12-2022

High-Performance VLSI Architectures for Lattice-Based Cryptography

Weihang Tan
wtan@g.clemson.edu

High-Performance VLSI Architectures for Lattice-Based Cryptography

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Electrical Engineering

---

by
Weihang Tan
December 2022

Accepted by:
Dr. Yingjie Lao, Committee Chair
Dr. Jon C. Calhoun
Dr. Shuhong Gao
Dr. Rajendra Singh

Abstract

Lattice-based cryptography is a cryptographic primitive built upon the hard problems on point lattices. Cryptosystems relying on lattice-based cryptography have attracted huge attention in the last decade since they have post-quantum-resistant security and the remarkable construction of the algorithm. In particular, homomorphic encryption (HE) and post-quantum cryptography (PQC) are the two main applications of lattice-based cryptography. Meanwhile, the efficient hardware implementations for these advanced cryptography schemes are demanding to achieve a high-performance implementation.

This dissertation aims to investigate the novel and high-performance very large-scale integration (VLSI) architectures for lattice-based cryptography, including the HE and PQC schemes. This dissertation first presents different architectures for the number-theoretic transform (NTT)-based polynomial multiplication, one of the crucial parts of the fundamental arithmetic for lattice-based HE and PQC schemes. Then a high-speed modular integer multiplier is proposed, particularly for lattice-based cryptography. In addition, a novel modular polynomial multiplier is presented to exploit the fast finite impulse response (FIR) filter architecture to reduce the computational complexity of the schoolbook modular polynomial multiplication for lattice-based PQC scheme. Afterward, an NTT and Chinese remainder theorem (CRT)-based high-speed modular polynomial multiplier is presented for HE schemes whose moduli are large integers.

Acknowledgments

I would like to thank my advisor, Dr. Yingjie Lao, for his guidance, recognition, and encouragement throughout my Ph.D. study at Clemson University. I will always treasure the opportunity to be under his supervision. Through his encouragement, I have learned to better myself. Since I was an undergraduate student, Dr. Lao has been getting me a start in hardware security/design research, teaching me everything in patience, and providing tremendous help in my research and study. He is always there to support and motivate me whenever I feel lost and get stuck on any problems. We had many great discussions and brainstormed ideas while working on the research project. As his student, I have developed expertise in different research areas and learned a lot of important knowledge on research. It is only with his support that I have been able to fulfill my dream of obtaining a Ph.D. degree.

I would like to express my appreciation to Dr. Jon C. Calhoun, Dr. Shuhong Gao, and Dr. Rajendra Singh for their valuable suggestions on my research as my Ph.D. dissertation committee members. They have taught me a lot in the field of integrated circuits, cryptography, and computer engineering.

My sincere thanks to Dr. Keshab Parhi at the University of Minnesota and Dr. Xinmiao Zhang at The Ohio State University for their mentoring and advice. Their insightful discussions and suggestions help me produce several good quality research works. I especially thank Dr. Parhi, whom I have learned so much from. He has always been willing to help me improve my research skills and discuss research ideas with me.

I want to thank my past and current research group members: Joseph Clements, Azadeh Famili, Urmil Joshi, Quinton Kinzie, Ling Qiu, Ankur Sharma, Dr. Jianchi Sun, Antian Wang, Xiaojia Wang, Yuejiang Wen, Yunhao Xu (Southeast University, China), and Bingyin Zhao, for their assistance, support, and feedback to my research and study. I also extend my thanks to my

collaborators: Dr. Benjamin Case, Dr. Gengran Hu (Hangzhou Dianzi University, China), Antian Wang, and Sin-Wei Chiu (University of Minnesota). This dissertation could not have been finished without their help and contributions, especially thanks to Antian Wang for his huge efforts on our papers. I also want to thank the instructors and staff at Clemson University, who helped and taught me a lot during my study.

I also would like to thank my friends in my past few years of study at Clemson University, particularly Chenyan Chang, Xiang Lan, Quan Li, Dr. Yuqing Hang, Dr. Zhenyu Zhou, Kewei Li, Jianxin Gao, Siyuan Yu, Dr. Xiaonan Zhang, Sihan Yu, Yu Xuan, and Dr. Xiurui Zhao, for their help and support in my Ph.D. life. I will never forget the time we spent together; they will always be like family to me.

Lastly, I am truly grateful to my beloved parents for their endless support. They have always encouraged me to do what I love. While I could not physically accompany them during the global pandemic, they have always stood by me through the good times and hard.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet is a system interconnecting computer networks worldwide. There were 307.2 million internet users in the United States in January 2022, which shows the internet penetration rate stood at 92.0% of the total population at the start of 2022 [1]. As the advantages of the Internet, people are more and more dependent on it in their daily lives. However, the interaction between users and the internet servers makes the users' data more vulnerable to leak to threats and unauthorized parties. Therefore, safeguarding users' identities and sensitive data is one of the important research topics. In order to avoid information leakage in the network communication between the users and internet servers, a secure communication system is needed.

A secure communication system is required for data transmission in some untrusted platforms. As shown in Fig. 1.1, the cryptography scheme plays a crucial role during secure communication to prevent the eavesdropper from learning the knowledge in the communication channel. In general, four goals are desired to protect information security using cryptographic schemes [2]. The *confidentiality* of a cryptosystem is to prevent the information from leaking to the unauthorized party, which is also called the security of the information. *Data integrity* ensures that the unauthorized party does not manipulate the content of the information. *Authentication* is to guarantee that two parties have to identify with each other before the communication. *Non-repudiation* is to prevent any party from denying the previous actions and sent information. During the encryption and decryption processes, a cryptographic scheme is needed, where private-key and public-key cryptography are two of the most important topics in modern cryptography. Private key encryption uses the same key to convert the plaintext (information) to the ciphertext (encrypted data), which

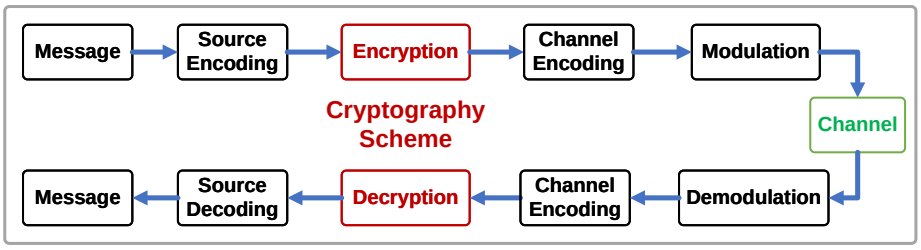

Figure 1.1: Secure communication system block diagram [3].

requires two parties to hold the same key for encryption and decryption. Such a setting requires two parties to agree on and share a private key in advance. In contrast, public key encryption holds two different keys during secure communication. The keys are generated by the receiver, called the private and public keys. The public key is shared with the sender, so the senders can encrypt the information using the public key. Only the private key holder (receiver) can use the private key to decrypt the ciphertext and receive the information. Public key encryption only requires the receiver to keep a private secret, but the public key can be shared with any party in the public channel.

Various public key encryption schemes such as RSA and elliptic-curve are currently employed for secure communication. Though the above public-key cryptosystems showed excellent performance on security previously, they are no longer secure in the world of quantum computing. It has been shown that quantum computers can efficiently break these cryptosystems relying on the computational difficulty in factoring large integers. As a result, it is crucial to develop new cryptosystems that can withstand attacks under quantum computers and classic computers, known as post-quantum cryptography (PQC) [4].

The PQC schemes, building on the lattice-based cryptographic primitive, are the most popular cryptosystems during the National Institute of Standards and Technology (NIST) PQC standardization process since 2017 [4].

In addition, lattice-based cryptographic primitive has also been utilized in an important cryptographic method for secure processing, called homomorphic encryption (HE) [5]. HE allows the processing of the data directly on the ciphertexts without decryption. HE is a revolutionary breakthrough for privacy-preserving applications. It differs from private-key or public-key encryption schemes that cannot prevent information leakage on the cloud since the data must be decrypted to plaintext before the computation.

These powerful schemes provide the preliminary that shows superior performance compared

to the conventional cryptosystems in many perspectives. However, the software implementation is still inefficient because of the high computational complexity of the lattice-based schemes. For example, the computations inside lattice-based cryptography involve arithmetic over the polynomial ring, which also requires a large number of polynomial multiplication, and additions over the ring. Due to the long polynomial degree and long word-length of the coefficient modulus, such computations are expensive. Thus a hardware acceleration for the lattice-based PQC and HE schemes is desired.

This dissertation considers high-performance hardware implementations for lattice-based cryptography. In particular, this dissertation proposes different hardware accelerators to fulfill diverse lattice-based cryptography applications. Modular polynomial multiplier design is one of the targets of this dissertation, where number theoretic transform (NTT)-based modular polynomial multiplication is first presented in this dissertation to improve the performance of the HE and PQC schemes. Besides, the novel schoolbook-based modular polynomial multiplier is considered for the schemes whose parameters or mathematical primitives do not allow using the NTT for modular polynomial multiplication. Finally, the other functional blocks customized for the specific schemes are also presented in this dissertation.

- The first work in Chapter 3 presents a hardware accelerator for the external product step of the recent FHE schemes [6, 7]. In this work, we mainly study an ultra-high parallel number theoretic transform (NTT)-based modular polynomial multiplier. This hardware accelerator leverages a memory-based design framework to achieve a scalable and reconfigurable architecture. The result of this work is published in [8, 9].

- The second work in Chapter 4 introduces another NTT architecture along with an efficient Barrett reduction-based modular (integer) multiplier. This architecture targets not only the acceleration for the FHE schemes but also the lattice-based PQC schemes. The result of this work is published in [10].

- Chapter 5 introduces the third work that targets the hardware and software co-optimization for the modular (integer) multiplication for lattice-based cryptography. This work utilizes the Karatsuba-based algorithm and explores the special prime to accelerate modular multiplication. The result of this work is published in [11].

- The fourth work investigates the schoolbook modular polynomial multiplier by exploiting the fast finite impulse response (FIR) filter-like and scalable architecture, which is presented

in Chapter 6. Note that this work targets the schemes whose parameters or mathematical primitives do not allow using the NTT for modular polynomial multiplication. Furthermore, this work utilizes the Saber PQC scheme as a case study. The result of this work is presented in [12].

- In Chapter 7, we present a high-speed long polynomial multiplication architecture, PaReNTT, for the HE schemes. This architecture first investigates a feed-forward NTT-based polynomial multiplier with low clock cycle consumption by using a novel folding transformation technique. Moreover, since the moduli used in the HE schemes are large, decomposing the moduli based on the Chinese remainder theorem (CRT) and executing the operations in parallel allows the modular polynomial multiplier has a better timing performance. Therefore, efficient architectures for the pre-processing and post-processing in the CRT are also studied in this chapter.

The outline of this dissertation is as follows. This preliminary and mathematical background of this dissertation is reviewed in Chapter 2. Afterward, we proposed our novel architectures and designs in Chapters 3, 4, 5, 6, and 7. Finally, Chapter 8 concludes the contributions of this dissertation and future work.

Chapter 2

Background

In this chapter, the mathematical background closely related to this dissertation is introduced, which is helpful for understanding the proposed hardware designs. This chapter first reviews modular arithmetic, which is a foundation for cryptography. Then, the preliminary for the lattice-based cryptography is also reviewed.

2.1 Modular Arithmetic

Modular arithmetic is system arithmetic that requires the results to fall in the range $[0, q-1]$, where q is called modulus.

2.1.1 Congruence

The modular arithmetic is based on the congruence relation, defined as follows.

Given an integer $q \neq 0$. If $x - y$ is divisible by q (i.e., $x - y = kq$ for the integer k), then q is the modulus and y is the remainder of integer x with respect to modulus q:

$$x \equiv y \pmod{q}. \tag{2.1}$$

It can also be denoted as $y = [x]_q$.

Specifically, all the integers congruent to x modulo q is called the congruence class (residue)

of x modulo q:

$$[x]_q = \{z \in \mathbb{Z} \quad | \quad x - z = kq \text{ for } k \in \mathbb{Z}\}. \tag{2.2}$$

2.1.2 Modular Arithmetic Operations

Modular addition, modular subtraction, and multiplication are three essential operations for modular arithmetic. In particular, modular addition and modular subtraction are relatively simple, and the algorithms are shown in Algorithms 1 and 2. Such algorithms are straightforward to implement in software and hardware. However, the modular multiplication is non-trivial, one of the targeted blocks to be optimized in the prior works. The high-level expression for modular multiplication is illustrated as

$$c = a \cdot b \mod q \tag{2.3}$$

$$= a \cdot b - k \cdot q, \tag{2.4}$$

where $k \in \mathbb{Z}$.

Besides, the division operation of a and b in modular arithmetic is based on the modular multiplication of the reciprocal of the divisor (i.e., the multiplicative inverse):

$$c = a \cdot b^{-1}, \tag{2.5}$$

where b^{-1} can be calculated by Fermat's little theorem [13] if q is a prime number:

$$b^{-1} \equiv b^{q-2} \mod q, \quad b \nmid q. \tag{2.6}$$

Algorithm 1 Modular Addition

Input: a and $b \in [0, q-1]$
Output: $c = a + b \mod q$,

1: $c = a + b$
2: **if** $c >= q$ **then**
3: $\quad c = c - q$
4: **end if**
5: **return** c

Algorithm 2 Modular Subtraction

Input: a and $b \in [0, q-1]$
Output: $c = a - b \mod q$,

1: $c = a - b$
2: **if** $c < 0$ **then**
3: $\quad c = c + q$
4: **end if**
5: **return** c

2.2 Lattice-based Cryptography

Lattice-based cryptography represents the cryptographic schemes that are based on the hardness of the lattice problem. The learning with errors (LWE) problem is a computational problem built upon the lattice problem, which is NP-hard for quantum computers [14, 15]. Due to its arithmetic simplicity and strong security performance, cryptosystems related to the LWE problem are widely studied for post-quantum secure applications.

Ring-LWE (RLWE) is a ring-based analogue for the LWE problem [15]. The RLWE problem has been adopted in the recent popular HE schemes [16–18]. For an RLWE-based scheme, the polynomial computation is computed over ring $R_q = \mathbb{Z}_q/(x^n + 1)$, where $x^n + 1$ is an irreducible polynomial of degree (dimension) n, and the polynomial coefficient modulus is q. The RLWE sample $(a(x), b(x)) \in R_q \times R_q$ is defined as follows: $a(x)$ is an uniformly random polynomial over ring R_q, and the corresponding $b(x)$ is expressed as

$$b(x) = a(x) \cdot s(x) + e(x) \in R_q, \tag{2.7}$$

where $s(x) \in R_q$ is the secret, and $e(x) \in R_q$ is the error term. These two polynomials are random polynomials sampled from a discrete Gaussian distribution with standard deviation σ [19]. The key generation, encryption, and decryption steps of the RLWE problem can be defined based on the above setting.

The key generation step is used to obtain the public key $(a(x), p(x))$ and private key $r_2(x)$, where $a(x) \in R_q$ is taken from the uniform distribution. Two polynomials $r_1(x), r_2(x) \in R_q$ are firstly sampled from the discrete Gaussian distribution, and then compute $p(x)$ by

$$p(x) = r_1(x) - a(x) \cdot r_2(x) \in R_q. \tag{2.8}$$

During the encryption step, a pair of ciphertexts $(c_1(x), c_2(x))$ is calculated based on the public key. In particular, $c_1(x)$ as well as $c_2(x)$ are expressed as

$$\begin{aligned} c_1(x) &= a(x) \cdot e_1(x) + e_2(x) \in R_q, \\ c_2(x) &= p(x) \cdot e_1(x) + e_3(x) + \hat{m}(x) \in R_q, \end{aligned} \tag{2.9}$$

where $\hat{m}(x)$ is encoded from message $m(x)$, and three error terms $e_1(x), e_2(x), e_3(x) \in R_q$ are also sampled from the discrete Gaussian distribution.

For decryption step, polynomials $m'(x) = c_1(x) \cdot r_2(x) + c_2(x) \in R_q$ is computed by using the private key $r_2(x)$, which will then be decoded to $m(x)$.

Besides, the Module-LWE (MLWE) is another important computational problem used in CRYSTALS-KYBER scheme [20], a candidate has chosen to be standardized by NIST. The construction of RLWE and MLWE-based variants are similar, while the only difference is $s(x)$ and $a(x)$ are vectors, and their entries are polynomials in R_q (i.e., $s(x), a(x) \in (R_q)^d$), where d is the dimension of the vector.

Chapter 3

An Ultra-Highly Parallel Polynomial Multiplier for the Bootstrapping Algorithm in A Fully Homomorphic Encryption Scheme

Fully homomorphic encryption (FHE) is a post-quantum secure cryptographic technology that enables privacy-preserving computing on an untrusted platform without divulging any secret or sensitive information. The core of FHE is the bootstrapping algorithm, which is the intermediate refreshing procedure of a processed ciphertext. However, this step has been the computational bottleneck that prevents real-world deployments among various FHE schemes. This chapter, to the best of our knowledge, for the first time, presents a scalable and ultra-highly parallel design for the number theoretic transform (NTT)-based polynomial multiplier with a variable number of reconfigurable processing elements (PEs). Hence, the highest degree of acceleration can be achieved for any targeted hardware platform by implementing as many PEs as possible under resource constraints. The corresponding addressing and scheduling schemes are also proposed to avoid memory access

conflict for the PEs, which yields an extremely high utilization ratio of 99.18% on average. In addition, the latency of the proposed design with the general negative wrapped convolution algorithm is reduced by 59.20% compared to prior works. [1]

3.1 Introduction

The recent development of cloud computing and the Internet of Things (IoT) demands an efficient cryptosystem to protect sensitive data through public networks and computing platforms. To this end, fully homomorphic encryption (FHE) has emerged as a promising secure function evaluation (SFE) scheme that allows operations on encrypted data without decryption. As opposed to conventional encryption methods, data can remain encrypted under the user's secret key, and hence only the secret key owner has access to the plaintext. In addition, different from partially or somewhat homomorphic encryption which only permits a limited type or a limited number of operations, FHE allows the function to be directly evaluated using the encrypted data for an unlimited number of operations.

However, FHE schemes are still computationally expensive, especially for the bootstrapping step, which is the intermediate refreshing procedure of a processed ciphertext. Hence, one approach to improve the efficiency is to bypass the bootstrapping but only allow a limited number of homomorphic operations (i.e., implement somewhat homomorphic encryption instead) [16,21]. Alternatively, many works in past years aimed at reducing the complexity of homomorphic computation, including the bootstrapping or recryption steps. Some recent breakthroughs have significantly reduced the computation time of bootstrapping by utilizing GSW-based scheme [17,22] and some novel homomorphic embedding [16,23,24]. However, their schemes still have large cipher expansions (i.e., the ratio between the ciphertext and the plaintext), according to [22]. Meanwhile, our recent works successfully reduce the cipher expansion to 6 with private-key encryption and 20 with public-key encryption in [6], which are further optimized to 2.5 with private-key encryption and 6.5 with public-key encryption for a 4-bit plaintext space in [7].

In this chapter, we present a scalable and ultra-highly parallel design with a variable number of processing elements (PEs) for the most computational-intensive step in this efficient FHE scheme, i.e., external product in the bootstrapping, whose fundamental operation is polynomial multiplica-

[1] This work is presented in [8,9].

tion over the ring. This chapter is extended from our prior work in [8]. The proposed PE can be reconfigured as either a butterfly processor for computing the NTT/iNTT or a modular multiplier. Additionally, the reconfigurable PE eliminates the need for other individual computational units, which helps increase the parallelism of the computation for eventually reducing the latency.

The main contributions of this chapter are summarized as follows:

- We introduce an ultra-highly parallel and scalable design by extending our prior work [8] to a *variable* number of reconfigurable PEs without the power-of-two constraint. The optimal setup between the number of PEs and out-of-chip memory banks is also proposed.
- The corresponding novel conflict-free memory management and the scheduling schemes help the proposed architecture to achieve a nearly full utilization ratio. Specifically, a special read-and-write pattern for the memory addressing organizes the data-flow in an optimal order across the PEs.
- As opposed to most prior works on hardware implementations of homomorphic encryption that usually bypass the expensive bootstrapping step (i.e., somewhat homomorphic encryption) [25–29], our proposed design for the bootstrapping algorithm in [6,7,22] could ensure homomorphic operations can be performed in an arbitrary depth.
- The proposed reconfigurable architecture and scheduling schemes can be easily extended to other FHE schemes that involve polynomial multiplications, facilitating more efficient architectures for future deployment of FHE.

The rest of this chapter is organized as follows: Section 3.2 reviews the mathematical background and the existing hardware implementations of homomorphic encryption. Section 3.3 introduces the details of our proposed hardware architecture design and conflict-free memory management scheme. The performance of our proposed architecture is presented and analyzed in Section 3.4. Finally, Section 3.5 presents the conclusion and discusses future work.

3.2 Background

Table 3.1 summarizes essential notations that are used in this chapter.

A high-level diagram of homomorphic encryption in the cloud computing scenario is illustrated in Fig. 3.1. The users will first generate two keys: private key sk and public key pk. Specifically, pk is used for encrypting, bootstrapping, or recryption the data, which can be accessed

Table 3.1: Notation used in this chapter

Notation	Explanation
N	Degree of polynomial
Q	Modulus of coefficient in the polynomial
S	Current global stage of polynomial multiplication
k	Current stage of NTT/iNTT
L	Number of PE(s)
M	Number of memory banks
R	Number of words in each memory bank
P	Number of steps in each type of operation
v	Index of memory bank
R_b	Index in binary of the word in memory bank

by anyone and stored in any untrusted platform. In contrast, sk should be securely kept by the users. Operations on the cloud are only performed on the encrypted data. As a result, the cloud server can only access the ciphertexts C_i, public key pk, and the homomorphic computation function $\hat{h}(\cdot)$. Finally, the ciphertext-form results will be returned to the users, who can then decrypt with the private key sk to obtain the result y. The result should be the same as the operation directly performed in the plaintext domain using the function $h(\cdot)$.

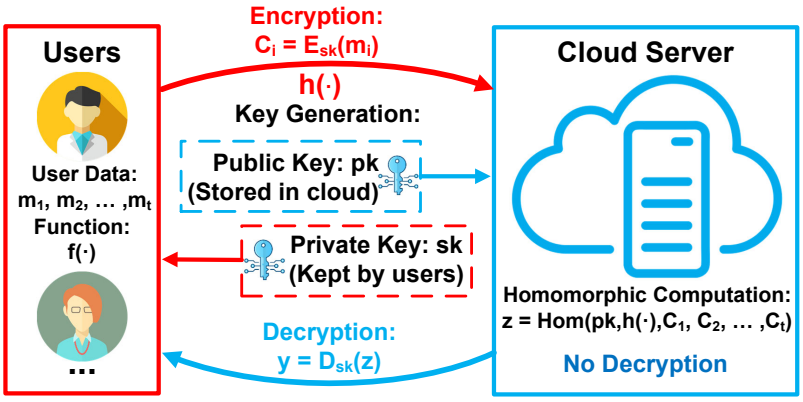

Figure 3.1: FHE-based computation on the cloud.

Currently, the models of existing homomorphic encryption schemes can be broadly classified into integer-based [16, 21] and Boolean logic-based [6, 7, 22]. On the one hand, the plaintext of the integer-based schemes is represented by an integer. Their computations in the ciphertext domain are based on homomorphic addition, multiplication, and bit shifting. This model of schemes usually requires a large prime in order to be capable of limited (leveled) multiplication. On the other hand, the plaintext of Boolean logic-based is bit representation, whose basic components on the

homomorphic computations are the encrypted ciphertext-forms of logic AND, OR, and XOR. Fig. 3.2 shows the relationship between the plaintext domain and the ciphertext domain of the Boolean logic-based model.

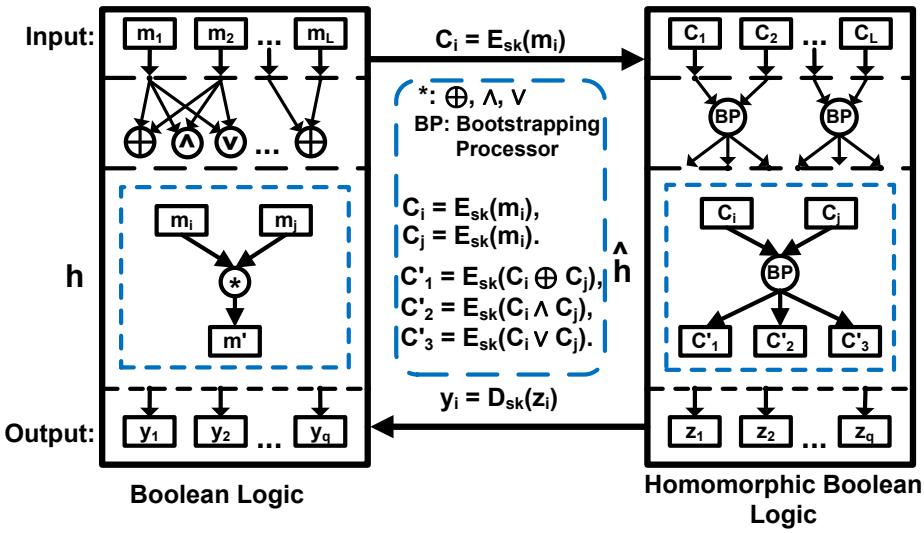

Figure 3.2: Homomorphic computation of Boolean logic-based FHE schemes.

3.2.1 External Product of the Bootstrapping Algorithm

Our recently proposed FHE schemes significantly reduced the cipher expansion [6, 7]. The bootstrapping algorithm involves λ iterations, where λ is the bit length of the private key. All the time-consuming external product steps need to be performed between a trivial RLWE cipher ($\mathrm{RE}_{sk}(t(x))$) and all the GSW ciphers (from $\mathrm{GSW}(x^{u_0 sk_0})$ to $\mathrm{GSW}(x^{u_{\lambda-1} sk_{\lambda-1}})$), which can be defined as:

$$\mathrm{RE}_{sk}(t(x)) \odot \mathrm{GSW}(x^{u_0 sk_0}) \odot \cdots \odot \mathrm{GSW}(x^{u_{n-1} sk_{\lambda-1}}), \tag{3.1}$$

where $\odot$ represents the operand of the external product [7].

In general, the external product in each iteration $\epsilon \in [0, \lambda - 1]$ generates the new RLWE cipher ($f(x)[\epsilon]$) based on the previous RLWE cipher ($f(x)[\epsilon-1]$) and the GSW cipher ($\mathrm{GSW}(x^{u_\epsilon sk_\epsilon})$):

$$f(x)[\epsilon] = f(x)[\epsilon - 1] \odot \mathrm{GSW}(x^{u_\epsilon sk_\epsilon}), \tag{3.2}$$

where u_ϵ and sk_ϵ are the ϵ-th coefficients in the sum of two LWE ciphers u and ϵ-th bit of secret

key sk, respectively.

The iterative data-flow for the external product is shown in Fig. 3.3. Specifically, the RLWE cipher can be represented as two N-degree polynomials $f(x)_0$ and $f(x)_1$ (i.e., a 1×2 matrix) over the ring $R_{N,Q}$, which are then transformed into a 1×4 matrix by the random flattening step. Consequently, a random flattening step is used to split each polynomial into a lower half ($f(x)_{0L}$ or $f(x)_{1L}$) and a higher half ($f(x)_{0H}$ or $f(x)_{1H}$) [7]. Then, in order to perform an efficient point-wise multiplication with GSW cipher, we transform these four polynomials into the NTT-domain as $F(x)_{0L}$, $F(x)_{1L}$, $F(x)_{0H}$, and $F(x)_{1H}$.

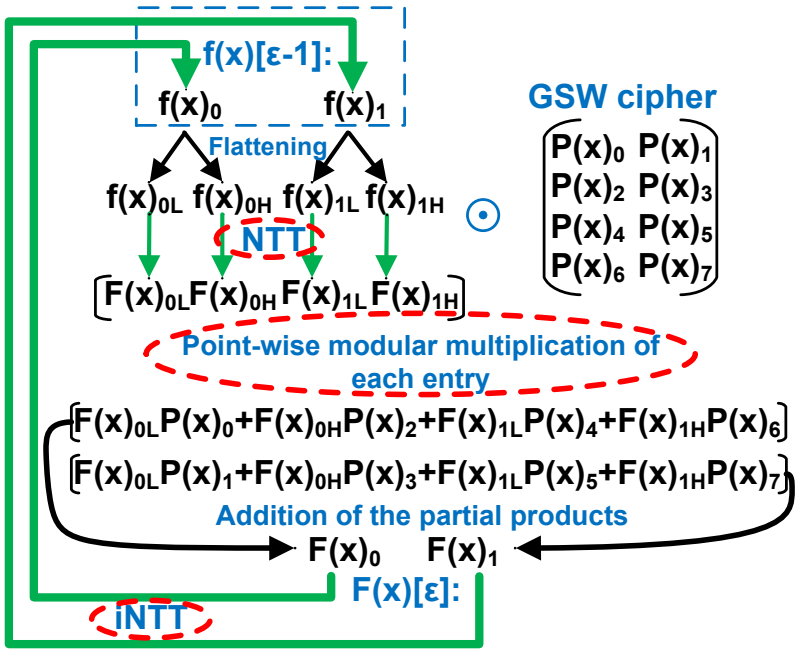

Figure 3.3: Data-flow for the external product $f(x) \odot \mathrm{GSW}(x^{u_\epsilon sk_\epsilon})$.

GSW cipher (a 4×2 matrix) is used to maintain the same size for the ciphertext by using the idea of the gadget matrix [17], which can be defined as:

$$\begin{aligned} &\mathrm{GSW}(x^{u_\epsilon sk_\epsilon}) = G + (x^{u_\epsilon} - 1)bk_\epsilon, \\ &G = \begin{bmatrix} 1 & 0 \\ B & 0 \\ 0 & 1 \\ 0 & B \end{bmatrix}, bk_\epsilon = \begin{bmatrix} a_{1\epsilon}(x) & b_{1\epsilon}(x) \\ a_{2\epsilon}(x) & b_{2\epsilon}(x) \\ a_{3\epsilon}(x) & b_{3\epsilon}(x) \\ a_{4\epsilon}(x) & b_{4\epsilon}(x) \end{bmatrix} + sk_\epsilon G, \end{aligned} \tag{3.3}$$

where G and bk_ϵ represent the gadget matrix (B inside the matrix represents a selected prime number) and bootstrapping key, respectively. Note that the entries of the GSW cipher are denoted from $P(x)_0$ to $P(x)_7$ in Fig. 3.3. More details can be found in papers [6, 7].

The RLWE cipher and the GSW cipher perform an N-degree polynomial multiplication to generate the new $F(x)_0$ and $F(x)_1$ for the next iteration. These operations take most part of the entire computational time, which is the bottleneck of the external product step.

3.2.2 NTT-based Polynomial Multiplication over Ring

Most of the recent schemes are based on the RLWE problem that requires the computations over the ring $R_{N,Q} = \mathbb{Z}[x]/(x^N+1, Q)$, where (x^N+1, Q) is the ideal of $\mathbb{Z}[x]$ generated by x^N+1 and Q [19]. In this case, all the polynomials are reduced so that their degree is smaller than N, while the integer-based coefficients of the polynomial are bounded by $Q-1$. For the binary operations over the ring $R_{N,Q}$, the polynomial additions are fairly simple. However, the polynomial multiplications have a quadratic complexity (i.e., $\mathcal{O}(N^2)$) by using the schoolbook polynomial multiplication. In prior works, the polynomial multiplication modulo x^N-1 is usually accelerated using the NTT to achieve a complexity of $\mathcal{O}(N \log N)$, which is also referred as FFT over the finite field [30]. An N-point NTT is formally expressed as:

$$A_i = \sum_{j=0}^{N-1} a_j \alpha^{ij}, \quad 0 \le i \le N-1, \tag{3.4}$$

where α is the primitive N-th root modulo Q. For its inverse form (iNTT), the expression is given by:

$$a_i = N^{-1} \sum_{j=0}^{N-1} A_j \alpha^{-ij}, \quad 0 \le i \le N-1, \tag{3.5}$$

where N^{-1} is the modular multiplicative inverse of N modulo Q, which can be pre-calculated by the extended Euclidean algorithm [31]. Note that N is required to be a power-of-two in order to perform the radix-2 NTT transform.

In previous designs, the zero-padding steps need to be used for the polynomial multiplier with the modulus x^N+1, which will extend the length to $2N$ [32]. In order to address this issue of doubling the length, one can use a weighted operation on coefficients to reduce the multiplication of two polynomials modulo x^N+1 to the multiplication of two polynomials modulo x^N-1. More

precisely, let Q be a prime such that $2N$ divides $Q-1$, so there is an integer ϕ that has order $2N$ modulo Q, hence $\alpha = \phi^2$ has order N. Then all the roots of $x^N + 1$ are

$$\phi^{2i+1} = \phi\alpha^i, \quad 0 \le i \le N-1,$$

and all the roots of $x^N - 1$ are

$$\phi^{2i} = \alpha^i, \quad 0 \le i \le N-1.$$

Then, for any polynomial $a(x) = \sum_{j=0}^{N-1} a_j x^j$, we have

$$a(\phi^{2i+1}) = \sum_{j=0}^{N-1} a_j \phi^j \alpha^{ij} = \tilde{a}(\alpha^i), \quad 0 \le i \le N-1, \tag{3.6}$$

where $\tilde{a}(x) = \sum_{j=0}^{N-1}(a_j\phi^j)x^j$. Hence the weighted operation from $a(x)$ to $\tilde{a}(x)$ is to multiply the coefficient of x^j by ϕ^j and ϕ^{-j} before the NTT and after the iNTT operation, respectively, where $0 \le j \le N-1$.

The negative wrapped convolution is illustrated in Algorithm 3, and its data-flow chart is shown in Fig. 3.4. Two polynomials $a(x)$ and $b(x)$ will have the weighted operation by multiplying the corresponding ϕ^j to generate two new polynomials $\widetilde{a}(x)$ and $\widetilde{b}(x)$. Then, these two weighted polynomials will perform the NTT transform based on Equation (3.4), followed by a point-wise (element-wise) multiplication in the NTT-domain. Finally, $g(x)$ (i.e., the product of $a(x)$ and $b(x)$) in the NTT-domain will be converted using the iNTT transform in Equation (3.5) and another weighted operation to obtain the final result.

Algorithm 3 Negative Wrapped Convolution [33]

Input: $a(x) = \sum_{j=0}^{N-1} a_j x^j$, $b(x) = \sum_{j=0}^{N-1} b_j x^j$
Output: $g(x) = a(x) \cdot b(x) \mod (x^N + 1)$
1: $\widetilde{a}(x) = \sum_{j=0}^{N-1} a_j \phi^j x^j$ //Weighted $a(x)$
2: $\widetilde{b}(x) = \sum_{j=0}^{N-1} b_j \phi^j x^j$ //Weighted $b(x)$
3: $\widetilde{A}(x) = \text{NTT}(\widetilde{a}(x))$
4: $\widetilde{B}(x) = \text{NTT}(\widetilde{b}(x))$
5: $\widetilde{A}(x) \circ \widetilde{B}(x) = \sum_{i=0}^{N-1} \widetilde{A}_i \widetilde{B}_i x^i$ //Point-wise multipl.
6: $\tilde{g}(x) = \text{iNTT}(\widetilde{A}(x) \circ \widetilde{B}(x))$
7: $g(x) = \sum_{j=0}^{N-1} \widetilde{a} \cdot \widetilde{b}\phi^{-j} x^j$
8: **return** $g(x)$

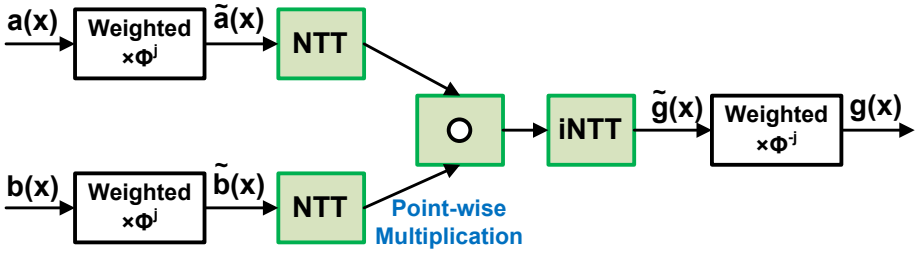

Figure 3.4: NTT-based multiplier with modulus $x^N + 1$.

