RNS-Based NTT Polynomial Multiplier

for Lattice-Based Cryptography

by

Logan Brist

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2020 by the
Graduate Supervisory Committee:

Chaitali Chakrabarti, Chair
Daniel Bliss
Antonia Papandreou-Suppappola

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

Lattice-based Cryptography is an up and coming field of cryptography that utilizes the difficulty of lattice problems to design lattice-based cryptosystems that are resistant to quantum attacks and applicable to Fully Homomorphic Encryption schemes (FHE). In this thesis, the parallelization of the Residue Number System (RNS) and algorithmic efficiency of the Number Theoretic Transform (NTT) are combined to tackle the most significant bottleneck of polynomial ring multiplication with the hardware design of an optimized RNS-based NTT polynomial multiplier. The design utilizes Negative Wrapped Convolution, the NTT, RNS Montgomery reduction with Bajard and Shenoy extensions, and optimized modular 32-bit channel arithmetic for nine RNS channels to accomplish an RNS polynomial multiplication. In addition to a full software implementation of the whole system, a pipelined and optimized RNS-based NTT unit with 4 RNS butterflies is implemented on the Xilinx Artix-7 FPGA(xc7a200tlffg1156-2L) for size and delay estimates. The hardware implementation achieves an operating frequency of 47.043 MHz and utilizes 13239 LUT's, 4010 FF's, and 330 DSP blocks, allowing for multiple simultaneously operating NTT units depending on FGPA size constraints.

ACKNOWLEDGMENTS

I want to give great thanks to my mentor Professor Chaitali Chakrabarti for being my committee head, for putting the effort and resources into my studies, for guiding the subject matter and scope of the thesis, and for using her hardware expertise to consistently provide valuable and sensible feedback. I greatly appreciate being able to work with her, not only for the thesis but in her independent study course and in her lab as a research assistant for a year and a half. My graduate experience has been invaluable and I owe my thanks to her for that.

I want to thank Professor Daniel Bliss for being on my committee and for the great feedback he has provided. As an undergraduate, he gave me coursework considerations and helped me join the graduate electrical engineering program at ASU. I also appreciate being able to have a part in his Airbus Flex Connect project working under Prof. Chakrabarti. His communication systems class and textbook are highly enjoyable and guided much of my interest in taking communication coursework.

I want to thank Professor Antonia Papandreou-Suppappola for being on my committee and for the academic and industry opportunities she provided. Her Time-Frequency Processing course helped me get a taste of report writing and helped me consider the thesis option for the Masters degree. She was also the one who initially helped me get into contact with Prof. Chakrabarti for a research assistant position, as well as other industry connections. Her help has been exceptional.

I lastly want to thank my family and friends because they have consistently been there for me and have always helped guide decision making in the right direction. Finding success in school as well as in other important areas of life is notably easier when someone has a strong foundation. So to friends and family, and viewers like you, thank you.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

DEFINITIONS

Variable	Name
M	System modulus
n	Input polynomial length
w	Modulus bitwidth
w_{ch}	RNS channel bitwidth
$\mathbb{Z}_M^n$	Length n vector of integers mod M
ω_n	n-th root of unity
k	Number of RNS channels
D	Dynamic range of RNS base
$\hat{X}$	Variables in base 2
X_{RNS}	RNS representation of X
x_i	RNS value of X for channel i
D_i	D divided by moduli i
$\lvert X \rvert_i$	X mod channel i
X_i^{-1}	Modular inverse of X_i with respect to channel i
W_i	RNS conversion weight for channel i
d	Propagation delay
f	Operating frequency
ℓ	FIFO Length
B	Buffer length

Table 1. Table of Definitions.

Chapter 1

INTRODUCTION

Cryptography is the field of study concerning how to send secure messages through an insecure channel. In today's world, the need to securely send information for the sake of business, banking, personal communication, or any internet communication is unparalleled and requires the application of cryptography schemes that have extremely fast, resource-efficient, and secure encryption and decryption operations. With the recent discoveries of quantum-computing attacks that present security risks to classical cryptography, established cryptographic schemes will soon need to be replaced with new schemes that are future-proofed in terms of both attack resilience and performance needs. One paramount area of cryptographic research that is anticipated by researchers to have both the benefits of future-proof security and speed is Lattice-based Cryptography.

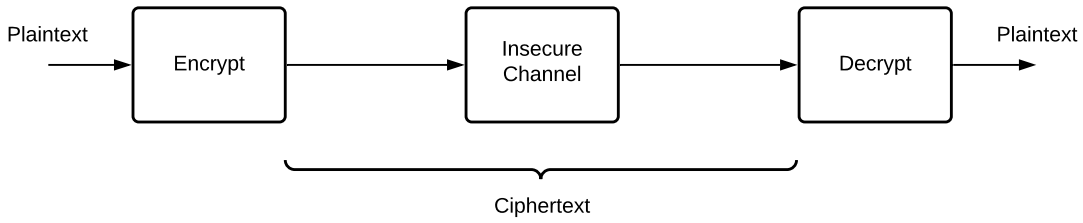

Figure 1. Cryptography Model showing encryption and decryption of data on both ends of an insecure channel.

Lattice-based Cryptography is a branch of cryptography which utilizes the mathematics of lattices to provide security for a cryptographic scheme. The core benefit of

Lattice-based Cryptography is that lattice problem hardness enables cryptographic schemes to be resistant to quantum attacks. Additionally, lattice-based cryptosystem algorithms are relatively simple and able to be run in parallel due to their dependency on operations on rings of integers for certain cryptosystems. Often paired with lattice-based cryptosystems is Fully Homomorphic Encryption, which is an encryption that enables arbitrary calculations on encrypted data while maintaining correct intermediate results without decrypting the data to plaintext. Both arenas are currently in the research spotlight. If these areas of cryptography are to be realized and implemented in greater scale in the future, their time-critical elements need to be understood and accelerated. Modular polynomial multiplication is one of the most critical modules in Lattice-based Cryptography and Fully Homomorphic encryption so this thesis focuses on an efficient design of a RNS-based NTT polynomial multiplier.

1.1 Problem Statement and Space for Hardware Optimizations

As determined by a host of Lattice-based Cryptography hardware acceleration papers and general survey papers ([1], [2], [3],[4], [5], [6]), there are two primary bottlenecks of lattice-based cryptographic schemes: Modular polynomial multiplication and Gaussian sampling. Lattice-based Cryptography algorithms rely on massive numbers of polynomial multiplications to encode and decode polynomial plaintext/ciphertext using key values. These keys then rely on a large number of Gaussian samples because they are required to be random polynomials. Together they make up the two areas of hardware optimization currently seen in the field. A deeper review of the two primary areas of optimization has been provided in Chapter 2.

There is a diverse range of devices that will potentially be using Lattice-based

Cryptography and they will require high performance or resource constrained computation that can't be obtained with a general-purpose processor. The problem of polynomial multiplication is chosen because it is one of the most time critical elements that can benefit from hardware acceleration. The goal of this thesis is to design a hardware-optimized Residue Number System-based (RNS) Number Theoretic Transform (NTT) polynomial multiplier for application in Lattice-based Cryptography. An NTT-based multiplication method is chosen for its flexibility, lowest computational complexity, simplified extensions into hardware, and established foothold in existing literature. An RNS dataflow is then adopted because of the speed increase found with a parallel channel system and its natural extension into parallel hardware.

1.2 Related Work

The first notable hardware papers for Lattice based cryptography comes from Göttert [1] and Pöppelmann [7] [8], which start pinpointing the necessary building blocks for lattice hardware and begin to make a flexible accelerator given the partially defined parameter set of lattice cryptography at the time. Howe introduces an area-optimized FPGA design for the Ring Learning with Errors scheme in [9]. The Sapphire co-processor [10] was also introduced which is a configurable crypto-processor focused on lattice-based cryptography. Looking at papers that assisted in hardware design, work in Lattice-based Cryptography also yielded a Haskell library for lattice constructions [11], released by Peikert and Crockett with the intention of improving testing and prototyping. A review of the history and direction of lattice cryptography can also be seen in [12] by Peikert.

Considering FHE hardware accelerators, there are several existing FPGA accel-

erators for Homomorphic encryption. A R-LWE and SHE polynomial multiplier was explored in [13]. Reconfigurable hardware for FHE was explored in [14] and by Pöppelmann [15]. An FHE co-processor was designed in [16]. [17] created domain specific accelerators for FHE. Most recently, [18] made an FPGA based hardware accelerator for R-LWE FHE. The introduction of HEAWS [19], an FHE accelerator which utilizes RNS and NTT, was also seen recently. In [16], a co-processor that accelerates a Chinese Remainder Transform, ring addition, ring subtraction, and ring multiplication is designed and obtains several orders of magnitude acceleration over CPU runtime. An SHE residue polynomial multiplier was designed in [20]. It uses an accelerated RNS variant of negative wrapped convolution and is designed for flexibility. It also features a twiddle factor generator.

Specific to the problem under study, the following papers on RNS and NTT polynomial multipliers are recently published and relevant to work being built on. RNS is used in [20] to achieve SHE acceleration. A Negative Wrapped Convolution polynomial multiplier is used in [21] to multiply integers. Additionally, an NTT-based polynomial multiplier is designed in [22]. In [23], a systolic architecture is chosen to make an energy efficient polynomial multiplier. They systolically perform a standard convolution based multiplication and an NTT multiplication and compare it to a sequential NTT multiplier. Both systolic designs achieve 1.7x to 7.5x higher throughput over a sequential NTT multiplier. This work builds on many of these papers by combining the RNS elements and other lattice-based cryptography acceleration methods. RNS is also used in [24] on RSA and Elliptic curve cryptography.

1.3 Thesis Contributions

This thesis focuses on designing an RNS-based NTT polynomial multiplier for application in Lattice-based cryptography and FHE. At the algorithmic level, polynomial multiplication is accomplished using an NTT because of its low computational complexity and its ability to perform a polynomial multiplication using the negative wrapped convolution algorithm. This allows input polynomials to be unchanged in size compared to the zero padded inputs in the NTT. The NTT approach also enables the use of parallel NTT butterfly units when implemented on an FPGA. At the datapath level, parallelism via RNS is chosen because it can break large bit-width operations into smaller channel operations and consequently can lower propagation delays in the NTT butterfly. To accomplish the required modular multiplication in RNS, an optimized RNS Montgomery multiplication with efficient Shenoy [25] and Bajard [26] base extensions is implemented. RNS Montgomery is chosen over a selection of other RNS modular multiplication methods for its overall lower size when compared to Sum of Residues and higher speed when compared to Barrett reduction. Lastly at the computational level, modular channel arithmetic is simplified and streamlined using hardware efficient modular addition, modular subtraction, and Barrett reduction. Arithmetic units such as the base extensions and accumulators are also streamlined by being pipelined and having internal MAC units unrolled into parallel multiplications. Registers are then properly placed to lower combinational delay per cycle and to achieve an average throughput of one butterfly operation per cycle. The entire design is then implemented in software with hardware units simulated in hardware.

With these design elements briefly stated and summarized, the specific contributions of this thesis are providing in the following list:

- Algorithm level/data-path level/computation level design for an RNS-based NTT polynomial multiplier that targets a polynomial length of $n = 1024$, $w = 32$-bit coefficients, and $k = 4$ RNS channels of width $w_{ch} = 8$-bit for lattice schemes and a polynomial length of $n = 4096$, $w = 128$-bit coefficients, $k = 4$ RNS channels of width $w_{ch} = 32$-bit for FHE schemes. The design uses 4 sequential/simultaneously operating NTT units with an RNS dataflow that takes advantage of hardware parallelism.
- C++ code for the fully parameterizable RNS-based NTT polynomial multiplier. Includes the RNS-based butterfly NTT with classical NTT and radix-2 butterfly NTT for comparison, negative wrapped convolution, parameter/coefficient generation using modular functions, RNS Montgomery Reduction, Bajard and Shenoy base extensions, RNS forward and reverse conversion, Barrett channel reductions, and supporting test functions to verify all units.
- Verilog FPGA results for size and timing of the RNS-based NTT unit, RNS Montgomery multiplication unit, RNS Bajard and Shenoy base extension units, and optimized modular channel arithmetic units on a Xilinx Artix-7 FPGA (xc7a200tlffg1156-2L).
- Literature survey of lattice bottlenecks including polynomial multiplication (School Book, Comba, 3-way Toom-Cook, k-way Toom-Cook, Furer, Karatsuba, Schonhage-strassen, and FFT algorithms) and Gaussian sampling.
- Background and explanation for the NTT, RNS, RNS Montgomery multiplication, RNS Bajard and Shenoy base extensions, modular arithmetic optimizations, and parameter selection.

1.4 Section Summary

The general chapter contents for this thesis are as follows:

- Chapter 1 introduces the hardware optimization problem, related work, thesis contributions, and a self-referential section summary.
- Chapter 2 contains background information relevant to cryptography and the direction of hardware/software design.
- Chapter 3 contains documentation and algorithms for each component of the RNS-based NTT polynomial multiplier.
- Chapter 4 contains details of hardware implementation including size/timing results for the major modules implemented on an Artix-7 FPGA.
- Chapter 5 contains final commentary on the design and future design considerations.

Chapter 2

BACKGROUND

2.1 Quantum-Proof Cryptography

Cryptography can be broken down into public-key (asymmetric) and symmetric-key cryptography. A key is generally a string of numbers that can be arithmetically applied to plaintext information to make it appear random. When a party knows the key(s) used in a scheme, they are able to translate between plaintext and ciphertext using protocol encryption and decryption operations. Symmetric-key schemes are the oldest computation-era cryptographic schemes, where only one common private key is known to the sender and recipient of a message. The security level of a symmetric-key cryptographic scheme is generally equal to the number of operations needed to randomly guess the key.

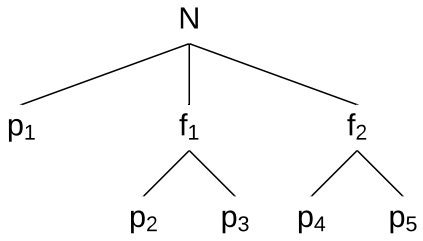

Figure 2. The Integer Factorization problem which is solved in polynomial time by Shor's algorithm [27].

Symmetric-key schemes suffer when many communication channels are open, as each pair of communicators requires a unique and safely conveyed private key. Public-

key schemes, the primary focus for lattice-based cryptography, make use of both a private key shared only between sender and recipient and a public key, which is generated from the private key in an unknown way but shared with all on the channel. The public key is then used to encrypt messages and the private key is used for decryption. Low key sizes make lattice-based cryptosystems a strong candidate for full scale implementation.

Traditional cryptographic schemes make use of difficult mathematical problems to obtain their security in transmitting digital information. These problems have mainly pertained to factorization and historically have not been considered breakable in polynomial time on standard computational devices (polynomial time being the generally acceptable measure of maximum complexity before a computation is deemed infeasible). The difficulty in solving public-key cryptographic problems can be measured in terms of average-case hardness and worst-case hardness. Average-case hardness can be thought of as the resistance to random guesses from a probability distribution and it means the cryptographic security problem is difficult to solve in most cases. Worst-case hardness is when the cryptographic security problem is difficult to solve for only some cases. Some traditional cryptography problems include the following.

- Integer Factorization - Factoring large numbers down to their prime factors.
- Discrete Logarithm - Computing the discrete logarithm of certain large numbers.
- Elliptic-Curve Discrete Logarithm - Computing the discrete logarithm of a random elliptic curve element with respect to a known base point.

While these problems have been effective for past cryptography applications, Shor's algorithm in 1995 [27] finds a number's prime factors in polynomial time on a quantum computer and will break each of these traditional problems. From here, the search for

a quantum proof problem has become an increasingly important area of cryptography research.

2.2 Lattice Problems

Lattice problems have been found to be one solution to the quantum computing-proof problem. A lattice is described as an infinite repetition of points in an n-dimensional space. To represent a lattice, a matrix of basis vectors which make up each contained point is used.

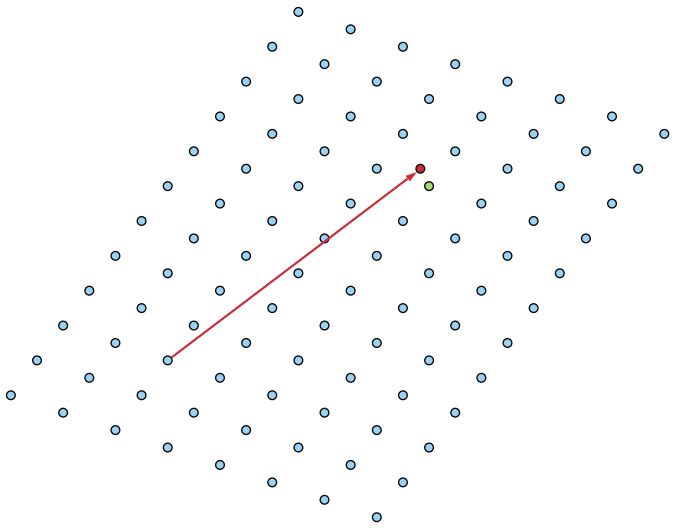

Figure 3. Closest Vector Problem. Given a random vector (red) find the closest lattice point (blue) to the random vector.

A linear combination of these vectors is then determinable for each point in the lattice. This matrix can be written as

$$L(B) = Bx : x \in \mathbb{Z}^n = L(b_1, ..., b_n) = \sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z} \tag{2.1}$$

Given this mathematically constructed environment, several problems involving lattices can be generated. These include:

- Shortest Vector - Find the shortest nonzero vector in a lattice given the basis of the lattice.
- Closest Vector - Given a basis of a lattice and a vector v not in the lattice, find the closest vector to v that is in the lattice.
- Covering Radius - Given a basis for a lattice, find the smallest sphere that always includes exactly 2 lattice points when placed at every lattice point.

In 1996, Ajtai [28] found that proving worst-case security for the shortest vector problem also yielded average-case security. The implications of this meant that worst-case lattice problems can only be reduced to average-case cryptography problems when tackled with quantum computing and it has thusly given rise to mass interest in lattice-based cryptography, specifically for the next generation of quantum-resistant cryptographic schemes.

2.3 Lattice Cryptography Schemes

Following the discover of Ajtai in 1996 [28], the use of lattices appeared in cryptography in multiple schemes including NTRU, LWE, and R-LWE. The following sections contain brief descriptions of them.

It should be noted that the use of rings from number theory are often critical to most cryptography schemes. A ring is an abstract algebraic set that is closed under addition and multiplication. Some simple examples of rings include the integers $\mathbb{Z}$, complex numbers $\mathbb{C}$, and integers modulo some number q which is written as $\mathbb{Z}_q$. While many sets of numbers can be considered a ring, the general functionality of wrapping around

a modulus as with $\mathbb{Z}_q$ is of most concern in the context of cryptography. Lattice-based cryptography schemes make use of rings in its polynomials to speed up calculations. A polynomial ring is written as $\mathbb{Z}_q[x]/\Phi$ where Φ is generally $\Phi = x^n + 1$. This has the effective impact of limiting the size of polynomial used in the cryptographic scheme to that of size Φ. The degree of these polynomials are generally a large prime number or power of two with coefficients modulo a small integer. By keeping computations in rings, computation is made more efficient and precise as operations take place in a finite space and because reduction in the integer space results in reduced key sizes.

2.3.1 NTRU

NTRU [29], developed in 1996, is a public-key encryption scheme. It was the first scheme to use polynomial rings for efficiency and compactness and is the fastest known lattice-based encryption scheme. It is also listed in the IEEE P1363 standard. The original NTRUEncrypt scheme a uses secret key that consists of two sparse polynomials with coefficients of $-1, 0, 1$ and limited to a degree $< n$. The public key is the quotient of the secret key in $\mathbb{Z}_q[x]/(x^n - 1)$ where the degree of polynomial n is prime and q is a power of two integer that is the same order of magnitude as n. The public key is generated as

$$h = pg/f \mod q \tag{2.2}$$

where $p \in \mathbb{Z}[x]/(x^2 - 1)$ and g and f are the two secret key polynomials. If the polynomial f is found to not be invertible modulo q and modulo p, the key needs regenerated. The ciphertext of a plaintext message M is encrypted as

$$C = hs + M \mod q \tag{2.3}$$

Decryption is then done by multiplying the ciphertext by the secret key and eliminating certain terms due to many of them being small. For the original NTRU cryptographic scheme, there is no discovered relation between the cryptography problem in NTRU and a worst-case lattice problem. One issue with the given formulation of the secret and private keys in NTRU is that there is a non-uniform distribution among the ring used for the private key. As sampling uniformity provides optimal resistance to random guesses of the key, this leads to difficulty in achieving hardness. This problem was answered in [30] where a relationship between ring-LWE hardness and a version of NRTU proved the new version secure. It was found that if the secret key polynomials are rejection sampled from a discrete Gaussian distribution (rejecting sample if it is not invertible), the public key will be statistically indistinguishable from a uniform distribution. The updated NTRU uses a ring of $\mathbb{Z}_q[x]/(x^n + 1)$ instead of $\mathbb{Z}_q[x]/(x^n - 1)$. In 2019, NTRU is optimized using the NTT in [31].

