Number Theoretic Transform (NTT) FPGA Accelerator

Austin Hartshorn, Humberto Leon, Noel Qiao, and Scott Weber
Advisor: Yarkin Doroz, Dr.

Worcester Polytechnic Institute, Worcester MA 01609, USA

Abstract. Post-quantum cryptography has become popular in recent years due to advances in quantum computing. Current cryptographic solutions are vulnerable to post-quantum attacks as they can solve computationally hard problems. Many post-quantum cryptographic schemes rely on lattice-based solutions. Polynomial multiplication is required for these cryptographic primitives. However, those latticed-based schemes are inefficient without hardware to accelerate the polynomial multiplication computations that are performed. We introduce a hardware implementation of the Number Theoretic Transform to speed up polynomial multiplication. As a result, two methods are implemented. The first is a 64 point NTT module. The second is an iterative NTT module. We developed our design on the xc7vx690t-2ffg1761c Virtex 7 platform. We are able to compute a forward NTT calculation in 5.315μs. A full polynomial multiplication can be done in 20.310μs.

1 Introduction

Classical cryptographic algorithms protect data against malicious actors. Files can be encrypted prior to transmission, then decrypted once received. These encryption and decryption schemes can either rely on symmetric or asymmetric keys. Symmetric key encryption means that both sides need to have the key. On the other hand, asymmetric key encryption relies on computationally hard functions to generate a public key easily while preventing someone who knows that public key from gaining access to a user's private key and as a result, their data. Algorithms created for quantum computers can solve some previously created computationally hard problems in a shorter time period which has led to major research into post-quantum cryptography algorithms that are secure against quantum computers.

Research into post-quantum algorithms have led to a number of different types of algorithms being created. We chose to focus on lattice-based schemes, most of which see significant speedups when using hardware accelerators. This is due to the fact that many lattice-based cryptographic primitives use polynomial multiplication as a basic operation. The Number Theoretic Transform (NTT) is used for the polynomial multiplication within the algorithm. The NTT is used to perform a transform, then multiply two polynomials then transform back. Hardware accelerators, such as GPUs or FPGAs, can be used to perform

polynomial multiplication faster because they can have multiple functional units compute different parts of the Number Theoretic Transform in parallel.

2 Background

In this background section, first, we explain classical cryptography along with public-key encryption to give a better understanding of the current schemes in use. Then we describe quantum computing to introduce how it is used to attack modern cryptographic schemes. We further explain this project by discussing how post-quantum cryptography, specifically lattice-based encryption schemes fight off these attacks. Finally, we give an overview of the Number Theory Transform and how hardware can be used to accelerate its implementation.

2.1 Cryptography

A huge amount of data is at risk of being stolen by hackers. Cryptographic schemes provide security to said data. For example, files are encrypted when in transit and at rest, but are decrypted when in use. However, current cryptographic solutions are still vulnerable. The security measures we have today are not able to maintain security with future technology. The most popular cryptographic algorithms that exist today are secure because they rely on computationally hard problems that classical computers are unable to solve quickly. Quantum computing algorithms, however, can solve these problems in a feasible amount of time. It has been realized that the strongest common encryption could be decrypted and public-key cryptosystems will be beaten.

There are two main types of encryption. Symmetric encryption requires a sender and a receiver to have identical keys to encrypt and decrypt data. Asymmetric, or public-key encryption, uses a pair of keys: public keys that can be distributed widely and private keys that are known only to the owner. These keys are generated by one-way functions and only the private key needs to stay private to maintain security.

The following functions explain public-key encryption where m is the message, c is the cipher text, pk is the public-key, and sk is the private (secret) key:

$$c = Enc(m, pk) \tag{1}$$

$$d = Dec(c, sk) \tag{2}$$

$$sig = sign(m, sk) \tag{3}$$

$$verify = verify(sig, pk) \tag{4}$$

With public-key encryption, a user can encrypt a message using the public key as seen in Equation 1. The message can only be decrypted with the private key

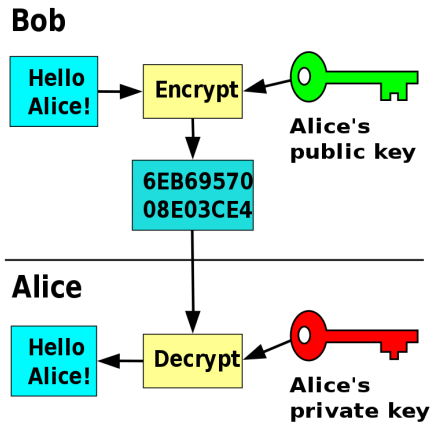

Fig. 1. Public-key Encryption and Decryption Scheme

as seen in Equation 2. Another possibility for the sender is creating a digital signature on the message by combining a message with a private key, as seen in Equation 3. Anyone with the sender's corresponding public key can combine the same message and the digital signature with it to verify if the signature was valid. Equation 4 shows the verification function.

Popular cryptographic methods use trapdoor functions, mathematical constructs that are relatively easy to compute in one direction, to create extensive keys. The keys are hard to compute in the opposite direction. Classical computers take a very long time to brute force all possible permutations of the private keys. Quantum computers, on the other hand, can easily break trapdoor functions.

2.2 Quantum Computing

Quantum computers are machines that use quantum mechanics. They leverage the strange and unreasonable physical properties of matter at an atomic scale to perform computations. Classical computers encode data in binary digits (bits) that are represented as "1" or "0". Quantum computers generate and manipulate quantum bits or qubits that can encode more than two states. Qubits are typically subatomic particles such as electrons or photons that are isolated in a controlled quantum state.

Superposition is an important qubit property. It is the ability to "be" in multiple states. Therefore, a quantum computer with several qubits in superposition can represent an exponent of potential outcomes simultaneously.

Entanglement is another important qubit property. This allows the generation of pairs of qubits where two members of a pair exist in a single quantum state. When the state of one of the qubits is changed, the state of the other one changes instantaneously in a predictable way.

The processing power of a quantum computer increases greatly with the number of qubits. public-key algorithms like RSA and ECDSA will be obsolete with further quantum computer development. Unique algorithms for quantum computers reduce the time it takes to break these mathematically hard problems.

Grover's algorithm allows quantum computers to search all possible permutations in a relatively short amount of time. According to this algorithm, quantum computers can find with high probability the unique input to a black box function that produces a particular output value, using just $\mathbf{O}(\sqrt{N})$ evaluations of the function, where N is the size of the function's domain. All 2^n possible n-bit keys can be searched in $2^{n/2}$ time, suggesting that the lengths of symmetric keys be doubled to protect against these attacks. Using this, hackers could brute-force search a 128-bit symmetric cryptographic key in about 2^{64} iterations, or a 256-bit key in roughly 2^{128} iterations. Generally, this is all quantum computers can do against the symmetric algorithms used for encryption today. public-key cryptographic schemes are in more danger due to Shor's algorithm.

Current popular security algorithms rely on one of three hard mathematical problems: the integer factorization problem, the discrete logarithm problem, or the elliptic-curve discrete logarithm problem. Shor's algorithm running on a quantum computer can solve all of these problems and can factor integers in polynomial time (the time taken is polynomial in $logN$, the size of the integer given as input). This algorithm solves the following problem: Given an integer $\boldsymbol{N}$, find its prime factors.

Shor's algorithm consists of two parts. First, the factoring problem is translated into the problem of finding the period of a function. This may be implemented on a classical computer. The second part of the algorithm uses the quantum Fourier transform to find the period of a modular operation. This is responsible for the quantum speedup. If the period of this function using the factorable integer as a modulus is found, the prime factors are found quickly with some additional calculations. Specifically, it takes quantum gates of order $O((logN)^2(loglogN)(logloglogN))$ using fast multiplication. This demonstrates that the integer-factorization problem can be solved efficiently. Computation is exponentially faster than the most efficient classical factoring algorithm (the general number field sieve), which works in sub-exponential time.

Given the evident threat, new cryptographic algorithms are being developed that can resist quantum computing. These methods are known as post-quantum cryptography and are focused on the following six approaches:

- Lattice-based cryptography
- Multivariate cryptography
- Hash-based cryptography
- Code-based cryptography
- Super-singular elliptic curve isogeny cryptography
- Symmetric key quantum resistance

Schemes have been submitted in the competition initiated by the US National Institute of Standardization and Technology (NIST). NIST launched a

standardization process in 2016 to develop standards for post-quantum encryption for protecting electronic information and government use in light of the potential threat to current cryptography by quantum computers.

2.3 Lattice-based Cryptography

For this project, we focused on lattice-based cryptography. This is considered the most promising and well-studied method. Among the 26 selected Round-2 candidates for the NIST competition in 2019, 11 are based on lattices (eight for public-key encryption and three for digital signature) with most of these primitives focusing on polynomial operations, specifically multiplication. In addition to security, we also focused on performance since it is also a criteria for standardization.