3.2.3 Prior Works of FHE Implementation

Several software open-source libraries have been implemented for various schemes: SEAL [34], cuHE [35], HElib [36], and TFHE [22]. Specifically, SEAL, cuHE, and HElib implemented somewhat homomorphic encryption schemes, which allow for fixed depths of multiplication but are not limited to additions. TFHE involves the bootstrapping step after every operation, and hence it can perform unlimited homomorphic operations. These software implementations are still fairly slow to be used in real-world applications since the computational time for recryption in the somewhat homomorphic encryption schemes or the bootstrapping steps in FHE is prohibitively long.

For practical deployment of FHE, customized hardware acceleration has emerged as a not only promising but also essential direction. In fact, FPGA and ASIC designs have shown superior performance compared to software implementations on CPU or GPU. This effort has yielded a factor of almost 3,000 in the acceleration of FHE implementation from the first FHE implementation. In past years, a number of VLSI architectures for several homomorphic encryption schemes have been proposed [27–29, 32, 37, 38]. Most of these architectures are integer-based schemes utilizing large integer multiplications and RLWE-based schemes with polynomial multiplications, as these achieve superior computational efficiency compared to early lattice-based schemes that usually involve relatively large public keys and ciphertext sizes. The effort of the hardware accelerations has mainly focused on the large integer or polynomial multiplication, as it is the major performance bottleneck in homomorphic encryption.

For architectures considering the entire schemes, the costly bootstrapping step is usually not included, which thus only supports a limited number of homomorphic operations. FV [16], BGV [21], YASHE [24], and CKKS [39] schemes are generally adopted in these hardware architectures. For instance, some recent works presented the optimized architectures for the evaluation step

of the FV scheme by increasing the parallelism [28] or using the Chinese remainder theorem (CRT) representation for the polynomials [25].

Moreover, the first custom FHE architecture was developed in [40], which includes a recryption step that is homomorphically evaluating the decryption circuit using encrypted secret key bits on the noisy ciphertext. Hardware acceleration for the SEAL library [34], which includes the FV and CKKS schemes was done by applying 4 NTT cores and 2 iNTT cores in parallel to improve the efficiency [29]. Similar designs with 4 NTT cores and special addressing schemes are proposed in [34, 41]. Furthermore, an architecture for a recryption box is also proposed in [27] to ignore the costly bootstrapping operation and accelerate the homomorphic evaluation.

In most of the existing polynomial multiplier architectures, PEs are designed separately for computing different components: NTT, iNTT, point-wise multipliers, and so on [26, 38, 42]. In general, the NTT requires two times the resources as the iNTT, since one polynomial multiplication involves two NTTs on both inputs and only one iNTT after point-wise multiplication. Several prior works consider sharing the same PE between NTT and iNTT [26, 40, 42–45], which could reduce the hardware area usage. However, the parallel property of NTT architectures is not exploited in these works.

Besides, the prior designs have demonstrated that increasing the parallelism of the NTT cores (PEs) can accelerate the computation time compared with the single-core/PE design. However, these designs usually have a fixed number of PEs, which are not scalable or applicable to an arbitrary number of PEs.

3.3 Proposed Architecture

In this section, we propose an efficient architecture of polynomial multiplier for the external product, which leverages the advantages that our specific data-flow brings. As shown in Fig. 3.3, the external product is basically a matrix multiplication between a 1×4 and a 4×2 matrix. The NTT domain GSW cipher can be easily computed in the customized software implementation for each iteration since it does not require a standard NTT transform for the special polynomial $(x^{u_\epsilon} - 1)$ [7]. As a result, the NTT transforms for all the multiplicands (i.e., the 4×2 matrix) in hardware are eliminated. In addition, the matrix multiplication results in 8 point-wise multiplications, which are required to be summed up in a group of 4 by using the adder tree. Finally, iNTT is performed

individually on each of the two sums. To summarize, there are 4 separate NTT operations, 8 point-wise multiplications, 6 modular additions, and 2 separate iNTT operations in our external product step.

3.3.1 Reconfigurable PE

To this end, we first propose a novel time-multiplexed polynomial multiplier architecture with multiple PEs by exploring the parallelism of our scheme. Each PE can be reconfigured to use for NTT, iNTT, and point-wise multiplication (marked in the dashed circle in Fig. 3.3). The proposed reconfigurable PE is customized from the memory-based decimation-in-frequency (DIF) butterfly unit, as shown in Fig. 3.5.

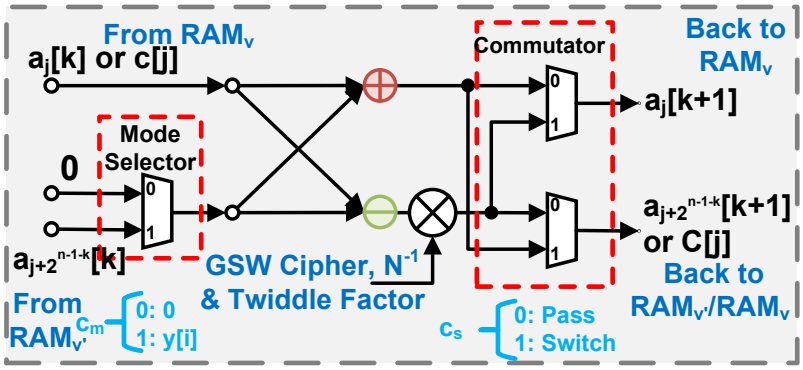

Figure 3.5: The proposed reconfigurable PE.

Each PE consists of one modular multiplier, three multiplexers (MUXs), one modular adder, and one modular subtractor. Specifically, the mode selector (the MUX on the left-hand side in Fig. 3.5) acts as a switch between a butterfly operation and a basic modular multiplication controlled by the signal c_m. When c_m is 1, the PE will be configured as a butterfly unit, whose mathematical expression is given as

$$\begin{aligned} a_j[k+1] &= a_j[k] + a_{j+2^{n-1-k}}[k], \\ a_{j+2^{n-1-k}}[k+1] &= a_j[k] - a_{j+2^{n-1-k}}[k]\alpha^{ij}. \end{aligned} \tag{3.7}$$

Fig. 3.6 illustrates the corresponding data-flow in this mode. In each stage, k, two inputs (i.e., $a_j[k]$ and $a_{j+2^{n-1-k}}[k]$) are read from the corresponding RAMs for each PE. The difference between the two inputs will perform modular multiplication with the twiddle factors in the bottom path. The

intermediate results will then be written back to the memory through the commutator for the next stage $(k+1)$.

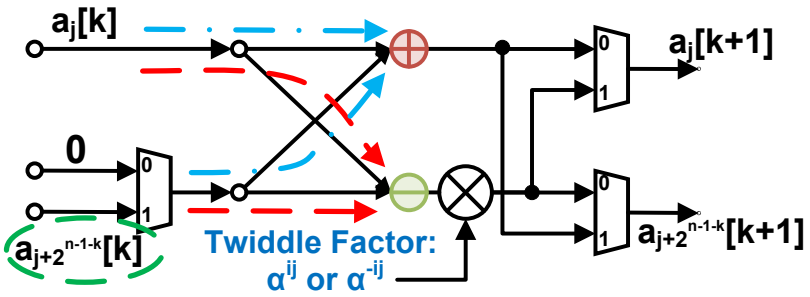

Figure 3.6: Reconfigurable PE in butterfly mode.

When c_m is 0, the modular multiplication mode will be activated, as shown in Fig. 3.7. Consequently, the upper input $c[j]$ will multiply with an integer, which is expressed as

$$C[j] = c[j]\sigma, \tag{3.8}$$

where σ can be ϕ^j, ϕ^{-j}, N^{-1}, or the GSW cipher, while the lower input is always 0. In this case, only one input and one output will be used.

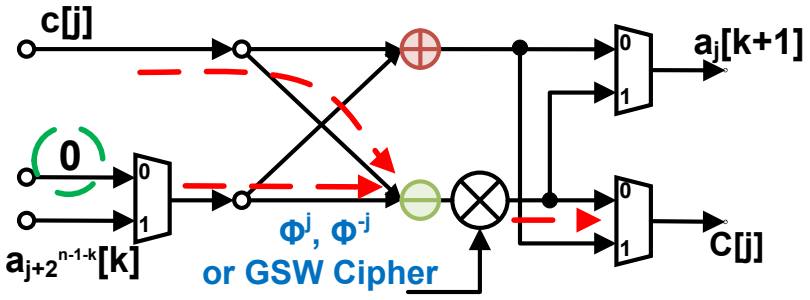

Figure 3.7: Reconfigurable PE in multiplication mode.

3.3.2 Ultra-Highly Parallel Architecture with Variable Number of PEs

Our proposed top-level architecture of the polynomial multiplier is shown in Fig. 3.8, which mainly consists of three components: memories, control units, and arithmetic units. We use the read-only memory (ROM) to store pre-computed values (i.e., twiddle factors ϕ^j, α^{ij} for NTT, ϕ^{-j}, α^{-ij} for iNTT, N^{-1} and the NTT-domain form of the GSW ciphers), and random-access memory (RAM) for the intermediate results of each iteration. Since each PE still requires a significant amount

of computational resources in the existing FHE schemes, it is impractical to implement fully-parallel architectures on most of the current hardware platforms [25,38,46]. In addition, it is also of interest to deploy on embedded or edge devices with fewer resources.

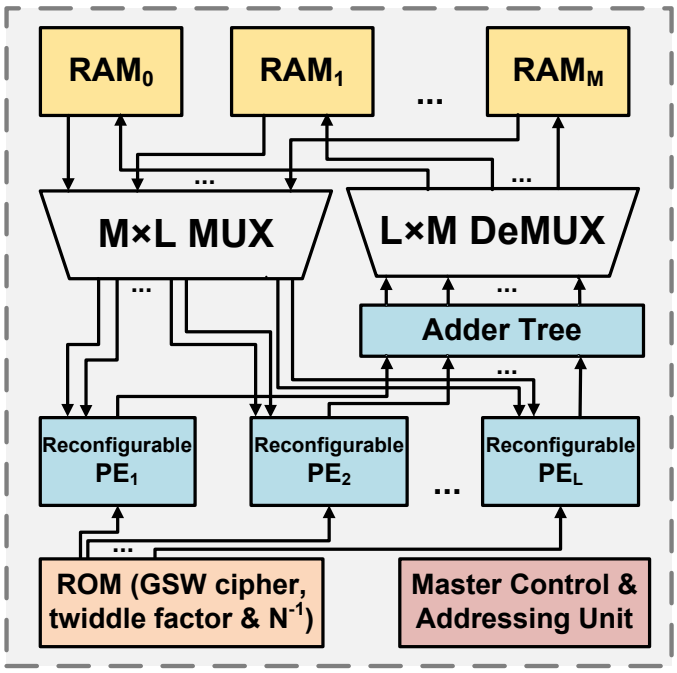

Figure 3.8: NTT-based polynomial multiplier with L PEs.

In general, we wish to implement as many PEs as possible on the targeted hardware platform under the resource constraint so that the highest degree of acceleration is achieved. In fact, the proposed reconfigurable PE eliminates the need for other individual computational units, which also increases the number of PEs that a hardware platform can accommodate. To this end, a proper method for scheduling PE operations is critical. Better scheduling could improve hardware utilization as well as reduce the overall latency of the architecture.

In this chapter, instead of restricting the number of PEs to be a power-of-two form as in our prior work [8] and other existing designs, we consider the general case of L PEs in parallel in the arithmetic unit, where $L \in \mathbb{Z}^+$.

In our prior scheme, each PE has two I/Os, and hence there are $2L$ memory banks in total. However, when L is not a power-of-two, directly using $2L$ memory banks will lead some PEs to be not fully used, which consequently reduces the overall utilization ratio.

Instead, we choose to use M memory banks in the generalized scheme, where M is defined

as

$$M = 2^{\lceil \log_2(L) \rceil + 1} = 2^m. \tag{3.9}$$

Among these memory banks, $2^{\lfloor \log_2 L \rfloor}$ PEs will have direct access to half of the memory banks while the other half can be selected by the rest $(L - 2^{\lfloor \log_2 L \rfloor})$ PEs.

There are $R = 2^r$ words in each memory bank, where r is given by:

$$r = \log_2 N - \log_2 M - 1,\ r \in \mathbb{Z}^+. \tag{3.10}$$

Each word in the memory bank has a size of W, which is determined by the selected modulus Q, i.e., $W = \lceil \log_2 Q \rceil$. Additionally, we denote the index of each word by an (r-1)-bit binary representation:

$$R_b = e_{r-1} e_{r-2} \cdots e_0. \tag{3.11}$$

Note that our design requires dual-port memories that allow us to read the inputs and write the outputs of the PEs simultaneously.

3.3.3 Conflict-Free Memory Addressing and Scheduling Scheme

Memory addressing schemes for NTT-based FHE or lattice-based cryptosystems with one or two PEs have been extensively studied in the literature [29, 38, 43, 46]. However, these works generate irregular memory patterns after the computation, which cannot be used in our scheme that involves multiple iterations. To this end, we develop a conflict-free memory management scheme to integrate all the stages of each polynomial multiplication based on the schedule of L reconfigurable PEs, as presented in Table 3.2. We use $P = P_{end} - P_{begin} + 1$ to denote the number of steps in each type of operation, where every step involves R clock cycles for the words in the memory banks. Thus, the global stage S represents the parallel computations of all L PEs by simply using L to divide the steps P.

When L is a power-of-two [8], the number of steps in each type of operation P is always divisible by L, which implies that all PEs are always in the same type of operation. Besides, when computing NTT/iNTT, since the base of L is the same as the radix of NTT/iNTT, all the PEs are always operating in the same NTT/iNTT stage as well. However, when L is an arbitrary integer,

Table 3.2: Schedule of L reconfigurable PEs in each polynomial multiplication

P_{begin}	P_{end}	Type of Operations	Lower Input	Multiplier Input
1	M	weighted (ϕ^j)	0	twiddle factor
$M+1$	$M+M/2(\log_2 N)$	butterfly	$y[j]$	twiddle factor
$M+M/2(\log_2 N)+1$	$2M+M/2(\log_2 N)$	point-wise MUL	0	GSW cipher
$2M+M/2(\log_2 N)+1$	$2M+M(\log_2 N)$	butterfly	$y[j]$	twiddle factor
$2M+M(\log_2 N)+1$	$3M+M(\log_2 N)$	weighted (ϕ^{-j})	0	twiddle factor
$3M+M(\log_2 N)+1$	$4M+M(\log_2 N)$	$\times\ N^{-1}$	0	N^{-1}

PEs have to be loaded with different operations for certain clock cycles in order to maintain a high utilization ratio.

It can be seen from Table 3.2 that P is not always divisible by L if $L \neq 2^l$, so the global stage S in this type of operation cannot be denoted as an integer. Instead, we express the operations of PEs in global stage S by using a mixed fraction:

$$\frac{P}{L} = S_{fu} + \frac{L_{used}}{L}, \tag{3.12}$$

where S_{fu} is the number of global stages fully using all L PEs and $L_{used} = [0, L-1]$ is the number of PE(s) additionally used in the current type of operation with P steps. We also define the first global stage as $S = 1$.

We illustrate the proposed concept by an example with $M = 8$ and $L = 3$. The number of steps in the weighted operation is $P = P_{end} - P_{begin} + 1 = 8 - 1 + 1 = 8$ which can be further represented as $S_{fu} = 2$ and $L_{used} = 2$, according to Equation (3.12). In other words, the weighted operation fully uses all the 3 PEs in the first and second stages ($S = 1, 2$), and an additional 2 PEs in the third stage. Thus, the next type of operation, i.e., NTT transform, will start with the remaining single PE ($L - L_{used} = 3 - 2 = 1$). In other words, the last computation of the weighted operation and the beginning of the NTT operation are in the same global stage $S = 3$.

In the beginning, we set the coefficients a_j in each RAM_v to be initially located as:

$$j = R\lfloor\frac{v}{2}\rfloor + 2\theta + \text{mod}(v, 2), \tag{3.13}$$

where $\theta \in [0, R-1]$, and $v \in [0, M-1]$ that is the index of the RAM. An example of the initial ordering of $N = 32$ is shown in the blue dashed box of Fig. 3.9.

3.3.3.1 RAM Selection

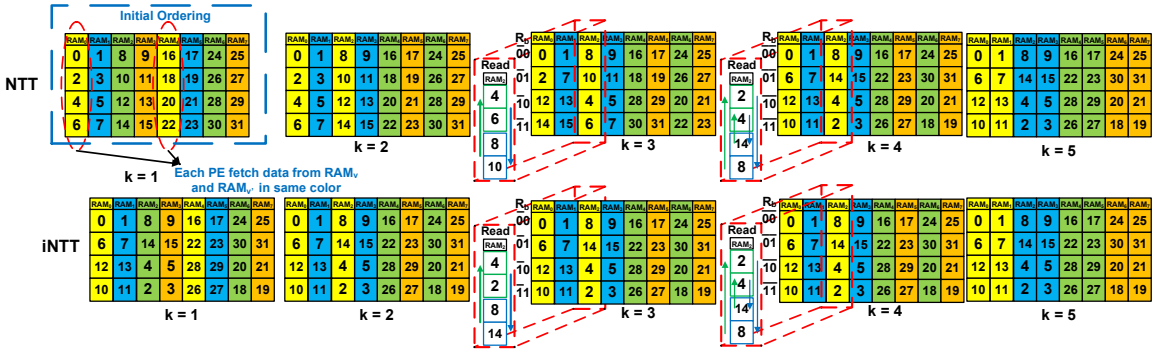

Figure 3.9: An addressing example for the NTT and iNTT with $N = 32$ and $L = 4$.

The RAM selection controlled by the addressing unit is important for the NTT/iNTT operations to avoid memory access conflict. According to Table 3.2, when the PEs are in the NTT/iNTT operations (i.e., configure in the butterfly mode, $c_m = 0$), two banks will be selected to process the data. Therefore, the memory banks chosen by the butterfly mode PEs are always in pairs. For each RAM_v, the other entry in the pair $\text{RAM}_{v'}$ is generally determined by $v' = v + 2^{m-\xi}$, where $\xi \in [1, m]$. We use β to denote a sequence of such pairs (i.e., RAM_v and $\text{RAM}_{v'}$) for the NTT/iNTT. The overall addressing scheme is presented in Algorithm 4, while the corresponding tree-based diagram is illustrated in Fig. 3.10. A total of 2^{m-1} pairs of the RAMs will be selected

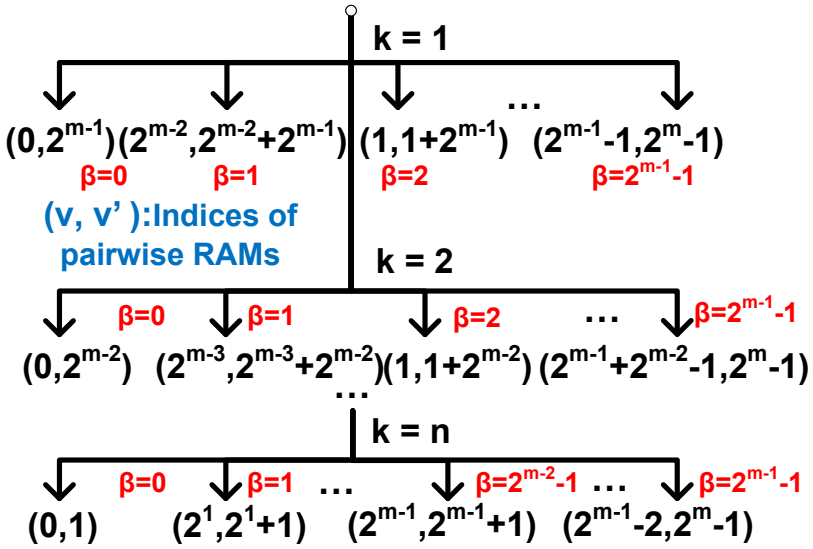

Figure 3.10: Tree-based diagram of the scheduling scheme for NTT/iNTT.

in each stage. We denote the first stage of each NTT/iNTT operation as $k = 1$. As shown in Algorithm 4, when $k \in [1, m-1]$, each pair of RAMs will fetch the coefficients from $\text{RAM}_v(k, \beta)$ and $\text{RAM}_{v'}(k, \beta)$. After the NTT operation is finished, the data in the RAMs will then be used for point-wise multiplication (i.e., the PEs will be in modular multiplier mode). When the PEs are configured as modular multipliers (i.e., $c_m = 1$), each PE can directly read from and write to the same RAM.

Algorithm 4 Conflict-Free Addressing/Scheduling Scheme - RAM Selection (NTT/iNTT)

Input: m
Result: $\text{RAM}_v(k, \beta)$, $\text{RAM}_{v'}(k, \beta)$,
$k \in [1, n]$, $\beta \in [0, 2^{m-1} - 1]$

for $k = 1$ **to** $m - 1$ **do**
  for $\eta = 0$ **to** $2^{k-1} - 1$ **do**
    for $\gamma = 0$ **to** $2^{m-k-1} - 1$ **do**
      for $\psi = 0$ **to** 1 **do**
        $\beta = \text{mod}((\eta + \gamma + \psi 2^{m-k-1}), 2^{m-1})$
        $v = \gamma + \eta 2^{m-k+1} + \psi 2^{m-k-1}$
        $v' = \gamma + \eta 2^{m-k+1} + \psi 2^{m-k-1} + 2^{m-k}$
      end for
    end for
  end for
end for
for $k = m$ **to** $n - 1$ **do**
  for $\eta = 0$ **to** $2^{m-2} - 1$ **do**
    for $\gamma = 0$ **to** 1 **do**
      $\beta = \text{mod}(\eta, 2^{m-1})$
      $v = \gamma + 4\eta$
      $v' = \gamma + 4\eta + 2$
    end for
  end for
end for
if $k = n$ **then**
  for $\eta = 0$ **to** $2^{m-1} - 1$ **do**
    $\beta = \text{mod}(\eta, 2^{m-1})$
    $v = 2\eta$
    $v' = 2\eta + 1$
  end for
end if

Thus, in order to achieve a high utilization ratio, we also ensure the data is stored in the memory banks properly and continuously. Therefore, we merge the scheduling scheme of the NTT/iNTT into memory management in Algorithm 4. As shown in Fig. 3.10, the indices of the second pair RAMs in the first stage are selected as $\text{RAM}_{v=2^{m-2}}$ along with its corresponding pair

$\text{RAM}_{v'=2^{m-2}+2^{m-1}}$, since the data in RAM_0 need to operate with the data in $\text{RAM}_{2^{m-2}}$ at the beginning of the next stage. Thus, this scheduling will lead to no conflict even if two NTT/iNTT stages are overlapped in a single global stage.

3.3.3.2 Read and Write Pattern of NTT/iNTT

In order to realize the proposed scheduling, the commutator in each PE, as shown in Fig. 3.5, is used to control the memory banks of the upper and lower outputs to assist addressing for the NTT/iNTT operation by using a parameter c_s. For example, in the NTT that adopts a DIF FFT data-flow, the interval between the two inputs of the butterfly operation starts from $\frac{N}{2}$ for the first stage and then decreases by a factor of two for each subsequent stage.

The commutator switches the outputs to avoid memory conflict at the first NTT operation only when $k \geq m-1$. The write pattern of the RAM banks is controlled by a counter w, which is expressed as:

$$w = n - k - 1. \tag{3.14}$$

When the w-th bit of R_b (i.e., b_w) is 1, the two outputs of the PEs will be switched (i.e., $c_s = 1$); otherwise, the two outputs are written back to the same RAMs directly. Based on our special initial ordering as shown in Equation (3.13), the iNTT operations do not require switching since it only depends on the read pattern to access the words in the memory.

For all NTT/iNTT operations at stage $k \in [m, n-1]$, the read pattern of the addressing unit is dependent on the written pattern of the commutator. The words located in RAM_v can be fed into the corresponding PE directly, while the words in $\text{RAM}_{v'}$ will be accessed based on the binary representation of its read pattern:

$$R_b^{'} = \overline{e_{r-1} \dots e_{r-z}} \dots e_0, \tag{3.15}$$

where $z \in [1, r]$.

An example for $N = 32$ with four PEs ($L = 4$) is shown in Fig. 3.9. In this case, $m = \log_2 8 = 3$ and $n = \log_2 32 = 5$. For the RAM selection, the banks with the same color indicate that the words in these two RAMs will be fed into the same PE. Then, when the second stage of the NTT operation begins, data whose indices with the most significant bit (MSB) of 1 (i.e., $e_w = 1$) will be switched by the commutator.

In the next stage, the data with indices, whose second significant bit equals to 1 and 0, will be switched and passed, respectively. However, the rest of the NTT/iNTT and modular multiplication operations in each iteration will bypass the switches.

According to the read pattern of NTT/iNTT, the first input of all the PEs ($a_j[k+1]$ in Fig. 3.5) will read from RAM$_v$, while the other input ($a_{j+2^{n-1-k}}[k+1]$ in Fig. 3.5) will read the words with the indices of $R_b^{'} = \overline{e_1}e_0$ or $R_b^{'} = \overline{e_1 e_0}$ from RAM$_{v'}$ for the stages 3 or 4 of the NTT.

3.3.3.3 Scheduling of the Global Stages

When $L \neq 2^l$, since the PEs might perform different types of operations in the same global stage, we need to guarantee that all the operations in this global stage are uncorrelated, even though they correspond to different types of operations. The scheduling scheme for NTT/iNTT is illustrated in Algorithm 4. Meanwhile, we also propose a scheduling scheme for the modular multiplication operation to assist with the NTT/iNTT operation, which is summarized in Algorithm 5. The output of the algorithm is the order set **D**, which is used by L PEs in the modular multiplication mode.

Algorithm 5 Scheduling Scheme of Modular Multiplier for $L \neq 2^l$

Input: m
Result: Order set **D**,

Initialization: Empty vector **D**
for $\psi = 0$ **to** 1 **do**
  $\mathbf{D} \lhd v = \psi$ //$\lhd$: push back at the end of vector
  $\mathbf{D} \lhd v = 2^{m-1} + \psi$
  for $\rho = 2 : m-1$ **do**
    $\mathbf{D} \lhd v = 2^{m-\rho} + \psi$
    $\mathbf{D} \lhd v = 2^{m-\rho} + \psi + 2^{m-1}$
  end for
end for

An example of the scheduling for $N = 32$ and $L = 3$ is shown in Fig. 3.11. The weighted operations (i.e., PEs will be in the modular multiplication mode) will be computed at the beginning of the polynomial multiplication. Based on the order set **D** generated from Algorithm 5, the data in RAM$_{v=\{0,4,2\}}$ will be firstly fed into 3 PEs and data in RAM$_{v=\{6,1,5\}}$ will follow in the next global stage. Then, there are two types of operations when $S = 3$, which involve weighted operation for the data in RAM$_{v=\{3,7\}}$ and one NTT operation for the data in pair RAMs (RAM$_0$ and RAM$_4$). Then, the next global stage ($S = 4$) only includes the NTT operation to compute the data in RAM$_{v=\{2,1,3\}}$ along with the data in another RAM$_{v'=\{6,5,7\}}$ in the 3 PEs. The rest of the global stages remain in

the same pattern. The entire polynomial multiplication will be completed at global stage 25.

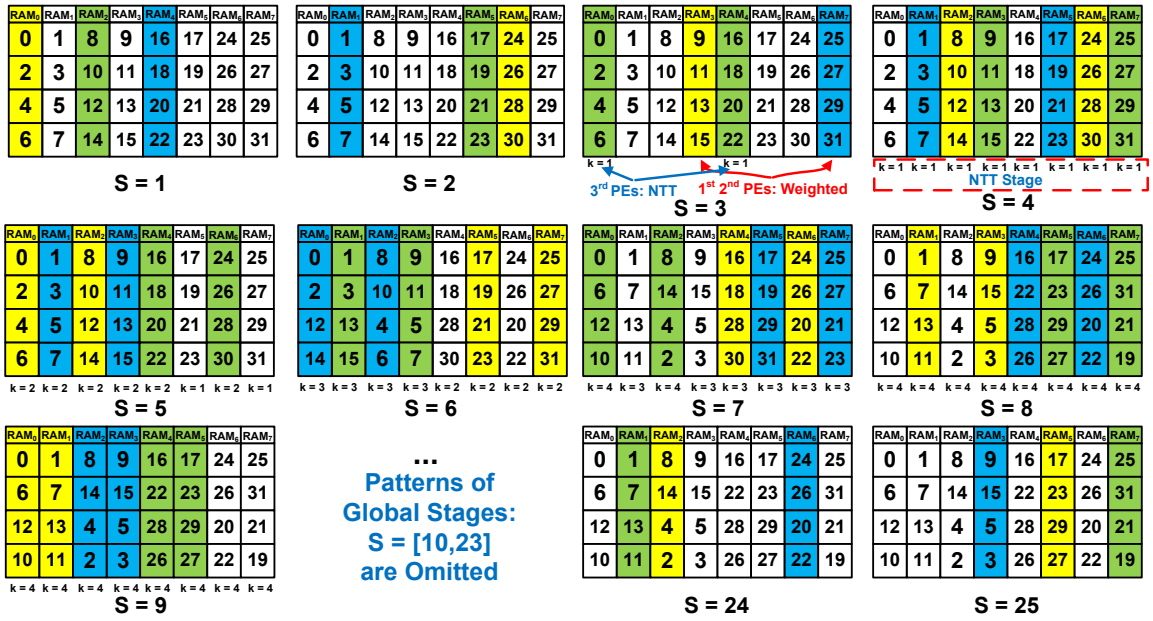

Figure 3.11: Scheduling example for the polynomial multiplication with $N = 32$ and $L = 3$.

Note that the proposed scheduling and memory management methods can also be extended to other FHE schemes that involve polynomial multiplications.

3.4 Experimental Results

3.4.1 Evaluation of NTT-based Polynomial Multiplication

One objective of our design is to achieve a high utilization ratio for all the PEs. In our prior work for a power-of-two number of PEs [8], the utilization ratio is always 100% based on the properties of radix-2 NTT transform. However, in this proposed design with a variable number of PEs, the last global stage, in some cases, will not fully use all L PEs. Overall, the utilization ratio is calculated as:

$$U = \frac{\lfloor M(4+n)/L \rfloor (L + \mathrm{mod}(M(4+n), L))}{S_{total} L}, \tag{3.16}$$

where $S_{total} = \lceil M(4+n)/L \rceil$ is the total number of global stages.

We plot the utilization ratio of our design for the number of PEs from 1 to 16 and the polynomial degree from 32 to 4096, as shown in Fig. 3.12. It can be observed that our design

achieves a very high utilization ratio, as all samples are located in the interval between 96% to 100%, with an average of 99.18%.

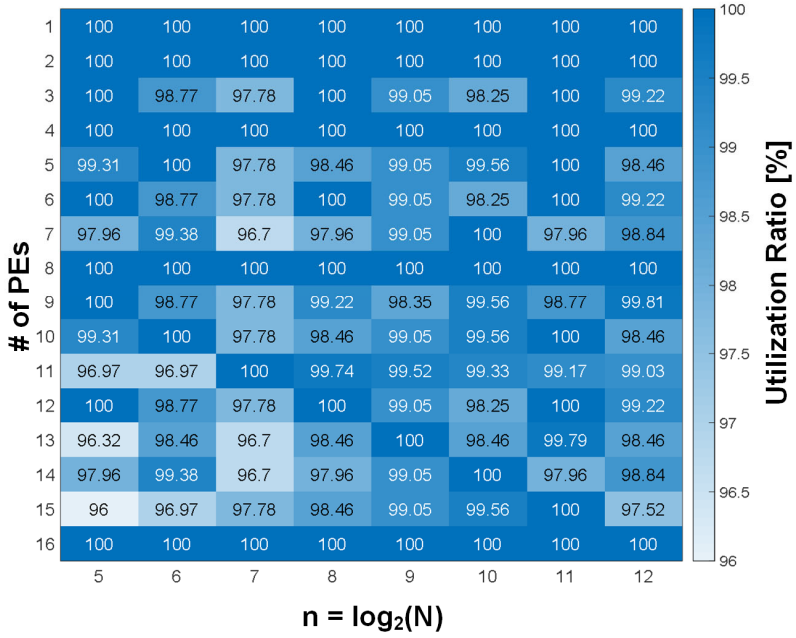

Figure 3.12: Utilization ratio of proposed design ($L = [1, 16]$) with $N = 2^n, n \in [5, 12]$.

In addition, we also expect our proposed design can significantly improve efficiency by exploring the reconfigurability of the PE and maximizing the number of PEs for the targeted hardware platform. For a fair comparison, we define a general form of polynomial multiplication by using the negative wrapped convolution as shown in Fig. 3.4, which consists of two NTT, one iNTT, and three modular multiplication operations. The latency expression of our proposed architecture with degree N polynomial multiplication is given by:

$$T_{poly} = (3N/2(\log_2 N) + 3N)/L. \tag{3.17}$$

Meanwhile, by following a similar expression, the latency in the works of Chen [38] and Pöppelmann [47] are $3N/4(\log_2 N) + N/4$ and $3N/2(\log_2 N) + 11N/2$, respectively. The comparison for a wide range of polynomial degrees with four PEs is illustrated in Fig. 3.13. Clearly, our proposed design has the lowest latency under any polynomial degree. Averagely, the latency in our proposed design is reduced by 39.84% and 78.55%.

To evaluate the hardware cost, we implement our proposed design with different PEs using

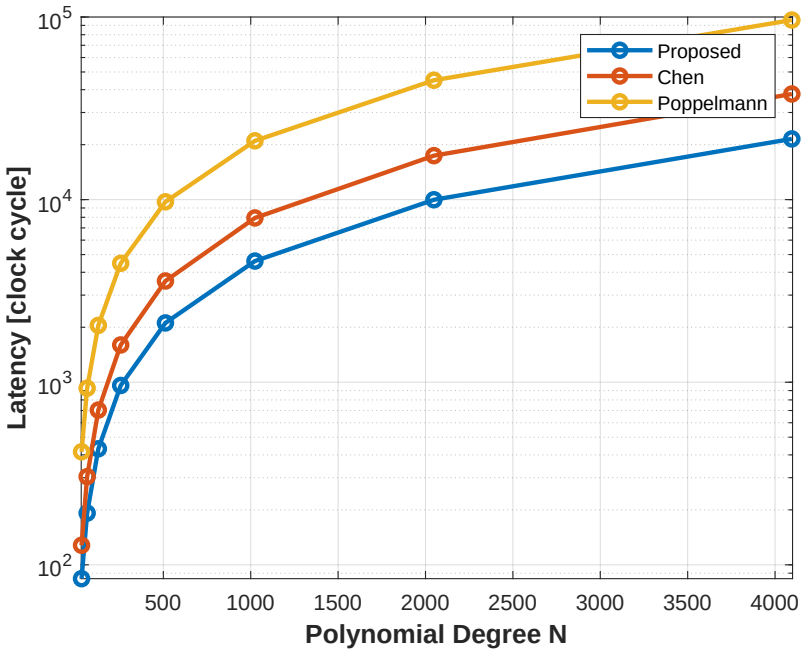

Figure 3.13: Comparison of latency with prior works: Chen [38], Pöppelmann [47]

Verilog HDL, which are then mapped into a 32nm technology node based on Synopsys SAED library. We select a 32-bit prime Q and polynomial degrees as $N = 1024, 4096$. Table 3.3 summarizes the experimental results of area, latency, and area-time product (ATP) from single PE to 16 PEs. We also plot the relationship between area and latency in Fig. 3.14. As expected, area consumption increases, and the latency drops with the increase in the number of PEs. Since our proposed scheme has the flexibility to increase the number of PEs without a power-of-two constraint, our proposed scheme has the potential to further decrease the latency compared to prior works.

