.2-6.6 where $c[x:y]$ represents the bits of integer c from y^{th} bit to x^{th} bit with right-most bit is being 0^{th}bit.

$$d = 2^{12}c[23:12] + c[11:0] \tag{6.2}$$

$$d = 2^9 c[23:12] + 2^8 c[23:12] - c[23:12] + c[11:0] \tag{6.3}$$

$$\begin{aligned} d = {} & 2^{12}c[23:15] + 2^{12}c[23:16] + 2^9 c[14:12] + \\ & 2^8 c[15:12] - c[23:12] + c[11:0] \end{aligned} \tag{6.4}$$

$$\begin{aligned} d = {} & (2^9 + 2^8 - 1)(c[23:15] + c[23:16]) + \\ & 2^9 c[14:12] + 2^8 c[15:12] - c[23:12] + c[11:0] \end{aligned} \tag{6.5}$$

$$\begin{aligned} d = {} & 2^9 c[23:15] + 2^8 c[23:15] - c[23:15] + \\ & 2^9 c[23:16] + 2^8 c[23:16] - c[23:16] + \\ & 2^9 c[14:12] + 2^8 c[15:12] - c[23:12] + c[11:0] \end{aligned} \tag{6.6}$$

We apply this approach recursively until no bits left at a position greater than 12. Throughout this process, we eliminate redundant operations by combining identical bits at the same position. For example, there are two $2^8 c[15]$ in Eqn. 6.6 where they can be combined as single $2^9 c[15]$. We also convert subtraction operation into addition by negating the subtracted integer, adding 1 to the final result, and extending the sign bit to 15^{th} bit. As shown in Fig. 6.1, this recursive process generates a tree of bits where white and red boxes represent bit and negated bit of the input c, respectively.

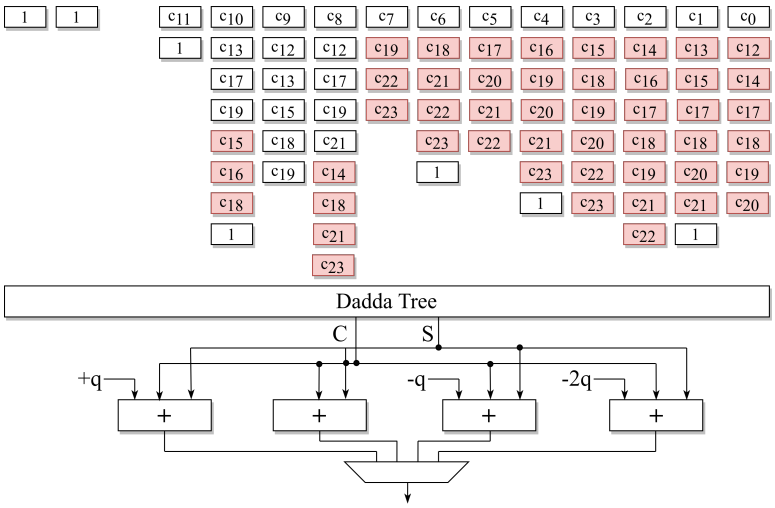


Figure 6.1 Modular Reduction Unit

To reduce the tree, we utilize Dadda's method (Dadda, 1965) which produces an incomplete result with two integers, C and S, using 50 full adders (FAs) and 13 half adders (HAs). To generate the final result, the resulting integers need to be added. We observe that the final result at the end of the addition operation is between 9271 and -3264 which is not in the desired range $[0, q)$. Therefore, $(C + S + q)$, $(C + S)$, $(C + S - q)$ and $(C + S - 2q)$ are calculated separately after the Dadda tree and the result in the range $[0, q)$ is selected as the final result. The proposed modular reduction unit for $q = 3329$ has 1 clock cycle latency. There are other modular reduction methods such as Barrett and Montgomery (Mert et al., 2020). Both Barrett and Montgomery reductions require 2 multiplication operations with additional addition and subtraction operations. Montgomery modular reduction also requires its operands to be converted back and forth in Montgomery space. On the other hand, our proposed implementation utilizes only addition and subtraction operations.

6.3.2 Unified Butterfly Unit

NTT operation requires CT butterfly which performs $A + B \cdot W \pmod q$ and $A - B \cdot W \pmod q$ while INTT operation utilizes GS butterfly structure which performs $A + B \pmod q$ and $(A - B) \cdot W \pmod q$. Since the proposed design aims to use different butterfly structures for NTT and INTT operations, we propose a unified butterfly unit shown in Fig. 6.2.

The proposed butterfly unit uses no extra modular multiplier, adder or subtractor

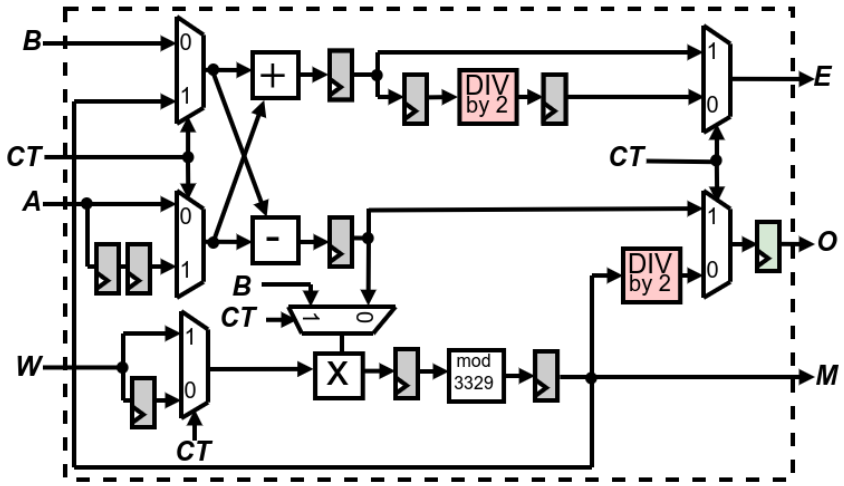


Figure 6.2 Unified Butterfly Unit

than a dedicated CT or GS butterfly unit. The proposed butterfly unit has one modular multiplier, one modular adder and one modular subtractor. It also has pipeline registers for synchronizing output coefficients and multiplexers for reconfigurability. The proposed butterfly unit takes A, B and W as inputs, performs butterfly operation, and generates two coefficients E and O as outputs corresponding to the steps 6–9 of the Algorithm 13 and the steps 6–8 of the Algorithm 14, respectively. It also takes control signal CT as input which is used as a selection signal for multiplexers in the butterfly unit. When the CT signal is set as 0 and 1, the butterfly unit is configured to work as GS and CT butterfly, respectively. The butterfly unit has three clock cycles latency for both CT and GS configurations.

The butterfly unit employs the technique proposed in (Zhang et al., 2020) to eliminate the multiplication of resulting coefficients with $n^{-1} \pmod q$ after the INTT operation. This technique multiplies the resulting coefficients of GS butterfly with $2^{-1} \pmod q$, which can be performed using addition and shift operations. For an odd prime q, $x \cdot 2^{-1} = \frac{x}{2} \pmod q$ can be performed as shown in Eqn. 6.7 where $\gg$ represents right shift operation.

$$\frac{x}{2} \pmod q = (x \gg 1) + x[0] \cdot (\frac{q+1}{2}) \tag{6.7}$$

Utilizing this property, one can divide the output of GS butterfly by two and calculate $\frac{A+B}{2} \pmod q$ and $\frac{(A-B)\cdot W}{2} \pmod q$ coefficients instead of $A+B \pmod q$ and $(A-B)\cdot W \pmod q$ as shown in the Step 8 of Algorithm 14. In this work, we adopt this technique and insert two `DIVby2` units into our butterfly units (shown as red boxes in Fig. 6.2) which perform the operation shown in Eqn. 6.7. Therefore, we eliminate extra 256 multiplication operations in $\mathbb{Z}_q$ after INTT operation at the

expense of extra hardware in the butterfly unit. This technique reduces the number of modular multiplication operations in INTT by 22%.

The proposed butterfly unit can also be configured to perform CWM operations defined in Algorithm 15. A polynomial multiplication in $\mathbb{Z}_q[x]/(x^2 - \omega^i)$ requires five multiplication and two addition operations as shown in the steps of 5–6 of Algorithm 15. This operation can be realized using a series of multiply-accumulate operation which can be performed using CT butterfly configuration and reading E output from the butterfly unit. Fig. 6.3 illustrates the multiplication of $(a_0 + a_1x)$ and $(b_0 + b_1x)$ in $\mathbb{Z}_q[x]/(x^2 - \omega)$ where inputs coefficients are mapped to A, W and B inputs of butterfly unit for calculating $A + B \cdot W \pmod q$. For example, 0, a_1 and b_0 coefficients are sent to A, W and B inputs, respectively, for calculating $0 + a_1 \cdot b_0$. To improve the latency, multiplication results $a_1 \cdot b_0$, $a_1 \cdot b_1$ and $a_0 \cdot b_0$, which are shown with red, blue and green in Fig 6.3, are forwarded to the input of the butterfly unit before being stored back to the memory. It takes five clock cycles for a CWM operation after filling the pipeline.

There are other unified butterfly structures proposed in previous works (Fritzmann et al., 2020), (Xin et al., 2020), (Xing & Li, 2021). The butterfly units proposed in (Fritzmann et al., 2020), (Xin et al., 2020) use two multipliers, whereas our design uses only a single modular multiplier unit. Also, our butterfly unit realizes the multiplication with $2^{-1} \pmod q$ operation after GS butterfly using adder and therefore, eliminates the multiplications with n^{-1} at the end of INTT operation. The butterfly unit in (Xing & Li, 2021) utilizes one modular multiplier unit with two modular adder and subtractor units while our butterfly unit employs one modular adder and one modular subtractor.

6.3.3 Overall Design

Fig. 6.4 shows the high-level block diagram of the proposed hardware architecture with one butterfly unit. There are four dual-port BRAMs for each butterfly unit where two BRAMs are used to store the first input polynomial and output polynomial, and two BRAMs are used to store the second input polynomial. There is also one BROM for each butterfly unit to store the pre-computed powers of the twiddle factor. Prior to any operation, the pre-computed powers of twiddle factor are loaded and stored in BROM blocks as explained in Chapter 4 (Mert et al., 2020). The input polynomials are loaded into the BRAMs with input multiplexers. Then, the proposed hardware starts its operation according to start signals and the resulting

CC#1	$0 + a_1 \cdot b_0$	–	–	
CC#2	$0 + a_1 \cdot b_1$	$0 + a_1 \cdot b_0$	–	
CC#3	$0 + a_0 \cdot b_0$	$0 + a_1 \cdot b_1$	$a_1 \cdot b_0$	
CC#4	$a_1 \cdot b_0 + a_0 \cdot b_1$	$0 + a_0 \cdot b_0$	$a_1 \cdot b_1$	
CC#5	$0 + (a_1 \cdot b_1) \cdot w$	$a_1 \cdot b_0 + a_0 \cdot b_1$	$a_0 \cdot b_0$	
CC#6	–	$a_0 \cdot b_0 + (a_1 \cdot b_1) \cdot w$	$a_1 \cdot b_0 + a_0 \cdot b_1$	
CC#7	–	–	$a_0 \cdot b_+ (a_1 \cdot b_1) \cdot w$	c_0
CC#8	–	–	–	c_1

Figure 6.3 Scheduling of CWM Operation for CRYSTALS-Kyber

polynomial is read using the output multiplexers after the operation.

The proposed architecture implements NTT and INTT schemes shown in Algorithm 13 and 14, respectively. Both operations consist of 7 stages and 128 butterfly operations should be performed in each stage. An n-pt NTT operation can be performed as separate two $(n/2)$-pt NTT operations after the first stage and this approach recursively can be applied to the smaller NTT operations. In this work, we utilize this property as detailed in Chapter 4 (Mert et al., 2020) and Chapter 7 (Mert et al., 2020), and propose an efficient memory access pattern, which is illustrated in Fig. 6.5 where numbers inside the boxes represent indices of stored coefficients. In an NTT stage, coefficients stored in the first half of the memory blocks should be read and written into the first memory block due to changing access pattern of NTT operation in each stage. Similarly, coefficients in the second half of the memory blocks should be read and written into the second memory block as shown in Fig. 6.5.

We adopt a changing memory read pattern for efficient memory management. Since the coefficients in consecutive memory addresses should be written into the same memory block after the butterfly operation for the next stage, they should not be read consecutively. Instead, coefficients from the first and second half of the memory blocks should be read in an alternating way. For example, in the first stage of the NTT operation, after reading the coefficients in address 0, coefficients in address $(n/4)$ should be read instead of address 1 for the next butterfly operation. Since the coefficients read in a clock cycle should be written into the same memory block after butterfly operation, we place an extra register at the output of O output of butterfly

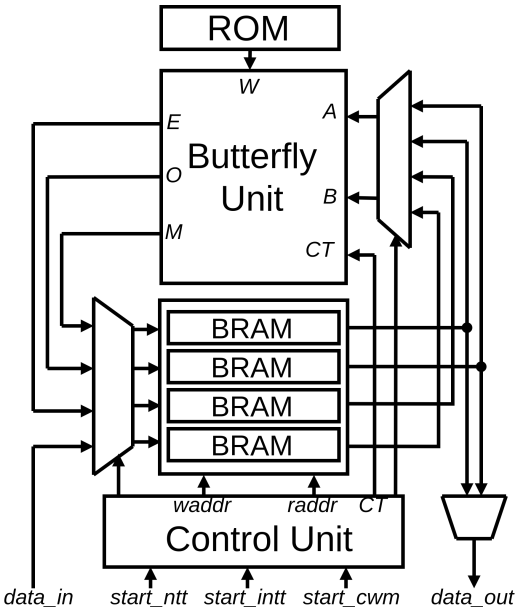


Figure 6.4 Overall Design with one Butterfly Unit

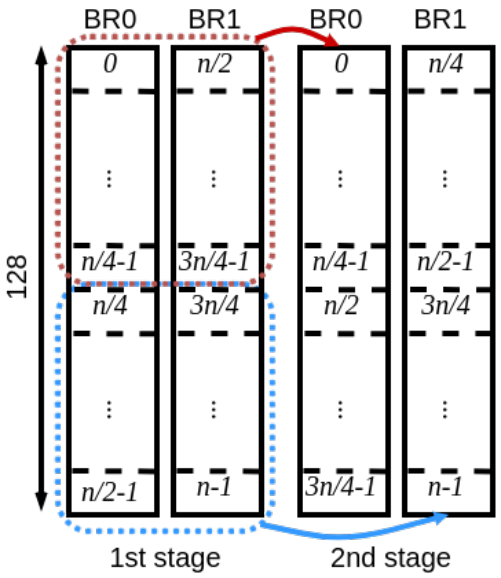


Figure 6.5 Memory Access Pattern for one Butterfly Unit

unit as shown with green in Fig. 6.2. Therefore, two coefficients produced by one butterfly unit can be written into the same memory block in two consecutive clock cycles. Since the proposed architecture is pipelined, this extra register does not cause any stall. Similarly, INTT operation uses the same memory access pattern.

