POLITECNICO DI TORINO

Master Degree
in Computer Engineering

Master's Degree Thesis

Additive Fast Fourier Polynomial Multiplier For Code Based Algorithms

Supervisors
prof. Guido MASERA
prof. Maurizio MARTINA
PhD. Alessandra DOLMETA

Candidate
Abdallah EL MOUAATAMID

Academic Year 2023/2024

Abstract

In the modern era, devices require cryptography to protect information against malicious behavior that could exploit and damage the user. To address this, various algorithms have been developed for different purposes. These algorithms leverage complex mathematical problems to make it computationally infeasible to break a system within a reasonable timeframe, even for powerful computers. However, the advent of quantum computers has the potential to significantly impact cryptography by rendering many traditional algorithms vulnerable to attacks. For example, Shor's Algorithm can break the factoring problem of RSA in polynomial time.

In response, NIST initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptography algorithms. Different categories of algorithms were proposed. After four rounds of submission, NIST recommends two primary algorithms to be implemented for most use cases: CRYSTALS-KYBER and CRYSTALS-Dilithium, FALCON, and SPHINCS+. Now, NIST will create new draft standards for the algorithms to be standardized and will coordinate with the submission teams to ensure that the standards comply with the specifications.

NIST expects to select at most one of these two candidates for standardization at the conclusion of the fourth round. Therefore, in this study, code-based algorithms were selected. The study focused on improving the execution of polynomial multiplication within HQC and McEliece, as it was one of the heaviest operations from a computational standpoint in the respective lattice-based algorithms.

The primary objective centers on the development of a polynomial multiplier based on the Fast Fourier Transform. The exploration of the finite fields used by HQC and McEliece led to the subsequent decision to utilize a new class of FFTs, the additive FFTs, which have the advantage of evaluating and interpolating polynomials in fields that do not have the desired nth roots of unity, by exploiting the additive vector space construction of the finite fields. For this specific purpose, the Additive Fast Fourier Transform developed by Gao and Mateer was chosen for multiplying polynomials that lie in finite fields of characteristic two, as the one present in HQC and McEliece.

The research is conducted in two main stages. Initially, a software model of the polynomial multiplier was created to run on all levels of security of HQC and McEliece. Subsequently, The model was then integrated into the official algorithms to verify the correct functioning of the algorithms when working with the multiplier. The second phase involved the development of the hardware structure of the multiplier based on the software model. Employing a memory-based approach to tackle the high number of points to evaluate and interpolate, thereby reducing the hardware cost by minimizing the required number of processing elements.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cryptography is the study and design of methods to protect information against unauthorized access or modification. It serves as the cornerstone in safeguarding sensitive information from unauthorized access or alteration. It plays a pivotal role in ensuring secure communication channels amidst the omnipresent threat posed by adversaries seeking to exploit vulnerabilities in digital systems. However, the landscape of digital security is continually evolving, presenting new challenges that demand innovative solutions.

With the rapid advancement of technologies and the proliferation of sophisticated techniques to breach systems, the efficacy of traditional cryptographic methods is increasingly being called into question. Of particular concern is the looming specter of quantum computing paradigm-shifting technology poised to revolutionize computational capabilities.

Quantum computers harness the principles of quantum mechanics, such as superposition and entanglement, to perform computations that transcend the limitations of classical computing. These systems possess the potential to solve complex mathematical problems, such as integer factorization and discrete logarithms, at an unprecedented speed, rendering conventional cryptographic schemes vulnerable to exploitation.

The foundational security of many prevalent public-key cryptosystems hinges on the computational complexity of these mathematical problems. Consequently, the emergence of quantum computers poses a significant threat to the integrity of digital communications and underscores the urgent need to fortify cryptographic defenses against this impending paradigm shift.

In response to this looming threat, the cryptographic community has embarked on a quest to develop and standardize new cryptographic protocols resilient to quantum attacks. This burgeoning field, known as post-quantum cryptography (PQC), seeks to redefine the cryptographic landscape by furnishing algorithms impervious to the computational prowess of quantum adversaries.

PQC endeavors to furnish cryptographic primitives and protocols that afford the same level of security and functionality as their classical counterparts while leveraging novel mathematical constructs resilient to quantum threats. By embracing innovative approaches and mathematical problems immune to quantum attacks, PQC endeavors to ensure the longevity and resilience of cryptographic systems in the face of impending quantum supremacy

1.2 Thesis organization

This study focuses on developing a polynomial multiplier, able to run on the two main code-based algorithms, McEliece and HQC. This aims to reduce the computational load of the systems, implementing a Hardware-Based Additive Fast Fourier Polynomial Multiplier Accelerator. The development interested first the creation of the software model of the accelerator, and then the hardware design was approached. After this first introductory chapter 1, the thesis is structured as follows:

- **chapter 2** is an introduction to cryptography and post-quantum cryptography, with a particular focus on McEliece and HQC.
- **chapter 3** is dedicated to introducing the main mathematical concepts required for the understanding of the study.
- **chapter 4** presents the software implementation of the polynomial multiplier.
- **chapter 5** discusses the hardware implementation of the accelerator.
- **chapter 6** exhibits and compares the result of the implementation of the accelerator.
- **chapter 7** summarizes the conclusion of this thesis.

Chapter 2

Cryptography

This chapter introduces the basic ideas of cryptography, covering both asymmetric and symmetric cryptography, as well as touching on the main post-quantum cryptographic systems, with a particular focus on McEliece and HQC schemes.

2.1 Cryptography

Cryptography involves using mathematical principles to encode and decode data, ensuring secure storage and transmission of sensitive information over unsecured networks, making it inaccessible to unauthorized individuals. These mathematical concepts are applied to create cryptographic algorithms, which are mathematical functions employed to encrypt and decrypt data. These algorithms take a key and a plaintext as input and produce a ciphertext as output.

Cryptographic algorithms are primarily classified into two categories:

- **Symmetric cryptography**: in this approach, the same secret key is used both for encryption and decryption.

- **Asymmetric cryptography**: this method involves key pairs, consisting of a public key, which is accessible to anyone, and a private key, known only to the key pair owner.

2.1.1 Symmetric cryptography

Symmetric cryptography involves algorithms that employ a single key both for encrypting and decrypting data. In Figure 2.1, the algorithm uses the secret key and plaintext to generate a ciphertext, and vice versa for decryption.

Like every algorithm, Symmetric cryptography has its advantages and disadvantages. It is very fast for encrypting and decrypting data. However, Symmetric cryptography is not secure for transmitting secure data due to the difficulty of secure key distribution.

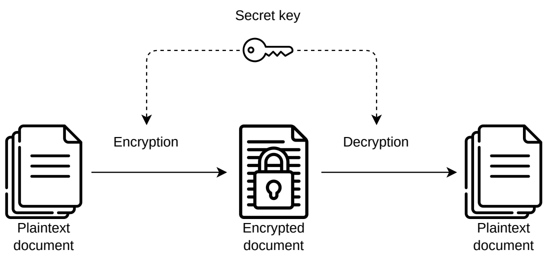


Figure 2.1: Symmetric cryptography

2.1.2 Asymmetric cryptography

Asymmetric cryptography, called Public Key Cryptography, is an asymmetric scheme that uses a pair of keys for encryption and decryption. As shown in Figure 2.2, the private and public keys have a reciprocal function, when one is used to encrypt, the other one must be used to decrypt. This feature provides separate benefits, the primary benefit is that it allows people who have no secure channel to exchange messages securely. This feature

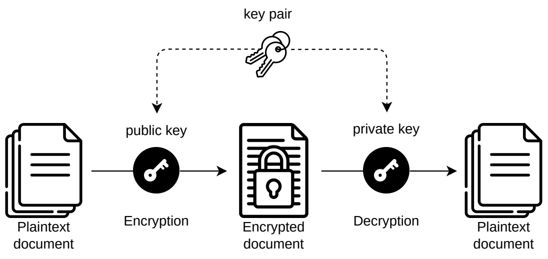


Figure 2.2: Asymmetric cryptography

is for example used to share a common secret key on an unsecured channel because all communications involve only public keys, and no private key is transmitted or shared. The other benefit is that it could be used as a digital signature. Digital signatures let the recipient of information verify the authenticity of the information origin, and also verify that the information was not altered. In this case, the private key is used for encrypting the data, if the information can be decrypted by the public key this demonstrates that the file origin is verified (Figure 2.3).

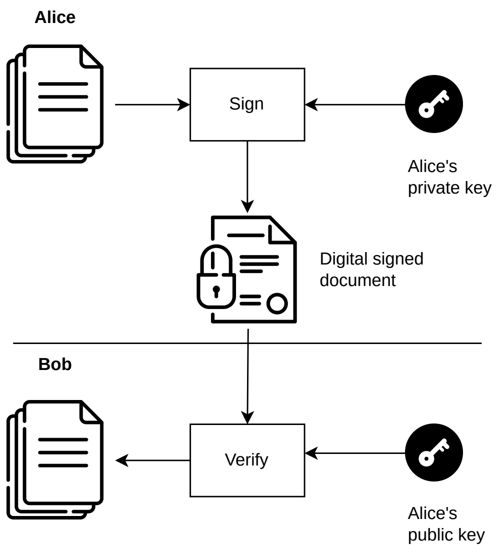


Figure 2.3: Digital signature

2.2 Post Quantum Cryptography

Post-quantum cryptography is the development of cryptographic algorithms, that are resistant to quantum computer attacks. The main development of these algorithms interests public key algorithms, the problem with the popular algorithms such as RSA, DSA, and ECDSA, is that they rely on mathematical problems that could be solved easily by a quantum computer running Shor's algorithm. As of 2023, quantum computers do not present the required computational power to be able to break the widely used cryptographic algorithms. IBM aims to develop a 100,000-qubit system by 2033[1]. Therefore is crucial to develop a cryptographic system able to withstand an attack from a quantum computer. Post Quantum Cryptography research is mostly focused on 4 approaches:

- **Lattice-based cryptography**: based on hard problems over lattices.
- **Multivariate cryptography**: based on the NP-complete problem of solving multivariate equations over a finite field.
- **Hash-based cryptography**: based on finding the isogeny map between two supersingular elliptic curves.
- **Code-based cryptography**: resorts to problems from algebraic coding theory.

This study focuses only on code-based cryptography, with a particular focus on HQC and McEliece schemes.

2.2.1 Code-based cryptography

Code-based cryptographic algorithms offer a unique approach to achieving cryptographic security by leveraging the mathematical properties of error-correcting codes. These algorithms introduce intentional noise into the encoded message, requiring the recipient to effectively filter out this noise to retrieve the original message. This noise-based approach serves as a robust foundation for cryptographic security, as it relies on the inherent complexity of decoding random linear codes.

The pioneering work in this field is credited to Robert J. McEliece, whose seminal contribution laid the groundwork for code-based cryptography. In McEliece's formulation, the private key comprises a linear code $\mathcal{C}$ capable of correcting a certain number of errors. Conversely, the public key $\mathcal{C}'$ is also constructed as a linear code [21]. The encryption process involves encoding the message using the public key $\mathcal{C}'$, while decryption relies on the secret key $\mathcal{C}$ to decode the message accurately.

One of the distinguishing features of code-based cryptographic algorithms is their resilience to quantum attacks. Unlike many traditional cryptographic schemes vulnerable to quantum algorithms, code-based cryptography remains robust against quantum computing threats. This inherent resistance to quantum attacks has positioned code-based cryptography as a promising candidate for securing sensitive information in the era of quantum computing.

Furthermore, ongoing research in code-based cryptography continues to explore new techniques and optimizations to enhance both security and efficiency. By leveraging advancements in coding theory and algorithmic optimization, code-based cryptographic algorithms strive to address emerging challenges and adapt to evolving threat landscapes, ensuring their relevance and efficacy in modern cryptographic applications.

HQC (Hamming-based Quasi-Cyclic) is a cryptographic scheme that falls under the umbrella of code-based cryptography. HQC is designed to provide post-quantum security by leveraging the properties of quasi-cyclic codes derived from Hamming codes. HQC offers a balance between security and efficiency, making it a promising candidate for securing sensitive information in the face of quantum computing threats. Ongoing research and standardization efforts aim to further refine and optimize HQC for widespread adoption in practical cryptographic applications.

Let's now see the two algorithms a little bit more in detail.

Classic McEliece

The Classic Mceliece is a code-based public key encryption scheme. This cryptosystem is based on the Niederreiter framework with binary Goppa code as secret codes. Beginning with the depiction of the scheme. Suppose m is a positive integer such that $q = 2^m$, where n is less than or equal to q, and t is a positive integer greater than or equal to 2, satisfying the condition $mt < n$. Set $k = n - mt$. [21].

Moreover, select a monic irreducible polynomial $f(z) \in \mathbb{F}_2[z]$ of degree m and associate $\mathbb{F}_q$ with $\mathbb{F}_2[z]/f(z)$. Given these conditions, each element in $\mathbb{F}_{2^m}$ can be expressed as

$$u_0 + u_1 z + \ldots + u_{m-1} z^{m-1}$$

corresponding to a unique vector $(u_0, u_1, \ldots, u_{m-1}) \in \mathbb{F}_2^m$.

With these preliminary definitions established, we can now proceed to describe the public key encryption scheme:

- **Key Generation**

  1. Generate a random monic irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ of degree t and select n distinct random elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$.
  2. Compute a parity-check matrix $\tilde{\mathbf{H}} = \{\tilde{h}_{ij}\}_{ij}$ of the binary Gopppa code with parameters $(g, \alpha_1, \ldots, \alpha_n)$ by evaluating $\tilde{h}_{ij} = \alpha_j^{i-1}/g(\alpha_j)$.
  3. Apply an invertible matrix to $\tilde{\mathbf{H}}$ and permute the columns to obtain a matrix in systematic form $\mathbf{H} = (\mathrm{Id}_{n-k}|\mathbf{T})$.
     Denote with $(\alpha'_1, \ldots, \alpha'_n)$ as the permutation of $(\alpha_1, \ldots, \alpha_n)$.
     Note that $(\mathrm{Id}_{n-k}|\mathbf{T})$ is a parity-check matrix of the Goppa code defined by $(g, \alpha'_1, \ldots, \alpha'_n)$

  The **private key** is the $(n+1)$-tuple $\Gamma' = (g, \alpha'_1, \ldots, \alpha'_n)$.
  The **public key** consists of the $(n-k) \times (n-k)$ matrix $\mathbf{T}$ and the number t

- **Encryption**: Encrypt the message as weight t vector $\mathbf{e} \in \mathbb{F}_2^n$ and compute

$$\mathbf{c}_0 = \mathbf{H}\mathbf{e}^\top \in \mathbb{F}_2^{n-k}.$$

- **Decryption**: Extend $\mathbf{c}_0$ to $\mathbf{v} = (\mathbf{c}_0^\top, 0, \ldots, 0) \in \mathbb{F}_2^n$ .The parameters Γ' of the private key define a Goppa code, enabling the use of a decoding algorithm for Goppa codes to find a codeword $\mathbf{c}$ with distance $\leq t$ to $\mathbf{v}$ (if it exists).

  Recover $\mathbf{e}$ as $\mathbf{e} = \mathbf{v} - \mathbf{c}$ and verify that it satisfies $\mathbf{H}\mathbf{e}^\top = \mathbf{c}_0$ and is of weight t.

The decryption process functions as follows: Since $\mathbf{H} = (\mathrm{Id}_{n-k}|\mathbf{T})$, we have

$$\mathbf{H}\mathbf{v}^\top = \mathrm{Id}_{n-k}\,\mathbf{c}_0 = \mathbf{c}_0.$$

Therefore,

$$\mathbf{H}(\mathbf{v} + \mathbf{e})^\top = 0,$$

and $\mathbf{c} = \mathbf{v} - \mathbf{e}$ forms a codeword of the Goppa code defined by Γ'.

Since this code has minimum distance at least $2t - 1$, we deduce that $\mathbf{v} - \mathbf{e}$ is also the unique codeword within distance t from $\mathbf{v}$. Thus, we can recover the error vector as $\mathbf{e} = \mathbf{v} - \mathbf{c}$.

Classic Mceliece, as depicted in Table 2.1, offers various proposed parameter sets, with corresponding input and output sizes for different security levels. Level 1 provides a security level of 128 bits, level 3 offers 192 bits, and level 5 provides 256 bits of security.

Parameter set	m	n	t	Public key	Private key	Ciphertext	Security level
mceliece348864/f	12	3488	64	261120	6492	128	1
mceliece460896/f	13	4608	96	524160	13608	188	3
mceliece6688128/f	13	6688	128	1044992	13932	240	5
mceliece6960119/f	13	6960	119	1047319	13948	226	5
mceliece8192128/f	13	8192	128	1357824	14120	240	5

Table 2.1: Parameters for Classic McEliece

HQC

HQC (Hamming Quasi-Cyclic) is a public key encryption scheme based on code-based cryptography. It leverages the quasi-cyclic framework, combining a decodable code of choice with circulant matrices. Suppose n is such that $(x^n - 1)/(x - 1)$ is irreducible over $\mathbb{F}_2$. Choose a positive integer $k < n$ and an $[n, k]$ linear code $\mathcal{C}$ with an efficient decoding algorithm, having error-correcting capacity given by t [21]. Considering the error weights w, w_r, and w_e, all within the range of $\frac{\sqrt{n}}{2}$, define $R := \mathbb{F}_2[x]/(x^n - 1)$. Any element $a \in R$ can be expressed as:

$$a = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0$$

for unique $a_0, a_1, \ldots, a_{n-1} \in \mathbb{F}_2$. For such an element its Hamming weight is denoted

$$\mathrm{wt}_H(a) = |\{i \in \{0, 1, \ldots, n-1\} \mid a_i \neq 0\}|.$$

It is also possible to associate a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{F}_2^n$ with the element $a = \sum_{i=0}^{n-1} a_i x^i \in R$, and vice versa. In the following description, any bold letter, e.g., $\mathbf{u}$, refers to the associated vector in $\mathbb{F}_2^n$ of an element in R, e.g., $u \in R$.

- **Key generation:** Given the parameters (n, k, t, w, w_e, w_r) , select a generator matrix $\mathbf{G}$ of the code $\mathcal{C}$ and generate a random $h \in R$.
  - **Private key:** Randomly generate a pair $(y, z) \in R^2$ such that $\mathrm{wt}_H(y) = \mathrm{wt}_H(z) = w$.
  - **Public key:** Compute $s = y + hz \in R$. The public key is $(\mathbf{G}, h, s, t)$.

- **Encryption:** randomly generate an element $e \in R$ such that $\mathrm{wt}(e) = w_e$ and a pair $(r_1, r_2) \in R^2$ such that $\mathrm{wt}_H(r_1) = \mathrm{wt}_H(r_2) = w_r$.
  Let $\mathbf{m} \in \mathbb{F}_2^k$ be the message, encrypted as the pair $\mathbf{c} = (\mathbf{u}, \mathbf{v}) \in R^2$, where $u = r_1 + hr_2$ and $\mathbf{v} = \mathbf{mG} + \mathbf{sr}_2 + \mathbf{e}$.

- **Decryption:** According to the quasi-cyclic framework, compute:

$$\mathbf{v} - \mathbf{uz} = \mathbf{mG} + (\mathbf{yr}_2 - \mathbf{r}_1\mathbf{z} + \mathbf{e}).$$

  The term $\mathbf{yr}_2 - \mathbf{r}_1\mathbf{z} + \mathbf{e}$ has Hamming weight $\leq t$ with high probability (this follows non-trivially from the choice of the parameters). If so, use the decoding algorithm of $\mathcal{C}$ to recover the message $\mathbf{m}$.

The table displayed in Table 2.2 provides the proposed parameters for HQC, accompanied by an upper estimate of the decoding failure rate (DFR), as well as the sizes of ciphertext and keys, all expressed in bytes. The security levels correspond to 128-bit, 192-bit, and 256-bit security, respectively.

Security	n	w	$w_r = w_e$	Public key	Private key	Ciphertext	DFR
Level 1	17669	66	75	2249	40	4481	2^{-128}
Level 3	35851	100	114	4522	40	9026	2^{-192}
Level 5	57637	131	149	7245	40	14469	2^{-256}

Table 2.2: Parameters for HQC

Chapter 3

Background

The forthcoming section explores the landscape of mathematical foundations essential for a profound comprehension of the subsequent code implementations. Because of the complexity of the code, it becomes imperative to establish a small mathematical groundwork. This section serves as a gateway, offering a concise exploration of the mathematical concepts and principles that underpin the algorithms and operations to follow.

The chapter begins with an introduction to the concept of finite fields, focusing specifically on the finite field of characteristic 2. Following this, the discussion transitions to the introduction of polynomial multiplication, employing the fast Fourier transform (FFT) for its application. Finally, the chapter concludes by extending the concept of polynomial multiplication through FFTs to finite fields.

3.1 Finite Fields

In mathematics, a finite field, also known as a Galois field, is a field that consists of a finite number of elements. A finite field is characterized by an order p^n, where p is a prime number and n is a positive integer. The field is defined by two operations: multiplication and addition, which satisfy the following properties:

- Elements exhibit closure under addition modulo p^n.
- Elements demonstrate closure under multiplication modulo p^n.
- Every non-zero element possesses a multiplicative inverse.

Fields with prime order can be used to construct prime-power fields, extending the field of smaller order using a primitive polynomial. Considering a polynomial

$$f(x) = a_0 + a_1x + \cdots + a_nx^n, \qquad a_i \in \mathbb{F}_q \tag{3.1}$$

$f(x)$ is defined to be irreducible if it can not be factored into the product of two non-constant polynomials. That implies:

$$f(x) = a_0 + a_1x + \cdots + a_nx^n = 0 \tag{3.2}$$

has no roots in the field $\mathbb{F}_q$.

A primitive element of $\mathbb{F}_q$ is a generator of the multiplicative group of the field. A primitive polynomial is defined as a polynomial in which all the roots belong to the field's set of primitive elements. To form a field $\mathbb{F}_{p^n}$, extending the prime field $\mathbb{F}_p$ necessitates the use of a primitive polynomial of degree n. Defined σ a primitive element, the field is generated as follows:

$$\mathbb{F}_{p^n} = \{0, \sigma, \sigma^2, \ldots, \sigma^{p^n-1}\} \tag{3.3}$$

The common extension fields encountered in this study are the fields $\mathbb{F}_{2^n}$. In the following sections, we will describe in detail the operations.

