Open-Source FPGA Implementation of Number-Theoretic Transform for CRYSTALS-Dilithium

A Major Qualifying Project Report

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science in

Electrical & Computer Engineering

Mathematical Sciences

By:

Brandon Voci

Advisor(s):

Patrick Schaumont [1]

Herman Servatius [2]

April 23, 2023

[1]Department of Electrical & Computer Engineering, Worcester Polytechnic Institute, Worcester MA, 01609

[2]Department of Mathematical Sciences, Worcester Polytechnic Institute, Worcester MA, 01609

1 Abstract

In Ring-Learning with Error (R-LWE) and R-LWE related cryptosystems, the Number Theoretic Transform (NTT) is the most important mathematical operation. The NTT effectively reduces the time complexity of polynomial multiplication from $O(N^2)$ to $O(N \log N)$ by exploiting a certain algebraic isomorphism. One of such R-LWE related cryptosystems is the CRYSTALS-Dilithium (Dilithium) digital signature scheme, a Round 3 finalist in the National Institute for Standards in Technology's Post-Quantum Cryptography (PQC) standardization project. This paper discusses the mathematical theory underpinning the NTT and implementation details concerning the Dilithium NTT specifically. Ultimately, we develop an open-source FPGA implementation of the Dilithium NTT at the Register-Transfer Level (RTL). Targeting speed and high-throughput, our RTL design exploits the full parallelism of the Radix-2 Decimation-in-Time (DIT) Cooley-Tukey NTT algorithm. We hence performed behavioral simulation on our RTL design via unit and regression tests, estimating a 40-48 clock cycle delay to compute the full forward transform. However, given the increased utilization of FPGA resources, our design is better suited for dedicated parallel hardware.

2 Acknowledgments

I would like to thank my advisors, Professor Patrick Schaumont and Professor Herman Servatius, for their willingness to take on this project and for their exceptional mentorship thereafter. This project would not have possible without them. Additionally, I would like to thank my family for their unwavering support.

Link to FPGA Project Repository:
`https://github.com/bvoc59/CRYSTALS_Dilithium_NTT`
Link to Python Utility Repository:
`https://github.com/bvoc59/CRYSTALS_Dilithium_NTT_Py_Util`

Contents

3 Introduction:

Learning With Errors (LWE): Over the past decade, the Learning with Errors (LWE) problem has become an extremely versatile tool in cryptographic constructions. The problem essentially consists of solving a system of linear equations with noise from uniformly sampled values over a particular finite-dimensional lattice [1]. In particular, given the integers $N \geq 1$, $q > 1$, and a real number $\epsilon \geq 0$, the LWE problems asks to recover a secret vector or lattice coordinate, $\mathbf{s} \in \mathbb{Z}_q^N$, given the list of approximate equations [1, 2]:

$$\langle \mathbf{s}, \mathbf{a}_i \rangle \approx_\epsilon b_i \mod q \ , \ i = 1, 2, ... \tag{1}$$

where $\langle \mathbf{s}, \mathbf{a}_i \rangle := \sum_j s_j (a_i)_j$, for $j = 1, 2, ...$, and ϵ represents a certain noise or error factor. For $\epsilon = 0$, (1) can be solved by taking N samples, $1 \leq i \leq N$, and thereafter solving an $N \times N$ linear system over a finite dimensional lattice. Using a technique such as Gaussian elimination requires $O(N)$ equations and $O(N^k)$ (polynomial) time, for a positive integer k [1]. On the other hand, for $\epsilon > 0$, a simple maximum likelihood algorithm requires $O(N)$ equations and is of time complexity $2^{O(N)}$ [1]. Figure 1 illustrates this problem for $N = 2$ over the 2 dimensional lattice $\mathbb{Z}_q^2$. Here, the blue vector represents the secret vector, $\mathbf{s} \in \mathbb{Z}_q^2$, while the orange vector, $\mathbf{s}_\epsilon$, represents the solution to the erroneous system $\langle \mathbf{s}_\epsilon, \mathbf{a}_i \rangle = b_i$, for $1 \leq i \leq N$. Of course, in the case of $N = 2$, inferring the correct vector $\mathbf{s}$ is rather simple

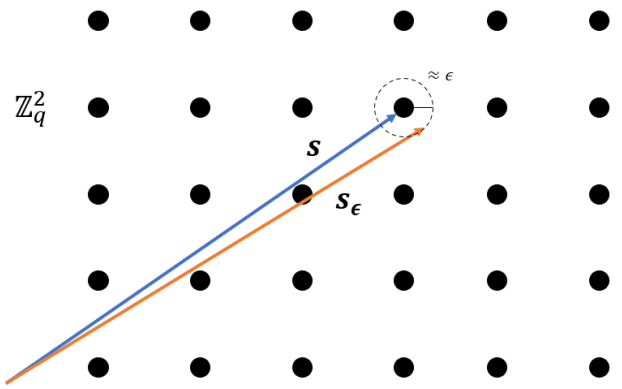

Figure 1: Two dimensional, $N = 2$, illustration of the LWE problem over the lattice $\mathbb{Z}_q^2$. The blue vector represents our secret vector, $\mathbf{s} \in \mathbb{Z}_q^2$, while $\mathbf{s}_\epsilon$ represents the vector which solves the erroneous system. Our error factor can be thought of as a small radius of ϵ about the secret vector.

given that there are only a handful of nearby lattice points to choose from. However, for a sufficiently large of value of N, this problem becomes extremely difficult as there are a multitude of different directions one can move. One of the best known algorithms for solving this problem requires both $2^{O(N/\log N)}$ time and equations [3]. Indeed, for a sufficiently large N, it has been shown that the computational hardness of LWE is equivalent to worst-case lattice problems [2].

Ring-Learning With Errors (R-LWE): Despite the promising outlook of cryptography based on the hardness of LWE, two major inefficiencies involve large key sizes and computationally expensive operations, with key sizes of spatial complexity $O(N^2)$ [4,5]. Indeed, one solution involves adding an algebraic structure to the space of secret key vectors, replacing the lattice $\mathbb{Z}_q^N$ with the polynomial ring:

$$R_q := \mathbb{Z}_q[x]/\langle x^N + 1\rangle \tag{2}$$

where $N = 2^d$ for some integer $d > 1$ and $q \equiv 1 \mod 2N$ is a sufficiently large public prime modulus [4]. Set-wise, R_q is equivalent to the our former space, though we shall find that (2) enables much more efficient arithmetic operations and reduces the spatial complexity of key sizes from $O(N^2)$ to $O(N)$. Thus, the variation of the LWE problem which utilizes (2) is known as the Ring-LWE problem, or simply the R-LWE problem. With respect to security, Balbas gives a proof of hardness which, similar to the case of LWE, involves a reduction from lattice problems [5]. Lyubashevsky et al. prove a comparable hardness result, assuming that worst-case problems on ideal lattices are hard for both classical and quantum computers [4]. This alleged computational hardness together with aforementioned versatility has rendered R-LWE a major foundational element of Post-Quantum Cryptography (PQC) [5].

CRYSTALS & Number Theoretic Transform (NTT): The preeminence of PQC based on the R-LWE problem can be seen from the National Institute of Standards and Technology (NIST) PQC standardization project. Among the Round 3 finalists for public-key encryption and digital signature algorithms were the two cryptographic primitives of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), CRYSTALS-Kyber (Kyber) and CRYSTALS-Dilithium (Dilithum), respectively [6]. These cryptosystems utilize so-called "module lattices"which have less algebraic structure than those typically associated with R-LWE and are in fact closer to those associated with LWE [7]. Note that the rendition of LWE utilizing such modular lattices is hence termed M-LWE. Indeed, while the hardness of Dilithium explicitly depends on M-LWE, for a sufficient N, Kyber and Dilithium inherit the same efficiency as R-LWE [7,8].

In CRYSTALS and in general R-LWE settings, the most important mathematical operation is the Number Theoretic Transform (NTT). The NTT can be thought of as an algebraically-compatible version of the Discrete-Fourier Transform (DFT) for the finite-field $\mathbb{Z}_q$, where q denotes a positive prime integer [9]. Ultimately, the NTT simplifies multiplicative operations over R_q by exploiting a certain algebraic isomorphism, analogous to the application of the DFT in simplifying complex-valued polynomial multiplication. More precisely, the NTT computes the ring isomorphism:

$$\mathbf{NTT} : R_q \to \prod_{i=1}^{N} R_q^i := R_q^1 \times R_q^2 \times \cdots \times R_q^N \tag{3}$$

where each R_q^i is associated with an irreducible, linear factor of $x^N + 1$ over $\mathbb{Z}_q$. This renders ordinary polynomial multiplication into pointwise-multiplication. Furthermore, the same

basic techniques which enable Fast-Fourier Transforms (FFT) extend to the NTT [9, 10]. Thus, with respect to multiplication over R_q, the NTT enables a reduction from quadratic time, $O(N^2)$, to logarithmic, $O(N \log N)$. Note that in Dilithium, one has $N = 2^8 = 256$ and $q = 2^{23} - 2^{13} + 1 \equiv 1 \mod 512$.

FPGA Implementations: Already, there are a number of Field-Programmable Gate Array (FPGA) implementations of the Dilithium digital signature scheme [11–13]. These implementations can be regarded as advancements towards non-programmable hardware solutions, such as Application-Specific Integrated Circuits (ASIC). Despite the potential move towards non-programmable solutions, efficient and optimized FPGA implementations are valuable as starting points in the design flow. Indeed, because the NTT is generally the most computationally expensive operation in Dilithium, many of these FPGA implementations tend to focus on optimized NTT implementations foremost. In [11], the authors describe an iterative NTT which processes 4 polynomial coefficients per clock cycle and 2 NTT layers at once, utilizing an arrangement of $4, 2 \times 2$ Cooley-Tukey butterflies. Here, the ideal cost of the forward transform is $N/4 \cdot \log_2 N/2 = 256$ clock cycles. The authors in [12] utilize a similar butterfly arrangement, compromising between speed and hardware resources. On the other hand, the NTT architecture in [13] requires 533 clock cycles, optimizing the number of Look-Up Tables (LUTs) required.

Our Contributions: In this paper, we first discuss the algebraic framework associated with R-LWE and R-LWE related cryptosystems, including Dilithium, so as to develop the mathematical theory underpinning the NTT. We hence discuss the implementation of the Dilithium NTT, focusing on the Radix-2 and Radix-4 Decimation-in-Time (DIT) Cooley-Tukey NTT algorithms. Ultimately, we establish an FPGA implementation of the (forward) Dilithium NTT, utilizing the Radix-2 version of this algorithm and designing at the Register-Transfer Level (RTL). Targeting speed and high-throughput, our RTL design exploits the full parallelism of the Cooley-Tukey NTT algorithm, performing the forward transform on $3N$ Byte = 7.68 kB blocks of data at a time. By performing behavioral simulations on our RTL design, we estimate the cost of the forward transform to be $5 \cdot \log_2 N = 40$ clock cycles at best and $6 \cdot \log_2 N = 48$ cycles at worst, with respect to speed. However, the fully-parallel architecture which enables this speed incurs a significant number of FPGA resources. Thus, our design is better suited for dedicated parallel-computational hardware.

4 Mathematical Background:

In this Section, we expose and discuss the algebraic framework utilized in R-LWE and R-LWE related cryptography, in which arithmetic operations are performed over certain polynomial rings. As previously noted, this particular framework enables efficient multiplicative operations by exploiting a particular ring isomorphism. More specifically, suppose F is a field and let $\Phi(x)$ denote an $F-$valued polynomial of degree N. If $\Phi(x)$ splits into irreducible, linear factors over F, then given the quotient ring $F[x]/\langle\Phi(x)\rangle$, one can find an isomorphism into the product ring:

$$\prod_{i=1}^{N} F[x]/\langle\Phi_i(x)\rangle := F[x]/\langle\Phi_1(x)\rangle \times F[x]/\langle\Phi_2(x)\rangle \times \cdots \times F[x]/\langle\Phi_N(x)\rangle \tag{4}$$

where each $\Phi_i(x)$ is an irreducible, linear factor of $\Phi(x)$, associated with its Chinese Remainder Theorem (CRT) factorization, and $F[x]/\langle\Phi_i(x)\rangle \cong F$. In the product ring, multiplication is pointwise, hence rendering multiplicative operations less computationally expensive. Figure 2 illustrates this algebraic framework given $a(x), b(x) \in F[x]/\langle\Phi(x)\rangle$. If $F = \mathbb{C}$ then

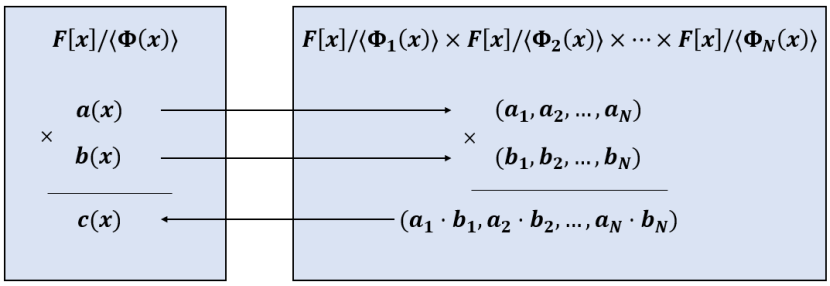

Figure 2: Algebraic framework utilized in R-LWE and R-LWE related cryptography. One can perform multiplicative operations in the product ring, which requires less computational effort.

this isomorphism can be computed using the DFT. On the other hand if F is the finite field of order q, with q prime such that $\mathrm{GF}(q) \cong \mathbb{Z}_q$, then one can use the NTT. It is in this sense that the NTT is the finite-field equivalent of the DFT.

In Section 4.1, we discuss polynomial rings, focusing primarily on the finite ring $\mathbb{Z}_q[x]/\langle\Phi(x)\rangle$ and proving the finite-field case of the isomorphism shown in (4). In Sections 4.2 and 4.3, we turn to discrete-time signals analysis, which gives explicit maps for certain $\Phi(x)$. These explicit maps are in fact the direct forms of the DFT and NTT, which we present in Sections 4.4 and 4.5, respectively.

4.1 Polynomial Rings:

Let R be a commutative ring with $1 \in R$ and let $R[x]$ denote the ring of $R-$valued polynomials, in the indeterminate x. We write polynomials as formal linear combinations: thus, given $a(x) \in R[x]$:

$$a(x) = \sum_i a_i x^i \ , \ a_i \in R$$

For $a(x), b(x) \in R[x]$, we recall that addition is defined component-wise and multiplication, $c(x) = a(x) \cdot b(x)$, is defined as follows:

$$c(x) = (\sum_i a_i x^i) \cdot (\sum_j b_j x^j) = \sum_i \sum_j a_i b_j x^{i+j} \tag{5}$$

We note that if $a(x), b(x)$ are each of degree $N-1$, then computing $c(x)$ requires $O(N^2)$ time since for each $0 \leq i < N$ we must calculate $a_i b_j$ for $0 \leq j < N$. This assumes that a single ring multiplication requires $O(1)$ time: we shall continue to make this assumption.

Suppose $\Phi(x) \in R[x]$: then $\langle \Phi(x) \rangle$ denotes the principal ideal generated by $\Phi(x)$. Setwise:

$$\langle \Phi(x) \rangle = \{ a(x) \cdot \Phi(x) \mid a(x) \in R[x] \}$$

Let F denote a field. As expected, we are primarily interested in the quotient ring:

$$R(\Phi; F) := F[x] / \langle \Phi(x) \rangle \tag{6}$$

for some $\Phi(x) \in F[x]$. In the case when F is the finite field of order q, we denote this ring as $R_q(\Phi, F)$. Note that because F is a field, we have that $F[x]$ is a euclidean domain [14]. Thus, to characterize the elements of $R(\Phi; F)$, we use the fact that for every $a'(x) \in F[x]$ there exists $a''(x), a(x) \in F[x]$ with $\deg a''(x) > \deg a(x)$ or $a(x) = 0$, such that:

$$a'(x) = a''(x) \cdot \Phi(x) + a(x)$$

Then, using the canonical homomorphism $\psi : F[x] \to R(\Phi; F)$ we have that $\psi(a'(x)) = a(x)$. We hence write $a(x) \equiv a'(x) \mod \Phi(x) \in R(\Phi; F)$ and impose the following rules for all $a(x), b(x) \in R(\Phi; F)$:

$$\begin{aligned} 1.\ & a(x) + b(x) && \equiv (a'(x) + b'(x)) && \mod \Phi(x) \\ 2.\ & a(x) \cdot b(x) && \equiv (a'(x) \cdot b'(x)) && \mod \Phi(x) \end{aligned}$$

which can be done since $F[x]$ is a euclidean domain. Let $N := \deg \Phi(x) > 1$. We remark that for every $a(x) \in R(\Phi; F)$, we have that $\deg a(x) < N$. Thus, in the case of the finite field of order q, we have that $|R_q(\Phi; F)| = q^N$.

We shall now turn our attention to the Chinese-Remainder Theorem (CRT). In its full generality, the CRT establishes an important relationship between a commutative ring and its comaximal ideals. Note that two ideals I_1, I_2 are comaximal if $I_1 + I_2 = 1 \in R$ [15].

Theorem 1. *Let $I_1, I_2, ..I_N$ be ideals in a commutative ring R with $1 \in R$. The map $R \to R/I_1 \times R/I_2 \times \cdots \times R/I_N$ defined by $r \mapsto (r+I_1, r+I_2, ..., r+I_N)$ is a ring homomorphism. Additionally, if the list of ideals $I_1, I_2, ..I_N$ is pairwise comaximal then this map is surjective and $I_1 \cap I_2 \cap \cdots \cap I_N = I_1 I_2 \cdots I_N$ such that:*

$$R/(I_1 I_2 \cdots I_N) \cong R/I_1 \times R/I_2 \times \cdots \times R/I_N \tag{7}$$

A proof of this general case can be found in [15]. Nonetheless, we shall prove this result for a slightly more general form of the R-LWE ring, $R_q(\Phi; \mathbb{Z}_q)$ for q prime.

Theorem 2. *Let $\Phi(x) \in \mathbb{Z}_q[x]$ be a degree N polynomial and suppose $\Phi(x)$ splits into irreducible linear factors $\Phi_1(x), \Phi_2(x), ...\Phi_N(x)$ over $\mathbb{Z}_q$. That is, $\Phi(x) = \prod_{i=1}^{N} \Phi_i(x)$ with each $\Phi_i(x)$ an irreducible linear factor over $\mathbb{Z}_q$. Then it follows that:*

$$R_q(\Phi; \mathbb{Z}_q) \cong \prod_{i=1}^{N} R_q^i(\Phi; \mathbb{Z}_q) := R_q^1(\Phi; \mathbb{Z}_q) \times R_q^2(\Phi; \mathbb{Z}_q) \times \cdots \times R_q^N(\Phi; \mathbb{Z}_q) \tag{8}$$

where $R_q^i(\Phi; \mathbb{Z}_q) := \mathbb{Z}_q[x]/\langle \Phi_i(x) \rangle$.