2.3.2 Learning With Errors

Learning with errors (LWE) is a cryptographic problem that was introduced in 2005 by Regev [32] and has average-case hardness. LWE has a search version and a decision version and the problem formulation is as follows:

1. Given a dimension $n > 1$, an integer $p \geq 2$ used as the modulus, a discrete Gaussian error distribution X, and $\mathbb{Z}_q$ the integers modulo some integer q:
2. Generate some vector $s \in \mathbb{Z}_q^n$, a uniformly random vector $a \in \mathbb{Z}_q^n$, and an error value e from X.
3. Generate a sample point (a, t) where $t = \langle a, s \rangle + e \mod p$.

4. Search LWE problem: Given a certain number of computed pairs of points (a_i, t_i), find s.
5. Decision LWE problem: Given a certain number of computed pairs of points (a_i, t_i) and random points from a uniform distribution, determine which points originate from s.

In 2012, a LWE-based system was implemented in software and hardware [1] highlighting the core hardware-related problems of polynomial multiplication and sampling from discrete Gaussian distributions. It is possible for there to be both a polynomial and matrix based variant of LWE. In the paper, it was found that the polynomial version of a LWE-based cryptosystem was more successful in all metrics (speed, memory requirement, etc.) with the exception of the matrix version having smaller message expansion factors (ratio of the encoded data length to decoded data length). LWE has been used to prove the security of many cryptosystems however they are limited on efficiency due to their keys being matrices randomly generated over $\mathbb{Z}_q$ for a small integer q, with dimension that linearly increases with security. The solution to this comes in the form of ring variants.

2.3.3 Ring Learning With Errors

In 2009, a varinant of LWE called Ring Learning with Errors (Ring-LWE) was proposed by Lyubashevsky, Peikert, and Regev ([33] and [34]). Ring-LWE allows for more natural and efficient design of cryptography functions by utilizing polynomials in rings. In Ring-LWE, the key size is reduced by a factor of n compared to its LWE parent. LWE may use up to several thousand bits to secure a message that would take Ring-LWE only hundreds of bits, making Ring-LWE a more feasible system for any

implementation that has constrained computation resources. Ring-LWE additionally has been the basis for a published paper that supports fully Homomorphic encryption [35].

2.3.4 Other Schemes

Short integer solution (SIS) is a cryptographic problem that was introduced by Ajtai in 1996 [28] in his paper proposing an average-case lattice problem. The problem was the first to be proven as difficult on average as lattice problems in the worst case leading to future investigation in lattice-based cryptography. Additional schemes such as Goldreich–Goldwasser–Halevi signature/encryption (GGH) and Bimodal Lattice Signature Scheme (BLISS) rely on lattices. BLISS is a digital signature scheme introduced in 2013 in [5]. One of the innovations in BLISS was the presentation of a new rejection sampling algorithm. The authors improve the computation of both the exponential and hyperbolic cosine for the sake of implementation efficiency and they make use of a variant of rejection sampling. Goldreich–Goldwasser–Halevi signature and encryption scheme (GGH) is a lattice signiture scheme presented at Crypto conference '95 and '97 and explained and suggested insecure in [36]. While not highly relevant, it is based the closest vector problem and was the original inspiration for NTRU signatures. Gentry and Peikert also show further potential theoretical ability to create efficient cryptographic constructions that utilized lattices in 2008 [37].

2.4 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is a form of encryption that allows computation on encoded data without revealing the true contents to the computer handling the data. FHE is not a lattice-based scheme, but it is a concept in cryptography that is finding priority in research along with quantum-proofing cryptographic schemes for its massive implications in future security. The main benefit of FHE is that an encoded chain of data flow will remain encrypted for mathematical operations, leaving fewer instances and consequent vulnerabilities on plain-text data. Gentry published the first FHE scheme in 2009 in [38]. FHE is often paired with lattice-based schemes and consequently relies on similar core building blocks such as polynomial multiplication. Due to the need for frequent bootstrapping in FHE to maintain a decodable noise level in data, efficient hardware as well as efficient lattice-based schemes are being researched to pair with FHE. Several of these hardware papers are included in the following: [13], [14], [15],[19],[35]. Somewhat Homomorphic Encryption (SHE) is also referenced periodically. It is a homomorphic scheme which only supports a limited number of ciphertext computations before needing fully decoded.

2.5 The Cryptography Bottlenecks

As seen in Section 2.3, the introduction, and as determined by a host of Lattice-based Cryptography hardware acceleration papers, there are two primary bottlenecks of Lattice-based cryptographic schemes: Modular Polynomial multiplication and Gaussian sampling ([1], [2], [3], [4], [5], and [6]). The following sections are a review

of both and give the validation for choosing NTT polynomial multiplication as the primary focus for this project.

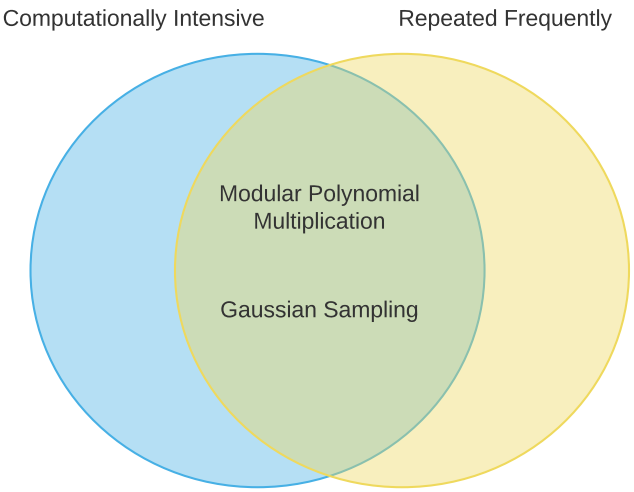

Figure 4. Lattice-based Cryptography's Bottlenecks. Operations which are computationally intensive and frequently used.

2.5.1 Gaussian Sampling

Gaussian sampling is the first mentionable mathematical component in lattice-based cryptographic schemes, however it is only relevant to the overall scope of Lattice-based Cryptography and not touched on again for the rest of the thesis. Gaussian sampling is important because matrices of random numbers are generated and used to encrypt information, with decryption performed using probabilistic inference. In LWE-based systems, the private key is a uniformly random matrix and the public key is calculated using a uniformly random matrix and a certain error distribution. A Gaussian sampler relies on the standard deviation of the samples σ, precision

parameter λ which controls how close to statistically ideal the distribution is, and τ which is the value on the tail of the distribution where all past values are ignored (sampling in $\{0, \sigma\tau\}$ instead of $\{0, \infty\}$) [2]. A higher standard deviation σ requires more memory. Precision parameter λ affects security as a more precise distribution is more secure but results in slower computation. A lattice-based Gaussian sampler pulls numbers from a discrete Gaussian over an n-dimensional lattice where the mean and standard deviation are input parameters $\mu \in \mathbb{R}^n$ and $\sigma > 0$. An additional term mentioned in Gaussian sampler discussion is statistical distance (Equation 2.4) from a desired distribution. On any computed distribution (and especially distributions drawn from resource-limited hardware), the generated distribution will not perfectly match an ideal distribution and the difference must be considered for the sake of ensuring cryptographic security.

$$\Delta(X, Y) = \frac{1}{2} \sum_{x \in L} |P(X = x) - P(Y = x)| \tag{2.4}$$

Here X and Y are two random variables and x is a distribution on a lattice L. Several ways to generate random numbers as well as accelerated versions in hardware are listed below. Additional methods not covered include the binomial method. According to [2], binomial is not suitable for digital signatures but it is listed among lattice-cryptography hardware papers as an alternative Gaussian sampling method.

Rejection sampling for Lattice-based Cryptography takes a uniformly random distribution and removes samples that fall outside of the range $(-\tau\sigma, \tau\sigma)$, bounds calculated using the tail cut value and standard deviation of a Gaussian distribution. For arbitrary probability distributions X and Y and respective probability density functions $X(x)$ and $Y(x)$, the distribution of Y can be generated by sampling X and throwing out samples with a probability $Y(x)/MX(x)$ where $M \geq 1$ and $Y(x) \leq$

$MX(x)$. M is generally thought of as the number of times the rejection will occur and the number of times sampling will need to be repeated. When the above conditions are true for all x, the exact distribution of Y will be produced and when they are true for only some x, the distribution will be statistically near to Y. Rejection sampling is inefficient because sampling needs to be repeated every time a sample is rejected and current algorithms may take multiple trials on average to accept a sample. More information on rejection sampling for lattice-based cryptography can be found in [39].

Knuth-Yao Gaussian sampling is performed by generating a binary tree based on arbitrarily chosen draw probabilities for each element of a set. With the probabilities of each draw written in binary and moving from most significant bit to least significant bit, terminating tree arms are drawn for binary places with a 1. At each new bit, the next layer of the tree is generated. It can be seen that with uniform probability of traversing the branching options at each tier and with the most significant binary bits being earlier in the tree, the probability of ending at any one element in the set can be accurately achieved via a uniform sampling regardless of the set's true probability distribution. The Knuth-Yao algorithm is beneficial because it takes a minimal number of initial random bits. The disadvantage of this method is that storage of the discrete distribution generating tree (DDG) requires ROM storage. For more information see [40], where a Knuth-Yao hardware design is proposed.

The Bernoulli sampler is a form of rejection sampling that does not rely on calculating the exponential function or performing a large number of repeat samples. Bernoulli was first implemented for lattice-based cryptography in [5] where the digital signature scheme BLISS was introduced. To avoid computing the exponential function, the authors produce samples of $e^{-x/f}$ using precomputed exponentials of powers of two $exp(-2^{i/f})$. They additionally replace the uniform distribution used in standard

rejection sampling with a discrete Gaussian, lowering the rejection rate from an average of 10 times per accepted sample to 1.5.

In [39], a bimodal distribution is used for lattice-based cryptography which achieved reduced table sizes from linear to logarithmic at the expense of longer execution time. The authors present three different rejection sampling methods where all are performed without using floating point arithmetic and where one method relies on no tables and two methods rely on small tables. A core algorithm in the paper is in generating a Bernoulli random variable with the probability of drawing a 1 equal to $2^{-z/k^2}$ where z and k are integers and $z < k^2$ using no floating-point exponential calculation. For more information on Bernoulli sampling, see [5].

Ziggurat sampling is a form of rejection sampling that divides the area under a Gaussian curve into rectangular areas that are equal in size. Increasing the number of rectangles lowers the probability of a sample being rejected. The method essentially optimizes rejection sampling using a uniform distribution. Ziggurat has been reported as unsuitable for hardware implementation by [2] in 2017 and does not have any hardware implementation. However in March 2018, a discrete Ziggurat hardware design is proposed with an architecture diagram in [40].

The Cumulative Distribution Table sampler (CDT) uses a lookup table filled with the discrete Gaussian cumulative distribution to perform Gaussian sampling using a uniformly random distribution. To converge on a value in the table, each iteration of the sampling algorithm removes half of the possible options. Given that the distribution is symmetric, the lookup table can be half as large. In [40], a CDT hardware design is the chosen sampler out of several hardware implementations, achieving 59.4 million samples per second for encryption and 16.3 million samples per second for signatures with minimal hardware resources. Algorithmic and hardware

improvements include hashing the most significant bits of the search space to make the lookup table smaller, reducing the standard deviation of the CDT, and using floating point numbers with a changing mantissa size.

While Gaussian number sampling is another bottleneck of Lattice Cryptography, the focus of this thesis is chosen to be polynomial multiplication using the NTT. For more information on sampling methods, see [3], [5], [6], [41], [42], [43], [44], and [45].

2.5.2 Modular Polynomial Multiplication

Polynomial multiplication is the primary bottleneck in lattice-based schemes and is the focus in this thesis. The multiplication between polynomials A and B with coefficients a_i and b_i in the form

$$A(x) = a_0 + a_1 x + ... + a_{n-1} x^{n-1}$$

$$B(x) = b_0 + b_1 x + ... + b_{n-1} x^{n-1}$$

results in the product polynomial C with coefficients c_i:

$$C = A(x)B(x) = c_0 + c_1 x + ... + c_{2n-2} x^{2n-2}$$

Polynomials are used as private and public keys and random numbers used for intermittent decryption/encryption steps are often times in the form of polynomials. Matrix multiplication is associated with standard lattices and makes computation slow and storage requirements high. Polynomials are associated with ideal lattices and are chosen over matrix multiplication when a ring-based scheme is used. This is beneficial because polynomials are generally more efficient and have lower space requirements from n^2 to $n \log n$ elements. Polynomial multipliers include integer multipliers and their complexities are measured treating n as the number of digits in the multiplicands.

The multiplication of two polynomials can be interpreted as the convolution between the polynomial coefficients. This is relevant in methods such as the FFT method, which make use of transforms to avoid the classical approach of convolution. A hardware and software survey of Lattice-based Cryptography in [2] gives a broad outline of the possible algorithms for modular polynomial multiplication. In order of descending computational complexity, these include the School Book, Comba, 3-way Toom-Cook, k-way Toom-Cook, Furer, Karatsuba, Schonhage-strassen, and FFT algorithms.

$$\text{Polynomial multiplication} = \begin{cases} \text{School Book: } \mathcal{O}(n^2) \\ \text{3-way Toom-Cook:} \mathcal{O}(n^{1.58}) \\ \text{k-way Toom-Cook: } \mathcal{O}(n^{\log(2k-1)/\log k}) \\ \text{Furer: } \mathcal{O}(n \log n * 2^{\log n}) \\ \text{Comba:} \mathcal{O}(n^2) \\ \text{Karatsuba:} \mathcal{O}(n^{1.58}) \\ \text{Schonhage-strassen:} \mathcal{O}(n \log(n) \log(\log(n)) \\ \text{FFT: } \mathcal{O}(n \log n) \end{cases}$$

A hardware analysis of modular multiplication in [4] and [46] directly compares these methods, with some of the results written below.

2.5.2.1 School Book Multiplication

The schoolbook polynomial multiplication algorithm has complexity $\mathcal{O}(n^2)$ and is based on multiplying the coefficients of A and B directly using convolution. The new

product polynomial C with coefficients c_i is found by multiplying each coefficient in a_i by every b_j and is written as:

$$C = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \tag{2.5}$$

At the lowest level, the schoolbook algorithm for integer multiplication can be accomplished with just shift and additions. Starting with two n-bit integers, one of the binary-represented integers is shifted 2^i times and added to an accumulator for each i^{th} bit of the other integer that equals 1. The process for polynomial multiplication is given in Algorithm 1.

Algorithm 1: Schoolbook Convolution Polynomial Multiplication

Input : Polynomials $A, B \in \mathbb{Z}_M[x]/(x^n + 1)$ of length n.

```
C = 0
for i = 0 to n − 1 do
    for j = 0 to i − 1 do
        sign = (−1)^⌊(i+j)/n⌋
        index = (i + j) mod n
        coef = a_i b_i mod M
        c_index = int(c_index + sign × coef) mod M
```

Output : $C = \{c_0, c_1, ...c_n\} = A \times B$

Despite the algorithm's high computational complexity, the schoolbook algorithm has been chosen in hardware design papers such as [23] over more efficient polynomial multiplication algorithms because of it's simplicity when implemented using systolic arrays. The authors create both a convolution and NTT based systolic multiplier and achieve performance gains over the standard sequential NTT implementation of a polynomial multiplier. The systolic convolution was performed using a series of processors that contained multiply-accumulators (MACs) in combination with modular reduction blocks. Using n MAC and n reduction processing blocks resulted in

complexity lowering from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The systolic schoolbook algorithm was able to finish processing a polynomial multiplication several times faster than the systolic NTT and sequential NTT, at the expense of significantly more energy expenditure and more DSP, FF, and LUT blocks. For future projects, it would be logical to look at other overlooked polynomial multiplication algorithms other than the NTT for hardware acceleration in the same way as the schoolbook algorithm was looked at and implemented.

2.5.2.2 Comba Multiplication

Comba is a scheduling multiplication algorithm with computational complexity of $\mathcal{O}(n^2)$ similarly to the schoolbook algorithm and improves on it by reordering the partial products encountered in the multiplication [47]. The algorithm has been looked at for cryptographic hardware applications because it requires fewer resources when compared to the schoolbook algorithm and reduces the amount of read and write operations by using DSP blocks on FPGAs. Analysis on the architecture in [47] reveals that the number of DSP blocks increases linearly with the binary length of the multiplied numbers but also shows that the structure of Comba multiplication does not allow for pipelining when designed in hardware. When two integers of length n words are multiplied, Comba requires $2n - 1$ partial products. Diagrams in [47] provide the multiplier architecture for the Comba multiplier, as well as the Karatsuba multiplier and could form the basis for hardware improvements.

2.5.2.3 k-way Toom-Cook Multiplication

Toom-Cook is the general name for an integer multiplication algorithm which splits a multiplication's operands into k smaller integers to perform smaller operations on each and to reduce complexity. 3-way Toom-Cook is the Toom-Cook algorithm when $k = 3$ and has complexity $\mathcal{O}(n^{1.46})$. The algorithm was introduced in 1963 by Toom [48] and revisited in 1969 by Cook [49] to produce an improved version of the algorithm. Toom-Cook is a generalized version of Karatsuba, in which the integers are split into two parts. This means the hardware for both algorithms looks similar, however the Toom-Cook methods requires more memory for intermediate values making Karatsuba the chosen polynomial multiplication algorithm over Toom-Cook in several papers despite having slightly higher computational complexity over 3-way Toom-Cook.

2.5.2.4 Karatsuba Multiplication

Karatsuba was introduced in 1962 in [50] and has complexity $\mathcal{O}(n^{1.58})$. It is a special instance of the k-way Toom-Cook algorithm where the two multiplied integers are separated into two parts. Karatsuba is based off of the idea that an integer of length n-bits can be represented by the addition of it's least significant bits and it's most significant bits multiplied by a base $2^{n/2}$ (i.e. shifted by $n/2$ bits). Given two integers that are split between the upper and lower bits in this fashion, the algorithm for Karasuba multiplication is given in Algorithm 2.

Karatsuba has found success when it's operands are of 32 bits or more. Karatsuba works by breaking down the large initial multiplication into several $n/2$-bit multipli-

Algorithm 2: Karatsuba Polynomial Multiplication

Input : n-bit integers A and B, split into l least significant bits A_0, B_0 and $n - l$ most significant bits A_1, B_1

1. Determine (A_1, A_0) and (B_1, B_0) where $A = A_1 \times 2^l + A_0$ and $B = B_1 \times 2^l + B_0$
2. $t = \begin{pmatrix} t_0 & t_1 & t_2 & t_3 \end{pmatrix} = \begin{pmatrix} A_0 B_0, & A_1 B_1, & A_1 + A_0, & B_1 + B_0 \end{pmatrix}$
3. $m = (t_2 \times t_3) - t_0 - t_1$
4. $C = t_0 + m \times 2^l + t_1 \times 2^{2l}$

Output : $C = A \times B$

cations and several additions and subtractions. When considering the multiples of powers of two as shifts, the algorithm runs using only three multiplications. Karatsuba is a computationally efficient multiplication algorithm and is applied in many hardware level designs of lattice-based cryptosystems. Diagrams in [47] show the architecture of a Karatsuba multiplier with Comba multiplier units as sub-units and could show the basis for further hardware acceleration.