3.2 Finite Fields Of Characteristic Two

Finite fields of characteristic two are fields that derive from the extension of the field $\mathbb{F}_2$, obtaining as a result a field $\mathbb{F}_{2^n}$. The field elements can be represented as binary vectors of dimension n, relative to a given basis (α_0, α_1, $\ldots$ α_{n-1}) of $\mathbb{F}_{2^n}$ as linear space over $\mathbb{F}_2$. Within this context, the core operations of **addition** and **multiplication** are of paramount importance. In particular:

- The addition operation is seamlessly executed through the exclusive OR (XOR) operation;
- The multiplication is contingent upon the chosen basis.

Beyond these fundamental operations, other crucial algorithms contribute to the robustness and versatility of finite fields of characteristic two. This section will delve into the intricacies of Modular Reduction, Multiplication, Squaring, and Inversion. These algorithms not only enhance the computational efficiency of finite fields but also play a pivotal role in various cryptographic applications and error-correcting codes. Understanding and harnessing these algorithms are imperative for navigating the intricacies of finite fields, particularly in the realm of characteristic two.

3.2.1 Modular Reduction

When performing arithmetic operations, using an irreducible polynomial ensures that the result of arithmetic operations stays within the field. This operation is called modular reduction. The choice of the irreducible polynomial can diminish the complexity of this operation. By choosing $f(x)$ as a low-weight polynomial, One with the minimum number of non-zero coefficients, reduction modulo $f(x)$ becomes an operation that can be executed in time $O(Wn)$, where W is the weight of the irreducible polynomial [3]. In practical contexts, the irreducible polynomial is either a trinomial or pentanomial ($W = 3$ or 5). For a finite field $\mathbb{F}_{2^n}$, empirical studies for values of n into the thousands show that trinomials exist for over half of the values of n covered ([4], [17]). In addition, in all cases where a trinomial is not available, an irreducible polynomial can be found.

Algorithm 1 operates using an irreducible trinomial. It performs reduction of a polynomial of degree $2n-2$, obtaining a polynomial of degree $n-1$.

Algorithm 1 Reduction Modulo $f(x) = x^n + x^t + 1$, $0 < t < n$.

```
Input:  $a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{2n-2}x^{2n-2} \in \mathbb{F}_2[x]$
Output: $r(x) \equiv a(x) \pmod{f(x)}$,   $\deg r(x) < n$ .
1: for $i = 2n-2$ to $n$ by $-1$ do
2:     $a_{i-n} \leftarrow a_{i-n} + a_i$ ,  $a_{i-n+t} \leftarrow a_{i-n+t} + a_i$ .
3: end for
4: return  $r(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$
```

The equivalent for an irreducible pentanomial is defined in Algorithm 2. The considered algorithms operate with an "in-place" method, avoiding the use of extra storage for the result $r(x)$.

Algorithm 2 Reduction Modulo $f(x) = x^n + x^t + x^j + 1$, $0 < t < n$.

```
Input:  $a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{2n-2}x^{2n-2} \in \mathbb{F}_2[x]$
Output: $r(x) \equiv a(x) \pmod{f(x)}$,   $\deg r(x) < n$ .
1: for $i = 2n-2$ to $n$ by $-1$ do
2:     $a_{i-n} \leftarrow a_{i-n} + a_i$ ,  $a_{i-n+t} \leftarrow a_{i-n+t} + a_i$ ,  $a_{i-n+t+j} \leftarrow a_{i-n+t+j} + a_i$ .
3: end for
4: return  $r(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$
```

3.2.2 Multiplication

The multiplication of two elements of $\mathbb{F}_{2^n}$ is the product of two polynomials of degree at most $n-1$ in $\mathbb{F}_2[x]$. The multiplication of elements in $\mathbb{F}_{2^n}$ is a carryless version of the multiplication of two n-bit integers. The majority of the methods for large integer multiplication could be applied, such as Schönhage and Karatsuba algorithms. The multiplication of two elements $A, B \in \mathbb{F}_{2^m}$, with $A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i$ and $B(\alpha) = \sum_{i=0}^{m-1} b_i\alpha^i$ is given as:

$$C(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i \equiv A(\alpha) \cdot B(\alpha) \bmod F(\alpha)$$

where the multiplication is polynomial multiplication, and all α^t , with $t \geq m$ are reduced with $F(\alpha)$ [11]. Two algorithms can be applied to calculate the product :

- Shift-and-add method with the reduction step inter-leaved [10]
- Comb method [15]

The shift-and-add method with the reduction step interleaved performs the multiplication by interleaving shift and XOR with the modular reduction, as shown in Algorithm 3.

Software implementation of this algorithm is not suitable due to the difficulty of executing bitwise shifts across the words on a processor

Algorithm 3 Shift-and-Add Most Significant Bit (MSB) first $\mathbb{F}_{2^m}$ multiplication

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$ where $a_i, b_i \in \mathbb{F}_2$
Output: $C = A \cdot B \mod F(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$ where $c_i \in \mathbb{F}_2$
1: $C \leftarrow 0$
2: for $i = m-1$ downto 0 do
3:     $C_i \leftarrow b_i \cdot (\sum_{i=0}^{m-1} a_i\alpha^i) + (\sum_{i=0}^{m-1} c_i\alpha^i) \cdot \alpha \mod F(\alpha)$
4: end for
5: Return $(C)$
```

The Comb method shown in Algorithm 4 presents a more efficient way to implement a multiplier. The operation is managed in two separate steps: first, it performs the polynomial multiplication, obtaining a $2n$-bit length polynomial; then, it reduces the polynomial to the length n using modular reduction.

Algorithm 4 Comb Method for $\mathbb{F}_{2^m}$ multiplication on a w-bit processor

```
Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$ where $a_i, b_i \in \mathbb{F}_2$
Output: $C = A \cdot B \mod F(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$ where $c_i \in \mathbb{F}_2$
1: $C \leftarrow 0$
2: for $j = 0$ to $w-1$ do
3:     for i = 0 to $s-1$ do
4:         $C \leftarrow b_{wi+j} \cdot \text{SHIFT}(A << w.i) + C$
5:     end for
6:     $A \leftarrow \text{SHIFT}(A << 1)$
7: end for
8: Return $(C)$
```

3.2.3 Squaring

Squaring is an operation that does not require XOR and shifting. To square a polynomial:

$$\begin{aligned} C &\equiv A^2 \bmod F(\alpha) \\ &\equiv (a_{m-1}\alpha^{2(m-1)} + a_{m-2}\alpha^{2(m-2)} + \ldots + a_1\alpha^2 + a_0) \bmod F(\alpha) \end{aligned} \tag{3.4}$$

Polynomial squaring is implemented by increasing the size of C to double its bit-length and interleaving 0 bits in between the original bits of C, then reducing the double-length result [11].

3.2.4 Inversion

The inversion is usually computed using the extended Euclidean algorithm. This algorithm is an extension of the Euclidean algorithm, which is used to find the greatest common divisor (GCD) of two integers. The extended version also finds the coefficients of Bézout's identity. Inversion is an operation that is significantly slower than multiplication. An alternative to the extended Euclidean algorithm is to perform inversion by replacing this operation with a chain of multiplications and squaring [3]. These schemes are based on the field equation, which can be reformulated as:

$$\beta^{-1} = \beta^{2^n-2} = \left(\beta^{2^{n-1}-1}\right)^2 \tag{3.5}$$

for all $\beta \neq 0$ in $\mathbb{F}_{2^n}$. The objective is to reduce the number of multiplications by utilizing squaring, which is cheaper. A technique to minimize the number of multiplications is described by Itoh and Tsuji (**Itoh–Tsujii inversion algorithm**) [8]. The method is based on the identities:

$$\beta^{2^{n-1}-1} = \begin{cases} \beta^{(2^{\frac{n-1}{2}}-1)(2^{\frac{n-1}{2}}+1)} = \left(\beta^{2^{\frac{n-1}{2}}-1}\right)^{2^{\frac{n-1}{2}}} \beta^{2^{\frac{n-1}{2}}-1}, & n \text{ odd,} \\ \beta\beta^{2^{n-1}-2} = \beta\left(\beta^{2^{n-2}-1}\right)^2, & n \text{ even.} \end{cases} \tag{3.6}$$

3.3 Polynomial Multiplication

Given two polynomials $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$, the result of the product is a polynomial $C(x) = \sum_{j=0}^{2n-2} c_j x^j$ where $c_j = \sum_{k=0}^{j} a_k b_{j-k}$. Computing $C(x)$ using the schoolbook method would require $\Theta(n^2)$. There are methods that efficiently solve polynomial multiplication.

- Karatsuba algorithm $\Theta(n^{\log_2 3})$
- Fast Fourier Transform $\Theta(n \lg n)$

In this study, we will explore the techniques and methodologies related to polynomial multiplication, focusing particularly on the application of the FFT. This exploration will be structured into the following subsections:

1. **Polynomial Representation**: Discussing different representations of polynomials, namely coefficient representation and point-value representation, and the operations involved in converting between the two representations.
2. **Fast Multiplication of Polynomials in Coefficient Form**: Introducing practical methods for multiplying polynomials represented in coefficient form efficiently, laying the groundwork for the subsequent discussion on FFT.
3. **Discrete Fourier Transform (DFT)**: Providing an overview of DFT, which serves as the foundation for FFT, and its significance in polynomial multiplication.
4. **Fast Fourier Transform (FFT)**: Exploring the FFT algorithm, its divide-and-conquer strategy, and its application in polynomial multiplication.

3.3.1 Polynomial representation

A polynomial could be represented in two different ways:

- **Coefficient representation**.
  The coefficient representation of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree n is an array of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$.
- **Point-value representation**.
  The point-value representation of a polynomial $A(x)$, bounded by a degree of n, consists of a collection of n pairs of point-value coordinates:

  $$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

  where each x_k is unique, and $y_k = A(x_k)$ for $k = 0,1, \ldots, n-1$. Each polynomial can be represented with different point-value pairs.

Converting from one representation to the other can be done through two operations:

- from coefficient to point-value, it is required to evaluate the polynomial in coefficient representation in n distinct points. Using Horner's method requires $\Theta(n^2)$ time to evaluate a polynomial of length n.
- from point-value to coefficient, the operation that is performed is called interpolation. Starting from a set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$, the coefficients are computed by
$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1}y \tag{3.7}$$
where V is the inverse Vandermode matrix and y is the column vector of the y_i points. Solving the linear equations, requires $O(n^3)$. A more efficient algorithm for n-point interpolation relies on Lagrange's formula:
$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x - x_j)}{\prod_{j\neq k}(x_k - x_j)} \ . \tag{3.8}$$
The algorithms based on the Lagrange's formula require $O(n^2)$.

The different representations have different advantages. Given two polynomials, computing the polynomial multiplication in the point-value form takes $O(n)$, while in the coefficient form, the time required is $O(n^2)$.

3.3.2 Fast multiplication of polynomials in coefficient form

For practical reasons, polynomials are always represented in coefficient form. To make the use of polynomial multiplication in point-value form feasible, an efficient algorithm for evaluation and interpolation should be devised. The **Fast Fourier Transform**, using the special properties of the complex roots of unity as the evaluation points, evaluates and interpolates a polynomial in $\Theta(n \log n)$.
This approach is summarized in 3 steps, Fig 3.1:

1. **Evaluate**: compute the point-value representations of $A(x)$ and $B(x)$ of length $2n$, by applying the FFT of order $2n$ on each polynomial.
2. **Pointwise multiply**: compute
$$C(x) = A(x)B(x)$$
by multiplying these values together pointwise.
3. **Interpolate**: compute the coefficient representation of $C(x)$ applying the IFFT on the point-values pairs.

In terms of computational complexity, the required steps exhibit the following complexities:

- Evaluation: $\Theta(n \lg n)$
- Pointwise Multiplication: $\Theta(n)$
- Interpolation: $\Theta(n \lg n)$

Overall the complexity over the entire computation is $\Theta(n \lg n)$.

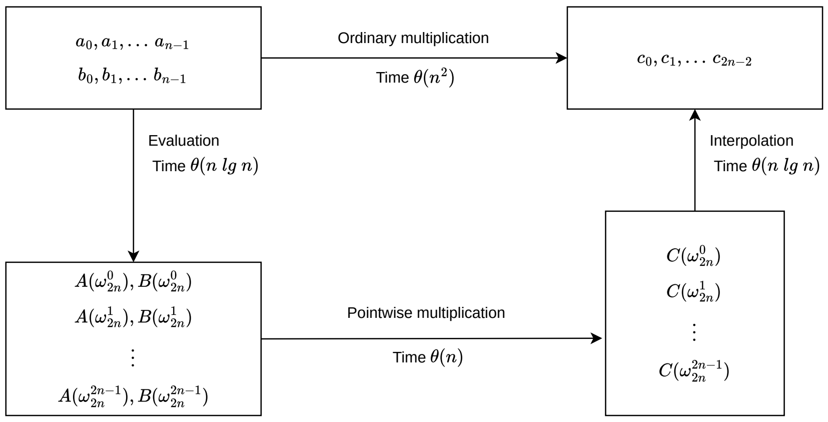


Figure 3.1: Fast Polynomial Multiplication

3.3.3 Discrete Fourier Transform

Before diving into the Fast Fourier Transform, let's first describe the starting point of the fast Fourier transform: the Discrete Fourier Transform. The Discrete Fourier Transform is an operation that transforms a sequence of N complex numbers into another sequence of complex numbers.

A complex nth root of unity refers to a complex number ω satisfying the equation:

$$\omega^n = 1$$

There are n complex nth roots of unity, that are described by $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$. In Figure 3.2, it is possible to observe the 8th roots of unity. the value $\omega_n = e^{2\pi i/n}$ is

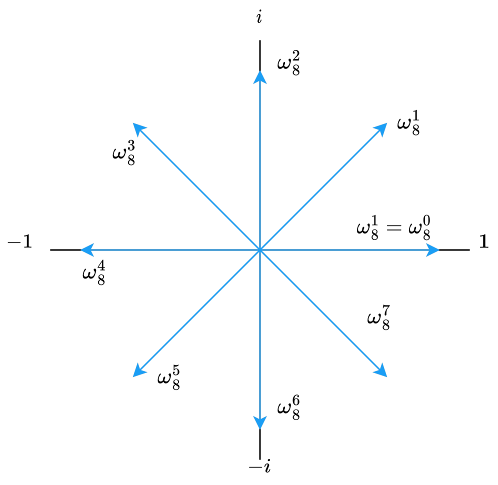


Figure 3.2: 8th roots of unity

called the principal nth root of unity. All the other values are powers of ω_n.

Given $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n, the polynomial will be evaluated at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$. The y_k points are computed as:

$$y_k = A({\omega_n}^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{3.9}$$

the vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the discrete Fourier transform of the polynomial $A(x)$.

3.3.4 Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm designed to efficiently compute the Discrete Fourier Transform (DFT) and its inverse (IDFT) in $\Theta(n \log n)$ time complexity. It achieves this efficiency by leveraging the properties of complex roots of unity.
When addressing a problem of size n, particularly when n is a power of 2, the Fast Fourier Transform (FFT) method utilizes a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ separately to define two new polynomials $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ of degree-bound $n/2$. In particular:

$$\begin{aligned} A^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ A^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned} \tag{3.10}$$

Considering that $A(x)$ can be presented as:

$$A(x) = A^{\text{even}}(x^2) + x A^{\text{odd}}(x^2) \tag{3.11}$$

the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to :

1. evaluating the degree-bound $n/2$ polynomials $A^{\text{even}}(x^2)$ and $A^{\text{odd}}(x^2)$ at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$$

2. combining the results according to the equation(3.11).

Applying the halving lemma, the list of values is halved, resulting in $n/2$ complex $(n/2)$th roots of unity, where each root appears exactly twice. Therefore, the FFT recursively evaluates the two polynomials of degree $n/2$ at $n/2$ complex $(n/2)$th roots of unity.

The final algorithm is described in Algorithm 5.

Algorithm 5 Fast Fourier Transform

```
FFT(a,n)
1: **if** $n == 1$
2:     **return** $a$
3: $\omega_n = e^{2\pi i/n}$
4: $\omega = 1$
5: $a^{\text{even}} = (a_0, a_2, \dots, a_{n-2})$
6: $a^{\text{odd}} = (a_1, a_3, \dots, a_{n-1})$
7: $y^{\text{even}} = \text{FFT}(a^{\text{even}}, n/2)$
8: $y^{\text{odd}} = \text{FFT}(a^{\text{odd}}, n/2)$
9: for $k = 0$ to $n/2 - 1$
10:     $y_k = y_k^{\text{even}} + \omega\, y_k^{\text{odd}}$
11:     $y_{k+(n/2)} = y_k^{\text{even}} - \omega\, y_k^{\text{odd}}$
12:     $\omega = \omega\omega_n$
13: **return** y
```

The FFT works as follows. Lines 1-2, represent the base case of the recursion. Lines 5-6, divide the problem into two subproblems. Lines 3,4, and 12 keep ω update. Lines 7-8 compose the two recursive calls. Lines 10-11 are used to combine the results of the two recursive calls. The inverse of the Fast Fourier Transform can be constructed by performing the reverse operations in reverse order.

3.4 Fast Polynomial Multiplication in Finite Fields

The previous section explained how polynomial multiplication can be computed in $\Theta(n \log n)$ time by exploiting the Fast Fourier Transform as a means to evaluate and interpolate polynomials. Various techniques can be employed for polynomial multiplication when coefficients belong to a finite field, with common methods including Karatsuba and the Fast Fourier Transform. This study specifically concentrates on the utilization of modified Fast Fourier Transform variants to optimize the execution time of polynomial multiplication within finite fields.

The two algorithms that will be treated are:

- Number Theoretic Transform (discussed in subsection 3.4.1);
- Additive Fast Fourier Transform (discussed in subsection 3.4.2);

The same approach is used to perform the multiplication:

1. Evaluation
2. Pointwise multiplication
3. Interpolation

In this case, the arithmetic operations are substituted with the relative operations of the field.

3.4.1 Number Theoretic Transform

The Number Theoretic Transform is a special version of the discrete Fourier transform over a finite field [14]. Given a field $\mathbb{Z}_q$, in order to apply the Number Theoretic Transform, the primitive n-th root of unity ω_n should exist [13]. The transformed polynomial is defined as $\hat{\boldsymbol{a}} = \text{NTT}(\boldsymbol{a})$, where

$$\hat{a}_j = \sum_{i=0}^{n-1} a_i \omega_n^{ij} \bmod q, j = 0, 1, \ldots, n-1. \tag{3.12}$$

The inverse transform, denoted by INTT, is defined as $\boldsymbol{a} = \text{INTT}(\hat{\boldsymbol{a}})$, where

$$a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-ij} \bmod q, i = 0, 1, \ldots, n-1. \tag{3.13}$$

3.4.2 Additive Fast Fourier Transform

The Additive FFT algorithms are a class of algorithms that are based on the additive properties of the field. Traditional FFT algorithms like the Number Theoretic Transform and Fast Fourier Transform are applicable when the n points form an n-th roots of unity (thus constituting a cyclic multiplicative group of order n), where n is either a power of 2 or a product of small primes [7]. These FFT algorithms rely on the factorization of the polynomial $x^n - 1$, which corresponds to the subgroup structure of the multiplicative group of order n. Due to this factorization, these transforms are often referred to as multiplicative FFTs.

The limitation of multiplicative Fast Fourier Transforms (FFTs) lies in their inability to handle problem sizes n that are not products of small primes, or when the underlying fields lack the necessary n-th roots of unity. This issue arises notably in Fourier Transforms over finite fields. When working with a field $\mathbb{F}_{2^k}$, where n is a power of two, the field cannot support multiplicative FFTs of length n due to the absence of primitive n-th roots of unity in any field with characteristic two [7].
An additive Fast Fourier Transform (FFT) is characterized by having evaluation points that form an additive group. In these algorithms, the polynomial is evaluated at each of the roots of the polynomial $x^n - x$. By employing an Additive FFT is possible to perform the evaluation and interpolation on those finite fields where the classic multiplicative FFTs are not an option.
There are different algorithms based on the Additive Fast Fourier Transform. The major ones are:

- The additive FFT of von zur Gathen and Gerhard [19]
- The additive FFT of Wang and Zhu [20]
- The additive FFT of Cantor [5]
- Frobenius Additive Fast Fourier Transform [12]
- Gao and Mateer Additive Fast Fourier Transform [7]

This study opts for the Additive Fast Fourier Transform (AFFT) as the primary focus, citing the abundance of resources on the Gao and Mateer Additive Fast Fourier Transform and its widespread application in decoding Reed-Solomon codes. Consequently, in subsequent references, the Gao and Mateer Additive Fast Fourier Transform will be denoted simply as the Additive Fast Fourier (AFFT).

Gao and Mateer Additive Fast Fourier Transform

Starting from a field $\mathbb{F}_{2^m}$, the number of elements of the field is 2^m.
Let $n = 2^m$ and $f \in \mathbb{F}[x]$ of degree less than n. Define $B = \langle \beta_1, \ldots, \beta_m \rangle = \{a_1\beta_1 + \cdots + a_m\beta_m : a_1, \ldots, a_m \in \mathbb{F}_2\}$, where β_i are linearly independent over $\mathbb{F}_2$.
The elements of B are ordered as follows. For any $0 \leq i \leq 2^m$ the binary representation of i is

$$i = a_1 + a_2 \cdot 2 + \cdots + a_m \cdot 2^{m-1} = (a_1, a_2, \cdots, a_m)_2, \tag{3.14}$$

where every $a_j = 0$ or 1. The ith element of B is presented as

$$B[i] = a_1\beta_1 + a_2\beta_2 + \cdots + a_m\beta_m. \tag{3.15}$$

provided a polynomial $f(x)$ of degree less than n, the Additive Fast Fourier Transform computes the values of f at specified points within a set B, represented as

$$\mathrm{FFT}(f, m, B) = (f(B[0]), f(B[1]), \ldots, f(B[n-1])). \tag{3.16}$$

The algorithm reduces the problem of size $n > 1$ to two problems of half the dimension.

Algorithm 6 Additive FFT of length $n = 2^m$ (arbitrary m)

```
Input:   $(f, m, B)$ where $m \geq 1$, $f(x) \in \mathbb{F}[x]$ of degree $< n = 2^m$ , and
         $B = \langle \beta_1, \ldots, \beta_m \rangle$ where $\beta_i$'s are linearly independent over $\mathbb{F}_2$.
Output:  FFT$(f, m, B) = (f(B[0]), f(B[1]), \ldots, f(B[n-1]))$.