Proof: We first point that every euclidean domain is a unique factorization domain such that the factorization $\Phi(x) = \Phi_1(x)\Phi_2(x)\cdots\Phi_n(x)$ is unique. Indeed, as suggested in the previous theorem, let us define the map $\phi : R_q(\Phi; \mathbb{Z}_q) \to \prod_i R_q^i(\Phi; \mathbb{Z}_q)$ as follows:

$$\begin{aligned} a(x) \mapsto & (a'(x) \mod \Phi_1(x), a'(x) \mod \Phi_2(x), ..., a'(x) \mod \Phi_N(x)) := \\ & (a_1(x), a_2(x), ..., a_N(x)) := (a_i(x)) \end{aligned}$$

In other words, the factors $\Phi_1(x), \Phi_2(x), ..., \Phi_N(x)$ generate a list of comaximal ideals. One can verify that this map is indeed a ring homomorphism. In the case of addition, we observe that:

$$\begin{aligned} \phi(a(x) + b(x)) &= \phi((a'(x) + b'(x)) \mod \Phi(x)) \\ &= ((a'(x) + b'(x)) \mod \Phi_i(x)) \\ &= (a'(x) \mod \Phi_i(x) + b'(x) \mod \Phi_i(x)) \\ &= (a'(x) \mod \Phi_i(x)) + (b'(x) \mod \Phi_i(x)) \\ &= (a_i(x)) + (b_i(x)) \\ &= \phi(a(x)) + \phi(b(x)) \end{aligned}$$

with the case of multiplication (defined pointwise) following nearly identical reasoning. With respect to injectivity, since each $\Phi_i(x)$ is irreducible over $\mathbb{Z}_q$, the representation $(a_i(x))$ is coordinate-wise unique. With respect to surjectivity, since each $R_q^i(\Phi; \mathbb{Z}_q)$ is a linear factor we have that:

$$R_q^i(\Phi; \mathbb{Z}_q) \cong \mathbb{Z}_q$$

Therefore:

$$|\prod_{i=1}^{N} R_q^i(\Phi; \mathbb{Z}_q)| = \prod_{i=1}^{N} |R_q^i(\Phi; \mathbb{Z}_q)| = |\mathbb{Z}_q|^N = q^N$$

and on the other hand we have that $|R_q(\Phi;\mathbb{Z}_q)| = q^N$. Thus, ϕ is necessarily surjective given that it is injective and the domain and codomain have the same size. We conclude that ϕ is indeed a ring isomorphism $\square$.

Although implicitly understood in the previous theorems, we remark that given two rings R_1, R_2 then the product ring $R_1 \times R_2$ is naturally equipped with pointwise multiplication, $\odot$. Now, in the product $\prod_i R_q^i(\Phi;\mathbb{Z}_q)$, since $R_q^i(\Phi;\mathbb{Z}_q) \cong \mathbb{Z}_q$, we note that each $(a_i(x))$ reduces to some $a_i \in \mathbb{Z}_q$. Thus, given $(a_i), (b_i) \in \prod_i R_q^i(\Phi;\mathbb{Z}_q)$ the pointwise product is:

$$\begin{aligned}(a_i) \odot (b_i) &= (a_1, a_2, ..., a_N) \odot (b_1, b_2, ..., b_N) \\ &= (a_1 \cdot a_2, a_2 \cdot b_2, ..., a_N \cdot b_N) \\ &= (c_1, c_2, ..., c_N) \\ &= (c_i)\end{aligned}$$

where $c_i := a_i \cdot b_i$ for $1 \leq i \leq N$. One can readily see from this calculation that multiplication in $\prod_i R_q^i(\Phi;\mathbb{Z}_q)$ requires $O(N)$ time.

4.2 Discrete-Time Signals:

Given the ring $F[x]/\langle\Phi(x)\rangle$, we know that for a sufficient choice of $\Phi(x)$ the CRT predicates the existence of an isomorphism into the product ring $\prod_i F[x]/\langle\Phi_i(x)\rangle$. The maps given in Theorem(s) 1 and 2, however, are somewhat inexplicit. Indeed, discrete-time signals analysis gives explicit maps, $F[x]/\langle\Phi(x)\rangle \to \prod_i F[x]/\langle\Phi_i(x)\rangle$ for a certain class of $\Phi(x)$.

A discrete-time signal is nothing more than an integer-indexed sequence. More precisely, given the sets $D \subseteq \mathbb{Z}$ and $S \subseteq F$, F a field, an F-valued discrete-time signal is a sequence of the form $a : D \to S$, denoted by $\{a[n]\}_{n\in D}$ or simply $a[n]$. As expected, we generally take $F = \mathbb{C}$ or $F = \mathbb{Z}_q$. We shall write $\mathcal{D}(D;S)$ for the set of such sequences: thus, $\{a[n]\} \in \mathcal{D}(D;S)$. If $|D| < \infty$ then $a[n]$ is said to be finite-duration: otherwise, $a[n]$ is infinite-duration. Note that any finite-duration signal, $a[n] \in \mathcal{D}(D;S)$, can be naturally extended into an infinite-duration one by setting $a[n] := 0$ for each $n \in \mathbb{Z} \setminus D$. Often, we refer to the terms of the sequence as samples.

Generally, we are interested in finite-duration signals defined over $D = [N] := \{0, 1, ..., N-1\}$, for some $N \in \mathbb{Z}^{>0}$: that is, $a[n] \in \mathcal{D}([N];S)$. Here, $a[n]$ can be described as a finite list of length N, indexed by n. That is:

$$a[n] = \{a[0], a[1], ..., a[N-1]\}\ ,\ 0 \leq n < N$$

As we shall come to see, it is useful to realize discrete-time signals as the coefficients of F-valued polynomials, in some indeterminate, x. To do this precisely, given $a[n] \in \mathcal{D}(D;S)$, we define the discrete-time polynomial operator, $\mathcal{P} : \mathcal{D}(D;S) \to F[x]$, as follows:

$$a(x) := \mathcal{P}\{a[n]\} := \sum_{n\in D} a[n]x^n \tag{9}$$

We further define it's inverse, the polynomial discrete-time operator, $\mathcal{P}^{-1} : F[x] \to \mathcal{D}(D; S)$, described by:

$$a[n] := \mathcal{P}^{-1}\{a(x)\} = \mathcal{P}^{-1}\{\sum_{n\in D} a[n]x^n\} := \{a[n]\}_{n\in D} \tag{10}$$

In other words, $a(x)$ is the polynomial with coefficients $a[n]$, multiplied by the indeterminate x^n. Thus, the image of $a[n] \in \mathcal{D}([N]; S)$ under $\mathcal{P}$ is:

$$\{a[0], a[1], ..., a[N-1]\} \mapsto a[0] + a[1]x + \cdots + a[N-1]x^{N-1} = \sum_{i=0}^{N-1} a[i]x^i$$

So the discrete-time polynomial operator maps discrete-time signals of length N to polynomials of degree $N-1$. Conversely, the polynomial discrete-time operator maps polynomials of degree $N-1$ to signals of length N since $\mathcal{P}^{-1}\{\mathcal{P}\{a[n]\}\} = a[n]$ for all $a[n] \in \mathcal{D}(D; S)$.

In the next Section, we shall make use of this operator in proving three critical propositions.

4.3 Convolution Operations:

The set $\mathcal{D}(\mathbb{Z}; F)$ is equipped with three important variations of a type of mathematical operation known as a (discrete) convolution. These are essentially binary operators, $\mathcal{D}(\mathbb{Z}; F) \times \mathcal{D}(\mathbb{Z}; F) \to \mathcal{D}(\mathbb{Z}; F)$, which exhibit a certain correspondence with polynomial multiplication over various rings. Here we shall introduce these operations and prove their respective connection(s) to multiplication over such rings. Throughout, we shall assume that $N > 0$ is an integer value.

To begin, we define the linear convolution of two discrete-time signals. Namely, given $a[n], b[n] \in \mathcal{D}(\mathbb{Z}; F)$ then the (discrete) linear convolution is the binary operator $* : \mathcal{D}(\mathbb{Z}; F) \times \mathcal{D}(\mathbb{Z}; F) \to \mathcal{D}(\mathbb{Z}; F)$ described in [16] by:

$$c[n] := a[n] * b[n] := \sum_{m\in\mathbb{Z}} a[m]b[n-m] \tag{11}$$

Implicitly, we apply the infinite-duration extension detailed in the previous Section so that $a[m], b[n-m]$ are well-defined for every integer.

Although (11) gives a complete description of the linear convolution for infinite-duration signals, it lacks computational insight for finite-duration signals, given the unspecified lower and upper bounds of the summation. Thus, given $a[n], b[n] \in \mathcal{D}([N]; F)$, a modified, computationally practical version of the linear convolution is:

$$c[n] := a[n] * b[n] := \begin{cases} \sum_{m=0}^{n} a[m]b[n-m] & 0 \le n < N \\ \sum_{m=n-(N-1)}^{N-1} a[m]b[n-m] & N \le n < 2N-1 \end{cases} \tag{12}$$

With this definition, we have that $c[n] \in \mathcal{D}([2N-1]; F)$: that is, $c[n]$ is of length $2N-1$:

$$c[n] = a[n] * b[n] = \{c[0], c[1], ..., c[2N-2]\}\ ,\ 0 \le n < 2N-1$$

We claim that the linear convolution of two discrete-time signals corresponds to the product of the associated polynomials. This is made precise in the following proposition.

Proposition 1. *Suppose we have* $a[n], b[n] \in \mathcal{D}([N]; F)$. *Then:*

$$a[n] * b[n] = \mathcal{P}^{-1}\{\mathcal{P}\{a[n]\} \cdot \mathcal{P}\{b[n]\}\} \tag{13}$$

Proof: Let $\mathcal{P}\{a[i]\} = \sum_{i=0}^{N-1} a[i]x^i := a(x)$ and $\mathcal{P}\{b[j]\} = \sum_{j=0}^{N-1} b[j]x^j := b(x)$, wherein we replace n with the dummy indices i and j in the summation. Further let $c(x) := a(x) \cdot b(x)$. Note that $\deg a(x) = \deg b(x) = N-1$ so $\deg c(x) = 2N-2$ implying that $\mathcal{P}^{-1}\{c(x)\}$ is a signal of length $2N-1$, as expected. It remains to be shown that the coefficients of $c(x)$ correspond to the discrete-time signal $c[n] := a[n] * b[n]$. To see this, we note that the coefficients of product polynomial:

$$c(x) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a[i]b[j]x^{i+j} = \sum_{k=0}^{2N-2}\Bigg(\sum_{i+j=k} a[i]b[j]\Bigg)x^k$$

can be represented by an $N \times N$ square matrix, where the indices i, j specify the row and column:

$$\begin{bmatrix} a[0]b[0] & a[0]b[1] & \cdots & a[0]b[N-1] \\ a[1]b[0] & a[1]b[1] & \cdots & a[1]b[N-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[N-1]b[0] & a[N-1]b[1] & \cdots & a[N-1]b[N-1] \end{bmatrix}$$

The kth coefficient of the product polynomial, c_k, is a summation over the indices in which $i+j = k$. With respect to our matrix, this corresponds to $a[0]b[0]$ for $k = 0$, $a[0]b[1]+a[1]b[0]$ for $k = 1$, and the remaining cross diagonals for $k > 1$. For $0 \leq k < N$, these diagonal summations are given by:

$$c_k = \sum_{i=0}^{k} a[i]b[k-i] \ , \ 0 \leq k < N$$

which are precisely the elements of the convolution, $c[k]$, for the values of k in this range. Furthermore, for $N \leq n < 2N-1$ we have:

$$c_k = \sum_{i=k-(N-1)}^{N-1} a[i]b[k-i] \ , \ N \leq k < 2N-1$$

which again corresponds to the elements of the convolution $c[k]$ for the necessary values of k $\square$.

Thus, given the result of the above proposition, we have that linear convolution corresponds to multiplication in $F[x]$. The next convolution operation we shall consider is the cyclic or circular convolution. In particular, given $a[n], b[n] \in \mathcal{D}([N]; F)$, the cyclic convolution of $a[n]$ and $b[n]$ corresponds to the binary operator $*_N : \mathcal{D}([N]; F) \times \mathcal{D}([N]; F) \to \mathcal{D}([N]; F)$ described in [17] by:

$$c[n] := a[n] *_N b[n] = \sum_{m=0}^{N-1} a[m]b[(n-m) \mod N] \tag{14}$$

Note that while the linear convolution of two discrete-time signals of length N produces another discrete-time signal of length $2N+1$, the length of the cyclic convolution remains N. With respect to index of the second discrete-time signal in the summation, the difference $n-m$ is reduced modulo N such that one computes $\bar{n} \equiv (n-m) \mod N$. This corresponds to a circular shift of $b[n]$ by m samples of the signal [17]. With this definition, we claim that cyclic convolution corresponds to polynomial multiplication modulo x^N+1.

Proposition 2. *Let $a[n], b[n] \in \mathcal{D}([N]; F)$. Then:*

$$a[n] *_N b[n] = \mathcal{P}^{-1}\{\mathcal{P}\{a[n]\} \cdot \mathcal{P}\{b[n]\} \mod (x^N - 1)\} \tag{15}$$

Proof: We seek to evaluate the right-hand side of (16). Let $\mathcal{P}\{a[i]\} = \sum_{i=0}^{N-1} a[i]x^i$ and $\mathcal{P}\{b[j]\} = \sum_{j=0}^{N-1} b[j]x^j$ The product polynomial, $\mathcal{P}\{a[i]\} \cdot \mathcal{P}\{b[j]\}$, is therefore:

$$\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a[i]b[j]x^{i+j} = \sum_{i=0}^{N-1}\sum_{k=i}^{(N-1)+i} a[i]b[k-i]x^k$$

where the right-hand side corresponds to the ordinary convolution form and $k = i+j$. Extracting the x^N terms from the inner summation:

$$\begin{aligned}\sum_{i=0}^{N-1}\sum_{k=i}^{(N-1)+i} a[i]b[k-i]x^k &= \sum_{i=0}^{N-1} a[i]\Big(\sum_{k=i}^{N-1} b[k-i]x^k + \eta[i]\sum_{k=N}^{(N-1)+i} b[k-i]x^k\Big) \\ &= \sum_{i=0}^{N-1} a[i]\Big(\sum_{k=i}^{N-1} b[k-i]x^k + \eta[i]x^N\sum_{k=N}^{(N-1)+i} b[k-i]x^{k-N}\Big)\end{aligned}$$

where $\eta[i] := 0$ if $i = 0$ and $\eta[i] := 1$ otherwise. Here, we note that the inside of the parenthesis can be rewritten as follows for $i = 0, 1, ..., N-1$:

$$\sum_{k=i}^{N-1} b[k-i]x^k + \eta[i](x^N-1)\sum_{k=N}^{(N-1)+i} b[k-i]x^{k-N} + \eta[i]\sum_{k=N}^{(N-1)+i} b[k-i]x^{k-N}$$

such that modulo $x^N - 1$, the inner term simply becomes:

$$\sum_{k=i}^{N-1} b[k-i]x^k + \eta[i]\sum_{k=N}^{(N-1)+i} b[k-i]x^{k-N}$$

with the resulting product given by:

$$\sum_{i=0}^{N-1} a[i]\Big(\sum_{k=i}^{N-1} b[k-i]x^k + \eta[i]\sum_{k=N}^{(N-1)+i} b[k-i]x^{k-N}\Big)$$

We hence define $l := k - N$, rendering the product:

$$\sum_{i=0}^{N-1} a[i]\Bigg(\sum_{k=i}^{N-1} b[k-i]x^k + \eta[i]\sum_{l=0}^{i-1} b[l-i+N]x^l\Bigg)$$

We can now rewrite this expression in terms of the difference $k-i$ modulo N, allowing us to recombine the inner summations. That is:

$$\sum_{i=0}^{N-1} a[i]\Bigg(\sum_{k=0}^{N-1} b[(k-i) \bmod N]x^k\Bigg) = \sum_{k=0}^{N-1}\sum_{i=0}^{N-1} a[i]b[(k-i) \bmod N]x^k$$

Applying the inverse polynomial-discrete-time operator, $\mathcal{P}^{-1}$, gives the desired result $\square$.

Thus, because the cyclic convolution of two discrete-time signals corresponds to the product of the associated polynomials modulo x^N+1, we infer that a cyclic convolution corresponds to multiplication in the ring $F[x]/\langle\Phi(x)\rangle$ where $\Phi(x) = x^N - 1$.

One convenient intuition for performing the cyclic convolution of two discrete-time signals of length N includes computing the linear convolution for N points and then successively adding the remaining points at each subsequent index. To motivate the negacyclic, sometimes termed negative-wrapped convolution, we can imagine successively subtracting instead. Thus, given $a[n], b[n] \in \mathcal{D}([N]; F)$ the negacylic convolution is given as follows:

$$a[i] *_N^- b[i] := \sum_{i=0}^{N-1} a[i]\Bigg(\sum_{k=i}^{N-1} b[k-i] - \eta[i]\sum_{k=0}^{i-1} b[(k-i)\bmod N]\Bigg) \tag{16}$$

where the indices in the rightmost summation are again reduced modulo N. Indeed, complementing (16), we can write:

$$a[i] *_N b[i] := \sum_{i=0}^{N-1} a[i]\Bigg(\sum_{k=i}^{N-1} b[k-i] + \eta[i]\sum_{k=0}^{i-1} b[(k-i)\bmod N]\Bigg)$$

which justifies our computational intuition. Furthermore, just as the cyclic convolution corresponds to multiplication modulo $x^N + 1$, we have that the negacyclic convolution corresponds to multiplication modulo $x^N - 1$.

Proposition 3. *Let $a[n], b[n] \in \mathcal{D}([N]; F)$. Then:*

$$a[n] *_N^- b[n] = \mathcal{P}^{-1}\{\mathcal{P}\{a[n]\} \cdot \mathcal{P}\{b[n]\} \mod (x^N + 1)\} \tag{17}$$

Proof: The proof follows a nearly identical argument to that of the previous proposition, though instead of creating the term $x^N - 1$ we create the term $x^N + 1$ and thereafter reduce, hence causing the term $-\eta[i]$ instead of $+\eta[i]$ which corresponds exactly to how we have defined the negacyclic convolution $\square$.

Therefore, the negacylic convolution corresponds to multiplication over the ring $F[x]/\langle\Phi(x)\rangle$ for $\Phi(x) = x^N + 1$. Indeed, having introduced all three discrete convolution operations, we summarize their relationship(s) with polynomial multiplication, given discrete-time signals $a[n], b[n] \in \mathcal{D}([N]; F)$:

Convolution:	Polynomial Form:	Polynomial Ring:
Linear ($*$)	$\mathcal{P}\{a[n]\} \cdot \mathcal{P}\{b[n]\}$	$F[x]$
Cyclic ($*_N$)	$\mathcal{P}\{a[n]\} \cdot \mathcal{P}\{b[n]\} \mod (x^N - 1)$	$F[x]/\langle x^N - 1 \rangle$
Negacyclic ($*_N^-$)	$\mathcal{P}\{a[n]\} \cdot \mathcal{P}\{b[n]\} \mod (x^N + 1)$	$F[x]/\langle x^N + 1 \rangle$

Table 1: Different (discrete) convolution operations with their corresponding polynomial multiplication and the associated polynomial rings.

4.4 The Discrete Fourier Transform (DFT):

Having examined various convolution operations, we shall now return to the problem of computing the isomorphism detailed in (4) over a particular class of rings. We first detail this process using the more familiar DFT. Here, we fix $F = \mathbb{C}$.

Generally, the DFT is applied to periodic discrete-time signals, with period $N \in \mathbb{Z}^{>0}$. Despite this, we can consider signals in $\mathcal{D}([N];\mathbb{C})$ by treating the values of $a[n]$ for $0 \leq n < N$ as one period. Thus, given $a[n] \in \mathcal{D}([N];\mathbb{C})$, the forward transform associated with the DFT, $\mathbf{DFT} : \mathcal{D}([N];\mathbb{C}) \to \mathcal{D}([N];\mathbb{C})$ is the map given by [16]:

$$A[k] := \mathbf{DFT}\{a[n]\} = \sum_{n=0}^{N-1} a[n] \exp(-\frac{j2\pi kn}{N}) \ , \ 0 \leq k < N \tag{18}$$

where j denotes the imaginary unit, satisfying $j^2 = -1$. As stated in (18), the DFT converts complex-valued discrete-time signals of length N to complex-valued ones of the same length: in the language of discrete-time signals analysis, $A[k]$ is the frequency-domain representation of $a[n]$. The backwards transform associated with the DFT, $\mathbf{DFT}^{-1} : \mathcal{D}([N];\mathbb{C}) \to \mathcal{D}([N];\mathbb{C})$, is simply [16]:

$$a[n] = \mathbf{DFT}^{-1}\{A[k]\} = \frac{1}{N} \sum_{k=0}^{N-1} A[k] \exp(\frac{j2\pi kn}{N}) \ , \ 0 \leq n < N \tag{19}$$

Indeed, $\mathbf{DFT}^{-1}\{\mathbf{DFT}\{a[n]\}\} = a[n]$ for all $a[n]$ in $\mathcal{D}([N];\mathbb{C})$.

