Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Energy Engineering (35^{th}cycle)

Architectures for Code-based Post-Quantum Cryptography

Exploration of architectures efficient for QC-MPDC codes multiplication and approaches for generic Post-Quantum cryptographic primitives

By

Kristjane Koleci

******

Supervisor(s):
Prof. Maurizio Martina, Supervisor
Prof. Guido Masera, Co-Supervisor

Doctoral Examination Committee:
Prof. Sergio Saponara, Referee, Università degli Studi di Pisa
Prof. William Fornaciari, Referee, Politecnico di Milano
Prof. E.F, University of...
Prof. G.H, University of...
Prof. I.J, University of...

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Kristjane Koleci
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my brother, my mum and my dad who has always been there for me.

Abstract

Post-Quantum Cryptography has become popular in recent years due to overcoming the threat posed by Quantum Computers; the field is a branch of Cryptography and aims to provide algorithms that can secure communication even if a quantum computer is employed.

In Post-Quantum Cryptography, the research focuses on algorithms for Asymmetric Cryptography, where the Public Key and a Secret Key pair are employed to secure our data. The main focus of this thesis is Code-based Post-Quantum Cryptographic schemes; the security of such schemes is based on the NP-hardness of decoding a general linear code. The algorithms considered are LEDAcrypt and BIKE; they adopt a variant of the Classic McEliece cryptosystems based on Quasi-Cyclic Moderate Density Parity Check Codes. The scheme has three primitives for Key Generation, Encryption, and Decryption.

The Encryption and Decryption algorithms have been implemented in this thesis; exploring the efficient design of such schemes is fundamental since they require more computational capabilities than preexisting algorithms. In LEDAcrypt and BIKE, the implementation exploits the Quasi-Cyclic structure and sparsity of the matrices to accelerate the decoding while simultaneously trying to keep the total area bounded. The fundamental component of Encryption and Decryption is the multiplier for cyclic matrices. Initially, the direction of the research has been dedicated to designing an efficient multiplier with sparse cyclic matrices. The decoder and encoder have been designed such that they can adapt to the updates on the parameters and the algorithm.

The architecture that has been developed targeted both Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC); the design is scalable and thus can adapt to both low-end and high-end applications. The FPGA results for the Artix-7 200 device obtained a decoder latency of 0.6 *ms* and resource utilization of at most 30%. The ASIC design has been synthesized for the STM

FDSOI 28 nm technological node and achieved a latency of 0.15 *ms* and a total area of 0.0 $9\mu m^2$.

The decoder is the most critical unit for the security of the whole system since the Secret Key is fundamental in the inner computations. In the present work, a Side-channel attack that exploits the dynamic power consumption is applied to the multiplier in the decoder; the target is the Secret Key which is employed in the multiplication and results in the complete disclosing of the key. The method is applied to real and simulated measurements (from the netlist) to validate a methodology that aims to include a preliminary study of the security during the design phase.

In the last study, the multiplier is implemented as a Number Theoretic Transform-based multiplier due to the analogy between the cyclic matrix product and the cyclic convolution. The unit is fundamental for both the encryption and decryption stage of Code-based Post-Quantum schemes; in this work, has optimized it for both sparse and dense vectors to fully benefit from using the Number Theoretic Transform based multiplier. The result of such an approach has been provided for Field Programmable Gate Arrays (FPGA) and Application Specific Integrated Circuits (ASIC). In the end, it has been showing its flexibility when applied to both encryption and decryption since if we consider as a metric the product of latency and area; it is 3 to 10 times more efficient than other proposals. Moreover, such an approach's flexibility also extends to generic PQC primitives, where such a multiplier is employed.

Contents

List of Figures **x**

List of Tables **xiv**

1 Introduction **1**

2 Cryptography **6**

2.1 Asymmetric Cryptography 7

2.1.1 Modern Cryptography Schemes 9

2.2 Symmetric Cryptography 11

2.3 Post-Quantum Cryptography 12

2.3.1 Lattice-based Schemes 13

3 Code-Based Post Quantum Cryptography **15**

3.1 Notations and definitions 16

3.1.1 Constants, vector and matrices 16

3.1.2 Polynomials 18

3.1.3 Operators 18

3.2 McEliece Cyrptosystem 19

3.2.1 Scheme algorithms 20

3.2.2 Niederreiter Cyrptosystem 21

3.2.3 Limitations and attacks . . . 23
3.3 Low-Density Parity Check Codes . . . 23
3.3.1 Structured Codes: QC-LDPC/MDPC . . . 27
3.4 Code-based QC-LDPC schemes . . . 28
3.4.1 LEDAcrypt . . . 29
3.4.2 BIKE . . . 35
3.5 Arithmetic in Code-based PQC . . . 38
3.5.1 Polynomial Sum . . . 38
3.5.2 Polynomial Inversion . . . 39
3.5.3 Polynomial Multiplication . . . 39

4 Fast Polynomial Multiplication 41
4.1 Polynomial Multiplication Overview . . . 41
4.1.1 Schoolbook Multiplication . . . 42
4.1.2 Karatsuba Multiplication . . . 44
4.1.3 Schönhage Stressen Multiplication . . . 45
4.2 Polynomial Multiplication with QC-MDPC/LDPC Matrices . . . 46
4.2.1 Large and Binary/Integer Polynomial Multiplication . . . 46
4.2.2 Sparse Polynomial multiplication in Schoolbook Multiplier 48

5 Architectures for Large Polynomial Multiplication 51
5.1 Schoolbook architecture . . . 53
5.1.1 VectorByCirculant . . . 53
5.1.2 VectorByCirculant Pipelined . . . 68
5.1.3 SparseVectorByCirculant . . . 73
5.2 Schönhage Stressen Architecture . . . 78
5.2.1 Number Theoretic Transform Computation . . . 78

5.2.2 Number Theoretic Transform parameters 79
5.2.3 Number Theoretic Transform Architecture 82
5.2.4 NTT-based multiplier for Code-based PQC 90
5.2.5 Architecture 91
5.3 Results 97
5.3.1 ASIC Results 97
5.3.2 FPGA Results 98
5.3.3 Latency 99
5.4 Side-Channel Attack on VectorByCirculant 100
5.4.1 Results validation and conclusions 106

6 Decryption and Encryption in QC-MDPC Codes for PQC 107
6.1 Encryption and encapsulation 108
6.1.1 Implementation and Comparison 108
6.2 Decryption and decapsulation 109
6.2.1 Decoder analysis 109
6.2.2 Decoder Architecture 112
6.2.3 Implementation Results 117
6.3 Flexible-NTT in Code based Post Quantum Cryptography 120
6.3.1 Encoder 121
6.3.2 Decoder 121

7 Conclusions 125

References 127

List of Figures

2.1 Message ubfuscation by means of a cryptographic function 7

2.2 Block Diagram of the Key Encapsulation Mechanism schemes . . . 9

2.3 Block Diagram of the Public Key Cryptography schemes 9

2.4 Block Diagram of a Symmetric Cryptosystem 11

2.5 The example of a two dimensional lattice with the basis, b_1 and b_2 basis vector highlighted and the shortest vector c, ... 14

3.1 Block scheme for McElice-like crytosystem 20

3.2 The example of a codeword of length n with the different sections highlighted: the message is in yellow, with $n-k$ bit, the redundant part with k bits, the t error bit. 21

3.3 Block scheme for Niederreiter-like crytosystem 22

4.1 Time Complexity 47

4.2 Time Complexity with sparse Schoolbook 49

5.1 Memories format with the sizes specified. 56

5.2 Memories in input to `VectorByCirculant` 57

5.3 Shift Register content to rotate $\texttt{MEM}_{\mathbf{a}}$ by b_2 58

5.4 Shift Register loaded with the complete row with n_b reads from $\texttt{REG}_{\mathbf{A}}$ and $\texttt{REG}_{\mathbf{B}}$ 58

5.5 Register with the new row loaded in parallel. 59

5.6 Memories of VectorByCirculant with Input, Output and addressed highlighted. 60

5.7 VectorByCirculant Data Path with the control signal form $\mathtt{MEM_b}$, the consecutive rows from $\mathtt{MEM_a}$ stored in $\mathtt{REG_A}$ and $\mathtt{REG_B}$ and the first row of $\mathtt{MEM_c}$ evaluated. 61

5.8 Memories of VectorByCirculant with Input, Output and addressed highlighted for the writing of $\mathtt{MEM_c}$ at $\mathtt{ADX}_c = 1$. 61

5.9 Memories of VectorByCirculant with Input, Output and addressed highlighted for $\mathbf{a}^1$ computation. 62

5.10 VectorByCirculant Data Path with the control signal form $\mathtt{MEM_b}$, the consecutive rows from $\mathtt{MEM_a}$ stored in $\mathtt{REG_A}$ and $\mathtt{REG_B}$ and the first row of $\mathtt{MEM_c}$ evaluated. 63

5.11 $\mathtt{MEM_a}$ with a vector of length $n = 13$ and padded with zeros 63

5.12 VectorByCirculant Data Path with the control signal form $\mathtt{MEM_b}$, the consecutive rows from $\mathtt{MEM_a}$ stored in $\mathtt{REG_A}$ and $\mathtt{REG_b}$ and the first row of $\mathtt{MEM_c}$ evaluated. 64

5.13 Control Unit of VectorByCirculant 64

5.14 The time evolution of the Control Unit to compute a single row in $\mathtt{MEM_b}$ after the initialization steps. 65

5.15 Logarithmic approach for the Barrel Shifter 67

5.16 VectorByCirculant alternatives 68

5.17 The time evolution of the Control Unit for the coarse grain pipelined architecture to compute a single row in $\mathtt{MEM_c}$ after the initialization steps. 69

5.18 The time evolution of the Control Unit for the fine grained pipelined architecture to compute a single row in $\mathtt{MEM_c}$ after the initialization steps. 71

5.19 VectorByCirculant Pipelined Data Path with the control signal form $\mathtt{MEM_b}$, the consecutive rows from $\mathtt{MEM_a}$ stored in $\mathtt{REG_A}$ and $\mathtt{REG_B}$ and the first row of $\mathtt{MEM_c}$ evaluated. 72

5.20 The Data Path of `SparseVectorByCirculant` multiplier. The modular sum is in the orange box. 75

5.21 Time evolution of `SparseVectorByCirculant` 76

5.22 Time evolution of Pipelined `SparseVectorByCirculant` 76

5.23 Memories of `SparseVectorByCirculant` with sparse Input, and Output with highlighted position of the asserted bits from the input. 77

5.24 Butterfly Unit with one multiplier (`MPY`), adder (`ADD`), subtractor (`SUB`). 83

5.25 NTT structure with $n = 8$ 84

5.26 8-points NTT with a dense input 85

5.27 8.points NTT with sparse input, asserted elements: $v_0(3)$ and $v_0(2)$. 87

5.28 8.points NTT with sparse input, asserted elements: $v_0(3)$ and $v_0(6)$. 88

5.29 The DataPath shared between the 92

5.30 Control Unit hierarchy. The Multiplier is the master CU, while the sub-CU communicate independently with the Twiddle Factor Management CUs. 92

5.31 Input memory for the Sparse NTT execution 93

5.32 Twiddle Factors Management Units 96

5.33 Shared resources in NTT-based multiplier 97

5.34 The Power Trace derived with Synopsys Prime time during the execution of a complete product [1]. 102

5.35 The netlist, .sdf and technological node files are the inputs of both QuestaSim and Prime Time, the .vdc file, containing the information on the switching activity, is derived from a random input provided in the test bench. 103

5.36 Correlation trace corresponding to asserted value11 in the SK. The dashed box highlights the correlation peak[1]. 103

5.37 The plot shows the value for correlation peaks associated to correct key value, in blue, and the average value that is assume by the correlation associated to wrong keys. 105

6.1 Implementation of the Encryption 109

6.2 *Version0*:The first version implements LEDAcrypt Decoder submitted to Round 1 of the PQC Competition. the evaluation of *Syndrome*, *Unsatisfied Parity Check* and the `Syndrome Update` is computed with two distinct modules for binary and integer output, the additional elements are in charge of computing the threshold and correct the errors. 113

6.3 *Version1*:The Round 3 Decoder computes the *Syndrome*, *Unsatisfied Parity Check* and the `Syndrome Update` by means of a single matrix, thus the input matrix is **H**. The other modules remains unchanged. . 114

6.4 *Version2*:The total area occupation is reduced by implementing the *Syndrome*, *Unsatisfied Parity Check* computation with a single unit, the additional modules are kept unchanged. 115

6.5 The total area occupation of the *Version0* Decoder with the partial area occupation of the main units[2]. 119

6.6 The comparison of latency in *Version0* Decoder with and without the the use of *VectorByCirculant* to update the Syndrome 119

6.7 Latency of the Q-Decoder comparison. 120

List of Tables

3.1 Considered LEDAcrypt parameters for KEM. 31

3.2 Considered LEDAcrypt parameters for KEM and PKE submitted for Round 3. 32

3.3 BIKE parameters submitted to Round 4 and Round 3. 35

5.1 QC-MDPC Code-based PQC crytopsystems parameters. 52

5.2 The synthesis results of `VectorByCirculant` with linear unit barrel shifter. 66

5.3 The synthesis results of `VectorByCirculant` with logarithmic barrel shifter. 66

5.4 The synthesis results of `VectorByCirculant` with logarithmic barrel shifter and pipelined execution. 73

5.5 The synthesis results of `VectorByCirculant` with logarithmic barrel shifter and fine grain pipelining. 73

5.6 The synthesis results of `SparseVectorByCirculant` with and without pipelining. 77

5.7 ASIC synthesis results for the UMC 65 nm technology. $d_v = 10$. Percentage delay and area values are given with respect to the reference synthesis with constraint $t_{cp} = 5$ ns. 98

5.8 FPGA resource utilization with Artix-7 200. The operating frequency is set to 100 MHz and $d_v = 10$. 99

5.9 The execution time with different length N' and density (d_v). The results are provided in terms of both number of clock cycles and *ms*, assuming the largest achievable clock frequency. 99

6.1 ASIC synthesis of the whole decoder with the UMC 65 nm technology, the results refers to a block length of $m = 27,779$ (LEDAcrypt parameters submitted to Round 1). The reference architecture is in [3]. 118

6.2 QC-MDPC encoding latency in terms of number of cycles vs LUTs utilization. 121

6.3 The clock cycles required in each iteration of the decoder. 123

6.4 LC figure of merit for combined encoder and decoder: comparison between the proposed solution and state-of-the-art solutions. 124

Chapter 1

Introduction

Quantum computing is considered the next big revolution in the area of computing; the concept of Quantum Computers was introduced in 1981 by the physicists Richard Feynman [4]. It was considered a method that could improve simulations of quantum systems, but recently it has become clear that its impact could be more profound on our everyday lives, since it can reduce the simulation time of complex physical systems, enabling new stages of our knowledge of the world. Despite the theoretical background that has been developed in the 80s and 90s, from a technological point of view. Quantum computing is at its early stages. However, Quantum Computing has encountered rapid advancement in recent years[5–8], thanks to contributions from a number of companies, research institutions, and governments interested in this topic.

However, the advent of quantum computers also raises significant concerns regarding the security of our data. Traditional cryptographic algorithms, such as those used in Public Key Cryptography (PKC) (or Asymmetric Cryptography), rely on the difficulty of certain mathematical problems to ensure the confidentiality and integrity of sensitive information. Quantum computers possess the ability to solve these problems exponentially faster than classical computers, rendering many conventional encryption schemes vulnerable to attacks. One of the security requirements upon which an asymmetric cryptosystem is built is the NP hardness of discovering the Secret Key (SK) from the Public Key (PK): the PK is computed by applying a one-way function to the SK, and inverting this function is computationally infeasible. One of the most widespread one-way functions used in current PKC

is based on integer factorization. Factorizing large integers is believed to be a Non-Polynomial (NP) problem with classical computers.

However, it has been demonstrated that by using Shor's algorithm [9] on a Quantum Computing, the problem reduces to polynomial time complexity. Similarly, the security of cryptographic schemes based on Elliptic Curve Cryptography (ECC) relies on the difficulty of discovering the discrete logarithm of a random elliptic curve, but this easily be solved by means of quantum computers and Shor's algorithm [9]. Quantum Computers breaks the security of all the asymmetric cryptosystems, that have been adopted to secure our data or communications.

The above threats have motivated governments and institutions to begin a migration to quantum-resistant asymmetric algorithms. The research area is known as Post-Quantum Cryptography (PQC), the field was first described by Bernstein [10]. In recent years, by recognizing the urgency and importance of this endeavor, the National Institute of Standards and Technology (NIST) initiated a comprehensive selection process in December 2016 to define new standards for Post-Quantum PKC [11]. Initially, 69 candidate cryptographic schemes were submitted, based on various approaches arising from different families of hard mathematical problems. This concerted effort underscores the pressing need to develop cryptographic schemes that can effectively mitigate the vulnerabilities introduced by quantum computers while ensuring the continued security of our communications and sensitive information, the possible issues of underlying problems has to be carefully researched since they can break the schemes.

One of these problems is that of decoding an arbitrary linear code, which in [12] has been proven to be NP-hard. The most remarkable and oldest example of cryptosystems based on such a problem, is the Classic McEliece[13], which is among the schemes admitted to the fourth round of the NIST competition [14], and is widely recognized as one of the most secure and promising solutions to the threats posed by Quantum Computers. The McEliece Cryptosystem remained unbroken for decades. Classic McEliece [15] is the cryptographic scheme submitted to the competition is based on the work of McEliece[13] and the key pair size have been adapted to the current security requirements. The work is the first-ever proposed code-based cryptosystem. The choice of the error correcting code for this purpose, McElice chose binary Goppa codes as secret keys. Despite its largely recognized security, the

McEliece scheme relying on Goppa codes has a major drawback with its very large public keys (hundreds of kilobytes).

In response to the aforementioned issues, researchers have put significant effort into the goal of reducing the public key size of McEliece-like cryptosystems. One solution in this respect consists of replacing the underlying secret error correcting code employed in the original McEliece system with a Quasi-Cyclic (QC) random or pseudorandom code [16].

The idea is that by using codes that admit a compact geometric construction, results in the implementation using less memory, with respect to unstructured large linear codes. Indeed, such an approach has been used in different NIST submissions, like LEDAcrypt [17], BIKE [18] and HQC [19], with BIKE [18] scheme still running as a finalist in the NIST competition.

In this work, we focus on the implementation of the LEDAcrypt/BIKE cryptosystem, which uses a class of state-of-the-art error correcting codes, named Moderate-Density Parity-Check (MDPC) codes. These are similar to Low-Density parity-Check codes described in [20] and [21], as the Secret Key of the asymmetric cryptographic scheme.

The secret MDPC code used in LEDAcrypt is defined by a sparse Quasi-Cyclic parity-check matrix. which was the the result of product of two sparse quasi-cyclic matrices. The additional structure enabled the use of a very efficient decoding algorithm, named *Q-decoder*, which derives from the classical Bit Flipping (BF) decoder [20]. However, the scheme was discovered to be insecure, thus the approach of BIKE has been preferred. The Q-decoder in LEDAcrypt and the BFG-Decoder in BIKE, are specifically tailored for the code structure of LEDAcrypt/BIKE, and they are significantly faster than original BF decoding on unstructured LDPC codes. LEDAcrypt was successfully admitted to the first two selection rounds of the NIST PQC competition, but it was not admitted to the third round due to the discovery a families of weak keys [22].

In this thesis, we refer to two schemes submitted to the NIST competition: LEDAcrypt and BIKE with their submission to the four round of the PQC call. It is important to remark that the initial implementation that has been provided is scalable, and adapts to all versions of LEDAcrypt and BIKE.

In LEDAcrypt and BIKE, the decoder is a fundamental component of cryptographic functions based on error-correcting codes. For this thesis, we focus on implementing the Bit Flipping-based Decoder, one of the LEDAcrypt/BIKE peculiarities since its design is fundamental to have a scheme which is secure and with a low latency. The interest in such a decoder goes beyond its cryptographic application, since efficient decoders are also required in conventional coded transmissions.

The BF decoder is implemented with an efficient polynomial multiplier, since this is the main processing unit for both encryption and decryption primitives of QC-MDPC cryptosystems. The polynomial product is fundamental in all PQC primitives, including one of the winner of the selection process in Public Key Encryption [23, 24].

The design of the architecture of LEDAcrypt/BIKE considers the parameters that affect both hardware complexity and latency in polynomial multiplication. These are the length of polynomials, the number of non-zero elements, and the ring/field on which the operations are carried on.

The algorithms known in the literature for polynomial product calculation are: the Schoolbook, the Karatsuba [25] and the Schönhage-Strassen [26] methods. Because the polynomial product in LEDAcrypt/BIKE involves very large binary polynomials, the two use the aforementioned methods as follows:

- The Schoolbook algorithm is adopted in implementations of schemes based on QC-LDPC/MDPC codes [27–29]. The results showed that the reduced area and latency of the implementation of this approach strongly depends on the density of the involved matrices: the latency is low for the sparse matrix in the decoder, it becomes rather poor when the density of the matrix is increased, as in the encoder.

- The Karatsuba algorithm achieves a the lowest latency, if used to compute the polynomial products required in the encryption [30].

- The Schönhage-Strassen algorithm [31], with the Number Theoretic transform (NTT), has the lowest time complexity among the algorithms with all tyoe of vectors. Therefore, has the flexibility to be applied to both encoder and decoder.

The thesis describes a Schoolbook architecture that implement polynomial product optimized for LEDAcrypt/BIKE encoder/decoder. In addition, the security of the Schoolbook approach is studied by applying a side-channel attack to retrieve the secret key. The encoder and decoder architectures are described and compared with state-of-the-art designs. Moreover, the Schönhage-Strassen approach, despite not being the most efficient solution, is more flexible among PQC primitives.

The work is organized as follows: in Chapter II, general concepts on cryptography are presented, while in Chapter III describes the Code-based PQC in details, in Chapter IV and V the algorithms for polynomial multiplication and the architectures are described, Chapter VI presents the implementation of the decoder and encoder and finally the Conclusions are provided.

Chapter 2

Cryptography

The security of private messages exchanged between two parties or the integrity of a received stream of data, are ensured by Cryptography. The field of Cryptography offers methods to establish secure communication channels, safeguarding secret information from third-party access, and expose potential threats that may compromise the reliability of communication.

To protect a secret message, this is transformed into an obfuscated text, known as *ciphertext*. The transformation of plaintext into ciphertext, represented in Figure 2.1, relies on specific cryptographic functions with provable security through mathematical theorems. Although in theory, the message could be derived from the ciphertext through brute force or known attacks, the computational cost of such tasks renders them infeasible given current and foreseeable computing capabilities.

The study of secure cryptosystem extends beyond Mathematics and encompasses contributions from Computer Science, Electronics and Physics (Quantum Mechanics) to ensure data protection. The methods and their implementations are continuously adapted into different requirements, and their security is regularly studied to find unexpected issues.

The methods, or protocols, used to secure information between two parties can be classified into two groups of algorithms: Symmetric Cryptography, which utilizes a single Secret Key for establishing a communication between trusted parties over a secure channel, and Asymmetric Cryptography, with a Secret Key and a Public Key, to establish communication between trusted parties over an insecure channel. It also provides a secure channel or prove the identity and integrity of a message or party

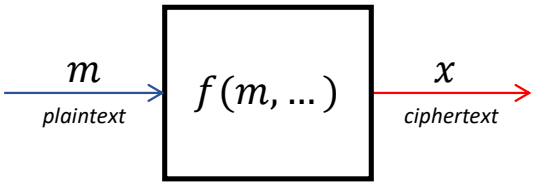


Fig. 2.1 Message ubfuscation by means of a cryptographic function

involved in the communication, together with Cryptographic Hash Functions. These methods evolved over time, with the increasing size of the keys implemented to align with the computational capabilities of computers. Recently, the advent of Quantum Computers and their potential to break the currently employed Asymmetric Schemes, has prompted research into new algorithms that can provide security in the coming decades. This has led to the emergence of Post-Quantum Cryptography.

This chapter aims to provide a comprehensive overview of the current standardized cryptographic methods that have been developed over the past decades and rely on mathematically proven secure problems. The impact of Quantum Computers on these algorithms will also be examined.

2.1 Asymmetric Cryptography

Asymmetric Cryptography, also known as Public-Key Cryptography (PKC), is a pivotal field in cryptography, as it enables the sharing of a common secret between two unknown parties, using a pair of Secret Keys (SK) and Public Keys (PK).

For example, the exchange of a secret message between Alice and Bob using PKC operates as follows: Bob wishes to share a secret with Alice, so he selects Alice's Public Key and encrypts the message, transforming it into ciphertext. Alice receives the ciphertext and decrypts it using her Secret Key to retrieve the message sent by Bob. The asymmetry in the use of the Secret Key and the Private Key is inherited from functions, these are computationally easy to solve in one direction, but difficult to invert.

The security of the communication is guaranteed by the asymmetry inherent in the selected mathematical function: the encryption of a message into a secret using the Public Key is "easy", while making it computationally difficult to invert

or decrypt without the Secret Key. As a result, disclosing the secret message is infeasible without the Secret Key. However, careful consideration must be given to the choice of the underlying problem, as the strength of the decryption process without the Secret Key can be compromised by the availability of more powerful or efficient methods.

The most common application of an Asymmetric Cryptosystem is to share the Secret Key of a Symmetric Systems, in order to establish a communication through a symmetric secure channel. In general, symmetric cryptosystems are more efficient due to their reduced latency and are referred to as Key Encapsulation Mechanism (KEM) schemes. The second main application is in Signature Schemes that are employed to prove the identity of the parties involved in a communication. The schemes that establish an asymmetric secure channel are referred as Public Key Encryption (PKE) schemes.

The schemes are a set of primitives that are in charge of generating the key pair (Key Generation). The block diagrams for KEM and PKC are in Figures 2.2 and 2.3. Encrypt and Decrypt for PKE, Encapsulate and Decapsulate for KEM and Sign and Verify for the Signature Schemes.

The above applications result in schemes that can build on algorithms that are resistant to classical or quantum threads. Modern Cryptography researches the state-of-the-art of methods that considers only classical threats, these come from classical computer (laptops) or supercomputers. More recently, the threads posed by Quantum Computers and quantum algorithms required an additional effort to find the algorithms and protocol that best suite to achieve security. In the following sections, the most common mathematical problems and the protocols are described in order to provide and overview of the cryptosystems.

The examples of mathematical problems that are the basis of PKC schemes are briefly described to have an overview of the different algorithms that are available. In addition, KEMs/PKCs and Signature scheme are detailed in order to understand how they apply a mathematical problem to encrypt messages.

Security level The security of a cryptosystem is given as *n-bit of security*. This is referred to the 2^n number of operations that an attacker has to perform, in order to retrieve the Secret Key from the Public Key. The security level is defined assuming that the best known attack is applied. The estimate is computed by different agencies

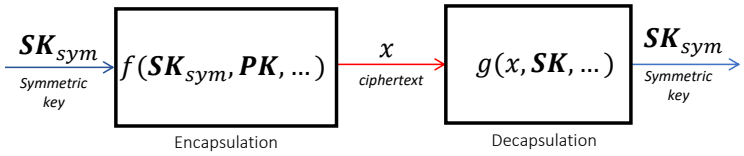


Fig. 2.2 Block Diagram of the Key Encapsulation Mechanism schemes

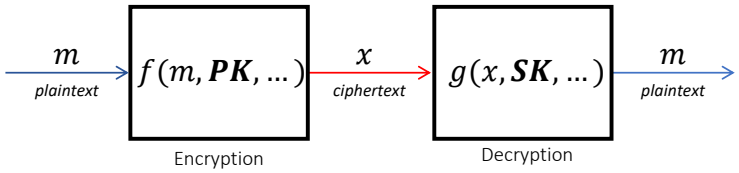


Fig. 2.3 Block Diagram of the Public Key Cryptography schemes

across the world and result in the suggested parameters of the selected scheme. The recommendations require to have at least 128 bit of security.

2.1.1 Modern Cryptography Schemes

In Modern Cryptography, Asymmetric schemes depend on two problems: the inefficiency of factoring a large number, which is the product of two large primes (RSA[32]) and the inefficiency to solve the discrete logarithm problem, referred as Elliptic Curve Cryptography[33, 34]. The functions has been adopted for decades to secure our data.

The methods that can recover the secret message from the ciphertext are inefficient to run on a classical computer, but methods to speed-up them up are known if a quantum computer with a sufficient number of bits is available.

RSA Cryptosystem

The RSA cryptosystem firstly appeared in 1977 and its name derived from the inventors of the scheme: Ron **R**ivest, Adi **S**hamir and Leonard **A**dleman[32].

In RSA, Alice would start the Key Generation, as mentioned above, with the extraction of two large prime numbers with different length, these are p and q, which has not to he disclosed. The product of these numbers, $n = pq$, is part of the Public Key (**PK**), which will be transmitted to Bob. The second number

of PK is e, this is selected such that $1 < e < \lambda(n)$ and coprime with $\lambda(n)$, with $l\lambda(n) = lcm(p-1, q-1)$ (*lcm* is the *least common multiple*). The value of SK is $d = e^{-1}$.

The Encryption, on Bob's side, has in input the secret message m that becomes the ciphertext $c = m^e \pmod{n}$.

The Decryption, which only Alice can perform, is $m = c^d \pmod{n}$.

The method is based on a specific group of functions referred as *trapdoor one-way*, which are very popular in PKC: the *one-way* property is useful in the encryption since it refers to a problem that is hard to invert, while the *trapdoor* is useful in the decryption because it can efficiently invert the function. Obviously, the trapdoor has to be only one otherwise the method is not suitable for cryptography.

Elliptic Curve Cryptography

The use of Elliptic Curve in cryptographic systems has been firstly studied in the 80's and then applied to existing schemes to build algorithms public key cryptography.

Elliptic Curves are described with the equation:

$$y^3 = x^3 + ax + b \tag{2.1}$$

The Elliptic Curve in Cryptography are defined over finite fields, thus, the equation becomes:

$$y^3 = x^3 + ax + b \; mod \; p \tag{2.2}$$

The representation of the elliptic curves changes when only the integer (*mod p*) coordinates are considered, Secret and Public Key, are points selected by Alice on such a curve. The search of the point associated to the Secret key is referred as the the discrete logarithm problem.

The advantage of ECC over RSA is that the size of the keys is reduced.

Elliptic Curve Diffie-Hellman The Diffie-Hellman Key Exchanged Protocol (DH)[35] paired with Elliptic Curves ensures that the secret cannot be found since it requires to solve the Discrete Logarithm Problem, there are no known methods

that can efficiently solve it. The idea is that Alice and Bob combines PK and SK and then derived a secret value which is the same between the two parties.

2.2 Symmetric Cryptography

The Symmetric Cryptography provides a set of algorithms that ensure a secure and efficient communication over a channel by adopting a single shared key, the Secret Key, in Figure 2.4.

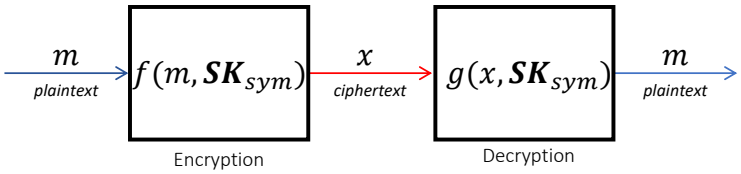


Fig. 2.4 Block Diagram of a Symmetric Cryptosystem

The communication between Alice and Bob consists in ciphering (encrypting) a secret message with SK and then de-cipher (decrypting) it with the same key. The algorithms available are divided into:

- Stream Ciphers, the encryption is applied digit by digit to the secret message;
- Block Ciphers, the encryption is applied to a block of digits of the secret message.

There is a number of algorithms that has been developed for symmetric cryptography, the most popular and adopted is the Advanced Encryption Standard (AES)[36].

The AES algorithm performs a set of operations that combines matrices with the secret message and matrices with the Secret Key to perform the encryption which are then reverted in order to decrypt the message.

The algorithm security is not affected by Quantum Computers since it is enough to double the size of the keys to have a equivalent security level on Quantum Computers.

2.3 Post-Quantum Cryptography

Shor's Algorithm[9] running on Quantum computers can break the RSA and ECC schemes, in such a way that is not enough to increase the size of the key to have a secure system. In Post-Quantum Cryptography researches schemes that are called *quantum-safe*, which guarantees the 'n-bit security' of RSA and ECC, but against quantum algorithms, such as Shor's[9] and Grover's[37] Algorithms.

The field of Post-Quantum Cryptography aims to provide algorithms for Asymmetric Cryptography and Signature scheme that relies on problems that are infeasible on both Classic and Quantum Computers.

The topic has become relevant in recent years, but quantum resistant schemes have been known for years ago and rediscovered recently. In 2015, the National Institutes for Standards and Technology (NIST)[38] launched a Call for Submission for Post-Quantum Schemes and different agencies across the world warned about the need of a transition towards quantum safe algorithms.

The selection process has encountered 4 Rounds and in the end selected a winner for the PKC/KEMs and Signature schemes.

Round 1, 2 and 3 The story of the NIST PQC Standardization process started with in 2015 with the call for proposal. The first set of submission for Round 1, which included in the competition 69 schemes [39], but a set of these has been broken in the very first days and hours after being made publicly available. In the next months, the remaining schemes has been carefully studied and a subset of them admitted to Round 2.

The PKE schemes admitted to Round 2 where 17 among the initial 69, further reduced to 9 at Round 3. The schemes admitted are based on four problems that were considered the most promising ones for PQC, one can find schemes based on Lattices, Code-based schemes, Isogeny based schemes and Multivariate ones. Despite that, in the end, the Isogeny-based and Multivariate-based turned out to be insecure, while Code-based and Lattice based schemes has been admitted to the last Round[14].

Round 4: selected algorithms and finalists The final phase of the competition is Round 4. The selected candidate to be standardized for KEMs has been found in Crystals-Kyber[40], while Dilithium[41], Falcon[42] and SPHINCS+[43] have been the selected as Signatures schemes[44]. The selection process is still open for both KEMs and Signature schemes, while for KEMs there are Code-based schemes, for signatures the proposals are not diversified, a specific competition has been launched [45].

The algorithms that are still running in the NIST competition for KEMs are:

- BIKE[18]: a Code-based scheme with Moderate Density Parity Check codes, this is reduced density variant of the McEliece Cryptosystem;
- Classic Mceliece[15]: a Code-based scheme that the original MceEliece Cryptosystem, with parameters which makes it quantum safe;
- HQC[19]: a Code-based cryptosystems that proposes a different model for code-based KEMs;
- SIKE [46]: a Isogeny-based cryptosystem which appeared to be a promising candidate, but has been found that is insecure by its inventors as states in the note published on NIST website [47].

The Code-based and Lattice based schemes are the oldest and most valid options for quantum safe algorithms, they have been both described decades ago with the McEliece Cryptosystems [13] and NTRU [48].

The Code-based cryptosystem history, details and implementations are discussed in the next chapter, in the following a brief description of Lattice-based schemes is provided.