3.4.2 Comparison of External Product Implementations

To further evaluate and analyze the performance of our proposed design, we compare our synthesized results with some prior works in [38, 47] by implementing our recently proposed FHE scheme [7]. Since the cost of the control flow or addressing units in these prior works is not presented, we only compare the performance of the arithmetic units. In fact, according to our experimental results, the overhead of the control logic yielded from our proposed addressing scheme is rather negligible, compared to the cost of the PEs. We set the modulo Q as a 71-bit prime, which is consistent with the schemes in [7]. Although the number of PEs in our design can vary for different

Table 3.3: Performance of proposed design ($L = [1, 16]$) with $N = 1024, 4096$

	Area		Latency		ATP[$\times 10^8$]	
L	**Gates**	[μm^2]	**1024**	**4096**	**1024**	**4096**
1	10353	15530	18432	86016	1.91	7.63
2	20712	31068	9216	43008	1.91	7.64
3	30617	45925	6144	28672	1.88	7.46
4	40458	60687	4608	21504	1.86	7.57
5	51311	76966	3687	17204	1.89	7.42
6	60353	90529	3072	14336	1.85	7.42
7	69474	104211	2634	12288	1.83	7.32
8	78367	117550	2304	10752	1.80	7.22
9	86628	129942	2048	9558	1.77	7.09
10	101350	152025	1844	8602	1.87	7.47
11	111938	167908	1676	7820	1.88	7.50
12	121110	181665	1536	7168	1.86	7.44
13	125954	188931	1418	6617	1.79	7.14
14	135074	202612	1317	6144	1.78	7.11
15	144681	217022	1229	5735	1.78	7.11
16	153386	230083	1152	5376	1.77	7.07

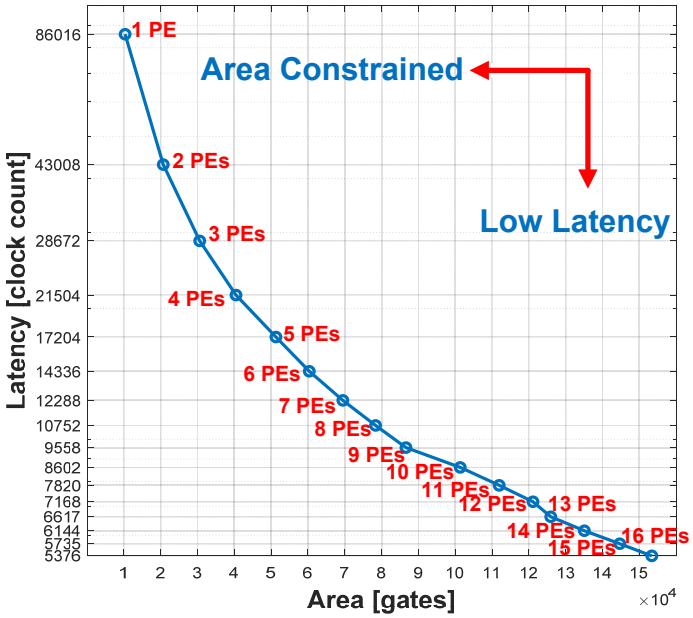

Figure 3.14: Performance of $L = [1, 16]$: Area vs. Latency

hardware platforms, we present the performance of the proposed design with the same number of modular multipliers as the previous works for a fair comparison. The results are summarized in Table 3.4.

The latency of our design implemented in the external product in [7] is

$$T = (3N \log_2 N + 8N)/L. \tag{3.18}$$

In particular, for the architecture with one PE, the NTT and weighted operations take $4(\frac{N}{2}(\log_2 N - 1) + N)$ cycles. In addition, 8 polynomial multiplications consume $8N$ cycles in total. Then, the two iNTT operations, along with the weighted step and multiplying by N^{-1}, require $2(\frac{N}{2}(\log_2 N - 1) + 2N)$ cycles.

Note that since the α^{-ij} at the last stage of NTT and the first stage of iNTT are 1, the modular multiplication can be bypassed for these butterfly operations. For the architecture with L PEs, the latency is reduced by a factor of L, given that all the PEs are fully utilized.

It can be seen from Table 3.4 that our proposed design reduces the ATP by 43.44% and 39.08% on average, compared to [47] and [38], respectively. Furthermore, as the proposed method could achieve full utilization in this case for all the PEs, and the latency is also significantly reduced, i.e., by 42.03% compared to that of the 4-PE architecture in [38].

Table 3.4: Performance of the proposed method and comparison

	Proposed ($L = 1$)		Pöppelmann [47]		**Proposed** ($L = 4$)		Chen [38]	
# Mod. Multipliers	1		1		4		4	
# Area [μm^2]	57071		87422		205181		195449	
# NAND Gates	38047		58281		136787		130299	
Power [$\times 10^4 \mu$W]	2.30		3.03		8.64		8.64	
N	Cycles	ATP ($\times 10^9$)	Cycles	ATP ($\times 10^9$)	Cycles	ATP ($\times 10^9$)	Cycles	ATP ($\times 10^9$)
512	20992	0.79	24064	1.40	5120	0.70	8704	1.13
1024	45056	1.71	51200	2.98	11008	1.50	18944	2.47
4096	204800	7.79	229376	13.33	51200	6.86	88064	11.4

3.5 Conclusion

This chapter presented a novel ultra-highly parallel polynomial multiplier for accelerating homomorphic computations. The architecture employs a novel reconfigurable PE, which can be used as a butterfly operation for the NTT/iNTT or modular multiplication.

Our design, to the best of our knowledge, for the first time, considers a variable number of PEs to accelerate the polynomial multiplication of external products in the bootstrapping algorithm. In addition, efficient conflict-free memory management and scheduling schemes are also proposed to ensure the architecture has a nearly full utilization ratio. Comprehensive experimental results are presented to verify the effectiveness of the proposed design.

Future work will be directed towards further increasing the parallelism of our architecture by applying the CRT to split the large prime into several small primes.

Chapter 4

Pipelined High-throughput NTT Architecture for Lattice-Based Cryptography

Lattice-based cryptography is a powerful cryptographic primitive that can achieve post-quantum security. The most computationally-intensive operations in the lattice-based cryptographic schemes are the polynomial multiplications over the ring, which can be accelerated by adopting the number theoretic transform (NTT) in practical applications. This chapter proposes a novel hardware accelerator for the NTT algorithm for lattice-based cryptography applications, which can achieve full utilization for all the hardware components. The key ideas involve exploiting well-designed folding sets and applying the *folding* transformations to adapt the fast Fourier transform (FFT) multi-path delay commutator architectures and a lightweight modular multiplier. [1]

4.1 Introduction

Lattice-based cryptography has emerged as a promising technique for both post-quantum cryptography (PQC) [4] and homomorphic encryption (HE) [5]. PQC schemes, including quantum-resistant public-key encryption, key encapsulation mechanism (KEM) algorithms, and digital signa-

[1] This work is presented in [10].

ture algorithms, are believed to be secure even under attacks from quantum computers. Besides, HE is a cryptographic method that allows processing the data (e.g., multiplication and addition) directly on the ciphertexts without endangering data privacy.

However, the computations over the polynomial ring in lattice-based cryptography, including polynomial multiplication and polynomial addition, are quite expensive. Specifically, the polynomial multiplication over the ring is the main bottleneck since the degree and coefficients' moduli for the polynomial are typically large in order to ensure high-level security of the cryptosystems and have sufficient space to carry the information in the ciphertext. Therefore, hardware accelerations for the polynomial multiplication over the ring are essential in facilitating the practical deployment of real-world applications. Most prior works of hardware accelerations for lattice-based cryptography applied the number theoretic transform (NTT) algorithm to the polynomial multiplication, which can reduce the quadratic complexity to the log-linear complexity [33].

These prior works only utilize a limited number of processing elements (PEs) (only one or two PEs) due to the resource-constrained hardware platforms [48–50], resulting in a long processing time. Fortunately, recent efficient PE designs [9,51], along with the technology scaling and significant advances in computing power, enable parallelizing more PEs in FPGA boards or ASIC platforms. In this chapter, we propose a high-throughput and efficient accelerator for the NTT-based polynomial multiplication with $\log_2 n$ PEs operating in parallel by adopting the techniques from fast Fourier transform (FFT) multi-path delay commutator (MDC) architectures, where n is the degree of the polynomial. The contributions are summarized below:

- A pipelined NTT architecture is developed to accelerate polynomial multiplication for lattice-based cryptography, which achieves full hardware utilization and a higher level of parallelism.
- The proposed architecture utilizes a lightweight Barrett reduction algorithm-based modular multiplier, which allows more PEs to process in parallel while only incurring a small resource overhead than prior works.
- Our experimental results show that the proposed design can significantly improve the throughput and area-time product (ATP), compared to prior works.

The rest of this chapter is organized as follows: Section 4.2 reviews the mathematical background for the NTT-based polynomial multiplication and the corresponding hardware architectures in prior works. Section 4.3 presents the details of our hardware architecture design. The performance

of our proposed architecture is provided and analyzed in Section 4.4. Finally, Section 4.5 presents remarks and concludes the chapter.

4.2 Background

4.2.1 Basic Notations

The element (polynomial) of the ring $R_{n,q} = \mathbb{Z}_q[x]/(x^n + 1)$ is denoted as $a(x)$, where n is a power-of-two number and q is a prime. The i-th coefficient inside the polynomial $a(x)$ is represented as a_i, i.e., $a(x) = \sum_{i=0}^{n-1} a_i x^i$. We also denote the total stages of the NTT transform as m where $m = \log_2 n$, and k is the current stage of the NTT transform in the rest of this chapter.

4.2.2 NTT-based Polynomial Multiplication

In general, the polynomial multiplication between two polynomials

$$a(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}, \tag{4.1}$$

and

$$b(x) = b_0 + b_1 x + b_2 x^2 + ... + b_{n-1} x^{n-1} \tag{4.2}$$

over ring $R_{n,q}$ can be represented as

$$p(x) = a(x) \cdot b(x) \mod (x^n + 1, q). \tag{4.3}$$

This operation requires the coefficients of the produced polynomial $p(x)$ to be less than q but greater or equal to zero, and the degree of $p(x)$ to be less than n. For the lattice-based cryptography applications, the polynomial degree n can be in the range of hundreds of thousands for the PQC and HE schemes, which becomes the bottleneck for the implementations in both software and hardware. Using the conventional schoolbook yields a complexity of $\mathcal{O}(n^2)$. In this case, acceleration by adapting NTT can reduce the time complexity to $\mathcal{O}(n \log n)$.

NTT-based polynomial multiplication first converts the polynomials $a(x)$ and $b(x)$ to their NTT-domain $A(x)$ and $B(x)$, respectively. Next, the point-wise multiplication is executed to generate the NTT-domain produced polynomial $P(x)$. Then, an inverse NTT (iNTT) transform is

followed to transform $P(x)$ back to the original algebraic domain polynomial $p(x)$. Thus, an n-point NTT transform is mathematically expressed as:

$$A_i = \sum_{j=0}^{n-1} a_j \alpha^{ij} \mod q, \quad i \in [0, n-1], \tag{4.4}$$

where α is the primitive n-th root of unity modulo q (i.e., twiddle factor), which satisfies $\alpha^n \equiv 1 \mod q$. For its inverse form (iNTT), the expression is given by:

$$a_i = n^{-1} \sum_{j=0}^{n-1} A_j \alpha^{-ij} \mod q, \quad i \in [0, n-1], \tag{4.5}$$

where n^{-1} is the modular multiplicative inverse of n with respect to modulo q, calculated by the extended Euclidean algorithm. The conventional NTT-based polynomial multiplication for a ring $x^n + 1$ employs zero padding to pad additional n zeros and performs $2n$-point NTT transform for each polynomial. A more efficient method, namely negative wrapped convolution, is able to apply an n-point NTT transform to reduce the redundant computations for the NTT transform as in conventional methods [33]. The negative wrapped convolution is illustrated in Algorithm 6. The input polynomials $a(x)$ and $b(x)$ need to be multiplied by the weights ϕ to generate $\widetilde{a}(x)$ and $\widetilde{b}(x)$ before performing the NTT transform, where ϕ is the primitive $2n$-th root of unity modulo q. A similar weighted operation needs to be applied after the iNTT transform (as described in Step 4 of Algorithm 6). Note that an NTT-compatible prime is also required, i.e., q must satisfy that $(q-1)$ is divisible by $2n$.

Algorithm 6 Negative Wrapped Convolution [33]

Input: $a(x), b(x) \in R_{n,q}$

Output: $p(x) = a(x) \cdot b(x) \mod (x^n + 1, q)$

1: $\widetilde{a}(x) = \sum_{j=0}^{n-1} a_j \phi^j x^j$, $\widetilde{b}(x) = \sum_{j=0}^{n-1} b_j \phi^j x^j$

2: $\widetilde{A}(x) = \mathrm{NTT}(\widetilde{a}(x))$, $\widetilde{B}(x) = \mathrm{NTT}(\widetilde{b}(x))$

3: $\widetilde{P}(x) = \widetilde{A}(x) \circ \widetilde{B}(x) = \sum_{i=0}^{n-1} \widetilde{A}_i \widetilde{B}_i x^i$

4: $\tilde{p}(x) = \mathrm{iNTT}(\widetilde{P}(x))$

5: $p(x) = \sum_{j=0}^{n-1} \widetilde{p_j} \phi^{-j} x^j$

4.2.3 Prior Works of NTT-based Polynomial Multiplier

In the literature, NTT-based polynomial multiplication for lattice-based cryptography can be broadly classified into two categories: the memory-based architecture [9, 49–51] and pipelined architecture [52–54].

The memory-based architecture typically consists of a small number of the PEs along with the external memory to communicate with the PEs for each addition/subtraction and multiplication (i.e., the butterfly operation for NTT). Such architectures allow the designers to reconfigure the number of PEs under the area budget for different performance requirements. However, since all the intermediate results need to be read from and written to memory, the incurred communication overhead is very large. Besides, based on the construction of the NTT algorithm, complicated addressing and control schemes are often required to feed data to the correct PEs.

The pipelined architectures have also been studied recently for both PQC and HE schemes [53, 54]. All of these works adopt the original radix-2 multi-path delay commutator (R2MDC) FFT architecture. The R2MDC architectures maintain a low area consumption compared to the fully parallel architecture while still achieving a low latency compared to the prior memory-based designs with only a few PEs. However, to the best of our knowledge, these prior R2MDC-based architectures only achieve 50% hardware utilization. In this chapter, we propose a novel design with full hardware utilization.

4.3 Hardware Architecture for Proposed Design

4.3.1 Processing Element for Butterfly Operation

The core step for the NTT-based polynomial multiplication is the butterfly operation. The architecture for the butterfly operation is shown in Fig. 4.1, and the data-flow graph is the same as the radix-2 decimation-in-frequency (DIF) FFT algorithm [55]. This PE mainly consists of one modular adder, one modular subtractor, and one modular multiplier.

The modular adder and modular subtractor are simply constructed by two adders/subtractors. However, the modular multiplier is much more costly than the general multiplier. The requirement that the modulus of the modular multiplier has to be an NTT-compatible prime makes the modular reduction method more expensive than the case with a power-of-two modulus.

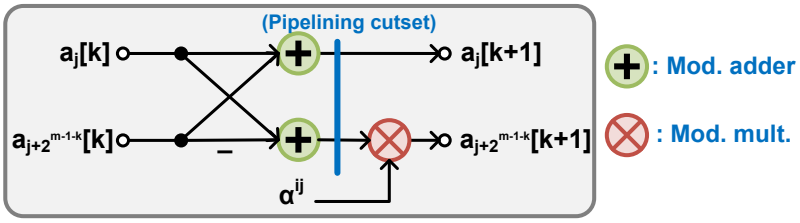

Figure 4.1: PE of the DIF-based butterfly operation.

Modular multiplication for PQC in the literature typically employs a Barrett reduction-based algorithm [56]. Barrett reduction algorithm uses integer multiplications instead of expensive division to compute the modular reduction by mapping the product of two integers back to $\mathbb{Z}_q$. Since the bit-lengths of the moduli used in PQC KEM schemes are usually less than twenty bits, the integer multiplication in the Barrett reduction algorithm can be completed in a short period of time using only one DSP unit or a small amount of look-up tables (LUTs) in an FPGA. The Barrett reduction for the product z of integer u times v ($u, v \in [0, q-1]$) can be expressed as

$$p \approx z - \Big((z \cdot q_{inv}) \gg len(z) \Big) \cdot q, \tag{4.6}$$

where p is the modular product in the range of $[0, q-1]$, $len(z)$ is equal to or slightly less than the bit-length of z, $q_{inv} = \lfloor 2^{len(z)}/q \rfloor$, while "$\gg$" represents the right shift operation [56].

To reduce the implementation cost, the modulus is constructed such that it can be expressed using few positive or negative power-of-two terms as is the case in most of the PQC standards. For example, a commonly used modulus format is $q = 2^{r1} + 2^{r2} + 1$, where $r1 > r2$. The multiplication between this q and the intermediate result z_k ($z_k = (z \cdot q_{inv}) \gg len(z)$) is calculated as

$$\begin{aligned} z_q &= z_k \cdot q = z_k \ll r1 + z_k \ll r2 + z_k \\ &= \Big((z_k + (z_k \ll d1)) + (z_k \gg r2) \Big) \Big|\Big| \Big(z_k[r2-1:0] \Big), \end{aligned} \tag{4.7}$$

where $d1 = r1 - r2$, and "$\ll$", "$||$" represent the left shift and concatenation of two binary numbers, respectively. $z_k[r2-1:0]$ denotes the bits from the $(r2-1)$-th bit to the least-significant bit (LSB) of z_k.

Based on Equations (4.6) and (4.7), the architecture for this modular multiplier using the Barrett reduction algorithm is shown in Fig. 4.2. Specifically, the components boxed in green

implement the operations of Equation (4.7), which only has two adders with short bit-lengths (around the same bit-length of q). Then, z_q is subtracted by the product z, which generates an intermediate result p' of desired modular product p. Finally, a multiplexer (MUX) selects the correct result for p by comparing p' with q. If it is higher than q, p' needs to be subtracted by q. Otherwise, $p = p'$. Apart from the algorithm-based optimization, three pipelining stages are added to this modular multiplier to reduce the critical path for achieving a higher frequency.

As opposed to PQC, the modular multiplications for the HE schemes are usually accelerated by applying the Karatsuba multiplication along with simplifications of the modular reduction based on special NTT-compatible primes. For example, the work in [11] introduced an efficient modular multiplier design for certain prime patterns, which only requires several additions/subtractions to combine the short bit-length partial products into the modular product. During each butterfly operation, the indices of two inputs ($a_j[k]$ and $a_{j+2^{m-1-k}}[k]$) vary in different stages k, which is another main challenge for the NTT transform hardware design.

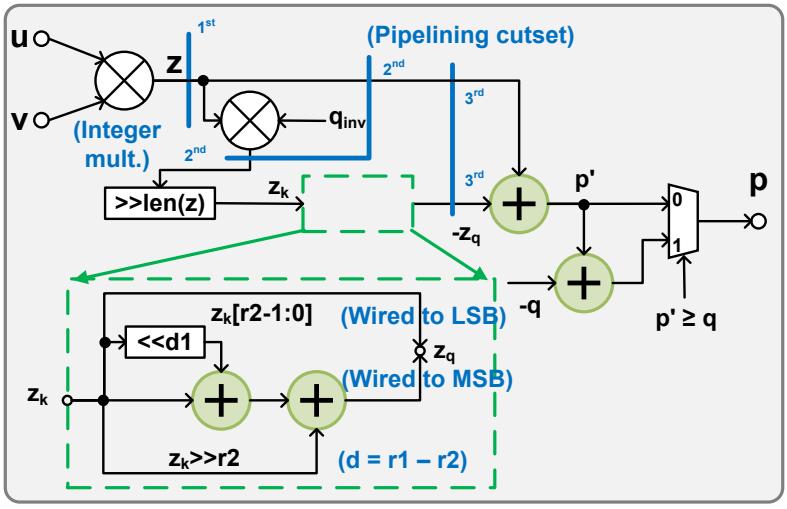

Figure 4.2: Barrett reduction-based modular multiplier architecture.

4.3.2 Top-level Architecture Using Optimized Folding Transformation

For each NTT transform, there are $m\frac{n}{2}$ butterfly operations in total. Using a fully parallel design is not feasible since the hardware cost of each PE is expensive, and the polynomial degree is large. Compared to memory-based architectures as in prior works, the pipelined architectures can significantly reduce the communication time between the PEs and the memory in each butterfly

operation. However, as we discussed in Section 4.2.3, the existing pipelined designs suffer from low hardware utilization, which results in low throughput.

A pipelined architecture applies *folding transformation* to map several operations from the fully parallel design to a single PE in a time-multiplexed manner [57, 58]. A common method for the pipelined FFT/NTT architectures is to reduce the number of PEs from $m\frac{n}{2}$ to m. In a folded architecture, the intermediate results are stored in the registers of the data-path, which are allocated to an appropriate PE for computation by the control unit. The data-flow graph for the 16-point forward NTT transform is shown in Fig. 4.3, which includes steps 1 and 2 for the polynomial $a(x)$ in Algorithm 6. Each colored circle represents one butterfly operation. The index differences of butterfly operation are halved from the first stage to the last stage. Since all the butterfly operations in the same stage have the same index difference for the two inputs, they can be executed using the same PE after folding. As in this 16-point example, butterfly operations in the same color are operated in a single PE, where the input index differences of these four PEs are reduced from 8 in stage 0 to 1 in stage 3.

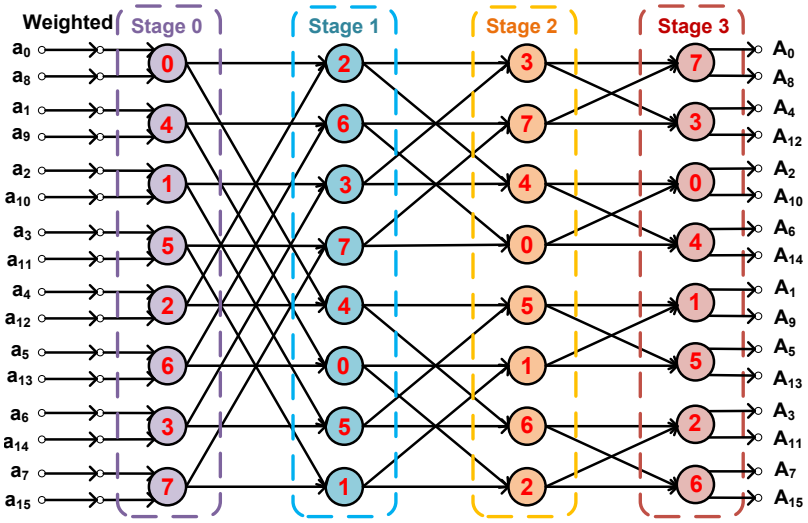

Figure 4.3: Data-flow graph of the 16-point forward NTT transform. Folding orders for each stage are highlighted in red in circles. Operations in the same color will be fed into the same PE. Note that the multiplications and additions/subtractions are omitted in this diagram.

To control the data for each PE, a conventional method for the NTT-based polynomial multiplier using the R2MDC architectures is to add an identical amount of registers as the index differences in both the upper and lower paths with an additional commutator, which results in a low-throughput and utilization [53, 54]. As opposed to these prior methods, we improve the throughput

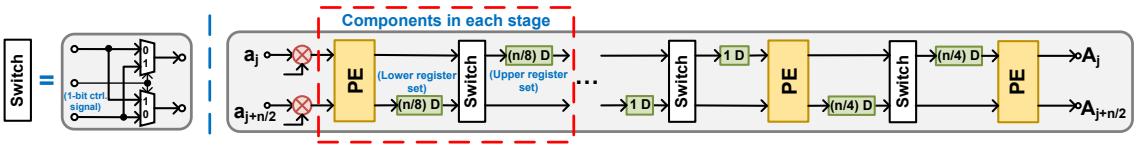

Figure 4.4: Top-level architecture of the R2MDC NTT transform.

and utilization by optimizing the folding transformation. To address this issue, we adopt the design of a folding set (i.e., the folding orders of the butterfly operations in each PE) in [59] to achieve a 100% utilization for the PEs. Fig. 4.3 shows the optimized folding order of butterfly operations for each PE, which is highlighted in red in the circles. To ensure the data is transmitted from one PE to the next stage PE properly, retiming and pipelining techniques are also incorporated to optimize the registers in the data path. The proposed top-level R2MDC architecture for an m-stage NTT-based polynomial multiplier is shown in Fig. 4.4, where each stage consists of one PE, one upper register set, one lower register set, and one switch. Based on the folding order described in Fig. 4.3, two data with the indices j and $j + n/2$, where $0 \leq j \leq n/2 - 1$, are loaded into the inputs in parallel. Two modular multipliers perform the weighted operations for the input data before executing the DIF-based butterfly operation. Then the intermediate results are stored in the upper/lower registers or fed into the PE in the next stage. The register sets and switch ensure the sequence of butterfly operations is the same as the folding order. In general, the number of registers (delay elements) τ for the k-th stage upper/lower register set is expressed as

$$\tau[k] = \begin{cases} \frac{n}{2^{k+2}}, & k \in [0, m-2], \\ \frac{n}{2}, & k = m-1. \end{cases} \tag{4.8}$$

Note that the proposed optimized folding transformation does not increase the total number of registers compared with the prior works [52–54]. Compared to prior works, our design 1) performs butterfly operations that are adjacent in the data-flow graph consecutively, instead of performing the butterfly operations in the natural order; 2) allocates the upper/lower registers set with $n/2$ delay elements at the last stage rather than the first stage as in prior works. These two properties enable all the PEs to be fully utilized during the computations. As opposed to prior works where the next NTT transform has to wait for the previous NTT transform to be completed, both the input and output nodes in the proposed architecture can have a continuous flow even between two

NTT transforms. Thus, we can achieve a high-throughput performance with two output samples per clock cycle. In this case, our proposed architecture can yield better overall timing performance for processing multiple consecutive operations. The total latency (i.e., first data in and last data out) in our design for L NTT transforms is expressed as

$$T_{Lat} = m \cdot T_{pp} + n/2 + n/2 \cdot L, \tag{4.9}$$

where T_{pp} is the clock cycles consumed by one PE. In our design, we add one pipelining cut-set in each PE, as shown in Fig. 4.1, in order to avoid a long critical path in a chain of modular adders.

4.3.3 Schemes for the Control and Addressing Units

Another advantage of the proposed pipelined architecture over memory-based architectures is a simple control unit. The control unit (denoted as "switch" in Fig. 4.4) is easily implemented as two MUXs, which propagates the signals to go to either the upper register set or the PE in the next stage. This 1-bit select signal $ctrl[k]$ is controlled by an m-bit global counter β. The control signals for the first and last stages are the $(m-2)$-th bit of β, while the other stages' control signals are connected to the $(m-2-k)$-th bit of β in k-th stage. The rationale behind this is to study the switch rate of the signals that are always in power-of-two, which is mapped to the global counter with the same changing rate in its bits.

We also propose an efficient addressing scheme for the twiddle factor α^{ij}. In our design, the number of twiddle factors that need to be stored is decreased from $n/2$ in the first stage to 2 in $(m-2)$-th stage. Note that no twiddle factor or modular multiplier is required in the last stage since the twiddle factor is always equal to 1.

Each addressing unit outputs an $(m-1-k)$-bit signal δ_k to select the twiddle factors, where the first $(m-2-k)$ bits of δ_k (i.e., $\delta_k[m-2-k:1]$) is generated from a counter with the same bit-length, and the last bit (i.e., $\delta_k[0]$) is derived from the following fashion. In k-th stage, if all bits in $\beta[m-3:m-3-k]$ are 1, $\delta_k[0] = \overline{\beta[m-2]}$ (i.e., the inverse of $\beta[m-2]$); otherwise, $\delta_k[0] = \beta[m-2]$.

4.4 Experimental Results

Our design is implemented using Verilog HDL and then mapped to the Xilinx Artix-7 series FPGA board. We select prior works that report results on the same FPGA board [48–50] for a fair comparison. We apply one of the most commonly used parameters ($n = 1024$ and $q = 12289$), which is also the same as in [49–51]. Under this parameter set, our design is instantiated as ten stages of PEs ($m = 10$). The experimental results, including the area performance (LUTs, FFs, DSPs, and BRAM) and timing performance (frequency and actual computational time in μs) for the NTT transforms, are presented in Table 4.1.

Table 4.1: Performance comparison of our proposed design and prior works

Design	Oder [48]	Xing [49]	Kuo [50]	**Ours**
LUTs	1390	4823	2832	5386
ATP (LUT)	$35.9(\times 10^5)$	$3.6(\times 10^5)$	$4.5(\times 10^5)$	$1.7(\times 10^5)$
FFs	615	2901	1381	2056
DSPs	2	8	8	11
ATP (DSP)	$51.6(\times 10^2)$	$6.0(\times 10^2)$	$12.6(\times 10^2)$	$3.4(\times 10^2)$
BRAM	1	–	10	0
Freq.[MHz]	125	153	150	167
1 NTT: cc (μs)	35845 (286.8)	1280 (8.4)	2616 (17.4)	1064 (6.3)
9 NTTs: cc (μs)	322605 (2580.8)	11520 (75.3)	23544 (157.0)	5160 (30.8)

Our proposed lightweight modular multiplier only consumes one DSP for the first multiplication between two inputs, while the second multiplication with a fixed input only uses the LUTs after synthesis. Overall, our NTT architecture consumes 5386 LUTs and only 11 DSPs, which can be operated at a 167 MHz frequency. For one NTT transform, our design reduces the computational time by 25.0%, 63.8% and 97.8% compared to the works in [49], [50] and [48], respectively. We also evaluate the performance of nine consecutive NTT transforms, as many of the LWE schemes involve matrix multiplication with a three-dimensional vector corresponding to nine polynomial multiplications. Due to the advantages of high-throughput and full utilization of the architecture, our design is even more superior when considering the actual time of computing nine NTT transforms, which achieves 59.1%, 80.4%, and 98.8% reductions, respectively, compared to the same prior works. When we take the hardware cost into account, our design still significantly outperforms prior works, as seen from the ATP in Table 4.1. Note that the ATP results are calculated by multiplying the number of LUTs or DSPs with the actual computation time for nine NTT transforms. It can be observed that our design has 52.8% (43.3%), 62.2% (72.9%), and 95.3% (93.4%) lower ATPs on LUTs (DSPs),

compared to these prior works.

4.5 Conclusion

This chapter proposed a novel pipelined high-throughput architecture to accelerate the NTT algorithm for lattice-based cryptography. We exploited the FFT MDC architecture and a lightweight Barrett reduction algorithm-based modular multiplier to optimize the efficiency of the proposed architecture. The effectiveness of the proposed design was verified by comprehensive experimental results.

Future work will be directed towards a high radix NTT algorithm to improve the efficiency of polynomial multiplication architecture.

Chapter 5

High-Speed Modular Multiplier for Lattice-Based Cryptosystems

Thanks to the inherent post-quantum resistant properties, lattice-based cryptography has gained increasing attention in various cryptographic applications recently. To facilitate the practical deployment, efficient hardware architectures are demanded to accelerate the operations and reduce the computational resources, especially for polynomial multiplication, which is the bottleneck of lattice-based cryptosystems. In this chapter, we present a novel high-speed modular multiplier architecture for polynomial multiplication. The proposed architecture employs a divide-and-conquer strategy and exploits a special modulus to increase the parallelism and speed up the calculation while enabling wider applications across various cryptosystems. The experimental results show that our work achieves around 27% and 39% reduction in the area consumption and delay, respectively, compared to prior designs.[1]

5.1 Introduction

Post-quantum cryptography (PQC) is a category of cryptographic algorithms that are believed to be able to protect sensitive information in the world of quantum computers. The National Institute of Standards and Technology (NIST) is currently in the process of standardizing PQC. More than half of the second round and three out of four of the third round PQC candidates are

[1] This work is presented in [11].

lattice-based schemes [60]. Meanwhile, lattice-based schemes are also prevalent in the field of homomorphic encryption (HE), whose goal is to enable secure function evaluation on encrypted data so that user privacy can be protected [5].

The lattice-based cryptography is built upon the NP-hard lattice problems that even quantum computers cannot solve efficiently. One example of the lattice problem is the shortest vector problem (SVP), whose security relies on the hardness of approximating SVP in the Euclidean norm [5]. Modular multiplication is a fundamental yet the most computationally-intensive operation in the polynomial multiplication of lattice-based cryptography. Therefore, improving the efficiency of modular multiplication, especially under the umbrella of resource-constrained platforms such as IoT and mobile devices, is critical to the practical deployment of lattice-based cryptosystems.

This chapter proposes an efficient and high-speed modular multiplier by innovatively utilizing the Karatsuba multiplication [61] to generate several partial products that can then be processed by a low-cost and efficient modular reduction unit. The main contributions of this chapter are summarized below:

- We utilize a divide-and-conquer strategy, and exploit a special modulus to increase the parallelism and speed up the calculation, while simultaneously reducing the hardware complexity.
- Our design shortens the bit-lengths of intermediate values by performing the modular reduction directly on the partial products. The largest bit-length among all the intermediate values is only $(3v + 2)$-bit for multiplication modulo a $2v$-bit prime.
- In contrast to some previous works that only support one specific prime [53, 62], our design can be reconfigured as various moduli (primes) with arbitrary bit-lengths, which is suitable for both PQC and HE schemes.

The rest of this chapter is organized as follows: Section 5.2 reviews the mathematical background and the prior works on the hardware implementation of modular multipliers for different lattice-based cryptographic algorithms. Section 5.3 introduces the details of our optimized modular multiplication algorithm. The proposed novel hardware architecture and its performance are presented and analyzed in Section 5.4. Finally, Section 5.5 concludes the chapter.

5.2 Background

5.2.1 Modular Multiplication

The operations in lattice-based cryptography in a ring such as $R_{n,Q} := \mathbb{Z}[x]/(x^n + 1, Q)$ [7] are computationally heavy, where n is a power of two. The multiplication modulo Q in a polynomial multiplication can be expressed as:

$$a \cdot b = t \pmod{Q}, \tag{5.1}$$

where $0 \leq a, b, t < Q$.

In the literature, various algorithms have been proposed to improve the efficiency of modular multiplication, including Montgomery reduction [63] and Barrett reduction [56].

The Montgomery reduction requires two $2v$-bit inputs a and b to multiply with a factor $r = 2^{2v}$. Then, the reduction can be achieved by

$$t = a \cdot b \cdot r^{-1}, \tag{5.2}$$

where r^{-1} is the inverse of r modulo Q. Barrett reduction also first computes the product t from the general integer multiplication of a and b. Different from Montgomery reduction, it reduces t back to the range $[0, Q-1]$ by

$$t = t - ((t \cdot m) >> 4v) \cdot Q, \tag{5.3}$$

where Q is the modulus and $m = \lfloor 2^{4v}/Q \rfloor$ [56].

5.2.2 Hardware Implementation for Modular Multiplication

Besides the theoretical improvement, many hardware optimization techniques for modular multiplication as well as polynomial multiplication have been developed recently [8, 9, 41, 64]. The majority of modular multipliers for HE and PQC are based on Barrett reduction [29, 62], as the primes are usually fixed in these applications. It has been shown that the Barrett reduction can be implemented by using multipliers and shifting operations only [29, 62]. Recently, a method called Shift-Add-Multiply-Subtract-Subtract (SAMS2) has been proposed [65], which uses the addition

and shifting instead of the expensive multiplication/division in Barrett reduction and hence leads to more efficient hardware implementation [53]. Hardware optimization methods based on Montgomery reduction have also been developed for both HE [28] and PQC digital signature [66] schemes.

However, for a $2v$-bit prime, most of these previous designs need to expand their results to $4v$-bit, followed by a modular reduction unit (i.e., use either multiplications or subtractions to reduce the results modulo Q). In contrast, our work shortens the bit-lengths in the intermediate values to improve efficiency.

5.3 Optimized Modular Karatsuba Multiplication

Different from the Barrett reduction or Montgomery reduction, our proposed design calculates several partial products in parallel and then performs reduction before merging them, which has the potential to accelerate the operation as well as reduce the area consumption of the hardware implementation. Besides, while the modular Karatsuba multiplication for other applications has been studied by the prior works [67, 68], to the best of our knowledge, our design is the first work that leverages the Karatsuba multiplication for the coefficient multiplication of the polynomial multiplication in the lattice-based cryptosystems.

5.3.1 Base B Decomposition

We first perform a base B decomposition to split the operands and then reduce the partial products. For positive integers Q, B, ℓ, if $Q < B^{\ell}$, then any integer $y \in [0, Q-1]$ can be expressed uniquely in base B as

$$y = y_0 + y_1 B + y_2 B^2 + \cdots + y_{\ell-1} B^{\ell-1} \tag{5.4}$$

where $0 \leq y_i < B$. If B is a power-of-two integer (i.e., $B = 2^v$), it is simple to convert from the binary representation into its base B decomposition. In this case, $y_i = [(i+1)v - 1, iv]$, where i is an integer and $0 \leq i \leq \ell - 1$.

5.3.2 Optimally Chosen Prime

NTT-based multiplication is advantageous, especially when the polynomial degree is large. An NTT-compatible prime Q must satisfy that $2n$ divides $(Q-1)$. To further optimize the modular

multiplication, we choose the prime to have a "sparse" representation, similar to the idea of Solinas prime in [67]. We use a power-of-two base B, and a prime Q that is less than B^2 in our algorithm. In particular, for integers v_1 and v_2, where $v_2 < v_1 \leq v - 2$, we consider the primes in these two forms:

$$Q = 2^{2v} - 2^{v_1} \pm 2^{v_2} + 1, \tag{5.5}$$

and

$$Q = 2^{2v} - 2^{v_1} + 1. \tag{5.6}$$

We refer to the forms in Equations (5.5) and (5.6) as *4-sparse* and *3-sparse* primes, respectively. The rationale behind using these primes are: 1) our technique represents the inputs as two base B digits based on the fact that $Q < 2^{2v} = B^2$. Thus we must subtract the power 2^{v_1}; 2) the condition of $v_1 \leq v - 2$ allows for further optimization in hardware implementation; 3) the least significant bit (LSB) of a prime has to be one. Otherwise, Q is even; 4) in contrast to the original Solinas prime [67], we add rather than subtract one so that $Q-1$ will be divisible by a power-of-two, becoming NTT-friendly; 5) finally, 2^{v_2} is added to enable more prime choices.

