Open-Source IC Design for Post-Quantum Cryptography

Alexander Demirs, Bruce Huynh, Matthew Wong
March 3rd, 2023

Abstract

The development and research into quantum computers endanger the security of current cryptographic algorithms. To prepare for this, the National Institute of Standards and Technology (NIST) has created a competition to develop post-quantum cryptography algorithms. Post-quantum cryptography lends itself to algorithms that are resistant to attacks by quantum computers. In this project, our team analyzes the final round of NIST algorithm submissions and selects the CRYSTALS-Dilithium algorithm to attempt an ASIC design implementation. We discuss the mathematics behind CRYSTALS-Dilithium and the design decisions of available FPGA implementations. Specifically, we focus on the Decomposer module of the design, which optimizes the output that the number theoretic transform (NTT) block to send to other modules. Small changes made to the register sizes of the Decomposer module result in a critical path 196% smaller and a 40.12% size reduction. Additionally, synthesis scripts are used to optimize our design further for area or speed; our new design is tested against the original design to verify the correctness of the optimizations. Other optimizations and recommendations are suggested at the end of the report to assist future scholars in exploring post-quantum ASIC design or the CRYSTALS-Dilithium algorithm.

Table of Contents

I. Introduction

A. What is Post-Quantum Cryptography?

Throughout human history, secure and private data and communication have been central to information systems and are prone to adversarial interactions. In ancient times, cryptology was essentially synonymous with cryptography, which is the study of plaintext, ciphertext, encryption, and decryption. However, contemporary cryptology is based on algorithms and heavily involves mathematics, as the cryptanalytic difficulty determines the strength of the algorithm. The cornerstone of cryptographic security is the idea of "computational hardness". Advances in computing technology cause the algorithms to be constantly reevaluated and reworked as cryptanalysis becomes increasingly more efficient. Most contemporary algorithms are theoretically breakable; however, they are computationally secure, because computers currently cannot break them in a practical timeframe.
The two worldwide, most commonly-used asymmetric cryptographic algorithms are Rivest-Shamir-Adleman (RSA) cryptography and Elliptic Curve Cryptography (ECC).

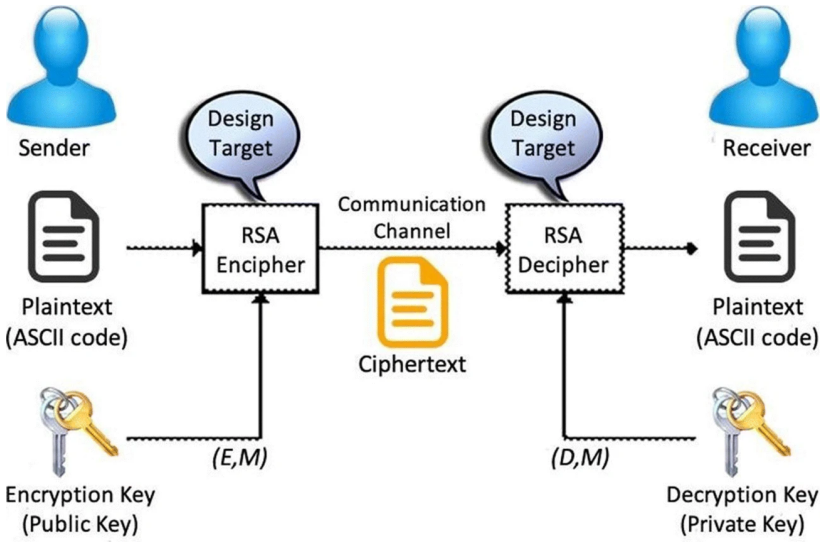


Fig. 1. Rivest-Shamir-Adleman (RSA) Cryptography [1]

RSA utilizes a large integer factoring problem consisting of two large prime integers (private keys) and a public key integer [2]. People can use the public key to encrypt new messages; however, only people who know the value of the private keys can decrypt messages. There are currently no existing algorithms for quickly factoring the product of the two large primes, but as computers advance, this will inevitably change.

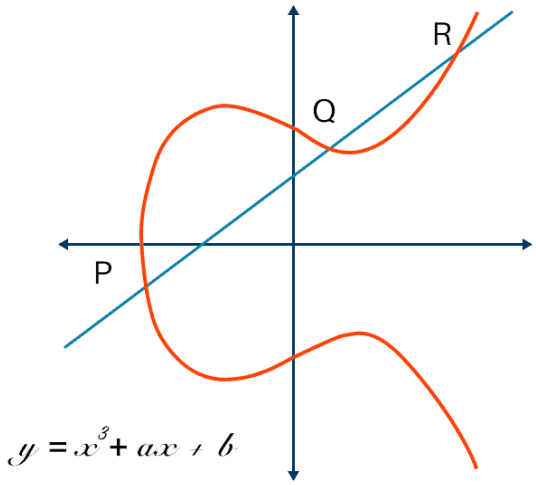


Fig. 2. Elliptic Curve Cryptography [3]

ECC uses elliptic curves and the discrete logarithm connecting to a publicly known base point value [2]. There is currently no timely and practical way to solve this problem with modern technology. Governments, institutions, and businesses that control vital global structures rely on these kinds of cryptography when dealing with sensitive/private data. Any electronic infrastructure over the internet or between institutions will eventually need a revamped form of cryptography when technology advances far enough to break the current cryptographic standards.

Modern-day research into quantum mechanics has allowed scientists and engineers to create quantum computers. Quantum computers are an evolved version of the modern computer that exploits quantum mechanics to improve their processes. Instead of using the traditionary binary bits of 1's and 0's, quantum computers use "qubits".

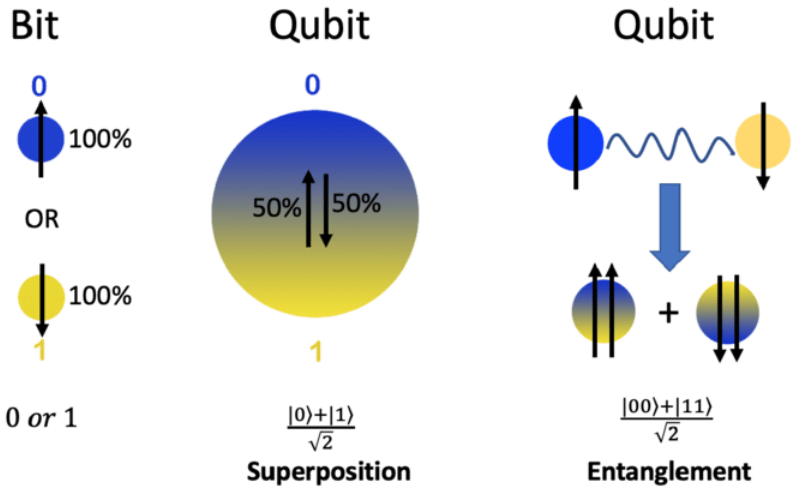


Fig. 3. Bits vs Qubits [4]

Qubits function similarly to common binary bits, but they allow for superposition states, which allow the qubit to be in both states at the same time. Simultaneous state superposition is a

central element of quantum computing [5]. In addition, qubits can become entangled, linking two qubits together in a state of matching superposition (qubit pairing) [4].

These computers will be magnitudes more efficient than current computers and would consequently be able to perform algorithmic processes many times faster. Security issues arise from these powerful quantum computers being able to break through modern cryptographic algorithms. Asymmetric cryptography, algorithms that employ public-private keys, will be made obsolete by the development of quantum computers. These algorithms protect governments, institutions, and vital global infrastructure. The current cryptographic standards that apply integer factorization, discrete logarithm, and the elliptic-curve discrete logarithm will indubitably break under attack by quantum computers. According to the Department of Homeland Security, a real risk involves malicious parties gathering sensitive intelligence or data encrypted by today's standards and later decrypting it using future quantum computers [6]. Some engineers predict that quantum computers will be able to break all the currently available public key schemes in approximately the next twenty years [7]. Consequently, researchers are studying and testing the capabilities of post-quantum cryptography.

Post-Quantum Cryptography is the research, development, and application of cryptographic algorithms secure against cryptanalytic attacks by quantum computers. For example, Shor's Algorithm is a post-quantum algorithm that can solve the integer factorization problem in polynomial time [8]. This endangers anything protected by RSA and any encryption based on integer prime factorization. To combat the development of stronger decryption tools, cryptographic standards must remain secure; a designation of this is named NP-hardness. NP-hardness is a classification of problem difficulty that has computational complexity higher than the capability of a "nondeterministic Turing machine in polynomial time" [9]. Some examples of cryptography that have a higher level of complexity (quantum-resistant) are symmetric algorithm, hash-based, lattice-based, multivariate, code-based, supersingular elliptic curve isogeny, and symmetric key quantum-resistant cryptography. Each of these styles contains a complex mathematical problem that quantum computers are unable to solve in a realistic amount of time.

In a post-quantum computing world, global security depends on cryptographic primitives being quantum resistant. Although quantum technology provides us with newfound computing power and unlocks doors to global technological advancement, quantum computing raises many security issues that must be addressed. To prepare for this, the National Institute of Standards and Technology (NIST) has initiated a competition that looks for public-key cryptographic algorithms that will be able to withstand attacks from quantum computers.

B. NIST Competition

In early 2017, the National Institute of Standards and Technology (NIST) released a "Call for Proposals" for researchers and institutions to submit potential algorithms. The National Institute of Standards and Technology and the Department of Homeland Security are still working together to facilitate research into information security in the coming age of quantum computing. For this reason, the NIST has run many rounds of the "Post-Quantum Cryptography Standardization" competition. Teams consisting of researchers strive to create better quantum-resistant public-key cryptographic algorithms in preparation of future developments in quantum computing. Each of the three complete rounds of submissions since 2017 has laid the foundation for the next round's submissions. Since the beginning of the competition, many

groups have merged and/or broken up depending on the analysis and reception of the algorithm in their submissions.

For some of the algorithms, the baseline mathematics has been proven unable to prevent quantum computers from decrypting. Because quantum computing is still in development, some current submissions may probably be phased out of the competition. The NIST anticipates the final round of submissions will end in 2024. Whichever post-quantum cryptographic scheme remains after the last round of submissions will replace many current cryptographies. The expected result will be an overhaul of all current cryptographic standards. For the current iteration of public key cryptography, it took about twenty years to implement onto critical infrastructure. This is why NIST pushes to create post-quantum cryptographic algorithms before the widespread use of quantum computers.

C. Our Goals

These algorithms are highly complex and deeply researched. Almost all of the algorithms are performed in software or on a Field-Programmable Gate Array (FPGA). Our team aimed to implement a post-quantum cryptographic algorithm on an Application-Specific Integrated Chip (ASIC) and additionally optimize it for the ASIC environment. We planned to analyze the constraints, timings, or memory/storage of the chip to optimize for size. The study of the algorithm along with the design process of an ASIC implementation of the entire algorithm was too difficult for our team, so the scope of the project was reconsidered. After researching and dissecting the algorithm for months, we decided to focus on optimizing the hardware implementation for a specific module. By reducing the complexity of the project, the feasibility of the open-source hardware implementation greatly increased.

II. Analyzing NIST Options

A. Available Submissions

Initially, our team selected our algorithm from the round 3 submissions of the competition. These algorithms included: Classic McEliece, CRYSTALS-KYBER, NTRU, SABER, CRYSTALS-Dilithium, FALCON, and Rainbow. Our team studied each of these algorithms and created a table (Appendix A) that contains a summary of each of them and links to any resources or documentation.

Classic McEliece [10] is a public-key cryptosystem employing a random binary Goppa code. It is designed for one-wayness against chosen plaintext attack security.

Crystals-Kyber [11] applies indistinguishability under chosen ciphertext attack/adaptive chosen ciphertext attack key encapsulation mechanism (KEM). This algorithm applies a Learning with errors encryption.

A different algorithm by NTRU or Number Theory Research Unit (originally known as Number Theorists 'R' Us) uses a partially correct probabilistic public-key encryption scheme [12]. It is a lattice-based alternative to RSA and elliptic curve cryptography, but unlike pre-quantum schemes, it is resistant to attacks by Shor's algorithm.

SABER is a "Mod-LWR"-based KEM that contains a version of the Fujisaki-Okamoto transform [13]. The name "ModLWR" stands for a module learning-with-rounding problem.

Crystals-Dilithium is a lattice-based digital signature algorithm that uses the "Fiat-Shamir with Aborts" approach [14]. It is known to be extremely strong against chosen message attacks.

Fast Fourier Lattice-based compact signatures over NTRU (FALCON) is a post-quantum signature scheme that applies the hash-and-sign techniques using NTRU-style lattices. FALCON is based on lattice-based signature schemes and short integer solution problems (SIS) over NTRU lattices. It yields higher security, compactness, speed, scalability, and RAM usage [15].

Rainbow is one of the algorithms in the multivariate public key cryptosystems. It employs the Oil-Vinegar signature scheme, which applies multivariate quadratic systems and algebraic geometry. Rainbow provides small signatures and efficient signature generation and verification via simple operations over small finite fields. It is quantum-resistant and NP-hard because quantum computers do not have an advantage over normal computers when solving multivariate systems of equations [16] [17].

Table 1 below is an abbreviated version of the table in Appendix A.