2.3.1 Lattice-based Schemes

The Lattice-based schemes has been developed to consider two NP-hard problems that can be described on a given lattice, this is the set of points described over a n-dimensional space, the example is in Figure 2.5, which are linear combination of the base vectors, over this space one can define:

- the Shortest Vector Problems, aims to find the shortest vector on the lattice;

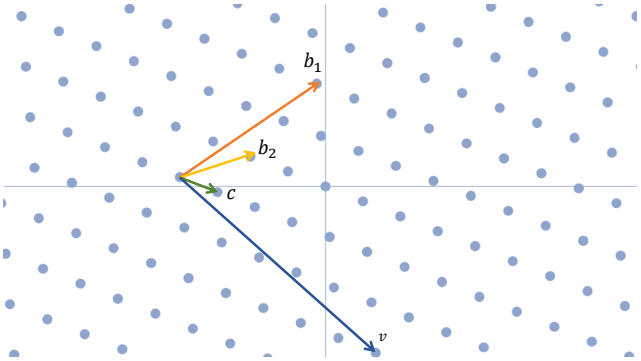


Fig. 2.5 The example of a two dimensional lattice with the basis, b_1 and b_2 basis vector highlighted and the shortest vector c,

- the Closest Vector Problem, aims to find in the lattices the closest vector to another vector.

The Shortest Vector Problem is the basis of Lattice-based cryptography, but recently a variant of the original problem has been developed, these are referred to as Learning With Error (LWE) schemes, since the proposal is the search of a vector on a lattice which has been perturbed by noise.

Despite the LWE schemes has been selected for standardization, the Code-based schemes are still considered as an alternative method to build a secure cryptosystem.

Code-based Schemes

The Code-based schemes are based on the NP-hard problem of decoding a general linear code. The NP-hardness has been adopted in the McEliece Cryptosystem, which has selected Goppa Codes to encrypt and decrypt the secret message. More recently, families of error correcting codes has been researched in order to lower the memory footprint of such schemes.

The detail of the Code-based schemes are described in Chapter III.

Chapter 3

Code-Based Post Quantum Cryptography

Error correcting codes, which are a tool in information theory, have been used for decades to transmit information over noisy channels by adding redundancy through encoding, which is later utilized by decoders to remove errors.

The first example of error correcting codes was proposed by Richard Hamming in the 1940s, the code developed is known as the Hamming code. Subsequently, several error correcting codes with varying structures, complexities, and correction capabilities have been developed, such as Reed-Solomon Codes, BCH Codes, Polar Codes, LDPC Codes, and Goppa Codes. These codes find wide applications in satellite communication, data storage, digital television protocols, and cryptography. More recently, error correcting codes have gained significance in securing data, with the rediscovery of McEliece Cryptosystems[49] based on Goppa Codes, and the successful application of LDPC[20] Codes in Quantum Key Distribution (QKD) protocols[50].

In the field of cryptography, the secret message, the *plaintext*, is encoded using a generator matrix into *codeword*, errors are intentionally introduced to obtain the *ciphertext*. The Decoding algorithm is then applied to recover the secret message. The security of such schemes is based on the NP-harness of the decoding a general linear code[12].

The oldest Code-based proposal iw the McEliece[49] and Niederreiter[51] Cryptosystems, and recently, variants of the method have been proposed and demonstrated promising results.

The description of Code-based Post-Quantum Cryptography (PQC) involves matrices, vectors, and notation specific to this chapter, which will be defined in dedicated sections in this and subsequent chapters.

3.1 Notations and definitions

The present section is dedicated to describe the notations and to define some concepts that are used in the present manuscript.

3.1.1 Constants, vector and matrices

The constants and integers are referred by letters a, vectors of size $1 \times m$ are defined in bold with $\mathbf{a}$ the row vector is:

$$\mathbf{a} = \begin{bmatrix} a_0 & a_1 & \dots & a_{m-1} \end{bmatrix} \tag{3.1}$$

the row vector has length m, the column vector, of size $m \times 1$,is the transpose of $\mathbf{a}^T$ is:

$$\mathbf{a}^T = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{bmatrix} \tag{3.2}$$

The elements of the vector are indexed starts from 0. The vector can be a binary vector, with elements a_i ($i \in [m-1, 0]$) that can be 0 or 1. The weight of a binary vector is a_w is the sum of the 1a in $\mathbf{a}$.

The $n \times m$ matrix is defined with capital and bold letter $\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \dots & a_{n-1,m-1} \end{bmatrix} \tag{3.3}$$

The square matrix has size $m \times m$, this is a circular matrix derived from vector $\mathbf{a}$:

$$\mathbf{A} = \begin{bmatrix} a_0 & a_1 & \vdots & a_{m-1} \\ a_{m-1} & a_0 & \vdots & a_{m-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \vdots & a_0 \end{bmatrix} \tag{3.4}$$

The transpose of the matrix is $\mathbf{A}^T$:

$$\mathbf{A}^T = \begin{bmatrix} a_0 & a_{m-1} & \vdots & a_1 \\ a_1 & a_0 & \vdots & a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1} & a_1 & \vdots & a_0 \end{bmatrix} \tag{3.5}$$

There is specific case of cyclic matrix, this is the *identity matrix*, it is referred as $\mathbf{I}$, it is a binary matrix with the first row $\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix}$, the matrix is:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \vdots & 0 \\ 0 & 1 & \vdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \vdots & 1 \end{bmatrix} \tag{3.6}$$

The binary matrices are associated to the weight of each column or row in matrix $\mathbf{A}$, these are a row vector, $\mathbf{a}_{row}$, and a column vector, $\mathbf{a}_{column}$,that counts the number of 1s in each column and rows. The binary cyclic matrix $\mathbf{A}$ is *regular* since the weight of rows and columns is constant, the row weight it referred as d_A, the density of the matrix (which is identical to a_w of vector $\mathbf{a}$).

The block matrices with $n_0 \times m_0$ are is defined by:

$$\mathbf{B} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \cdots & \mathbf{A}_{0,m_0-1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,m_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{n_0-1,0} & \mathbf{A}_{n_0-1,1} & \cdots & \mathbf{A}_{n_0-1,m_0-1} \end{bmatrix} \tag{3.7}$$

The size of $\mathbf{B}$, considering the $m \times m$ matrix $\mathbf{A}$, is $n_0 m \times m_0 m$. the block matrix can be 'cyclic' too, since the property can apply to the block matrices.

The block matrix with cyclic matrices is defined as *quasi-cyclic matrix*, the matrix is associated to a weight matrix $w(\mathbf{B})$ with each elements (i, j) that corresponds to the weight of each $\mathbf{A}_{i,j}$.

3.1.2 Polynomials

The polynomials are referred in a similar way as vectors, polynomial $\mathbf{a}(x)$ of degree m is:

$$\mathbf{a}(x) = a_0 + a_1 x^1 + \cdots + a_{m-1} x^{m-1} \tag{3.8}$$

The coefficients of $\mathbf{a}(x)$ are elements a_i, these corresponds to the elements of vector $\mathbf{a}$ of length m previously defined. Polynomials are referred by its coefficients.

3.1.3 Operators

addition or subtraction The addition (or subtraction) between constants, vectors and matrices of proper size is straightforward. The present work makes use of the sum between binary or integer vectors, the sum between vectors $\mathbf{a}$ and $\mathbf{b}$ is referred in two ways:

- $\mathbf{a} + \mathbf{b}$, the result is an integer vector it is performed in the integer domain;
- $\mathbf{a} \oplus \mathbf{b}$, the result is a binary vector it is performed in $GF(2)$

multiplication and correlation The multiplication (or product) between constants is $c = ab$, between vector is $\mathbf{c} = \mathbf{ab}$ is the element wise product among the elements of the vectors.

The matrix multiplication is referred to a matrix by matrix product and a vector by matrix product, the product considered are:

- the row vector $\mathbf{a}$ with size $1 \times m$ by the square cyclic matrix $\mathbf{B}$ (or $\mathbf{B}^T$) with size $m \times m$

$$\mathbf{c} = \mathbf{aB} \tag{3.9}$$

- the square cyclic matrix $\mathbf{B}$ (or $\mathbf{B}^T$) with size $m \times m$ by the column vector $\mathbf{a}^T$

$$\mathbf{c} = \mathbf{Ba} \tag{3.10}$$

The same notation is used for multiplication that result in integer or binary vectors, the domain of the output is specified.

The correlations, which is applied over two vectors is referred as: $\mathbf{a} \star \mathbf{b}$.

transforms The transform of a vector $\mathbf{a}$ is referred with the capital letter A (despite it is a vector it is not in bold to avoid confusion with matrix $\mathbf{A}$). For example, the Fourier transform of a signal store in vector $\mathbf{a}$, is:

$$A = \mathscr{F}(\mathbf{a}) \tag{3.11}$$

3.2 McEliece Cyrptosystem

The McEliece Cryptosystem was described by Robert McEliece in 1978 [49] based on binary Goppa Codes, the scheme is suitable to exchange a secret between two parties with an asymmetric cryptosystem, this is a Public-key Encryption scheme.

The cryptosystem, RSA[32] and ECC[34][33] has been proposed in 1977 and 1986, respectively. The RSA and ECC schemes have been selected as standards for asymmetric cryptography because of their efficiency, compared to McEliece. The overcoming thread of quantum computers has a milder effect on the McEliece cryptosystem due to the NP-hardness of the decoding of a general linear code. The

surprising aspect of McEliece cryptosystem is that since 1978 only one attack has been discovered which improves brute force attack, but with an increase of the size of the secret key the attack becomes infeasible.

The method establishes a secure channel between Alice and Bob, the general idea of the scheme is in Figure 3.1, the main elements are the Encryption and Decryption, these correspond to the encoder and decoder present in a telecommunication system, the difference is that the errors are part of the encryption, thanks to the redundancy of the codeword it is possible to recover the `plaintext`.

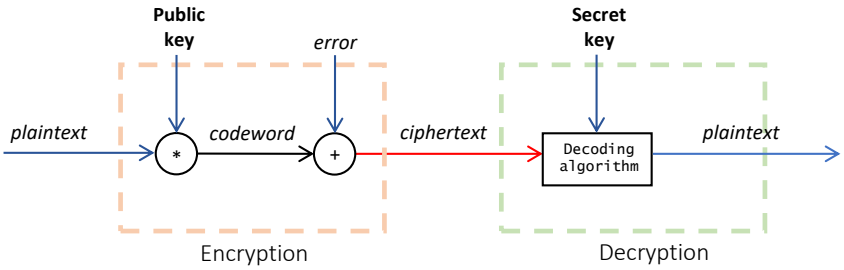


Fig. 3.1 Block scheme for McElice-like crytosystem

The original method consists in three algorithms that run in polynomial time: secret key pair generation, encryption, decryption.

The cryptosystem has a set of parameters for which the security is guaranteed, these are:

- n: the length of the code, which is the length of the redundant message,
- k: the number redundant bit added to the message,
- t: the weight of the error added to the redundant message.

3.2.1 Scheme algorithms

Key Pair generation The Secret Key (SK) is the binary Goppa Code capable of correcting t errors with and with dimensions n and k, the Public Key (PK) is the Generator Matrix, $\mathbf{G}$, the randomly selected $\mathbf{S}$ and $\mathbf{P}$ matrices which are the non-singular matrix with dimensions $k \times k$ and the permutation matrix with dimensions $n \times n$ respectively. The Private Key is $\hat{G} = SGP$.

The Public Key is shared by Alice to Bob.

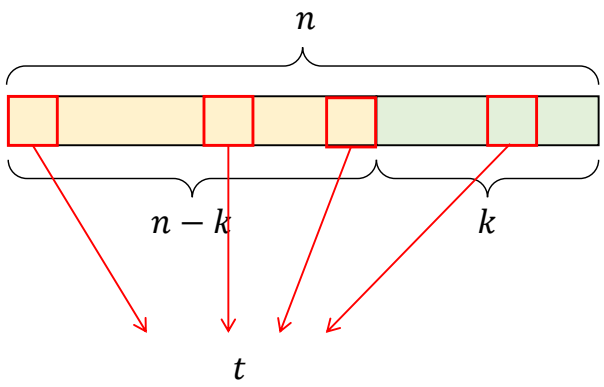


Fig. 3.2 The example of a codeword of length n with the different sections highlighted: the message is in yellow, with $n-k$ bit, the redundant part with k bits, the t error bit.

Encryption The secret message is $\mathbf{m}$ with length $n-k$, it is encrypt by Bob with the error vector with length n and t asserted bits is derived, the ciphertext is:

$$\mathbf{c} = \mathbf{m}\hat{\mathbf{G}} + \mathbf{e} \tag{3.12}$$

The ciphertext is a vector of length n that can be shared over an insecure channel.

Decryption The Decryption is performed by Alice in three steps:

- the inverses of $\mathbf{P}$ and $\mathbf{S}$ are computed;
- the decoding algorithm is applied to $\hat{\mathbf{c}} = \mathbf{c} * \mathbf{P}^{-1}$ and $\hat{\mathbf{m}}$ is obtained;
- the message is: $\mathbf{m} = \hat{\mathbf{m}}\mathbf{S}^{-1}$

The Decoder, with the parameters and algorithm, is designed to correct up to t errors, then the failures of the decoder (referred as DFR, Decoding Failure Rate) is guaranteed to be zero, this important to consider when designing a Code-based cryptosystem since the failures of the decoder are adopted by an attacker to retrieve information on the Secret Key.

3.2.2 Niederreiter Cyrptosystem

The Niederreiter Cryptosystem [51], with the general idea of the scheme in Figure 3.3, is variant of the McEliece Cryptosystem, the algorithm was originally based on

generalized Reed-Solomon Codes and has been broken in [52], but the cryptosystem is secure with binary Goppa Codes since they are both based on the NP-hardness of linear codes (the equivalence of the two scheme is described in [53]), thus the method is still valid if a suitable family of codes is employed. The original proposal is based on the parity-check matrix of a linear code and not the generator matrix, as in the original McEliece scheme.

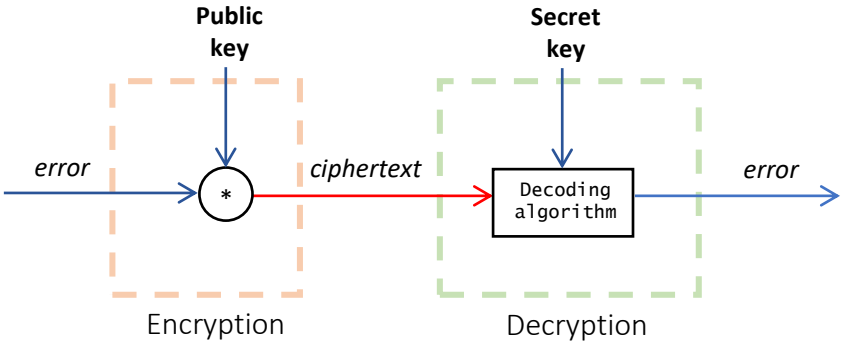


Fig. 3.3 Block scheme for Niederreiter-like crytosystem

The cryptosystem is defined by the code length, n, the redundancy, k, and the weight of the error vector, t.

Key Pair Generation The Secret Key is the parity check matrix of a Reed-Solomon error correcting code, $\mathbf{H}$, and the invertible matrix $\mathbf{S}$. The Public Key is matrix $\mathbf{H}' = \mathbf{HS}$.

Encryption The encryption is performed in two steps:

- the secret message is encoded into a vector of weight t, the error vector $\mathbf{e}$;
- the ciphertext is $\mathbf{c} = \mathbf{eH}'$, which is the *Syndrome* of the vector.

Decryption The decryption is carried on with a syndrome decoding algorithm that recovers $\mathbf{e}$ from $\mathbf{S}^{-1}\mathbf{c}$.

3.2.3 Limitations and attacks

The factor that limited the adoption of the McEliece cryptosystem, despite its outstanding features, is the dimensions of the keys, the value of the parameters in its original[49] form are: $n = 1024$ and $k = 524$ and $t = 50$, with the length of the message $n - k = 500$. The Generator Matrix has form $\hat{\mathbf{G}} = [\tilde{\mathbf{G}}, \mathbf{I}]$, which makes it a systematic code, thus matrix $\tilde{\mathbf{G}}$ has $524 \times (1024 - 524)$ binary values to be stored. The sizes are increased for the McEliece Cryptosystem adapted to modern standards, this is referred as *Classic McEliece*[54].

The choice of the parameters is fundamental to guarantee efficient decoding and encoding, but most important the security. The attacks applied on McEliece to derive the Secret key from the Public key are brute force kind of attacks and can be divided in two categories: syndrome decoding and information set decoding [55], the latter tries to find the low weight codewords and thus is able to reduce the complexity of the Secret key recovery. Recently, the parameters of McEliece Cryptosystem have been updated to make such algorithms infeasible, the proposal has been submitted to NIST PQC Standardization competition as Classic McEliece for PKE, while the Niederreiter scheme based on Goppa codes is considered for KEM.

The size of the key is a big limitation for McEliece adoption, thus new families of codes have been used to derive variants of McEliece, but this requires additional studies to discover new attacks and test the effect of know attacks to them. The most promising options are LDPC/MDPC codes since they seem not affected by structural attacks (as the original McEliece), while Reed Solomon codes turned out to be insecure.

3.3 Low-Density Parity Check Codes

The Low-Density Parity Check (LDPC) had described by Gallager in its doctoral thesis [20] with their encoding and decoding algorithm. The LDPC Codes have been rediscovered in the past decade thanks to their outstanding correcting capabilities.

The binary Low-Density Parity Check Codes are linear codes defined by two matrices: *Parity Check Matrix*, $\mathbf{H}$, with dimensions $n \times k$, and the *Generator Matrix*, $\mathbf{G}$.

The code is constructed with the *Parity Check Matrix*, a binary sparse matrix (low density) defined by a set of parity equations. The *Generator Matrix*, with dimensions $(n-k) \times n)$, encodes the input message, $\mathbf{m}$ with length $n-k$, into a *codeword*, $\mathbf{c}$ of length n.

$$\mathbf{c} = \mathbf{mG} \tag{3.13}$$

The *codeword*s of the code are the set of inputs that satisfy the parity check equation, the *Parity Check Matrix* is a linear combination of codewords.

The parameters of the LDPC code are:

- n, the length of the codeword;
- k, the number of redundant bit;
- t, the maximum number of errors.

The LDPC codes have additional elements that characterizes the *Parity Check Matrix*, this is d_c (or d_r) the weight of each column (or row), this is the number of asserted bit over a column (or row). In LDPC Codes the value of d_c (or d_r) is $<< n$.

Code Construction and Encoding The *Parity Check Matrix*, $\mathbf{H}$, is a $k \times n$ matrix generated randomly of with a construction algorithm. The *Generator Matrix* is obtained from $\mathbf{H}$ by rearranging it as:

$$\mathbf{H} = [\mathbf{A}, \mathbf{B}] \tag{3.14}$$

Where the sub matrix $\mathbf{A}$ is a non singular square matrix of size k, thus $\mathbf{B}$ is a matrix with size $k \times (n-k)$ then matrix $\mathbf{G}$ for a systematic code is $[\mathbf{I_{n-k}}, \mathbf{G}_l]$:

$$\mathbf{G_l} = [\mathbf{A}^{-1}\mathbf{B}]^T \tag{3.15}$$

The Identity Matrix, $\mathbf{I}$, is a square matrix of size $n-k$ and $\mathbf{G_l}$ is a $(n-k) \times k$ matrix.

The encoding, in details, is done as follows:

$$\mathbf{c} = \mathbf{m}[\mathbf{I}, \mathbf{G}_l] = [\mathbf{m}, \mathbf{mG}_l] \tag{3.16}$$

The encryption is an additional step in cryptosystems, since it requires to intentionally flip a set of bits in the codeword, making it a *ciphertext*.

$$\mathbf{x} = \mathbf{c} + \mathbf{e} \tag{3.17}$$

The codeword is encrypted into a ciphertext and the attacks that aims to decode it without the parity-check matrix are infeasible for the parameters of the code.

Decoding The codewords are the set of vector with length n that satisfy the parity equations of the code, with the errors added to the codeword the set of parity equations have a different value. The result of the equation is the *Syndrome*, $\mathbf{s}$, this quantity is fundamental in the decoder side since it is used to correct the errors.

The LDPC Codes have a wide number of decoders depending on the type of error that is introduced, the one employed in cryptography is Bit Flipping Decoder, modified version of the one proposed by Gallager, since the errors are bit flips of the codeword.

The Decoder with the Bit Flipping approach two variables have to be computed:

- The *Syndrome* vector is computed, this is a binary vector and each entry is the result of the parity equation of the code, the vector has length $n-k$;
- The *Unsatisfied Parity Check* vector is computed, this is an integer vector and it counts the number of wrong equations in each value of the ciphertext is involved.

The search of the error bit to be flipped is done by looking at the *Unsatisfied Parity Check* vector, **upc**, given a threshold value, the positions which are above that threshold are more likely to be errors.

The Algorithm iteratively approximated the ciphertext to the codeword by reaching a null *Syndrome*:

The evaluation of the threshold depends on the variant of the Bit Flipping algorithm that is adopted, the number of errors bits in the encryption step and the function that calculates the threshold at each iteration are critical parameters in the security analysis, thus in the coding design phase they have to be considered while testing attacks running on classic and quantum algorithm.

Algorithm 1 `Bit Flipping Decoder`

Input: ciphertex $\mathbf{x}$, vector with length n; :Parity Check Matrix $\mathbf{H}$, with size $k \times n$

Output: error free message $\mathbf{m}$, with size $(n-k)$

$It = 0$ ▷ Iterations counter
$\mathbf{s}^{(0)} = \mathbf{x}\mathbf{H}^T$ ▷ Syndrome
while $It < It_{max}$ **or** $\mathbf{s}^{It} = \mathbf{0}$ **do**
    $\mathbf{upc}^{It} = \mathbf{s}\mathbf{H}$ ▷ Unsatisfied Parity Check
    $b^{It} = f(\mathbf{s}^{It})$ ▷ Compute threshold
    $P = \{i \in [0, n-1] | \rho_i > b\}$ ▷ Collect the position of the errors
    $\mathbf{x}^{It+1}(P) = \mathbf{x}^{It}(P) \oplus \mathbf{c}^{It}(P)$ ▷ Flipping of error position
    $\mathbf{s}^{(It)} = \mathbf{x}^{It}\mathbf{H}^T$ ▷ Update Syndrome
    $It = It + 1$ ▷ Increase iterations counter
end while
if $\mathbf{s} = \mathbf{0}$ **then**
    return $\mathbf{m} = \mathbf{c}(1 : n-k)$ ▷ Decoding is successful
else
    Report failure ▷ Decoding has failed
end if

Coding Design The design of LDPC Codes suitable for cryptography has to be both secure and efficient, the parameters that are considered during the design are the result of a trade off between the efficiency of the encoding/decoding. The most critical parameters are the number of errors introduced in the codewords and the function to evaluate the threshold since on it the correcting capabilities of the decoder are affected: it is fundamental to correct the errors in It_{max} iterations at most and avoid decoding failures since the decoding failures give additional information to discover the parity-check matrix from the generator matrix. The constraints imposed on the coding design are different for PKEs and KEMs, the latter generates ephemeral keys used only once, while PKE has long term keys, thus a statistical analysis that exploits failures of the decoder or analysis of the power consumption/execution time of the primitives can be applied.

The LDPC may have an additional structure that reduces the encoding and decoding complexity, without deteriorating the security [18][17]: the Quasi-Cyclic construction is adopted to further reduce the memory requirements of the scheme.

In cryptographic applications, impressively low error rates (in the order of 2^{-128} or less) must be achieved to avoid some types of attacks that gain information on the

secret key. The design of the codes, with its parameters and decoder, is fundamental to reaching this requirement and avoid a phenomenon known as 'error floor' [56, 57].

3.3.1 Structured Codes: QC-LDPC/MDPC

The variants of the McEliece and Niederreiter cryptosystem proposed for Post-Quantum Cryptography are based on QC-LDPC/MDPC Codes, Quasi-Cyclic Low-Density Parity-Check/Moderate-density Parity-Check Codes.

The Quasi-Cyclic matrices in LDPC Codes are referred to the structure of the Parity-check and generator matrix. The cyclic or (circulant) matrices are square matrices, the whole matrix is defined with the first row (or column) that are cyclically shifted to obtain the following row (or columns), the binary matrix $\mathbf{A}$ with size $m \times m$:

$$\mathbf{A} = \begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_{m-1} \\ a_{m-1} & a_0 & a_1 & \dots & a_{m-2} \\ a_{m-2} & a_{m-1} & a_0 & \dots & a_{m-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & a_3 & \dots & a_0 \end{bmatrix} \tag{3.18}$$

The matrix is regular since the weight is constant across column (and rows), this is referred as w_A, the cyclic structures allows to stored only the first column $\begin{bmatrix} a_0 & a_1 & a_2 & \dots & a_{m-1} \end{bmatrix}$.

The parity-check matrix, $\mathbf{H}$, is a block matrix which is assumed to have $x \times y$ blocks:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{0,0} & \mathbf{H}_{0,1} & \mathbf{H}_{0,2} & \dots & \mathbf{H}_{0,y-1} \\ \mathbf{H}_{1,0} & \mathbf{H}_{1,1} & \mathbf{H}_{1,2} & \dots & \mathbf{H}_{1,y-1} \\ \mathbf{H}_{2,0} & \mathbf{H}_{2,1} & \mathbf{H}_{2,2} & \dots & \mathbf{H}_{2,y-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{H}_{x-1,0} & \mathbf{H}_{0,0} & \mathbf{H}_{0,0} & \dots & \mathbf{H}_{x-1,y-1} \end{bmatrix} \tag{3.19}$$

The elements $H_{i,j}$ is cyclic, thus $\mathbf{H}$ is **quasi-cyclic** (QC), the knowledge of the firs row (or column) is enough to derive $\mathbf{H}$.

The additional interesting property of quasi-cyclic codes, as we will see in the next section, is its isomorphism with the polynomial ring $\mathbb{F}_2[x]/(x^m + 1)$. The matrix $\mathbf{A}$ and the polynomials $a(x)$ with coefficients in $\mathbb{F}_2$ (binary coefficients) are

equivalent:

$$a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1} x^{m-1} \tag{3.20}$$

The property is important to improve in the computation and it successfully applied in modern cryptosystems. The key generation involves a matrix inversion, with the parameters of the cryptosystem it is simplified into a polynomial inversion, the multiplications in the encoder and decoder are considered as polynomial multiplication and thus the techniques known in literature are adopted to speed-up the computation.

The cryptosystems that have been proposed for Code-based Post-quantum Cryptography are BIKE and LEDAcrypt, they are variants of McEliece with QC-LDPC/MDPC Codes adopted to implement Public Key Encryption (PKE) and Key Encapsulation Mechanisms (KEM) schemes.

3.4 Code-based QC-LDPC schemes

The NIST competition reached its final phases of the standardization process, with Round 4, BIKE is still running to be a possible candidate for standardization, while LEDAcrypt has been found vulnerable and did not made it to Round 3 of the standardization process. The two schemes are very similar, but not identical. The primitives and parameters of the schemes are described in the following and their specifications are reported.

The primitive that implements the Key Generation, Encryption and Decryption for PKE, Encapsulation and Decapsulation for KEMs, the methods are similar to the algorithms explained before but the polynomial structure, the security requires some additional elements to take care of.

The key pair are the Parity-Check matrix and Generator matrix of a QC-LDPC/MDPC code, the parameters are the number of blocks, n_0, the size of the blocks, p, the density of the matrix as the number of asserted bits in the first row, d_h, and the number of errors, t. The code length and size of the message are derived from the previous parameters.

3.4.1 LEDAcrypt

The LEDAcrypt parameters are in Tables 3.1 and 3.2, these are referred to KEM primitives submitted to Round 2 of the NIST competition, the parameters are grouped per level of security.

The parity-check matrix in LEDAcrypt is matrix **L**, a quasi-cyclic matrix, the result of the product **HQ**, which are again quasi-cyclic matrices. The cyclic matrices are defined as:

- $$\mathbf{H} = [\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_{n_0-1}], \tag{3.21}$$

  The matrix has n_0 blocks, the density of the cyclic blocks is constant and set to d_H, which refers to the number of asserted bit in the first row of the matrix.

  The matrix is calculated randomly extracting a set of d_H distinct positions for each n_0 block.

- $$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{0,1} & \cdots & \mathbf{Q}_{0,n_0-1} \\ \mathbf{Q}_{1,0} & \mathbf{Q}_{1,1} & \cdots & \mathbf{Q}_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{Q}_{n_0-1,0} & \mathbf{Q}_{n_0-1,1} & \cdots & \mathbf{Q}_{n_0-1,n_0-1} \end{bmatrix}. \tag{3.22}$$

  The matrix has $n_0 \times n_0$ blocks.

The matrix **Q** is associated to its *density matrix*, which is quasi-cyclic itself, the matrix is:

$$\mathbf{D}_(Q) = \begin{bmatrix} d_0 & d_1 & \ldots & d_{n_0-1} \\ d_{n_0-1} & d_0 & \ldots & d_{n_0-2} \\ \vdots & \vdots & \ddots & \vdots \\ d_1 & d_{n_0-1} & \ldots & d_0 \end{bmatrix} \tag{3.23}$$

The first row is $\begin{bmatrix} d_0 & d_1 & \ldots & d_{n_0-1} \end{bmatrix}$, referred as $\mathbf{d}_Q$, the sum of these elements is referred as d_Q.

The matrix is calculated by extracting $\mathbf{D}_(Q)(i, j)$ distinct position for the $\mathbf{Q}(i, j)$ block, for each (i, j) block with $i = 0 \ldots n_0 - 1$ and $j = 0 \ldots n_0 - 1$.

Decoder Efficiency and Security concerns The idea to split the parity-check matrix into **H** and **Q** was proposed in Round 1 of the LEDAcrypt submission, this for an efficient decoder: the cyclic matrix products are the core of the computation and, as it will be shown later, is the execution time of such multiplication strictly depends on the density of the matrices, with the 'factorized' version of the parity-check it is possible to compute the products with a reduced density. Unfortunately, the optimization showed weaknesses from the security point of view and thus has been dropped out from the NIST competition. The choice of the updated parameters is close to the one selected in BIKE. The details on LEDAcrypt are reported in the following since the initial study on the architectural implementation has been carried on considering LEDAcrypt, but can be extended to BIKE by changing the sizes of the parameters.

Parameters

The value of n_0 for a given Security Level can be 2, 3 and 4, which is associated to different code length ($n_0 p$) and redundancy ($(n_0 - 1)p$), despite the security being the same the length of the message to be encoded and the execution time of the decoder, can change, this allows to have a decryption with different performances but a given level of security.

The parameters referred to the submission to Round 2 of LEDAcypt for KEMs are in Table 3.1, the maximum number of iterations of the decoder is set to 5 (It_{max} in Algorithm 1).

The parameters referred to the submission to Round 3 of LEDAcypt for PKE and KEMs are in Table 3.2, the parity check matrix is **H** with its density d_H.

It is important to point out that the size of the block is a prime number, this choice is fundamental to have an efficient generation of the keys, but requires particular attention when implementing an efficient decoder and encoder.

The list of parameters is provided since these have been considered in the first implementation of the Decoder, the following parameters considered are derived from BIKE.

Table 3.1 Considered LEDAcrypt parameters for KEM.

Security Level (in bits)	n_0	m	d_H	$\mathbf{d}_Q$	t
128	2	14,939	11	$[4,3]$	136
	3	8,269	9	$[4,3,2]$	86
	4	7,547	13	$[2,2,2,1]$	69
192	2	25,693	13	$[5,3]$	199
	3	16,067	11	$[4,4,3]$	127
	4	14,341	15	$[3,2,2,2]$	101
256	2	36,877	11	$[7,6]$	267
	3	27,437	15	$[4,4,3]$	169
	4	22,691	13	$[4,3,3,3]$	134

LEDAkem

The Key Encapsulation mechanism for Code-based PQC is the Niederreiter cryptosystem with QC-LDPC Codes. The scheme is in charge of encapsulating a secret string, which is later used as the key of a Symmetric Cryptosystem. The Secret Key is the Parity-Check Matrix of a QC-LDPC Code, derived as $\mathbf{L} = \mathbf{HQ}$, and the Public Key is the Generator Matrix of a QC-LDPC Code.

Key Generation The key generation has to calculate firstly the Parity-Check Matrix $\mathbf{L}$, given the parameters n_0, m and d_H/$\mathbf{d}_Q$, the steps are:

- Calculate the asserted position in each n_0 block for matrix $\mathbf{H}$, this corresponds to calculating d_H distinct position for each n_0 block;
- Calculate the asserted position in each $n_0 \times n_0$ block for matrix $\mathbf{Q}$, this corresponds to calculating $\mathbf{d}_Q$ distinct position for each block;
- Calculate the product $\mathbf{HQ}$.

The result is the Quasi-Cyclic matrix $\mathbf{L} = [\mathbf{L}_0, \mathbf{L}_1, \ldots, \mathbf{L}_{n_0-1}]$, which has density $d_H * d_Q$.

The Generator Matrix, $\mathbf{G}$,is calculated in three steps:

Table 3.2 Considered LEDAcrypt parameters for KEM and PKE submitted for Round 3.

Security Level (in bits)	n_0	m	d_H	t	It_{max}
128	2	10,853	71	133	6
	3	8,237	79	84	5
	4	7,187	83	67	4
192	2	20,981	103	198	6
	3	15,331	117	125	5
	4	13,109	113	99	4
256	2	35,117	137	263	4
	3	25,579	155	166	4
	4	21,611	163	132	4

- Computation of the inverse of a block in $\mathbf{L}$

$$\mathbf{L}_{inv} = \mathbf{L}_{n_0-1}^{-1} \tag{3.24}$$

- The matrix $\mathbf{M}_l$ is calculated

$$\mathbf{M}_l = \mathbf{L}_{\mathbf{inv}} * [\mathbf{L}_0, \mathbf{L}_1, \ldots, \mathbf{L}_{n_0-2}] \tag{3.25}$$

  to obtain a $(m \times mn_0 - 2)$ matrix

$$\mathbf{M}_l = [\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_{n_0-2}] \tag{3.26}$$

- Finally, the matrix $\mathbf{G} = [\mathbf{M}_l, \mathbf{I}]$, with identity matrix $m \times m$.

The Secret key is $\mathbf{L}$, the Public Key is $\mathbf{G}$.

Encapsulation The encapsulation is in charge of encoding the a secret into an error vector, $\mathbf{e}$, with length mn_0 and t asserted bits. The encryption computes the syndrome, $\mathbf{s}$:

$$\mathbf{s} = \mathbf{e}\mathbf{G}^T \tag{3.27}$$

Decapsulation The decapsulation is in charge of recovering the secret $\mathbf{e}$ with the Secret Key, this is achieved with:

- the value $\mathbf{s}' = \mathbf{L}_{n_0-1}\mathbf{s}$;
- the decoder subroutine, with $\mathbf{L}$ and $\mathbf{s}'$, computes $\mathbf{e}$

LEDApkc

The Public Key Cryptography scheme with Code-based PQC is the McEliece cryptosystem with QC-LDPC Codes. The scheme is in charge of encrypting a secret string. The Secret key is the Generator Matrix of a QC-LDPC Code and the Public Key is the Parity-check Matrix of a QC-LDPC Code, $\mathbf{L} = \mathbf{HQ}$.

The algorithms of the scheme are very similar to the ones of LEDAkem.

Key Generation The Key Generation is carried on as in LEDAkem, the Secret Key is the Parity Check matrix $\mathbf{L} = \mathbf{HQ}$, the Public Key is derived from $\mathbf{M}_l$ by computing:

$$\mathbf{G} = [\mathbf{I}, \mathbf{M}_l^T] \tag{3.28}$$

The dimension of the identity matrix is $m(n_0-1) \times m(n_0-1)$. The Generator Matrix has systematic form.