Our proposed balanced and high-performance designs use 4 and 16 butterfly units respectively. The number of memory blocks increases with the number of butterfly units to not stall the pipeline and the coefficients are distributed across memory blocks accordingly. While the number of memory blocks is increasing, the used address space decreases accordingly. For example, the design with one butterfly unit (and with two BRAMs for each polynomial) uses BRAMs with a depth of 128 while the design with four butterfly units (and with eight BRAMs for each polynomial) uses BRAMs with a depth of 32. The memory access pattern for the design with four butterfly units is shown in Fig. 6.6.

The proposed designs can take two polynomials as inputs and they can perform NTT and INTT operations for both input polynomials. Also, the designs can perform CWM operation between input polynomials. The proposed architectures with one, four and sixteen butterfly units finish NTT operation in 904, 232, 69, INTT operation in 904, 233, 71, and CWM operation in 647, 167, 47 clock cycles, respectively, excluding the time for loading input polynomials into BRAMs.

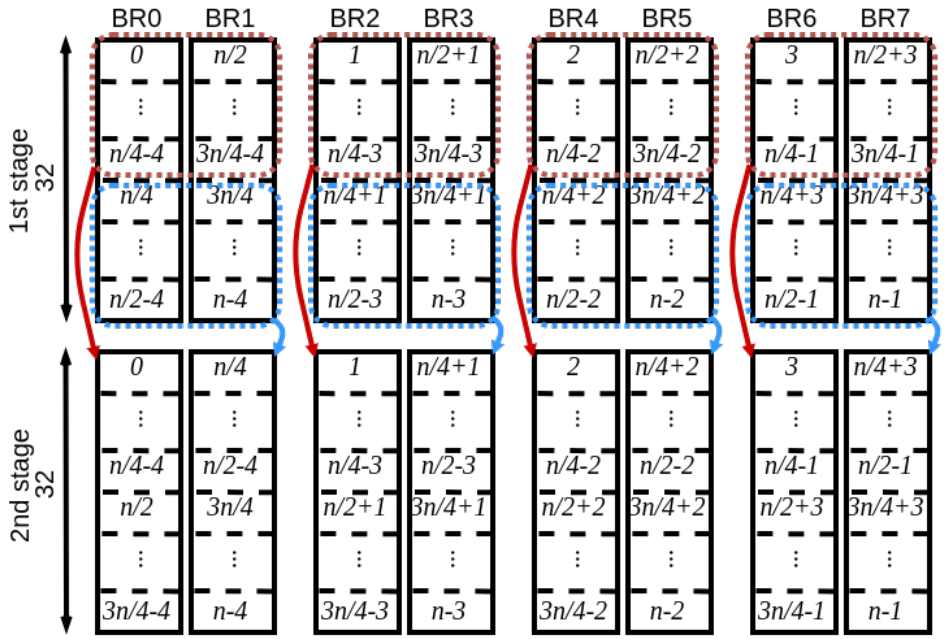


Figure 6.6 Memory Access Pattern for four Butterfly Units

6.4 Results and Comparison

In this section, we present our implementation results and their comparison with the works in the literature.

6.4.1 Prior Works

There are full software implementations of CRYSTALS-Kyber with old parameter sets in the literature. Previously, Seiler *et al.* (Seiler, 2018) optimized NTT, INTT and polynomial multiplication operations utilizing AVX2 instructions on Skylake and Haswell architecture processors. Later, they proposed a faster design in (Lyubashevsky & Seiler, 2019) using a new modular reduction technique. Another implementation of the CRYSTALS-Kyber scheme is published by Botros *et al.* (Botros et al., 2019). They proposed a new NTT module for new CRYSTALS-Kyber parameters with optimizations, such as merging NTT layers and instruction alignment of polynomials on ARM Cortex-M4 processors. Similarly, Alkim *et al.* (Alkim et al., 2020) proposed CRYSTALS-Kyber implementation on ARM Cortex-M4 which outperforms the work in (Botros et al., 2019). There are also hardware accelerators for CRYSTALS-Kyber proposed in the literature with new

CRYSTALS-Kyber parameters. In (Huang et al., 2020), the authors proposed full hardware implementation of CRYSTALS-Kyber with new parameters utilizing resource reusing. Recent work (Xin et al., 2020) implements PQC specific schemes on a vector co-processor with RISC-V SCR1 processor targeting ASIC platform. This work supports CRYSTALS-Kyber with new parameters. The study in (Fritzmann et al., 2020) implements an NTT accelerator with RISC-V architecture which can be applied to different PQC schemes. Alkim *et al.* (Alkim et al., 2020) proposed an instruction set extension to RISC-V for improving PQC schemes including CRYSTALS-Kyber. In (Xing & Li, 2021), the authors proposed a compact co-processor highly optimized for CRYSTALS-Kyber. Some of these works do not provide results for operations separately, so they are not included or are shown partially on Table 6.1. There are also hardware accelerators proposed for old CRYSTALS-Kyber parameters (Mert et al., 2020), (Pöppelmann & Güneysu, 2013), (Sinha Roy et al., 2014), (Banerjee et al., 2019), (Chen, Ma, Chen, Lin & Jing, 2020), (Chen, Ma, Chen, Lin & Jing, 2021), which are not included in the comparison.

6.4.2 Implementation Results

We developed three hardware architectures with one, four and sixteen butterfly units (lightweight, balanced and high-performance, respectively) proposed in this work into Verilog modules. Then, they are synthesized, placed & routed for different FPGA families. The proposed hardware architectures are first implemented for Spartan-6 FPGA (xc6slx75fgg676-3) using Xilinx ISE 14.7 with default synthesis options. Then, they are implemented for Artix-7 FPGA (xc7a200tffg1156-3) using Xilinx Vivado 2018.1 with default synthesis options. Implementation results of our architectures and the works in the literature are shown in Table 6.1. The proposed modular reduction and butterfly units are also synthesized, placed & routed as separate units for Spartan-6 and Artix-7 FPGAs. The modular reduction unit uses 236 LUTs with 154 MHz and 195 LUTs with 212 MHz for Spartan-6 and Artix-7, respectively. Butterfly unit uses 377 LUTs, 242 DFFs, 1 DSPs with 129 MHz and 312 LUTs, 207 DFFs, 1 DSPs with 192 MHz for Spartan-6 and Artix-7, respectively.

There are not many prior works in the literature using the new CRYSTALS-Kyber parameter set. However, not all works give performance figures for separate operations. Therefore, it is hard to compare our designs for each operation and parameter set with other works. Our high-performance hardware architecture outperforms the prior works for polynomial multiplication operation in terms of latency. The

proposed high-performance hardware shows up to 99×, 98× and 89× better performance for NTT, INTT and polynomial multiplication, respectively, compared to the high-speed software implementation on ARM Cortex-M4 (Alkim et al., 2020). Our high-performance design is also 6× faster for NTT operation compared to the software implementation of the CRYSTALS-Kyber PQC scheme with old parameter sets on Intel processors with Skylake and Haswell architectures (Seiler, 2018). For hardware implementations, our high-performance design shows better latency performance for NTT operation than the prior works in the literature except for the work in (Xin et al., 2020) proposing an ASIC design with a large 521K gate area. Our high-performance design accelerates NTT operation on the FPGA platform up to 28× compared to the work in (Fritzmann et al., 2020) which uses a 16-bits q. Our balanced design also shows better performance in terms of latency than the prior works in the literature. We also provide utilization results of our works for low-cost Spartan-6 FPGAs. The works (Seiler, 2018) and (Lyubashevsky & Seiler, 2019) are using Skylake CPUs which are not suitable for embedded systems. That makes our design affordable and practical in the use of the CRYSTALS-Kyber PQC scheme in embedded systems.

Table 6.1 Implementation Results and its Comparison to Prior Work

Work	**Platform**	n	q / $\lceil \log_2(q) \rceil$	**LUT/REG/DSP/BRAM**	**Clock (MHz)**	**Latency (Cycle Count)**		
						NTT	**INTT**	**PM**[d]
(Seiler, 2018)[a]	Intel Corei7-6600U	256	7681 / 13	– / – / – / –	–	419	394	1278
	Intel Core i7-4770K	256	7681 / 13	– / – / – / –	–	460	440	1432
(Botros et al., 2019)[a,b,c]	ARM Cortex-M4	256	7681 / 13	– / – / – / –	–	9452	10373	32576
		256	3329 / 12	– / – / – / –	–	7725	9347	27873
(Alkim et al., 2020)[a,b,c]	ARM Cortex-M4	256	3329 / 12	– / – / – / –	–	6855	6983	23018
(Alkim et al., 2020)[a,b]	Artix-7	256	3329 / 12	1738 / 1599 / 4 / 34	60	8595	9427	30667
				1842 / 1634 / 5 / 34	60	6868	6367	22498
(Fritzmann et al., 2020)[a,b]	Zynq-7000	256	– / 16	2908 / 170 / 9 / –	–	1935	1930	–
(Huang et al., 2020)[b,c]	Artix-7, Virtex-7	256	3329 / 12	– / – / – / –	225	1834	–	–
(Xing & Li, 2021)[b,c]	Artix-7	256	3329 / 12	1737 / 1167 / 2 / 3	161	512	512	1792
(Xin et al., 2020)[a,b]	28nm CMOS	256	3329 / 12	512K / – / – / –	–	41	–	–
		256	7681 / 13			45		
Ours-1 BTF[b,c]	Spartan-6	256	3329 / 12	985 / 444 / 1 / 5	138	904	904	3359
	Artix-7			948 / 352 / 1 / 2.5	190			
Ours-4 BTFs[b,c]	Spartan-6			2498 / 1046 / 4 / 18	127	232	233	864
	Artix-7			2543 / 792 / 4 / 9	182			
Ours-16 BTFs[b,c]	Spartan-6			9898 / 3688 / 16 / 70	115	69	71	256
	Artix-7			9508 / 2684 / 16 / 35	172			

[a]:Works with multiple n and q. [b]:Supports new CRYSTALS-Kyber parameters. [c]:Optimized for constant q. [d]:**PM**: INTT(CWM(NTT(A), NTT(B))).

6.5 Summary

In this work, we present three hardware architectures (lightweight, balanced, high-performance) performing NTT, INTT and polynomial multiplication operations for the CRYSTALS-Kyber scheme. These operations are extensively utilized in key generation, encryption and decryption operations of CRYSTALS-Kyber and our proposed polynomial multiplier hardware can be utilized as a hardware accelerator for these operations, hence for CRYSTALS-Kyber. Compared to the high-speed software implementations on Cortex-M4 (Alkim et al., 2020), the proposed high-performance hardware shows up to 99×, 98× and 89× better performance for NTT, INTT and polynomial multiplication, respectively.

7. AN EXTENSIVE STUDY OF FLEXIBLE DESIGN METHODS FOR THE NUMBER THEORETIC TRANSFORM

In this chapter, we evaluate two different design methods for *design-time* flexible NTT implementations. In our first approach, we present a parametric NTT hardware generator that provides design-time configurability for both algorithm parameters and throughput. The proposed hardware generator takes the degree of polynomial (n), coefficient size ($\lceil \log_2(q) \rceil$), the number of processing elements as inputs and generates the corresponding hardware. In the second approach, we present an HLS-based NTT implementation and we investigate the impact of different pragmas and HLS parameters on the performance and flexibility of the generated hardware. Finally, we present a comprehensive analysis of all the resulting designs and compare different design methods for implementing NTT architectures. [1]

7.1 Introduction

Flexibility is a key requirement for digital systems to reduce design costs (Keutzer, Newton, Rabaey & Sangiovanni-Vincentelli, 2000), (Verbauwhede & Schaumont, 2005) and consolidate changing standards, performance requirements, target platforms, and algorithmic alternatives (Milder, Franchetti, Hoe & Püschel, 2012). It is especially important for next-generation cryptographic systems such as lattice-based cryptography. There are three primary design methods to build *design-time* flexibility: (*i*) a parametric hardware generator with customized parameters, (*ii*) a software compiled on a processor, and (*iii*) a software directly transformed into hardware through an HLS tool. Among these options, software tends to be the most flexible solution with the worst performance and the parametric hardware is often the most efficient one with the worst flexibility. HLS, by comparison, forms

[1] This chapter presents the works in (Mert et al., 2020) and (Mert et al., 2020).

the middle ground, resulting in medium performance and flexible solutions. In this work, we will investigate the first and the last methods.

Efficient lattice-based cryptography operates with polynomial rings and polynomial multiplication is a well-known computational bottleneck of lattice-based cryptosystems—indeed, schoolbook multiplication corresponds to 95.7% and 98.8% percent of the overall computation time for the encryption and decryption processes in a lattice-based public-key encryption and decryption scheme, respectively (Pöppelmann & Güneysu, 2014). The NTT reduces the $\mathcal{O}(n^2)$ complexity of the schoolbook polynomial multiplication to the quasi-linear complexity of $\mathcal{O}(n \cdot \log_2(n))$. NTT is thus a major building block of lattice-based cryptography implementations.

