POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

DESIGN AND IMPLEMENTATION OF A QC-MDPC CODE-BASED POST-QUANTUM KEM TARGETING FPGAS

Doctoral Dissertation of:
Andrea Galimberti

Supervisor:
Prof. William Fornaciari

Co-supervisor:
Prof. Davide Zoni

Tutor:
Prof. Francesco Amigoni

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

Year 2022 – Cycle XXXV

Abstract

Quantum computing is expected to break the traditional public-key cryptography solutions in the upcoming decades, making it paramount to design new security solutions that can also resist attacks carried out by quantum computers. Post-quantum cryptography aims to design cryptoschemes that can be deployed on traditional computers and resist both traditional and quantum attacks. The deployed post-quantum cryptography solutions will have to satisfy not only security requirements but also performance ones. Providing effective hardware support for such cryptosystems is indeed one of the requirements set by NIST within its ongoing post-quantum cryptography standardization process, and it is particularly crucial to ensuring a wide adoption of post-quantum security solutions across embedded devices at the edge.

This thesis delivers a configurable FPGA-based hardware architecture to support BIKE, a post-quantum QC-MDPC code-based cryptoscheme. In particular, BIKE implements a key encapsulation mechanism, i.e., a cryptoscheme that generates and shares a symmetric key between two parties by employing a public-private key pair. The proposed architecture aims to improve performance over the existing state-of-the-art software and hardware solutions, and it is configurable through a set of architectural and code parameters, which make it efficient, providing good performance while using the resources available on FPGAs effectively, flexible, allowing to support different large QC-MDPC codes defined from the designers of the cryptosystem, and scalable, targeting the whole Xilinx Artix-7 FPGA family. The hardware components implementing QC-MDPC bit-flipping decoding,

binary polynomial inversion, and binary polynomial multiplication, i.e., the three most complex operations employed within the BIKE cryptoscheme, are indeed specifically designed in a parametric way to exploit parallelism as desired according to the performance requirements and the area constraints given by the target platform.

Two separate modules target the cryptographic functionality of the client and server nodes of the quantum-resistant key exchange, respectively. This thesis delivers a preliminary definition of a methodology to identify the best parameterization of the configurable hardware components implemented within the proposed architecture's BIKE client and server cores. The methodology uses a complexity-based heuristic that leverages the knowledge of the time and space complexity of such parametric components to steer the design space exploration.

The proposed architecture's performance was evaluated against state-of-the-art software and hardware implementations. The proposed architecture's client- and server-side instances outperform the state-of-the-art reference software, exploiting the Intel AVX2 extension and running on a desktop-class CPU, by up to 1.91 and 1.83 times, respectively. Moreover, compared to the fastest reference state-of-the-art FPGA-based architecture, which targets the same Xilinx Artix-7 FPGA family, the architecture described in this thesis provides a performance speedup of up to six times. In particular, the proposed architecture executes the whole BIKE key encapsulation mechanism in a time ranging from 5.74ms to 0.61ms for AES-128-equivalent security and from 19.35ms to 1.77ms for AES-192-equivalent security, with the lowest performance obtained on the smallest FPGAs and the highest performance when targeting the largest Artix-7 200 chips.

Sommario

Si prevede che i computer quantistici romperanno le tradizionali soluzioni di crittografia a chiave pubblica nei prossimi decenni, rendendo fondamentale la progettazione di nuove soluzioni di sicurezza in grado di resistere anche agli attacchi effettuati dai computer quantistici. La crittografia post-quantistica ha l'obiettivo di progettare schemi crittografici che possano essere implementati sui computer tradizionali e che siano in grado di resistere sia agli attacchi tradizionali che a quelli quantistici. Le soluzioni di crittografia post-quantistica implementate dovranno soddisfare non solo i requisiti di sicurezza ma anche quelli prestazionali. Fornire un supporto hardware efficace per tali sistemi crittografici è infatti uno dei requisiti fissati dal NIST nell'ambito del processo di standardizzazione della crittografia post-quantistica attualmente in corso, ed è particolarmente cruciale per garantire un'ampia adozione di soluzioni di sicurezza post-quantistica su dispositivi embedded all'edge.

Questa tesi fornisce un'architettura hardware configurabile basata su FPGA per supportare BIKE, un sistema crittografico post-quantistico basato su codici QC-MDPC. In particolare, BIKE implementa un meccanismo di incapsulamento della chiave, ovvero uno schema crittografico che genera e condivide una chiave simmetrica tra due parti impiegando una coppia di chiavi pubblica e privata. L'architettura proposta punta a migliorare le prestazioni rispetto alle soluzioni software e hardware esistenti ed è configurabile attraverso una serie di parametri architetturali e del codice, che la rendono efficiente, fornendo buone prestazioni e allo stesso tempo utilizzando efficacemente le risorse disponibili sugli FPGA, flessibile, consentendo di

supportare diversi codici QC-MDPC di grandi dimensioni definiti dai progettisti del crittosistema, e scalabile, essendo rivolta all'intera famiglia di FPGA Xilinx Artix-7. I componenti hardware che implementano la decodifica bit-flipping di codici QC-MDPC, l'inversione di polinomi binari e la moltiplicazione di polinomi binari, ossia le tre operazioni più complesse impiegate all'interno dello schema crittografico BIKE, sono infatti appositamente progettati in modo parametrico per sfruttare il parallelismo desiderato in base ai requisiti prestazionali e i vincoli di area dati dalla piattaforma di destinazione.

Due componenti distinti realizzano rispettivamente la funzionalità crittografica dei nodi client e server dello scambio di chiave resistente ad attacchi quantistici. Questa tesi fornisce una definizione preliminare di una metodologia per identificare la migliore parametrizzazione dei componenti hardware configurabili implementati all'interno dei core client e server BIKE dell'architettura proposta. La metodologia utilizza un'euristica basata sulla complessità che sfrutta la conoscenza della complessità temporale e spaziale di tali componenti parametrici per guidare l'esplorazione dello spazio di progettazione.

Le prestazioni dell'architettura proposta sono state valutate rispetto alle implementazioni hardware e software dallo stato dell'arte. Le istanze lato client e lato server dell'architettura proposta superano le prestazioni del software di riferimento, eseguito su una CPU di classe desktop sfruttando l'estensione Intel AVX2, rispettivamente fino a 1.91 e 1.83 volte. Inoltre, rispetto all'architettura per FPGA di riferimento più performante, che ha come target la stessa famiglia di FPGA Xilinx Artix-7, l'architettura descritta in questa tesi fornisce un miglioramento delle prestazioni fino a sei volte. In particolare, l'architettura proposta esegue l'intero crittosistema BIKE in un tempo che va da 5.74ms a 0.61ms per una sicurezza equivalente ad AES-128 e da 19.35ms a 1.77ms per una sicurezza equivalente ad AES-192, con le prestazioni inferiori ottenute sugli FPGA con meno risorse e le prestazioni più elevate quando ha come target i chip di fascia più alta Artix-7 200.

Contents

CHAPTER 1

Introduction

The last few decades have seen significant technological improvements in the field of quantum computing. Since the first attempts in the early 1980s to devise abstract models of computers based on the principles of quantum physics [39], a vast amount of research has been carried out in the field of quantum computing to develop novel algorithms and design actual quantum computers.

The 1990s saw the introduction of the Shor's [94] and Grover's [47] quantum algorithms for integer factoring and database search algorithms, respectively, which highlighted the first notable examples of algorithms able to solve in polynomial time problems that would instead be hard, i.e., require exponential time for solving, on traditional computers. The first pioneering works also emerged in the academic environment for what concerns the realization of actual quantum computers, with the first 2-qubit quantum computer based on nuclear magnetic resonance (NMR) being presented in 1998 [28] and the number of qubits later increased to 7 and 12 in 2000 [65] and 2006 [79], respectively.

While NMR was discarded due to difficulty in scaling to a larger number of qubits, a variety of new solutions arose in the following decade from research on both the academic and industry sides. The rising commercial

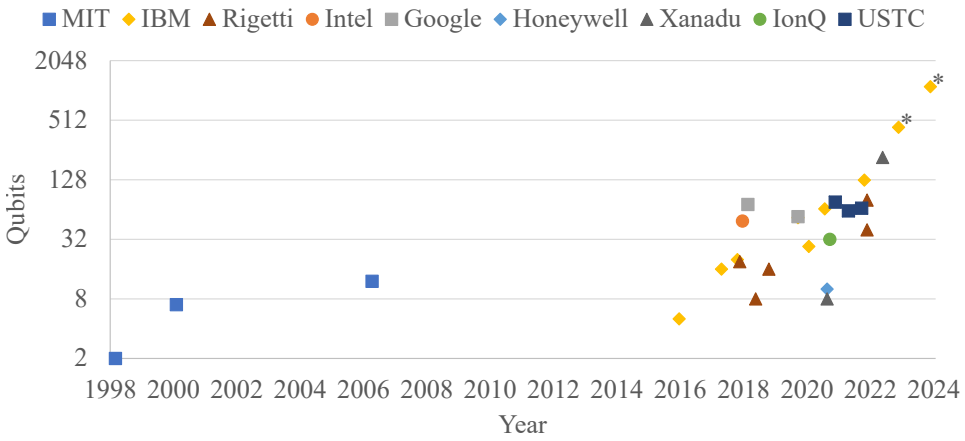


Figure 1.1: *Temporal evolution of the qubits per quantum computer by manufacturer. Data points marked with * are forecasts.*

interest for quantum computing saw indeed many of the largest tech companies heavily investing in research in such field. Most current solutions can be classified as either superconducting, photonics-based, or trapped ion quantum computers, depending on the underlying technology.

On the superconducting side, IBM presented in 2016 its IBM Quantum Experience, providing users from general public access to a five-qubit quantum processor through a graphical interface over the cloud [57]. Further research led IBM to introducing 27-, 65-, and 127-qubit superconducting quantum computers through 2020 and 2021 [57], with the expectation of presenting 433- and 1121-qubit models by the end of 2022 and 2023, respectively [58]. Other large companies involved in quantum computing research include Intel and Google, with the former introducing its Tangle Lake 49-qubit superconducting chip in 2018 [52] and the latter claiming for the first time the experimental demonstration of quantum supremacy in 2019 [8]. Photonics-based solutions were more recently introduced from Canada-based Xanadu, with its 8-qubit X8 [7] and 216-qubit Borealis [71] quantum processors presented in 2020 and 2022, respectively. On the academic side, Chinese USTC developed the Jiuzhang [108] and Zuchongzhi 2.1 [109] photonics-based quantum processors. Ion trapped quantum computing solutions include those by IonQ, which presented a 32-qubit model in 2020.

Although not yet widely available and powerful enough to be used in real applications, it is widely expected that quantum computers will achieve in the next decades a computing power that surpasses traditional ones in certain applications [82]. While actual applications will indeed require a

number of qubits in the order of millions and an error rate lower than 0.1%, Figure 1.1 highlights the exponential trend in the growth of the number of qubits per quantum computer.

The architecture of quantum computers, intrinsically different from the traditional one, allows them to efficiently solve operations research problems and drastically speed up the computations in applications ranging from machine learning to molecular docking. All these problems are instead computationally complex on traditional computers, thus quantum computing enables novel applications in a variety of fields, including finance, logistics, and chemistry. However, the use of quantum computers also opens up the implementation of mathematical algorithms that more efficiently solve the problems underlying traditional cryptography, decreasing its security.

Particularly critical is the impact of Shor's algorithm [94], which solves the factorization of an integer in polynomial rather than exponential time, and in a similar way can operate on the sum of two elliptic curves or compute a discrete logarithm. The availability of quantum computers would therefore make obsolete the currently used public key cryptosystems, whose security relies on the difficulty in solving such mathematical problems.

On the contrary, Grover's database search algorithm [47] highlights a security reduction by $2\times$ for AES and symmetric cryptoschemes and a $3\times$ security reduction for cryptographic hash functions such as SHA-2 and SHA-3. To obtain the same security as nowadays, where the only threat is given by traditional attacks, it will be sufficient to employ 2x larger symmetric keys and 3x larger hash digests, respectively.

Traditional public-key cryptosystems, including RSA [90], ECDSA [16], and Diffie-Hellman [31], underpin cryptographically secure key exchange mechanisms and digital signature schemes. The secure exchange of a shared secret is a fundamental component of secure communication protocols such as TLS [99] and SSH [105], which employ this secret to generate session keys, known to both parties of the communication, to be used within symmetric ciphers, which are more performing than public-key ones. The digital signature schemes also make it possible to guarantee the authenticity and integrity of a message, i.e., that the message was actually produced by its author and that it has not been modified by third parties.

Transport Layer Security (TLS) is a cryptographic protocol for secure communication between two nodes on a TCP/IP network that operates at the presentation layer. It is the basis of the HTTPS protocol, which uses TLS to obtain an encrypted connection using public key cryptography. TLS guarantees the confidentiality, authentication and integrity of the data transmitted between sender and recipient [87]. The TLS protocol involves

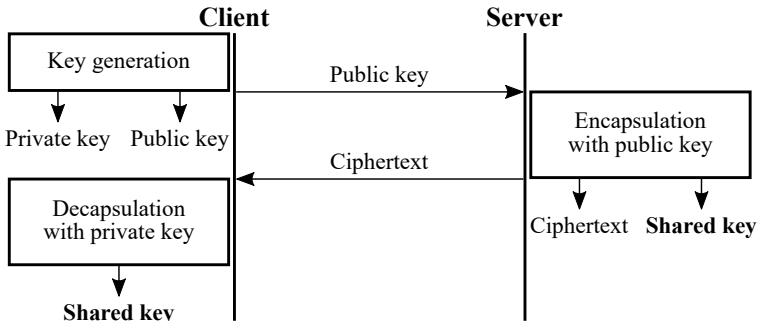


Figure 1.2: *Key exchange using a KEM.*

first the exchange of keys through a public key cryptosystem, such as RSA and Diffie-Hellman, and then a secure communication using symmetric encryption, such as AES, using the previously exchanged keys.

The threat posed by quantum computers to public key cryptosystems requires the definition and design of alternative cryptosystems that perform the same functions, maintaining security against traditional computer attacks but above all ensuring security against quantum computer attacks. Post-quantum cryptography (PQC) aims to develop new cryptosystems that are resistant to both traditional attacks and new quantum attack models, which can be implemented on traditional architecture computers and on existing devices, and that can be integrated into the networks and communication protocols currently in use [17]. Indeed, it is important to distinguish between post-quantum cryptography and quantum cryptography. The latter in fact specifically makes use of properties of quantum mechanics to ensure data security. For example, quantum key distribution (QKD) allows exchanging a secret key, preventing an attacker from intercepting it without this attack being detected by the two sides of the communication. To do this, QKD systems require a dedicated infrastructure for the transmission and detection of single photons. By making use of advanced, expensive, and not commonly available technologies on devices currently in use, such quantum cryptography techniques are therefore complex to deploy, unlike post-quantum cryptography.