Lattice cryptography hides data inside mathematically complex problems and geometric structures known as lattices. Cryptographers find the difficulty of these math problems useful because they can apply this intractability to protect information. One of these problems known as the Ring Learning with Errors Problem (R-LWE) is the basis for cryptographic schemes that include polynomial multiplication over a finite field as one of their fundamental operations. It is believed to be sufficiently hard, even for attackers with a large scale quantum computer, thus showing promise for post-quantum cryptography.

Lattice-based cryptography is also the basis of the Fully Homomorphic Encryption (FHE) scheme. FHE can perform calculations on a file without ever seeing sensitive data or exposing it to malicious actors. For example, a consumer credit reporting agency could analyze and produce credit scores without ever decrypting personal data. Alternatively, primary care physicians could share patient medical records in a way that enables each party to access pertinent data without revealing the identity of the patient.

Lattice-based cryptography provides fast, quantum-safe, fundamental primitives and allows for constructions of primitives that were previously thought impossible. This combination has established lattice-based cryptography as a fascinating research field with near future standardization potential.

2.4 Number Theory Transform

A versatile and efficient polynomial multiplier would be of great help to facilitate researchers' hardware designs of post-quantum algorithms. This is because the most computationally intensive operation in these cryptosystems is polynomial multiplication. The Number Theory Transform (NTT) is a powerful tool that enables this operation to be computed in quasi-polynomial complexity and improve the performance of the system. It is one of the core components in implementations of lattice-based cryptosystems because it provides efficient algorithms for cyclic and negacyclic convolutions, which help lower the runtime for polynomial multiplication. Ideal lattice-based schemes are usually defined in $R = Z_q[x]/ < x^n + 1 >$ with modulus $x^n + 1$ where n is a power of two and one can make use of the negacyclic convolution property of the NTT that allows

carrying out a polynomial multiplication in $Z_q[x]/ < x^n + 1 >$ using length-n transforms and no zero padding.

The NTT is a specialized version of the Discrete Fourier Transform(DFT) which avoids round-off errors for exact convolutions of integer sequences. The coefficient ring is taken to be a finite field containing the right roots of unity. It is computed with Fast Fourier Transform(FFT) algorithms that work over this specific finite field. The notation for the NTT is the following: with n being a power of 2 and q a prime with $q \equiv 1(mod\ 2n)$, let $a = (a[0], ..., a[n-1]) \in Z_q^n$, and let ω be a primitive n-th root of unity in Z_q, meaning that $\omega^n \equiv 1(mod\ q)$. The forward transformation $\tilde{a} = NTT(a)$ is defined as $\tilde{a}[i] = \sum_{j=0}^{n-1} a[j]\omega^{ij}(mod\ q)$ for $i = 0, 1, ..., n-1$.

Some previous optimizations of the NTT-based polynomial multiplication explain how to merge multiplications by the powers of ω with the powers of ψ and ψ^{-1} inside the NTT. This improves the performance of the system by precomputing and storing in memory the values related to these parameters. Expensive reordering or a bit-reversal step before or after NTT computation is necessary because of the restrictive nature of certain algorithms to only accept inputs in standard ordering and producing results in but-reversed ordering.

The forward NTT algorithm that computes the transformation based on the Cooley-Tukey butterfly that absorbs the powers of ψ in bit-reversed ordering [7] as seen in Algorithm 1. This function receives the inputs in standard ordering and gives a result in bit-reversed ordering. Later on in the paper, we use ω in place of ψ.

Algorithm 1: NTT Algorithm based on Cooley-Tukey butterfly

input : A vector $a = (a[0], a[1], ..., a[n-1]) \in Z_q^n$ in standard ordering, where q is a prime such that $q \equiv 1 \mod 2n$ and n is a power of two, and a pre-computed table $\psi \in Z_q^n$ storing power of ψ in bit-reversed order

output: $a \leftarrow NTT(a)$ in bit-reversed ordering

```
1  t = n;
2  for (m = 1; m < n; m = 2m) do
3      t = t/2;
4      for (i = 0; i < m; i++) do
5          j1 = 2 · i · t;
6          j2 = j1 + t − 1;
7          S = ψ_rev[m + i];
8          for (j = j1; j <= j2; j++) do
9              U = a[j];
10             V = a[j + t] · S;
11             a[j] = U + V mod q;
12             a[j + t] = U − V mod q;
13 return a
```

The cyclic convolution of two integer sequences coming from a finite field of length n can be computed by applying the algorithm to both sequences, then multiplying the resulting NTT sequences of length $n$coefficient-wise and transforming the result back via an inverse NTT. Because of this cyclic convolution produced, computing $c = a \cdot b \mod (X^n + 1)$ with two polynomials a and b would require applying the NTT of length $2n$ and thus n zeros to be appended to each input. This effectively doubles the length of the inputs and requires the computation of an explicit reduction modulo $X^n + 1$.

The polynomial product of two n-1 degree polynomials a and b has degree $2n$-1, so it requires evaluations in at least $2n$ distinct points to be uniquely identified. To do this, we use the $2n$th primitive roots of unity, meaning that a polynomial with a coefficient vector of at least length $2n$ is needed for our NTT algorithm. Following this, we pad the coefficient vectors of polynomials a and b to at least length $2n$ using zeros. For more precision since it is required that n is a power of 2, the coefficient vectors are padded to length 2^k, where k is the lowest integer such that $2^k \geq 2n$. The NTT algorithm is then applied to get evaluations of the polynomials a and b at the same $2n$ distinct inputs. If we then multiply the $2n$ evaluations of a with the respective $2n$ evaluations of b, we calculate $2n$ products in total that together represent the polynomial product of the two original polynomials. The INTT of this product is then computed to transform the vector of polynomial evaluations into the vector of its coefficients.

2.5 FPGA Acceleration

In cryptographic algorithms, the most computationally rigorous parts are relatively simple operations that have to be performed in iterations and can be parallelized for high performance. Because of this simplicity, these algorithms, specifically lattice-based, can use available hardware resources from Field Programmable Gate Arrays (FPGAs) with high efficiency to perform these tasks. In general, implementations of specialized hardware architectures for specific operations give a significant advantage over software implementations because of all the repetitive computations needed in software running on a general-purpose central processing unit (CPU).

Hardware acceleration has many advantages such as reduced power consumption, lower latency, increased parallelism, higher throughput and better utilization of area and functional components available on an integrated circuit. By offloading critical functions to an FPGA, system performance of a cryptographic algorithm can be accelerated. The programmable logic cell structure and large built-in memory of FPGAs allow for the implementation of bit-wise and memory-intensive operations. Also, with its reconfigurability at run time, the FPGA can be reused for different algorithms based on system specifications and specific applications. FPGAs offer the availability of more fixed parameters for better efficiency of cryptographic systems. Since FPGAs consist of an array of configurable logic blocks (CLBs) surrounded by input/output blocks that provide an interlace between the configurable logic block and package pins, the architecture is suitable for the implementation of deeply pipelined designs.

NTT arithmetic includes a large amount of modular addition, subtraction, and multiplication operations. The iterative NTT algorithm for a hardware implementation presented in this project is built with a high degree of parallelism so that arithmetic processing, memory fetches and writes occur at the same time, speeding up throughput and reducing processing latency. All the operations are divided into smaller ones that each perform one task in a clock cycle and pass the result to the next stage and retrieve the result of the previous stage. This pipelining allows for processing of data at a high speed. With this, our generalized NTT implementation can be scaled to fit different specifications and systems.

3 Methodology

We approached the hardware-based polynomial multiplication problem for post-quantum computing with a combination of two solutions.

The first solution is a special 64-point NTT that skips some of the initial stages of a standard iterative NTT algorithm by utilizing special numbers and shifts. NTT is simply a sped up and modified Discrete Fourier Transform (DFT) that uses integers instead of floats. This method should be faster than the iterative NTT algorithm on hardware.

The second solution is an iterative NTT algorithm designed for hardware implementation. This module is scalable depending on the desired performance and resource allocation. The faster 64-point NTT is initially used and switches to the standard iterative NTT. Figure 2 shows the overall block diagram. For both these parts, a fast modulus reduction was implemented as well.

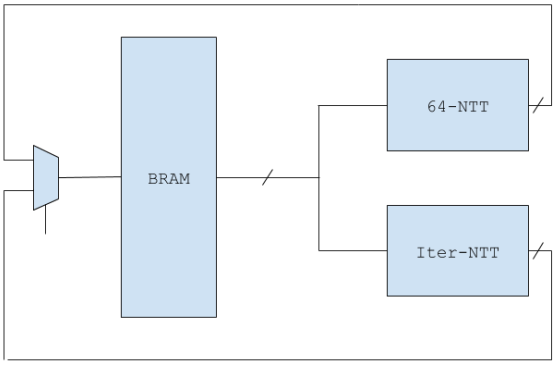

Fig. 2. Overall NTT Diagram

We created two fast modulus blocks that compute $z \pmod{p}$ where z is either an arbitrary 128-bit number or an arbitrary 256-bit number. Since we are using an NTT algorithm with 64-bit inputs, modulo operations occur on 64-bit

by 64-bit multiplications. Additionally, for the 64-point NTT, we must perform modulo operations on up to 254-bit numbers.