Encryption The encryption of a secret message $\mathbf{m}$, with length $m(n_0-1)$, is carried out by computing the codeword:

$$\mathbf{c} = [\mathbf{m}, \mathbf{m}\mathbf{M}_l^T] \tag{3.29}$$

the Encryption is the sum of the codeword and an error vector $\mathbf{e}$ of weight t:

$$\mathbf{x} = \mathbf{c} + \mathbf{e} \tag{3.30}$$

Decryption The Decryption is in charge of retrieving the secret vector $\mathbf{m}$. The same decoder subroutine of LEDAkem is run to remove the errors added to $\mathbf{c}$, the input is the secret vector $\mathbf{L}$ and the Syndrome $\mathbf{s} = \mathbf{x}\mathbf{H}^T$.

Algorithm 2 Q-Decoder

Input: syndrome $\mathbf{s}$, with length $m(n_0-1)$, parity-check matrix $\mathbf{H}$, with size $m(n_0-1)\times mn_0$,
sparse matrix $\mathbf{Q}$,with size $mn_0 \times mn_0$, maximum number of iterations It_{max}

Output: estimated error $\mathbf{e}$ or decoding result

$It = 0$ ▷ Iteration count
$\mathbf{s}^{(0)} = \mathbf{s}$ ▷ Save the initial syndrome in $\mathbf{s}^{(0)}$
$\mathbf{e} = \mathbf{0}$, $\tilde{\mathbf{e}} = \mathbf{0}$ ▷ approximated error $n_0 m$
while $It < It_{max}$ **or** $\mathbf{s} = \mathbf{0}$ **do**
    $\sigma = \mathbf{s}^T\mathbf{H}$
    $\rho = [\rho_0, \ldots, \rho_{n-1}] = \sigma\mathbf{Q}$
    $b = f(\mathbf{s})$ ▷ Compute threshold
    $P = \{i \in [0, n-1] | \rho_i > b\}$
    for $i \in P$ **do**
        $e_i = e_i \oplus 1$ ▷ Update error vector estimate
        $\tilde{\mathbf{e}} = \tilde{\mathbf{e}} + \mathbf{q}_i$
    end for
    $\mathbf{s} = \mathbf{s}^{(0)} + \mathbf{H}\tilde{\mathbf{e}}^T$ ▷ Update syndrome
    $It = It + 1$ ▷ Increase iteration count
end while
if $\mathbf{s} = \mathbf{0}$ **then**
    return $\mathbf{e}$ ▷ Decoding success
else
    Report failure ▷ Decoding failure
end if

Q-Decoder

The Decoder applied in Round 2 submission of LEDAcrypt is referred as Q-Decoder since it fully exploits the presence of $\mathbf{H}$ and $\mathbf{Q}$. The Algorithm is in 2, the inputs are the Syndrome, $\mathbf{s}$, and the Parity-Check Matrix, $\mathbf{H}$. The Decoder is a Bit Flipping Decoder where the main quantities to be computed are the *Syndrome* and *Unsatisfied Parity Check* that allows to compute the error vector applied to the codeword. The presence of $\mathbf{H}$ and $\mathbf{Q}$ is employed to compute the *Unsatisfied Parity Check* in two steps and it is referred to as ρ, instead of **upc**.

The function $f(\mathbf{s})$, which evaluates the Threshold b, is performed with the use of a Look Up table, provided as a parameter of LEDAcrypt. The LUT is structures as follows, the value of the Syndrome weight points to a specific value of the threshold.

The optimal values and the methods to derive the table have been widely described in the documentation of the scheme [17].

The decoder submitted to Round 3 computes $\rho = \mathbf{s}^T\mathbf{L}$.

3.4.2 BIKE

The BIKE scheme is a Key Encapsulation Mechanism based on QC-MDPC Codes, the proposal successfully made to Round 4 and it is a possible candidate for the standardization.

The scheme is Niederreiter-like, thus the secret key is encrypted into a syndrome, similarly to LEDAcrypt there are three main algorithms, which are reported below in their core part, that cryptosystem and a set of parameters that defines the code length, density of the matrix and number of error.

The parity-check matrix is characterized by matrix $\mathbf{H} = [\mathbf{H}_0, \mathbf{H}_1]$, which is a quasi-cyclic matrix with two blocks, the blocks of the matrix are referred by its first row, then as vectors $\mathbf{h}_0$ and $\mathbf{h}_1$.

Parameters

The set of parameters submitted to Round 4 is in Table 3.3.

Table 3.3 BIKE parameters submitted to Round 4 and Round 3.

Security Level (in bits)	n_0	m	d_H	t	It_{max}
128	2	12,323	142	134	5
192	2	24,206	206	198	5
256	2	40,973	274	264	5

Key Generation The Secret Key and Public Key are derived as follows:

- the Secret Key consists in the vectors $\mathbf{h}_0$ and $\mathbf{h}_1$, these are derived by extracting d_H distinct position of each vector;

- the Public Key is derived as $\mathbf{h} = \mathbf{h}_1\mathbf{h}_0^{-1}$, the inverse and the multiplication are performed to the polynomials associated to $\mathbf{h}_i$.

It is worth to mention that the choice of m as a prime number reduces the complexity of the polynomial inversion of $\mathbf{h}_1^{-1}$, the same property is applied to the matrix inversion (that has in general complexity $O(n^3)$) to have an efficient inversion.

Encapsulation The encapsulation starts by encoding the secret $\mathbf{m}$ in the error vector $\mathbf{e} = [\mathbf{e}_0, \mathbf{e}_1]$, with weight t. The cipertext is then obtained with:

$$\mathbf{c}_0 = \mathbf{e}_0 + \mathbf{e}_1\mathbf{h} \tag{3.31}$$

The result involves again a polynomial multiplication, where the polynomials are associated to vectors $\mathbf{e}$ and $\mathbf{h}$.

The ciphertext in the original description contains a term added by the has function, this is skipped here since it is not part of the study which has been carried on.

Decapsulation The decapsulation has in input the ciphertext and the Secret Key, the main part of the primitives applied the decoding subroutine to the ciphertext and retrieves the error vector and then the secret $\mathbf{m}$.

the decoding algorithm is the Black-Gray-Flip.

BlackGrayFlip (BGF) Decoder

The decapsulation employs a Decoder referred as BlackGrayFlip (BGF) Decoder with parameters from Table 3.3, this has been initially proposed in BIKE documentation that can be found in [18] with 'shades' of the errors instead of a single threshold, and applied to BIKE scheme. The Decoder is in charge of computing $\mathbf{e}$, this is carried out by means of two support binary vectors referred as **black** and **gray** with length $2m$.

The decoder is an iterative bit flipping decoder, which computes the syndrome, error approximation and syndrome update, the difference is in the evaluation of the threshold and the presence of a 'gray' are in the threshold, this is issued on the first

iteration of the decoder which include a further mask in the decoder. The algorithms have additional subroutines which are defined: `ctr(`$\mathbf{H}$`,`$\mathbf{s}$`,`j`)`, `ctr(`$\mathbf{H}$`,`$\mathbf{s}$`,`j`)`, this computes each i element of **upc** and the syndrome updated and error approximation are implicit in the definition; `threshold(`$\mathbf{s}, i$`)`, the function is a parameter of the scheme and evaluates the threshold based on the syndrome weight and the value of the iteration; functions `BFIter` and `BFMaskedIter` are defined in the following.

The algorithm in its original form from BIKE documentation appears different from a Bit Flipping Decoder, it is important to mention that the relevant computations performed in such a decoder are presented here, the Syndrome is updated with $\mathbf{s}+\mathbf{e}\mathbf{H}^t$ and Unsatisfied Parity Check is computed with the function `ctr()`, despite it is not clearly expressed they are computed as polynomial multiplication.

LEDA and BIKE: differences and similarities

The LEDAcrypt and BIKE Cryptosystems, despite being both low-density variants of the McEliece Cryptosystem, possess some differences that are summarized in the following:

- BIKE adopts QC-MDPC Codes, while LEDAcryp adopts QC-LDPC Codes: the structure of the matrices is Quasi-Cyclic, but MDPC Codes are slightly more dense then LDPC Codes. This different existed up to Round 2, while in Round 3 the Secret Key of LEDAcrypt and BIKE have similar densities.
- In LEDAcrypt more options of block lengths are proposed, introducing a further flexibility in the cryptosystem.
- The Decryption is based for both on a Bit-Flipping Decoder, but BIKE introduces a 'shade' in the type of error that is detected introducing two thresholds, while the Q-Decoder the threshold of the error detection is only one.

The similarity is in the choice of the length of polynomials, thus the blocks of the Secret Key, they have both selected length equal to prime numbers, thus comparing the Security Levels in Table 3.2 and 3.3 the value of m is similar.

3.5 Arithmetic in Code-based PQC

The two main proposal that has been submitted for Code-based PQC scheme with QC-MDPC codes has been widely described in the previous section. The scope of the present work is to provide the implementation of such primitives, in particular of the Encoder (Encryption/Encapsulation) and Decoder (Decryption/Decapsulation).

The calculation that involves QC-MDPC codes are carried on by applying the arithmetic of polynomials, with sum, polynomial inversion and polynomial multiplication. The sum between Polynomials is a straightforward operation, while Polynomial multiplication and inversion are the most time consuming section of the computation.

The Polynomial multiplication and the proposal of an efficient decoder/encoder is provided in the next chapters, while efficient algorithms for polynomial inversions have been listed in the documentation of BIKE and LEDAcrypt.

In the following, the steps of the schemes are reported withe the arithmetic unit required.

3.5.1 Polynomial Sum

The Polynomial Sum or the sum between two vectors is performed as an element wise sum between the elements of the two inputs. The operation is performed for:

- Codeword Encryption with $\mathbf{c}+\mathbf{e}$;
- Syndrome Update with $\mathbf{e}\mathbf{H}^T$

The sum $\mathbf{c}=\mathbf{a}+\mathbf{b}$ with vectors of length m that corresponds to $\mathbf{a}=\begin{bmatrix}a_0 & a_1 & a_2 & \dots & a_{m-1}\end{bmatrix}$ and $\mathbf{b}=\begin{bmatrix}b_0 & b_1 & b_2 & \dots & b_{m-1}\end{bmatrix}$ is:

$$\mathbf{c}=\begin{bmatrix}b_0+a_0 & b_1+a_1 & b_2+a_2 & \dots & b_{m-1}+a_{m-1}\end{bmatrix} \tag{3.32}$$

The sum between b_i+a_i is a binary or integer sum.

3.5.2 Polynomial Inversion

The Polynomial inversion is $\mathbf{h}_0^{-1}$, in BIKE, and the cyclic matrix inversion *LInv*, in LEDA. The step is fundamental to computing the key pair in McEliece or Niederreter schemes.

The efficient computation of the inverse and the determinants is crucial to have an efficient generation of the keys. The computation of determinants and inverses with the conventional methods is avoided thanks to particular choices of block sizes.

The matrix inversion in LEDAcrypt is computed as a polynomial inversion for QC-LDPC Codes by considering the isomorphism between circulant matrices and finite fields. The choice of m as a prime number simplifies the computation of the inverse, which is performed with Euclid's Algorithm. The Key Generation in BIKE is performed with the algorithm described in the documentation of [18].

3.5.3 Polynomial Multiplication

The polynomial multiplication is employed both in the encoder a decoder to perform multiplication that involves a multiplication with a quasi-cyclic matrix and polynomial multiplication. In general, three methods are known in literature to efficiently perform such products, the detail of these algorithms is provided in the next chapter.

Algorithm 3 BlackGrayFlip (BGF) Decoder

Input: syndrome $\mathbf{s}$, with length m, parity-check matrix $\mathbf{H}$, with size $m \times 2m$, maximum number of iterations It_{max}, grey threshold τ

Output: estimated error $\mathbf{e}$ or decoding result

$\mathbf{e} = \mathbf{0}$ ▷ Initialization of the error vector
for $i = 0, \ldots, It_{max}$ **do**
  $b =$ `threshold`$(\mathbf{s} + \mathbf{e}\mathbf{H}^T, i)$
  $[e, black, gray] =$ `BFIter`$(\mathbf{s} + \mathbf{e}\mathbf{H}^T, \mathbf{e}, b, \mathbf{H})$
  if $i = 0$ **then**
    $\mathbf{e} =$ `BFMaskedIter`$(\mathbf{s} + \mathbf{e}\mathbf{H}^T, \mathbf{e}, \mathbf{black}, (d_H/2+1)/2+1, \mathbf{H}))$
    $\mathbf{e} =$ `BFMaskedIter`$(\mathbf{s} + \mathbf{e}\mathbf{H}^T, \mathbf{e}, \mathbf{gray}, (d_H/2+1)/2+1, \mathbf{H}))$
  end if
end for
if $\mathbf{s} = \mathbf{e}\mathbf{H}^T$ **then**
  return $\mathbf{e}$ ▷ Decoding success
else
  Report failure ▷ Decoding failure
end if

`function BFIter`$(\mathbf{s}, \mathbf{e}, b, \mathbf{H})$
for $j = 1, \ldots, 2m-1$ **do**
  if `ctr`$(\mathbf{H}, \mathbf{s}, j) \geq b$ **then**
    $e_j = e_j \oplus 1$
    $\mathbf{black}_j = 1$
  end if
  if `ctr`$(\mathbf{H}, \mathbf{s}, j) \geq b - \tau$ **then**
    $\mathbf{gray}_j = 1$
  end if
end for

`function BFMaskedIter`$(\mathbf{s}, \mathbf{e}, \mathbf{mask}, b, \mathbf{H})$
for $j = 1, \ldots, 2m-1$ **do**
  if `ctr`$(\mathbf{H}, \mathbf{s}, j) \geq b$ **then**
    $e_j = e_j \oplus \mathbf{mask}_j$
  end if
end for

Chapter 4

Fast Polynomial Multiplication

The chapter is dedicated to the algorithms known in literature to compute Polynomial Multiplication, this is widely present in cryptographic primitives for Post Quantum Cryptography and in Code-based PQC is particularly interesting due to the isomorphism between cyclic matrices, with block size $m \times m$, and polynomial ring $\mathbb{F}_2[x]/(x^m+1)$, such products involved large values of m has binary coefficients, thus require a particular attention. The description of the most known algorithms for polynomial multiplication is provided in the following, then the comparison among the various methods are provided considering their effect on Code-based Post-Quantum Cryptography, where the specific characteristics of the variables are considered to find the best solution.

The description provides the time complexity of each algorithm, the detailed description of the hardware complexity is extensively described in the next chapter. Despite that, some consideration on this aspect can be made before looking at the final results.

The algorithms that are described are the Schoolbook, Karastuba, Toom-Cook and Schönhage Stressen algorithms.

4.1 Polynomial Multiplication Overview

The Multiplication Algorithms for integer and polynomials have been widely studied in the past years for large integer multiplication, which are common in cryptosystems

such as RSA. The main feature to define the effectiveness of an algorithm is the Time Complexity, which is the number of single digit addition/multiplication given the length of two operands. The Time Complexity metric is defined with the Big-O notation: gives the order of magnitude of such operation as function of the length of the polynomial. The Space Complexity, which measures the cost in terms of memory, thus the support variable to store during the computation.

The polynomial multiplication is as a carry-less integer multiplication. The equivalence between polynomials, integer, cyclic matrices is shown, together with matrix multiplication and convolution.

4.1.1 Schoolbook Multiplication

The Schoolbook multiplier is the simplest way to perform a multiplication, with integers it is:

$$\begin{array}{cccc} & 1 & 1 & 4 & \times \\ & & 5 & 2 & \\ \hline & 2 & 2 & 8 & \\ 5 & 6 & 0 & & \\ \hline 5 & 8 & 2 & 8 & \end{array} \tag{4.1}$$

The expression complicates for polynomial multiplication. The degree n polynomials $\mathbf{a}(x) = a_0 + a_1x^1 + \cdots + a_{n-1}x^{n-1}$ and $\mathbf{b}(x) = b_0 + b_1x^1 + \cdots + b_{n-1}x^{n-1}$, results in a polynomial $\mathbf{c}(x)$ with degree $2n-1$. The value of $c(x)$, with the Schoolbook method, is:

$$\begin{array}{ccccccccc} & & & a_{n-1}x^{n-1} & \dots & \dots & a_1x^1 & a_0x^0 & \times \\ & & & b_{n-1}x^{n-1} & \dots & \dots & b_1x^1 & b_0x^0 & \\ \hline & & & b_0a_{n-1}x^{n-1} & \dots & \dots & b_0a_1x^1 & b_0a_0x^0 & \\ & & b_1a_{n-1}x^n & \dots & \dots & b_1a_1x^2 & b_1a_0x^1 & \ddots & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & & \\ b_{n-1}a_{n-1}x^{2n-1} & \dots & b_{n-1}a_1x^n & b_{n-1}a_0x^{n-1} & & & & & \\ \hline c_{2n-1}x^{2n-1} & \dots & c_nx^n & c_{n-1}x^{n-1} & \dots & c_2x^2 & c_1x^1 & c_0x^0 & \end{array} \tag{4.2}$$

The similarities between polynomials $\mathbf{a}(x)$ and $\mathbf{b}(x)$ and vectors $\mathbf{a}$ and $\mathbf{b}$, defines the convolution between a vectors $\mathbf{a} = [a_{n-1}, \dots, a_1, a_0]$ and $\mathbf{b} = [a_{n-1}, \dots, b_1, b_0]$

is equivalent to the polynomial multiplication in 4.2. Then, the result is **c** equivalent to $\mathbf{c}(x)$, $c = [c_{2n-1},\ \ldots,\ c_1,\ c_0]$.

In the description, we obtain a set of n polynomials referred as $\mathbf{pp}_i = b_i x^i \times \mathbf{a}$, with $i \in 0,\ \ldots,\ n-1$, these the partial products of the multiplication.

The cyclic convolution between vectors **a** and **b**, is a vector **c** with length n. The result is evaluated as:

$$\begin{array}{cccccc|c}
a_{n-1} & a_{n-2} & \ldots & a_2 & a_1 & a_0 & \times \\
b_{n-1} & b_{n-2} & \ldots & b_2 & b_1 & b_0 & \\
\hline
b_0 a_{n-1} & b_0 a_{n-2} & \ldots & b_0 a_2 & b_0 a_1 & b_0 a_0 & \\
b_1 a_{n-2} & b_1 a_{n-3} & \ldots & b_1 a_1 & b_1 a_0 & b_1 a_{n-1} & \\
\vdots & \vdots & \iddots & \vdots & \vdots & \vdots & \\
b_{n-1} a_0 & b_{n-1} a_{n-1} & \ldots & b_{n-1} a_3 & b_{n-1} a_2 & b_{n-1} a_1 & \\
\hline
c_{n-1} & c_{n-2} & \ldots & c_2 & c_1 & c_0 &
\end{array} \tag{4.3}$$

The result is equivalent to the product among a vector **a** and a cyclic matrix associated to vector **b**. The cyclic matrix, as described in the previous chapter, is **B**:

$$\mathbf{B} = \begin{bmatrix}
b_0 & b_1 & \ldots & b_{n-1} \\
b_{n-1} & b_0 & \ldots & b_{n-2} \\
\vdots & \vdots & \ddots & \vdots \\
b_1 & b_2 & \ldots & b_0
\end{bmatrix} \tag{4.4}$$

The result **aB** is the vector **c** in (4.3).

The Schoolbook multiplier evaluates the result by computing each $a_i b_j$, with $i \in (0,\ldots,n-1)$ and $j \in (0,\ldots,n-1)$, and by adding each element to obtain the desired c_k, with $k \in (0,\ldots,n-1)$ for the cyclic convolution or $k \in (0,\ldots,2n-1)$ for the linear convolution.

The complexity of the method is the number of integer multiplications that are going to be computed, in Schoolbook algorithm the complexity is: $O(n^2)$.

The Schoolbook Algorithm has the highest Time Complexity, but the Space Complexity in terms of memory is $O(n)$.

4.1.2 Karatsuba Multiplication

The Karatsuba Algorithm is an improvement of the Schoolbook method. The Time Complexity is reduced to $O(n^{log_2 3}) \approx O(n^{1.585})$, while the Space complexity is $O(n)$.

The multiplication between polynomials $\mathbf{a}(x)$ and $\mathbf{b}(x)$ with length n is computed by firstly rewriting the two factors.

$$\begin{cases} \mathbf{a}(x) = (a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots a_{\lceil n/2 \rceil}x^{\lceil n/2 \rceil}) + \\ \qquad\qquad + (a_{\lceil n/2 \rceil - 1}x^{\lceil n/2 \rceil - 1} + a_{\lceil n/2 \rceil - 2}x^{\lceil n/2 \rceil - 2} + \dots a_0 x^0) \\ \mathbf{b}(x) = (b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \dots b_{\lceil n/2 \rceil}x^{\lceil n/2 \rceil}) + \\ \qquad\qquad + (b_{\lceil n/2 \rceil - 1}x^{\lceil n/2 \rceil - 1} + b_{\lceil n/2 \rceil - 2}x^{\lceil n/2 \rceil - 2} + \dots b_0 x^0) \end{cases} \tag{4.5}$$

and then the polynomials are split in two intermediate polynomials, $\mathbf{a}_0(x)$, $\mathbf{a}_1(x)$ and $\mathbf{b}_0(x)$, $\mathbf{b}_1(x)$:

$$\begin{cases} \mathbf{a}(x) = (a_{n-1}x^{\lceil n/2 \rceil - 1} + a_{n-2}x^{\lceil n/2 \rceil - 2} + \dots a_{\lceil n/2 \rceil}x^{\lceil 0 \rceil}) \cdot x^{\lceil n/2 \rceil} + \\ \qquad\qquad + (a_{\lceil n/2 \rceil - 1}x^{\lceil n/2 \rceil - 1} + a_{\lceil n/2 \rceil - 2}x^{\lceil n/2 \rceil - 2} + \dots a_0 x^0) \\ \mathbf{b}(x) = (b_{n-1}x^{\lceil n/2 \rceil - 1} + b_{n-2}x^{\lceil n/2 \rceil - 2} + \dots b_{\lceil n/2 \rceil}x^0) \cdot x^{\lceil n/2 \rceil} + \\ \qquad\qquad + (b_{\lceil n/2 \rceil - 1}x^{\lceil n/2 \rceil - 1} + b_{\lceil n/2 \rceil - 2}x^{\lceil n/2 \rceil - 2} + \dots b_0 x^0) \end{cases} \tag{4.6}$$

The final form is:

$$\begin{cases} \mathbf{a}(x) = \mathbf{a}_1(x) \cdot x^{\lceil n/2 \rceil} + \mathbf{a}_0(x) \\ \mathbf{b}(x) = \mathbf{b}_1(x) \cdot x^{\lceil n/2 \rceil} + \mathbf{b}_0(x) \end{cases} \tag{4.7}$$

The product is:

$$\begin{cases} \mathbf{c}(x) = \mathbf{a}_1(x)\mathbf{b}_1(x) \cdot x^n + \\ \qquad\qquad + (\mathbf{a}_1(x)\mathbf{b}_0(x) + \mathbf{a}_0(x)\mathbf{b}_1(x)) \cdot x^{\lceil n/2 \rceil} + \\ \qquad\qquad\qquad + \mathbf{a}_0(x)\mathbf{b}_0(x) \end{cases} \tag{4.8}$$

The further improvement reduces two multiplications, $\mathbf{a}_1(x)\mathbf{b}_0(x) + \mathbf{a}_0(x)\mathbf{b}_1(x)$, to:

$$\mathbf{a}_1(x)\mathbf{b}_0(x) + \mathbf{a}_0(x)\mathbf{b}_1(x) = (\mathbf{a}_0 + \mathbf{a}_1)(\mathbf{b}_0 + \mathbf{b}_1) - \mathbf{a}_0(x)\mathbf{b}_0(x) - \mathbf{a}_1(x)\mathbf{b}_1(x) \quad (4.9)$$

The method has a recursive form that computes the three multiplication applying Karatsuba method.

The asymptotic Time Complexity with n is $O(n^{log_2 3})$ and a Space Complexity of $O(n)$.

Toom-Cook Generalization

The Toom-Cook multiplication is a generalized formulation of the Karatsuba Algorithm. The method is referred as Toom-k, where k is referred to the number of 'splits' of the polynomials **a** and **b**, Karatsuba can be considered Toom-2. The additional split of the polynomials can further reduce the number of multiplications.

The Time Complexity of the method is $O(n^{log(2k-1)/log(k)})$, for 'small' k, with Toom-3 the result is $O(n^{1.46})$. The increase in the k-splits introduces an overhead due to the increasing number of additions, which makes the method less efficient. Moreover, due to its complexity to implement Toom-Cook with bigger k, it is preferable to select methods that have a lower Time Complexity.

4.1.3 Schönhage Stressen Multiplication

The Schönhage Stressen Algorithm is for its lowest Time Complexity, it is $O(nlog(n) * log(log(n)))$. The reduced complexity is obtained by taking advantage of the FFT efficiency. The method for the multiplication of two polynomials $\mathbf{a}(x)$ and $\mathbf{b}(x)$ of degree n. The resulting coefficients ($\mathbf{c}(x)$) of multiplication, as it is clear from Equation (4.2), are the result of a convolution among the two factors. Then, by recalling the properties of the Fourier Transform, the convolution in time domain becomes an element-wise multiplication in frequency domain. The Fourier Transform is computed with the *DFT* (*Discrete Fourier Transform*) algorithm and the inverse transform is computed with the *DFT* (*Inverse Discrete Fourier Transform*) algorithm.

Then, the polynomial multiplication is computed as:

$$\mathbf{c}(x) = IDFT(DFT(\mathbf{a}(x)) \cdot DFT(\mathbf{b}(x))) \tag{4.10}$$

The present formulation of the problem does not introduce an improvement since the Time Complexity is $O(n^2)$, due to use of the *DFT*. The method is known for the adoption of the *Fast Fourier Transform*, *FFT*. This strongly reduces the cost of the direct and inverse transforms. The equations are:

$$\mathbf{c}(x) = IFFT(FFT(\mathbf{a}(x)) \cdot FFT(\mathbf{b}(x))) \tag{4.11}$$

The *Fast Fourier Transform* algorithms is a method to compute the *Fourier Transform* with Time Complexity $O(nlog(n)log(log(n)))$.

The Schönhage Stressen method involves the use of the *Number Theoretic Transform* (*NTT*), the equivalent of *Fourier Transform* in finite fields. The use of the *NTT* is motivated by the loss of precision when computing the *FFT*.

4.2 Polynomial Multiplication with QC-MDPC/LDPC Matrices

The multiplication in Code-based Post Quantum Cryptography is involved in the Encryption and Decryption primitives, with long binary/integer vector cyclic convolution with dense or sparse binary matrices with the circular structure in (4.4). In the following, the Time Complexity of the multipliers is evaluated considering the implementation of the algorithms in Post-Quantum primitives, where the algorithms are adapted to efficiently compute the variables in Encryption/Decryption.

4.2.1 Large and Binary/Integer Polynomial Multiplication

The cyclic multiplication in LEDAcrypt/BIKE involves a vector and a matrix, their block size is $n \times 1$ and $n \times n$ respectively, since $n \approx 10,000$ the Schönhage Stressen algorithm has the lowest time complexity.

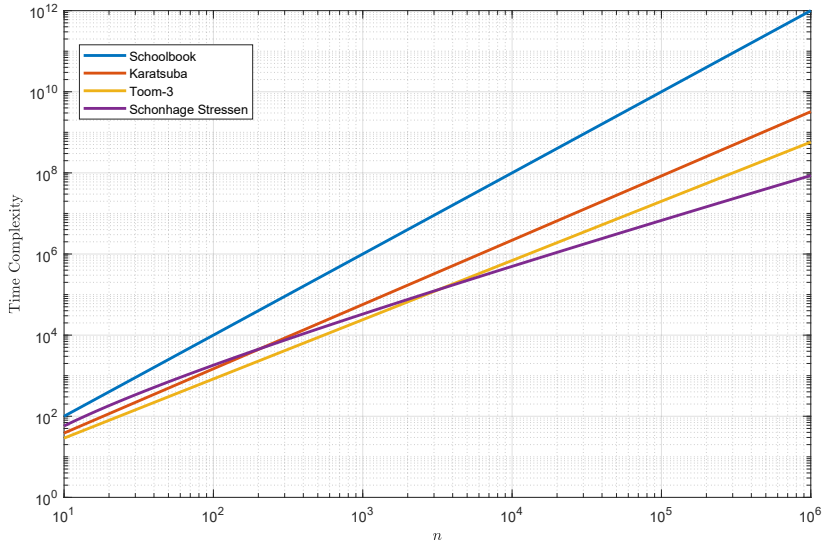


Fig. 4.1 Time Complexity

In the Encryption in Public Key Cryptography has $n_0 - 1$ multiplication between a vector of length n and a circulant matrix with size $n \times n$, this is performed in $GF(2)$.

The Encapsulation and the LDPC/MDPC Decoding in Decryption/Encryption have an additional feature, the multiplication is between a sparse vector (or circulant matrix) and a dense vector, the result is binary in the Encapsulation and Syndrome evaluation (in BF-Decoding) and integer in the Unsatisfied Parity Check evaluation (in BF-Decoding).

The comparison of the Time Complexity in Figure 4.1 clearly shows that the Schönhage Stressen Algorithm is suggested for the variables size in LEDAcrypt/BIKE, since the multiplication can have sparse factors, the simplification that it introduces in Schoolbook multiplier is described in the following.

The Time Complexity and Space Complexity provided for each algorithm do not include the implementation cost of each algorithm, this is measured in terms of gates that are required. The discussion on the implementation is provided in the following sections and it will be more clear that the one with the lowest 'Gate Complexity' is the Schoolbook multiplier. For this reason, the Schoolbook algorithm is applied to the convolution among vectors, with one or more input factor which is very sparse.

4.2.2 Sparse Polynomial multiplication in Schoolbook Multiplier

The Schoolbook multiplier for cyclic convolution is in Equation (4.3). The equation can be strongly simplified, less elements to compute, in case very few elements in **a** or **b** are non-zero, say that a_h and b_h are the number of non-zero elements in **a** and **b** respectively, and the vectors are binary.

The multiplication $n = 9$ and **b** sparse. The result with factors

$\mathbf{a} = [a_8,\ a_7,\ a_6,\ a_5,\ a_4,\ a_3,\ a_2,\ a_1,\ a_0]$ and $\mathbf{b} = [\,0,\ 0,\ b_6,\ 0,\ 0,\ 0,\ 0,\ b_1,\ b_0]$

is:

$$
\begin{array}{ccccccccc|c}
a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & \times \\
0 & 0 & b_6 & 0 & 0 & 0 & 0 & b_1 & b_0 & \\
\hline
b_0a_8 & b_0a_7 & b_0a_6 & b_0a_5 & b_0a_4 & b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 & \\
b_1a_7 & b_1a_6 & b_1a_5 & b_1a_4 & b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & b_1a_8 & \\
b_6a_2 & b_6a_1 & b_6a_0 & b_6a_8 & b_6a_7 & b_6a_6 & b_6a_5 & b_6a_4 & b_6a_3 & \\
\hline
c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 &
\end{array}
\tag{4.12}
$$

The number of elements to compute is strongly reduced, since the number of multiplication and addition becomes: $C_{mpy}/C_{add} = n * b_h$.

The additional simplification is that the sparse vector is in general a binary vector for our application, then to compute the result there is no need of the multiplier and the result **c** is the sum of cyclic shifts of the input vector **a**:

$$
\begin{array}{ccccccccc|c}
a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & \times \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & \\
\hline
a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & \\
a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & a_8 & \\
a_2 & a_1 & a_0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & \\
\hline
c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 &
\end{array}
\tag{4.13}
$$

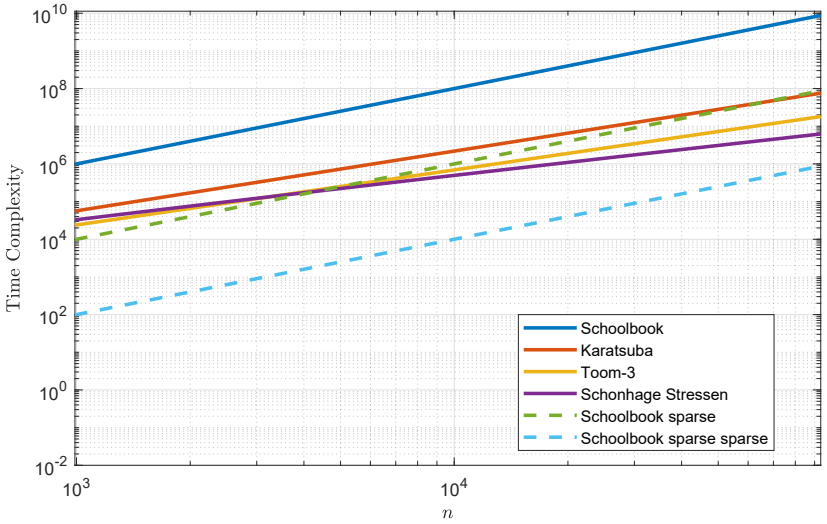


Fig. 4.2 Time Complexity with sparse Schoolbook

The reduction applies to the Decoder in LEDAcrypt/BIKE, since the multiplications involves a very sparse binary cyclic matrix, the result has to be both a binary vector and a integer vector.

The following condition that has to be handled, has a sparse vector too. Again, the simplified result assuming that $\mathbf{a} = [a_8, 0, 0, 0, 0, a_3, a_2, 0, a_0]$, the vector $\mathbf{c}$ is:

$$\begin{array}{ccccccccc|c}
a_8 & 0 & 0 & 0 & 0 & a_3 & a_2 & 0 & a_0 & \times \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & \\
\hline
a_8 & 0 & 0 & 0 & 0 & a_3 & a_2 & 0 & a_0 & \\
0 & 0 & 0 & 0 & a_3 & a_2 & 0 & a_0 & a_8 & \\
a_2 & 0 & a_0 & a_8 & 0 & 0 & 0 & 0 & a_3 & \\
\hline
c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 &
\end{array} \tag{4.14}$$

The number of sums is proportional to $a_h \times b_h$.

The Time Complexity reduction introduced by sparse inputs, is compared to the previous algorithm. Considering that 'sparse' means a_h and $b_h << n$, in the comparison a_h and $b_h \approx n/100$.

The Schoolbook complexity reduction with sparse inputs is not detailed for the Toom-Cook and Karatsuba Algorithms due to their cost to implement them, as it will

be shown later it can be more convenient to adopt the Schönhage Stressen, despite its complexity, since it has, overall, the lowest Time Complexity.

Chapter 5

Architectures for Large Polynomial Multiplication

The Polynomial Multiplication is a fundamental step in Code-based PQC proposal submitted to NIST competition, the present chapter is dedicated to describe the architectures that computes and accelerates the large polynomial multiplication required by BIKE and LEDAcrypt. The peculiar case of QC-MDPC Codes requires an extensive architectural exploration to find a good balance between total area and execution time.

The present chapter describes two architectures that implements Schoolbook and Schönhage Stressen algorithm for binary and large polynomial multiplication. The Schoolbook algorithm is implemented and compared to designs in charge of computing the multipliers that implements the same primitives, in addition the Schönhage Stressen method has been implemented to provide a flexible multiplier that can be applied in more PQC primitives. The multipliers are tailored for Code-based PQC. The speed-up introduced exploits the sparsity of the binary vectors that are present in the MDPC/LDPC matrices. In the end, the gate complexity, presented in terms of area occupation (ASIC) or resource utilization (FPGA), and execution time are compared with the state of art of polynomial multipliers and polynomial multipliers in PQC.