The National Institute of Standards and Technology (NIST), a USA government agency that has among its aims the definition of cryptographic standards, is conducting a process for the standardization of PQC cryptosystems, in particular key encapsulation mechanisms (KEM) and digital signature schemes [76]. At the same time, the USA's National Security Agency (NSA) expects to complete the transition to quantum-resistant algo-

Table 1.1: *Status of the NIST PQC standardization process after the third round [77]. Legend: ^{L} lattice-based, ^{C} code-based, ^{I} isogeny-based cryptoscheme.*

Status	KEMs	Digital signatures
Selected for standardization	CRYSTALS-Kyber ^{L}	CRYSTALS-Dilithium ^{L} Falcon ^{L} SPHINCS^{+} ^{L}
Advancing to the fourth round	BIKE ^{C} Classic McEliece ^{C} HQC ^{C} SIKE ^{I}	

rithms for national security systems (NSS) to be complete by 2035. Such transition foresees the effective deprecation of the current public-key systems currently employed in NSS, i.e., RSA, DH, ECDH, and ECDSA, and their substitution with the post-quantum KEMs and digital signature schemes defined in the NIST PQC standardization process [78].

A KEM allows to securely transmit, through a public key algorithm, a shared secret, which can then be expanded to generate keys to be used in a symmetric cryptosystem, more efficient for the transmission of long messages than a public key cryptosystem [96]. After generating a random element of the finite group that underlies the implemented public key cryptosystem, such element is exchanged between the two parties of the communication, which can finally derive the shared secret by applying a hash function to the element of the finite group. A KEM consists of three main primitives, namely the generation of a pair of public and private keys, the encapsulation of the shared secret and its decapsulation. A key exchange performed through a KEM is depicted in Figure 1.2.

After the third round of the PQC standardization process, NIST selected the CRYSTALS-Kyber KEM for standardization, while appointing a fourth round of evaluation to further analyze BIKE, Classic McEliece, HQC, and SIKE, as shown in Table 1.1 [77]. Of particular importance for the evaluation of cryptosystems are not only the guaranteed security with respect to traditional and quantum attacks as well as the performance of software execution, but also the performance of the hardware implementations of such cryptosystems. In particular, NIST takes Intel Haswell processors and Xilinx Artix-7 FPGAs as references for software and hardware implementations, respectively. The latter FPGAs are specifically targeted by NIST for the hardware ones in order to provide a fair comparison environment for all proposals by avoiding differences due to the usage of different FPGA technologies or technology nodes of ASIC implementations.

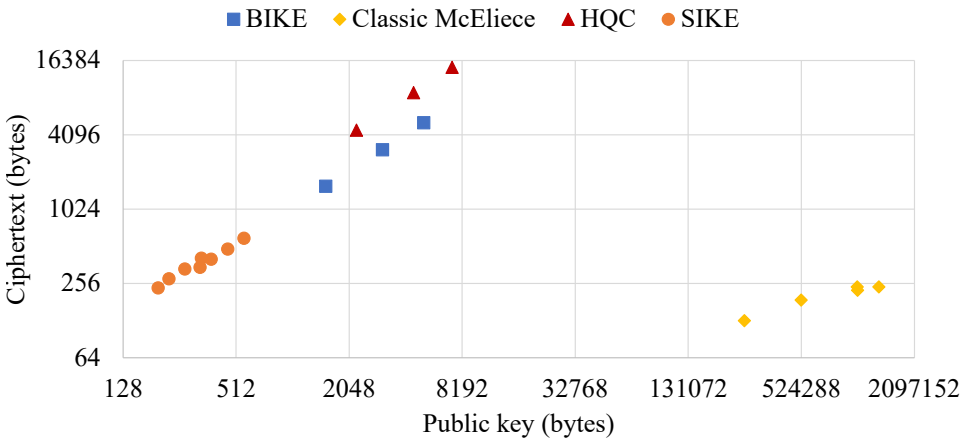


Figure 1.3: *Size in bytes of the public key and ciphertext of the KEMs advancing to the fourth round of the NIST PQC standardization process [75].*

Post-quantum KEMs can be mainly divided into lattice-based, code-based, and isogeny-based cryptography schemes.

The family of lattice-based cryptosystems [73, 80], that are characterized by a combination of good performance and small public key size, includes the upcoming NIST PQC standard CRYSTALS-Kyber [20]. From the mathematical point of view, a lattice L in is a discrete subgroup of R^n that generates the real vector space R^n. That is, each lattice L in R^n is generated from a base of R^n by means of linear combinations with integer coefficients. The main computational problem based on lattices is the shortest vector problem (SVP), which requires to search for the non-zero vector of a lattice having minimum norm. This problem is considered hard for both traditional and quantum computers [83].

Code-based cryptography dates back to the McEliece cryptosystem, introduced in 1978, which is based on the difficulty of decoding a generic linear code [72]. This problem is in fact recognized as NP-hard. The original scheme proposed by Robert McEliece makes use of binary Goppa codes, which allow a particularly efficient decoding [15]. More generally, the private key is an error correction code for which an efficient decoding algorithm is available and which is capable of correcting the desired number of errors. BIKE [5], Classic McEliece [2], and HQC [1] are post-quantum code-based cryptosystems in the fourth round of the NIST PQC standardization process.

Finally, isogeny-based cryptography builds its security on the problem of identifying an isogeny between two isogenous elliptic curves [38]. Isogeny-based cryptosystems are characterized by the smallest keys and cipher-

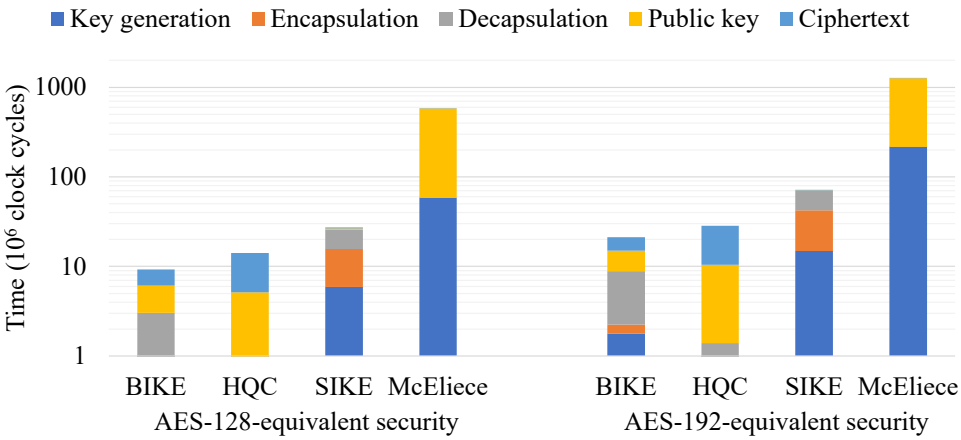


Figure 1.4: *Performance of NIST Round 4 KEMs on a x86-64 CPU, considering a 2000 cycles/byte transmission cost [77].*

texts among the PQC solutions, although the complexity of the underlying problem severely hinders their performance. SIKE is a post-quantum isogeny-based cryptosystems currently in the fourth round of the NIST PQC standardization process [61].

While NIST already selected a lattice-based post-quantum KEM, there is a crucial need to differentiate on multiple cryptoscheme families that are based on different computational problems, due to the uncertainty behind the actual security of these problems against quantum attacks. BIKE or HQC may represent the most attractive option for general-purpose applications among the fourth round candidates. SIKE might be a better solution for applications that need a very small ciphertext, but an attack was recently demonstrated to break it in one hour on a single-core CPU [24]. For what concerns Classic McEliece, it is not clear that there are any use cases that justify its usage of massively large public keys, with a length in the order of megabits [77].

In particular, BIKE is a KEM based on quasi-cyclic moderate-density parity-check (QC-MDPC) codes [5]. These codes are used in a scheme similar to that originally proposed by Niederreiter, which exploits the same problem as the McEliece cryptosystem [81]. Compared to Classic McEliece, that implements the traditional Niederreiter cryptosystem with binary Goppa codes, BIKE has a particularly small public key size, thanks to the quasi-cyclic nature and the reduced density of the QC-MDPC codes, which allow a more compact representation [10]. The performances are however markedly lower than those of Classic McEliece, which using binary Goppa codes has a much more efficient decoding. BIKE distinguishes itself for two aspects

related to ciphertext and key lengths and performance. On the one hand, compared to the other two code-based cryptosystems left in the NIST PQC standardization process, as shown in Figure 1.3, BIKE has both public key and ciphertext representations that are more compact than HQC, and a larger ciphertext but also a dramatically smaller public key than Classic McEliece. The key size of Classic McEliece is indeed a particularly critical aspect that hinders the applicability of the cryptosystem in real-world scenarios. On the other hand, BIKE has the most competitive performance among the non-lattice-based KEMs [77], as depicted in Figure 1.4, which compares BIKE to other candidate KEMs in NIST PQC Round 4, when considering the impact of the transmission cost deriving from sending the public key and ciphertext between the two nodes of the key exchange.

1.1 Contributions

Post-quantum security will be a crucial aspect across the whole continuum of computing, ranging from the Internet of things to high-performance computing, and the deployed PQC solutions will have to satisfy not only security requirements, but also performance ones. Providing an effective hardware support for post-quantum cryptosystems is indeed one of the requirements set by NIST within its standardization process, and it is particularly paramount to ensuring a wide adoption of post-quantum security solutions across embedded devices at the edge.

This thesis delivers a configurable FPGA-based hardware architecture that implements the BIKE post-quantum QC-MDPC code-based cryptosystem, with the aim of improving performance over the existing state-of-the-art software and hardware solutions. The proposed architecture consists of two client and server modules that support the BIKE KEM execution on the respective nodes of the quantum-resistant key exchange. This research provides three main contributions.

First, although there already exists a multitude of hardware solutions to support QC-MDPC codes, those solutions are often tailored to codes with dimensions in the order of hundreds of bits. Therefore, they cannot scale effectively to tens of thousands of bits as required by PQC applications, where there is a direct relationship between security of a cryptosystem and the size of the underlying code. This thesis provides the design of an architecture to effectively support QC-MDPC codes suitable to post-quantum cryptography applications.

Second, the proposed architecture is not a hard-coded one that is custom-tailored to a particular QC-MDPC code and to a specific FPGA target.

Instead, the result of this research is a parametric architecture that is *efficient*, providing good performance while using effectively the resources available on FPGAs, *flexible*, allowing to support different large QC-MDPC codes defined from the designers of the cryptosystem, and *scalable*, targeting the whole Xilinx Artix-7 FPGA family, i.e., the hardware platform identified by NIST for the proposals in its PQC standardization process.

Finally, this work provides a preliminary definition of a methodology to identify the best parameterization of the configurable hardware components. A complexity-based heuristic leverages the knowledge of the time and space complexity of such parametric components to steer the design space exploration and efficiently identify the combination of parameters that delivers the best hardware support.

The rest of the thesis is organized as follows. Chapter 2 discusses the theoretical background for QC-MDPC code-based post-quantum cryptography and BIKE. Chapter 3 overviews the state-of-the-art literature concerning implementations of QC-MDPC code-based cryptography and in particular of BIKE. Chapter 4 presents a top-down overview of the architecture that implements the BIKE cryptosystem, starting from the KEM primitives and then detailing the most complex operations, with a focus on the configurability of the various components by means of parameters that allow tuning their degree of parallelism. Chapter 5 discusses the experimental results of the main components and of the whole architecture, comparing them to reference state-of-the-art software and hardware implementations. Finally, Chapter 6 presents the conclusions and future works.

CHAPTER 2

Background

This chapter overviews the theoretical background that underlies the BIKE post-quantum QC-MDPC code-based key encapsulation mechanism. Understanding such theoretical aspects provides the reader with knowledge about how BIKE works, what makes it secure against both traditional and quantum model attacks, and how it can be implemented in software and hardware and optimized according to the desired figures of merit. The rest of this chapter details the structure of a generic key encapsulation mechanism, overviews the BIKE cryptoscheme and the three main primitives that compose it, and outlines the theory and algorithms underlying the principal operations associated with binary polynomials and QC-MDPC codes that are employed in BIKE.

Parts of this chapter are derived from previously published works co-authored by the author of this thesis. In particular, Section 2.1, Section 2.2, and Section 2.3 derive from [41], parts of Section 2.4 come from [111] and [43], and Section 2.5 originated from the work in [110]. More details about the referenced publications are provided in Appendix A.

2.1 BIKE key encapsulation mechanism

A key encapsulation mechanism (KEM) is a public-key cryptoscheme that performs the secure transmission between two communicating nodes of a shared secret, which can then be expanded to generate keys to be used in a symmetric cryptosystem, since the latter is in general more efficient for the transmission of long messages than a public key cryptosystem [96]. As previously shown in Figure 1.2, three main steps compose the key exchange between client and server nodes performed by means of a KEM. Such key exchange is the core task within the handshake phase between TLS clients and servers. First, the client performs the key generation primitive, producing a private-public key pair and sending the public key to the server. The server node then generates a shared secret and encrypts it with the public key of the client. Finally, the client retrieves the shared secret by decapsulating with its own private key the ciphertext received by the server node. As a result, the client and server endpoints obtained the same shared secret.

Secure communication protocols such as TLS 1.3 mandate the usage of ephemeral keys to enforce the perfect forward secrecy (PFS) property, thus the design of a computationally efficient key generation primitive is as important as for the encapsulation and decapsulation ones. It should be noted that the use of static keys would instead make it possible for a malicious attacker, once the private key has been compromised, to obtain all the session keys generated from it. The attacker would therefore be able to decrypt all the messages exchanged between the two secure communication nodes. Employing ephemeral keys avoids this vulnerability, since each session key is generated from a new public-private key pair.

2.2 BIKE

BIKE [6] is a QC-MDPC code-based KEM, based on the Niederreiter cryptosystem, that leverages quasi-cyclic matrices with coefficients over $\mathbb{Z}_2$. The employed quasi-cyclic (QC) matrices are composed of n_0 circulant blocks with size $p \times p$, that can be equivalently represented by n_0 binary polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$, with coefficients equal to the first row of the corresponding circulant blocks.

The arithmetic of $p \times p$ circulant matrices over $\mathbb{Z}_2$ is thus equivalent to the arithmetic of binary polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$. The addition of two binary polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$ corresponds to their bit-wise XOR, while their multiplication consists in their carry-less multiplication

Table 2.1: *Parameters of QC-MDPC codes for BIKE [6].*

Code	NIST security level	Equivalent security	p	t	v	$iter$
B1	Level 1	AES-128	12323	134	71	5
B3	Level 3	AES-192	24659	199	103	5
B5	Level 5	AES-256	40973	264	137	5

followed by a modular reduction with respect to the $x^p + 1$ irreducible polynomial. Moderate-density parity-check (MDPC) codes feature sparse parity-check H matrices, i.e., only a small percentage of values are set to 1, allowing for a sparse representation by enumerating the positions of bits set to 1. QC-MDPC codes possess both the quasi-cyclic and moderate-density properties.

We describe BIKE by using the following notation:

- $e = [e_0|e_1]$ is a random n-bit error vector with $t \approx \sqrt{n}$ bits set to 1, where $n = n_0 \cdot p = 2p$ and each e_i is a p-bit vector;
- $H = [h_0|h_1]$ is the private key, composed of $n_0 = 2$ circulant blocks h_i of size $p \times p$, with $v \approx \sqrt{n}$ bits set to 1 for each row of each block h_i;
- h is the public key, which is a circulant block of size $p \times p$;
- s is the p-bit syndrome;
- m, m', m'', and σ are 256-bit messages;
- c is the $(p + 256)$-bit ciphertext;
- K is the 256-bit shared secret.