3.1 Fast Modulus Reduction

128-bit Number Reduction First, we implemented the 128-bit fast modulus based on Emmart et. al [6]. Since we are also using a 64-bit input polynomials, we have a fairly large range of primes to choose. There are special primes, known as Solinas Primes, that we are able to use that allow us to compute the modulus in. We use $p = 0xFFFFFFFF00000001$ as our prime, as Emmart et. al. use. Following Emmart et. al, they utilize two identities of p, $2^{96} \pmod p = -1$ and $2^{64} \pmod p = 2^{32} - 1$. In the following equation, where a, b, c, and d are 32-bit numbers, the modulus can then be calculated as follows:

$$\begin{aligned} z &\equiv (2^{96} \pmod p)a + (2^{64} \pmod p)b + 2^{32}c + d \\ &\equiv (-1)a + (2^{32} - 1)b + (2^{32})c + d \\ &\equiv (2^{32})(b + c) - a - b + d \end{aligned} \tag{5}$$

Equation 5 above allows the 128-bit input to be split up into four 32-bit numbers. The following operations are then easily able to be computed on an FPGA with only addition, subtraction, and shift operations. Once z is calculated, an additional check needs to occur to check if the result is within $\pmod p$. We checked for worst case scenarios of values for a, b, c, and d and determined that in the worst case, $p < z < 2p$. This results in subtracting p if z is greater than p, otherwise return z. Figure 3 shows the pipeline stages required to perform the shifts and the subtractions.

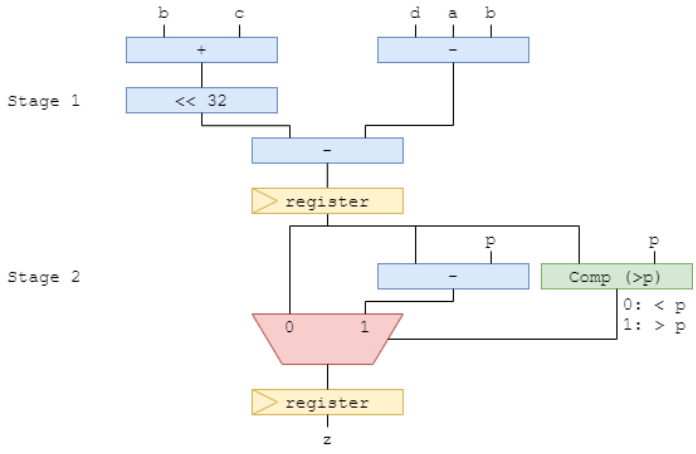

Fig. 3. 128-bit Fast Modulus Pipeline Stages

256-bit Number Reduction We utilize this same concept and expand it to support a 254-bit fast modulus reduction. Using the same prime p and the same identities, we were able quickly calculate $z \pmod p$ for a 256-bit input. The following is our equation where a, b, c, d, e, f, g, and h are 32-bit unsigned numbers.

$$\begin{aligned} z &\equiv (2^{32})a + b + (-2^{32} + 1)c + d(-2^{32}) + (-1)e + (2^{32} - 1)f + (2^{32})g + h \\ &\equiv 2^{32}(a - c - d + f + g) + b + c - e - f + h \\ &\equiv 2^{32}((a + f + g) - (c + d)) + (b + c + h) - (e + f) \end{aligned} \tag{6}$$

Unlike the 128-bit fast modulus reduction, there are many more scenarios for z values that we need to account for. We found 6 different cases for z, such that $-3p < z < 2p$. In equation 6, we create groupings of additions to prevent underflow. Since we are using unsigned datatypes, we need to be careful when performing subtractions to ensure we never go below zero. Using the groupings, we can easily check for various cases when we must add p or $2p$ to prevent underflow. For the last step, we subtract p or $2p$ as necessary. Even with more additions, shifts, subtractions, and comparison, we are able to pipeline the 256-bit fast modulus in three stages. Figure 4 shows the pipeline stages.

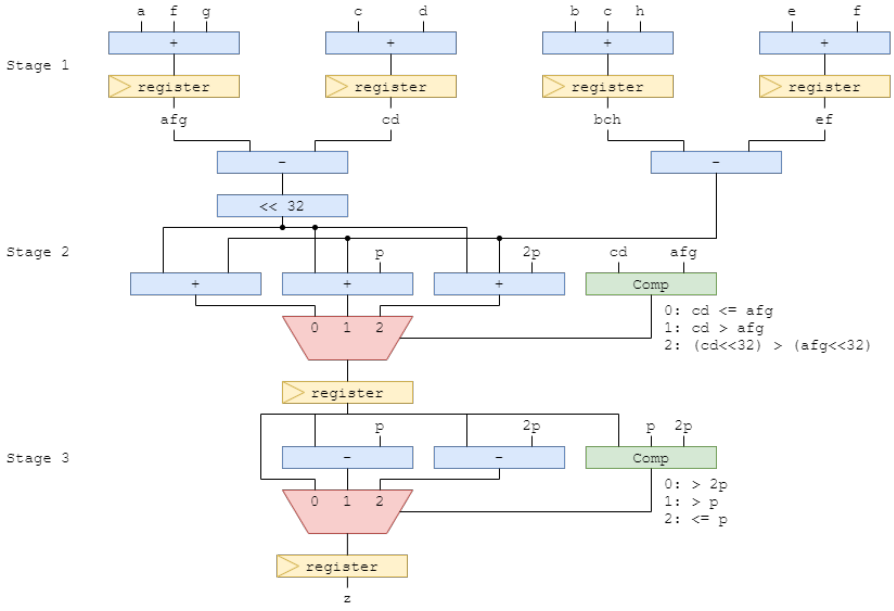

Fig. 4. 256-bit Fast Modulus Pipeline Stages

3.2 64-point Number Theoretic Transform

The 64-point NTT we implemented has a few specific properties that allow it to work. The first is the use of 64-bit input numbers which allows use to utilize bit shifting since 8 is a 64^{th} root of unity. The second requirement is that the number of shifts follow a set pattern which allows us to pre-calculate the shifts and store them ahead of time in BRAM blocks. The input to the NTT is 64 64-bit numbers. The corresponding output is another set of 64 64-bit numbers.

Normally when calculating the 64-point NTT, many multiplication operations are needed which are extremely slow on FPGA hardware. Essentially, a matrix of 64 by 64 multiplication operations is needed for the calculations. As mentioned before, using 64-bit input values allows us to shift instead of multiply which greatly improves the speed performance. These shift values are known as omegas.

The first step of the NTT involves computing rows of shifts. Each row is calculated by taking the summation of input numbers shifted by 64 corresponding omegas. After the summation, a modulus needs to be taken. In order to calculate ω, the indexes of the input and output are needed. The following equation calculates the correct shifts:

$$\omega_{i,j} = (i \times j) \pmod{p} \tag{7}$$

We use Equation 7 to precalculate omega and load those values into BRAM blocks. A Python script was used to calculate the 64 by 64 omega values.

Below is the calculations of the 64 output numbers, y, for the input numbers x and the ω values where i and j are the indexes of x and y respectively:

$$\begin{aligned}
y(0) &= x(0) \ll \omega_{0,0} + x(1) \ll \omega_{0,1} + \cdots + x(62) \ll \omega_{0,62} + x(63) \ll \omega_{0,63} \\
y(1) &= x(0) \ll \omega_{1,0} + x(1) \ll \omega_{1,1} + \cdots + x(62) \ll \omega_{1,62} + x(63) \ll \omega_{1,63} \\
&\vdots \\
y(62) &= x(0) \ll \omega_{62,0} + x(1) \ll \omega_{62,1} + \cdots + x(62) \ll \omega_{62,62} + x(63) \ll \omega_{62,63} \\
y(63) &= x(0) \ll \omega_{63,0} + x(1) \ll \omega_{63,1} + \cdots + x(62) \ll \omega_{63,62} + x(63) \ll \omega_{63,63}
\end{aligned} \tag{8}$$

For the hardware implementation, we decided to put more importance on the speed up of the calculations over the space requirements. We continued by treating each output as one row of calculations. We were able to easily pipeline each row in hardware in a Row Calculation block. The Row Calc. blocks consisted of shifts and modulus reductions. Figure 5 shows the detailed pipeline for the entire NTT calculation while Figure 6 shows the Block Diagram of the process. By creating 64 rows of calculations, we thus also created the need for 64 BRAM blocks to store our ω values. Each BRAM block is routed to the corresponding Row Calc. block as well as all 64 input numbers. Below is a block diagram of the 64-point NTT module.