A critical feature of both the forwards and backwards transforms, is that the kernels of these transforms are the Nth roots of unity over the complex plane, $\mathbb{C}$. This can be demonstrated as follows:

$$\exp(\pm\frac{j2\pi kn}{N})^N = \exp(\pm j2\pi kn) = \cos(2\pi kn) \pm j\sin(2\pi kn) = 1$$

In fact, it can be shown that every Nth root of unity over $\mathbb{C}$, $\omega \in \mathbb{C}$, is of the form:

$$\omega = \exp(\pm j2\pi\frac{m}{N}) \ , \ m \in \mathbb{Z}$$

Although the DFT satisfies a number of remarkable properties, with respect to our discussion, its most important property involves its relation with cyclic convolution. We state and prove this relation in the proposition given below.

Proposition 4. *Let $a[n], b[n] \in \mathcal{D}([N]; \mathbb{C})$. Then:*

$$a[n] *_N b[n] = \mathbf{DFT}^{-1}\{\mathbf{DFT}\{a[n]\} \odot \mathbf{DFT}\{b[n]\}\} \tag{20}$$

Proof: Write $A[k] := \mathbf{DFT}\{a[n]\}$ and $B[k] := \mathbf{DFT}\{b[m]\}$, where the dummy index n is replaced with m in the second input signal. The point-wise product $C[k] := A[k] \odot B[k]$ is therefore given by:

$$\begin{aligned} C[k] &= (\sum_{n=0}^{N-1} a[n] \exp(-\frac{j2\pi nk}{N}))(\sum_{m=0}^{N-1} b[m] \exp(-\frac{j2\pi mk}{N})) \\ &= \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} a[n]b[m] \exp(-\frac{j2\pi(n+m)k}{N}) \end{aligned}$$

We hence define the index $l := n + m \Rightarrow m = l - n$ such that:

$$\begin{aligned} C[k] &= \sum_{n=0}^{N-1} \sum_{l=n}^{(N-1)+n} a[n]b[l-n] \exp(-\frac{j2\pi lk}{N}) \\ &= \sum_{l=n}^{(N-1)+n} \left(\sum_{n=0}^{N-1} a[n]b[l-n]) \exp(-\frac{j2\pi lk}{N}) \right) \end{aligned}$$

Observe that the index $n \leq l \leq (N-1) + n$ implies a circular shift of $b[l]$ by n samples. Indeed, since each of our signals are of length N, we can write:

$$\begin{aligned} C[k] &= \sum_{l=0}^{(N-1)} (\sum_{n=0}^{N-1} a[n]b[(l-n) \mod N]) \exp(-\frac{j2\pi lk}{N}) \\ &= \sum_{l=0}^{(N-1)} (a[l] *_N b[l]) \exp(-\frac{j2\pi lk}{N}) \end{aligned}$$

such that:

$$\mathbf{DFT}^{-1}\{ \sum_{l=0}^{(N-1)} (a[l] *_N b[l]) \exp(-\frac{j2\pi lk}{N})\} = a[l] *_N b[l]$$

completing our proof $\square$.

As previously noted, the kernel of the DFT corresponds to the Nth roots of unity over $\mathbb{C}$. More explicitly, they are the solutions to the polynomial equation $\Phi(x) := x^N - 1 = 0$ over this particular field. By the fundamental theorem of algebra, we can factor $\Phi(x)$ as follows:

$$\begin{aligned} \Phi(x) &= (x - \exp(j2\pi/N))(x - \exp(j4\pi/N)) \cdots (x - 1) \\ &= \prod_{k=1}^{N} (x - \exp(j2\pi k/N)) = \prod_{k=1}^{N} \Phi_k(x) \end{aligned}$$

where each $\Phi_k(x) := x - \exp(j2\pi k/N)$ is an irreducible, linear factor over $\mathbb{C}$, corresponding to a distinct root of $\Phi(x)$. Thus, using the CRT, we have:

$$\mathbb{C}[x]/\langle\Phi(x)\rangle \cong \prod_{k=1}^{N} \mathbb{C}[x]/\langle\Phi_k(x)\rangle \tag{21}$$

Now based on the result of (20), we know that the DFT renders circular convolution, or multiplication modulo $x^N - 1$, into pointwise multiplication. In fact, the DFT is precisely the ring isomorphism:

$$\mathbf{DFT} : \mathbb{C}[x]/\langle\Phi(x)\rangle \to \prod_{k=1}^{N} \mathbb{C}[x]/\langle\Phi_k(x)\rangle$$

We apply nearly identical reasoning in our contextualization of the NTT, discussed in the next Section.

4.5 The Number Theoretic Transform (NTT):

Just as the DFT is the ring isomorphism from $\mathbb{C}[x]/\langle\Phi(x)\rangle$ to $\prod_{k=1}^{N} \mathbb{C}[x]/\langle\Phi_k(x)\rangle$, for $\Phi(x) = x^N - 1$, the NTT is the isomorphism associated with the finite field $\mathbb{Z}_q \cong \mathrm{GF}(q)$, q a positive prime integer. Additionally, while the DFT solely utilizes, $\Phi(x) = x^N - 1$, the NTT has two distinct forms, one corresponding to the factorization of $\Phi(x) := x^N + 1$ and the other corresponding to $\Phi(x) := x^N - 1$. These are termed the cyclic form of the NTT and negacyclic form, respectively.

In order for $\mathbb{Z}_q$ to admit an $N-$point NTT, $\mathbb{Z}_q$ must contain an Nth root of unity, $\omega \in \mathbb{Z}_q$. That is, $\omega^N \equiv 1 \mod q$, such that the solutions to the polynomial equation $\Phi(x) = x^N - 1 = 0$ over $\mathbb{Z}_q$ are of the form $x = \omega^k$. $x - \omega^k$ is therefore a root of $\Phi(x)$ for $1 \leq k \leq N$. Given that such a N-th root of unity exists, we can define the cyclic form of the transform, $\mathbf{NTT} : \mathcal{D}([N]; \mathbb{Z}_q) \to \mathcal{D}([N]; \mathbb{Z}_q)$, by [9]:

$$A[k] := \mathbf{NTT}\{a[n]\} = \sum_{n=0}^{N-1} a[n]\omega^{nk} \mod q \ , \ 0 \leq k < N \tag{22}$$

and the backwards transform, $\mathbf{NTT}^{-1} : \mathcal{D}([N]; \mathbb{Z}_q) \to \mathcal{D}([N]; \mathbb{Z}_q)$, by [9]:

$$a[n] := \mathbf{NTT}^{-1}\{A[k]\} = N^{-1} \sum_{k=0}^{N-1} A[k]\omega^{-kn} \mod q \ , \ 0 \leq n < N \tag{23}$$

Of course, as in the case of the DFT, one has that $\mathbf{NTT}^{-1}\{\mathbf{NTT}\{a[n]\}\} = a[n]$ for any discrete-time signal input, valued in $\mathbb{Z}_q$ [10]. Further analogous to the DFT, (22) satisfies the proposition shown below.

Proposition 5. *Suppose* $a[n], b[n] \in \mathcal{D}([N]; \mathbb{Z}_q)$ *Then:*

$$a[n] *_N b[n] = \mathbf{NTT}^{-1}\{\mathbf{NTT}\{a[n]\} \odot \mathbf{NTT}\{b[n]\}\} \tag{24}$$

Proof: The proof of (24) is nearly identical to that of (20) in Proposition 2.4.1. Namely, let:

$$A[k] := \mathbf{NTT}\{a[i]\} = \sum_{i=0}^{N-1} a[i]\omega^{ik} \text{mod } q \ , \ B[k] := \mathbf{NTT}\{b[j]\} = \sum_{j=0}^{N-1} b[j]\omega^{jk} \text{mod } q$$

such that the point-wise product, $\mathbf{NTT}\{a[i]\} \odot \mathbf{NTT}\{b[j]\} = A[k] \cdot B[k] := C[k]$, $k = 0, 1, ..., N-1$ is:

$$C[k] = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a[i]b[j]\omega^{(i+j)k} \text{mod } q$$

As usual, let $l := i + j$ such that:

$$C[k] = \sum_{i=0}^{N-1} \sum_{l=i}^{(N-1)+i} a[i]y[l-i]\omega^{lk} \text{mod } q$$

Once again, the index $i \leq l \leq (N-1)+i$ corresponds to a circular shift of $y[l]$ by i samples such that:

$$\begin{aligned} C[k] &= \sum_{i=0}^{N-1} \sum_{l=0}^{N-1} a[i]y[(l-i) \text{mod } N]\omega^{lk} \text{mod } q \\ &= \sum_{l=0}^{N-1} \Bigg(\sum_{i=0}^{N-1} a[i]y[(l-i) \text{mod } N] \Bigg) \omega^{lk} \text{mod } q \\ &= \sum_{l=0}^{N-1} (a[i] *_N b[j])\omega^{lk} \text{mod } q = \mathbf{NTT}\{a[n] *_N b[n]\} \end{aligned}$$

Therefore, $\mathbf{NTT}\{a[n] *_N b[n]\} = \mathbf{NTT}\{a[n]\} \odot \mathbf{NTT}\{b[n]\}$ so we necessarily have that $a[n] *_N b[n] = \mathbf{NTT}^{-1}\{\mathbf{NTT}\{a[n]\} \odot \mathbf{NTT}\{b[n]\}\}$ $\square$.

To clarify the result of this proposition, we have rendered circular convolution, or polynomial multiplication modulo $x^N - 1$, into pointwise multiplication. With respect to the CRT, we have that for the factorization:

$$\Phi(x) = (x-\omega)(x-\omega^2)\cdots(x-1) = \prod_{k=1}^{N}(x-\omega^k) = \prod_{k=1}^{N} \Phi_k(x)$$

and associated ring-isomorphism:

$$\mathbb{Z}_q[x]/\langle \Phi(x) \rangle \cong \prod_{k=1}^{N} \mathbb{Z}_q[x]/\langle \Phi_k(x) \rangle$$

the cyclic form of the NTT is the map:

$$\mathbf{NTT} : \mathbb{Z}_q[x]/\langle \Phi(x) \rangle \to \prod_{k=1}^{N} \mathbb{Z}_q[x]/\langle \Phi_k(x) \rangle \ , \ \Phi(x) = x^N - 1 \ , \ \Phi_k(x) = x - \omega^k$$

To discern the negacyclic form of the transform we require an additional $\psi \in \mathbb{Z}_q$ such that $\psi^2 = \omega$. That is, ψ is a $2N$th root of unity over $\mathbb{Z}_q$. This condition is equivalent to requiring $q \equiv 1 \mod 2N$. Additionally, we require ψ to satisfy the identity $\psi \equiv -1 \mod q$. If we define the discrete-time signal:

$$\psi[n] := \psi^n \ , \ 0 \leq n < N \tag{25}$$

then we can write the negacyclic form of the NTT as follows (26):

$$A[k] := \mathbf{NTT}^{\psi}\{a[n]\} := \mathbf{NTT}\{\psi[n] \odot a[n]\} = \sum_{n=0}^{N-1} a[n]\psi^n\omega^{nk} \mod q \tag{26}$$

Similarly, defining $\psi^{-1}[n] := \psi^{-n}$ for $0 \leq n < N$ gives the negacylic form of the backwards transform:

$$a[n] := \mathbf{NTT}^{-\psi}\{A[k]\} := \psi^{-1}[n] \odot \mathbf{NTT}^{-1}\{a[n]\} = N^{-1}\psi^{-n} \sum_{k=0}^{N-1} A[k]\omega^{-nk} \mod q \tag{27}$$

It can be shown that: $a[n] = \mathbf{NTT}^{-\psi}\{\mathbf{NTT}^{\psi}\{a[n]\}\}$ [10]. Moreover, as previously alluded to, the negacyclic form of the NTT satisfies:

Proposition 6. *Suppose* $a[n], b[n] \in \mathcal{D}([N]; \mathbb{Z}_q)$. *Then:*

$$a[n] *_N^- b[n] = \mathbf{NTT}^{-\psi}\{\mathbf{NTT}^{\psi}\{a[n]\} \odot \mathbf{NTT}^{\psi}\{b[n]\}\} \tag{28}$$

Proof: As done in the previous proposition, let:

$$A[k] := \mathbf{NTT}^{\psi}\{a[i]\} = \sum_{i=0}^{N-1} a[i]\psi^i\omega^{ik} \mod q \ , \ B[k] := \mathbf{NTT}^{\psi}\{b[j]\} = \sum_{j=0}^{N-1} b[j]\psi^j\omega^{jk} \mod q$$

The point wise product $A[k] \odot B[k] := C[k]$, for $k = 0, 1, ...N-1$, is therefore:

$$\begin{aligned} C[k] &= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a[i]b[j]\psi^{i+j}\omega^{(i+j)k} \mod q \\ &= \sum_{i=0}^{N-1}\sum_{l=i}^{(N-1)+i} a[i]b[l-i]\psi^l\omega^{lk} \mod q \end{aligned}$$

where $l := i + j$. Analogous to the expansion performed in the proof of Proposition 2, we hence expand the inner summation:

$$\begin{aligned} C[k] &= \sum_{i=0}^{N-1} a[i]\Bigg(\sum_{l=i}^{N-1} b[l-i]\psi^l\omega^{lk} \mod q + \eta[i]\sum_{l=N}^{(N-1)+i} b[k-i]\psi^l\omega^{lk} \mod q\Bigg) \\ &= \sum_{i=0}^{N-1} a[i]\Bigg(\sum_{l=i}^{N-1} b[k-i]\psi^l\omega^{lk} \mod q + \eta[i]\psi^N\omega^{Nk}\sum_{m=0}^{i-1} b[m-i+N]\psi^m\omega^{mk} \mod q\Bigg) \end{aligned}$$

where $m := l - N$. Now since $\psi^N \equiv -1 \mod q$ and ω is an Nth root of unity over $\mathbb{Z}_q$, we have that:

$$C[k] = \sum_{i=0}^{N-1} a[i] \Bigg(\sum_{l=i}^{N-1} b[k-i] \psi^l \omega^{lk} \text{mod } q - \eta[i] \sum_{m=0}^{i-1} b[m-i \mod N] \psi^m \omega^{mk} \text{mod } q \Bigg)$$

which we can readily identify as the negacyclic form of the NTT of $c[n] := a[n] *_N^- b[n]$. Thus, $C[k] = \mathbf{NTT}^{\psi}\{a[k] *_N^- b[k]\}$ such that $a[k] *_N^- b[k] = \mathbf{NTT}^{-\psi}\{C[k]\} = \mathbf{NTT}^{\psi}\{\mathbf{NTT}^{\psi}\{a[k]\} \odot \mathbf{NTT}^{\psi}\{b[k]\}\}$ $\square$.

We should point out that using the identity $\psi^2 = \omega$, a more straightforward form of the forward transform in (26) is:

$$\mathbf{NTT}^{\psi}\{a[n]\} = \sum_{n=0}^{N-1} a[n] \psi^{(2k+1)n} \mod q \tag{29}$$

The CRT contextualization of the negacyclic form of the NTT utilizes the fact that over over $\mathbb{Z}_q$, the solutions to the polynomial equation $\Phi(x) = x^N + 1 = 0$ are of the form $x = \psi^{(2k+1)}$ such that $x - \psi^{2k+1}$ is a root of $\Phi(x)$ for $1 \leq k \leq N$. Thus, we have that:

$$\Phi(x) = (x - \psi^3)(x - \psi^5) \cdots (x - \psi) = \prod_{k=1}^{N} (x - \psi^{2k+1}) = \prod_{k=1}^{N} \Phi_k(x)$$

such that, given the associated ring isomorphism, the negacyclic form of the NTT is the map:

$$\mathbf{NTT}^{\psi} : \mathbb{Z}_q[x]/\langle \Phi(x) \rangle \to \prod_{k=1}^{N} \mathbb{Z}_q[x]/\langle \Phi_k(x) \rangle \ , \ \Phi(x) = x^N + 1 \ , \ \Phi_k(x) = x - \psi^{2k+1}$$

Indeed, for $\Phi(x) = x^N + 1$ as shown above, the ring $\mathbb{Z}_q[x]/\langle \Phi(x) \rangle = R_q(\Phi; \mathbb{Z}_q) := R_q$ corresponds to the classical R-LWE setting: in actuality, however, one further prescribes $N = 2^d$ for some integer $d > 0$ [4]. This condition provides computational assistance, as well shall soon see.

Given the above discussion, if we wish to compute the product $c(x) = a(x) \cdot b(x)$, for some $a(x), b(x) \in R_q$ one can simply compute:

$$c[n] = \mathbf{NTT}^{-\psi}\{\mathbf{NTT}^{\psi}\{a[n]\} \odot \mathbf{NTT}^{\psi}\{b[n]\}\} \tag{30}$$

where $a[n] := \mathcal{P}^{-1}\{a(x)\}$, $b[n], c[n]$ defined similarly. We summarize this in Algorithm 1, shown on the next page. Per our remarks at the end of Section 4.1, for $\Phi_k(x) = x - \psi^{2k+1}$, we know that multiplication in the product ring $\prod_{k=1}^{N} \mathbb{Z}_q[x]/\langle \Phi_k(x) \rangle := \prod_{k=1}^{N} R_q^k$ requires only $O(N)$ time. However, the utility of Algorithm 1 and ultimately the NTT rests on the assumption that we can compute the forward and inverse transform in reasonable time. Despite this, computational equations such as (26), and (22) imply that we require $O(N^2)$

Algorithm 1 *NTT-based Negacyclic Convolution Algorithm*

Require: $a(x), b(x) \in R_q$
  $a[n] \leftarrow \mathcal{P}^{-1}\{a(x)\}$
  $b[n] \leftarrow \mathcal{P}^{-1}\{b(x)\}$
  $A[k] \leftarrow \mathbf{NTT}^{\psi}\{a[n]\}$
  $B[k] \leftarrow \mathbf{NTT}^{\psi}\{b[n]\}$
  for $k = 0; k < N-1; k = k+1$ **do**
    $C[k] \leftarrow A[k] \cdot B[k]$
  end for
  $c(x) := a(x) \cdot b(x) \leftarrow \mathcal{P}\{\mathbf{NTT}^{-\psi}\{C[k]\}\}$

time to compute the NTT.

To exemplify this, consider computing the negacyclic form of the NTT using (26). We note that (26) can be expressed as the following matrix-vector product, where the modulo q's in our matrix are omitted for brevity:

$$\begin{bmatrix} A[0] \\ A[1] \\ \vdots \\ A[N-1] \end{bmatrix} = \begin{bmatrix} 1 & \psi & \cdots & \psi^{N-1} \\ 1 & \psi\omega & \cdots & (\psi\omega)^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & (\psi\omega^{N-1}) & \cdots & (\psi\omega^{N-1})^{N-1} \end{bmatrix} \begin{bmatrix} a[0] \\ a[1] \\ \vdots \\ a[N-1] \end{bmatrix} \tag{31}$$

In this light, the NTT is a sort of linear transformation over the vector space $\mathbb{Z}_q^N$. Nonetheless, a direct computation of the NTT involves N^2 multiplications for each $(\psi\omega^i)^j a[j]$, followed by $N(N-1) = N^2 - N$ additions across each row of this matrix, rendering its time-complexity $O(N^2)$.

Thus, if we chose to utilize a direct form the NTT, we would be no better off in time-complexity than in comparison to ordinary polynomial multiplication. Fortunately, there are a number of different FFT-inspired algorithms which compute the NTT in $O(N \log N)$ time [9, 10]. We shall discuss one particular algorithm at length in the next section.

5 Implementation Details:

Given the promising outlook of the NTT, its utility in R-LWE and R-LWE related cryptography ultimately depends on efficient implementation. In this Section, we discuss non-hardware specific implementation details regarding the the Dilithium NTT, which shall provide concrete transform parameters. Namely, the desired prime modulus $q \in \mathbb{Z}^{>1}$, the length of discrete-time signal inputs $N \in \mathbb{Z}^{>0}$, and our chosen root of unity over $\mathbb{Z}_q$, $\omega \in \mathbb{Z}_q$. Thus, in Section 5.1, given q and inferring the remaining parameters, we derive the direct form equation of the forward transform.