There are two design-time flexibility requirements for specialized NTT designs. The first one is due to varying algorithmic parameters which are the degree of the polynomial, n, and coefficient size, $k = \lceil \log_2(q) \rceil$. For example, while NewHope algorithm (Alkim et al., 2016) uses polynomials of degree 1024 with 14-bit coefficients, most HE schemes operate with larger parameters. For example, CryptoNets (Brutzkus et al., 2019) operates with polynomials of degree 4096 with up to 60-bit coefficients. The second flexibility need is due to a consequence of performance requirements of the applications, even for a fixed algorithm. For instance, while a cloud computing infrastructure demands high-throughput hardware, an IoT/embedded device would favor a low area/energy design. Therefore, throughput is the second flexibility parameter, mainly determined by the number of processing elements (PEs), which carry out the fundamental arithmetic operations in the design.

In Table 7.1, we report a summary of the previous works in the literature, which lists the specific setting for parameters, the design method, and the target platform thereof. The prior NTT designs have so far been fixed in both aspects of algorithm parameters and throughput. Notwithstanding, extending the hardware from a specific setting to a more generic and flexible design is non-trivial due to memory access and control flow challenges. Although there are NTT hardware designs in the literature offering run-time configurability for algorithm parameters, they have fixed throughput (Banerjee et al., 2019), (Song et al., 2018). Thus, as various settings in Table 7.1 indicate, the quest for a flexible and efficient design that supports a wide range of parameters is still outstanding.

This work provides one of the first evaluation of *design-time* flexible NTT designs and uses the Xilinx FPGA devices as the common demonstrator platform. To that end, we investigate two different design approaches. In our first approach, we propose an optimized, parametric hardware generator for the NTT operation. This design offers flexibility for both algorithm parameters and throughput, and supports a

Table 7.1 Previous NTT Implementations

Method	Work	n	$\lceil \log_2(q) \rceil$	PE	Target Platform
HLS	(Ozcan & Aysu, 2020)[c]	1024	14	10	Virtex-7
	(Kawamura et al., 2018)[c]	1024	10	–	Virtex-7
	(Millar, 2019)[c]	512	17	–	Zynq US+
Software	(Botros et al., 2019)[a,b]	256	12	2	ARM Cortex-M4
	(Seiler, 2018)	256 1024	13 14	4	Intel Core i7-4770K
	(Alkim et al., 2016)[a,b]	1024	14	1	ARM Cortex-M0 ARM Cortex-M4
	(Mert et al., 2020),(Microsoft, 2019)	1024	27	–	Intel Core i9-7900X
Hardware	(Aysu et al., 2013)[a]	256	17	1	Spartan-6
	(Pöppelmann & Güneysu, 2013)[a,b]	256	13	1	Virtex-6
	(Sinha Roy et al., 2019)[b]	4096	30	2	Zynq US
	(Öztürk et al., 2017)[b]	32768	32	256	Virtex-7
	(Mert et al., 2019)[b]	1024	32	1 64	Spartan-6 Virtex-7
	(Mert et al., 2020)[b]	1024	32	32 64	Virtex7
	(Sinha Roy et al., 2014)[c]	256	13	1	Virtex-6
	(Banerjee et al., 2019)[c]	256 512 1024	13 14 14	1	40nm CMOS
	(Song et al., 2018)[c]	256	13	16	40nm CMOS
	(Fritzmann & Sepúlveda, 2019)[c]	256	13	1	UMC 65nm
	(Xing & Li, 2020)[a,b]	1024	14	4	Artix-7
	(Zhang et al., 2020)[b]	16384	32	1	Virtex-7
	(Sinha Roy et al., 2018)[b]	65536	30	16	Virtex-6

[a]:Uses fixed q. [b]:Uses fixed n. [c]:Supports multiple n and q.

wide range of cryptographic algorithms. First, it can cater different arithmetic structures for varying polynomial degrees and coefficient sizes. Second, it can provide a trade-off in area vs. performance by incorporating a different number of PEs. The user of our generator simply enters polynomial degree and coefficient size and a desired number of PEs, and our tool automatically produces a corresponding efficient hardware [2]. Prior works, by contrast, are either ad-hoc efforts fixed for a specific setting (Aysu et al., 2013), (Pöppelmann & Güneysu, 2013), (Sinha Roy et al., 2019), (Öztürk et al., 2017), (Mert et al., 2019), (Mert et al., 2020), (Xing & Li, 2020), (Zhang et al., 2020), (Sinha Roy et al., 2018) or employ a fixed number of PEs (Banerjee et al., 2019), (Sinha Roy et al., 2014), (Song et al., 2018), (Fritzmann & Sepúlveda, 2019). We furthermore investigate the HLS approach for generating flexible NTT designs and provide implementation results. We finally analyze the resulting implementations and compare them with each other and to prior fixed NTT designs in the literature.

[2]Code is available at `https://github.com/acmert/parametric-ntt`

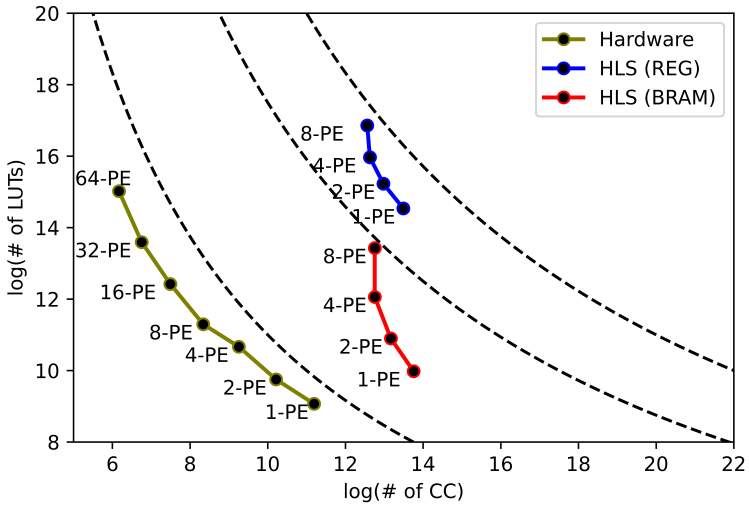


Figure 7.1 An overview of the design method's results and comparison for the NTT of NewHope-512. The hand-tuned hardware designs lead to most efficient results.

For both approaches, we implement the NTT designs with parameters suitable for post-quantum KEMs (CRYSTALS-Kyber (Bos et al., 2018), NewHope (Alkim et al., 2016)), DS schemes (CRYSTALS-Dilithium (Ducas et al., 2018), Falcon (Fouque et al., 2018), qTESLA (Alkim et al., 2019)) as well as HE applications (SEAL (Microsoft, 2019), CryptoNets (Brutzkus et al., 2019)) using the parametric hardware generator and HLS, and quantify the area-cost and the latency of the resulting designs.

Fig. 7.1 shows the summary of our results for a particular NTT with $n = 512$ and $\lceil \log_2(q) \rceil = 14$ used in NewHope-512. The results, in $\log_2$ scale, quantify the superiority of hand-tuned hardware over HLS-based implementation. For a similar area (LUT) cost, the hardware achieves an 11.6× faster design than HLS-based design. Furthermore, the results show that our hardware generator automates the design space exploration of hardware—the results show a coverage of 61.9× in area and 32.5× in latency. Such a coverage is not yet possible with HLS tools because they fail to generate a hardware with more than 8 PEs.

The rest of the chapter is organized as follows. Section 7.2 summarizes the related prior work. Section 7.3 introduces the parametric hardware design with novel optimizations. Section 7.4 discusses the HLS-based design method and tuning the HLS framework for efficient exploration of the design space. Section 7.5 presents implementation results, compares the resulting implementations among themselves and with prior work, and Section 7.6 concludes the chapter.

7.2 Prior Implementations of NTT

Different algorithms and architectures targeting various platforms for NTT are proposed in the literature in order to facilitate practical implementations of lattice-based cryptography. There are many NTT architectures proposed in the literature using different methods and targeting different platforms: low-level hardware implementations ((Aysu et al., 2013), (Pöppelmann & Güneysu, 2013), (Sinha Roy et al., 2019), (Öztürk et al., 2017), (Mert et al., 2019), (Mert et al., 2020), (Sinha Roy et al., 2014), (Banerjee et al., 2019), (Song et al., 2018), (Fritzmann & Sepúlveda, 2019), (Xing & Li, 2020), (Zhang et al., 2020), (Sinha Roy et al., 2018)), HLS implementations ((Ozcan & Aysu, 2020), (Kawamura et al., 2018), (Millar, 2019)) and software implementations ((Brutzkus et al., 2019), (Microsoft, 2019), (Halevi & Shoup, 2014), (Aguilar-Melchor et al., 2016), (Dai & Sunar, 2015), (Botros et al., 2019), (Seiler, 2018)).

The works (Öztürk et al., 2017), (Mert et al., 2019) use the iterative radix-2 DIF NTT algorithm (Chu & George, 1999) and propose balanced NTT implementations considering I/O requirements of the CPU-FPGA system. In (Mert et al., 2020), two different NTT architectures utilizing radix-2 DIF NTT algorithm (Chu & George, 1999), (Longa & Naehrig, 2016) and Four-Step (Dai & Sunar, 2015) NTT algorithms are presented. The NTT architectures are utilized to accelerate the encryption and decryption operations implemented in SEAL library (Microsoft, 2019) on the FPGA. In (Sinha Roy et al., 2019) and (Sinha Roy et al., 2018), the authors implement an iterative version of the NTT algorithm on FPGA; while the work in (Sinha Roy et al., 2019) implements modular reduction operation using a *sliding window* method, (Sinha Roy et al., 2018) uses Barrett modular reduction. The work in (Zhang et al., 2020), which targets both FPGA and ASIC platforms, proposes a method to eliminate the first stage of the NTT operation.

In (Fritzmann & Sepúlveda, 2019), (Banerjee et al., 2019) and (Song et al., 2018), the works target ASIC platform. In (Banerjee et al., 2019), the authors propose a reconfigurable crypto-processor supporting multiple lattice-based cryptosystems. The work in (Banerjee et al., 2019) utilizes the constant-geometry NTT algorithm (Pollard, 1971), which features a constant read/write pattern in each stage of NTT operation and uses single-port RAM in order to reduce hardware cost. Also, it proposes a configurable Barrett modular reduction implementation. In (Fritzmann & Sepúlveda, 2019), the authors focus on low-power NTT design for battery-powered IoT devices with a discussion on countermeasures for side-channel attacks. The

method in (Song et al., 2018) proposes an accelerator for R-LWE based cryptosystems for multiple parameter sets, which uses 16 butterfly units in its NTT core and divides large NTT operations into smaller 64-pt NTTs.

The works in (Ozcan & Aysu, 2020), (Kawamura et al., 2018), (Millar, 2019) use HLS to generate NTT hardware targeting FPGA. These works use HLS-friendly NTT algorithms and HLS directives to generate efficient NTT hardware. In (Ozcan & Aysu, 2020), the authors propose an NTT-based polynomial multiplier implementing the memory-efficient NTT algorithm introduced in (Aysu et al., 2013). In (Kawamura et al., 2018), the *for* loop structure of the NTT algorithm is modified for efficiently applying Vivado HLS directives. In (Millar, 2019), the authors propose an FFT-based polynomial multiplier that uses an FFT algorithm with *ping-pong* memory buffer utilizing 2D arrays in HLS.

Different software implementations for NTT are also proposed in the literature. The software libraries CryptoNets (Brutzkus et al., 2019), SEAL (Microsoft, 2019), HElib (Halevi & Shoup, 2014) and NFLlib (Aguilar-Melchor et al., 2016) provide efficient software implementations of arithmetic blocks for the lattice-based HE schemes, which include efficient NTT implementations. The work in (Botros et al., 2019) proposes a high-speed implementation of CRYSTALS-Kyber (Bos et al., 2018) on ARM Cortex-M4 micro-controller, where an NTT implementation for $n = 256$ and $q = 3329$ was also introduced. The work in (Seiler, 2018) proposes an AVX2 optimized NTT implementation that utilizes a modified Montgomery modular reduction algorithm. In (Alkim et al., 2016), NewHope-1024 (Alkim et al., 2016) is implemented on ARM Cortex-M0 and Cortex-M4 micro-controllers with NTT/INTT implementation optimized for the parameter set of NewHope-1024. Also, there are GPU accelerators for NTT and HE applications such as cuHe (Dai & Sunar, 2015).

7.3 Design Method I: Parametric Hardware Generator Design

NTT computations have three parts: loading related data (steps 5 and 6 in Algorithm 1), performing arithmetic computations (steps 7 and 8 in Algorithm 1) and storing the result (steps 9 and 10 in Algorithm 1). Within each PE, the so-called butterfly units execute arithmetic computations, which are composed of modular addition, subtraction, and multiplication. The Algorithm 1 loops over $\log_2(n)$ stages and performs $n/2$ butterfly operations at each stage. To achieve higher throughput,

it is possible to unroll NTT loops and parallelize the butterfly operations by using multiple PEs.

The parametric NTT hardware generator takes the polynomial degree, the coefficient size, and the number of PEs as input parameters, and generates an optimized NTT hardware that performs NTT operation for the given parameters. It is notable that the design provides flexibility not only for polynomial degree and coefficient size, but also the number of PEs, which determines the throughput of the hardware. The same unit can also execute the INTT. We will discuss the design in a bottom-up fashion, starting with the design of efficient modular multiplication. We will then describe the construction of the PE and finally explain its parallelization along with the optimized memory access structure and the organization.