Step 1.  If $m = 1$ then return $(f(0), f(\beta_1))$
Step 2.  Compute $g(x) = f(\beta_m x)$.
Step 3.  Compute the Taylor expansion of $g(x)$ and
         let $g_0(x)$ and $g_1(x)$ be as in (3.17).
Step 4.  Compute $\gamma_i = \beta_i \cdot \beta_m^{-1}$ and $\delta_i = \gamma_i^2 - \gamma_i$ for $1 \leq i \leq m-1$
         Let $G = \langle \gamma_1, \ldots, \gamma_{m-1} \rangle$, and $D = \langle \delta_1, \ldots, \delta_{m-1} \rangle$.
Step 5.  Let $k = 2^{m-1}$ . Compute
             FFT($g_o$,$m-1$,D) $= (u_0, u_1, \ldots, u_{k-1})$, and
             FFT($g_1$,$m-1$,D) $= (v_0, v_1, \ldots, v_{k-1})$.
Step 6.  For $0 \leq i < 2^{m-1}$ , set $w_i = u_i + G[i] \cdot v_i$ and $w_{k+i} = w_i + v_i$
Step 7.  Return $(w_0, w_1, \ldots, w_{n-1})$.
```

In Algorithm 6, the AFFT is presented. It employs the same structure as traditional FFTs. In the previous FFT, the problem was reduced by dividing $A(x)$ into two functions, $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$, each with a degree which is half of the original polynomial. In the additive FFT, this is achieved using the Taylor expansion, as shown in step 3 of Algorithm 6.
Starting from the polynomial $g(x)$, the Taylor expansion computes

$$h_0(x), h_1(x), \ldots, h_{m-1}(x) \in \mathbb{F}[x]$$

such that

$$f(x) = h_0(x) + h_1(x) \cdot (x^t - x) + \cdots + h_{m-1}(x) \cdot (x^t - x)^{m-1}$$

In the finite fields of characteristic two, the result is represented as:

$$f(x) = \sum_{i=0}^{N/2-1} h_i(x) \cdot (x^2 - 1)^i,$$

The Algorithm 7 describes the Taylor expansion.

Algorithm 7 TaylorExpansion(f,N)

```
Input: $f \in \mathbb{F}[x]$ of degree $< N$
Output: The Taylor expansion $\{h_0, \dots, h_{N/2-1}\}$ of $f$ w.r.t. $(x^2 - x)$
 1: if $N \leq 2$ then
 2:     return $\{f\}$
 3: $k \leftarrow N/4$
 4: $g_0 \leftarrow \sum_{i=0}^{2k} f_i \cdot x^i$
 5: $g_1 \leftarrow \sum_{i=0}^{2k} f_{i+2k} \cdot x^i$
 6: for $i = 0, \dots, k-1$ do
 7:     $g_1(x) \leftarrow g_1(x) \oplus g_{1,i+k} \cdot x^i$
 8:     $g_0(x) \leftarrow g_0(x) \oplus g_{1,i} \cdot x^{i+k}$
 9: $V_0 \leftarrow$ TaylorExpansion $(g_0, N/2)$
10: $V_1 \leftarrow$ TaylorExpansion $(g_1, N/2)$
11: return $V_0||V_1$
```

This operation performs only field additions. At the end, the original polynomial can be described as

$$g(x) = g_0(x) + (x^2 - x)g_1(x), \tag{3.17}$$

where $g_0(x)$ and $g_1(x)$ are polynomials dimensions halved with respect to the original. The taylor expansion in terms of complexity requires $\frac{1}{2}N \log N - \frac{1}{2}N$ additions. In step 2 of Algorithm 6, the weighted polynomial is computed. In step 4, Gammas and Deltas are calculated using

$$\gamma_i = \beta_i \cdot \beta_m^{-1}, \quad 1 \leq i \leq m-1, \tag{3.18}$$

$$\delta_i = \gamma_i^2 - \gamma_i \quad 1 \leq i \leq m-1, \tag{3.19}$$

obtaining

$$D = \langle \delta_1, \dots, \delta_{m-1} \rangle$$

$$G = \langle \gamma_1, \dots, \gamma_{m-1} \rangle,$$

where delta is the new basis for the next recursive call, and gamma is used in step 6. Step 6, combines the result coming from the recursive calls through the butterfly.

Chapter 4

Software Implementation

This chapter is dedicated to the description of the software model of the polynomial multiplier. The core element of this model is the Additive Fast Fourier Transform, which will be our method for evaluating and interpolating polynomials. Functionally, the model is organized as follows

- Evaluation of the two polynomials $A(x)$ and $B(x)$
- Pointwise multiplication of the coefficients of $A(x)$ and $B(x)$
- Interpolation of the resulting polynomial $C(x)$

Before and after these steps, depending on the algorithm and the level of security chosen, there will be steps of pre-processing and post-processing of the data to follow the specifications of the algorithms. The following section will present:

- The arithmetic unit, and all its elements;
- The Additive Fourier Transform;
- The complete multiplier.
- The results achieved after integrating the models in HQC and McEliece

The present section will deal with the presentation of the C code of the different functions interesting the polynomial multiplier. The model will reuse and readapt algorithms coming from the PQClean library [1], McEliece[2] and HQC[3] official algorithms.

[1]https://github.com/PQClean/PQClean

[2]https://classic.mceliece.org/impl.html

[3]https://pqc-hqc.org/implementation.html

4.1 Arithmetic Unit

The arithmetic operations required for the model are based on the algebra of the finite fields of characteristic two, described in the previous section. In the various algorithms, the fields of interest are:

- $\mathbb{F}_{2^{12}}/(z^{12} + z^3 + 1)$ and $\mathbb{F}_{2^{13}}/(z^{13} + z^4 + z^3 + z + 1)$ for McEliece
- $\mathbb{F}_{2^{16}}/(z^{16} + z^5 + z^3 + z^2 + 1)$ for HQC

Each field is characterized by an irreducible polynomial used to reduce an element outside of the field during arithmetic operations. The model necessitates the following operations: squaring, inversion, multiplication, exponentiation, and addition. Each operation is specifically developed for a particular field, except for addition, which is the same for all.

In the implementation of each finite field, distinct source files detail the operations involved. To enhance modularity and streamline functionality, a wrapper function is crafted for each operation. These wrapper functions serve as centralized points of access, organizing the execution of multiple underlying functions. Notably, the software architecture capitalizes on the wealth of algorithms already delineated in the official codebases of HQC and McEliece. This decision is driven by the desire to expedite model development, and this ensures an efficient integration of well-established cryptographic algorithms.

4.1.1 Multiplication

As explained in subsection 3.2.2, there are two styles of algorithms that can be applied: Comb Method and the shift-add method.
For **McEliece**, the authors opted for the Comb Method; therefore, the multiplication is conducted in two steps: multiplication and then reduction. In Code 4.1, defined inside the file **gf.c** of **mceliece460896** from the PQClean library, the multiplication is performed in lines 15-16 using bitwise operations, storing the intermediate results in the variable **tmp**. Finally, from line 19 to line 23, the product is reduced using the irreducible polynomial $f(x) = z^{12} + z^3 + 1$. The code implements Algorithm 1 as the modular reduction, through masking and shifting.

The implementation discussed involved the field $\mathbb{F}_{2^{13}}$, and the same approach is applied for $\mathbb{F}_{2^{12}}$. The difference lies in the number of iterations in the for loop and the mask and shifting done in the reduction step.

For **HQC**, the implementation of multiplication does not follow the Comb Method but instead the shift-and-add method. Therefore, each step considers an intermediate product followed by the reduction. The Code 4.2, defined in the file **gf_16.c** of the software model, implements the multiplication in $\mathbb{F}_{2^{16}}$ having $f(z) = z^{16} + z^5 + z^3 + z^2 + 1$ as the reduction polynomial. from line 7 to line 15, the multiplication and reduction are performed using bitwise operations. At each iteration of the loop, for each bit of **b** that is set, **a** is added to the intermediate result. if the most significant bit of **a** is set to 1, **a** is shifted to the left by one position and reduced with the irreducible polynomial.

```c
uint16_t gf_mul_13(uint16_t in0, uint16_t in1)
{
  int i;

  uint64_t tmp;
  uint64_t t0;
  uint64_t t1;
  uint64_t t;

  t0 = in0;
  t1 = in1;

  tmp = t0 * (t1 & 1);

  for (i = 1; i < GFBITS_13; i++)
    tmp ^= (t0 * (t1 & (1 << i)));


  t = tmp & 0x1FF0000;
  tmp ^= (t >> 9) ^ (t >> 10) ^ (t >> 12) ^ (t >> 13);

  t = tmp & 0x000E000;
  tmp ^= (t >> 9) ^ (t >> 10) ^ (t >> 12) ^ (t >> 13);

  return tmp & GFMASK_13;
}
```

Code 4.1: gf_mul_13

4.1.2 Squaring

In the fields $\mathbb{F}_{2^{12}}$ and $\mathbb{F}_{2^{13}}$, squaring is implemented following Equation 3.4. The equation is implemented through two phases: squaring and then reduction.

In the case of $\mathbb{F}_{2^{12}}$, in Code 4.3, defined in **gf.c** of **mceliece348864** in the PQClean library, the first four lines after the variables declaration perform bit manipulation on **x** using the 4-bit masks defined in array **B**. This corresponds to interleaving 0 bits in between the original bits of the input. The last lines compute the reduction through the irreducible polynomial using masking and shifting.

In the codebase of the McEliece algorithms, which employ the field $\mathbb{F}_{2^{13}}$, it is possible to encounter other variants of squaring that are used to perform efficiently the inversion. These are:

- **gf_sq2_13(in)**, corresponds to performing $out = (in^2)^2$
- **gf_sqmul_13(in,m)** corresponds to performing $out = (in^2)\ m$
- **gf_sq2_13mul(in,m)**, corresponds to performing $out = ((in^2))^2\ m$

The normal square (in^2) in $\mathbb{F}_{2^{13}}$ is performed through **gf_sqmul_13(in, 1)**.
In HQC where the field employed is $\mathbb{F}_{2^{16}}$, the operation is done using multiplication.

```c
uint16_t gf_mul_16(uint16_t a, uint16_t b)
{
    uint32_t result = 0;
    uint32_t mask = 1;

    for (int i = 0; i < 16; ++i) {
        if (b & mask) {
            result ^= a;
        }
        if (a & 0x8000) {
            a = (a << 1) ^ IRRED_POLY;
        } else {
            a <<= 1;
        }
        mask <<= 1;
    }

    return result;
}
```

Code 4.2: gf_mul_16

```c
uint16_t gf_square_12(uint16_t in)
{
    const uint32_t B[] = {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF
   };

    uint32_t x = in;
    uint32_t t;

    x = (x | (x << 8)) & B[3];
    x = (x | (x << 4)) & B[2];
    x = (x | (x << 2)) & B[1];
    x = (x | (x << 1)) & B[0];

    t = x & 0x7FC000;
    x ^= t >> 9;
    x ^= t >> 12;

    t = x & 0x3000;
    x ^= t >> 9;
    x ^= t >> 12;

    return x & ((1 << GFBITS_12) - 1);
}
```

Code 4.3: gf_square_12

4.1.3 Inversion and Exponentiation

Inversion in all three fields $\mathbb{F}_{2^{12}}$, $\mathbb{F}_{2^{13}}$, and $\mathbb{F}_{2^{16}}$ is implemented using an addition chain of squaring and multiplication. In particular, as explained previously in subsection 4.1.2, in $\mathbb{F}_{2^{13}}$, variant functions of squaring were developed by the authors to aid in the computation. The inversion makes use of the relationship in Equation 3.5, therefore computing β^{2^n-2} is equivalent to computing β^{-1}.
In $\mathbb{F}_{2^{12}}$, $\beta^{2^n-2} = \beta^{4094}$, To achieve the inversion, the operations in Algorithm 8 are performed.

Algorithm 8 Inversion in $\mathbb{F}_{2^{12}}$

```
1: out = gf_square_12(in)= $in^2$
2: tmp_11 = gf_mul_12(out, in)=$in^2 \cdot in = in^3$
3: out = gf_square_12(tmp_11)=$in^6$
4: out = gf_square_12(out)=$in^{12}$
5: tmp_1111 = gf_mul_12(out, tmp_11)=$in^{12} \cdot in^3 = in^{15}$
6: out = gf_square_12(tmp_1111)=$in^{30}$
7: out = gf_square_12(out)=$in^{60}$
8: out = gf_square_12(out)=$in^{120}$
9: out = gf_square_12(out)=$in^{240}$
10: out = gf_mul_12(out, tmp_1111)=$in^{240} \cdot in^{15} = in^{255}$
11: out = gf_square_12(out)=$in^{510}$
12: out = gf_square_12(out)=$in^{1020}$
13: out = gf_mul_12(out, tmp_11)=$in^{1020} \cdot in^3 = in^{1023}$
14: out = gf_square_12(out)=$in^{2046}$
15: out = gf_mul_12(out, in)=$in^{2046} \cdot in^1 = in^{2047}$
16: out = gf_square_12(out)=$in^{4094}$
```

In $\mathbb{F}_{2^{13}}$, $\beta^{2^n-2} = \beta^{8190}$, To achieve the inversion, the operations in Algorithm 9 are performed.

Algorithm 9 Inversion in $\mathbb{F}_{2^{13}}$

```
1: tmp_11 = gf_sqmul(in, in) = $in^2 \cdot in = in^3$
2: tmp_1111 = gf_sq2mul(tmp_11, tmp_11) = $(in^3)^2)^2 \cdot in = in^{15}$
3: out = gf_sq2(tmp_1111) = $((in^{15})^2)^2 = in^{60}$
4: out = gf_sq2mul(out, tmp_1111) = $(in^{60})^2)^2 \cdot in^{15} = in^{255}$
5: out = gf_sq2(out) = $((in^{255})^2)^2 = in^{1020}$
6: out = gf_sq2mul(out, tmp_1111) = $(in^{1020})^2)^2 \cdot in^{15} = in^{4095}$
7: out = gf_sqmul(out, 1) = $((in^{4095})^2 \cdot 1 = in^{8190}$
```

In $\mathbb{F}_{2^{13}}$, $\beta^{2^n-2} = \beta^{65534}$, To achieve the inversion, the operations in Algorithm 10 are performed.

Algorithm 10 Inversion in $\mathbb{F}_{2^{16}}$

```
1: b = gf_square_16(a) = $a^2$
2: c= gf_mul_16(b,a) = $a^3$
3: d = gf_square_16(b) = $a^4$
4: e = gf_mul_16(c,d) = $a^7$
5: f = gf_mul_16(e,d) = $a^{11}$
6: g = gf_mul_16(f,d) = $a^{15}$
7: h = gf_square_16(g) = $a^{30}$
8: h = gf_square_16(h) = $a^{60}$
9: h = gf_square_16(h) = $a^{120}$
10: i = gf_mul_16(h,e) = $a^{127}$
11: j= gf_square_16(i) = $a^{254}$
12: k= gf_mul_16(j,a) = $a^{255}$
13: l= gf_square_16(k) = $a^{510}$
14: l= gf_square_16(l) = $a^{1020}$
15: l= gf_square_16(l) = $a^{2040}$
16: l= gf_square_16(l) = $a^{4080}$
17: l= gf_square_16(l) = $a^{8160}$
18: n=gf_square_16(l)= = $a^{16320}$
19: n=gf_square_16(n) = $a^{32640}$
20: n=gf_square_16(n) = $a^{65280}$
21: b=gf_mul_16(n,j) = $a^{65534}$
```

The addition chains used for McEliece, are the ones already present in the official algorithms, while for HQC the addition chain has been generated from a tool.

The exponentiation has been developed as an iterative algorithm performing multiplication. All three algorithms resort to this approach.

4.2 Additive Fast Fourier Transform

The Additive Fast Fourier Transform is the direct development of the algorithm presented by Gao and Mateer in software. The development of the Additive Fast Fourier Transform started first from the recursive form presented in the paper by Gao and Mateer [7], followed by the work done in the Additive Fast Fourier Transform in HQC, which includes optimizations proposed by Bernstein, Chou, and Schwabe [2], and the work done by the library Quadiron, by Vianney Rancurel and Lam Pham-Sy [18]. The additive Fast Fourier Transform can be divided into multiple steps.

The algorithm takes as input a polynomial with coefficients in $\mathbb{F}_{2^m}$ with a degree less than 2^m(where m is the power of the field) and a beta basis. The recursive algorithm will divide the problem into two subproblems of half the dimension.
The polynomial will go through the following steps:

- **Twisting**

- **Taylor expansion**
- **Gammas and Deltas computation**

After the recursive calls, the last step is the calculation of ω_i and ω_{k+i}.

The model investigation adopted a recursive approach initially, later transitioning to an iterative implementation. To facilitate comprehension, the recursive algorithm will be comprehensively detailed first, succeeded by an exposition of the iterative algorithm.

4.2.1 Recursive Additive Fast Fourier Transform

The Recursive Additive Fast Fourier Transform follows the structure given by the AFFT realized in HQC. This version is composed of two functions: the wrapper and the recursive function. The wrapper realizes the first level of the Additive FFT. Then, it calls the recursive function, which deals with the other levels. The wrapper function is used to compute some values, before the recursive calls. It computes mainly the starter beta basis.

Beta Basis Computation

The function **compute_fft_betas(uint*betas, pqc_algorithm algorithm)**, defined in the file **fft_rec.c** of the software model, computes the beta basis required as the initial point for the Additive Fast Fourier Transform.

```c
static void compute_fft_betas(uint16_t *betas, pqc_algorithm algorithm)
{
    size_t i;
    uint16_t beta = primitive_polynomial(algorithm);
    uint16_t m = parameter_m(algorithm);
    for (i = 0; i < m - 1; ++i)
    {
        if (beta == 2)
            betas[i] = 1 << (m - 1 - i);
        else
            betas[i] = gf_exp_general(beta,i+1,algorithm);
    }
}
```

Code 4.4: Beta Basis Computation

The function (Code 4.4) takes as input the pointer of the array that will store the basis and the algorithm used. The basis is computed starting from the primitive polynomial relative to the field used. To generate linearly independent values, for each iteration, the primitive polynomial is elevated. Defined α as the primitive polynomial, the basis is described as $B = \langle \alpha, \alpha^2, \ldots, \alpha^{m-1} \rangle$.

This operation is implemented in two ways:

- when the primitive polynomial is 2, the computation resorts to shifting.

- when the primitive polynomial is not 2, the computation resorts to exponentiation

This is done to optimize the case where the primitive polynomial is 2, without resorting to complex operations such as the exponentiation.

Twisting

The twisting of the polynomial is an operation that takes a polynomial $f(x)$ of length n and performs the following computation:

$$f(\beta_m x) = \sum_{j=0}^{n-1} a_j \beta_m^j$$

Code 4.5, defined in the file **fft_rec.c** of the software model, iterates over the coefficient of the polynomial $f(x)$, multiplying each term by beta_m_pow, which contains the power β_m^j.

```c
static void fft_rec_v2(uint16_t *w, uint16_t *f, size_t f_coeffs,
    uint8_t m, uint32_t m_f, const uint16_t *betas,pqc_algorithm
    algorithm) {
     .
     .
     .
     if (betas[m - 1] != 1) {
          beta_m_pow = 1;
          x = 1;
          x <<= m_f;
          for (i = 1; i < x; ++i) {
               beta_m_pow = gf_mul_general(beta_m_pow, betas[m - 1],
    algorithm);
               f[i] = gf_mul_general(beta_m_pow, f[i],algorithm);
          }
     }
     .
     .
     .
}
```

Code 4.5: Twisting of the polynomial

Gammas and Deltas Computation

Gammas and deltas are computed using Equation 3.18 and Equation 3.19. In Code 4.6, defined in the file **fft_rec.c** of the software model, the computation is done iterating on the beta basis, and performing inversion and multiplication.

γ_i is computed by multiplying β_i with β_m^{-1}, the results generated are then used to compute the deltas, which will serve as the new beta basis in the next recursion call, while gamma is used in the computation of the gammas sums.

```c
static void fft_rec_v2(uint16_t *w, uint16_t *f, size_t f_coeffs,
    uint8_t m, uint32_t m_f, const uint16_t *betas,pqc_algorithm
    algorithm) {
    .
    .
    .
    for (i = 0; i + 1 < m; ++i) {
        gammas[i] = gf_mul_general(betas[i], gf_inverse_general(betas[m
    - 1],algorithm),algorithm);
        deltas[i] = gf_square_general(gammas[i],algorithm) ^ gammas[i];
    }
    .
    .
    .
}
```

Code 4.6: gf_square_12

Gammas sums

Gammas sums are the values derived from the computation of the subset of the gammas, these are the values effectively used in the AFFT butterfly. The computation derives from Equation 3.15, to compute the $B[j]$ of a certain j, it is necessary to observe the binary representation of j. Therefore, the operation is done in the following way:

$$B[j] = \sum_{i=0}^{n-1} a_i \beta^i \tag{4.1}$$

where a_i is the bit in position i , of the binary representation of j. Code 4.7, defined inside **fft_rec.c** of the software model, itself optimizes the operation by reusing during the computation the values previously calculated.

```c
static void compute_subset_sums(uint16_t *subset_sums, const uint16_t *
    set, uint16_t set_size) {
    uint16_t i, j;
    subset_sums[0] = 0;

    for (i = 0; i < set_size; ++i) {
        for (j = 0; j < (1 << i); ++j) {
            subset_sums[(1 << i) + j] = set[i] ^ subset_sums[j];
        }
    }
}
```

Code 4.7: Computation of the subset of gammas

Taylor Expansion

The Taylor expansion is a directly iterative implementation of the Algorithm 7.