2.5.2.5 Schönhage-Strassen Multiplication

The Schönage-Strassen integer multiplication algorithm [51] was introduced in 1971 and found computational complexity improvements over Toom-Cook with a complexity of $\mathcal{O}(n \log n \log \log n)$ when dealing with several thousand decimal digit numbers. It is used in the established GNU Multiple Precision Arithmetic Library (GMP), one of the most commonly used libraries for handling large integers and floating point numbers without limitations to precision. Schönage-Strassen is an FFT-based integer multiplication algorithm which uses the NTT. It is generally not chosen over Karatsuba unless numbers being used are significantly large. A result which provides an upper bound to the Schönhage-Strassen multiplication method is seen in [52].

2.5.2.6 Fürer Multiplication

The Fürer integer multiplication method was introduced in 2007 and is an asymptotically faster algorithm than the Schönhage-Strassen algorithm but only for very large numbers. While it uses complex numbers to achieve fast multiplication, ideas from it were taken and used to create a modular polynomial multiplication algorithm in [53]. The core idea of Fürer is to reduce a large multiplication to many exponentially smaller multiplications that are then performed recursively. While the algorithm is well researched and the inspiration for many variants, the Fürer polynomial multiplication method is rarely the chosen algorithm for lattice-based schemes due to it's impracticality and ineffectiveness on small numbers.

2.5.2.7 FFT Multiplication

In this thesis, the FFT polynomial multiplication method is realized using NTT. This method is chosen because of its flexibility, lowest computational complexity, extensions into hardware, and established foothold in existing literature. The RNS-based NTT utilizes the same algorithm as the NTT but replaces arithmetic with RNS operations by distributing the original workload among k parallel channels. The NTT is discussed in detail in Section 3.5.

Chapter 3

DESIGN OF THE RNS-BASED NTT POLYNOMIAL MULTIPLIER

This thesis focuses on an RNS-based NTT polynomial multiplier for accelerating lattice-based and FHE schemes. Development of the NTT modular polynomial multiplier unit is broken into two components; a software implementation in C++ and hardware implementation in Verilog [1]. The software design is a proof-of-concept of the entire system an the algorithmic level and is also used to produce initialization files for the FPGA design. The hardware design is the realization of the algorithmic system and used to report on area, power, and timing expectations for the optimized system. Both are scalable to varying polynomial lengths, ring sizes, and channel widths. The following is a review of the system's parameters, modular functions, RNS application, NTT computation, and outer negative wrapped convolution algorithm.

3.1 Parameter Sets

A table including popular parameter sets for lattice and FHE schemes is provided in Table 4. The chosen target to approximately match this polynomial multiplier design's parameter set is [19] and [54]. For FHE schemes, the design targets polynomial length $n = 4096$, modulus bitwidth $w = 128$, RNS channels $k = 4$, and channel width $w_{ch} = 32$. For lattice schemes, this can be lowered to $n = 1024$, modulus bitwidth $w = 32$, RNS channels $k = 4$, and channel width $w_{ch} = 8$.

[1] All C++ and Verilog code can be found at: <https://github.com/LoganBrist/C-Verilog_NTT_RNS_Polymultiplier>

name	year	Scheme	n	M	w
[1]	2012	LWE	128 256 512	3329 7681 12289	12 13 14
[55]	2014	R-LWE	256 512	7681 12289	13 14
[54]	2014	R-LWE	256 512	7581 12289	13 14
[8]	2015	R-LWE	256 512 2048	65537 8383489	30
[56]	2016	R-LWE	256 512 1024	12289	32+
[57]	2017	R-LWE	256 512	7681 12289	13 14
[23]	2020	R-LWE	128 256 512 1024	-	-
[13]	2014	R-LWE/SHE	256 1024 2048	$2^{21}-k, 2^{30}-k$	21 30
[14]	2015	FHE	2^{15}	$2^{32}-k$	32
[15]	2015	FHE	4096 2^{14}	$2^{w}-2^{k}+1$	124 512
[58]	2018	FV SHE	2^{15}	$2^{1228}-k$	1228
[19]	2020	FV SHE	4096	$2^{180}-k$	180

Table 2. Parameter set for R-LWE, SHE, and FHE lattice-based accelerators

The chosen parameters are highly flexible due to the nature of RNS. Lattice-based Cryptography is of interest because it has relatively low key sizes while FHE schemes have significantly larger coefficient widths and polynomial lengths that can better utilize the RNS. This design targets both, to make an ideal high speed and low resource multiplier that is flexible.

3.2 Design Components

At the algorithmic level, polynomial multiplication is accomplished using a NTT approach. The hardware design realizes this with a single NTT unit containing 4 butterfly units. The wide range of alternative multiplication methods can be seen in [2] as well as in Chapter 2. An NTT approach is chosen because of its flexibility with hardware implementation and its $\mathcal{O}(n \log n)$ complexity when implemented with an FFT butterfly structure. At the highest system level, using the NTT means the polynomial multiplication can be implemented using a negative wrapped convolution. This algorithm allows input polynomials to be unchanged in size compared to the

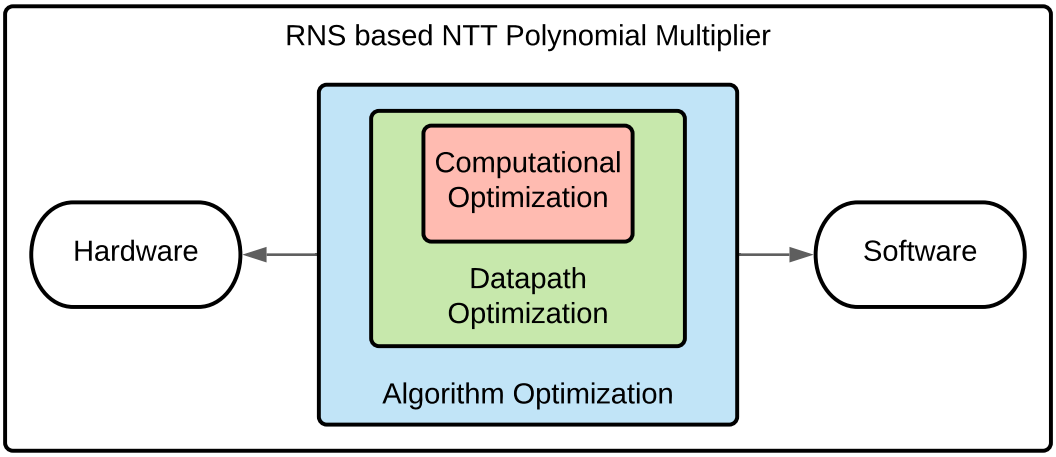

Figure 5. Hierarchy of Design for the RNS based NTT Polynomial Multiplier

zero padded inputs in the NTT. The NTT approach also enables the use of parallel NTT butterfly units when implemented on an FPGA. Using p parallel units turns complexity into a further improved $\mathcal{O}(n/p \log n)$ and achieves significant FPGA-enabled speed-up before any small scale hardware optimizations are made. For these same reasons, the NTT polynomial multiplication method has been a popular choice in other lattice-based cryptography hardware accelerators ([7], [8], [15], [22]).

At the datapath level, parallelism is achieved by completing the full NTT in an RNS. This design chooses an RNS of $k = 4$ (9 channels total) with $w = 32$. RNS has been used in a variety of applications ([13], [20], [59], and [60]), however few lattice-based cryptography designs have utilized it. Employment of an RNS was chosen because a polynomial multiplication can require modular arithmetic operations with bit-widths ranging anywhere from < 32 bits for lattice cryptography and leveled FHE schemes ([18] and [56]) to 180-372 bits in large FHE schemes ([19] and [61]). An RNS can break these potentially large arithmetic operations into smaller channel

operations and can consequently bring lower propagation delays to the repeated large word length arithmetic that is found in the NTT butterfly. To accomplish the required modular multiplication in RNS, an optimized RNS Montgomery multiplication with efficient Shenoy [25] and Bajard [26] base extensions is used. RNS Montgomery is chosen over a selection of other RNS modular multiplication methods seen in [62] for its overall lower size when compared to Sum of Residues and higher speed when compared to Barrett reduction.

Lastly at the computational level, all modular channel arithmetic of the RNS in the polynomial multiplication unit is simplified and streamlined using bit select-based 32-bit modular addition, modular subtraction, and Barrett reduction. Individual RNS channels use this optimized modular arithmetic and can additionally be improved using properly chosen RNS moduli. With the proper reduction values, individual modular arithmetic blocks can be simplified or turned into logic expressions. Arithmetic units such as the base extensions and accumulators are also streamlined by being pipelined and having internal MAC units unrolled into parallel multiplications, which allows a fluent pipeline of the entire design. With properly placed registers, the average throughput of one butterfly operation is one per cycle, as long as it is when all registers are being used. The critical path in the overall design is also shortened by placing registers intermittently, thereby increasing the maximum allowable clock frequency.

3.3 Modular Functions for Precomputations and Channel Operations

Modular functions are used to calculate all precomputed constants, channel operations, and cryptographic functions in the polynomial multiplier and are especially prominent in precomputations in the software implementation. Modular arithmetic is

the collection of standard integer arithmetic operations where computed results that are greater than or equal to a modulus M are wrapped back into the range $[0, M-1]$ and it is intertwined in the NTT and Residue Number System. Modular reduction is used in all modular arithmetic and is written as

$$X \bmod M \tag{3.1}$$

for some integer X and a chosen modulus M. The modular reduction is equivalent to performing a division between X and M and only keeping the remainder (e.g. $17 \bmod 5 = 2$, as $17/5 = 3$ rem 2). A general arithmetic approach to performing a modular reduction is

$$X \bmod M = X - \lfloor \frac{X}{M} \rfloor \times M \tag{3.2}$$

which achieves reduction via a floored division. Modular reduction similarly can be thought of as arithmetic performed on a wrapped number line, where the values at the boundary of the line wrap around to the beginning:

$$X = \{M-1, M, M+1\} \bmod M = \{M-1, 0, 1\} \tag{3.3}$$

Extended to any arbitrary computation resulting in $X \geq M$, the reduction results for

$$X = \{nM, nM+1, nM+2, ..., nM+(M-1)\}$$

results in

$$X \bmod M = \{0, 1, 2, ..., (M-1)\}$$

This is when provided a sufficiently large modulus M and some positive integer n. For a negative X, the reduction wraps in the opposite direction around the circular

number line. Expressed in the same way as addition, the reduction result for

$$X = \{nM, nM - 1, nM - 2, ..., nM - (M - 1)\}$$

results in wrapping to

$$X \bmod M = \{0, M - 1, M - 2, ..., 1\}$$

This again is when provided a sufficiently large modulus M and some negative integer n. Negative modular reductions are generally not necessary in cryptographic schemes as the schemes rely on additive and multiplicative functions and are closed in $[0, M - 1]$ by those operations. When not dealing with assignments such as ($c = a \bmod b$), modular arithmetic equations often replace equalities ($=$) with congruences ($\equiv$) as the computation of modular arithmetic functions result in the same value for integer multiples of the modulus. For example, both

$$(5 + 4) \bmod 7 = 2$$

$$(5 + 11) \bmod 7 = 2$$

therefore both are congruent to each other and can be written as

$$5 + 4 \equiv 5 + 11 \equiv 2 \pmod 7$$

The equation leaves the mod at the end of the line and it is interpreted as applying to the entire line. When written as a congruence, $\bmod M$ is parenthesized and is no longer the modulus operator but an expression claiming a common modulus across the entire equation. Modular arithmetic is vital in cryptography because operations take place in a well defined finite sized set (finite field or ring). The wrapping nature of modular functions makes modular reduction fitting for cryptography because it produce results that are not easily linked back to function inputs and consequentially

introduce difficulty in cryptographic problems. Cryptographic schemes rely on fields $\mathbb{Z}_q$ and rings $\mathbb{Z}_{q_{prime}}$ and consequently modular arithmetic is a natural way to realize a field combined with easy mathematical properties.

Modular functions used in the design include:

- Montgomery Multiplication - To achieve reduction in the RNS-based NTT unit
- Barrett Reduction - To achieve reduction on individual RNS channels
- Modular inverse - To precalculate n-th root/Base Extension Inverses (in software)
- Modular square root - To precalculate NTT twiddle factors (in software)
- Modular n-th root - To find n-th root of unity for NTT (in software)

3.3.1 Montgomery Multiplication

Montgomery multiplication [63] is a modular multiplication scheme which uses a Montgomery constant R in replacement of a modulus M to achieve fast modular reduction. Montgomery multiplication is important in this RNS-based NTT polynomial multiplier as fast modular multiplication is necessary to perform an efficient NTT. The classical Montgomery multiplication algorithm is described in Algorithm 3.

Algorithm 3: Classical Montgomery Multiplication

Input: A, B

Data: R co-prime to M and $R > M$, M' and R^{-1} such that $R^{-1} < M$ and $\bar{M} < R$ and $RR^{-1} - M\bar{M} = 1$.

1 $X = A \times B$
2 $m = X \bmod R$
3 $n = m \times \bar{M} \bmod R$
4 $t = (X + mM)/R$
5 **if** $M \leq t$ **then**
6 $\quad t = t - M$

Output: $t = ABR^{-1} \bmod M$

For computational efficiency, the Montgomery constant R ideally is a power of 2 as it allows the reduction's division to be replaced by bitshifts. The calculation of constants makes Montgomery multiplication not efficient for single multiplications, however it is highly efficient when repeated and the initial conversion is unneeded. A Montgomery Multiplication result is not a full reduction as it comes in the form $ABR^{-1} \bmod M$. As given in Equation 3.4, the Montgomery Multiplication can be adjusted to produce a fully reduced result $AB \bmod M$ by multiplying both input values by R^2.

$$Z = ABR^{-1} \times R^2 \times R^{-1} \bmod M \tag{3.4}$$

$$= AB \bmod M \tag{3.5}$$

Montgomery Multiplication is the chosen modular multiplication method in the RNS-based NTT polynomial multiplier. It is realized as an RNS Montgomery multiplication in the NTT unit (NTT discussed in Section 3.5). The adjustments that are needed to perform it include two base extensions. These changes are detailed in Section 3.4.3.

3.3.2 Barrett Reduction

Barrett Reduction [64] is a fast modular reduction technique that works by estimating the floor function of the standard modular reduction (Equation 3.2) without division. Barrett relies on two precomputed values L an K. K is chosen to be the smallest value that makes $1/M - (\lfloor 2^K/M \rfloor / 2^K) < 1/M^2$ true. Then one can solve $L = \lfloor 2^K/M \rfloor$. The Barrett multiplication algorithm is presented in Algorithm 4.

Algorithm 4: Barrett Modular Multiplication for channel multiplications

Input: A, B

Data: Smallest K such that $1/M - (\lfloor 2^K/M \rfloor / 2^K) < 1/M^2$, $L = \lfloor 2^K/M \rfloor$

1 $Z = A \times B$

2 $T = Z \times L >> K$

3 $Z = Z - (T \times M)$

4 **if** $Z \geq M$ **then**

5 $\quad Z = Z - M$

Output: $Z = AB \bmod M$

Barrett reduction is the reduction method used in this design for channel reductions on the individual RNS channels. As opposed to the RNS-based NTT's modular multiplication, it is chosen over the Montgomery method because of its flexibility with channel moduli of varying size and varying mathematical properties. To ensure the conditions on K are met for arbitrary M, this design's hardware implementation assumes K = $2\lceil log_2(M) \rceil$. The software implementation calculates them precisely for the chosen moduli set. The hardware results for Barrett reduction units are given in Section 4.7.1.

3.3.3 Modular Inverse

Modular inverse is used to calculate variables such as ω_n^{-1}, ϕ^{-1}, and D_i^{-1} in the pre-computation stage of the software polynomial multiplier design. The modular inverse of A is X if

$$A \times X \equiv 1 \bmod M. \tag{3.6}$$

The software design of the polynomial multiplier utilizes the modular inverse function in the BigInteger Library [65] which furthermore utilizes the Euclidean

algorithm. The constants calculated from modular inverse are provided in Section 3.4.2.

3.3.4 Modular Square Root

Modular square root is used to calculate $\phi^2 = \omega_n$ in the software design. There are multiple ways to perform the calculation including Tonelli-Shank's algorithm [66]. Depending on whether the modular square root exists or not, the equivalency can have two or zero solutions. The square relationship between A and B can be represented as

$$A^2 \equiv B(\mathrm{mod}M) \tag{3.7}$$

In the software implementation, this value is discovered by repeated trials of the modular exponentiation function in [65]. The variable associated with this is discussed in Section 3.4.2.

3.3.5 Modular n-th Root

The modular n-th root is the conceptual relation behind this design's NTT root of unity. It is expressed as

$$A^n \equiv B(\mathrm{mod}M) \tag{3.8}$$

The modular n-th root is not calculated directly in the software implementation. It is determined via generator functions that narrow the search for the root. Finding the n-th root is given in Section 3.5.3.

3.4 Using the Residue Number System

The Residue Number System (RNS) is the primary means of accelerating the polynomial multiplier at the datapath level. The polynomial multiplier design embeds RNS in it's variable storage, the NTT unit's addition/subtraction/reductions, and the NTT's computationally expensive modular multiplication.

3.4.1 RNS Operation

RNS is an arithmetic scheme where mathematical computations on large bit-width numbers is divided into smaller independent modular operations on multiple thinner channels. RNS is the chosen method to optimize the NTT polynomial multiplier on a datapath level because of its parallelization, lack of carry arithmetic, and division of workload for large arithmetic. An RNS is defined by a chosen base of k relatively co-prime moduli $(m_1, m_2, ..., m_k)$. For each of these values, the RNS has an independent channel performing arithmetic over its corresponding channel modulus that contributes to the overall correct computation. The dynamic range of a specific RNS system is the product of its channels $D = \prod_{i=1}^{k} m_i$. Representing an integer in RNS is done by keeping the remainders of an integer after it is divided by each channel modulus. For an integer X, the RNS representation consists of as a set of residues written as

$$X_{RNS} = [x_1, x_2, ...x_k] = X \bmod m_i \qquad \text{for } i \in [1, k]$$

where X is guaranteed to have unique representation in RNS form for as long as $0 \leq X < D$ by the Chinese Remainder Theorem (CRT) [67]. The CRT states that if

p, q are co-prime, the following set of equations

$$x = a \bmod p$$

$$x = b \bmod q$$

have only one unique solution for x mod pq. With this scheme of representation in place, one can perform further operations on RNS numbers and retrieve integer values after those results are complete.

Addition, subtraction, and multiplication are straightforward in an RNS. The arithmetic $x = a \oplus b$ between two RNS integers and where $\oplus$ is one of the operators $[+, -, x]$ can be performed as

$$X_{RNS} = [x_1, x_2, \ ... \ x_k] = [a_1 \oplus b_1, a_2 \oplus b_2, \ ... \ a_N \oplus b_k] \tag{3.9}$$

This set of operations can be performed exclusively and in parallel within each individual channel to produce another RNS number that is the correct result as long as the resulting computation still falls within the range $[0, D-1]$. If an operation returns a value outside of this range, the value will wrap with D and result in an overflow. Taking advantage of overflow is initially appealing to achieve free modular arithmetic within an RNS, however any prime modulus will not be representable as the wrapping value because the dynamic range is a product. Signed representation and arithmetic with these operations is possible. To include signed values, one must assign a fraction of the dynamic range to cover negative numbers and shift all output values. This is similar way to the wrapping of binary 2's complement.

RNS is advantageous for this design because channel operations are smaller, carry-free, and independent of one another compared to a standard NTT. The operations needed in the system are also RNS-friendly as the butterfly operation involves mainly

addition and subtraction. These operations are low level arithmetic on large bit-width values and can be repeated without converting out of RNS. In traditional systems, core arithmetic functions like multiplication have a high propagation delay due to the carry's involved in adding multiple partial products. RNS spreads this load over multiple smaller multiplications and in parallel among fewer bits, only to be converted back into integer (binary) form later for a usable answer.

With these benefits acknowledged, it should be noted that conversions to, and especially from the RNS system are not efficient as they require several full resolution multiplies, accumulates, and modular reductions. To make the system efficient, it is important to remain in RNS form for an entire polynomial multiplication. The treatment of RNS values when being converted conversion is discussed in Section 3.4.6. Additionally, arithmetic such as division, rounding, magnitude comparison, and modular reduction are not straightforward in an RNS system so to retain RNS form throughout a multiplier's modular multiplication, an intelligent RNS modular multiplication unit needs investigated.