BIKE provides implementations for NIST security levels 1, 3, and 5, each of them characterized by a different underlying QC-MDPC code. The security levels 1, 3, and 5 correspond to AES-128-, AES-192-, and AES-256-equivalent security, respectively. The employed QC-MDPC codes have a $2p$-bit code word length and a p-bit information word length. For simplicity, we denote each BIKE code as *Bj*, where *j* corresponds to the security level. Table 2.1 reports the QC-MDPC code parameters for each security level of BIKE.

2.3 BIKE primitives

The BIKE key encapsulation mechanism can be split into three primitives, which are the key generation, the encapsulation, and the decapsulation. Their corresponding algorithms are detailed in the rest of this section.

Algorithm 1 BIKE key generation primitive.

1: **function** $[H, \sigma, h]$ KEYGEN ( )
2: $seed =$ TRNG ();
3: $H =$ PRNG(SHAKE256($seed$));
4: $h_0^{-1} =$ INVERT(h_0);
5: $h = h_1 \odot h_0^{-1}$;
6: $\sigma =$ TRNG ();
7: **return** $\{H, \sigma, h\}$;
8: **end function**

Algorithm 2 BIKE encapsulation primitive.

1: **function** $[K, c]$ ENCAPS (h)
2: $m =$ TRNG ();
3: $e =$ PRNG(SHAKE256(m));
4: $s = e_0 \oplus (e_1 \odot h)$;
5: $m' = m \oplus$ TRUNC$_{256}$(SHA3-384(e));
6: $c = \{s, m'\}$;
7: $K =$ TRUNC$_{256}$(SHA3-384($\{m, c\}$));
8: **return** $\{K, c\}$;
9: **end function**

2.3.1 Key generation

First, a 32-bit random $seed$ must be produced by a true random number generator (TRNG) (line 2 of Algorithm 1). Such random seed is then employed within the pseudorandom number generation (PRNG) of the private key $H = [h_0|h_1]$ (line 3), which is composed of two polynomials each with Hamming weight equal to v, making use of the SHAKE256 extendable output function [35]. $seed$ is thus expanded into two uniform random sequences of v integer values comprised between 0 and $p - 1$, corresponding to the positions of bits set to 1 within the h_0 and h_1 binary polynomials. A binary polynomial inversion allows to compute h_0^{-1} starting from h_0 (line 4). The public key h is computed from the binary polynomial multiplication between h_1 and h_0^{-1}, where h_1 is sparse, with Hamming weight equal to v, while h_0^{-1} is dense, i.e., it has a Hamming weight $\approx p/2$ (line 5). In addition, a 256-bit message σ is also obtained by a TRNG (line 6). The key generation primitive outputs the private key H, the corresponding public key h, and the 256-bit message σ (line 7).

Algorithm 3 BIKE decapsulation primitive.

1: **function** $[K]$ DECAPS (H, σ, c)
2: $s' = h_0 \odot s$;
3: $e' = \text{DECODE}(s', H)$;
4: $m'' = m' \oplus \text{TRUNC}_{256}(\text{SHA3-384}(e'))$;
5: $a = (e' == \text{PRNG}(\text{SHAKE256}(m''))) \ ? \ m'' : \sigma$;
6: $K = \text{TRUNC}_{256}(\text{SHA3-384}(\{a, c\}))$;
7: **return** K;
8: **end function**

2.3.2 Encapsulation

Algorithm 2 details the encapsulation primitive, which takes as its only input the public key h received from the client node after the key generation. A 256-bit random message m is first obtained by a TRNG (line 2 of Algorithm 2). Such random seed is then expanded through the SHAKE256-based PRNG into the n-bit error vector $e = [e_0|e_1]$, which has Hamming weight equal to t (line 3). The syndrome s is subsequently obtained by XORing e_0 and the product of the binary polynomial multiplication between e_1 and h, where e_1 is a sparse polynomial with Hamming weight up to t (line 4). m' is the result of the XOR between m and the 384-bit SHA-3 [35] hash digest of the error vector e', after truncating the digest to 256 bits (line 5). The concatenation of the syndrome s and the message m' corresponds to the ciphertext c (line 6). The shared secret K is instead obtained by truncating to 256 bits the SHA-3 digest of the concatenation of m and c (line 7). The encapsulation primitive outputs the shared secret K and the ciphertext c (line 8).

2.3.3 Decapsulation

Algorithm 3 details the decapsulation KEM primitive, which takes as its inputs the public key H and the message σ stored on the client side as well as the ciphertext $c = \{s, m'\}$ received from the server node after the latter executed the encapsulation. A binary polynomial multiplication between h_0 and s, where the h_0 operand is sparse with Hamming weight equal to v, first produces s' (line 2 of Algorithm 3), QC-MDPC bit-flipping decoding allows retrieving the error vector e', starting from the syndrome s' and the private key H (line 3). In particular, BIKE makes us of the Black-Grey-Flip (BGF) variant of QC-MDPC bit-flipping decoding [34]. m' is then XORed with the 384-bit SHA-3 hash digest of the retrieved n-bit error vector e', after truncating the digest to 256 bits (line 4). Subsequently, e' is compared to the

output of seeding the SHAKE384-based PRNG with m''. If the two n-bit vectors coincide, then a is assigned m'', otherwise it is assigned σ (line 5). Finally, the shared secret K is computed as the SHA3-384 hash digest of the concatenation of a and c, that is then truncated to 256 bits (line 6). Such 256-bit shared secret K is the output of the decapsulation primitive (line 7).

2.4 Binary polynomial arithmetic

A finite field, also called Galois field, is a set that contains a finite number of elements on which the addition, subtraction, multiplication and division operations are defined. $\mathbb{Z}_2$, or $GF(2)$, is the Galois field of two elements, i.e., the smallest Galois field. The two elements of $\mathbb{Z}_2$ are usually referred to as 0 and 1, and they are respectively the additive and the multiplicative identities. The field's addition operation corresponds to the logical XOR operation, while the multiplication operation corresponds to the logical AND operation.

Polynomials with coefficients in $\mathbb{Z}_2$, i.e., 0 and 1, form a Galois field, which is commonly referred to as $\mathbb{Z}_2[x]$ or $GF(2)[x]$. The Galois field of binary polynomials with degree $m-1$ also referred to as $GF(2^m)$. The addition of two elements of such field corresponds to a bitwise XOR. The multiplication, instead, consists in the multiplication of the two binary polynomials, followed by a reduction with respect to an irreducible polynomial, which is taken from the construction of the field. For example, $\mathbb{Z}_2[x]/(x^p+1)$ is the Galois field of polynomials with coefficients in $\mathbb{Z}_2$ for which the irreducible polynomial is x^p+1, thus polynomials which belong to such field have degree at most equal to $p-1$.

2.4.1 Binary polynomial inversion

In $\mathbb{Z}_2[x]/(x^p+1)$, a multiplicative inverse for a polynomial $a(x)$, denoted by $a(x)^{-1}$, is a polynomial that when multiplied by $a(x)$ yields the multiplicative identity 1, i.e., $a(x)\cdot a(x)^{-1}=1$.

Inversion algorithms can be split in two families, deriving from Euclid's algorithm and from Fermat's little theorem, respectively. Euclid's algorithm allows to compute the greatest common divisor between two polynomials, and polynomial-time algorithms based on it are proposed by [18, 23, 66]. Algorithms based on Fermat's little theorem date back to the Itoh-Tsujii algorithm (ITA) introduced by [60] and are employed in the software implementations of BIKE [33] and LEDAcrypt [14] and in the hardware implementation of BIKE [89].

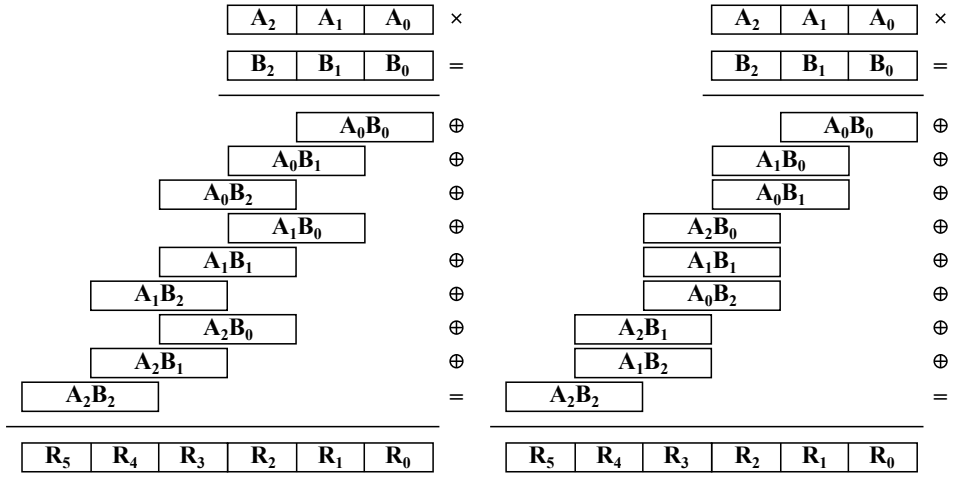
Algorithm 4 Inversion procedure from [14]. $a(x)$ is a binary polynomial in $\mathbb{Z}_2[x]/(x^p+1)$ with a multiplicative inverse, where p is a prime such that $ord_2(p) = p-1$. $d(x)$ is the multiplicative inverse of $a(x)$, i.e., $d(x) = a(x)^{-1}$.

1: **function** $[d(x)]$ INVERSION($a(x)$)
2: $b(x) = a(x)$;
3: $c(x) = a(x)$;
4: **for** $i \in 1 : (\lceil \log_2(p-2) \rceil - 1)$ **do**
5: $d(x) = c(x)^{2^{2^{i-1}}}$;
6: $c(x) = d(x) \cdot c(x)$;
7: **if** $(p-2)_2[i] == 1_2$ **then**
8: $d(x) = b(x)^{2^{2^i}}$;
9: $b(x) = d(x) \cdot c(x)$;
10: **end if**
11: **end for**
12: $d(x) = b(x)^2$;
13: **return** $d(x)$;
14: **end function**

The inversion algorithm employed by the software implementation of the LEDAcrypt-KEM-CPA cryptoscheme [14] is detailed in Algorithm 4. It takes a $\mathbb{Z}_2[x]/(x^p+1)$ binary polynomial $a(x)$ as input and executes a fixed number of iterations to output its multiplicative inverse $d(x) = a(x)^{-1}$. Each iteration (lines 4-11 in Algorithm 4) consists of two exponentiations (lines 5 and 8) and two multiplications (lines 6 and 9). However, lines 8 and 9 of iteration i are executed only when the condition at line 7 is verified, i.e., if the i-th bit of $p-2$ is equal to 1. Finally, a squaring operation produces the inverse polynomial (line 12). Algorithm 4 requires $(\log_2(p-2) + hw(p-2) - 1)$ multiplications and $(\log_2(p-2) + hw(p-2))$ exponentiations, where $hw(y)$ represents the Hamming weight, i.e., the number of bits set to 1, of y. The amount of required operations depends thus not on the input $a(x)$, but exclusively on the polynomial length p, that is a fixed parameter of the QC-MDPC code.

2.4.2 Binary polynomial multiplication

Multiplication in $\mathbb{Z}_2[x]$ conceptually works like long multiplication between integer numbers, except for the fact that the carry is always discarded instead of added to the more significant position. This property derives from the fact that the addition in $\mathbb{Z}_2$ corresponds to the logical XOR. For this reason, the multiplication operation in $\mathbb{Z}_2[x]$ is also commonly referred to as carry-less multiplication.



(a) *3-digit schoolbook* **(b)** *3-digit Comba*

Figure 2.1: *Two multiplication methods implementing the long multiplication algorithm on digital systems. The number of partial products and additions grows up quadratically for both schoolbook and Comba algorithms. Comba offers a more efficient scheduling of partial products, thus optimizing the memory write access pattern.*

Considering the quasi-cyclic codes employed in many proposals for post-quantum code-based cryptosystems, the arithmetic of $p \times p$ circulant matrices over $\mathbb{Z}_2$ can be substituted with the arithmetic of polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$. In code-based cryptosystems, matrix multiplication is the most computationally intensive operation of the encryption primitives. Since matrix multiplication corresponds to polynomial multiplication when considering quasi-cyclic codes, it is crucial for the performance of these post-quantum cryptosystems to implement the latter operation in an effective way.

The rest of this section overviews a few state-of-the-art algorithms to perform polynomial multiplication, i.e., the ones used in the proposed implementation. It is important to note that the multiplication algorithms have been selected to provide top-notch performance at reasonable complexity cost, according to the range of sizes employed in quantum-resistant code-based cryptography. The use of more complex algorithms provides no extra performance but a non-negligible resource overhead, since they are expected to perform better when the operand sizes are orders of magnitude higher than what is needed to support code-based cryptography.

Schoolbook multiplication

The schoolbook multiplication method implements the long multiplication for the execution on a digital system. Starting from the binary representation of the $\mathbb{Z}_2[x]$ polynomials, each factor is split into digits according to the actual operand size of the digital system, e.g., 32 or 64 bits on current general-purpose computers. The long multiplication algorithm is then implemented considering the digits as elementary units in the multiplication algorithm.

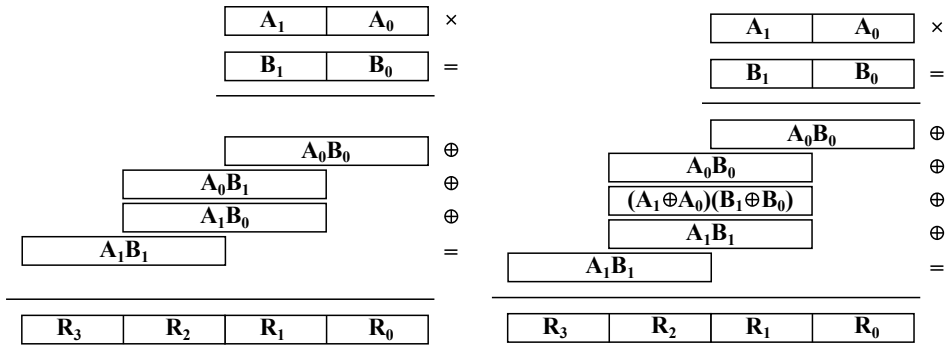
Figure 2.1a depicts the schoolbook multiplication between two polynomials A and B. Each polynomial has been split in three digits A_i and B_i, where their size is not explicitly reported since the method works for any possible digit size and consequently number of digits. Remarkably, a larger digit size corresponds to a smaller number of digits for each polynomial and therefore a smaller number of corresponding partial products to be computed, thus resulting in a faster computation. In particular, the number of A_iB_j partial products and the number of additions increase quadratically with the number of digits.

Comba multiplication

The Comba multiplication method [30] minimizes the number of memory writes by optimizing the order of computation of the partial products (see Figure 2.1b). In particular, the Comba algorithm requires exactly the same number of partial products and corresponding additions as the schoolbook approach, but it minimizes the number of bits required to store in memory the sum of the partial products.