Table I
Summary of NIST Round 3 Submission Algorithms

Post-Quantum Cryptography Algorithms
Classic McEliece [10]
• Uses a public-key cryptosystem with a random binary Goppa code   ○ Ciphertext with errors are sometimes used • Private-key decodes ciphertext and identifies/removes errors • Algorithm also known as a KEM (Key Encapsulation Method) • Designed for *OW-CPA security* (One-Wayness Against Chosen Plaintext Attack)
CRYSTALS-KYBER [11]

- Uses two cryptographic primitives (overlaps with CRYSTALS-Dilithium):
- *Kyber*, which uses IND-CCA2 (Indistinguishability under chosen ciphertext attack/adaptive chosen ciphertext attack) - also a KEM
- Uses LWE encryption (Learning With Errors)
- *Dilithium*, a strongly EUF-CMA-secure digital signature algorithm (Existential Unforgeability under Chosen Message Attack)
- Module lattices can withstand quantum attacks
- The only operations required for all security levels are variants of Keccak

NTRU - Number Theory Research Unit [12]

- NTRU originally described as "partially correct probabilistic public key encryption scheme (partially correct PPKE)" but can be made deterministic and perfectly correct
- KEM by Hoffstein, Pipher, Silverman
- Lattice based alternative to RSA and elliptic curve cryptography
- Only feasible attack is lattice-based attack → system of equations in an attempt to solve for any chosen polynomial
- Quantum variants of sieve algorithms studied
- Improvements rely on unit-cost superposition queries to classical memory (QRAM), the best claimed operation count is $2^{(0.265...+o(1))\cdot b}$

SABER [13]

- Lattice based, designed to offer better resistance to quantum computers
- Built on the hardness of the "Module learning with rounding problem" (Mod-LWR)
- Saber.PKE to Saber.KEM using a version of Fujisaki-Okamoto transform (no idea what this is)
- LWR reduces the randomness required (½) the amount of LWE-based schemes, reduces bandwidth
- Module structure provides flexibility by reusing one core component for multiple security levels
    - Parameters: n, L → degree 256 of the polynomial ring
        - Rank L of the module (determines the dimension of the lattice problem)
        - Dimension of lattice increase -> better security, less correctness
    - q,p,T → moduli powers of 2 → 2^{xq}
        - Higher the parameter, lower security, higher correctness
- Security in the Quantum Random Oracle Model
    - Limited to encryption scheme and KEM, no signature scheme included

CRYSTALS-Dilithium [14]

Key Generation:

- Begin with a number, q, and integers (k, l). Generate a $k \times l$ matrix with entries in the previously described field, F. q is defined by some formula, and the degree of the polynomial in the maximal ideal is $n := 256$. We then sample random "secret" key

vectors, (s_1, s_2), and the key is generated according to the rule: $t = As_1 + s_2$. So t is an affine combination.

Signing Procedure:
- The signing algorithm generates a masking vector of polynomials with coefficients less than a given γ and a challenge is created as the hash of the message and w where w is computed after some operations from the masking vector and is then further algebraically manipulated.

Verification:
- Verifier computes w' in the higher-order bits and accepts if all of the coefficients of the vector-part are less than a certain $\gamma - \beta$.

Implementations:
- Has a documented implementation on Intel Core-i7 6600U (Skylake) CPU.
- Furthermore, there is a C reference implementation, located on a GitHub repository

FALCON [15]

- Fast-Fourier Lattice-based algorithm.
- Similar to Crystals-Dilithium
- True Gaussian Sampling for negligible leakage
- NTRU lattices allow for short signatures and public keys
- O(n log n)
- uses 30 kilobytes of ram
- Falcon, submitted to NIST Post-Quantum Cryptography Project in 2017, is based on lattice-based signature schemes and short integer solution problem (SIS) over NTRU lattices.
- It offers high security, compactness, speed, scalability, and RAM economy.

Rainbow [16] [17]

Background:
- Based on the Unbalanced Oil and Vinegar scheme (UOV)
- Uses random multivariate quadratic systems
- Rainbow, designed in 2004, is based on multivariate quadratic systems and algebraic geometry. It was selected as one of the three NIST Post-quantum signature finalists in 2020 and offers very small signatures and efficient signature generation and verification through the use of simple operations over small finite fields.
- Rainbow offers very small signatures.
- efficient signature generation and verification

After in-depth research into each of these algorithms, our team created criteria to evaluate which cryptographic algorithm would be best for this project.

B. Our Criteria

Our team analyzed each of the cryptographic algorithms and scored them against a rubric based on the amount/quality of documentation, available code, ease of understanding, and ease of implementation. Documentation was our primary scoring criterion for the selection of a cryptographic algorithm. For a successful open-source study or hardware implementation, there must exist a solid foundation of research and source material. In addition, available code was the second heaviest criterion. For our team to feasibly study or plan a successful hardware implementation, a strong codebase would support any of the well-documented algorithms. Code was weighted less than documentation, as the only thing required was essentially the port and block diagrams. Furthermore, we weighed the ease of understanding and ease of implementation equally when comparing the cryptographic algorithms. Many algorithms were eliminated due to their lack of a hardware model either in Verilog or VHDL, and others were factored out due to the large size of their existing or possible implementations. Algorithm simplicity and implementation theoretical simplicity were also considered when selecting the algorithms to eliminate.

C. CRYSTALS-Dilithium

After much deliberation, our team decided to select CRYSTALS-Dilithium, as there is a large amount of available documentation, the algorithm is Lattice-based (which simplifies familiarization), and there are FPGA implementations available to reference. Particularly, the CRYSTALS-Dilithium NIST submission report was the foundational material for our understanding of the cryptographic algorithm. The report presents the algorithm as a digital signature scheme proven “secure under chosen message attacks based on the hardness of lattice problems over module lattices” [14]. As our project is geared toward an open-source implementation, we frequently reference publications and research projects from other institutions. The primarily referenced external project is “High-Performance Hardware Implementation of CRYSTALS-Dilithium” by Luke Beckwith, Duc Tri Nguyen, and Professor Kris Gaj of George Mason University. The team at George Mason University (GMU) produced a hardware implementation design for the NIST Round 3 post-quantum digital signature algorithm. The project consists of a combined architecture with key and signature generation, signature verification, and security-level runtime options [18]. GMU’s design is the lowest latency, small-area FPGA hardware implementation of CRYSTALS-Dilithium. Our project tightens its scope of design, focusing on optimizing the decomposer unit from the FPGA project for our open-source ASIC design.

III. CRYSTALS-Dilithium

The CRYSTALS-Dilithium scheme is a post-quantum cryptographic algorithm submitted to the NIST challenge and was designed with four primary goals: a simple implementation, conservative parameters, minimal key and signature size, and modularity. CRYSTALS-Dilithium implements a lattice-based scheme that evaluates the shortest vector in a modular space [14].

The Shortest Vector Problem (SVP) is a widely-used problem in the field of lattice cryptography. The SVP asks for the shortest non-zero vector in a lattice, where the length of a vector is defined by a chosen norm.

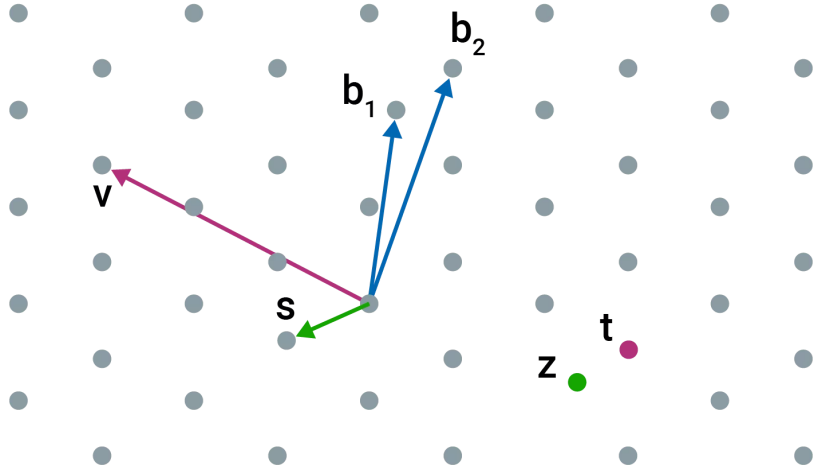


Fig. 4. Shortest Vector Problem [20]

The SVP is parametrized, where the underlying hardness is based on finding the shortest vector with the standard not smaller than the chosen parameter. Lattices are structures that consist of a set of vectors in a modular multidimensional space that is closed under addition and scalar multiplication. Lattices are defined by their basis, which is a set of linearly independent vectors that span the entire lattice. The SVP is an NP-hard problem, it is computationally infeasible to find the exact solution in polynomial time for large inputs. This makes SVP a useful tool for cryptography, as finding the solution to SVP would allow an attacker to break several lattice-based cryptographic systems.

A. Dilithium Key Generation

The key generation algorithm in the given system is used to create a public key and a secret key. It begins by generating a $k \times L$ matrix A, where each entry is a polynomial in a specific polynomial ring,

$$R_q = Z[X]/<X^n + 1>.$$

This ring has specific parameters: a prime, $q = 2^{23} - 2^{13} + 1$ and $n = 256$.

Next, the algorithm generates two secret key vectors, s_1 and s_2, where each coefficient of these vectors is a small element from the polynomial ring R_q with small coefficients. Finally, the public key is calculated as the sum of the product of matrix A and vector s_1, and vector s_2

$$t = A*s_1 + s_2.$$

All the mathematical operations in this system are done using the polynomial ring R_q.

Algorithm 1: Dilithium Key Generation

Input: $\zeta \in \{0,1\}^{256}$

Output: $pk = (\rho, t_1)$, $sk = (\rho, K, tr, s_1, s_2, t_0)$

1 $(\rho, \sigma, K) \leftarrow H(\zeta)$

2 $A \leftarrow$ExpandA(ρ) $\quad s_1, s_2 \leftarrow$ExpandS$(\sigma)$

3 $t \leftarrow A \cdot s_1 + s_2$

4 $(t_0, t_1) \leftarrow$ Power2Round(t, d)

5 $tr \leftarrow H(\rho || t_1)$

Fig. 5. Key generation algorithm steps [18]

B. Digital Signature Generation

Signing algorithms are used to create signatures for messages [21]. This signing algorithm starts by generating a masking vector of polynomials y with coefficients that are less than a certain value (γ_1). This value is specifically chosen to be large enough that the signature does not disclose the secret key (so the signing process is secure), but small enough that the signature cannot be easily forged. The algorithm then calculates an A_y and a W_1 as "high order" bits of the vector coefficient. Each coefficient in A_y can be expressed as

$$w = w_1 * 2_{y2} + w_0 \text{ where } |w_0| \leq y_2.$$

A challenge C is then the result of the hash of the message and of W_1, which results in a polynomial in the ring R_q that contains exactly 60 ±1's and has the rest of its values equal to 0. The distribution is the direct result of the small norm of C and the domain size of $D > 2^{256}$. Then, the first potential signature is calculated as $z = y + cs_1$.

However, z is withheld from the output at this stage, because the secret key would be vulnerable to extrapolation at this point. Rejection sampling is then used to prevent the dependence of z on the secret key. A parameter B is defined as the maximum number of coefficients of Cs_i. This causes B to $\leq 60n$. Coefficients larger than γ_1 - B in z are rejected and cause us to restart the signing procedure.

Additionally, the coefficient of the low-order bits of $A_z - c_t$ also cannot be larger than $\gamma_2 - \beta$. These verifications ensure the accuracy and security of the signature. The signing procedure continues to loop until both of these parameters are fulfilled, which is estimated to take an average of four to seven times.

Algorithm 2: Dilithium Signature Generation

Input: $sk = (\rho, K, tr, s_1, s_2, t_0), M \in \{0,1\}^*$
Output: $\sigma = (\hat{c}, z, h)$

$\mathbf{A} \leftarrow$ExpandA(ρ) $\quad \mu \leftarrow H(tr||M)$ $\quad \rho' \leftarrow H(K||\mu)$
$k \leftarrow 0$ $\quad abort \leftarrow 1$
while $abort$ **do**
  $abort \leftarrow 0$
  $y \leftarrow$ExpandMask(ρ', k)
  $w \leftarrow \mathbf{A} \cdot y$
  $w_1 \leftarrow$HighBits$(w, 2\gamma_2)$
  $\hat{c} \leftarrow H(\mu||w_1)$
  $c \leftarrow$SampleInBall$(\hat{c})$
  $z \leftarrow y + c \cdot s_1$
  $w_0 \leftarrow$LowBits$(w, 2\gamma_2)$
  if $||z||_\infty \geq \gamma_1 - \beta$ ***or*** $||w_0 - c \cdot s_2||_\infty \geq \gamma_2 - \beta$ **then**
    $abort \leftarrow 1$
  else
    $h \leftarrow$MakeHint$(w_1, w_0 - c \cdot s_2 + c \cdot t_0, 2\gamma_2)$
    if $||c \cdot t_0||_\infty \geq \gamma_2$ ***or*** $\sum h_i > \omega$ **then**
      $abort \leftarrow 1$
  $k = k + l$

Fig. 6. Signature generation steps [18]

C. Digital Signature Verification

The verification of the signature takes place in the form of a computation of W'_1 as the high-order bits of A_z-c_t. Then, it verifies that all the coefficients of z are smaller than γ_1 - B and C is a hash of the message and W'_1. This verification functions properly because

$$\text{HighBits}(Ay, 2\gamma_2) = \text{HighBits}(Ay - cs_s, 2\gamma_2).$$

For a signature to be valid, it must have

$$||\text{LowBits}||(Ay - cs_2, 2\gamma_2)||_{\text{infinity}} < \gamma_2 - B.$$

And since coefficients of cs_2 are smaller than B, the addition of Cs_2 does not add any carries and does not increase any low-order bit to a magnitude equal to or greater than γ_2.

Algorithm 3: Dilithium Signature Verification

Input: $pk = (\rho, t_1)$, $M \in \{0,1\}^*, \sigma = (\hat{c}, z, h)$
Output: Valid or Invalid

$\mathbf{A} \leftarrow$ExpandA(ρ)
$\mu \leftarrow H(H(\rho\|t_1)\|M)$
$c \leftarrow$SampleInBall$(\hat{c})$
$(w_1, w_0) \leftarrow$UseHint$(h, A \cdot z - c \cdot t_1 \cdot 2^d)$
if $||z||_\infty < \gamma_1 - \beta$ **&** $\hat{c} = H(\mu||w_1)$ **&** $\sum h_i \leq \omega$ **then**
  return Valid
return Invalid

Fig. 7. Signature verification steps [18]

IV. Focusing Our Scope

Our team found that the complexity of the project was very high, so we spent a lot of time deciding how to optimize or implement the CRYSTALS-Dilithium algorithm. We iterated through many ideas such as creating the design in a higher-level language to understand the architecture, converting the entire FPGA design, converting the polyarithmetic block to VHDL and then Verilog, and then finally finding a suitable block to convert.