7.3.1 A Design-time Configurable Word-Level Montgomery Modular Multiplier Unit

The modular multiplier is the key component of NTT arithmetic. This unit consists of two parts: an integer multiplier followed by a modular reduction. In this work, we developed the parameterized version of both parts. The design-time configurable parametric integer multiplier hardware uses the coefficient size ($\lceil \log_2(q) \rceil$) as a parameter. Each input of the multiplier is divided into 16-bit pieces and one DSP block is used for each 16-bit$\times$16-bit multiplication operation. The resulting intermediate values are then added up to calculate multiplication result using carry-save adders. The proposed integer multiplier is also fully pipelined and it can produce one multiplication result per clock cycle. The proposed parametric hardware generator, for $\lceil \log_2(q) \rceil = 32$, generates a 32-bit multiplier similar to the one shown in Fig. 3.1. The number and configuration of the DSP blocks along with the adder tree are automatically synthesized based on the input parameters. Although it may be possible to partition input integers more efficiently, we divide all inputs into 16-bit (power-of-2) pieces to preserve regularity and reduce the complexity of the control unit.

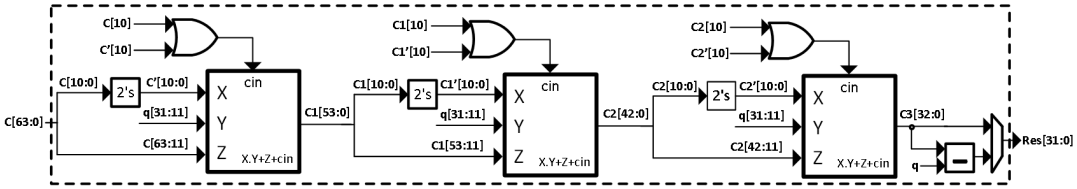
There are mainly two alternative methods for efficient modular reduction: Montgomery (Mert et al., 2020) and Barrett algorithms (Banerjee et al., 2019). Their selection for parametric hardware is a non-trivial design decision. Banerjee *et al.*, for instance, uses a Barrett modular reduction hardware (Banerjee et al., 2019), which includes two different reduction units. The first one is a configurable Barrett reduction hardware which enables run-time flexibility. The second one employs a

separate, specialized reduction hardware that is only compatible with a small set of pre-determined special moduli.

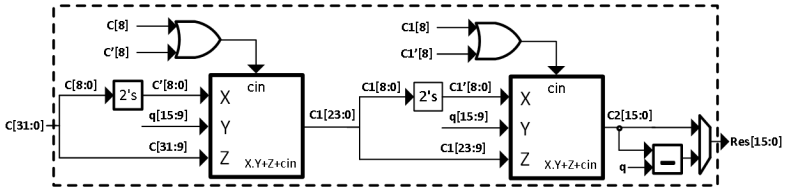
We argue that for the design-time flexibility, Montgomery reduction can offer a more efficient solution than the Barrett algorithm. Nevertheless, the regular Montgomery has to be optimized for NTT primes and extended to support various polynomial degrees and coefficient sizes. To that end, we propose a design-time configurable parametric word-level Montgomery reduction unit that generalizes the word-level Montgomery modular reduction algorithm (Algorithm 6) for NTT-friendly primes presented in Section 3.2.2.2. Compared to the configurable Barrett reduction (Banerjee et al., 2019), our solution either reduces the number of multipliers or results in smaller multiplier units.

Algorithm 6 provides the details of our generalized algorithm. The algorithm requires a different number of integer multipliers of varying bit lengths depending on q and n as detailed in Section 3.2.2.2. The proposed word-level Montgomery modular reduction algorithm divides reduction operation into a set of MAC operations, which can be implemented using DSP blocks in Xilinx FPGAs. Therefore, the algorithm itself is amenable to generating flexible designs. Our hardware generator makes use of this to automatically generate the reduction hardware for the given polynomial degree and coefficient size. After the Montgomery reduction operation, the extra R^{-1} in the result needs to be corrected using an extra multiplication with R. Similar to the approaches presented in Section 3.2.2.2 and Section 4.2.1, since one of the inputs in NTT is the constant twiddle factor (ω), we can fuse the multiplication by pre-computing it ($\omega \cdot R \pmod q$) and loading it into the related memory at design-time to save one multiplication at run-time.

Fig. 7.2 illustrates two examples of modular reduction hardware units for (a) polynomial degree of 1024 with 32-bit modulus and (b) polynomial degree of 256 with 16-bit modulus, where the rectangular boxes represent DSP blocks performing $X \cdot Y + Z + cin$. Both designs offer advantage over the Barrett method (Banerjee et al., 2019)—while the first design uses smaller multiplier units, the second one saves one multiplication. The first implementation requires 3 MAC operations while the second implementation uses only two MAC operations. The proposed reduction hardware is fully pipelined, and it produces one output each clock cycle after filling the pipeline. The generated modular reduction hardware runs in constant time for a given arithmetic configuration.



(a) For $n = 1024$ and $\lceil \log_2(q) \rceil = 32$



(b) For $n = 256$ and $\lceil \log_2(q) \rceil = 16$

Figure 7.2 Word-Level Montgomery Modular Reduction Unit

7.3.2 PEs and Butterfly Units

As we already obtain the efficient hardware for the core modular arithmetic, it is now time to discuss the design of the butterfly units that use modular operations and the PEs that contain butterfly units. Each PE implements the Gentleman-Sande butterfly configuration (Chu & George, 1999) corresponding to steps 5–10 of Algorithm 1. Fig. 7.3 illustrates one PE, which takes two coefficients and one twiddle factor as inputs, perform the butterfly operation, and output two resulting coefficients, namely even (E) and odd (O) coefficients. A PE consists of one modular adder, one modular subtractor, and one modular multiplier for implementing the GS butterfly operation. Each PE also uses three dual-port BRAMs, where two data BRAMs store input and intermediate coefficients while the other, twiddle factor TW BRAM, stores the twiddle factors (with Montgomery correction), which are the design-time constants pre-computed based on input parameters.

Fig. 7.3 depicts that the even coefficient is the output of the modular addition operation (Step 7 in Algorithm 1) and the odd output coefficient is the output of the modular subtraction and multiplication (Step 8 in Algorithm 1). To synchronize the output generation of even and odd coefficients, the proposed design inserts a parametric number of registers, shown in green, on the even path based on the input parameters (namely, the polynomial degree and the coefficient size). The number of PEs can only be a power-of-2 and their maximum is $(n/2)$ for an n-pt NTT.

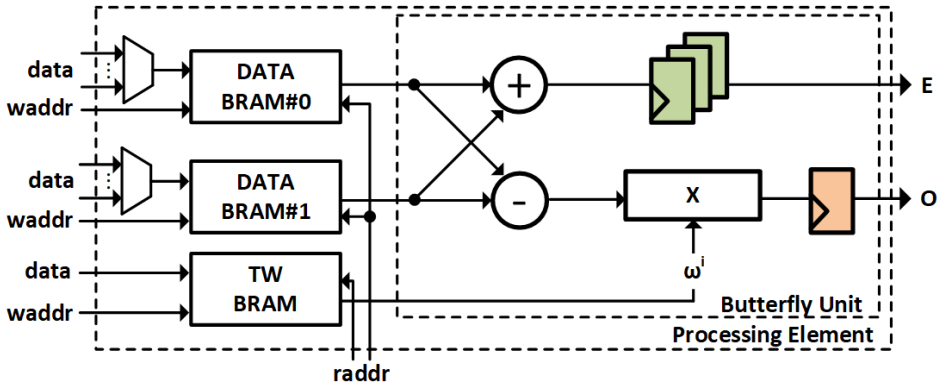


Figure 7.3 PE and the Butterfly Unit

7.3.3 Flexible Memory Access and Overall Design

A significant challenge for the NTT hardware design is managing the complex memory access schedule. This problem becomes more challenging for us because our hardware aims to provide flexibility in the number of core PEs. We need our parametric hardware generator to synthesize the address generation logic that will control the two BRAMs in each PE without adding any stalls to the NTT pipeline.

The proposed design uses the NTT scheme of Algorithm 1, which consists of $\log_2(n)$ stages and performs $(n/2)$ butterfly operations at each stage. Fig. 7.4 (a) demonstrates an example of the memory read access pattern of coefficients for $n = 8$. Each yellow dot represents a butterfly operation, which consumes and produces two coefficients mapping to the same degree. For example, 0^{th} and 4^{th} coefficients in the first stage will correspond to the 0^{th} and 4^{th} coefficients of the second stage in Fig. 7.4 (a).

The irregular access pattern of the NTT enforces storing each coefficient to a unique address. Fig. 7.4 (b) demonstrates the one PE case, where coefficients in the same butterfly operation are stored in two distinct memory blocks. For example, at the first stage of the 8-pt NTT, four coefficient pairs (0, 4), (1, 5), (2, 6), (3, 7) go into butterfly operation. These pairs need to be read at the same clock cycle. Therefore, coefficients 0, 1, 2, 3 and 4, 5, 6, 7 should be stored in separate memory blocks and accessed in parallel. Unfortunately, the pairings of the coefficients change at each stage. The output of a stage, therefore, has to be stored back into the memory blocks based on the pairing of the next stage. Fig. 7.4 (b) shows the example for $n = 8$ where the coefficient pairings for the first and second stages are respectively $\{(0, 4), (1, 5), (2, 6), (3, 7)\}$ and $\{(0, 2), (1, 3), (4, 6), (5, 7)\}$. Hence, both outputs of the pairs (0, 4) and (1, 5) should be written into the first memory block. Likewise, (2, 6) and (3, 7) output will be placed in the second memory. This guarantees that all coefficient pairs at the second stage can be read in a cycle.

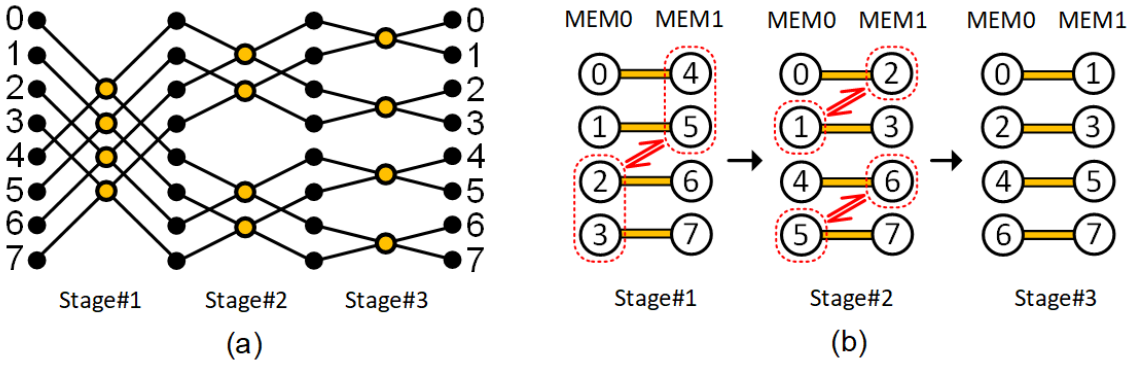


Figure 7.4 (a) Coefficient Access Pattern; (b) Memory Access Pattern for $n = 8$

Our parametric hardware design automatically generates the required access pattern to handle memory access operations for different numbers of PEs. This pattern, however, requires coefficient pairs to be written into the same memory block. For example, for $n = 8$, the coefficient pair (0, 4) should be written into the first memory block after the first stage to improve coalescing. This is enabled by adding one extra register to the output of the modular multiplier unit in the PE, shown as orange register in Fig. 7.3. This extra latency allows storing coefficients into the same memory in 2 cycles. Since the PEs are pipelined, this extra register does not affect the throughput.

The proposed design uses an alternating memory read pattern because the first and second half of coefficient pairs should be written into the first and second memories, respectively. For example, as shown in Fig. 7.4, only the coefficients (0, 1), (4, 5) are written into the first memory while (2, 3), (6, 7) are written into the second memory. Therefore, the memory read pattern for 8-pt NTT should be in the order {(0, 4), (2, 6), (1, 5), (3, 7)} instead of {(0, 4), (1, 5), (2, 6), (3, 7)}. Fig. 7.5 (a) and Fig. 7.5 (b) give the memory access examples for 8-pt NTT with one and two PEs, respectively. Green and blue boxes represent the read and write operations, respectively, and the letters in red represent coefficients written into the memory. For the 8-pt NTT with one PE, coefficients (0, 4) and (1, 5) need to be stored in the same memory block. The proposed addressing scheme thus first reads coefficients (0, 4) and (2, 6) in two consecutive clock cycles, which should be written into the first and second memories, respectively. Therefore, the operation can continue without any stall. For the 8-pt NTT with two PEs (see Fig. 7.5 (b)), the PEs perform the butterfly operation for the first half of the coefficient pairs first. In this case, the first and second PEs can read coefficients (0,4) and (1,5) at the same time because coefficients 0, 1 and 4, 5 will be written into different memory blocks. Since we have two PEs instead of one, the latency of each NTT stage is reduced.