For example, the final value for the R_0 digit is written in memory when the A_0B_0 partial product is ready. Thereafter, the Comba method operates a right shift of digit size to trash the lower part of the A_0B_0 partial product, since it is not necessary anymore. In a similar manner, the final value for the R_1 digit is written in memory when the subsequent A_0B_1 and A_1B_0 partial products have been computed and added to the upper part of A_0B_0. Once again, the lower part of the intermediate result is trashed out since it is no longer useful.

To this extent, the Comba method ensures a maximum of $2 \times size(digit)$ bits for the intermediate result, while the schoolbook algorithm requires at least $N \times size(digit)$ bits, where N is the number of digits of each operand. In general, even though the number of required multiplications and additions remains the same, the optimized memory access pattern of the Comba solution provides better performance than the schoolbook approach.



(a) *2-digit schoolbook* **(b)** *2-digit Karatsuba*

Figure 2.2: *Multiplication of two-digit polynomials considering schoolbook and Karatsuba algorithms. Karatsuba optimizes the computation by leveraging the intuition for which multiplications are more computationally expensive than additions in $\mathbb{Z}_2[x]$. In particular, schoolbook requires four multiplications and three additions, while Karatsuba performs the same computation using three multiplications and six additions.*

Karatsuba multiplication

The Karatsuba algorithm [64] optimizes the performance of the polynomial multiplication by reducing the number of partial products to be computes. This method leverages the intuition for which the multiplication is far more computationally expensive than the addition in $\mathbb{Z}_2[x]$. Figure 2.2 depicts the multiplication of two operands, each of them split in two digits, using either the schoolbook (see Figure 2.2a) or the Karatsuba (see Figure 2.2b) approaches. The schoolbook solution requires four multiplications and three additions to perform the polynomial multiplication. In contrast, the Karatsuba approach requires three multiplications and six additions.

The recursive application of the Karatsuba multiplication formula, i.e., computing the partial products through Karatsuba multiplications with smaller operands, allows further reducing the time complexity compared to schoolbook and Comba algorithms.

2.4.3 Binary polynomial exponentiation

Exponentiation in $\mathbb{Z}_2[x]/(x^p + 1)$ is the operation that computes $g(x) = f(x)^k$, where the base $f(x)$ and the result $g(x)$ are polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$ while the exponent k is a number. If k is a positive integer, then the exponentiation corresponds to iterating k times the multiplication of the base $f(x)$. Squaring, i.e., $g(x) = f(x)^2$, is a basic case of exponentiation, where k is equal to 2. Notably, in $\mathbb{Z}_2[x]/(x^p + 1)$ it is computed by interleaving the

Algorithm 5 Exponentiation procedure. $f(x)$ is a binary polynomial in $\mathbb{Z}_2[x]/(x^p+1)$, where p is a prime such that $ord_2(p) = p-1$. k is a non-zero positive integer, i.e., $k > 0$. $g(x) = f(x)^k$.

1: **function** $[g(x)]$ EXPONENTIATION($f(x), k$)
2: $g(x) = 0$;
3: **for** $i \in 0 : (p-1)$ **do**
4: $g(x)[(i \cdot k) \bmod p] = f(x)[i] \oplus g(x)[(i \cdot k) \bmod p]$;
5: **end for**
6: **return** $g(x)$;
7: **end function**

bits of the input polynomial $f(x)$ with bits set to 0. Extending the squaring case to the more general exponentiation in $\mathbb{Z}_2[x]/(x^p+1)$, the computation of $g(x) = f(x)^k$ revolves around the idea that two consecutive elements in $f(x)$ will be separated by $k-1$ other elements in $g(x)$. Coefficients of $g(x)$ are initialized to 0. Bits in the $f(x)$ polynomial are read one by one starting from bit 0 up to bit $(p-1)$, and they are added (XORed, since coefficients are in $\mathbb{Z}_2$) to the $g(x)$ polynomial starting from bit 0 and incrementing each time by k, with the increment operation being performed modulo p, until all the p bits have been written.

For generic m and k, where m is the binary polynomial length, there may be cancellations, i.e., there could be two 1s of the input polynomial that contribute to the same bit of the result polynomial, thus canceling each other because of the XOR operation. However, it is guaranteed that there are no cancellations if m and k are coprime, i.e., their greatest common divisor (GCD) is 1. In the considered application of QC-MDPC codes to cryptography, i.e., the BIKE and LEDAcrypt cryptosystems, m values are always prime numbers p, while k values are powers of 2, therefore the coprimality condition is always verified and it is guaranteed that there are no cancellations. The exponentiations can therefore be considered equivalent to permutations. The $g(x) = f(x)^{2^s}$ exponentiation, where $f(x)$ and $g(x)$ are polynomials in $\mathbb{Z}_2[x]/(x^p+1)$, p is a prime number, and s holds a positive integer value, can be equivalently expressed as the $P_i \rightarrow P_j$, $j = (i \cdot ((2^s) \bmod p)) \bmod p$ permutation, where P_i and P_j indicate the positions of coefficients in $f(x)$ and $g(x)$, respectively.

Restricting the exponentiations to the ones with power k equal to 2^s, with s holding a positive integer value, the computation can also be performed by iterating s times the squaring of the $f(x)$ polynomial.

Algorithm 5 details the procedure to compute the exponentiation of a binary polynomial in $\mathbb{Z}_2[x]/(x^p+1)$. It takes as inputs the $f(x)$ polynomial

$$f(x) = x^{10} + x^9 + x^3 + x^1 + x^0 = 11000001011_2$$
$$g(x) = f(x)^k = f(x)^4 = \mathbf{00010011011}$$

Time	$f(x)$	$g(x)$
0	11000001011	00000000000
1	1100000101**1**	0000000000**1**
2	1100000**1**11	000000**1**0001
3	1100000**1**011	00**0**00010001
4	1100000**1**011	00000010**1**11
5	1100000**1**011	000000**1**0011
6	1100**0**001011	0**0**000010011
7	1100**0**001011	0000001**0**011
8	110**0**0001011	00000**0**10011
9	11**0**00001011	**0**0000010011
10	11000001011	000000**1**1011
11	**1**1000001011	**0001**0011011

Figure 2.3: *Example of exponentiation.*

and the k non-zero positive integer value, which constitute the base and the exponent, respectively, and it produces the corresponding $g(x)$ polynomial, where $g(x) = f(x)^k$. The exponentiation procedure starts by setting the $g(x)$ polynomial to 0, i.e., its corresponding binary representation is initially constituted by all p bits set to 0 (see line 2 in Algorithm 5). Then, for each i ranging from 0 to $(p-1)$, the algorithm computes the value of the bit in position $i \cdot k \bmod p$ of the $g(x)$ polynomial, i.e., $g(x)[i \cdot k \bmod p]$, as the bit-wise exclusive OR between the values of $g(x)[i \cdot k \bmod p]$ and the i-th bit of the $f(x)$ polynomial, i.e., $f(x)[i]$ (see lines 3-5 in Algorithm 5). Notably, if k and p are coprime, each bit of $g(x)$ is assigned exactly once inside the *for* loop in Algorithm 5, hence each bit of $g(x)$ can be computed independently from the other bits of the same polynomial. Line 2 of Algorithm 5 thus becomes $g(x)[(i \cdot k) \bmod p] = f(x)[i]$. The coprimality condition is verified in the considered application of QC-MDPC codes to cryptography, i.e., the BIKE and LEDAcrypt cryptosystems.

Figure 2.3 shows an example of the iterative exponentiation procedure in Algorithm 5 to compute $g(x)$ as the 4-th power of $f(x)$. $f(x)$ and $g(x)$ are polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$ represented as p-bit binary values, where k is equal to 4 and p is equal to 11. The procedure takes 12 timesteps. At timestep 0, all bits in $g(x)$ are cleared, i.e., set to 0. One bit of the $f(x)$ polynomial is then processed at each of the subsequent 11 timesteps, with the i-th bit in the $f(x)$ polynomial contributing to generate the bit in position $i \cdot k \bmod p$ in the $g(x)$ polynomial (see line 4 in Algorithm 5), where i

ranges from 0 to 10. For each timestep, the processed and generated bits in $f(x)$ and $g(x)$ polynomials are highlighted in red. At the final timestep, the value of $g(x)$ is the result of the exponentiation, i.e., $g(x) = f(x)^4$.

2.5 Quasi-cyclic moderate-density parity check codes

Quasi-cyclic low-density parity check (QC-LDPC) and quasi-cyclic moderate-density parity check (QC-MDPC) codes emerged as viable solutions to design post quantum cryptosystems due to two main advantages enabled by their internal structure, i.e., their quasi-cyclicness and their sparse nature. On the one hand, such structure allows a significant decrease of the computational complexity of both the software and hardware implementations of KEMs that make use of such codes. On the other hand, the size of the private-public keypair and the ciphertext is also significantly reduced, resulting in a tighter memory footprint.

The rest of this section provides an overview of the basics of QC-LDPC and QC-MDPC codes with a focus on their decoding algorithms.

2.5.1 Moderate-density parity-check codes

Low-density parity-check (LDPC) codes are linear error correction codes that were first introduced by Gallager [44] and that allow transmitting messages over noisy channels. Their parity-check matrices are characterized by a low Hamming weight, i.e., a low number of bits set to 1, which enables a sparse representation. Moderate-density parity-check (MDPC) codes are defined by parity-check matrices with a higher density than LDPC ones, but they can still be represented effectively in a sparse way.

Without loss of generality, we are focusing on binary MDPC codes since they were the most widely adopted in code-based post-quantum cryptography. Starting from the definition of the Galois field of order 2, i.e., GF_2, we denote as GF_2^k the k-dimensional vector space defined over GF_2. To this end, a binary linear code denoted as $\mathbf{C}(n, k)$ is defined as a mapping which univocally associates each binary k-tuple, i.e., the information vector, to a binary n-tuple, i.e., the codeword (see Equation (2.1)).

$$\mathbf{C} : GF_2^k \rightarrow GF_2^n \tag{2.1}$$

In general, an MDPC code $\mathbf{C}(n, k)$ is defined by its parity-check matrix H that has r rows and n columns, where $r = n - k$ [10]. Such matrix can be graphically represented by the associated Tanner graph, that is a bipartite graph made of n variable nodes and r check nodes. A codeword bit

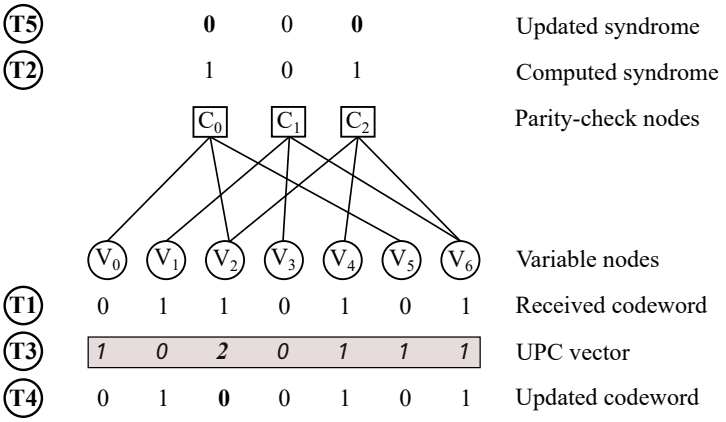


Figure 2.4: *Tanner graph of an MDPC code with $n = 7$ and $r = 3$. The steps of the bit-flipping algorithm used to correct the bit of the codeword associated to the V_6 variable node are marked from T_0 to T_4.*

is associated to each variable node, while each parity-check bit is associated to a check node. In particular, the set of all the parity-check bits defines the so-called syndrome vector s. Each $h_{i,j}$ element of the H matrix set to 1 indicates that the j-th bit in the codeword participates in the i-th parity check equation. The i-th syndrome bit is therefore computed as the bitwise XOR of all the codeword bits involved in the i-th parity-check equation. For example, the Tanner graph of a binary MDPC code with $n = 7$ and $r = 3$ is depicted in Figure 2.4, while Equation (2.2) defines the corresponding H matrix. For each parity-check node, the number of incoming edges is equal to the number of ones in the corresponding row of the H matrix, while the number of incoming edges to each variable node is equal to the ones in the corresponding column of the H matrix.

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Once a codeword is received, the decoding procedure analyzes the parity-check equations by generating the syndrome s of the codeword c through H, according to Equation (2.3).

$$s = c \cdot H^T \tag{2.3}$$

The received codeword is considered to be error-free when the syndrome is a null vector. In case the received codeword contains errors, the error correction algorithm iteratively recovers such errors in the codeword until either all

the parity check equations are satisfied, i.e., the syndrome is the null vector, or the codeword is declared unrecoverable and, thus, it has to be retransmitted by the sender. We note that, regardless of the use of soft-decision, e.g., Logarithmic-Likelihood-Ratio Sum-Product Algorithm (LLR-SPA) [10], or hard-decision, e.g., bit-flipping (BF), error correction algorithms, all the available codeword decoding algorithms implement an iterative procedure. Soft-decision decoders represent the most employed decoding solutions in telecommunication applications due to their superior performance coming from the exploitation of the available channel information [10]. In contrast, the bit-flipping algorithm represents the most employed decoding solution when no medium information is available, the floating point support is not available, and an efficient decoder design is required [10]. Considering its vast applicability and the possibility of delivering efficient decoders, the bit-flipping decoding algorithm is the sole solution adopted by the code-based cryptosystems participating to the NIST post-quantum competition.

Figure 2.4 depicts an example of the iterative bit-flipping decoding procedure, made of one iteration and five time-steps, i.e., T_1-T_5, to correct a received codeword by means of the bit-flipping algorithm. At time T_0 the sender transmits the codeword $c = 0100101$, that is received with an error by the receiver at time T_1 as $c = 0110101$. In this example, the received codeword contains an error in the bit associated to V_2, i.e., its value is 1 instead of 0. The bit-flipping decoding algorithm associates each bit of the received codeword to the corresponding variable node, and the syndrome is computed at time T_2 according to Equation (2.3). We note that the syndrome is made of three bits, i.e., one bit for each parity-check node. In particular, the parity-check equations corresponding to the parity-check nodes C_0 and C_2 are not satisfied and, thus, the error-recovery strategy of the bit-flipping algorithm takes place. For each iteration, the bit-flipping algorithm can flip one or more bits in the received codeword according to the information contained in the unsatisfied parity-checks (UPC) vector. For each variable node, the corresponding UPC value corresponds to the number of failed parity-check equations, i.e, the number of connected parity-check nodes whose associated syndrome bit has a value equal to 1. The UPC vector is defined by Equation (2.4) and it is computed at time T_3 (see Figure 2.4).

$$UPC = s \cdot H \tag{2.4}$$

Starting from the UPC vector, the bit-flipping algorithm flips each bit in the codeword for which the corresponding UPC value is above a certain threshold. We note that the threshold selection is a parameter of the bit-flipping algorithm and it strongly depends on the specific MDPC code.