A. Analyzing Higher-Level Code

When analyzing how to approach this algorithm, our team decided to analyze the higher-level code available in GMU's FPGA implementation. The researchers at GMU provided a hardware implementation in mostly Verilog and some VHDL, and they included software implementation in C++ that performed the same mathematic operations. By first analyzing the C++ code, we obtained a more fundamental and complete understanding of the overall architecture than we would have if we had analyzed only the Verilog code. We mapped all of the inputs and outputs and studied the intricacies of the functions. We then created a table containing all of the functions to reference them later. This table is available in Appendix B.

Initially, we considered creating our own high-level C/C++ implementation from the high-level reference implementation so that we could eventually create our own Verilog implementation. However, we realized that an implementation of the whole algorithm would be extremely difficult, so we pivoted the focus of our project.

B. VHDL to Verilog Conversion

When deciding how to change the design, our team focused our efforts on analyzing and converting the Keccak module. Keccak is currently the most secure hashing SHA-3 algorithm and sponge function, winning the 2012 NIST competition for the SHA-3 Cryptographic Hash Algorithm Competition [22]. To calculate the hashing value, first, determine the length of the message and add padding to it. The padding starts and ends with a '1' bit, and the rest of the bits in between are '0's. Then, the padded message is divided into n blocks, each with a length of r. The value of r depends on the hash length that was chosen. After that, a modulo operation is performed on the first padding block (P_0), which initially is all '0's. Subsequently, the focus shifts to the 24 rounds of the hash function, which consist of five functions: θ, ρ, π, χ, and ι. These rounds are repeated n times in the "absorb" function until reaching the "squeeze" function. The desired length of the output hash is determined at the beginning, so it is possible to extract that exact number of bits from the final value to obtain the complete hash [23] [24].

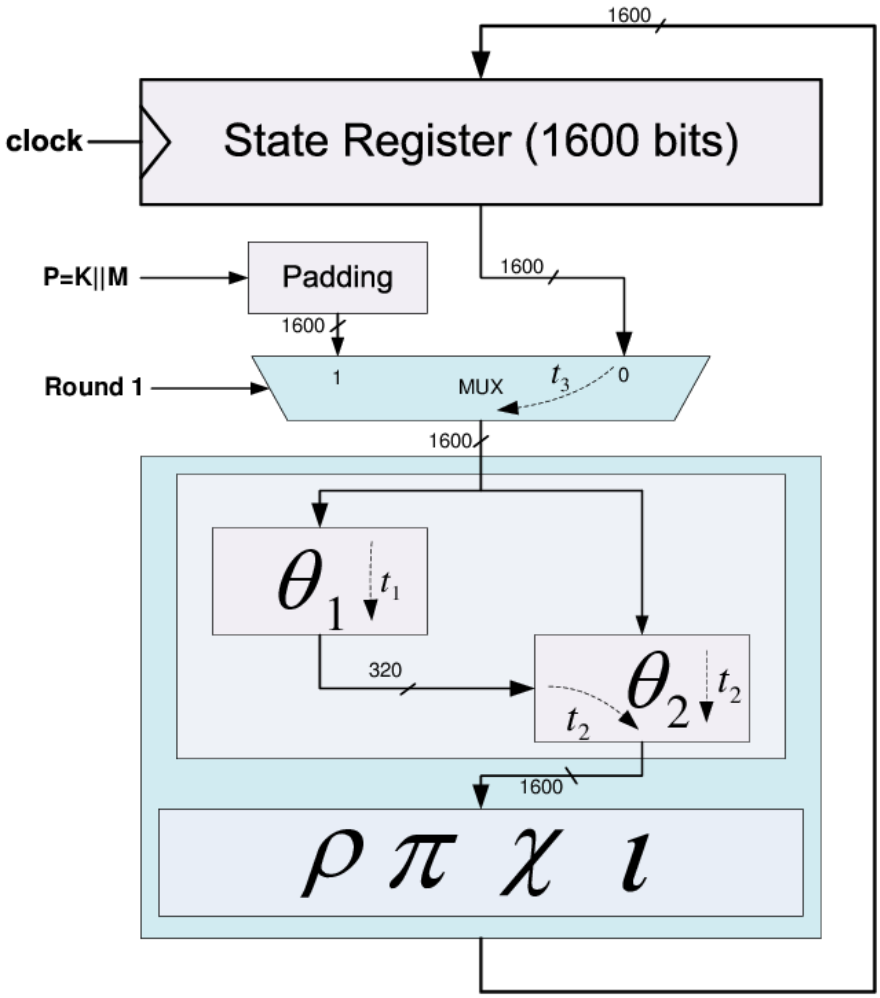


Fig. 8. Keccak algorithm high-level block diagram [23]

The designers at George Mason University created the FPGA design with Xilinx Vivado to target Xilinx FPGA boards. This allowed them to utilize both Verilog and VHDL in their design. The polyarithmetic modules in their design consisted of solely VHDL files. In an interview with Luke Beckwith of the GMU team, he stated that his decision was made because VHDL supported multidimensional arrays while Verilog did not. This posed a large issue for us at the time, as we could not integrate both of the languages into our ASIC design. We tried to convert the VHDL files into Verilog manually; however, since Verilog does not support multidimensional arrays, we lost the main functionality of the original files.

To overcome this, we used a tool called Design Compiler shell (DC shell) by Synopsys. This tool would allow us to convert the VHDL into a Verilog netlist with its standard cell library and combine that netlist with the rest of the Verilog modules. These standard cells contained Verilog descriptions of basic components such as gates (an example of this is in Fig. 9), and they had predefined physical properties like timing and power.

```
module AND2 (
```

```
      input A,
      input B,
      output Z
      );

assign Z = A & B;

endmodule
```

Fig. 9. AND gate

The Keccak modules were broken down into components like Fig. 9 were combined into a single file containing over 60,000 lines. Using that file, we were able to derive the area in the DC shell. This was when we reached a complication in our methodology. Creating a netlist for the VHDL components allowed us to run the whole design in purely Verilog, but when optimizing and testing, it would be extremely difficult. We could not debug or modify any of the converted code, as each module was broken down into smaller components. In addition, we could not easily study and comprehend the converted code. After this, we reevaluated our goals and searched for a module that would be feasible to work with.

C. FPGA vs ASIC design

One of the goals of this project was to convert a digital design originally made for an FPGA into a digital design created for an ASIC. In most undergraduate classes, students work with FPGAs due to their relatively low cost of entry and less need to optimize the design. One main benefit of working with an FPGA is that it is reconfigurable, allowing students trial and error with debugging their projects. On the contrary, an ASIC is not reconfigurable, and once it is fabricated, it cannot be changed. This means much more time is spent on the verification and optimization of the ASIC design.

Another large difference is how FPGAs and ASICs synthesize code. FPGAs turn a register transfer level (RTL) design into lookup tables (LUTs), which are combinational components that receive inputs and generate the corresponding outputs. Inside the LUTs is a truth table generated from the design that performs boolean logic to get the desired outputs. These LUTs allow a user to easily configure their design at the cost of consuming a greater amount of area. When an FPGA designer synthesizes their RTL design, the design gets transformed into a series of LUTs that handle combinational logic and sequential logic flip-flops. This design structure makes it ideal for prototyping and implementations that will potentially require many changes over time. For instance, the development of a hardware implementation of a cryptography algorithm will potentially change throughout the competition. This is the reason why the GMU team decided to go with an FPGA approach [18].

ASICs use standard cells from the same libraries to create their designs. These predesigned cells range from components such as gates and flip-flops to multiplexers, non-logical cells, and more [25]. These cells have known area, delay, and power consumption which is great for a designer looking to make an optimized design. When an ASIC designer synthesizes their RTL design, the design gets transformed into these cells, and a netlist of all the cells is created. Since these cells are predefined, they offer known information and are easier to optimize at the cost of less flexibility. Moreover, an ASIC does not have to endure the bloat of an

FPGA through its inflexible design. An FPGA must onboard a variety of hardware components and unused registers while ASIC features only specifically required parts. This means an equivalent ASIC will have a much smaller size, and consequently, this lends itself to reduced power consumption. Once a design is established, there is no option for future spontaneous editing and modifications.

There are many other design considerations when comparing ASICs and FPGAs. ASIC designs tend to use many clocks while FPGA designs usually feature fewer clocks [19]. This means that ASIC designers have to be wary of clock domain crossing in their designs. Latches are sometimes used in ASIC designs but should not be used in FPGA designs. This is because ASICs can be clocked at higher speeds and FPGAs use LUTs and flip-flops as their components. ASIC designs have more flexibility for sequentially combining gates and components together while FPGA designs prefer pipeline stages because of long latency.

A conversion of the FPGA implementation to an ASIC would have noticeable benefits such as efficient optimization of size and power. We studied the GMU team's FPGA implementation to find a suitable module to convert to an ASIC design.

D. Hardware Module Analysis

The implementation consists of eight modules, each interwoven with the memory and interconnected (Fig. 10). The modular design of GMU's FPGA implementation [18] makes it easier to understand and potentially convert it to an ASIC design. The implementation does key and signature generation with signature verification. This design uses limited DSP usage (multiplication only) because look-up tables (LUTs) are applicable for more basic arithmetic. In Fig. 10, the hardware implementation design for CRYSTALS-Dilithium is shown. Each of the modules is implemented to perform the cryptographic algorithm with minimal latency.

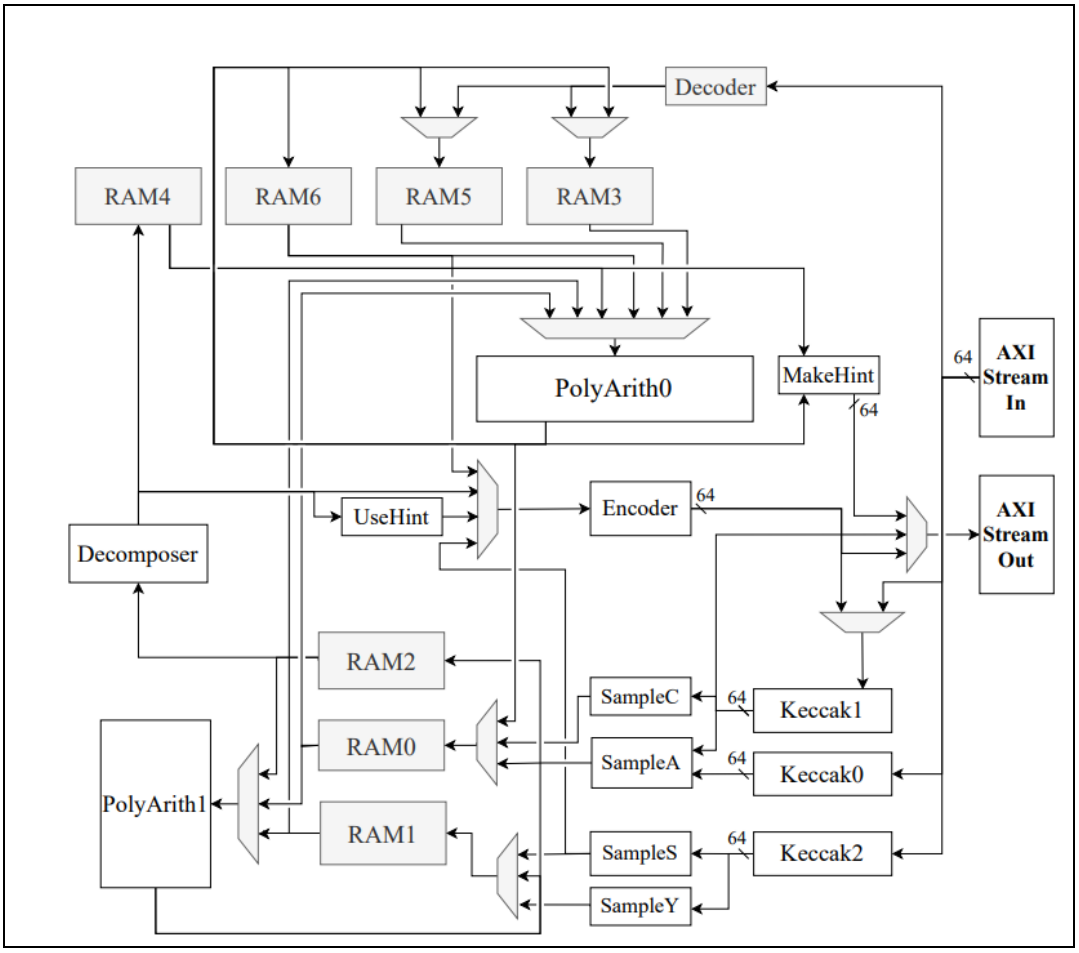


Fig. 10. Block Diagram Architecture of CRYSTALS-Dilithium [18]

The Polynomial Arithmetic unit executes standard polynomial addition, subtraction, multiplication, and Number Theoretic Transform (NTT) operations. It increases the speed of the polynomial multiplications. The design that GMU uses contains Barrett reductions and employs butterfly units for standard arithmetic. Within the "PolyArith" unit, there are four butterfly units that carry out simple arithmetic, Gentlemen-Sandle, and Cooley-Tukey operations. Using those four butterfly units in tandem, the design can handle four coefficients simultaneously, performing NTT in two layers each time the memory is accessed. To make sure the four coefficients are correctly pipelined, the design implements a module that contains a 4x4 matrix transposer.

In addition to the FIFO, the PolyArith unit contains an address resolver unit. It uses two 64x6 ROMs with LUTs so that the RAM can correctly map the address without changes when executing simple arithmetic or overwrite operations. This negates the need for shuffling and re-ordering, saving memory. The PolyArith module stores the two polynomial vectors within three BRAM units. Furthermore, GMU's design utilizes Keccak units; they are used for sampling and hashing matrixes.

Another few modules used in this design to simplify its operation are the decomposer, encoder, decoder, sample, and use/make hint units. Within the signature generation portion, the vector derived from the secret key and the message hash is decomposed. The high and low bits are split into two polynomial vectors and then mapped to coefficients. A Fisher-Yates shuffle [26] is applied to a polynomial to sample out a lower-length polynomial. This design, within a

key generation, requires the creation of a hint value that helps the verification portion recover missing bits. MakeHint and UseHint are applied in conjunction with the sampling units to recreate the value of the originally sampled polynomial. Once the hint is found, the resulting polynomial is encoded, comparing it with the original value before sampling to verify it.