Choosing the RNS moduli is also an important aspect of the hardware design for the individual channel arithmetic. Properly chosen values turn many of the modular operations into simplified logic. Enabling these improvements by selecting appropriate moduli is discussed in Section 4.1. A review of the RNS system and its datapath applications can also be found at [68].

3.4.2 System RNS Initialization

Before usage, run-time for base extensions and RNS conversion can be decreased if a few constants are derived from the moduli set $m_i = \{m_1, m_2, ...m_k\}$ at instantiation. The first of these constants are the dynamic range

$$D = \prod_{i=1}^{k} m_i \tag{3.10}$$

the dynamic range divided by each channel modulus

$$D_i = \frac{D}{m_i} \tag{3.11}$$

and the modular inverse of that result with respect to each channel moduli m_i.

$$D_i^{-1} = \text{modinverse}(D_i, m_i) \tag{3.12}$$

These three constants are used in the convert-to-integer weight calculation for a simple RNS. Using RNS in the NTT additionally requires the use of a Montgomery reduction as explained in Section 3.4.3.

$$\text{Ext. Constants} = \begin{cases} \left|D_i^{-1}\right|_i & = \text{modinverse}(\frac{D}{m_i}, m_i) \\ \left|D_i\right|_j & = \frac{D}{m_i} \bmod m_j \\ \left|\hat{D}_j^{-1}\right|_j & = \text{modinverse}(\frac{\hat{D}}{m_j}, m_j) \\ \left|\hat{D}_j\right|_r & = \frac{\hat{D}}{m_j} \bmod m_r \\ \left|\hat{D}_j^{-1}\right|_r & = \text{modinverse}(\frac{\hat{D}}{m_j}, m_r) \\ \left|\hat{D}_j\right|_i & = \frac{\hat{D}}{m_j} \bmod m_i \\ \left|\hat{D}\right|_i & = \hat{D} \bmod m_i \end{cases}$$

The above are found for $i = \{1, 2, ...k\}$ and $j = \{k+1, k+2, ...2k\}$. The precomputed constants from the Bajard extension include: $|D_i^{-1}|_i$ and $|D_i|_j$. For the Shenoy

extension these are: $|\hat{D}_j^{-1}|$, $|\hat{D}_j|_r$, $|\hat{D}_j^{-1}|_r$, $|\hat{D}_j|_i$, and $|\hat{D}|_i$. In the hardware design, these values are saved in local memory at each base extension.

The software implementation of the RNS-based NTT polynomial multiplier finds all of the above constants in the precomputation step of the program. They are then saved as text files and used as parameters in the hardware implementation.

3.4.3 RNS Montgomery Multiplication in the NTT

RNS Montgomery Multiplication is used in the design's RNS NTT to perform reduced multiplication. There are multiple known methods to perform modular reduction in RNS ([60], [62], [69]) which include the core function classical modular multiplication, the short word length modular multiplication using the look-up table method, the RNS Barrett Reduction, the Sum of Residues method, and the RNS Montgomery Multiplication. The modular multiplication utilized in the NTT is chosen to be an RNS Montgomery Multiplication. Each of these methods are carried over from a standard positional number system, with appropriate adjustments made to make them RNS applicable. The look-up table method uses tables to reduce computational requirements, however it is found to be slow and hardly implementable for larger word lengths given the significant storage space making it a non-realizable choice. The classical core function method attempts to execute the traditional reduction in Equation 3.2 by solving for $\lfloor \frac{X}{M} \rfloor$ using the core function.

Unfortunately, the floor function requires knowing the relative magnitude of numbers in RNS which is difficult and consequently requires the core function to computationally intensive. For long word length RNS operations, the core function method was found to be infeasible compared to the Barrett, Montgomery, and Sum

of Residue methods. RNS Barrett reduction is performed by similarly finding the reduction with Algorithm 4. While most of the operations can be simply performed in RNS, it was found that delay for RNS Barrett is higher than RNS Montgomery for larger than 20-bit width RNS modular multiplication. Sum of Residues worked to find a partially reduced result that can then have bounds placed on it to achieve optimization. An improvement to the Sum of Residues method can be found in [62] and [70]. The algorithm was found to be the most competitive to RNS Montgomery Multiplication in terms of speed, however it was highly inefficient in space for little speed improvement. For these reasons, RNS Montgomery Multiplication was found to be the most optimal with regard to speed and size. A detailed review of each method can be found in [62].

3.4.3.1 Chosen Reduction Method

RNS reduction is performed in the software and hardware design using an RNS-adjusted version of the Montgomery Multiplication with Bajard and Shenoy extensions. It is based on the proposed algorithm by Bajard in [26], who created the approximated base extension that is utilized in the algorithm. In the algorithm, the Montgomery constant R is chosen to be the dynamic range, D, to achieve reduction. This comes at the expense of not being able to express D in base 1, and therefore a second base is needed. To allow conversion to a second RNS base in Montgomery multiplication, an additional set of coprime moduli

$$\hat{m} = \{m_{k+1}, m_{k+2}, ...m_{2k}\} \tag{3.13}$$

and one redundant modulus m_r needs to be chosen. To distinguish between bases, the original moduli is referred to as base 1 and uses the i index while the additional

moduli is referred to as base 2 and uses the j index as well as hats on variables. Montgomery reduction requires two base extensions to be performed. These include pre-computable constants that are explained in Section 3.4.2. The RNS Montgomery Multiplication algorithm is described in Algorithm 5.

Algorithm 5: RNS Montgomery Multiplication algorithm

Input: A, B

1 $X = A \times B$; ▷ `in parallel for` $\{m_1, m_2, ...m_k\}$

2 $Q = X \times -M^{-1}$; ▷ `in parallel for` $\{m_1, m_2, ...m_k\}$

3 $\hat{Q} = \text{baseExtension}(Q)$

4 $\hat{Z} = (\hat{X} + \hat{Q}\hat{M})\hat{D}^{-1}$; ▷ `in parallel for` $\{m_{k+1}, m_{k+2}, ...m_{2k}, m_r\}$

5 $Z = \text{baseExtension}(\hat{Z})$

Output: $Z = ABD^{-1} \bmod M$

As seen in [62], RNS Montgomery reduction requires $4k + 2$ multiplications and $k + 1$ additions on the outside of the base extensions. RNS Montgomery reduction is speed and size efficient for larger channels when compared to Barrett reduction and Sum of Residues but at the cost of requiring arithmetic in an additional base. This necessitates the use of an efficient base extension.

The simplest method of base extension is integer conversion, where a value is reverse converted to integer form and forward converted to a new base. This is highly inefficient. The Shenoy base extension and Bajard base extension are efficient means of achieving extension without leaving RNS form. Shenoy [25] is an exact extension which produces an RNS result of the true integer value when converted out of the system. Bajard [26] is an estimated extension which produces an RNS representation that is valid for future computation but offset by a certain calculable multiple of the dynamic range. As seen later, a pairing of estimated and exact base extensions can be used together to balance speed and resource consumption if that offset is changed to an offset of the modulus. Other applications of RNS Montgomery multiplication can be

found in [69] where a new algorithm and VLSI design for Montgomery multiplication in RNS was suggested. The design also came with improved forward and reverse conversion units. RNS Montgomery multiplication is also used in [71] where the objective is to perform Modular exponentiation. The well documented report obtains significant acceleration.

3.4.4 Bajard Base Extension from Base 1 to Base 2

The Bajard base extension [26] is used as the first extension in this design's RNS Montgomery multiplication and achieves increased speed and lower space by ignoring the need for an exact extension. It is also seen in extension papers [59] and [72]. As long as initial conditions are met, a slight offset α is allowed to appear in the output. This falls within a certain integer multiple of the modulus and can be corrected in the Shenoy base extension or at the end of repeated Montgomery multiplication. As seen in [71] and Equation 3.14, a value of Q base extended using the Bajard extension produced a result of $\hat{Q}$.

$$\hat{Q} = \left| \sum_{i=1}^{k} \left| q_i D_i^{-1} \right|_i |D_i|_j \right|_j = Q + \alpha D \tag{3.14}$$

Here D_i is equal to D divided by moduli i and $||_i$ is a reduction by channel i. The calculation that occurs in between base extensions in the RNS Montgomery multiplication changes this offset by a factor of the dynamic range to an offset of solely a scaled modulus. Inclusion of the offset factor is expressed as the following.

$$Z = (X + Q'M)D^{-1} \tag{3.15}$$

$$= (X + QM + \alpha DM)D^{-1} \tag{3.16}$$

$$= (X + QM)D^{-1} + \alpha M \tag{3.17}$$

Now that the offset is a multiple of the system modulus the calculation can continue with congruence. However, an exact extension is needed on the back end of the Montgomery multiplication because the offset turns into a multiple of D in the second base and that removes congruence when only dealing with a system modulus M. The algorithm for the Bajard base extension is the following:

Algorithm 6: Bajard base extension algorithm

Input: A_i in $\{m_1, m_2, ...m_k\}$

1 $\sigma_i = A_i \times |D_i^{-1}|_i$; ▷ `in parallel for` $\{m_1, m_2, ...m_k\}$

2 $t = 0$

3 **for** *i = 1,...k* **do**

4 $\quad t = (t + \sigma_i \times |D_i|_j)$; ▷ `in parallel for` $\{m_{k+1}, m_{k+2}, ...m_{2k}, m_r\}$

5 $\quad A_j = t$

Output: A_j in $\{m_{k+1}, m_{k+2}, ...m_{2k}\}$ and m_r

Bajard takes k^2+2k multiplications and k^2+k additions. The initial conditions for Bajard are that $(k+2)^2M < D$ and $(k+2)M < \hat{D}$. The output is then $Z < (k+2)M$, a certain multiple of the modulus no greater than the number of moduli + 2.

3.4.5 Shenoy Base Extension from Base 2 to Base 1

The Shenoy base extension is used as the second extension in this design's RNS Montgomery multiplication algorithm and was presented in [25] by Shenoy and Kumaresan. It utilizes an extra moduli in the original base to produce an exact

extension into the new base. When used in combination with the Bajard extension, the extra moduli produced in the forward extension can be used as the redundant moduli required in the Shenoy extension. It obtains slightly less efficient performance as the Bajard extension but ensures a usable result on the output of the RNS Montgomery Multiplication unit. The Shenoy unit is similar to the Bajard unit in that it relies on the Chinese Remainder Theorem, except the offset α is fully calculated so it can be removed.

Algorithm 7: Shenoy base extension algorithm

Input: A_j in $\{m_{k+1}, m_{k+2}, ...m_{2k}\}$ and m_r

$E_j = A_j \times |\hat{D}_j^{-1}|_j$; ▷ `in parallel for` $\{m_{k+1}, m_{k+2}, ...m_{2k}\}$

$t = 0$

for $j = k+1, k+2, ...2k$ **do**

  $t = t + E_j \times |\hat{D}_j|_r$

$\beta = |\hat{D}^{-1}|_r \times (t - A_j[2k]) \bmod m_r$

$t = 0$

for $j = k+1, k+2, ..2k$ **do**

  $t = t + E_j \times |\hat{D}_j|_i$; ▷ `in parallel for` $\{m_1, m_2, ...m_k\}$

  $t_i = t$

$A_i = (t_i + m_i) - (\beta \times |\hat{D}|_i) \bmod m_i$; ▷ `in parallel for` $\{m_1, m_2, ...m_k\}$

Output: A_i in $\{m_1, m_2, ...m_n\}$

Shenoy takes $k^2 + 3k + 1$ multiplications and $k^2 + 2k + 1$ additions. The redundant moduli m_r must be $\geq n$ and similarly coprime to the remaining moduli in the two bases. The output of the algorithm is then the exact conversion from the input base to output base.

3.4.6 RNS Forward and Reverse Conversion

RNS conversion is a functional block on the outside of this design's RNS-based NTT unit which is needed to perform the conversion to and from RNS before and after the negative wrapped convolution begins. As mentioned in earlier sections, forward conversion from a binary integer X to X_{RNS} is performed as

$$x_i = X \bmod m_i \tag{3.18}$$

This reduction has the benefit of being implementable in parallel over k channels and is reduced only by an w_{ch}-wide modulus m_i, however it still suffers in overhead because X is the size of the original non-RNS polynomial elements and consequently of size $\lceil \log_2 M \rceil$. To compare to the standard channel modular multiplication in RNS, RNS operations only take a $2\lceil \log_2 m_i \rceil$ bit input and reduces it by a $\lceil \log m_i \rceil$ bit channel modulus. To decode an RNS value X_{RNS} from RNS representation $\{x_1, x_2, ...x_k\}$ via the Chinese Remainder theorem, Equation 3.19 is used.

$$X = \left| \sum_{i=1}^{k} D_i |D_i^{-1}|_i \times x_i \right|_D \tag{3.19}$$

This computation includes several large multiplications and a summation. This can be simplified by precomputing conversion weights for each channel. These weights can be interpreted as the i integer values found when setting the RNS system to 1|0|0|..., 0|1|0|... and up to ...0|0|1.

$$W_i = (D_i \times |D_i^{-1}|_i) \bmod D \tag{3.20}$$

Now replacing the precomputed constants in the reverse conversion equation, we have

$$X = \left| \sum_{i=1}^{k} W_i \times x_i \right|_D \tag{3.21}$$

These steps offer small precomputation and simplification, however reverse conversion still introduced significant overhead into RNS-based designs because of the $\log_2(D) \times \log_2(D)$ multiplication repeated nk times for one polynomial. There are efforts to perform this deconverison efficiently [73]. These offer speed improvements but often rely on specific moduli which limits flexibility.

This design uses the RNS conversion before and after the negative wrapped convolution to convert the RNS polynomial to and from integer form. By maintaining RNS throughout the computations, this design avoids performing forward and reverse conversion in the multiplication algorithm. To accomplish this, memory is initialized with RNS values via the RNS conversion unit. After the algorithm has been completed, the values are deconverted using values of W_i that are precomputed in the software version of the design.

3.5 Processing with the Number Theoretic Transform

The core computational instrument in this design of an RNS-based polynomial multiplier is the Number Theoretic Transform (NTT). The NTT is a linear transform function that maps an input integer vector $A \in \mathbb{Z}_M^n$ to another similar vector $Z \in \mathbb{Z}_M^n$ according to the following equation:

$$Z[i] = \sum_{j=0}^{n-1} A[j]\omega_n^{ij} \mod M \tag{3.22}$$

for $i = 0, 1, ...n-1$. The NTT is equivalent to an integer version of the Discrete Fourier Transform (DFT), with the complex exponential basis function $e^{-i2\pi/n}$ being replaced by the integer-valued n-th root of unity ω_n and with values kept in the range $\{0, M-1\}$ via a modular reduction. In number theory, this is equivalent to saying

the DFT is taken over the field of complex numbers while all results of the NTT remains in a ring of integers $\mathbb{Z}_M$. This is beneficial in hardware and cryptography because it removes the need for complex and floating point arithmetic and prevents rounding errors. Setting up the NTT relies on three parameters: input vector length, the modulus, and the n-th root of unity.

To perform the inverse NTT (INTT), only slight variation to the equation needs to be made as seen in Equation 3.23. In the INTT, the powers of the n-th root of unity need to be changed to its modular inverse and the output needs uniformly scaled by the modular inverse of the transform size. This is generally produced by a Hadamard product.

$$A[i] = n^{-1} \sum_{j=0}^{n-1} Z[j] \omega_n^{-ij} \mod M \tag{3.23}$$

for $i = 0, 1, ...n-1$. With these two equations in consideration, the general classical algorithm for the NTT and INTT is given in Algorithm 8. The only changes between the NTT and INTT are in the input data and final Hadamard product with n^{-1}, which are both preset once the NTT parameters are known.

While the classical NTT function serves as a reference for other NTT algorithms, it includes two loops that iterate through the entire input vector and is consequently computed in $\mathcal{O}(n^2)$ time and is not efficient enough for large transform sizes. The classical algorithm needs to be used when the size of transform is composite. To improve upon the classical algorithm, a radix-2 butterfly NTT with RNS arithmetic is adopted in this design. This butterfly structure can be utilized to achieve $\mathcal{O}(n \log n)$ complexity as long as the NTT input length is a power of 2. The algorithm is presented in Algorithm 9.

Algorithm 8: Classical NTT and INTT algorithm

Input: vector $A = \{A_1, A_2..., A_n\} \in \mathbb{Z}_M^n$
Output: vector $Z = \{A_1, A_2, ..., A_n\} \in \mathbb{Z}_M^n$
Data: modulus $M \in \mathbb{Z}_{prime}$, vector length $n \in \mathbb{Z}$, root of unity $\omega = \omega_n$ if NTT or $\omega = \omega_n^{-1}$ for INTT, arithmetic performed mod M

```
for i = 0; i < n; i = i + 1 do
    sum = 0 ;
    for j = 0; j < n; j = j + 1 do
        sum+ = ω^ij × A[j]
    Z[i] = sum ;
Z = bitreverse(Z) ;
if INTT then
    Z = n^-1 Z
```

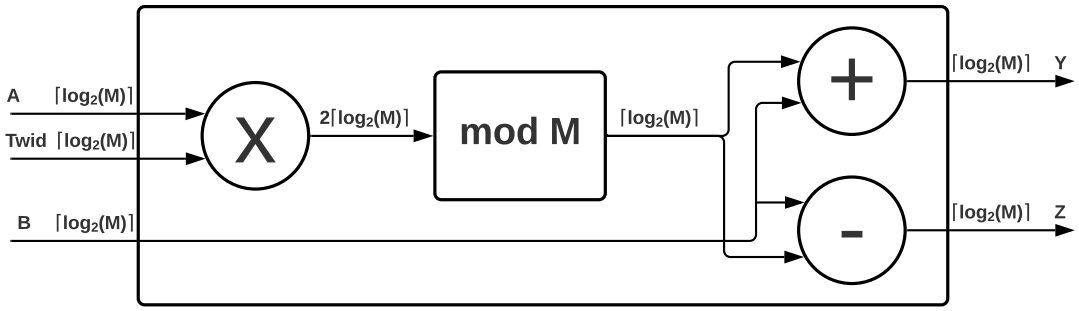

Figure 6. NTT butterfly for a standard NTT

There are three core loops in the butterfly structure which are the stage, step size, and block number. The NTT is broken into $log(n)$ stages. In the first stage, the data is broken into groups of 2 and the butterfly operation is performed on each group (on data that is spread by 1). In the next stage, the group size and spread is doubled, grouping every 4 values of data with the butterfly now being performed twice per group (on the data that's spread by 2). By repeating this with group sizes of increasing powers of two, by the log(n)-th stage there will be a single group of size n that performs the butterfly on elements that are $n/2$ apart and will produce the

Algorithm 9: Radix 2 butterfly NTT and INTT algorithm

Input: vector $A = \{A_1, A_2..., A_n\}$ where $A_i = \{a_1, a_2, ...a_k\} \in \mathbb{Z}^k_{m_i}$ in RNS
Data: modulus $M \in \mathbb{Z}_{prime}$, vector length $n \in 2^{\mathbb{Z}}$,root of unity $\omega = \omega_n$ if NTT or $\omega = \omega_n^{-1}$ for INTT, arithmetic performed mod M

```
1  if INTT then
2  |  ω = ω_n^{-1}
3  else
4  |  ω = ω_n
5  A = bitReverse(A) ;                          ▷ via index reversal algorithm
6  size = 2
7  count = 0
8  while size <= n do
9  |  halfsize = size/2
10 |  tablestep = n/size
11 |  for i = 0; i < n; i+ = size do
12 |  |  k = 0
13 |  |  for start = i; start < i + halfsize; start + + do
14 |  |  |  end = start + halfsize
15 |  |  |  L = A[start]
16 |  |  |  R = (A[end] × powtable[k]) mod M ;        ▷ via RNS ModMult
17 |  |  |  A[start] = (L + R) mod M ;               ▷ via RNS ModAdd
18 |  |  |  A[end] = (L + M − R) mod M ;             ▷ via RNS ModSub
19 |  |  |  k+ = tablestep
20 |  size = size × 2
21 Z = A
22 if INTT then
23 |  Z = n^{-1}Z ;                                 ▷ via RNS Hadamard
```

Output: $Z = NTT(A)$ where $Z_i = \{z_1, z_2, ...z_k\} \in \mathbb{Z}^k_{m_i}$ in RNS

final NTT result. Summing up the number of butterfly operations on the diagram, it is clear that from a consistent $n/2$ butterflies per stage and $\log n$ stages with a butterfly that contains a sequential multiplication and addition, the final complexity is $\mathcal{O}(n \log n)$.