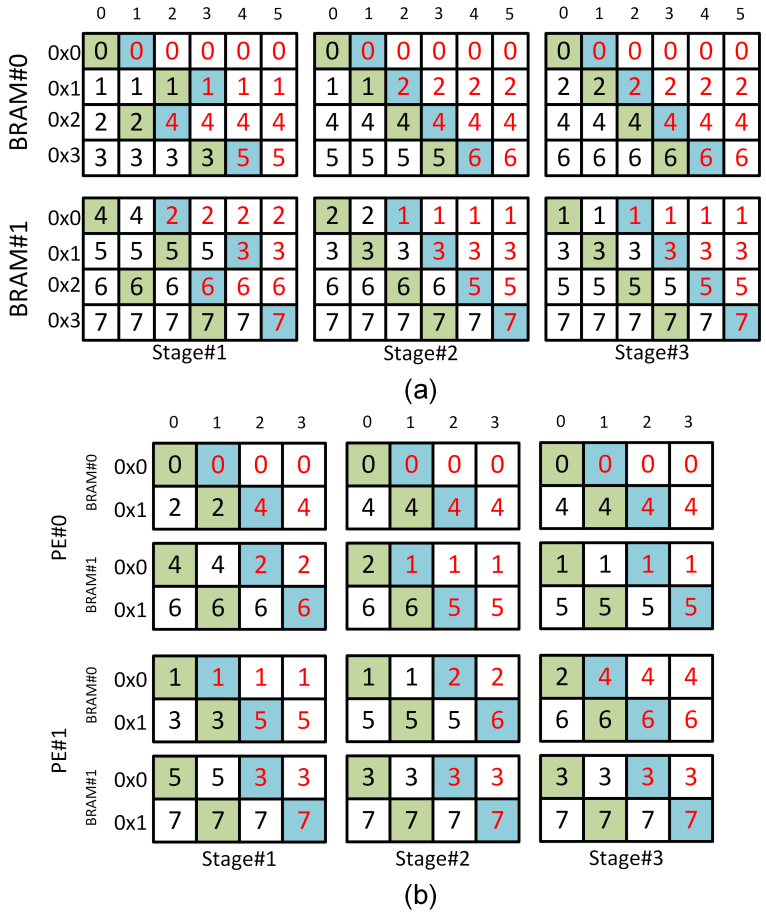


Figure 7.5 Memory Access for 8-pt NTT with (a) one PE, (b) two PEs

Fig. 7.6 outlines the high-level block diagram of the generated NTT hardware with (a) one and (b) two PEs. The outputs of one PE are connected to all PEs in the design to broadcast the coefficients needed due to the memory dependency of the NTT. Before the NTT starts, the hardware first takes twiddle factors followed by the input coefficient as inputs and writes them to their related BRAMs within each PE. The data BRAMs also keep the resulting output coefficients, which are read via output multiplexers.

7.4 Design Method II: HLS-Based Design

We aim to synthesize our design-time flexible NTT hardware using the popular Xilinx Vivado HLS tool, which takes C or C++ codes as input and generates synthesizable Verilog or VHDL codes. It also provides specific directives (*pragmas*) for

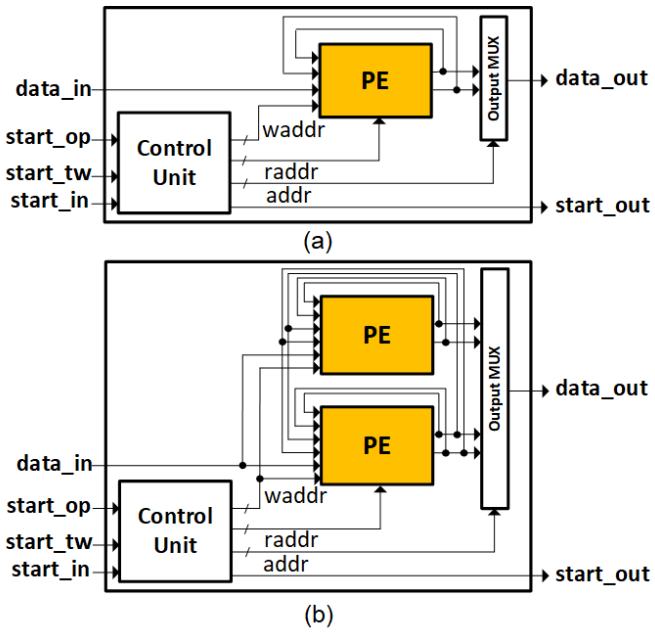


Figure 7.6 NTT Hardware (a) with one PE; (b) two PEs

users to intervene in the C/C++-to-RTL synthesis process and optimize the generated RTL design. The Vivado HLS tool offers several optimization parameters such as `loop unrolling`, `loop merging`, `pipeline` and `array partitioning`, among others (Xilinx, 2014).

To generate an efficient hardware from C/C++ code, the code should include appropriate HLS directives. This may require re-writing or changing the structure of the code (Kawamura et al., 2018), in addition to the exploration of optimization parameters. Indeed, we first observe that the straightforward transition of the software leads to inefficient results or may not even produce a synthesizable code. Specifically, in the original algorithm we used for implementing the parametric hardware generator (Algorithm 1), the NTT operation has three loops where indices of middle and inner loops depend on the index of the outer loop. This loop structure makes applying HLS directives complicated and causes the HLS tool to improperly handle the synthesis process.

Algorithm 16 shows our proposed HLS-friendly NTT algorithm, which is a modified version of Algorithm 1. To overcome HLS problems and make the C++ code more HLS-friendly, we modified the loop structure of the Algorithm 1 such that dependencies between indices of the loops are removed and the trip count of each loop structure is fixed. In Algorithm 16, the trip count of the outer loop is the number of stages in the n-pt NTT operation, $\log_2(n)$, as in Algorithm 1. In the middle loop, we modified the trip count to $(n/2)$, which is the number of butterfly operations in one stage of an n-pt NTT. Finally, we set the trip count of the innermost loop as the number of butterfly operations to be performed in parallel, which can be at

Algorithm 16 HLS-Friendly NTT Algorithm

Input: $\boldsymbol{a}(x) \in \boldsymbol{R}_{q,n}$
Input: $\boldsymbol{\omega}$ where $\boldsymbol{\omega}[i] = \omega^i \pmod{q}$ (an array of size $n/2$)
Input: B (number of butterfly operations in parallel)
Output: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_{q,n}$

1: $l, v = \log_2 n, n/2$
2: `STAGE_LOOP:`
3: **for** $(i = 0;\ i < l;\ i{+}{+})$ **do** ▷ *outer loop*
4: `BUTTERFLY_LOOP:`
5: **for** $(s = 0;\ s < v;\ s = s + B)$ **do** ▷ *middle loop*
6: `IDX_CALC_LOOP:` ▷ *index calculation*
7: **for** $(b = 0;\ b < B;\ b{+}{+})$ **do** ▷ *inner loop#1*
8: $j[b] = (s + b) \gg (l - 1 - i)$
9: $k[b] = (s + b)\ \&\ ((v \gg 1) - 1)$
10: $i_e[b] = j[b] \cdot (1 \ll (l - i)) + k[b]$
11: $i_o[b] = i_e[b] \cdot (1 \ll (l - i - 1))$
12: $i_w[b] = (1 \ll i) \cdot k[b]$
13: **end for**
14: `MEM_READ_LOOP:` ▷ *read data*
15: **for** $(b = 0;\ b < B;\ b{+}{+})$ **do** ▷ *inner loop#2*
16: $U[b] = \boldsymbol{a}[i_e[b]]$
17: $V[b] = \boldsymbol{a}[i_o[b]]$
18: $W[b] = \boldsymbol{\omega}[i_w[b]]$
19: **end for**
20: `OP_LOOP:` ▷ *butterfly operation*
21: **for** $(b = 0;\ b < B;\ b{+}{+})$ **do** ▷ *inner loop#3*
22: $E[b] = (U[b] + V[b]) \pmod{q}$
23: $O[b] = (U[b] - V[b]) \cdot W[b] \pmod{q}$
24: **end for**
25: `MEM_WRITE_LOOP:` ▷ *write data*
26: **for** $(b = 0;\ b < B;\ b{+}{+})$ **do** ▷ *inner loop#4*
27: $\boldsymbol{a}[i_e[b]] = E[b]$
28: $\boldsymbol{a}[i_o[b]] = O[b]$
29: **end for**
30: **end for**
31: **end for**
32: **return** $\boldsymbol{a}$

most $(n/2)$ for an n-pt NTT operation. Then, we divided butterfly operation into four separate loops with the same trip count, whereby four loops perform four steps of a butterfly operation: (i) calculating memory addresses for reading coefficients and twiddle factor, (ii) reading coefficients and twiddle factors from the memory, (iii) performing arithmetic operations and (iv) writing output coefficients into the memory.

Based on the implemented algorithm and the code section where the directives

Table 7.2 Vivado HLS *pragmas* Used in Our Work

Code	*pragma*
array $\boldsymbol{a}$	`#pragma HLS ARRAY_PARTITION`
array $\boldsymbol{\omega}$	`#pragma HLS ARRAY_PARTITION`
`STAGE_LOOP`	–
`BUTTERFLY_LOOP`	`#pragma HLS PIPELINE`
`IDX_CALC_LOOP`	`#pragma HLS UNROLL`
`MEM_READ_LOOP`	
`OP_LOOP`	
`MEM_WRITE_LOOP`	

are used, different combinations of directives can have different impacts on the architecture. Therefore, we applied a set of different directives to the C++ implementation of NTT operation and selected `loop unrolling`, `pipeline` and `array partitioning` directives at different code parts as shown in Table 7.2.

The `loop unrolling` directive (UNROLL) converts a single operation performed by a loop into multiple independent copies of the operation performed in the loop body and runs these operations concurrently. It can unroll a loop fully or partially, and it increases the parallelism and the performance of the operation. This directive is applied to the innermost four loops in our design where it generates B butterfly units running concurrently. The `pipeline` directive (PIPELINE) allows concurrent execution of operations and reduces the interval required to start processing a new input, e.g., it allows an operation to take a new input every clock cycle. We use PIPELINE directive at the middle loop to pipeline the four steps of butterfly operations.

Memory control is one of the most challenging parts of the NTT design as it requires many read/write operations with an irregular pattern. The addressing becomes even more complex with the increasing number of parallel butterfly operations. For instance, an NTT operation with $n = 256$ and $B = 8$ requires reading 16 coefficients and 8 coefficients from the arrays $\boldsymbol{a}$ and $\boldsymbol{\omega}$, respectively, at the same time. This enforces a structure with data stored on multiple small memory blocks or a large memory with multiple read/write ports (i.e., a register file). The straightforward software implementation cannot realize this structure—it generates a single BRAM with insufficient bandwidth. But the `array partitioning` (ARRAY_PARTITION) directive can automate the partition of an array into smaller memories or individual registers instead of instantiating a single large memory block. Therefore, it can effectively increase the read/write ports of a memory, and hence, the throughput, at the expense of more hardware resources.

In the proposed design, we applied the ARRAY_PARTITION directive to the array, $\boldsymbol{a}$, storing the input polynomial, and the array, $\boldsymbol{\omega}$, storing twiddle factors. Here, we used two different approaches: (i) we partitioned each array into a register file using the `complete` option of the ARRAY_PARTITION directive, where any data can be read/written at any time, (ii) we partitioned each array into a number of BRAMs using the `block` option of the ARRAY_PARTITION directive so that the algorithm can access multiple data at the same time. Although the former approach generates faster design, it uses high hardware resources due to large multiplexers generated for reading/writing multiple data. We also used the `ap_int.h` library provided by Vivado HLS tool that contains bit accurate models for the C++ code. This helps us to set $\lceil \log_2(q) \rceil$ and bit-lengths of intermediate calculation steps.

Fig. 7.7 shows the design flow of the proposed NTT architecture in Vivado HLS. The flow starts with the C++ implementation of the NTT operation and its verification. Then, Vivado HLS directives are applied to the code, the synthesis is performed and the Verilog code is generated. Finally, the generated Verilog code is verified with a Verilog testbench. The proposed HLS-based design uses regular Montgomery algorithm for the modular reduction operation as specified in Algorithm 5. Although our parametric hardware generator uses the word-level Montgomery algorithm, our HLS-based design with word-level Montgomery algorithm leads to inefficient architectures due to the loop-based structure of the word-level Montgomery algorithm. Also, as the number of PEs is increased, the HLS tool shows a significant increase in hardware resources and synthesis time because the HLS tool cannot resolve unrolling efficiently although it shows similar performance results with the regular Montgomery algorithm. Therefore, we utilize the regular Montgomery algorithm for modular reduction operation in our HLS-based design. The proposed work implements constant-time modular addition and subtraction operations (Mert et al., 2019). Fig. 7.8 illustrates the architecture of the NTT hardware generated by the Vivado HLS tool.

The NTT operation in (Ozcan & Aysu, 2020) is implemented using a loop structure with three nested loops, where trip counts of the inner loops are not fixed. This complicates applying HLS directives efficiently and setting the amount of parallelism in the generated hardware architecture. Our work is superior to the work in (Ozcan & Aysu, 2020) because our modified loop structure uses fixed trip counts which ease applying HLS directives and we can change the parallelism easily by tuning a single parameter. In (Millar, 2019) and (Kawamura et al., 2018), the authors modify the NTT loop structure such that it uses two nested loops with fixed trip counts. However, both works lack flexibility for setting the level of parallelism. The method in (Millar, 2019) manually unrolls its inner loop with a factor of two which limits the

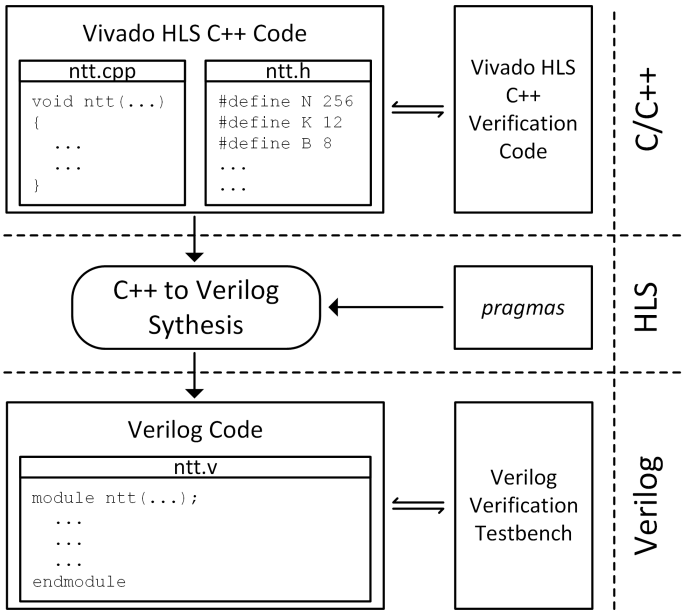


Figure 7.7 Xilinx Vivado HLS Flow

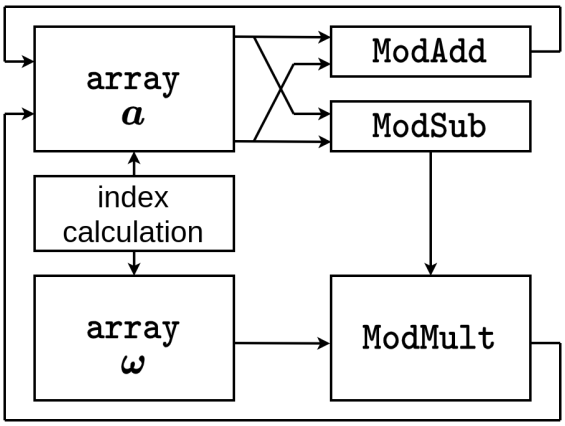


Figure 7.8 NTT Hardware Generated by Xilinx Vivado HLS Tool

parallelism and throughput flexibility. Although it shows better performance for a single parameter set, hardware resource usage details are not provided (as shown in Section 7.5). Similarly, the method in (Kawamura et al., 2018) uses manual unrolling and provides only partial flexibility by changing the structure of the code. In our work, the code structure is fixed and only parameter sets are changed for yielding different architectures.