In order to support an NTT in rings with large n (e.g., the Ring Learning with Errors (RLWE) problem [69] with a strong security level), v_2 and v_1 need to be large in the cases of *4-sparse* and *3-sparse* primes, respectively.

For example, a 64-bit prime $Q = 2^{64} - 2^{24} + 1$ supports an NTT with n up to 23 bits. For other applications such as schoolbook polynomial multiplication in [70, 71], we may choose a prime with small v_1 and v_2 for better efficiency, e.g., a 12-bit prime $Q = 2^{12} - 2^3 + 2^1 + 1$.

5.3.3 Proposed Algorithm for the Modular Karatsuba Multiplication

For simplicity, we integrate the *4-sparse* and *3-sparse* cases together by representing Q as $2^{2v} - 2^{v_1} + e2^{v_2} + 1$ where $e \in \{0, \pm 1\}$. The only modification is that for the case of $e = -1$, the proof of Lemma 2 requires an additional assumption of $v_1 \leq v - 3$.

Algorithm 7 describes our optimized modular multiplication, which also takes the hardware implementation into consideration. Only three small multiplications are required, while the other operations are simple shifting and additions/subtractions since multiplication with a power-of-two integer can be realized as shifting. The corresponding data-flow chart is shown in Fig. 5.1.

We first represent a and b in base B as $a = a_0 + a_1 B$ and $b = b_0 + b_1 B$. Then, similar to

Algorithm 7 Optimized Modular Multiplication

Input: a, b and $Q = 2^{2v} - 2^{v_1} + e2^{v_2} + 1$
where $0 \leqslant a, b < Q < 2^{2v}$.
Output: $t = a \cdot b \mod Q$

1: Initialization: Let $B = 2^v$, $0 \leqslant a_0$, a_1, b_0, $b_1 < B$
$a = a_0 + a_1 B$ //split a as two parts
$b = b_0 + b_1 B$ // split b as two parts
2: $c_0 = a_0 b_0$ // v-bit multiplication
$c_1 = a_1 b_1$ // v-bit multiplication
$c_2 = (a_0 + a_1)(b_0 + b_1)$ // $(v+1)$-bit multiplication
3: $c = c_0 + (c_2 - (c_1 + c_0))B + c_1(2^{v_1} - e2^{v_2} - 1)$
4: $f_0 = c[2v - 1 : 0]$
$f_1 = c[3v - 1 : 2v]$
$f_2 = c[3v + 1 : 3v]$
5: $f = f_0 + f_1(2^{v_1} - e2^{v_2} - 1) + f_2(2^{v_1} - e2^{v_2} - 1)B$
6: **if** $f \geq Q$ **then**
7: $t = f - Q$
8: **else**
9: $t = f$
10: **end if**
11: **return** t

the original Karatsuba multiplication, we generate the partial products of $c_0 = a_0 b_0$, $c_1 = a_1 b_1$, and $c_2 = (a_0 + a_1)(b_0 + b_1)$ in Step 2. As a result, the product $a \cdot b = (a_0 + a_1 B)(b_0 + b_1 B)$ can be expressed in terms of these three partial products as $a \cdot b = c_0 + (c_2 - (c_1 + c_0))B + c_1 B^2$ [61]. As opposed

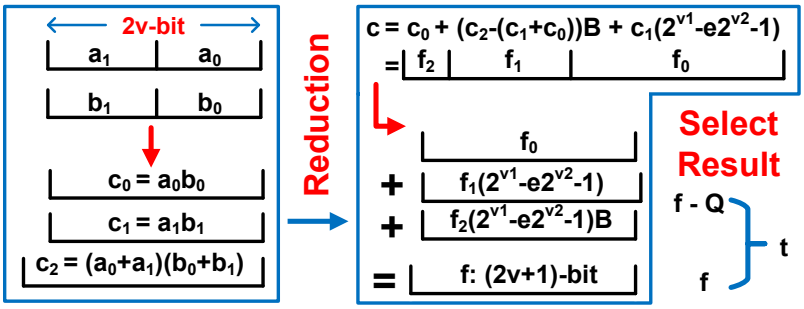

Figure 5.1: Data-flow of the optimized modular multiplication.

to multiplying c_1 by B^2 as in the original Karatsuba algorithm, we multiply it by $(2^{v_1} - e2^{v_2} - 1)$ since $B^2 = 2^{2v} \equiv 2^{v_1} - e2^{v_2} - 1 \pmod{Q}$. We note this intermediate value in Step 3 as

$$c := c_0 + (c_2 - (c_1 + c_0))B + c_1(2^{v_1} - e2^{v_2} - 1). \tag{5.7}$$

At this point, c is not guaranteed to be in the range $[0, Q - 1]$.

Lemma 1 *The intermediate value c obtained in Step 3 can be strictly upper-bounded by* 2^{3v+2}*, i.e.,*

$$c = c_0 + (c_2 - (c_1 + c_0))B + c_1(2^{v_1} - e2^{v_2} - 1) < 2^{3v+2}.$$

According to Lemma 1, our optimized parameter yields smaller bit-lengths of intermediate values than prior works in the reduction process. Hence, efficiency can be improved.

In Step 4, we split c in a base B decomposition as

$$c = f_0 + f_1 B^2 + f_2 B^3, \tag{5.8}$$

where $0 \leq f_0 < B^2$, $0 \leq f_1 < B$ and $0 \leq f_2 < 2^2$. It is important to note that f_2 is only 2-bit, according to the upper bound in Lemma 1. Thus, as opposed to a chain of multiplexers (i.e., several *if-else* condition statements) as the work in [53], we only need one multiplexer in Step 6.

Since $B^3 \equiv (2^{v_1} - e2^{v_2} - 1)2^v \pmod{Q}$, we reduce this modulo Q in Step 5 as

$$f = f_0 + f_1(2^{v_1} - e2^{v_2} - 1) + f_2(2^{v_1} - e2^{v_2} - 1)2^v. \tag{5.9}$$

Under the conditions as described in Lemma 2, we only need to check whether $f \geq Q$ is in Step 6. If this is the case, one subtraction by Q is enough to reduce it to the range $[0, Q-1]$.

Lemma 2 *Given* $Q = 2^{2v} - 2^{v_1} + e2^{v_2} + 1$*,*

1) if $v_2 < v_1 \leq v - 2$ *in the case that* $e = 0, 1$*,* f *is always less than* $2Q - 1$*. When* Q *is in 3-sparse form, the* v_2 *term is eliminated;*

2) if $v_2 < v_1 \leq v - 3$ *in the case that* $e = -1$*,* f *is always less than* $2Q - 1$*. When* Q *is in 3-sparse form, the* v_2 *term is eliminated.*

Note that the selection of our special primes also helps address the issue of overflow by setting the upper-bound of signals.

5.4 High-Speed Modular Multiplier

Our proposed high-speed modular multiplier mainly consists of two blocks: a semi-Karatsuba multiplier and a partial product reduction unit. The semi-Karatsuba multiplier is used to calculate

the partial products that will then be fed into the partial product reduction unit to generate the final result. Note that, as opposed to the pipelined modular multiplier in [66], these two blocks are operated at the same clock cycle in order to avoid any potential timing attack [62], but still remain a short critical path.

5.4.1 Semi-Karatsuba Multiplier

The architecture for Steps 1 and 2 in Algorithm 7 is shown in Fig. 5.2. As described in Section 5.3, two inputs a and b will first be split into a higher half (i.e., higher v bits) and a lower half (i.e., lower v bits). It then computes three v-bit multiplications in parallel, which is faster than a direct $2v$-bit multiplication. In the original Karatsuba multiplication algorithm, two v-bit and one

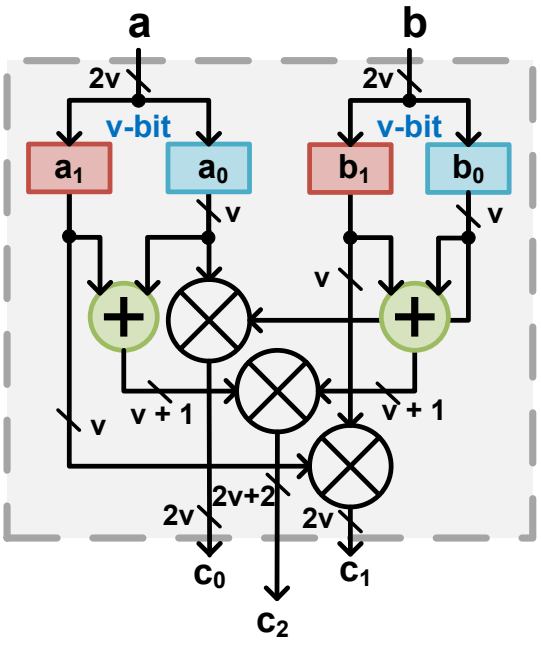

Figure 5.2: Semi-Karatsuba multiplier.

$(v+1)$-bit multipliers are used to generate three partial products c_0, c_1 and c_2, as shown in Step 2 of Algorithm 7. The addition of between a_0 and a_1 or between b_0 and b_1 may introduce an additional carry bit into the multiplication. Therefore, a $(v+1)$-bit multiplier is used to avoid overflow.

To better integrate modular reduction into the multiplication, we only use the split and multiplication steps from the original Karatsuba multiplication for the semi-Karatsuba multiplier, which will then connect its outputs c_0, c_1, and c_2 to the inputs of the partial product reduction unit.

5.4.2 Architecture for Partial Product Reduction Unit

The novel partial product reduction unit only consists of adders/subtractors and one multiplexer. In addition, this unit can be easily reconfigured for different moduli according to the choice of the prime form as defined in Equations (5.5) and (5.6) by changing the parameter of v_1, v_2 and v, which only requires to adjust the bit-length of adders/subtractors at the compile time.

Fig. 5.3 depicts the architecture for the prime $Q = 2^{2v} - 2^{v_1} \pm 2^{v_2} + 1$. For a *3-sparse* prime, the components in the blue dashed box can be eliminated, which saves three adders/subtractors. Besides, the operation in the form of $x(2^{v_1} \pm 2^{v_2})$ can be calculated as $(x2^d \pm x)2^{v_2}$, where $d = v_1 - v_2$. As a result, the adders/subtractor can be reduced from $(m(x) + v_1)$-bit to $(m(x) + d)$-bit, where $m(x)$ is the bit-length of x. If $d = 1$, this operation is eliminated.

At the beginning of this unit, two $2v$-bit and one $(2v + 2)$-bit output signals from the semi-Karatsuba multiplier are used to compute the value c based on Step 3 in Algorithm 7. After that, c is divided into f_0, f_1 and f_2 according to the base B decomposition. Then, the reduction process converts them to three new signals that are added into a $(2v+1)$-bit value f. Finally, one multiplexer is used to select either f or $f - Q$ as the final output t.

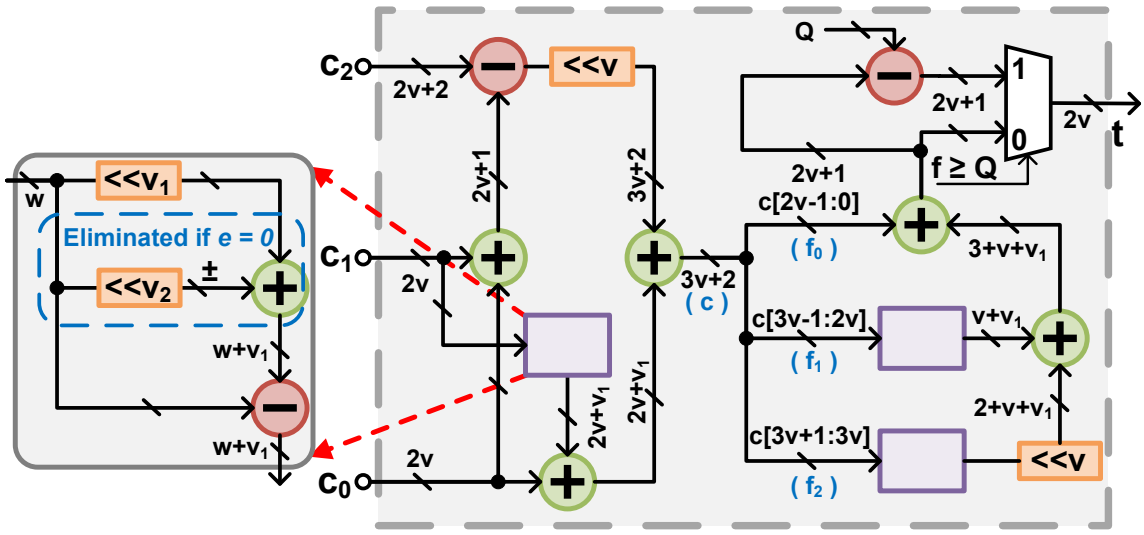

Figure 5.3: Architecture for partial product reduction unit.

5.4.3 Experimental Results

We implement our proposed design using Verilog HDL, which is then mapped to the Synopsys SAED 32nm technology node. For comparison, we also implement several recent designs [28, 29, 53, 62, 66]. These prior works are selected to include a wide range of modular multiplication methods used in different algorithm classes, including the Montgomery reduction, SAMS2, and the Barrett reduction. Besides, the comparison also includes designs employing the special primes in [28, 53, 62].

Experimental results of efficiencies and hardware complexities are presented in Table 5.1. In order to reduce the deviation from the optimization process by the electronic design automation (EDA) tools to achieve a fairer comparison, we map all the designs to the same gate-level arithmetic blocks, which is consistent with the work in [62].

Table 5.1: Comparisons of the proposed architecture with prior works

Designs	**Proposed**	Mert [28]	Riazi [29]	Wang [66]	**Proposed**	Mejía [53]	Banerjee [62]
Scheme	HE	HE	HE	PQC	PQC	PQC	PQC
Algorithm Class	**Karat.**	Mont.	Barrett	Mont.	**Karat.**	SAMS2	Barrett
Prime Bit-length	**32**	32	32	30	**14**	14	14
# Add. [a]	**12.09**	6.03	4	4	**13.5**	7.85	7
# Mul. [a]	**0.78**	1.67	2.06	3	**0.82**	2	3
Area [μm^2]	**16226**	17787	32967	17957	**3393**	4394	5574
# Gate	**10817**	11858	21978	11971	**2262**	2929	3716
Delay [ns]	**10.90**	15.22	23.23	13.23	**6.32**	11.85	11.34
Power [μW]	**768**	576	776	725	**163**	165	231
ADP [b]	**17.69**	27.07	76.58	23.76	**2.14**	5.2	6.32

[a] Normalized the additions/subtractions and multiplications to $2v$-bit (i.e., the bit-length of Q).
[b] ADP: ns $\times$ μm^2 $\times 10^4$

Since our design has high flexibility in various primes, we pick the most representative parameters for primes Q in the HE and PQC schemes (i.e., 32-bit and 14-bit primes) and configure the architectures by changing the bit-lengths for multipliers, adders, and subtractors. Note that, the modulus for most of the lattice-based cryptosystems is constant during the polynomial multiplication when the parameters are determined [60]. For example, we set 32-bit prime as $Q = 2^{32} - 2^{13} - 2^{12} + 1$ that can allow the degree of NTT-based polynomial multiplication to be $n = 2048$, which can mostly support the existing HE schemes to ensure a high-security level. Moreover, we use the *4-sparse* primes for our experiments, as shown in Table 5.1, to show a worse-case comparison. If a *3-sparse* prime is used, the performance of our proposed architecture will be further improved.

For better illustration, we normalize the cost of all the adders/subtractors and multipliers with respect to $2v$-bit (i.e., the bit-length of the prime). It can be seen from Table 5.1 that the proposed architecture has the lowest number of multipliers, while the designs in [29, 62, 66] employ the most multiplications among these designs. The architecture in [53] only has two multipliers but requires relatively large numbers of adders/subtractors.

Compared to the 32-bit prime, our design achieves a more significant reduction for the 14-bit prime in both area consumption and delay. Note that the pipelining technique is not applied in our evaluation in order to perform a fair comparison with prior works that are without the pipelining (e.g., [53, 62]). Even so, our design still remains a short critical path. Overall, the proposed design achieves 27.01% and 39.25% reductions on average for the area consumption and delay, respectively. By considering area and delay jointly, the area-delay product (ADP) of the proposed design is 54.10% less than the best among these prior works.

5.5 Conclusion

This chapter presented a novel high-speed modular multiplier along with optimized modular multiplication. The proposed architecture parallelizes partial product computation and modular reduction to improve efficiency. Comprehensive experimental results were provided to verify the effectiveness of the proposed design.

Proofs in This Chapter

Proof of Lemma 1: We have $c_2 = (a_0 + a_1)(b_0 + b_1)$, $c_1 = a_1b_1$, and $c_0 = a_0b_0$, so $c_2 - (c_1 + c_0)$ simplifies to $a_0b_1 + a_1b_0$. Since $a_0, a_1, b_0, b_1 \leq 2^v - 1$, all the products a_0b_0, a_1b_1, a_0b_1, and a_1b_0 are less than or equal to $(2^v - 1)^2$. Thus, we can first simplify and then express c as

$$\begin{aligned} c &= c_0 + (a_0b_1 + a_1b_0)B + c_1(2^{v_1} - e2^{v_2} - 1) \\ &\leq (2^v - 1)^2 + 2(2^v - 1)^2 2^v + (2^v - 1)^2(2^{v_1} - e2^{v_2} - 1) \\ &= (2^v - 1)^2[1 + 2^{v+1} + 2^{v_1} - e2^{v_2} - 1] \\ &= (2^v - 1)^2[2^{v+1} + 2^{v_1} - e2^{v_2}], \end{aligned}$$

since the terms contribute positively.

Since $v_1 < v$, we have $2^{v_1} \leq 2^{v+1}$. As $(2^v - 1)^2 < 2^{2v}$, and in the case of $e = 0, 1$, we drop the term $-e2^{v_2}$ and upper bound of c as

$$\begin{aligned} c &< (2^v - 1)^2[2^{v+1} + 2^v] \\ &< 2^{2v}(2^{v+2}) = 2^{3v+2}. \end{aligned}$$

In the case $e = -1$, we use $v_2 < v_1 \leq v - 3$ to upper bound c as

$$\begin{aligned} c &< 2^{2v}(2^{v+1} + 2^{v-3} + 2^{v-4}) \\ &\leq 2^{2v}(2^{v+1} + 2^{v+1}) = 2^{3v+2}. \qquad \square \end{aligned}$$

Proof of Lemma 2: We have $f_0 \leq 2^{2v} - 1$, $f_1 \leq 2^v - 1$ and $f_2 \leq 2^2 - 1$ from Lemma 1. Thus, we can define f as

$$\begin{aligned} f &= f_0 + f_1(2^{v_1} - e2^{v_2} - 1) + f_2 2^v(2^{v_1} - e2^{v_2} - 1) \\ &\leq 2^{2v} - 1 + (2^v - 1 + 3 \cdot 2^v)(2^{v_1} - e2^{v_2} - 1) \end{aligned}$$

We drop some negative terms,

$$\begin{aligned} f &\leq 2^{2v} - 1 + (2^{v_1} - e2^{v_2} - 1)[2^{v+2} - 1] \\ &< 2^{2v} + (2^{v_1} - e2^{v_2} - 1)2^{v+2}. \end{aligned}$$

Then in the case $e = 0, 1$ we can drop the term -2^{v_2} and use the assumption that $v_1 \leq v - 2$ to show $2^{v_1} \leq 2^{v-2}$, so the upper-bound of f is

$$\begin{aligned} f &< 2^{2v} + (2^{v-2} - 1)2^{v+2} \\ &= 2^{2v+1} - 2^{v+2} \\ &< 2Q - 2. \end{aligned}$$

In the case $e = -1$, we use $v_2 < v_1 \leq v - 3$ to upper bound f as

$$\begin{aligned} f &< 2^{2v} + (2^{v_1} + 2^{v_2} - 1)2^{v+2} \\ &\leq 2^{2v} + (2^{v-3} + 2^{v-4} - 1)2^{v+2} \\ &= 2^{2v} + (2^{2v-1} + 2^{2v-2} - 2^{v+2}) \\ &< 2^{2v} + 2^{2v} - 2^{v+2} \\ &= 2^{2v+1} - 2^{v+2} \\ &< 2Q - 2. \qquad \square \end{aligned}$$

Chapter 6

High-Speed VLSI Architectures for Modular Polynomial Multiplication via Fast Filtering and Applications to Lattice-Based Cryptography

This chapter presents a low-latency hardware accelerator for modular polynomial multiplication for lattice-based post-quantum cryptography and homomorphic encryption applications. The proposed novel modular polynomial multiplier exploits the fast finite impulse response (FIR) filter architecture to reduce the computational complexity of the schoolbook modular polynomial multiplication. We also extend this structure to fast M-parallel architectures while achieving low-latency, high-speed, and full hardware utilization. We comprehensively evaluate the performance of the proposed architectures under various polynomial settings as well as in the Saber scheme for post-quantum cryptography as a case study. The experimental results show that our design reduces the computational time and area-time product by 61% and 32%, respectively, compared to the state-of-the-art designs. [1]

[1] This work is presented in [12].

6.1 Introduction

Modular polynomial multiplication is commonly used in lattice-based post-quantum cryptography (PQC) and homomorphic encryption applications. While homomorphic encryption aims at allowing computations to be directly carried out in the encrypted domain without decryption [5], lattice-based cryptographic algorithms are also designed to be resistant against attacks from both traditional and quantum computers and are thus well suited for PQC. Three out of the four finalists for the ongoing NIST PQC standardization in round-3 fall into the category of lattice-based cryptography [60]. In prior works, the modular polynomial multiplication for the lattice-based cryptography scheme has mostly been implemented by schoolbook polynomial multiplication [72], number theoretic transform (NTT) [73] or the Karatsuba multiplication [74]. Different from the prior works, this chapter proposes *novel* high-speed architectures by exploiting the fast finite impulse response (FIR) parallel filter architecture [57, 75–77]. The chapter also proposes a novel *weight-stationary* systolic array for modular polynomial multiplication; these are used as building blocks for the fast parallel architecture. The proposed architecture is feed-forward and can be pipelined at arbitrary levels to achieve the desired speed. To the best of our knowledge, this is the first work to utilize the fast parallel filter architecture to accelerate the modular polynomial multiplication for lattice-based schemes.

Exploiting the fast parallel filter approach to modular polynomial multiplication is neither straightforward nor trivial. Since the fast parallel filters contain several subfilters and merging operations, the modular operations must be incorporated at the subfilter level and at the merging level. No prior work has addressed these design aspects. The subfilters should correspond to single-input single-output architectures that should integrate the modular operation and should operate in real-time with no hardware under-utilization. These should also require simpler control circuits. Such designs have not been presented before.

The contributions of this chapter are four-fold. First, using systolic mapping methodology [57, 78–80], we derive a sequential weight-stationary systolic array for modular polynomial operation. This structure is *partly* similar to the *transpose-form* FIR digital filter [57] and is the main building block of the proposed architecture. The low-latency systolic array achieves full hardware utilization. Second, we propose a low-latency fast modular polynomial multiplication architecture that integrates the modular reduction at the merging level, achieves full hardware utilization and

minimizes latency. Third, using *iterated* fast parallel filter design approach, we propose highly parallel architectures where the level of parallelism is the product of short lengths. The modular operation is also carried out at the merging step of each iteration to reduce overall latency and achieve full hardware utilization. Fourth, the advantages of the proposed architecture are demonstrated using the Saber scheme as a PQC benchmark.

The rest of the chapter is organized as follows: Section 6.2 reviews the mathematical background and the prior works on modular polynomial multiplication. Section 6.3 and Section 6.4 present the details of the proposed hardware architecture, including the modular polynomial multiplier and fast M-parallel architecture. Section 6.5 describes the experimental results and comparisons with the state-of-the-art designs. Finally, Section 6.6 concludes the chapter.

6.2 Background and Related Work

Since this chapter targets the schoolbook polynomial multiplication, we briefly review the essential notations, mathematical background, and related works of the schoolbook polynomial multiplication.

6.2.1 Lattice-based Cryptography

Lattice-based cryptography relies on the NP-hard lattice problems that even quantum computers cannot solve efficiently. One example of the lattice problems is the shortest vector problem (SVP), whose security relies on the hardness of approximating SVP in the Euclidean norm [5].

There are several representative schemes for homomorphic encryption and PQC based on the lattice-based cryptography primitive. For example, in lattice-based homomorphic encryption, BFV scheme [16] and CKKS scheme [39] support a limited number of homomorphic computations, which are also categorized as somewhat homomorphic encryption (SHE) schemes. The fully homomorphic encryption (FHE) schemes such as [5, 17] allow an unlimited number of homomorphic computations by using the bootstrapping algorithm.

On the other hand, the NIST finalist lattice-based PQC schemes can be classified as either NTRU-based (e.g., NTRU [81]) and learning with errors-based (LWE) (e.g., Crystals-Kyber [20] and Saber [82]) in general.

In this chapter, we evaluate and compare the performance of our proposed architectures used

in the Saber scheme [82] as a case study. Saber is indistinguishability under chosen-ciphertext attack secure Key Encapsulation Mechanism (KEM), which consists of three algorithms: key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps), which are outlined in Algorithms 8, 9, and 10, respectively [82].

For the key generation in Algorithm 8, the pseudo-random matrix $\mathbf{A}$ is first constructed from the SHAKE-128 function [83] using the uniform random seed $seed_A$ and the secret key s that is sampled from the centered binomial distribution β_μ. Then, vector $\mathbf{b}$ is generated by the matrix and vector multiplication along with the rounding operation and becomes the public key (pk) by grouping with the seed $seed_{\mathbf{A}}$. Note that distinct from other LWE-based PQC schemes, Saber does not require adding error from the random distribution since it is replaced by the rounding operation.

The encryption process encrypts a 256-bit message $m(x) \in R_2$ to generate a ciphertext c, as described in Algorithm 9. The first five steps are similar to the key generation algorithm with an additional inner product between vectors $\mathbf{b}^T$ and ($\mathbf{s}' \mod p$), which will generate the polynomial $v'(x)$. The polynomial is then used to produce the elements of ciphertext $c_m(x)$. Finally, the ciphertext $c = (c_m(x), \mathbf{b}')$ will be returned and used for decryption.

During the decryption process, as illustrated in Algorithm 10, the polynomial $v(x)$ is computed from the inner product between one of the elements of ciphertext ($\mathbf{b}'$) and the secret key $\mathbf{s}$ at first. Then, the decrypted message $m'(x)$ can be obtained using simple shift and addition operations based on $v(x)$.

Algorithm 8 Saber.PKE.KeyGen()

1: $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $\mathbf{A} = \text{gen}(seed_{\mathbf{A}}) \in R_q^{l \times l}$
3: $r = \mathcal{U}(\{0,1\}^{256})$
4: $\mathbf{s} = \beta_\mu(R_q^{l \times 1}; r)$
5: $\mathbf{b} = (\mathbf{A}^T\mathbf{s} + \mathbf{h} \mod q) >> (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6: **return** $pk := (seed_{\mathbf{A}}, \mathbf{b}), sk := \mathbf{s}$

Algorithm 9 Saber.PKE.Enc($pk := (seed_{\mathbf{A}}, \mathbf{b})$, $m \in R_2; r$)

1: $\mathbf{A} = gen(seed_{\mathbf{A}}) \in R_q^{l \times l}$
2: $\mathbf{s}' = \beta_\mu(R_q^{l \times l}; r)$
3: $\mathbf{b}' = (\mathbf{A}\mathbf{s}' + \mathbf{h} \mod q) >> (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
4: $v'(x) = \mathbf{b}^T(\mathbf{s}' \mod p) \in R_p$
5: $c_m(x) = (v'(x) + h_1(x) - 2^{\epsilon_p - 1}m(x) \mod p) >> (\epsilon_p - \epsilon_T) \in R_T$
6: **return** $c := (c_m(x), \mathbf{b}')$

Algorithm 10 Saber.PKE.Dec(s, $c = (c_m(x), \mathbf{b}')$

1: $v(x) = \mathbf{b}^T(\mathbf{s} \mod p) \in R_p$
2: $m'(x) = ((v(x) - 2^{\epsilon_p - \epsilon_T} c_m(x) + h_2(x)) \mod p) >> (\epsilon_p - 1) \in R_2$
3: **return** $m'(x)$

More specifically, the primitive of the Saber scheme is based on the hardness of the module-learning with rounding (M-LWR) problem and the use of the Fujisaki-Okamoto transform [84]. Besides, IND-CCA KEM (i.e., Saber.KEM) can be constructed based on the Saber.PKE, which requires more hash functions operated on the public key, message, and ciphertext to ensure higher security. More details are explained in [82]. In the Saber scheme, the polynomial degree n is always fixed as 256. The lattice dimension relies on $l \cdot n$, where l can be chosen as 2/3/4, which correspond to three different security levels, i.e., LightSaber (lightweight), Saber (Standard), and FireSaber (high security), where Table 6.1 summarizes the number of polynomial multiplications for the Saber scheme in different security levels. The bit-lengths ϵ of the moduli q and p are given as $\epsilon_q = 13$ and $\epsilon_p = 10$, respectively, while the bit-length of modulus T is varied across the three different security levels, i.e., $\epsilon_T = 3/4/5$.

Table 6.1: The number of polynomial multiplication for different security levels for Saber.

Security Level	KeyGen	Encapsulation	Decapsulation
l	l^2	$l^2 + l$	$l^2 + 2l$
Lightweight ($l = 2$)	4	6	8
Standard ($l = 3$)	9	12	15
Fire ($l = 4$)	16	20	24

Among all the steps in the Saber scheme, the most widely used functions are the matrix-vector multiplication and inner product of two vectors i.e., degree-256 modular polynomial multiplication. For the medium-security level of Saber (post-quantum security level similar to AES-192), there are 9, 12, and 15 polynomial multiplications in the key generation, encapsulation, and decapsulation, respectively. In this chapter, we only consider the medium-security level.

Modular polynomial multiplication is a fundamental and yet the most computationally intensive operation of lattice-based cryptography. For the lattice-based PQC, modular polynomial multiplication dominates the computations across key-generation, encryption, and decryption steps in the prior works [72, 85]. Similarly, the most expensive operation for homomorphic encryption schemes is also modular polynomial multiplication. Therefore, improving the efficiency of modular polynomial multiplication is critical to the practical deployment of lattice-based PQC schemes and

homomorphic encryption.

6.2.2 Schoolbook Modular Polynomial Multiplication

For the product $P(x)$ of two polynomials

$$A(x) = a[0] + a[1]x + a[2]x^2 + ... + a[n-1]x^{n-1}, \tag{6.1}$$

$$B(x) = b[0] + b[1]x + b[2]x^2 + ... + b[n-1]x^{n-1}, \tag{6.2}$$

over R_q, all the coefficients of $P(x)$ need to be less than q but non-negative integers, while the degree of $P(x)$ should be less than n, where $R_q = \mathbb{Z}_q/(x^n+1)$ is ring of the polynomial, and $\mathbb{Z}_q$ is the ring of integers modulo a integer q. The schoolbook modular polynomial multiplication between $A(x)$ and $B(x)$ modulo (x^n+1, q) can be described as

$$\begin{aligned} & A(x) \cdot B(x) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a[i]b[j]x^{i+j} \bmod (x^n+1, q) \\ &= \sum_{i=0}^{n-1} \Big(\sum_{j=0}^{n-1} (-1)^{\lfloor (i+j)/n \rfloor} a[i]b[j] \bmod q \Big) \cdot x^{(i+j) \bmod n}. \end{aligned} \tag{6.3}$$

For the schoolbook modular polynomial multiplication, the moduli are not required to be prime, which is different from the NTT-based polynomial multiplication. Consequently, the polynomial multiplication used in the M-LWR problem [86] and ring-learning with errors (R-LWE) problem [19] can benefit from these moduli. In these cases, since all the moduli can be selected as power-of-two integers, the modular reduction for the coefficients on the schoolbook polynomial multiplication can be simply performed by keeping the least significant ϵ bits (ϵ is the bit-length of the modulus q, i.e., $\epsilon = \lceil \log_2(q) \rceil$) instead of using the expensive Barrett reduction [56] or Montgomery modular multiplication [63]. Meanwhile, schemes based on the M-LWR problem (such as the Saber scheme) obtain the error term by rounding, while naturally aligning with the power-of-two modulus. Besides, recent work shows that a power-of-two modulus can simplify and improve the polynomial multiplication for the R-LWE-based homomorphic encryption schemes without affecting the computational hardness [87]. The modulus in this format has been applied in some popular schemes such as BFV scheme [16]. Based on this advantage, shortening the word-length of the operand

and eliminating the modular reduction for the coefficients can increase the resource available which can then enable the designer to increase the level of parallelism to achieve a high-speed modular polynomial multiplier.

It may be noted that the use of the power-of-two moduli, as needed in the Saber scheme, cannot leverage the acceleration from the NTT-based polynomial multiplication without further expensive transformation. However, NTT-based polynomial multiplication has been widely applied in many lattice-based cryptography schemes when the moduli are not power-of-two [10, 27, 62, 73, 88–91].

6.2.3 Karatsuba Polynomial Multiplication

To improve the efficiency and reduce the complexity of schoolbook polynomial multiplication, methods based on the divide-and-conquer strategy to increase parallelism are of great interest. One of the examples is the Karatsuba algorithm [61], which has been utilized in some prior modular polynomial multiplier designs for Saber scheme [85, 92]. The 2-level Karatsuba polynomial multiplication first decomposes the input polynomials into higher-degree and lower-degree parts as $A(x) = A_0(x) + A_1(x) \cdot x^{n/2}$ and $B(x) = B_0(x) + B_1(x) \cdot x^{n/2}$ and computes

$$\begin{aligned} C_0(x) &= A_0(x) \cdot B_0(x) \\ C_1(x) &= (A_0(x) + A_1(x)) \cdot (B_0(x) + B_1(x)) \\ C_2(x) &= A_1(x) \cdot B_1(x). \end{aligned} \tag{6.4}$$

Then the above products are summed up and polynomial modular reduction is carried out to derive the product $P(x)$ over the ring as

$$P(x) = C_0(x) + C_3(x) \cdot x^{n/2} + C_2(x) \cdot x^n \mod (x^n + 1), \tag{6.5}$$

where

$$C_3(x) = (C_1(x) - C_0(x) - C_2(x)). \tag{6.6}$$

Note that the degrees of $C_3(x) \cdot x^{n/2}$ and $C_2(x) \cdot x^n$ are $\frac{3}{2}n$ and $2n$, respectively. Hence polynomial

subtractions are needed to perform the modular reduction by $x^n + 1$. Based on this divide-and-conquer strategy of the Karatsuba algorithm, the number of coefficient multiplications is reduced from n^2 to $3(n/2)^2$.

6.2.4 Prior Hardware Implementations

Several hardware accelerators for lattice-based cryptography without using the NTT algorithm have been proposed recently [70, 72, 85, 92–96]. As expected, optimizing the polynomial multiplier is the main focus of these works, since it is the bottleneck. The hardware/software co-design for the modular polynomial multiplication accelerator in [93] shows a significant acceleration compared with the software implementation. Subsequently, the work in [85] introduced the compact hardware/software interfacing design, which applies a hybrid method of Toom-Cook multiplication [97] (a generalized form of Karatsuba algorithm) and a degree-64 schoolbook polynomial multiplier to optimize the modular polynomial multiplication. A full hardware implementation is proposed in [72], which utilizes a memory-based schoolbook polynomial multiplier. This design achieves a higher speed where each degree-256 polynomial multiplication only consumes 256 clock cycles. Later, an extended work of [72] is presented in [98], which is based on a design called centralized multiplier architecture. This optimized design retains the same timing performance but requires fewer hardware resources since each multiply-and-accumulate (MAC) is replaced by one multiplexer (MUX) and one adder. Furthermore, an 8-level recursive split hierarchical Karatsuba algorithm-based implementation is introduced in [92], which reduces a degree-256 polynomial multiplication to only 81 clock cycles without considering the pipelining startup time. Besides, several architectures of modular polynomial multipliers for the R-LWE schemes are introduced in [71, 74, 95]. The works in [71, 95] investigate the low-area design for the schoolbook modular polynomial multiplication, which only consumes a small amount of LUTs and DSPs. Meanwhile, the design in [74] proposes a modular polynomial multiplier using the Karatsuba algorithm and reduces the complexity by merging the polynomial modular reduction on the post-processing stage of the Karatsuba algorithm.