The threshold is selected to minimize the trade-off between the decoding failure rate (DFR), i.e., the number of times the algorithm fails decoding the received codeword, and the number of decoding iterations. At time T_4, the codeword is updated by flipping the bits corresponding to UPC values greater or equal to the threshold (which, as an example, can be set to the maximum of the values assumed by the UPC vector). In our case, the codeword bit corresponding to the variable node V_2 is flipped from 1 to 0, since its UPC value is equal to 2 (the maximum value assumed by the UPC vector). Finally, at time $T5$, the syndrome bits associated to the flipped codeword bits are also flipped, which is a faster way to update the syndrome vector than recomputing the vector-matrix multiplication in Equation (2.3). In the example in Figure 2.4, the syndrome bits corresponding to parity-check nodes C_0 and C_2 are both flipped from 1 to 0. Being the syndrome after T_5 equal to the null vector, the decoding procedure can be interrupted since the codeword has been certainly recovered correctly, i.e., all the transmission errors have been corrected. Otherwise, if the syndrome vector were not a null vector, the iterative procedure would have been continued by repeating the steps executed at the time-steps from T_3 to T_5.

2.5.2 Circulant matrices

A circulant matrix is defined as a square matrix where each row is obtained by shift-rotating the preceding row to the right by one position. By construction, a circulant matrix is therefore regular, i.e., both columns and rows have constant weight. A $p \times p$ circulant matrix A, where each element is denoted as a_i with $i \in [0, ..., p-1]$, is shown in Equation (2.5).

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & ... & a_{p-1} \\ a_{p-1} & a_0 & a_1 & ... & a_{p-2} \\ a_{p-2} & a_{p-1} & a_0 & ... & a_{p-3} \\ ... & ... & ... & ... & ... \\ a_1 & a_2 & a_3 & ... & a_0 \end{bmatrix} \tag{2.5}$$

We note that the arithmetic of circulant matrices of size p is isomorphic to the arithmetic of the polynomials modulo $x^p - 1$ over the same field as the coefficients of the circulant matrices. The circulant matrix A is therefore isomorphic to a polynomial $a(x)$ with coefficients given by the elements of the first row of the matrix, as shown in Equation (2.6).

$$a(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_{p-1} \cdot x^{p-1} \tag{2.6}$$

Considering the case of binary linear block codes, the arithmetic of $p \times p$ circulant matrices over $\mathbb{Z}_2$ can be substituted by the arithmetic of polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$, which provides a reduction in the storage requirements and a faster execution of the arithmetic operations.

2.5.3 QC-MDPC codes

Quasi-cyclic (QC) codes are linear block codes $\mathbf{C}(n, k)$ whose parity-check matrices H are composed of $r_0 \times n_0$ circulant blocks, each of size $p \times p$, where $n = n_0 \cdot p$, $k = k_0 \cdot p$ and $r_0 = n_0 - k_0$. Considering post-quantum code-based cryptosystems, we focus on the $r_0 = 1$ case, for which the corresponding family of QC codes has a rate of $(n_0 - 1)/n_0$. In this case, the parity check matrix is defined by Equation (2.7), where each block H_i with $i \in [0, \ldots, n_0 - 1]$ is a circulant matrix of size $p \times p$.

$$H = \begin{bmatrix} H_0 & H_1 & \ldots & H_{n_0-1} \end{bmatrix} \tag{2.7}$$

The structure of quasi-cyclic codes enables efficient encoding implementations by means of fast binary polynomial multipliers. However, the lack of an efficient decoding support, due to the inherent structure of the H matrix, prevented their widespread use for a long time. QC-LDPC and QC-MDPC codes have been explored as particular classes of quasi-cyclic codes that are characterized by parity-check matrices which are well-suited for LDPC and MDPC decoding algorithms, where the matrix is sparse and it avoids the presence of short length cycles in the associated Tanner graph [10]. In particular, QC-LDPC and QC-MDPC codes combine the efficient decoding and the low decoding failure rate (DFR) of LDPC and MDPC codes with the efficient encoding and the small memory footprint of QC codes.

2.5.4 QC-MDPC bit-flipping decoding

This part overviews the baseline bit-flipping decoding algorithm for QC-MDPC codes and its Black-Gray-Flip (BGF) variant, which is employed within the BIKE cryptoscheme, highlighting their general structure as well as the optimizations introduced by the BGF algorithm.

Baseline bit-flipping decoding

Algorithm 6 describes the baseline QC-MDPC bit-flipping decoding procedure. The main bit-flipping decoder function, i.e., `BFDecoding`, executes a predefined number of iterations (see lines 3-10 in Algorithm 6) to produce the error vector (`e`) and the decoding failure boolean flag (`fail`) as outputs.

Algorithm 6 Bit-flipping decoding procedure for MDPC codes [12]. H is a $p \times n$ parity check matrix, where $n = p \cdot n_0$. s is a p-bit syndrome vector. e is an n-bit error vector. $fail$ is a 1-bit flag that is set in case of decoding failure.

1: **function** $[e, fail]$ BFDECODING(H, s)
2: $e = 0; fail = 0;$
3: **for** $i \in 1 : iter_{max}$ **do**
4: $thr =$ THRESHOLD(s);
5: $upc = s \cdot H$;
6: $e_{bf} = (upc \geq thr)\ ?\ 1 : 0$;
7: $e = e \oplus e_{bf}$;
8: $s_{bf} = e_{bf} \odot H^T$;
9: $s = s \oplus s_{bf}$;
10: **end for**
11: $fail = (s == 0)\ ?\ 0 : 1$;
12: **return** $e, fail$;
13: **end function**

Each decoding iteration consists of a sequence of six operations. First, a threshold is computed as a function of the syndrome vector (line 4). Then, the vector of unsatisfied parity checks (UPC) is computed as the product of the syndrome vector s and the parity-check matrix H, treating both s and H as integer numbers rather than values in $\mathbb{Z}_2$ (line 5). The obtained integer UPC values are compared to the threshold value, producing an error bit-flips vector (e_{bf}) with bits set to 1 in positions corresponding to UPC values greater than or equal to the threshold and set to 0 otherwise (line 6). The error vector is then updated by XORing the error bit-flips vector (line 7), and the syndrome bit-flips vector s_{bf} is computed through the carry-less multiplication between e_{bf} and H (line 8). Finally, the syndrome vector is updated by XORing s_{bf} (line 9). Once all the decoding iterations have been executed, the $fail$ flag is set to 1 to signal a decoding failure if the syndrome has Hamming weight not equal to 0, while it is set to 0 otherwise.

From the computational complexity viewpoint, the UPC computation (see line 5 in Algorithm 6) and the syndrome bit-flips computation (see line 8 in Algorithm 6) represent the two most critical operations. In fact, both the UPC and the syndrome bit-flips computations impose a vector-matrix multiplication, i.e., the former between the syndrome and the H matrix and the latter between the error bit-flips vector and H. In particular, the UPC computation is performed in the integer domain, while the syndrome bit-flips computation is instead performed in the binary domain. Concerning the remaining operations, the threshold computation procedure is usually customized to minimize both the decoding failure rate (DFR) of the under-

Algorithm 7 BGF decoding procedure [34, 89]. H is a $p \times n$ parity check matrix, where $n = 2p$. s is a p-bit syndrome vector. e is an n-bit error vector. $black$ and $gray$ are n-bit vectors. $fail$ is a 1-bit flag that is set in case of decoding failure.

function $[e, fail]$ BGFDECODING(H, s)
  $e = 0; fail = 0;$
  for $i \in 1 : iter_{max}$ **do**
    $thr =$ THRESHOLD$(s + e \cdot H^T);$
    $e, black, gray =$ BITFLIPITER$(s + e \cdot H^T, e, T, H);$
    if $i = 1$ **then**
      $e =$ BITFLIPMASKEDITER$(s + e \cdot H^T, e, black, H);$
      $e =$ BITFLIPMASKEDITER$(s + e \cdot H^T, e, gray, H);$
    end if
  end for
  $fail = (s == 0)\ ?\ 0 : 1;$
  return $e, fail;$
end function

function $[e, black, gray]$ BITFLIPITER(s, e, thr, H)
  $black = 0; gray = 0;$
  for $j \in 0 : n - 1$ **do**
    if $upc(H, s, j) \geq thr$ **then**
      $e_j = e_j \oplus 1;$
      $black_j = 1; gray_j = 0;$
    else if $upc(H, s, j) \geq thr - 3$ **then**
      $black_j = 0; gray_j = 1;$
    end if
  end for
  return $e, black, gray;$
end function

function $[e]$ BITFLIPMASKEDITER$(s, e, mask, H)$
  for $j \in 0 : n - 1$ **do**
    if $upc(H, s, j) \geq (((v + 1)/2) + 1)$ **then**
      $e_j = e_j \oplus mask_j;$
    end if
  end for
  return $e;$
end function

lying QC-MDPC code and the number of iterations required to decode a codeword, and it is negligible from the computational viewpoint, similarly to the other three operations, i.e., the error bit-flips computation and the update of both the error and syndrome vectors, which consist of vector operations with a linear complexity.

Black-Gray-Flip decoding

The BIKE cryptoscheme makes use of the Black-Gray-Flip (BGF) variant of the QC-MDPC bit-flipping decoding algorithm. The BGF decoding algorithm, which was introduced in [34], is listed in Algorithm 7. Similarly to the previously detailed `BFDecoding` function, the `BGFDecoding` function, which implements the BGF decoding procedure, executes a predefined number of iterations (see lines 3-10 in Algorithm 7) to produce the error vector (`e`) and the decoding failure boolean flag (`fail`) as outputs.

The BGF decoding procedure differs from the baseline one only during the first decoding iteration. In such case, two $black$ and $gray$ n-bit vectors are generated, setting their bits to 1 if the UPCs in the corresponding positions are larger than thr (lines 17-19) and $thr - 3$ (lines 20-21), respectively, where thr is a threshold computed as a function of the syndrome, as in the general bit-flipping algorithm. The $black$ and $gray$ vectors are thereafter XORed, in two separate iterations (lines 7 and 8), with the error vector bits corresponding to UPC values larger than $(((v + 1)/2) + 1)$ (lines 32-33), where v is the Hamming weight of a row of a H_i block of the parity-check matrix.

Notably, the additional black and gray iterations introduced by the BGF algorithm do not increase the complexity of the generic bit-flipping decoding procedure, since they require simple comparison and XOR operations, over n-bit vectors, which are thus characterized by linear complexity.

CHAPTER 3

State of the art

This chapter provides an overview of the state-of-the-art literature concerning the implementation of QC-MDPC code-based cryptosystems. Such implementations comprise both hardware and software ones and range from low-end embedded platforms up to higher-end desktop-class systems.

The state of the art is discussed by analyzing first the implementations of standalone binary polynomial arithmetic and QC-MDPC decoding operations, and then the complete implementations of QC-MDPC code-based cryptosystems with a particular focus on BIKE.

Parts of this chapter are derived from previously published works co-authored by the author of this thesis. In particular, the multiplication state-of-the-art discussion in Section 3.1 was adapted from [111], the exponentiation and inversion literature analysis in Section 3.2 and Section 3.3 resulted from [43], the decoding state-of-the-art discussion in Section 3.4 came from [110], and parts of Section 3.5 overviewing complete cryptoscheme implementations were taken from [41]. More details about the referenced publications are provided in Appendix A.

3.1 Binary polynomial multiplication

The state-of-the-art contains several proposals that implement multiplication for the Galois field of binary polynomials, both in the form of software libraries, custom extensions to the instruction set architecture (ISA), and hardware accelerators.

On the software side, the *gf2x* [22] software library is the de-facto reference for fast multiplication of polynomials over $\mathbb{Z}_2$, implementing several multiplication algorithms to optimize the computation for different operand sizes. In contrast, the *NTL* [95] library either implements only the Karatsuba multiplication algorithm, or it can act as an overlay to the *gf2x* library, while the *MPFQ* [45] library is specifically tailored to deliver high performance for finite fields of moderate size, when the modulus size is known in advance.

From the ISA point of view, Intel introduced the *PCLMULQDQ* instruction and the corresponding hardware support in its Westmere architecture for the purpose of accelerating the computation of the AES Galois Counter Mode (AES-GCM) authenticated encryption algorithm [48]. The *PCLMULQDQ* instruction performs the carry-less multiplication of two 64-bit operands. The work in [32] leverages the *VPCLMULQDQ* instruction, which is intended to further accelerate AES-GCM and which is the vectorized extension of *PCLMULQDQ*, to compute multiplications between large-degree binary polynomials, i.e.. polynomials with degree greater than 511. In particular, results considering polynomials of degree up to 2^{16} predict a $2\times$ speed-up compared to the previous computing platforms. Similarly, the ARMv8-A architecture provides the *VMULL.P64* instruction, which takes as inputs two 64-bit NEON registers and outputs their product, computing according to binary polynomial multiplication, on a 128-bit NEON register.

On the hardware side, the state-of-the-art contains several architectures implementing ad-hoc hardware accelerators for the Galois field of binary polynomials, either in the form of bit-serial, digit-serial, or bit-parallel multipliers.

The bit-serial architectures have a low hardware complexity, thus they are well-suited for low-power and resource-constrained implementations. In particular, such hardware accelerators output the M-bit result after M clock cycles, thus their latency strictly depends on the size of the input. [46] presents a low-power bit-serial multiplier architecture for binary polynomials for which an M-bit multiplier implementation requires $28 \times M$ gates. The limited performance and flexibility in trading performance and area utilization of bit-serial architectures, prevents their use to design multipliers with operands of size in the order of tens of thousands of bits.

In contrast, bit-parallel architectures are intended for performance-oriented implementations, since they perform the M-bit multiplication in one clock cycle. However, they are characterized by a high critical path delay and a high area consumption, which grows up more than linearly with the size of the operands [36]. To this end, the bit-parallel multipliers in the state-of-the-art are limited to relatively small operand sizes, i.e., one or two thousands of bits at the most. [29] details the realization of the optimal bit-parallel design given the structure of the target binary polynomial Galois field, i.e., the size of the polynomials of the field and its associated irreducible polynomial.

We note that all bit-parallel solutions leverage the size of the operands to deliver efficient ad-hoc architectures. To this end, each architecture is customized for a specific Galois field and it is therefore not reusable. The limited flexibility and the hardware complexity that grows with the size of the operands make the bit-parallel architectures unsuitable to design large binary multipliers intended to be implemented on a large variety of FPGA devices, regardless of the size of the operands.

Differently from bit-parallel solutions, digit-serial polynomial basis multipliers offer a superior design flexibility. In particular, the operands are organized in digits, i.e., chunks with a fixed number of bits, and the multiplication proceeds on a digit-by-digit basis. The possibility to configure the size of the digit allows to trade the performance with the resource utilization. [84] presents a low-area and scalable digit-serial architecture to perform polynomial basis multiplications over $\mathbb{Z}_2[x]$. Two digit-serial architectures for multiplication over Galois fields employing the normal basis representation are presented in [9, 85]. By rewriting the multiplication equations in a normal basis form, the design in [85] can reduce both the hardware complexity and the combinational critical path. In contrast, the digit-serial multiplier presented in [9] aims to speedup the exponentiation and the point multiplication, in any case a double multiplication is required and traditional schemes are performance-limited due to data dependences.

We note that the scalability offered by digit-serial solutions is limited to the possibility of configuring the size of the digit, i.e., the number of bits that are processed in parallel. Normally, state-of-the-art solutions are validated on limited operand sizes, less than few thousands of bits, thus the scalability issues of such solutions have not been fully highlighted. Differently, the implementation of large binary multipliers requires to extend the flexibility of current digit-serial architectures with the use of fast multiplication algorithms to aggressively reduce the number of computed partial products, without increasing the design complexity.