The radix-2 butterfly NTT is used in this design to perform the NTT and INTT with butterfly units. It is combined with RNS arithmetic in its addition, subtraction,

and modular multiplication blocks and it is used inside the design's RNS negative wrapped convolution to produce a full RNS-based polynomial multiplication.

3.5.0.1 Polynomial Multiplication with NTT

The NTT-based polynomial multiplication algorithm in this design makes use of two NTTs and one INTT to multiply polynomials in $O(n \log n)$ time. Given two polynomials A and B, both polynomials are first zero padded and then the NTT of each are taken and the corresponding point-by-point terms are multiplied. The INTT of the resulting vector is then taken, producing the polynomial multiplication result and skipping the need for a longer convolutional multiplication.

$$A \circledast B = INTT(NTT(A) \times NTT(B)) \tag{3.24}$$

The NTT multiplication approach is comparable to the FFT multiplication method, however standard polynomial multiplication algorithms in lattice-based schemes use the Number theoretic transform (NTT) instead of the FFT. In [61], cryptography and FHE multiplication methods for large integers are tested on an FPGA platform and it was found that an NTT-Karatsuba-Schoolbook variant is best for large integers. Modular multiplication and modular exponentiation tests in hardware are performed in [46] show similar NTT success for large bit operands.

The NTT based multiplication method is the most popular hardware implementation due to it's minimal computational complexity. In 2012, the first full hardware implementation of a LWE-based cryptosystem was designed and their paper includes efficient modules for performing both the NTT and discrete Gaussian sampling in hardware [1]. To speed up polynomial reduction, the authors choose a cyclotomic polynomial in the form $f(x) = x^n + 1$ with n being a power of 2. Their hardware

improvements are able to be extended to numerous encryption and signature schemes. They were able to obtain speed gains of 200 and 70 times for encryption and decryption using a 0.5 KB secret key and 1 KB public key polynomials. The authors argue that they chose the NTT instead of other listed polynomial multiplication methods because the algorithm was able to be parallelized and could make use of the form of polynomial, but they also claim that other multiplication algorithms may be faster if given practical parameters.

The proposed design applies the above procedure and improves upon it by applying the Negative Wrapped Convolution as in [21]. While NTT multiplication is highly efficient, the doubling of input size as a result of zero padding requires space. Negative Wrapped Convolution allows the initial polynomial size to remain at their original size leading to less memory access and faster execution. This is discussed in detail in Section 3.6.1.

3.5.1 System NTT Initialization

The proposed design performs NTT initialization in the software implementation to validate and find input parameters. The NTT relies on input parameters of the polynomial length n and the minimum operating modulus M_{min}. The length n is generally a power of two for lattice-based schemes and is necessarily so for a radix-2 butterfly structure. The operating modulus has several criteria it must follow and is discussed in Section 3.5.2. The n-th root of unity also has criteria and needs precomputed as discussed in Section 3.5.3. Both parameters remain constant for any one system but are allowed to vary based on the cryptographic scheme being used.

3.5.2 Modulus Validation

At the NTT's instantiation in this design's software implementation, the minimum modulus is validated to ensure it meets the conditions to be the working modulus. To be valid, the working modulus M must be prime, larger than M_{min}, and one higher than some integer multiple k of the vector length n:

$$M \geq M_{min} \tag{3.25}$$

$$\text{factorize}(M) = \{1, M\} \tag{3.26}$$

$$M = kn + 1 \tag{3.27}$$

For example, attempting to use a minimum modulus of $M_{min} = 23$ and a vector length of $n = 8$, a new working modulus will need to be generated to satisfy all conditions. Testing for primes that are near multiples of the vector length, the next usable prime modulus M = 41 must be used as $M = kn + 1$ for $k = 5$.

The proposed design performs modulus validation in the precomputation stage of the software implementation of the RNS-based NTT polynomial multiplier. The program is provided with a minimum modulus and via the validation conditions, either finds a new valid modulus or accepts the given minimum modulus as working modulus. This is performed at parameter initialization and before any polynomial multiplication is performed.

3.5.3 Finding n-th Root of Unity

After verifying the modulus conditions, the proposed design's modular n-th root of unity ω_n is found in the software implementation using the polynomial length and

working modulus. This is also when the n-th root's modular square root $\phi^2 = \omega_n$ and their corresponding modular inverses with respect to the working modulus are found and saved in a table of the NTT's twiddle factors of increasing powers of ω_n and ϕ. The n-th root of unity's distinctive quality is that when raised to powers of $\{1, 2...n\}$, it wraps through n values in $\mathbb{Z}_M$ and has the result $\omega_n^n = 1 \bmod M$ at the last index. This can be paralleled to the DFT, where its n-th root of unity $e^{-i2\pi m/n}$ moves through the unit circle and wraps to 1 upon reaching $m = n$. This can also be thought of as a value where:

$$\omega_n^m = 1 \pmod{M} \qquad \text{for } m = n$$

$$\omega_n^m \neq 1 \pmod{M} \qquad \text{for } 0 \leq m < n$$

For an n-th root of unity to be used in the NTT, it must also be a primitive root. A primitive root r mod M is an integer where every number a that is relatively prime to M has a solution g in the equation $r^g = a \pmod{M}$. In the case of a prime M and in the case of the NTT, this means that all values in $\mathbb{Z}_M$ will be hit with some power of r (despite possibly being hit out of order). Finding a primitive root is not simple, however it can be shown that all prime numbers can be guaranteed to have an existing primitive root. A primitive root r of the ring $\mathbb{Z}_M$ will be an n-th root and valid candidate for ω_n as long as $M = bn + 1$ for some integer b. This is by virtue of the equation:

$$a^{M-1} = 1 \pmod{M} \tag{3.28}$$

which holds for all integers a because M is assumed to be prime and $M - 1$ is consequently a multiple of 2. Setting $M - 1 = n$, we have

$$a^n = 1 \pmod{M} \tag{3.29}$$

and we automatically see that a will consequently be both an n-th root and a primitive root on the condition that the prime modulus M (minus one) is a multiple of the transform size n. As an example of a primitive n-th root of unity, assume values of $n = 4$ and $M = 5$. With a guess of $\omega_n = 2$, powers of ω_n cover all relative primes to M as

$$2^0 \equiv 1, 2^1 \equiv 2, 2^3 \equiv 3, 2^2 \equiv 4 \pmod 5 \tag{3.30}$$

and ω_n is therefore a primitive root. As M and n were chosen to satisfy Equation 3.28, it is also the case that

$$\omega_n^n \equiv 2^4 \equiv 1 \pmod 5 \tag{3.31}$$

so ω_n is therefore a valid primitive n-th root of unity. The n-th root can be found by using a generator integer Θ. The generator must satisfy two conditions that purely rely on the working modulus:

$$\Theta^x \equiv 1 \pmod M$$
$$\Theta^{x/y} \not\equiv 1 \pmod M$$

Here $x = M - 1$ and $y \in \{\text{prime factors of } x\}$. A usable generator is solved by guessing at values of Θ until its modular exponentiation is the only power congruent to one and its factors are not congruent. The n-th root of unity is then found simply with the equation $\omega_n = \Theta^{\frac{x}{n}}$.

The implementation of the above n-th root of unity solver is found in the software precomputation component of this thesis. The resulting value for ω_n is used for the remaining RNS-based NTT polynomial multiplication in both the software and hardware implementation.

3.5.3.1 Example: Solving for n-th Root of Unity

As an example in solving for ω_n, assume the following NTT parameters of vector length $n = 4$ and modulus $M = 13$.

First assign values to x and y:

$$x = M - 1 = 12$$

$$y = \text{factor}(x) = \{2, 6\}$$

Guess values of Θ. Here we try $\Theta = 2$ on the two conditions involving x and y:

$$\Theta^x = 2^{12} = 4096 \equiv 1 \pmod{13}$$

$$\Theta^{y_1} = 2^2 = 4 \not\equiv 1 \pmod{13}$$

$$\Theta^{y_2} = 2^6 = 64 \not\equiv 1 \pmod{13}$$

These conditions are true so use equation for n-th root to solve:

$$\omega_n = \Theta^{\frac{x}{n}}$$

$$= 2^{12/4} = 9$$

For more information on the NTT or on its parameters, see [74] and [75].

3.6 Outer Layer Control

The proposed design combines the mentioned modular functions, RNS, NTT, and system initialization in an outer layer algorithm to complete the RNS-based NTT polynomial multiplier. A negative wrapped convolution block is used to multiply the two input polynomials. RNS conversion is responsible for converting values to and from

RNS before and after negative wrapped convolution is performed. Lastly, a memory unit is in charge of storing the input polynomials, negative wrapped convolution twiddle factors, and RNS conversion weights. The negative wrapped convolution is explained here, while the RNS conversion is discussed in Section 3.4.6 and memory is discussed in Section 4.8.

3.6.1 Negative Wrapped Convolution Procedure

Negative wrapped convolution [13] (NWC) is used in this thesis as the outer algorithm to control multiplying two polynomials A and B. Such an approach is beneficial because it prevents NTT input polynomials from needing zero padding. The equation for the NWC is given in Equation 3.32.

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \tag{3.32}$$

In the NWC, the NTT is called three times with Hadamard products being performed on intermediate results and scaling factors. A designer can choose whether to use a single NTT unit or two NTT units to perform the convolution. If two units are chosen, the two forward NTTs found in the NWC can be performed in parallel and will significantly cut back timing at the expense of doubling the space taken on the FPGA and having one unit underutilized for the inverse NTT.

Choosing one unit means all NTTs will need to be sequential. For the sake of hardware space and easier comparison/testing, the hardware implementation of the polynomial multiplication is an individual NTT unit. The algorithm for the NWC is given in Algorithm 10.

The NWC of two vectors can be shown to be equivalent to a polynomial multi-

Algorithm 10: NTT polynomial multiplication via negative wrapped convolution

Input: vectors $A = \{a_1, a_2..., a_n\}, B = \{b_1, b_2..., b_n\}$

Data: modulus M, vector length n. Precomputed: $\phi = \omega_n^2$ and ϕ^{-1} with respect to M

for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $\bar{a}_i = a_i \phi^i \bmod M$
    $\bar{b}_i = b_i \phi^i \bmod M$
$\bar{A} = NTT(\bar{a})$
$\bar{B} = NTT(\bar{b})$
for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $\bar{Z}_i = \bar{A}\bar{B} \bmod M$
$\bar{z} = INTT(\bar{Z})$
for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $Z = \bar{z}_i \phi^{-i} \bmod M$

Output: vector $Z = \{z_1, z_2..., z_n\} = A \times B \bmod (x^n + 1)$

plication in the $x^n - 1$ ring, meaning it gets modular reduction for free. It requires two NTT's and one inverse NTT, with the input and output vectors scaled by powers of $\phi = \omega_n^2$ and a Hadamard product between the forward and reverse NTT's. As in this design, Pöppelmann looked to use the NWC in [7] to minimize hardware costs on a lattice accelerator. Using the NWC saves many operations compared to the zero padding NTT multiplication. By automatically getting reduction in the algorithm, it saves a final reduction by $\bmod(x^n + 1)$ that's needed in the zero padded version. As seen in Table 3, the number addition and subtractions is more than halved along with great reductions in the number of multiplies and general reductions by M.

Operation	Zero Padding	Negative wrapped convolution
Add/subtract	$6n \log_2 2n$	$3n \log_2 n$
Multiply	$32n \log_2 2n + 4n$	$(3/2)n \log_2 n + 5n$
$\bmod M$	$9n \log_2 2n + 4n$	$(9/2)n \log_2 n + 5n$

Table 3. Polynomial multiplication operation count comparison between zero padding and negative wrapped convolution with data from [13].

3.7 Proposed Design

The full RNS-based NTT polynomial multiplier design is presented in Figure 7 with the corresponding algorithm in Algorithm 11. Starting with a binary represented polynomial loaded into memory, the polynomial is first converted to RNS in the forward converter in n cycles. This converts the polynomial into RNS representation in Base 1, Base 2, and the redundant modulus. After this conversion, the RNS Negative Wrapped Convolution begins and performs the combination of Hadamard products and RNS-based NTTs. This results in the full polynomial multiplication in RNS form. this can then be recovered from RNS form with the reverse converter and saved back into memory. The flow diagram highlights the critical components in the software design. The hardware implementation would handle the same structure but run all computations through a single NTT unit, Hadamard unit, and conversion unit.

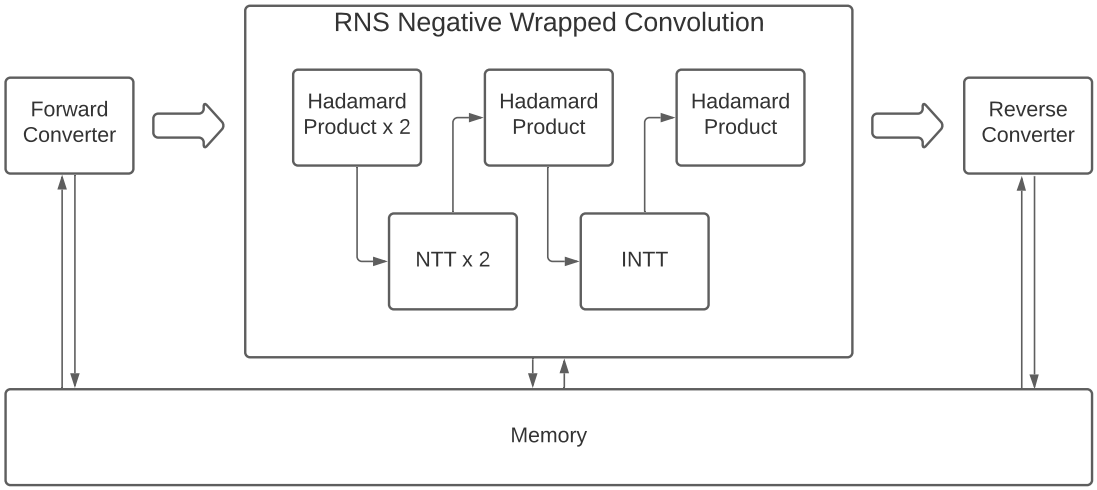

Figure 7. Full RNS-based NTT polynomial multiplication procedure via negative wrapped convolution.

For FHE schemes, the proposed design targets polynomial length $n = 4096$,

Algorithm 11: RNS-based NTT polynomial multiplication via negative wrapped convolution

Input: vectors $A = \{a_1, a_2..., a_n\}, B = \{b_1, b_2..., b_n\}$
Data: modulus M, vector length n. Precomputed: $\phi = \omega_n^2$ and ϕ^{-1} with respect to M, Bajard and Shenoy constants

A = RNSforwardConverter(A)
for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $\bar{a}_i = a_i \phi^i \bmod M$; ▷ via RNS Hadamard
    $\bar{b}_i = b_i \phi^i \bmod M$; ▷ via RNS Hadamard
$\bar{A} = NTT(\bar{a})$; ▷ via RNS NTT
$\bar{B} = NTT(\bar{b})$; ▷ via RNS NTT
for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $\bar{Z}_i = \bar{A}\bar{B} \bmod M$; ▷ via RNS Hadamard
$\bar{z} = INTT(\bar{Z})$; ▷ via RNS INTT
for $(i = 0;\ i < n;\ i = i + 1)$ **do**
    $Z = \bar{z}_i \phi^{-i} \bmod M$; ▷ via RNS Hadamard
Z = RNSreverseConverter(Z)

Output: vector $Z = \{z_1, z_2..., z_n\} = A \times B \bmod (x^n + 1)$

modulus bitwidth $w = 128$, RNS channels $k = 4$, and channel width $w_{ch} = 32$. For lattice schemes, this can be lowered to polynomial length $n = 1024$, modulus bitwidth $w = 32$, RNS channels $k = 4$, and channel width $w_{ch} = 8$. These are given in Table 4.

Scheme	Polynomial Length	Coefficient size	RNS channels	Channel width
Lattice	1024 coefficients	32 bits	4	8 bits
FHE	4096 coefficients	128 bits	4	32 bits

Table 4. System parameters for the proposed RNS-based NTT polynomial multiplier.

3.7.1 Computational Complexity

Four Hadamard products takes a total of $4nk$ multiplications. Three NTTs take $3n/2\log(n)$ cycles to complete so the butterfly addition and subtraction will require

a total of $2k \times 3n/2\log(n)$ operations. The RNS Montgomery Multiplication takes five k channel multiplications, two k channel additions/subtractions, and $2k^2 + 5k + 1$ multiplications and $k^2 + 3k + 1$ additions for the Bajard and Shenoy extensions respectively. The INTT requires an extra nk channel Hadamard product. The total multiplications in the RNS-based NTT polynomial multiplication is therefore $3n/2\log(n)(2k^2 + 10k + 1) + 4nk$ multiplications with the total additions/subtractions being $3n/2\log(n)(k^2+7k+1)$. For these estimates, keep in mind that the RNS channel width k is generally very small while the polynomial length n is large ($k = 4, n = 4096$ in this design). Also keep in mind that RNS operations are ran in parallel and the RNS butterfly elements are pipelined and ran simultaneously, making the target number of cycles to complete an NTT closer to $\sum_{i=0}^{m} 2^i$ for an $m = n_{stages} = \log_2(n)$ sequential butterfly system as seen in Equation 4.1 in Section 4.2.1.

With this established RNS-based NTT polynomial multiplier design and targeted parameters, Chapter 4 will discuss the hardware implementation of the NTT unit and the FGPA implementation results.

Chapter 4

HARDWARE IMPLEMENTATION

Hardware implementation of the RNS based NTT polynomial multiplier is important as commercial processors can not support the high throughput that is required in lattice-based cryptography and FHE schemes. Systems in cloud computing and Internet-of-Things (IoT) require an optimized hardware approach that offers adjustable, low latency, energy efficient, and tamper resistant security. Cloud computing requires ASIC-like speed and power efficiency to be achieved on large cloud computers to encrypt and decrypt for large amounts of data at a reasonable cost. IoT devices, on the other hand, generally operate on scaled-back hardware and consequentially have restrictions on energy and cost that make achieving speed with larger key sizes difficult. In this chapter, details of hardware implementation of the RNS-based polynomial multiplication is explained. The hardware building blocks and integrated design are implemented on a Xilinx Artix-7 (xc7a200tlffg1156-2L) for estimated timing, power, and sizing. The device contains 1156 IO, 500 IOB, 134600 LUT, 269200 FF, 365 BRAM, 740 DSP.

4.1 Choosing Hardware-friendly Moduli

There are multiple valid primes and co-primes to use when determining RNS moduli and the choice of them varies the core channel hardware in an RNS system. Choosing moduli based on bit-width [68] is an option that allows equally distributed channel widths in the RNS. This is more flexible and leads to easier design of RNS

cores. Choosing moduli on either side of powers of 2 (2^w, $2^w + 1$, and $2^w - 1$) is an option suggested in papers such as [26]. This has the advantage of nice mathematical properties for hardware reductions and conversion as reductions modulo $\{2^w, 2^w + 1, 2^w - 1\}$ can be realized with bit-shifts and simplified logic operations that are optimized for specific moduli sets ([76], [77], [78], [79], [80], [81]). This thesis uses a moduli set of 9 channels with moduli of size w_{ch}. The original RNS base (Base 1) uses the first 4 of these moduli, with the extended base (Base 2) and redundant modulus that are needed for the RNS Montgomery Multiplication using the remaining 4 and 1 moduli respectively.

$$\text{Chosen RNS moduli} = \begin{cases} m_1\text{: } 4294967291 & \text{base 1} \\ m_2\text{: } 4294967279 & \text{base 1} \\ m_3\text{: } 4294967231 & \text{base 1} \\ m_4\text{: } 4294967197 & \text{base 1} \\ m_5\text{: } 4294967189 & \text{base 2} \\ m_6\text{: } 4294967161 & \text{base 2} \\ m_7\text{: } 4294967143 & \text{base 2} \\ m_8\text{: } 4294967111 & \text{base 2} \\ m_r\text{: } 4294967087 & \text{redundant modulus} \end{cases}$$

These moduli are chosen in the form $2^w - k$ as to select the bit-width of each channel by setting $w = w_{ch}$ and to adjust to the nearest prime with an offset k. An advantage of taking these primes is that they offer the largest dynamic range by sitting close to the max binary value of an w-bit channel while not requiring a channel of $(w + 1)$-bits. They also allow a design to include a large number of moduli if needed, as the set $\{2^w, 2^w + 1, \text{and } 2^w - 1\}$ only offers speed advantages when all parallel

channels achieve the optimizations of a 2^w bit-shift. Primes with this structure can either be generated for low n or found at [82] where w, k for less-than-power-of-two primes of < 400 bits are listed.