Per our remarks at the end of the previous Section, however, such a direct form equation is computationally insufficient. Thus, in Section 5.2 and 5.3, we mathematically and graphically describe the Radix-2 and Radix-4 Decimation-in-Time (DIT) Cooley-Tukey algorithms. In the case of the Radix-2 algorithm, we recursively decompose our N point NTT into 2, $N/2$ point NTT's. Similarly, in the case of the Radix-4 algorithm, we decompose our N point NTT into 4, $N/4$ point NTT's. In Section 5.4, in the context of the Radix-2 algorithm, we motivate performing bit reversals on the indices of the input discrete-time signal, which gives a more natural order for applying the transform. Finally, in Section 5.5, we discuss the Barrett reduction algorithm, enabling efficient modular reductions over $\mathbb{Z}_q$.

5.1 CRYSTALS-Dilithium NTT:

In Dilithium, we have the prime integer $q := 2^{23} - 2^{13} + 1 = 8380417$, corresponding to the public prime modulus [8]. It follows that there exists a 512-th root of unity over $\mathbb{Z}_q$, denoted r: according to the algorithm specifications, r is taken to be 1753 [8]. Therefore:

$$r^{512} = 1753^{512} \equiv 1 \mod q$$

We further note that:

$$r^{256} = 1753^{256} \equiv -1 \mod q$$

We hence consider the polynomial equation $x^{512} - 1 = 0$. It follows that the solutions to this equation are integer powers of r. Namely: $x = r^l$ for $0 \leq l < 512$. If we factor our original equation:

$$x^{512} - 1 = (x^{256} + 1)(x^{256} - 1) = 0$$

we observe that the even integer solutions, $x = r^{2k}$, correspond to the solutions of $x^{256} - 1 = 0$, for $0 \leq k < 256$. On the other hand, the odd integer solutions, $x = r^{2k+1}$ correspond to the solutions of $x^{256} + 1 = 0$, for $0 \leq k < 256$. Therefore, the polynomial $x^{256} + 1$ splits into linear factors over $\mathbb{Z}_q$:

$$x^{256} + 1 = \prod_{k=0}^{255} (x - r^{2k+1}) \tag{32}$$

and by Theorem 2, we have:

$$R_q = \mathbb{Z}_q[x]/\langle x^{256} + 1 \rangle \cong \prod_{k=0}^{255} \mathbb{Z}_q[x]/\langle x - r^{2k+1} \rangle := \prod_{k=0}^{255} R_q^k \tag{33}$$

As the right hand side of (33) represents the CRT factorization of the original polynomial ring, R_q, our NTT must compute the ring isomorphism $\mathbf{NTT} : R_q \to \prod_k R_q^k$. Indeed, one can easily verify that $\psi = r$ and $\omega = r^2$. In particular, given $N = 256$, $\psi^N = r^{256} \equiv -1 \mod q$ and $\omega^N = (r^2)^{256} \equiv 1 \mod q$. Furthermore, $r^2 = \psi^2 = \omega$. Therefore, given $a[n] \in \mathcal{D}([256]; \mathbb{Z}_q)$, the direct form the Dilithium NTT is:

$$A[k] := \mathbf{NTT}^r\{a[n]\} = \sum_{n=0}^{255} a[n] r^{(2k+1)n} = \sum_{n=0}^{255} a[n] r^n (r^2)^{nk} \ , \ 0 \leq k < 256 \tag{34}$$

where the modulo q's are once again omitted for the sake of brevity. Henceforth, we shall continue this notation, unless otherwise noted.

As expected, we have utilized the negacyclic form of the forward transform, since we have $\Phi(x) = x^N + 1$. For the sake of contextualization, the direct form associated with the backwards transform is:

$$a[n] := \mathbf{NTT}^{-r}\{A[k]\} = 256^{-1} r^{-n} \sum_{k=0}^{255} A[k] (r^2)^{-nk} \ , \ 0 \leq n < 256 \tag{35}$$

where $256^{-1} = 8347681$ is meant to denote the multiplicative inverse of 256 in $\mathbb{Z}_q$. Now, with our direct form equations, we can exploit the Cooley-Tukey algorithm.

5.2 Radix-2 & Radix-4 DIT Cooley-Tukey NTT Algorithms:

While there a variety of different Cooley-Tukey Algorithms available, all act on the principle of division of labor, recursively reducing a larger problem into smaller, sub-problems. Generally, this procedure involves splitting or decimating a length N discrete-time signal into smaller signals at each layer or stage of the algorithm. Decimating the input signal, $a[n]$, corresponds to a Decimation-in-Time (DIT) algorithm, while decimating the output signal, $A[k] := \mathbf{NTT}\{a[n]\}$, corresponds to a Decimation-in-Frequency (DIF) algorithm [16].

For either decimation, if N is an integer power of p, that is, $N = p^d$ for some integer $d > 0$, then recursively decimating the signal of length N into signals of length $N/(p^j)$, for an integer $0 \leq j < d$, corresponds to a Radix-p algorithm [16]. Such an algorithm consists of $\log_p N = d$ stages and requires $O(N \log_p N)$ time. Generally speaking, choosing N such that $p = 2$ is ideal given that powers of 2 are highly composite, enabling effective recursion [16]. This commentary justifies the fact that the R-LWE formulation takes $N = 2^d$, as previously referenced. Since in the case of Dilithium we have $N = 256$, we shall focus on the Radix-2 and Radix-4 DIT algorithms as $256 = 2^8$ or $256 = 4^4$. Indeed, in the Radix-2 case, we require 8 stages while the Radix-4 case requires only 4.

Radix-2 Case: Let $a[n] \in \mathcal{D}([256], \mathbb{Z}_q)$ denote our input discrete-time signal. We hence define $f_0[m], f_1[m] \in \mathcal{D}([128]; \mathbb{Z}_q)$ as follows:

$$\begin{aligned} f_0[m] &:= a[2m] \qquad 0 \leq m < 128 \\ f_1[m] &:= a[2m+1] \end{aligned}$$

More explicitly, we separate the even and odd indexed terms of our original discrete-time signal, assigning to $f_0[m]$ the terms corresponding to the even indices and assigning to $f_1[m]$ the terms corresponding to the odd ones. Thus, if $F_0[k] := \mathbf{NTT}^r\{f_0[m]\}$ and $F_1[k] := \mathbf{NTT}^r\{f_1[m]\}$, then (34) can be written as follows:

$$\begin{aligned} A[k] = \sum_{n=0}^{255} a[n] r^n (r^2)^{nk} &= \sum_{m=0}^{127} f_0[m] r^{2m} (r^2)^{2mk} + \sum_{m=0}^{127} f_1[m] r^{2m+1} (r^2)^{(2m+1)k} \\ &= \sum_{m=0}^{127} f_0[m] (r^2)^m (r^4)^{mk} + r^{2k+1} \sum_{m=0}^{127} f_1[m] (r^2)^m (r^4)^{mk} \\ &= F_0[k] + r^{2k+1} F_1[k] \end{aligned}$$

So $A[k] = F_0[k] + r^{2k+1} F_1[k]$, meaning we have rewritten our original 256-point NTT as a sum of two, 128-point NTT's. Indeed, one can verify that the kernel's associated with these 128-point NTT's are correct, by noting that: $(r^2)^{128} = r^{256} \equiv -1 \mod q$ and $(r^4)^{128} = r^{512} \equiv 1 \mod q$. A subtle point of this summation, however, is that this expression is only valid for $0 \leq k < 128$ since this corresponds to the range of m. To resolve this issue, we use the result of the following proposition:

Proposition 7. *Given the decomposition $A[k] = F_0[k] + r^{2k+1} F_1[k]$, for $0 \leq k < 128$, then $A[k + 128] = F_0[k] - r^{2k+1} F_1[k]$.*

Proof: We have that:

$$\begin{aligned} A[k+128] &= F_0[k+128] + r^{2(k+128)+1} F_1[k+128] \\ A[k+128] &= \sum_{m=0}^{127} f_0[m] (r^2)^m (r^4)^{m(k+128)} + r^{2(k+128)+1} \sum_{m=0}^{127} f_1[m] (r^2)^m (r^4)^{m(k+128)} \end{aligned}$$

In the summations, we find that $(r^4)^{m(k+128)} = (r^4)^{mk+128m} = (r^4)^{mk} r^{512m} = (r^4)^{mk}$, implying that these expressions remain the same. For the coefficient multiplying $F_1[k]$, we have: $r^{2(k+128)+1} = r^{2k+256+1} = r^{256} r^{2k+1} = -r^{2k+1}$, which explains the subtraction of $F_1[k]$ $\square$.

To summarize the above discussion, instead of computing $A[k] \in \mathcal{D}([256]; \mathbb{Z}_q)$, we can instead compute $F_0[k], F_1[k] \in \mathcal{D}([128]; \mathbb{Z}_q)$ as:

$$\begin{aligned} A[k] &= F_0[k] + r^{2k+1} F_1[k] \ , \ 0 \leq k < 128 \\ A[k+128] &= F_0[k] - r^{2k+1} F_1[k] \end{aligned} \tag{36}$$

Of course, the efficacy of Cooley-Tukey lies in the power of recursion: in particular, we can successively apply the above procedure on $F_0[k], F_1[k]$, developing a recursive scheme which will ultimately reduce our problem to 128, 2-point NTT's. At this stage, we can take full advantage of the parallelism afforded by this algorithm.

To motivate this recursion and organize the various stages of our algorithm, we shall refer

to (36) as stage 0 and annotate the associated discrete-time signals with a superscript 0. That is:

$$\begin{aligned} A^0[k] &= F_0^0[k] + r^{2k+1} F_1^0[k] \ , \ 0 \leq k < 128 \\ A^0[k+128] &= F_0^0[k] - r^{2k+1} F_1^0[k] \end{aligned}$$

We further adopt the notation:

$$F_i^0[k] := \mathbf{NTT}^r\{f_i^0[m]\} = \sum_{m=0}^{127} f_i^0[m](r^2)^m(r^4)^{mk} \ , \ 0 \leq i \leq 1$$

as each of the summations will contain identical terms, aside from the input discrete-time signals. Let $A^1[k] := F_i^0[k]$. It follows that:

$$\begin{aligned} A^1[k] = \sum_{m=0}^{127} f_i^0[m](r^2)^m(r^4)^{mk} &= \sum_{m=0}^{63} f_0^1[m](r^2)^{2m}(r^4)^{2mk} + \sum_{m=0}^{63} f_1^1[m](r^2)^{2m+1}(r^4)^{(2m+1)k} \\ &= \sum_{m=0}^{63} f_0^1[m](r^4)^m(r^8)^{mk} + (r^2)^{2k+1} \sum_{m=0}^{63} f_1^1[m](r^4)^m(r^8)^{mk} \\ &= F_0^1[k] + (r^2)^{2k+1} F_1^1[k] \end{aligned}$$

where $f_0^1[m], f_1^1[m] \in \mathcal{D}([64]; \mathbb{Z}_q)$ are defined in the same manner as before: that is, $f_0^1[m]$ represents the even indexed terms of $f_i^0[m]$ and $f_1^1[m]$ the odd indexed ones. Here, we have a slight abuse of notation, as we have repurposed our prior m indices: we shall continue to permit this, as the explicit bounds of the summations provide a concrete reference for the range of our indices.

Given the above derivation and a result nearly identical to Proposition 7, the stage 1 equations can be expressed as follows:

$$\begin{aligned} A^1[k] &= F_0^1[k] + (r^2)^{2k+1} F_1^1[k] \ , \ 0 \leq k < 64 \\ A^1[k+64] &= F_0^1[k] - (r^2)^{2k+1} F_1^1[k] \end{aligned} \tag{37}$$

where, once again:

$$F_i^1[k] := \mathbf{NTT}^r\{f_i^1[m]\} = \sum_{m=0}^{64} f_i^1[m](r^4)^m(r^8)^{mk} \ , \ 0 \leq i \leq 1$$

Proceeding forth, let $0 \leq j < 8$ denote the index of our recursive scheme. In particular, each $0 \leq j < 8 = \log_2 256$ refers to a specific layer or stage of our algorithm. Inductively, we can show that for an arbitrary value of j in this range, the equations associated with stage j are:

$$\begin{aligned} A^j[k] &= F_0^j[k] + r^{2^j(2k+1)} F_1^j[k] \ , \ 0 \leq k < 128/2^j \\ A^j[k+128/2^j] &= F_0^j[k] - r^{2^j(2k+1)} F_1^j[k] \end{aligned} \tag{38}$$

In the context of Cooley-Tukey, the coefficients multiplying $F_1^j[k]$, $\pm r^{2^j(2k+1)}$, are known as twiddle factors. We compute $F_i^j[k]$ as follows:

$$F_i^j[k] := \mathbf{NTT}^r\{f_i^j[m]\} = \sum_{m=0}^{\frac{128}{2^j}-1} f_i^j[m](r^2)^{2^j m}(r^4)^{2^j mk} := A^{j+1}[k] \ , \ 0 \leq i \leq 1 \tag{39}$$

and $f_0^j[m], f_1^j[m] \in \mathcal{D}([128/2^j]; \mathbb{Z}_q)$ are formed in the usual manner.

At the maximum value of $j = 7$, our scheme decomposes the two-point NTT $A^7[k]$ as two, one-point NTTs. Namely:

$$\begin{aligned} A^7[0] &= F_0^7[0] + r^{128}F_1^7[0] \\ A^7[1] &= F_0^7[0] - r^{128}F_1^7[0] \end{aligned} \tag{40}$$

To understand what is meant by $F_0^7[0], F_1^7[0]$ we note that, from (39), $F_0^7[0] = f_0^7[0]$ and $F_1^7[0] = f_1^7[0]$. Now, $f_0^7[0]$ refers to the even index of the 2-point discrete-time signal $f_i^6[m]$. Thus, $f_0^7[0] = f_i^6[0]$. Similarly, $f_1^7[0]$ means the odd index of $f_i^6[m]$ so $f_1^7[0] = f_i^6[1]$. Furthermore, $A^7[0] = F_i^6[0]$ and $A^7[1] = F_i^6[1]$. Thus, (40) can be rewritten as follows:

$$\begin{aligned} F_i^6[0] &= f_i^6[0] + r^{128}f_i^6[1] \\ F_i^6[1] &= f_i^6[0] - r^{128}f_i^6[1] \end{aligned} \tag{41}$$

Indeed, $f_i^6[0], f_i^6[1]$ represents the result of performing an even-odd indexed term separation of our signal, 7 times.

Radix-4 Case: In the case of the Radix-4 algorithm, rather than decimating our signal into smaller signals of length 2, we instead decimate into smaller signals of length 4, with a parity of 4 rather than 2. In particular:

$$\begin{aligned} f_0[m] &:= a[4m] \qquad 0 \leq m < 64 \\ f_1[m] &:= a[4m+1] \\ f_2[m] &:= a[4m+2] \\ f_3[m] &:= a[4m+3] \end{aligned}$$

The NTT in (34) hence becomes:

$$A[k] = \sum_{m=0}^{63} f_0[m]r^{(2k+1)4m} + \sum_{m=0}^{63} f_1[m]r^{(2k+1)(4m+1)} + \sum_{m=0}^{63} f_2[m]r^{(2k+1)(4m+2)} + \sum_{m=0}^{63} f_3[m]r^{(2k+1)(4m+3)}$$

Or more compactly:

$$A[k] = \sum_{i=0}^{3}\sum_{m=0}^{63} f_i[m]r^{(2k+1)(4m+i)} = \sum_{i=0}^{3} r^{(2k+1)i}\sum_{m=0}^{63} f_i[m]r^{(2k+1)4m} = \sum_{i=0}^{3} r^{(2k+1)i}F_i[k]$$

where $F_i[k] := \mathbf{NTT}^r\{f_i[k]\}$. Indeed, this formulations holds for $0 \leq m < 64$. We must therefore evaluate $A[k+64], A[k+128], A[k+192]$ or $A[k+64i']$ for $i' = 1, 2, 3$. Indeed:

$$
\begin{aligned}
A[k+64i'] &= \sum_{i=0}^{3} r^{(2(k+64i')+1)i} \sum_{m=0}^{63} f_i[m] r^{(2(k+64i')+1)4m} \\
&= \sum_{i=0}^{3} r^{(2k+1)i} r^{128i'i} \sum_{m=0}^{63} f_i[m] r^{(2k+1)4m} r^{512mi'} \\
&= \sum_{i=0}^{3} r^{(2k+1)i} r^{128i'i} \sum_{m=0}^{63} f_i[m] r^{(2k+1)4m} = \sum_{i=0}^{3} r^{(2k+1)i} r^{128i'i} F_i[k]
\end{aligned}
$$

More explicitly, we have the set of four equations:

$$
\begin{aligned}
A[k] &= F_0[k] &&+r^{(2k+1)}F_1[k] + r^{2(2k+1)}F_2[k] &&+r^{3(2k+1)}F_3[k] \\
A[k+64] &= F_0[k] &&+r^{128}r^{(2k+1)}F_1[k] + r^{256}r^{(2k+1)}F_2[k] &&+r^{384}r^{(2k+1)}F_3[k] \\
A[k+128] &= F_0[k] &&+r^{256}r^{(2k+1)}F_1[k] + r^{512}r^{(2k+1)}F_2[k] &&+r^{768}r^{(2k+1)}F_3[k] \\
A[k+192] &= F_0[k] &&+r^{384}r^{(2k+1)}F_1[k] + r^{768}r^{(2k+1)}F_2[k] &&+r^{1152}r^{(2k+1)}F_3[k]
\end{aligned}
$$

which reduces to:

$$
\begin{aligned}
A[k] &= F_0[k] &&+r^{(2k+1)}F_1[k] + r^{2(2k+1)}F_2[k] &&+r^{3(2k+1)}F_3[k] \\
A[k+64] &= F_0[k] &&+r^{128}r^{(2k+1)}F_1[k] - r^{2(2k+1)}F_2[k] &&-r^{128}r^{3(2k+1)}F_3[k] \\
A[k+128] &= F_0[k] &&-r^{(2k+1)}F_1[k] + r^{2(2k+1)}F_2[k] &&-r^{3(2k+1)}F_3[k] \\
A[k+192] &= F_0[k] &&-r^{128}r^{(2k+1)}F_1[k] - r^{2(2k+1)}F_2[k] &&+r^{128}r^{3(2k+1)}F_3[k]
\end{aligned}
$$

As these equations correspond to the 0th stage of our scheme, we hence annotate with $A^0[k+64i'], F_i^0[k]$. Given that we expect $\log_4 256 = 4$ stages we seek the equations associated with stage j for $0 \leq j < 4$. Similar to the Radix-2 case, it can be shown that:

$$
\begin{aligned}
A^j[k] &= \sum_{i=0}^{3} \sum_{m=0}^{64/4^j-1} f_i^j[m] r^{(2k+1)(4^{j+1}m+4^j i)} \\
A^j[k] &= \sum_{i=0}^{3} r^{4^j(2k+1)i} \sum_{m=0}^{64/4^j-1} f_i^j[m] r^{(2k+1)4^{j+1}m} = \sum_{i=0}^{3} r^{4^j(2k+1)i} F_i^j[k] \ , \ 0 \leq j < 4
\end{aligned}
$$

where $A_i^j[k] = F_i^{j-1}[k]$. Thus, at $j = 3$, we have:

$$
F_i^2[k] = \sum_{i=0}^{3} r^{64(2k+1)i} F_i^3[k] \ , \ F^3[k] = f_i^2[k] \ , \ 0 \leq k < 4
$$

Hence:

$$
\begin{aligned}
F_i^2[0] &= f_i^2[0] &&+r^{64}f_i^2[1] + r^{128}f_i^2[2] &&+r^{192}f_i^2[3] \\
F_i^2[1] &= f_i^2[0] &&+r^{192}f_i^2[1] + r^{384}f_i^2[2] &&+r^{576}f_i^2[3] \\
F_i^2[2] &= f_i^2[0] &&+r^{320}f_i^2[1] + r^{640}f_i^2[2] &&+r^{960}f_i^2[3] \\
F_i^2[2] &= f_i^2[0] &&+r^{448}f_i^2[1] + r^{896}f_i^2[2] &&+r^{1344}f_i^2[3]
\end{aligned}
$$

which reduces to:

$$\begin{aligned} F_i^2[0] &= f_i^2[0] \quad + r^{64} f_i^2[1] + r^{128} f_i^2[2] \quad + r^{192} f_i^2[3] \\ F_i^2[1] &= f_i^2[0] \quad + r^{192} f_i^2[1] - r^{128} f_i^2[2] \quad + r^{64} f_i^2[3] \\ F_i^2[2] &= f_i^2[0] \quad - r^{64} f_i^2[1] + r^{128} f_i^2[2] \quad - r^{192} f_i^2[3] \\ F_i^2[2] &= f_i^2[0] \quad - r^{192} f_i^2[1] - r^{128} f_i^2[2] \quad - r^{64} f_i^2[3] \end{aligned} \tag{42}$$

5.3 Radix-2 & Radix-4 Cooley-Tukey Signal Flow Diagrams

While our derivations above give a complete description of our recursive scheme, working with them can be rather tedious and cumbersome. By using signal-flow diagrams, we can better convey this recursion and render the (fully-parallel) system architecture more intuitive.