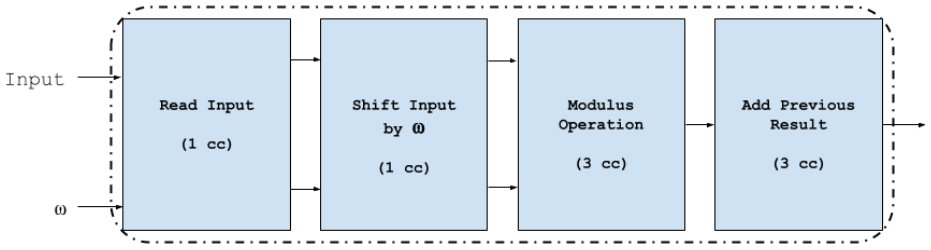

Fig. 5. Pipeline for 64-point NTT using Mod First Method

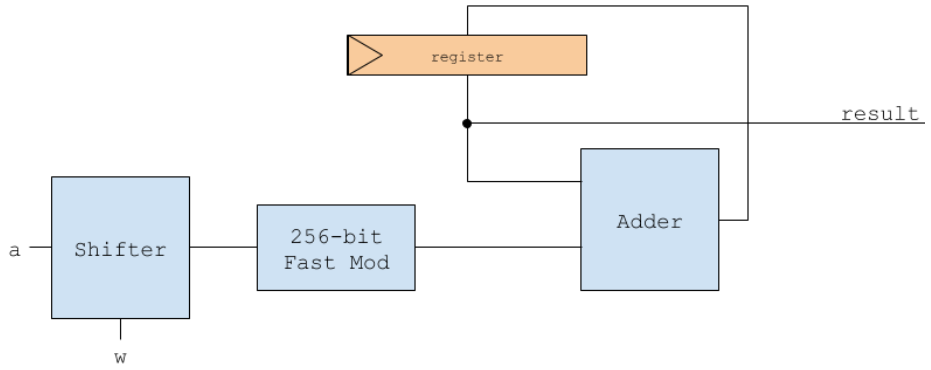

Fig. 6. Row Calculation Block Diagram

Differing Order of Operations There are two methods that we can use to compute the 64-point NTT. The first is to shift by w, perform a 256-bit fast modulus, then add the previous result. A final 128-bit fast modulus reduction is performed at the very end in case there is overflow. We call this first method the Mod First Method. The second is to shift by w, add the previous result, then perform a single 256-bit fast modulus at the very end. The second method requires a 256-bit fast modulus because the additions can cause the number to reach a maximum of 254-bits. We call this method the Add First Method. Figure 5, shows the first method of operations. Variations between the two methods are slight, but there are a few clock cycles of difference that are described in more detail in Section 4.2.

3.3 Iterative NTT

The top level implementation of the iterative NTT module is shown in Figure 8. Note that the control signals between the modules are not shown. Instead, the datapath is shown through the different modules. Each of the modules are fully pipelined. Additionally, the algorithms used in these modules are intended to be scalable. The scaling parameters are the input vector size (n), and the number

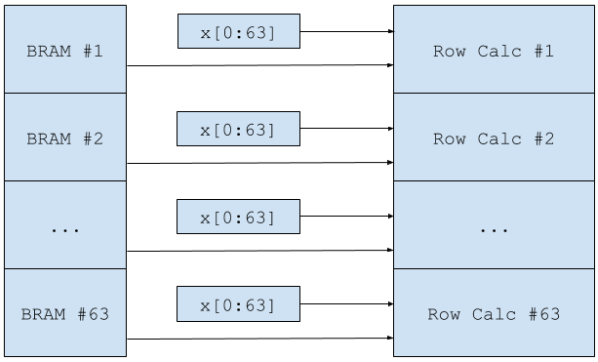

Fig. 7. 64-point NTT Input Block Diagram

of BRAMs / functional units (b). This section discusses the function of each of the blocks in the top level module. The blocks are:

- Index Calculator
- BRAM Router
- Write Back Controller
- ALU Router
- BRAM controller
- ALU Cores

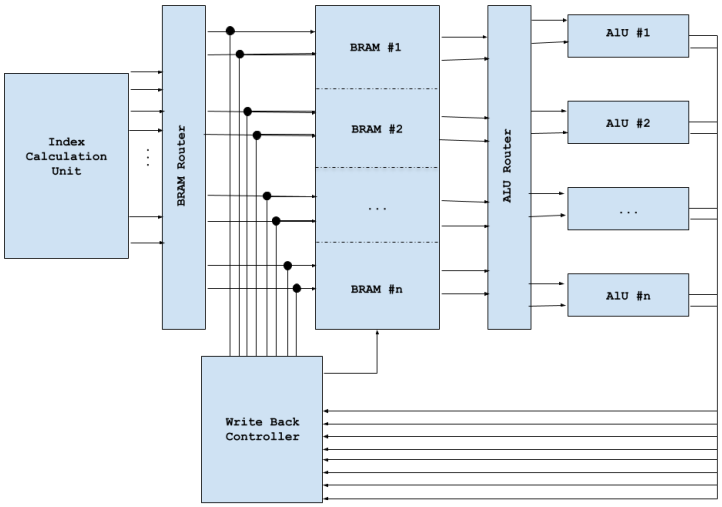

Fig. 8. Iterative NTT top level modules

Index Calculator The index calculator algorithm dynamically calculates the NTT index pairs to maximize utilization of a variable number of functional units. The two user defined parameters are BRAMs (b) and vector length(n). A rule set was developed to satisfy the index pairs based on these parameters. To realize the rule set, the index pairs were written out considering varying b. A Python implementation of the algorithm was developed first. A pseudocode model is included below. Also, pages from the spreadsheet are used to help explain the algorithm.

In software, NTT is easily modeled as a recursive function. For a hardware implementation, an iterative version was written. The iterative version also had to be modified to accommodate hardware limitations. Specifically, since 2 values can be read/written in a BRAM in a single cycle, the algorithm had to be adapted to calculate 2 values per BRAM per cycle. Algorithm 2 shown below achieves this.

Algorithm 2: Iterative NTT Index Calculation Algorithm

input : vector length n, bram number b
output: index pairs in the form of (x_pos, y_pos)
cc : clock cycles per stage, number of cycles for this power of 2
brams : how many groups of BRAMs need to be accessed for the *cc*

```
1  for i ← 0 to ℓ = log2(n) do
2      power_of_two = 2^(i+1);
3      cc = n/(2 * b);
4      Δy = power_of_two ≫ 1;
5      if b = n/2 then
6          groups = 1;
7      else if Δy >= n/b then
8          groups = 2;
9      else
10         groups = 1;
11     if power_of_two ≫ 2 >= n/(2b) or power_of_two = 2 then
12         Δx = 2;
13     else
14         Δx = 1;
15     if b = n/2 then
16         Δg = 1
17     else if Δy = n/b then
18         Δg = 2 × Δy
19     else
20         Δg = n/b
21     for cycle ← 0 to cc do
22         if cycle (mod Δy) = 0 and Δy ≠ 2 and cycle ≠ 0 then
23             Δx' = Δx' + Δy;
24         brams = b/groups;
25         for index_pair ← 0 to brams do
26             if Δg (mod Δy) = 0 and index_pair ≠ 0 and
                 Δg < power_of_two then
27                 Δg' = Δg' + Δy;
28             x_pos = Δg × index_pair + cycles × Δx + Δg' + Δx';
29             y_pos = x_pos + Δy;
```

An example of index calculation is shown in table 1. In this example, 2 BRAMs are used to store a 32 length vector. A cell contains an x and y coordinate pair. The color of the cell indicates the clock cycle the pair is calculated in. Each column denotes the next *power_of_two* (line 2). Notice the occasional non linear change in the x positions (columns 2 & 3).

Table 1. Index Calculation Table

(0,1)	(0,2)	(0,4)	(0,8)	(0,16)		Clock Cycle
(2,3)	(1,3)	(1,5)	(1,9)	(1,17)		1
(4,5)	(4,6)	(2,6)	(2,10)	(2,18)		2
(6,7)	(5,7)	(3,7)	(3,11)	(3,19)		3
(8,9)	(8,10)	(8,12)	(4,12)	(4,20)		4
(10,11)	(9,11)	(9,13)	(5,13)	(5,21)		5
(12,13)	(12,14)	(10,14)	(6,14)	(6,22)		6
(14,15)	(13,15)	(11,15)	(7,15)	(7,23)		7
(16,17)	(16,18)	(16,20)	(16,24)	(8,24)		8
(18,19)	(17,19)	(17,21)	(17,25)	(9,25)		
(20,21)	(20,22)	(18,22)	(18,26)	(10,26)		
(22,23)	(21,23)	(19,23)	(19,27)	(11,27)		
(24,25)	(24,26)	(24,28)	(20,28)	(12,28)		
(26,27)	(25,27)	(25,29)	(21,29)	(13,29)		
(28,29)	(28,30)	(26,30)	(22,30)	(14,30)		
(30,31)	(29,31)	(27,31)	(23,31)	(15,31)		

Algorithm Remarks In line 4, the change in the y index from the x is simple. It is the current $power\_of\_two$ shifted right by 1.

Next consider the logic starting at line 5. A variable, groups, is set to indicate that the pairs are grouped together in a single clock cycle as opposed to indexes that contain a jump.

The difference in x coordinates from each clock cycles is defined as Δx. This depends on the groups variable whether the jump is 2 or 1. The change in the x coordinate is not constant. To handle the occasional non constant change in x, the variable $\Delta x'$ (line 22) is used. The offset is incremented whenever the current clock cycle equals Δy.