The description provided in the following reports the area and execution time of Schoolbook and Schönhage Stressen for the parameters in LEDAcrypt[17], up

to Round 2, and BIKE[18], up to Round 4, Public Key Encryption (PKE) and Key Encapsulation Mechanism (KEM) schemes.

Parameters of the multiplier The polynomial multiplication is the equivalent of a vector by cyclic matrix (or cyclic matrix by vector), it is performed in the Encryption/Encapsulation and Decryption/Decapsulation primitives of BIKE/LEDAcrypt, with vectors (polynomials) of length n reported in Table 5.1 for Code-based PQC with QC-MDPC codes submitted to NIST Competition during Round 2. The vector (and first row or column of the matrix) is assumed to be dense or sparse, the sparsity is referred as the number of asserted position over n, this is labelled as d_v in the Table. The last parameters is the number or errors introduced, t_e, which is the second measure of 'sparsity' of the vectors that are multiplied. The first column is referred to the security category, with NIST parameters, of the lengths provided.

Table 5.1 QC-MDPC Code-based PQC crytopsystems parameters.

	LEDAcrypt[17]			BIKE[18]		
NIST Cat.	n	d_v	t_e	N	d_v	t_e
1	21,701	71	130	12,323	119	134
3	37,813	103	196	24,659	206	199
5	58,171	137	262	40,973	274	264

The multiplier has to handle a series of multiplication, $c = \mathbf{a} \times C(\mathbf{b})$, these are the following:

- dense binary vector by sparse cyclic binary matrix with binary result;
- dense binary vector by sparse cyclic binary matrix with integer result
- sparse binary vector by sparse cyclic binary matrix with binary result;

The vector is $\mathbf{a}$, the binary cyclic matrix is $C(\mathbf{b})$ and the result is $\mathbf{c}$.

The result of execution time, area occupation/resource utilization, is reported for different values of n and the three cases of multipliers.

5.1 Schoolbook architecture

The Schoolbook multiplier is widely acknowledged as the algorithm with the lowest gate complexity for binary inputs, owing to its straightforward implementation, basic control of operation, and minimal hardware requirements (i.e., an accumulator and a register). The literature on the implementations for large (but not as large as in BIKE/LEDAcrypt) binary multiplication is limited, but in [58] is is showed that the lowest complexity is achieved by combining different multipliers with Schoolbook multiplier, despite that the present architecture considers a specific case of multiplication (with one term very sparse), thus the Schoolbook approach has been preferred from the beginning and then compared to the state of art of multipliers for binary polynomial product.

The Schoolbook algorithm for computing the vector by binary sparse cyclic multiplication, as described in Chapter 4, is referred to as `VectorByCirculant` and `SparseVectorByCirculant` for sparse binary vectors and binary sparse cyclic multiplication with binary/integer and binary results.

The design of an efficient architecture for computing $\mathbf{c} = \mathbf{a} \cdot C(\mathbf{b})$ requires consideration not only of the cost in terms of area and execution time, but also of the need for implementation that is immune to side-channel attacks. Such attacks exploit information on the secret key that may leak from the execution time or power consumption of the device. Achieving immunity to side-channel attacks requires a rigorous design exploration to identify and address any potential vulnerabilities in the architecture.

5.1.1 VectorByCirculant

The `VectorByCicrculant` (`VbC`) multiplier computes the product among a large binary vector of length n and a cyclic sparse matrix with size $n \times n$, with b_h asserted bit is each row with a Schoolbook-like algorithm. The idea is to exploit the cyclic structure of the matrix, the same approach has been applied in [59] with a Time Constant software implementation. In the following the algorithm and then the hardware implementation are going to be described.

The multiplication that is going to be addressed is $\mathbf{a} \cdot C(\mathbf{b})$. The algorithm is derived considering that $C(\mathbf{b})$ is a sparse binary matrix. The result $\mathbf{c}$ for vectors of

length $n = 5$ for generic $\mathbf{a}$ and $\mathbf{b}$ is:

$$\begin{aligned} c_0 &= b_0a_0 + b_1a_4 + b_2a_3 + b_2a_3 + b_1a_4, \\ c_1 &= b_0a_1 + b_1a_0 + b_2a_4 + b_3a_3 + b_4a_2, \\ c_2 &= b_0a_2 + b_1a_1 + b_2a_0 + b_3a_4 + b_4a_3, \\ c_3 &= b_0a_3 + b_1a_3 + b_2a_4 + b_3a_0 + b_4a_1, \\ c_4 &= b_0a_4 + b_1a_3 + b_2a_2 + b_3a_1 + b_4a_0. \end{aligned} \tag{5.1}$$

The result is simplified for sparse and binary $C(\mathbf{b})$, with b_0 and b_4 as asserted element (they are equal to 0), the vector $\mathbf{c}$ is:

$$\begin{aligned} c_0 &= a_0 + a_4, \\ c_1 &= a_1 + a_2, \\ c_2 &= a_2 + a_3, \\ c_3 &= a_3 + a_1, \\ c_4 &= a_4 + a_0. \end{aligned} \tag{5.2}$$

It is clear that to compute the result it is enough to sum together shifted versions of $\mathbf{a}$.

The formal description of `VectorByCirculant` is in Algorithm 4 and 5, the result is updated by deriving shifted versions of the input $\mathbf{a}^i$ by the amount i given by the asserted positions in $C(\mathbf{b})$.

Algorithm 4 `VectorbyCirculant (VbC) (binary)`

Input: length-n vector $\mathbf{a}$, weight of $C(\mathbf{b})$ (interpreted as the weight of each row and column) $b_h \in [0, n-1]$, first row of $C(\mathbf{b})$ represented as a list of positions S_b with size b_h

Output: length-n binary vector $\mathbf{c} = \mathbf{a}C(\mathbf{b})$

1: $\mathbf{c} = \mathbf{0}$ ▷ Initialized as null vector with length n
2: **for** $i = 0$ **to** $b_h - 1$ **do**
3: $k = S_b(i)$
4: $\mathbf{a}^{(i)} = [a_k, a_{k+1}, \cdots, a_{p-1}, a_0, a_1, \cdots, a_{k-1}]$
5: $\mathbf{c} = \mathbf{c} \oplus \mathbf{a}^{(i)}$ ▷ new partial product xored with $\mathbf{c}$
6: **end for**
7: **return c**

Algorithm 5 `VectorbyCirculant (VbC) (integer)`

Input: length-n vector $\mathbf{a}$, weight of $C(\mathbf{b})$ (interpreted as the weight of each row and column) $b_h \in [0, n-1]$, first row of $C(\mathbf{b})$ represented as a list of positions S_b with size b_h

Output: length-n integer vector $\mathbf{c} = \mathbf{a}C(\mathbf{b})$

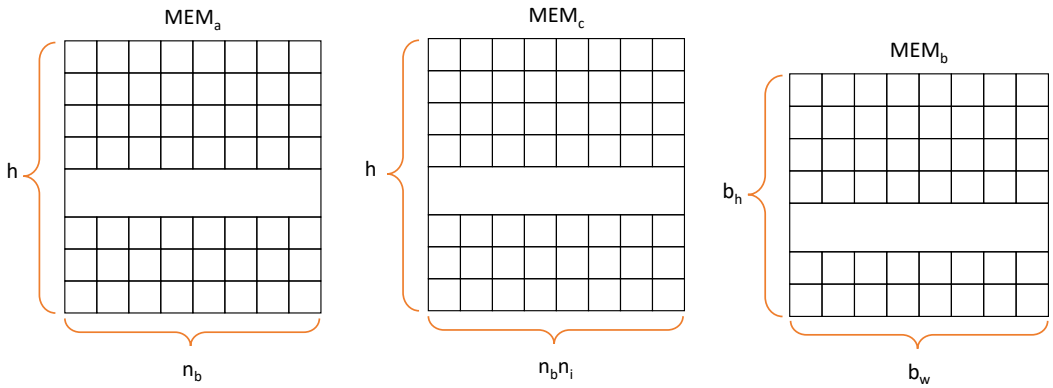
1: $\mathbf{c} = \mathbf{0}$ ▷ Initialized as null vector with length n
2: **for** $i = 0$ **to** $b_h - 1$ **do**
3: $\quad k = S_b(i)$
4: $\quad \mathbf{a}^{(i)} = [a_k, a_{k+1}, \cdots, a_{p-1}, a_0, a_1, \cdots, a_{k-1}]$
5: $\quad \mathbf{c} = \mathbf{c} + \mathbf{a}^{(i)}$ ▷ new partial product summed with $\mathbf{c}$
6: **end for**
7: **return c**

The computation in example (5.2) is then performed by computing $\mathbf{a}^{(1)} = [a_0, a_1, a_2, a_3, a_4]$ and summing it to $\mathbf{c}$, then the second vector $\mathbf{a}^{(4)} = [a_4, a_2, a_3, a_1, a_0]$ is derived and summed or xored again to updates $\mathbf{c}$.

The vector, $\mathbf{a}$, is stored as it is, a binary sequence of 1 and 0 or (integers). The matrix has a specific format, due to the prohibitive sizes of $n \times n$5.1 does not fit the width of the memories available in commercial FPGAs and for ASIC modules. The sparsity of the cyclic matrix reduces the memory needs, the b_h asserted positions of the first column $C(\mathbf{b})$, then in $\mathbf{b}$ are stored.

Memories The vectors $\mathbf{a}$ and $\mathbf{c}$ are binary and integer, respectively. They have a matrix format with a width (in bits) that is power of 2. Depending on the available memories and the desired execution time the vectors $\mathbf{a}$ and $\mathbf{c}$ are stored with a matrix format that matches the size of the selected memory, the size is referred as $h \times n_b$, with n_b the width of the memory and h its height that corresponds to $h = \lceil n/n_b \rceil$. The width of the memory is n_b for binary vector, for integer vectors represented on n_i bit the total width is $n_b * n_i$. The memory is addressed as $\texttt{MEM}_{\mathbf{a}}$ and $\texttt{MEM}_{\mathbf{c}}$ for vectors $\mathbf{a}$ and $\mathbf{c}$ respectively. The values of n_b, since it is a power of 2, is assumed equal to [8, 16, 32, 64, 128, 256], while n_i is equal to log_{b_h} since it is the width of the integer resulting vector, thus it is assumed that all the 1s over a row of the result are summed and then it is at most b_h.

The memories are asyncronous in reading, while synchronous in writing. These are modeled as single or dual port SRAMs in the design.



(a) $\texttt{MEM}_\mathbf{a}$ (b) $\texttt{MEM}_\mathbf{c}$ (c) $\texttt{MEM}_\mathbf{b}$

Fig. 5.1 Memories format with the sizes specified.

The cyclic matrix $C(\mathbf{b})$, is stored as a list of asserted position in the first column of the matrix in order to reduce the memory occupation since the matrix is binary and sparse for LEDAcrypt/BIKE decoder. The memory is referred as $\texttt{MEM}_\mathbf{b}$, the positions stored can be 0 up to n, thus the width of the memory is $b_w = log_2(n)$ and its height is b_h. The binary values of the i-th position point the first value of vector $\mathbf{a}^i$ in $\texttt{MEM}_\mathbf{a}$: the row is $\lfloor b_i/n_b \rfloor$ and the column is $b_i \% n_b$ (modulo operation: b_i mod n_b), the two quantities are referred as `ADX` and `Shift` respectively. These are easy to compute since n_b is a power of 2 and b_i is a represented as a binary value: the `ADX` are the first $log_2(h)$ bits (MSBs) and the `Shift` are the first $log_2(n_b)$ bits (LSBs) of b_i.

The `VectorByCirculant` implementation is reported in Algorithm 6. The unit derives from memory $\texttt{MEM}_\mathbf{a}$ for the i-th position in $\texttt{MEM}_\mathbf{b}$ the partial product $\mathbf{a}^{(\mathbf{i})}$ to update $\texttt{MEM}_\mathbf{c}$ row by row, that corresponds to the sum of partial products ($\mathbf{c} = \mathbf{c} + \mathbf{a}^{(i)}$ in Algorithm 4 and 5).

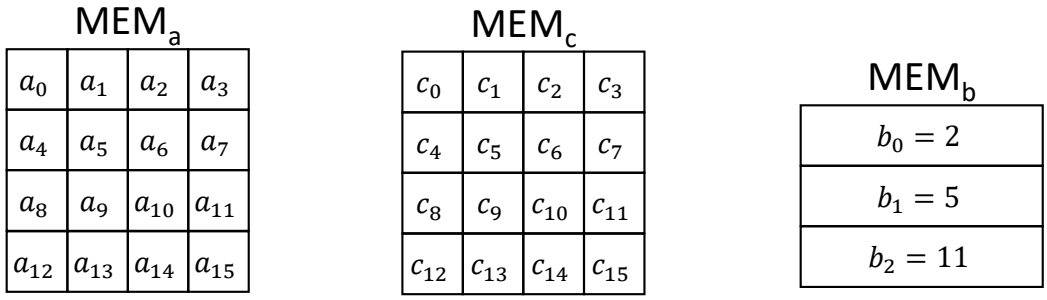
The new row that updates $\texttt{MEM}_\mathbf{c}$ are, in general, over two rows of $\texttt{MEM}_\mathbf{a}$ selected with `ADX`, these are saved in the variable $\texttt{row}_{2n_b}$, the vector `NewRow` is mapped with `Shift`. The different hardware implementations that derives from Algorithm 6 depends on *how* these n_b elements are obtained, the best option has to be selected considering the efficiency and security of each option.

The different options that has been studied to rotate $\texttt{MEM}_\mathbf{a}$ are described and the whole architecture for the multiplier is then presented.

Algorithm 6 `VectorbyCirculant Implementation`

Input: $\texttt{MEM}_\mathbf{a}$ with size $h \times n_b$, $\texttt{MEM}_\mathbf{c}$ with size $b_h \times b_w$
Output: $\texttt{MEM}_\mathbf{c}$ with size $h \times n_b$

1: **for** $i = 0$ **to** $b_h - 1$ **do**
2: $k = b_i$
3: ADX = $\lfloor k/n_b \rfloor$
4: Shift = $k\%n_b$
5: **for** $j = 0$ **to** $h - 1$ **do**
6: ADX = mod(ADX$+ j + 1$),h)
7: $\texttt{row}_{2n_b}$= [$\texttt{MEM}_\mathbf{a}$(ADX-1),$\texttt{MEM}_\mathbf{a}$(ADX)]
8: NewRow = $\texttt{row}_{2n_b}$(Shift,Shift+n_b)
9: $\texttt{MEM}_\mathbf{c}(j)$ = $\texttt{MEM}_\mathbf{c}(j)$+NewRow
10: **end for**
11: **end for**
12: **return** $\texttt{MEM}_\mathbf{c}$



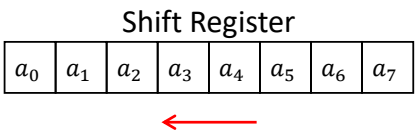
(a) $\texttt{MEM}_\mathbf{a}$ (b) $\texttt{MEM}_\mathbf{c}$ (c) $\texttt{MEM}_\mathbf{b}$

Fig. 5.2 Memories in input to `VectorByCirculant`

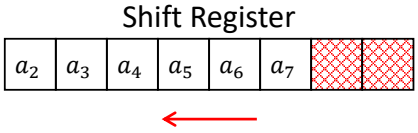
The following examples to describe the architectures considers $n = 16$ and $b_h = 3$, the dimensions of the memories is $n_b = 4$ $h = 4$ and $b_w = log_2(n) = 4$, then $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{c}$ have size 4×4 and $\texttt{MEM}_\mathbf{c}$ has size 3×4.

Rotation with Shift Register The first idea is to use shift registers to rotate the rows in $\texttt{MEM}_\mathbf{a}$.

The first row of partial product $\mathbf{a}^{(0)}$ is over rows 0 and 1 of $\texttt{MEM}_\mathbf{a}$, the value of the row is given by $b_0 = 2$ in $\texttt{MEM}_\mathbf{b}$, which points to row 0 (ADX) and column 2 (Shift) of $\texttt{MEM}_\mathbf{a}$. Two consecutive rows are then read and loaded in a $2n_b$ bit shift register, with two right shifts (two clock cycles) (the amount of the shift is 2) the first n_b bits are the first row written in $\texttt{MEM}_\mathbf{c}$.



(a) Shift Register with two consecutive rows of $\texttt{MEM}_\mathbf{a}$



(b) Shift Register with the first n_b elements that updates $\texttt{MEM}_\mathbf{c}$

Fig. 5.3 Shift Register content to rotate $\texttt{MEM}_\mathbf{a}$ by b_2

The number of clock cycles to derive a single partial product associate to position b_i with `Shift` s_i is:

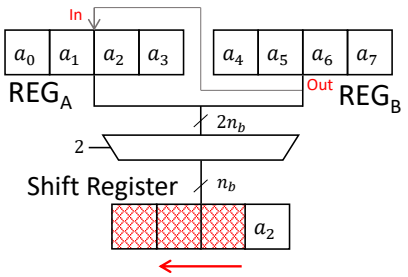
$$T_{clk} = h * s_i \tag{5.3}$$

The option has been discarded since the execution time is proportional to the asserted position in $C(\mathbf{b})$ (the Secret Key), this makes the architecture not safe for partial key recovery attacks.

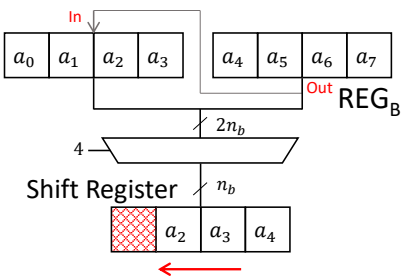
Rotation with multiplexer The rotation of $\mathbf{a}$ from $\texttt{MEM}_\mathbf{a}$ can be computed connecting the $2n_b$ register to a multiplexer that loads shift register element by element.

The rows 0 and 1 (`ADX` for $\mathbf{a}^{(0)}$) from $\texttt{MEM}_\mathbf{a}$ are loaded in two regular register with n_b elements, $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$. The output is connected to a $2n_b$ input multiplexer, the selection of the multiplexer is a counter and the output is connected to a n_b shift register.

The selection of the multiplexer changes from `Shift` to `Shift`$+n_b$ (`Shift`$= 2$), thus the shift register is loaded element by element to form the new row for $\texttt{MEM}_\mathbf{c}$.



(a) Selection of element 2 of the output from $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$



(b) Selection of element 4 of the output from $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$

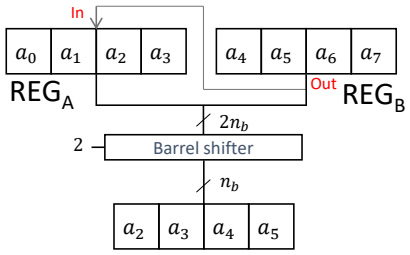
Fig. 5.4 Shift Register loaded with the complete row with n_b reads from $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$

The number of cycles to derive the i-th partial product is:

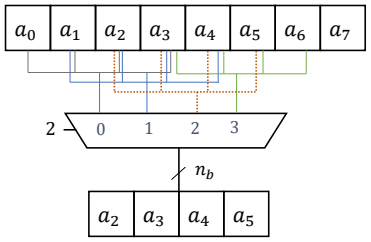
$$T_{clk} = h * n_b = n \tag{5.4}$$

The execution time is not proportional to the the asserted bits of the Secret Key, this comes at the increased cost of resources.

Rotation with Barrel Shifter The rows of each $\mathbf{a}^i$ can be obtained in parallel with the use of a barrel shifter. The first row of $\mathbf{a}^0$ is across rows 0 and 1 from $\texttt{MEM}_\mathbf{a}$, these are loaded in $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$, their output is connected to the barrel shifter that selects the correct range for the `NewRow` vector. The barrel shifter is a multiplexer with n_b input vector and a selection bit equal to 2.



(a) Selection of a complete row with the barrel shifter



(b) Detail of the barrel shifter (BS_1)

Fig. 5.5 Register with the new row loaded in parallel.

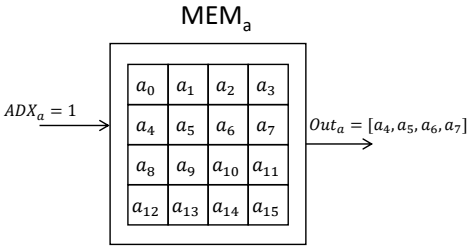
The number of clock cycles to derive a rotated vector is:

$$T_{clk} = h \tag{5.5}$$

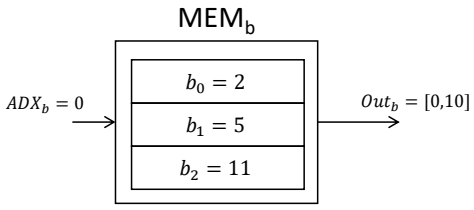
The barrel shifter introduces an additional cost in terms of resources, but it has been preferred due to its reduced number of cycles that is proportional to the selected parallelism.

The `VectorByCirculant` complete architecture and execution is described with the parameters considered in the previous example. The execution firstly loads $\mathbf{a}^0$ in $\texttt{MEM}_\mathbf{c}$, and then updates the content with $\mathbf{a}^1$ and $\mathbf{a}^3$.

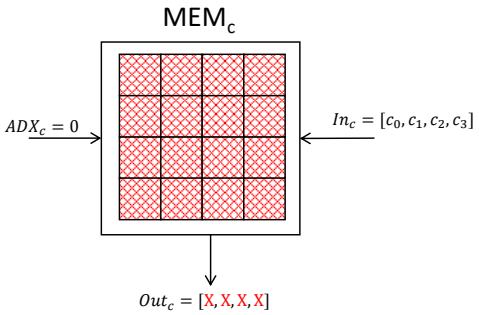
The first row ($\mathbf{a}^0$) to be loaded in $\texttt{MEM}_\mathbf{c}$ obtained from $\texttt{MEM}_\mathbf{a}$ starts with the following content of the memories:



(a) $\texttt{MEM}_\mathbf{a}$ with the second row in output



(b) $\texttt{MEM}_\mathbf{b}$ with the output position representing ADX (0) and Shift (10) as binary values.



(c) $\texttt{MEM}_\mathbf{c}$ before $\mathbf{a}^0$ is written has an undefined content.

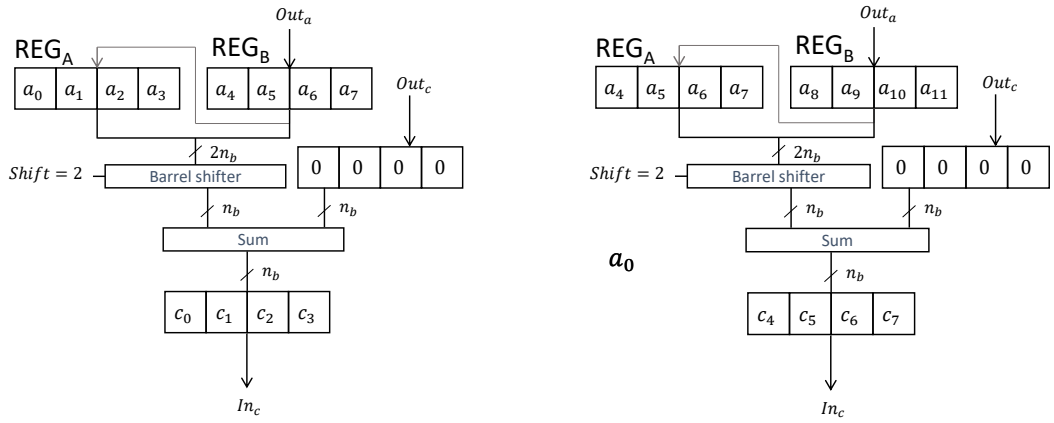
Fig. 5.6 Memories of `VectorByCirculant` with Input, Output and addressed highlighted.

The values of $\texttt{ADX}_c$, $\texttt{ADX}_b$ and $\texttt{ADX}_a$ are provided by counters, the latter is loaded with ADX.

The Data Path is in Figure 5.7, it includes the registers where the rows from $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{c}$ are loaded, barrel shifter and the element wise sum, the counters and the register with the position in $\texttt{MEM}_\mathbf{b}$ are not depicted. The loading of $\mathbf{a}^0$ does not involve the content $\texttt{MEM}_\mathbf{c}$, thus register $\texttt{REG}_\mathbf{C}$ is at zero, the register is cleared. The value from the barrel shifter is then loaded in the output register, the row is written in $\texttt{MEM}_\mathbf{c}$. Then, to update row $\texttt{ADX}_c = 1$, the a new row is read ($\texttt{ADX}_a = 2$), $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ are updated, the value shifted and then loaded in the output register.

The next $\mathbf{a}^1$ that updates $\texttt{MEM}_\mathbf{c}$ is obtained in a similarly, the initial content of the memory is in Figure 5.9. The content of the registers in the Data Path for $\texttt{ADX}_c = 0$ and $\texttt{ADX}_c = 1$ is in Figures 5.16

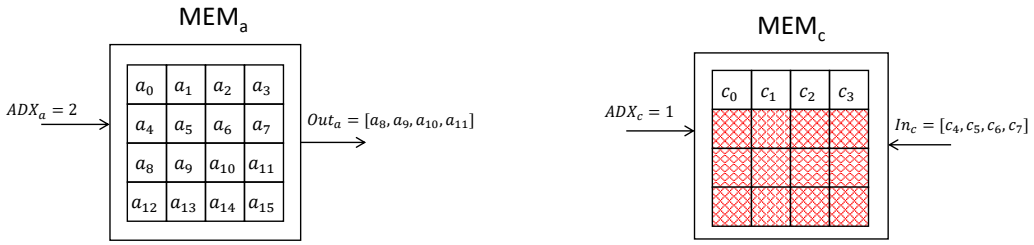
Rotation of Vectors with Generic Length The execution of the vector rotation algorithm presents slight changes when the vector length is not a multiple of n_b. In this case, the last row of memory contains values that do not belong to the original vector $\mathbf{a}$. To address this, an additional step is required, which involves storing two



(a) $\texttt{ADX}_c = 0$ available

(b) $\texttt{ADX}_c = 1$ available

Fig. 5.7 `VectorByCirculant` Data Path with the control signal form $\texttt{MEM}_\mathbf{b}$, the consecutive rows from $\texttt{MEM}_\mathbf{a}$ stored in $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ and the first row of $\texttt{MEM}_\mathbf{c}$ evaluated.



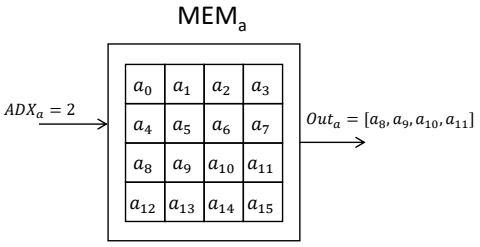
(a) $\texttt{MEM}_\mathbf{a}$ with the third row in output

(b) $\texttt{MEM}_\mathbf{c}$ has the first row loaded

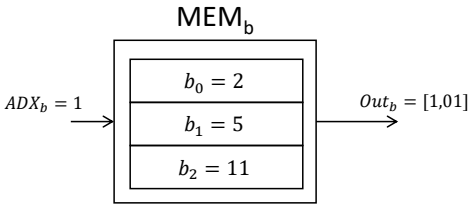
Fig. 5.8 Memories of `VectorByCirculant` with Input, Output and addressed highlighted for the writing of $\texttt{MEM}_\mathbf{c}$ at $\texttt{ADX}_c = 1$.

partial rows in $\texttt{MEM}\mathbf{c}$ and modifying the values of `Shift` for the next rows of $\texttt{MEM}\mathbf{a}$. For example, Figure 5.11 shows the vector $\mathbf{a}$ of length $n = 13$ along with the other parameters used in the previous examples.

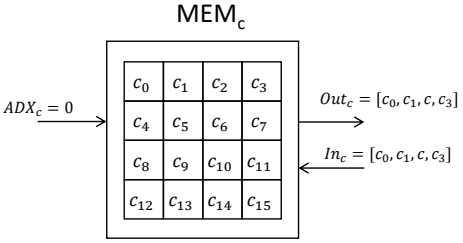
The total number of cycles required to fill $\texttt{MEM}\mathbf{c}$ is proportional to the number of rows in both $\texttt{MEM}\mathbf{a}$ and $\texttt{MEM}_\mathbf{c}$. The detailed number of cycles can be computed by starting from the description of the Control Unit, which is depicted in Figure 5.13. The execution of the algorithm is organized into three phases: the initialization of `ADX` and `Shift`, the first rotation of the vector, and the subsequent rotations of the vector to obtain the result. The initialization phase loads the first value of the rotation to be performed, as highlighted by the green box in the Control Unit diagram shown in Figure 5.13. This phase involves the following state, the name of the state is the same as the command that is executed:



(a) $\texttt{MEM}_\mathbf{a}$ with the second row in output



(b) $\texttt{MEM}_\mathbf{b}$ with the output position representing `ADX` (1) and `Shift` (01) as binary values.



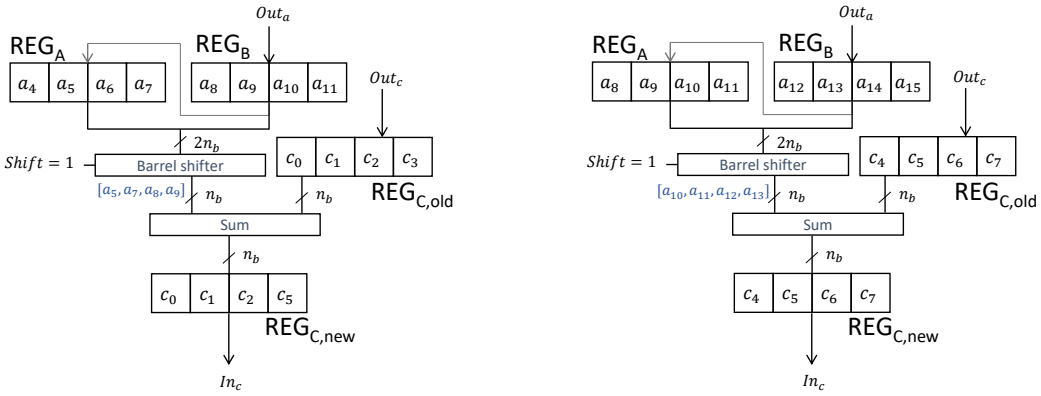
(c) $\texttt{MEM}_\mathbf{c}$ with $\mathbf{a}^0$ loaded.

Fig. 5.9 Memories of `VectorByCirculant` with Input, Output and addressed highlighted for $\mathbf{a}^1$ computation.

- *Start*, the registers and counters are cleared ;
- *Load Rotation*, the *i*-th position from $\texttt{MEM}_\mathbf{b}$ is loaded in `Position Register`, `ADX` ad `Shift`, the counter that points $\texttt{MEM}_\mathbf{b}$ is incremented and the counter for $\texttt{MEM}_\mathbf{a}$ is loaded with `ADX`;
- *Load first Row*: the `ADX` row in $\texttt{MEM}_\mathbf{a}$ is loaded in $\texttt{REG}_\mathbf{B}$ and the counter of $\texttt{ADX}_\mathbf{a}$ is incremented;

The states that compute the first rotation are highlighted in the orange box in Figure 5.13. This is achieved by iterating *h* times through the following states:

- *Load*, the row $\texttt{ADX}_a$ is loaded in $\texttt{REG}_\mathbf{B}$ and its previous values moved to $\texttt{REG}_\mathbf{A}$, the row $\texttt{ADX}_c$ is loaded in $\texttt{REG}_{\mathbf{C},old}$, the counter of $\texttt{ADX}_\mathbf{a}$ is incremented;
- *Compute*: the contents of the pair $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ is shifted and its content xores/summed up with $\texttt{REG}_{\mathbf{C},old}$, thus $\texttt{REG}_{\mathbf{C},new}$ is available in the next cycle.
- *Store*: $\texttt{REG}_{\mathbf{C},new}$ is stored in $\texttt{MEM}_\mathbf{c}$ and the counter of $\texttt{ADX}_\mathbf{c}$ is incremented.



(a) $\texttt{ADX}_c = 0$ available (b) $\texttt{ADX}_c = 1$ available

Fig. 5.10 `VectorByCirculant` Data Path with the control signal form $\texttt{MEM}_\mathbf{b}$, the consecutive rows from $\texttt{MEM}_\mathbf{a}$ stored in $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ and the first row of $\texttt{MEM}_\mathbf{c}$ evaluated.

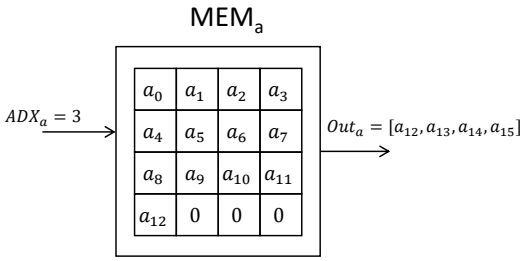


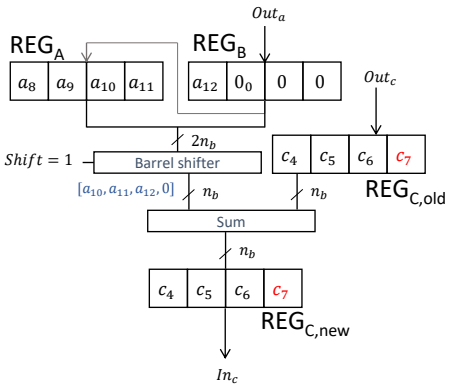
Fig. 5.11 $\texttt{MEM}_\mathbf{a}$ with a vector of length $n = 13$ and padded with zeros

The state *Load* is dual: the first rotation assumes that the content of $\texttt{MEM}_\mathbf{c}$ is uninitialized, thus $\texttt{REG}_{\mathbf{C},old}$ is cleared and the shifted row is $\texttt{REG}_{\mathbf{C},new}$; in the next rotations $\texttt{REG}_{\mathbf{C},old}$ is actually loaded.

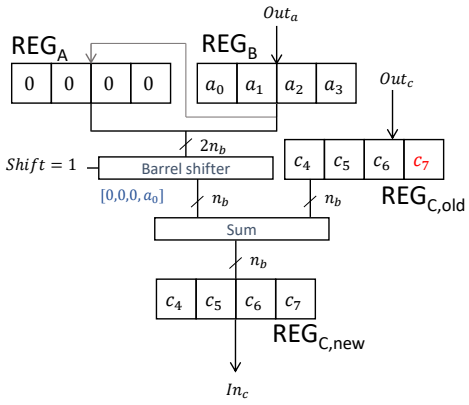
The rotation with generic n requires two more states to perform the partial read:

- *Partial Read*, the the pair $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ is rotated and the content and its content xores/summed up with $\texttt{REG}_{\mathbf{C},old}$, thus $\texttt{REG}_{\mathbf{C},new}$ is available in the next cycle;
- *Partial Store*, $\texttt{REG}_{\mathbf{C},new}$ is stored in $\texttt{MEM}_\mathbf{c}$ and the counter of $\texttt{ADX}_\mathbf{c}$ is **not** incremented;

The shift of the row is partial since the last row in $\texttt{MEM}_\mathbf{a}$ is loaded with zeros after the n values, then $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ has only part of the new row, moreover the value of `Shift` is updated in next `Load`, `Compute` and `Store` states.