In particular, several works in the state-of-the-art demonstrate the pos-

sibility of implementing the Karatsuba algorithm into the multiplier to minimize the number of computed partial product and, thus, to improve the overall multiplication performance. [101] proposes a hardware multiplier employing an ad-hoc implementation of the Karatsuba algorithm for 240-bit polynomials. The design takes 30 clock cycles to perform a single multiplication, but the ad-hoc combinational logic structure severely thwarts the scalability of the multiplier. [40] presents a hardware multiplier relying on a Karatsuba-like approach. Depending on the operand size, the solution optimizes the performance by allowing to split the operands either into 4, 5, 6 or 7 blocks. However, the fixed architecture limits the scalability of the solution in the exploitation of the resources available in large FPGAs. Moreover, the design has been validated against polynomials of degree up to 99. [13] compares two implementations of binary polynomial multipliers targeting the encryption function of LEDAcrypt. Depending on the actual Hamming weight of one of the two polynomials, i.e., the number of its coefficients set to 1, the authors discuss the possibility of using dense-sparse binary polynomial multipliers rather than traditional Karatsuba-like dense-dense architectures. In particular, the dense-dense multiplier implements a single iteration of the Karatsuba algorithm and either one serial or three parallel Comba multipliers to compute the three partial products. Such multiplier works at 100 MHz and the parallel and serial versions are provided as two separate implementations.

Few hardware implementations of multipliers specifically targeting the BIKE cryptosystem are also available in the state-of-the-art literature. [56] considered the FPGA-based design of two polynomial multipliers for BIKE, one implementing the multiplication between two dense operands and one implementing the multiplications between a dense and a sparse operand. The FPGA-based implementation by the authors of BIKE in [89] employed a multiplication module that minimized the BRAM usage by parallelizing the computation of a simpler schoolbook multiplication algorithm rather than applying a more complex one such as Karatsuba's. Such area-optimized approach comes at the expense of performance, with the execution time having a quadratic relation to the p/b ratio, i.e., $\mathcal{O}(\lceil p/b \rceil^2)$, where p is the polynomial length and b is the bandwidth of the multiplication module. The latter is the only configurable parameter in the proposed multiplication architecture. Finally, the later FPGA-based implementation of BIKE in [88] only implements a dense-sparse multiplication module, which exploits the sparse representation of one of the two operands in the binary polynomial multiplication.

3.2 Binary polynomial exponentiation

Few implementations of the exponentiation algorithm have been proposed in the last decade to efficiently support the key generation algorithm in post-quantum QC-MDPC code-based cryptosystems and more in general the computation of binary polynomial inversion.

[33] proposed two main optimizations for the software computation of $GF(2^m)$ exponentiations. First, the permutation corresponding to a $f(x)^{2^k}$ exponentiation is fully precomputed by storing in a lookup table the positions of bits in the inverse polynomial and indexing them by the original positions in the input polynomial $f(x)$. Lookup tables can be precomputed for all values held by k during the inversion algorithm, which depend exclusively on p. Second, $f(x)^{2^k}$ exponentiations are executed faster as a chain of k squarings, when k is smaller than a threshold value, which depends on the underlying computing architecture. The authors exploited the Intel AVX2 and AVX512 extensions to further improve the performance of the software computation.

However, the proposed lookup tables required $p \cdot (\lceil \log_2(p-2) \rceil - 1) \cdot \lceil \log_2 p \rceil$ bits of memory, and may thus not be suitable to constrained devices such as microcontrollers. [14] optimized the memory requirements by using a smaller lookup table, that holds only the $(\lceil \log_2(p-2) \rceil - 1)$ values obtained as $2^i \bmod p$, with $i \in \{1, 2, ..., \lceil \log_2(p-2) \rceil\}$. The position of the j-th coefficient, where $0 \leq j \leq p-1$, of $a(x)^{2^i}$ is instead computed at run-time as $(j \cdot (2^i \bmod p)) \bmod p$, i.e., through a multiplication and a modulus operation.

[89] proposed three different strategies for the computation of the $f(x)^{2^k}$ exponentiation with arbitrary k. The first one iterates k squaring operations, i.e., $f(x)^2$, processed by a squaring module. The second one implements two modules, one computing $f(x)^2$ and the other computing $f(x)^{2^4}$. The latter is used as long as the remaining exponent of the squaring chain is ≥ 4, otherwise the iterative computation is done by the former. The third strategy combines a fixed squaring module computing $f(x)^2$ and a module that directly computes $f(x)^{2^k}$ exponentiations with arbitrary k. $f(x)^{2^k}$ exponentiations are executed by the latter module when $k \geq BW$, where BW is the width of the architecture datapath, otherwise they are computed by iterative squaring. The third strategy was shown to provide the best performance, while occupying slightly more resources than the first one.

3.3 Binary polynomial inversion

Several algorithms to compute the multiplicative inverses in $GF(2^m)$ and their hardware implementations have been proposed since the 1980s. In general, the state-of-the-art proposals can be split into two main families, deriving respectively from Fermat's little theorem and Euclid's algorithm.

Fermat's little theorem states that, if p is a prime number, then for any integer a the number $a^p - a$ is an integer multiple of p, and it was at the core of the first state-of-the-art proposals, such as [103] and ITA [60], and other Fermat-based software and hardware implementations were later introduced by [37, 55, 70, 86, 91, 92, 97].

Euclid's algorithm computes the the greatest common divisor of two natural, i.e., positive integer numbers, and it can be generalized to binary polynomials. It was first adapted to compute multiplicative inverses in $GF(2^m)$ by [23], which introduced the inversion procedure known as Brunner's algorithm. Subsequent proposals based on Euclid's and Brunner's algorithms were [18, 49, 66].

The following paragraph details the earlier literature proposals for binary polynomial inversion algorithms and their hardware and software implementations. [103] first proposed in 1985 an algorithm that required $(m-2)$ multiplications and $(m-1)$ cyclic shifts, where m is the length of the binary polynomials, and targeted a VLSI implementation. [60] proposed the Itoh-Tsujii algorithm (ITA), requiring at most $(m-1)$ cyclic shifts and $2(\log_2(m-1))$ multiplications, with a significant reduction from [103]. [37] introduced an algorithm with $O(m \log_2 m)$ time complexity, that employs serial-in-parallel-out multiplication and is suited for VLSI implementation. While the previous proposals derived from Fermat's little theorem, [23] presented a design based on the Euclid's algorithm. It computes the inverse by repeated shifts and subtractions, enabling an efficient hardware implementation with $O(m)$ complexity for both area and time. [49] proposed a variant of the algorithm in [23] that made it more suitable to a systolic architecture. The two proposed architectures are parallel-in parallel-out and serial-in serial-out systolic arrays, with throughput 1 and $1/m$, respectively, and both with $(8m-1)$ cycles latency. [97] proposed a Fermat-based algorithm that reduced the number of required multiplications by decomposing $m-1$ into several factors and a small remainder. Such decomposition is applied recursively until finding the minimum required number of multiplications [91] proposed an efficient architecture based on ITA that makes use of addition chains and targets FPGA devices. It is tailored to a special class of irreducible trinomials, and results are shown for $GF(2^{193})$. [92]

further improved the performance on the same Galois field by deriving a parallel formulation of the ITA. [86] modified the ITA algorithm to better use the available FPGA resources and require shorter addition chains, and showed the proposed method to be scalable for polynomial degrees comprised between 100 and 300. [66] proposed an algorithm that targeted modern CPUs with hardware support for fast carry-less multiplication. The operations in several contiguous iterations of the extended Euclid's algorithm are represented as a matrix nd can then be performed at once by means of a single carry-less multiplication instruction, resulting in fewer multiplication and XOR instructions. [55] improved and parallelized ITA, targeting FPGAs and polynomials with length in the order of hundreds. [70] modified ITA to reduce latency by enabling the parallel computation between some multiplications and squarings. It was implemented on FPGA and targeted polynomials with degree in the order of few hundreds for elliptic curve cryptography. [18] introduced a divide-et-impera strategy for computing greatest common divisors and inverses for generic $GF(p^m)$ rings in constant time.

Remarkably, all the previously listed state-of-the-art proposals targeted binary polynomials with degree in the order of few hundreds at most, due to the lesser requirements of traditional PKC and error control coding schemes.

Only few and more recent proposals target instead polynomials with degrees in the order of tens of thousands, that are thus suitable to post-quantum QC-MDPC cryptography. They are software and hardware implementations of the BIKE and LEDAcrypt cryptosystems. The software ones target modern *x86_64* CPUs that support custom instructions for carry-less multiplication, while the hardware one targets Artix-7 FPGAs.

[33] introduced a constant-time algorithm for polynomial inversion, targeting the software implementation of BIKE and based on Fermat's little theorem. The authors optimized the exponentiation operation and further improved performance by means of a source code targeting the latest Intel Ice Lake CPUs, that support the AVX512 and *VPCLMULQDQ* instructions.

[14] presented a Fermat-based algorithm, not dissimilar from the one introduced in [33], that is employed in the software implementation of LEDAcrypt and was previously detailed in Section 2.4.1.

[89] presented the FPGA-based implementation of BIKE that employs an inversion algorithm based on [55]. The employed algorithm differs from the one used in [33], requiring the same number of exponentiations, but slightly less operations if the exponentiations are computed by means of iterated squarings. Notably, the inversion algorithms employed in the [14, 33, 89] implementations of BIKE and LEDAcrypt require the same

number of exponentiations and multiplications.

The authors of [89] later applied instead the extended Euclidean algorithm first proposed by [18], rather than a Fermat-based one, to another FPGA-based implementation of BIKE [88].

3.4 QC-MDPC bit-flipping decoding

LDPC and QC-LDPC codes have traditionally been used in telecommunication applications ranging from wired, e.g., 10GBase-T Ethernet [106], to wireless ones, e.g. WiMax (IEEE 802.16e) and WiFi (802.11n) [50], due to their superior error-correction capabilities [10]. However, the recent advances in quantum computing highlighted the possibility of employing the class of QC-LDPC and QC-MDPC codes as the codes underlying code-based quantum-resistant cryptosystems [93]. QC-LDPC codes were indeed employed by the LEDAcrypt cryptoscheme, which was admitted to the third round of the NIST PQC standardization process [11], while BIKE, currently a candidate for standardization in the fourth round, akes use of QC-MDPC codes [5].

From the decoding point of view, the state of the art contains several proposals addressing the optimized design of decoders to support QC-LDPC codes. In the following we classified them in two main groups: i) soft-decision decoders, e.g., Belief Propagation (BP), Sum-Product Algorithm (SPA), and their variations, that employ a message-passing structure, and ii) hard-decision decoders, i.e., bit-flipping algorithms, designed to offer a simple decoder implementation. Traditionally, soft-decision decoders offer superior decoding performance than hard-decision ones, i.e., bit-flipping solutions, thanks to the use of the channel information. In contrast, bit-flipping decoders have a favorable less complex design. The rest of this section discusses the state-of-the-art proposals on decoding, targeting QC-LDPC codes. We note that the review aims to highlight the main limitations and constraints that prevent the use of current state-of-the-art solutions in the design of QC-LDPC decoders for post-quantum cryptography.

Among the soft-decision decoders, [62] proposed a FPGA-based QC-LDPC decoder for the Chinese Digital Television Terrestrial Broadcasting (DTTB) standard, which is based on the soft-decision min-sum algorithm. [104] describes a parallel GPU implementation of the soft-decision min-sum decoder for QC-LDPC codes, targeting both the WiMax and WiFi standards. Despite the interesting performance of the proposed parallel GPU decoder, the underlying QC-LDPC $\mathbf{C}(n, k)$ codes for WiFi and WiMax have the (n, k) pair of parameters equal to $(1944, 972)$ and $(2304, 1152)$ for WiFi

and WiMax, respectively, thus tens of times smaller than the ones employed in post-quantum QC-LDPC cryptosystems. An additional hardware implementation of a soft-decision decoder for the 802.11n WiFi standard, thus targeting small codes, is proposed in [98]. In contrast, [107] presents a 90nm CMOS implementation of a soft-decision decoder for QC-LDPC codes with n values up to 96000 bits. Despite the code size is aligned with the one employed in current QC-LDPC-based cryptosystems, the decoder in [107] is tailored to a specific code structure that is intended for telecommunications. To this end, the underlying code cannot offer the security margin required by post-quantum code-based cryptosystems.

Considering telecommunication applications, the use of soft-decision decoders represents the optimal choice due to the possibility of implementing a system approaching the channel capacity limit [10]. However, such superior performance is achieved by leveraging the channel information in the decoding procedure. To this end, QC-LDPC and QC-MDPC codes meant for post-quantum cryptosystems can not employ soft-decision algorithms, since the cryptosystem is expected to operate even when the channel information is not available, e.g., encryption and decryption of digitally stored data. Moreover, the complexity of soft-decision decoders limits their scalability in supporting large QC-LDPC and QC-MDPC codes [59]. The state-of-the-art contains several families of proposals, i.e., Weighted Bit Flipping (WBF) [68], Modified WBF (MWBF) [63], and Gradient Descent Bit Flipping (GDBF) [102], aiming at optimizing the performance of the baseline bit-flipping algorithm, i.e., hard-decision decoders. However, for each of them, the performance improvement is obtained by leveraging some sort of channel information, thus preventing their use in the design of decoders for post-quantum cryptosystems [59]. In summary, the baseline bit-flipping algorithm represents the most important candidate to deliver hardware accelerated decoders for quantum-resistant QC-LDPC and QC-MDPC cryptosystems. We note that such design choice is also supported by the fact that all the QC-LDPC and QC-MDPC code-based cryptosystems participating in the third round of the NIST PQC standardization process made use of the bit-flipping decoding procedure in its baseline version or in a variant with slight modifications.

Among the hard-decision decoders, [53] presents a hardware implementation of the LEDAcrypt KEM submitted to the first round of the NIST competition. Such version of the cryptosystem proposes a variant of the bit-flipping decoder, i.e., the Q-decoder, that has been dismissed due to a set of security vulnerabilities in the theoretical decoding scheme [4]. In fact, the current LEDAcrypt submission to the third round of the NIST competition

employs a baseline bit-flipping decoder, thus making the work in [53] obsolete. [100] proposes a lightweight implementation of a bit-flipping decoder for QC-LDPC codes. Despite the fact that the solution in [100] does not offer a configurable area-performance trade-off, it suffers two other limitations. On the one hand, the decoding execution time is in the order of tens of milliseconds. Onthe other hand, the design is limited to small QC-LDPC codes that offer an 80-bit security level, while post-quantum cryptography mandates larger codes to achieve a security level comprised between 128 and 256 bits. The BIKE round 3 specification document [5] discusses the decoder implementation of the BIKE QC-MDPC code-based cryptosystem, that leverages a light variant of the bit-flipping algorithm. In particular, the baseline bit-flipping algorithm has been slightly modified in its first iteration to conditionally perform an additional error correction task. We also note that a software implementation of the BIKE bit-flipping decoder employing the *Intel AVX512* extension is discussed in [34]. In a similar manner, both the reference *C11* and the optimized *Intel AVX2* software implementations of the bit-flipping decoder employed in the current version of the LEDAcrypt cryptosystem, are discussed in its third round specification document [11].