However, these designs cannot consider an architecture using the fast filtering technique to reduce the latency. Also, the architectures based on the Karatsuba algorithm generally consider the polynomial modular reduction after the multiplication. These designs do not reduce the number of addition operations. Therefore, it is possible to further reduce the number of additions/subtractions at the post-processing stage thereby reducing the total number of addition/subtraction operations.

Since our objective is to improve the speed under a given hardware budget, we define the following two metrics in evaluating the performance from the speed perspective:

- Response time: clock cycles between the first input and the first output sample.
- Total latency: clock cycles between the first input and the last output sample.

6.3 Modular Polynomial Multiplier Based on Weight-Stationary Systolic Array

Consider the design of a degree-n modular polynomial multiplier described by Equation 6.3. In this section, we use $n = 4$ as an example to illustrate our proposed novel modular polynomial multiplier. The modular polynomial multiplication is described by:

$$\begin{aligned} P(x) &= A(x) \cdot B(x) \mod (x^4 + 1, q) \\ &= p[0] + p[1]x + p[2]x^2 + p[3]x^3, \end{aligned} \tag{6.7}$$

where

$$\begin{aligned} A(x) &= a[0] + a[1]x + a[2]x^2 + a[3]x^3, \\ B(x) &= b[0] + b[1]x + b[2]x^2 + b[3]x^3. \end{aligned}$$

The polynomial multiplication of $A(x)$ and $B(x)$ leads to

$$\begin{aligned} P'(x) = {} & p'[0] + p'[1]x + p'[2]x^2 + p'[3]x^3 \\ & + p'[4]x^4 + p'[5]x^5 + p'[6]x^6. \end{aligned} \tag{6.8}$$

Since the polynomial multiplication has a degree higher than three, the terms x^4, x^5, and x^6 are replaced by -1, $-x$, and $-x^2$, respectively, to perform the modular reduction. Thus, the

coefficients of the modular polynomial multiplication are:

$$
\begin{aligned}
p[3] &= a[3]b[0] + a[2]b[1] + a[1]b[2] + a[0]b[3], \\
p[2] &= a[2]b[0] + a[1]b[1] + a[0]b[2] - a[3]b[3], \\
p[1] &= a[1]b[0] + a[0]b[1] - a[3]b[2] - a[2]b[3], \\
p[0] &= a[0]b[0] - a[3]b[1] - a[2]b[2] - a[1]b[3]. \qquad (6.9)
\end{aligned}
$$

A dependence graph (DG) of the modular polynomial multiplication for the $n = 4$ example is shown in Fig. 6.1.

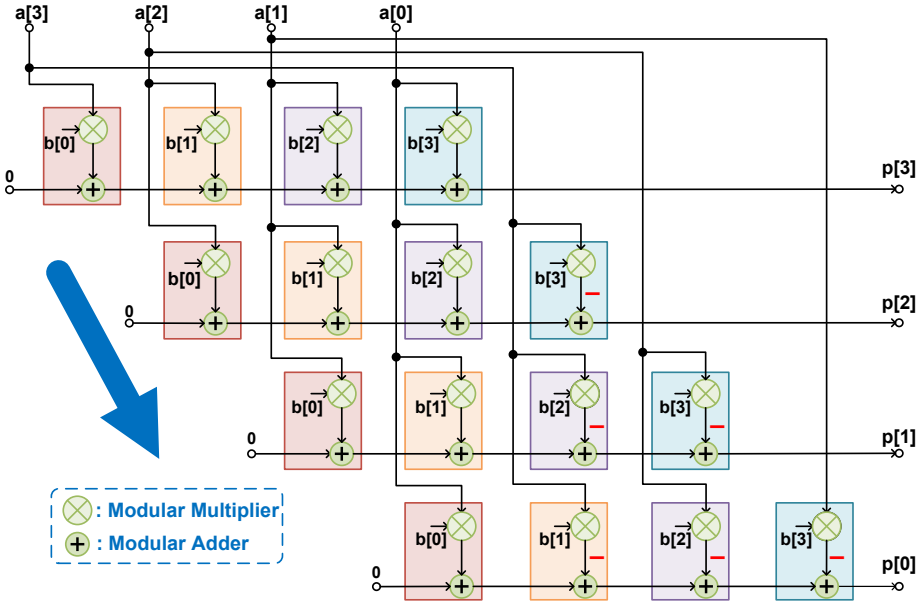

Figure 6.1: DG of the modular polynomial multiplication when $n = 4$. The DG is mapped to a systolic array using the projection vector shown in blue.

6.3.1 Architecture of Modular Polynomial Multiplier Using Transpose-Form FIR Filter

Given the similarity between modular polynomial multiplication and FIR filter, it is useful to consider three common types of FIR filter structures [57], i.e., direct-form, transpose-form, and hybrid-form, respectively, as shown in Fig. 6.2, Fig. 6.3, Fig. 6.4.

FIR filter is one of the digital filters that is used to modify the frequency properties of the

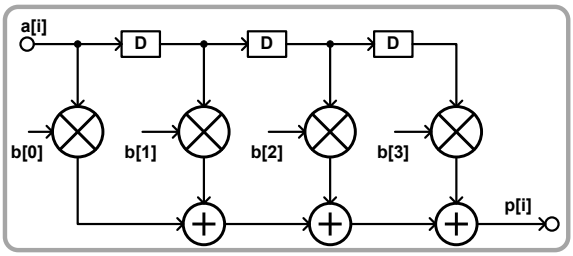

Figure 6.2: Direct-form FIR filter architecture when $n = 4$.

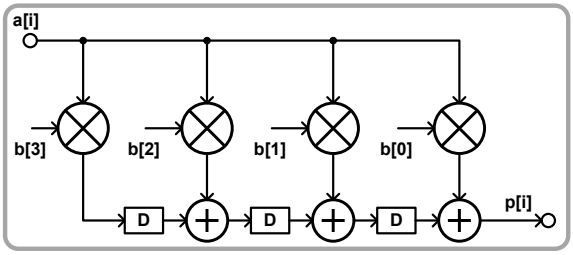

Figure 6.3: Transpose-form FIR filter architecture when $n = 4$.

input signal to achieve specific design requirements [57]. It can also be mathematically expressed as a discrete convolution of two signals, which can be defined as

$$p[n] = \sum_{j=0}^{n-1} b[j]a[n-j], \tag{6.10}$$

where n is the number of taps, $a[n]$ is the input signal, $b[j]$ is value of the impulse response at the j-th instant ($j \in [0, n-1]$), and $p[n]$ is the output signal. Though using any of the FIR filter structures in Fig. 6.2, Fig. 6.3, Fig. 6.4. can sufficiently instantiate Equation 6.10 and show a negligible difference in the overall performance in most of the digital signal processing applications, the FIR structure for modular polynomial multiplication needs to be carefully selected.

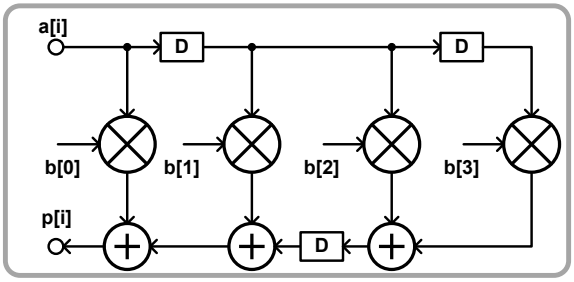

Figure 6.4: Hybrid-form FIR filter architecture when $n = 4$.

The direct-form FIR filter is shown in Fig. 6.2. leads to a long critical path, which consists of one multiplier and n adders. It can also be observed from Fig. 6.4 that the hybrid-form architecture generates its first output immediately after loading the first input, and requires additional registers to store the intermediate results; however, the architecture is not feed-forward and has a slightly longer critical path than the transpose-form. Thus, the best choice for implementing polynomial multiplication in lattice-based schemes is the transpose-form as shown in Fig. 6.3 as it has the least critical path and a feed-forward datapath. Fortunately, the DG in Fig. 6.1 can be mapped to a weight-stationary systolic array using the projection vector shown in blue in the figure. Alternatively, the systolic array can also be derived using the folding algorithm [58]. Note that all multiplications with coefficient $b[j]$ are mapped to the same hardware multiplier.

Fig. 6.5 shows an example systolic architecture for modular polynomial multiplication for a degree-4 where the components in each tap (node) are illustrated in Fig. 6.6. The systolic array contains additional switches and a shift register of size-n (see the top of Fig. 6.5) for continuous processing of input polynomials and polynomial modular reduction. Note that using a conventional transpose-form-like structure to perform the polynomial multiplication would require padding zeros until the entire operation finishes; otherwise, it will lead to conflicts and produce wrong results. Furthermore, to perform polynomial modular reduction, the shift register as well as the switches can control the signals (coefficients of polynomial $A(x)$) properly based on the expression in Equation 6.9.

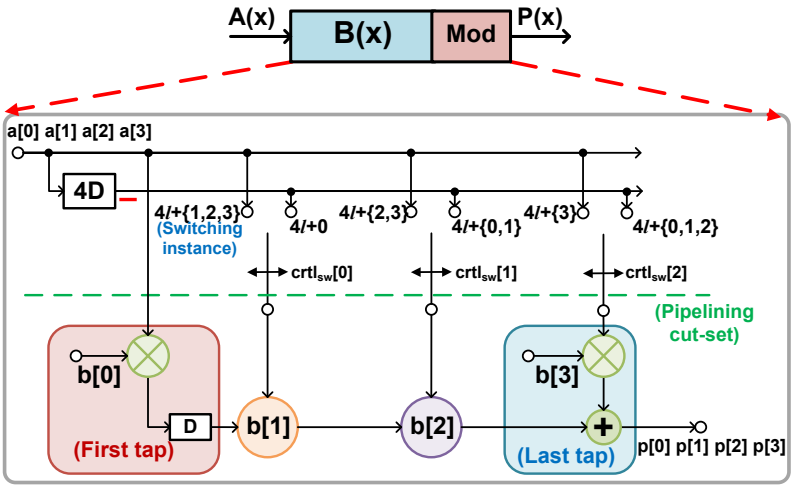

Figure 6.5: Top-level architecture of degree-4 weight-stationary systolic modular polynomial multiplier.

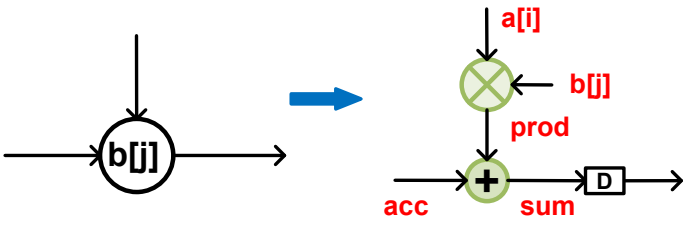

Figure 6.6: Details of each tap in the architecture of degree-4 weight-stationary systolic modular polynomial multiplier.

Specifically, the coefficients of polynomial $A(x)$ with the negative signs are extracted from the shift register in its negative form. Then, the switches select either the negative form or the original form coefficients from polynomial $A(x)$ in different clock cycles. As shown in Fig. 6.5, the proposed degree-4 modular polynomial multiplier consists of four modular multipliers, three modular adders, three delay elements, three switches, and one shift register. Specifically, the shift register consists of four delay elements, and the switches are constructed using multiplexers (MUXs). The design in Fig. 6.5 can be easily extended to degree-n. A degree-n modular polynomial multiplier requires n modular multipliers, $(n-1)$ modular adders, $(n-1)$ delay elements, $(n-1)$ switches at the lower data paths, and one shift register (consists of n delay elements). For one modular polynomial multiplication, the response time is n clock cycles, while the total latency is $(2n-1)$ clock cycles. For L polynomial multiplications, the response time remains the same, while the total latency in clock cycles is given by:

$$T_{lat} = n \cdot (L+1) - 1. \tag{6.11}$$

This architecture also has a full hardware utilization after the first output is computed. Hardware utilization is the percentage of the components inside this circuit that are performing useful operations, and full hardware utilization means no component is performing null operations.

The modular reduction can be performed by simply keeping the least ϵ bits for a 2^{ϵ} modulus. For the lattice-based cryptography schemes, the degrees of the polynomial are relatively large, i.e., n can be up to hundreds or thousands, which could cause a high fan-out issue on the output of the shift register and the input node. To overcome this, buffers (registers) are inserted after the switches, as shown in the green dashed line in Fig. 6.5. As a result, the critical path is one modular multiplier and one modular adder.

6.3.2 Scheduling for the Modular Polynomial Multiplier

The scheduling and control logic in the proposed architecture are very simple and efficient. The coefficients of polynomial $A(x)$ are loaded sequentially from the most significant (highest degree) coefficient to the least significant (lowest degree) coefficient, while the coefficients of polynomial $B(x)$ are stored starting with the least significant coefficient to the most significant coefficient from left to right. Finally, the result coefficients are output in the same order as $A(x)$ (i.e., from the most significant coefficient to the least significant coefficient).

The notation $4l+\{0,1,2,3\}$ represents the switch instances with a switch period of 4 clock cycles. Hence, l can be interpreted as the l-th period (iteration). For example, the left node will be connected when the switch instances are $4l+$ {1, 2, and 3}, while the right node will be connected at switch instance of $4l+0$. Each switch is controlled by a one-bit signal from the $(n-1)$-bit controller $ctrl_{sw}$: if this bit is equal to 1, the operand from polynomial $A(x)$ of the modular multiplier is loaded from the input node; otherwise, it is loaded from the shift register. These control signals $ctrl_{sw}$ can be simply generated by a counter (ranging from 0 to $(n-1)$), as:

$$ctrl_{sw} = \begin{cases} \{0, ..., 0, 0\}, & \text{if counter} = 0, \\ \{ctrl_{sw}[n-2:1], 1\}, & \text{otherwise.} \end{cases} \tag{6.12}$$

After resetting the counter, all $(n-1)$-bit control signals $ctrl_{sw}$ are zeros. Then, in every subsequent clock cycle, $ctrl_{sw}$ shifts left by padding a "1" to the least significant bit (LSB).

6.4 Fast Polynomial Multiplier Using Fast M-Parallel Filter Architecture

In this section, we derive a highly parallel hardware architecture for the polynomial multiplication based on the fast parallel filter algorithm [57, 75–77].

The proposed design requires less resource overhead than prior Karatsuba-based polynomial multipliers in the post-processing stage. Parallel structures for modular polynomial multiplication for small lengths are first derived. These can then be *iterated* to obtain architectures for larger levels of parallelism. For example, a fast 2-parallel (i.e., $M = 2$) modular polynomial multiplier can be iterated twice (or thrice) to design a 4-parallel (or 8-parallel) multiplier.

6.4.1 Fast 2-parallel Architecture

The fast 2-parallel modular polynomial multiplication, referred to as Fast.2.PolyMult, is described in Algorithm 11, which mainly consists of three stages: pre-processing (Step 1), intermediate polynomial multiplication (Step 2), and post-processing (Steps 3 and 4).

Algorithm 11 Fast.2.PolyMult($A(x), B(x)$)

Input: $A(x)$ and $B(x) \in R_q$
Output: $P(x) = (P_0(x^2), P_1(x^2))$
$//P(x) = A(x) \cdot B(x) \mod (x^n + 1, q)$

1: $A(x) = A_0(x^2) + A_1(x^2) \cdot x$
$//$split $A(x)$ as two parts based odd and even indices
$B(x) = B_0(x^2) + B_1(x^2) \cdot x$
$//$split $B(x)$ as two parts based odd and even indices
2: $U(y) = A_0(y)B_0(y) \mod (y^{n/2} + 1, q)$, where $y = x^2$
$//$ intermediate polynomial multiplication
$V(y) = A_1(y)B_1(y) \mod (y^{n/2} + 1, q)$
$W(y) = (A_0(y) + A_1(y))(B_0(y) + B_1(y)) \mod (y^{n/2} + 1, q)$
3: $P_0(y) = U(y) + V(y) \cdot y \mod (y^{n/2} + 1, q)$
$P_1(y) = W(y) - (U(y) + V(y)) \mod (y^{n/2} + 1, q)$
4: $P(x) = P_0(x^2) + P_1(x^2) \cdot x$, where $y = x^2$
5: **return** $P(x)$

We first decompose the polynomials $A(x)$ and $B(x)$ based on the even and odd indices, as shown in Step 1, also called polyphase decomposition [57]. We denote $y = x^2$, and the polynomial $A(x)$ is expressed as:

$$A(x) = A_0(x^2) + A_1(x^2) \cdot x = A_0(y) + A_1(y) \cdot x, \tag{6.13}$$

where the even indexed polynomial $A_0(y)$ and the odd indexed polynomial $A_1(y)$ are expressed as:

$$\begin{aligned} A_0(y) &= a[0] + a[2]y + a[4]y^2 + \ldots \\ &\quad + a[n-2]y^{n/2-1} \mod (y^{n/2} + 1), \end{aligned} \tag{6.14}$$

$$\begin{aligned} A_1(y) &= a[1] + a[3]y + a[5]y^2 + \ldots \\ &\quad + a[n-1]y^{n/2-1} \mod (y^{n/2} + 1). \end{aligned} \tag{6.15}$$

A similar decomposition is applied to $B(x)$ to obtain its even and odd polynomials $B_0(y)$ and $B_1(y)$.

The product $P(x)$ can be computed as:

$$\begin{aligned} P(x) &= P_0(y) + P_1(y) \cdot x \\ &= (A_0(y) + A_1(y) \cdot x) \cdot (B_0(y) + B_1(y) \cdot x) \\ &= A_0(y) B_0(y) + [A_0(y) B_1(y) + A_1(y) B_0(y)] \cdot x \\ &\quad + [A_1(y) B_1(y)] \cdot y \end{aligned} \tag{6.16}$$

The polyphase decomposition describes one polynomial multiplication of length-n in terms of four polynomial multiplications of length-$n/2$. While this step in itself does not reduce the computation complexity, it is an essential first step. In Step 2, the fast filter algorithm describes the modular polynomial multiplication in terms of three polynomial multiplications of half-length; this reduces the complexity by 25%. Denote the three intermediate modular polynomial multiplication outputs as $U(y)$, $V(y)$, and $W(y)$. In the fast algorithm, $P_1(y)$ is computed as:

$$\begin{aligned} P_1(y) &= A_0(y) B_1(y) + A_1(y) B_0(y), \\ &= (A_0(y) + A_1(y)) (B_0(y) + B_1(y)) \\ &\quad - A_0(y) B_0(y) - A_1(y) B_1(y) \qquad (6.17) \\ &= W(y) - (U(y) + V(y)), \qquad (6.18) \end{aligned}$$

where

$$U(y) = A_0(y) B_0(y), \tag{6.19}$$

$$V(y) = A_1(y) B_1(y), \tag{6.20}$$

$$W(y) = (A_0(y) + A_1(y))(B_0(y) + B_1(y)). \tag{6.21}$$

Note that unlike $P_1(y)$, $P_0(y) = U(y) + V(y) \cdot y \bmod (y^{n/2} + 1)$ requires further modular polynomial reduction, which is achieved in the post-processing step. Since $V(y)$ needs to be multiplied by y before adding the coefficients of $U(y)$, the highest degree of coefficient exceeds the range of the ring $(y^{n/2} + 1)$, (i.e., $U(y) + V(y) \cdot y = u[0] + p_0[1]y + p_0[2]y^2 + ... + v[n/2 - 1]y^{n/2}$). As a

result, the even polynomial $P_0(y)$ requires an additional subtraction and is computed as:

$$\begin{aligned} P_0(y) =&(u[0] - v[n/2-1]) + p_0[1]y + p_0[2]y^2 \\ &+ \ldots + p_0[n/2-1]y^{n/2-1}. \end{aligned} \tag{6.22}$$

The data-flow of the proposed fast parallel architecture is shown in Fig. 6.7. Different from the traditional methods that execute the polynomial modular reduction during or after post-processing (i.e., combining the intermediate polynomials back to a single polynomial) [74, 85], we integrate polynomial modular reduction into the three intermediate polynomial multiplications. This is achieved by using the sequential systolic modular polynomial multiplication described in the previous section. A 2-level Karatsuba polynomial multiplication requires at least $(n-1)$ clock cycles to output n coefficients sequentially for the three intermediate polynomials and $(\frac{7}{2}n-4)$ or $(3n-3)$ modular additions/subtractions for post-processing [74].

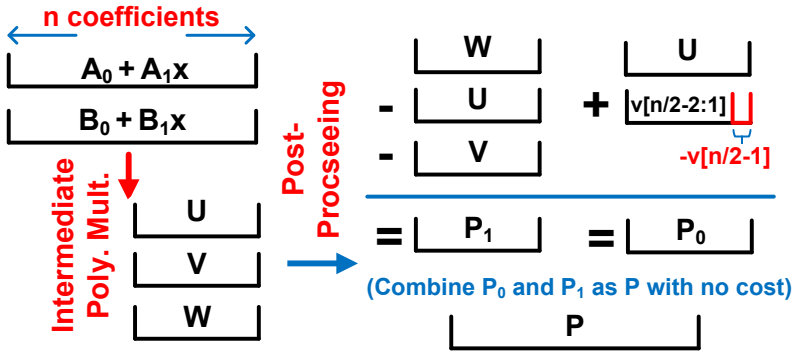

Figure 6.7: Data-flow of the Fast.2.PolyMult algorithm.

In contrast, by employing the sequential weight-stationary systolic polynomial modular multiplier as shown in Fig. 6.6, $\frac{n}{2}$ coefficients of $U(y)$, $V(y)$, and $W(y)$ are output in the same $(n-1)$ clock cycles without requiring additional elements. As these three intermediate polynomials are already in the ring R_q, the post-processing stage has a lower cost, which only needs $\frac{3}{2}n$ modular additions/subtractions.

Fig. 6.8 depicts the proposed hardware architecture for Algorithm 11 for a degree-n polynomial multiplication. It mainly consists of four adders/subtractors, three registers, and three degree-$\frac{n}{2}$ modular polynomial multipliers that also include three shift registers of length-$\frac{n}{2}$ (as described in Fig. 6.5). Besides, the bottom path can store $v[n/2-1]$ (coefficient from $V(y)$) for n clock cycles

and feed its negative form $(-v[n/2-1])$ to the adder at the upper path in each iteration l. This operation is controlled by two switches. When the left switch's instance is at $(n/2)l+0$, the output coefficient of $V(y)$ is loaded into a register, while the right switch will release the stored data to the next operation at $(n/2)l+(n/2-1)$.

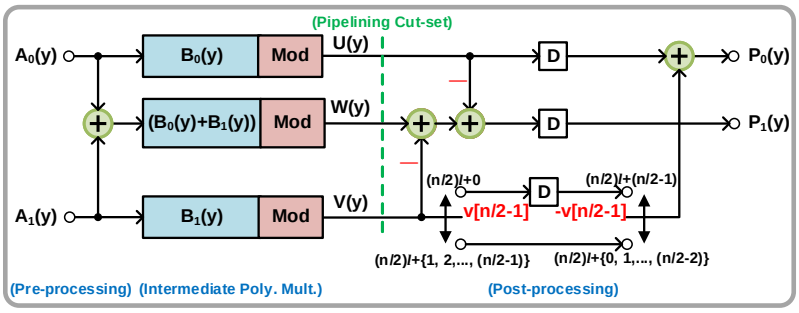

Figure 6.8: Fast 2-parallel modular polynomial multiplier.

The coefficients of $P_1(y)$ can be simply obtained by using two subtractors, while the coefficients for $P_0(y)$ are more complicated to generate. The addition between $U(y)$ and $V(y)\cdot y$ is explained using the timing diagrams for $n=8$ shown in Fig. 6.9.

clk:	4	5	6	7
U(y):	u[3]	u[2]	u[1]	u[0]
V(y):	v[3]	v[2]	v[1]	v[0]

(a)

clk:	4	5	6	7	8
U(y):		u[3]	u[2]	u[1]	u[0]
V(y)·y:	v[3]	v[2]	v[1]	v[0]	

(b)

clk:	4	5	6	7	8
U:		u[3]	u[2]	u[1]	u[0]
V:		v[2]	v[1]	v[0]	-v[3]
		p₀[3]	p₀[2]	p₀[1]	p₀[0]

(c)

Figure 6.9: Timing diagram for $P_0[y]$ at post-processing stage when $n=8$.

As the coefficients of $U(y)$ and $V(y)$ are generated in the same pattern as shown in Fig. 6.9(a), directly calculating $P_0(y)$ is infeasible without multiplying y for $V(y)$. However, delaying $U(y)$ by one cycle can enable the addition operation as shown in Fig. 6.9(b). Furthermore, to perform the polynomial modular reduction as in Equation 6.22, as described in Fig. 6.9(c), two switches and two delay elements are required. For the subtraction of $v[3]$ from $u[0]$, the first switch passes $v[3]$ to the delay element and the second switch releases its negative after four clock cycles ($\frac{n}{2}$ clock cycles for general case), as $u[0]$ is output four clock cycles ($\frac{n}{2}$ clock cycles for general case) after $v[3]$. Note that no additional adder/subtractor is needed and full hardware utilization is retained for all the components in the circuit. Moreover, this optimization technique still allows continuous processing of modular polynomial multiplications without requiring any null operations. To align the coefficients of $P_1(y)$ with $P_0(y)$, one delay element is placed at the end of $P_1(y)$'s output.

While the fast modular polynomial multiplier structure is similar to the fast parallel filter, there is one fundamental difference. Unlike the fast parallel filter where all computations are causal, the computation $V(y) \cdot y$ is inherently a *non-causal* operation. This is transformed into a causal operation by introducing a latency of one clock cycle; this can be achieved by placing delays at one feed-forward cut-set in the post-processing step. The proposed *novel* approach of computing $V(y) \cdot y$ does not increase the latency beyond one clock cycle and preserves the feed-forward property of the architecture and continuous data-flow property.

6.4.2 Fast 4-parallel Architecture

A fast 4-parallel architecture can be derived by iterating the fast 2-parallel architecture twice [57, 75–77]. The fast 4-parallel schoolbook modular polynomial multiplication algorithm (also denoted as Fast.4.PolyMult) is presented in Algorithm 12, while Fig. 6.10 shows the corresponding architecture.

Algorithm 12 Fast.4.PolyMult($A(x), B(x)$)

Input: $A(x)$ and $B(x) \in R_q$
Output: $P(x) = (P_0(x^4), P_1(x^4), P_2(x^4), P_3(x^4))$,
$//P(x) = A(x) \cdot B(x) \mod (x^n + 1, q)$

1: $A(x) = A_0(x^2) + A_1(x^2) \cdot x^2$
//split $A(x)$ as two parts based odd and even indices
$B(x) = B_0(x^2) + B_1(x^2) \cdot x^2$
//split $B(x)$ as two parts based odd and even indices
2: $(C_0(y), C_1(y)) =$ Fast.2.PolyMult$(A_0(x^2), B_0(x^2))$,
where $y = x^4$
$(C_2(y), C_3(y)) =$ Fast.2.PolyMult$\Big((A_0(x^2) + A_1(x^2)),$
$(B_0(x^2) + B_1(x^2))\Big)$
$(C_4(y), C_5(y)) =$ Fast.2.PolyMult$(A_1(x^2), B_1(x^2))$
3: $P_0(y) = C_0(y) + C_5(y) \cdot y \mod (y^{n/4} + 1, q)$
$P_1(y) = C_2(y) - C_1(y) - C_4(y) \mod (y^{n/4} + 1, q)$
$P_2(y) = C_1(y) + C_4(y) \mod (y^{n/4} + 1, q)$
$P_3(y) = C_3(y) - C_0(y) - C_5(y) \mod (y^{n/4} + 1, q)$
4: $P(x) = P_0(x^4) + P_1(x^4) \cdot x + P_2(x^4) \cdot x^2 + P_3(x^4) \cdot x^3$, where $y = x^4$
5: **return** $P(x)$

The Fast.4.PolyMult algorithm has four steps. In Step 1 of Algorithm 12, $A(x)$ is split into two parts based on the odd and even indices. Then, $A_0(x^2)$, $A_1(x^2)$, and their sum $(A_0(x^2) + A_1(x^2))$ are further split based on Step 1 in Fast.2.PolyMult (Algorithm 11). $A_0(x^2)$ and $A_1(x^2)$ are decomposed as four polynomials $(A_{00}(x^4), A_{01}(x^4), A_{10}(x^4), A_{11}(x^4))$ which are fed to upper

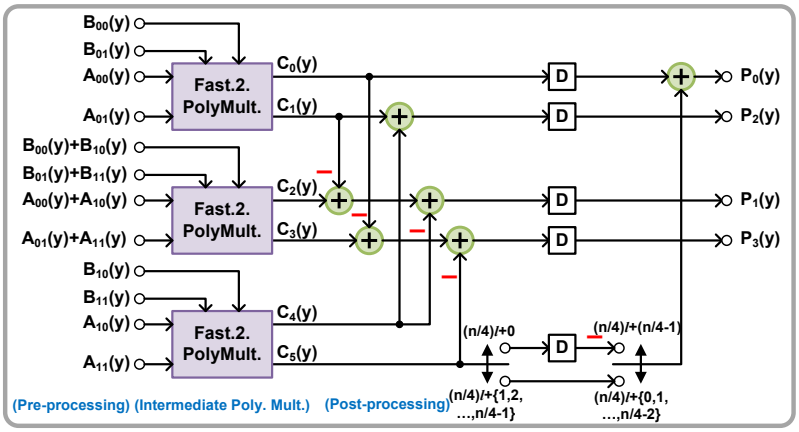

Figure 6.10: Fast 4-parallel modular polynomial multiplier.

and lower fast 2-parallel modular polynomial multipliers (denoted Fast-2 PolyMult. in Fig. 6.10), respectively. Meanwhile, as the fast 2-parallel modular polynomial multiplier has two inputs in parallel, $(A_0(x^2) + A_1(x^2))$ in Step 2 is simply implemented as two adders in the middle fast 2-parallel modular polynomial multiplier, i.e., $(A_{00}(x^4) + A_{10}(x^4))$ and $(A_{01}(x^4) + A_{11}(x^4))$. Let y represent x^4; Hence the four polynomials decomposed from $A_0(x^2)$ and $A_1(x^2)$ can be expressed as

$$\begin{aligned} A_{00}(y) &= a[0] + a[4]y + a[8]y^2 + ... \\ &\quad + a[n-4]y^{n/4-1} \mod (y^{n/4}+1), \end{aligned} \tag{6.23}$$

$$\begin{aligned} A_{10}(y) &= a[1] + a[5]y + a[9]y^2 + ... \\ &\quad + a[n-3]y^{n/4-1} \mod (y^{n/4}+1), \end{aligned} \tag{6.24}$$

$$\begin{aligned} A_{01}(y) &= a[2] + a[6]y + a[10]y^2 + ... \\ &\quad + a[n-2]y^{n/4-1} \mod (y^{n/4}+1), \end{aligned} \tag{6.25}$$

$$\begin{aligned} A_{11}(y) &= a[3] + a[7]y + a[11]y^2 + ... \\ &\quad + a[n-1]y^{n/4-1} \mod (y^{n/4}+1), \end{aligned} \tag{6.26}$$

where

$$A(x) = A_{00}(x^4) + A_{10}(x^4) \cdot x + A_{01}(x^4) \cdot x^2 + A_{11}(x^4) \cdot x^3. \tag{6.27}$$

$B(x)$ can be decomposed in a similar manner.

In the intermediate polynomial multiplication stage, three degree-$\frac{n}{4}$ fast 2-parallel modular polynomial multipliers (Fig. 6.8) generate six degree-$\frac{n}{4}$ polynomials, $C_0(y), C_1(y), \ldots, C_5(y)$. As shown in Fig. 6.10, $\frac{3}{2}n$ additions/subtractions are carried out by six adders/subtractors, where each adder/subtractor performs $\frac{n}{4}$ additions/subtractions. Finally, polynomial modular reduction for $C_5(y)$ and $C_0(y)$ are performed in a manner similar to the fast 2-parallel architecture (Fig. 6.8).

6.4.3 Fast 3-parallel Architecture

We also present the design for a fast 3-parallel schoolbook modular polynomial multiplication algorithm (denoted as Fast.3.PolyMult), allowing M to be a multiple of 3, enabling various levels of parallelism.

Algorithm 13 Fast.3.PolyMult($A(x), B(x)$)

Input: $A(x)$ and $B(x) \in R_q$
Output: $P(x) = (P_0(x^3), P_1(x^3), P_2(x^3))$,
$//P(x) = A(x) \cdot B(x) \mod (x^n + 1, q)$

1: $A(x) = A_0(x^3) + A_1(x^3) \cdot x + A_2(x^3) \cdot x^2$
$B(x) = B_0(x^3) + B_1(x^3) \cdot x + B_2(x^3) \cdot x^2$
2: $C_0(y) = A_0(y)B_0(y) \mod (y^{n/3} + 1, q)$
$C_1(y) = A_1(y)B_1(y) \mod (y^{n/3} + 1, q)$
$C_2(y) = A_2(y)B_2(y) \mod (y^{n/3} + 1, q)$
$C_3(y) = \big(A_0(y) + A_1(y)\big)\big(B_0(y) + B_1(y)\big) \mod (y^{n/3} + 1, q)$
$C_4(y) = \big(A_1(y) + A_2(y)\big)\big(B_1(y) + B_2(y)\big) \mod (y^{n/3} + 1, q)$
$C_5(y) = \big(A_0(y) + A_1(y) + A_2(y)\big)\big(B_0(y) + B_1(y) + B_2(y)\big) \mod (y^{n/3} + 1, q)$, where $y = x^3$
3: $D_0(y) = C_3(y) - C_1(y) \mod (y^{n/3} + 1, q)$
$D_1(y) = C_4(y) - C_1(y) \mod (y^{n/3} + 1, q)$
$D_2(y) = C_0(y) - C_2(y) \cdot y \mod (y^{n/3} + 1, q)$
$D_3(y) = C_5(y) \mod (y^{n/3} + 1, q)$
4: $P_0(y) = D_2(y) + D_1(y) \cdot y \mod (y^{n/3} + 1, q)$
$P_1(y) = D_0(y) - D_2(y) \mod (y^{n/3} + 1, q)$
$P_2(y) = D_3(y) - D_0(y) - D_1(y) \mod (y^{n/3} + 1, q)$
5: $P(x) = P_0(x^3) + P_1(x^3) \cdot x + P_2(x^3) \cdot x^2$, where $y = x^3$
6: **return** $P(x)$

Fast.3.PolyMult algorithm also consists of three stages, which is illustrated in Algorithm 13. During the polyphase decomposition (pre-processing stage), polynomial $A(x)$ is decomposed as

$$A(x) = A_0(x^3) + A_1(x^3) \cdot x + A_2(x^3) \cdot x^2. \tag{6.28}$$

The modular multiplication result $P(x)$ can be defined as:

$$P(x) = P_0(y) + P_1(y) \cdot x + P_2(y) \cdot x^2, \tag{6.29}$$

where $y = x^3$, and these three sub-polynomials are presented in Step 4 in Algorithm 13. The derivation of the fast 3-parallel modular multiplier is similar to the fast parallel filter derivation; the reader is referred to [57, 75, 77].

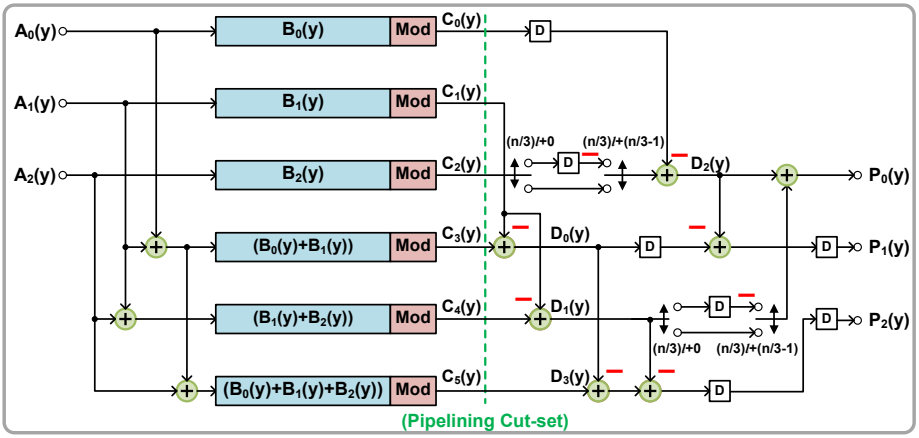

Figure 6.11: Fast 3-parallel modular polynomial multiplier.