(a) Partial write in $\texttt{MEM}_\mathbf{c}$, the value c_7 is not updated



(b) $\texttt{MEM}_\mathbf{c}$ row is completely updated

Fig. 5.12 `VectorByCirculant` Data Path with the control signal form $\texttt{MEM}_\mathbf{b}$, the consecutive rows from $\texttt{MEM}_\mathbf{a}$ stored in $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{b}$ and the first row of $\texttt{MEM}_\mathbf{c}$ evaluated.

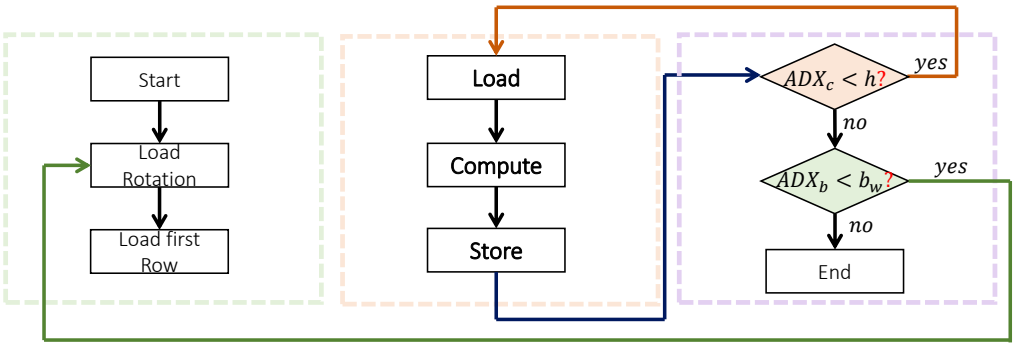


Fig. 5.13 Control Unit of `VectorByCirculant`

The addresses are provided by the counters that are incremented after each read/write of the given memory.

The evolution of the Control Unit when a rotation is computed is in Figure 5.14.

The total number of cycles is easily obtained considering the initial setup of the registers, load of the position from $\texttt{MEM}_\mathbf{b}$ and the evolution in Figure 5.14:

$$N_{cycles}^{VbC} = 1 + b_h + 3 * a_h * b_h \tag{5.6}$$

The total execution time depends on the synthesized architecture, since the scalability of the inner units affect the maximum achievable frequency (f_{max}, found

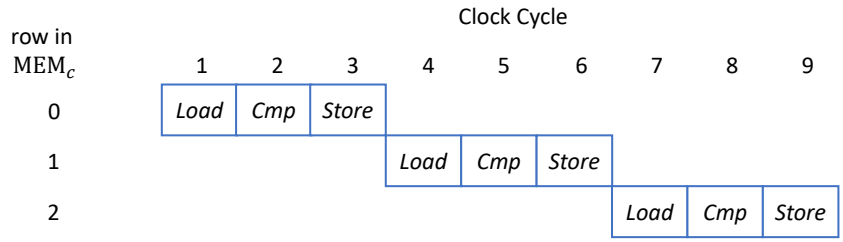


Fig. 5.14 The time evolution of the Control Unit to compute a single row in $\mathtt{MEM_b}$ after the initialization steps.

from the minimum critical path $1/T_{clk}$). Then, the parameter n_b affects both the critical path and the total number of cycles.

The total execution time is:

$$T_{ex} = N_{cycles}^{VbC} * T_{clk} \tag{5.7}$$

The total area, A, and the frequency, f,are reported in Table 5.2, considering a multiplication with a binary cyclic matrix and:

- binary input vector and binary result;
- binary input vector and integer result, with integers represented on $n_i = 4$;
- integer input vector (with $n_i = 4$) and integer result (with $n_i = 4$).

The considered length of the vector is $\approx 10^4$.

The results has been obtained with Synopsys Design Compiler and the STM FDSOI 28nm technology. The frequency f is the maximum frequency of the binary to integer multiplier.

The value of n_b is limited to 64 bit, since it is enough to show the limitations of the previous approach. The value of n_b is increased in the following improvements of the multiplier.

The total area, A, and frequency, f, with a logarithmic unit are provided in Table 5.3 for the STM FDSOI 28nm technological node.

The multiplier is meant to implement a fundamental step in the QC-LDPC Decoder for Code-based PQC, the limitations that showed in [3] when implementing the

		n_b					
		8	16	32	64	128	256
A (μm^2)	bin to bin	677	925	1440	4802	-	-
	bin to int	1019	1598	2774	7461	-	-
	int to int	1932	3467	6551	22033	-	-
f (MHz)		250	250	250	250	-	-

Table 5.2 The synthesis results of `VectorByCirculant` with linear unit barrel shifter.

		n_b					
		8	16	32	64	128	256
A (μm^2)	bin to bin	753	1028	1566	2699	4956	9953
	bin to int	990	1569	2774	7572	9793	19646
	int to int	1714	3358	6350	16175	23918	48457
f (MHz)		330	500	500	400	380	-

Table 5.3 The synthesis results of `VectorByCirculant` with logarithmic barrel shifter.

multipliers motivated the research on different approaches. The resource utilization is not provided in the present description, since the limitations are evident only in the ASIC implementaiton, whith the sythnesis on FPGA the resource usage is linear with n_b. The final results and comparison will be provided for bot ASIC and FPGA implementation.

Alternative Barrel Shifter: Logarithmic Rotate Unit The increase of the total area in the ASIC implementation depends on the parameter n_b, the effect of the increase of the parallelism is not bounded by the reduction in the execution time since the total area increased becomes exponential with n_b, as can be clearly seen from [3], where the architecture scaling is limited to $n_b = 64$. Then a different approach has been implemented, this is referred as **Logarithmic Rotate Unit**. The example is in Figure 5.15 for $n_b = 4$ and `Shift` $= 2$, similarly to the examples in Figure 5.5.

The multiplexer is split over $log_2(4) = 2$ levels, with each level containing a two input multiplexer. The split strongly reduced the area increase, while keeping the T_{clk} equivalent to the previous approaches.

The single level of multiplexing applies a rotation by approximating the value `Shift`. In the example, the input is a $2n_b = 8$ vector with a rotation of `Shift`$= 2_{10} = 10_2$, the value of the rotation can be $0,\ 1,\ 2,\ 3$, this group of 4 number is in the first

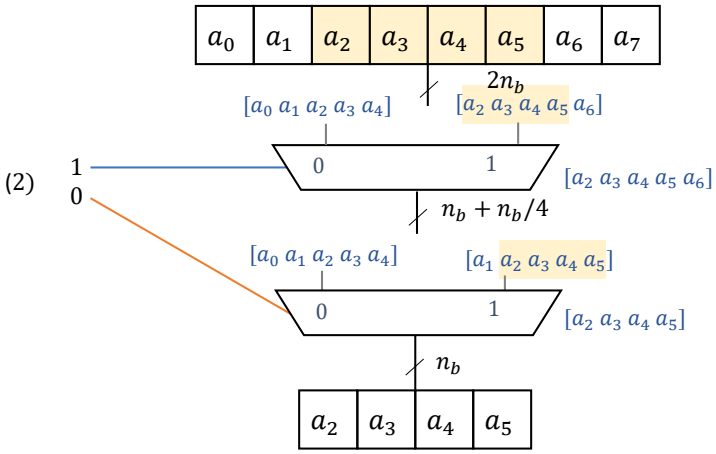


Fig. 5.15 Logarithmic approach for the Barrel Shifter

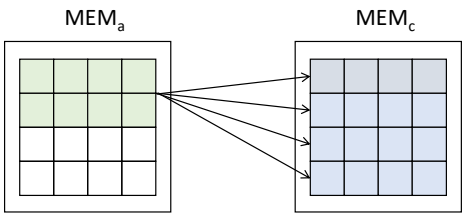
layer divided in two option with $4/2 = 2$ elements each: selection 0 has a portion of the complete vector that contains sub-vectors corresponding to `Shift` $[0,\ 1]$ and selection 1 has $[2,\ 3]$, the choice depends on the binary representation of the `Shift`, 10, the first bit (1) maps selection 1, then the group is again split in selection 0 $[2]$ and selection $1 = 3$, the last bit, 0, maps g_0, then the correct shift is selected. The layer of multiplexer selects a portion of the complete vector that contains the options of the group, for example with selection 0 in the first layers both $[a_0,\ a_1,\ a_2,\ a_3]$ and $[a_1,\ a_2,\ a_3,\ a_4]$ can be derived.

The same applied to a different n_b, the number of level is always $log_2(n_b)$ and each multiplier is driven by the binary representation of `Shift`, with its MSB that drives the first level of multiplexer.

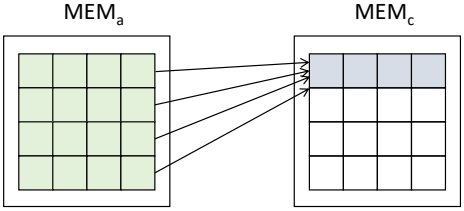
The logarithmic collapse unit, thanks to its efficiency in the ASIC implementation and the similar results for FPGA [**?**] is considered in the next sections as the only rotate unit adopted.

Alternative VbC The content of $\texttt{MEM}_\mathbf{c}$ can be obtained in an alternative way. Instead of reading the each row of $\texttt{MEM}_\mathbf{a}$ several times, the b_w row pair is loaded in the pair $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$, the b_w shifts are applied completely generate one row of the result, then only one write per row is required. This idea has been discarded because the new address of $\texttt{MEM}_\mathbf{a}$ has to be computed per each iteration resulting in an increase of the total number of cycles. The benefit of the following option is the

reduction of writes to the result memory that becomes read of the input memory, this may affect the dynamic power of the architecture since the number of writes is reduced.



(a) The single row of $\texttt{MEM}_\mathbf{c}$ is completely evaluated before writing it in memory.



(b) The content in $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ is kept constant.

Fig. 5.16 `VectorByCirculant` alternatives

5.1.2 VectorByCirculant Pipelined

The total execution time is improved increasing the throughput at which the rows of the result are computed, in `VectorByCirculant` the throughput is $1/latency$. The parameter is increased by providing a pipelined `VectorByCirculant`. The pipelining applied is both *coarse* and *fine grain*, thus both the throughput and T_{clk} are improved.

The coarse pipelining has complex Control Unit and the $\texttt{MEM}_\mathbf{c}$ must be a dual port memory.

Coarse-grained Pipeline The coarse-grained pipelining (*c-pipe*) has the same Data Path of `VectorByCirculant`, the memory include a dual port memory for $\texttt{MEM}_\mathbf{c}$, while the Control Unit execution is modified to have a pipelined execution and overlap the commands.

The Control Unit execution to provide a rotated **a** in `c-pipe VectorByCirculant` is organized in the following stages:

- Stage 1: the i-th row is loaded from $\texttt{MEM}_\mathbf{a}$, the command is *Load*;
- Stage 2: the $(i+1)\%h$-th row is loaded from $\texttt{MEM}_\mathbf{a}$, *Load*, the output from the barrel shifted computed, *Cmp*, that updates the row 0 in $\texttt{MEM}_\mathbf{c}$;

- Stage 3: the $(i+2)\%h$-th row is loaded from $\texttt{MEM}_\mathbf{a}$, *Load*, the output from the barrel shifted computed, *Cmp*, that updates the row 1 in $\texttt{MEM}_\mathbf{c}$ and row 0 is stored $\texttt{MEM}_\mathbf{c}$, *Store*;

The execution after the first two stages, in each cycle, the *Load*, *Cmp*, and *Store* from `VectorByCirculant` are computed, thus updating $\texttt{MEM}_\mathbf{c}$ at every clock cycle. The overhead is introduced by the first three stages before the pipelining execution can start and the last three stages to exit the pipelining execution. The time evolution is in Figure 5.17.

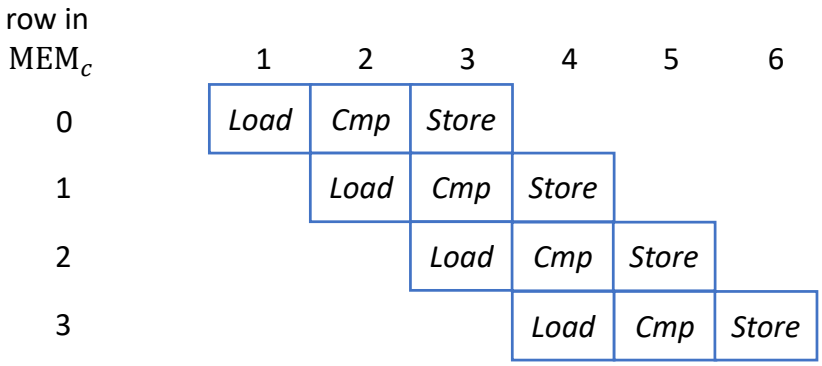


Fig. 5.17 The time evolution of the Control Unit for the coarse grain pipelined architecture to compute a single row in $\texttt{MEM}_\mathbf{c}$ after the initialization steps.

The read/write operation from $\texttt{MEM}_\mathbf{c}$ can cause a potential hazard, but the operations are performed on different rows, thus the read and write do not overlap on the same row. The Memory $\texttt{MEM}_\mathbf{c}$ is a dual port in order to have the read and write of two different rows in the same cycle, with a single port memory it is not possible to fully benefit of the pipelined `VectorbyCirculant`.

The total number of cycles for the coarse pipelined architecture is computed considering the number of clock cycles to:

- read each position in $\texttt{MEM}_\mathbf{b}$, which counts b_h cycles;
- read the starting row in $\text{MEM}_\mathbf{a}$, which counts b_h
- the two states to start and end the pipelining execution for each rotation, which counts $2 * b_h$;
- the rotations of $\texttt{MEM}_\mathbf{a}$, which counts $a_h * b_h$.

The overall number of cycles is:

$$N_{cycles}^{c-pipe} = 4 * b_h + a_h * b_h \tag{5.8}$$

The total execution time is thus improved since with the same DataPath the number of cycles is reduced and T_{clk} is kept constant.

Fine-grained Pipeline The fine-grained pipeling aims to reduce the T_{clk}, it required to study how the combinatorial paths are distributed over the circuit. In `VectorByCirculant` exists two combinatorial path:

- the first one is located between $\texttt{REG}_{\mathbf{A}}$/$\texttt{REG}_{\mathbf{B}}$ and $\texttt{REG}_{\mathbf{C,new}}$, it includes the barrel shifter and the sum block;
- the second one is located between $\texttt{REG}_{\mathbf{C,old}}$ and $\texttt{REG}_{\mathbf{C,new}}$, it include the sum block only.

Clearly, the longest path is the first one and a reduction of T_{clk} implies the use of an additional register between the barrel shifted and the sum block.

The final Data Path is in Figures 5.19, the architecture has one additional layer of registers ($\texttt{REG}_{\mathbf{S}}$) that reduces T_{clk}, together with the Control Unit the overall execution time of a complete cyclic multiplication is reduced by a factor of 3.

The execution has an additional command: $\texttt{Load}_{\mathbf{S}}$ that loads $\texttt{REG}_{\mathbf{S}}$. The Control Unit requires then three stages to enter the pipelining execution.

The Control Unit execution to provide a rotated **a**, for the fine grained pipelined `VectorByCirculant` is organized in the following stages:

- Stage 1: the i-th row is loaded from $\texttt{MEM}_{\mathbf{a}}$, the command is *Load*;
- Stage 2: the $(i+1)\%h$-th row is loaded from $\texttt{MEM}_{\mathbf{a}}$, *Load*, the output from the barrel shifter is loaded, $Load_{\mathbf{S}}$;
- Stage 3: the $(i+2)\%h$-th row is loaded from $\texttt{MEM}_{\mathbf{a}}$,*Load*, the output from the barrel shifter is loaded, $Load_{\mathbf{S}}$, the shifted vector is summed with the row 0 from $\texttt{MEM}_{\mathbf{c}}$, *Cmp*;

- Stage 4: the $(i+3)\%h$-th row is loaded from $\texttt{MEM}_\mathbf{a}$,*Load*, the output from the barrel shifter is loaded, $Load_\mathbf{S}$, the shifted vector is summed with the row 1 from $\texttt{MEM}_\mathbf{c}$, *Cmp*, row 0 is stored $\texttt{MEM}_\mathbf{c}$, *Store*;

The execution after the first three stages, in each cycle, the *Load*, *Shift*, *Cmp*, and *Store* from `VectorByCirculant` are computed, thus updating $\texttt{MEM}_\mathbf{c}$ at every clock cycle. The overhead is introduced by the first three stages before the pipelining execution can start and the last three stages to exit the pipelining execution. The time evolution is in Figure 5.18.

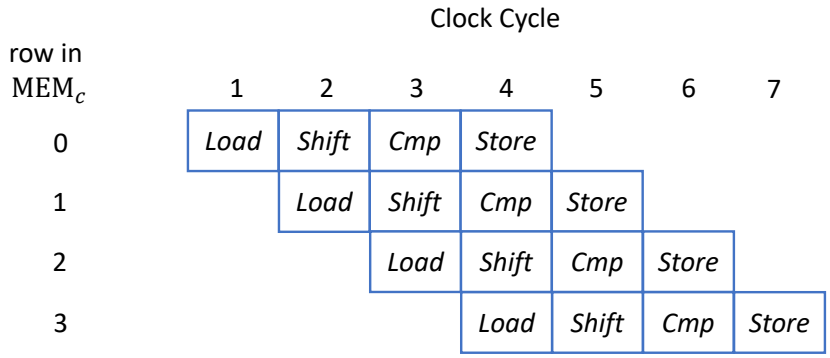


Fig. 5.18 The time evolution of the Control Unit for the fine grained pipelined architecture to compute a single row in $\texttt{MEM}_\mathbf{c}$ after the initialization steps.

The read/write operation from $\texttt{MEM}_\mathbf{c}$ are performed on different rows, thus the read and write do not overlap on the same row. The Memory $\texttt{MEM}_\mathbf{c}$ is a dual port in order to have the read and write of two different rows in the same cycle, with a single port memory it is not possible to fully benefit of the pipelined `VectorbyCirculant`.

The total number of cycles for the coarse pipelined architecture is computed considering the number of clock cycles to:

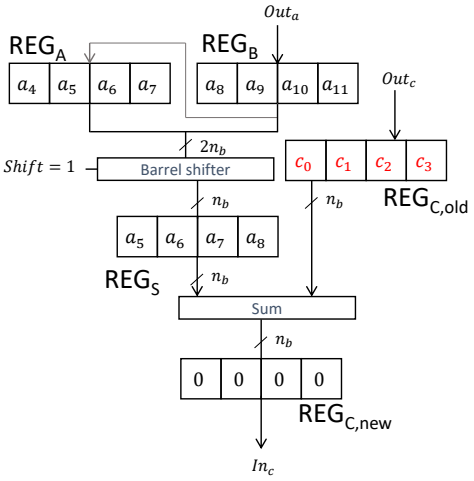
- read each position in $\texttt{MEM}_\mathbf{b}$, which counts b_h cycles;
- read the starting row in $\text{MEM}_\mathbf{a}$, which counts b_h
- the three states to start and end the pipelining execution for each rotation, which counts $3 * b_h$;
- the rotations of $\texttt{MEM}_\mathbf{a}$, which counts $a_h * b_h$.

The total number of cycles for the coarse pipelined architecture is:

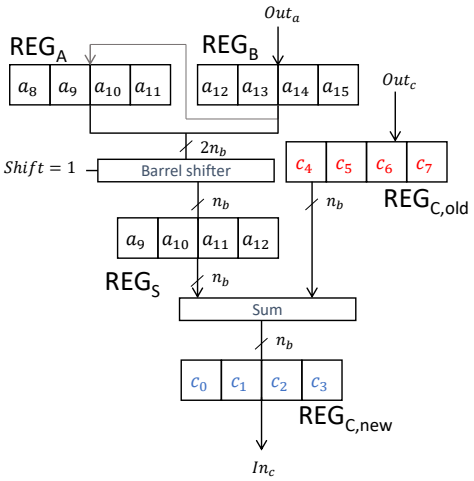
$$N_{cycles}^{f-pipe} = 6 * b_h + a_h * b_h \tag{5.9}$$

and the T_{clk}^{f-pipe} is reduced thanks to the additional registers that can cut the critical path of the architecture.

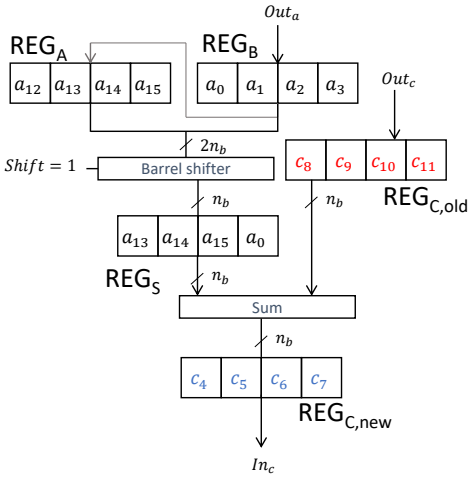
The pipelined architecture is in Figure 5.19. The split with a layer of registers depends on the factors mentioned above since at this point the throughput of the architecture is improved and the maximum clock frequency increased.



(a) $\texttt{ADX}_c = 0$ computed



(b) $\texttt{ADX}_c = 0$ available, $\texttt{ADX}_c = 1$ computed



(c) $\texttt{ADX}_c = 1$ available,$\texttt{ADX}_c = 2$ computed

Fig. 5.19 `VectorByCirculant` Pipelined Data Path with the control signal form $\texttt{MEM}_\mathbf{b}$, the consecutive rows from $\texttt{MEM}_\mathbf{a}$ stored in $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$ and the first row of $\texttt{MEM}_\mathbf{c}$ evaluated.

The final pipelined architecture, at the cost of a new register reduces the throughput to $1/3 * Latency$, since once the pipe is filled every clock cycle a new row is provided. In the first proposal the throughput is equal to $1/Latency$.

The two option resulted in two designs with different maximum frequency and area, since the key aspects to compare are the total area and execution time.

		n_b					
		8	16	32	64	128	256
A (μm^2)	bin to bin	794	1057	1601	2735	4991	9992
	bin to int	1032	1509	2774	7572	9728	19673
	int to int	1745	3387	6350	16175	23946	48487
f (MHz)		500	500	500	400	380	330

Table 5.4 The synthesis results of `VectorByCirculant` with logarithmic barrel shifter and pipelined execution.

		n_b					
		8	16	32	64	128	256
A (μm^2)	bin to bin	811	1134	1766	2699	4956	9953
	bin to int	1206	1838	3184	7572	9793	19646
	int to int	2180	3733	6823	16175	23918	48457
f (MHz)		735	617	500	400	380	-

Table 5.5 The synthesis results of `VectorByCirculant` with logarithmic barrel shifter and fine grain pipelining.

5.1.3 SparseVectorByCirculant

The `SparseVectorByCirculant` multiplier further improves the previous architecture for a sparse vector multiplied by a sparse matrix, this is another type of multiplication required in algorithms such as LEDAcrypt/BIKE.

The idea is to evaluate only the non zero positions of the resulting vector to further reduce the execution time. The algorithms is straightforward in its implementation since it is enough to combine the input positions **a** and **b**. The algorithm is a further optimization of the Schoolbook algorithm.

The multiplication has sparse inputs, thus it is more convenient to store them as a list of asserted position, while the result is in general dense and then it is stored as a

conventional vector. The SparseVectorByCirculant implementation is Algorithm 7, the input sparse vectors **a** and **b** are stored in memories $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{b}$ as a list of asserted position, the weight of the vectors are a_h and b_h, the positions has a value between $[0; n-1]$, thus $a_w = b_w = log_2(n)$ is the width that memories should have to store each position. The use of an optimized module can reduce the latency of a sparse vector sparse matrix multiplication when compare to VectorByCirculant. The use of the architecture improves the total latency of VectorByCirculant up to $n_b = 64$, since otherwise VectorByCirculant is better as can be seen in Figure **??**.

Algorithm 7 SparseVectorbyCirculant Implementation

Input: $\texttt{MEM}_\mathbf{a}$ with size $a_h \times a_w$, $\texttt{MEM}_\mathbf{c}$ with size $b_h \times b_w$
Output: $\texttt{MEM}_\mathbf{c}$ with size $h \times n_b$

1: **for** $i = 0$ **to** $b_h - 1$ **do**
2: **for** $j = 0$ **to** $a_h - 1$ **do**
3: k = ($\texttt{MEM}_\mathbf{a}$(j)+$\texttt{MEM}_\mathbf{b}$(i))$\%n$
4: ADX = $\lfloor (k/n_b) \rfloor$
5: Shift = $k\%n_b$
6: NewRow = **0** ▷ Vector of length n_b
7: NewRow(Shift) = 1 ▷ Value 1 in position Shift
8: $\texttt{MEM}_\mathbf{c}$(ADX) = $\texttt{MEM}_\mathbf{c}$(ADX)+NewRow
9: **end for**
10: **end for**
11: **return** $\texttt{MEM}_\mathbf{c}$

The DataPath and Control Unit of SparseVectorByCirculant are quite simple, the positions from $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{b}$ are loaded in registers $\texttt{REG}_\mathbf{A}$ and $\texttt{REG}_\mathbf{B}$. The values are combined, the modulo sum is performed, and it is stored in $\texttt{REG}_\mathbf{R}$, then the ADX and Shift area easily obtained: the Shift corresponds to the $log_2(n_b)$ LSBs of the resulting position while the other MSB bits are the ADX. Then the NewRow is the result of a decoded Shift, the ADX row in $\texttt{MEM}_\mathbf{c}$ is loaded in $\texttt{REG}_\mathbf{C}$, which accumulates the output of the decoder and then updates the ADX row of the memory. The Data Path is in Figure 5.20.

The Control Unit issues the command to load the positions from $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{c}$, *Load*,compute the sum, *Sum*, compute the modulo sum, *ModSum*, load the row from $\texttt{MEM}_\mathbf{c}$, *LoadC*, update the row, *UpdateC* and finally store it in memory, *Store*. The modulo sum step is split in two cycles in order to reduce the critical path.

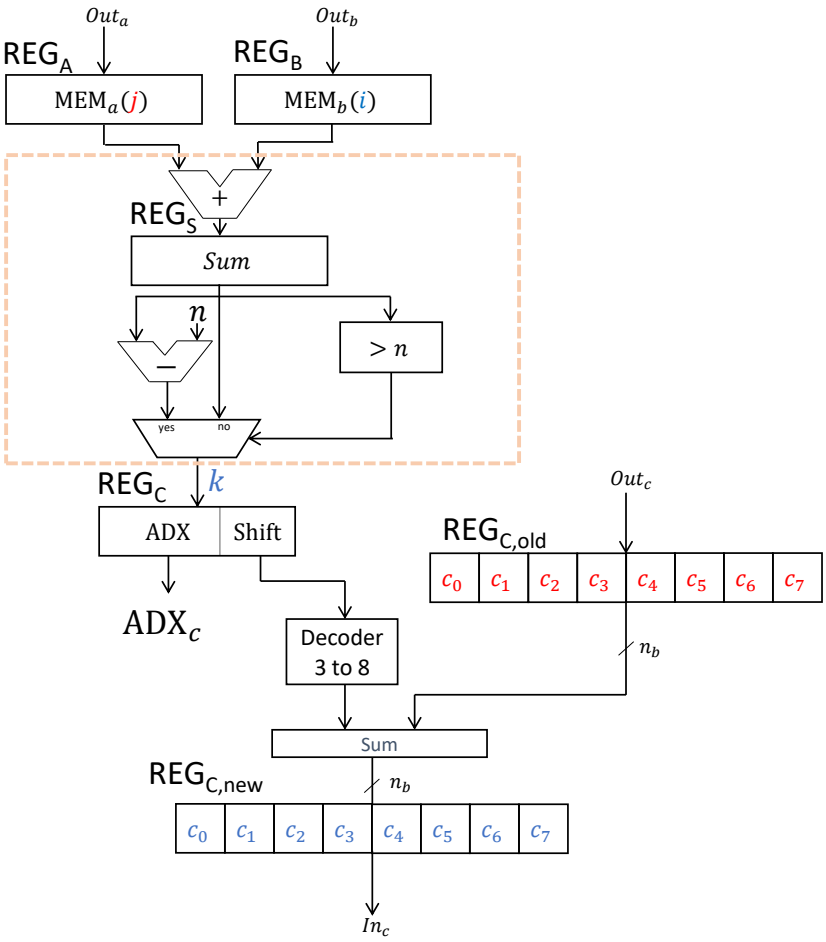


Fig. 5.20 The Data Path of `SparseVectorByCirculant` multiplier. The modular sum is in the orange box.

The total number of cycles required to compute the result is:

$$N_{cycles}^{sVbC} = 6 * a_h * b_h \tag{5.10}$$

The Time Evolution for the pipelined execution is in Figure 5.22:

Resource sharing The improvement of the unit is to combine both `VectorByCirculant` and `SparseVectorByCirculant` in order to reuse the shift unit. The barel shifter and sum unit in `VectorByCirculant` can be employed to derive the value `NewRow` by rotating the vector $[1, 0, 0, \ldots, 0]$ (with length n_b). The Data Path can be shared between the multipliers, the values of $\texttt{ADX}_C$ is connected to $\texttt{MEM}_\textbf{c}$, while in $\texttt{REG}_\textbf{A}$ and $\texttt{REG}_\textbf{B}$ (from `VectorByCirculant`) the vector $[1, 0, 0, \ldots, 0]$ (with length $2 * n_b$) is

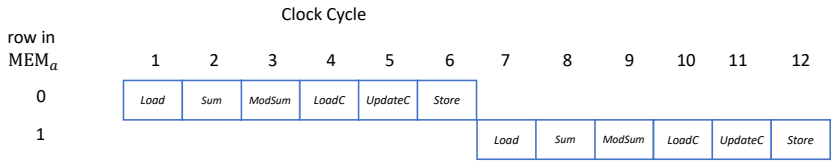


Fig. 5.21 Time evolution of `SparseVectorByCirculant`

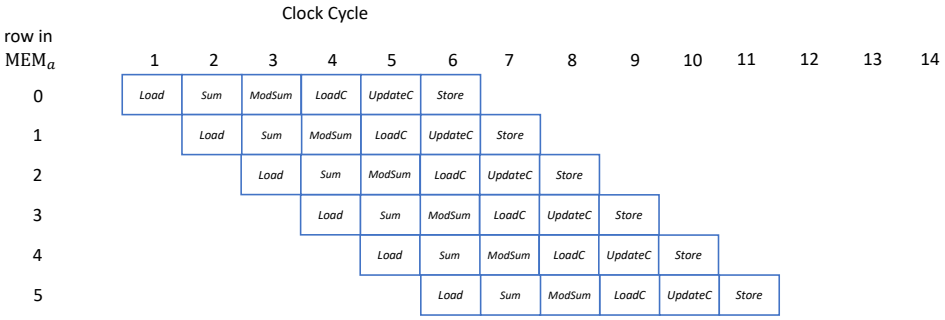


Fig. 5.22 Time evolution of Pipelined `SparseVectorByCirculant`

loaded during the execution and rotated, this the additional cost of a decoder that for increasing values of n_b can be an issue.

SparseVectorByCirculant Pipelining The pipelining can be easily applied to the present architecture executing commands in parallel. The architectures requires a Dual Port Memory for $\mathtt{MEM_c}$. The final execution time is then:

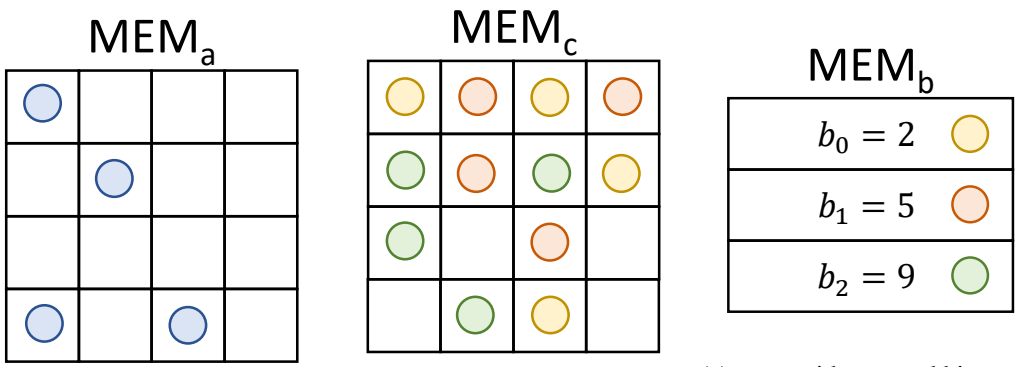
$$N_{cycles}^{sVbC} = 10 * b_h + a_h * b_h \tag{5.11}$$

The Time Evolution for the pipelined execution is in Figure 5.22:

Initial optimization goal The initial goal was to optimize the execution by rearranging the computation such that each row of the result was completely computed before storing it in memory. The idea can be clearly understood by looking at the result, this is the sum of b_w sparse vectors that ends up not to 'sum' over each row of the result memory, but they are distributed over the row are separate elements. This feature of the result is due to the presence of a sparse vector input and to the low probability of a sum to occurs.

The idea is described in Figures 5.23, with $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{c}$ described with dots: the input vector is very sparse, while the result is dense since it is the result of shifted version of the input (highlighted with a color corresponding to the amount of the rotation).

The initial idea was to calculate the rows $\texttt{MEM}_\mathbf{c}$ bit by bit and then write it in memory, but this approach requires a preliminar study of the positions in $\texttt{MEM}_\mathbf{a}$ and $\texttt{MEM}_\mathbf{b}$ and then to arrange arrange the sVbC calculations such that the result is evaluated row by row, but this would have required extra time that is not balanced by the benefit of this approach: it is not possible to use the bit by bit generation of sVbC and the row by row generation of VbC.



(a) Sparse $\texttt{MEM}_\mathbf{a}$

(b) $\texttt{MEM}_\mathbf{c}$.

(c) $\texttt{MEM}_\mathbf{b}$ with asserted bits associated to a color.

Fig. 5.23 Memories of `SparseVectorByCirculant` with sparse Input, and Output with highlighted position of the asserted bits from the input.

The total area occupation and target frequency, the same as `VectorBycirculant`, of the module is provided in Table 5.6 considering a vector of size $\approx 10^4$, results has been obtained with Synopsys Design Compiler and the STM FDSOI 28nm technology.

		n_b					
		8	16	32	64	128	256
$A\ (\mu m^2)$	pipeline	812 880	952 1013	1226 1301	1757 1785	2832 2899	5131 5203
$f\ (MHz)$		500	500	500	400	380	330

Table 5.6 The synthesis results of `SparseVectorByCirculant` with and without pipelining.

SparseVectorByCirculant and VectorByCirculant

The use of an optimized module can reduce the latency of a sparse vector sparse matrix multiplication when compare to `VectorByCirculant`. The use of the non pipelined architectures improves the total latency of `VectorByCirculant` vs `SparseVectorByCirculant` up to $n_b = 64$, since the total area is considered, the latency for a $n = 12$ since otherwise `VectorByCirculant` is better.

5.2 Schönhage Stressen Architecture

The Schönhage Stressen algorithm has been implemented to compute the multiplication between a vector and a cyclic matrix, which is considered as a convolution. The 'convolution' approach is widely used in PQC primitives that involve a polynomial multiplication, the multiplier is applied to LDPC/MDPC PQC primitives in order to understand the impact on the area of the most efficient algorithms for large multiplication in the domain of Code-based Post Quantum Cryptography. The length of the polynomials involved in such algorithms ($n > 10^4$) suggests that the method is the best in terms of time complexity, but the architecture is more complex than the Schoolbook approach.