4.2 RNS-based NTT unit

The RNS-based NTT unit is based on the RNS-based NTT described in Section 3.5. It contains $m = 4$ daisy chained butterfly units, operating in parallel with data loaded serially at each clock cycle. The unit additionally has twiddle factor memory routing factors to each unit and an intermediate buffer to capture the intermediate NTT result after every four stages. Data is stored in both RNS bases because the RNS Montgomery Multiplication requires inputs in both bases. This results in a total datapath width of $w_D = 9w_{ch}$ for all arithmetic and memory outside of the modmult_RNS unit. Connected to the first input and output of each butterfly unit is a FIFO which stores varying block sizes of the single stream of input values. The use of FIFO's accomplishes the three loop functionality of the radix-2 algorithm while only loading one value into the butterfly at a time. When the transform size is larger than m, multiple passes through the butterfly network are needed and the FIFO lengths are adjusted to the next power of 2 lower until the computation finishes. This structure benefits from a decimation in frequency approach as it allows the first FIFO to start with a size $n/2$.

Each butterfly unit is responsible for one stage of the NTT by adjusting the FIFO length and controlling input twiddle factors. The hardware architecture in Figure 8 can calculate the NTT in sets of 4 stages. If the NTT length is not a multiple of 4, remaining data can be bypassed at the cost of several clock cycles. For any number of

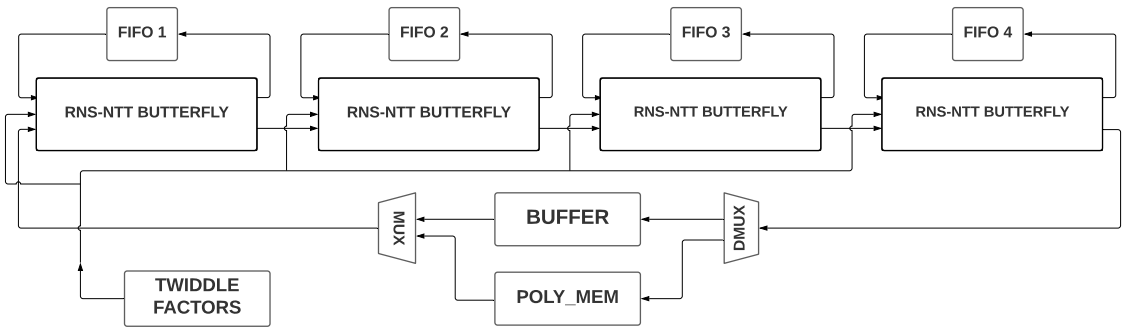

Figure 8. RNS-based NTT unit. Contains four chained butterfly units with bypass FIFOs and polynomial buffer to hold intermediate NTT result.

stages over 4, the buffer holds all data until the butterflies have completed processing for that set. This prevents errors due to data being bypassed too early. The size of the buffer is equal to the length of the initial polynomial. The polynomial memory block can not be used as a buffer because bypasses allow the fourth and final butterfly to begin calculation before the first butterfly ends calculation and values may be overwritten early.

4.2.1 FIFO and Buffer Sizes

Since the NTT butterfly computation units often have to be mapped to a small number of hardware NTT butterfly units, computation using a variable number of sequentially chained butterfly units and changing FIFO sizes should be understood. Every RNS-NTT butterfly has a corresponding FIFO of size $\ell_i = 2^i$ for some stage $0 \leq i \leq n_{stages} - 1$, where the number of stages in the size n RNS-based NTT is $n_{stages} = \log_2(n)$. The sequential butterfly operation performs three stages until input data is no longer available. These are the bypass, computing data, and final bypass and take ℓ_i cycles for each stage each time they're performed. The butterflies use the

first ℓ_i cycles to fill all of the FIFO with half of the input vector. This is followed by ℓ_i cycles to calculate the butterfly between all of the FIFO data and the second half of the input data, loading Y_{RNS} into the FIFO again and Z_{RNS} into the next butterfly. This is followed by a final ℓ_i cycles to bypass the saved computed data to the next butterfly for processing. In this cycle, input data is either bypassed to the FIFO again if the previous butterfly still has data to pass to it or the butterfly finishes operating.

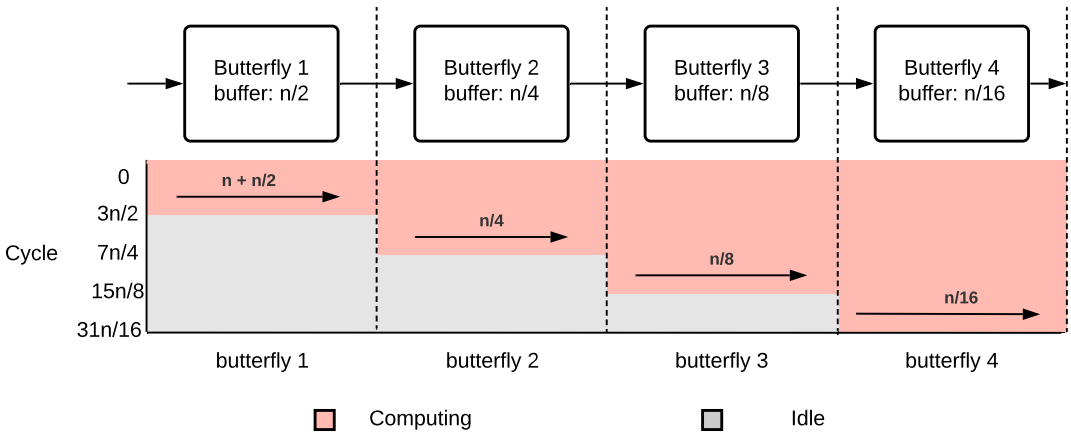

Figure 9. Timing diagram for a sequential 4-butterfly NTT unit. Each butterfly performs a bypass-compute-bypass routine taking ℓ_i cycles for each of the three steps. When a butterfly is finished, the following butterfly has ℓ_i cycles to complete before finishing.

If the FPGA is very large, a size n NTT that uses $m = n_{stages}$ butterflies to complete the NTT in one pass can be housed. In that case, no extra buffer would be required and returning values can be sent directly to the polynomial memory. The total number of cycles taken by the sequential RNS-NTT with $m = n_{stages}$ butterflies takes can be found by looking at the clock delay between the first value loaded into the first FIFO and the final FIFO bypassing the final value. It can be seen that when the first sequential butterfly finishes its last bypass-calculate-bypass cycles, the next

butterfly only has its last ℓ_i bypass cycles to finish before similarly being finished. Therefore, the total cycles it takes for an $m = n_{stages} = \log_2(n)$ sequential butterfly system to complete an n sized NTT with buffer sizes ℓ_i is given as

$$n + \sum_{i=0}^{m-1} \ell_i = \sum_{i=0}^{m} 2^i \tag{4.1}$$

For a transform size of $n = 4096$, this equals 8191 cycles beating the standard processor complexity of $n\log(n) = 49152$. This experiences some extra cycles compared to the estimated $n/m\log(n) = 4096$ cycles for m parallel butterfly units that aren't serially chained due to the bypassing of FIFO values.

m	ℓ_a, ℓ_b	First Storage Cycle	First FIFO Availability	Required Buffer size
1	$\ell, \ell/2$	ℓ	$2\ell + \ell/2$	$\ell + \ell/2$
2	$\ell, \ell/4$	$\ell + \ell/2$	$2\ell + 3\ell/4$	$\ell + \ell/4$
3	$\ell, \ell/8$	$\ell + \ell/2 + \ell/4$	$2\ell + 7\ell/8$	$\ell + \ell/8$
...	...	...	...	...
m	$\ell, \ell/2^m$	$c = \sum_{i=1}^{m} 1/(2^{i-1})\ell$	$3\ell - c/2$	$3\ell - 3c/2$

Table 5. FIFO buffer sizes for large NTT systems ($m < n_{stages}$) where a butterfly needs reused. Found for m butterflies, first and second pass FIFO lengths ℓ_a, ℓ_b, the cycle count when buffer storage is needed, cycle count where the FIFO becomes available again, and the corresponding buffer size.

An m butterfly system where $m < n_{stages}$ requires a buffer to store data before the first FIFO can begin calculating. The generalization for buffer size can be seen in Table 5. The first storage requirement comes when the last butterfly finishes its first bypass and starts computing. This occurs after data has traversed through each preceding butterfly via their first length ℓ_i bypass cycles. The FIFO is available for reuse after it has space for data and will provide it with no more delay than ℓ_b, the second FIFO size. This means its available for loading after it has spent $\ell_a - \ell_b$ cycles of its last bypass, hence the second value in the First FIFO available column. The

buffer size is then the difference in clock cycles between the first storage requirement and first FIFO availability. The generalized buffer size can be given by the equation

$$B = 3\ell - 3/2 \sum_{i=1}^{m} 1/(2^{i-1})\ell \tag{4.2}$$

$$= 3/2\ell \times (2 - \sum_{i=1}^{m} 1/2^{i-1}) \tag{4.3}$$

where $\ell = n/2$ is the length of the first FIFO and m butterflies are used in an NTT system where $m < n_{stages}$. In this design, FHE parameters $\ell = 2048$ and $m = 4$ suggesting an optimal buffer size of

$$B = \frac{3}{16}\ell = 384 \tag{4.4}$$

The total FHE buffer space can then be found using the RNS channel width w_{ch} and channel count k to be

$$B \times w_{ch} \times k = (384)(32)(9) = 108\text{Kb} \tag{4.5}$$

Summing the largest butterfly FIFOs of sizes 1/2, 1/4, 1/8, and 1/16 as ratios of n, the total required FIFO space is found to be

$$(\sum sizes) \times n \times w_{ch} \times k = (.9375)(4096)(32)(9) = 1.055\text{Mb} \tag{4.6}$$

For lattice parameters we have $\ell = 512$ and $m = 4$ suggesting an optimal buffer size of

$$B = \frac{3}{16}\ell = 96 \tag{4.7}$$

The total lattice buffer space required is then

$$B \times w_{ch} \times k = (96)(8)(9) = 6.75\text{Kb} \tag{4.8}$$

Summing the FIFOs and applying parameters from lattice schemes sets the required FIFO space to be

$$(\sum sizes) \times n \times w_{ch} \times k = (.9375)(1024)(8)(9) = 67.5\text{Kb} \tag{4.9}$$

4.3 RNS Butterfly Unit

The RNS-based NTT butterfly unit is based on the radix-2 NTT butterfly unit in Section 3.5 and shown in Figure 10. It holds the modular multiplication unit realized with the RNS Montgomery Multiplication and the butterfly addition and subtraction units realized with RNS arithmetic. Details of the RNS Montgomery Multiplication unit are included in Section 4.4.

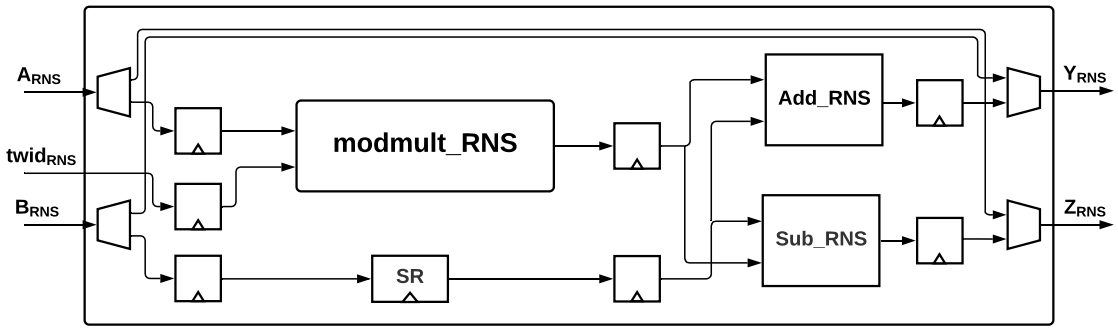

Figure 10. NTT butterfly unit. Contains the modmult_RNS unit, Add_RNS unit, Sub_RNS unit, intermediate registers, and the bypass line.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF	DSP
Fully pipelined	9	32	21.257	47.043	0.639	13239	4010	330

Table 6. RNS butterfly implementation results for fully pipelined unit for k RNS channels with channel bitwidth w_{ch}, propagation delay d, operating frequency f, power P, and LUT/FF/DSP FPGA resources. Implemented on Artix-7 (xc7a200tlffg1156-2L)

The RNS addition and RNS subtraction subtraction units is shown in Figure 11. These contain the k parallel channel operations that perform an RNS addition or subtraction. This design uses channel counts of $k = \{4, 5, 9\}$ to support arithmetic in base 1, base 2 with m_r, and all bases. The addition units are realized as w_{ch}-bit

adders, where w_{ch} is again the bitwidth of each RNS channel. In the case of the chosen design, $w_{ch} = 32$ bits. The result of the channel additions produce a $w_{ch} + 1$-bit result which is then reduced back to w-bit by the constant modulus blocks. The reduction units for each moduli can be individually optimized according to methods in Section 4.7. The subtraction unit first uses channel additions to add the channel moduli to input A_{RNS}. This ensures the subtraction is positive and will result in an accurate result for C_{RNS} as the channel is reduced by the channel moduli before a final result is produced.

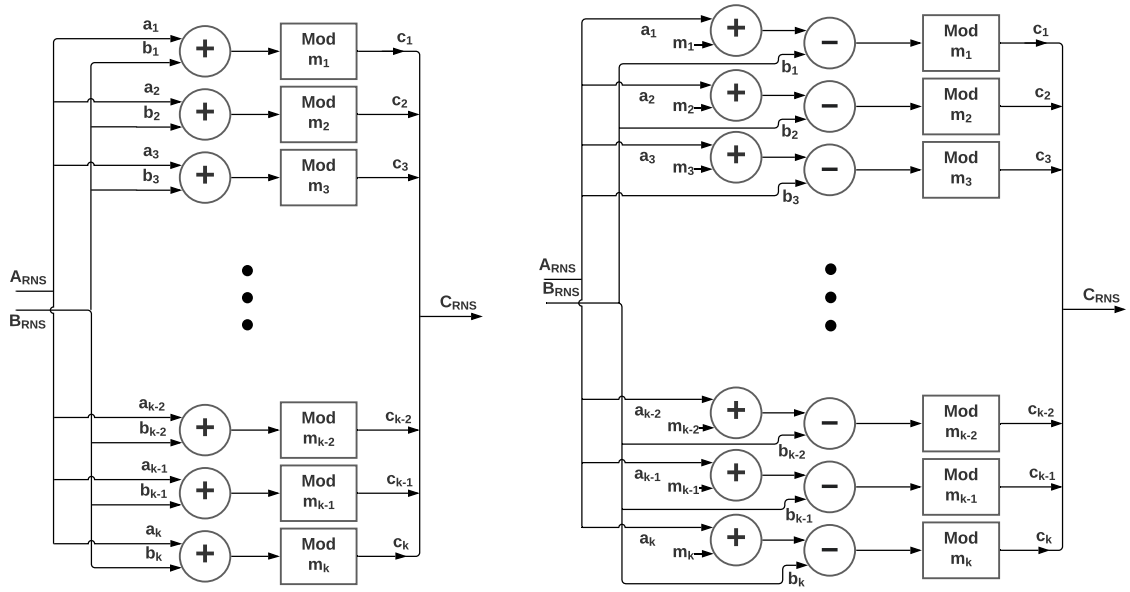

Figure 11. Hardware implementation of Add_RNS and Sub_RNS units. Contains k modular arithmetic channels performing independent 32-bit arithmetic.

The butterfly unit includes intermediate registers that are placed in between critical elements to shorten the critical path through the butterfly unit. The add_RNS and sub_RNS blocks are relatively fast compared to the modmult_RNS which contains the highly expensive base extensions and several addition, subtraction, and multiplication

blocks. To reduce this critical path through the modmult_RNS block, another set of t intermediate registers needs to be placed inside the module. These internal registers need accounted for on the datapaths that run parallel to the modmult_RNS block to ensure data arrives at the add_RNS and sub_RNS units at the same clock cycle. A parallel delay to the modmult_RNS block can be realized as either a shift register or a memory block with additional control such as a FIFO. Here it is represented as a shift register with t delays to match the t registers found in modmult_RNS.

Unique to the hardware implementation, the top of the NTT butterfly unit runs a parallel bypass line that directly connect inputs A_{RNS} and B_{RNS} to outputs Z_{RNS} and Y_{RNS} respectively. The external FIFO and next sequential butterfly then alternate between receiving bypassed data and calculated data. When bypass occurs, the internal clock that routes to internal computational elements can also be disabled. This limits data from unnecessarily moving through the butterfly and ideally saves power. As discussed in detail in Section 4.2.1, this bypass is ideally controlled by an external finite state machine.

4.4 RNS Montgomery Multiplication Unit

The RNS Montgomery Multiplication unit is the most complex block present in each RNS-based NTT butterfly unit. It is based on the algorithm level design in Section 3.4.3. The unit consists of four RNS multiplications, one RNS addition, one RNS subtraction, and two base extensions. The module takes inputs A and B represented in both RNS bases, performs the necessary arithmetic with precomputed constants, converts an intermediate value Q to and from and produces the Montgomery

result $Z = ABD^{-1} \bmod M$ where D is the RNS Base 1 dynamic range and M is the system modulus.

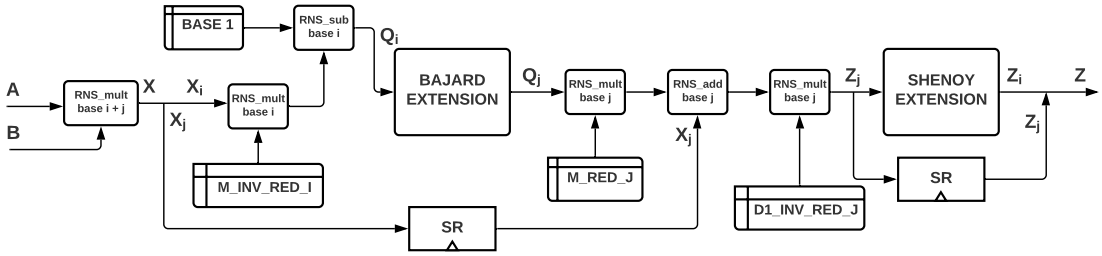

Figure 12. RNS Montgomery reduction unit. Obtains $Z = ABD^{-1} \bmod M$ via the Bajard extension, Shenoy extension, and intermediate arithmetic utilizing precomputable constants. Inputs A,B and output Z are represented in both bases.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF	DSP
non-pipelined	9	32	173.968	5.748	0.576	10829	288	328
Fully pipelined	9	32	21.642	46.206	0.627	12094	3488	330

Table 7. Montgomery Multiplication implementation results for a non-pipelined unit and a fully pipelined unit. Implemented on Artix-7 (xc7a200tlffg1156-2L)

The hardware realization uses memory blocks for the constants BASE 1, M_INV_RED_I, M_RED_J, and D1_INV_RED_J. As these are generally single values, they can also be stored in RTL memory as registers. The constant BASE 1 is the set of original moduli and takes $w_{ch} \times k_{base1}$ bits of storage for a total of $32 \times 4 = 128$ bits. M_INV_RED_I, M_RED_J, and D1_INV_RED_J are the precomputed constants that depend on system modulus M and dynamic range D. Constants that are i-labeled are computed for each channel in base 1 and those that are j-labeled are computed for each channel in base 2 and m_r. This results in the i-labeled constant taking $w_{ch} \times k$ bits of storage and the j-labeled constants taking $w_{ch} \times (k + 1)$ bits. The redundant modulus is only included in base 2 calculations

for the final Bajard calculation (Section 3.4.4) and the intermediate Montgomery computation between base extensions (Section 3.4.3). The Shenoy extension keeps the redundant modulus separate and finds the correction factor in parallel to the base 2 calculations (Figure 14). The Montgomery multiplication module requires internal registers between the core functional blocks to minimize the critical path delay. For clarity, these are not shown in Figure 12. Similar to the delays found in the butterfly units, shift registers are placed in parallel datapaths to the two base extensions to account for internal registers in those modules.