Radix-2 Case: In the case of the Radix-2 algorithm, we begin with the stage 7 equations, given in (41). Here, the corresponding signal flow diagram is shown in Figure 3. Note that our recursive scheme utilizes 128 of these 2-point computational units. In the context of the Cooley-Tukey algorithm, we refer to such a computational unit as a Butterfly unit. To clarify this diagram, the inputs $f_i^6[0], f_i^6[1]$ refer to either the even indexed terms of the

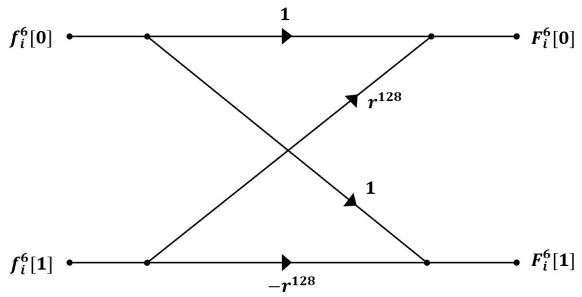

Figure 3: Stage 7, 2-point Radix-2 Cooley-Tukey NTT "Butterfly" Unit.

4-point signal $f_i^5[m]$, or to the odd indexed terms of this signal, depending on weather $i = 0$ or $i = 1$, respectively. For the stage 6 equations, we recall that:

$$\begin{aligned} F_i^5[k] &= F_0^6[k] + r^{64(2k+1)} F_1^6[k] \ , \ 0 \leq k < 2 \\ F_i^5[k+2] &= F_0^6[k] - r^{64(2k+1)} F_1^6[k] \end{aligned}$$

Hence the 4-point NTT $F_i^5[k]$ can be reconstructed as follows:

$$\begin{aligned} F_i^5[0] &= F_0^6[0] + r^{64} F_1^6[0] \\ F_i^5[1] &= F_0^6[1] + r^{192} F_1^6[1] \\ F_i^5[2] &= F_0^6[0] - r^{64} F_1^6[0] \\ F_i^5[3] &= F_0^6[1] - r^{192} F_1^6[1] \end{aligned}$$

We capture this in Figure 4. For the next stage, we have the equations:

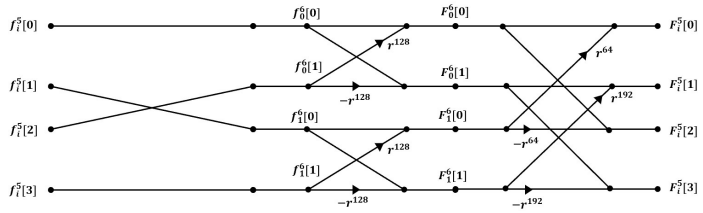

Figure 4: Stage 6, 4-point Radix-2 Cooley-Tukey NTT Unit.

$$F_i^4[k] = F_0^5[k] + r^{32(2k+1)} F_1^5[k] \ , \ 0 \leq k < 4$$
$$F_i^4[k+2] = F_0^5[k] - r^{32(2k+1)} F_1^5[k]$$

which, upon a twiddle factor calculation, gives the diagram shown in Figure 5. Because of

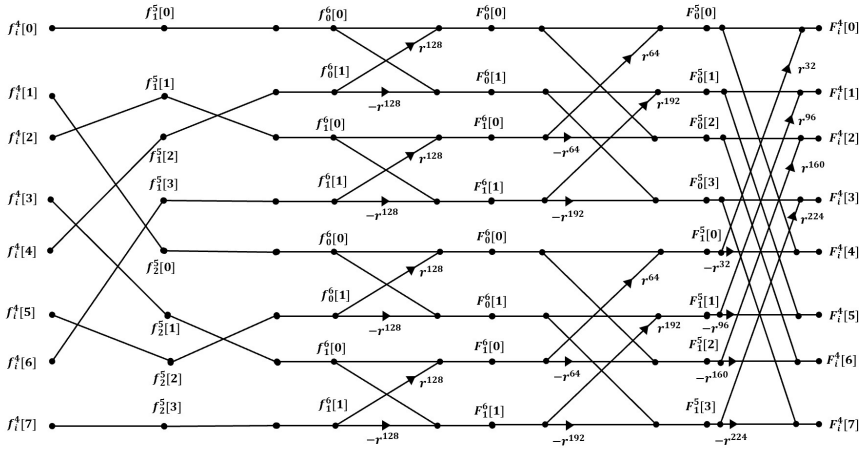

Figure 5: Stage 5, 8-point Radix-2 Cooley-Tukey NTT Unit.

the increasing complexity of these Cooley-Tukey NTT units, we shall not proceed further than this stage in the Radix-2 case. Nevertheless, having gone through the signal-flow diagrams for some of the smaller stages, we now have a better sense of how the different stages of our recursive scheme fit together.

Radix-4 Case: In the case of the Radix-4, the equations given in (42) describe the most basic computational units. Figure 6 illustrates these stage 3, 4-point units. Indeed, the Radix-4 algorithm requires $64 = 256/4$ of such 4-point units. At stage 2, we combine 4 of these 4-point units and perform additional arithmetic so as to partially reconstruct $F_i^1[k]$. We illustrate this stage in Figure 7. As expected, our Radix-4 algorithm utilizes $16 = 256/16$ of these 16-point units.

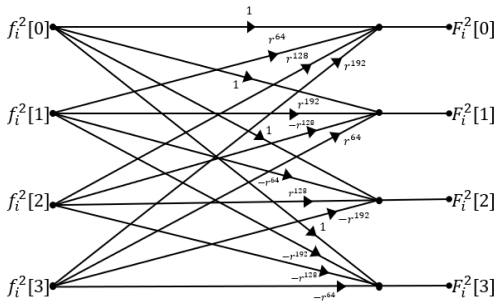

Figure 6: Stage 3, 4-point Radix-4 Cooley-Tukey NTT Unit.

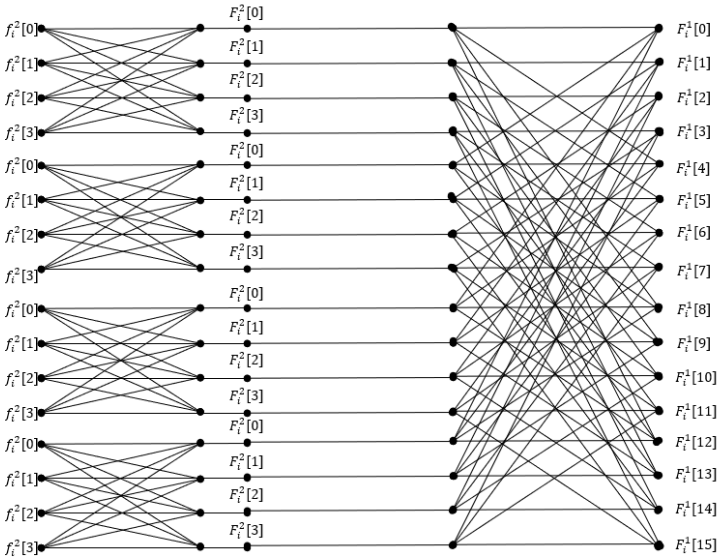

Figure 7: Stage 2, 16-point Radix-4 Cooley-Tukey NTT Unit. Note that the twiddle factors have been removed for ease of illustration.

5.4 Bit Reversals: Radix-2

In the case of the Radix-2 algorithm, by merely inspecting the 8-point Cooley-Tukey NTT unit shown in Figure 5, it is clear that there is a better input order we can utilize. In particular, throughout the entire input stage, none of our signals are scaled or modified. Rather, we have a series of long-winded connections, which seemingly detract from an otherwise more natural input ordering.

To see this more natural order, consider the following table, which displays the discrete-time signal index assuming our input stage is bypassed, referred to as the bypassed index, preceded by our original indexing. We displays both indices in both decimal (d) and binary (b) form. By inspection of this table, we see that the bypassed index reflects a sort of bit reversal

Original Index (d):	Original Index (b):	Bypassed Index (d):	Bypassed Index (b):
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

Table 2: Original indexing associated with the input(s) to our 8-point Cooley-Tukey NTT unit, in binary and decimal format, as well as the bypassed indexing, assuming we bypass the input stage, further in binary and decimal format.

of the original index. To make this precise, suppose we have $d \in \mathbb{Z}^{\geq 0}$. Recall, the decimal representation of d is simply $d = \sum_{i \geq 0} d_i 10^i$ where $d_i \in \{0, 1, ..., 9\}$ are the decimal digits of d. The (unsigned) binary representation of d is hence $d = \sum_{i \geq 0} b_i 2^i$ where $b_i \in \{0, 1\}$ are the binary digits (bits) of d. If the binary representation of b is M bits long, then the bit reversal of d is the function $\mathbf{bitRev} : \mathbb{Z}^{\geq 0} \to \mathbb{Z}^{\geq 0}$ defined by $\sum_{i \geq 0} b_i 2^i \mapsto \sum_{i \geq 0} b_{M-i} 2^i$. Note that $d = \mathbf{bitRev}(\mathbf{bitRev}(d))$ for all d of exactly M bits long: we therefore hold our bit length fixed and zero-pad as necessary, as suggested in the above table. We further note that $d = \mathbf{bitRev}(d)$ if and only if the M bit representation of d is symmetric.

Thus, for our Radix-2 algorithm, we develop an auxiliary algorithm, **bitRevSort**(), which will sort our input signal $a[n] \in \mathcal{D}([256]; \mathbb{Z}_q)$. This algorithm should iterate through each of the indices $n = 0, 1, ..., 255$, and swap $a[n]$ with $a[\mathbf{bitRev}(n)]$ if the 8-bit representation of n is asymmetric and n is less than its bit reversal. This condition is added to guarantee an element is swapped only once. Indeed, our **swap**() function, shown in Algorithm 3, can

Algorithm 2 bitRevSort() *Algorithm*

Require: $a[n] \in \mathcal{D}([256]; \mathbb{Z}_q)$
  for $n = 0;\ n < 256;\ n = n + 1$ **do**
    if bitRev$(n) \neq n$ **and** $n <$ **bitRev**(n) **then**
      swap$(n,$ **bitRev**$(n))$
    end if
  end for

be simply implemented by using a single temporary variable, t.

Algorithm 3 swap(n_0, m_0) *Algorithm*

Require: $a[n] \in \mathcal{D}([256]; \mathbb{Z}_q)$, $0 \leq n_0, m_0 \leq 256$, $n_0 \neq m_0$
  $t \in \mathbb{Z}_q \leftarrow a[n_0]$
  $a[n_0] \leftarrow a[m_0]$
  $a[m_0] \leftarrow t$

5.5 Barrett Reductions:

Polynomial arithmetic aside, all of our integral arithmetic should be performed over the finite field $\mathbb{Z}_q$. Thus, every integral arithmetic operation must map the resulting $a' \in \mathbb{Z}$ to some $a \in \mathbb{Z}_q$ wherein $a \equiv a' \mod q$. In other words, we must perform modular reduction on $a' \in \mathbb{Z}$ so as to maintain finite field arithmetic.

While one could ascertain $a \in \mathbb{Z}_q$ via a fast-division algorithm, a more elegant approach involves the Barret Reduction technique. To enable this technique, we fix a modulus $q \in \mathbb{Z}^{\geq 3}$ [18]: in the case of Dilithium $q = 2^{23} - 2^{13} + 1$ is constant throughout. We hence choose a $k \in \mathbb{Z}^{>1}$ such that $2^k > q$. Note that the smallest choice of k is $\lceil \log_2 q \rceil$ [18]. Thus, given our particular q associated with Dilithium, we choose:

$$k := \lceil \log_2 (2^{23} - 2^{13} + 1) \rceil = 23$$

Indeed, $2^{23} > 2^{23} - 2^{13} + 1$. Next, we compute the factor $\tilde{r} := \lfloor 4^k / q \rfloor$ [18]. Four our chosen k, we have:

$$\tilde{r} := \lfloor \frac{4^{23}}{2^{23} - 2^{13} + 1} \rfloor = 8396807$$

Finally, we define the following quantity, t [18]:

$$t := a' - \lfloor \frac{a'\tilde{r}}{4^k} \rfloor q \tag{43}$$

If $t < q$ we are done. Otherwise, we perform $t - q \mapsto t$. In either case, it can be shown that $a := t \equiv a' \mod q$. A proof of this can be found in [18].

One important feature of (43) is the fact that the floored fractional value can be computed quickly by bit shifting since $\lfloor a'\tilde{r}/4^k \rfloor = \lfloor a'\tilde{r}/2^{2k} \rfloor$. Indeed, this corresponds to right shifting the bits associated with $a'\tilde{r} = 8396807 \cdot a'$ by $2k = 46$ places. Thus, according to 43, we can compute the modular reduction of a' in two multiplications and, at most, two subtractions. We conclude by summarizing our Dilithium Barret reduction algorithm:

Algorithm 4 *Dilithium Barret Reduction Algorithm*

Require: $a' \in \mathbb{Z}^{\geq q}$
  $t \leftarrow a' - [(8396807 \cdot a') >> 46] \cdot q$
  if $t \geq q$ **then**
    $a \leftarrow (t - q)$
  else
    $a \leftarrow t$
  end if
  return a

6 FPGA Implementation:

In this section, we discuss an FPGA implementation of the Dilithium NTT at at the Register-Transfer Level (RTL). Our design utilizes the Radix-2 version of the DIT Cooley-Tukey algorithm, described at length in the previous section. Our motivation for implementing the Radix-2 version of this algorithm rather than the Radix-4 version is twofold. First, the most basic computational units involved in this algorithm, the 2-point or 2×2 Cooley-Tukey butterflies, have a more straightforward implementation. Second, in the Radix-2 algorithm, one can perform a simple bit reversal sort beforehand, whereas the Radix-4 algorithm requires additional work to establish the necessary input order. Note that our entire project has been made open-source, as we include links to the GitHub repository's containing our design files: such links can be found on the Contents page.

Our RTL design consists of a fully-parallel architecture. That is, at any given stage in our algorithm, $j = 0, 1, ..., 7$, all of the associated butterfly calculations occur in parallel. This contrasts with an iterative NTT approaches in which one only calculates one section of a given stage at any time. Such a fully-parallel architecture leverages optimal speed and throughput, but requires an immense number of FPGA resources and power.

The fundamental hardware module housing this architecture is a consolidated NTT-core, which performs the transform on $3N$ Byte or 7.68 kB blocks of data at a given time. In the context of an FPGA implementation, it is difficult to operate on actual data blocks of this size given potential Input/Output (IO) limitations. Our FPGA implementation circumvents this by multiplexing the input as well as the output data. In other words, our NTT core requires peripheral hardware support for such multiplexing. Furthermore, our design requires additional peripheral hardware support for interfacing with FPGA block RAM (BRAM), if desired. These factors suggest that our fully-parallel design is better suited for dedicated parallel-hardware, such as a Single-Instruction Multiple-Data (SIMD) processor or certain Graphics Processing Units (GPUs) which demand optimal speed. Nonetheless, our FPGA implementation can be regarded as a conceptual starting point for a more efficient fully-parallel implementation.

In Section 6.1, we discuss this architecture in greater detail. In Sections 6.2 and 6.3, we describe the RTL modules associated with our Radix-2 design. Then, in Section 6.4, using the Verilog Hardware Description Language (HDL), we give an overview of the RTL to HDL translation and make use of python scripting in assisting with this task.

6.1 System Architecture:

To reiterate, the fundamental system architecture consists of a consolidated NTT core, `CT_radix_2_ntt_core`. Using the parallelism of the Radix-2 DIT Cooley-Tukey algorithm, this module performs the full 256-point forward transform on the input port data and writes directly to the output port. `CT_radix_2_ntt_core` hence consists of 256 input ports and 256 output ports, in addition to a generic clock signal. Since the largest field element, $q - 1 = 8380416$, has a 24-bit unsigned representation, `0x7FE000`, each input is a 24-bit bus

and similarly each output is a 24-bit bus. This corresponds to $2 \cdot 7.68 = 15.36$ kB of IO, excluding the aforementioned clock signal. In this light, our NTT core is a Multiple-Input, Multiple-Output (MIMO) hardware module.

Within `CT_radix_2_ntt_core`, we have various stages of fixed-architecture butterfly modules, labelled `ntt_butterfly_nxn`, corresponding to each of the different stages of the Cooley-Tukey NTT algorithm. Recall that for a Radix-r Cooley-Tukey algorithm there are $d = \log_p N = 8$ different stages. Thus, $n = 2, 4, ..., 256$ in the case of the Radix-2 algorithm. Figure 8 gives a partial block diagrams of our NTT-core, showing the inter-connectivity of various $n \times n$ butterfly modules. That is, butterfly modules with n 24-bit inputs and n 24-bit outputs. While one can readily observe from Figure 8 that our NTT core has

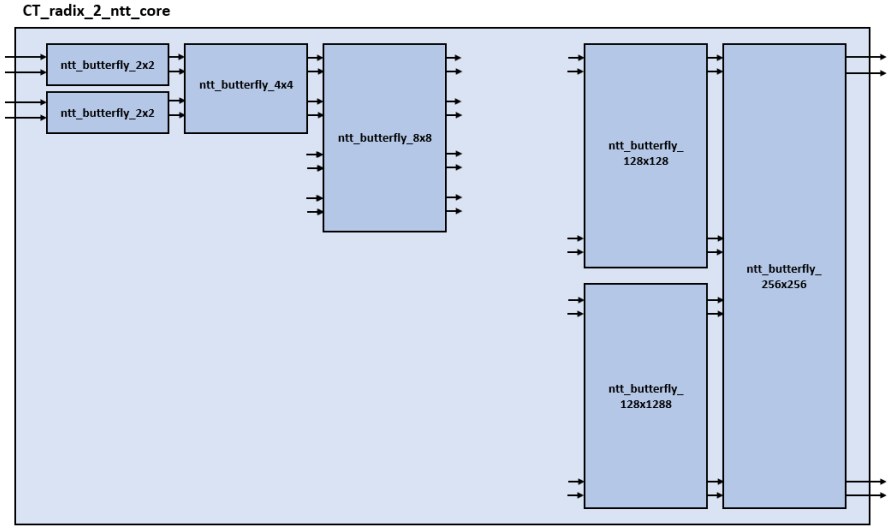

Figure 8: NTT Core showing the inter-connectivity of various fixed-architecture butterfly modules.

the potential for handling 7.68 kB of input data at a time, this is difficult to achieve on an FPGA given potential IO limitations, as previously mentioned. Thus, our FPGA implementation includes an input demultiplexer of dimensions $1 : N = 1 : 256$, `ntt_core_demuxi`, which maps a 24-bit input value to the appropriate input port of the NTT core, given an 8-bit index. Similarly, we include an output multiplexer of dimensions $N : 1 = 256 : 1$, `ntt_core_mux`, which maps the 24-bit NTT-core output associated with the specified 8-bit index to a 24-bit output port. This effectively renders our system a Single-Input, Single-Output (SISO) hardware module, capable of handing input streams of data with additional logic.