All the previously listed solutions however either are software-implemented ones, with a few of them specifically designed for post-quantum QC-LDPC or QC-MDPC code-based cryptosystems, or hardware-implemented ones that only target telecommunications QC-LDPC codes.

To this end, it is of paramount importance to provide efficient and scalable hardware decoders to support the emerging QC-MDPC cryptosystems, since the available software solutions reveal the impossibility to cope with the required performance, more so when considering the steep increase of the key length expected in the near future. Few recent works proposed hardware bit-flipping decoders specifically designed to be suitable for post-quantum QC-MDPC cryptography applications.

[67] proposed a bit-flipping decoder for QC-MDPC codes, that is, however, only configurable in the bandwidth of its datapath, with the goal of supporting the LEDAcrypt PQC cryptosystem. [89] provided the first FPGA-based implementation of the BGF decoder employed by BIKE. The proposed decoding architecture is composed of three modules, respectively devoted to computing the threshold function, computing the Hamming weight of the updated syndrome vector, and updating the error vector in both the traditional bit-flipping and black and gray iterations. The architecture is configurable in the bandwidth parameter, which allows delivering different decoding instances depending on the desired area and performance. For instance, such parameter corresponds to the number of UPCs computed in

parallel and to the number of error vector bits concurrently evaluated in the bit-flipping phase. A later FPGA-based implementation of BIKE by the same authors [88] also uses the same decoding architecture as [89].

3.5 KEM primitives

While several works focused on the optimal design of single key operations to support QC-MDPC code-based cryptosystems, the literature also contains few proposals that provide complete hardware or software implementations of QC-MDPC code-based post-quantum cryptosystems.

[51, 100] proposed the implementation of the McEliece cryptosystem with QC-MDPC codes on FPGAs. In particular, [51] targeted a performance-oriented design while [100] focused on a resource-optimized one. [54] discussed a fast implementation of QC-MDPC Niederreiter encryption for FPGAs, outperforming the work in [51] thanks to using a hardware module to estimate the Hamming weight of large vectors and proposing a hardware implementation tailored to low-area devices for encryption and decryption used in QC-MDPC code-based cryptosystems. The authors of BIKE presented a VHDL FPGA-based implementation, targeting Xilinx Artix-7 FPGAs and providing support for the key generation, encryption, and decryption KEM primitives on a unique design [89]. However, the proposed architecture was custom-tailored to smaller FPGA targets, up to Artix-7 100, and it employed the AES and SHA-2 cryptographic functions as random oracles, thus supporting a now obsolete specification of BIKE. Finally, [88] proposed an updated FPGA-based implementation that employed a Keccak core rather than AES and SHA-2, as specified in the latest version of the BIKE cryptoscheme [6]. The proposed architecture targets Artix-7 FPGAs and the authors listed three instances implementing the whole KEM providing a range of area-performance trade-offs. The smallest one requires less resources than the lightweight one from [89] and provides a more than $3\times$ speedup, while the largest one takes 3.7ms compared to the 4.8ms of the high-speed one from [89] while also occupying a smaller area.

On the software side, implementations of QC-MDPC code-based cryptosystems participating in the NIST PQC competition are open-source and publicly available. Two separate software versions of LEDAcrypt, a reference one written in plain C11 and an optimized one that exploits the AVX2 extension for recent Intel Core CPUs, are available at [11]. [5] provides instead two software implementations of BIKE, a reference one written in plain C11 and an optimized one that exploits the Intel AVX512 extension. Other works from literature provide software implementations for ISAs

other than the Intel x86 one, with [26] targeting ARM Cortex-M4 microcontrollers and [27] introducing support for RISC-V computing platforms. Further additional implementations of BIKE, including a fully portable one, versions optimized for AVX2 and AVX512 instruction set extensions, and implementations optimized for CPUs that support *PCLMULQDQ* and *VPCLMULQDQ* instructions, are also publicly available on Github [3].

Finally, [74] proposed a mixed hardware-software (HW/SW) approach that made use of three HLS-generated accelerators, each implementing one of the BIKE primitives. The HW/SW approach allowed mixing the usage of hardware acceleration for the most computationally expensive primitives with the software execution of the least complex ones. The proposed solution resulted in three different combinations of hardware-implemented and software-executed KEM primitives on three chips from the Xilinx Zynq-7000 family of heterogeneous SoCs, which feature ARM CPUs coupled with programmable FPGA logic equivalent to the Artix-7 one.

CHAPTER 4

Methodology

This chapter details the proposed hardware architecture by following a top-down approach.

First, it provides an overview of the architecture of client and server top modules implementing the three BIKE KEM primitives [41] and discusses the performance profile of software across different computing platforms, allowing to identify the operations on which to focus RTL design effort. Then, it details the architecture of modules implementing such most complex operations, namely bit-flipping decoding [110] and binary polynomial inversion [43], multiplication [111], and exponentiation [43], as well as pseudorandom number generation and SHA-3. Finally, it discusses a heuristic for design space exploration that leverages the time- and space-complexity analysis of the employed configurable components to steer the fast identification of the architectural parameters that deliver the best hardware support [41].

Parts of this chapter are derived from previously published works co-authored by the author of this thesis. The description of the KEM primitives architecture in Section 4.1 and of the complexity-based DSE heuristic in Section 4.8 were derived from [41]. The architectural description of the decoding component in Section 4.2 was updated from the one introduced in [110]. The descriptions of the inversion and exponentiation components'

architectures in Section 4.3 and Section 4.5, respectively, originated from [43]. Finally, the discussion of the dense-dense multiplication architecture in Section 4.6 was derived from [111]. More details about the referenced publications are provided in Appendix A.

4.1 KEM primitives architecture and software profiling

The proposed hardware architecture foresees separating the components that implement the operations being carried out on the KEM client and server nodes. The client node is tasked with performing the key generation and decapsulation primitives, while the server one is limited to executing the encapsulation, as previously discussed in Section 2.1, The two client and server architectures are detailed separately in the following.

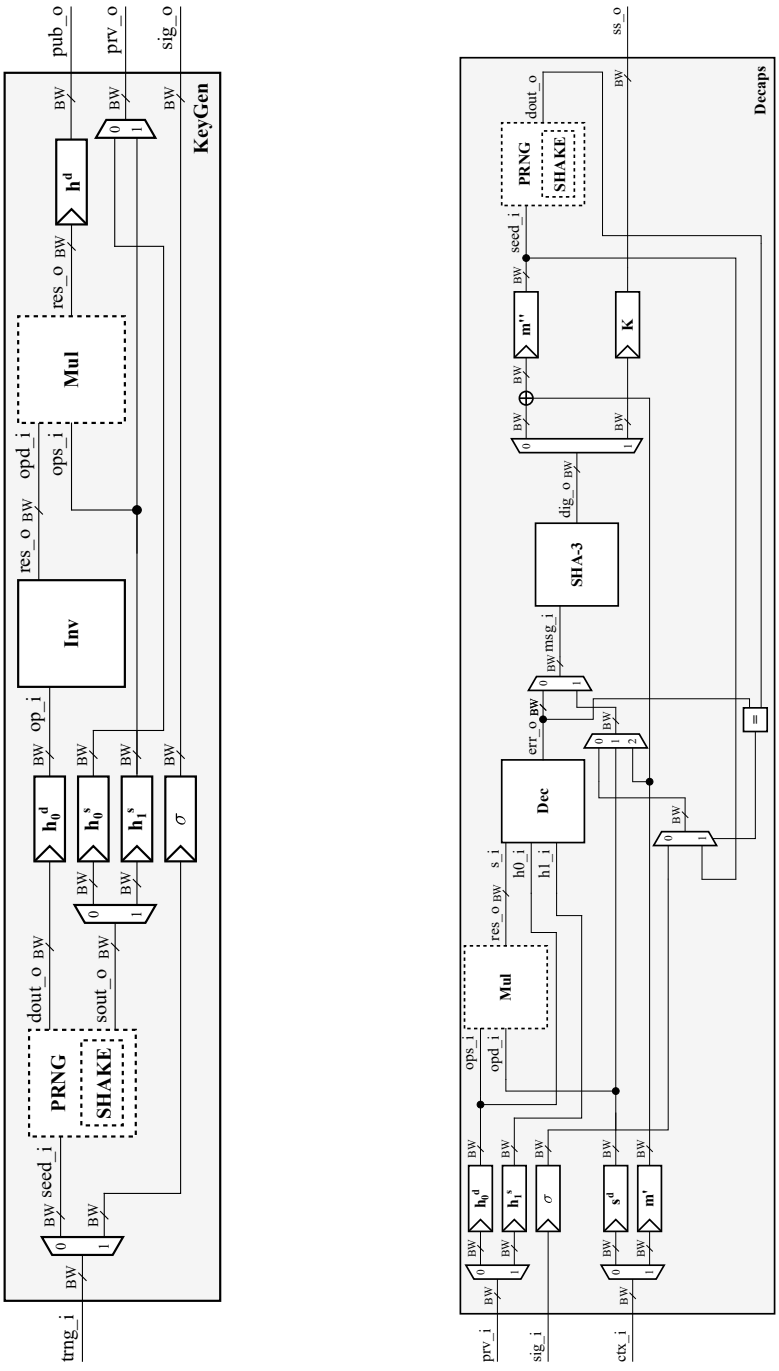
4.1.1 Client architecture

The client architecture implements the cryptographic core to support the client-side execution of BIKE. The client architecture consists of two main modules, key generation (`KeyGen`) and decapsulation (`Decaps`), which are depicted in Figure 4.1a and Figure 4.1b, respectively. In order to minimize duplicate hardware resources, the pseudorandom generator (`PRNG`) and the multiplier (`Mul`) are shared between the `KeyGen` and `Decaps` modules and are depicted as dashed blocks in Figure 4.1a and Figure 4.1b.

Key generation

The `KeyGen` module, depicted in Figure 4.1a, implements Algorithm 1. It features a BW-bit input (`trng_i`) to receive a 256-bit random value from the external true random number generator (TRNG), and outputs the public (h) and private (H) keys using the `pub_o` and `prv_o` outputs, respectively. Notably, the random output value σ that is part of the key generation procedure (see Algorithm 1), is produced by the external TRNG during the key generation procedure and output via the `sig_o` port.

The implementation of the key generation algorithm of BIKE requires performing three subsequent hardware operations, namely pseudorandom generation (`PRNG`), binary polynomial inversion (`Inv`), and binary polynomial multiplication (`Mul`), each computed by a dedicated component. The `PRNG` component is tasked with generating the private key H, that is composed of the h_0 and h_1 binary polynomials. The h_0 polynomial is produced in both the sparse and dense forms, while h_1 is stored only in its sparse form. The dense-represented h_0 polynomial is then inverted by the



(a) *Key generation* **(b)** *Decapsulation*

Figure 4.1: *Top-view architecture of the key generation and decapsulation modules. The blocks with dashed lines are shared by the two modules.*

Inv component, and the result of the binary polynomial inversion is finally multiplied by the sparse h_1 polynomial within the Mul component, that performs binary polynomial multiplication between a dense polynomial and a sparse one. The output of the Mul component, i.e., the result of the binary polynomial multiplication, corresponds to the public key h, that is output through the pub_o BW-bit port, while the previously obtained private key H is output through the prv_o BW-bit port.

Decapsulation

The Decaps module, shown in Figure 4.1b, implements Algorithm 3. It features three BW-bit inputs to receive the private key H (prv_i), σ (sig_i), and the shared secret encrypted by the server c (ctx_i), and it outputs the shared secret K through ss_o.

The implementation of the decapsulation primitive of BIKE requires performing a sequence of four hardware operations, namely binary polynomial multiplication (Mul), QC-MDPC bit-flipping decoding (Dec), computation of SHA-3 hash digest (SHA-3), and pseudorandom generation (PRNG), each computed by a dedicated component. The dense syndrome s, which is part of the ciphertext c, is first multiplied by the sparse polynomial h_0 within the dense-sparse binary polynomial multiplier Mul. The resulting product corresponds to the s' vector, which is then fed to the bit-flipping decoding component Dec together with the private key H to decode it and obtain the e' n-bit error vector. The latter is subsequently hashed by the SHA-3 module and the resulting digest is XORed with m', obtaining m''. Thereafter, if the result of the pseudorandom generation seeded by m'' is equal to e', then the SHA-3 module computes the digest of $\{m'', c\}$, i.e., m'' concatenated to the ciphertext c, otherwise it hashes $\{\sigma, c\}$ to avoid information leakage while also raising a decapsulation error flag. In case of a successful decoding, the resulting digest is the shared secret, which is output by the ss_o BW-bit port.

4.1.2 Server architecture

The server architecture implements the cryptographic core to support the server-side execution of BIKE. The architecture of the server consists of the encapsulation module (Encaps). Although the software execution of the encapsulation is significantly faster than the more complex key generation and decapsulation, the web server scenario foresees a multitude of concurrently active connections, thus mandating for efficient hardware support also for encapsulation.

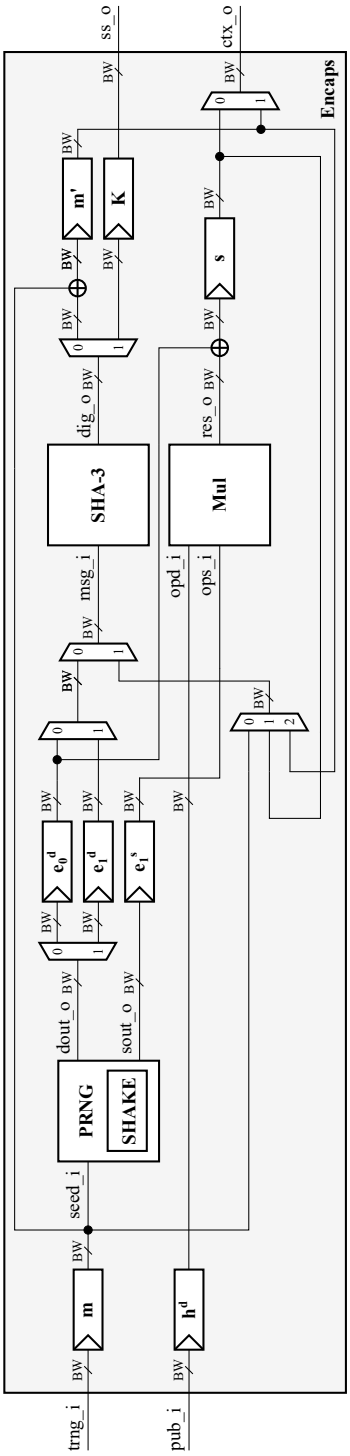


Figure 4.2: *Top-view architecture of the encapsulation module.*

Encapsulation

The Encaps module, which is depicted in Figure 4.2, implements Algorithm 2. It takes as inputs a 256-bit random message m and the public key h through the BW-bit trng_i and pub_i ports, respectively, and outputs the shared secret K, through ss_i, and the ciphertext c, i.e., the shared secret encrypted with the public key received from the client, through ctx_o.