The architecture for the Fast.3.PolyMult algorithm is shown in Fig. 6.11, which consists of six degree-$\frac{n}{3}$ modular polynomial multipliers, thirteen modular adders/subtractors with additional delay elements. These six degree-$\frac{n}{3}$ modular multipliers compute the intermediate polynomials $C_0(y)$ to $C_5(y)$ with an additional pipelining stage at the end of the modular multipliers' output.

In the post-processing stage, six intermediate polynomials are used to generate four new intermediate polynomials $D_0(y)$ to $D_3(y)$ before computing the outputs $P_0(y)$, $P_1(y)$, and $P_2(y)$ using less number of additions/subtractions.

6.4.4 Fast M-parallel Architecture

Using *iterated* approach, we can use fast 2-parallel architecture and/or fast 3-parallel architecture to achieve higher levels of parallelism. Therefore, we can implement various fast M-parallel architectures, where the level of parallelism M can be a power-of-two integer, power-of-three integer, or product of any power-of-two and power-of-three.

The high-level overview of the generalized fast M-parallel architecture is shown in Fig. 6.12. This architecture mainly has M sub-modular polynomial multipliers of degree-n/M operating in parallel to generate M sub-polynomials of $P(x)$. In addition, the components for post-processing as well as the control unit are used to align the coefficients from all the output sub-polynomials of $P(x)$. This is similar to inserting a pipelining cut-set to transform non-causal operations into causal operations, at the expense of an increase in latency by one cycle.

During the computation, the data can be either accessed from the host's personal computer (PC) to achieve the input polynomials' coefficients directly or communicated to the FPGA's RAM.

The timing performance can be theoretically derived as follows. The fast M-parallel design can reduce the response time to approximately n/M clock cycles. In general, the total latency of an M-parallel modular polynomial multiplier for L polynomial multiplications can be expressed as:

$$T_{lat} = n(1+L)/M + \lceil \log_2(M) \rceil. \tag{6.30}$$

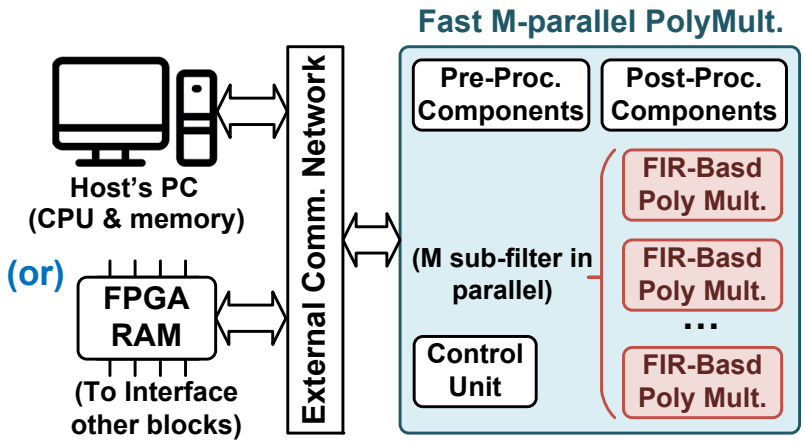

Figure 6.12: High-level overview of generalized fast M-parallel modular polynomial multiplier.

6.5 Experimental Results

The performance of the proposed modular polynomial multiplication is demonstrated for the Saber scheme using Verilog HDL implementation. Several changes are adopted specifically for the Saber scheme.

6.5.1 Basic Arithmetic Units for Saber

Due to the Saber scheme's advantages, the basic components do not consume a large number of hardware resources. In particular, the modular multiplier discussed in Section 6.3 can be replaced by a few adders since the coefficients of polynomial $B(x)$ are in the range of $[-4, 4]$ [82]. As the moduli are power-of-two integers, the modular reduction can again be performed by simply keeping the lower bits. Note that, the coefficients in both polynomials $A(x)$ and $B(x)$ are represented in the sign-magnitude form, and the word-lengths of the magnitudes of these two polynomials are 13-bit and 3-bit, respectively. The modular adder calculates the 13-bit sum (sum) by adding the product ($prod$) of the corresponding $a[i]$ and $b[j]$, and the output from the register acc as shown in Fig. 6.6, which can also be mathematically expressed as:

$$sum = \begin{cases} acc - prod, & \text{if } a_{sign} \bigoplus b_{sign} = 1, \\ acc + prod, & \text{otherwise,} \end{cases} \tag{6.31}$$

where a_{sign} and b_{sign} are the sign bits of the two operands $a[i]$ and $b[j]$, respectively. Note that all the modular polynomial multiplications correspond to degree $n = 256$.

6.5.2 Unified Hash Function Block

The hash functions used in Saber are SHA3-256, SHA3-512, and SHAKE-128 extendable output functions (XOF), which are from the Keccak family [83]. For any message (input to the hash function), the length of the digest (hashed output) is always fixed. SHA3-256 and SHA3-512 produce 256 bits and 512 bits, respectively, while SHAKE-128 runs the permutation iteratively and stops until the results reach the required output length. In the Saber scheme, the SHAKE-128 function is used to generate pseudo-random matrix $\mathbf{A}$, the secret key $\mathbf{s}$, and $\mathbf{s}'$ as described in Algorithms 8 and 9. SHA3-256 and SHA3-512 are applied only in the IND-CCA secure case for enhancing the protection of the keys, message, and ciphertext. These functions share an identical underlying sponge function Keccak-f[1600] with different rates, paddings, and output patterns. Therefore, similar to prior works [72, 89, 99], we also consider a unified hash block in our LARK architecture.

As the sponge function Keccak-f[1600] has been standardized and widely used in many hardware implementations for cryptographic algorithms, many architectures for PQC schemes directly adopt the open-source packages to design a unified hash function block [72,73,89]. Most open-source

packages add stages of pipelining to achieve a high frequency (low critical path) design in order to adapt to general applications [100]. However, the critical path among the prior works is under the NTT-based or schoolbook modular polynomial multiplier that requires addition or multiplication, which is much higher than the Keccak core provided in the open-source packages, thus implying that the pipelines are redundant. Different from the prior works, we implement our own unified hash function block as we aim to reduce the total latency for computing the hash functions by eliminating unnecessary pipelining stages.

The Keccak hash functions generally divide into two parts, i.e., absorb and squeeze, with both including 24 rounds of permutation to handle different input and output length requirements [83]. The absorb step truncates the input message to equal or less than the given rate and permutes the new states generated by bitwise XOR of previous states and message. The process is completed when all input messages are absorbed and permutations are finished. The squeeze process operates the 24 rounds of permutation after absorb generates the required length of digest using the state at the end of every 24 rounds of permutations.

The unified hash function block consists of three steps: message input, permutation, and digest output. In particular, 24 rounds of permutation are finished in 24 clock cycles to a single round for permutation in a single clock cycle, which has less latency compared with the works in [72] (28 clock cycles) and [89] (25 clock cycles) for the same rounds of permutation. Moreover, the message input and digest output are transmitted in 64-bit per clock cycle, thus the required clock cycles are determined by the bit-length of the message and digest. Besides, our unified hash function block needs to be alternatively reconfigured as SHA3-256, SHA3-512, and SHAKE-128, whereby we construct the control logic to handle varied input and output requirements in permutation rounds of the absorb and squeeze processes. However, there is a bottleneck in that the bit-lengths of the message or digest are in thousands or hundreds of bits, which requires many clock cycles to load or output the data. Similar to the prior works [72], we buffer the new message/calculated digest during the permutation without interfering with the permutations to improve the utilization of the permutation core as shown in Fig. 6.13.

The experiment is performed on the Xilinx Artix-7 AC701 FPGA board since Artix-7 family FPGA boards are recommended by NIST for PQC hardware implementation. In addition, since several prior works also used the high-performance Xilinx UltraScale+ FPGA board, we also demonstrate the performance of our architecture on this board for more comparisons. The communication

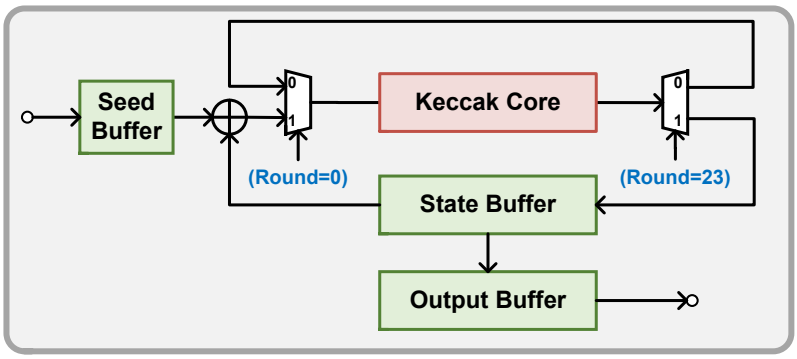

Figure 6.13: Architecture for unified hash function block.

and data transmission between FPGA and PC use the universal asynchronous receiver transmitter (UART) module provided by the AC701 device for functionality verification.

6.5.3 Evaluation of Modular Polynomial Multiplier

We first examine the performance of our proposed modular polynomial multipliers, including the FIR filter-based (Fig. 6.5), fast 2-parallel architecture, and fast 4-parallel architecture.

The experimental results and comparison with prior works [72, 85, 92, 98] are summarized in Table 6.2. A further comparison of the timing performance is presented in Tables 6.3 and 6.4. The clock frequencies are set as 250MHz and 133MHz for UltraScale+ and Artix-7, respectively. Tables 6.3 and 6.4 summarize the number of clock cycles and actual latency for one modular polynomial multiplication (PolyMult.), and all the modular polynomial multiplications in KeyGen, Encaps, and Decaps steps of Saber scheme with the medium security level. Note that the number of modular polynomial multiplications in encryption (decryption) is the same as in Encaps (Decaps). It can be seen from Table 6.2 that our design has a shorter critical path than those of the designs in [85, 92] and the same as the work in [72, 98].

For a fair comparison, we focus on the evaluation against the architecture [72], and its extended work [98] since both works use the same clock frequency and target high-speed design. Note that the high-speed designs in [98] do not provide the timing performance in the KeyGen, Encaps, and Decaps steps for Saber scheme but only the result for one modular polynomial multiplication, so we adopt the number of clock cycles from their previous work [72] and present in Table 6.3. In particular, the framework and the timing performance (including the number of clock cycles and frequency) for one modular polynomial multiplication of [98] are maintained to be the same as their

Table 6.2: Comparison of area consumption and frequency for modular polynomial multiplier when $n = 256$

Design	Device	LUTs	FFs	DSPs	BRAM	Freq. [MHz]
Roy (1 Mult.) [72]	Ultrascale+	17406	5083	0	0	250
Roy (2 Mult.) [72]	Ultrascale+	31853	8844	0	0	250
Zhu [92]	Ultrascale+	13954	3943	85	6	100
Basso (HS-I 256) [98]	Ultrascale+	10844	5150	0	0	250
Basso (HS-I 512) [98]	Ultrascale+	22118	4920	0	0	250
FIR.PolyMult	Ultrascale+	16971	8755	0	0	250
Fast.2.PolyMult	Ultrascale+	25831	12850	0	0	250
Fast.4.PolyMult	Ultrascale+	35306	19143	64	0	250
Mera [85]	Artix-7	7400	7331	38	2	125
FIR.PolyMult	Artix-7	16902	8755	0	0	133
Fast.2.PolyMult	Artix-7	25854	12850	0	0	133
Fast.4.PolyMult	Artix-7	35396	19143	64	0	133

Table 6.3: Timing performance of modular polynomial multiplier when $n = 256$ in medium security level of Saber based on Ultrascale+ FPGA board

Design	1 PolyMult.	KeyGen	Encaps	Decaps	ATP-LUT[b]
Roy (1 Mult.) [72]	256[c] (1.02)[a]	2685 (10.74)	3592 (14.37)	4484 (17.94)	7.49 $\times 10^5$
Roy (2 Mult.) [72]	128[c] (0.51)	1552 (6.21)	2205 (8.82)	2911 (11.64)	8.50 $\times 10^5$
Zhu [92]	81 (0.81)	(Not Reported)	978 (9.78)	1227 (12.27)	-
Basso (HS-I 256) [98]	256[c] (1.02)	2685 (10.74)	3592 (14.37)	4484 (17.94)	4.67 $\times 10^5$
Basso (HS-I 512) [98]	128[c] (0.51)	1552 (6.21)	2205 (8.82)	2911 (11.64)	5.89 $\times 10^5$
FIR.PolyMult	511 (2.04)	2560 (10.24)	3328 (13.31)	4096 (16.38)	6.77 $\times 10^5$
Fast.2.PolyMult	255 (1.02)	1281 (5.12)	1665 (6.66)	2049 (8.20)	5.16 $\times 10^5$
Fast.4.PolyMult	127 (0.51)	642 (2.57)	834 (3.34)	1026 (4.10)	3.50 $\times 10^5$

[a]: Total latency in the unit of clock cycles (actual latency in the unit of μs) of one modular polynomial multiplication, or all the modular polynomial multiplications in Saber's specific step
[b]: ATP-LUT (area-time product of LUTs) is calculated from the number of LUTs times the sum of actual latency (μs) of the total number of modular polynomial multiplications in KeyGen, Encaps, and Decaps steps
[c]: Clock cycles for reading and writing operations are not counted

Table 6.4: Timing performance of modular polynomial multiplier when $n = 256$ in medium security level of Saber based on Artix-7 FPGA board

Design	1 PolyMult.	KeyGen	Encaps	Decaps	ATP-LUT[b]
Mera [85]	1299 (10.30)	11592 (92.74)	15456 (123.65)	19320 (154.56)	27.45 $\times 10^5$
FIR.PolyMult	511 (3.83)	2560 (19.25)	3328 (25.02)	4096 (30.80)	12.66 $\times 10^5$
Fast.2.PolyMult	255 (1.92)	1281 (9.63)	1665 (12.52)	2049 (15.41)	9.71 $\times 10^5$
Fast.4.PolyMult	127 (0.95)	642 (4.83)	834 (6.27)	1026 (7.71)	6.65 $\times 10^5$

[a]: Total latency in the unit of clock cycles (actual latency in the unit of μs) of one modular polynomial multiplication, or all the modular polynomial multiplications in Saber's specific step
[b]: ATP-LUT (area-time product of LUTs) is calculated from the number of LUTs times the sum of actual latency (μs) of the total number of modular polynomial multiplications in KeyGen, Encaps, and Decaps steps
[c]: Clock cycles for reading and writing operations are not counted

Table 6.5: Performance of modular polynomial multiplier using fast M-parallel architecture when $n = 180$ based on Artix-7 FPGA board

Design	**Fast.2.PolyMult**	**Fast.3.PolyMult**	**Fast.4.PolyMult**
LUTs	17902	21729	25110
FFs	9096	11996	13633
DSPs	0	60	45
Freq. [MHz]	133	133	133
1 PolyMult.*	181 (1.36)[a]	122 (0.91)	92 (0.69)
9 PolyMult.*	901 (6.77)	602 (4.53)	452 (3.40)
ATP-LUT[a]	1.21 $\times 10^5$	0.98 $\times 10^5$	0.85 $\times 10^5$
Throughput[c]	2	3	4

[a]: Total latency in the unit of clock cycles (actual latency in the unit of μs)
[b]: ATP-LUT (area-time product of LUTs) is calculated from the number of LUTs times the sum of actual latency (μs) of nine modular polynomial multiplications
[c]: Throughput in the unit of samples per clock cycle

previous work [72], while their optimized centralized multiplier design for the MAC unit in [98] significantly reduces LUTs. Besides, these two works present the general design and parallel design. In Tables 6.2 and 6.3, the general designs correspond to Roy (1 Mult.) as well as Basso (HS-I 256), and parallel designs correspond to Roy (2 Mult.) as well as Basso (HS-I 512).

Compared to the general design in [72], our proposed FIR filter-based modular polynomial multiplier (i.e., FIR.PolyMult) has slightly less number of LUTs and less total latency in the modular polynomial multiplications used in three steps of the Saber scheme; however, a higher number of flip-flops (FFs) is needed due to the additional shift registers. Note that the clock cycles used in one modular polynomial multiplication in Roy (1 Mult.) are only 256 clock cycles due to the fact that their result does not count the number of clock cycles used for reading and writing operations [98], which is same as the response time in our proposed FIR.PolyMult design. When compared to Roy (2 Mult.), our fast 2-parallel architecture achieves 18.91%, 25.08%, and 39.88% reduction on the number of LUTs, latency, and area-time product of LUTs (ATP-LUT), respectively.

Besides, their extended work [98] is also taken into comparison. Our FIR.PolyMult and fast 2-parallel architectures require a larger number of LUTs compared to their general design (Basso (HS-I 256)) and parallel design (Basso (HS-I 512)), but the latency of our two proposed designs is smaller than theirs. In this case, the performance of ATP-LUT is utilized for a fair comparison. The Basso (HS-I 256) design has 31.02% reduction compared to our FIR.PolyMult architecture. However, the ATP-LUT product of our fast 2-parallel architecture is reduced by 13.24% compared to the Basso (HS-I 512). This result implies that our fast M-parallel design has a superior performance compared

to conventional parallel processing techniques.

Even though our design requires more FFs in the data-path and shift registers, we argue that it has a small influence on the overall performance of UltraScale+ and Artix-7 FPGA boards since both devices have a much higher resource budget for FFs than LUTs [101].

Furthermore, both the modular polynomial multiplier in LWRpro [92] and the compact modular polynomial multiplier in [85, 92] use the Toom-Cook/Karatsuba algorithm with 8-level and 4-level, respectively. The compact polynomial multiplier in [85] has a long critical path of five adders/subtractors and two multipliers in the interpolation part, which requires two pipelining stages to reduce the critical path for maintaining a high frequency. This design targets the low-area performance, which only requires limited numbers of LUTs, FFs, and only 38 DSP units, as shown in Table 6.2. While this design has lower LUT usage than our architecture, it suffers from a low speed since their degree-64 polynomial multipliers require 1168 clock cycles for each computation, which causes the actual latency in such a compact design to be around 19 times of the latency in our fast 4-parallel architecture as presented in Table 6.4. If we consider the ATP-LUT as the performance metric to compare our proposed fast 4-parallel architecture and this prior low-area design, it shows that our design achieves a 75.77% reduction.

Besides, the modular polynomial multiplier in [92] requires the lowest number of clock cycles among all the prior works, while having the lower clock frequency as illustrated in Table 6.3. In comparison, our fast 4-parallel architecture requires 15.65% less number of clock cycles and achieves a 66.26% reduction in the actual latency for all the modular polynomial multiplications in the Encaps and Decaps steps. Considering the area performance of this work, their modular polynomial multiplier uses 60.48% fewer LUTs but 24.71% more DSPs compared to the proposed fast 4-parallel architecture. Thus, the ATP products of DSP and LUT need to be considered separately. Since the clock cycles used for KeyGen are not reported in [92], the ATP-LUT (ATP-DSP) for the comparison with this work is defined as the number of LUTs (DSPs) times the sum of actual latency (μs) of the total number of modular polynomial multiplications in two steps only, Encaps and Decaps. Specifically, the ATP-LUT and ATP-DSP in their modular polynomial multiplier are 3.08×10^5 and 1874.20, respectively. The ATP-LUT and ATP-DSP in the fast 4-parallel architecture are 2.63×10^5 and 476.16, respectively. Under this comparison, ATP-LUT and ATP-DSP are reduced by 14.61% and 75.07%, respectively, in our design.

Thus, we can conclude that our design achieves a significant reduction in the latency or the

delay (critical path) which leads to reductions in ATP when compared to the two prior works that employ the Karatsuba/Toom-Cook algorithm-based modular polynomial multiplication.

Our proposed modular polynomial multipliers can be sufficient to support different security levels of Saber without any change of the area consumption and the frequency presented in Table 6.2, but only requires the different number of clock cycles.

6.5.4 Parallel Architectures

The works in [72] and [98] also present a parallel architecture, which is a scaled version. The parallel design in [72] (Roy (2 Mult.)) uses two multipliers in one MAC unit, and the parallel design in [98] (Basso (HS-I 512)) doubles their MAC units (each MAC unit has one MUX and one adder). When compared to the Roy (2 Mult.) design, our fast 2-parallel architecture achieves a significant reduction in the area overhead and latency. In particular, our fast 2-parallel architecture consumes only about 34% higher area consumption than the FIR filter polynomial multiplier while reducing latency by 50%, while their scaled version of the parallel modular polynomial multiplier has 45% overhead compared with the general design architecture (Roy (1 Mult.)). Along with parallelization, the delay also increases, as the critical path will change from one multiplication and one addition to one multiplication and two additions. In this case, an additional pipeline is added in the design of Roy (2 Mult.) [72] to maintain the same high frequency. Under the same number of pipeline stages, our fast 2-parallel architecture achieves a lower critical path and hence can be driven by a clock with a higher frequency.

Table 6.6: Comparison with recent Saber scheme implementation in medium security level

Design	Roy [72]	**Ours**	**Ours**
Platform	UltraScale+	UltraScale+	Artix-7
Time in (μs): KeyGen/Encaps/Decaps	21.8/26.5/32.1	10.2/12.6/15.6	19.2/23.6/29.2
Freq. [MHz]	250	250	133
Area: LUTs/FFs/DSPs/BRAM	23.6k/9.8k/0/2	41.5k/22.3k/64/2	41.5k/22.3k/64/2
ATP-LUT[a]	1.9 $\times 10^6$	1.6 $\times 10^6$	3.0 $\times 10^6$

[a]: ATP-LUT (area-time product of LUTs) is calculated from the number of LUTs times the sum of actual latency (μs) of KeyGen, Encaps, and Decaps steps

We also compare different fast M-parallel architectures for $n = 180$ in Table 6.5. It can be noticed that when the level of parallelism increases (M becomes larger), the actual latency is reduced at the expense of higher area consumption. Besides, the throughput of the designs are

increased when M becomes larger. The ATP-LUT product for the fast 2-parallel, 3-parallel, and 4-parallel architectures are listed for computing nine modular polynomial multiplications, which indicates that a higher level of parallelism can provide a more efficient design if sufficient resource budget is available.

6.5.5 Comparison with Saber PQC Scheme Implementations

For the implementation of the entire Saber scheme, the modular polynomial multiplication is implemented by the proposed fast 4-parallel architecture, while other simple functional blocks are modified from the open-source codes provided in [82] and [72].

Table 6.6 presents the comparison of the FPGA performance with recent hardware implementation [72] for the Saber PQC scheme at a medium security level. The latency in our design is 52% less than the latency in [72] with the cost of more LUTs and DSPs consumed. In fact, the reduction is mainly from our optimized low-latency modular polynomial multiplier. Besides, instead of directly adopting the open-source SHA3 hash function block as in [72], we also implement the hash function block when implementing the entire scheme. For example, the total latency of SHA3-256 (needs to process 32-byte, 64-byte, 992-byte, and 1088-byte seeds) operating in the hash function block is reduced from 585 clock cycles to 526 clock cycles in the Saber Encaps. The rationale behind this latency reduction is as follows. Most open-source packages add stages of pipelining to achieve a high frequency (low critical path) design in order to adapt to general applications [100]. However, the critical path of the modular polynomial multiplier that requires addition or multiplication from prior work is much higher than the Keccak core provided in the open-source packages, thus implying that some pipelines are redundant. Different from the prior work, we implement our own hash function block as we aim to reduce the total latency for computing the hash functions by eliminating unnecessary pipelining stages.

For the area performance, although we have increased hardware costs, both Artix-7 and UltraScale+ FPGA boards still have sufficient resources to accommodate our fast 4-parallel design. In other words, our proposed fast 4-parallel architecture is under the constraint of hardware complexity specified by NIST (Artix-7).

6.6 Conclusion

This chapter has presented a novel modular polynomial multiplier and demonstrated its applications for lattice-based cryptography. The proposed hardware design exploits the fast filtering technique to achieve low latency, high scalability, and full hardware utilization. We proposed efficient parallel architectures with much lower hardware overhead and latency than prior works. Our design can be easily generalized across different levels of parallelism. Comprehensive experimental results are presented. We show that our design achieves superior performance than the state-of-the-art modular polynomial multipliers based on schoolbook polynomial multiplication or the Karatsuba algorithm. A case study of instantiating Saber scheme is also presented, which shows that our proposed design can accelerate the computation and reduce the actual latency of the cryptosystem compared with the prior work.

Chapter 7

PaReNTT: Low-Latency Parallel Residue Number System and NTT-Based Long Polynomial Modular Multiplication for Homomorphic Encryption

High-speed long polynomial multiplication is important for applications in homomorphic encryption (HE) and lattice-based cryptosystems. This chapter addresses low-latency hardware architectures for long polynomial modular multiplication using the number-theoretic transform (NTT) and inverse NTT (iNTT). Chinese remainder theorem (CRT) is used to decompose the modulus into multiple smaller moduli. Our proposed architecture, namely PaReNTT, makes four novel contributions. First, parallel NTT and iNTT architectures are proposed to reduce the number of clock cycles to process the polynomials. This can enable real-time processing for HE applications, as the number of clock cycles to process the polynomial is inversely proportional to the level of parallelism. Second, the proposed architecture eliminates the need for permuting the NTT outputs before their product is input to the iNTT. This reduces latency by $n/4$ clock cycles, where n is the

length of the polynomial, and reduces buffer requirement by one delay-switch-delay circuit of size n. Third, an approach to select special moduli is presented where the moduli can be expressed in terms of a few signed power-of-two terms. Fourth, novel architectures for pre-processing for computing residual polynomials using the CRT and post-processing for combining the residual polynomials are proposed. These architectures significantly reduce the area consumption of the pre-processing and post-processing steps. The proposed long modular polynomial multiplications are ideal for applications that require low latency and high sample rate as these feed-forward architectures can be pipelined at arbitrary levels. The experimental results show that the proposed architecture reduces the area-block processing product (ABP) by a factor of 43.2 times with respect to LUT and 11.5 times with respect to DSP, when compared without the use of CRT, for a polynomial degree of 4096 and word-length of 192 bits, for a two-parallel architecture.

7.1 Introduction

Privacy-preserving protocols and the security of the information are essential for cloud computing. To this end, cloud platforms typically encrypt the data by certain conventional symmetric-key or asymmetric-key cryptosystems to protect user privacy. However, these methods cannot prevent information leakage during the computation on the cloud since the data must be decrypted before the computation. To further enhance privacy, homomorphic encryption (HE) has emerged as a promising tool that can guarantee the confidentiality of information in an untrusted cloud. Homomorphic encryption is also deployed in privacy-preserving federated learning [102] and neural network inference [103].

Homomorphic multiplication and homomorphic addition are two fundamental operations for the HE schemes. Most of the existing HE schemes are constructed from the ring-learning with errors (R-LWE) problem [19] that adds some noise to the ciphertext to ensure post-quantum security. However, the quadratic noise growth of homomorphic multiplication requires the ciphertext modulus to be very large, which results in inefficient arithmetic operations. One possible solution to address this issue is to decompose the modulus and execute it in parallel. This approach has been used in residue number system (RNS) representation. In the literature, RNS-based implementations have been employed in several software [34, 104] and hardware implementations [25, 29, 37]. However, RNS relies on the Chinese remainder theorem (CRT), which requires additional pre-processing and

post-processing operations. The hardware building blocks for these steps need to be optimized; otherwise, the complexity of the RNS system will negate the advantages of parallelism of the RNS. Meanwhile, modular polynomial multiplication is one of the essential arithmetic operations for the R-LWE problem-based cryptosystems and, indeed, HE schemes. The complexity of the number-theoretic transform (NTT)-based modular polynomial multiplication can be reduced dramatically compared to the schoolbook-based modular polynomial multiplication.

Different modular long polynomial multiplier architectures can be adopted for different applications. For example, a low-area time-multiplexed architecture is well-suited for an edge device. However, the cloud requires very high-speed architectures where multiple coefficients of the polynomial need to be processed in a clock cycle. This inherently requires a parallel architecture where the level of parallelism corresponds to the number of coefficients processed in a clock cycle. While substantial research has been devoted to designing and implementing sequential and time-multiplexed architectures, much less research on parallel NTT-based architectures has been presented. Computing the inverse NTT (iNTT) of the product of NTT of the two polynomials can lead to long latency and extra buffer requirement if its scheduling aspects are not considered as the product needs to be shuffled before the iNTT is computed.

Although parallel NTT-based architectures can achieve low latency and high speed, these require a large silicon area for the arithmetic operations as the word-lengths of the coefficients can be large. To reduce the area, residue arithmetic is used to convert the coefficient into several smaller coefficients that can be implemented using shorter word-lengths. This chapter proposes parallel residue arithmetic and NTT-based modular long polynomial multiplication referred to as PaReNTT. The use of different scheduling (folding) of the NTT and iNTT operations eliminates the need for additional buffers. Thus, the latency of the complete operation is reduced. The use of parallel NTT architecture reduces the number of clock cycles needed to process the long polynomial modular multiplication. The proposed parallel NTT and iNTT architectures are completely feed-forward and achieve full hardware utilization. These can be pipelined at any arbitrary level. To the best of our knowledge, the proposed architecture is the first approach for feed-forward and parallel NTT-based implementation that eliminates intermediate shuffling or buffer requirement.

Fig. 7.1 shows the overview of the proposed PaReNTT architecture, which can be broken down into three steps. The first step, referred to as residual polynomials computations (pre-processing operation), splits the two input polynomials into several polynomials whose coefficients

are small. Then, instead of using only one modular polynomial multiplier, several modular polynomial multiplications are executed in parallel in the residual domain. Finally, the post-processing operation performs the inverse mapping for the product polynomials to one polynomial using the CRT. The result is the same as directly performing the modular polynomial multiplication for two input polynomials. In the literature, most CRT and NTT-based modular polynomial multipliers are

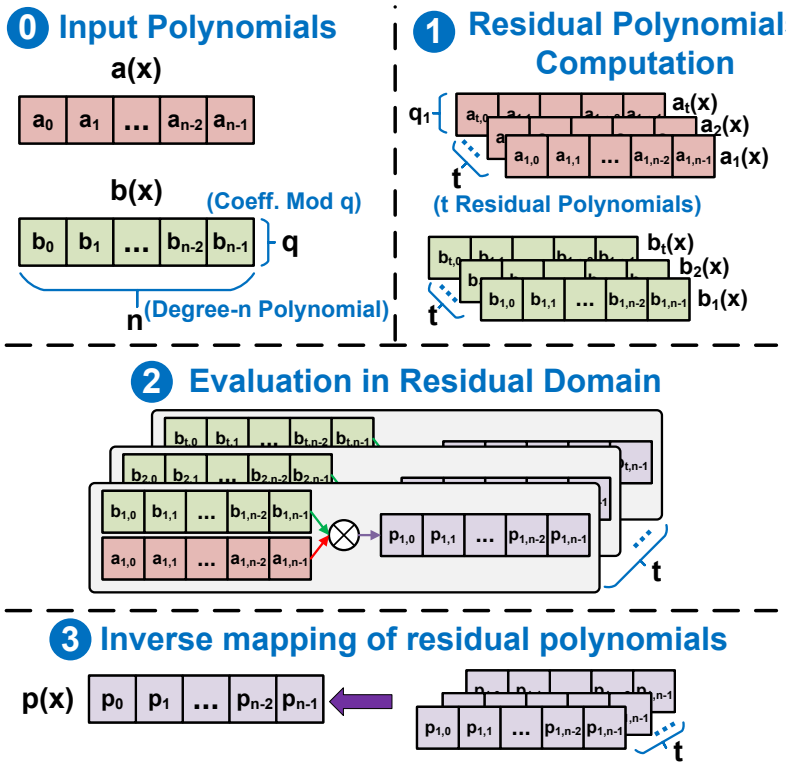

Figure 7.1: Overview of the use of residue arithmetic and CRT in the proposed PaReNTT architecture.

based on feedback architectures with loops for executing multiple operations in a time-multiplexed manner [25, 37, 105]. In particular, these prior works consider a unified architecture for the NTT and iNTT architecture, which is typically constructed from a memory-based or folded architecture framework. This design strategy can reduce the number of required processing elements (PEs). However, a feedback architecture requires feeding the intermediate results to the storage (memories or registers) and returning back to the input port of the architecture via the loops in multiple cycles. This method does not allow continuous loading of the new inputs since the input/output unit (I/O) is occupied by the intermediate results. Different from the prior works, our proposed architecture exploits a feed-forward and parallel architecture. Our proposed architecture has no feedback loop-

s/data paths. Therefore, the intermediate results can be executed and passed through to the next stage PE, directly.

The contributions of this chapter are four-fold and are summarized below.

- We propose a novel *parallel* NTT-based polynomial multiplier where the number of clock cycles to process the coefficients of the polynomials is inversely proportional to the level of parallelism. This real-time architecture requires a linear increase in area with respect to the level of parallelism.

- Our proposed architecture does not require intermediate shuffling operations. Different folding sets for the NTT and iNTT are used such that the product of the two NTTs can be processed immediately in the iNTT. This leads to a significant reduction in latency and complete elimination of intermediate buffer requirement.

- We propose a CRT-based implementation for the modular multiplication, which allows each NTT-based polynomial multiplication to operate over a small prime (co-prime factor). A special format of primes is also considered to reduce the cost of the implementation. Specifically, all the primes are not only NTT-compatible and CRT-friendly but also have low Hamming weights (i.e., these contain only a few signed power-of-two terms).

- Novel optimized architectures for pre-processing and post-processing for residue arithmetic are proposed; these architectures reduce area and power consumption. Finally, the low-cost pre-processing and post-processing blocks for the residue arithmetic are integrated into the parallel NTT-based modular polynomial multiplier to achieve high speed, low latency, and low area designs. In particular, optimizations in the pre-processing step can replace $(t-1)$ Barrett reduction units with only one Barrett reduction unit. The optimized post-processing step has a lower cost since it bypasses any expensive modular multipliers over a large modulus q.

The rest of this chapter is organized as follows: Section 7.2 reviews the mathematical background for the RNS and NTT-based polynomial modular multiplication and the corresponding hardware architectures in prior works. Section 7.3 presents a parallel architecture for the NTT-based polynomial multiplication that eliminates intermediate storage requirements. Then, Section 7.4 introduces our optimized RNS and CRT-based polynomial multiplier. The performance of our proposed architecture is presented and analyzed in Section 7.5. Finally, Section 7.6 concludes the chapter.

7.2 Background

7.2.1 Notation

For a polynomial ring $R_{n,q} = \mathbb{Z}_q[x]/(x^n+1)$, its coefficients have to be modulo q (i.e., these lie in the range $[0, q-1]$) and the degree of the polynomial is less than n (n is a power-of-two integer). To ensure all the intermediate results will not exceed such a polynomial ring, a modular reduction operation is needed, which is expressed as "mod (x^n+1, q)" or $[\circ]_q$. The polynomial of the ring $R_{n,q}$ is denoted as $a(x) = \sum_{j=0}^{n-1} a_j x^j$, where the j-th coefficient inside the polynomial $a(x)$ is represented as a_j.

The addition and multiplication of two polynomials modulo (x^n+1, q) (i.e., modular polynomial addition and multiplication) are written as $a(x)+b(x)$ and $a(x)\cdot b(x)$, respectively. We also use $\odot$ to denote the point-wise multiplication over (x^n+1, q) between two polynomials. Parameters $m = \log_2 n$ and $s \in [0, m-1]$ represent the total number of stages and the current stage in the NTT (iNTT), respectively.

7.2.2 Homomorphic Encryption and Residue Number System

HE allows the computations (e.g., multiplication, addition) directly on the ciphertext, without decryption, so that the users can upload their data to any (even untrusted) cloud servers while preserving privacy. The HE schemes can be broadly classified as fully HE (FHE) and somewhat HE (SHE). The FHE schemes allow an arbitrary number of homomorphic evaluations while suffering from high computational complexity [5]. SHE is an alternative solution with better efficiency than the FHE, which only allows performing a limited number of operations without decryption [16,19,39].

High-level steps for HE schemes can be summarized in four stages: key generation, encryption, evaluation, and decryption. In particular, the key generation step is used to output three keys: the secret key, public key, and relinearization key, based on the security parameter λ. Then, using the public key, the encryption algorithm encrypts a message into a ciphertext ct. During the evaluation step, a secure evaluation function performs a computation homomorphically for all input ciphertexts and outputs a new ciphertext ct' using the relinearization key. Finally, the result can be obtained using the secret key and ct' in the decryption step.