For the SISO system described, one may like to interface with dedicated FPGA BRAM. This can be achieve by adding an additional memory manager module, `mem_mananger`, which reads and writes to a 7.68 kB NTT BRAM module, `ntt_BRAM`. The `mem_manager` module additionally interfaces with the multiplexer modules described above to read and

write from the NTT core. Figure 9 gives a simplified block diagram of this FPGA application system. We further include a dedicated IO stream for the NTT data, consisting of the 24-bit input and output buses labelled `ntt_data_path_in` and `ntt_data_path_out`: we assume the user enters the discrete-time signal inputs in bit reversed order. Note that

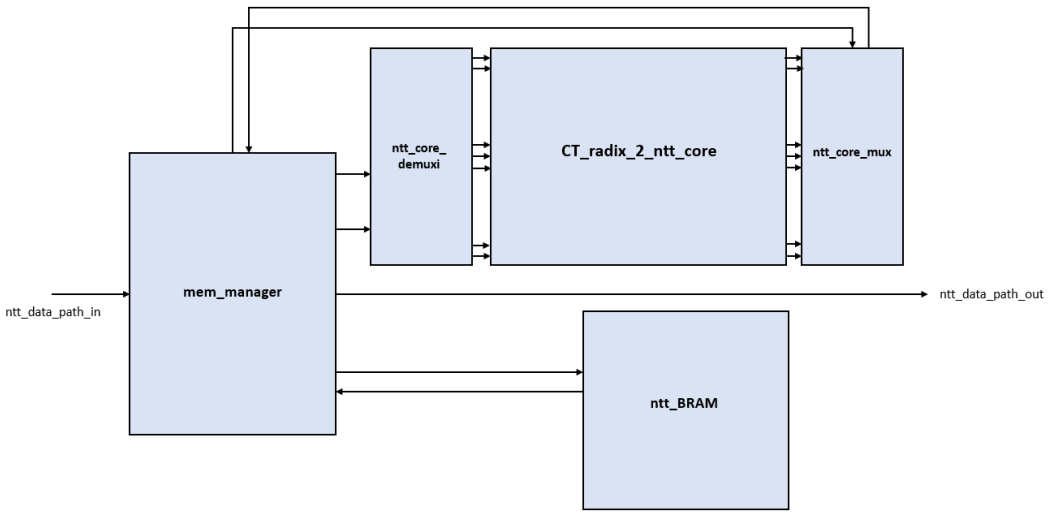

Figure 9: Simplified block diagram of FPGA application system, consisting of memory management module and dedicated FPGA BRAM.

Figure 9 only represents an application system, as we do not account for `mem_mananger` nor `ntt_BRAM` in our design. Indeed, our design solely accounts for our multiplexer modules togethter with the NTT core itself. Given the straightforward implementation of our multiplexer modules, we shall only describe the modules contained inside our NTT core in greater detail. Indeed, this amounts to our barret reduction module, `barret`, as well as $n \times n$ butterfly modules, `ntt_butterfly_nxn`. Table 3 gives the individual quantities of the different butterfly modules contained in `CT_radix_2_ntt_core`.

Butterfly Module:	Quantity:
`ntt_butterfly_2x2`	128
`ntt_butterfly_4x4`	64
`ntt_butterfly_8x8`	32
`ntt_butterfly_16x16`	16
`ntt_butterfly_32x32`	8
`ntt_butterfly_64x64`	4
`ntt_butterfly_128x128`	2
`ntt_butterfly_256x256`	1

Table 3: Quantity of different butterfly modules in our NTT core.

6.2 Description of NTT-Core RTL Modules:

Before discussing our RTL modules, we assume than any integer arithmetic operation requires exactly one clock cycle. Indeed, in all of our block diagrams, we shall generally exclude clock signals for ease of illustration.

`barret`: Our Barrett reduction modules performs reduction modulo q using the Barret reduction algorithm described in Section 5.5. The input to this module, `a_in` corresponds to the element $a \in \mathbb{Z}$ which we must reduce modulo q. The output wire, `a_out`, hence corresponds to the finite-field element $a' \in \mathbb{Z}_q$ satisfying $a \equiv a' \mod q$. As suggested in Algorithm 4, our RTL design utilizes 2 instantiated multipliers, 2 instantiated subtractors, 1 combinational right-shift unit, 1 combinational comparator element, and a 2:1 multiplexer (MUX). Our logic is primarily asynchronous, with only the arithmetic units requiring a clock signal. Figure 5 illustrates the block diagram associated with our RTL module. As seen in

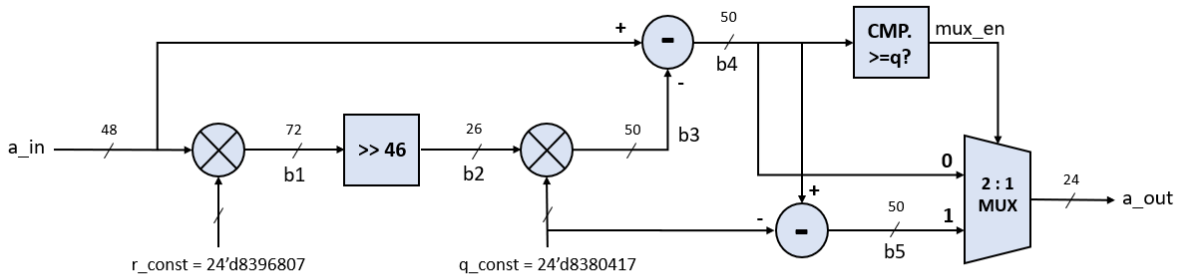

Figure 10: Block diagram of Barrett reduction module. Note that only the arithmetic units require a clock signal.

Figure 10, the primary data path consists of wires `b1, b2, b3, b4`. `b4` represents the first potential output of the Barrett reduction algorithm, $t := a' - [(8396807 \cdot a') >> 46] \cdot q$. Note that both multipliers along this data path are constant-coefficient multipliers. In the case when $t \geq q$, however, we must output $t - q$. Thus, `b4` is compared to q such that the logical output of this comparison becomes the input to our MUX. For $t < q$, the MUX output is 0 which corresponds to t; `b4` is thus mapped to the output wire, `a_out`. For $t \geq q$, the MUX output is 1 which corresponds to $t - q$; the wire `b5` which includes a branch of `b4` with `q_const` subtracted is hence mapped to `a_out`.

Since the delay through each of our multipliers and subtractors is uniform (1 clock cycle), we expect the delay to be around 4 clock cycles for worst-case performance, from input to output. In the best-case performance, we do not require this extra subtraction rendering the delay to 3 clock cycles from input to output.

`ntt_butterfly_2x2.v`: Our most basic Cooley-Tukey NTT computational units, these modules compute the stage 7, 2×2 equations, described in 41. The input wires `fi_0, fi_1` correspond to $f_i^6[0], f_i^6[1]$, respectively. Similarly, the output wires `Fi_0, Fi_1` correspond to $F_i^6[0], F_i^6[1]$, respectively. The RTL design closely resembles the signal flow diagram outlined in Figure 3.

To mitigate delay, our 2×2 butterfly modules only perform modular reduction once on each expression $f_i^6[0] \pm r^{128} f_i^6[1]$. This contrasts with performing the reduction on each $r^{128} f_i^6[1]$ and thereafter performing an additional subtraction if the addition of $f_i^6[0]$ exceeds the maximum field element. That is, $f_i^6[0] + r^{128} f_i^6[1] > q - 1$. This is done to avoid the presence of an additional arithmetic unit, which requires one extra clock cycle. Instead, we simply add an extra bit to each of the data paths in `barret`. 11 displays our 2×2 butterfly design: With respect to worst-case performance, we expect a delay of 4 clock cycles

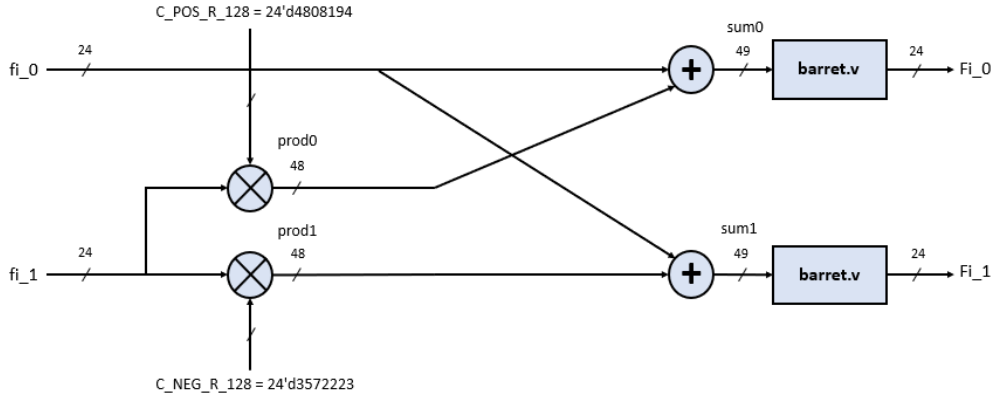

Figure 11: Block diagram of 2×2 Cooley-Tukey NTT butterfly module, consisting of 2 multipliers, 2 adders, and 2 Barret reduction modules

from each of the Barret reduction modules. Thus, because the computation of `Fi_0` occurs in parallel to the computation of `Fi_1`, we expect a delay of 4 clock cycles + 2 additional clock cycles from each of the multiplier and adder totaling to 6 clock cycles from input to output. With respect to best-case performance, our Barret reduction modules incurs only 3 clock cycles, such that the delay from input to output becomes $3 + 2 = 5$ clock cycles.

`ntt_butterfly_nxn.v`: Since each of our Radix-2 Cooley-Tukey butterfly modules utilize a fixed architecture, we require dedicated $n \times n$ butterfly modules for $n = 2, 4, ..., 256$, corresponding to each stage of the Radix-2 Cooley-Tukey NTT algorithm. With respect to design, any given $n \times n$ butterfly module essentially consists of $n/2, 2 \times 2$ butterfly modules. Figure 12 illustrates this for $n = 4$, shown on the next page. Therefore, given that the delay through a 2×2 butterfly module is 5 clock cycles at best and 6 clock cycles at worst, we expect the delay through any given $n \times n$ butterfly to also be 5 clock cycles at best and 6

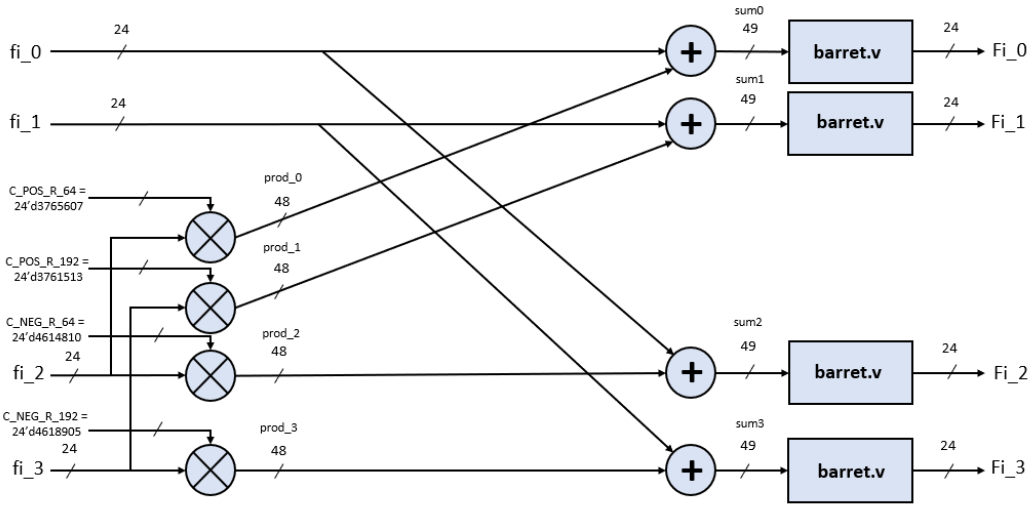

Figure 12: Block diagram of 4×4 Radix-2 Cooley-Tukey butterfly module. One can see that such a 4×4 butterfly module is nothing more than 2 parallel, 2×2 butterfly modules.

clock cycles at worst. This hence renders the net delay across our NTT core to $5 \log_2 N = 40$ clock cycles at best and $6 \log_2 N = 48$ clock cycles at worst.

One can further observe that for any given $n \times n$ butterfly, the number of multiplier, adder and Barrett module instantiations is simply n. This allows us to calculate the number of multipliers, adders, and Barret module instantiations in our NTT core: we display these results in table 4, shown below. Note that we exclude the multipliers inside our Barret

Instantiation:	Quantity:
Multiplier	2048
Adder	2048
Barret	2048

Table 4: Quantity of multipliers, adders, and Barret instantiations in our NTT core.

reduction modules. To ascertain the total number of multipliers in our design, we note that each Barret module incurs an additional two multipliers which gives $2048 + 2(2048) = 6144$ total multipliers. Similarly, each Barret module has two subtractors which implies $2(2048) = 4096$ total subtractors.

6.3 Overview of HDL and Python Utility Scripts:

We hence translate our designs into the Verilog Hardware Description Language (HDL), writing Verilog RTL modules, so as to perform logical synthesis. We wrote our Verilog code in the Vivado environment, with our Vivado project targeting an Artix-7 family Evaluation Platform. In particular, we targeted the Artix-7 AC701 Evaluation Platform, with associated FPGA part xc7a200tfbg676-2.

To implement basic arithemtic, we utilized Xilinx LogiCORE Multiplier IPs as well as Xilinx LogiCORE Adder/Subtracter IPs . In both IPs, one specifies the widths of the input ports `A,B`. In the case of the Multiplier, the IP hence dictates the bit width of the output, `C`, while one further specifies the width of the output in the case of the Adder/Subtracter. Indeed, for the Adder/Subtracter, one must further select which of the two operations one would like the IP to perform. Upon specifiying bit-width(s), in both IPs one can select to implement using either LUTs or DSP slice logic. In the case of DSP slices, the maximum bit width is 48 bits. Thus, in all our $n \times n$ butterfly modules, we implement our twiddle factor calculations using available DSP slices.

For `barret.v`, our Verilog code primarily consists of instantiating the dedicated arithemtic IPs. The only additional logic requires right shifting `b1` by 46 bits, setting the MUX enble based on weather `b4` exceeds the maximum field element, and inferring the MUX itself. In the case of right-shifting the bits of `b1` and setting the MUX enable, the following combinational logic suffices:

```
assign b2     = (b1 >> k_const);
assign mux_en = (b4 >= q_const) ? 1'b1 : 1'b0
```

Additionally, we can infer our 2:1 MUX as follows:

```
always @ (*) begin
    case(mux_en)
        1'b1: a_out = b5[23:0];
        1'b0: a_out = b4[23:0];
    endcase
end
```

Recall that `b5` represents the output which corresponds to the additional subtraction with q whereas `b4` is the original output from the primary data path. The complete Verilog code can be found in the Appendices Section, under Subsection 10.1.

In the case of `ntt_butterfly_2x2.v`, our implementation requires nothing more than arithmetic IP and Barret module instantiation, as there is no additional combinational or sequential logic. In general, such instantiations fall into three distinct blocks of code. The first block instantiates the multiplier IPs, corresponding to the initial multiplication(s) of `fi_1` with the twiddle factor(s) `C_POS_R_128` and `C_NEG_R_128`. The next block consists of adder IPs corresponding to addition(s) of `fi_0`. Lastly, we instantiate two pairs of barret modules, for performing modular reduction. Indeed, the complete Verilog code for this module can be found in the Appendices Section, in Subsection 10.2.

Since any `ntt_butterfly_nxn` for $n = 2, 4, ..., 256$ is just a collection of $n/2, 2 \times 2$ butterfly modules, these distinct blocks of code generalize to all subsequent butterfly modules. We hence wrote python code which exploits the formulaic nature of such butterfly modules to write the correct Verilog code to some output text file. This python code, `dilithium_ntt_butterfly_gen.py` consists of a function `butterfly_gen(n)` wherein `n`

corresponds to the dimensions of the associated butterfly. The function first opens a generic text file and writes the module header, with the necessary IO ports. Upon calculating and parameterizing the associated twiddle factors, it then instantiates the three distinct blocks of code. Namely, the multipliers, adders, and then Barret reduction modules. We further include an option to add additional `start, done` signals to each butterfly module, along with a 3-bit counter which asserts this `done` signal after 6 clock cycles has passed from the rising edge of the `start` signal. Although our actual design does not utilize these signals, these can be helpful in verification or if a different module requires some indication that a given butterfly module has finished its calculation. Regardless, one can then copy the output text file to the .v file titled `ntt_butterfly_nxn.v` for digital synthesis: we include this python code in Subsection 10.3 of the appendices.

7 Performance:

Given our Verilog modules, we verified intended functionality by writing RTL testbenches and thereafter performing behavioral simulation. In particular, we simulated unit tests as well as regression tests of various butterfly modules. Our unit tests consisted of evaluating the functionality of an individual `ntt_butterfly_nxn` module while our regression tests consisted of evaluating a family of different butterfly modules, together connected. In Section 7.1, we describe the testbenches and testbench logic associated with these tests. Then, in Section 7.2, we comment on the behavioral simulation results.

7.1 RTL Verification: Unit & Regression Test

In essence, both our unit and regression tests consist of sets of input test vectors as well as sets of expected output test vectors. In the case of unit tests, test vectors are sequentially mapped to the input port of an instantiated device-under-test (DUT) such that the output port is read and compared to a set of expected output test vectors. We hence display a pass or fail message accordingly. In the case of regression tests, we have virtually the same procedure though instead of mapping inputs to a single instantiated DUT, we have a family of DUTs which model a specific region of our NTT core.

With respect to concrete testbench files, we labelled our unit tests `ntt_butterfly_nxn_unit_tb.v` for $n = 2, 4, ...256$. Intuitively, `ntt_butterfly_nxn_unit_tb.v` verifies the functionality of the associated $n \times n$ butterfly module, placing this module as the testbench DUT. We labelled our regression tests `ntt_butterfly_nxn_reg_tb.v`. With respect to our concrete testbenches, we developed a sufficiently flexible unified framework for both test forms.

In this framework, we specify a test-length, L, which corresponds to the number of sets of inputs to be tested. Recall that a given $n \times n$ butterfly module requires a set of n inputs and maps to set of n outputs. Therefore, for an $n \times n$ test, we have n many 24-bit register-arrays, each of size L, which correspond to input test vectors. In the case of $n = 8$, these are labelled as follows:

```
reg [23:0] test_vec_in_fi_0[0:L-1]
```

```
reg [23:0] test_vec_in_fi_1[0:L-1]
.
.
reg [23:0] test_vec_in_fi_7[0:L-1]
```

On the other hand, we also have $n = 8$ many 24-bit register-arrays which correspond to expected output test vectors, for a particular test index $0 \leq l < L$:

```
reg [23:0] test_vec_exp_out_Fi_0[0:L-1]
reg [23:0] test_vec_exp_out_Fi_1[0:L-1]
.
.
reg [23:0] test_vec_exp_out_Fi_7[0:L-1]
```

We developed a python script to generate random input test-vectors, calculate the expected output test-vectors, and thereafter write to a test-file the initialized registers. In particular, our script first generates n many 24-bit arrays of length L, corresponding to the input test vectors, with each array containing arbitrary elements of the finite field, $\mathbb{Z}_q$. Note that each array is associated with a specific port of the DUT or family of DUTs. We next perform some Cooley-Tukey butterfly calculations depending on weather we specify a unit or regression test, and finally write L many `initial` blocks to an output text file, wherein `test_vec_in_fi_j[l]` and `test_vec_exp_out_Fi_j[l]` have been initialized accordingly. This python script can be found in the appendices section, underneath subsection 10.4.