```c
static void iterative_optimized_taylor_expand(uint16_t* f0,uint16_t *f1,
    uint16_t* f,int n,int m){
    int level, node, step_curr, step_next;

    for (level = m; level >= 2; level--)
    {
        step_curr = 1 << level;
        step_next = 1 << (level - 1);
        int elements = 1<<level;
        int k = elements/4;
        for (node = 0; node < n; node = node + step_curr)
        {
            uint16_t* g0 = &f[node];
            uint16_t* g1 = &f[node+step_next];
            for(int i=0;i<k;i++){
                g1[i] = g1[i] ^ g1[i + k];
                g0[i + k] = g0[i + k] ^ g1[i];
            }
        }
    }

    for(int i=0;i<n;i+=2){
      f0[i / 2] = f[i];
      f1[i / 2 ] = f[i + 1];
    }

}
}
```

Code 4.8: Taylor Expansion of $f(x)$

This recursive algorithm could be identified as a tail-recursive function, where the recursive calls are executed at the end of the function. Transforming this recursive algorithm into an iterative algorithm is straightforward. Code 4.8, defined in the file **fft_iterative.c** of the software model, is implemented as two nested loops: the first loop emulates the recursive calls and the second loop emulates the node's processing. In the inner loop the computation of $g_1(x)$ and $g_0(x)$ is performed, when each node is processed the outer loop goes to the next level. The termination of the recursive function translates into restricting the outer loop between m and 2, as the recursive calls terminate when the length of the sub-problem is equal to 2 (level=1).
At the end, the algorithm generates the Taylor expansion of the polynomial $h(x)$. The last four lines, extract $g_0(x)$ and $g_1(x)$. The even indexed values go to the array $g_0(x)$ and the odd indexed value go to the array $g_1(x)$.

Butterfly

The butterfly function in Code 4.9, defined in the file **fft_rec.c** of the software model, takes as inputs the gammas sums and values returned by the recursive calls. It computes the following operation:

$$w_i = u_i + G[i] \cdot v_i$$

$$w_{k+i} = w_i + v_i.$$

For $0 \leq i < 2^{m-1}$. The software implementation performs this resorting to a for loop.

```c
 static void fft_rec_v2(uint16_t *w, uint16_t *f, size_t f_coeffs,
uint8_t m, uint32_t m_f, const uint16_t *betas,pqc_algorithm
algorithm) {
 .
 .
 .
 k = 1;
 k <<= ((m - 1)); // &0xf is to let the compiler know that m-1 is
small.

 for (i = 0; i < k; ++i) {
     w[i] = u[i] ^ gf_mul_general(gammas_sums[i], v[i],algorithm);
     w[k+i] = w[i] ^ v[i];
 }
 .
 .
 .
 }
```

Code 4.9: Butterfly

4.2.2 Recursive Inverse Additive Fast Fourier Transform

The inverse of the AFFT is constructed by executing the inverse of each operation of the algorithm in reverse order. This implies:

1. compute gammas and deltas
2. compute Gammas Sums
3. compute the inverse butterfly
4. Execute the recursive calls
5. compute the inverse Taylor expansion of the polynomial
6. compute the inverse twisting of the polynomial

Not all operations are executed in reverse order. The computation of gammas, deltas, and gammas sums needs to be done first to compute the inverse butterfly. The other operations are developed to perform the inverse operation.

The inverse butterfly will perform:

$$v_i = w_i + w_{i+k}.$$

$$u_i = u_i + G[i] \cdot v_i$$

for $0 \leq i < 2^{m-1}$.
After the recursive calls, the algorithm executes the inverse Taylor expansion. The inverse Taylor expansion, starting from two polynomials half the dimension $g_0(x)$ and $g_1(x)$ reconstructs the polynomial $g(x)$. While the direct algorithm is a tail recursive algorithm, the inverse algorithm presents the recursive calls as the first operations. In the iterative algorithm, this translates, into starting the outer loop not from m but from 2, to emulate the bottom-up approach of the recursive calls tree. The last step is the inverse twisting of the polynomial

$$f(x) = g(\beta_m^{-1}x)$$

The operation done is to multiply each coefficient with the inverse power β_m^{-i}.

4.2.3 Iterative Additive Fast Fourier Transform

When transitioning from a software model to a hardware implementation, the recursive nature of algorithms, inherent to computer architecture, often hinders parallel execution. In response to this, the Additive Fast Fourier Transform was transformed into an iterative structure. This process involved a thorough analysis of the recursive structure, revealing that the Additive Fast Fourier Transform does not conform to a strictly tail-recursive or head-recursive algorithm due to its unique characteristics. Consequently, no straightforward method exists. Among the potential techniques, two approaches emerged as viable solutions:

- Stack simulation.
- Custom approach.

In the stack simulation, an iterative approach is employed to emulate calls and returns, structured around the definition of a stack data structure. As recursive calls occur, the simulated stack facilitates the pushing of new stack frames with corresponding arguments. However, this approach lacks the concurrency essential for hardware development, prompting the study to explore a custom approach.

Upon analyzing the algorithm, two distinct portions emerge, which are the functions executed before the recursive calls and those executed after. This allows for a logical division of the function into the following segments:

- Before the recursive calls: twisting and Taylor expansion.
- After the recursive calls: butterfly.

Upon observing the code execution, it becomes apparent that functions before the recursive calls are executed from top to bottom, whereas those after are executed from bottom to top. Recognizing that the function involves two recursive calls, the recursive call tree can be discerned in Figure 4.1.

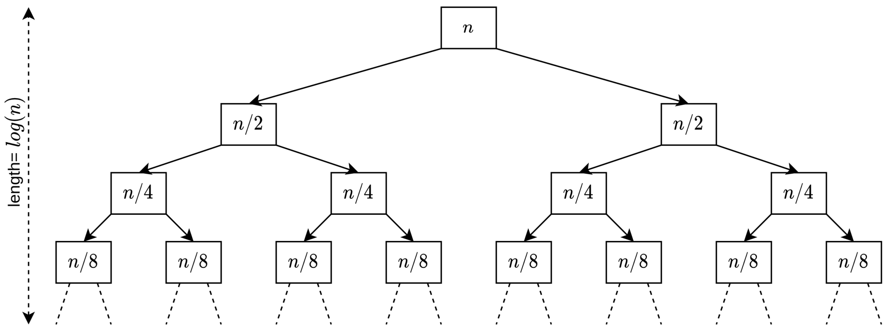


Figure 4.1: recursive calls tree

The tree will have depth $depth = log(n)$ and each level has $2^{depth-level}$ nodes. To use a parallel approach the nodes are traversed in Breadth First. This implies elaborating all the nodes before going to the next level.

The traversal of a tree is implemented using a nested loop, the outer loop iterates over the levels, and the inner loop iterates on the nodes of a level. The functions before the recursive calls, traverse the tree from the top to the bottom, therefore the outer loop will start from $log(n)$ to 2. The termination, is the processing of the nodes, at level 1, and this is managed by a single loop. Starting from the termination, the function after the recursive calls are executed. The tree, in this case, is traversed in reverse, therefore the outer loop starts from 2 to $log(n)$.

Some of the functions such as those that compute the beta basis, gammas, and deltas are computed at the beginning. The function computing Gammas sums has been moved after the recursive calls because its values are only used by the butterfly. The final algorithm has been organized into four main blocks.

Initialization

The first block computes the beta basis, gammas, and deltas for each level. This choice implies to store $log(n)$ arrays of the maximum dimension of $log(n)$.

Depth first traversal

This part interests those functions that are executed before the recursive calls. The execution is done by processing all nodes of a level, before going to the next level.

Termination

This block manages the termination. It executes the instruction present in it.

Inverse depth first traversal

This segment specifically pertains to functions executed after the recursive calls, where the execution unfolds from the leaves of the tree upward, culminating at the top.

4.2.4 Inverse Iterative Additive Fast Fourier Transform

The construction of the Inverse Iterative Additive Fast Fourier Transform mirrors the structure of its iterative counterpart, with the key distinction lying in the reversed order of execution for inverse functions. The segment preceding the recursive calls executes the inverse butterfly and computes the sums of gammas. Subsequently, the portion following the recursive calls handles the computation of the inverse Taylor expansion and the inverse twisting.
In the code, the Depth First Traversal processes each node using the inverse butterfly. Simultaneously, the Inverse Depth First Traversal of the tree applies, at each node, the inverse Taylor expansion, followed by the inverse twisting.

4.3 Polynomial Multiplication Structure

The polynomial multiplier is organized into five parts:

- pre-processing blocks
- evaluation
- pointwise multiplication
- interpolation
- post-processing blocks

From the software model point of view, the polynomial multiplication is performed in different functions for McEliece and HQC. This section presents the main pre-processing blocks and post-processing.

4.3.1 Pre Processing

Pre-processing is an operation performed on the data before going into the evaluation-pointwise multiplication-interpolation steps. The pre-processing is done only on the inputs provided by HQC.

HQC during the encryption and decryption of data performs multiplication on polynomial where the coefficients are in $\mathbb{F}_2$. Due to the nature of the field, polynomials cannot be evaluated and interpolated in their original field $\mathbb{F}_2$. This is due to the requirements of the additive fast Fourier transform.

The AFFT requires that the field presents enough evaluation points to evaluate the polynomial. In the field $\mathbb{F}_2$, the number of points is less than the length of the polynomial, and therefore the condition $deg(f(x)) < 2^m$ is not satisfied. To solve that, what is done is to map a polynomial in $\mathbb{F}_2$ to a field $\mathbb{F}_{2^m}$ [7], where there are enough evaluation points. This operation is performed through the Kronecker segmentation [6]. The operation is performed as follows:

1. Partition the polynomial to w-bits blocks.

$$\begin{gathered} a(x) = a_0 + a_1 x + \cdots + a_{d-1} x^{d-1} \\ \to (a_0 + \cdots + a_{w-1} x^{w-1}) + (a_w + \cdots + a_{2w-1} x^{w-1}) x^w + \cdots + (\cdots) x^{w(n-1)} \end{gathered} .$$

2. Define a field $\mathbb{F}_{2^m}$.

3. Map each block to the field $\mathbb{F}_{2^m}$:

$$a'(y) \quad := \quad a'_0 + a'_1 y + \cdots + a'_{n-1} y^{n-1} \in \mathbb{F}_{2^{2w}}[y],$$

such that $a'_0 = (a_0 + a_1 z + \ldots + a_{w-1} z^{w-1}), a'_1 = (a_w + \ldots + a_{2w-1} z^{w-1}), \ldots, a'_{n-1} = (a_{(n-1)w} + a_{(n-1)w+1} z + \ldots + a_{nw-1} z^{w-1})$.

Defined as $q = 2^m$ the order of the field, w the dimension of the block, and n the degree of the polynomial resulting from the product of $A(x)$ and $B(x)$.

The following conditions are defined:

$$w < 1 + \frac{1}{2}\log q \tag{4.2}$$

$$ceil\left(\frac{n-1}{w}\right) < q \tag{4.3}$$

These conditions determine the choice of q and w, to be able to apply the Additive Fast Fourier Transform. For HQC has been chosen to use the field $\mathbb{F}_{2^{16}}$, the value of w chosen is 8, this has been validated against the conditions, Table 4.1.

algorithm	**n**	**q**	**w(4.2)**	**condition(4.3)**
HQC-128	17669	65536	w < 9 = 8	2209 < 65536
HQC-192	35851	65536	w < 9 = 8	4482 < 65536
HQC-256	57637	65536	w < 9 = 8	7205 < 65536

Table 4.1: choice of q and w

Code 4.3.1 of the Kronecker segmentation, implemented in the file **kronecker_segmentation.c** in the software model, is done through a for loop that uses a mask to select 8 bits and position them in an array cell.