The Schönhage Stressen algorithm is applied to **aB**, with **B** a binary cyclic and sparse matrix.

5.2.1 Number Theoretic Transform Computation

The transform which is applied is *Fast Fourier Transform* (FFT/IFFT), the Fourier Transform posses the property of the convolution in time that becomes a multiplication in frequency, the computations involves complex and floating point numbers. The Schönhage Stressen algorithm applied cyclic and polynomial multiplications applies Number Theoretic Transform, instead of the Fourier Transform: the NTT and the INTT. The benefit with the use of finite field arithmetic, which is carried on with integers, while the FFT/IFFT requires floating point arithmetic.

The Number Theoretic Transfom is introduced from the Fourier Transform, the approach of the FFT is applied to the NTT too. The *Discrete Fourier Transform*

(DFT) is applied to a signal $x(n)$ in time domain, this is transformed to $X(k)$ which is the expression of the input signal in frequency domain. The result is obtained with:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i2\pi kn/N} \tag{5.12}$$

The NTT equation is:

$$X(k) = \sum_{i=0}^{N-1} (x(i) \cdot \alpha^{ik}) \mod P, \tag{5.13}$$

In the NTT the main parameters are the modulus P and radix α, these quantities are integer numbers thus the operations are fixed point multiplication and additions.

The execution has a complexity $O(n^2)$, with the FFT and the Cooley-Tuckey FFT (CT-FFT) algorithm, the complexity is reduced to $O(nlog(2))$. The speed up in the Schönhage Stressen algorithm is due to the use of the FFT algorithm.

The expression in (5.13) with the CT speed up is rewritten as:

$$\begin{aligned} X(k) = & \sum_{j=0}^{N/2-1} x(2j)\alpha^{(2j)k} \mod P \\ & + \sum_{j=0}^{N/2-1} x(2j+1)\alpha^{(2j+1)k} \mod P \end{aligned} \tag{5.14}$$

The values of α and P are parameters of the NTT.

The parameters and adaptation of the NTT to LEDAcrypt/BIKE sizes is described in the following.

5.2.2 Number Theoretic Transform parameters

The NTT applied to the vectors of LEDAcrypt/BIKE has specific requirements: the length of the vectors is not a power of 2, in particular their length is a prime number and the result of the multiplication should be exact, since some uncertainty. In addition, the addressed multiplication is the vector by sparse cyclic matrix, thus the transform has to be applied to dense or sparse vectors.

The Direct and Inverse Number Theoretic Transform algorithm and implementation are described in the following, considering that the application is Code-based PQC the architecture that is described has the same Data Path and by changing the.

The parameters are the modulo, P, and the radix, α of the NTT and the radix α_{inv} and N_{inv} for the INTT. These values are selected starting from the length of the input vector,n, and the maximum integer of the vector, $max(\mathbf{a})$. The input is binary, thus $max(\mathbf{a}) = 1$.

Modulus P The modulus of the transform for binary vectors is found by selecting a prime number $P > n$, where n is the length of the binary vector, with the modulus the primitive root of unit,r, can be computed. The primitive n^{th} root of unity. The value of P si selected in the form $P = kN + 1$. The modulus P is computed, in general, by exhaustive search.

radix α The value of the radix is selected starting from P and r, such that $a = r^k \ mod \ P$ and $a^n \ mod \ P = 1$.

The multiplication by α^k (with $k = 0, \cdot, n-1$) and *mod P* are the equivalent of the multiplication by frequencies $e^{-i2\pi k/n}$, in fact the *radix* condition is present in the FFT, since the radix is $e^{-i2\pi/n}$, then in fact the value $(e^{-i2\pi/n})^n = 1$ (from the Euler's identity). The property of the frequencies that corresponds to n divisions of a 2π circle is equivalent to the $\alpha^i \ mod \ P$ operation, which *rotates* over the range $0, \cdot P-1$ range.

Prime and Zero Padding

The CT-NTT has a constraint on the length of the input vectors, thus must be a power of 2, the usual sizes of vectors in LEDAcrypt/BIKE has to be properly padded to apply the transform.

The vector padding is applied to the inputs of the cyclic matrix multiplication, since the design of Code-based PQC primitive requires that the length is a prime number.

In literature a series of methods are know to efficiently handle such a condition: Radar's [60] and Bluestein's [61] algorithm can compute the CT-FFT, for length of

the signals prime number or generic; the methods can be applied to the NTT. The padding to be applied is derived starting from the cyclic multiplication in order to have the same result by multiplying bigger vectors.

The inputs has length n which becomes n', the result of the cyclic multiplication is the same if the matrix vector $\mathbf{b}$ is extended with a flipped replica of itself an becomes $\mathbf{b}'$: this way $C(\mathbf{b}')$ ($\mathbf{B}'$) is a cyclic matrix and contains $C(\mathbf{b})$ ($\mathbf{B}$) in the upper left corner. The vector $\mathbf{a}$ becomes $\mathbf{a}'$, this time it is enough to apply the zero padding to have a correct result. The value of the equation and the correctness of the result is proved in the following example, with vector length $n = 5$.

Prime Padding The padding applied to the input $\mathbf{b}$ is referred as PrimePadding. The vector is $\mathbf{b} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \end{bmatrix}$, the corresponding cyclic matrix is:

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{5.15}$$

The vector $\mathbf{b}$ is extended to $\mathbf{b}' = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$, with length $n' = 9$, the cyclic matrix is then:

$$\mathbf{B}' = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{5.16}$$

In general, the extension applied to the vector associated to cyclic matrix $\mathbf{B}'$ is in Equation (5.17).

$$\mathbf{b}' = \begin{bmatrix} b_0 & \dots & b_{n-1} & 0 & \dots & 0 & b_{n-1} & \dots & b_1 \end{bmatrix} \tag{5.17}$$

The value of n' is $2 * n - 1$ approximated to next power of 2 value, the $n' - 2n + 1$ values wich are not b_i are set to 0.

Result The vector **a** can be padded with the method in Equation (5.17) or with zeros in Equation (5.18)

$$\mathbf{a}' = \begin{bmatrix} a_0 & \dots & a_{n-1} & 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix} \tag{5.18}$$

The multiplication which is performed is $\mathbf{a}'\mathbf{B}'$, with result $\mathbf{c}'$, this is reduced to vector **c** by removing the last $n' - n$ bits, as showed in Equation (5.19).

$$\mathbf{c}' = \begin{bmatrix} c'_0 & c'_1 & \dots & c'_{n-1} & c'_n & \dots & c'_{n'-2} & c'_{n'-1} \end{bmatrix} \tag{5.19}$$

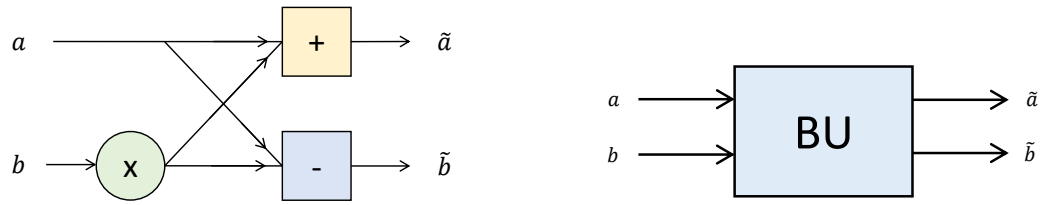
The result is $\mathbf{c} = \begin{bmatrix} c_0 & c_1 & \dots & c_{n-1} \end{bmatrix}$ The choice of the padding applied to **a** does not affect the correctness of the result, but with the PrimePadding applied the result would be:

$$\mathbf{c}' = \begin{bmatrix} c_0 & \dots & c_{n-1} & 0 & \dots & 0 & c_{n-1} & \dots & c_1 \end{bmatrix} \tag{5.20}$$

5.2.3 Number Theoretic Transform Architecture

The Number Theoretic Transform with the Cooley-Tuckey speed-up rearranges the multiplication/summation in a more efficient way, the approach, in hardware, applied a 'butterfly' unit to perform such computations, the unit is in Figures 5.24 with the basic elements of the computation highlighted: in green for the multiplication by a twiddle factor, in blue the sum and in yellow the subtraction. The unit is referred in two ways: with the inner elements highlighted, as in Figure 5.24a; with a block BU, in Figure 5.24b.

The example for a input signal **x** transformed into **X** with a $n = 8$ points NTT is depicted in Figure 5.25, some elements of the BU unit are highlighted to clarify their location.



(a) Inner structure (b) Block

Fig. 5.24 Butterfly Unit with one multiplier (`MPY`), adder (`ADD`), subtractor (`SUB`).

The parallelism, in this contest, is referred to the number of BU that has been implemented.

The algorithm computes the transform of a dense input **x** stage by stage (s) until the last stage, $m = log_2(n)$. The input of NTT is in bit reverse order, thus for clarity its elements are renamed into $v_i(j)$, where i refers to the stage and j refers to the element in a stage. The output, X, is in natural order.

The CT-NTT pseudo code is in Algorithm 8.

Algorithm 8 `NTT multiplier Implementation (iterative CT)`

Input: signal **a** with length $n = 2^m$
Output: the transform A

1: $A = a$
2: **for** $s = 1 : m$ **do**
3: $\quad n_{op} = 2^s$
4: $\quad$ **for** $i = 1 : s : m$ **do**
5: $\quad\quad$ **for** $j = 1 : n_{op}/2$ **do**
6: $\quad\quad\quad T = W_{j,s} * A(i + j + n_{op}/2)\ mod\ P$ ▷ `MPY`
7: $\quad\quad\quad U = A(i + j)$
8: $\quad\quad\quad A(i + j + n_{op}/2) = (U + T)\ mod\ P$ ▷ `ADD`
9: $\quad\quad\quad A(i + j) = (U - T)\ mod\ P$ ▷ `SUB`
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: **end for**

The implementation in Algorithm 8 computes one T and U pair per iteration, while the architecture provided parallelize the task by computing $n_u = 8$ elements

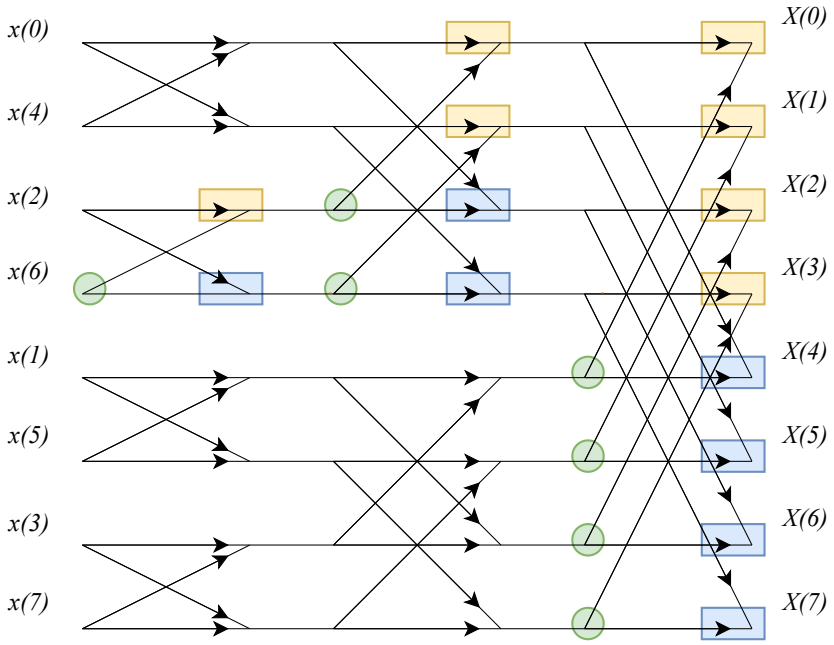


Fig. 5.25 NTT structure with $n = 8$

per iteration. This increases the complexity of the Control Unit, since the correct addresses of input/output and constant memories has to be provided.

The CT-NTT scheme for a dense input considering 8-points is in Figure 5.26. The input is vector $\mathbf{x}$, the output is X. The number of stages in the example is 3 ($log_2(8)$), the transform at each stage is addressed with vector $\mathbf{v}_i$, with index i that is referred to the the actual stage.

The transform of a dense vector with the NTT is a straightforward application of the CT Algorithm, with a progress over a single stage before moving to the next one: the value of $\mathbf{v}_0$ is computed, then $\mathbf{v}_1$ and so on. The twiddle factors are not reported in Figure 5.26, they are located in the following points:

- $s = 0$, the twiddle factor is W_0^0 and it is multiplied by $v_0(1)$, $v_0(3)$, $v_0(5)$, $v_0(7)$;
- $s = 1$, the twiddle factors are W_0^1 and W_2^1 multiplied by $v_1(2)$ and $v_1(3)$ then $v_1(6)$ and $v_1(7)$, respectively;
- $s = 2$, the twiddle factors are W_0^2, W_1^2, W_2^2 and W_3^2 multiplied by $v_2(4)$, $v_2(5)$, $v_2(6)$ and $v_2(7)$.

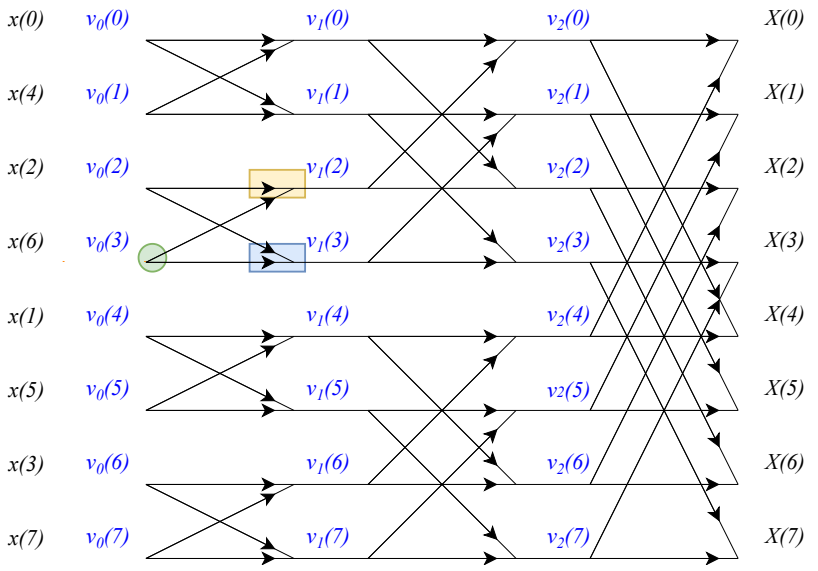


Fig. 5.26 8-points NTT with a dense input

The twiddle factors, despite being labelled in a different way, are actually the same set of values from $s = 2$, since $W_0^0 \equiv W_0^2 \equiv W_0^1$ and $W_2^2 \equiv W_2^2$.

The condition with a sparse inputs implements a different strategy in order to reduce the number of computation and avoid to compute the zeros. The sparse implementation of the NTT, with the pseudo code in Algorithm 9, schedules the operations in a different way: the single asserted bits of **a**, since they are sparse, will be computed completely up to stage $s = s_{lim}$ before moving to the next position. The transforms of each element are mixed together to have **A** at stage s_{lim} and moved to the usual NTT algorithm from stage $s = s_{lim+1}$. The algorithm iterates over the asserted values in **a**, d_a, which ares stored as binary values in $\mathbf{Pos}_a$ and consideres as binary vectors in the algorithm. The, since a sparse value is processed the computation is a multiplication and subtraction or a copy of the input value into two outputs, this choice depends on the value assumed by the l bit of Pos_v (a single row of $\mathbf{Pos}_a$), then the execution:

- with $Pos_v(l) = 1$, the single input value is multiplied by a twiddle factor and copied as it is for one output and subtracted by P for the second output;
- with $Pos_v(l) = 0$, the input is copied copied in two identical outputs;

The example of a 8-point sparse NTT is in Figures 5.27 and 5.28, the execution is both sparse and dense. The example shows two conditions, with the asserted bits 'near' of 'far', independently of their distance the asserted bits are processed separately.

The examples in Figure 5.27 has $v_0(3)$ and $v_0(2)$ asserted. The element $v_0(2)$ is expanded, in stage $s = 0$ the value is copied into $v_1(2)$ and $v_1(3)$, in stage $s = 1$ $v_1(2)$ and $v_1(2)$ are multiplied by the twiddle factors W_1^0 and W_1^2, then copied into $v_2(2)$ and $v_2(3)$, the subtraction is performed $(P - v_i(j))$ to store the *mod P* values into $v_2(0)$ and $v_2(1)$. The process is similar for $v_0(3)$.

The transform at stage s_{lim} is not a vector, but a set of transforms of each asserted position: for the example in Figure 5.27, with 2 asserted bit the transform stores two rows that contains the value of $[v_2(0)\ v_2(1)\ v_2(2)\ v_2(3)]$ for $v_0(2)$ and $v_0(3)$. The following step merges the rows in order to obtain the complete $\mathbf{v}_{s_{lim}}$ for the dense stages. The merge step is performed by mixing the resulting rows of the dense input: such rows can be summed up or alternatively concatenated. The merge in Figure 5.27 is performed by summing the $[v_2(0)\ v_2(1)\ v_2(2)\ v_2(3)]$ rows of the asserted bits and then concatenated to a set of zeros. The merge step for the example in Figure 5.28 is a concatenation between vector $[v_2(0)\ v_2(1)\ v_2(2)\ v_2(3)]$ that derives from $v_0(3)$ and $[v_2(4)\ v_2(5)\ v_2(6)\ v_2(7)]$ that derives from $v_0(6)$

bit-reverse order The input vector $\mathbf{x}$ is provided in 'bit-reverse order', while the output X is in the linear order. The input vector is stored in natural order, thus an additional post-processing is need to 'reverse' its order.

Modular arithmetic

The updated values store in vector $\mathbf{A}$, when computing the multiplication and addition/subtraction the additional *mod P* operation is applied to have the result in the range $[0; P-1]$.

The definition of *mod P* for integer a is:

$$a_P = a\%P \tag{5.21}$$

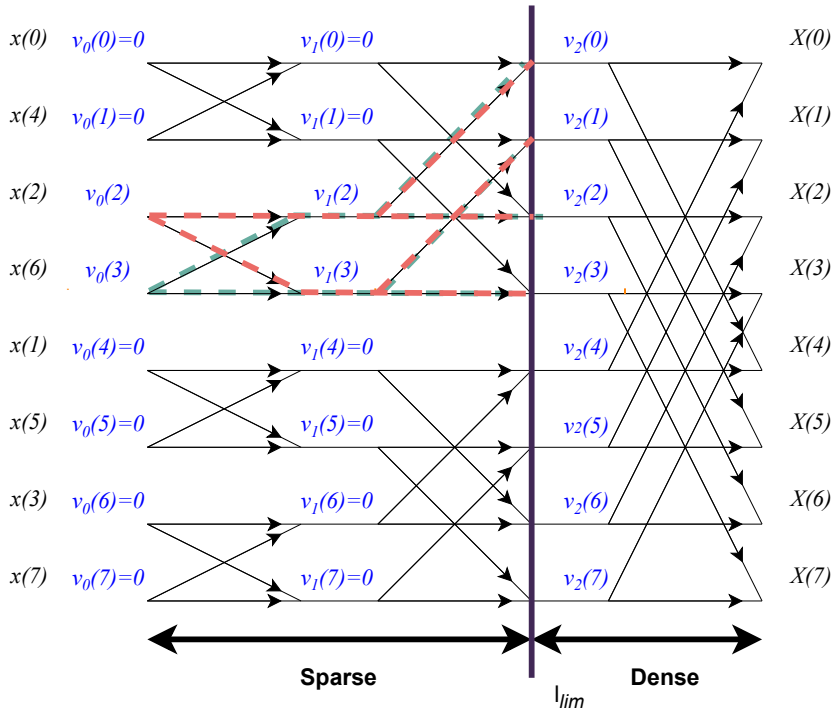


Fig. 5.27 8.points NTT with sparse input, asserted elements: $v_0(3)$ and $v_0(2)$.

Despite the operation requires a division, it is simplified for modular addition/subtraction between integers that are *mod* P, in the example the addition $c = a + b$ or $c = a - b$:

$$c_p = c \,||c - P \tag{5.22}$$

It is enough to compare the result with P and then subtract or not the modulus, this is due to the fact that c is in the range $-2P < c < 2P$, thus it is straightforward to adapt the result to $[0; P-1]$.

The multiplication between two integers, to be scaled in $[0; P-1]$ may involve divider, since the result is in $[0; P^2]$ for modulo P inputs. The idea to issue multiple times a subtraction is not feasible and the presence of a divider (hardware divider) increases the gate complexity of the whole architecture. The idea is to use a technique that reduces c by exploiting other properties of the operations, these are referred as **modular reduction**, the most common methods are: **Montgomery Reduction**[62] and **Barrett Reduction**[63].

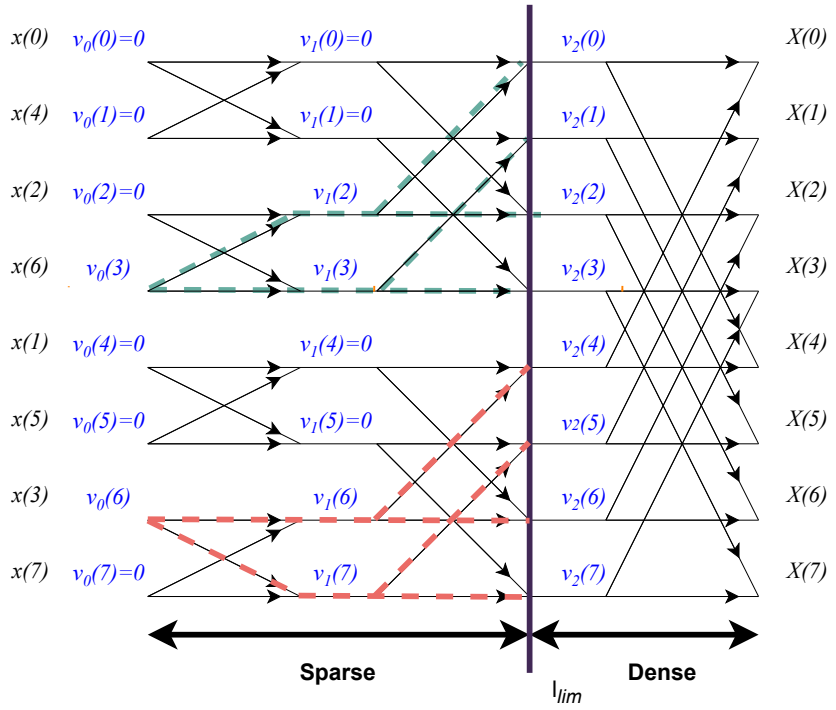


Fig. 5.28 8.points NTT with sparse input, asserted elements: $v_0(3)$ and $v_0(6)$.

Montgomery reduction The Montgomery reduction evaluates the modular multiplication $c = ab \; mod \; P$ such that the division by a generic P becomes a division by P_s, which is selected to be a power of two, making the division just a set of shifts.

The multiplication with the Montgomery Reduction is performed is in Algorithm 10, it include both the multiplication and modulo P reduction.

The modular multiplication evaluated *by definition* has the additional cost of a divider, while the Montgomery multiplication introduces an overhead because of input and output conversion to have the Montgomery form:

$$a_m = a \cdot P_s \; b_m = b \cdot P_s \; c = c_m / P_s \tag{5.23}$$

The advantage of the algorithm is the fact that the multiplications and additions/subtractions in the NTT (INTT) can be performed in Montgomery form, the distributive property is preserved, and applying the conversions only to the input and output of the transform, thus saving the cost of the input and output conversions for every multiplication. The second important advantage is that the result of the

Algorithm 9 `sparse NTT multiplier Implementation`

Input: signal **a** with length $n = 2^m$, provided by its asserted positions $\mathbf{Pos}_a(i_v)$
Output: the transform A of each $\mathbf{Pos}_a(i_v)$ upt to stage s_{lim}.

for $i_v = 1 : d_a$ **do**
  $Pos_v = \mathbf{Pos}_a(i_v)$
  for $s = 0 : s_{lim}$ **do**
    $A^{old}(i_v) = A(i_v)$
    $n_{op} = 2^l$
    for $k = 1 : n_{op}$ **do**
      if $Pos_v(s) = 1$ **then**
        $A(i_v, 2k) = W_s A^{old}(i_v, k)$
        $A(i_v, 2k+1) = P - W_s A^{old}(i_v, k)$
      else
        $A(i_v, 2k) = A^{old}(i_v, k)$
        $A(i_v, 2k+1) = A^{old}(i_v, k)$
      end if
    end for
  end for
end for

multiplications is always exact, which in the Barrett multiplication the correctness of the multiplication has to be studied.

In the present work the Montgomery multiplication is applied to compute the modular multiplication in the NTT/INTT applied to Code-based PQC.

Barrett multiplication The Barrett multiplication is performed by means of a precomputed values, which is the reciprocal of an integer.

The Barrett Reduction is in Algorithm 11

Algorithm 10 Montgomery Reduction

Input: integer factors of the multiplication in Montgomery form, a_m, b_m;
integer $y_s \in [0, P_s - 1]$ s.t. $y_s = -1 \ mod \ P_s$;
Output: product in Montgomery form, c_m;

1: $x = a_m \cdot b_m$ ▷ Product
2: $x_s = x \bmod P_s$ ▷ Modular Reduction with P_s
3: $s = (x_s \cdot y_s) \bmod P_s$
4: $c_s = (x + s \cdot P)/P_s$ ▷ Result modulus P_s
5: **if** $c_s < P$ **then**
6: $c_m = c_s$
7: **else**
8: $c_m = c_s - P$
9: **end if**

Algorithm 11 Barrett Reduction

Input: result of the multiplication, $c = ab$;
modulus P, not power of 2;
integer k s.t. $2^k > P$;
Output: product $mod \ P$, $c_P = c \ mod P$;

1: $n = \lfloor 4^k / P \rfloor$ ▷ Constant of the method
2: $c_m = c - \lfloor c * \frac{n}{4^k} \rfloor P$
3: **if** $c_m < P$ **then**
4: $c_P = c_m$
5: **else**
6: $c_P = c_m - P$
7: **end if**

The correctness of the computation depends on how the constant n is approximated, thus the Montgomery multiplication is preferred.

5.2.4 NTT-based multiplier for Code-based PQC

The NTT-based multiplier is applied to a cyclic matrix multiplication between input vector $\mathbf{a}$ and cyclic matrix $C(\mathbf{b})$. The result, with the Schönhage Stressen method, transforms the vectors $\mathbf{a}$ and $\mathbf{b}$ into $\mathbf{A}NTT(\mathbf{a})$ and $\mathbf{B} = NTT(\mathbf{b})$ with the NTT algorithm. Then $\mathbf{C} = \mathbf{A} * \mathbf{B}$, is the element wise multiplication among the

transformed vectors, then the final result is $\mathbf{c} = INTT(\mathbf{C})$. The vectors are padded, and bit-reversed before the NTT routine is issued, the result is linear order.

5.2.5 Architecture

The complete architecture computes the NTTs, Element-wise multiplication and INTT to evaluate the cyclic matrix multiplication. The multiplier has a shared Data-Path and specific Control Units to perform each step of the Schönhage Stressen algorithm. The Memory is partially shared to save resources. The Control Unit is the most complex part of the multiplier due to the shared elements. There is an additional element to be included: the (Direct/Inverse) Constants Memory Management Unit which consists in two blocks connected to the main Data-Path that posses their own Memory and Control Unit.

The `Complete RAM D`, where the Dense NTT, Element-wise product result and the Dense INTT are stored and computed.

The Memories includes `Starting ROM`, `Sparse RAM` and `Complete RAM S`, these are the support memories to compute the Sparse NTT and where the transform of the sparse vector is finally stored.

The Data-Path has a parallelism of n_u, then this numer of multiplier, referred as `MPY`, and n_u adders and subtractors, referred as `ADD` and `SUB` respectively, are implemented to compute at the same time n_u elements of the transform. The arithmetic units are connected to the memories. The operands have a layer of multiplexers in order to select the correct inputs, these multiplexers are driven by a Control Unit.

The complexity of the Control Unit, with its hierarchi in Figure 5.30, is such that each step of the NTT-based multiplier has to be analyzed separately: the Sparse-NTT, the Dense-NTT and Dense-INTT, the Element wise multiplication. The Control Unit is organized in three levels: the first level controls which step of the multiplier is issued, thus the start signal is sent to the second level of Control Unit and their end signals collected; in the second level the inner steps of the multiplier are computed,tThe last level of control unit is in the `Constant Management Unit`, which controls the twiddle factors in input to the `MPY`.

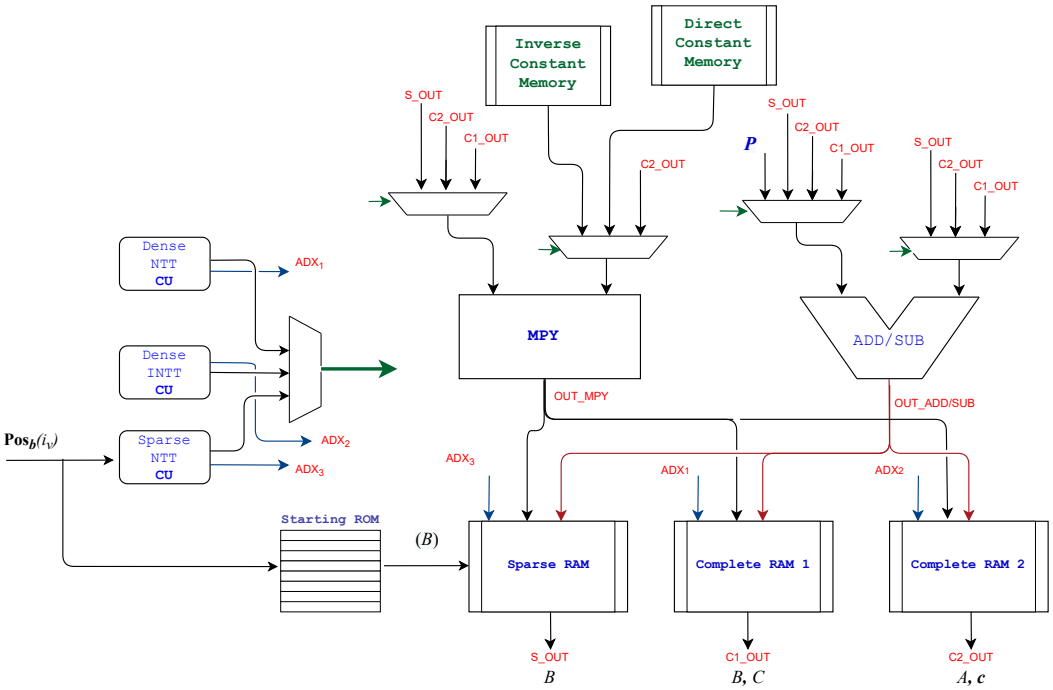


Fig. 5.29 The DataPath shared between the

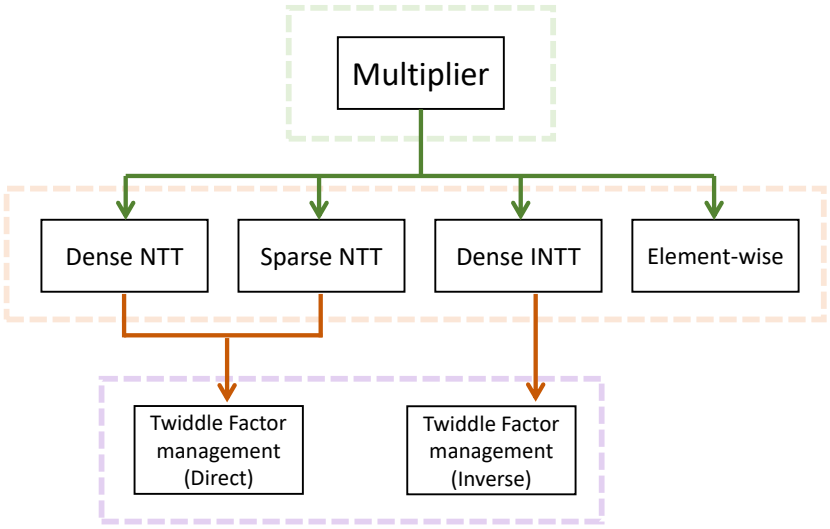


Fig. 5.30 Control Unit hierarchy. The Multiplier is the master CU, while the sub-CU communicate independently with the Twiddle Factor Management CUs.

Sparse-NTT

The Sparse-NTT Control Unit transforms the sparse of a signal up to a defined stage, s_{lim}. The computation, as reported in Algorithm 9, is carried out by transforming the single asserted position in the input signal up to s_{lim}.

Input Sparse/Output Sparse The input is a set of asserted positions (d_v), listed by their position as a binary value and the output is a set of d_v transforms of each position up to stage s_{lim}, where each transform is integer vector of length $2^{s_{lim}}$. The transform of the whole signal, with length n, is a merge merge of the d_v transforms.

Sparse NTT merged to Dense NTT The merge of the Sparse NTT into the Dense NTT transforms the d_v rows of the transform in a vector of length n, which is the transform of a signal at stage s_{lim}. The rows of Sparse NTT points to a specific section of the complete vector, these rows are copied on a section or summed to a preexisting value in that section. The sum is performed thanks to the linearity of the transform.

Sparse Execution The single asserted positions are firstly loaded from Starting ROM (input memory of the Data Path in Figure 5.29), depicted in Figure 5.31, which is a memory with 8 rows and for each of them has the transform of a binary vector with length 8 and weight 1, where the 1 assumes all the 8 positions of the input. For example, in row 2 there is the transform up to stage 3 of vector [0, 0, 1, 0, 0, 0, 0, 0].

$NTT([00000001], \alpha, P)$
$NTT([00000010], \alpha, P)$
$NTT([00000100], \alpha, P)$
$NTT([00001000], \alpha, P)$
$NTT([00010000], \alpha, P)$
$NTT([00100000], \alpha, P)$
$NTT([01000000], \alpha, P)$
$NTT([10000000], \alpha, P)$

Fig. 5.31 Input memory for the Sparse NTT execution

The position of the asserted, $\mathbf{Pos}_a$, input value is in binary form, thus only one row is selected from Starting ROM, these are the 'transform input' to the 8 butterfly units of the Data Path. The next stage, $s = 4$, is computed by requesting the constants to the Direct/Inverse Twiddle Factor Management unit, the multiplications and then addition subtractions are computed. In the end, the content in Sparse-NTT at a given position is updated. The expansion from stage 3 to 4 doubles the size of the vector from 8 to 16, thus from stage 4 to 5 the elements are expanded from 16 to 32, this requires to perform run the cycles `ADD/SUB` and `MPY` two times, the double of the previous stage.

The operations of the Sparse execution involves:

- evaluate the addressed for the Starting ROM;
- evaluate the correct address of the Sparse RAM to be read and then write;
- request the twiddle factors for a given stage and range over n;
- issue the execution of the multiplication and addition/subtraction;

Dense-NTT

The Dense-NTT computation takes in input a vector and schedules the operations to compute the transform, which is then stored in Complete RAM 1. The transform is evaluated from stage $s = 0$ or a given stage, for example $s = s_{lim}$.

The Complete RAM is updated stage after stage, thus the single asserted elements are not expanded as in the Sparse-NTT, but the whole signal is transformed for the next stage $s + 1$. There is only support memory employed to update the transform, this is updated with a parallelism of $n_u = 8$ by evaluating 'sections' of the complete n vector.

The transform of the sparse input is stored in Complete RAM 1, while the transform of the binary vector is stored in Complete RAM 2.