After analysis of all of the modules, our team decided to tighten the scope of the project to the decomposer module. This module is simpler and contains more straightforward elements that allow for a straightforward and more in-depth study of one of the key parts of the algorithm.

V. Decomposer Module

A. Connection To Larger Architecture

The decomposer performs the separation of high-bits and low-bits and is frequently used in the digital signature verification of the algorithm. It allows us to save space in memory by extracting the important bits, but the missing data can be recovered as well. The MakeHint module uses the high-bits from the decomposer output, and the UseHint module directly uses the decomposer module for both high and low bits. UseHint runs the decompose function and compares it to the hint to verify the signature.

The decomposer module gets its data inputs from RAM2, which is the output from the PolyArith1 module that contains the NTT operation. UseHint pulls the value directly from the output of the decomposer. The decomposer output values are also stored in the RAM for usage in MakeHint to prepare that information for when the state changes. (Refer to Fig. 10)

B. Purpose

The decomposer unit requires the high-order bits of a given element in Z_q. The implementation of CRYSTALS-Dilithium utilizes a specific algorithm to perform this extraction, referred to as Decompose [14]. Given an element r in Z_q, decompose$_q$ writes r in the form:

$$r = r_1 a + r_0, \text{ where } a \text{ is chosen so that } ka = (q - 1) \text{ for some positive integer } k.$$

Here, r_1 represents the high-order bits of r, and r_0 represents the low-order bits of r. This decomposition gives

$$r_1 = (r - r_0)/a, \text{ with } r_0 = r \bmod a.$$

However, if $r - r_0 = q - 1$, such that $r_1 a = (q - 1) \Rightarrow r_1 = k$, then we round

$$r_1 = 0 \text{ and } r_0 = r_0 - 1.$$

This is done to prevent the high-order bits from changing by more than 1.

$\underline{\mathsf{Power2Round}_q(r, d)}$
$r := r \bmod^+ q$
$r_0 := r \bmod^\pm 2^d$
return $\big((r - r_0)/2^d, r_0\big)$

$\underline{\mathsf{MakeHint}_q(z, r, \alpha)}$
$r_1 := \mathsf{HighBits}_q(r, \alpha)$
$v_1 := \mathsf{HighBits}_q(r + z, \alpha)$
return $[\![r_1 \neq v_1]\!]$

$\underline{\mathsf{UseHint}_q(h, r, \alpha)}$
$m := (q - 1)/\alpha$
$(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
if $h = 1$ and $r_0 > 0$ **return** $(r_1 + 1) \bmod^+ m$
if $h = 1$ and $r_0 \leq 0$ **return** $(r_1 - 1) \bmod^+ m$
return r_1

$\underline{\mathsf{Decompose}_q(r, \alpha)}$
$r := r \bmod^+ q$
$r_0 := r \bmod^\pm \alpha$
if $r - r_0 = q - 1$
  then $r_1 := 0$; $r_0 := r_0 - 1$
else $r_1 := (r - r_0)/\alpha$
return (r_1, r_0)

$\underline{\mathsf{HighBits}_q(r, \alpha)}$
$(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
return r_1

$\underline{\mathsf{LowBits}_q(r, \alpha)}$
$(r_1, r_0) := \mathsf{Decompose}_q(r, \alpha)$
return r_0

Fig. 11. Decompose, MakeHint, UseHint algorithms [14]

C. Architecture and Layout

The Decomposer unit runs four instances of the `coeff_decomposer` module, allowing it to calculate the 96-bit value coeff_w in 24-bit chunks. Within these `coeff_decomposer` units is a `decomp_map1` unit which performs the calculation for the high-bits. The `coeff_decomposer` module then uses the high-bits from the `decomp_map1` to retrieve the low-bits. Each of these modules runs simultaneously to increase speed and efficiency.

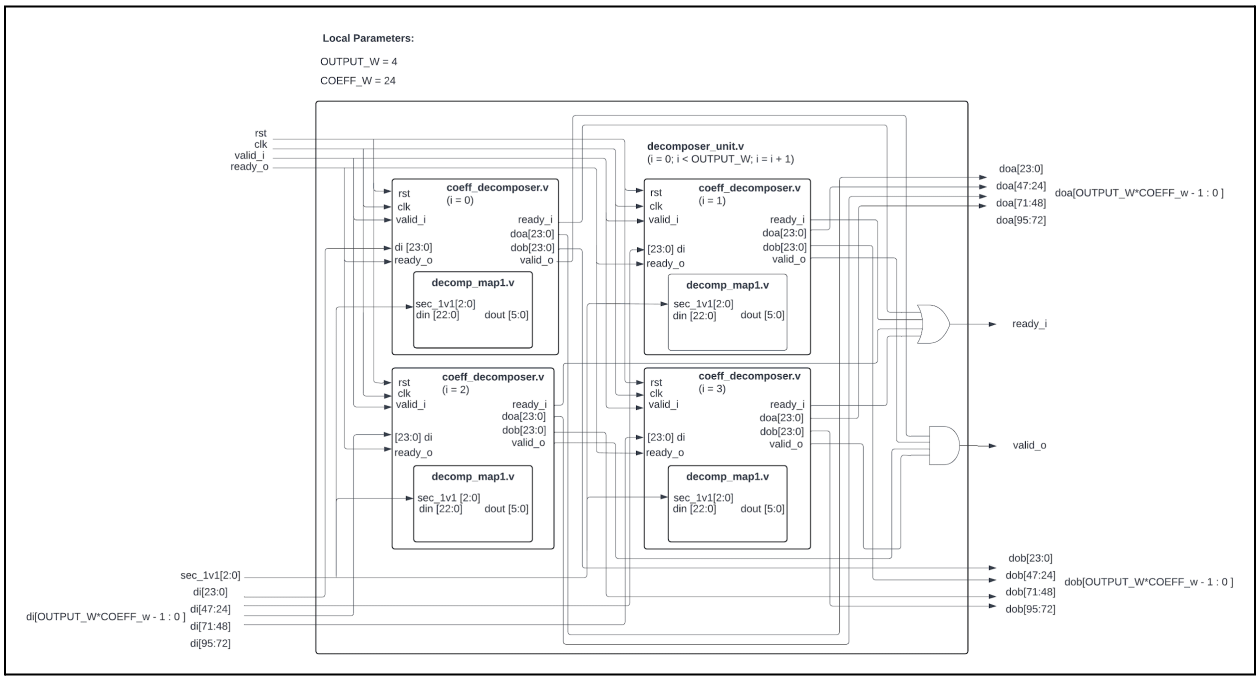


Fig. 12. Port diagram of the decomposer unit

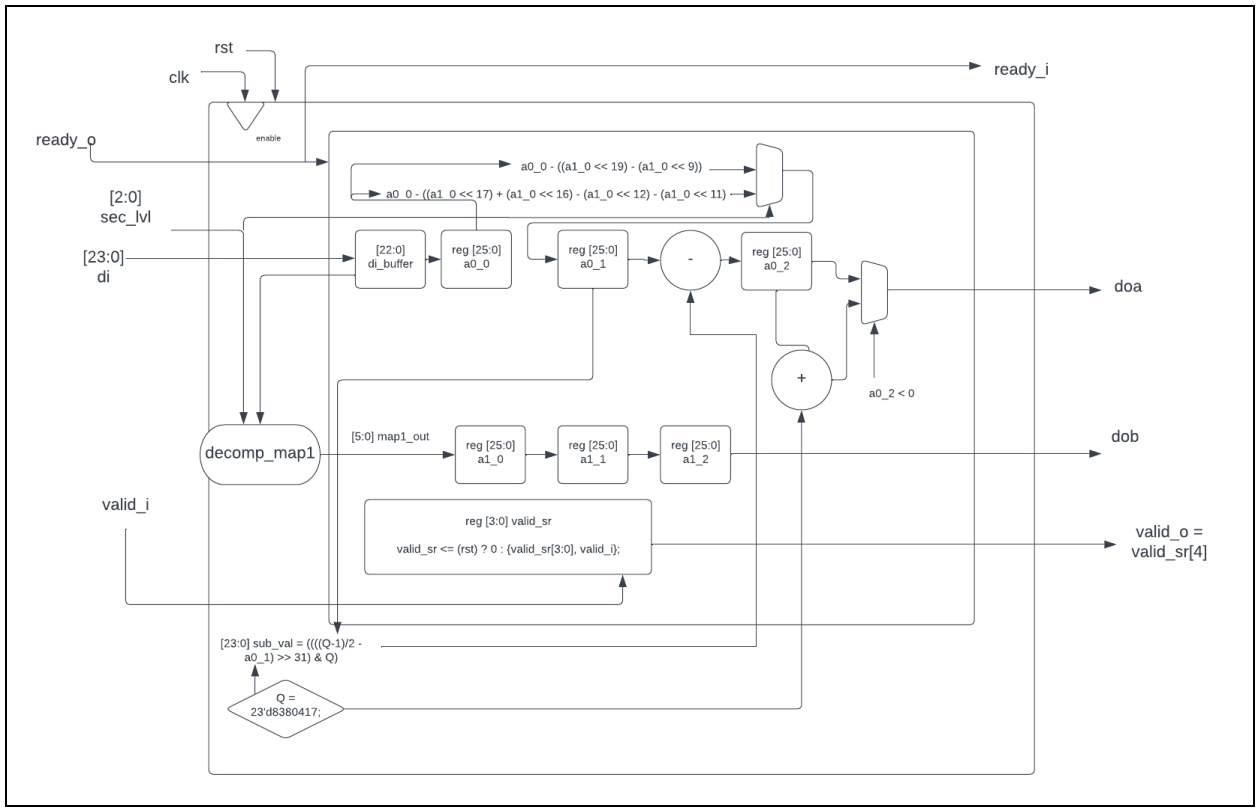


Fig. 13. Block diagram of the coeff_decomposer unit

The `coeff_decomposer` can be divided into a few core components: the local parameter registers, the looping data buffer, and the `coeff_map`.

D. Optimization

Optimizing this module was quite challenging, as we needed to modify the architecture in such a way that it would still perform the same function and match the exact same timing in the larger architecture. We analyzed possible changes to the block and found that the register sizes can be truncated to save area. In the `coeff_decomposer` module, there are six buffers that have a signed-bit length of 56. The maximum value that these buffers can take is 26 bits (1 extra bit for a signed bit). This maximum number of bits was calculated as the largest output of the decomposer map shifted 19 bits to the right. Maximum values for the buffer were also tested through our testbench (Appendix F), which resulted in a maximum of 26 bits.

The current size of the decomposer is quite large at over 80,000 μm^2. If we target the `coeff_decomposer` modules, we can reduce the size drastically since there are four of them in the decomposer unit. To view the area of the chip, we used Yosys [27], an open-source software that allows us to calculate and view the area of our design. For this design, we specifically used the Skywater 130nm standard cell library. To view timing, we used a software called OpenSTA and ran our design under a 100 MHZ clock (10,000 ps period). Here are all the variations of the decomposer module:

Table II
Table of our designs and their characteristics

	Critical Path (ps)	Area (μm²)	Number of Cells (sky130)
Regular Design	9280	81665.824	10209
Optimized Design	4720	48906.9056	5795
Speed-Centered Optimized Design	3200	54563.5808	6214
Area-Centered Optimized Design	3280	56816.992	9102

We see results that are nearly half the area of the original decomposer unit, specifically 59.88% smaller for the optimized design. We can verify that this is valid by noticing the changed number of cells. This means that the tool we are using does not automatically truncate unused bits. For the critical path, the optimization decreased the critical path by 196% of the original amount and increased the slack by 730% of the original slack.

Yosys has a library of scripts that can be used to optimize for specifically speed or area. We modified these scripts for our design and documented their results. Interestingly enough, the speed-centered optimized design requires smaller area than the area-centered optimized design. They are also both faster than the optimized design but are both larger in area. It is also interesting to note that the number of standard cells is much higher in the area-centered design compared to both the optimized and speed-centered designs.

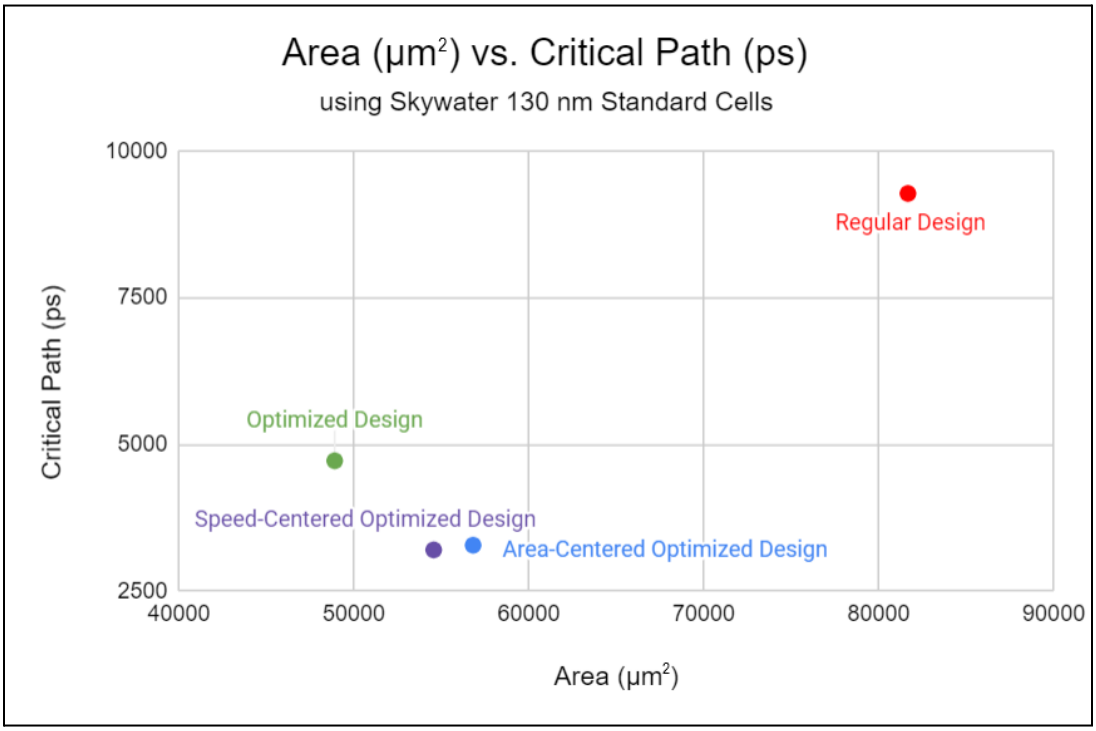


Fig. 14. Area (μm²) vs Critical Path (ps) using Skywater 130nm cells

When looking at the tradeoffs on the chart, we want to be close to (0,0) as possible, as we want the lowest critical path as well as the smallest area. The Pareto optimal designs would be the regular optimized design if we wanted a faster decomposer or the speed-centered design, as both are points on the Pareto optimal set [28]. Compared to the regular design, the speed-centered optimized design has a 290% smaller critical path. The speed-centered optimized design is overall better than the area-centered optimized design; therefore, the area-centered optimized design should not be considered, as it is pareto suboptimal.