```c
    void kronecker_segmentation(const uint64_t *binary_polynomial,
   size_t bin_length, uint16_t *polynomial)
{
    size_t index = 0;
    for (size_t i = 0; i < bin_length; i++)
    {
        uint64_t currentValue = binary_polynomial[i];
        for (int j = 7; j >= 0; j--)
        {
            polynomial[index++] = (uint16_t)((currentValue >> (j * 8)) &
     0xFF);
        }
    }
}
```

The polynomial generated after the Kronecker segmentation will have the coefficients in $\mathbb{F}_{2^{16}}$, and it will be then evaluated with the Additive Fast Fourier Transform. After the interpolation and the pointwise multiplication, the polynomial will be converted back to the original field $\mathbb{F}_2$. In this case, the operation that will be performed is the inverse Kronecker segmentation. In the next subsection, we will discuss the post-processing blocks, which will include the inverse Kronecker segmentation and the reduction functions for HQC and McEliece.

4.3.2 Post Processing

The Post-processing for McEliece and HQC is done separately. In McEliece, it implies only the reduction of the polynomial, while in HQC it implies first the inverse Kronecker segmentation and then the reduction.

HQC Post Processing

The first step is the inverse Kronecker segmentation. Considering two binary polynomials

$$a(x) = a_0{+}a_1x{+}\cdots{+}a_{d-1}x^{d-1} \in \mathbb{F}_2[x]$$

$$b(x) = b_0 + \cdots + b_{d-1}x^{d-1} \in \mathbb{F}_2[x]$$

They can be mapped to:

$$\begin{aligned} a'(y) &:= a'_0 + a'_1y + \cdots + a'_{n-1}y^{n-1} \in \mathbb{F}_{2^{2w}}[y], \\ b'(y) &:= b'_0 + b'_1y + \cdots + b'_{n-1}y^{n-1} \in \mathbb{F}_{2^{2w}}[y], \end{aligned}$$

After evaluating $a'(y)$ and $b'(y)$, pointwise multiplication is performed by multiplying the coefficients of the transformed function. The resulting product $c'(y)$ will present coefficients that have a maximum length of $2w-1$, to map back the polynomial to the original field. We need to compute the binary polynomial adding together at most two coefficients at any power.

$$c'(y) = c'_0 + c'_1y + \cdots + c'_{2(n-1)}y^{2(n-1)} \in \mathbb{F}_{2^m}[y]$$

$$c(x) = (c_0 + \ldots + c_{(2w-1)}x^{w-1}) + (c_{2w} + \ldots + c_{(4w-1)}x^{w-1})x^w + \ldots + (\ldots) \tag{4.4}$$

$$c(x) = c_0 + c_1x + \cdots + c_{2(d-1)}x^{2(d-1)} \in \mathbb{F}_2[x]$$

This operation in code is computed with XORing and shifting, the coefficients are shifted, of as many positions as the exponent, and then it is added to the partial result. After converting the polynomial the to original field $\mathbb{F}_2$, the polynomial is reduced with the polynomial $x^n - 1$, where n is the binary polynomial length defined by HQC. This is done to reduce the polynomial to the dimension of the inputs of polynomial multiplication.

McEliece Post Processing

The post-processing in McEliece interests only a reduction of the dimension of the polynomials. This is done through a reduction function $F[z]$, which is different for every level of security. The operations consist of the computation of the remainder of the product polynomial $C(x)$.

4.4 Testing And Integration

The polynomial multiplier, in conclusion, presents two different functions dealing with polynomial multiplication for McEliece and HQC. These functions have been integrated and successfully tested in the PQClean collection [9].
The execution time concerning the method used by PQClean has been compared, and the results are reported in Table 4.2.

Algorithm	PQClean + Additive FFT Multiplier	PQClean
hqc-128	16.470s	0.099s
hqc-192	21.967s	0.360s
hqc-256	30.669s	0.432s
mceliece348864	3.981s	1.553s
mceliece348864f	1.950s	1.191s
mceliece460896	25.781s	3.437s
mceliece460896f	8.498s	2.575s
mceliece6688128	57.289s	9.529s
mceliece6688128f	12.575	4.569s
mceliece690119	1m9.299s	10.939s
mceliece690119f	11.967s	4.512s
mceliece8192128	34.011s	8.361s
mceliece8192128f	13.527s	5.407s

Table 4.2: HQC and Mceliece execution time comparison

The observed results exhibit less-than-expected performance, and the root cause can be traced back to the Additive Fast Fourier Transform (AFFT). During polynomial evaluation, the AFFT assesses the polynomials across all $B[i]$ elements of the fields, leading to a phenomenon of over-evaluation. Specifically, the polynomial is evaluated on more points than necessary.

Taking McEliece348864 as an example, where the polynomial length is 64, the resultant product spans 128 coefficients. In a conventional Fast Fourier Transform (FFT), this would entail 127 evaluations at points. However, due to the nature of the AFFT, the polynomial undergoes evaluation at 4096 points, significantly exceeding the required number.

The adoption of the AFFT in the new software versions of HQC and McEliece introduces a noteworthy consideration for the observed decrease in performance. This deviation from classical methods results in a slower execution due to the inherent over-evaluation characteristics of the AFFT, necessitating a reevaluation of the chosen transform methodology in the context of these specific cryptographic algorithms.

Chapter 5

Hardware Implementation

This chapter presents the hardware architecture of the polynomial multiplier. The hardware design follows a memory-based approach, that relies on the use of memory elements to store intermediate results. This allows to achieve a system that is more flexible and uses fewer hardware resources. The architecture is shown in Figure 5.1.

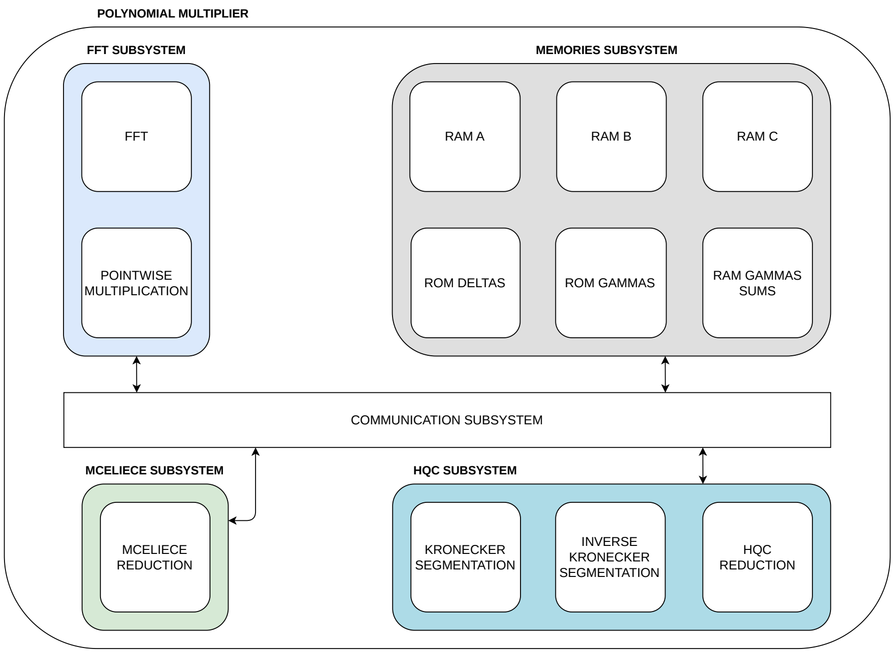


Figure 5.1: Accelerator architecture

It is organized mainly into five subsystems:

- The **FFT subsystem** that performs the direct and inverse Fast Fourier Transform, and the pointwise multiplication of the two resulting transformed polynomials.
- The **memories subsystem** is organised in six memories, ram a, b and c that store

the polynomials. rom deltas and gammas that store values required by the FFT, and a memory that store the gammas sums computed.

- The **McEliece subsystem**, contains the polynomial reduction for all levels of security.
- The **HQC subsystem**, contains the post and pre processing blocks specific to HQC, and the final reduction.
- The **communication subsystem**, is a layer of logic, that manages the access to the memories.

The architecture has been designed following a top-down approach: each block has been organized in different submodules, in order to simplify the testing and validation of the system. The development of the accelerator relies on SystemVerilog for the design, and for the testing, it relies both on SystemVerilog and Python. Before delving into the implementation of the different subsystems, it will be first presented the arithmetic logic and the memories used for the polynomials. This choice is motivated by the fact, that all the subsystems evolve around those components. The section will then be concluded by the presentation of the functioning of the polynomial multiplier.

5.1 Field arithmetic logic

The arithmetic logic deals with the development of the computational units dedicated to the fields $\mathbb{F}_{2^{12}}$,$\mathbb{F}_{2^{13}}$, and $\mathbb{F}_{2^{16}}$. The operations required for the functioning of the system are:

- Addition
- Multiplication
- Squaring
- Inversion

Each operation is organized as shown in Figure 5.2, Figure 5.3, Figure 5.3. For each operation architecture, the arithmetic logic for each field is defined. The correct selection of the field operation is performed through the `algorithm` signal, which selects the appropriate output. For the addition operation, it is not provided an architecture, instead, it relies directly on the behavioral XOR operation.

The next subsection is going to provide the implementation of the field arithmetic logic of three required operations(multiplication, squaring, and inversion).

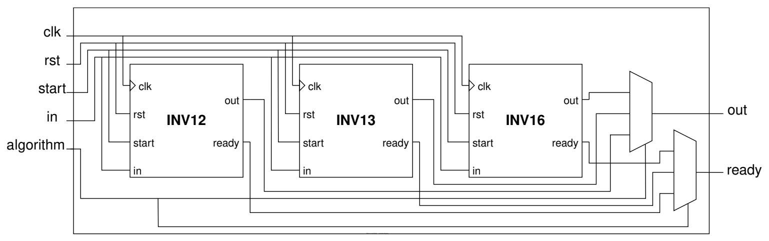


Figure 5.2: Inversion architecture

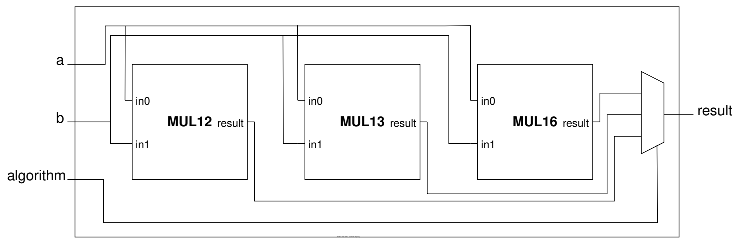


Figure 5.3: Multiplication architecture

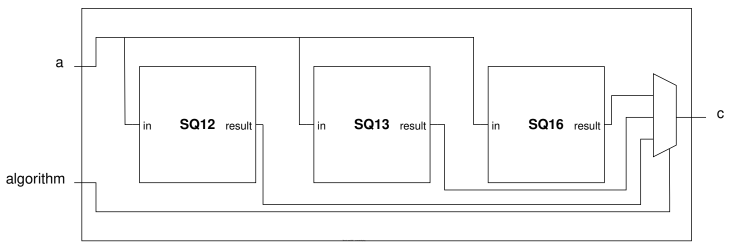


Figure 5.4: Squaring architecture

5.1.1 Multiplication

The multiplication in hardware is realized as a fully combinational circuit. This operation is carried out in two steps, multiplication and field reduction. The multiplication as shown in Figure 5.5, is implemented following the classic structure of the normal multiplication. The difference lies in the fact that instead of using carry adders, xor gates are used to sum the values. The code computes in parallel all the product bits, by XORing and ANDing the correct bits. The field reduction is implemented through masking and shifting, following Algorithm 1. The same approach is used for the fields $\mathbb{F}_{2^{12}}$,$\mathbb{F}_{2^{13}}$, and $\mathbb{F}_{2^{16}}$.

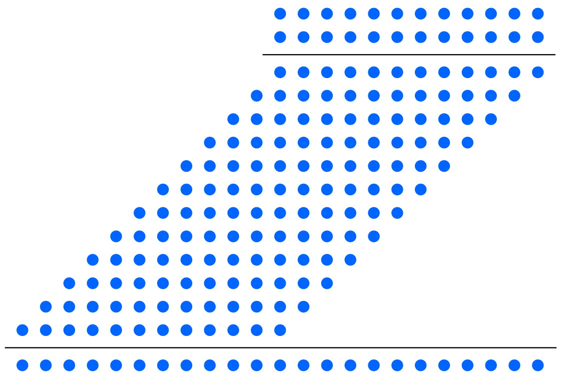
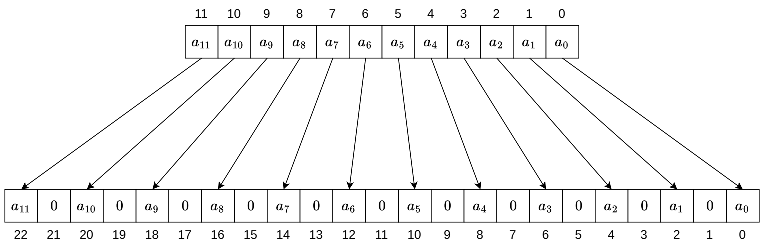


Figure 5.5: Multiplication in $\mathbb{F}_{2^{12}}$

5.1.2 Squaring

Squaring in hardware, concerning multiplication, does not imply any calculation (except reduction), it is implemented through bit manipulation as shown in Figure 5.6. The bits of the input are positioned in the even-indexed positions, while the odd-indexed bits are set to zero. The result is then reduced to the correct dimension, according to the field.



11 10 9 8 7 6 5 4 3 2 1 0
a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
a_{11} 0 a_{10} 0 a_9 0 a_8 0 a_7 0 a_6 0 a_5 0 a_4 0 a_3 0 a_2 0 a_1 0 a_0
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Figure 5.6: Squaring in $\mathbb{F}_{2^{12}}$

5.1.3 Inversion

The inversion in hardware is implemented resorting to squaring and multiplication. Following the addition chain realized in the software model, an FSM employing one multiplier and one squaring has been created. As shown in Figure 5.7, the FSM, in the case of the field $\mathbb{F}_{2^{12}}$ goes through 18 states, from S0 to S15 the input follows the chain of squaring and multiplications. The three fields $\mathbb{F}_{2^{12}}$,$\mathbb{F}_{2^{13}}$, and $\mathbb{F}_{2^{16}}$, follow the same architecture,

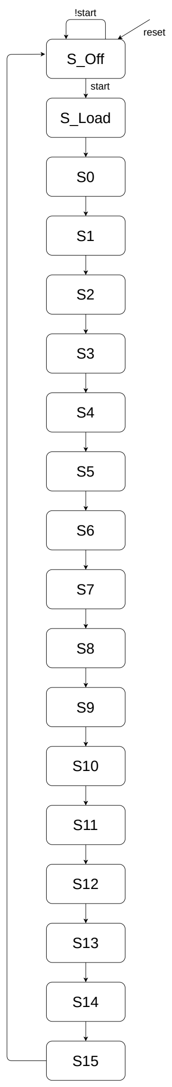


Figure 5.7: FSM inversion $\mathbb{F}_{2^{12}}$

what differs is the length of the chain and sequence of multiplications and squares.

5.2 Memories subsystem

The memories subsystem is structured with different types of memories:

- The polynomial memories are organized into 3 RAMs.
- The gammas sums memory.
- ROMs which are organized into 3 ROMs.

The polynomial memories are made by 4 dual port RAMs, that form a unique memory with 4 ports for reading and writing. This solution is adopted in order to fetch data in parallel and avoid collisions when accessing the data. As shown in Figure 5.8, the memory is organized in two sub-memories(RAM A and RAM B), where each one is made up of 2 dual port RAMs. Through the `sel` signal, when its value is 0 the RAM A outputs are selected and the enable signals are forwarded to RAM A.

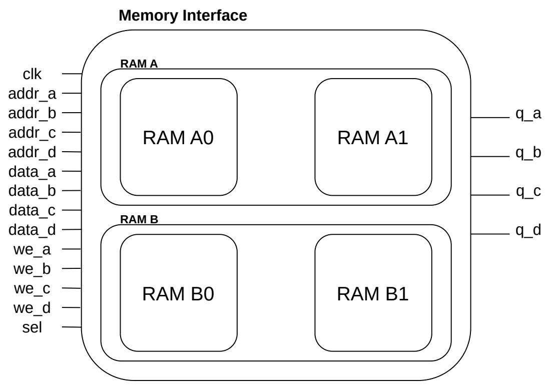


Figure 5.8: Memory interface

The ROMs, as shown in Figure 5.9, are organized in as many ROMs as the number of fields, each ROM is a single port memory. In order to access the data of specific ROM, the signals `m`,`i`, and `field` are used. In particular:

- `field` signal selects the memory relative to the field.
- `m` identifies the values belonging to a certain level of the recursion, as the values of deltas and gammas are computed for each level.
- `i` selects a particular value of delta and gamma of the level `m`.

The last memory is the gammas sums, which is a dual port RAM memory, that is used to store the computed values from the module compute subset sums.

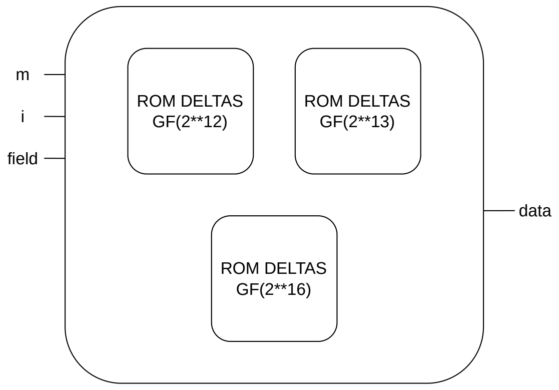


Figure 5.9: Rom interface

5.3 Fast Fourier Transform subsystem

The FFT subsystem is organized into two submodules: pointwise multiplication and FFT. Being the FFT the larger module, this will be the main focus in this section. The FFT is structured logically in two parts, as shown in Figure 5.11:

- Control
- Datapath

The control implements the iteration of the two nested for loops of the iterative implementation of the Fast Fourier Transform. The datapath is the direct mapping of the FFT C functions into hardware. Together, datapath and control, implement the direct and inverse Fast Fourier Transform. Each module performs both the direct and inverse operations. The process begins by taking in input the signals `field`, `fft_ifft` and `in_sel`. The signal `field`, selects the field in which the FFT is executed.
The signal `fft_ifft`, selects the FFT if 0 and the IFFT if 1; The signal `in_sel`, tells the FFT from which sub memory start the operation. Following that, the finite state machine, as shown in Figure 5.10, starts from the state **M__TO__TWO**. This state executes those functions that in the recursive FFT are done before the recursive calls. Vice versa in **TWO__TO__M** those functions that are after the recursive calls are executed. While the condition of termination is computed in the state **CIW**. The states denoted with suffix **__WAIT** are used to wait until the operation is terminated before going to the next one.

Before going into detail, about the different modules, it will be first presented the memory access scheme and organization of the data.

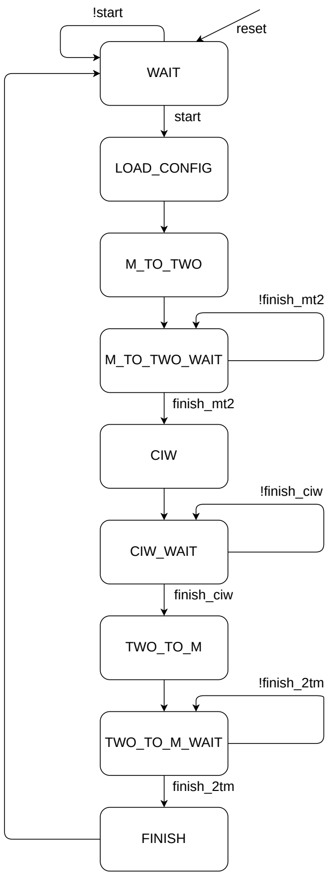


Figure 5.10: FSM Fast Fourier Transform

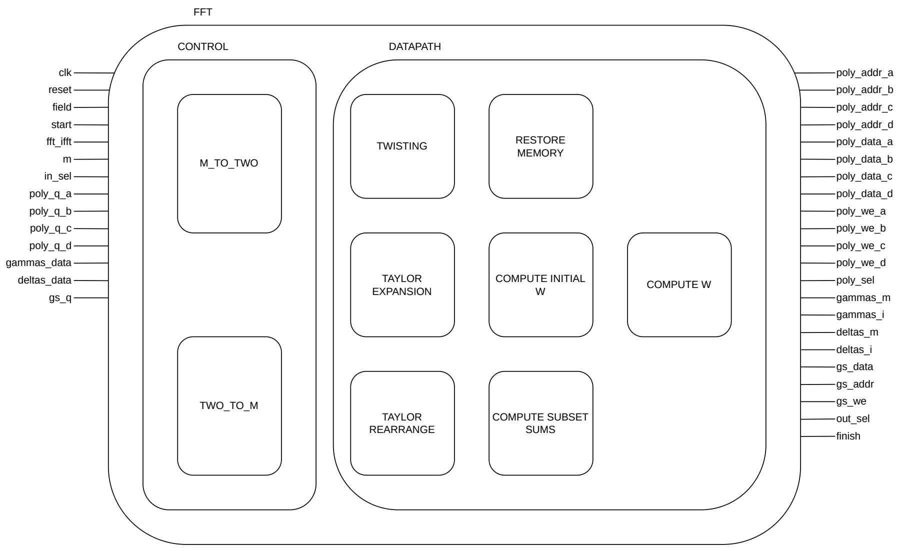


Figure 5.11: Fast Fourier Transform architecture

5.3.1 Memory access scheme

The memory access setup was designed having in mind to use single-port memories, although it also works by extension with dual-port RAMs. To explain its functionality, let's consider a simplified memory configuration compared to what's utilized in the accelerator.

Data in memory is arranged with the understanding that a processing element requires access to two memory locations simultaneously. Consequently, the memory storing the polynomials is divided into two parts. However, merely dividing the memory isn't sufficient for parallel access. To address this, a method has been developed to ensure that data requiring parallel access is distributed across two different memories, thereby enabling simultaneous retrieval.

The access scheme is designed following how the computation is done by the Taylor expansion and the butterfly. In the case of the butterfly, each time two values are read from the array, one value belongs to the first half of the array and the second value belongs to the second half of the array. The Figure 5.12, represents the execution of recursion together with the butterfly function, starting from level 4 to level 3. In the case of level 4, the data in red and blue should be in two different memories to allow parallel access. The same concept can be applied to all the other levels, therefore all values identified in red and blue are organized in two different memories.

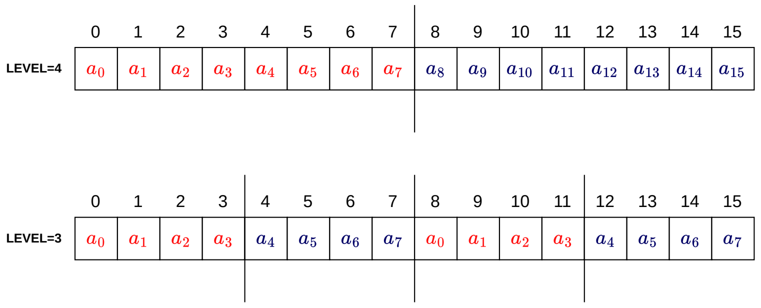

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
LEVEL=4
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
LEVEL=3
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7

Figure 5.12: Butterfly array data access

The case is different for the Taylor expansion. In the case of the butterfly, the function was simply an execution of a for loop, the Taylor expansion presented a nested loop (due to the fact that it is a recursive algorithm). Therefore what happens is on each subarray generated by the recursive calls, the Taylor expansion is executed.
In Figure 5.13, the case considers a problem of dimension 16. The figure represents, in particular, the execution at level 3, where the problem is divided into two sub-problems of dimension 8, on each array of the Taylor expansion executed. Knowing that the Taylor expansion is a recursive algorithm. The data will need to follow the evolution of the access scheme. The data arrangement will change in relationship to the recursion and the Taylor expansion. In the case of the butterfly, the access scheme evolves only with the traversal of the recursive calls of the FFT since the butterfly is only an iterative algorithm with one for loop.