Similar to Δx, Δg is used to denote the jump in pairs in the same clock cycle. This is calculated in line 15. There is also a non constant change in Δg. $\Delta g'$ is used to compensate for the non constant Δg. It is shown in line 26.

Router Modules The BRAM router, write back controller, and ALU router have similar functions. They are responsible for separating the index / ALU pairs into BRAM pairs. The values are multiplexed to their appropriate port because of the BRAM port tag in the datagram. The write back controller also issues a signal to stall the pipeline for a cycle to write back into the BRAMs. The ALU router acts like the BRAM router except for the datagram tag. Recall the datagram contains a separate tag to send the pair to an ALU. Figure 9 shows how values can be routed using a butterfly circuit. Although different BRAMs' values are needed, they can still be fully utilized and routed to an ALU. The ALU performs the addition / subtraction and uses a similar process to store back into the BRAM.

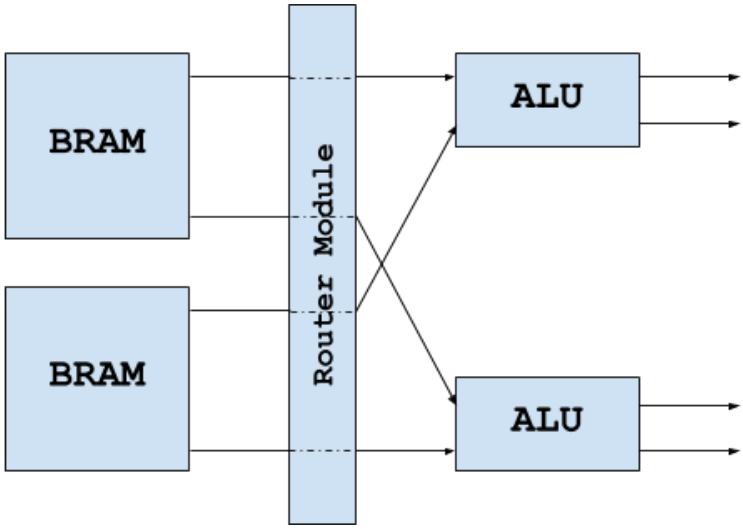

Fig. 9. Example of routing from BRAM to ALU units

BRAM Controller Users can specify the number of BRAMs to include in the module. The BRAM number must be a power of two. The default Xilinx dual port BRAM module was slightly modified to include our NTT parameters. The address and data port widths are automatically calculated based on the vector size and number of BRAMs. The BRAM modules are wrapped in a BRAM top module that contains the bus assignments.

ALU Cores The final module used is the ALU core. The number of ALU cores matches the number of BRAMs. The ALU cores are wrapped in an ALU top level module to assign the busses. The figure below shows the pipeline stages for the ALU core. The pipeline fills in six clock cycles. Note that the wb values in the multiplication stage are stored in a ROM and are not dynamically computed. This is because in our implementation we consider a fixed omega value. Moreover, we store the omega values in ROM. They are pre-calculated prior to execution.

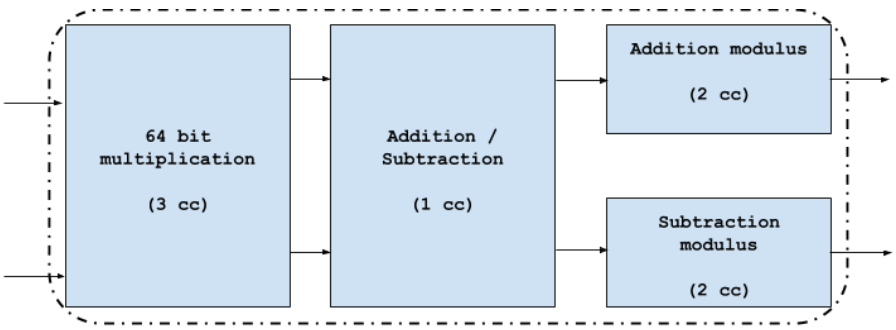

Fig. 10. ALU core pipeline

4 Results

In this section, we discuss timing and utilization of each element of our design. Our experiments for timing and overall usage were performed on a Xilinx Virtex-7 FPGA. Our device consisted of 693,120 logic cells and 3600 DSP slices. It has 433,200 LUTs, 865,400 FFs, and 1470 BRAMs available for use and has a system clock which runs at 200 MHz. The device also has a PCIe connector allowing it to be used for high performance applications such as hardware acceleration.

4.1 Fast Modulus

The metrics we focused on to assess our two fast modulus reductions were clock cycles to completion, timing, and hardware utilization. Each of these metrics tell a difference story, and each metric needs to be taken into account when used in our final larger design. The primary goal for each design is to beat timing, which is a 5ns clock period based on the 200 MHz clock speed of our FPGA. We are then most concerned about utilization and how much space the designs require. For the fast modulus especially, space is a big concern because we need to instantiate multiple copies of each block.

Table 2. Performance of 128-bit Fast Modulus vs 256-bit Fast Modulus

Modulus Size (bit)	Timing (clk cycles)	Frequency (MHz)	LuT Util.	FF Util.
128	2	307.7	192 (0.04%)	129 (0.01%)
256	3	229.7	467 (0.11%)	266 (0.03%)

Our final results can be seen in Table 2. For both 128-bit fast modulus and 256-bit fast modulus, we went through many iterations to decrease the clock cycle count and to meet timing requirements. We were able to meet the 200MHz frequency with both designs, though 256-bit fast modulus does take longer to complete and require more resources. The 128-bit Modulus Reduction copmletes in 6.500ns while the 256-bit Modulus Reduction completes in 13.059ns. We expected to see these results because 256-bit fast modulus is working with much more data and much larger numbers. The hardware utilization for each module is also very low which is very beneficial to us.

4.2 64-point NTT

We focus on the same metrics for 64-point NTT as we do for the fast modulus reductions. In this section, the performances of our two 64-point NTT method as described at the bottom of Section 3.2 is discussed. Again, the primary goal is to meet timing, and only considering hardware utilization afterwards. In Table 3, we reference the two methods based on their declaration in Section 3.2.

Table 3. Performance of 64-point NTT

Vector Size (bit)	Method	Timing (clk cycles)	Frequency (MHz)	LuT Util.	FF Util.
64	Mod First	72	204.6	73,487 (16.95%)	54,695 (6.31%)
	Add First	69	185.8	74,354 (17.15%)	55,205 (6.37%)
1024	Mod First	1152	204.6	73,487 (16.95%)	54,695 (6.31%)
	Add First	1104	185.8	74,354 (17.15%)	55,205 (6.37%)

The results of the two methods are very similar, as seen in Table 3. Unfortunately, only the 128-bit fast modulus method meets timings even though it takes 3 more clock cycles to complete. The Mod First method takes a total of 352.8ns to complete, whereas the Add First method takes 372.6ns to complete. Both methods have similar hardware utilization numbers which was surprising to us.

The Mod First method uses 64 256-bit fast modulus blocks and a single 128-bit fast modulus block. The Add First method only requires a single 256-bit fast modulus block. Based on the results of our standalone fast modulus reduction, we can see that the 256-bit method is much more complex. With slightly more LuT and FF utilization, but much less fast modulus blocks, the utilization of the Add First method happen primarily in the routing of many 256-bit numbers. The Mod First method has a much better time routing 64-bit numbers and thus is able to meet our timing requirements.

Since we only designed a 64-point NTT module, in order to do a 1024-bit NTT, we need to run the 64-point NTT 16 times. We are reading from BRAMs and writing to BRAMs which means that there doesn't need to be new routing hardware. As a result, the hardware utilization remains the same. For timing, we simply multiply the number of clock cycles by 16 in order to estimate completion timing. The Mod First method would complete in 5629.8ns while the Add First method would complete in 5941.7ns. It is clear that as the NTT vector size increases, the Mod First method performs better.

4.3 Iterative NTT

Table 4 shows the iterative NTT results for the Full NTT. This means the full power of 2 iteration is done. As a disclaimer, the module still needs some tinkering. However, we do not believe these results to change significantly. We vary the size of n and b and observed how the unit performs. When n is increased, the cycle count increases linearly. When b is increased, the resource usage is increased and the cycle count is decreased. It becomes increasingly difficult to route the design as the functional units increase. This causes the core clock speed to decrease. The area of the design is relatively small compared to the FPGA.

We tested two different combinations of functional units. A functional unit contains an ALU and BRAM. 4 units or less hits our target frequency of 200MHz. The resource usage stays the same for varying vector length sizes. This is because the BRAM utilization is increased but the combinational logic stays the same. The 8 unit test yielded a max frequency of around 179 MHz. We assume the clock speed linearly decreases with additional functional units.

4.4 Summation and Estimation

In this section, we estimate the results of the combined 64-point NTT and Iterative NTT. When combining the 64-point NTT and the Iterative NTT, we use the 64-point NTT in place of power of two iterations 2 through 64 in the Iterative NTT. This means we can skip the first six iterations of the Iterative NTT just by performing 64-point NTT. The Iterative NTT takes over after the 64-point NTT completes, starting with power of two equals 128.