Other optimizations could be made if the larger architecture is changed as well. For example, the buffer registers *a1_1* and *a1_2* only exist to delay the output by two additional clock cycles. They do not serve any purpose besides that.

To show the layout of the decomposer module, we used the OpenROAD software, which is an open-source electronic design automation (EDA) tool. This tool also helped us see the chip layout, the critical path, and other timing information.

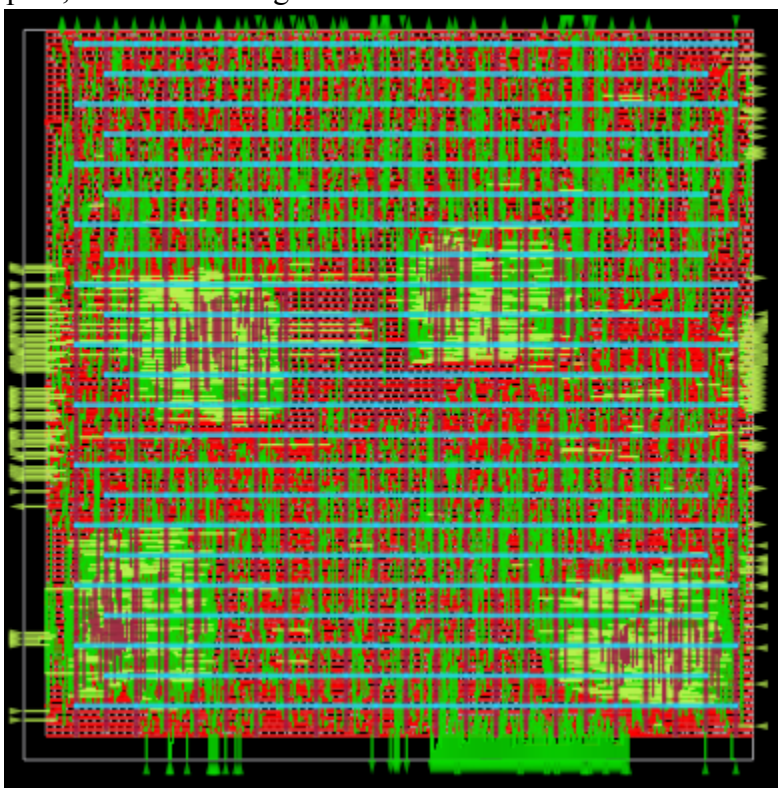


Fig. 15. Chip layout of the decomposer module

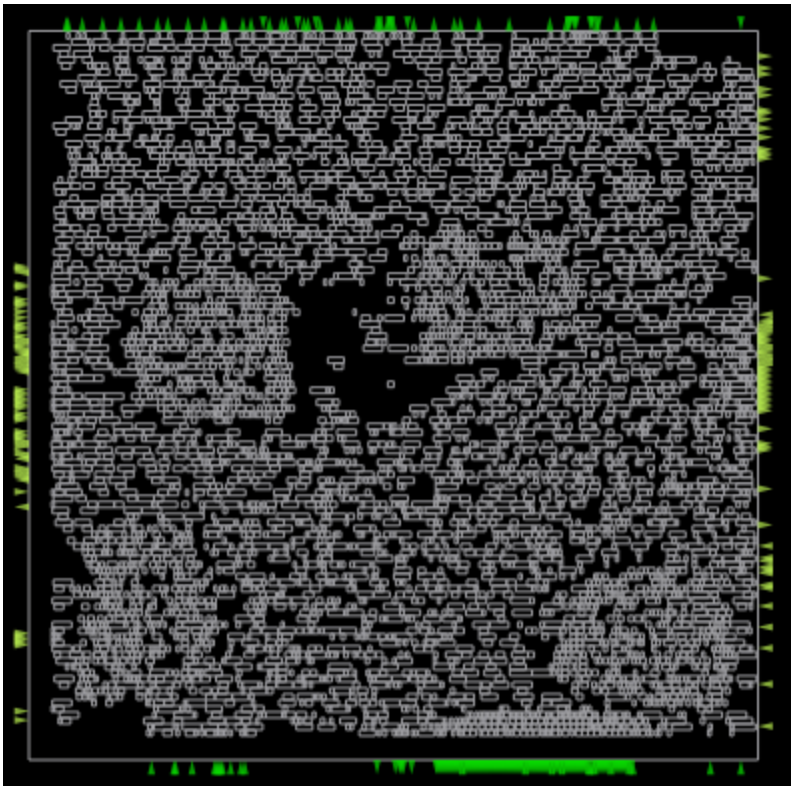


Fig. 16. Chip layout of the decomposer module with just standard cells

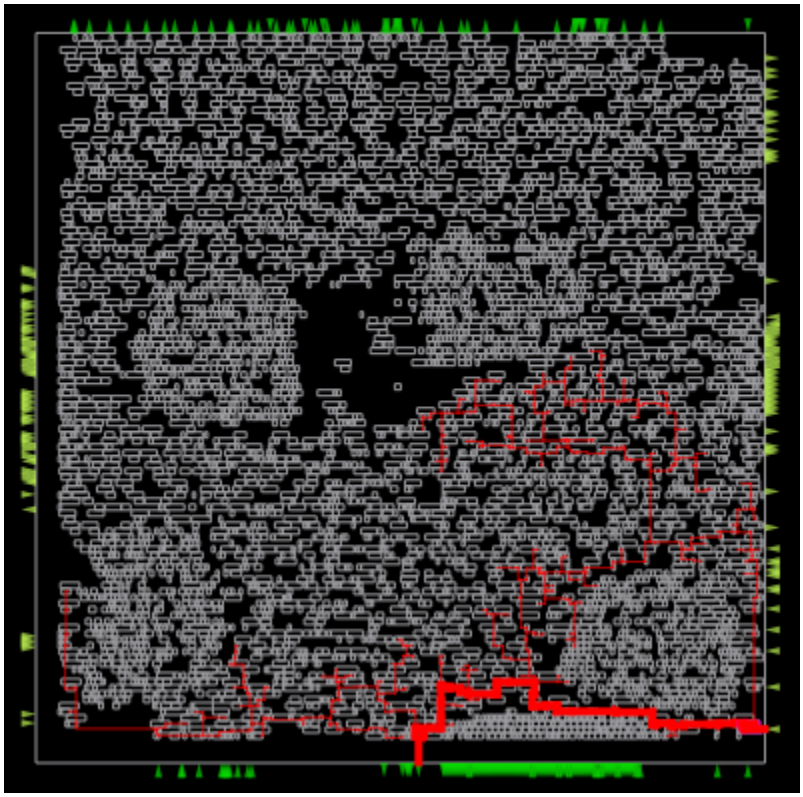


Fig 17. The critical path of the decomposer module

From viewing the chip through the OpenROAD GUI, we can find more information about the critical path and the timing of other paths. Our main observation was that the update to the register sizes slightly increased the slack in many other paths, leading to a large change in the overall slack. It still seems that the critical path in both the original and optimized design is the path from *ready_i* to *ready_o*. However, it is not considered slow: it still has over 5000 ps of slack.

With further inspection, this path traces to the `decomp_map1` module and gets values in and out of the map. This module is essentially a very large priority encoder which would translate to a very large LUT. The largest bit is taken and assigned to the value of the high-bit output. These values are stored in the two registers, one 17-bit register and one 45-bit register for security level 0 and security level 1 respectively. By doing the calculation instead of the priority encoder, we can save even more area and increase our overall slack further.

E. Testing

When testing the decomposer, out team tested all possible settings. This included when the ready signals were on/off and the different security levels. Changing the security level on the decomposer allows for larger outputs and more precision between inputs. In terms of main inputs, we included the same data with multiple security levels to test the effects of security level alongside values that will test the outputs of the maximum and minimum inputs. Furthermore, we displayed the results in the console log with transaction numbers to make reading easier, since the correct output would print four cycles after the input first appears. It is also important to note that most of the tests were performed on a single `coeff_decomposer` module, as the higher-level decomposer unit simply combines four `coeff_decomposer` modules.

```
    always @(di) begin
        $display("transaction input [%d]", transaction_num1);
        $display("di = %d", di);
        transaction_num1 = transaction_num1 + 1;
    end

    always @(doa or dob) begin
        if (transaction_num1 >= 4) begin
            $display("transaction output [%d]", transaction_num2);
            $display("doa = %d, dob = %d", doa, dob);
            transaction_num2 = transaction_num2 + 1;
        end
    end
```

Fig. 18. Monitoring code for the `coeff_decomposer` module

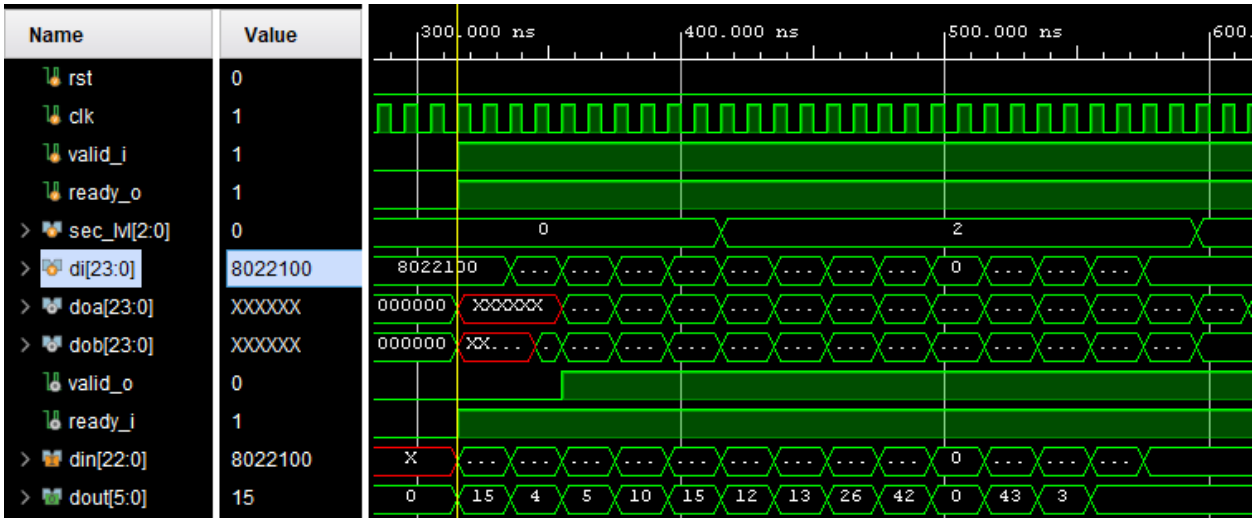


Fig. 19. Waveform of the `coeff_decomposer` module

```
transaction input [id:          5]
di =  2312250
```

```
transaction output [id:          5]
doa =   217146, dob =        4
```

Fig. 20. and 21. Transactions in the `coeff_decomposer`

In addition to the deterministic tests, we tested random values at set intervals within the bounds of the decomposer. We made sure to test both security levels to get code coverage.

```
        sec_lvl = SEC_LVL_0;
        repeat(10) #20 di = $urandom_range(0, 8118529);
        sec_lvl = SEC_LVL_2;
        repeat(10) #20 di = $urandom_range(0, 8285185);
```

Fig. 22. Random range input variables

VI. Difficulties and Recommendations

Our team faced many difficulties over the course of this project, and we have recommendations for other students who want to attempt a continuation of this project or a related project that deals with cryptography and ASIC design.

A. Scope of Knowledge Required

The most challenging part of this Major Qualifying Project was the breadth of advanced background knowledge required to successfully achieve our original design goals. Our original goals required a much deeper knowledge of digital design, ASIC design, advanced mathematics, and cryptography. Our team had individual members who were proficient in each of these individual fields, but not a single member had sufficient experience in every field. We were able to mitigate some of our individual inexperience by working closely as a team and dividing tasks. However, there were limits to our ability to do this. With how interconnected the project was, it became apparent that our group simply did not have enough combined experience. Two-thirds of the way through the project, a team member split from the project to work on a required adjacent project for a separate degree. This impacted our functionality substantially, as our team size shrunk to three people.

Two core issues that prevented us from making more progress were our lack of understanding of post-quantum cryptography, something that requires several graduate-level courses to be able to understand, and our lack of experience in ASIC design, another graduate-course-exclusive topic.

Another challenge emerged from our incomplete understanding of the topic. When we attempted to reevaluate our scope, we were unaware of the course knowledge we lacked. While our scope kept on tightening, we repeatedly reached the conclusion that we were still being too ambitious in our goals. By the time we quantified our limits and settled on a more manageable scope, we had spent months generating data and studying material that was no longer directly usable in our current endeavor.

B. Tools

Another obstacle we faced was the usage of tools. As mentioned previously, knowledge of ASIC design was needed for this course, and one of our members took the relevant class concurrently with the MQP. However, this made it difficult to teach the other members who had only taken the prerequisite FPGA courses. There were many tools that needed to be installed, and the installation process was lengthy, time-consuming, and confusing. This led to copious time spent trying to set up virtual environments and several weeks attempting to learn the workings of each of the tools.