Aside from test-vector generation, the test bench logic itself is driven by an `always` block which counts clock cycles. The testbench clock is generated by simply toggling a 1-bit register every 5 ns to create a 100 Mhz signal. Indeed, at every number of clock cycles that is a multiple of the number of clock cycles we expect to wait for valid butterfly data, we read the output port of our DUT or family of DUT's and check to see if they match the expected outputs at that particular test index, $0 \leq l < L$. More specifically, for $n = 8$, we logically evaluate expressions of the form:

```
(Fi_0 == test_vec_exp_out_Fi_0[l]) &&
(Fi_1 == test_vec_exp_out_Fi_1[l]) &&
.
.
.
(Fi_7 == test_vec_exp_out_Fi_7[l])
```

If this expression evaluates to true we print a pass message to the TCL console. Conversely, if this expression evaluates to false we print a fail message, along with the expected results and the actual results. We then increment the test index, l, and write the next set of input test vectors to the input port of the DUT(s). In our appendices section underneath subsection 10.5, we include a sample unit test `ntt_butterfly_8x8_unit_tb.v`. Note that `ntt_butterfly_8x8_reg_tb.v` is essentially the same, with the only difference being that this testbench instantiates 4 $\times$ `ntt_butterfly_2x2.v`, 2 $\times$ `ntt_butterfly_4x4.v`, and 1 $\times$ `ntt_butterfly_8x8.v` rather than a single `ntt_butterfly_8x8.v` DUT.

7.2 Behavioral Simulation Results

Having written testbenches to verify the functionality of individual butterfly modules or families of butterfly modules in the form of unit and regression tests, respectively, we performed behavioral simulation in the Vivado environment. As far as intended functionality goes, we simply check the TCL console to determine the results of the test-bench logic. Additionally, however, we can use the waveform viewer to verify the conjectured 5-6 clock cycle delay to receive valid butterfly data.

As an example of a typical output waveform, we examine the result from our 2×2 unit testbench in the waveform viewer. This is shown below in Figure 13: Here, the red cursor

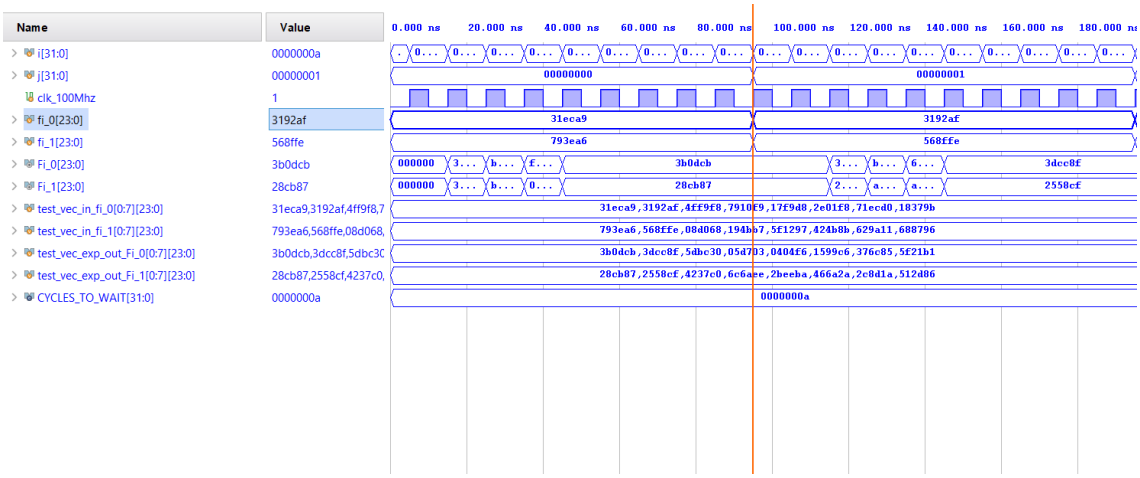

Figure 13: Waveform view from 2 x 2 unit testbench. The red cursor indicates the time at which new data was mapped to the input port(s) of the DUT.

indicates the time at which input test vector data was written to the 24-bit DUT input(s) `fi_0, fi_1`. One can easily see that after 6 clock cycles, the DUT output(s) of the same bit-width, `Fi_0, Fi_1`, do not change value, indicating that the final result from the 2×2 butterfly calculation has been obtained.

Furthermore, in the 4×4 unit test, we observe a similar delay. This is captured in Figure 14. Once again, the red cursor indicates the time at which input test vector data was written

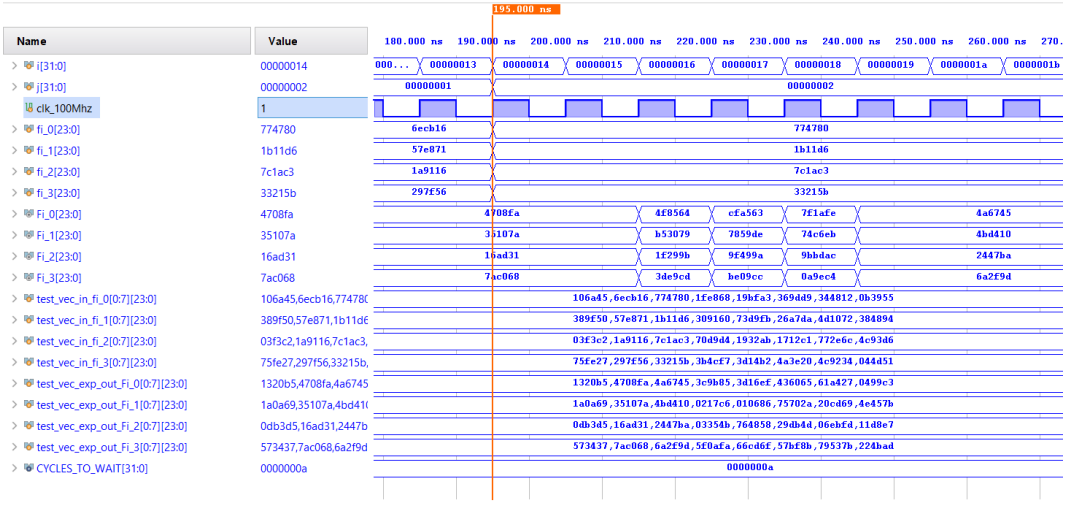

Figure 14: Waveform view from 4 x 4 unit testbench. Once again, the red cursor indicates the time at which data was mapped to input port(s) of the DUT.

to each of the four 24-bit DUT input port(s), `fi_0, fi_1, fi_2, fi_3`. One can readily see that after 6 clock cycles, the DUT output(s) `Fi_0, Fi_1, Fi_2, Fi_3` do not change implying that the device has completed its 4×4 butterfly calculation.

Indeed, in each of our unit and regression test benches conducted, the number of clock cycles required to perform an $n \times n$ butterfly calculation never exceeds 6 clock cycles nor does it settle in less than 5 clock cycles. These measurements support our previous estimate of the delay through our NTT core remaining bounded between $5 \cdot \log_2 N = 40$ and $6 \log_2 N = 48$ clock cycles.

We should point out, however, that these quantities merely reflect the results of our behavioral simulation. Indeed, the physical implementation results may differ, depending on weather or not the associated arithmetic IP indeed requires a single clock cycle, as observed in simulation.

8 Conclusions:

To summarize the content of this paper, we investigated the ring-theoretic foundations of the Number Theoretic Transform (NTT) and exposed the algebraic framework associated with R-LWE and R-LWE related cryptography. With respect to the Cooley-Tukey algorithm, this enabled a reduction from $O(N^2)$ time to $O(N \log N)$ time. We hence proposed a fully-parallel FPGA implementation which sought to exploit the full-parallelism afforded by the Cooley-Tukey algorithm. We concluded by performing behavioral simulation on the associated RTL modules.

Although such a design leverages optimal speed and high-throughput, we noted that on an FPGA it is difficult to achieve this maximum throughput given possible IO constraints. Furthermore, our design requires an immense number of FPGA resources. From a preliminary synthesis, Vivado estimates 79,720 LUT's which corresponds to an almost 60% utilization of LUTs for the targeted FPGA part, xc7a200tfbg676-2. Furthermore, the size of this implementation gave rise to difficulties in simulating large areas of the NTT core.

Further work therefore includes reducing the size of our design to aid the feasibility of our FPGA implementation. This could be achieved by returning to an iterative approach so as to reduce the number of butterfly modules associated with the NTT core. Alternatively, one could investigate fully-parallel hardware implementations on devices other than FPGAs. In either case, we have outlined seemingly efficient individual, butterfly units and a somewhat-optimized modular reduction unit. These items could be of potential use in pursuing either of these two routes.

9 References:

[1] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography." "https://cims.nyu.edu/~regev/papers/qcrypto.pdf", 2009.

[2] O. Regev, "The learning with errors problem." https://cims.nyu.edu/~regev/papers/lwesurvey.pdf, 2010.

[3] A. Blum, A. Kalai, and H. Wasserman, "Noise tolerant learning, the parity problem, and the statistical query model," *Journal of the ACM*, pp. 506–519, 2003.

[4] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," 2013.

[5] D. Balbás, "The hardness of lwe and ring-lwe: A survey." Cryptology ePrint Archive, Paper 2021/1358, 2021. https://eprint.iacr.org/2021/1358.

[6] N. I. of Standards and Technology, "Post quantum cryptography: Round 3 submissions." https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, 2023.

[7] "Crystals: Cryptographic suite for algebraic lattices." https://pq-crystals.org/index.shtml, 2022.

[8] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle, "Crystals-dilithium: Algorithm specifications and supporting documentation (version 3.1)." https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf, 2021.

[9] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," *International Conference on Cryptology and Network Security*, pp. 124–139, 2016.

[10] Z. Liang and Y. Zhao, "Number theoretic transform and its applications in lattice-based cryptosystems: A survey," 2022.

[11] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of crystals-dilithium," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–10, 2021.

[12] S. Ricci, L. Malina, P. Jedlicka, D. Smekal, J. Hajny, P. Cibik, P. Dzurenda, and P. Dobias, "Implementing crystals-dilithium signature scheme on fpgas," *ARES 21: Proceedings of the 16th International Conference on Availability, Reliability, and Security*, pp. 1–11, 2021.

[13] G. Land, P. Sasdrich, and T. Guneysu, "A hard crystal: Implementing dilithium on reconfigurable hardware," *SmartCard Research and Advanced Applications: 20th International Conference, CARDIS 2021, Lubeck, Germany, November 11-12, 2021, Revised Selected Papers*, pp. 210–230, 2021.

[14] S. Shahriari, *Algebra in Action: A Course in Groups, Rings, and Fields*. American Mathematical Society, 2017.

[15] "Chinese remainder theorem." https://www.math.columbia.edu/~khovanov/ma2_fall/files/crt.pdf.

[16] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*. Pearson Education, Inc., 4th ed., 2007.

[17] I. Selesnick, "Properties of the dft." https://eeweb.engineering.nyu.edu/iselesni/EL713/zoom/dftprop.pdf.

[18] "Barret reduction algorithm." https://www.nayuki.io/page/barrett-reduction-algorithm.

10 Appendices:

10.1 barret.v

Verilog HDL implementation of `barret` module.

```
module barret(
    input clk_100Mhz,
    input      [48:0] a_in,
    output reg [23:0] a_out
    );

    parameter q_const = 24'd8380417;
    parameter k_const =  6'd46;

    wire [72:0] b1;
    wire [26:0] b2;
    wire [49:0] b3;
    wire [50:0] b4;
    wire [50:0] b5;
    wire mux_en;

    mult_gen_49_bit mult_gen_49_biti(.CLK(clk_100Mhz), .A(a_in), .P(b1));
    mult_gen_27_bit mult_gen_27_biti(.CLK(clk_100Mhz), .A(b2),   .P(b3));

    c_sub_51_bit c_sub_51_biti0(.A(a_in),     .B({1'd0, b3}),
    .CLK(clk_100Mhz), .CE(1'b1), .S(b4));
    c_sub_51_bit c_sub_51_biti1(.A(b4[48:0]), .B({27'd0, q_const}),
    .CLK(clk_100Mhz), .CE(1'b1), .S(b5));

    assign b2     = (b1 >> k_const);
    assign mux_en = (b4 >= q_const) ? 1'b1 : 1'b0;

    always @ (*) begin
        case(mux_en)
            1'b1: a_out = b5[23:0];
            1'b0: a_out = b4[23:0];
        endcase
    end

endmodule
```

10.2 ntt_butterfly_2x2.v

Verilog HDL implementation of `ntt_butterfly_2x2.v` module.

```
module ntt_butterfly_2x2(
         input clk_100Mhz,
         input [23:0] fi_0,
         input [23:0] fi_1,
         output [23:0] Fi_0,
         output [23:0] Fi_1
```

```
        );

        parameter C_POS_R_128 = 24'd4808194;
        parameter C_NEG_R_128 = 24'd3572223;

        wire [48:0] sum0;
        wire [48:0] sum1;

        wire [47:0] prod0;
        wire [47:0] prod1;

        mult_gen_0 mult_gen_0i(.CLK(clk_100Mhz), .A(fi_1), .B(C_POS_R_128), .P(prod0));
        mult_gen_0 mult_gen_1i(.CLK(clk_100Mhz), .A(fi_1), .B(C_NEG_R_128), .P(prod1));

        c_add_0 c_add_0i(.A(prod0), .B(fi_0), .CLK(clk_100Mhz),  .CE(1'b1), .S(sum0));
        c_add_0 c_add_1i(.A(prod1), .B(fi_0), .CLK(clk_100Mhz),  .CE(1'b1), .S(sum1));

        barret barret_0i(.clk_100Mhz(clk_100Mhz), .a_in(sum0), .a_out(Fi_0));
        barret barret_1i(.clk_100Mhz(clk_100Mhz), .a_in(sum1), .a_out(Fi_1));

endmodule
```

10.3 dilithium_ntt_butterfly_gen.py

Python script which generates Verilog butterfly modules, given $2 \leq n \leq 256$.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Mar 31 14:58:56 2023
@author: bvoc5

Generate n x n butterfly module for CRYSTALS-Dilithium
Radix-2 Cooley-Tukey NTT FPGA Implementation
"""

#Enable cycle counter for start/done signals
clocked_en = 0

def butterfly_gen(n):

    r               = 1753
    q               = 8380417
    exp_factor_list = []

    bit_width      = 24
    num_cycles     = 15
    cycles_width   =  4
    barret_count   =  0

    clk_signal = "clk_100Mhz"
    rst_signal = "rst_n"
```

```
str_signal = "start"
dne_signal = "done"

#Open file with writing priveleges
butterfly_file_name = "ntt_butterfly_"  + str(n) + "x" + str(n)
butterfly_file      = open(butterfly_file_name + ".txt", "w")

#Port declaration
butterfly_file.write("module "   + butterfly_file_name + "(\n")
butterfly_file.write("\t input " + clk_signal + ",\n")

if(clocked_en == 1):
    butterfly_file.write("\t input " + rst_signal + ",\n")
    butterfly_file.write("\t input " + str_signal + ",\n")

#Butterfly input(s)
for i in range(n):
    butterfly_file.write("\t input [" + str(bit_width - 1) + ":0] ")
    butterfly_file.write("fi_" + str(i) + ",\n")

#Butterfly output(s)
for i in range(n):
    butterfly_file.write("\t output [" + str(bit_width - 1) + ":0] ")
    butterfly_file.write("Fi_" + str(i))

    if(i != (n - 1)):
        butterfly_file.write(",\n")
    else:
        if(clocked_en == 0):
            butterfly_file.write("\n \t );\n")
            butterfly_file.write("\n")
        else:
            butterfly_file.write(",\n")

if(clocked_en == 1):
    butterfly_file.write("\t output " + dne_signal + "\n")
    butterfly_file.write("\t );\n")
    butterfly_file.write("\n")

#Twiddle factor calculation
for k in range(int(n/2)):

    exp_factor      =    int((256 / n)*(2*k + 1))
    pos_twid_factor =      pow( r, exp_factor, q)
    neg_twid_factor =  q - pow( r, exp_factor, q)

    butterfly_file.write("\t parameter C_POS_R_" + str(exp_factor) + " = 24'd" +
    str(pos_twid_factor) + ";\n")
    butterfly_file.write("\t parameter C_NEG_R_" + str(exp_factor) + " = 24'd" +
    str(neg_twid_factor) + ";\n")
```

```
    exp_factor_list.append(exp_factor)

#Cycles to wait
if(clocked_en == 1):
    butterfly_file.write("\t parameter C_CYCLES_TO_WAIT = " +
    str(cycles_width) + "'d" + str(num_cycles) + ";\n")

butterfly_file.write("\n")

#Sum wire declaration
for i in range(n):
    butterfly_file.write("\t wire [48:0] ")
    #butterfly_file.write("\t wire [" + str(bit_width) + ":0] ")
    butterfly_file.write("sum" + str(i) + ";\n")

butterfly_file.write("\n")

#Product wire declaration
for i in range(n):
    butterfly_file.write("\t wire [" + str(2*bit_width - 1) + ":0] ")
    butterfly_file.write("prod" + str(i) + ";\n")

butterfly_file.write("\n")

if(clocked_en == 1):
    butterfly_file.write("\t reg [" + str(cycles_width - 1) + ":0] clk_count;\n")
    butterfly_file.write("\n")

#mult_gen instantiations
for i in range(n):

    butterfly_file.write("\t mult_gen_0 mult_gen_" + str(i)     + "i(")
    butterfly_file.write(".CLK("                   + clk_signal + "), ")

    if(i < int(n/2)):
        butterfly_file.write(".A(fi_"      + str(i + int(n/2))       + "), ")
        butterfly_file.write(".B(C_POS_R_" + str(exp_factor_list[i]) + "), ")
    else:
        butterfly_file.write(".A(fi_"      + str(i)
        + "), ")
        #butterfly_file.write(".B(sub"      + str(i - int(n/2))   + "), ")
        butterfly_file.write(".B(C_NEG_R_" + str(exp_factor_list[i - int(n/2)])
        + "), ")

    butterfly_file.write(".P(prod" + str(i) + "));\n")

butterfly_file.write("\n")

#c_add_0 instantiations
for i in range(n):
```

```
    butterfly_file.write("\t c_add_0 c_add_"   + str(i) + "i(")
    butterfly_file.write(".A(prod"             + str(i) + "), ")
    #butterfly_file.write(".A(prod"             + str(i) + "_rdd), ")

    if(i < int(n/2)):
        butterfly_file.write(".B(fi_" + str(i)             + "), ")
    else:
        butterfly_file.write(".B(fi_" + str(i - int(n/2))  + "), ")

    butterfly_file.write(".CLK("      + clk_signal + "),  ")
    butterfly_file.write(".CE(1'b1), ")
    butterfly_file.write(".S(sum"   + str(i)     + "));\n")

butterfly_file.write("\n")

#barret instantiations
for i in range(n):

    butterfly_file.write("\t barret barret_" + str(barret_count) +
    "i(.clk_100Mhz(" + clk_signal + "), ")
    #butterfly_file.write(".rst_n("     + rst_signal + "), ")
    #butterfly_file.write(".a_in(sum"   + str(i)     + "_pdd), ")
    butterfly_file.write(".a_in(sum"   + str(i)     + "), ")
    butterfly_file.write(".a_out(Fi_"  + str(i)     + ")); \n")
    barret_count = barret_count + 1

if(clocked_en == 1):
    butterfly_file.write("\n")
    butterfly_file.write("\t assign " + dne_signal + " =
    (clk_count == C_CYCLES_TO_WAIT) ? 1'b1 : 1'b0;\n")
    butterfly_file.write("\n")

    #Clock counter
    butterfly_file.write("\t always @ (posedge "  + clk_signal + ") begin \n")
    butterfly_file.write("\t \t if("              + rst_signal + " == 1'b1) begin \n")
    butterfly_file.write("\t \t \t clk_count <= " + str(cycles_width) + "'d0;\n")
    butterfly_file.write("\t \t end \n")

    butterfly_file.write("\t \t else begin\n")
    butterfly_file.write("\t \t \t if(clk_count == C_CYCLES_TO_WAIT) begin\n")
    butterfly_file.write("\t \t \t \t clk_count <= " + str(cycles_width) + "'d0;\n")
    butterfly_file.write("\t \t \t end \n")

    butterfly_file.write("\t \t \t else begin\n")
    butterfly_file.write("\t \t \t \t if(" + str_signal + " == 1'b1) begin\n")
    butterfly_file.write("\t \t \t \t \t clk_count <= clk_count + " +
    str(cycles_width) + "'d1;\n")
    butterfly_file.write("\t \t \t \t end \n")
    butterfly_file.write("\t \t \t end \n")
    butterfly_file.write("\t \t end \n")
    butterfly_file.write("\t end \n")
```

```
    #End module
    butterfly_file.write("\n")
    butterfly_file.write("endmodule")

    #Close file
    butterfly_file.close()

#Code to Execute
butterfly_gen(2)
butterfly_gen(4)
butterfly_gen(8)
butterfly_gen(16)
butterfly_gen(32)
butterfly_gen(64)
butterfly_gen(128)
butterfly_gen(256)
```

10.4 ntt_butterfly_nxn_test_vec.py

Python script to generate input test vectors as well as expected-output vectors.