We calculated the estimated performance of a shortened Iterative NTT in Table 5. In order to compute a 1024 vector length NTT using a single 64-point, we need to perform the 64-point NTT 16 times. By increasing the number of 64-point blocks to 4, we can decrease the number of 64-NTT operations to 4 times. The 128-1024 Iterative NTT is estimated by taking a proportional number of iterations for the number of clock cycles. For example, 128-1024 NTT is four power of 2 iterations (i.e. 128, 256, 512, 1024) while the full 1024 Iterative NTT is 10 power of 2 iterations. Thus we estimated the number of clock cycles of 128-1024 Iterative NTT as needing 4/10ths of the number of clock cycles.

In Table 5, we use the results of Iterative NTT using 4 BRAMs. To estimate the combined timing, we simply add the 64-point NTT results with the 128-1024 Iterative NTT results. We estimate two versions of the combined NTT,

Table 4. Performance of Iterative NTT for a Full NTT

Vector Length (n)	BRAMs (b)	Timing (clk cycles)	Frequency (MHz)	LuT Util.	FF Util.	DSP Util.
32	4	80	200	4670 (1.60%)	4315 (0.71%)	64 (2.29%)
64	"	136	"	"	"	"
128	"	256	"	"	"	"
256	"	496	"	"	"	"
512	"	976	"	"	"	"
1024	"	1936	"	"	"	"
1024	8	1294	179	9233 (3.04%)	8585 (1.41%)	128 (4.57%)
1024	16	654	160*	18351 (6.04%)	17125 (2.82%)	256 (9.14%)
1024	32	334	140*	36587 (12.1%)	34205 (5.63%)	512 (18.3%)
1024	64	174	120*	73059 (24.1%)	68365 (11.3%)	1024 (36.6%)

*Assume linear decrease in clock speed.

one using a single 64-point NTT and one using 4 64-point NTT blocks. As you can see, the results of the combined NTT using 4 64-point blocks is almost 50% faster in terms of the number of clock cycles. On the other hand, using just a single NTT does not make a significant difference in terms of performance. Only a few clock cycles are saved. The 1024 Iterative NTT takes 9680ns to complete while the combined NTT takes 5315ns to complete. Unfortunately, the cost of the speed up is a signficant increase to the hardware utilization. The speed up of using 4 64-point NTTs increases LuT utilization and FF utilization by over 30 times. Using a single 64-point becomes horribly inefficient because the increase in hardware utilization only saves a few clock cycles.

We also estimated the performance gains of the combined NTT against the Iterative NTT for different BRAM usage. In Table 6, we see that the combined NTT using a single 64-point block is only faster than the standard Iterative NTT using 4 BRAMs. Using 4 64-point NTTs is faster up to 16 BRAMs. At 32 BRAMs, the Iterative NTT is more efficient. This means that the Iterative NTT actually scales wells in terms of BRAM usage. At 16 BRAMs, the combined NTT using 4 64-point blocks is just over 500ns faster in terms of execution time. Where as at 64 BRAMs, the iterative NTT is twice as fast.

The hardware utilization is heavily sacrificed for the increase in speed since the 64-point NTT requires a lot of resources. For a low BRAM iterative NTT module, speed can be gained by using more 64-point NTT blocks. However, as the number of BRAMs increases, the number of 64-point NTT modules also decreases as the hardware limit becomes an issue. Note that utilizing more BRAMs is more efficient even in terms of speed. Using 64 BRAMs in the iterative NTT

Table 5. Performace of Combined NTT using Iterative NTT with 4 BRAMs

Vector Length (n)	NTT Method	Timing (clk cycles)	Frequency (MHz)	LuT Util.	FF Util.
64	1 64-point	1152	204.6	73,487 (16.95%)	54,695 (6.31%)
64	4 64-point	288	204.6	293948 (67.80%)	218780 (25.24%)
128 - 1024	Iterative NTT	775	200	4670 (1.60%)	4315 (0.71%)
1024	Iterative NTT	1936	200	4670 (1.60%)	4315 (0.71%)
1024	Combined w/ 1 64-point	1927	200	78157 (18.55%)	59010 (7.02%)
1024	Combined w/ 4 64-point	1063	200	298618 (69.40%)	277790 (32.26%)

Table 6. Performace of Combined 1024-bit NTT based on Number of BRAMs

BRAMs (bit)	Timing (clk cycles)	Frequency (MHz)	Execution Time (ns)	LuT Util.	FF Util.
4	1936	200	9680.0	4670 (1.60%)	4315 (0.71%)
	1927	200	9635.0	78157 (18.55%)	59010 (7.02%)
	1063	200	5315.0	298618 (69.40%)	223095 (25.95%)
8	1294	179	7233.5	9233 (3.04%)	8585 (1.41%)
	1670	179	9335.3	82720 (19.99%)	63280 (10.88%)
	806	179	4505.5	303181 (70.84%)	227365 (26.65%)
16	654	160*	4087.5	18351 (6.04%)	17125 (2.82%)
	1414	160*	8837.5	91838 (22.99%)	71820 (15.45%)
	550	160*	3437.5	312299 (73.84%)	235905 (28.06%)
32	334	140*	2384.8	36587 (12.1%)	34205 (5.63%)
	1286	140*	9182.0	110074 (29.05%)	88900 (24.61%)
	422	140*	3013.1	330535 (79.90%)	252985 (30.87%)
64	174	120*	1449.4	73059 (24.1%)	68365 (11.3%)
	1222	120*	10179.3	146546 (41.05%)	127754 (42.91%)
	358	120*	2982.1	367007 (91.90%)	287145 (36.54%)

*Assume linear decrease in clock speed

The first row of each section is the Iterative NTT performance. The second row of each section is the combined NTT performance using 1 64-point NTT. The third row of each section is the combined NTT performance using 4 64-point NTTs.

is more than 5 times faster than using 4 BRAMs with 4 64-point NTT blocks. These results are assuming that the hardware routing can be done. The number of DSP slices available allow the iterative NTT to scale well.

Full NTT Multiplication For a complete NTT operation, two forward NTT operations are needed. The result of those two forward NTT operations are multiplied together and then a single inverse NTT is performed. The total number of clock cycles required is computed by the addition of two forward NTT operations with an inverse NTT operation. It is important to note that the forward NTT operations can use a combined NTT while the inverse NTT operation must be a standard iterative NTT in terms of clock cycles. In Table 7, we estimate the execution time of the full NTT Multiplication. Again, the results prove that the iterative NTT scales much better than using multiple 64-point NTTs.

Table 7. Timing for Full NTT Multiplication

BRAMs (bit)	# 64-point	Timing (clk cycles)	Frequency (MHz)	Execution Time (μs)
4	0	5808	200	29.04
	1	5790	200	28.95
	4	4062	200	20.31
8	0	3882	179	19.41
	1	4634	179	25.90
	4	2906	179	16.24

The first row of each section uses only Iterative NTT performance. The second row of each section is the combined NTT performance using 1 64-point NTT. The third row of each section is the combined NTT performance using 4 64-point NTTs.

We have not added any comparisons to previous works because we were not able to find any fair comparisons. Some of the previous research that we read had results based on GPU or software performance. Additionally, the hardware that previous research has been on is dated at the time of our findings. For those hardware research results that we did see, the design choices were different from the ones we choose. Some research had a heavy focus on hardware utilization reduction rather than performance increase, while other research drastically increased hardware utilization for performance gains. Furthermore, differences in vector size and bit size of the NTT can change results heavily.

5 Future Work and Conclusion

This project was a good start to creating a scalable NTT module. There is room for improvement in our design. For example, we can save clock cycles in the 64 point by adjusting the way it is integrated with the iterative module. Also, the iterative module still exhibits errors in some computations. We would have to hunt for these edge cases and see what causes the module to produce inaccurate results.

In post-quantum cryptography, the NTT is a commonly used yet expensive operation. NTT operations are used over 600 times in each post-quantum algorithm in the NIST competition[2][3]. FPGAs can be used to implement the operation and provide speedups to execution time. We exploited parallelization of the NTT to create a hardware accelerator. A dedicated 64 point module was combined with a generic iterative module. The unit is capable of a complete polynomial multiplication on the order of 20,000 nanoseconds. Our project was focused on a generic model that can be scaled according to desired specifications. This allows the NTT to be implemented in a variety of applications from simple microcontrollers to high end servers.