The field of ASIC development is less popular when compared to FPGA design within computer science and software engineering. Due to the lack of online information, each issue we encountered with each tool cost us substantial amounts of precious project time. This made us rely on software manuals and repeatedly attempt working with the systems until success. This was especially prevalent for the DC shell and OpenROAD GUI, where both had many errors that were only solved by trial and error. The OpenROAD GUI for the decomposer was only available near the end of the project, so we were not able to do as much layout research as we had desired.

C. Recommendation

Our recommendations fall into two categories: who should attempt this project and what background information is needed to be successful and technical recommendations for future implementation.

Firstly, we believe that it is extremely important to have a sufficient in-depth background in graduate-level electrical and computer engineering classes. Whether future groups attempt to continue optimizing the GMU hardware implementation or attempt to implement a custom design, we recommend both knowledge of post-quantum cryptography alongside ASIC design and optimization.

If an undergraduate student team begins the project without strong fundamentals, they may reach a similar conclusion to us. We tried to learn as the project progressed, but met the issue of confused redirection. Similarly to us, other undergraduate teams may not know when to refocus the main objective of the project.

A math background is also recommended. Having a team member with a strong mathematics knowledge base working with us for a portion of the project greatly benefitted our understanding of the cryptography scheme.

As for technical advice, we recommend attempting to optimize the `decomp_map1` module. In our discussion with the GMU team, they explained that their FPGA implementation has inefficiencies that could be solved with ASIC hardware implementation (Section IV.C). We recommend taking the Decomposer Map module and performing the calculations instead of using a priority encoder.

If a group wants to implement or optimize another block, we recommend something with more documentation such as the NTT block or the butterfly block as those are well-researched and well-documented blocks that groups have studied in the past.

VII. Acknowledgments

Our team would like to express our thanks and gratitude to Professor Patrick Schaumont for all his support and guidance throughout this project. We would also like to thank Professor Herman Servatius for all his help in the mathematical aspects of this project. This project would not be possible without the help of both of our advising professors.

Finally, we would like to thank Brandon Voci for his contributions to this project's cryptography understanding and mathematical analysis.

VIII. References

[1] R. Abid et al., “An optimised homomorphic CRT-RSA algorithm for secure and efficient communication,” Personal and Ubiquitous Computing. Springer Science and Business Media LLC, Sep. 01, 2021. doi: 10.1007/s00779-021-01607-3.

[2] “What is cryptography or a cryptographic algorithm?,” *What is cryptography or a Cryptographic Algorithm? | DigiCert FAQ*. [Online]. Available: https://www.digicert.com/support/resources/faq/cryptography/what-is-cryptography-or-a-cryptographic-algorithm.

[3] VMWare, “What is Elliptic Curve Cryptography? Definition & FAQs,” Avi Networks. https://avinetworks.com/glossary/elliptic-curve-cryptography/

[4] S. S. Gill et al., “Quantum computing: A taxonomy, systematic review and future directions,” Software: Practice and Experience, vol. 52, no. 1. Wiley, pp. 66–114, Oct. 07, 2021. doi: 10.1002/spe.3039.

[5] B. Schumacher, “Quantum coding,” *Physical Review A*, vol. 51, no. 4, pp. 2738–2747, 1995.

[6] “Post-quantum cryptography,” *Post-Quantum Cryptography | Homeland Security*. [Online]. Available: https://www.dhs.gov/quantum.

[7] I. T. L. Computer Security Division, “Post-Quantum Cryptography Standardization - Post-Quantum Cryptography | CSRC | CSRC,” CSRC | NIST, Jan. 03, 2017. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[8] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," Proceedings 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 1994, pp. 124-134, doi: 10.1109/SFCS.1994.365700.

[9] P. E. Black, “NP-hard,” *Algorithms and Theory of Computation Handbook*, 05-Jan-2021. [Online]. Available: https://xlinux.nist.gov/dads/HTML/nphard.html.

[10] M. R. Albrecht et al., “Classic McEliece: conservative code-based cryptography”, 2022. [Online]. Available: https://classic.mceliece.org/nist/mceliece-submission-20221023.pdf

[11] A. Avanzi et al., “CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.02)”, 2021. [Online]. Available: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf

[12] C. Chen et al., “NTRU Algorithm Specifications And Supporting Documentation”, 2019. [Online]. Available: https://ntru.org/f/ntru-20190330.pdf

[13] J-P. D’Anvers, A. K. S. S. Roy, F. Vercauteren, “Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM”, 2018. [Online]. Available: https://eprint.iacr.org/2018/230.pdf

[14] S. Bai *et al.*, “CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (Version 3.1)”, 2021. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf

[15] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, S. Whyte, and Z. Zhang, “Falcon Fast-Fourier Lattice-based Compact Signatures over NTRU,” *Falcon*, 17-Nov-2017. [Online]. Available: https://falcon-sign.info/

[16] J. Ding, M.-S. Chen, A. Petzoldt, D. Schmidt, and B.-Y. Yang, “Rainbow The 2nd NIST Standardization Conference for Post-Quantum Cryptosystems,” *PQCRainbow*. [Online]. Available:

https://csrc.nist.gov/CSRC/media/Presentations/rainbow-round-2-presentation/images-media/rainbow-ding.pdf.
[17] M.-shing Chen, J. Ding, M. Kannwischer, J. Patarin, A. Petzoldt, D. Schmidt, and B.-Y. Yang, "Pqcrainbow One of the Three NIST Post-quantum Signature Finalists," *PQCRainbow*. [Online]. Available: https://www.pqcrainbow.org/.
[18] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-Performance Hardware Implementation of CRYSTALS-Dilithium," *2021 International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2021, doi: 10.1109/icfpt52863.2021.9609917. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9609917&isnumber=9609698
[19] C. Maxfield, "FPGA vs. ASIC Designs," in *FPGAs: Instant access*, Amsterdam: Elsevier/Newnes, 2008, pp. 1–2.
[20] Utimaco, "What is lattice-based cryptography?," What is Lattice-based Cryptography? [Online].https://utimaco.com/products/technologies/post-quantum-cryptography/what-lattice-based-cryptography.
[21] A. Hartshorn, S. Weber, N. Qiao, & H. Leon Liu, "Number Theoretic Transform (NTT) FPGA Accelerator", Worcester Polytechnic Institute, 2020. [Online]. Available: https://digital.wpi.edu/pdfviewer/p2676z164
[22] Bertoni, Daemen, J., Peeters, M., & Van Assche, G, "Keccak," *Advances in Cryptology – EUROCRYPT 2013*, 313–314. https://doi.org/10.1007/978-3-642-38348-9_19
[23] P. Luo, Y. Fei, X. Fang, A. A. Ding, M. Leeser and D. R. Kaeli, "Power analysis attack on hardware implementation of MAC-Keccak on FPGAs," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Cancun, Mexico, 2014, pp. 1-7, doi: 10.1109/ReConFig.2014.7032549.
[24] A. Anand, "Breaking down : Sha-3 algorithm," *Medium*, 13-Jan-2020. [Online]. Available: https://infosecwriteups.com/breaking-down-sha-3-algorithm-70fe25e125b6.
[25] Team VLSI, "Standard Cell Library for ASIC design", 2020. [Online]. Available: https://teamvlsi.com/2020/08/standard-cell-library-in-asic-design.html
[26] F. Panca Juniawan, H. Arie Pradana, Laurentinus, and D. Yuny Sylfania, "Performance Comparison of Linear Congruent Method and Fisher-Yates Shuffle for Data Randomization," Journal of Physics: Conference Series, vol. 1196. IOP Publishing, p. 012035, Mar. 2019. doi: 10.1088/1742-6596/1196/1/012035.
[27] C. Wolf, "Yosys Open SYnthesis Suite". [Online]. Available: https://yosyshq.net/yosys/
[28] J. S. Arora, "Multi-objective Optimum Design Concepts and Methods," Introduction to Optimum Design. Elsevier, pp. 771–794, 2017. doi: 10.1016/b978-0-12-800806-5.00018-4.

Appendix A: NIST Cryptographic Algorithms

Classic McEliece	Documentation: Mid	Ease of Understanding: Easy	Ease of Digital Design: Easy

Notes:

- Uses a public-key cryptosystem
  - Public-key uses a random binary Goppa code
  - A Goppa code is a polynomial g(x) of degree t over a finite field GF(2^{m}) with no repeated roots, and a sequence {L_{1},...,L_{n}} of n distinct elements from GF(2^{m}) that are not roots of g.
  - Ciphertext with errors are used as well
- Private-key decodes the ciphertext and identifies and removes errors
- First introduced in 1978 so it is quite old
  - Still holds up (prequantum) now despite that
- Algorithm also known as a KEM (Key Encapsulation Method)
- Designed for OW-CPA security (One Wayness Against Chosen Plaintext Attack)
- Decoding equation:

$$s(x) \equiv \sum_{i=0}^{n-1} \frac{c_i}{x - L_i} \mod g(x)$$

- Uses square root

$$v(x) \equiv \sqrt{s(x)^{-1} - x} \mod g(x)$$

-
- Roots are used for correcting errors

Documentation/Code notes:

- Has a lot of references on website
- Has 22 page paper that includes the math and a block diagram
- Includes the code
- Encoding Verilog code looks simple @ first glance
- Looks like the keys are generated in Python
- Decryption is a bit harder (of course) and uses a lot of row multiplication and vectors
- Code has comments!

https://classic.mceliece.org/index.html
https://eprint.iacr.org/2017/1180
https://classic.mceliece.org/nist/mceliece-20201010.pdf

CRYSTALS-KYBER	Documentation: Mid	Ease of Understanding: Easy	Ease of Digital Design: Hard

Notes:

- Uses two cryptographic primitives:
  - Kyber, which uses IND-CCA2 (Indistinguishability under chosen ciphertext attack/adaptive chosen ciphertext attack) which is also a KEM
    - Uses LWE encryption (Learning With Errors)
  - Dilithium, a strongly EUF-CMA-secure digital signature algorithm (Existential Unforgeability under Chosen Message Attack)
  - Both use module lattices and can withstand quantum attacks
  - The only operations required for Kyber and Dilithium for all security levels are variants of Keccak, additions/multiplications in Zq for a fixed q, and the NTT (number theoretic transform) for the ring Zq[X]/(X^256+1)
    - Easy to understand and looks not terrible to implement
  - We would have to learn two algorithms but since they both use module lattices it will be a bit easier
  - Code is in C

Notes on C code:

- Split into many functions for easier understanding
  - These split functions have mostly simple operations
  - Contains 17 .c files
  - Has IEEE hardware implementation (2021)

NTRU (1996)	Documentation: High	Ease of Understanding: Mid-hard	Ease of Digital Design: Hard

Notes:

- Open source public key
- NTRU originally described as “partially correct probabilistic public key encryption scheme (partially correct PPKE)” but can be made deterministic and perfectly correct
- KEM by Hoffstein, Pipher, Silverman
- Lattice based alternative to RSA and elliptic curve cryptography
  - (based on shortest vector problem in a lattice)
  - Quantum computers are not known to be able to break shortest vector problem
- Two algos for different uses:
  - NTRUEncrypt → encryption
  - NTRUSign → digital signature
- Math:
  - Based on the truncated polynomial ring: $R = \mathbb{Z}[X]/(X^N - 1)$
    - Convolution multiplication and all polynomials have integer coefficients w/ N-1 being the highest degree

- Parameters: N, q, p
- Polynomial arithmetic modulo → ciphertext is private key and public key, receiver uses their own private key to decrypt. But all of the keys are polynomials

- Security:
  - Only feasible attack is lattice-based attack → system of equations in an attempt to solve for any chosen polynomial
  - Needs a matrix of size NxN ; if N is less than 100, NTRU can be broken, but if N is of larger magnitude, it takes years for computers to solve. So, if N is way bigger, (impossible)
- Quantum variants of sieve algorithms studied
  - Improvements rely on unit-cost superposition queries to classical memory (QRAM), the best claimed operation count is $2^{(0.265...+o(1))\cdot b}$
    - (NTRU people say that quantum advancements don't really benefit the way that this algorithm is attacked) → but it might in the future
- Advantages:
  - Correct, well studied, flexible, simple, fast, compact, patent free.
- Limitations:
  - Not best in speed, compactness, or security, but is really good in all categories
  - Similar to all lattice based cryptosystems, the optimal parameters are limited by non-asymptotic behavior of new algos for SVP
  - There are extra structures in the NTRU system that are excess, and people may construe this to be a limitation (stuff can be eliminated but it will make it less correct and compact)

https://ntru.org/f/ntru-20190330.pdf
^ this one is the main documentation file from the NTRU group
https://ntru.org/talks/20190823_nist_round2.pdf
https://ideaexchange.uakron.edu/cgi/viewcontent.cgi?article=1880&context=honors_research_projects

SABER:	Documentation: Mid	Ease of Understanding: Mid	Ease of Digital Design: Mid-Hard

Notes:
- Mod -LWR based KEM
  - Lattice based , designed to offer better resistance to quantum computers
  - Relies on the hardness of the:
    - Module learning with rounding problem: Mod-LWR
  - Saber.PKE to Saber.KEM using a version of Fujisaki-Okamoto transform (no idea what this is)
- Integer moduli are powers of 2 to prevent modular reduction and rejection sampling
- LWR reduces the randomness required (½) the amount of LWE based schemes, reduces

bandwidth
- Module structure provides flexibility by reusing one core component for multiple security levels
  - Ring: $\mathbb{Z}_q[X]/(X^n+1)$
  - Parameters: n, L → degree 256 of the polynomial ring
    - Rank L of the module (determines the dimension of the lattice problem)
    - Dimension of lattice increase -> better security, less correctness
  - q,p,T → moduli powers of 2 → 2^{xq}
    - Higher the parameter, lower security, higher correctness
- Security in the Quantum Random Oracle Model
  - See theorem 6.5 page 20 of saberspecround3.pdf
- Advantages:
  - No modular reduction (since all moduli are power of 2)
  - Modular structure is flexible: to change security, change module to higher rank one for better security
  - Less pseudorandomness required bc of the use of Mod-LWR (½)
  - Scaling and rounding simplified bc power of 2 mod p,q,T
  - Low bandwidth bc of Mod-LWR
  - Generic polynomial multiplication → implementation ease
  - No full multiplications: random element in R_q x small element from B_u → circular shifts and additions
  - Good for anonymous communication → constant time over different pub key
  - Efficient masking
- Limitations:
  - Use of 2-power moduli makes NTT-like polynomial multiplication not natively supported.
  - Limited to encryption scheme and KEM, no signature scheme included

https://www.esat.kuleuven.be/cosic/pqcrypto/saber/
^main website
https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf
^documentation by SABER designers
https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/ribeiro-saber-pq-key-pqc2021.pdf

CRYSTALS-DILITHIUM:	Documentation: High	Ease of Understanding: Mid	Ease of Digital Design: Mid

Background:

Lattices:

- Lattice over R^n (*Geometric Interpretation*): Set of linearly independent points in R^n that is also a free abelian (commutative) group and spans R^n. In other words, a lattice can be constructed from a basis of R^n, by taking linear combinations with integral coefficients.