Key generation, encryption, and decryption steps are generally executed by the client. Meanwhile, the evaluation step is distributed to the cloud server for homomorphic computation. Different

homomorphic evaluation functions have different computational costs. The homomorphic addition is relatively simple since it is implemented by modular polynomial additions. However, homomorphic multiplication requires expensive modular polynomial multiplication. Thus the hardware or software accelerations for the modular polynomial multiplier, especially under the HE parameters with large degrees of polynomial and long word-length coefficients, are demanding.

As an example, performing a depth of four homomorphic multiplications with an 80-bit security level requires a 180-bit ciphertext modulus and degree-4096 polynomial in prior works [37]. However, the computation involving the long word-length coefficients is not trivial, which is also inefficient without high-level transformations. Since the moduli in most widely-used SHE schemes, e.g., BGV [19], BFV [16], CKKS [39], are not restricted to be primes, it is possible to choose each modulus to be a product of several distinct primes by using CRT, where each prime is an NTT-compatible prime with a small word-length.

The CRT algorithm decomposes q to $q_1, q_2, \ldots, q_t$ (i.e., $q = \prod_{i=1}^{t} q_i$, q_i's are mutually co-prime), and the ring isomorphism $R_q \equiv R_{q_1} \times R_{q_2}, \ldots, \times R_{q_t}$. After this decomposition, ring operation in each R_{q_i} is performed separately, which thus can be executed in parallel. From the implementation perspective, the larger the parameter t, the smaller each q_i and the simpler arithmetic operation over R_{q_i}.

7.2.3 Prior Hardware Implementations

Several hardware architectures based on CRT-based optimization have been proposed in [25, 37,105,106]. The works in [25,37] introduce an approximate CRT method for the BFV scheme, which involves the lifting and scaling operations to switch between a small ciphertext modulus q and a large ciphertext modulus Q. Later, a multi-level parallel accelerator utilizing the RNS and NTT algorithms is presented in [105]. Nevertheless, these works mainly focus on optimizing the NTT blocks but not on the CRT system's pre-processing and post-processing functional blocks.

7.3 Parallel NTT-based Polynomial Multiplier without Shuffling Operations

The long polynomial degree n can be in the range of thousands for the HE schemes, which becomes the bottleneck for the implementations in both software and hardware [35, 107]. Therefore, an efficient NTT-based polynomial multiplication method with a time complexity of $\mathcal{O}(n \log n)$ is used. This method significantly reduces the time complexity compared to the $\mathcal{O}(n^2)$ complexity method of the schoolbook polynomial multiplication along with the modular polynomial reduction.

7.3.1 NTT-based Polynomial Multiplication Using Negative Wrapped Convolution

The prior work in [33] presents an efficient algorithm for the NTT-based polynomial multiplication computing $p(x) = a(x) \cdot b(x) \mod (x^n + 1, q)$, namely negative wrapped convolution, as shown in Algorithm 14. Note that the weighted operations are needed before NTT and after iNTT during the negative wrapped convolution to avoid the expensive zero padding [33]. The core step of

Algorithm 14 Negative Wrapped Convolution [33]

Input: $a(x), b(x) \in R_{n,q}$
Output: $p(x) = a(x) \cdot b(x) \mod (x^n + 1, q)$
1: $\widetilde{a}(x) = \sum_{j=0}^{n-1} a_j \psi_{2n}^j x^j \mod q$
   $\widetilde{b}(x) = \sum_{j=0}^{n-1} b_j \psi_{2n}^j x^j \mod q$
2: $\widetilde{A}(x) : A_k = \sum_{j=0}^{n-1} \widetilde{a}_j \omega_n^{kj} \mod q,\ k \in [0, n-1]$
   $\widetilde{B}(x) : B_k = \sum_{j=0}^{n-1} \widetilde{b}_j \omega_n^{kj} \mod q,\ k \in [0, n-1]$
3: $\widetilde{P}(x) = \widetilde{A}(x) \odot \widetilde{B}(x) = \sum_{k=0}^{n-1} \widetilde{A}_k \widetilde{B}_k x^k$
4: $\tilde{p}(x) = n^{-1} \sum_{j=0}^{n-1} \tilde{P}_j \omega_n^{-kj} \mod q,\ k \in [0, n-1]$
5: $p(x) = \sum_{j=0}^{n-1} \widetilde{p}_j \psi_{2n}^{-j} x^j$

this algorithm is the NTT that converts the polynomials $a(x)$ and $b(x)$ to their NTT-domain $\widetilde{A}(x)$ and $\widetilde{B}(x)$ as in Step 2. The NTT for polynomial $a(x)$ is mathematically expressed as

$$\widetilde{A}_k = \sum_{j=0}^{n-1} a_j \psi_{2n}^j \omega_n^{kj} \mod q, \quad k \in [0, n-1]. \tag{7.1}$$

Polynomial $b(x)$ is similarly transformed to $\widetilde{B}(x)$. Specifically, ω is the primitive n-th root of unity modulo q (i.e., twiddle factor), which satisfies $\omega^n \equiv 1 \mod q$. ψ_{2n} is the primitive $2n$-th root of

unity modulo q, and thus $\omega = \psi_{2n}^2 \mod q$. After using the NTT algorithm, the efficient point-wise multiplication between $\widetilde{A}(x)$ and $\widetilde{B}(x)$ is performed, which is followed by the iNTT. The iNTT transforms product, $\widetilde{P}$, to the original algebraic domain polynomial $p(x)$, which is defined as

$$p_k = n^{-1}\psi_{2n}^{-k}\sum_{j=0}^{n-1}\widetilde{P}_j\omega_n^{-kj} \mod q, \quad k \in [0, n-1], \tag{7.2}$$

where n^{-1} is the modular multiplicative inverse of n with respect to modulo q.

During the NTT and iNTT, the weighted operation requires the multiplication of the polynomials by the weights $\psi_{2n}^j \mod q$ for NTT or $\psi_{2n}^{-j} \mod q$ for iNTT. Furthermore, an NTT-compatible prime is also utilized, i.e., q must satisfy that $(q-1)$ is divisible by $2n$.

Since the weighted operations in NTT/iNTT require a large number of expensive modular multiplications, the recent works in [73,108] present a new method to merge the weighted operations into the butterfly operations. In particular, the new NTT in Equation 7.1 is re-represented as $\widetilde{A}_k$ and $\widetilde{A}_{k+n/2}$ by using the decimation-in-time (DIT) method:

$$\widetilde{A}_k = a_k^{\diamond} + \psi_{2n}\omega_n^k a_k^{\square} \mod q, \tag{7.3}$$

$$\widetilde{A}_{k+n/2} = a_k^{\diamond} - \psi_{2n}\omega_n^k a_k^{\square} \mod q, \tag{7.4}$$

where $k \in [0, \frac{n}{2}-1]$ and

$$a_k^{\diamond} = \sum_{j=0}^{n/2-1} a_{2j}\psi_n^j\omega_{n/2}^{kj} \mod q, \tag{7.5}$$

$$a_k^{\square} = \sum_{j=0}^{n/2-1} a_{2j+1}\psi_n^j\omega_{n/2}^{kj} \mod q. \tag{7.6}$$

Since $\omega = \psi_{2n}^2 \mod q$, the integers ψ_{2n}^j and $\omega_{n/2}^{kj}$ can be merged as an integer $\psi_{2^{s+1}}^{(2j+1)n/2^s}$ in the s-th stage; Thus, only one multiplication is required in the butterfly operation, and the architecture is shown in Fig. 7.2.

The improved iNTT algorithm merges not only the weighted operation but also the multiplication with constant n^{-1} into the butterfly operations, as presented in [73]. Based on Equation 7.2

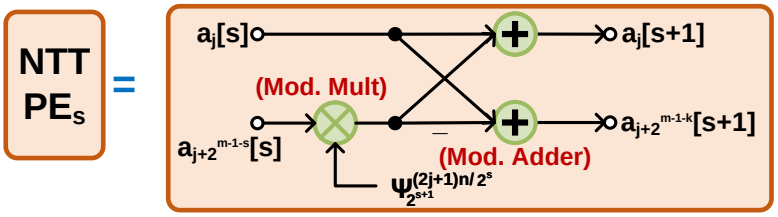

Figure 7.2: Architecture for DIT-based butterfly with merging the weighted operation in NTT.

and the decimation-in-frequency (DIF) method, the new iNTT algorithm is expressed as

$$p_{2k} = \frac{n}{2}^{-1} \psi_n^{-k} \sum_{j=0}^{n/2-1} \widetilde{P}_j^{even} \omega_{n/2}^{-kj} \mod q, \tag{7.7}$$

$$p_{2k+1} = \frac{n}{2}^{-1} \psi_n^{-k} \sum_{j=0}^{n/2-1} \widetilde{P}_j^{odd} \omega_{n/2}^{-kj} \mod q, \tag{7.8}$$

where $k \in [0, \frac{n}{2} - 1]$, and

$$\widetilde{P}_j^{even} = \frac{\widetilde{P}_j + \widetilde{P}_{j+n/2}}{2} \mod q \tag{7.9}$$

$$\widetilde{P}_j^{odd} = \frac{\widetilde{P}_j - \widetilde{P}_{j+n/2}}{2} \omega_n^{-j} \psi_{2n}^{-1} \mod q. \tag{7.10}$$

Similarly, the integers ω_n^{-j} and ψ_{2n}^{-1} are combined as an integer $\psi_{2^{s+1}}^{-(2j+1)n/2^s}$ in the s-th stage. Different from the NTT butterfly architecture, the modular addition and modular subtraction intermediate results in the iNTT butterfly need to be divided by two. Fig. 7.3 shows a hardware-friendly architecture, which only involves one left shift operation, one modular addition with constant $\frac{q+1}{2}$, and one multiplexer (MUX) for one modular division by two [71].

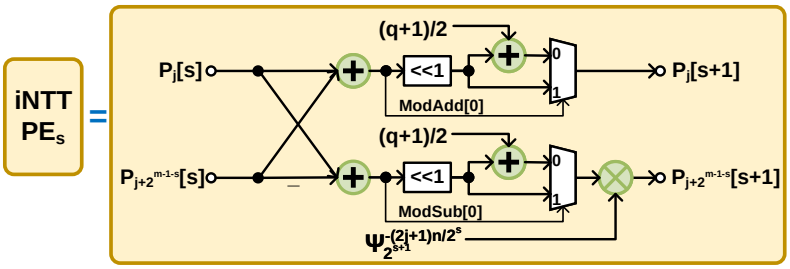

Figure 7.3: Architecture for DIF-based butterfly with merging the weighted operation in iNTT.

7.3.2 Two-Parallel Architecture

We propose a novel real-time, feed-forward, and parallel NTT-based polynomial multiplication architecture design that does not require intermediate shuffling, as shown in Fig. 7.4.

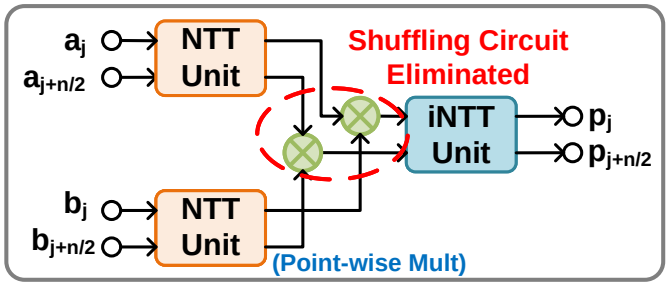

Figure 7.4: Architecture for the modular polynomial multiplier using two-parallel NTT and iNTT.

In particular, the NTT/iNTT units in Fig. 7.4 are based on a two-parallel architecture; these are derived using appropriate folding sets and the folding transformation [57, 58]. Fig. 7.5 and Fig. 7.6 show the data-flow graphs for 16-point forward NTT of $a(x)$ and iNTT for $P(x)$, respectively, where each circle represents one butterfly operation.

After applying the folding transformation, the operations in the same color are fed into the same PE and then executed in a time-multiplexed manner. The order in which the butterfly operations are executed in the same PE is referred to as the folding order. Also, the corresponding clock cycle for each butterfly operation is highlighted in blue in Fig. 7.5 and Fig. 7.6. In this 16-point example, the folding set (i.e., the ordered set of operations executed in each PE) of the forward NTT is expressed as:

$$\begin{aligned}
\mathcal{A} &= \{\mathcal{A}_0,\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3,\mathcal{A}_4,\mathcal{A}_5,\mathcal{A}_6,\mathcal{A}_7\} \\
\mathcal{B} &= \{\mathcal{B}_4,\ \mathcal{B}_5,\mathcal{B}_6,\ \mathcal{B}_7,\mathcal{B}_0,\ \mathcal{B}_1,\mathcal{B}_2,\ \mathcal{B}_3\} \\
\mathcal{C} &= \{\mathcal{C}_2,\ \ \mathcal{C}_3,\mathcal{C}_4,\ \ \mathcal{C}_5,\mathcal{C}_6,\ \ \mathcal{C}_7,\mathcal{C}_0,\ \ \mathcal{C}_1\} \\
\mathcal{D} &= \{\mathcal{D}_1,\mathcal{D}_2,\mathcal{D}_3,\mathcal{D}_4,\mathcal{D}_5,\mathcal{D}_6,\mathcal{D}_7,\mathcal{D}_0\}.
\end{aligned} \tag{7.11}$$

An additional shuffling circuit is typically used for reordering output data before computing iNTT. However, such a shuffling circuit requires a large number of clock cycles and registers.

In the proposed novel architecture, the two-parallel products are fed into a two-parallel iNTT architecture such that no intermediate buffer is needed. Thus, the outputs of the product

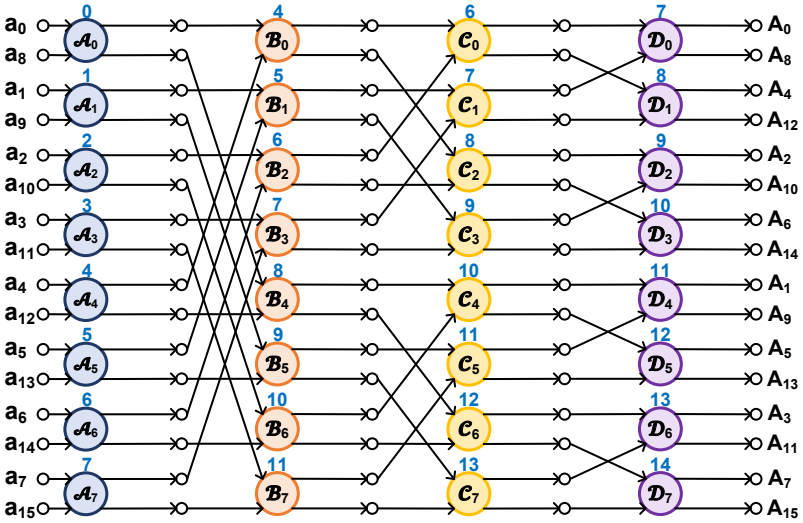

Figure 7.5: Data-flow graph of the 16-point forward NTT.

are executed immediately by the iNTT. This is possible as we can select different folding sets for the NTT and iNTT. It may be noted that reconfigurable memory-based NTT-based polynomial

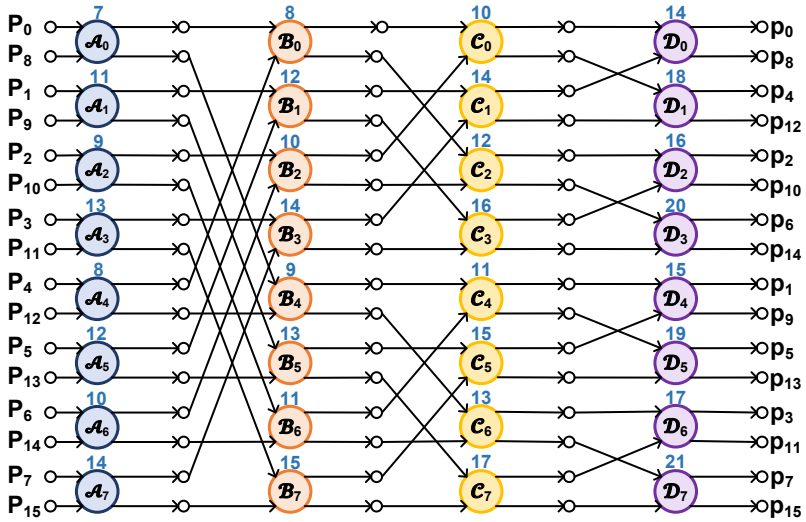

Figure 7.6: Data-flow graph of the 16-point iNTT.

multipliers have been published in [73, 89, 105, 109–111]. However, the intermediate results and the twiddle factors in the NTT/iNTT algorithm have data dependencies. As a result, the memory-based architectures easily have control/data hazards and cause bubbles in the pipeline, which will waste extra clock cycles [25].

In order to avoid intermediate buffer or data format conversion from NTT to iNTT, the

output samples from the last PE in the NTT unit should be fed into the first PE in the iNTT unit at the same clock cycle. This is achieved using the following folding set for the iNTT:

$$\begin{aligned}
\mathcal{A} &= \{\mathcal{A}_4,\mathcal{A}_2,\mathcal{A}_6,\mathcal{A}_1,\mathcal{A}_5,\mathcal{A}_3,\mathcal{A}_7,\ \mathcal{A}_0\} \\
\mathcal{B} &= \{\mathcal{B}_0,\ \mathcal{B}_4,\mathcal{B}_2,\ \mathcal{B}_6,\mathcal{B}_1,\ \mathcal{B}_5,\mathcal{B}_3,\ \ \mathcal{B}_7\} \\
\mathcal{C} &= \{\mathcal{C}_3,\ \ \mathcal{C}_7,\mathcal{C}_0,\ \ \mathcal{C}_4,\mathcal{C}_2,\ \ \mathcal{C}_6,\mathcal{C}_1,\ \ \ \mathcal{C}_5\} \\
\mathcal{D} &= \{\mathcal{D}_2,\mathcal{D}_6,\mathcal{D}_1,\mathcal{D}_5,\mathcal{D}_3,\mathcal{D}_7,\mathcal{D}_0,\mathcal{D}_4\}.
\end{aligned} \tag{7.12}$$

The NTT and iNTT designs are inspired by the design of parallel FFT architectures based on folding sets [59,112]. Parallel NTT architectures based on folding sets was presented in our earlier work [10]. The NTT architecture in Fig. 7.7 is derived using the folding sets shown in Equation 7.11. Specifically, this architecture has four PEs and three delay-switch-delays (DSDs), where the structures for PE and DSD are illustrated in Fig. 7.2 and Fig. 7.8. Besides, the DSD block utilizes two MUXs and two register sets, such that it can store the specific data in the data-path and then either switch or pass the data to the PE. Note that the number of registers inside each register set is varied in different stages. In the s-th stage, each register set has 2^{m-s-2} registers in the DSD block for the NTT architecture.

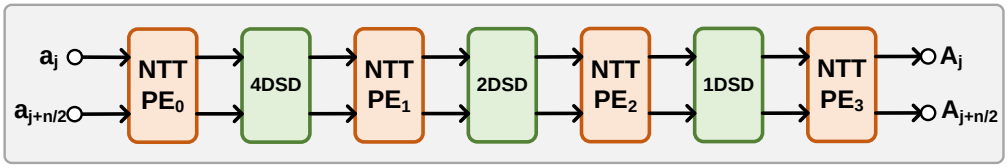

Figure 7.7: Architecture of the 16-point forward NTT unit.

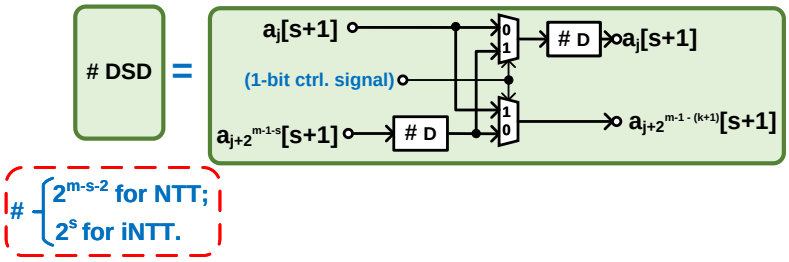

Figure 7.8: Architecture for DSD unit.

Furthermore, the architecture for iNTT is shown in Fig. 7.9, and its components are de-

scribed in Fig. 7.3 and Fig. 7.8. One of the main differences between NTT and iNTT architectures is the number of registers located inside each DSD block since they are determined by the folding set as in Equation 7.12. Specifically, 2^s registers are required for each register set in the s-th stage for the iNTT architecture. Even though the operations of NTT and iNTT are very similar, we consider two separate architectures instead of considering a unified and reconfigurable architecture. The rationale is as follows. Since modular multiplications are heavily used in homomorphic multiplication, using two different architectures for NTT and iNTT allows a continuous flow of the input polynomials and thus can highly accelerate the HE multiplication.

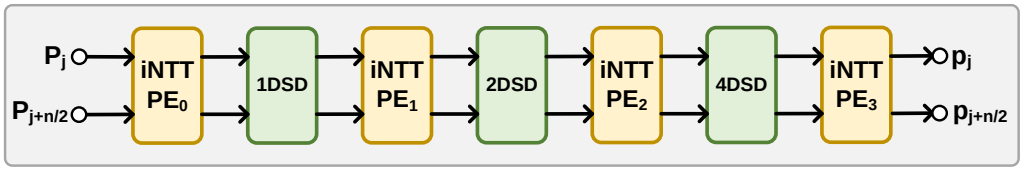

Figure 7.9: Architecture of the 16-point iNTT unit.

The 16-point architectures in Fig. 7.7 and Fig. 7.9 can also be easily generalized to any power-of-two length n by having m PEs and $(m-1)$ DSDs blocks. Furthermore, the general case NTT and iNTT folding sets are defined as follows. We denote the PE in s-th stage as PE_s, and the NTT folding set for the butterfly operations performing inside this PE is expressed in Table 7.1. The entries in the Table describe the node index of the node of that stage in the data-flow graph. The folding order describes the clock partition at which the node is executed. For example, a folding order s implies that the node is executed at clock cycle $(n/2)l + s$ where l is an integer. The cardinality of the folding set is $n/2$ as there are $n/2$ operations (nodes) in an NTT stage. Thus the scheduling period is $n/2$.

Table 7.1: Generalized folding order for NTT

Folding Order	0	1		l		$\frac{n}{2}-1$
PE_0	0	1	...	l	...	$\frac{n}{2}-1$
PE_1	2^{m-2}	$2^{m-2}+1$	...	$2^{m-2}+l \mod \frac{n}{2}$	...	$2^{m-2}-1$
				...		
PE_s	2^{m-s-1}	$2^{m-s-1}+1$	...	$2^{m-s-1}+l \mod \frac{n}{2}$	...	$2^{m-s-1}-1 \mod \frac{n}{2}$
				...		
PE_{m-1}	1	2	...	$l+1$	...	0

The folding set for iNTT can also be generalized as in Table 7.2, where the symbol $\langle\circ\rangle$ means the bit-reverse representation for the folding set with respect to a $(m-1)$-bit integer (e.g.,

$\langle 1 \rangle = \langle 001_b \rangle = 100_b = 4$ when $m = 4$). Specifically, if a node i in the NTT has folding order i, the folding order of the corresponding node in iNTT is $\langle i \rangle - 1$ modulo $(n/2)$. While the bit-reversed scheduling has been known to eliminate latency and buffer requirements at the data-flow graph level, the observation that the same property holds in a parallel NTT-iNTT cascade is non-intuitive and new.

Note that if the iNTT was designed using the same folding set in Equation 7.11, the product would need to be input to a DSD of size 4 ($n/4$ in general). This would introduce an additional latency of 4 ($n/4$ in general) clock cycles. The use of different folding sets for NTT and iNTT eliminates any additional DSD circuit and its associated latency.

Table 7.2: Generalized folding order for iNTT

Folding Order	0	1	...	l	...	$\frac{n}{2}-1$
PE_0	$\langle 1 \rangle$	$\langle 2 \rangle$	...	$\langle l+1 \rangle$	...	$\langle 0 \rangle$
PE_1	$\langle 0 \rangle$	$\langle 1 \rangle$	...	$\langle l \rangle$	...	$\langle 2^{m-1}-1 \rangle$
				...		
PE_s	$\langle 2-2^s \mod \frac{n}{2} \rangle$	$\langle 2-2^s+1 \mod \frac{n}{2} \rangle$	...	$\langle 2-2^s+l \mod \frac{n}{2} \rangle$	...	$\langle 2-2^s-1 \mod \frac{n}{2} \rangle$
				...		
PE_{m-1}	$\langle 2 \rangle$	$\langle 3 \rangle$	...	$\langle l+2 \mod \frac{n}{2} \rangle$	...	$\langle 1 \rangle$

7.4 Moduli Selection and Architectures for Pre-Processing and Post-Processing for CRT

The CRT and NTT-based modular polynomial multiplication can be divided into three stages: pre-processing, NTT-based polynomial multiplication over R_{n,q_i}, and post-processing, with high-level architecture shown in Fig. 7.10.

7.4.1 Special NTT-Compatible and CRT-Friendly Primes Selection

As opposed to the prior works that randomly select the co-primes, this work studies and utilizes the property of the special co-primes to reduce the computational cost and the silicon area. The main idea of this optimization is to trade the flexibility of the co-primes selection for the timing/area performance of the architectures.

In the proposed architecture, each q_i not only needs to be an NTT-compatible prime but

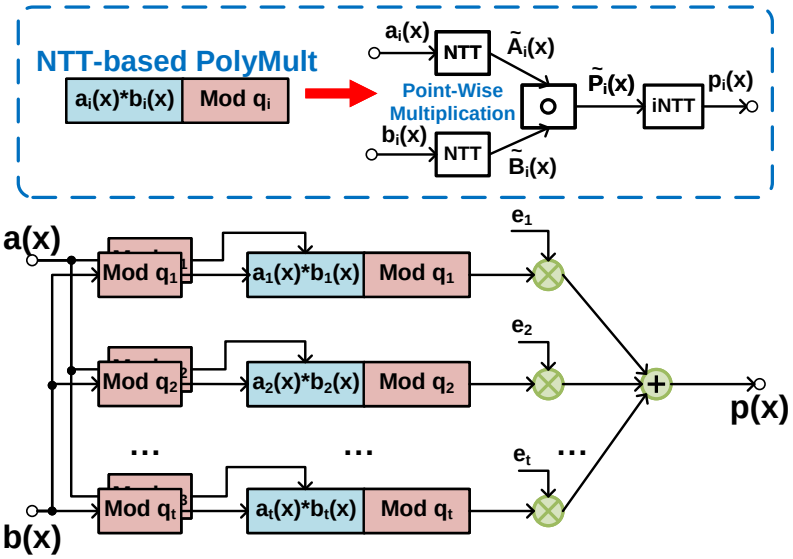

Figure 7.10: High-level block diagram of CRT and NTT-based modular polynomial multiplication.

also has a short word-length, which is defined as

$$q_i = 2^v - \beta_i, \quad \beta_i = 2^{v_{1i}} \pm 2^{v_{2i}} \pm 2^{v_{3i}} \pm \ldots - 1, \tag{7.13}$$

where v is the word-length of q_i. $\lceil \frac{v-1}{t-1} \rceil > v_{1i} > v_{2i}$, which can ensure q_i to be also CRT-friendly for our later optimization. A CRT-friendly prime leads to an optimized hardware architecture with respect to the overall timing and area performance for the pre-processing and post-processing steps. Our exhaustive approach generates q_i that are similar to the Solinas prime, and contain a few signed power-of-two terms [11, 67].

The integer multipliers have a larger area consumption and longer delay than the integer adders for the hardware implementation. Besides, the area and delay are proportional to the word-length. Therefore, one possible direction to optimize the modular multiplier, pre-processing stage, and post-processing stage architectures is to reduce the number of integer multipliers, especially the long integer multipliers. In the proposed approach, all the integer multipliers are eliminated when multiplying by q_i, which significantly reduces the computation cost.

7.4.2 Residual Polynomials Computation Unit

The pre-processing stage maps the input polynomials to their residual polynomials by applying the CRT algorithm. For the polynomial $a(x)$, its residual polynomials are

$$a_i(x) = [a(x)]_{q_i} = \sum_{j=0}^{n-1} (a_{i,j} \bmod q_i) x^j, \quad i \in [1, t]. \tag{7.14}$$

The same method is applied to the polynomial $b(x)$ to obtain its residual polynomials.

However, this process is not trivial since the word-length of q is much larger than that of q_i. One of the key steps in the pre-processing stage is modular reduction. While Barrett reduction is widely used in modular reduction algorithms for the HE schemes [56], it cannot be directly used in this process since the input for the *efficient* Barrett reduction algorithm has to be smaller than q_i^2 when q_i is the modulus. Note that the original Barrett reduction in [56] utilizes a *while*-loop at the end of the algorithm so there is no restriction for the input word-length. However, it is inefficient for hardware implementation. A more popular hardware implementation method uses an efficient Barrett reduction that replaces the *while*-loop by a simple *if-else* statement and restricts the input word-length. To utilize the efficient Barrett reduction algorithm for the residual polynomial computation, further transformations in the pre-processing stage are needed.

Algorithm 15 presents a novel and hardware-friendly optimization to implement Equation 7.14. For a large integer a_j, the first step is to split it into several segments where each segment has v bits (v is the word-length of q_i). We define the base $B = 2^v$. Thus, each segment in $a_{i,j}$ can be represented as $z_k \cdot B^k$, $k \in [0, t-1]$. The second step performs the modular reduction for each term, which is the main focus of our hardware optimization.

Algorithm 15 Efficient residual coefficient computation

Input: $a_j \in [0, q-1]$, q_i, and $B = 2^v$ ($v = \lceil \log_2(q_i) \rceil$) **Output:** $a_{i,j} = a_j \mod q_i$, $a_{i,j} \in R_{q_i}$

1: $a_j = z_0 + z_1 \cdot B + z_2 \cdot B^2 +, ..., + z_{t-1} \cdot B^{t-1}$ (Step 1)
2: **for** $k = 1$ **to** $t - 1$ **do**
3: $\quad r_k = z_k \times \beta_i^k \quad$ // $\beta_i = B \mod q_i$ (Step 2)
4: **end for**
5: $a_{i,j} = z_0 + r_1 + ... + r_{t-1} \mod q_i$ (Step 3)

Since we consider a special q_i, Step 2 in Algorithm 15 no longer requires $v \times v$-bit integer multiplication with β_i^k to obtain each r_k. Thus, our proposed method eliminates the expensive modular multiplications.

Besides, different from the prior work [25] where the Barrett reductions are required to reduce each r_k modulo q_i in Step 2, our design reduces $(t-1)$ Barrett reduction units to only one required in Step 3. The rationale behind this is as follows. The product r_k in the prior work is a large integer that approximately equals q_i^2, as β_i^k is any constant modulo q_i. In contrast, since the special q_i is utilized in our proposed design, all the β_i^k are small integers, guaranteeing that all the products, r_k, are sufficiently small.

Since q_i only contains only a few signed power-of-two terms, no integer multiplication is required in Step 2. For example, for a special prime $q_i = 2^v - 2^{v_{1i}} - 2^{v_{2i}} + 1$, β_i in Step 2 can be expressed as

$$\beta_i = [2^v]_{q_i} \equiv 2^{v_{1i}} + 2^{v_{2i}} - 1. \tag{7.15}$$

Note that more signed-power-two terms can be added to accommodate the desired number of moduli at the cost of an additional adder/subtractor per term.

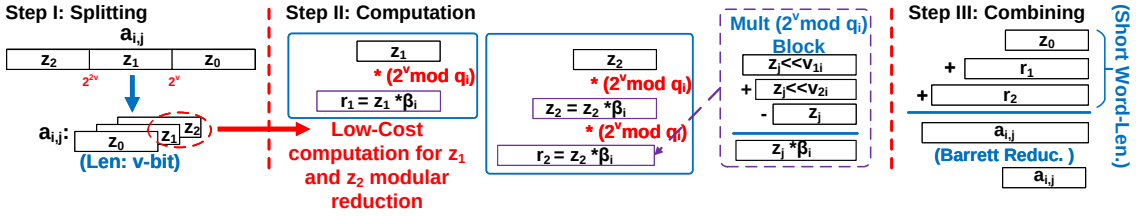

Figure 7.11: Flow chart for the residual coefficient computation unit when $t = 3$.

We use a flow chart for $t = 3$ as an example (illustrated in Fig. 7.11) to show the overall computation based on this efficient algorithm. It can be seen that the modular multiplication in $z_k \times \beta_i^k$ in [25] (Fig. 7.12(a)) can be replaced by the shift and add operations, which can reduce the hardware cost. When k becomes larger, a deeper shift and add unit (SAU) is required, as shown in Fig. 7.11 and Fig. 7.12(b). However, since a multiplier is typically quadratically more expensive than an adder with respect to word-length, using such shift and add operation is still much more efficient than using a multiplier to obtain its result r_k. In our CRT-friendly q_i, v_{1i} is at most equal to $(\lceil \frac{v-1}{t-1} \rceil - 1)$-bit to ensure r_k is smaller than q_i^2, as shown in Fig. 7.12(b). Thus, since the operating segments r_k are still represented in short word-lengths, combining all the r_k and z_0 to obtain $a_{i,j}$ only requires adders and a Barrett reduction unit.

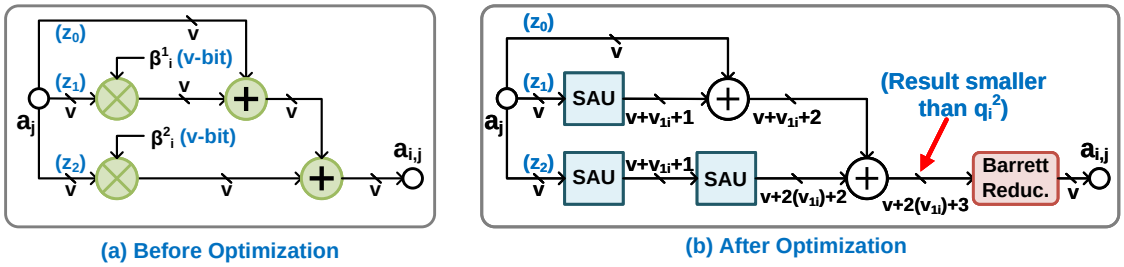

Figure 7.12: Top-level architecture of residual coefficient computation unit when $t = 3$.

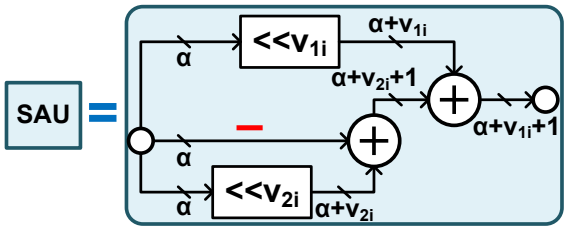

Figure 7.13: SAU unit of residual coefficient computation unit when $t = 3$ whose input word-length is α.

7.4.3 Increasing the Number of Primes When Needed

Table 7.3 shows the number of special NTT-compatible and CRT-friendly primes that can be found by exhaustive search when 3, 4, and 5 signed power-of-two-terms are considered when $n = 1024$, and each co-prime factor is 48-bit. It can be noticed that the number of generated co-prime factors is not enough for constructing a modulus with four co-prime factors when $n = 1024$ or $n = 4096$ in Table 7.3. However, this bottleneck can be overcome by either using more signed power-of-two terms to construct the co-prime factors or using an additional Barrett reduction. Using the latter method, the co-primes factors for $t = 3$ can also be used in the $t = 4$ case. This approach is used in the experimental evaluation in the chapter.

Table 7.3: The number of special NTT-compatible and CRT-friendly primes under different settings when $v = 48$

Parameter Setting	3 terms	4 terms	5 terms
$t = 2$, $n = 1024$, 1 BR	3	98	1501
$t = 3$, $n = 1024$, 1 BR	2	10	34
$t = 4$, $n = 1024$, 1 BR	0	0	0
$t = 4$, $n = 1024$, 2 BR	2	10	34
$t = 2$, $n = 4096$, 1 BR	3	87	1235
$t = 3$, $n = 4096$, 1 BR	2	8	20
$t = 4$, $n = 4096$, 1 BR	0	0	0
$t = 4$, $n = 4096$, 2 BR	2	8	20

Fig. 7.14 shows an example for the residual polynomials computation unit with more flexible co-prime factors selection when $t = 4$, $\log_2(q_i) = 48$, and $n = 1024$. Utilizing an additional Barrett reduction unit (inside the green box) can reduce the word-length of the intermediate results, so the co-prime factors for $(t-1)$ in Table 7.3 can be employed. Note that the overhead of this alternative solution is only one Barrett reduction unit and one modular adder, compared to the architecture in Fig. 7.12(b) when it is in the same parameter setting. Nevertheless, this alternative solution still requires fewer hardware resources than the conventional design in Fig. 7.12(a) since the number of Barrett reduction units is lower and no integer multiplier is required.