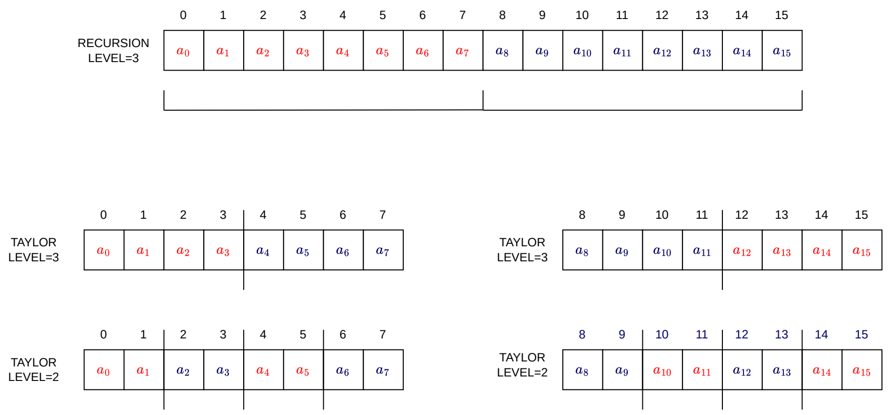
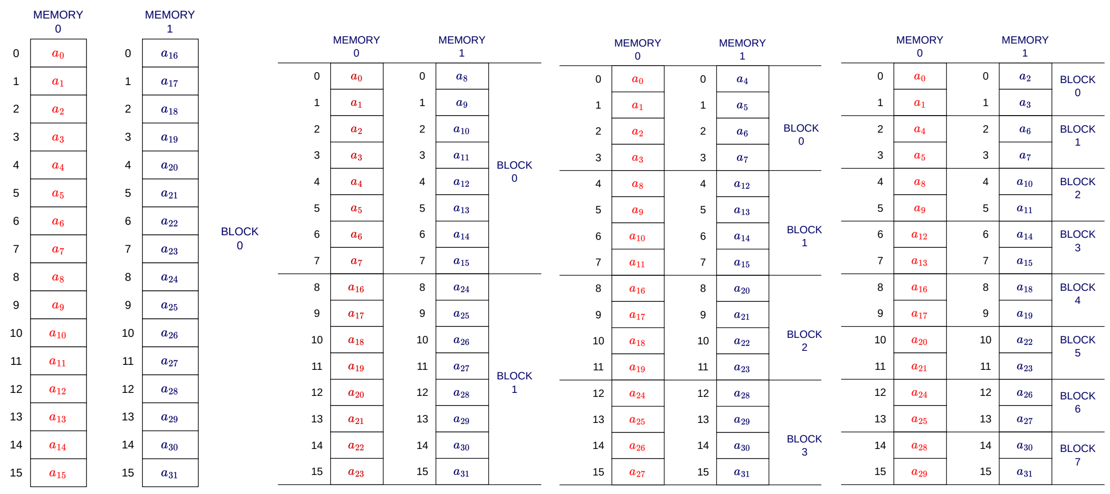


Figure 5.13: Taylor expansion data access

The Figure 5.14, illustrates the functioning access scheme across the different levels of recursion, considering an array of dimension 32. The scheme views the memories differently. In level 5 the memory is considered as a unique block, in level 4 the memory is considered as two different blocks, and so on for level 2 and level 3.



(a) Level 5 (b) Level 4 (c) Level 3 (d) Level 2

Figure 5.14: Access scheme from level 5 to level 2

The access scheme for reading and writing is implemented through the module **address_translation.sv**. This module takes as input the current level of the recursion and the address of the memory location. The effective address where the data is stored is computed through the following steps:

1. The first step is to compute in which memory block the data is stored
2. The second step is to compute in which word of the memory block the data is stored

3. The third step is to compute in which memory the data is stored
4. The final step is to compute the final address.

The address translation unit is used in all the modules that compose the datapath, in order to access the correct memory location.

5.3.2 Twisting

The twisting of the polynomial is executed by the module **twisting.sv**. The function is implemented as a finite state machine, as shown in Figure 5.15. It performs both the direct and inverse operations. The module starts by receiving the signals `fft_ifft`, `poly_len`, `start_addr`, `m`, `field` and `mem_sel`. These signals indicate: the operation to perform; the length of the input, the start address; the level of the recursion; the field used by the FFT; and the sub-memory to access. The FSM has an initial state called **LOAD_CONFIG**, where the control data in input is memorized in the system. From there it moves to the state **BETA_RETRIEVAL**, where beta is read from the **deltas rom**. In case, the system is performing the inverse operation, beta needs to be inverted. Therefore it moves to the state **BETA_INV**, where the value of beta is inverted. The iteration present in the C code is implemented through the states **FOR_INIT**, **FOR_EVAL** and **FOR_INC**, that respectively emulate the for initialization, the condition evaluation, and the increment. The states **READ** and **WRITE**, perform the reading of the data from memory and writing of the result in memory. The computation is done in the state **COMPUTE**, where the product of the beta power and the coefficients of $f(x)$ is done, also the next power of beta is computed.

5.3.3 Taylor Expansion

The Taylor expansion is realized by the module **taylor_expand.sv**, as finite state machine as shown in Figure 5.16. The module realizes both the inverse and direct operation. The system takes in input:

- `fft_ifft`: the selection between direct and inverse operation.
- `level`: indicates the depth of the recursion.
- `field`: indicates the field use in the FFT.
- `address`: the starting address of the operation.
- `len`: indicate the length of the input.
- `sel`: identifies the sub-memory from what starts the operation.

The operation starts from the state **LOAD_CONFIG**, where the input control data is saved in the system. The Taylor expansion as function is realized as a triple loop, the nested loop emulates the traversal of the tree while the inner loop is used to operate on the data. In the FSM this translated into having the following states:

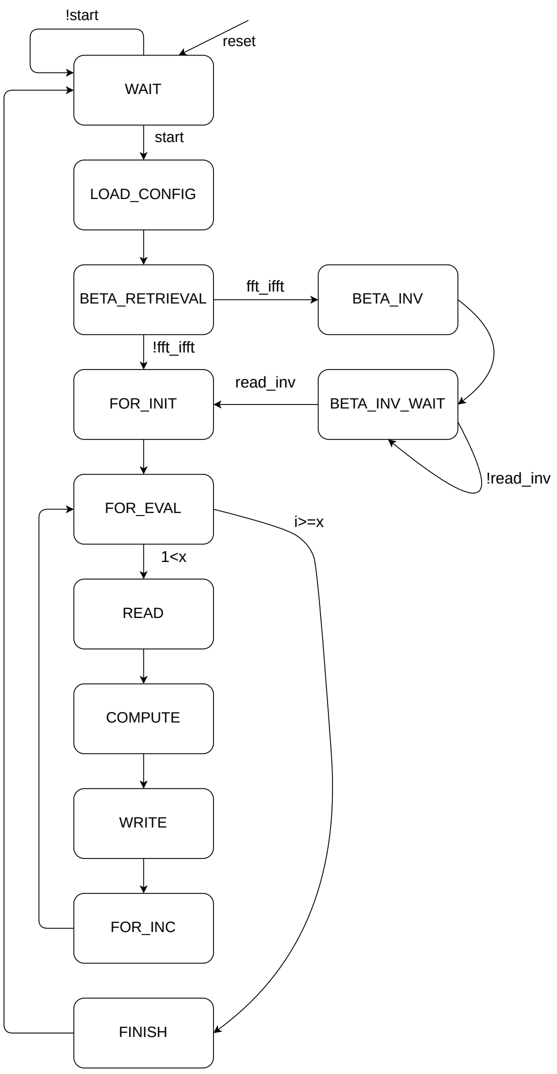


Figure 5.15: Twisting finite state machine

- **INIT_LEVEL, EVAL_LEVEL, DECREM_LEVEL**, emulate the outer loop.
- **INIT_NODE, EVAL_NODE, INC_NODE**, emulate the iteration on the node of a level.
- **INIT_ELEMENT**, **EVAL_ELEMENT**, **INC_ELEMENT**, identifies the loop where the data is computed.

In case, the inverse operation is executed, the behavior of the outer loop changes. First of all, the evaluation condition becomes `LevelTaylor <= m`, and the loop will decrement the variable `LevelTaylor` instead of incrementing. The other loops will remain unchanged in both operations.

The computation of the data is performed in the states **READ_PE**, **COMPUTE_PE** and **WRITE_PE**. the states **READ_PE** and **WRITE_PE**, will perform the reading and writing of the data. The memory address for reading and writing is computed through the address translation module. The state **COMPUTE_PE**, resorts to the submodule **taylor_pe.sv** , in order to compute the $g_1(i)$ and $g_0(i)$ from $g_1(i)$,$g_1(i+k)$ and $g_0(i+k)$. The finite state machine terminates when all the nodes of the tree till `level`=2 (`level=m` in the inverse operation) are computed.

5.3.4 Taylor rearrange

The Taylor rearrange module(**taylor_rearrange.sv**), implements the last for loop present in the Taylor expansion function. This is implemented as FSM, as shown in Figure 5.17. The for loop is identified by the states **FOR_INIT**, **FOR_EVAL**, and **FOR_INC**. The Taylor rearrange operation consists of reordering the memory, this is implemented through the states, **READ, STORE** and **WRITE**, which respectively read 2 values from memory and store them in 2 new memory locations. The new memory locations are given through **start_address + i/2 + n/2**, **start_address + i/2**, that are stored in `temp_address_n_2` and `temp_address`

5.3.5 Restore memory

The restore memory operation is an operation executed after all the nodes of a level have been elaborated by the twisting, Taylor expansion and Taylor rearrange modules. This is done due to the effect of the taylor expansion to the memory. The Taylor expansion as an operation, leaves the memory in a different organisation.
In Figure 5.14, the Taylor expansion will leave the data organized following the structure of level 2. The restore memory operation will reorganize the data to the next level of recursion. For example, if the current level of the recursion is 5, the next will be 4, in this case, the restore memory operation will reorganize the data from the structure of level 2 to the structure of level 4.

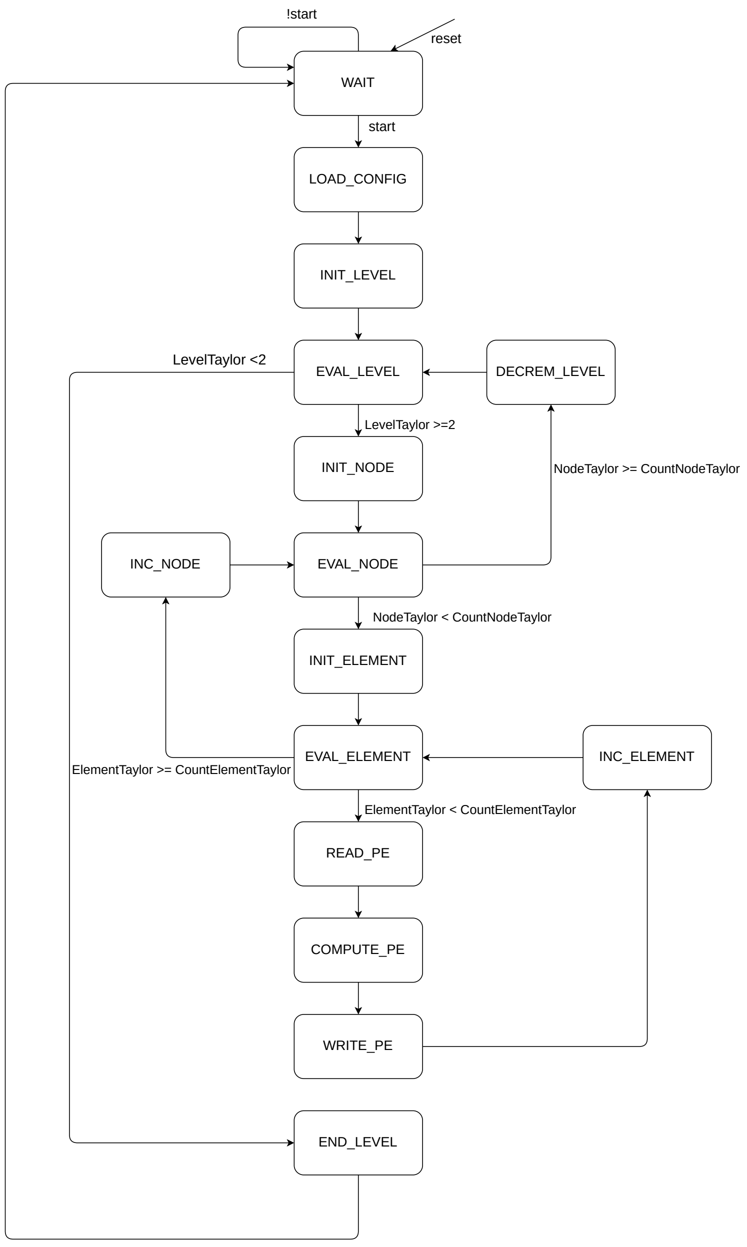


Figure 5.16: Taylor expansion finite state machine

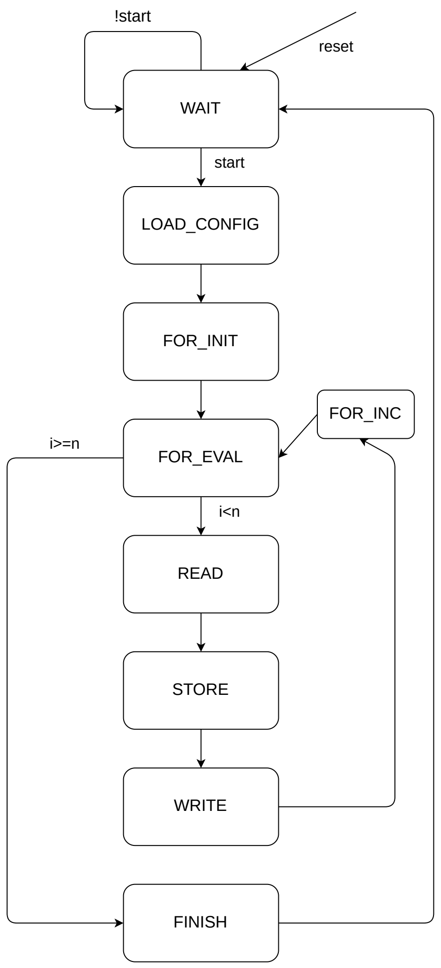


Figure 5.17: Taylor rearrange finite state machine

This action is implemented by the module **restore_memory.sv**, which implementation is based on an FSM, as shown in Figure 5.18.
The states **FOR_INIT**, **FOR_EVAL**, and **FOR_INC**, represent the for loop structure. The states **READ**, **STORE** and **WRITE**, read the data from memory and write it in a different location. The location is computed using the address translation unit, by providing the next level and current iteration of the loop, the unit will provide the new address.

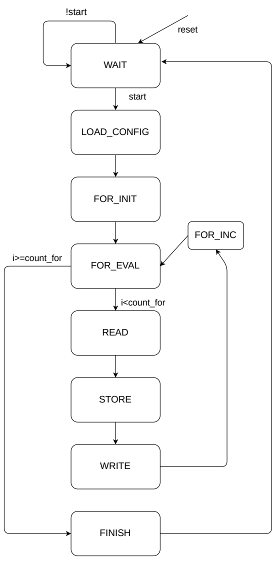


Figure 5.18: Restore memory finite state machine

5.3.6 Compute initial w

Compute initial w is identified by the module **compute_initial_w.sv**, it implements the termination of the FFT. The component is realized through an FSM, as shown in Figure 5.19. It implements a for loop through the states **FOR_INIT, FOR_EVAL** and **FOR_INC**. The module takes as input the signals:

- `fft_ifft`, selects the direct or inverse operation
- `mem_sel`, selects the sub memory
- `start_address`, selects the start address of the operation
- `poly_len`, identifies the length of the input
- `m`, identifies the level
- `field`, identifies the field of the FFT

After saving the input control data and reading the value of beta, in the state **LOAD_CONFIG**, the loop starts. At each iteration, the signal `start_address_reg`, which stores the address for reading and writing, gets incremented. In the states **READ**, **COMPUTE**, and **WRITE** the data is read, computed, and written. The system will read 4 consecutive values(`value_a`,`value_b`,`value_c`,`value_d`), and compute the operations $((\texttt{value\_b} \cdot \texttt{betas\_0\_reg}) \oplus \texttt{value\_a})$ and $((\texttt{value\_d} \cdot \texttt{betas\_0\_reg}) \oplus \texttt{value\_c})$, then result is then stored in the memory location of `value_b` and `value_d`.
The inverse operation performs $((\texttt{value\_b} \oplus \texttt{value\_a}) \cdot \texttt{betas\_0\_reg})$ and $((\texttt{value\_d} \oplus \texttt{value\_c}) \cdot \texttt{betas\_0\_reg})$, where `betas_0_reg`, is the inverse of beta.

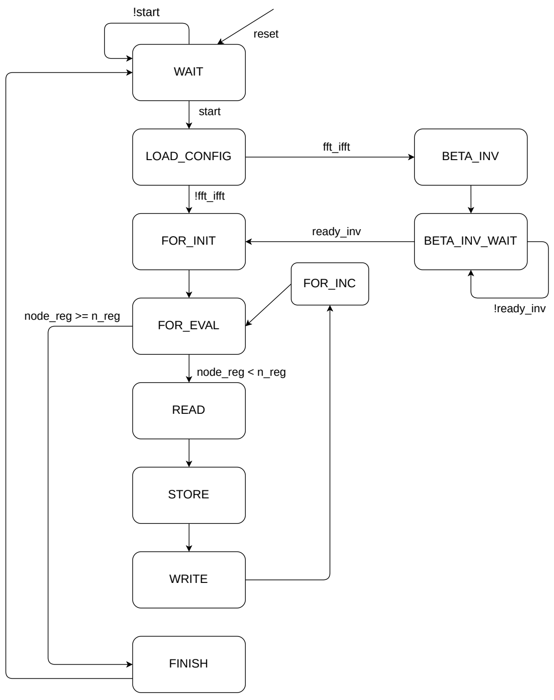


Figure 5.19: Compute initial w finite state machine

5.3.7 Compute subset sums

Compute subset sums, implemented in **compute_subset_sums.sv**, perform the computation of the gammas sums starting from the values stored in the deltas and gammas ROMs. The module as shown in Figure 5.20, is implemented as FSM. The C code implements the function as a nested loop, this is translated in the states:

- **FOR_I_INIT**, **FOR_I_EVAL**, and **FOR_I_INC**, for the outer loop.
- **FOR_J_INIT**, **FOR_J_EVAL**, and **FOR_J_INC**, for the inner loop.

The module receives as input the signal `m`, which identifies the level of the recursion. The FSM performs the following operations:

1. **LOAD__CONFIG**, saves the value of the signal `m`, in the register `m_reg`.
2. **SUBSET__SET__ZERO**, initialize the address 0 of the gammas sums memory to 0.
3. **FOR__I__INIT**, **FOR__I__EVAL**, **FOR__I__INC**, **FOR__J__INIT**, **FOR__J__EVAL**, and **FOR__J__INC** execute the nested loop.
4. **READ**, in this state the value of gamma(from gammas ROM), together with the value of gammas sums at address `j_reg` are read.
5. **STORE**, saves the value read from memory.
6. **WRITE**, performs the computation and stores the value in the memory location `j_reg`

The process ends when the nested loop is completed.

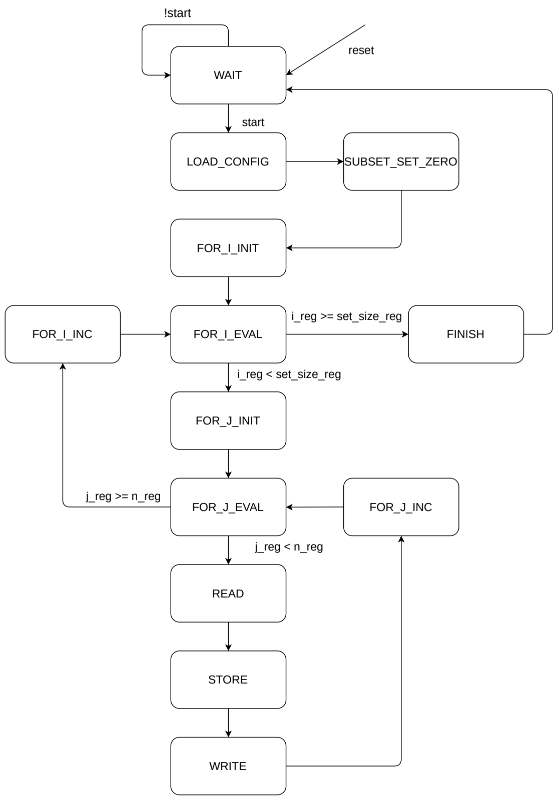


Figure 5.20: Compute subset sums finite state machine

5.3.8 Compute w

Compute w, is the unit developed to compute the butterfly of the FFT. It performs both the direct and inverse operations. The module is implemented through an FSM, as shown in Figure 5.21, that manages the for-loop iterations. The system takes as control input signals:

- `fft_ifft` selects the direct or inverse operation.
- `start_address` is the starting address of the operation.
- `m_max` is the maximum level of the FFT.
- `m` is the current level of the FFT.
- `field` is the field employed by the FFT.
- `mem_sel` identifies the sub-memory from where the operation starts.

The operation starts, after the signal `start` is raised. The system will follow this flow:

1. **LOAD_CONFIG**, stored the control input values.
2. **FOR_INIT, FOR_EVAL** and **FOR_INC**, represent the for loop.
3. **READ**, the values at address `v_address` and `u_address` are forwarded to the address translation units, and the result is then used to access the polynomial memory.
4. **STORE**, the values are stored in the registers `v` and `u`.
5. **WRITE**, the computation is performed through the module **butterfly_pe.sv**, which computes:
$$w_i = u_i + G[i] \cdot v_i$$
$$w_{k+i} = w_i + v_i.$$
   the values are then saved in the memory locations generated by the address translation unit.

In the reverse operation, the module **butterfly_pe.sv**, performs the reverse operation.

5.3.9 M to two

M to two is a control module, that manages the traversal of the tree of the recursive calls. The component performs the top-down visiting of the nodes. This is implemented through an FSM, as shown in Figure 5.22, that manages the execution of the datapath components. The system can follow two different state flow executions, in relationship if the FFT or the IFFT are executed. In the case of the FFT, the states executed are **TWISTING, TAYLOR_EXPAND, TAYLOR_REARRANGE** and **RESTORE_MEMORY**. instead in the IFFT, the states are **COMPUTE_SUBSET_SUMS** and **COMPUTE_W**. The module receives as input the following signals:

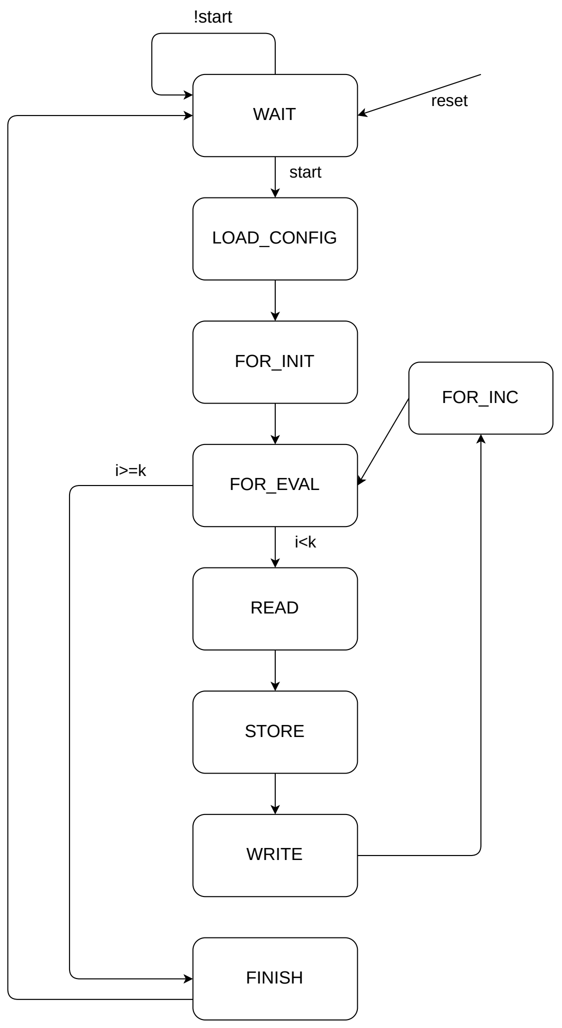


Figure 5.21: Compute w finite state machine

- `field`, identifies the field of execution of the FFT
- `m`, identified the degree of the finite field
- `fft_ifft`, indicates the operation to perform
- `in_sel`, identifies the sub memory to access

The system will behave differently when dealing with FFT or the IFFT. In the case of the FFT, each node will be processed by the twisting, taylor_expand and taylor_rearrange modules. After executing all the nodes of a level, the restore_memory module is executed, in order to prepare the memory for the next level. The processing completes when all level are processed. In the case of the IFFT, before executing compute_w, on each node of a level. The gammas sums are calculated for the level. Note that compute_w is performing the reverse operation.

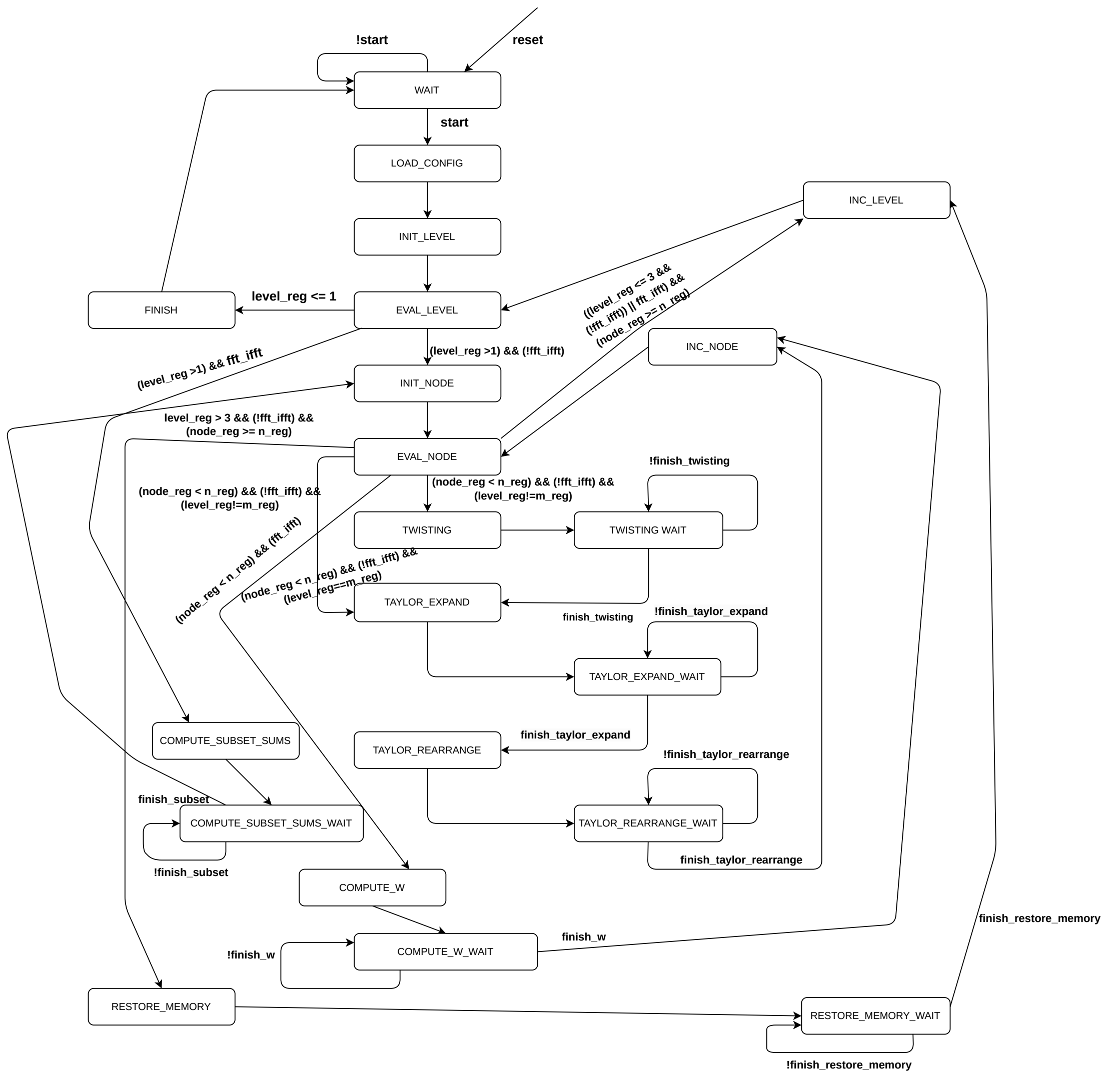


Figure 5.22: M to two finite state machine

5.3.10 Two to m

Two to m, with respect to m to two, is the module that performs the bottom-up traversal of the recursive calls tree. It is realized as FSM, as shown in Figure 5.23, that manages the execution of the datapath components. The system can follow two different state flow executions, in relationship if the FFT or the IFFT is executed. In the case of the IFFT, the states executed are **TWISTING, TAYLOR_EXPAND, TAYLOR_REARRANGE** and **RESTORE_MEMORY**. instead in the FFT, the state is **COMPUTE_SUBSET_SUMS** and **COMPUTE_W**. The module receives as input the following signals:

- `field`, identifies the field of execution of the FFT
- `m`, identified the degree of the finite field
- `fft_ifft`, indicates the operation to perform
- `in_sel`, identifies the sub memory to access

The system will behave differently when dealing with FFT or the IFFT.
In the case of the IFFT, each node will be processed by the twisting, taylor_expand and taylor_rearrange modules. Before executing all the nodes of a level, the restore_memory module is executed, in order to prepare the memory for the current level. Note that all modules are performing the reverse operation. In the case of the FFT, before executing compute_w, on each node of a level. The gammas sums are calculated for the level.

5.3.11 Pointwise mul

Pointwise mul is the module, dedicate to the pointwise multiplication of the transformed polynomials $A(x)$ and $B(x)$. This component is implemented through a FSM, as show in Figure 5.24. The system will receive in input the following values:

- `m`, indicates degree of the field
- `field`, indicates the field employed in the FFT
- `in_sel`, indicates the submemory from which start reading the values.

The component will iterate over all the values, and it will compute the field multiplication of the coefficient. The result is then saved in the RAM C.

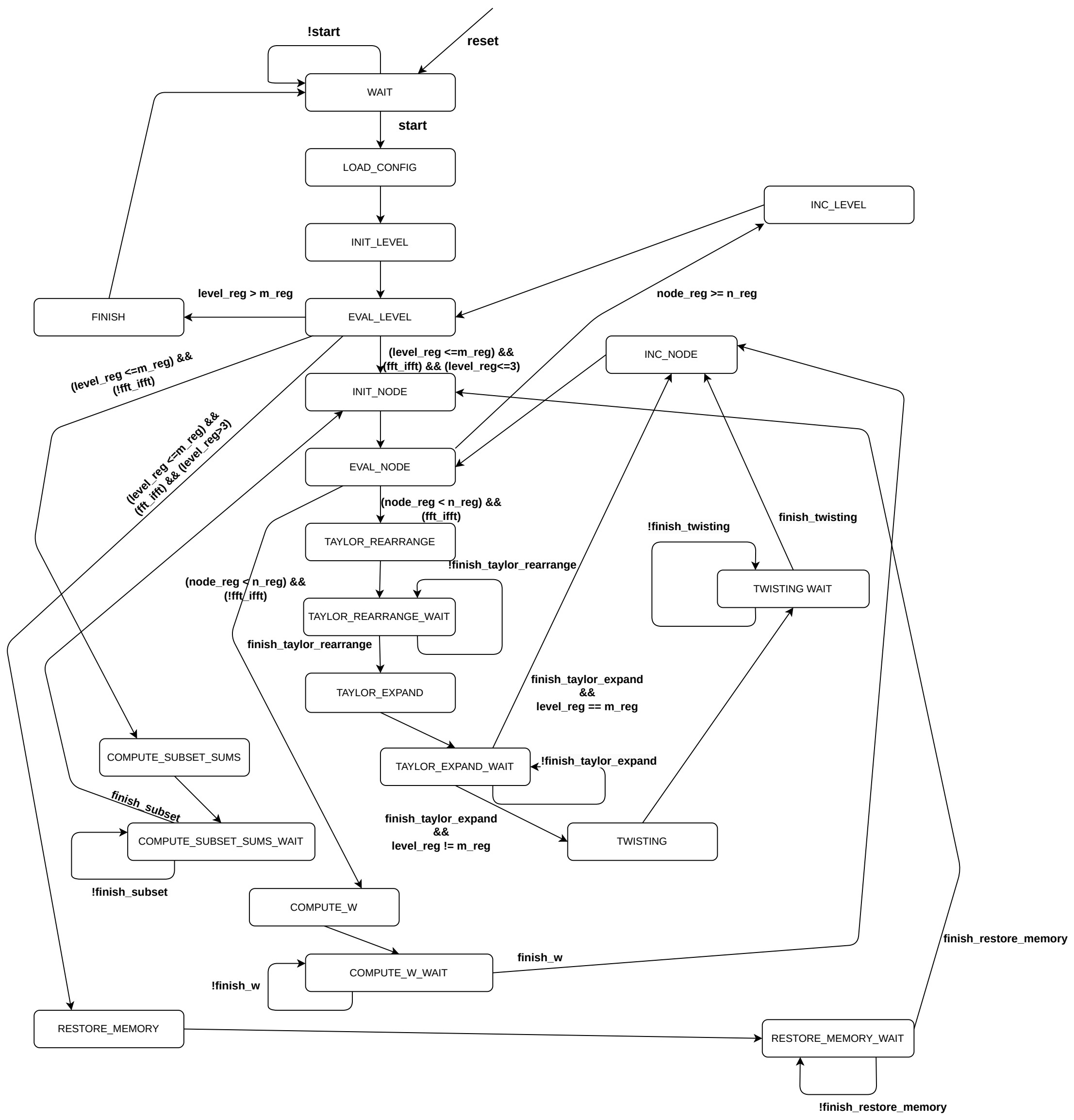


Figure 5.23: Two to m finiste state machine

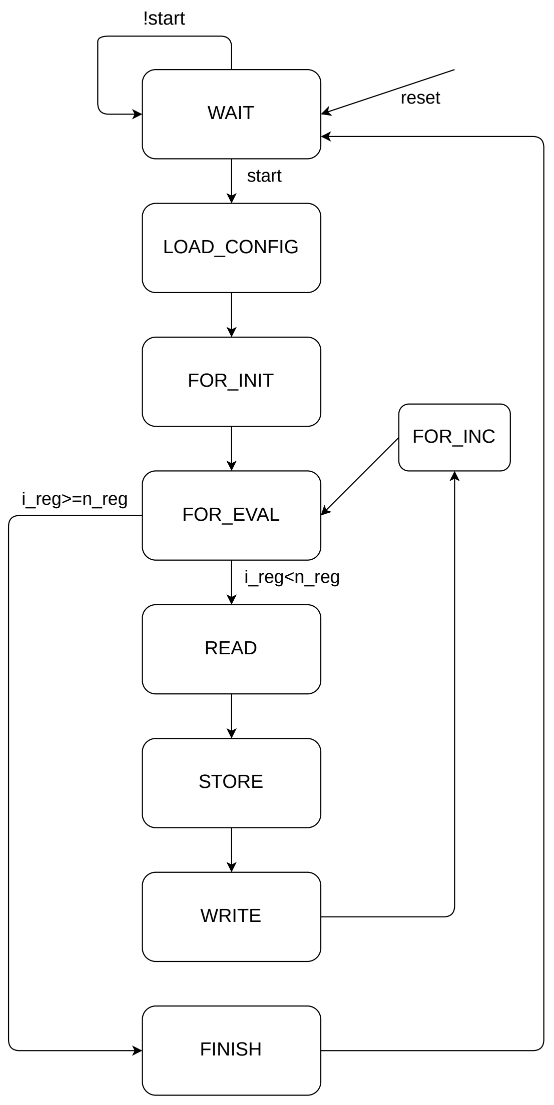


Figure 5.24: pointwise multiplication FSM

5.4 McEliece subsystem

The McEliece subsystem consists only of the McEliece reduction. This module performs the polynomial reduction of the result to the dimension **t** (refer to Table 2.1), specified by the parameter set of the algorithm.
McEliece implements 4 reduction algorithms, and due to that the FSM presents 4 flows of execution. The states of the finite state machine are organized in 3 categories: **READ**,**STORE**, and **COMPUTE**. Those states represent the reading and writing of a certain element from the memory.
Analyzing Code 5.1, Code 5.2, Code 5.3 and Code 5.4 (which are the four reduction algorithms) it is possible to observe that some operations are shared through the reduction algorithms.