References

1. Aysu, Aydin & Patterson, Cameron & Schaumont, Patrick. (2013). Low-cost and area-efficient FPGA implementations of lattice-based cryptography. Proceedings of the 2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013. 81-86. 10.1109/HST.2013.6581570. https://rijndael.ece.vt.edu/schaum//pdf/papers/2013hostb.pdf
2. Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. (2019). National Institute of Standards and Technology. https://csrc.nist.gov/projects/ post-quantum-cryptography/round-2-submissions
3. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: Ntru prime: reducing attack surface at low cost. Cryptology ePrint Archive, Report 2016/461 (2016). http://eprint.iacr.org/2016/461
4. Nayuki. "Number-Theoretic Transform (Integer DFT)." Project Nayuki, 7 June 2017, www.nayuki.io/page/number-theoretic-transform-integer-dft.
5. A. Emerencia. (2007). Multiplying huge integers using Fourier transforms. http://www.cs.rug.nl/ ando/pdfs/Ando_Emerencia_multiplying_huge_integers_using_fourier_transforms_paper.pdf.
6. Emmart, Niall & Weems, Charles. (2011). High Precision Integer Multiplication with a GPU Using Strassen's Algorithm with Multiple FFT Sizes. Parallel Processing Letters. 21. 359-375. 10.1142/S0129626411000266.
7. Longa, P., Naehrig, M.: Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. Cryptology ePrint Archive, Report 2016/504 (2016). https://eprint.iacr.org/2016/504.pdf
8. Chen, D.D., Mentens, N., Vercauteren, F., Roy, S.S., Cheung, R.C.C., Pao, D., Verbauwhede, I.: High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems. Cryptology ePrint Archive, Report 2014/646 (2014). https://eprint.iacr.org/2014/646.pdf
9. Mert, A.C., Ozturk, E., Savas, E.: Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture. Cryptology ePrint Archive, Report 2019/109 (2019). https://eprint.iacr.org/2019/109.pdf
10. T. Poppelmann, T. Oder, and T. Guneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. Cryptology ePrint Archive, Report 2015/382 (2015). https://eprint.iacr.org/2015/382.pdf
11. J. W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 19(90):297–301, 1965.
12. Slade, George. (2013). The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation. https://www.researchgate.net/publication/235995761_The_Fast_Fourier_Transform_in_Hardware_A_Tutorial_Based_on_an_FPGA_Implementation
13. Giles, Martin. (2019). Explainer: What is a quantum computer? How it works, why it's so powerful, and where it's likely to be most useful first. MIT Technology Review. https://www.technologyreview.com/2019/01/29/66141/what-is-quantum-computing/
14. Lyubashevsku, Vadim. (2016). Preparing for the Next Era of Computing With Quantum-Safe Cryptography. SecurityIntelligence. https://securityintelligence.com/preparing-next-era-computing-quantum-safe-cryptography/

15. Boutin, Chad. (30 January 2019). NIST Reveals 26 Algorithms Advancing to the Post-Quantum Crypto 'Semifinals'. National Institute of Standards and Technology. https://www.nist.gov/news-events/news/2019/01/nist-reveals-26-algorithms-advancing-post-quantum-crypto-semifinals
16. Public-key cryptography. (n.d). Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Public-key_cryptography ([Online; accessed 10-March-2020])
17. Post-quantum cryptography. (n.d). Wikipedia, the free encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Post-quantum_cryptography ([Online; accessed 14-February-2020])

Appendix A 128-Fast Modulus Verilog Code

```
// compute (2^32)(b+c)-a-b+d mod p
// pipeline into two stages
// checks for overflow
module fast_mod(
    input [31:0] a,
    input [31:0] b,
    input [31:0] c,
    input [31:0] d,
    input clk,
    output reg [63:0] z
    );

    parameter p = 64'hffffffff00000001;

    reg [64:0] ztemp1 = 0;

    always @ (posedge clk) begin
    //first stage
        ztemp1 <= {(b+c), 32'b0} + d - a - b;

    //second stage (checking for overflow)
        if((ztemp1) > p)
            z <= ztemp1 - p;
        else
            z <= ztemp1;
    end

endmodule
```

Appendix B 256-Fast Modulus Verilog Code

```
module fast_mod_256(
    input [31:0] a,
    input [31:0] b,
    input [31:0] c,
    input [31:0] d,
    input [31:0] e,
    input [31:0] f,
    input [31:0] g,
    input [31:0] h,
    input clk,
    output reg [63:0] z
    );
```

```
    parameter p = 64'hffffffff00000001;

    reg  [34:0] afg = 35'b0;
    reg  [34:0] bch = 35'b0;
    reg  [34:0] cd = 35'b0;
    reg  [34:0] ef = 35'b0;
    reg  [65:0] ztemp = 66'b0;
    reg  [65:0] ztemp2 = 66'b0;
    reg  [34:0] bch2 = 35'b0;
    reg  [34:0] ef2 = 35'b0;

    always @ (posedge clk) begin
        afg <= (a+f+g);
        bch <= (b+c+h);
        cd <= (c+d);
        ef <= (e+f);

        if({cd, 32'b0} > {afg, 32'b0})
            ztemp <= p + p + {afg - cd, 32'b0} + bch - ef;
        else if(cd > afg)
            ztemp <= p + {afg - cd, 32'b0}+ bch - ef;
        else
            ztemp <= {afg - cd, 32'b0} + bch - ef;

        if(ztemp > (p+p))
            z <= ztemp - p - p;
        else if(ztemp > p)
            z <= ztemp - p;
        else
            z <= ztemp;

    end
endmodule
```

Appendix C Row Calculation Verilog Code

```
module rowcalc(
    input clk,
    input [63:0] a,
    input [7:0] w,
    output [63:0] out
    );

    wire [63:0] aw;
```

```
    reg [63:0] temp = 0;
    reg [63:0] temp2 = 0;
    reg [127:0] result = 0;
    reg [255:0] b;
    reg done;
    reg [8:0] count = 0;

    always @ (posedge clk) begin
        b <= a << w;
    end

    fast_mod_256 m1 (.a(b[255:224]), .b(b[223:192]), .c(b[191:160]),
                     .d(b[159:128]), .e(b[127:96]), .f(b[95:64]),
                     .g(b[63:32]), .h(b[31:0]), .clk(clk), .z(aw));

    always@ (posedge clk) begin
        if (count == 71)
            count <= 0;
        else
            count <= count + 1'b1;
    end

    always@ (posedge clk) begin
        if (count < 7)
            result <= aw;
        else
            result <= result + aw;
    end

    fast_mod m2 (.a(result[127:96]), .b(result[95:64]), .c(result[63:32]),
                 .d(result[31:0]), .clk(clk), .z(out));

endmodule
```

Appendix D Index Calculation Python Code

```
if __name__ == "__main__":
    #Constants
    BRAMS = 2
    VECTOR_LEN = 32
    trans_size_array = [2, 4, 8, 16, 32]

    cycles = int(VECTOR_LEN / (2*BRAMS))

    #Variables
```

```
groups = 0
pairs_per_cycle = 0
delta_x = 0
delta_y = 0
delta_g = 0
delta_g_bonus_offset = 0
delta_x_bonus_offset = 0
x_pos = 0
y_pos = 0

for trans_size in trans_size_array:
    print("----- Next trans_size: " + str(trans_size) + " -----")

    # Assign groups variable. Groups are defined as the
    # number of adjacent pairs
    if(BRAMS == VECTOR_LEN / 2):
        groups = 1
    elif((trans_size >> 1) >= VECTOR_LEN / BRAMS):
        groups = 2
    else:
        groups = 1

    # Assign the delta_x variable. Delta_x is defined as the change
    # between pairs in different cycles
    if((trans_size >> 2) >= VECTOR_LEN / (2*BRAMS) or trans_size == 2):
        delta_x = 2
    else:
        delta_x = 1

    # Assign the delta_y variable. Delta_y is equal to trans_size >> 1.
    # Included for readability
    delta_y = trans_size >> 1

    # Assign the delta_g variable. Delta_g is defined as the change
    # between pairs in the same cycle
    if(BRAMS == VECTOR_LEN / 2):
        delta_g = 1
    elif((trans_size >> 1) == VECTOR_LEN / BRAMS):
        delta_g = 2*(trans_size >> 1)
    else:
        delta_g = VECTOR_LEN / BRAMS

    delta_x_bonus_offset = 0
    pairs_per_cycle = int(BRAMS / groups)
```

```
        # Begin calculating index pairs
        # i is the current cycle
        # j is the current BRAM
        for i in range(0, cycles):
            print("----- Next clock cycle -----")
            delta_g_bonus_offset = 0

            # Handle the occasional non linear change of delta_x
            if(i % (trans_size >> 1) == 0 and trans_size != 2 and i != 0):
                delta_x_bonus_offset = delta_x_bonus_offset + (trans_size >> 1)

            for j in range(0, pairs_per_cycle):
                # Handle the occasional non linear change of delta_g
                if( delta_g*j%(trans_size >> 1) == 0 and \
                    j != 0 and delta_g < trans_size):
                    delta_g_bonus_offset = delta_g_bonus_offset + (trans_size >> 1)

                x_pos = int(delta_g * j + i * delta_x + \
                    delta_g_bonus_offset + delta_x_bonus_offset)
                y_pos = int(x_pos + delta_y)

                if(groups == 1):
                    print("(" + str(x_pos) + ", " + str(y_pos) + ")")
                elif(groups == 2):
                    print("(" + str(x_pos) + ", " + str(y_pos) + ")")
                    print("(" + str(x_pos + 1) + ", " + str(y_pos + 1) + ")")
```