4.5 Bajard Base Extension Unit

The Bajard base extension unit (Section 3.4.4) is responsible for converting RNS values from base 1 to base 2 in the RNS Montgomery multiplication unit. It contains an RNS multiplication on input values by a constant D1_I_INV_I and an RNS MAC operation for each outbound channel. The Bajard result is one channel greater than base 1 because the Shenoy extension requires a redundant modulus, which is easily generated by appending an additional MAC unit. If saving space is a necessary consideration, an alternative method when implementing Bajard is to take k cycles to perform the RNS MAC once per outbound channel. However, this results in a pipelining difficulty and is not chosen in an attempt to maintain high levels of pipelining availability.

As seen in Table 8, implementation results for the Bajard unit for non-pipelined, sub-pipelined, and fully pipelined results are presented. The non-pipelined version does not contain any intermediate registers in the Bajard module, as well as in the multipliers or MAC units below. The partially-pipelined version has the lower modules

pipelined only. The fully pipelined version has registers added in between each critical element.

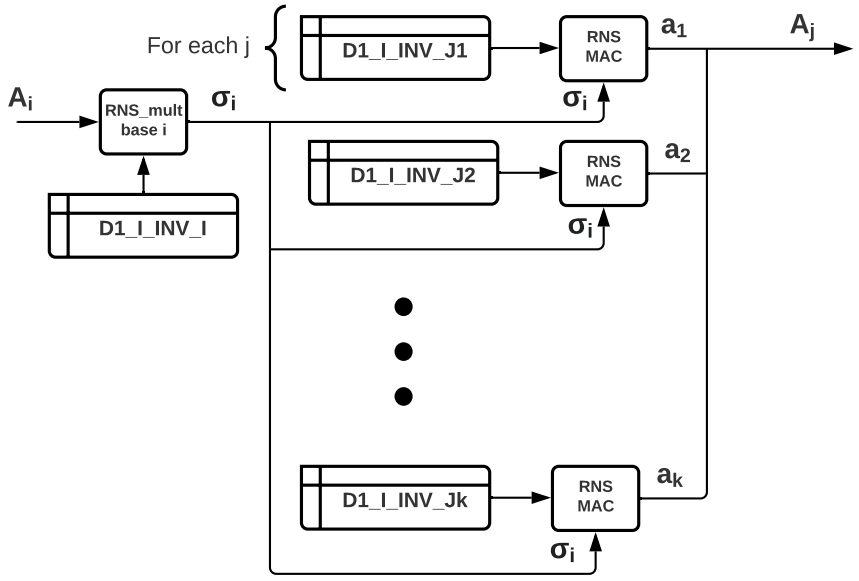

Figure 13. Bajard base extension unit including RNS MAC units for each output channel and precomputed constants D1_I_INV_I and D1_I_INV_J.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF	DSP
Non-pipelined	4/5	32	35.337	28.299	0.257	3599	160	96
Partially-pipelined	4/5	32	20.111	49.724	0.254	3599	160	96
Fully pipelined	4/5	32	14.914	67.051	0.252	3424	800	96

Table 8. Bajard Extension implementation results for a non-pipelined unit, a sub-pipelined unit where only sub-modules are pipelined, and a fully pipelined unit. Implemented on Artix-7 (xc7a200tlffg1156-2L)

4.6 Shenoy Base Extension Unit

The Shenoy base extension (Section 3.4.5) is the second base extension in the RNS Montgomery multiplication responsible for converting RNS values from base 2 to base 1. The hardware unit is presented in Figure 14. The module contains two critical elements; one being the conversion logic executed on k channel RNS arithmetic and the other being a calculation of a correction factor β. The channel β is calculated on is a single channel width. The constant is used at the end of the Shenoy unit and ensures a fully reduced output.

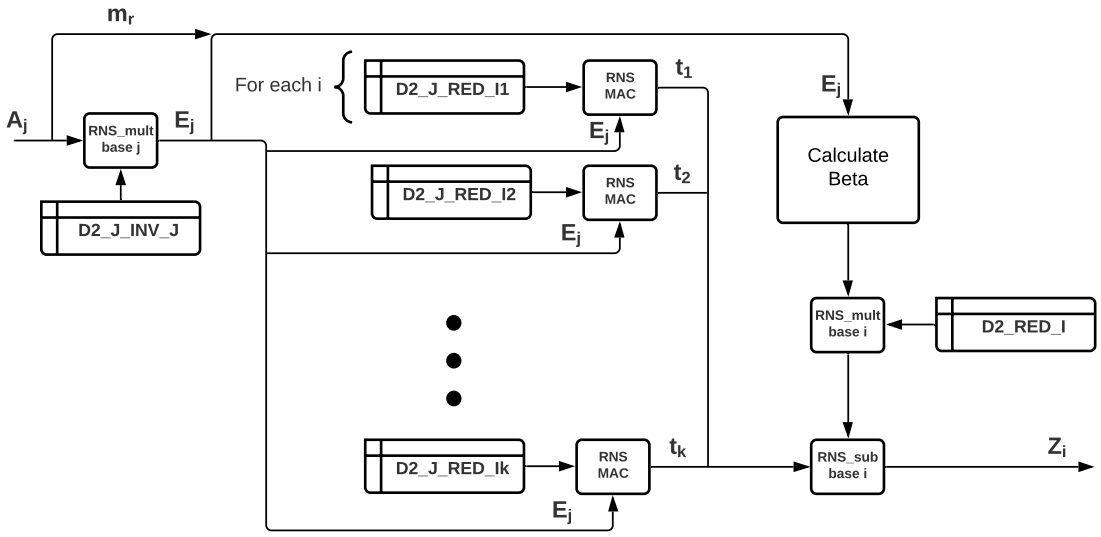

Figure 14. Shenoy base extension unit.

Table 9 presents the implementation results of the Shenoy unit. Like the Bajard unit, it is simulated for a non-pipelined version, sub-pipelined version, and a fully pipelined version. Due to the calculation of β, the sizing in terms of LUT, FF, and DSP are all slightly greater than the Bajard unit. The unit is also the bottleneck in

the RNS Montgomery multiplication unit. The critical path has a delay of 20.8 ns and corresponding frequency of 47.9 MHz.

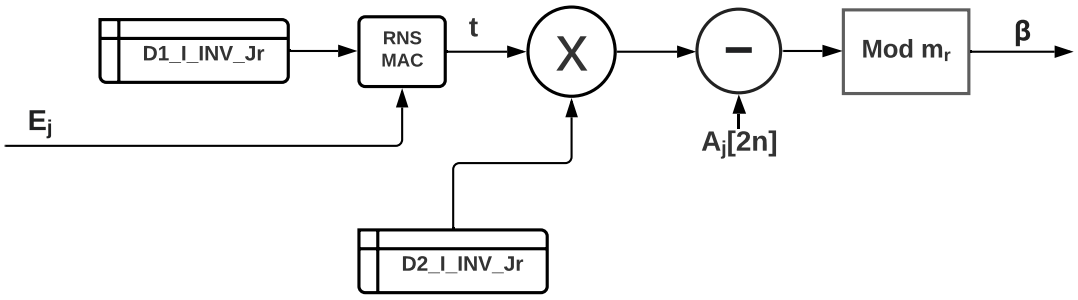

Figure 15. Calculating Beta for Shenoy extension.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF	DSP
Non-pipelined	5/4	32	73.76	13.557	0.305	4735	128	140
Partially-pipelined	5/4	32	27.142	36.843	0.303	4178	768	116
Fully pipelined	5/4	32	20.858	47.943	0.302	4168	1120	116

Table 9. Shenoy Extension implementation results for a non-pipelined unit, a sub-pipelined unit where only sub-modules are pipelined, and a fully pipelined unit. Implemented on Artix-7 (xc7a200tlffg1156-2L)

4.7 Arithmetic Optimizations

The third stage of optimization mentioned in Chapter 3 is computational optimization of the core modular arithmetic units of the hardware system. As the individual RNS channels perform a standard 32-bit modular arithmetic, they each can be optimized via effective hardware acceleration techniques for low level arithmetic. Modular addition, subtraction, modular multiplication, and modular MAC operations are the key focuses of this optimization. A general survey of high level synthesis

systems can be found at [83] and can help guide optimization. The basis for relevant modular arithmetic is also discussed in Section 3.3.

4.7.1 Modular Adder

The design of the 32-bit modular addition unit is given in Figure 16. Implementation results for the channel modular adder is given in Table 10. The standard method is found using system tools to infer a modular addition. Reductions via the VHDL modulo operator were generally found to be highly inefficient. To optimize these modular additions, a hardware optimizing technique utilizing a magnitude comparison and conditional subtraction was first applied. The last attempted method is found is Sapphire [10], which replaces the inferred reduction with one addition, one subtraction, and a logic-based select bit to send one of the signals through. This was found to be most optimal and hardware friendly and was the chosen channel addition technique.

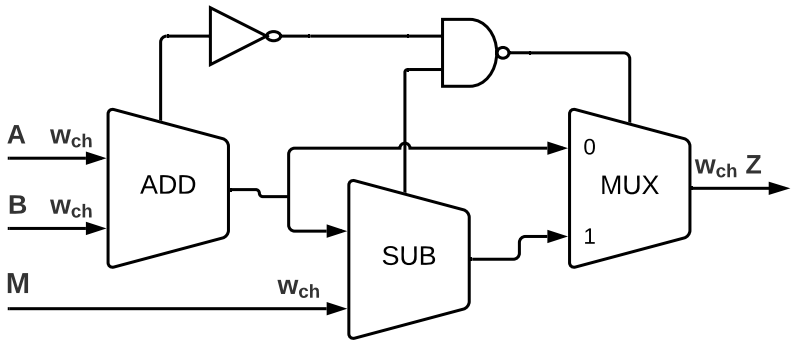

Figure 16. Channel modular adder unit.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF
Standard	9	32	8.753	114.246	0.133	743	864
Comparison	9	32	7.727	129.416	0.132	599	864
Optimized Carry	9	32	2.645	135.962	0.129	590	864
Standard	5	32	8.430	118.623	0.290	417	480
Comparison	5	32	7.029	142.268	0.120	375	480
Optimized Carry	5	32	2.566	134.517	0.120	325	480
Standard	4	32	8.611	116.131	0.256	397	384
Comparison	4	32	7.434	134.517	0.118	300	384
Optimized Carry	4	32	3.177	146.563	0.118	258	384

Table 10. Channel Modular Adder on Artix-7 (xc7a200tlffg1156-2L) using standard inferred reduction, modular addition with a magnitude comparator-based subtraction, and modular addition utilizing a carry select bit from Saphire

4.7.2 Modular Subtractor

The design of the 32-bit modular subtraction unit is given in Figure 17. Table 11 shows the modular subtractor implementation results obtained using a variant of the optimized modular addition hardware design discussed in Section 4.7.1. It benefits from requiring only one select line from the subtraction.

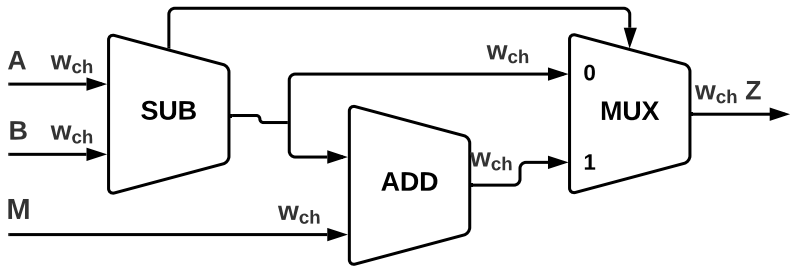

Figure 17. Channel modular subtractor unit.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF
Optimized Select	9	32	2.747	137.873	0.128	569	864
Optimized Select	5	32	2.666	136.351	0.119	315	480
Optimized Select	4	32	2.379	131.216	0.117	255	384

Table 11. Channel Modular Subtractor on Artix-7 (xc7a200tlffg1156-2L) using the subtractor select bit method from Saphire

4.7.3 Modular Multiplier via Barrett Reduction

The design of the 32-bit modular multiplication unit is given in Figure 18. In Table 12, a comparison of modular multiplication is given. It compares the FPGA tool's inferred channel modular multiplication to an optimized Barrett multiplication (Section 3.3.2). There is a great improvement in delay and area due to difficulties the tools face in implementing the reduction portion of the modular multiplication. Barrett relies on an adjustment factor β. For maximum flexibility and testing purposes, this values is assumed to be maximum in the design. This should eventually be lowered when the channel modulus for the Barrett reduction is ensured to be static and remaining for the design.

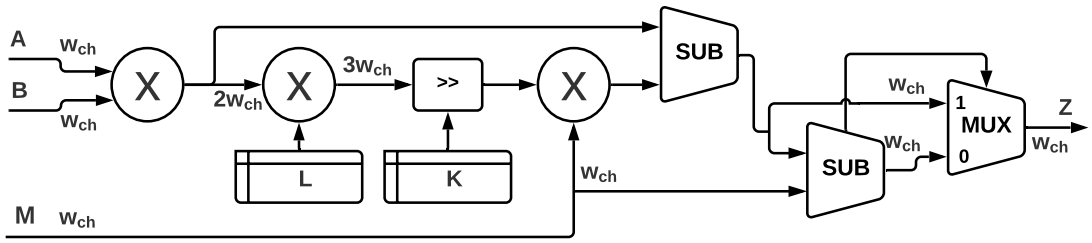

Figure 18. Channel modular multiplier via Barrett Reduction.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF	DSP
Inferred	9	32	205.713	4.861	0.454	18481	288	36
Barrett	9	32	12.949	77.226	0.156	1022	288	36
Inferred	5	32	205.857	4.858	0.303	10348	160	20
Barrett	5	32	12.899	77.525	0.136	565	160	20
Inferred	4	32	205.00	4.878	0.265	8297	128	16
Barrett	4	32	12.964	77.136	0.120	450	128	16

Table 12. Channel modular multiplier on Artix-7 (xc7a200tlffg1156-2L) using standard inferred modular reduction and modular multiplication using a flexible Barrett reduction

4.7.4 Modular MAC

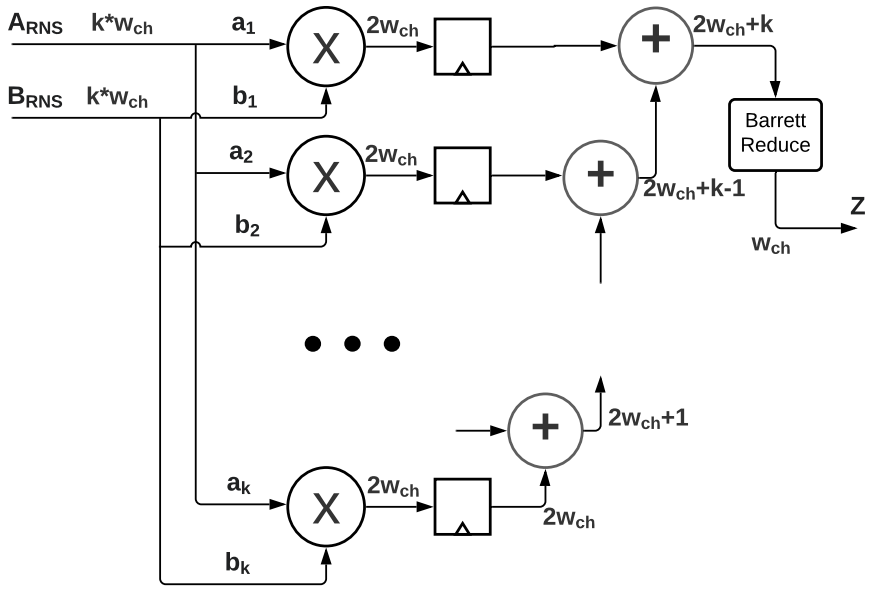

Figure 19. Unrolled RNS MAC operation.

The modular MAC design is presented in Figure 19. The MAC unit found in the base extension units has two different approaches to optimal realization. For an area sensitive design, an accumulation register approach can be implemented along

side a multiplication unit. This comes at the expense of multiple clock cycles for a single summation. The second option is to unroll the MAC operation and pipeline at the intermediate additions. This enables a higher throughput at the expense of more resources on the FPGA. The unrolled MAC is chosen in this design and has implementation results provided in Table 13.

Method	k	w_{ch}	d (ns)	f (Mhz)	P (W)	LUT	FF	DSP
Mod Adder	9	32	46.09	21.696	0.142	447	32	27
Barrett	9	32	14.07	71.073	0.139	314	32	27
Pipelined	9	32	9.742	102.648	0.142	263	320	27
Mod Adder	5	32	27.467	36.407	0.128	251	32	15
Barrett	5	32	13.079	76.458	0.126	164	32	15
Pipelined	5	32	9.737	102.701	0.128	139	192	15
Mod Adder	4	32	22.791	43.876	0.124	197	32	12
Barrett	4	32	12.112	82.562	0.124	164	32	12
Pipelined	4	32	9.676	103.348	0.124	151	160	12

Table 13. Unrolled channel MAC unit on Artix-7 (xc7a200tlffg1156-2L) using standard multiplier and modular addition units, using a final Barrett reduction to replace the modular adders, and a pipelined version of the barrett MAC unit

4.8 On-chip Memory

The two input polynomials of size n and bitwidth $w = \lceil \log_2 M \rceil$ are stored in on-chip memory. Due to the polynomials being stored in both bases, the total storage requirements of both polynomials combined is $2 \times n \times k \times w_{ch}$ bits for $k = 9$ channels. For lattice schemes using a parameter set $\{k = 9, w_{ch} = 8, n = 1024\}$, a single polynomial size is 72 Kb with 144 Kb the total space for two input polynomials. For FHE schemes using a parameter set $\{k = 9, w_{ch} = 32, n = 4096\}$, a single polynomial size is 1.125 Mb with 2.25 Mb total for two polynomials. When dealing with a single

NTT unit of sequential butterfly units, one input and one output value per cycle via a dual port BRAM is acceptable. This is by virtue of the butterflies operating in a chained fashion and passing half of its input values into a FIFO before processing occurs.

Scheme	n	w	FIFO	Polynomial	Twid. Factors	RNS Weights
Lattice	1024	32	67.5 Kb	72 Kb	72 Kb	2.53 Kb
FHE	4096	128	1.055 Mb	1.125 Mb	1.125 Mb	648 bits

Table 14. Memory requirements for butterfly FIFO, input polynomials, twiddle factors and RNS conversion weights given targeted schemes' NTT size n and coefficient width w.

In an alternate hardware setup where the butterflies are not chained and looped from FIFO buffers, two memory banks of size $n/2 \times \lceil \log_2 M \rceil$ can be used. This is due to the butterfly algorithm's use of alternating indices, by which the even and odd Hamming weights of the indices' binary values can be grouped together and will always be fed into the butterfly along with one element from the other group. If the negative wrapped convolution uses two NTT units in parallel, the two full sized input polynomials will also need to be stored in separate memory banks.

The on-chip memory also stores the negative wrapped convolution twiddle factors and RNS conversion weights. The convolution twiddle factors are powers of ϕ, where $\phi^2 = \omega_n \bmod M$. As these constants have been reduced by the system modulus M, they consequently have sizes of $n \times \lceil \log_2 M \rceil$ for n different powers of ϕ in a standard setup and a size $n \times k \times w_{ch}$ for an RNS system. Under certain memory constraints, certain powers of ϕ and ω_n can be saved in the same address given the modular square root relationship. The k RNS conversion weights are also included in memory and are values that fall somewhere within the dynamic range of the RNS moduli. The total memory requirement for n twiddle factors is therefore 1.125 Mb and 72 Kb for the

FHE and lattice parameter set, respectively. The k conversion weights then require 2.53 Kb and 648 bits for the FHE and lattice parameter set, respectively.