The computation is carried on with a parallelism equal to $n_u = 8$ (the number of `MPY`). The tasks are the following:

- evaluate the addresses of the Complete RAM to be inputs of the DataPah;

- request the set of Twiddle Factors to the Twiddle Factors Unit;
- read and then write the Complete RAM;

The transforms of the input vectors is in memories Complete RAM 1 and Complete RAM 2.

Element-wise multiplication

The intermediate step of the element-wise multiplication enables the connection between `MPY` inputs and the Complete RAMs with the transform of the inputs, the result is evaluates with a parallelism of $n_u = 8$ and stored in Complete RAM (1 or 2), the results replaces the input factors in input of the `MPY`. The result is then transformed wit the Dense-INTT.

Dense-INTT

The Dense-INTT computes the final transform that evaluates the product between the two input vectors.

The strategy is the same as the Dense-NTT, the inverse transform is computed from stage $s = 0$ up to the end. The memory is the Complete RAM, while the twiddle factors are computed by the Inverse Twiddle Factor Management Units.

Twiddle Factors Management

The Twiddle Factor Management Unit is organized in a Control Units and a set of Memories, this block efficiently evaluates the correct set of twiddle factors for the Dense-NTT, Sparse-NTT and INTT. Their value is precomputed and stored in the memories, thus the CU is in charge of efficiently accessing them.

The unit is organized such that the latency to provide the twiddle factors is reduce for the Sparse-NTT and Dense-NTT, since the two algorithms requests the twiddle factors with a different strategy, it was convenient to introduced both modes.

The main structure of the unit is in Figure 5.32, with the memory block organization and connections highlighted. The memory is organized in four blocks, referred as *mod i*, where $i = 0, \ldots, 4$. The organizations reduced the latency for the given

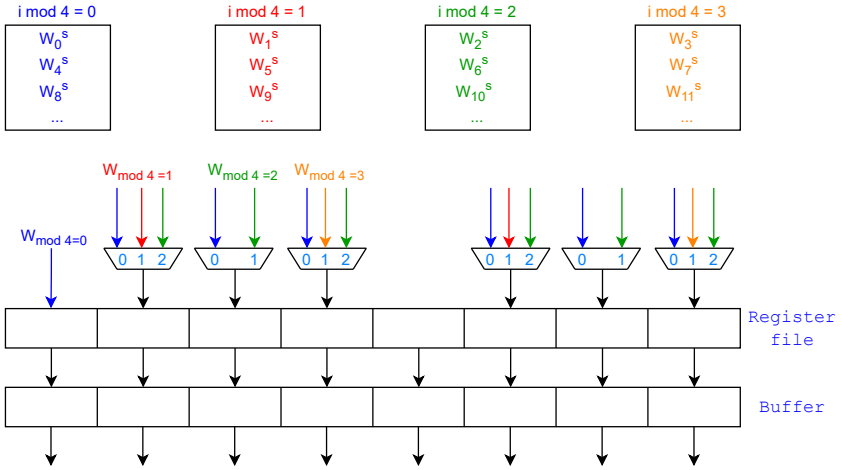


Fig. 5.32 Twiddle Factors Management Units

$n_u = 8$ parallelism. The memories are connected with a layer of multiplexers to Register File and then to a Buffer. The Control Unit has to execute the following steps when a twiddle factor request is issued:

- drive the multiplexers to collect the correct set of twiddle factors;
- fill the RF with the twiddle factors required in the next request;
- fill the BF with the content in RF for the current execution and kept until needed.

The set of twiddle factors present in BF for a NTT evaluation are, in general, required for more than one butterfly evaluation, thus the RF can be filled with the next set of factors required by the NTT.

The organization is efficient for $n_b = 8$, the increase of the parallelism requires to consider a different organization of the twiddle factor memories.

Resource sharing The architecture is meant to implement a NTT-based multiplier and takes advantage of a resource reuse in order to reduce the total area occupation and memory requirement. The DataPath is shared among the Sparse-NTT, Dense-NTT, INTT and the multipliers are shared with the previous tasks and the element wise multiplication. The Twiddle factors management unit with direct constants is

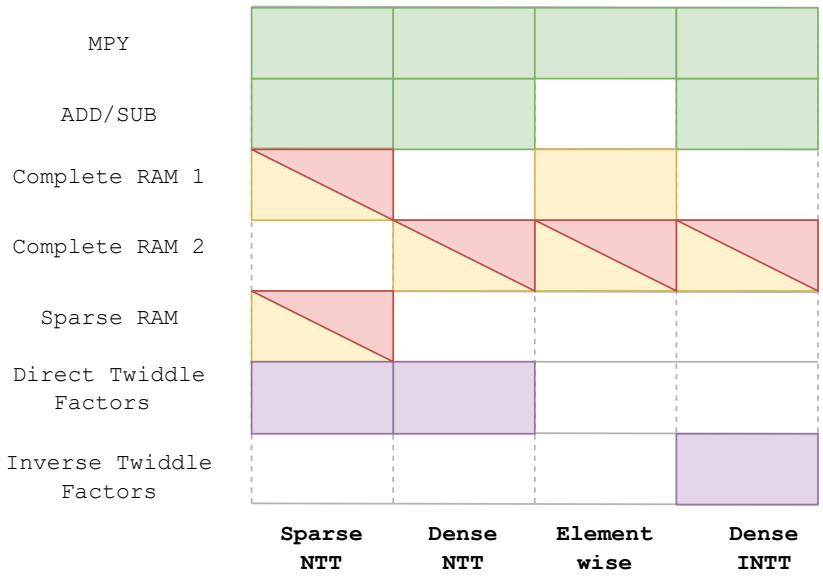


Fig. 5.33 Shared resources in NTT-based multiplier

shared between the Dense and Sparse NTT. The Complete RAMs are shared by all the tasks that are performed.

The resource sharing is summarized in Figures 5.33.

5.3 Results

The NTT-based multiplier has been implemented as an ASIC component, using Synopsys Design Compiler and the UMC 65 nm technology; the same architecture has also been implemented for an Artix-7 200 FPGA target, using Vivado.

5.3.1 ASIC Results

The ASIC synthesis results are reported in Table 5.7 and include the occupied area A, when the clock period is set to $t_{ck} = 5$ ns (column 3), the shortest achievable clock period $t_{ck,min}$ before obtaining a negative slack (column 4), and the occupied area A_f when the shortest clock period is set as a constraint (column 5). The critical path, with the longest combinatorial delay, is found within the element wise multiplier and

it depends on the choice of the modulus P, since P affects the size of the arithmetic units.

The Table also provides the percentage area and delay differences with respect to the synthesis case with clock period set to 5 ns: for a large N', the feasible increment of the clock frequency is higher than the corresponding area penalty.

N'	N	A (μm^2)	$t_{ck,min}$ (ns)	A_f (μm^2)
32,768	12,323	94,315	3.3 (-34%)	128,497 (+36%)
65,536	21,701 24,659	95,543	3.5 (-30%)	113,344 (+18%)
131,072	37,813 40,973 58,171	124,639	3.7 (-24%)	139,719 (+7%)

Table 5.7 ASIC synthesis results for the UMC 65 nm technology. $d_v = 10$. Percentage delay and area values are given with respect to the reference synthesis with constraint $t_{cp} = 5$ ns.

It is worth mentioning that the area occupation is referred to a flexible kind of polynomial multiplier that performs the product of binary vectors and computes the result either in $\mathbb{GF}(2)$ or in the integer field.

5.3.2 FPGA Results

The FPGA synthesis results are reported in Table 5.8 for several choices of the input vector length (N), while the density of the vector is set $d_v = 10$. The critical path has been set to 10 *ns* in order to enable a fair comparison against published implementations, where the same constraint was imposed. It has been verified that the effect of the density on the number of required resources is negligible. In the Table, n' is the length of the input vector extended with *PrimePadding*. Columns 2-6 provide the required number of hardware resources, namely Look-Up Tables (LUT), Flip-Flops (FF), Block RAMs (BRAM), and arithmetic processing units (DSP). It is worth noting that the number of BRAMs increases almost linearly with n', whereas the amount of allocated logic elements (LUTs and FFs) is sub-linear with n' , leading to hardware utilization improvement.

n'	n	LUT	FF	BRAM	DSP	Latency (ms)
32,768	12,323	5396	3774	52	40	2
65,536	21,701 24,659	6259	4053	100	40	4.3
131,072	37,813 40,973 58,171	7062	4358	199	40	9.1

Table 5.8 FPGA resource utilization with Artix-7 200. The operating frequency is set to 100 MHz and $d_v = 10$.

5.3.3 Latency

The multiplier latency is reported in Table 5.9 in terms of number of clock cycles, for a few choices of vector length and sparsity. The last two columns also show the latency value in ms, assuming for each case the corresponding highest clock frequency in the ASIC and FPGA synthesis. From Table 5.9, it can be noticed that the latency scales linearly with n', while it scales logarithmically with the matrix density.

n'	d_v	Clock cycles	Time (ms)	
			ASIC	FPGA
$32,768$	10	207k	0.83	2
	100	226k	0.90	2.2
	1000	244k	0.97	2.4
	$\approx N'$	261k	1	2.6
$65,536$	10	437k	1.7	4.4
	100	474k	1.9	4.7
	1000	512k	2	5.1
	$\approx N'$	565k	5.6	
$131,072$	10	919k	3.4	9.2
	100	996k	3.7	1
	1000	1M	3.7	1
	$\approx N'$	1.2M	4.4	1.2

Table 5.9 The execution time with different length N' and density (d_v). The results are provided in terms of both number of clock cycles and ms, assuming the largest achievable clock frequency.

5.4 Side-Channel Attack on VectorByCirculant

The *VectorByCirculant* architecture has been developed in order to achieve a reduced execution time and area occupation, the execution has been organized such the execution time does not depend on the secret key, the architecture is safe towards attacks that analyzed the total execution time of the multiplier to obtain more information about the matrix the SK. The immunity towards Side-channel attacks has to be studied by considering leakage of information from the dynamic power consumption during the execution of a primitive or its electromagnetic radiations of the device.

The present work analyzes the dynamic power consumption of the post-synthesis design in order to predict possible weaknesses resulting from information leaked by the architecture during the execution of a primitive that involves the Secret Key in the computation, which is the case of the LDPC Decoder in LEDAcrypt/BIKE. The study presents an initial study of the device in charge of evaluating a multiplication, the approach is similar to the work presented in [64], which has been applied to the cryptosystem in [59]. The attack is based on the simulated power consumption of the multiplier, which is later validated considering a real device.

The Side-Channel Attacks based on the power consumption of a device recorded during the execution of a primitive that involves the secret key in the computation, the power consumption is recorded, this is referred as power trace, over a time span and then processed with method in order to retrieved some information. In literature, there are few types of know attacks suitable:

- the Single Power Analysis (SPA), only one power trace is derived,
- the Differential Power Analysis (DPA), analyzes the difference of a set of power traces,

The goal of such studies is to identify a behaviour or pattern that is typical of some secret key. The present research considered an attack which is a further improvement of DPA and SPA, this is the Correlation Power Analysis (CPA) attack, which has been extensively described in [65]. The method is applied to the *VectorByCirculant* multiplier, the power traces resulting from dynamic power consumption (P_{dyn}). The LDPC Decoder employes a multiplier in which an input is the Secret Key, thus both the Syndrome and the Unsatisfied Parity Check vectors computation can be targeted to retrieve $\mathbf{H}$.

The Secret Key is a set of d_H positions, the developed algorithm iteratively selects these position that are *correlated* to the power traces that has been derived. The complexity of the search is linear with the density of the code, with BIKE[18]/LEDAcrypt[17] the value of d_H is 10^2.

The method has two main elements to be set-up:

- the first step is to select a target for the attacks, this is referred as *intermediate value*, this corresponds to the updated result in the *VectorByCirculant* architecture ad has been selected since include the information from the input, which is supposed to be random and the secret key;
- the transitions that occur on the intermediate value are modeled with the Hamming Distance (HD) [65], which measures the difference between two consecutive values.

The model is fundamental for the following steps that are issues, but firstly the power consumption is derived.

The attack is referred as *Correlation Power Attacks* because the modeled power consumption (due to transitions) is correlated with the actual power consumption of the device while a primitive is executed, the set of positions that are correlated to the power consumption are processed and the Secret Key is derived.

Power Traces Derivation

The power consumption is the power consumed by the device during the execution of a multiplication, this is referred as *power trace*, this is obtained by executing a multiplication with `VectorByCirculant` between a dense and sparse vector (the first row of cyclic matrix), these have been obtained with a post-synthesis simulation of the multiplier netlist. The sparse vector, in Code-based PQC is the Secret Key.

The netlist has been synthesized with Synopsys Design Compiler, the node is the UMC CMOS 65 nm technology. The netlist, the Standard Delay Format (.sdf) file are obtained with Synopsys Design Compiler and will feed the developed methodology.

The power trace is derived by running the the *VectorByCirculant* algorithm with the secret key vector as a constant input, while the input message is a random message, the transitions that occurs during the execution of the algorithm in each gate

of the netlist, the switching activity, are derived with QuestaSim by Mentor Graphics. The result is the Value Dump Change (.vcd) file, with contains the information of the switching activity. This corresponds to a single power trace.

The last step involves with Synopsys PrimeTime with the so called Time Based Power Analysis, is executed and the single power trace computed. An example of power trace is shown in Figure 5.34 from [1].

The methodology is organized in various step, since the number of power traces to be computed can be large, the process has been automatized with a Python script in order to run the tools required by each step with a random input dense vector. In particular the `Simulation Script` derives the switching activity with QuestaSim and the `Power Script` derives the simulated power trace of the netlist with Synopsys PrimeTime.

The visual representation of the method in in Figure 5.35

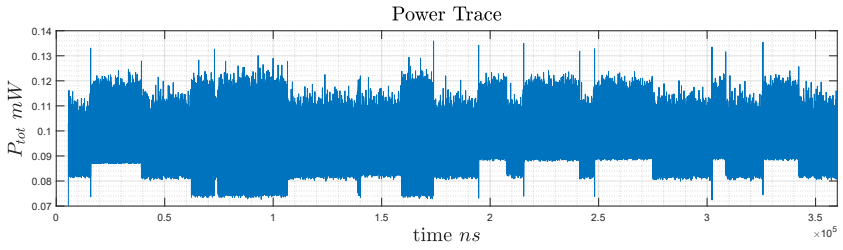


Fig. 5.34 The Power Trace derived with Synopsys Prime time during the execution of a complete product [1].

The power prediction is computed as described in [65] with the hamming distance model. The power prediction is evaluated considering two consecutive values loaded in $\texttt{REG}_C, new$, the values has been selected since it includes both the information on the input vector and on the asserted position (the Secret Key), the consecutive values are organized in two matrices $\mathbf{IV}_1$ and $\mathbf{IV}_2$ (*intermediate values*), which contains over the columns all the possible inputs of the Barrel Shifter, while on the columns all the possible values that the asserted position can assume. The modeled power is then matrix **PT** (*predicted power*) which corresponds to the hamming distance between $\mathbf{IV}_1$ and $\mathbf{IV}_2$.

The correlation combines the predicted power with the simulated power, for each point of **PT**, these are referred as *correlation traces*, **CP**, have been computed with the Pearson's Correlation coefficient described in [65]:

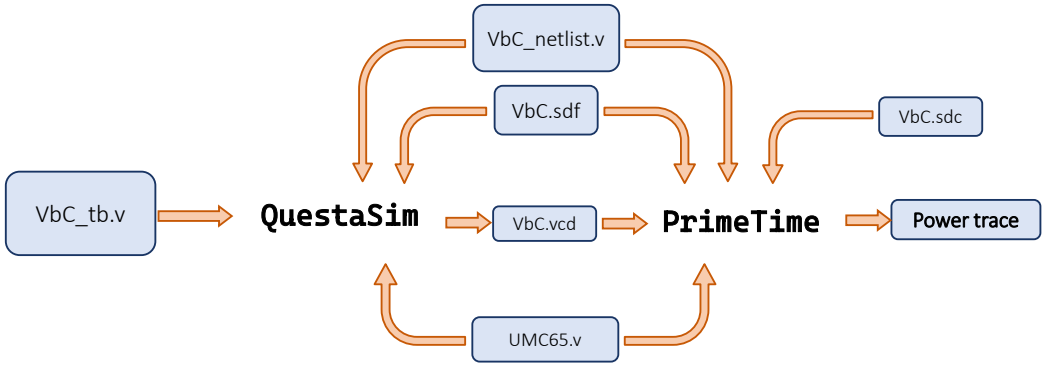


Fig. 5.35 The netlist, .sdf and technological node files are the inputs of both QuestaSim and Prime Time, the .vdc file, containing the information on the switching activity, is derived from a random input provided in the test bench.

$$CP_{i,j} = \frac{\sum_{n=1}^{N} (PM_{n,i} - PM_i) \cdot (PT_{n,j}) - PT_j}{\sqrt{\sum_{n=1}^{N} (\sum_{n=1}^{N} (PM_{n,i} - PM_i)^2 \cdot \sum_{n=1}^{N} (PT_{n,j} - PT_j)^2}} \tag{5.24}$$

The value $PM_{n,i}$ of the modeled power refers to the point n,i of matrix **PM**, while the value PM_i is the average over one row i, thus for a single asserted position. The matrix CP has the rows associated to an asserted position, while the column are associated to the value the input vector can assume.

The Figure 5.36 shows the format of a single correlation trace, over one row of matrix **CP**. The vector shows the presence of a peak, which is referred as the is the *Correlation Peak*.

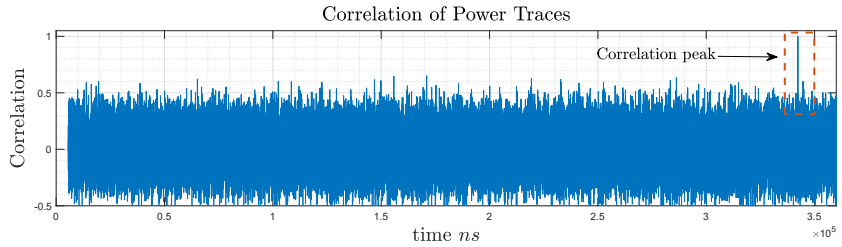


Fig. 5.36 Correlation trace corresponding to asserted value11 in the SK. The dashed box highlights the correlation peak[1].

The peak is, in general, not unique but associated to a set of position, in order to select the correct one an algorithm for the key recovery has to be run.

Secret Key Recovery

The CPA attack involves one last step with an algorithm that selects the position among the candidates and then recovers the whole sparse vector (stored in $\texttt{MEM}_b$, value $C(\mathbf{b})$ in Algorithm 4). In general, the candidate position values coming out from the CPA algorithm are associated to correlation traces that shows a peak with a high correlation, thus close to 1. However, more than one positions for each iteration may be found, this is inherited by the rotational nature of the *VectorByCirculant* algorithm.

In particular, the first position in $C(\mathbf{b})$ requires to select the correct position among various candidates. The correct value has to be computed from the first peak, over the column, that appears, this corresponds to the first peak *in time*. The selection of the second position is simpler, only one trace presents a peak, thus there is only one possible candidate. On the contrary, the third position is selected among different candidate positions, as the first one. This time, one can have two possible scenarios: three or two candidates. In the second case, the candidates are $C(\mathbf{b}(2)$ and $C(\mathbf{b}(3)$. The candidates are three if $C(\mathbf{b}(2) - C(\mathbf{b}(1)$ is a multiple of n_b, with $C(\mathbf{b}(1) < C(\mathbf{b}(2)$. In this case the candidates are $C(\mathbf{b}(1)$, $C(\mathbf{b}(2)$ and $C(\mathbf{b}(3)$, then the position is found by exclusion.

the other condition, the $i-th$ (with $4 \leq i \leq b_H$) asserted bit approximated by the CPA attack corresponds requires to select between $C(\mathbf{b}(i)$ and $C(\mathbf{b}(i-1)$, and the correct position is found by exclusion since the other candidates has already appeared.

In the end, the attack turn out to be successful and the position of the asserted bits in SK is guessed. This makes the *VectorbyCirculant* multiplier not safe against such attacks.

Threshold frequency limit detection

The attack on the architecture is successful for the weaknesses of the implementation and because the operating frequency at which the power consumption has been 'measured' is the maximum achievable. The total power is the sum of static and dynamic power, P_{static} and P_{dyn}, but for high values of operating frequency this is driven by the dynamic power, the same attack is implemented by considering

different operating frequency , this is meant to find the limit of the CPA attack presented in the previous paragraphs. In particular, the goal is to find the minimum operating frequency for which there correlation peak is not evident.

The study on this limit is motivated by the fact that the P_{dyn} of the correlation traces strongly depends on the operating frequency. If the dynamic power has a contribution too small, it is not possible to compute a reliable correlation value because of the limitation in the representation of the measured power.

The results shown in Figure 5.37 from [1] have been obtained guessing the second value of the key and represent the behaviour of the correlation peak of the correct key as a function of the operating frequency.

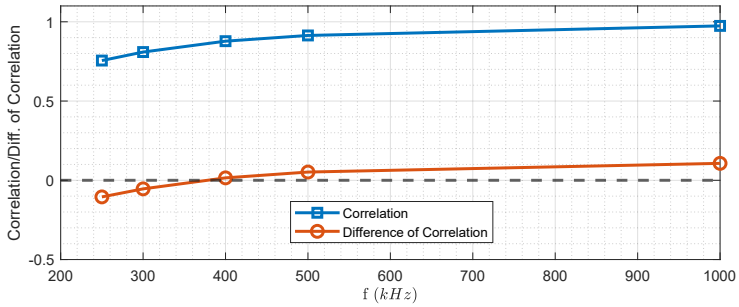


Fig. 5.37 The plot shows the value for correlation peaks associated to correct key value, in blue, and the average value that is assume by the correlation associated to wrong keys.

The blue curve is associated the maximum value the correlation assumes and found in the correct key, this is directly proportional to the frequency, as expected. The orange curve shows the difference between the correlation peak in the correct key and the highest correlation value found from all the wrong key values. If the difference is > 0, the attack is feasible and provided the correct key in SK. Alternatively, if the difference is < 0 the highest correlation value corresponds to a incorrect set of positions. In the Figure, the threshold is around 400 kHz but since the data has been 'measured' from an ideal device, it is clear that in a real case this value would be higher.

5.4.1 Results validation and conclusions

The Simulated power for the CPA attack can be applied also with measured power, then to validate the results obtained previously with 'ideal' measures, the same approach has been followed has been on a real implementation of the multiplier.

VirtLab board [66][67] has been employed to conduct the study. The board is equipped with a Cyclone 10 LP 10CL025YE144C8G FPGA and an STM32L496VET6 microcontroller. The multiplier architecture has been synthesized and then implemented on the Cyclone 10 FPGA, while the microcontroller stimulates the circuit and it can collect the measures of the power consumption of the device, by working as a Digital Storage Oscilloscope (DSO). The maximum DSO sampling frequency is limited to 500 kHz, thus the working frequency of the multiplier to 250 kHz in order to record two samples per clock cycle and to stay below the Shannon limit. The attack, due to such limitations, turned out to be unsuccessful and the SK has not been found, since the maximum working frequency is lower than the previously found threshold. The same study should be carried out with a device that has not such limitations. In the end, the validation consists in applying the same methodology previously explained on samples that are derived from a real device, with the additional difficulty of adopting 'real measurements'.

The proposed method of analysis is useful to prevent the security issues of an architecture during the design phase by identifying the effects of a CPA attack on a PQC multipler. The method can be applied to the whole decoder and for different types of Side-channel attacks. The future work will be dedicated to make the architecture immune to this type of attacks. The development of a secure architecture is at the cost of an increase total area or power consumption or execution: the most common approach is to mask the computations that involve the SK, this way the information that leaks is not the secret one.

Chapter 6

Decryption and Encryption in QC-MDPC Codes for PQC

The multiplier suitable for Code-based Post-Quantum Cryptography with QC-MDPC or LDPC codes is a fundamental part of the Encoder and Decoder algorithms of BIKE and LEDAcrypt schemes.

The multiplier which is adopted mostly in the decoder is the Schoolbook multiplier, while in the Encoder the choice depends on the scheme which is adopted: with PKE the encryption is a dense multiplication, both the vector and the matrix are dense, with KEM encapsulation the multiplication is sparse, thus the use of the Schoolbook multiplier is applied and compared to state-of-art implementations.

The encoder and decoder has additional elements that perform the sum between two vector and evaluates the approximated error vector, the additional arithmetic units are described and designed targeting an efficient decoder/encoder.

Moreover, the multipliers previously described in the previous section are applied to provide an efficient encoder/decoder and a flexible multiplier for a variety of PQC primitives, by optimizing their use in the QC-MDPC schemes.

The chapter is organizes as follows: the encoder, applied to encryption and encapsulation, and the decoder, applied to decryption and decapsulation, are described and their results in terms of area an execution time are provided for both FPGA and ASIC implementation, in the end the NTT-multiplies is applied to Code-based PQC in order to prove its feasibility without a degradation of the overall execution time

and area occupation, when considering the multiplier to be applied to a variaty of cases.

6.1 Encryption and encapsulation

The McElice cryptosystem in LEDAcrypt suite adopts QC-LDPC Codes for Public Key Encryption, the computation involves a cyclic multiplication and a sum. The dense cyclic matrix is $\mathbf{M}_l$, the result of $\mathbf{L}_{n_0-1}^{-1}\mathbf{L}$, the density of such a matrix is $(d_H + d_Q)(d_H + d_Q)$ with parameters in Round 2 from Table 3.1 or $d_H d_H$ with parameters in Round 3 from table 3.2, in both cases is referred as d_M. The `VectorByCirculant`, with binary input and output, unit is applied to compute the product $\mathbf{m}\mathbf{M}_l^T$, the codeword is then summed to the error vector, $\mathbf{e}$, which has been previously derived as a sequence of t positions.

The LEDAkem and BIKE schemes, which are meant for Key Encapsulation Mechanism perform the multiplication by sparse vector $\mathbf{e}$, with weight t, and a dense cyclic matrix and then the result is summed up with same input sparse vector. The `VectorbyCirculant`, with binary input and output, and the sum module are applied.

The implementation, represented in Figure 6.1, has two elements:

- the binary `VectorByCirculant`, only one cyclic multiplication is performed;
- the element wise sum unit, t elements expressed as positions are summed up to the vector.

The total area and resource utilization of the Encryption and Encapsulation is the same, considering the same parallelism (n_b) of the units, since the size of the modules is not affected by the density of the variables. The differences are in the memory occupation, a dense variable has to be store and in the total execution time.

6.1.1 Implementation and Comparison

The Implementation result are provided for different parameters of both the architecture, such as the parallelism n_b, and the scheme parameters, such as the density d_M or t, the length of the block n.

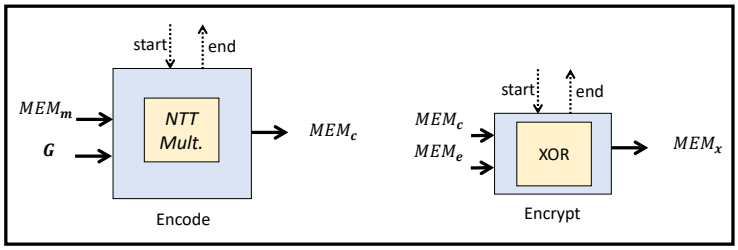


Fig. 6.1 Implementation of the Encryption

6.2 Decryption and decapsulation

The Decryption and Decapsulation for LEDA and BIKE are very similar despite the different scope of such schemes the 'DECODE' step is identical between LEDApkc and LEDAkem, moreover in BIKE a slightly different algorithm is applied but both proposals adopts variants of a Bit Flipping Decoder, the algorithms main tasks computes: Initial Syndrome, Unsatisfied Parity Check, Threshold, Error Approximation, Updated Syndrome.

6.2.1 Decoder analysis

The details of each task and their description in the BGF-Decoder (BIKE) and Q-Decoder (LEDA) are described in the following.

Initial Syndrome Computation The input of the decoder is the vector Syndrome $\mathbf{s}$ of length $(n_0-1)m$, which is $\mathbf{eH}$, a sparse vector by a sparse quasi-cyclic matrix, or $\mathbf{cH}$, a dense vector by a sparse quasi cyclic matrix, with $n_0 = 2$ the equations are:

- KEM

$$\mathbf{s} = \mathbf{e_0}\mathbf{H_0}^T + \mathbf{e_1}\mathbf{H_1}^T \tag{6.1}$$

- PKC

$$\mathbf{s} = \mathbf{c_0}\mathbf{H_0}^T + \mathbf{c_1}\mathbf{H_1}^T \tag{6.2}$$

The options to compute such a product is the binary *VectorByCirculant*, the algorithm is run n_0 times.

The Syndrome computation is the initialization step of the decoder or it is an input, the other task are iteratively computed by the decoders.

Unsatisfied Parity Check The Unsatisfied Parity Check is a vector with length $n_0 m$, referred as **upc**, is computed in both Q-Decoder, it is referred as *Sigma* and *Correlation*, and BFG Decoder, it is referred with function `ctr(H,s,j)`. The computations with $n_0 = 2$ are:

- In BIKE it is computed with

$$\mathbf{upc} = [\mathbf{sH_0}, \mathbf{sH_1}] \tag{6.3}$$

- In BIKE it is computed with

$$\mathbf{upc} = [\mathbf{sL_0}, \mathbf{sL_1}] \tag{6.4}$$

In the Q-Decoder submitted to Round 2 the computation has been carried our in two steps:

- The vector Sigma, σ, of length $n_0 m$, is computed:

$$\sigma = [\mathbf{sH_0}, \mathbf{sH_1}] \tag{6.5}$$

- The **upc** of length $n_0 m$, is computed:

$$\mathbf{upc} = [\sigma_0 \mathbf{Q_0 0} + \sigma_1 \mathbf{Q_1 0}, \sigma_0 \mathbf{Q_0 1} + \sigma_1 \mathbf{Q_1 1}] = [\mathbf{upc}_0, \mathbf{upc}_1] \tag{6.6}$$

The computation is carried out with the integer version of `VectorByCirculant`, the algorithm computes each cyclic product to finally obtain integer vector **upc**. The split into $\mathbf{L} = \mathbf{HQ}$ reduces the latency in computing **upc**, the execution time depends on the overall number of partial products to be computed: the parameters of LEDAcrypt submitted to Round 2 has the number to understand such an advantage, the total number of partial product for $\mathbf{L}$ is $d_L = d_Q * d_H$ (Tables 3.1), while with Q-Decoder the number of partial products becomes $2 * d_Q + d_H$.

Threshold The Threshold function is implemented in both functions as a LUT, despite the functions that calculates such a value are different for LEDAcrypt and BIKE.

In Q-Decoder the threshold function is organized as a LUT, thus a table with two columns: the Syndrome Weight Threshold and the Threshold of the Decoder. The Syndrome Weight is computed and compared to the syndrome thresholds, the corresponding threshold is then provided.

In the BFG-Decoder the threshold is computed with a function that is feed by the iteration number, i in Algorithm 3, and the syndrome weight, s_w, the function has the form:

$$max(\lfloor a * s_w + b \rfloor, c) \tag{6.7}$$

The value of a, b and c depends on the Security Level selected in Table 3.3, the function is then transformed into a LUT similar to the one provided by the Q-Decoder, the threshold is easily evaluated.

It is important to mention that in BIKE two threshold exits: the first one is evaluated with the method proposed above, the second one is $b - \tau$.

Error Approximation The Error Approximation is performed by collecting the position of **upc** which are over a threshold, the set of position is referred as the error vector. The vector can then be applied to correct the ciphertext, for LEDApkc, or is the approximation of the error vector to be recovered in BIKE or LEDAkem.

Syndrome Update The Syndrome Update is performed in two steps:

- the correction on the Syndrome is computed as: $\mathbf{eH}^T = \mathbf{e_0}\mathbf{H_0}^T + \mathbf{e_1}\mathbf{H_1}^T$
- the correction of the Syndrome is summed to the previous value to update is: $\mathbf{s} = \mathbf{s}^{It-1} + \mathbf{eH}^T$

The product $\mathbf{eH}^T$ is implemented with

6.2.2 Decoder Architecture

The element described above are implemented in the complete Data Path of the Decoder, their execution is scheduled by the Control Unit. The first proposed Decoders has been described in [3] and [2], the evolution of the decoder is described and the total execution time and area occupation is provided.

The implementation of the Decoder is described by considering its Memory, DataPath and Control Unit.

Memory

The Memory is organized in memory banks of variable width that is in charge of loading the inputs and outputs of the decoding algorithm and the support variables needed for the computation. The memories are are divided into two groups: memories that stores a vector and memories that stores a set of positions.

The vector memories are:

- $\texttt{MEM}_{\mathbf{s}}$ stores the Syndrome, a binary vector of length m, the memory has size $h \times n_b$, with $h = \lceil m/n_b \rceil$ and n_b the parallelism of the design;
- $\texttt{MEM}_{\mathbf{c_0}}$, ..., $\texttt{MEM}_{\mathbf{c_{n_0-1}}}$ store the ciphertext, a binary vector of length m, the memory has size $h \times n_b$, with $h = \lceil m/n_b \rceil$ and n_b the parallelism of the design;
- $\texttt{MEM}_{\mathbf{upc_0}}$, ..., $\texttt{MEM}_{\mathbf{upc_{n_0-1}}}$ store the unsatisfied parity check, a integer vector of length m, the memory has size $h \times n_b * d_H$ in Tables 3.2 and 3.3, with $h = \lceil m/n_b \rceil$ and n_b the parallelism of the design;
- $\texttt{MEM}_{\sigma_0}$, ..., $\texttt{MEM}_{\sigma_{n_0-1}}$ store the value of σ (in the first two submissions of LEDAcrypt), a integer vector of length m, the memory has size $h \times n_b * d_H$ in Table 3.1, with $h = \lceil m/n_b \rceil$ and n_b the parallelism of the design;

The set of positions are a way to efficiently store binary sparse vectors by storing the asserted elements alone, the position memories are:

- $\texttt{MEM}_{\mathbf{H_0}}$, ..., $\texttt{MEM}_{\mathbf{H_{n_0-1}}}$ store block matrix $\mathbf{H}$, each position assume a value from 0 to $m-1$, the single block memory has size $d_h \times n_m$, with $n_m = log_2 m$ and n_b the parallelism of the design;

- $\texttt{MEM}_{\mathbf{e_0}}, \ldots, \texttt{MEM}_{\mathbf{e_{n_0-1}}}$ store the block vector $\mathbf{e}$, a binary vector of length m, the whole memory has size $t \times n_b$, with $h = \lceil m/n_b \rceil$ and n_b the parallelism of the design;

DataPath

The Data Path has been modified during with the improvements introduced by the LEDAcrypt team and by applying resource reuse, this evolution in the *top view* is provided in Figures 6.2.