Appendix E Index Calculator Verilog Code with 4 Functional Units

```
‘timescale 1ns / 1ps

module index_calc_datapath(
    input clk,
    input [2:0] current_state,
    input [9:0] trans_size,
    output reg [6:0] i,
    output reg [1:0] j,
    output [9:0] x_pos1,
    output [9:0] y_pos1,
    output [9:0] x_pos2,
    output [9:0] y_pos2,
    output [9:0] x_pos3,
    output [9:0] y_pos3,
    output [9:0] x_pos4,
```

```
output [9:0] y_pos4,
output reg result_active,
output reg groups
);
parameter BRAMS = 'd4;
parameter VECTOR_LEN = 'd1024;
parameter   IDLE        = 2'b000,
            VAR_ASSIGN  = 2'b001,
            i_LOOP      = 2'b010,
            j_LOOP      = 2'b011,
            ADD_I       = 3'b100;

reg [1:0] delta_x;
reg [9:0] delta_y;
reg [9:0] delta_g;
reg [10:0] delta_g_bonus_offset;
reg [10:0] delta_x_bonus_offset;
reg [9:0] pos_reg;
reg [6:0] i_counter;
reg [1:0] j_counter;

wire [9:0] x_pos2_bus;
wire [9:0] y_pos2_bus;
wire [9:0] x_pos4_bus;
wire [9:0] y_pos4_bus;
wire [1:0] j3_bus;

index_calculator CALC1(
    .delta_g(delta_g),
    .j(0),
    .i(i),
    .delta_x(delta_x),
    .delta_y(delta_y),
    .delta_g_bonus_offset(delta_g_bonus_offset),
    .delta_x_bonus_offset(delta_x_bonus_offset),
    .x_pos(x_pos1),
    .y_pos(y_pos1)
);

index_calculator CALC2(
    .delta_g(delta_g),
    .j(1),
    .i(i),
    .delta_x(delta_x),
    .delta_y(delta_y),
```

```
    .delta_g_bonus_offset(delta_g_bonus_offset),
    .delta_x_bonus_offset(delta_x_bonus_offset),
    .x_pos(x_pos2_bus),
    .y_pos(y_pos2_bus)
);

index_calculator CALC3(
    .delta_g(delta_g),
    .j(j3_bus),
    .i(i),
    .delta_x(delta_x),
    .delta_y(delta_y),
    .delta_g_bonus_offset(delta_g_bonus_offset),
    .delta_x_bonus_offset(delta_x_bonus_offset),
    .x_pos(x_pos3),
    .y_pos(y_pos3)
);

index_calculator CALC4(
    .delta_g(delta_g),
    .j(3),
    .i(i),
    .delta_x(delta_x),
    .delta_y(delta_y),
    .delta_g_bonus_offset(delta_g_bonus_offset),
    .delta_x_bonus_offset(delta_x_bonus_offset),
    .x_pos(x_pos4_bus),
    .y_pos(y_pos4_bus)
);

always @(posedge clk) begin
    if(current_state == IDLE) begin
        groups <= 0;
        pairs_per_cycle <= 0;
        delta_x <= 0;
        delta_y <= 0;
        delta_g <= 0;
        delta_g_bonus_offset <= 0;
        delta_x_bonus_offset <= 0;
        i <= 0;
        j <= 0;
        i_counter <= 0;
        j_counter <= 0;
        result_active <= 1'b0;
    end
```

```
else if(current_state == VAR_ASSIGN) begin
    // set groups variable
    if(BRAMS == VECTOR_LEN >> 1) begin
        groups <= 0;
    end
    else if((trans_size >> 1) >= (VECTOR_LEN / BRAMS)) begin
        groups <= 1;
    end
    else begin
        groups <= 0;
    end
    // set delta_x variable
    if(((trans_size >> 2) >= VECTOR_LEN / (BRAMS << 1)) ||
        trans_size == 2) begin
        delta_x <= 2;
    end
    else begin
        delta_x <= 1;
    end
    // set delta_y variable
    delta_y <= trans_size >> 1;
    // set the delta_g variable
    if(BRAMS == VECTOR_LEN >> 1) begin
        delta_g <= 1;
    end
    else if((trans_size >> 1) == VECTOR_LEN / BRAMS) begin
        delta_g <= trans_size;
    end
    else begin
        delta_g <= VECTOR_LEN / BRAMS;
    end
    // set the delta_x_bonus_offset
    delta_x_bonus_offset <= 0;
    // set the pairs per cycle
    pairs_per_cycle <= BRAMS / groups;
end
else if(current_state == i_LOOP) begin
    result_active <= 1'b0;
    delta_g_bonus_offset <= 0;
    //calculate the delta_x value
    if(i_counter == (trans_size >> 1) && i != 0 && trans_size != 2) begin
        delta_x_bonus_offset = delta_x_bonus_offset + (trans_size >> 1);
        i_counter <= 0;
    end
end
```

```
        else if(current_state == j_LOOP) begin
            //calculate the delta_g value without using modulus
            if(j != 0 && delta_g < trans_size) begin
                delta_g_bonus_offset <= delta_g_bonus_offset + (trans_size >> 1);
            end
            result_active <= 1'b1;
        end
        else if(current_state == ADD_I) begin
            i <= i + 1;
            i_counter <= i_counter + 1;
            result_active <= 1'b0;
        end
    end

    assign x_pos2 = (!groups) ? x_pos2_bus : x_pos1 + 1;
    assign y_pos2 = (!groups) ? y_pos2_bus : y_pos1 + 1;
    assign x_pos4 = (!groups) ? x_pos4_bus : x_pos3 + 1;
    assign y_pos4 = (!groups) ? y_pos4_bus : y_pos3 + 1;
    assign j3_bus = (!groups) ? 2 : 1;

endmodule
```

Appendix F ALU Core Verilog Code

```
module alu_core(
    input clk,
    input [63:0] wb,
    input [73:0] op1,
    input [73:0] op2,
    output [73:0] add_pair,
    output [73:0] sub_pair
    );

    reg [127:0] add_pair_to_mod;
    reg [127:0] sub_pair_to_mod;
    reg [127:0] mul_reg;
    // Buffer the operand for 3cc
    reg [73:0] op1_reg_cc1;
    reg [73:0] op1_reg_cc2;
    reg [73:0] op1_reg_cc3;
    // Buffer the ram positions for 5cc
    reg [9:0] op2_ram_pos_cc1;
    reg [9:0] op2_ram_pos_cc2;
    reg [9:0] op1_ram_pos_cc3;
    reg [9:0] op2_ram_pos_cc3;
```

```
reg [9:0] op1_ram_pos_cc4;
reg [9:0] op2_ram_pos_cc4;
reg [9:0] op1_ram_pos_cc5;
reg [9:0] op2_ram_pos_cc5;
wire [127:0] mul_result;

// 2 cc to complete
fast_mod ADD_MOD (
    .a(add_pair_to_mod[127:96]),
    .b(add_pair_to_mod[95:64]),
    .c(add_pair_to_mod[63:32]),
    .d(add_pair_to_mod[31:0]),
    .clk(clk),
    .z(add_pair[63:0])
);
// 2 cc to complete (in parallel with other mod)
fast_mod SUB_MOD (
    .a(sub_pair_to_mod[127:96]),
    .b(sub_pair_to_mod[95:64]),
    .c(sub_pair_to_mod[63:32]),
    .d(sub_pair_to_mod[31:0]),
    .clk(clk),
    .z(sub_pair[63:0])
);

// 3 cc to complete
mul_64bit M1(
    .a(op2[63:0]),
    .b(wb),
    .clk(clk),
    .result(mul_result)
);

always @(posedge clk) begin
    //  MUL cc1
    op1_reg_cc1 <= op1;
    op2_ram_pos_cc1 <= op2[73:64];
    //  MUL cc2
    op1_reg_cc2 <= op1_reg_cc1;
    op2_ram_pos_cc2 <= op2_ram_pos_cc1;
    //  MUL cc3
    op1_reg_cc3 <= op1_reg_cc2;
    op2_ram_pos_cc3 <= op2_ram_pos_cc2;
    //  MUL FINISHED do Addition (1cc)
    add_pair_to_mod <= op1_reg_cc3[63:0]+mul_result;
```

```
        sub_pair_to_mod <= op1_reg_cc3[63:0]-mul_result;
        op1_ram_pos_cc3 <= op1_reg_cc3[73:64];
        op2_ram_pos_cc4 <= op2_ram_pos_cc3;
        //  MOD cc1
        op1_ram_pos_cc4 <= op1_ram_pos_cc3;
        op2_ram_pos_cc4 <= op2_ram_pos_cc3;
        //  MOD cc2
        op1_ram_pos_cc5 <= op1_ram_pos_cc4;
        op2_ram_pos_cc5 <= op2_ram_pos_cc4;
        //  MOD finished
    end

    assign add_pair[73:64] = op1_ram_pos_cc5;
    assign sub_pair[73:64] = op2_ram_pos_cc5;

endmodule
```