4.9 Full Design

The diagram for the proposed hardware implementation of the RNS-based NTT multiplier containing four RNS-based butterfly units can be seen in Figure 20. Encircled in the red box is the RNS-based NTT processing unit which performs the RNS-based NTT and INTT computations in the negative wrapped convolution. The RNS-based NTT unit contains 4 sequential RNS butterflies, FIFOs of size 2048, 1024, 512, and 256, a buffer to hold intermediate NTT results, a twiddle factor bank, and a polynomial memory block connected to on-chip memory. Inside each RNS-based butterfly unit is the RNS Montgomery Multiplication unit, Bajard Base extension unit, Shenoy Base Extension unit, and RNS MAC/Barrett/Adder/Subtractor operations.

The negative wrapped convolution is achieved with this design by preparing input polynomials via the RNS Conversion unit and RNS-Hadamard unit and then calling on the NTT block multiple times to perform the RNS-based NTTs on both input polynomials and an RNS-based INTT on the resulting RNS-Hadamard product between the polynomials. The RNS-based NTT processing unit has implementation results for final sizing and operational frequencies of different parameters given in Table 15.

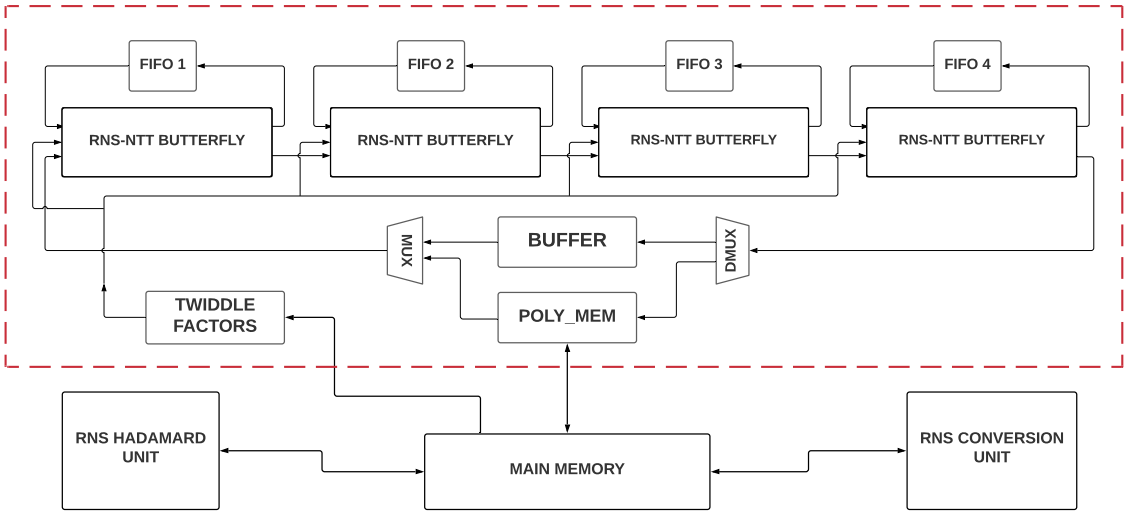

Figure 20. Full Proposed hardware design for RNS-based NTT multiplier with the Hadamard unit, RNS conversion unit, main memory, and encircled RNS-based NTT unit which has implementation results.

Name	k	w_{ch}	f (Mhz)	LUT	FF	DSP
RNS Butterfly	9	32	47.043	13239	4010	330
RNS Modmult	9	32	48.206	12094	3488	330
Shenoy Extension	5/4	32	47.943	4168	1120	116
Bajard Extension	4/5	32	67.051	3424	800	96
RNS MAC	4	32	103.348	151	160	12
RNS Barrett	9	32	77.226	1022	288	36
RNS Subtractor	9	32	131.216	255	384	0
RNS Adder	9	32	135.962	590	864	0

Table 15. RNS-based NTT polynomial multiplier hardware blocks on Artix-7 (xc7a200tlffg1156-2L) final implementation results.

Chapter 5

CONCLUSION

5.1 Summary

In this thesis, a pipelined and optimized RNS-based NTT polynomial multiplier is designed with the intention of accelerating Lattice-based cryptography and FHE polynomial multiplication. The thesis begins with research into Lattice-based Cryptography and FHE to find the most time-critical operations that are frequently used in lattice and FHE schemes. After determining that this is modular polynomial multiplication, the candidates for performing efficient polynomial multiplication in hardware such as Karatsuba, Schoolbook, and FFT were analyzed for their computational complexity and applicability in hardware. An NTT based approach to polynomial multiplication was chosen for its efficient radix-2 butterfly hardware implementation and use of the negative wrapped convolution. An RNS was then chosen as the primary means of obtaining datapath-level optimization in the polynomial multiplier due to how it takes advantage of hardware parallelism. All operations in the NTT and NWC were translated into RNS equivalents, thereby breaking large bitwidth computations into smaller independent operations on the RNS channels. Introduction of the RNS brought complexity to the design via the RNS Montgomery Multiplication, which relies on the Bajard and Shenoy base extensions to perform a Montgomery computation in a second RNS base.

The final RNS-based NTT polynomial multiplier design is implemented in software for algorithm validation and the RNS-based NTT unit is implemented in hardware

to obtain circuit size and timing results on a Xilinx Artix-7 FPGA. The software implementation is written in C++ and contains the Negative Wrapped Convolution, the RNS-based NTT, RNS Montgomery reduction with Bajard and Shenoy extensions, parameter generation, and test functions to accomplish the full RNS polynomial multiplication. The hardware implementation includes a pipelined and optimized RNS-based NTT unit implemented on the Xilinx Artix-7 FPGA(xc7a200tlffg1156-2L) for size and delay estimates. The design includes the RNS-based NTT unit with four RNS butterflies and FIFOs, the Montgomery multiplication unit, the Bajard Base Extension unit, the Shenoy Base Extension unit, parameterizable channel sizes and base extension constants, and optimized optimized modular 32-bit channel arithmetic for nine RNS channels including Barrett reduction and carry-based modular addition and subtraction.

For an NTT system with parameter set of nine 32-bit moduli that supports a system modulus $\lceil \log_2 M \rceil < 128$ bits and pipelining on either side of the base extension units, a maximum frequency of 47.043 MHz is achieved with space requirements totaling 13239 LUT's, 4010 FF's, and 330 DSP blocks. With these sizes, the individual butterfly unit can flexibly be implemented multiple times for higher parallelism depending on FGPA size constraints.

5.2 Future Considerations

Extra considerations for extensions to this project include utilizing new systems in RNS. An extension of the RNS system can be realized with hierarchical residue number systems (HRNS) [84]. HRNS is beneficial because it allows values outside of an RNS dynamic range to still be represented in its hierarchical number system.

This is accomplished by having a lower level RNS and higher level RNS working in combination. Further optimization to the existing design may also be found in memory access improvements. Multi-ported memory is an alternative memory access scheme that allows for more than single read/write access per cycle. In [85], multi-ported memories out of BRAM are introduced which allow for many read/writes at the expense of potential read/write misses. Implementing this on an FPGA for polynomial multiplication would lead to more restrictive design, however it has potential to greatly increase the throughput of the RNS-based NTT polynomial multiplier if the read/write count per cycle goes higher than 1. A final consideration is to map the computations onto multiple NTT units. Utilization of a single unit is usage efficient, however time-critical needs would benefit from performing multiple RNS-based NTT's in parallel.

REFERENCES

[1] Norman Göttert et al. "On the design of hardware building blocks for modern lattice-based encryption schemes". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2012, pp. 512–529.

[2] Hamid Nejatollahi et al. "Software and Hardware Implementation of Lattice-based Cryptography Schemes". In: *Center for Embedded Cyber-Physical Systems*. Nov. 2017, pp. 1–43.

[3] Chris Peikert. "An efficient and parallel Gaussian sampler for lattices". In: *Annual Cryptology Conference*. Springer. 2010, pp. 80–97.

[4] Murat Cenk et al. "Efficient big integer multiplication in cryptography". In: *International Journal of Information Security Science* 6.4 (2017), pp. 70–78.

[5] Léo Ducas et al. "Lattice signatures and bimodal Gaussians". In: *Annual Cryptology Conference*. Springer. 2013, pp. 40–56.

[6] Nagarjun C Dwarakanath and Steven D Galbraith. "Sampling from discrete Gaussians for lattice-based cryptography on a constrained device". In: *Applicable Algebra in Engineering, Communication and Computing* 25.3 (2014), pp. 159–180.

[7] Thomas Pöppelmann and Tim 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.

[8] Thomas Pöppelmann. "Efficient implementation of ideal lattice-based cryptography". In: *IT-Information Technology* 59.6 (2017), pp. 305–309.

[9] James Howe et al. "Lattice-based encryption over standard lattices in hardware". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2016, pp. 1–6.

[10] U. Banerjee, T. Ukyab, and A. Chandrakasan. "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols". In: *IACR Cryptol.* 2019 (2019), p. 1140.

[11] Eric Crockett and Chris Peikert. "LOL: Functional Lattice Cryptography". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications security*. CCS '16. Vienna, Austria, 2016, pp. 993–1005.

[12] Chris Peikert. "A decade of lattice cryptography". In: *Foundations and Trends® in Theoretical Computer Science* 10.4 (2016), pp. 283–424.

[13] Donald Donglong Chen et al. "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 62.1 (2014), pp. 157–166.

[14] Yarkin Doröz et al. "Accelerating LTV Based Homomorphic Encryption in Reconfigurable Hardware". In: *CHES*. 2015.

[15] T. Pöppelmann et al. "Accelerating Homomorphic Evaluation on Reconfigurable Hardware". In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 631.

[16] D. B. Cousins, K. Rohloff, and D. Sumorok. "Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor". In: *IEEE Transactions on Emerging Topics in Computing* 5.2 (2017), pp. 193–206.

[17] Hamid Nejatollahi et al. "Domain-specific Accelerators for Ideal Lattice-based Public Key Protocols". In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 608.

[18] Y. Su et al. "FPGA-Based Hardware Accelerator for Leveled Ring-LWE Fully Homomorphic Encryption". In: *IEEE Access* 8 (2020), pp. 168008–168025.

[19] F. Turan, S. S. Roy, and I. Verbauwhede. "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA". In: *IEEE Transactions on Computers* 69.8 (2020), pp. 1185–1196.

[20] Joël Cathébras et al. "Data flow oriented hardware design of RNS-based polynomial multiplication for SHE acceleration". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 69–88.

[21] X. Feng and S. Li. "Accelerating an FHE Integer Multiplier Using Negative Wrapped Convolution and Ping-Pong FFT". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 66.1 (2019), pp. 121–125.

[22] A. C. Mert, E. Öztürk, and E. Savaş. "Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture". In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. 2019, pp. 253–260.

[23] Hamid Nejatollahi et al. "Exploring energy efficient quantum-resistant signal processing using array processors". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 1539–1543.

[24] L. Sousa, S. Antao, and P. Martins. "Combining Residue Arithmetic to Design Efficient Cryptographic Circuits and Systems". In: *IEEE Circuits and Systems Magazine* 16.4 (2016), pp. 6–32.

[25] A. P. Shenoy and R. Kumaresan. "Fast base extension using a redundant modulus in RNS". In: *IEEE Transactions on Computers* 38.2 (1989), pp. 292–297.

[26] J.Bajard, L.Didier, and P. Kornerup. "Modular multiplication and base extensions in residue number systems". In: *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001.* 2001, pp. 59–65.

[27] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509.

[28] Miklós Ajtai. "Generating hard instances of lattice problems". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing.* 1996, pp. 99–108.

[29] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. "NTRU: A ring-based public key cryptosystem". In: *International Algorithmic Number Theory Symposium.* Springer. 1998, pp. 267–288.

[30] Damien Stehlé and Ron Steinfeld. "Making NTRU as secure as worst-case problems over ideal lattices". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2011, pp. 27–47.

[31] Vadim Lyubashevsky and Gregor Seiler. "NTTRU: truly fast NTRU using NTT". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 180–201.

[32] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.

[33] Vadim Lyubashevsky, Chris Peikert, and Oded 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.

[34] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: *Journal of the ACM (JACM)* 60.6 (2013), pp. 1–35.

[35] Zvika Brakerski and Vinod Vaikuntanathan. "Fully homomorphic encryption from ring-LWE and security for key dependent messages". In: *Annual Cryptology Conference*. Springer. 2011, pp. 505–524.

[36] Phong Nguyen. "Cryptanalysis of the Goldreich-Goldwasser-Halevi cryptosystem from crypto'97". In: *Annual International Cryptology Conference*. Springer. 1999, pp. 288–304.

[37] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 197–206.

[38] Craig Gentry and Dan Boneh. *A fully homomorphic encryption scheme*. Stanford university Stanford, 2009.

[39] Yusong Du and Baodian Wei. "On Rejection Sampling Algorithms for Centered Discrete Gaussian Distribution over Integers." In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 988.

[40] James Howe et al. "On practical discrete Gaussian samplers for lattice-based cryptography". In: *IEEE Transactions on Computers* 67.3 (2016), pp. 322–334.

[41] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "High precision discrete Gaussian sampling on FPGAs". In: *International Conference on Selected Areas in Cryptography*. Springer. 2013, pp. 383–401.

[42] J. Howe et al. "On Practical Discrete Gaussian Samplers for Lattice-Based Cryptography". In: *IEEE Transactions on Computers* 67.3 (2018), pp. 322–334.

[43] Yusong Du and Baodian Wei. "On Rejection Sampling Algorithms for Centered Discrete Gaussian Distribution over Integers." In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 988.

[44] Léo Ducas and Phong Q Nguyen. "Faster Gaussian lattice sampling using lazy floating-point arithmetic". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2012, pp. 415–432.

[45] Chaohui Du and Guoqiang Ba. "High-performance software implementation of discrete Gaussian sampling for lattice-based cryptography". In: *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*. IEEE. 2016, pp. 220–224.

[46] Jean Pierre David, Kassem Kalach, and Nicolas Tittley. “Hardware complexity of modular multiplication and exponentiation”. In: *IEEE Transactions on Computers* 56.10 (2007), pp. 1308–1319.

[47] Ciara Rafferty, Máire O’Neill, and Neil Hanley. “Evaluation of large integer multiplication methods on hardware”. In: *IEEE Transactions on Computers* 66.8 (2017), pp. 1369–1382.

[48] Andrei L Toom. “The complexity of a scheme of functional elements realizing the multiplication of integers”. In: *Soviet Mathematics-Doklady*.

[49] Stephen A Cook and Stål O Aanderaa. “On the minimum computation time of functions”. In: *Transactions of the American Mathematical Society* 142 (1969), pp. 291–314.

[50] Anatolii Karatsuba and Yu Ofman. “Multiplication of Multidigit Numbers on Automata”. In: *Soviet Physics Doklady* 7 (Dec. 1962), p. 595.

[51] Arnold Schönhage and Volker Strassen. “Schnelle multiplikation grosser zahlen”. In: *Computing* 7.3-4 (1971), pp. 281–292.

[52] David Harvey and Joris Van Der Hoeven. “Integer multiplication in time O (n log n)”. In: (2019).

[53] Anindya De et al. “Fast integer multiplication using modular arithmetic”. In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 499–506.

[54] Thomas Pöppelmann and Tim Güneysu. “Towards practical lattice-based public-key encryption on reconfigurable hardware”. In: *International Conference on Selected Areas in Cryptography*. Springer. 2013, pp. 68–85.

[55] Sujoy Sinha Roy et al. “Compact ring-LWE cryptoprocessor”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2014, pp. 371–391.

[56] Patrick Longa and Michael Naehrig. “Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography”. In: (2016).

[57] Claudia Patricia Renterıa-Mejıa and Jaime Velasco-Medina. “High-throughput ring-LWE cryptoprocessors”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.8 (2017), pp. 2332–2345.

[58] S. Sinha Roy et al. “HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation”. In: *IEEE Transactions on Computers* 67.11 (2018), pp. 1637–1650.

[59] J-C Bajard and Laurent Imbert. “A full RNS implementation of RSA”. In: *IEEE Transactions on Computers* 53.6 (2004), pp. 769–774.

[60] M. Esmaeildoust et al. “Efficient RNS Implementation of Elliptic Curve Point Multiplication Over $GF(p)$”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.8 (2013), pp. 1545–1549.

[61] Sujoy Sinha Roy et al. “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.

[62] Yinan Kong. “Modular multiplication in the residue number system”. PhD thesis. 2009.

[63] Peter L Montgomery. “Modular multiplication without trial division”. In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[64] Paul 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.

[65] Matt Mccutchen. *C++ Big Integer Library*. 2020. URL: <https://mattmccutchen.net/bigint/>.

[66] D. Shanks. “Five number-theoretic algorithms”. In: *Proceedings of the Second Manitoba Conference on Numerical Mathematics*. 1973.

[67] Ben Lynn. *The Chinese Remainder Theorem*. 2020. URL: <https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html>.

[68] C. Chang et al. “Residue Number Systems: A New Paradigm to Datapath Optimization for Low-Power and High-Performance Digital Signal Processing Applications”. In: *IEEE Circuits and Systems Magazine* 15.4 (2015), pp. 26–44.

[69] D. Schinianakis and T. Stouraitis. “A RNS Montgomery multiplication architecture”. In: *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. 2011, pp. 1167–1170.

[70] Mohamad Ali Mehrabi. "Improved Sum of Residues Modular Multiplication Algorithm". In: *Cryptography* 3.2 (2019), p. 14.

[71] Leon Noordam. "VHDL Implementation of 4096-bit RNS Montgomery Modular Exponentiation for RSA Encryption". 2019.

[72] Filippo Gandino et al. "A general approach for improving RNS Montgomery exponentiation using pre-processing". In: *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE. 2011, pp. 195–204.

[73] Hector Pettenghi, Ricardo Chaves, and Leonel Sousa. "RNS Reverse Converters for Moduli Sets With Dynamic Ranges up to-bit". In: *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS* 60.6 (2013), p. 1487.

[74] Julius Smith. 2020. URL: <https://ccrma.stanford.edu/~jos/mdft/Number_Theoretic_Transform.html>.

[75] Project Nayuki. 2017. URL: <https://www.nayuki.io/page/number-theoretic-transform-integer-dft>.

[76] P. M. Matutino et al. "RNS Arithmetic Units for Modulo $2^n + -k$". In: *2012 15th Euromicro Conference on Digital System Design*. 2012, pp. 795–802.

[77] Piotr Patronik and Stanisław J. Piestrak. "Design of Reverse Converters for a New Flexible RNS Five-Moduli Set $\{ 2^k, 2^n-1, 2^n+1, 2^{n+1}-1, 2^{n-1}-1\}\{2k, 2n-1, 2n+1, 2n+1-1, 2n-1-1\}$ (n Even)". In: *Circuits, Systems, and Signal Processing* 36 (2017), pp. 4593–4614.

[78] R. Muralidharan and C. Chang. "Area-Power Efficient Modulo 2^n-1 and Modulo 2^n+1 Multipliers for $\{2^n-1, 2^n, 2^n+1\}$ Based RNS". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 59.10 (2012), pp. 2263–2274.

[79] L. Sousa and R. Chaves. "A universal architecture for designing efficient modulo 2/sup n/+1 multipliers". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 52 (2005), pp. 1166–1178.

[80] Evangelos Vassalos, Dimitris Bakalis, and Haridimos T Vergos. "On the design of modulo 2 n±1 subtractors and adders/subtractors". In: *Circuits, Systems, and Signal Processing* 30.6 (2011), pp. 1445–1461.

[81] Gavin Xiaoxu Yao et al. "Novel RNS parameter selection for fast modular multiplication". In: *IEEE Transactions on Computers* 63.8 (2013), pp. 2099–2105.

[82] Chris K. Caldwell. 2020. URL: <https://primes.utm.edu/lists/2small/>.

[83] Robert A Walker and Raul Camposano. *A survey of high-level synthesis systems*. Vol. 135. Springer Science & Business Media, 2012.

[84] Tadeusz Tomczak. "Hierarchical residue number systems with small moduli and simple converters". In: *International Journal of Applied Mathematics and Computer Science* 21.1 (2011), pp. 173–192.

[85] Ameer Abdelhadi. "Architecture of block-RAM-based massively parallel memory structures: multi-ported memories and content-addressable memories". PhD thesis. University of British Columbia, 2016.