```
        for (i = (sys_t - 1) * 2; i >= sys_t; i--)
        {
            prod[i - sys_t + 3] ^= prod[i];
            prod[i - sys_t + 1] ^= prod[i];
            prod[i - sys_t + 0] ^= gf_mul_general(prod[i], 2, algorithm)
    ;
        }
```

Code 5.1: Reduction 1

```
        for (i = (sys_t - 1) * 2; i >= sys_t; i--)
        {
            prod[i - sys_t + 10] ^= prod[i];
            prod[i - sys_t + 9] ^= prod[i];
            prod[i - sys_t + 6] ^= prod[i];
            prod[i - sys_t + 0] ^= prod[i];
        }
```

Code 5.2: Reduction 2

In the FSM (Figure 5.25) this translates into representing each assignment as triplet READ-STORE-WRITE, in that way, the number of states is reduced.
During execution, the FSM will visit only the states relative to the reduction algorithm. The module will take as input the value `sys_t`, which represents **t** and the reduction algorithm `red`.

```
        for (i = (sys_t - 1) * 2; i >= sys_t; i--)
        {
            prod[i - sys_t + 7] ^= prod[i];
            prod[i - sys_t + 2] ^= prod[i];
            prod[i - sys_t + 1] ^= prod[i];
            prod[i - sys_t + 0] ^= prod[i];
        }
```

Code 5.3: Reduction 3

```
        for (i = (sys_t - 1) * 2; i >= sys_t; i--)
        {
            prod[i - sys_t + 8] ^= prod[i];
            prod[i - sys_t + 0] ^= prod[i];
        }
```

Code 5.4: Reduction 4

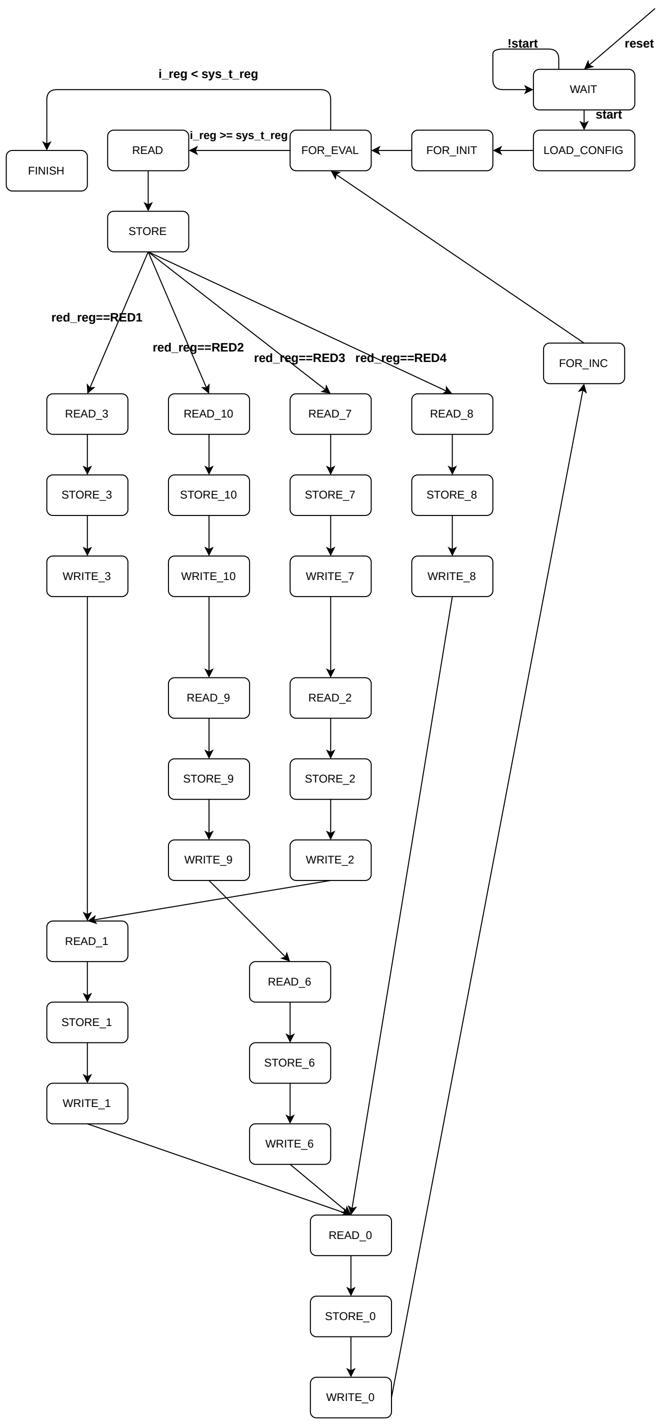


Figure 5.25: McEliece reduction finite state machine

5.5 HQC subsystem

The HQC subsystem is made by 3 blocks:

- Kronecker segmentation
- Inverse Kronecker segmentation
- HQC reduction

In this subsection, the hardware implementation of the modules will be presented

5.5.1 Kronecker segmentation

The Kronecker segmentation is the module responsible for dividing into blocks of dimension 8, a binary polynomial. The system is organized in an FSM, as shown in Figure 5.26. The component receives input signals:

- the length of binary polynomial in terms of words(`len`)
- the sub memory from which to operate(`mem_sel`)

The finite state machine follows this execution:

1. In state **LOAD_CONFIG**, the input values are saved inside registers.
2. States **FOR_INIT**,**FOR_EVAL** and **FOR_INC**, manage the for loop.
3. States **READ**,**STORE**, read and store word from the memory at address `i_reg`.
4. State **WRITE_0**,**WRITE_1**,**WRITE_2** and **WRITE_3**, each blocks of 8 bits is stored in 4 contiguous memory locations, starting from index `i_reg`.

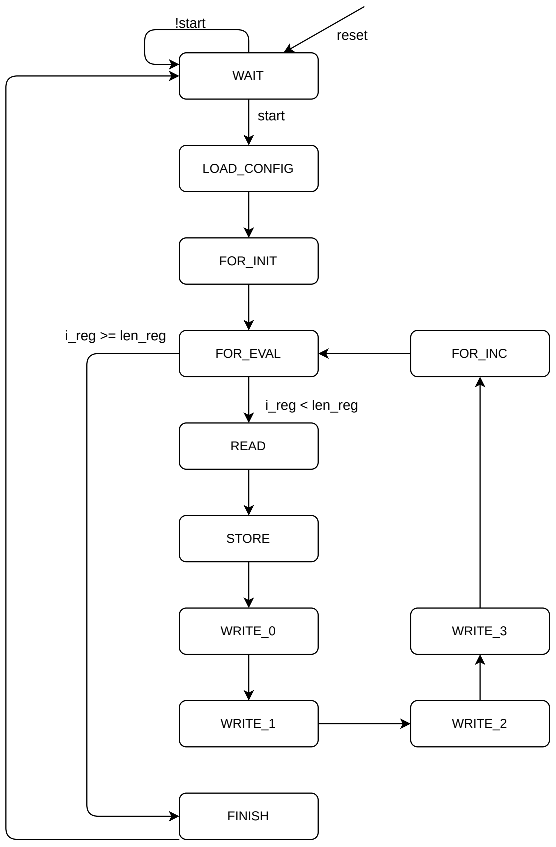


Figure 5.26: Kronecker segmentation finite state machine

5.5.2 Inverse Kronecker segmentation

The inverse Kronecker segmentation is the module responsible for converting a polynomial with coefficient in $\mathbb{F}_{2^{16}}$ in a binary polynomial. The module implements an FSM, as shown in Figure 5.28. Before the pointwise multiplication, the polynomial $A(x)$ and $B(x)$, have coefficients on 8 bits. After the pointwise multiplication, the coefficients are on 16 bits. The polynomial obtained will have this form:

$$c(x) = (a_0, a_1, a_2, \ldots, a_{15}) + (a_{16}, a_{17}, a_2, \ldots, a_{31})x^8 + \ldots$$

the inverse Kronecker segmentation has to sum those bits that share the same exponent, as shown in figure Figure 5.27.

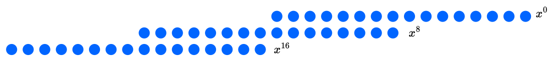


Figure 5.27: $c(x)$ polynomial dot view

This behavior is implemented by the finite state machine as follows:

- The finite state machine will read two words, knowing that they are on 16 bits, it will have to sum the upper half bits of word 1 and the lower half of word 2.
- The values then need to be stored in the correct word and the correct position in the word.
- The system fills each word sequentially, arranging the 8 bits of the result in order.

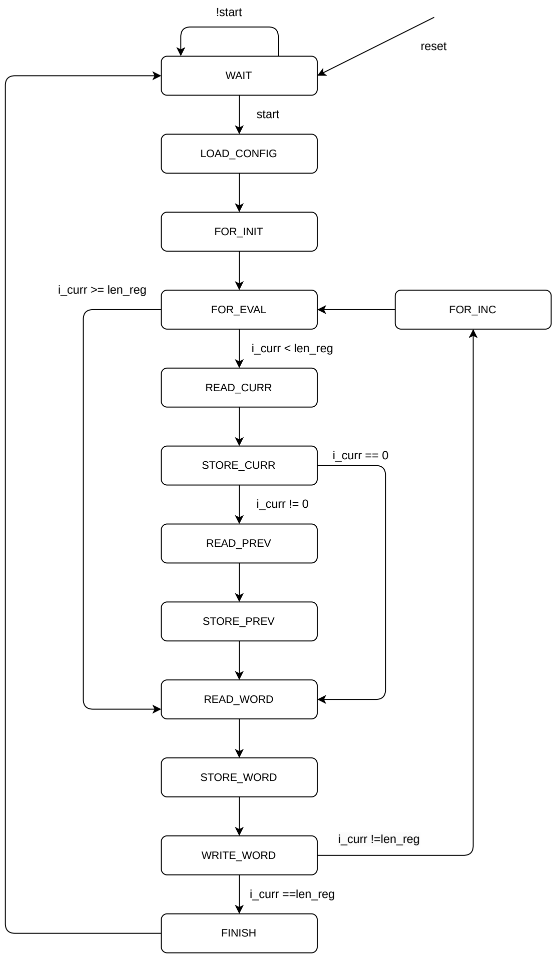


Figure 5.28: Inverse Kronecker segmentation finite state machine

5.5.3 HQC reduction

The module **hqc_reduction.sv**, implements the reduction of the binary polynomial, for all level of security of HQC. This component implements an FSM, as show in Figure 5.29. Looking at the C Code 5.5, it is possible to observe that the variables are defined on 64 bits. The accelerator being on 32 bits, has to perform two reads and writes. In the for loop the elements that are accessed from memory are:

- a[i + vec_n_size_64 - 1]
- a[i + vec_n_size_64]
- a[i]

The finite state machine for each instruction it defines a pair of **READS**, in the following way:

- **READ_A0** and **READ_A1** are used for a[i + vec_n_size_64 - 1]
- **READ_A2** and **READ_A3** are used for a[i + vec_n_size_64]
- **READ_A4** and **READ_A5** are used for a[i]

The state **COMPUTE** performs the operation **o**[**i**] = **a**[**i**] $\oplus$ **r** $\oplus$ **carry**. The value is written in memory through states **WRITE_0** and **WRITE_1**. The final operation in line 16, is done in **COMPUTE**, instead of performing **o**[i] = **a**[i] $\oplus$ **r** $\oplus$ **carry**, it performs **o**[i] = (**a**[i] $\oplus$ **r** $\oplus$ **carry**) & **red_mask**.