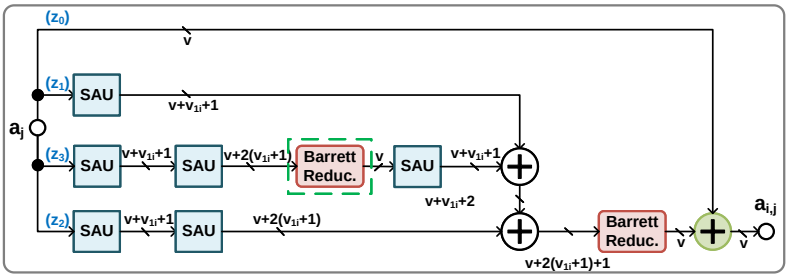

Figure 7.14: Residual coefficient computation unit with additional Barrett reduction unit when $t = 4$.

7.4.4 Evaluation in Residual Domain

After using CRT representation, the function $f(a_i(x), b_i(x))$ over R_{n,q_i} can be computed independently. As a result, the overall t operations can be executed in parallel. In our case, the function computes the residual products $p_i(x)$ for $i \in [1, t]$, by utilizing NTT-based polynomial multiplication over R_{n,q_i}. The architecture to compute $p_i(x) = a_i(x) \cdot b_i(x) \mod (x^n + 1, q_i)$ is based on our novel NTT-based polynomial multiplier in Fig. 7.4. Thus, our proposed architecture achieves high throughput and low latency by increasing the parallelism from the CRT representation.

7.4.5 Inverse Mapping of Residual Coefficients of Polynomials

The results obtained by the evaluation in the residual domain need to be converted back to over the ring $R_{n,q}$, which is the same as $f(a(x), b(x))$ over $R_{n,q}$ (i.e., result computed without using CRT representation).

This post-processing stage is based on the inverse CRT algorithm, which is

$$\begin{aligned} p(x) &= \sum_{i=1}^{t} p_i(x) \cdot e_i \bmod q \\ &= \sum_{i=1}^{t} \sum_{j=0}^{n-1} p_{i,j} \cdot e_i \cdot x^j \bmod q, \end{aligned} \tag{7.16}$$

where each $e_i = q_i^* \cdot \tilde{q}_i$ is a constant, $q_i^* = (\frac{q}{q_i}) \in \mathbb{Z}$, and $\tilde{q}_i = [(\frac{q}{q_i})^{-1}]_{q_i} \in \mathbb{Z}_{q_i}$.

However, direct multiplication by the constant e_i involves a long integer multiplication and expensive modular reduction over q (tv), which will result in an inefficient implementation and a long critical path. Meanwhile, the properties of the special co-primes can lower the cost of modular operations over q_i in the post-processing stage. Therefore, we leverage the technique in [113] to further express Equation 7.16 as:

$$\begin{aligned} p(x) &= \sum_{i=1}^{t} \left[p_i(x) \cdot \tilde{q}_i\right]_{q_i} \cdot q_i^* \bmod q \\ &= \sum_{i=1}^{t} \sum_{j=0}^{n-1} \left[p_{i,j} \cdot \tilde{q}_i\right]_{q_i} \cdot q_i^* \cdot x^j \bmod q. \end{aligned} \tag{7.17}$$

Note that the computation in $0 \leq \left[p_{i,j} \cdot \tilde{q}_i\right]_{q_i} < q_i$ can be performed efficiently since the modular reduction over q_i has a lower cost than the modular reduction q. As q_i^* is a $(t-1)v$-bit pre-computed constant, no division is required in the post-processing stage. Besides, the range of the coefficients from $\left[p_i(x) \cdot \tilde{q}_i\right]_{q_i} \cdot q_i^*$ is in $[0, q-1]$ so that no modular multiplication is required to compute the product.

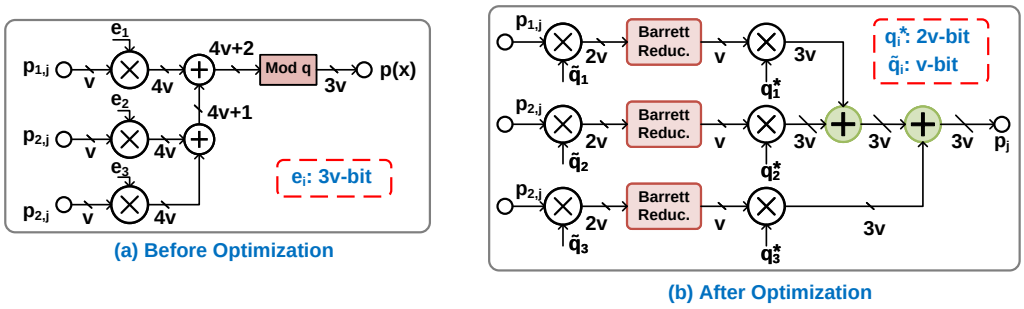

Figure 7.15: Inverse mapping architecture when $t = 3$. This circuit illustrates the post-processing step for the inverse CRT.

The optimized architecture of the inverse mapping of residual coefficients of polynomials is

shown in Fig. 7.15(b) (we use $t = 3$ as an example). In this architecture, each long word-length ($3v \times v$-bit) multiplier for multiplying e_i is split into $v \times v$-bit multiplier with constant $\tilde{q}_i$ and $v \times 2v$-bit multiplier with constant q_i^*. Instead of implementing an expensive modular reduction over a large modulus q block in Fig. 7.15(a), only two modular adders and three modular reductions over q_i are required to obtain the final result $p(x)$. Specifically, the modular reduction over q_i can reuse the same Barrett reduction block from the pre-processing stage, which is also efficient based on the special co-prime.

Overall, the proposed novel architecture can significantly reduce the area and power consumption.

7.5 Experimental Results

To evaluate the performance of our proposed design, we first introduce the experimental result of our proposed NTT-based polynomial multiplier, which is based on Section 7.3 without the CRT representation as the baseline design for the comparison. Then, the performance of PaReNTT architecture in Section 7.4 (based on the CRT and NTT-based polynomial multiplier) is presented. The proposed designs are implemented using SystemVerilog and then mapped to Xilinx Virtex Ultrascale FPGA board (XCVU440-1FLGA2892C, 20nm FinFET node).

We consider three different word-lengths for q (96-bit, 144-bit, and 192-bit) with the same degree of polynomial $n = 1024$ and $n = 4096$ to investigate the designs under different levels of CRT-based parallelism. Specifically, all the decomposed co-prime factors are in 48-bit special NTT-compatible and CRT-friendly format, while the designs without the CRT representation directly utilize the 96-bit, 144-bit, and 192-bit NTT-compatible primes. Note that our design can be easily extended to a longer word-length modulus by either having more co-primes or increasing the word length of each co-prime. Furthermore, each degree-1024 NTT-based polynomial multiplier has 30 PEs and 27 DSD units since $m = \log_2(1024) = 10$. A degree-4096 NTT-based polynomial multiplier applies 36 PEs and 33 DSD units since $m = \log_2(4096) = 12$. Note that a higher degree of the polynomial can be applied, which only requires more PEs and DSDs.

7.5.1 Evaluation Metrics and Performance of Parallel NTT-based polynomial multiplier for base Modulus

We first describe the evaluation metrics used in this work. We then evaluate the performance of our NTT-based polynomial multiplier that does not require any shuffling operations.

To analyze the timing performances of the implementations, we define two timing performance metrics, *block processing period* (BPP) and latency. BPP is defined as the time required to process n coefficient inputs or the time required to generate n coefficient outputs. For a degree-n NTT-based two-parallel polynomial multiplier, the expression for BPP is

$$T_{BPP} = n/2, \tag{7.18}$$

where the throughput is two samples per clock cycle. In addition, the latency for one modular polynomial multiplication is

$$T_{Lat} = (n-2) + T_{pipe}, \tag{7.19}$$

where T_{pipe} represents the additional pipelining stages added to the data-path in order to reduce the critical path. Furthermore, the total clock cycles consumed by L modular polynomial multiplications are

$$T_{total} = (T_{BPP} + T_{Lat} - 1) \cdot L. \tag{7.20}$$

For $n = 1024$, the BPP is 512 clock cycles, and the latency is 1,126 clock cycles (including extra clock cycles required for pipelining). The latency is significantly reduced compared to the NTT-based polynomial multipliers that use a shuffling circuit in the prior works. The comparison of our optimized and conventional methods (without considering the pipelining) is shown in Fig. 7.16. Specifically, the conventional method with the shuffling circuit needs additional 256 ($n/4$ in general) clock cycles for the re-ordering, leading to an increase in latency by around 20.0% for two-parallel design and $n = 1024$.

7.5.2 Evaluation on PaReNTT polynomial multiplier

This section considers the implementation of the two-parallel residue arithmetic-based NTT architecture for $n = 1024$ and $n = 4096$. The performances of the two-parallel architecture without

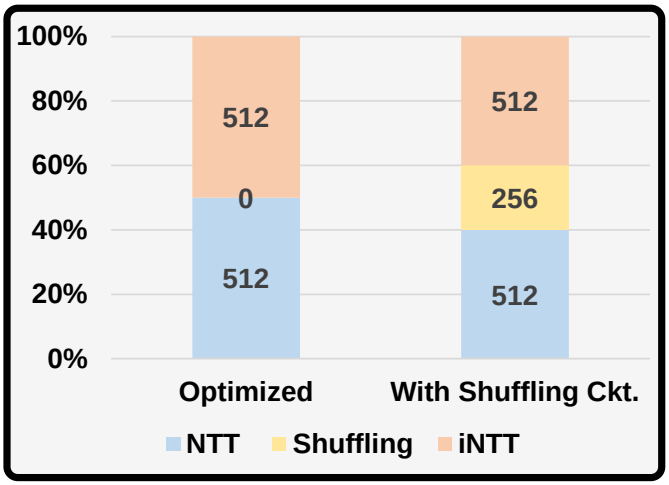

Figure 7.16: Comparison of latency of Two-parallel NTT-based polynomial multiplication with and without shuffling operations when $n = 1024$.

and with residue arithmetic (i.e., PaReNTT architecture design) are compared.

For the CRT case, the co-prime factors are 48 bits long (i.e., $v = \log_2(q_i) = 48$), where each co-prime factor has four signed power-of-two terms. Also, the number of co-prime factors, t, increases from 1 to 4.

Tables 7.4 and 7.5, respectively, describe the area and speed (latency) comparisons for a 2-parallel architecture with and without CRT when $n = 1024$. Four word-lengths are considered: 48, 96, 144, and 192 bits. Similarly, the experimental result and comparison when $n = 4096$ are presented for the same word-lengths in Tables 7.6 and 7.7.

Table 7.4: Area consumption and frequency for modular polynomial multipliers when $n = 1024$

$\lceil \log_2 q \rceil$	t	CRT	Freq.[MHz]	LUTs	DSPs	FFs
48	1	No	207	59,426 (2.4%)[a]	288 (10.0%)	15,419 (0.3%)
96	1	No	151	194,016 (7.7%)	1,152 (40.0%)	39,587 (0.8%)
96	2	Yes	199	142,014 (5.6%)	612 (21.3%)	35,655 (0.7%)
144	1	No	54	431,623 (17.0%)	2,080 (72.2%)	62,240 (1.2%)
144	3	Yes	168	221,183 (8.7%)	972 (33.8%)	60,228 (1.2%)
192	1	No	13	1,402,022 (55.4%)	1,696 (58.9%)	145,955 (2.9%)
192	4	Yes	113	323,209 (12.8%)	1,344 (46.7%)	87,150 (1.7%)

[a]: # of used resources (% utilization) on FPGA board.

When the word-length increases without using residue arithmetic, the clock frequencies for the FPGA implementation are reduced, as shown in Tables 7.5 and 7.7. For example, increasing the

Table 7.5: Timing performance for modular polynomial multipliers when $n = 1024$

$\lceil \log_2 q \rceil$	t	CRT	BPP[b]		Latency[c]		ABP (LUT)[d]	ABP (DSP)[e]
			# Cycles	Period [μs]	# Cycles	Period [μs]		
48	1	No	512	2.5	1,126	5.4	0.1M	0.7K
96	1	No	512	3.4	1,126	7.5	0.7M	3.9K
96	2	Yes	512	2.6	1,142	5.7	0.4M	1.6K
144	1	No	512	9.5	1,126	20.9	4.1M	19.7K
144	3	Yes	512	3.0	1,152	6.9	0.7M	3.0K
192	1	No	512	40.7	1,126	89.5	57.0M	69.0K
192	4	Yes	512	4.5	1,152	10.2	1.5M	6.1K

[b]: Block processing period (BPP) is the period (μs) for processing n coefficient inputs or for generating n sample outputs after the first sample out. [c]: Latency is the period (μs) of the first sample in and the first sample out. [d]: ABP (LUT) is calculated from the number of LUTs times BPP (μs). [e]: ABP (DSP) is calculated from the number of DSPs times BPP (μs).

word-length from 48-bit to 96-bit results in a 27.1% and 35.6% longer critical path for $n = 1024$ and $n = 4096$, respectively. Thus, for high-speed applications, the long word-length modulus architecture without the use of residue arithmetic is inefficient.

Table 7.6: Area consumption and frequency for modular polynomial multipliers for $n = 4096$

Design	$\lceil \log_2 q \rceil$	t	CRT	Freq.[MHz]	LUTs	DSPs	FFs
Ours	48	1	No	216	100,681 (4.0%)	342 (11.9%)	20,087 (0.4%)
	96	1	No	139	290,003 (11.5%)	1,368 (47.5%)	48,540 (1.0%)
	96	2	Yes	196	224,964 (8.9%)	720 (25.0%)	43,556 (0.9%)
	144	1	No	47	597,815 (23.6%)	2,470 (85.8%)	69,992 (1.4%)
	144	3	Yes	168	349,720 (13.8%)	1,134 (39.4%)	71,778 (1.4%)
	192	1	No	10	1,965,433 (77.6%)	1,752 (60.8%)	174,852 (3.5%)
	192	4	Yes	111	488,112 (19.3%)	1,632 (56.7%)	101,385 (2.0%)

We now compare the timing performance of the designs with and without residue arithmetic based on the experimental results in Table 7.5 for $n = 1024$ and Table 7.7 for $n = 4096$. Note that the implementations with and without the CRT representation have the same stage of pipelining for a fair comparison of the timing performance. When $n = 1024$, the latency of PaReNTT architectures with two, three, and four co-prime factors are reduced by 24.0%, 67.0%, and 88.6% compared to without residue arithmetic implementation. Similarly for $n = 4096$, the latency in the PaReNTT architectures are reduced by 28.5%, 71.0%, and 90.4% for q equal to 96-bit, 144-bit, and 192-bit, respectively. The trends of the delay variation with respect to different word-lengths for PaReNTT architecture design are illustrated in Fig. 7.17 for $n = 1024$. When the word-length of moduli becomes larger, the logic delay of the CRT-representation architectures remains almost the same as the 48-bit NTT-based polynomial multiplier, while the routing delays increase slightly. Thus, the

Table 7.7: Timing performance for modular polynomial multipliers for $n = 4096$

$\lceil \log_2 q \rceil$	t	CRT	BPP		Latency		ABP	ABP
			# Cycles	Period [μs]	# Cycles	Period [μs]	(LUT)	(DSP)
48	1	No	2048	9.5	4,218	19.5	1.0M	3.2K
96	1	No	2048	14.7	4,218	30.2	4.3M	20.1K
96	2	Yes	2048	10.4	4,234	21.6	2.3M	7.5K
144	1	No	2048	43.3	4,218	87.2	25.9M	106.8K
144	3	Yes	2048	12.2	4,244	25.3	4.3M	13.8K
192	1	No	2048	197.8	4,218	398.8	388.8M	346.6K
192	4	Yes	2048	18.5	4,244	38.4	9.0M	30.2K

clock frequencies in CRT-based architectures are similar. However, the delay for the architectures without CRT increases proportionally with respect to the word-length of q.

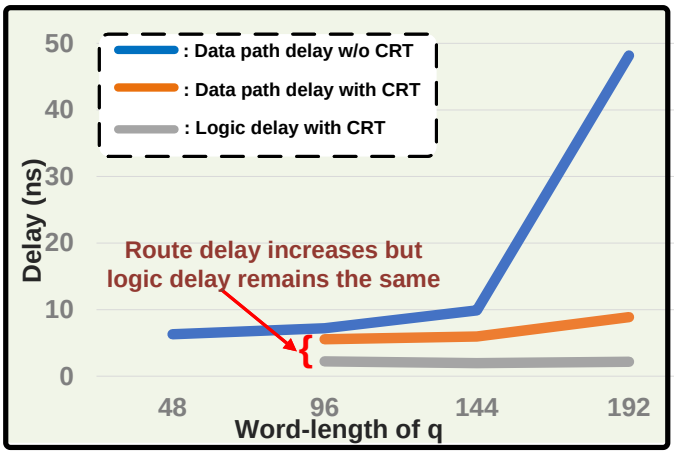

Figure 7.17: Delay variation in different word-lengths q when $n = 1024$.

We also examine the area performance of PaReNTT architectures and the implementations without residue arithmetic. The results are presented in Tables 7.4 and 7.6 for $n = 1024$ and $n = 4096$, respectively. Table 7.4 shows the residue arithmetic-based architectures for $n = 1024$ require 26.8%, 48.8%, and 76.9% fewer LUTs when q is a 96-bit, 144-bit, and 192-bit moduli, compared to the implementations without residue arithmetic. For $n = 4096$, residue arithmetic-based architectures require 22.4%, 41.5%, and 75.2% less LUTs for 96-bit, 144-bit, and 192-bit moduli, respectively. Furthermore, the residue arithmetic-based architectures require 46.9%, 53.3%, and 20.8% fewer DSPs for 96-bit, 144-bit, and 192-bit moduli for $n = 1024$, compared to the implementations without residue arithmetic. Similarly, residue arithmetic-based architectures reduce 47.4%, 54.1%, and 6.8% DSP utilization for 96-bit, 144-bit, and 192-bit moduli, respectively for

$n = 4096$.

We now compare the area-BPP product (ABP) for the PaReNTT architectures and the implementation without residue arithmetic to evaluate the timing and area performance together. Comparing the ABP (LUT), the reductions achieved by the PaReNTT architectures with two, three, and four co-prime factors are 42.9%, 82.9%, and 97.4%, responsively for $n = 1024$. When the degree of polynomial increases to 4096, the reductions for ABP (LUT) are 46.5%, 83.4%, and 97.7%, respectively. When further comparing the ABP (DSP) in a similar way for $n = 1024$, the PaReNTT architectures achieve 59.0%, 84.8%, and 91.2% reductions, respectively. For the degree-4096 case, ABP (DSP) savings in the PaReNTT architectures are 46.5%, 83.4%, and 97.6%, respectively.

7.6 Conclusion

This chapter has proposed PaReNTT, an efficient CRT and NTT-based long polynomial multiplier. This design leverages the characteristics of the specially selected primes to optimize the pre-processing and post-processing units for the CRT algorithm. In addition, a novel iNTT unit is designed based on bit-reversed scheduling to eliminate an expensive shuffling circuit and significantly reduce latency. Future work will be directed toward evaluating different homomorphic encryption algorithms such as BFV, BGV, and CKKS using the proposed efficient long polynomial multiplier based on hardware-software co-design.

Chapter 8

Conclusion and Future Works

8.1 Conclusion

This dissertation has considered the efficient and high-performance designs for modular (integer) multipliers, modular polynomial multipliers, and custom design blocks for lattice-based cryptography. Specifically, an ultra-high parallel number-theoretic transform (NTT)-based polynomial multiplication architecture with multiple processing elements (PEs) is presented in Chapter 3 for the fully homomorphic encryption (FHE) acceleration of this dissertation. Then, a pipelined NTT architecture for both homomorphic encryption (HE) and post-quantum cryptography (PQC) schemes are introduced in Chapter 4. We also present a novel optimized modular Karatsuba multiplication algorithm in Chapter 5 for the modular integer multiplications to achieve better timing and area performance. Besides, for the lattice-based schemes that cannot leverage the NTT algorithm for modular polynomial multiplication, we also provide a novel and high-speed schoolbook modular polynomial multiplier design based on the fast filtering technique. Finally, another efficient NTT-based polynomial multiplication architecture for HE schemes whose moduli are large is presented in Chapter 7. This architecture mainly targets the optimization of the pre-processing as well as post-processing steps of the Chinese remainder theorem (CRT) and a novel pipelined NTT-based polynomial multiplier.

These proposed designs and architectures are implemented in either field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC) platforms for the performance evaluation, showing our work performs better than the prior designs.

Furthermore, several future directions and extended works based on our proposed designs in this dissertation can be investigated.

8.2 High-Speed Architecture for the CRYSTALS-Kyber Post-Quantum Cryptography Scheme

CRYSTALS-Kyber (Kyber) scheme has been identified for key-establishment (KEM) algorithm standardization in the National Institute of Standards and Technology (NIST) post-quantum cryptography (PQC) standardization process. Due to the special construction of the Kyber algorithm, the NTT-based polynomial multiplier is slightly different from the conventional design for the negative wrapper convolution accelerator presented in Chapters 3 and 4. In particular, the degree-256 modular polynomial multiplications in the Kyber scheme need to be divided into two degree-128 polynomials first and then perform the degree-128 modular polynomial multiplications. Therefore, we can utilize the novel designs in Chapters 3 and 4 along with the efficient two-parallel algorithms to customize the efficient modular polynomial multiplier for the Kyber scheme.

8.3 Efficient VLSI Architecture for Homomorphic Evaluation for the Homomorphic Encryption Scheme

The goal of this work is to develop an efficient BFV-based homomorphic evaluation accelerator, including the homomorphic multiplier and homomorphic adder, to fulfill a wide range of HE applications. The high-level overview is shown in Fig. 8.1. However, due to the complex construction of the HE schemes, the computation is expensive and thus requires a long processing time. Since our targeted BFV scheme is built upon the ring-learning with errors (RLWE) problem, the fundamental arithmetic of the HE schemes is the polynomial multiplication/addition over the ring. Based on this reason, most of the prior hardware accelerations only focus on the polynomial multiplication over the ring using the NTT algorithm.

Built upon our prior works in this dissertation, future work will focus on the entire homomorphic evaluation step. The hardware implementation for the homomorphic adder is relatively simple since it only involves modular addition, while the homomorphic multiplier is challenging.

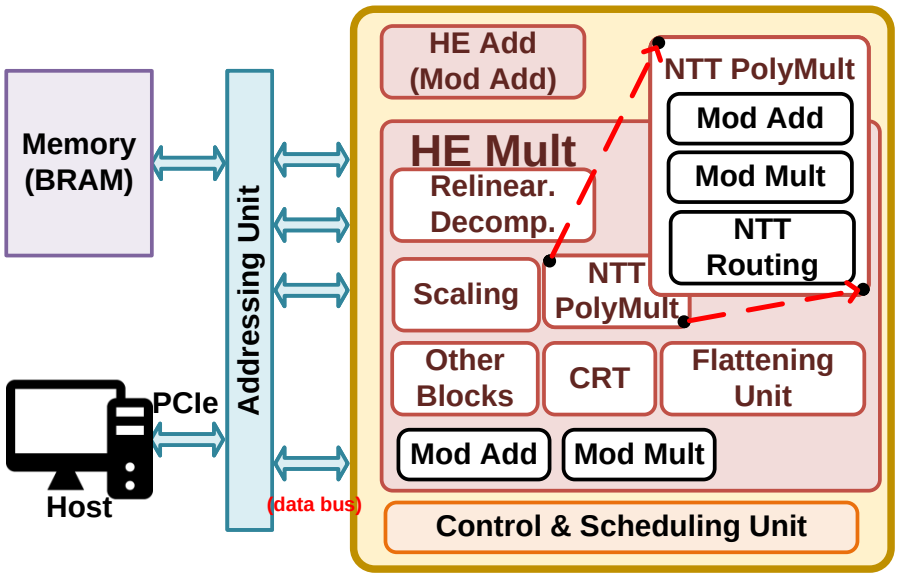

Figure 8.1: Top-level architecture for homomorphic evaluation processor.

Besides, large parameters are used in the HE scheme in order to have a high-security level and be able to perform depth of the homomorphic arithmetic. The future work further investigates the implementations of the detailed step inside the BFV-based homomorphic multiplier, mainly tensoring and relinearization steps. Besides, we will follow the software implementation in [113] and focus on a hardware-friendly optimization for the efficient residue number system-based BFV accelerator.

Bibliography

[1] DataReportal, "Digital 2022: The united states of america," https://datareportal.com/reports/digital-2022-united-states-of-america, 2022, accessed November, 20, 2022.

[2] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2018.

[3] A. E. Cohen, *Architectures for cryptography accelerators*. University of Minnesota, 2007.

[4] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[5] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[6] S. Gao, "Efficient fully homomorphic encryption scheme," *IACR Cryptology ePrint Archive*, p. 637, 2018.

[7] B. M. Case, S. Gao, G. Hu, and Q. Xu, "Fully homomorphic encryption with k-bit arithmetic operations," *Cryptology ePrint Archive, Report 2019/521*, 2019, https://eprint.iacr.org/2019/521.

[8] W. Tan, G. Hu, B. Case, S. Gao, and Y. Lao, "An efficient polynomial multiplier architecture for the bootstrapping algorithm in a fully homomorphic encryption scheme," in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2019, pp. 85–90.

[9] W. Tan, B. M. Case, G. Hu, S. Gao, and Y. Lao, "An ultra-highly parallel polynomial multiplier for the bootstrapping algorithm in a fully homomorphic encryption scheme," *Journal of Signal Processing Systems*, vol. 93, no. 6, pp. 643–656, 2021.

[10] W. Tan, A. Wang, Y. Lao, X. Zhang, and K. K. Parhi, "Pipelined high-throughput NTT architecture for lattice-based cryptography," in *2021 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2021, pp. 1–4.

[11] W. Tan, B. M. Case, A. Wang, S. Gao, and Y. Lao, "High-speed modular multiplier for lattice-based cryptosystems," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 8, pp. 2927–2931, 2021.

[12] W. Tan, A. Wang, Y. Lao, X. Zhang, and K. K. Parhi, "Low-latency vlsi architectures for modular polynomial multiplication via fast filtering and applications to lattice-based cryptography," *arXiv preprint arXiv:2110.12127*, 2021.

[13] C. J. Smyth, "A coloring proof of a generalisation of fermat's little theorem," *The American Mathematical Monthly*, vol. 93, no. 6, pp. 469–471, 1986.

[14] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[15] C. Peikert, "A decade of lattice cryptography," *Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 4, pp. 283–424, 2016.

[16] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[17] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 75–92.

[18] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun, "Batch fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013, pp. 315–335.

[19] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

[20] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 353–367.

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, p. 13, 2014.

[22] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[23] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[24] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *IMA International Conference on Cryptography and Coding*. Springer, 2013, pp. 45–64.

[25] S. S. Roy, K. Jarvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Transactions on Computers*, 2018.

[26] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 143–163.

[27] S. S. Roy, F. Vercauteren, J. Vliegen, and I. Verbauwhede, "Hardware assisted fully homomorphic function evaluation and encrypted search," *IEEE Transactions on Computers*, vol. 66, no. 9, pp. 1562–1572, 2017.

[28] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[29] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[30] J. M. Pollard, "The fast Fourier transform in a finite field," *Mathematics of computation*, vol. 25, no. 114, pp. 365–374, 1971.

[31] J. Naranjo, J. López-Ramos, and L. Casado, "Applications of the extended euclidean algorithm to privacy and secure communications," in *Proc. of 10th International Conference on Computational and Mathematical Methods in Science and Engineering*, 2010, pp. 702–713.

[32] X. Feng and S. Li, "Design of an area-effcient million-bit integer multiplier using double modulus NTT," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 9, pp. 2658–2662, 2017.

[33] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, "SWIFFT: A modest proposal for FFT hashing," in *International Workshop on Fast Software Encryption*. Springer, 2008, pp. 54–72.

[34] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-SEAL v2. 1," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 3–18.

[35] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[36] S. Halevi and V. Shoup, "Bootstrapping for HElib," Cryptology ePrint Archive, Report 2014/873, 2014, https://eprint.iacr.org/2014/873.

[37] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 387–398.

[38] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems." *IEEE Trans. on Circuits and Systems*, vol. 62, no. 1, pp. 157–166, 2015.

[39] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[40] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, 2015.

[41] X. Feng and S. Li, "Accelerating an FHE integer multiplier using negative wrapped convolution and ping-pong FFT," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 1, pp. 121–125, 2018.

[42] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, 2017.

[43] J.-H. Ye and M.-D. Shieh, "Low-complexity VLSI design of large integer multipliers for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[44] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *Digital System Design (DSD), 2013 Euromicro Conference on*. IEEE, 2013, pp. 955–962.

[45] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction." *IACR Cryptology ePrint Archive*, vol. 2013, p. 616, 2013.

[46] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014, pp. 371–391.

[47] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 139–158.

[48] T. Oder and T. Güneysu, "Implementing the NewHope-Simple key exchange on low-cost FPGAs," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2017, pp. 128–142.

[49] Y. Xing and S. Li, "An efficient implementation of the NewHope-Simple key exchange on FPGAs," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 3, pp. 866–878, 2019.

[50] P.-C. Kuo, W.-D. Li, Y.-W. Chen, Y.-C. Hsu, B.-Y. Peng, C.-M. Cheng, and B.-Y. Yang, "High performance post-quantum key exchange on FPGAs," *IACR Cryptology ePrint Archive*, p. 690, 2017.

[51] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, 2020.

[52] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Optimised multiplication architectures for accelerating fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2794–2806, 2015.

[53] C. P. Rentería-Mejía and J. Velasco-Medina, "High-throughput ring-LWE cryptoprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2332–2345, 2017.

[54] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt, "Exploring energy efficient quantum-resistant signal processing using array processors," in *ICASSP 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 1539–1543.

[55] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, fall joint computer conference*. ACM, 1966, pp. 563–578.

[56] P. Barrett, "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*. Springer, 1986, pp. 311–323.

[57] K. K. Parhi, *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[58] K. K. Parhi, C.-Y. Wang, and A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 1, pp. 29–43, 1992.

[59] M. Ayinala, M. Brown, and K. K. Parhi, "Pipelined parallel FFT architectures via folding transformation," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 20, no. 6, pp. 1068–1081, 2011.

[60] National Institute of Standards and Technology (NIST), "Post-quantum cryptography standardization," https://csrc.nist.gov/projects/post-quantum-cryptography, 2020.

[61] A. Karatsuba, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, pp. 595–596.

[62] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 17–61, 2019.

[63] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[64] X. Feng, S. Li, and S. Xu, "RLWE-oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 3, pp. 556–559, 2019.

[65] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede, "Efficient Ring-LWE encryption on 8-bit AVR processors," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 663–682.

[66] W. Wang, S. Tian, B. Jungk, N. Bindel, P. Longa, and J. Szefer, "Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qtesla," in *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES, September 2020.

[67] M. Hamburg, "Ed448-goldilocks, a new elliptic curve." *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 625, 2015.

[68] K. Sakiyama, M. Knežević, J. Fan, B. Preneel, and I. Verbauwhede, "Tripartite modular multiplication," *INTEGRATION, the VLSI journal*, vol. 44, no. 4, pp. 259–269, 2011.

[69] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[70] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, pp. 2459–2463, 2019.

[71] Y. Zhang, C. Wang, D. E. S. Kundi, A. Khalid, M. O'Neill, and W. Liu, "An efficient and parallel R-LWE cryptoprocessor," *IEEE Transactions on Circuits and Systems II*, vol. 67, no. 5, pp. 886–890, 2020.

[72] S. S. Roy and A. Basso, "High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 443–466, 2020.

[73] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49–72, 2020.

[74] X. Zhang and K. K. Parhi, "Reduced-complexity modular polynomial multiplication for R-LWE cryptosystems," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 7853–7857.

[75] D. A. Parker and K. K. Parhi, "Low-area/power parallel FIR digital filter implementations," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 17, no. 1, pp. 75–92, 1997.

[76] C. Cheng and K. K. Parhi, "Hardware efficient fast parallel FIR filter structures based on iterated short convolution," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 8, pp. 1492–1500, 2004.

[77] D. A. Parker and K. K. Parhi, "Area-efficient parallel FIR digital filter implementations," in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP'96*. IEEE, 1996, pp. 93–111.

[78] H.-T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 01, pp. 37–46, 1982.

[79] S. Y. Kung, "VLSI array processors," *Englewood Cliffs*, 1988.

[80] H. V. Jagadish, S. K. Rao, and T. Kailath, "Array architectures for iterative algorithms," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1304–1321, 1987.

[81] C. Chen, O. Danba, J. Hoffstein, A. Hülsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, and Z. Zhang, "Algorithm specifications and supporting documentation," *Brown University and Onboard security company, Wilmington USA*, 2019.

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

[83] N. I. of Standards and Technology, "FIPS PUB 202 SHA-3 standard: Permutation-based hash and extendable-output functions." 2015, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[84] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Annual international cryptology conference*. Springer, 1999, pp. 537–554.

[85] J. M. B. Mera, F. Turan, A. Karmakar, S. S. Roy, and I. Verbauwhede, "Compact domain-specific co-processor for accelerating module lattice-based KEM," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[86] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs, "Learning with rounding, revisited," in *Annual Cryptology Conference*. Springer, 2013, pp. 57–74.

[87] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé, "Classical hardness of learning with errors," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 575–584.

[88] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[89] Y. Xing and S. Li, "A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 328–356, 2021.

[90] D. T. Nguyen, V. B. Dang, and K. Gaj, "A high-level synthesis approach to the software/hardware codesign of NTT-based post-quantum cryptography algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 371–374.

[91] ——, "High-level synthesis in implementing and benchmarking number theoretic transform in lattice-based post-quantum cryptography using software/hardware codesign." in *ARC*, 2020, pp. 247–257.

[92] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "LWRpro: An energy-efficient configurable crypto-processor for Module-LWR," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1146–1159, 2021.

[93] V. B. Dang, F. Farahmand, M. Andrzejczak, and K. Gaj, "Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 206–214.

[94] S. Fan, W. Liu, J. Howe, A. Khalid, and M. O'Neill, "Lightweight hardware implementation of R-LWE lattice-based cryptography," in *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2018, pp. 403–406.

[95] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE international symposium on circuits and systems (ISCAS)*. IEEE, 2014, pp. 2796–2799.

[96] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/-software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 335–347, 2016.

[97] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," in *Soviet Mathematics Doklady*, vol. 3, 1963, pp. 714–716.

[98] A. Basso and S. S. Roy, "Optimized polynomial multiplier architectures for post-quantum kem saber," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1285–1290.

[99] W. Wang, S. Tian, B. Jungk, N. Bindel, P. Longa, and J. Szefer, "Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qtesla," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 269–306, 2020.

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

[101] Xilinx, "7 series FPGAs configurable logic block user guide (UG474)," Sep 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[102] F. Wibawa, F. O. Catak, S. Sarp, M. Kuzlu, and U. Cali, "Homomorphic encryption and federated learning based privacy-preserving CNN training: COVID-19 detection use-case," in *Proceedings of the 2022 European Interdisciplinary Cybersecurity Conference*, 2022, pp. 85–90.

[103] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 395–412.

[104] S. Halevi and V. Shoup, "Algorithms in helib," in *Annual Cryptology Conference*. Springer, 2014, pp. 554–571.

[105] G. Xin, Y. Zhao, and J. Han, "A multi-layer parallel hardware architecture for homomorphic computation in machine learning," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[106] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.

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

[108] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *International Conference on Cryptology and Network Security*. Springer, 2016, pp. 124–139.

[109] A. C. Mert, S. Kwon, Y. Shin, D. Yoo, Y. Lee, and S. S. Roy, "Medha: Microcoded hardware accelerator for computing on encrypted data," *Cryptology ePrint Archive*, 2022.

[110] A. C. Mert, E. Öztürk, and E. Savaş, "FPGA implementation of a run-time configurable ntt-based polynomial multiplication hardware," *Microprocessors and Microsystems*, vol. 78, p. 103219, 2020.

[111] R. Paludo and L. Sousa, "NTT architecture for a Linux-ready RISC-V fully-homomorphic encryption accelerator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[112] N. K. Unnikrishnan and K. K. Parhi, "Multi-channel FFT architectures designed via folding and interleaving," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 142–146.

[113] S. Halevi, Y. Polyakov, and V. Shoup, "An improved rns variant of the BFV homomorphic encryption scheme," in *Cryptographers' Track at the RSA Conference*. Springer, 2019, pp. 83–105.