A key takeaway of our HLS experience on NTT was that, in order to generate an efficient hardware, the designer needs an in-depth understanding of the resulting hardware designs and the effect of tool's parameters on the synthesized hardware, which arguably contradicts the goal of enabling easy solutions for software developers. Section 7.5 discusses the design complexity of the HLS method with respect to the others and compares the efficiency of resulting hardware.

7.5 Results and Comparison

To provide a fair comparison of the design methods, we use a common set of EDA tools and we target the same FPGA. We also compare our work with previous results in the literature and comment on the design complexity.

7.5.1 Experimental Setup

The parametric hardware generator code is written in Verilog RTL and the generated NTT hardware is synthesized, placed & routed using Xilinx Vivado tools on the Xilinx Virtex-7 FPGA (xc7vx690tffg1761-2). The proposed HLS-friendly software code is written in C++ and is enhanced with Xilinx HLS pragmas.

7.5.2 Implementation Results of the Design Methods

For our parametric hardware generator, the area-cost, number of clock cycles needed to finish one NTT, and LUT/latency improvements for 7 different $(n, \lceil \log_2(q) \rceil)$ sets with 1, 8 and 32 PEs are shown in Table 7.3. As the number of PEs increases from 1 to 32, the latency improves up to 25.4× at the expense of more hardware resources. The increase in the number of PEs (which can be as much as $(n/2)$) further improves the performance for larger n values. As expected, the parametric hardware generator yields (at least an order-of-magnitude) better results than the HLS-based implementation method in latency and/or area.

For HLS implementations, the area-cost, number of clock cycles to finish one NTT and latency improvements for 7 different $(n, \lceil \log_2(q) \rceil)$ sets with 1 and 8 PEs are shown in Table 7.4. We obtained two different synthesis results, register memory and BRAM memory, for each parameter set as explained in Section 7.4. In Table 7.4, the first and the second rows reflect the results for the implementation with register and BRAM memories, respectively, for a given parameter set. The implementations with register memory show slightly better performance at the expense of more hardware resources. The first 4 rows of Table 7.4 highlight the impact of the Montgomery modular reduction algorithm and the proposed HLS-friendly NTT al-

Table 7.3 Our Hardware Implementation Results

PE	$(n, \lceil \log_2(q) \rceil)$	**LUTs/DSPs/BRAMs**	**# of CC**	**LUT-Lat. Impr.**
1	(256, 13)	489 / 3 / 2.5	1056	–
8		2371 / 24 / 12	160	$\times 0.20, \times 6.60$
32		15888 / 96 / 48	64	$\times 0.03, \times 16.5$
1	(256, 23)	888 / 7 / 5	1096	–
8		5071 / 56 / 12	200	$\times 0.15, \times 5.48$
32		30847 / 224 / 48	104	$\times 0.02, \times 10.5$
1	(512, 14)	537 / 3 / 5.5	2340	–
8		2514 / 24 / 12	324	$\times 0.21, \times 7.20$
32		16983 / 96 / 48	108	$\times 0.03, \times 21.6$
1	(1024, 14)	575 / 3 / 11	5160	–
8		2584 / 24 / 16	680	$\times 0.22, \times 7.58$
32		17188 / 96 / 48	200	$\times 0.03, \times 25.8$
1	(1024, 29)	966 / 7 / 21.5	5210	–
8		6788 / 56 / 24	730	$\times 0.14, \times 7.13$
32		38093 / 224 / 48	250	$\times 0.02, \times 20.8$
1	(2048, 30)	991 / 7 / 45	11363	–
8		6821 / 56 / 44	1507	$\times 0.15, \times 7.54$
32		38598 / 224 / 64	451	$\times 0.02, \times 25.2$
1	(4096, 60)	2720 / 31 / 180	24708	–
8		23215 / 248 / 176	3276	$\times 0.11, \times 7.54$
32		99384 / 992 / 176	972	$\times 0.02, \times 25.4$

gorithm. The Montgomery modular reduction algorithm improves the performance up to 5× and the HLS-friendly NTT algorithm further improves the performance up to 767× compared to the baseline design (Algorithm 1 without Montgomery modular reduction algorithm) with similar hardware resources. Compared to the parametric hardware generator, the increase in the number of PEs has a weaker effect—the performance increases only by 2× as the number of PEs is increased by 8×, which reveals the inefficiencies in the HLS tools. Vivado HLS cannot synthesize the hardware with register memory for $(2048, 30)$ and $(4096, 60)$ because the `array partitioning` pragma is limited to arrays with length up to 1024. The number of PEs is limited to 8 because no significant performance improvement is observed with 16 or more PEs.

Table 7.4 Our HLS-Based Implementation Results

PE	$(n, \lceil \log_2(q) \rceil)$	**LUTs/FFs/DSPs/BRAMs**	**# of CC**	**Lat. Impr.**
$1^{(1)}$	(256, 13)	12336 / 6639 / 1 / –	3934226	–
		1045 / – / 1 / 2	4065298	–
$1^{(2)}$		12185 / 6390 / 3 / –	788498	×5.0
		893 / – / 3 / 2	919570	×4.4
$1^{(3)}$		12255 / 10197 / 3 / –	5124	×767.8
		979 / – / 3 / 2	6147	×661.3
$8^{(4)}$		61297 / 11187 / 24 / –	2436	×1615
		10487 / – / 24 / 16	3075	×1322
1	(256, 23)	12553 / 18763 / 6 / –	4100	–
		1213 / – / 6 / 6	5123	–
8		63241 / 19387 / 48 / –	2308	×1.8
		12565 / – / 48 / 32	3075	×1.7
1	(512, 14)	23757 / 21727 / 3 / –	11524	–
		1010 / – / 3 / 4	13827	–
8		118804 / 22841 / 24 / –	6050	×1.9
		11020 / – / 24 / 16	6915	×2.0
1	(1024, 14)	36061 / 43239 / 3 / –	25604	–
		1045 / – / 3 / 4	30723	–
8		167018 / 44373 / 24 / –	13442	×1.9
		11305 / – / 24 / 16	15363	×2.0
1	(1024, 29)	36405 / 89454 / 12 / –	25604	–
		1445 / – / 12 / 6	30723	–
8		169560 / 91578 / 96 / –	13442	×1.9
		13975 / – / 96 / 32	15363	×2.0
1	(2048, 30)	–	–	–
		1479 / – / 12 / 6	67587	–
8		–	–	–
		13886 / – / 96 / 64	33795	×2.0
1	(4096, 60)	–	–	–
		2145 / – / 45 / 22	147459	–
8		–	–	–
		17768 / – /360 / 128	73731	×2.0

(1): DIF NTT algorithm (no Mont. Mod. Red.).
(2): DIF NTT algorithm (with Mont. Mod. Red.).
(3): HLS-friendly NTT algorithm (with Mont. Mod. Red.).
(4): HLS-friendly NTT algorithm (with Mont. Mod. Red. and 8 PE).

7.5.3 Comparison to Prior Work

The target devices are implemented under different FPGA technology or even ASIC, hence, the comparison should serve as a first-order estimate rather than an idealized

method. Also, note that only the NTT implementations of the same polynomial degree (n) and coefficient size ($\lceil \log_2(q) \rceil$) make a meaningful comparison. In Table 7.5 and Table 7.6, a subset of our parametric hardware and HLS-based implementations for selected parameter sets are compared with the prior works, respectively.

Albeit target device and technology differences, the results in Table 7.5 and Table 7.6 show that our parametric NTT hardware generator can outperform most of the existing hardware and high-level synthesis designs respectively by up to 51.2× and 82.8× in terms of the number of cycle count. Our parametric generator can produce a hardware that is comparable to fixed setting hardware units and can even be better in some cases. Our designs can achieve either a lower area or a faster design (in clock cycle) compared to prior FPGA solutions. For example, our one PE design outperforms the work in (Banerjee et al., 2019) in terms of latency (cycle count). Some implementations, by contrast, show better performance results than our parametric generator since these implementations are optimized for fixed parameters. For example, the works in (Mert et al., 2019) and (Mert et al., 2020) (works presented in Section 3) show better performance than our work since they utilize 64 PEs. However, our parametric hardware generator can outperform these implementations by simply increasing the number of PEs. Our HLS implementations with 1 and 8 PEs show similar area×latency performance with the works in (Ozcan & Aysu, 2020) and (Kawamura et al., 2018), respectively.

Using our parametric hardware generator, we instantiate the hardware implementations of NTT for the parameters of CryptoNets ($n = 4096$ and $\lceil \log_2(q) \rceil = 60$) (Brutzkus et al., 2019) and qTESLA ($n = 1024$ and $\lceil \log_2(q) \rceil = 14$) (Alkim et al., 2019), which outperform the reference software by up to 25.3× and 5.5× on Intel Xeon processor, and 95.7× and 76.5× compared to HLS-based design, respectively.

We finally highlight the fast design-space exploration that can be achieved by our parametric hardware generator. To test this aspect, we sweep the parameter that controls the number of PEs used in NewHope-512 hardware and report the implementation results in Fig. 7.1. Our generator is able to cover a space of 61.9× in area cost for a tradeoff of 32.5× in performance by simply tuning a parameter knob. By contrast, 5× in area and 1.9× in performance is achievable in HLS since it cannot parallelize the hardware beyond 8 PEs. By contrast, our parametric generator can parallelize up to $(n/2)$ as long as the resulting hardware fits in the FPGA.

Table 7.5 A Summary of Our Hardware Implementation Results and its Comparison to Prior Works

Work	**Platform**	n	$\lceil \log_2(q) \rceil$	**LUT / REG / DSP / BRAM**	**Clock (MHz)**	**Latency**	
						CC	μs
(Aysu et al., 2013)[a]	Spartan-6	256	17	250 / – / 3 / 2	–	–	25
		512		240 / – / 3 / 2		–	50
		1024		250 / – / 3 / 2		–	100
(Sinha Roy et al., 2019)[b]	Zynq US	4096	30	64K / – / 200 / 400	225	–	73
(Mert et al., 2019)[b]	Spartan-6	1024	32	1208 / – / 14 / 14	212	–	12
	Virtex-7			34K / 16K / 476 / 228	200	80	0.4
(Mert et al., 2020)[b]	Virtex-7	1024	32	67K / – / 599 / 129	200	140	0.7
				77K / – / 952 / 325.5		80	0.4
(Sinha Roy et al., 2014)[c]	Virtex-6	256	13	1349 / 860 / 1 / 2	313	1691	5.4
		512	14	1536 / 953 / 1 / 3	278	3443	12.3
(Banerjee et al., 2019)[c]	40nm CMOS	256	13	106K / – / – / –	72	1289	17
		512	14			2826	32
		1024	14			6155	81
(Fritzmann & Sepúlveda, 2019)[c]	UMC 65nm	256	13	14K / – / – / –	25	2056	82
		512	14			4616	184
		1024	14			10248	409
(Xing & Li, 2020)[a,b]	Artix-7	1024	14	4823 / 2901 / 8 / –	153	1280	–
qTESLA (Alkim et al., 2019)[a,b]	Intel Xeon CE5-1650	1024	28	– / – / – / –	–	–	11
CryptoNets (Brutzkus et al., 2019)[a]	Intel Xeon CE5-1650	4096	60	– / – / – / –	–	–	195
Ours-1 PE	Virtex-7	1024	14	575 / – / 3 / 11	125	5160	41.2
		4096	60	2720 / – / 31 / 180		24708	197.6
Ours-8 PE	Virtex-7	1024	14	2584 / – / 24 / 16	125	680	5.4
		4096	60	23215 / – / 248 / 176		3276	26.2
Ours-32 PE	Virtex-7	1024	14	17188 / – / 96 / 48	125	200	1.6
		4096	60	99384 / – / 992 / 176		972	7.7

[a]:Uses fixed q. [b]:Uses fixed n. [c]:Works with multiple n and q.

Table 7.6 A Summary of Our HLS-based Implementation Results and its Comparison to Prior Works

Work	**Platform**	n	$\lceil \log_2(q) \rceil$	**LUT / REG / DSP / BRAM**	**Clock (MHz)**	**Latency**	
						CC	μs
(Ozcan & Aysu, 2020)	Virtex-7	1024	14	4737 / 3243 / 8 / 2	–	16569	76
(Kawamura et al., 2018)	Virtex-7	1024	10	38984 / 30498 / 19 / 21.5	100	5291	53
		2048		46738 / 38224 / 21 / 24.5		10731	107
		4096		58082 / 44767 / 22 / 41.5		22072	221
(Millar, 2019)	Zynq US+	512	17	– / – / – / –	250	1202	–
Ours-1 PE	Virtex-7	1024	14	1045 / – / 3 / 4	100	30723	307
Ours-8 PE	Virtex-7	1024	14	11305/ – / 24 / 16	100	15363	153

7.6 Summary

This study conducts an extensive study of flexible design methods for NTT, proposes a flexible yet efficient hardware generator, and compares its efficiency against the HLS-based design approach and other works in the literature. The results show the superiority of hand-tuned, parameterized hardware designs over other techniques (which is expected) and the inefficiencies of HLS tools. Therefore, this work calls for better HLS tools that can close the order(s)-of-magnitude gap compared to RTL-based designs.