  That is: $\Lambda := \{ \sum_{i=1}^{n} a_i v_i \mid a_i \in Z \}$ where $\{v_i\}$ is a basis for R^n
- In many cryptographic schemes, a "hard" problem involves finding "short" vectors in lattices: for a Lattice over R^n "short" can be defined in terms of the usual 2-norm on R^n: $\sqrt{{v_1}^2 + {v_2}^2 + \ldots}$ or in terms of <u>inf-norm</u> $max\{|v_i|\}$ (this is the norm used in the Dilithium scheme).
- Recall that R^n is a vector space over a field. Analogously, we can also construct lattices on modules over rings. The Dilithium scheme utilizes such modular lattices in the signature process.

Polynomial Rings & Constructing Fields:

- Let Z_q denote the integers modulo q. Then $Z_q[X]$ is the ring of polynomials with entries in Z_q. By choosing a maximal ideal, $< X^n + 1 >$ and obtaining the quotient ring generated. By $Z_q[X] / < X^n + 1 >$ we receive a field, F, wherein the usual arithmetic operations (+, -, x, /) are well-defined.
- Dilithium scheme constructs matrices with entries in $F := Z_q[X] / < X^n + 1 >$ and the <u>key space</u> is simply $GL(n, F) = GL(n, Z_q[X] / < X^n + 1 >)$.

Mechanics:

Key Generation:

- Begin with a number, q, and integers (k, l). Generate a $k \times l$ matrix with entries in the previously described field, F. q is defined by some formula, and the degree of the polynomial in the maximal ideal is $n := 256$. We then sample random "secret" key vectors, (s_1, s_2), and the key is generated according to the rule $t = As_1 + s_2$. So t is an affine combination.

Signing Procedure:

- Signing algorithm generates a masking vector of polynomials with coefficients less than a given γ an challenge is created as the hash of the message and w where w is computed after some operations from the masking vector and is then further

algebraically manipulated.

Verification:
- Verifier computes w' in the higher-order bits and accepts if all of the coefficients of the vector-part are less than a certain $\gamma - \beta$.

Implementations:
- Has a documented implementation on Intel Core-i7 6600U (Skylake) CPU.
- Furthermore, there is a C reference implementation, located on a GitHub repository

More on the Mathematical Background for Dilithium:
Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures | SpringerLink https://link.springer.com/chapter/10.1007/978-3-642-10366-7_35
Official Specifications Document & Supporting Documentation :
CRYSTALS-Dilithium (pq-crystals.org)
https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf
GitHub Repository:
GitHub - pq-crystals/dilithium

FALCON:	Documentation: Mid-high	Ease of Understanding: Mid	Ease of Digital Design: Hard

Notes:
- Fast-Fourier Lattice-based algorithm.
- Similar to Crystals-Dilithium
- True internal Gausian Sampling for negligible leakage
- NTRU latices allow for short signatures and public keys
    - O(n log n)
- uses less than 30 kilobyes of ram, over 100x better than old NTRUsign
- Falcon, submitted to NIST Post-Quantum Cryptography Project in 2017, is based on lattice-based signature schemes and short integer solution problem (SIS) over NTRU lattices.
- It offers high security, compactness, speed, scalability, and RAM economy.
- efficient signature generation and verification

Rainbow:	Documentation: Mid-high	Ease of Understanding: Mid	Ease of Digital Design: Hard

Notes:

Background:
- Based off of the Unbalanced Oil and Vinegar scheme (UOV)
- Uses random multivariate quadratic systems

- Rainbow, designed in 2004, is based on multivariate quadratic systems and algebraic geometry. It was selected as one of the three NIST Post-quantum signature finalists in 2020 and offers very small signatures and efficient signature generation and verification through the use of simple operations over small finite fields.
- Rainbow offering very small signatures.
- efficient signature generation and verification

Mechanics:

- Constructed from an inveratble quadratic map

$$\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^m$$

- Inverable linear maps

$$\mathcal{S} : \mathbb{F}^m \to \mathbb{F}^m \text{ and } \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$$

- Public key: $P = S$ o F o T
- Private key: S, F, T can be used to invert the public key

Signature Generation:

- Given a document $d \in \{0,1\}^\star$
  - The hash H finds $\mathbf{w} = \mathcal{H}(d) \in \mathbb{F}^m$
- Compute recursively $\mathbf{x} = \mathcal{S}^{-1}(\mathbf{w}) \in \mathbb{F}^m$, $\mathbf{y} = \mathcal{F}^{-1}(\mathbf{x}) \in \mathbb{F}^n$ and $\mathbf{z} = \mathcal{T}^{-1}(\mathbf{y})$
- Signature is $\mathbf{z} \in \mathbb{F}^n$

Signature Verification

- Given $\mathbf{z} \in \mathbb{F}^n$ $\mathbf{w} \in \mathbb{F}^m$
- Calculate $\mathbf{w}' = \mathcal{P}(\mathbf{z}) \in \mathbb{F}^m$
- If $\mathbf{w}' = \mathbf{w}$ the signature is true

Appendix B: Function List

Function:	Input(s):	Output(s):	Description:
address_encoder_decoder.cpp			
unsigned resolve_address	- enumerated type mapping (NATURAL, AFTER_NTT, AFTER_INVNTT) - unsigned int address	- unsigned int ram_i	Based on the values of MAPPING(NATURAL, AFTER_NTT, AFTER_INVNTT), it has case statements to compute the next address
void resolve_twiddle	- unsigned int tw_i[4] - unsigned int *last - unsigned tw_base_i[4] - const int k - const int s - enum OPERATION mode	- void, but changed the values of: unsigned intl1, l2, l3, l4 unsigned l1_base, l2_base, l3_base, l4_base unsigned int tw_i[3:0] and tw_i_base[3:0].	Enumerated type OPERATION (FORWARD_NTT_ MODE, INVERSE_NTT_ MODE, MUL_MODE) determine what operation is performed through if else statements. Based on the value of k and s, tw_base_i and tw_base_i are assigned with the values of l1 - l4 and l1base - l4_base
address_encoder_decoder.h			
unsigned resolve_address	- enumerated type mapping (NATURAL, AFTER_NTT, AFTER_INVNTT) - unsigned int address	- unsigned int ram_i	Header file
void resolve_twiddle	- unsigned int tw_i[4] - unsigned int *last - unsigned tw_base_i[4] - const int k - const int s	- void, but changed the values of: unsigned intl1, l2, l3, l4 unsigned l1_base,	Header file

	- enum OPERATION mode	l2_base, l3_base, l4_base unsigned int tw_i[3:0] and tw_i_base[3:0].	
butterfly_unit.h			
void butterfly	- template <typename T2, typename T> - enumerated type OPERATION mode - T *bj - T *bjlen - const T zeta - const T aj - const T ajlen	- void, but changes the value of aj1 - aj4, ajlen1 - ajlen4, and bj and bjlen.	Depending on the operation mode, it performs operations on the values of aj1 - aj4, ajlen1 - ajlen4, and bj and bjlen. These variables are of type T but I have yet to find the definition for T.
void buttefly_circuit	- template <typename T2, typename T> - T data_out[4] - const T data_in[4] - const T w[4] - enum OPERATION mode	- void, but changes the value of data_out[1] - data_out[4]. Also instantiates butterfly variable/class depending on mode.	A pipelined process with debug statements that set T (variable/class I'm not sure). Instantiates these butterfly classes (?). data_out depends on the mode.
config.h			
struct BRAM	- template <typename T> - T coeffs[BRAM_DEPT][4]	- struct definition	BRAM_DEPT = 64 4 arrays of 64 bits of class/type T
enum OPERATION	- FORWARD_NTT_MODE - INVERSE_NTT_MODE - MUL_MODE	- struct definition	Modes that help other functions know what operation the system is currently performing
enum MAPPING	- NATURAL - AFTER_NTT - AFTER_INVNTT	- struct definition	Modes that help other functions know where the system is currently operating
typedef			BRAM<int32_t> bram

BRAM<data_t> bram;			
fifo.h			
(int32_t) data_t FIFO	- (int32_t) data_t fifo[DEPT] - (int32_t) const data_t new_value	- (int32_t) data_t out	Sets data_out to fifo[DEPT - 1] and then does a for loop to set fifo[i] to fifo[i-1]
void PIPO	- template <int DEPT, typename T> - T w_out[4] - T fifo[DEPT][DEPT] - const T w_in[4]	- void but changes value of w_out and fifo	Sets w_out[i] = fifo[DEPT - 1][i] and fifo[0][i] = w_in[i]. Also performs two nested for loops with decrementing i and incrementing j: fifo[i][j] = fifo[i - 1][j];
T FIFO_PISO	- template <int DEPT, typename T> - T fifo[DEPT] - const bool piso_en - const T line[4] - const T new_value	- T out	Sets out to fifo[DEPT - 1] and also does work on the rest of the fifo. First performs a for loop for the size of DEPT - 1 to 4: fifo[i] = fifo[i - 1] Then manually changes the values of fifo and line based on array value (i.e. fifo[3] = line[0]) if piso_en is enabled. Otherwise, the fifo is pushed like a fifo and a new value is inserted in fifo[0]
void read_fifo	- template <typename T> - T data_out[4] - const unsigned count - const T fifo_a[DEPT_A] - const T fifo_b[DEPT_B] - const T fifo_c[DEPT_C], - const T	- void but changes the value of data_out as well as ta, tb, tc, td	Case statement depends on the value of count and bitwise AND of 3. ta - td will be set depending on the value being 0 - 3. Finally, data_out[0 - 3] will be assigned by the values of ta - td.

	fifo_d[DEPT_D]		
void read_write_fifo	- template <typename T> - enum OPERATION mode - T data_out[4], - const T data_in[4] - const T new_value[4] - T fifo_a[DEPT_A] - T fifo_b[DEPT_B] - T fifo_c[DEPT_C] - T fifo_d[DEPT_D] - const unsigned count	- void, but changes the value of a_piso_en - d_piso_en, fd - fa, and data_out[0 - 3]	Case statement depends on the value of count and bitwise AND of 3. Having a value of 0 will enable write to fifo D, 1 equates to enabling write to fifo C, etc. Performs compound ANDing on the fifo write enable variables depending on mode. fd - fa is set depending on the values of FIFO_PISO. data_out[0 - 3] = fd - fa if the mode is forward and read_fifo is call if the mode is something else.
ram_util.cpp			
read_ram	data_t data_out[4], const bram *ram, const unsigned ram_i	-void	Changes values of data_out[0-3] to the values of ram->coeffs[ram_i][0-3]
write_ram	bram *ram, const unsigned ram_i, const data_t data_in[4]	-void	Changes the values of ram->coeffs[ram_i][0-3] to data_out[0-3]
read_twiddle	data_t data_out[4], enum OPERATION mode, const unsigned tw_i[4]	-void	Sets data_out[0-3] to zetas_barrett{i1-4] or to -zetas_barrett{i1-4] based on wether OPERATION mode is imputed as Forward_NTT_MODE or INVERSE_NTT_MODE

ram_util.h			
read_ram	data_t data_out[4], const bram *ram, const unsigned ram_i	void	instantiation
write_ram	bram *ram, const unsigned ram_i, const data_t data_in[4]	Void	instantiation
read_twiddle	data_t data_out[4], enum OPERATION mode, const unsigned tw_i[4])	void	instantiation
util.cpp			
print_reshaped_array	bram *ram, int bound, const char *string	void	Prints the array modded array using the read_ram function and a nested for loop
print_index_reshaped_array	bram *ram, int index	void	Prints the index of the modded array
reshape	bram *ram, const data_t in[DILITHIUM_N]	void	
compare_array	data_t *a, data_t *b, int bound	Int 0 or 1	Returns 1 if the arrays are not the same returns 0 otherwise
compare_bram_array	bram *ram, data_t array[DILITHIUM_N], const char *string, enum MAPPING mapping, int print_out	Int 0 or 1 if error	
util.h			
print_array	T *a, int bound, const char *string	void	Prints the input array a within a given bound
print_reshaped_array	bram *ram, int bound, const char *string	void	instantiation

print_index_reshaped_array	bram *ram, int index	void	instantiation
reshape	bram *ram, const data_t in[DILITHIUM_N]	void	instantiation
compare_array	data_t *a, data_t *b, int bound	int	instantiation
compare_bram_array	bram *ram, data_t array[DILITHIUM_N], const char *string, enum MAPPING mapping, int print_out	int	instantiation
ntt2x2.h *(h file) - header instantiation*			
update_indexes	unsigned tw_i[4], const unsigned tw_base_i[4], const unsigned s, Enum OPERATION mode	void	Instantiation for same func in ntt2x2.cpp
ntt2x2_fwdntt	bram *ram, enum OPERATION mode, enum MAPPING mapping	void	instantiation
ntt2x2_mul	bram *ram, const bram *mul_ram, enum MAPPING mapping	void	instantiation
ntt2x2_invntt	bram *ram, enum OPERATION mode, enum MAPPING mapping	void	instantiation
ntt2x2.cpp			
MAX	typename T: a, b	typename T	Returns the larger value a or b -Uses inline conditional (a<b)? b:a -Option to use the comp() function to compare ‘a’ and ‘b’

update_indexes	unsigned tw_i[4], const unsigned tw_base_i[4], const unsigned s, Enum OPERATION mode	void	Changes index/memory addresses
ntt2x2_mul.cpp *(multiply)*			
ntt2x2_mul	bram *ram, const bram *mul_ram, enum MAPPING mapping	void	Point-wise multiplication Read address, calculate, write back
ntt2x2_fwdntt.cpp *(forward ntt)*			
ntt2x2_fwdntt	bram *ram, enum OPERATION mode, enum MAPPING mapping	void	Forward NTT for 256, uses FIFO, temp “twiddle”, implements butterfly circuit()
ntt2x2_invntt.cpp *(inverse ntt)*			
ntt2x2_invntt	bram *ram, enum OPERATION mode, enum MAPPING mapping	void	Inverse NTT, implements fwdntt pattern, rolling FIFO index, iterates on conditional to roll FIFO and extract data
ntt2x2_test.cpp *(testbench file - compares all ret values with reference code tbench results)*			
ntt2x2_NTT	data_t r_gold[DILITHIUM_N]	int ret	Load data into BRAM, complete fwdntt, prints array from compare_bram_array
ntt2x2_MUL	data_t r_mul[DILITHIUM_N], data_t test_ram[DILITHIUM_N]	int ret	ntt2x2_mul test
polymul	data_t a[DILITHIUM_N], data_t b[DILITHIUM_N]	int ret	Test hardware multiplication for mul function; uses the |= “or” operand for the ret value

reference_code Function(s):