```c
1  void hqc_reduce(uint64_t *o, const uint64_t *a, pqc_algorithm algorithm)
2  {
3      uint64_t r;
4      uint64_t carry;
5      int vec_n_size_64 = binary_2_vec64(algorithm);
6      int param_n = parameter_n(algorithm);
7      uint64_t red_mask = red_mask_sel(algorithm);
8  
9      for (int i = 0; i < vec_n_size_64; ++i)
10     {
11         r = a[i + vec_n_size_64 - 1] >> (param_n & 0x3F);
12         carry = a[i + vec_n_size_64] << (64 - (param_n & 0x3F));
13         o[i] = a[i] ^ r ^ carry;
14     }
15 
16     o[vec_n_size_64 - 1] &= red_mask;
17 }
```

Code 5.5: HQC reduction

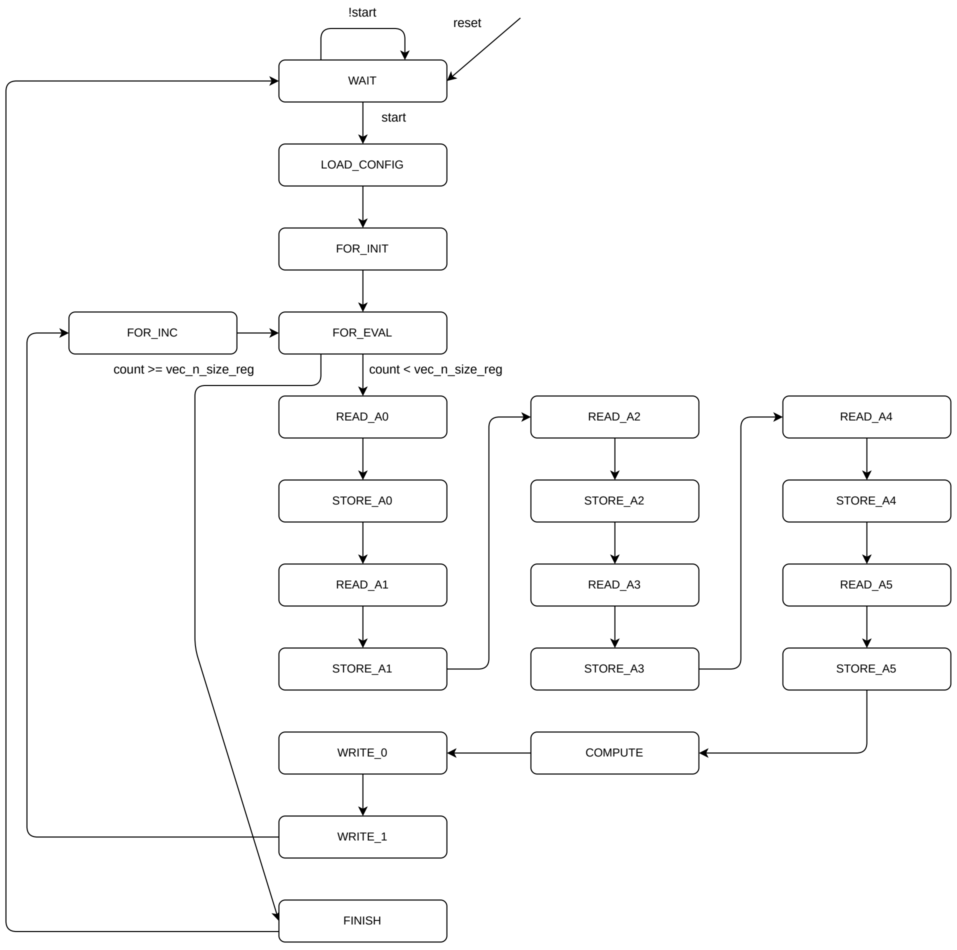


Figure 5.29: HQC reduction finite state machine

5.6 Polynomial multiplier

Now that we've covered all the main subsystems and their parts, let's dive into how the polynomial multiplier works. The Finite State Machine (Figure 5.30) operates differently for McEliece and HQC algorithms, but there are some common stages like **FFT_A, FFT_B, POINTWISE_B**, and **IFFT**, which handle the polynomial multiplication.

In HQC, before we do the FFT, we first split the polynomials $A(x)$ and $B(x)$ using Kronecker segmentation. After the multiplication, what happens next depends on the algorithm. For McEliece, the polynomial is just reduced to size `t`. But for HQC, we have to convert the polynomial back to binary form before reducing it further.

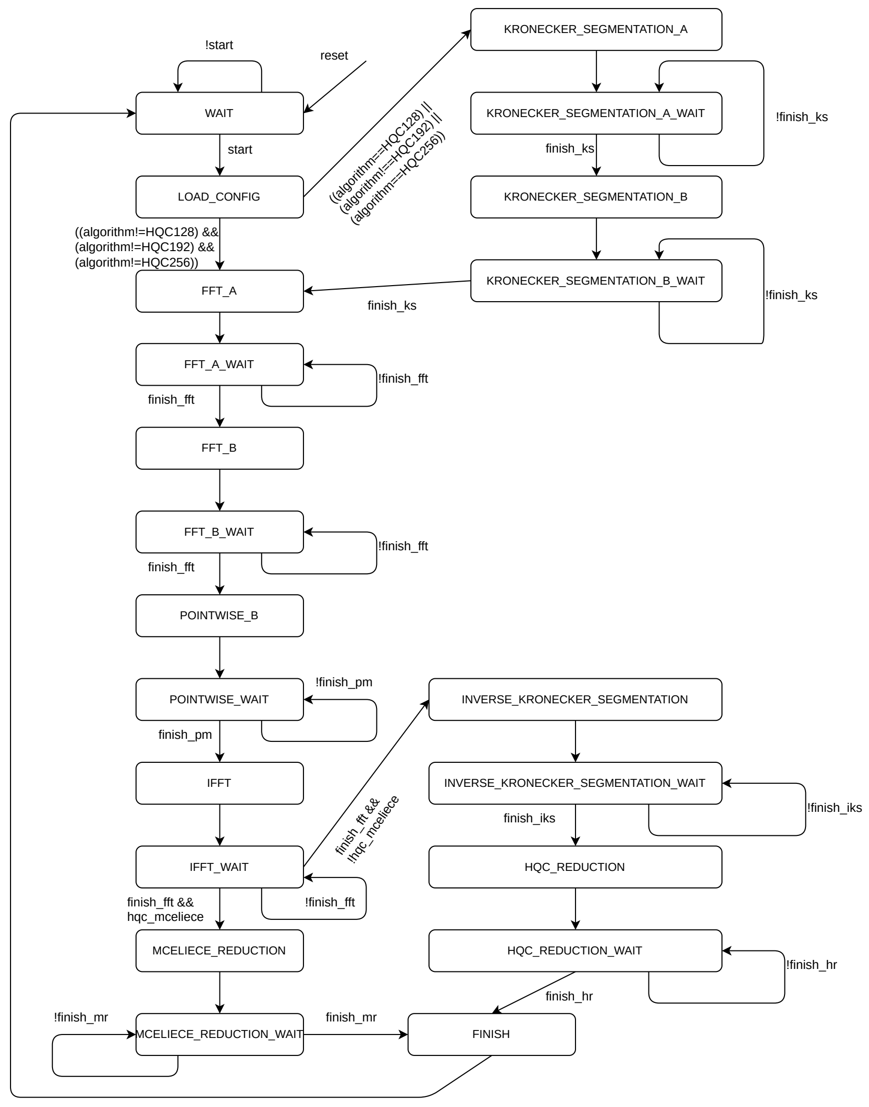


Figure 5.30: Polynomial multiplier finite state machine

Chapter 6

Results

The initial objective of the study was to design a hardware accelerator, that would be integrated into the X-HEEP[1][16] microcontroller. Due to time constraints, the hardware accelerator was tested as a standalone unit rather than being integrated into X-HEEP as initially planned. To prove that the accelerator presents a significant advantage with respect to the software model, the execution time of the software functions was confronted with the execution time of the hardware modules. In this chapter, the results achieved through the hardware implementation with those obtained from running the software implementation on the simulated RISC-V microcontroller X-HEEP are presented. The chapter will be organized in the following sections:

- X-HEEP: will give a brief introduction to the platform and explain the functioning of the simulation envirorment.
- Simulation results: will present the performance achieved in software and hardware.
- Comparison: will analyze and compare the results obtained in software and hardware.
- Potential improvements: will discuss the current limitation of the system, and propose different solutions in order to solve them.

6.1 X-HEEP

X-HEEP(eXtendable Heterogeneous Energy-Efficient Platform) is an open-source RISC-V microcontroller described in SystemVerilog. It is designed to be highly configurable, capable of targeting small and tiny platforms, and can be extended to support accelerators[16]. The unique feature of X-HEEP is its simplicity and customizability. It provides a basic microcontroller unit (MCU) with CPUs, common peripherals, and memories, allowing users to extend it with their own accelerator without modifying the MCU. This means

[1] https://github.com/esl-epfl/x-heep

users can focus on building their special hardware supported by the microcontroller. X-HEEP supports simulation with Verilator, Questasim, and others. Firmware can be built and linked using CMake with either gcc or clang.
In the following subsection, we will discuss the operation and configuration of the simulation environment utilized for the execution of the software functions.

6.1.1 Simulation Envirorment

The X-HEEP simulation environment offers various parameters for adjustment, including CPU type, BUS type, number of memory banks, and number of interleaved banks.
For the simulation of the software model functions, the following parameters were chosen:

- **CPU**=cv32e40x
- **BUS**=NtoM
- **MEMORY_BANKS**=64
- **MEMORY_BANKS_IL**=0

Due to the dimension of the algorithm employed in the FFT, the linker parameters relative to the heap and stack, were modified in order to accommodate the functions.
The simulation envirorment is organized through makefile, which manage the compilation and execution of the codes. The microcontroller offers performance counters, which are accessible through dedicated functions defined in the device libraries. In Code 6.1, a template illustrates the process of capturing the number of clock cycles of a function.
To utilize this template:

1. First, include the library `csr.h` to access the performance counter.
2. Enable the performance counter (**line 26**).
3. Reset the performance counter to 0 (**line 27**).
4. Execute the function of interest (**line 29**).
5. Capture the number of clock cycles (**line 31**).
6. Utilize a print function (**line 32**) to save the cycle count in the file uart0.log.

This process enables the capture of performance metrics, such as the number of clock cycles, for the specified function. The result is then stored in the file uart0.log for further analysis.
The performance capture of various software functions follows the template outlined in Code 6.1. The simulation was conducted using ModelSim.

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include "csr.h"
#include "rv_plic.h"
#include "rv_plic_regs.h"
#include "rv_plic_structs.h"
#include "hart.h"
#include "x-heep.h"

int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}



int main(int argc, char *argv[])
{
    unsigned int cycles;
    int n, i;
    n=10;

    CSR_CLEAR_BITS(CSR_REG_MCOUNTINHIBIT, 0x1);
    CSR_WRITE(CSR_REG_MCYCLE, 0);

    fibonacci(n);

    CSR_READ(CSR_REG_MCYCLE, &cycles);
    printf("Number of clock cycles keygen: %d\n", cycles);
    return EXIT_SUCCESS;
}
```

Code 6.1: Template performance

6.2 Simulation results

The accelerator underwent simulation and validation against the C-based software model. Testing procedures were conducted using SystemVerilog and Python. SystemVerilog was used to develop testbenches for individual modules, while Python was employed to convert C functions and generate corresponding tests for the modules. The testing flow structure is depicted in Figure 6.1. Each module includes a **test_generation.py** file responsible for generating input data and the golden output for the testbench. This data is then utilized by the testbench to produce results, which are subsequently compared with the golden output. If the results match, a pass value is generated.

Each module was successfully tested, and as it stands, the accelerator functions with all parameter sets of HQC and McEliece.

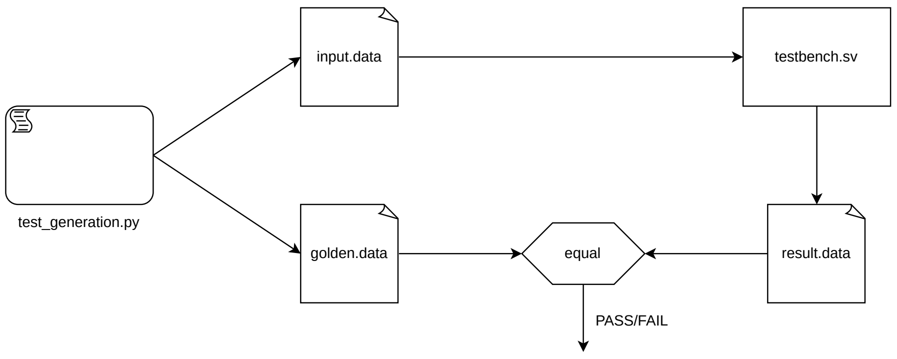


Figure 6.1: Testing flow

6.3 Modelsim simulation accelerator

For the different modules of the accelerator, the number of clock cycles to compute their execution has been computed. The result is divided in different tables:

- Table 6.1, Table 6.2, Table 6.3 summarize the clock cycles for the field operations.
- Table 6.4, Table 6.5 and Table 6.6 summarized the clock cycles for the FFT components for the fields considered.
- Table 6.7 summarize the clock cycles for the execution of the FFT and IFFT for the three fields.
- Table 6.8 and Table 6.9, present the reduction clock cycles for each algorithm and level of security.
- Table 6.10 and Table 6.11, present the clock cycles for each level of security of HQC for the direct and inverse kronecker segmentation.

From the result obtained, it is possible to identify as the heaviest operation the FFT, when executing the polynomial multiplication in HQC-256(the worst case) the FFT occupies 99% of the clock cycles. This implies also that is better to concentrate on improving the FFT since the other modules occupy just 1 % of the execution time.

Operation	**Clock Cycles**
mul	1
square	1
inversion	18

Table 6.1: Arithmetic logic $\mathbb{F}_{2^{12}}$ hardware

Operation	**Clock Cycles**
mul	1
square	1
inversion	18

Table 6.2: Arithmetic logic $\mathbb{F}_{2^{13}}$ hardware

Operation	**Clock Cycles**
mul	1
square	1
inversion	24

Table 6.3: Arithmetic logic $\mathbb{F}_{2^{16}}$ hardware

Module	**Clock cycles[FFT]**	**Clock cycles[IFFT]**
compute_g	20480	20498
iterative_optimized_taylor_expand	64556	64556
compute_initial_w	5124	5142
compute_subset_sums	10284	10284
compute_w	10244	10244
precompute_taylor_expand	964639	1001467
butterfly	147762	147762

Table 6.4: FFT and IFFT hardware modules in $\mathbb{F}_{2^{12}}$

Module	**Clock cycles[FFT]**	**Clock cycle[IFFT]**
compute_g	40960	40978
iterative_optimized_taylor_expand	139312	139312
compute_initial_w	10244	10262
compute_subset_sums	20528	20528
compute_w	20484	20484
precompute_taylor_expand	2170916	2244608
butterfly	315751	315751

Table 6.5: FFT and IFFT hardware modules in $\mathbb{F}_{2^{13}}$

Module	**Clock cycles[FFT]**	**Clock cycles[IFFT]**
compute_g	327680	327704
iterative_optimized_taylor_expand	1359932	1359932
compute_initial_w	81924	81948
compute_subset_sums	163900	163900
compute_w	163844	163844
precompute_taylor_expand	23658547	24444931
butterfly	3015198	3015198

Table 6.6: FFT and IFFT hardware modules in $\mathbb{F}_{2^{16}}$

Field	**Clock cycles[FFT]**	**Clock cycles[IFFT]**
$\mathbb{F}_{2^{12}}$	1117530	1154376
$\mathbb{F}_{2^{13}}$	2496916	2570626
$\mathbb{F}_{2^{16}}$	26755674	27542082

Table 6.7: FTT and IFFT hardware

Algorithm	**Clock cycles**
mceliece348864/f	823
mceliece460896/f	1524
mceliece6688128/f	2036
mceliece6960119/f	1184
mceliece8192128/f	2036

Table 6.8: Mceliece reduction hardware

Algorithm	**Clock cycles**
HQC-128	4713
HQC-192	9541
HQC-256	15321

Table 6.9: HQC reduction hardware

Algorithm	**Clock cycles**
HQC-128	4436
HQC-192	8980
HQC-256	14420

Table 6.10: Kronecker segmentation hardware

Algorithm	Clock cycles
HQC-128	39875
HQC-192	80771
HQC-256	129731

Table 6.11: Inverse Kronecker segmentation hardware

6.4 Modelsim simulation software

The software functions developed in the software model have been executed in the simulated environment of the RISC-V microcontroller X-HEEP. For each function, the execution time has been captured in terms of clock cycles. The result is organized in the following tables:

- Table 6.12, Table 6.13, Table 6.14 summarize the clock cycles for the field operations.
- Table 6.15, Table 6.16 and Table 6.17 summarize the clock cycles for the FFT components for the field considered.
- Table 6.18 and Table 6.19, present the reduction clock cycles for each algorithm and level of security.
- Table 6.20, presents the clock cycles for each level of security of HQC for the Kronecker segmentation.

The functions indicated with the value N/A, due to the time to complete the simulation, do not present a result. The tables relative to the FFT and IFFT, and the one interesting the inverse Kronecker segmentation are not presented for the same reason.

Operation	Bits	Clock Cycles
mul	12	137
square	12	41
inversion	12	1131

Table 6.12: Arithmetic logic $\mathbb{F}_{2^{12}}$ software

Operation	Bits	Clock Cycles
mul	13	155
square	13	155
inversion	13	2286

Table 6.13: Arithmetic logic $\mathbb{F}_{2^{13}}$ software

Operation	**Bits**	**Clock Cycles**
mul	16	314
square	16	314
inversion	16	6589

Table 6.14: Arithmetic logic $\mathbb{F}_{2^{16}}$ software

Module	**Clock cycles[FFT]**	**Clock cycles[IFFT]**
compute_g	1138449	5439249
iterative_optimized_taylor_expand	278764	225467
compute_initial_w	278572	2357293
compute_subset_sums	32886	32886
compute_w	294951	288806
precompute_taylor_expand	N/A	N/A
butterfly	N/A	N/A

Table 6.15: FFT and IFFT functions in $\mathbb{F}_{2^{12}}$

Module	**Clock cycles[FFT]**	**Clock cycle[IFFT]**
compute_g	2555625	15908585
iterative_optimized_taylor_expand	585984	485580
compute_initial_w	622631	7254056
compute_subset_sums	65661	65661
compute_w	655399	643110
precompute_taylor_expand	N/A	N/A
butterfly	N/A	N/A

Table 6.16: FFT and IFFT functions in $\mathbb{F}_{2^{13}}$

Module	**Clock cycles[FFT]**	**Clock cycles[IFFT]**
compute_g	35651074	468319747
iterative_optimized_taylor_expand	5374262	4606324
compute_initial_w	9633832	237076521
compute_subset_sums	524446	524446
compute_w	9895976	9797671
precompute_taylor_expand	N/A	N/A
butterfly	N/A	N/A

Table 6.17: FFT and IFFT functions in $\mathbb{F}_{2^{16}}$

Algorithm	Clock cycles
mceliece348864/f	9634
mceliece460896/f	3004
mceliece6688128/f	3870
mceliece6960119/f	2537
mceliece8192128/f	3870

Table 6.18: Mceliece reduction software

Algorithm	Clock cycles
HQC-128	8653
HQC-192	17445
HQC-256	28885

Table 6.19: HQC reduction software

Algorithm	Clock cycles
HQC-128	36578
HQC-192	74062
HQC-256	118942

Table 6.20: Kronecker segmentation software

6.4.1 Comparison simulations

The hardware accelerator has demonstrated significantly reduced execution times compared to a microcontroller-based execution. As stated in section 6.3, the execution time is heavily capitalized by the FFT. The object of discussion of this comparison will deal with the FFT since it is the heaviest module inside the polynomial multiplier. The Table 6.21 summarizes the execution time of the FFT functions in the software model and hardware accelerator. As it is possible to observe, the hardware modules present a significant speedup with respect to the software function. This behavior is mainly due to the arithmetic logic. Inside the software model, all the critical operations such as multiplication and squaring are implemented through for loops, this causes to have a high number of cycles in the operations and consequently in the inversion, as it can be seen in Table 6.13. The hardware approach solves this problem by implementing multiplication and squaring as combinational circuits. This translates also in a high performance in the inversion. Therefore in the FFT, it is possible to observe that when the field operations are used in the functions, a significant speed-up can be observed in the accelerator. This is particularly clear when looking at **compute_g** and **compute_initial_w**, as it is presented in Table 6.21. Those two functions during inversion rely heavily on the inversion when performing the inverse operation. The speedup is less noticeable when the functions are not resorting to the field arithmetic. Considering the **iterative_optimized_taylor_expand**, the only

operation that is executed is the xor, therefore in software, the overhead is given only by the instructions execution in the pipeline.
In summary, the system achieves a greater performance than the software model due to the efficient arithmetic logic and reduced overhead in execution, with respect to a micro-controller.

Module	**HW Clock cycles [FFT]**	**HW Clock cycles [IFFT]**	**SW Clock cycles [FFT]**	**SW Clock cycles [IFFT]**	**Speed up factor [FFT]**	**Speed up factor [IFFT]**
compute_g	327680	327704	35651074	468319747	108.79	1429.09
iterative_optimized_taylor_expand	1359932	1359932	5374262	4606324	3.95	3.38
compute_initial_w	81924	81948	9633832	237076521	117.59	2893.01
compute_subset_sums	163900	163900	524446	524446	3.19	3.19
compute_w	163844	163844	9895976	9797671	60.39	59.79
precompute_taylor_expand	23658547	24444931	N/A	N/A	N/A	N/A
butterfly	3015198	3015198	N/A	N/A	N/A	N/A

Table 6.21: FFT and IFFT functions in $\mathbb{F}_{2^{16}}$ in hardware

6.5 Potential improvements

At its current state, the software model of the accelerator falls short in comparison to the polynomial computation methods employed in HQC and McEliece schemes. This discrepancy, if replicated in hardware, would result in poor performance. To enhance the competitiveness of the accelerator, several strategies could be considered. One potential improvement is to replace the current Fast Fourier Transform (AFFT) with the Frobenius Additive FFT as proposed by Li et al. (2018) [12]. This alternative FFT algorithm may offer better efficiency and performance. Another approach worth exploring is the implementation of a truncated version of the existing AFFT. By reducing the number of evaluation points, this approach aims to reduce the execution time of the accelerator.

Chapter 7

Conclusion

In conclusion, this thesis has introduced a novel approach to polynomial multiplication in code-based cryptographic algorithms by utilizing the Additive Fast Fourier Transform (AFFT). Departing from conventional methods, this alternative approach aimed to enhance efficiency and performance in hardware acceleration for code-based cryptography. While the anticipated performance benchmarks were not fully realized, the thesis has nonetheless succeeded in developing a functional hardware accelerator.

Despite the challenges encountered and the deviation from initial performance expectations, the creation of a working hardware accelerator represents a significant milestone. This achievement underscores the feasibility and potential of employing AFFT in hardware implementations for polynomial computation in code-based cryptography.

Moreover, the thesis contributes to the broader landscape of algorithmic research and hardware development in code-based cryptography. By exploring innovative approaches to polynomial multiplication and hardware acceleration, this work enriches the arsenal of techniques available to cryptographic researchers and practitioners. The insights gained from this endeavor pave the way for further refinement and optimization of AFFT-based hardware accelerators, potentially unlocking new avenues for enhancing the efficiency and scalability of code-based cryptographic systems.

Looking ahead, future research directions may focus on addressing the performance limitations identified in this thesis, such as optimizing resource utilization and mitigating computational overhead. Additionally, continued exploration of alternative algorithms and hardware architectures could yield further improvements in hardware acceleration for code-based cryptography. Ultimately, the pursuit of innovative solutions and the collaborative effort of researchers and practitioners will continue to propel advancements in cryptographic techniques, ensuring the resilience and security of digital communication in an ever-evolving landscape of threats and challenges.

Bibliography

[1] IBM Quantum Computing Blog | Charting the course to 100,000 qubits — ibm.com. https://www.ibm.com/quantum/blog/100k-qubit-supercomputer. [Accessed 23-03-2024].

[2] D. J. Bernstein, T. Chou, and P. Schwabe. Mcbits: fast constant-time code-based cryptography. In Cryptographic Hardware and Embedded Systems-CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings 15, pages 250–272. Springer, 2013.

[3] I. Blake, G. Seroussi, and N. Smart. Finite Field Arithmetic, page 11–28. London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[4] I. F. Blake, S. Gao, and R. J. Lambert. Construction and distribution problems for irreducible trinomials over finite fields. In INSTITUTE OF MATHEMATICS AND ITS APPLICATIONS CONFERENCE SERIES, volume 59, pages 19–32. Citeseer, 1996.

[5] D. G. Cantor. On arithmetical algorithms over finite fields. Journal of Combinatorial Theory, Series A, 50(2):285–300, 1989.

[6] M.-S. Chen, C.-M. Cheng, P.-C. Kuo, W.-D. Li, and B.-Y. Yang. Faster multiplication for long binary polynomials. arXiv preprint arXiv:1708.09746, 2017.

[7] S. Gao and T. Mateer. Additive Fast Fourier Transforms Over Finite Fields. IEEE Transactions on Information Theory, 56(12):6265–6272, Dec. 2010.

[8] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in gf (2m) using normal bases. Information and computation, 78(3):171–177, 1988.

[9] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers. Improving software quality in cryptography standardization projects. In 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pages 19–30. IEEE, 2022.

[10] D. E. Knuth. The art of computer programming. 2: Seminumerical algorithms. Addison-Wesley, Reading, Mass, 2. ed., 24. [print.] - 1995 edition, 1995.

[11] S. S. Kumar. Elliptic curve cryptography for constrained devices. PhD thesis, Bochum, Univ., Diss., 2006, 2006.

[12] W.-D. Li, M.-S. Chen, P.-C. Kuo, C.-M. Cheng, and B.-Y. Yang. Frobenius additive fast fourier transform. In Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation, pages 263–270, 2018.

[13] Z. Liang and Y. Zhao. Number theoretic transform and its applications in lattice-based cryptosystems: A survey. arXiv preprint arXiv:2211.13546, 2022.

[14] P. Longa and M. Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15, pages 124–139. Springer, 2016.

[15] J. López and R. Dahab. High-speed software multiplication in f. volume 1977, pages 93–102, 01 2000.

[16] S. Machetti, P. D. Schiavone, T. C. Müller, M. Peón-Quirós, and D. Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators. arXiv preprint arXiv:2401.05548, 2024.

[17] G. Seroussi. Table of low-weight binary irreducible polynomials. 1998.

[18] S. L. Vianney Rancure, Lam Pham-Sy. Quadiron: A library for number theoretic transform-based erasure codes. 2020.

[19] J. Von zur Gathen and J. Gerhard. Arithmetic and factorization of polynomial over f 2. In Proceedings of the 1996 international symposium on Symbolic and algebraic computation, pages 1–9, 1996.

[20] Y. Wang and X. Zhu. A fast algorithm for the fourier transform over finite fields and its vlsi implementation. IEEE Journal on Selected Areas in Communications, 6(3):572–577, 1988.

[21] V. Weger, N. Gassner, and J. Rosenthal. A survey on code-based cryptography. arXiv preprint arXiv:2201.07119, 2022.