```
# -*- coding: utf-8 -*-
"""
Created on Sun Apr  2 01:06:59 2023
@author: bvoc5

Generate test vectors for ntt_butterfly_nxn unit testbench file
in CRYSTALS-Dilithium NTT FPGA Implementation
"""

import random
import math

q = 8380417
r = 1753
N = 8

# "U" -> Unit Test, "R" -> Regression Test
test        = "U"
test_length = 8

def ntt_nxn_butterfly_instance(fi, N_local):

    Fi = [0]*N_local

    f1 = fi[0 : int(N_local/2)]
    f2 = fi[int(N_local/2) : N_local]

    for k in range(int(N_local/2)):
        Fi[k]                  = (f1[k] +
```

```
        pow(r, int((256/N_local)*(2*k + 1)), q)*f2[k]) % q
        Fi[k + int(N_local/2)] = (f1[k] -
        pow(r, int((256/N_local)*(2*k + 1)), q)*f2[k]) % q

    return Fi

def ntt_nxn_butterfly(fi):

    Fi = [0]*N
    if test == "U":
        Fi = ntt_nxn_butterfly_instance(fi, N)

    elif test == "R":

        Fi_temp = []
        fi_temp = fi

        for i in range(int(math.log(N, 2))):
            M1 = 2**(i + 1)
            M2 = int(N/(2**M1))
            for j in range(M2):
                Fi_temp2 = ntt_nxn_butterfly_instance(fi_temp[M1*j : M1 + M1*j], M1)
                Fi_temp.append(Fi_temp2)
            fi_temp = Fi_temp

    return Fi


butterfly_test_vec_file = open("ntt_butterfly_" + str(N) + "x" +
str(N) + "_test_vec.txt", "w")
butterfly_test_vec_file.write("\n")

for i in range(N):
    butterfly_test_vec_file.write("\t reg  [23:0] fi_" + str(i) + ";\n")
butterfly_test_vec_file.write("\n")

for i in range(N):
    butterfly_test_vec_file.write("\t wire [23:0] Fi_" + str(i) + ";\n")
butterfly_test_vec_file.write("\n")

for i in range(N):
    butterfly_test_vec_file.write("\t //Input test vectors: fi_"  + str(i) + " \n")
    butterfly_test_vec_file.write("\t reg [23:0] test_vec_in_fi_" + str(i) +
    "[0:" + str(test_length-1) + "];\n")
    butterfly_test_vec_file.write("\n")

for i in range(N):
    butterfly_test_vec_file.write("\t //Expected-output test vectors: Fi_"
    + str(i) + " \n")
    butterfly_test_vec_file.write("\t reg [23:0] test_vec_exp_out_Fi_"
    + str(i) + "[0:" + str(test_length-1) + "];\n")
```

```
    butterfly_test_vec_file.write("\n")

butterfly_test_vec_file.write("\t //Initialization Blocks\n")
for i in range(test_length):

    fi = []
    for j in range(N):
        fi_in = int((q-1)*random.random())
        fi.append(fi_in)

    Fi = ntt_nxn_butterfly(fi)

    butterfly_test_vec_file.write("\t initial begin\n")

    for j in range(N):
        butterfly_test_vec_file.write("\t \t test_vec_in_fi_" + str(j) +
        "[" + str(i) + "] = ")
        butterfly_test_vec_file.write("24'd" + str(fi[j]) + ";\n")

    butterfly_test_vec_file.write("\n")

    for j in range(N):
        butterfly_test_vec_file.write("\t \t test_vec_exp_out_Fi_" + str(j) +
        "[" + str(i) + "] = ")
        butterfly_test_vec_file.write("24'd" + str(Fi[j]) + ";\n")

    butterfly_test_vec_file.write("\t end\n")
    butterfly_test_vec_file.write("\n")

butterfly_test_vec_file.close()
```

10.5 ntt_butterfly_8x8_unit_tb.v

Sample unit test verifying the functionality of the RTL module `ntt_butterfly_8x8.v`.

```
module ntt_butterfly_8x8_unit_tb();

     parameter CYCLES_TO_WAIT = 10;

         integer i;
         integer j;

     reg clk_100Mhz;

           reg  [23:0] fi_0;
         reg  [23:0] fi_1;
         reg  [23:0] fi_2;
         reg  [23:0] fi_3;
         reg  [23:0] fi_4;
         reg  [23:0] fi_5;
         reg  [23:0] fi_6;
```

```
reg  [23:0] fi_7;

wire [23:0] Fi_0;
wire [23:0] Fi_1;
wire [23:0] Fi_2;
wire [23:0] Fi_3;
wire [23:0] Fi_4;
wire [23:0] Fi_5;
wire [23:0] Fi_6;
wire [23:0] Fi_7;

//Input test vectors: fi_0
reg [23:0] test_vec_in_fi_0[0:7];

//Input test vectors: fi_1
reg [23:0] test_vec_in_fi_1[0:7];

//Input test vectors: fi_2
reg [23:0] test_vec_in_fi_2[0:7];

//Input test vectors: fi_3
reg [23:0] test_vec_in_fi_3[0:7];

//Input test vectors: fi_4
reg [23:0] test_vec_in_fi_4[0:7];

//Input test vectors: fi_5
reg [23:0] test_vec_in_fi_5[0:7];

//Input test vectors: fi_6
reg [23:0] test_vec_in_fi_6[0:7];

//Input test vectors: fi_7
reg [23:0] test_vec_in_fi_7[0:7];

//Expected-output test vectors: Fi_0
reg [23:0] test_vec_exp_out_Fi_0[0:7];

//Expected-output test vectors: Fi_1
reg [23:0] test_vec_exp_out_Fi_1[0:7];

//Expected-output test vectors: Fi_2
reg [23:0] test_vec_exp_out_Fi_2[0:7];

//Expected-output test vectors: Fi_3
reg [23:0] test_vec_exp_out_Fi_3[0:7];

//Expected-output test vectors: Fi_4
reg [23:0] test_vec_exp_out_Fi_4[0:7];

//Expected-output test vectors: Fi_5
```

```
    reg [23:0] test_vec_exp_out_Fi_5[0:7];

    //Expected-output test vectors: Fi_6
    reg [23:0] test_vec_exp_out_Fi_6[0:7];

    //Expected-output test vectors: Fi_7
    reg [23:0] test_vec_exp_out_Fi_7[0:7];

    initial begin
    clk_100Mhz = 1'b0;
end

//Clock source
always begin
   #5; clk_100Mhz = ~clk_100Mhz;
end

    //Initialization Blocks
    initial begin
             test_vec_in_fi_0[0] = 24'd6160794;
             test_vec_in_fi_1[0] = 24'd3729094;
             test_vec_in_fi_2[0] = 24'd821436;
             test_vec_in_fi_3[0] = 24'd1638510;
             test_vec_in_fi_4[0] = 24'd7514611;
             test_vec_in_fi_5[0] = 24'd6118607;
             test_vec_in_fi_6[0] = 24'd5530952;
             test_vec_in_fi_7[0] = 24'd7342218;

             test_vec_exp_out_Fi_0[0] = 24'd5669706;
             test_vec_exp_out_Fi_1[0] = 24'd5935776;
             test_vec_exp_out_Fi_2[0] = 24'd2835773;
             test_vec_exp_out_Fi_3[0] = 24'd7778795;
             test_vec_exp_out_Fi_4[0] = 24'd6651882;
             test_vec_exp_out_Fi_5[0] = 24'd1522412;
             test_vec_exp_out_Fi_6[0] = 24'd7187516;
             test_vec_exp_out_Fi_7[0] = 24'd3878642;
    end

    initial begin
             test_vec_in_fi_0[1] = 24'd6876389;
             test_vec_in_fi_1[1] = 24'd1975857;
             test_vec_in_fi_2[1] = 24'd4602930;
             test_vec_in_fi_3[1] = 24'd7808932;
             test_vec_in_fi_4[1] = 24'd283156;
             test_vec_in_fi_5[1] = 24'd8356797;
             test_vec_in_fi_6[1] = 24'd80761;
             test_vec_in_fi_7[1] = 24'd7536007;

             test_vec_exp_out_Fi_0[1] = 24'd2728632;
             test_vec_exp_out_Fi_1[1] = 24'd2113095;
             test_vec_exp_out_Fi_2[1] = 24'd3795874;
```

```
        test_vec_exp_out_Fi_3[1] = 24'd5921040;
        test_vec_exp_out_Fi_4[1] = 24'd2643729;
        test_vec_exp_out_Fi_5[1] = 24'd1838619;
        test_vec_exp_out_Fi_6[1] = 24'd5409986;
        test_vec_exp_out_Fi_7[1] = 24'd1316407;
end

initial begin
        test_vec_in_fi_0[2] = 24'd1244392;
        test_vec_in_fi_1[2] = 24'd3230516;
        test_vec_in_fi_2[2] = 24'd6747272;
        test_vec_in_fi_3[2] = 24'd1042654;
        test_vec_in_fi_4[2] = 24'd2464511;
        test_vec_in_fi_5[2] = 24'd2892116;
        test_vec_in_fi_6[2] = 24'd5141250;
        test_vec_in_fi_7[2] = 24'd7534089;

        test_vec_exp_out_Fi_0[2] = 24'd4768373;
        test_vec_exp_out_Fi_1[2] = 24'd925230;
        test_vec_exp_out_Fi_2[2] = 24'd5392395;
        test_vec_exp_out_Fi_3[2] = 24'd5032258;
        test_vec_exp_out_Fi_4[2] = 24'd6100828;
        test_vec_exp_out_Fi_5[2] = 24'd5535802;
        test_vec_exp_out_Fi_6[2] = 24'd8102149;
        test_vec_exp_out_Fi_7[2] = 24'd5433467;
end

initial begin
        test_vec_in_fi_0[3] = 24'd835924;
        test_vec_in_fi_1[3] = 24'd6030074;
        test_vec_in_fi_2[3] = 24'd1887552;
        test_vec_in_fi_3[3] = 24'd5492287;
        test_vec_in_fi_4[3] = 24'd2226060;
        test_vec_in_fi_5[3] = 24'd1897376;
        test_vec_in_fi_6[3] = 24'd5856029;
        test_vec_in_fi_7[3] = 24'd3248945;

        test_vec_exp_out_Fi_0[3] = 24'd6479918;
        test_vec_exp_out_Fi_1[3] = 24'd5076546;
        test_vec_exp_out_Fi_2[3] = 24'd4350607;
        test_vec_exp_out_Fi_3[3] = 24'd6256317;
        test_vec_exp_out_Fi_4[3] = 24'd3572347;
        test_vec_exp_out_Fi_5[3] = 24'd6983602;
        test_vec_exp_out_Fi_6[3] = 24'd7804914;
        test_vec_exp_out_Fi_7[3] = 24'd4728257;
end

initial begin
        test_vec_in_fi_0[4] = 24'd2113451;
        test_vec_in_fi_1[4] = 24'd3390569;
        test_vec_in_fi_2[4] = 24'd5629027;
```

```
        test_vec_in_fi_3[4] = 24'd5900066;
        test_vec_in_fi_4[4] = 24'd5492090;
        test_vec_in_fi_5[4] = 24'd2181125;
        test_vec_in_fi_6[4] = 24'd6374548;
        test_vec_in_fi_7[4] = 24'd1865695;

        test_vec_exp_out_Fi_0[4] = 24'd3175772;
        test_vec_exp_out_Fi_1[4] = 24'd904055;
        test_vec_exp_out_Fi_2[4] = 24'd512679;
        test_vec_exp_out_Fi_3[4] = 24'd4760456;
        test_vec_exp_out_Fi_4[4] = 24'd1051130;
        test_vec_exp_out_Fi_5[4] = 24'd5877083;
        test_vec_exp_out_Fi_6[4] = 24'd2364958;
        test_vec_exp_out_Fi_7[4] = 24'd7039676;
end

initial begin
        test_vec_in_fi_0[5] = 24'd6889110;
        test_vec_in_fi_1[5] = 24'd2253402;
        test_vec_in_fi_2[5] = 24'd7538125;
        test_vec_in_fi_3[5] = 24'd4834889;
        test_vec_in_fi_4[5] = 24'd4888133;
        test_vec_in_fi_5[5] = 24'd7304219;
        test_vec_in_fi_6[5] = 24'd7240836;
        test_vec_in_fi_7[5] = 24'd1347179;

        test_vec_exp_out_Fi_0[5] = 24'd950864;
        test_vec_exp_out_Fi_1[5] = 24'd4613492;
        test_vec_exp_out_Fi_2[5] = 24'd457053;
        test_vec_exp_out_Fi_3[5] = 24'd3373799;
        test_vec_exp_out_Fi_4[5] = 24'd4446939;
        test_vec_exp_out_Fi_5[5] = 24'd8273729;
        test_vec_exp_out_Fi_6[5] = 24'd6238780;
        test_vec_exp_out_Fi_7[5] = 24'd6295979;
end

initial begin
        test_vec_in_fi_0[6] = 24'd3536705;
        test_vec_in_fi_1[6] = 24'd3058919;
        test_vec_in_fi_2[6] = 24'd3474614;
        test_vec_in_fi_3[6] = 24'd5554529;
        test_vec_in_fi_4[6] = 24'd6723010;
        test_vec_in_fi_5[6] = 24'd6107763;
        test_vec_in_fi_6[6] = 24'd4994185;
        test_vec_in_fi_7[6] = 24'd3280394;

        test_vec_exp_out_Fi_0[6] = 24'd3753375;
        test_vec_exp_out_Fi_1[6] = 24'd320226;
        test_vec_exp_out_Fi_2[6] = 24'd6281642;
        test_vec_exp_out_Fi_3[6] = 24'd6876327;
        test_vec_exp_out_Fi_4[6] = 24'd3320035;
```

```
            test_vec_exp_out_Fi_5[6] = 24'd5797612;
            test_vec_exp_out_Fi_6[6] = 24'd667586;
            test_vec_exp_out_Fi_7[6] = 24'd4232731;
    end

    initial begin
            test_vec_in_fi_0[7] = 24'd6510849;
            test_vec_in_fi_1[7] = 24'd1126108;
            test_vec_in_fi_2[7] = 24'd6161972;
            test_vec_in_fi_3[7] = 24'd906439;
            test_vec_in_fi_4[7] = 24'd1310684;
            test_vec_in_fi_5[7] = 24'd6419979;
            test_vec_in_fi_6[7] = 24'd5670648;
            test_vec_in_fi_7[7] = 24'd5782392;

            test_vec_exp_out_Fi_0[7] = 24'd1384189;
            test_vec_exp_out_Fi_1[7] = 24'd1974865;
            test_vec_exp_out_Fi_2[7] = 24'd1453786;
            test_vec_exp_out_Fi_3[7] = 24'd8198446;
            test_vec_exp_out_Fi_4[7] = 24'd3257092;
            test_vec_exp_out_Fi_5[7] = 24'd277351;
            test_vec_exp_out_Fi_6[7] = 24'd2489741;
            test_vec_exp_out_Fi_7[7] = 24'd1994849;
    end


initial begin
        i = 0;
        j = 0;
        fi_0 = test_vec_in_fi_0[j];
        fi_1 = test_vec_in_fi_1[j];
        fi_2 = test_vec_in_fi_2[j];
        fi_3 = test_vec_in_fi_3[j];
        fi_4 = test_vec_in_fi_4[j];
        fi_5 = test_vec_in_fi_5[j];
        fi_6 = test_vec_in_fi_6[j];
        fi_7 = test_vec_in_fi_7[j];
    end

    //Test bench logic
 always @ (posedge clk_100Mhz) begin
   i = i + 1;
   if(i % CYCLES_TO_WAIT == 0 && j < 8) begin
       if(Fi_0 == test_vec_exp_out_Fi_0[j] &&
          Fi_1 == test_vec_exp_out_Fi_1[j] &&
          Fi_2 == test_vec_exp_out_Fi_2[j] &&
          Fi_3 == test_vec_exp_out_Fi_3[j] &&
          Fi_4 == test_vec_exp_out_Fi_4[j] &&
          Fi_5 == test_vec_exp_out_Fi_5[j] &&
          Fi_6 == test_vec_exp_out_Fi_6[j] &&
          Fi_7 == test_vec_exp_out_Fi_7[j]) begin
```

```
    $display("\t[Testbench, j = %d] PASS", j);
end
else begin
    $display("\t[Testbench, j = %d] FAIL", j);

    $display("\t \t[Testbench, j = %d] Fi_0 Exepcted:
    %h", j, test_vec_exp_out_Fi_0[j]);
    $display("\t \t[Testbench, j = %d] Fi_1 Expected:
    %h", j, test_vec_exp_out_Fi_1[j]);
    $display("\t \t[Testbench, j = %d] Fi_2 Exepcted:
    %h", j, test_vec_exp_out_Fi_2[j]);
    $display("\t \t[Testbench, j = %d] Fi_3 Expected:
    %h", j, test_vec_exp_out_Fi_3[j]);
    $display("\t \t[Testbench, j = %d] Fi_4 Exepcted:
    %h", j, test_vec_exp_out_Fi_4[j]);
    $display("\t \t[Testbench, j = %d] Fi_5 Expected:
    %h", j, test_vec_exp_out_Fi_5[j]);
    $display("\t \t[Testbench, j = %d] Fi_6 Exepcted:
    %h", j, test_vec_exp_out_Fi_6[j]);
    $display("\t \t[Testbench, j = %d] Fi_7 Expected:
    %h", j, test_vec_exp_out_Fi_7[j]);

    $display("\t \t[Testbench, j = %d] Fi_0 Received:
    %h", j, Fi_0);
    $display("\t \t[Testbench, j = %d] Fi_1 Received:
    %h", j, Fi_1);
    $display("\t \t[Testbench, j = %d] Fi_2 Received:
    %h", j, Fi_2);
    $display("\t \t[Testbench, j = %d] Fi_3 Received:
    %h", j, Fi_3);
    $display("\t \t[Testbench, j = %d] Fi_4 Received:
    %h", j, Fi_4);
    $display("\t \t[Testbench, j = %d] Fi_5 Received:
    %h", j, Fi_5);
    $display("\t \t[Testbench, j = %d] Fi_6 Received:
    %h", j, Fi_6);
    $display("\t \t[Testbench, j = %d] Fi_7 Received:
    %h", j, Fi_7);

end

j = j + 1;
fi_0 = test_vec_in_fi_0[j];
fi_1 = test_vec_in_fi_1[j];
fi_2 = test_vec_in_fi_2[j];
fi_3 = test_vec_in_fi_3[j];
fi_4 = test_vec_in_fi_4[j];
fi_5 = test_vec_in_fi_5[j];
fi_6 = test_vec_in_fi_6[j];
fi_7 = test_vec_in_fi_7[j];
```

```
        end
    end

    //Device under test (DUT)
    ntt_butterfly_8x8 ntt_butterfly_8x8_DUT(
        .clk_100Mhz(clk_100Mhz),
        .fi_0(fi_0),
        .fi_1(fi_1),
        .fi_2(fi_2),
        .fi_3(fi_3),
        .fi_4(fi_4),
        .fi_5(fi_5),
        .fi_6(fi_6),
        .fi_7(fi_7),
        .Fi_0(Fi_0),
        .Fi_1(Fi_1),
        .Fi_2(Fi_2),
        .Fi_3(Fi_3),
        .Fi_4(Fi_4),
        .Fi_5(Fi_5),
        .Fi_6(Fi_6),
        .Fi_7(Fi_7));

endmodule
```