Function:	Input(s):	Output(s):	Description:
ref_ntt.cpp			
ntt( )	a: Array of size DILITHIUM_N in data_t	Returns void but modifies input array	Performs NTT transform on input array. All operations are modulo DILITHIUM_Q.
pointwise_barrett( )	c: Array of size DILITHIUM_N in data_t a, b: Constant arrays of same size in data_t	Returns void but modifies input array	Iterative scheme for pointwise multiplication of a, b arrays, modulo value of DILITHIUM_Q. Resultant array stored in c.
invntt( )	a: Array of size DILITHIUM_N in data_t	Returns void	Performs inverse NTT transform on input array.
ref_ntt2x2.cpp			
ntt2x2_ref( )	a: Array of size DILITHIUM_N in data_t	Returns void but modifies input array	Performs 2x2 NTT; i.e matrix formulation of NTT
invntt2x2_ref( )	a: Array of size DILITHIUM_N in data_T	Returns void but modifies input array	2x2 inverse NTT using matrix formulation of NTT.
ref_test_ntt2x2.cpp			
compare_array()	*a: Pointer to test array in data_t *a_gold: Pointer to test array in data_t	Returns an integer: 0: Indicates arrays are incongruent modulo DILITHIUM_Q 1: Indicates arrays are NOT congruent modulo DILITHIUM_Q	Verifies whether two arrays (assumed of same length) are congruent modulo DILITHIUM_Q. Iterates through the length of the arrays and, at each index, check if the difference modulo DILITHIUM_Q is 0.
main( )	Void input	Returns an integer: 1: Successful 0: Unsuccessful	Instantiate two separate arrays: a, a_gold. Initialize with

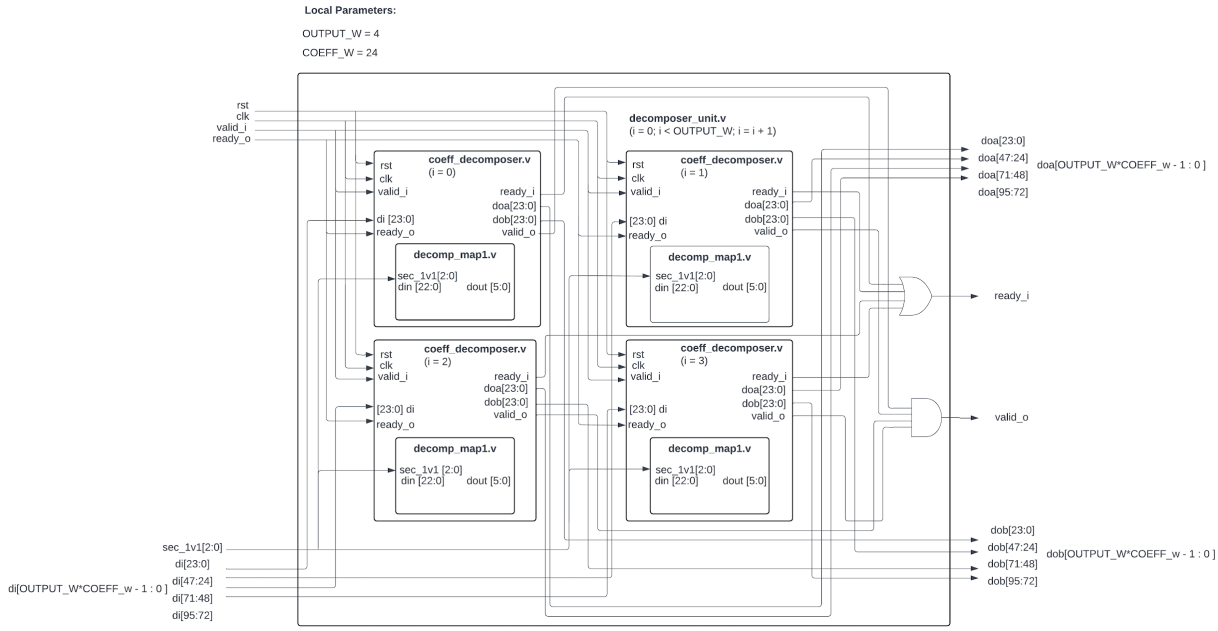
			identical data modulo DILITHIUM_Q generated from rand(). Perform forward NTT using ntt( ) on one and ntt2x2_ref( ) on another. Compare resulting arrays. Repeat for inverse NTT.
consts.cpp			
Contains an array called zetas_barrett that is referenced in file: [ram_until.cpp]			

Appendix C: RTL Modules and Functions in Files

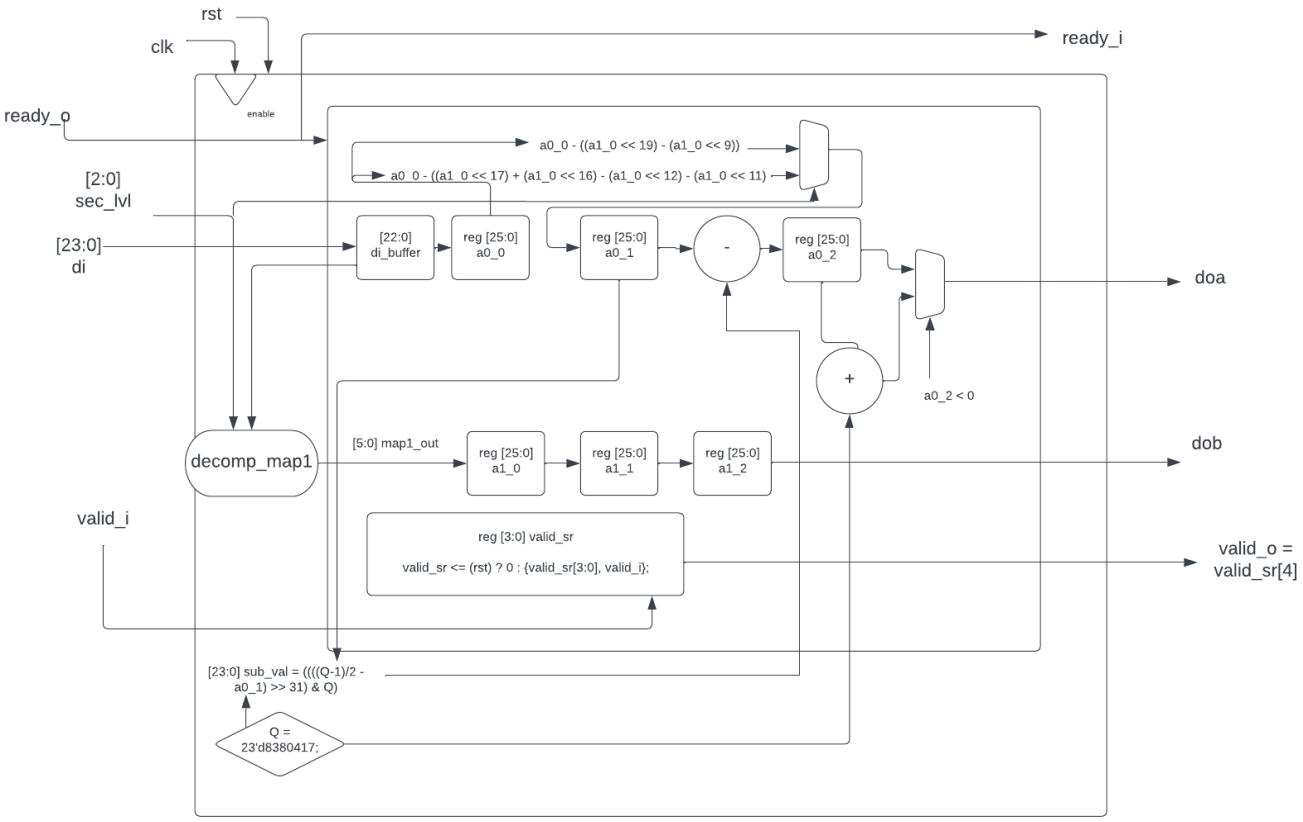
Block:	Files(s):
Keccak	keccak_top.vhd - keccak_control.vhd - sr_reg.vhd - keccak_datapath.vhd - keccak_bytepad.vhd - sipo.vhd - regn.vhd - keccak_cons.vhd - keccak_round.vhd - countern.vhd - piso.vhd - keccak_pkg.vhd keccak_fsm1.vhd - sr_reg.vhd - countern.vhd keccak_fsm2.vhd - countern.vhd - sr_reg.vhd sha_fsm3.vhd sha3_pkg.vhd
PolyArith	operation_module.v - address_unit.v - address_resolver.v - Twiddle_resolver.v - dual_port_rom.v - butterfly2x2.v - Butterfly.v - Barret_8380417.v - ntt_pipo.v - ntt_fifo.v - ntt_fifo_piso.v
Decomposer	decomposer_unit.v coeff_decomposer.v - decomp_map1.v
Encoder	encoder.v - zero_strip.v - uncenter_coeff.v
Decoder	decoder.v
MakeHint	makehint.v
UseHint	usehint.v

Sample	gen_a_ext.v (SampleA) gen_c.v (SampleC) gen_s.v (SampleS) expandmask_ext.v (SampleY) sampler_a_ext.v - rejection_a.v sampler_y_ext.v - rejection_y.v sampler_s_ext.v - rejection_s.v

Appendix D: Decomposer Unit Port Diagram



Appendix E: Coeff Decomposer Block Diagram



Appendix F: Coeff Decomposer Testbench

```
`timescale 1 ns/1 ps  // time-unit = 1 ns, precision = 1 ps

module coeff_decomposer_tb;

    reg  rst;
    reg  clk;
    reg  valid_i;
    reg  ready_o;

    reg [2:0]  sec_lvl;
    reg [23:0] di;
    wire [23:0] doa;
    wire [23:0] dob;
    wire valid_o;
    wire ready_i;

    parameter SEC_LVL_0  = 3'b000;
    parameter SEC_LVL_2  = 3'b010;

    parameter OUTPUT_W = 4;
    parameter COEFF_W  = 24;

    integer transaction_num1 = 0;
    integer transaction_num2 = 2;

    coeff_decomposer CDTB(
        .rst(rst),
        .clk(clk),
        .sec_lvl(sec_lvl),
        .valid_i(valid_i),
        .ready_o(ready_o),
        .di(di),
        .doa(doa),
        .dob(dob),
        .valid_o(valid_o),
        .ready_i(ready_i)
    );

    initial begin
        $dumpfile("cdtb.vcd");
        $dumpvars(0,coeff_decomposer_tb);
        di = 24'b0;
        rst     = 1'b1;
        clk     = 1'b0;
        sec_lvl = SEC_LVL_0;
```

```
        #55
        rst     = 1'b0;

        valid_i = 1'b0; //testing with off
        ready_o = 1'b0;
        #100
        di = 24'd2312250; // 4
        #20
        di = 24'd2500010; // 5
        #20
        di = 24'd5000020; // 10
        #20
        di = 24'd8022100; // 15

        #100 //all set to 0s
        valid_i = 1'b1;  //both on
        ready_o = 1'b1;
        #20
        di = 24'd2312250; // 4
        #20
        di = 24'd2500010; // 5
        #20
        di = 24'd5000020; // 10
        #20
        di = 24'd8022100; // 15
        #20
        sec_lvl = SEC_LVL_2; // Changing security levels

        di = 24'd2312250; // 12
        #20
        di = 24'd2500010; // 13
        #20
        di = 24'd5000020; // 26
        #20
        di = 24'd8022100; // 42
        #20
        di = 24'b0;
        #20
        di = 24'd8194721; // 43
        #20

        di = 24'd9000000; // 0
        #20
        di = 24'd10000; // 1
        #20
        di = 24'd16777215; // largest input value
        sec_lvl = SEC_LVL_0;
        repeat(10) #20 di = $urandom_range(0, 8118529);
        sec_lvl = SEC_LVL_2;
```

```
        repeat(10) #20 di = $urandom_range(0, 8285185);
        #1000
        $finish;
    end

    always  #5 clk = !clk;

    always @(di) begin
        $display("transaction input [id: %d]", transaction_num1);
        $display("di = %d", di);
        transaction_num1 = transaction_num1 + 1;
    end

    always @(doa or dob) begin
        if (transaction_num1 >= 4) begin
            $display("transaction output [id: %d]", transaction_num2);
            $display("doa = %d, dob = %d", doa, dob);
            transaction_num2 = transaction_num2 + 1;
        end
    end

endmodule
```