8. CONCLUSION AND FUTURE WORK

In this chapter, we present a summary of each work in this dissertation with a conclusion and potential future research areas.

8.1 Conclusions

This dissertation focuses on high-performance and efficient hardware implementations for lattice-based cryptography primitives. Specifically, we present hardware implementations for two main applications of lattice-based cryptography: HE and PQC. The first three chapters focus on HE applications. We first present high-performance and flexible hardware architectures that perform NTT, INTT and NTT-based polynomial multiplication operations, which are excessively utilized in lattice-based cryptographic applications. We use these high-performance blocks to design and implement efficient hardware architectures for encryption, decryption and homomorphic multiplication operations of the full RNS variant of the BFV scheme proposed in (Bajard et al., 2017) for the FPGA. We show that core hardware implementations on the FPGA take leverage of the intrinsically parallelizable structure of these operations and improve their performance compared to their pure software implementations. We also design and implement a proof-of-concept framework that establishes high-speed communication between the CPU and FPGA via PCIe link. We show that the proposed encryption and decryption hardware architectures can be utilized as actual accelerators, and provide up to one order of magnitude speedup for the encryption and decryption operations of the BFV scheme compared to the highly-optimized HE library SEAL (Microsoft, 2019) using the proposed framework. Therefore, we show that utilizing efficient FPGA accelerators for HE libraries such as SEAL is very promising and can be an enabler for the deployment of practical real-time HE applications. Although it is shown that the proposed architectures

are enablers for high-performance applications, we also observe that their impact is bounded by the I/O limitations of the FPGA devices.

In the second part of the thesis, we present one of the earliest hardware accelerators for the polynomial multiplication operation of CRYSTALS-Kyber PQC scheme (Bos et al., 2018) adopting a new set of parameters, which require a different method for NTT and NTT-based polynomial multiplication operations. We show that the proposed hardware on a low-cost Spartan-6 FPGA can show up to almost two orders of magnitude performance improvement compared to the high-speed implementation on Cortex-M4 (Alkim et al., 2020). This makes the design suitable and practical in the use of the Kyber PQC scheme in embedded or SoC systems. Also, it can serve as a dedicated accelerator unit coupled with a RISC-V processor (Fritzmann et al., 2020).

As the lattice-based cryptosystems mature and gear towards massive deployment, there will be a heavier emphasis on flexible design methods for faster adoption and design space exploration, which enforce scalable and configurable solutions. To that end, finally, we conduct an extensive study of flexible design methods for NTT, which is frequently used in lattice-based cryptographic schemes with different parameters and performance requirements. We propose a flexible yet efficient hardware generator, which shows comparable performance and area results with existing designs in the literature. We also show that the HLS-based design methodology is not mature enough to replace hand-written RTL designs. Since HLS-based design results reported in this study are an experience of a hardware design expert, this process will be significantly harder for software developers, which is against the target of HLS-based methodology.

Overall, we present a collection of hardware accelerators for lattice-based cryptography. The proposed architectures can be employed in various applications and platforms ranging from high-performance data centers to SoC platforms. HE applications require high processing power due to their algorithmic complexity and cloud computing services are suitable candidates for the applications utilizing HE. To that end, the high-performance architectures presented in this dissertation can be utilized in cloud services as hardware-as-a-service for performance-demanding operations. Similarly, they can be used in SoC platforms with a hardware/software co-design approach where only parallelizable and complex operations are offloaded to the FPGA.

8.2 Future Work

The proposed hardware architectures in this study are fast and they improve the performance of costly homomorphic operations compared to the software implementations of HE operations. The acceleration of HE applications would still be a very important research area due to the performance requirements of HE-based neural networks and the I/O overheads (Pulido-Gaytan, Tchernykh, Cortés-Mendoza, Babenko, Radchenko, Avetisyan & Drozdov, 2021). The communication cost between the CPU and FPGA creates a bottleneck for practical applications. Therefore, investigating approaches to reduce communication costs between CPU and the FPGA for enabling practical applications would be an important research direction.

There are several HE schemes in the literature where each scheme can show better performance when it is employed in a certain application. For example, the BFV (Fan & Vercauteren, 2012) and the BGV (Brakerski et al., 2014) schemes are suitable for homomorphic operations with integers while the CKKS (Cheon et al., 2017) scheme works with real numbers, which can be employed in privacy-preserving machine learning applications. Fully homomorphic encryption scheme over the torus is favored for Boolean operations (Chillotti, Gama, Georgieva & Izabachène, 2020). Besides, a recent study in the literature proposes efficient conversion between ciphertexts of different HE schemes (Chen, Dai, Kim & Song, 2021). Therefore, the design and implementation of flexible and unified hardware accelerators supporting multiple HE schemes would be an important research topic.

In addition to the performance and flexibility requirements of lattice-based cryptography, implementation attacks/defenses such as the fault and side-channel analysis on lattice-based cryptography (Reparaz, Sinha Roy, de Clercq, Vercauteren & Verbauwhede, 2016), (Aysu, Tobah, Tiwari, Gerstlauer & Orshansky, 2018), (Aysu, Orshansky & Tiwari, 2018), (Sarker, Mozaffari-Kermani & Azarderakhsh, 2019) are important emerging research direction.

BIBLIOGRAPHY

Acar, A., Aksu, H., Uluagac, A. S., & Conti, M. (2018). A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, *51*(4), 1–35.

Aguilar-Melchor, C., Barrier, J., Guelton, S., Guinet, A., Killijian, M.-O., & Lepoint, T. (2016). Nfllib: Ntt-based fast lattice library. In *Cryptographers' Track at the RSA Conference*, (pp. 341–356). Springer.

Ajtai, M. (1996). Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, (pp. 99–108).

Albrecht, M. R., Player, R., & Scott, S. (2015). On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046. `https://eprint.iacr.org/2015/046`.

Alkim, E., Barreto, P. S. L. M., Bindel, N., Kramer, J., Longa, P., & Ricardini, J. E. (2019). The Lattice-Based Digital Signature Scheme qTESLA. Cryptology ePrint Archive, Report 2019/085. `https://eprint.iacr.org/2019/085`.

Alkim, E., Bilgin, Y. A., Cenk, M., & Gérard, F. (2020). Cortex-m4 optimizations for {R,M}lwe schemes. Cryptology ePrint Archive, Report 2020/012. `https://eprint.iacr.org/2020/012`.

Alkim, E., Bos, J. W., Ducas, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., & Stebila, D. (2018). Frodokem: practical quantum-secure key encapsulation from generic lattices.

Alkim, E., Ducas, L., Pöppelmann, T., & Schwabe, P. (2016). Post-quantum key exchange—a new hope. In *25th USENIX*, (pp. 327–343).

Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., & Petri, R. (2020). Isa extensions for finite field arithmetic: Accelerating kyber and newhope on risc-v. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2020*(3), 219–242.

Alkim, E., Jakubeit, P., & Schwabe, P. (2016). Newhope on arm cortex-m. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, (pp. 332–349).

Angel, S., Chen, H., Laine, K., & Setty, S. (2018). Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, (pp. 962–979).

Aragon, N., Barreto, P., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Gaborit, P., Gueron, S., Guneysu, T., Melchor, C. A., et al. (2017). Bike: bit flipping key encapsulation. Submission to the NIST Post-Quantum Standardization project.

Aysu, A., Orshansky, M., & Tiwari, M. (2018). Binary ring-lwe hardware with power side-channel countermeasures. In *2018 Design, Automation Test in Europe Conference Exhibition*, (pp. 1253–1258).

Aysu, A., Patterson, C., & Schaumont, P. (2013). Low-cost and area-efficient fpga implementations of lattice-based cryptography. In *2013 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, (pp. 81–86).

Aysu, A., Tobah, Y., Tiwari, M., Gerstlauer, A., & Orshansky, M. (2018). Hor-

izontal side-channel vulnerabilities of post-quantum key exchange protocols. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, (pp. 81–88).

Azarderakhsh, R., Campagna, M., Costello, C., Feo, L., Hess, B., Jalali, A., Jao, D., Koziel, B., LaMacchia, B., Longa, P., et al. (2017). Supersingular isogeny key encapsulation. Submission to the NIST Post-Quantum Standardization project.

Badawi, A. A., Polyakov, Y., Aung, K. M. M., Veeravalli, B., & Rohloff, K. (2018). Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme. Cryptology ePrint Archive, Report 2018/589. `https://eprint.iacr.org/2018/589`.

Badawi, A. A., Veeravalli, B., Mun, C. F., & Aung, K. M. M. (2018). High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2018*(2), 70–95.

Bajard, J.-C., Eynard, J., Hasan, M. A., & Zucca, V. (2017). A full rns variant of fv like somewhat homomorphic encryption schemes. In Avanzi, R. & Heys, H. (Eds.), *Selected Areas in Cryptography – SAC 2016*, (pp. 423–442).

Bajard, J.-C., Eynard, J., Martins, P., Sousa, L., & Zucca, V. (2019). An hpr variant of the fv scheme: Computationally cheaper, asymptotically faster. *IACR Cryptology ePrint Archive*, *2019*, 500.

Banerjee, U., Ukyab, T. S., & Chandrakasan, A. P. (2019). Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2019*(4), 17–61.

Barrett, P. (1986). Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, (pp. 311–323). Springer.

Bernstein, D. J., Chou, T., Lange, T., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Schwabe, P., Sendrier, N., et al. (2017). Classic mceliece: conservative code-based cryptography. Submission to the NIST Post-Quantum Standardization project.

Bernstein, D. J., Chuengsatiansup, C., Lange, T., & van Vredendaal, C. (2017). Ntru prime: reducing attack surface at low cost. In *International Conference on Selected Areas in Cryptography*, (pp. 235–260). Springer.

Bernstein, D. J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., & Schwabe, P. (2019). The sphincs+ signature framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, (pp. 2129–2146).

Bernstein, D. J. & Lange, T. (2017). Post-quantum cryptography. *Nature*, *549*(7671), 188–194.

Beullens, W. (2020). Improved cryptanalysis of uov and rainbow. Cryptology ePrint Archive, Report 2020/1343. `https://eprint.iacr.org/2020/1343`.

Boneh, D. et al. (1999). Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, *46*(2), 203–213.

Boneh, D., Goh, E.-J., & Nissim, K. (2005). Evaluating 2-dnf formulas on ciphertexts. In Kilian, J. (Ed.), *Theory of Cryptography*, (pp. 325–341)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M.,

Schwabe, P., Seiler, G., & Stehlé, D. (2018). Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE EuroS&P*, (pp. 353–367). IEEE.

Bos, J. W., Lauter, K., Loftus, J., & Naehrig, M. (2013). Improved security for a ring-based fully homomorphic encryption scheme. In *IMA International Conference on Cryptography and Coding*, (pp. 45–64). Springer.

Botros, L., Kannwischer, M. J., & Schwabe, P. (2019). Memory-efficient high-speed implementation of kyber on cortex-m4. In *Int. Conference on Cryptology in Africa*, (pp. 209–228). Springer.

Brakerski, Z. (2012). Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, (pp. 868–886). Springer.

Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2014). (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, *6*(3), 1–36.

Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., & Storaasli, O. O. (2010). State-of-the-art in heterogeneous computing. *Sci. Program.*, *18*(1), 1–33.

Brutzkus, A., Gilad-Bachrach, R., & Elisha, O. (2019). Low latency privacy preserving inference. In Chaudhuri, K. & Salakhutdinov, R. (Eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, (pp. 812–821). PMLR.

Cathébras, J., Carbon, A., Milder, P., Sirdey, R., & Ventroux, N. (2018). Data flow oriented hardware design of rns-based polynomial multiplication for she acceleration. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2018*(3), 69–88.

Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., & Zaverucha, G. (2017). Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, (pp. 1825–1842).

Chen, C., Danba, O., Hoffstein, J., Hülsing, A., Rijneveld, J., Schanck, J. M., Schwabe, P., Whyte, W., & Zhang, Z. (2019). Ntru: Algorithm specifications and supporting documentation. https://ntru.org/f/ntru-20190330.pdf.

Chen, D. D., Mentens, N., Vercauteren, F., Sinha Roy, S., Cheung, R. C. C., Pao, D., & Verbauwhede, I. (2015). High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *62*(1), 157–166.

Chen, H., Dai, W., Kim, M., & Song, Y. (2019). Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. Cryptology ePrint Archive, Report 2019/524. `https://eprint.iacr.org/2019/524`.

Chen, H., Dai, W., Kim, M., & Song, Y. (2021). Efficient homomorphic conversion between (ring) lwe ciphertexts. In Sako, K. & Tippenhauer, N. O. (Eds.), *Applied Cryptography and Network Security*, (pp. 460–479)., Cham. Springer International Publishing.

Chen, L., Chen, L., Jordan, S., Liu, Y.-K., Moody, D., Peralta, R., Perlner, R., & Smith-Tone, D. (2016). Report on post-quantum cryptography. US Department of Commerce, National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf.

Chen, Z., Ma, Y., Chen, T., Lin, J., & Jing, J. (2020). Towards efficient kyber on fpgas: A processor for vector of polynomials. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, (pp. 247–252).

Chen, Z., Ma, Y., Chen, T., Lin, J., & Jing, J. (2021). High-performance area-efficient polynomial ring processor for crystals-kyber on fpgas. *Integration*, *78*, 25–35.

Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, (pp. 409–437). Springer.

Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2020). Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, *33*(1), 34–91.

Chu, E. & George, A. (1999). *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press.

Chung, C.-M. M., Hwang, V., Kannwischer, M. J., Seiler, G., Shih, C.-J., & Yang, B.-Y. (2021). Ntt multiplication for ntt-unfriendly rings: New speed records for saber and ntru on cortex-m4 and avx2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2021*(2), 159–188.

Cooley, J. W. & Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, *19*(90), 297–301.

Dadda, L. (1965). Some Schemes for Parallel Multipliers. In *Alta Frequenza*, volume 34, (pp. 349–356).

Dai, W. & Sunar, B. (2015). cuhe: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, (pp. 169–186). Springer.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., & Stehlé, D. (2018). Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2018*(1), 238–268.

D'Anvers, J.-P., Karmakar, A., Sinha Roy, S., & Vercauteren, F. (2018). Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *International Conference on Cryptology in Africa*, (pp. 282–305). Springer.

Fan, J. & Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, *2012*, 144.

Feng, X., Li, S., & Xu, S. (2019). Rlwe-oriented high-speed polynomial multiplier utilizing multi-lane stockham ntt algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *67*(3), 556–559.

Fouque, P.-A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., & Zhang, Z. (2018). Falcon: Fast-fourier lattice-based compact signatures over ntru.

Fritzmann, T. & Sepúlveda, J. (2019). Efficient and flexible low-power ntt for lattice-based cryptography. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, (pp. 141–150).

Fritzmann, T., Sigl, G., & Sepúlveda, J. (2020). RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2020*(4), 239–280.

Gentry, C. (2009). *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford

University, Stanford, CA, USA. AAI3382729.

Gentry, C., Sahai, A., & Waters, B. (2013). Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, (pp. 75–92). Springer.

Göttert, N., Feller, T., Schneider, M., Buchmann, J., & Huss, S. (2012). On the design of hardware building blocks for modern lattice-based encryption schemes. In Prouff, E. & Schaumont, P. (Eds.), *Cryptographic Hardware and Embedded Systems – CHES 2012*, (pp. 512–529)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Gouzien, E. & Sangouard, N. (2021). Factoring 2048 rsa integers in 177 days with 13436 qubits and a multimode memory.

Halevi, S., Polyakov, Y., & Shoup, V. (2019). An improved rns variant of the bfv homomorphic encryption scheme. In Matsui, M. (Ed.), *Topics in Cryptology – CT-RSA 2019*, (pp. 83–105)., Cham. Springer International Publishing.

Halevi, S. & Shoup, V. (2014). Algorithms in helib. In *Annual Cryptology Conference*, (pp. 554–571). Springer.

Huang, Y., Huang, M., Lei, Z., & Wu, J. (2020). A Pure Hardware Implementation of CRYSTALS-KYBER PQC Algorithm through Resource Reuse. *IEICE Electronics Express*, *advpub*.

ITU (2020). Measuring digital development: Facts and figures 2020. https://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx.

Jacobsen, M., Freund, Y., & Kastner, R. (2012). RIFFA: A Reusable Integration Framework for FPGA Accelerators. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, (pp. 216–219).

Kalali, E., Mert, A. C., & Hamzaoglu, I. (2016). A computation and energy reduction technique for hevc discrete cosine transform. *IEEE Transactions on Consumer Electronics*, *62*(2), 166–174.

Kannepalli, S., Laine, K., & Moreno, R. C. (2021). Password monitor: Safeguarding passwords in microsoft edge.

Kannwischer, M. J., Rijneveld, J., Schwabe, P., & Stoffelen, K. (2019). pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Report 2019/844. `https://eprint.iacr.org/2019/844`.

Karatsuba, A. A. & Ofman, Y. P. (1962). Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk*, *145*(2), 293–294.

Kawamura, K., Yanagisawa, M., & Togawa, N. (2018). A loop structure optimization targeting high-level synthesis of fast number theoretic transform. In *2018 19th ISQED*, (pp. 106–111).

Keutzer, K., Newton, A. R., Rabaey, J. M., & Sangiovanni-Vincentelli, A. (2000). System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *19*(12), 1523–1543.

Kurfalı, M., Arifoğlu, A., Tokdemir, G., & Paçin, Y. (2017). Adoption of e-government services in turkey. *Computers in Human Behavior*, *66*, 168–178.

Liu, W., Fan, S., Khalid, A., Rafferty, C., & O'Neill, M. (2019). Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *27*(10), 2459–2463.

Longa, P. & Naehrig, M. (Nov. 2016). Speeding up the number theoretic trans-

form for faster ideal lattice-based cryptography. In *Cryptology and Network Security*, (pp. 124–139)., Milan, Italy.

Lyubashevsky, V. & Seiler, G. (2019). Nttru: Truly fast ntru using ntt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2019*(3), 180–201.

Mert, A. C., Karabulut, E., Öztürk, E., Savaş, E., & Aysu, A. (2020). An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*.

Mert, A. C., Karabulut, E., Öztürk, E., Savaş, E., Becchi, M., & Aysu, A. (2020). A flexible and scalable ntt hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, (pp. 346–351).

Mert, A. C., Öztürk, E., & Savaş, E. (2019). Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, (pp. 253–260).

Mert, A. C., Öztürk, E., & Savaş, E. (2020). Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *28*(2), 353–362.

Mert, A. C., Öztürk, E., & Savaş, E. (2020). Fpga implementation of a run-time configurable ntt-based polynomial multiplication hardware. *Microprocessors and Microsystems*, *78*, 103219.

Micciancio, D. (2011). The geometry of lattice cryptography. In *International School on Foundations of Security Analysis and Design*, (pp. 185–210). Springer.

Microsoft (2019). Microsoft SEAL (release 3.2). `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA.

Microsoft (2020). Microsoft SEAL (release 3.5). `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA.

Migliore, V., Real, M. M., Lapotre, V., Tisserand, A., Fontaine, C., & Gogniat, G. (2018). Hardware/software co-design of an accelerator for fv homomorphic encryption scheme using karatsuba algorithm. *IEEE Transactions on Computers*, *67*(3), 335–347.

Milder, P., Franchetti, F., Hoe, J. C., & Püschel, M. (2012). Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. Des. Autom. Electron. Syst.*, *17*(2).

Millar, K. (2019). Design of a flexible schoenhage-strassen fft polynomial multiplier with high-level synthesis. Master's thesis, Rochester Institute of Technology.

Montgomery, P. L. (1985). Modular multiplication without trial division. *Mathematics of computation*, *44*(170), 519–521.

Nejatollahi, H., Dutt, N., Ray, S., Regazzoni, F., Banerjee, I., & Cammarota, R. (2019). Post-quantum lattice-based cryptography implementations: A survey. *ACM Comput. Surv.*, *51*(6).

Ozcan, E. & Aysu, A. (2020). High-level synthesis of number-theoretic transform: A case study for future cryptosystems. *IEEE Embedded Systems Letters*, *12*(4), 133–136.

Ozerk, O., Elgezen, C., Mert, A. C., Öztürk, E., & Savas, E. (2021). Efficient number theoretic transform implementation on gpu for homomorphic encryption. *IACR Cryptol. ePrint Arch.*, *2021*, 124.

Pease, M. C. (1968). An adaptation of the fast fourier transform for parallel pro-

cessing. *J. ACM*, *15*(2), 252–264.

Pollard, J. M. (1971). The fast fourier transform in a finite field. *Mathematics of computation*, *25*(114), 365–374.

Polyakov, Y., Rohloff, K., & Ryan, G. W. (2017). Palisade lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep*, *15*.

Pöppelmann, T. & Güneysu, T. (2012). Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In Hevia, A. & Neven, G. (Eds.), *Progress in Cryptology – LATINCRYPT 2012*, (pp. 139–158)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Pöppelmann, T. & Güneysu, T. (2013). Towards practical lattice-based public-key encryption on reconfigurable hardware. In *Int. Conf. on Selected Areas in Cryptography*, (pp. 68–85). Springer.

Pöppelmann, T. & Güneysu, T. (2014). Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *2014 IEEE Int. Symp. on Circuits and Systems*, (pp. 2796–2799).

Pöppelmann, T., Naehrig, M., Putnam, A., & Macias, A. (Sep. 2015). Accelerating homomorphic evaluation on reconfigurable hardware. In *CHES*, (pp. 143–163)., Saint-Malo, France.

Pöppelmann, T., Oder, T., & Güneysu, T. (2015). High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, (pp. 346–365). Springer.

Proos, J. & Zalka, C. (2003). Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Info. Comput.*, *3*(4), 317–344.

Pulido-Gaytan, B., Tchernykh, A., Cortés-Mendoza, J. M., Babenko, M., Radchenko, G., Avetisyan, A., & Drozdov, A. Y. (2021). Privacy-preserving neural networks with homomorphic encryption: Challenges and opportunities. *Peer-to-Peer Networking and Applications*, *14*(3), 1666–1691.

Regev, O. (2009). On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, *56*(6), 1–40.

Reis, D., Takeshita, J., Jung, T., Niemier, M., & Hu, X. S. (2020). Computing-in-memory for performance and energy-efficient homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *28*(11), 2300–2313.

Reparaz, O., Sinha Roy, S., de Clercq, R., Vercauteren, F., & Verbauwhede, I. (2016). Masking ring-lwe. *Journal of Cryptographic Engineering*, *6*(2), 139–153.

Rivest, R. L., Adleman, L., Dertouzos, M. L., et al. (1978). On data banks and privacy homomorphisms. *Foundations of secure computation*, *4*(11), 169–180.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120–126.

Sarker, A., Mozaffari-Kermani, M., & Azarderakhsh, R. (2019). Hardware constructions for error detection of number-theoretic transform utilized in secure cryptographic architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *27*(3), 738–741.

Seiler, G. (2018). Faster avx2 optimized ntt multiplication for ring-lwe lattice cryp-

tography. *IACR Cryptology ePrint Archive*, *2018*, 39.

Shenoy, A. & Kumaresan, R. (1989). Fast base extension using a redundant modulus in rns. *IEEE Transactions on Computers*, *38*(2), 292–297.

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, (pp. 124–134). Ieee.

Sinha Roy, S., Järvinen, K., Vliegen, J., Vercauteren, F., & Verbauwhede, I. (2018). Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation. *IEEE Transactions on Computers*, *67*(11), 1637–1650.

Sinha Roy, S., Turan, F., Jarvinen, K., Vercauteren, F., & Verbauwhede, I. (2019). Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (pp. 387–398).

Sinha Roy, S., Vercauteren, F., Mentens, N., Chen, D. D., & Verbauwhede, I. (2014). Compact ring-lwe cryptoprocessor. In Batina, L. & Robshaw, M. (Eds.), *Cryptographic Hardware and Embedded Systems – CHES 2014*, (pp. 371–391)., Berlin, Heidelberg. Springer Berlin Heidelberg.

Song, S., Tang, W., Chen, T., & Zhang, Z. (2018). Leia: A 2.05mm2140mw lattice encryption instruction accelerator in 40nm cmos. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, (pp. 1–4).

Takeshita, J., Reis, D., Gong, T., Niemier, M., Hu, X. S., & Jung, T. (2020). Algorithmic acceleration of b/fv-like somewhat homomorphic encryption for compute-enabled ram. Cryptology ePrint Archive, Report 2020/1223. `https://eprint.iacr.org/2020/1223`.

Takeshita, J., Schoenbauer, M., Karl, R., & Jung, T. (2020). Enabling faster operations for deeper circuits in full rns variants of fv-like somewhat homomorphic encryption. *IACR Cryptology ePrint Archive*, *2020*, 91.

Toom, A. L. (1963). The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, *3*(4), 714–716.

Turan, F., Roy, S. S., & Verbauwhede, I. (2020). Heaws: An accelerator for homomorphic encryption on the amazon aws fpga. *IEEE Transactions on Computers*, *69*(8), 1185–1196.

Van Dijk, M., Gentry, C., Halevi, S., & Vaikuntanathan, V. (2010). Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, (pp. 24–43). Springer.

Verbauwhede, I. & Schaumont, P. (2005). Skiing the embedded systems mountain. *ACM Trans. Embed. Comput. Syst.*, *4*(3), 529–548.

Viand, A., Jattke, P., & Hithnawi, A. (2021). Sok: Fully homomorphic encryption compilers. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, (pp. 1166–1182)., Los Alamitos, CA, USA. IEEE Computer Society.

Winkler, F. (2012). *Polynomial algorithms in computer algebra*. Springer Science & Business Media.

Xilinx (2014). Vivado design suite user guide-high-level synthesis.

Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., & Zeng, X. (2020). VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. *IEEE Trans. on Circuits and Systems I: Regular Papers*, *67*(8), 2672–2684.

Xing, Y. & Li, S. (2020). An efficient implementation of the newhope key exchange

on fpgas. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *67*(3), 866–878.

Xing, Y. & Li, S. (2021). A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2021*(2), 328–356.

Yaman, F., Mert, A. C., Öztürk, E., & Savaş, E. (2021). A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme. Cryptology ePrint Archive, Report 2021/485. `https://eprint.iacr.org/2021/485`.

Yanik, T., Savas, E., & Koc, C. K. (2002). Incomplete reduction in modular arithmetic. *IEE Proceedings - Computers and Digital Techniques*, *149*(2), 46–52.

Zhang, N., Qin, Q., Yuan, H., Zhou, C., Yin, S., Wei, S., & Liu, L. (2020). Nttu: An area-efficient low-power ntt-uncoupled architecture for ntt-based multiplication. *IEEE Transactions on Computers*, *69*(4), 520–533.

Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., & Liu, L. (2020). Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2020*(2), 49–72.

Öztürk, E., Doroz, Y., Savaş, E., & Sunar, B. (2017). A custom accelerator for homomorphic encryption applications. *IEEE Transactions on Computers*, *66*(1), 3–16.