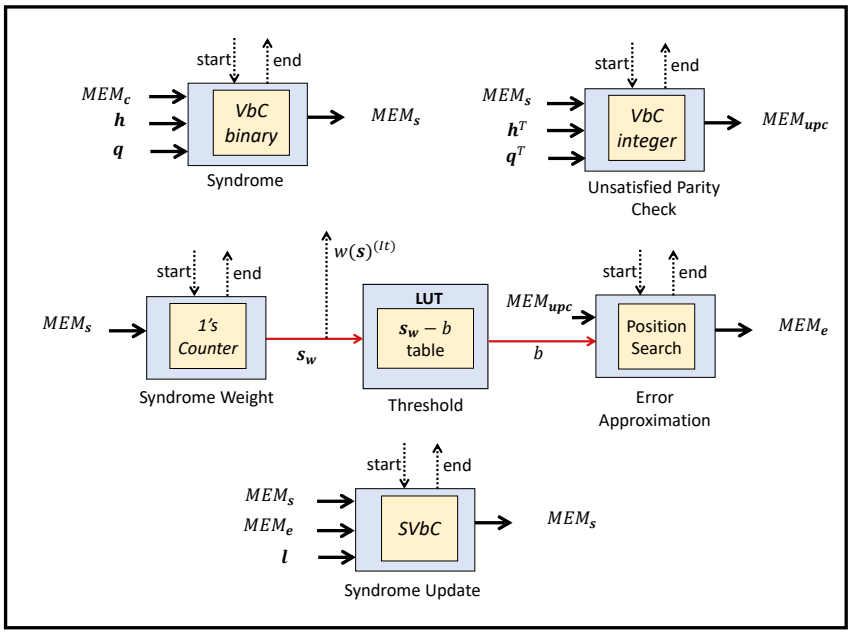


Fig. 6.2 *Version0*:The first version implements LEDAcrypt Decoder submitted to Round 1 of the PQC Competition. the evaluation of *Syndrome*, *Unsatisfied Parity Check* and the `Syndrome Update` is computed with two distinct modules for binary and integer output, the additional elements are in charge of computing the threshold and correct the errors.

The elements that can be identified are the blocks that executes the tasks of the BF-like algorithm:

- binary cyclic multiplication;
- integer cyclic multiplication;
- error position search;

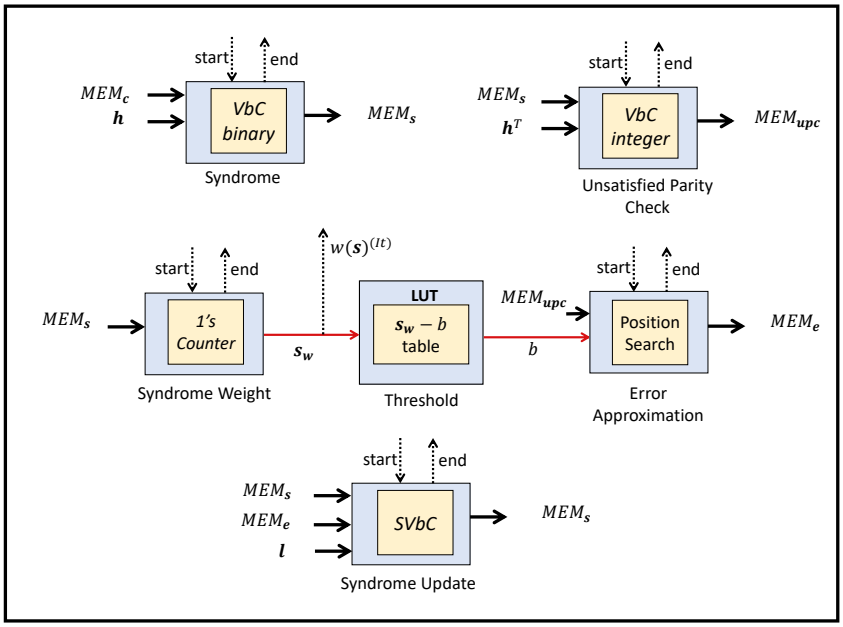


Fig. 6.3 *Version1*:The Round 3 Decoder computes the *Syndrome*, *Unsatisfied Parity Check* and the `Syndrome Update` by means of a single matrix, thus the input matrix is **H**. The other modules remains unchanged.

- bit filliping;

The multiplication can be carried out in various ways, the total are and execution time, which has been published in [3] and [2] are reported here with updated parameters and compared to a pipelined version of the multipliers.

The DataPath is designed with a parallelism of n_b, which is the parallelism of the whole architecture. The length of the vector, the number of blocks and density of the cyclic matrix affects the total execution time and area of the design.

The `VectorByCirculant`, binary and integer, and the `SparseVectorByCirculant` multiplier are present and connected to a set of multiplexer to select the correct inputs from the memories to be multiplied, their output is connected to the memories. The total number of cycles to compute the initial syndrome and the update of the syndrome, the unsatisfied parity check are respectively N_{syn}, N_{synupt} and N_{upc}.

The additional elements that are present are:

- Syndrome Weight module: is connected to $\texttt{MEM}_\textbf{s}$ and each row is read, summed up and updates the value of the syndrome weight, the process takes $2h$ cycles to compute the value, referred as N_{synw};

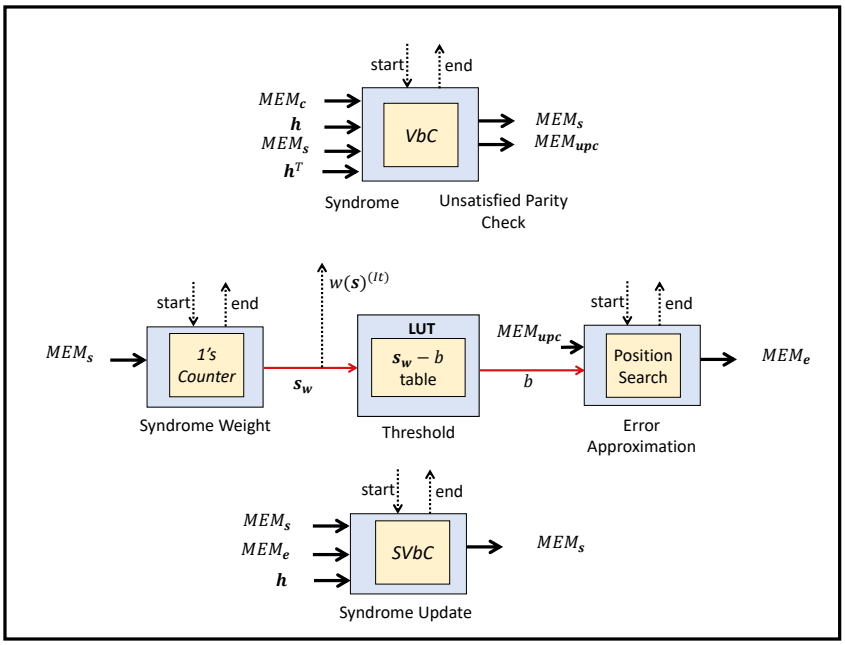


Fig. 6.4 *Version2*:The total area occupation is reduced by implementing the *Syndrome*, *Unsatisfied Parity Check* computation with a single unit, the additional modules are kept unchanged.

- Threshold module: the syndrome weight is compared to the inputs of a LUT, depending on its range the threshold is provided in output as b, the execution is performed in two cycles to access the Table and then provide the threshold in an output register, referred as N_{th}.
- Error Approximation module: the memories $\texttt{MEM}_{\mathbf{c_0}}, \ldots, \texttt{MEM}_{\mathbf{c_{n_0-1}}}$ are read row by row and compared to the threshold b, the asserted bits are then stored as position with the format `ADX`, the row, and `Shift` the over threshold position in a row, the execution is performed $n_b * t + 2 * h$ cycles, referred as $N_{errapprox}$.

The total number of cycles of each unit is calculated with the parameters n_b and the values in Tables 3.1, 3.2 and 3.3.

Control Unit

The Control Unit schedules the execution of each module in the orders suggested by the decoding algorithm. The main task is to provide the starting signal to each unit and then collect their end signal, in addition the execution of the complete

decoding algorithm is provided once two (or only one) conditions are met: the value of syndrome weight is 0 or the maximum number or iterations, It_{max}, is reached, with the BGF Decoder this condition alone is enough to stop the execution.

The total number of cycles is the sum of the total number of cycles of each task, since the decoders are not identical and there two alternative algorithms exists to compute the syndrome update, **eH**, these are referred as:

- *Version 1* computes the initial syndrome and the unsatisfied parity check with `VectorByCirculant Pipelined`, N_{syn}^{VbC}, while the syndrome update with `SparseVectorByCirculant Pipelined`, N_{synupt}^{sVbC};
- *Version 2* computes makes an efficient use of `VectorByCirculant Pipelined`, N_{syn}^{VbC},and `SparseVectorByCirculant Pipelined`, N_{synupt}^{sVbC}, to have a reduced execution time given the parallelism.

The total number of cycles is reported for Q-Decoder considering parameters $n_0 = 2$, $m = 10,853$, $d_H = 71$, $t = 133$ and $It_{max} = 6$, that are derive from Table 3.2.

The decoder, in one iteration, counts for the tasks are:

$$N_{it,V1} = N_{synupt}^{sVbC} + N_{upc} + N_{synw} + N_{th} + N_{errapprx} \tag{6.8}$$

$$N_{it,V2} = N_{synupt}^{VbC} + N_{upc} + N_{synw} + N_{th} + N_{errapprx} \tag{6.9}$$

The total number of cycles of the Decoder is then:

- the Decoder Version 1 counts

$$N_{dec,V1} = N_{syn}^{VbC} + \sum_{It=0}^{It_{max}} N_{it} \tag{6.10}$$

- the Decoder Version 2 counts

$$N_{dec,V2} = N_{syn}^{VbC} + N_{synupt}^{sVbC} \sum_{It=0}^{It_{max}-1} N_{it} \tag{6.11}$$

The distinction between Version 1 and Version 2, as pointed out in [**?**],in the Q-Decoder is motivated advantage of `SparseVectorByCirculant` over `VectorByCirculant`

only under some conditions: the parallelism and the density of the input vector. The condition to be considered is $N^{VbC} > N^{sVbC}$, by defining the density of the error vector at iteration It is d_e^{It}, it is easy to evaluate that `SparseVectorByCirculant` is more efficient than `VectorByCirculant` if d_e^{It} is:

$$h + 3d_h * h > d_e^{It} + d_h + 6 * d_h * d_e^{It} \rightarrow d_e^{It} < \frac{h + 3d_h * h - d_h}{(1 + d_H))} \tag{6.12}$$

The simulation of the Q-Decoder showed that the condition is satisfied with $n_b \geq 64$ for $It \geq 3$, thus the `SparseVectorByCirculant` has been used in the decoder under this conditions.

The total number of cycles that takes a single task (for It_{max} iteration) considering the parameters reported previously, parallelism $n_b = 8$ and *Version1* is:

- $N_{syn} = n_0 * N^{VbC} = 2 * 6 * 10,853/8 + 71 * 10,853/8 = 112600$
- $N_{upc} = N_{syn} * It_{max} = 563000$
- $N_{synw} + N_{th} = 2 * 10,853/8 + 2 = 2715$
- $N_{errorapprox} = 2 * 2 * 10,853/8 + 8 * 133 = 6490$
- $N_{synupt} = 10 * 71 * 2 + 71 * 2 * 133 = 20306$

The breakdown clearly shows the presence of an efficient multiplier for cyclic product introduces a huge benefit in the overall cycle count. The increase in n_b and the option of have Version 2 further improves the execution time. The reduction of the execution time depends on the total number of cycles and the critical path of the architecture, these depends on the actual synthesized architecture. The discussion on critical path, total area and latency is provided in the following.

The introduction of the pipelined multipliers (the coarse grained ones) can introduce a reduction in the execution time of the variables, the benefit of the approach is provided in the results discussion.

6.2.3 Implementation Results

The implementation results are provided for the decoder architecture presented in the previous subsection, the multiplier selected in each version is specified since there

are three options for *VectorByCirculant* and two options for *SparsevectorByCirculant*. The following tables are provided to understand the evolution of the Decoder in terms of total are and execution time.

The very first implementation of the decoder has been described in [3], it refers to *Version0* in Figure 6.2 with a non logarithmic *VectorByCirculant*, thus the results clearly shows a poor scaling with the parallelism. The increase on the total area is highlighted with % of overhead with respect to the previous value of n_b, despite the total execution time is linear with n_b, doubling the parallelism the latency is halved, the area between $n_b = 32$ and $n_b = 64b$ becomes four times higher.

Table 6.1 ASIC synthesis of the whole decoder with the UMC 65 nm technology, the results refers to a block length of $m = 27,779$ (LEDAcrypt parameters submitted to Round 1). The reference architecture is in [3].

	n_b			
	8	**16**	**32**	**64**
t_{cp} (ns)	3.72	3.72	3.72	3.86
Area (μm^2)	15,618	25,380 (62%)	44,493 (75%)	129,047 (190%)
P_{static} (mW)	1.11	1.73	2.98	5.52
Latency (ms)	6	3.3	1.9	1.2

The total area for *Version0* of the Decoder is reported in Figure 6.5 with a bar plot to identify the rate of the scaling of the total area for the single units. The total area with a logarithmic unit in *VectorByCirculant* scales linearly with the increase of n_b.

The synthesis is carries out with Synopsys Design Compiler with the STM FDSOI 28 nm technological node and referred to parameters of LEDAcrypt submitted to Round 2

the total execution is not affected by the choice of the logarithmic unit, since the scaling with n_b remains the same. The benefit is introduced by the use of `VectorByCirculant` to update the Syndrome when this is more convenient then *SparsevectorByCirculant*. The improvement on the latenncy introduced by this choice is in Figures 6.6: the latency is reduced for 128 and 256 bits of parallelism.

The major degradation in the Total execution time comes from the transition from Round 2 to Round 3 (with LEDAcrypt parameters), the evaluation of the *Syndrome*

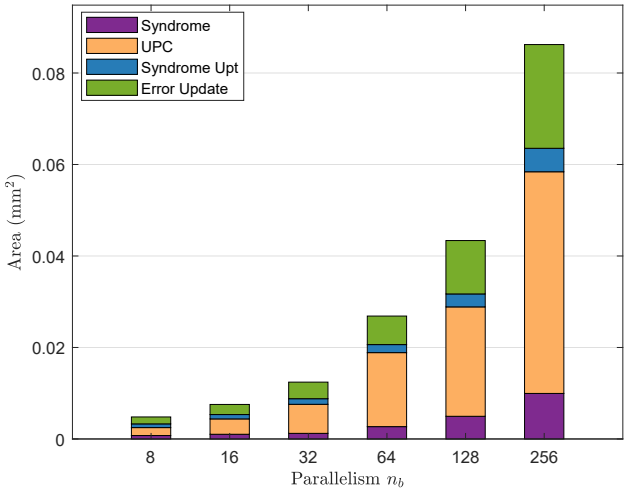
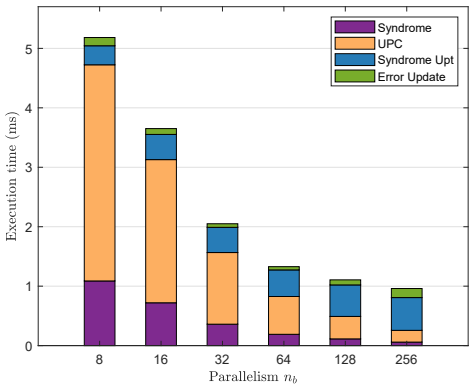
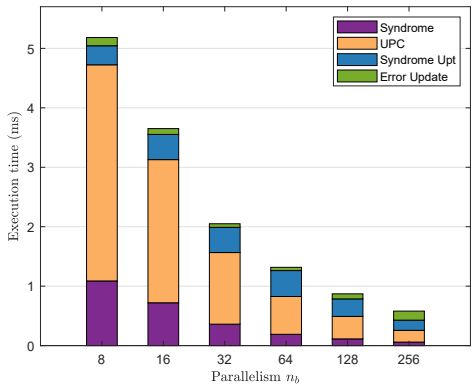


Fig. 6.5 The total area occupation of the *Version0* Decoder with the partial area occupation of the main units[2].



(a) *SparseVectorByCirculant* for Syndrome update



(b) *VectorByCirculant* and *SparseVectorByCirculant* for Syndrome update

Fig. 6.6 The comparison of latency in *Version0* Decoder with and without the the use of *VectorByCirculant* to update the Syndrome

and *Unsatisfied Parity Check* is with matrix **H** with an increased density, the use of pipelined multipliers (*c-pipe*) reduces the total execution time, with a negligible increase in the total area. The comparison of the total execution time with such conditions is in Figure 6.7: the blue line corresponds to the latency of the Q-Decoder submitted to Round 2, the latency strongly increased for the parameters of Round 3 (red line), since the density of the matrices is increased, the total execution time can be reduced with the use of *c-pipe* version of the multipliers (*VectorByCirculant*

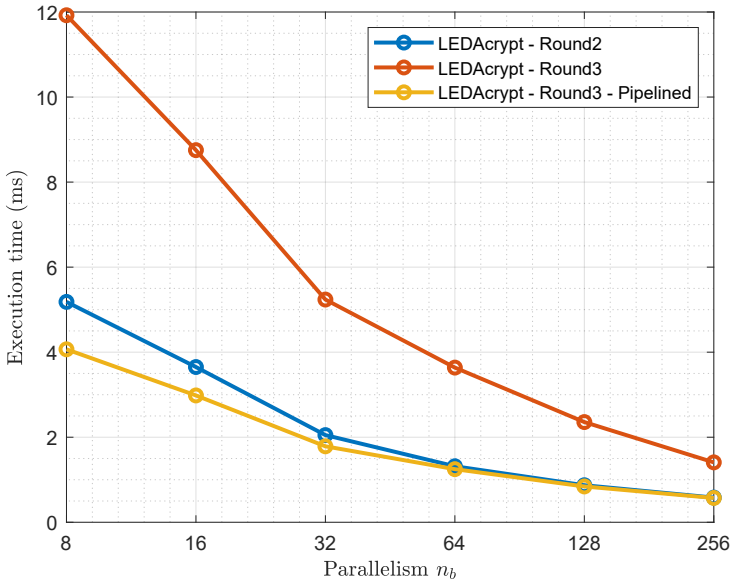


Fig. 6.7 Latency of the Q-Decoder comparison.

and *SparseVectorByCirculant*). The total area has no changes between *Version0* and *Version1*.

The use of the *c-pipe* VectorByCirculant is preferred over the *f-pipe* VectorByCirculant because the complete decoder does not benefit from the reduction of the critical path, thus the resulting decoder has a bigger area and similar total execution time.

6.3 Flexible-NTT in Code based Post Quantum Cryptography

The NTT-based multiplier perform the cyclic multiplications of the LEDAcrypt/BIKE Encoder and Decoder, despite being optimized to perform efficiently sparse computations, the decoder can be further optimized in order to reduce the latency.

The NTT-based multiplier is not intended to optimize the sparse cyclic multiplication, since the type of vectors involved has more efficient options. The present study is meant reduce the latency of the NTT in a Code-based PQC Decoder in order to allows the adoption of the NTT module to multiple PQC primitives. The result

will be provided considering both the sparse and dense multiplication that may arise in LEDAcrypt/BIKE.

6.3.1 Encoder

The multiplier is applied as it is to the $\mathbf{m}\mathbf{M}_l^T$.

The difference between PKC and KEM schemes is in the sparsity of vector $\mathbf{m}$, the NTT-based multiplier execution time similar between a sparse and a dense vector, as can be seen in Table 5.9.

The NTT-based multiplier shows its efficiency for dense multiplication of LEDAcrypt PKC, the present approach when compared, in Table 6.2, to the state of the art of multipliers shows comparable results.

	Latency / LUTs utilization			
n	NTT-mult.	[30]	[28]	[29]
12,323	261k / 5396	1k / 65k	3.5M / 1327	1.1M / 4700
24,659	565k / 6259	3k / 65k	7M / 1327	9.5M / 4400
37,813	1.2M / 7062	5.6k / 65k	11M / 1327	44M / 2800

Table 6.2 QC-MDPC encoding latency in terms of number of cycles vs LUTs utilization.

There is an additional set of results, proposed for BIKE KEM [27], which is not includes since since it implements a sparse multiplication.

The comparison of the Encoders shows that the approach followed in [30] outperforms the NTT and Schoolbook approaches [28] [29]. The final comparison, which includes results presented in this work are provided at the end of the chapter.

6.3.2 Decoder

The NTT-based products in the decoder can be arranged by exploiting the properties of the convolution, with the purpose of reducing the overall complexity. The decoding Algorithm 1 includes two multiplications in sequence, the $\mathbf{s} = \mathbf{x}\mathbf{H}^T$ (Syndrome) and $\mathbf{upc} = \mathbf{s}\mathbf{H}$ (Unsatisfied Parity Check). It is clear that to use the NTT-multiplier *as it is* the value of $\mathbf{x}$ and $\mathbf{h}$ are transformed, then multiplied and the transform of S

is obtained and transformed into **s** and then again into S to calculate **upc**, thus one transform can be saved by rearranging the computation of S^{It} as:

$$S = NTT(\mathbf{x}) \odot NTT(\mathbf{h}),$$

The calculation of upc is then

$$\mathbf{upc} = INTT(S \odot NTT(\mathbf{h}^T)).$$

The vectors $\mathbf{h}$ and $\mathbf{h}^T$ are the first row of matrices $\mathbf{H}$ and $\mathbf{H}^T$, respectively.

The total execution time is reduced by avoiding one inverse and direct transform computation.

The vector **upc** has to be transformed in the time domain, since the error position search is performed by selecting the values over the threshold, there is no efficient way to perform the same operation in the number (or frequency domain). The value of the threshold at a given iteration It is computed from the syndrome weight, s_w, since the value is not available this is deirved in the number domain by recallig that $S(0)$ (this is the first them of the transform), associated to $\alpha^{i \cdot 0}$ from Equation (5.13), thus it is the sum of all values in **s**.

The second optimization is to apply the Sparse-NTT Algoritm to error vector, e^{It},which is transformed into E^{It}.

The syndrome vector is then updated in the number domain by exploiting the linearity of the NTT transform, the updated syndrome is, at a given iteration It, is carried out by transforming only vector e^{It}:

$$S^{It+1} = S^{It} + E^{It} \bmod P \tag{6.13}$$

Then, as in the decoder Algorithm 1, after the check on the syndrome weight a new iteration of the procedure starts.

The optimizations exploits the properties of the NTT to reduce the latency, this is evident from first iteration due to the rearrangement of the operation, in the following iterations the execution time is further reduced: the transforms of $\mathbf{h}^T$ and $\mathbf{h}$, have

been already computed during the first iteration and one Sparse-NTT transform is enough.

The resulting latency of the decoding, with the computations performed in the frequency and time domain, is in the third column of Table 6.3, this is referred as total number of number of cycles. The last columns give the same information of the state of the art implementation that came up in literature, while the last row shows provided the resource utilization for LUTs usage.

It	Step in n_b Step in BF	Total Clock Cycles						
		NTT	[28]	[29]		[27]		
		8	8	32	128	32	64	128
1^{st} It	$\mathbf{s}^0$ $\mathbf{upc}^0$ $\mathbf{e}^0$	524k	346k	32k	2k	325k	100k	38k
$It \geq 2^{nd}$	$\mathbf{s}^1$ $\mathbf{upc}^1$ $\mathbf{e}^1$	236k	346k	32k	2k	325k	100k	38k
LUT		5396	1327	4480	6k	9k	16k	30k

Table 6.3 The clock cycles required in each iteration of the decoder.

In Table 6.3, the total numbers of clock cycles reported in [28] and [29] have been scaled to have $m = 12,323$ and $d_H = 100$, using the formulas provided in the two works. When comparing the different decoder implementations, one can see that the proposed solution achieves a similar latency as [28] and [27] ($n_b = 32$), while it is much slower than [29]. In terms of LUTs, the proposed decoder is better than [27], very similar to [29] and more expensive than [28].

We now consider the implementation of the cyclic multiplier in the context of a complete system, able to perform both encoding and decoding, and supporting both binary and integer formats. To compare the alternative implementations, we introduce the *LC* figure of merit, defined as a latency-complexity product, where the complexity C is given by the global number of LUTs allocated in the synthesis of both encoder and decoder, while the latency L is the execution time (in s) evaluated for encoder and decoder, assuming for the case $m = 12,323$, $d_H = 100$, a clock frequency of 100 MHz, and $It_{max} = 6$ (when possible the numbers are scaled to

match these requirements). The number of cycles in [29] for the encoder has been derived by scaling the total execution time of the decoder to match the dense polynomial multiplication requires, similarly to what has been done in Table 6.2.

m	LC			
	NTT-PQC	[28]†	[29]†	[27]*
12,323	112	786	324	92
24,659	282	1044	1605	N/A
37,813	677	7352	2984	N/A

*Sparse-dense multiplication in the encoder.
† The execution times of the encoder and decoder have been scaled to the nearest N, while the number of LUTs is kept constant for similar values of N.

Table 6.4 LC figure of merit for combined encoder and decoder: comparison between the proposed solution and state-of-the-art solutions.

Although the proposed NTT-based multiplier is not the best option when comparing standalone arithmetic units, the results in Table 6.4 show that it is more efficient than the other implementations proposed in the literature, when the unit is used in a complete application, where the same component must be exploited for multiple products in the encoding and decoding stages, which are characterized by different data formats and computational structures. As seen from Table 6.4, the advantage in terms of *LC* product over the alternative approaches grows quickly with the size of the problem: when N moves from 12,323 to 37,813, *LC* is 5 times the initial one. Considering the same N our proposal is almost 7 times more efficient when compared to [28] and 3 times more efficient than [29]. Moreover, the increase of N takes advantage of the Schonhage Stressen algorithm and makes out proposal even more efficient. The reported *LC* figure for [27] is better than all the other ones in Table 6.4; however, this comparison is not fair, because the encoding in [27] involves a product between a sparse vector and a dense vector, which drastically simplifies the computation with respect to the considered applications. Despite the good *LC* results that the NTT based approach reaches, the other works showed that the architecture fits also in low end FPGA, which has not been proved for our approach.

Chapter 7

Conclusions

The thesis explored the efficient implementation of Code-based Post-Quantum Cryptography primitives for BIKE and LEDAcrypt encoder and decoder. The first architecture that has been developed considered the Schoolbook algorithm to implement the Encoder in the Encapsulation/Encryption primitive and the Decoder in the Decapsulation/Decryption primitive for BIKE and LEDAcrypt. The Decoder developed for LEDAcrypt submitted to NIST Round 2 is the result of a trade-off between the area and execution time; the key feature of the proposed approach to improve the efficiency is the scalability. This makes the architecture able to adapt to different constrains (of resources) that are required by the application, in the comparison it is clear that the approach is the most efficient both for low-end FPGA and high-end ones. The efficiency has been reduced when considering NIST Round 3 submission, but the pipelined approach allowed to to keep a bounded total area.

The multiplier is the core of the Encoder and Decoder, since it drives the latency and total area, its implementation is critical for the complete architecture, but it has been shown that is can also be the target of attacks that tries to retrieve the secret key. A side-channel attack and methodology has been applied to the power consumption of the architecture to study its immunity towards such an issue and if some countermeasures need to be applied.

The last point that has been addressed is the option to use the same multiplier for multiple Post-Quantum primitives, for example considering it as an accelerator. The use of the Schönhage-Stressen algorithm in Code-based PQC has been studied, by proposing an NTT-based multiplier that is tailored to the BIKE and LEDAcrypt

specification, with an acceleration for sparse variables. The resulting multiplier does not improve the area-latency trade off of the decoder that has been found with Schoolbook, but with few rearrangements in the decoding algorithm the approach improves the state of the art. In the end, the NTT-based multiplier results suitable and efficient to be applied to a wide range of PQC-primitives.

The future work can include the study of hybrid architectures, with a combination of NTT-based and Schoolbook multipliers, to propose a better latency-trade off area. In addition, the use of a single accelerator associated to a processors for PQC-primitives has to be extensively studied.

References

[1] Kristjane Koleci, Lorenzo Cecchetti, Guido Masera, Maurizio Martina, and Massimo Ruo Roch. A side channel attack methodology applied to code-based post quantum cryptography. In *Applications in Electronics Pervading Industry, Environment and Society: APPLEPIES 2022*, pages 90–96. Springer, 2023.

[2] Kristjane Koleci, Paolo Santini, Marco Baldi, Franco Chiaraluce, Maurizio Martina, and Guido Masera. Efficient hardware implementation of the ledacrypt decoder. *IEEE Access*, 9:66223–66240, 2021.

[3] Kristjane Koleci, Marco Baldi, Maurizio Martina, and Guido Masera. A hardware implementation for code-based post-quantum asymmetric cryptography.

[4] Richard P Feynman. Simulating physics with computers. In *Feynman and computation*, pages 133–153. CRC Press, 2018.

[5] M. Giles. IBM new 53 qubit quantum computer is the most powerful machine you can use. *MIT*, 2018.

[6] J. Hsu. Ces 2018: Intel's 49-qubit chip shoots for quantum supremacy. *IEEE Spectrum Tech Talk*, 2018.

[7] J. Porter. Google confirms 'quantum supremacy' breakthrough. *VERGE*, 2019.

[8] Frank Arute, Kunal Arya, Ryan Babbush, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 2019.

[9] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[10] Daniel J Bernstein. *Introduction to post-quantum cryptography*. Springer, 2009.

[11] National Institute of Standards and Technology. Post-Quantum Crypto Project. http://csrc.nist.gov/groups/ST/post-quantum-crypto/. Accessed: 2023-10-16.

[12] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[13] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN Progress Report*, 44:114–116, 1978.

[14] Nist post-quantum cryptography round 4 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-4-submissions.

[15] D. J. Bernstein, Academia Sinica C. Cid T. Chou, J. Gilcher, T. Lange, V. Maram, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, N. Sendrier, J. Szefer, C. J. Tjhai, M. Tomlinson, and W. Wang. Classic Mceliece website. https://classic.mceliece.org/.

[16] Marco Baldi, Franco Chiaraluce, and Marco Bianchi. Security and complexity of the McEliece cryptosystem based on Quasi-Cyclic Low-Density Parity-Check codes. *IET Information Security*, 7(3):212–220, 2013.

[17] M.Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini. Ledacrypt home.

[18] N.Aragon, P. L. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.C. Deneuville, P. Gaborit, S. Ghosh, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, J. Richter-Brockmann, N. Sendrier, J.P. Tillich, V. Vasseur, and G. Zémor. BIKE website. https://bikesuite.org/.

[19] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J. Bos, J.C. Deneuville, A. Dion, P. Gaborit, J. Lacan, E. Persichetti, J.M. Robert, P. Véron, and G. Zémor. HQC website. http://pqc-hqc.org/.

[20] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[21] T.J. Richardson and R.L. Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *IEEE_J_IT*, 47(2):599–618, 2001.

[22] Daniel Apon, Ray Perlner, Angela Robinson, and Paolo Santini. Cryptanalysis of LEDAcrypt. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 389–418, Cham, 2020. Springer International Publishing.

[23] Yufei Xing and Shuguo Li. A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):328–356, Feb. 2021.

[24] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. Crystals-Kyber. https://pq-crystals.org/kyber/resources.shtml.

[25] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[26] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, September 1971.

[27] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding bike: Scalable hardware implementation for reconfigurable devices. *IEEE Transactions on Computers*, 71(5):1204–1215, 2022.

[28] Kristjane Koleci, Paolo Santini, Marco Baldi, Franco Chiaraluce, Maurizio Martina, and Guido Masera. Efficient hardware implementation of the ledacrypt decoder. *IEEE Access*, 9:66223–66240, 2021.

[29] Davide Zoni, Andrea Galimberti, and William Fornaciari. Efficient and scalable fpga-oriented design of qc-ldpc bit-flipping decoders for post-quantum cryptography. *IEEE Access*, 8:163419–163433, 2020.

[30] D. Zoni, A. Galimberti, and W. Fornaciari. Flexible and scalable fpga-oriented design of multipliers for large binary polynomials. *IEEE Access*, 8:75809–75821, 2020.

[31] K. Millar, M. Łukowiak, and S. Radziszowski. Design of a flexible schönhage-strassen fft polynomial multiplier with high- level synthesis to accelerate he in the cloud. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5, 2019.

[32] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[33] Victor S Miller. *Use of elliptic curves in cryptography*. Springer, 1986.

[34] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[35] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pages 365–390. 2022.

[36] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[37] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[38] Nist post-quantum cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography.

[39] Nist post-quantum cryptography round 1 submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions.

[40] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. Crystals-Kyber website. https://pq-crystals.org/kyber/index.shtml.

[41] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. Crystals-dilithium website.

[42] Falcon Team. Falcon website.

[43] J.P. Aumasson, D.J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, M. Kudinov, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, and B. Westerbaan. Sphincs website. https://sphincs.org/.

[44] Nist post-quantum cryptography selected algorithms. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[45] Call for post-quantum cryptography: Digital signature schemes. https://csrc.nist.gov/Projects/pqc-dig-sig.

[46] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Hutchinson, A. Jalali, K. Karabina, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik. SIKE website. https://sike.org/.

[47] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Hutchinson, A. Jalali, K. Karabina, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik. Sike note. https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/sike-team-note-insecure.pdf.

[48] Jeffery Hoffstein. Ntru: a new high speed public key cryptosystem. *presented at the rump session of Crypto 96*, 1996.

[49] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Thv*, 4244:114–116, 1978.

[50] David Elkouss, Anthony Leverrier, Romain Alléaume, and Joseph J Boutros. Efficient reconciliation protocol for discrete-variable quantum key distribution. In *2009 IEEE international symposium on information theory*, pages 1879–1883. IEEE, 2009.

[51] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Contr. Inform. Theory*, 15(2):157–166, 1986.

[52] Vladimir Michilovich Sidelnikov and Sergey O Shestakov. On insecurity of cryptosystems based on generalized reed-solomon codes. 1992.

[53] Yuan Xing Li, Robert H Deng, and Xin Mei Wang. On the equivalence of mceliece's and niederreiter's public-key cryptosystems. *IEEE Transactions on Information Theory*, 40(1):271–273, 1994.

[54] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. Classic mceliece: conservative code-based cryptography. *NIST submissions*, 2017.

[55] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[56] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt: QC-LDPC code-based cryptosystems with bounded decryption failure rate. In Marco Baldi, Edoardo Persichetti, and Paolo Santini, editors, *Code-Based Cryptography*, pages 11–43, Cham, 2019. Springer International Publishing.

[57] P. Santini, M. Battaglioni, M. Baldi, and F. Chiaraluce. Analysis of the error correction capability of LDPC and MDPC codes under parallel bit-flipping decoding and application to cryptography. *IEEE Transactions on Commmunications*, 68(8):4648–4660, 2020.

[58] Ciara Rafferty, Máire O'Neill, and Neil Hanley. Evaluation of large integer multiplication methods on hardware. *IEEE Transactions on Computers*, 66(8):1369–1382, 2017.

[59] Tung Chou. Qcbits: constant-time small-key code-based cryptography. In *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 280–300. Springer, 2016.

[60] Charles M Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

[61] Leo Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970.

[62] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[63] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology—CRYPTO'86: Proceedings*, pages 311–323. Springer, 2000.

[64] Alessandro Barenghi, Guido Bertoni, Fabrizio De Santis, and Filippo Melzani. On the efficiency of design time evaluation of the resistance to power attacks. In *2011 14th Euromicro Conference on Digital System Design*, pages 777–785, 2011.

[65] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[66] Massimo Ruo Roch and Maurizio Martina. vrLab: A Virtual and Remote Low Cost Electronics Lab Platform. In Sergio Saponara and Alessandro De Gloria, editors, *Applications in Electronics Pervading Industry, Environment and Society*, pages 213–220, Cham, 2021. Springer International Publishing.

[67] Massimo Ruo Roch and Maurizio Martina. Virtlab: A low-cost platform for electronics lab experiments. *Sensors*, 22(13):4840, Jun 2022.