The implementation of the encapsulation primitive of BIKE requires performing a sequence of three hardware operations, namely pseudorandom generation (PRNG), binary polynomial multiplication (Mul), and computation of the SHA-3 hash function (SHA-3), each computed by a dedicated component. m is first expanded by the PRNG component to generate the random error vector $e = [e_0|e_1]$ with Hamming weight t, and the dense-represented h is then multiplied by the sparse-represented e_1 in the dense-sparse binary polynomial multiplier Mul. The resulting product is then XORed with e_0, obtaining the syndrome s. m' is computed by XORing the message m with the SHA-3 hash digest of the error vector e, and the concatenation of s and m' corresponds to the ciphertext c. Finally, the shared secret K is produced as the SHA-3 hash digest of the message m concatenated to the ciphertext c. c is output through the ctx_o BW-bit port, while the ss_o BW-bit port outputs the shared secret K.

4.1.3 Profiling of software performance

In order to understand on which parts of the BIKE cryptoscheme to focus the design effort, we first evaluate how it performs when executed in software on a range of different computing platforms. Such an analysis considers 32- and 64-bit architectures, ARM and x86 ISAs, embedded- and desktop-class processors, plain-C99 and AVX2-optimized software, and NIST security level 1 and 3 instances of BIKE.

Table 4.1 details the performance profile of the software execution of BIKE on the different computing platforms, highlighting the ratio of execution time taken by the main operations on the client and server nodes of the key exchange. The BIKE cryptoscheme was executed 100 times for each considered combination of CPU, software implementation, and security level, collecting the execution times and computing their average.

The performance profile data was collected on a 32-bit ARM Cortex-A9 CPU, on a 64-bit ARM Cortex-A53 CPU, and on a Intel Core i5-10310U CPU. ARM Cortex-A9 is an embedded-class 32-bit processor implementing the ARMv7-A instruction set architecture (ISA), ARM Cortex-A53 is an

Table 4.1: *Breakdown of the percentage execution times of BIKE for different security levels, architectures, and software implementations. Legend: C client, S server node, Kg key generation, En encapsulation, De decapsulation primitive, PRNG and* ***H*** *pseudorandom generation, Inv. inversion, Mult. multiplication, Dec. decoding, Other other operations executed in the KEM primitives,* ***K*** *and* ***L*** *SHA-3 hash function.*

			Target CPU, software version and NIST security level							
			ARM32		ARM64		Intel		Intel	
KEM	KEM		C99 [5]		C99 [3]		C99 [3]		AVX2 [3]	
node	prim.	Op.	SL1	SL3	SL1	SL3	SL1	SL3	SL1	SL3
C	Kg	PRNG	0.1%	0.1%	0.8%	0.7%	0.5%	0.4%	1.0%	0.6%
		Inv.	40.0%	41.9%	35.7%	36.5%	44.7%	45.7%	17.8%	16.9%
		Mult.	1.6%	1.7%	1.8%	1.8%	2.2%	2.3%	0.8%	0.7%
		Other	0.0%	0.0%	0.1%	0.0%	0.1%	0.0%	0.4%	0.1%
	De	Dec.	58.1%	56.2%	58.3%	58.1%	50.4%	49.8%	73.8%	77.3%
		L func.	0.1%	0.1%	0.1%	0.1%	0.2%	0.1%	1.3%	0.9%
		H func.	0.1%	0.1%	1.4%	1.3%	0.8%	0.7%	1.4%	1.0%
		K func.	0.0%	0.0%	0.1%	0.0%	0.1%	0.1%	0.7%	0.4%
		Other	0.0%	0.0%	1.6%	1.4%	1.1%	0.9%	3.0%	1.9%
S	En	**H** func.	5.8%	3.0%	39.7%	40.0%	25.1%	23.3%	32.5%	33.0%
		Mult.	85.4%	90.9%	50.3%	54.5%	65.9%	71.2%	15.8%	22.7%
		L func.	4.3%	2.7%	3.9%	2.6%	4.7%	3.1%	26.1%	26.1%
		K func.	1.2%	0.8%	2.2%	1.4%	2.6%	1.6%	17.1%	13.6%
		Other	3.3%	2.6%	3.8%	1.5%	1.7%	0.7%	8.5%	4.5%

embedded-class 64-bit processor implementing the ARMv8-A ISA, and Intel Core i5-10310U is a desktop-class 64-bit processor implementing the x86-64 ISA and providing support for the Intel AVX2 extension.

On the ARMv7-A platform, the execution of the reference implementation from the official BIKE NIST submission [5] resulted in a performance profile characterized by binary polynomial inversion and BGF decoding occupying up to 42% and 58% of the execution time on the KEM client side, with binary polynomial multiplication taking instead up to 91% of the execution time on the server side.

The execution of the portable additional implementation of BIKE, compatible with 64-bit ARM architectures and written in C99 [3], on the ARMv8-A CPU highlighted binary polynomial inversion and BGF decoding taking up to 37% and 58% of the execution time on the client side, with uniform random number generation and binary polynomial multiplication taking instead up to 40% and 55% of the execution time on the server side.

Executing the same C99 software [3] on the Intel x86-64 processor saw client-side execution time being almost equally distributed between inversion and decoding, taking up to 46% and 50%, respectively, while the server-side

execution is more unbalanced with multiplication taking up to 71% and PRNG taking up to 25%. Overall, the results are quite similar to ARMv8-A software execution, due to not using any Intel-specific optimization.

On the contrary, the execution of the AVX2-optimized software [3] on the same Intel CPU produced quite different results. On the client side, the decoding procedure takes a larger portion of the execution time, up to 77%, while inversion only takes up to 18%. On the server side, the execution time is distributed between SHA3-384 computation, PRNG, and multiplication, taking up to 43%, 33%, and 23%, respectively. Notably, AVX2 instructions provide the higher speedup to the operations in binary polynomial arithmetic, namely multiplications and inversions, where the latter is computed as iterated multiplications and exponentiations. Binary polynomial multiplications and inversions end up therefore taking smaller shares of the KEM execution time.

Overall, the obtained results support the decision to focus the design effort for the proposed FPGA-based hardware architecture on the optimization of the QC-MDPC bit-flipping decoding, binary polynomial inversion, and binary polynomial multiplication operations. The architecture of each of these components is described in detail in the following sections.

4.2 QC-MDPC bit-flipping decoding architecture

The decoding architecture implements the QC-MDPC bit-flipping algorithm detailed in Algorithm 6. This section overviews the decoding component introduced in [110] discussing its parametric architecture and outlining the time and space complexity as functions of both the architecture and code parameters.

Figure 4.3 shows the architectural top level view of the proposed decoder (`BF-decoder`). The `BF-decoder` takes the syndrome (s) and the parity-check matrix (H) in input, and it outputs the error vector (e) and the boolean flag ($fail$) to signal any failure in the decoding procedure.

From the computational viewpoint, the BF-decoder is made of two stages to calculate the UPCs (`calcUpc`) and the syndrome bit-flips (`calcBf`). We note that the decoding architecture is optimized by leveraging the sparseness and quasi-cyclic properties of QC-MDPC codes. Indeed, only the positions of the v ones of the first row of each H_i block are stored, since each H_i block is both a sparse and a circulant matrix.

The `calcUpc` stage takes the syndrome and the blocks of the H matrix in input and it outputs the UPCs and the weight of the syndrome. At the beginning of a new decoding, i.e., when the `isNewDec` signal of `MUX1`

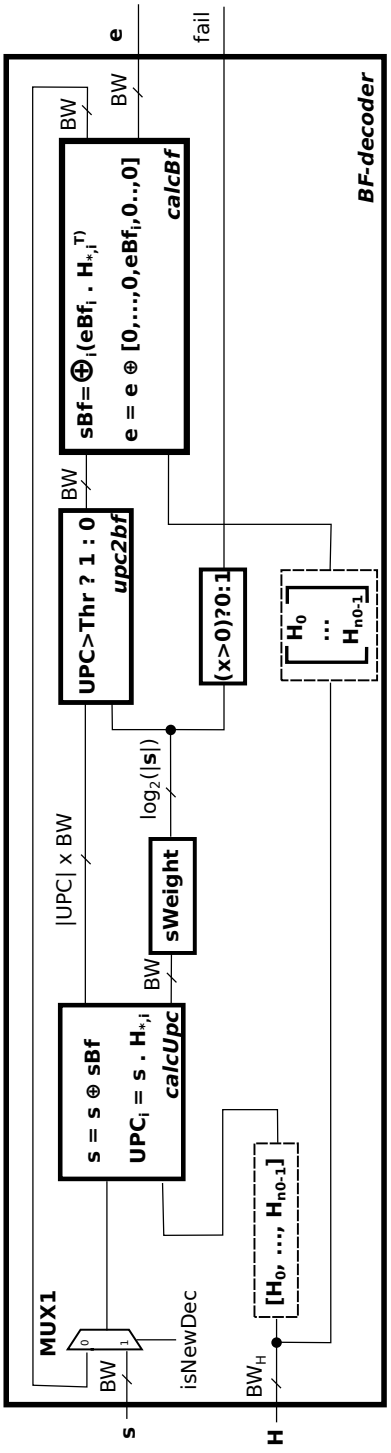


Figure 4.3: *Top-level view of the proposed bit-flipping decoding architecture.*

is equal to 1, the initial syndrome is received from the primary inputs. In contrast, for each subsequent iteration of the decoding algorithm, the syndrome is updated with the syndrome bit-flip vector generated by the `calcBf` stage. At the start of each iteration, the weight of the syndrome is computed by the `sWeight` module, and its value is used by the `upc2bf` module. The `calcUpc` module sequentially computes the UPCs for each H_i block of the H matrix. Once the UPCs from the H_i block, i.e., UPC_i, have been computed, they are passed to the `upc2bf` module with a bandwidth equal to BW times the size in bits of the maximum UPC value (which is v). The `upc2bf` module filters the incoming UPCs by comparing them with the UPC threshold to produce, as a result, the error bit-flip vector (eBf_i). We note that the eBf_i vector is fed to the `calcBf` stage to *i)* compute the syndrome bit-flips, and to *ii)* update the error vector.

Within each iteration of the bit-flipping algorithm, the `calcBf` stage updates both the syndrome bit-flips and the error vector starting from any incoming error bit-flip vector (eBf_i). In particular, the chain made of the `calcUpc` and the `upc2bf` modules produces a set of n_0 eBf_i vectors, where each of them corresponds to a specific H_i block of the parity-check matrix. To this end, the `calcBf` stage receives n_0 eBf_i vectors, i.e., one for each H_i block in the H matrix, to compute the fully updated syndrome bit-flip vector (sBf), as well as the update of the error vector.

We note that the error vector e is made by n_0 blocks of size $1 \times p$, thus each error bit-flip vector (eBf_i) updates a portion of the error vector (see the $e = e \oplus [\mathbf{0}, ..., \mathbf{0}, eBf_i, \mathbf{0}, ..., \mathbf{0}]$ update equation in Figure 4.3). In contrast, sBf is a $1 \times p$ row-vector obtained by performing the bitwise XOR of all the received eBf_i row-vectors (see $sBf = \oplus_i(eBf_i \cdot H*, i^T)$ equation in Figure 4.3). At the end of each iteration of the decoding procedure, i.e., lines $4 - 8$ in Algorithm 6, the bitwise XOR between the current syndrome and the syndrome bit-flip vector is performed in the next iteration (see line 9). From the architectural viewpoint, the syndrome update is performed by the `calcBf` module (see the $s = s \oplus sBf$ equation in Figure 4.3).

4.2.1 Dual-memory computing architecture

Apart from the configurable bandwidth (BW) that is used to trade the performance with the resource utilization, the proposed decoder implements a dual-memory architecture to perform the efficient and scalable computation of the two most time-consuming operations in the bit-flipping decoding procedure, i.e., the UPC computation ($UPC_i = s \cdot H_i$) and the generation of the syndrome bit-flip vector ($sBf_i = eBf_i \cdot H_i^T$).

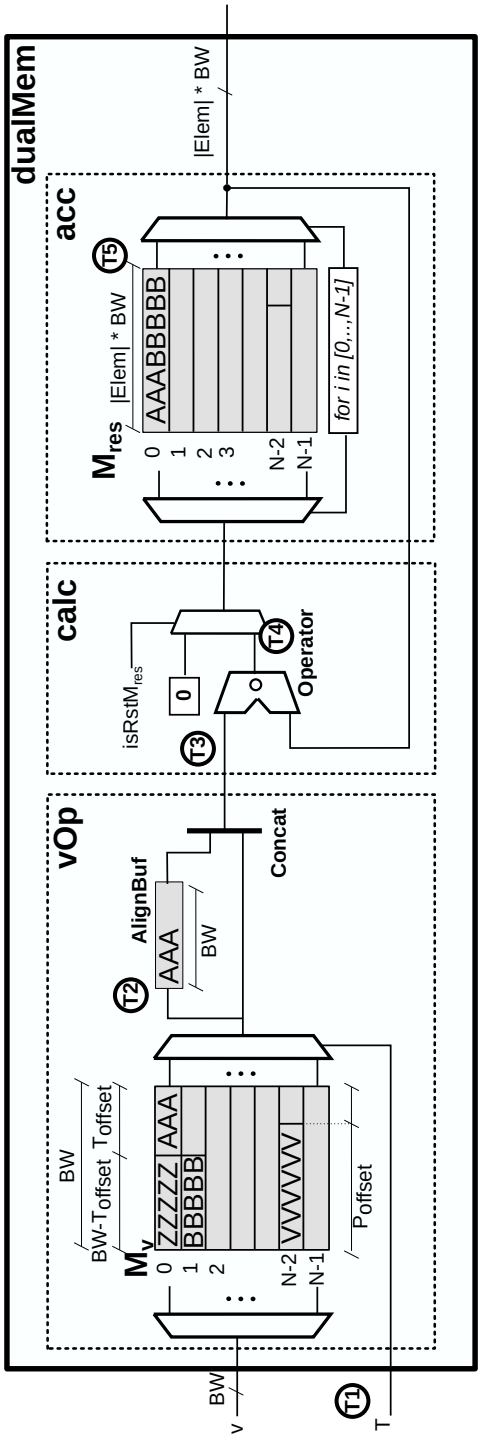


Figure 4.4: *Detailed view of the proposed dual-memory architecture.*

This section discusses the proposed dual-memory architecture that is meant to perform the efficient vector-matrix multiplication between a vector v and a sparse circulant matrix A. We demonstrate that the use of such an architecture allows to adopt an efficient divide-and-conquer approach in the computation, thus delivering an additional knob to trade the performance with the resource utilization.

Figure 4.4 shows the `dualMem` dual-memory architecture as made of three stages, i.e., `vOp`, `calc`, and `acc`. The operand (vOp) and accumulator (acc) stages implement two memory elements to store the vector v and the partial result vector, respectively. In contrast, the compute stage ($calc$) performs the actual vector-matrix computation starting from the inputs from both the accumulator and the operand stages. The dual-memory architecture receives two inputs, i.e., the vector v and the position of the ones in the A matrix, and outputs the vector resulting from the vector-matrix multiplication. The v vector is actually a primary input of the dual-memory module and it is stored in the memory of the operand stage (M_v). In contrast, the circulant matrix A is never stored nor received as input in its dense representation.

We note that the computational efficiency of the proposed dual-memory architecture sits on the possibility of substituting the time-consuming vector-matrix multiplication with a set of fast shift-rotate additions due to the fact that the A matrix, i.e., the H_i blocks of the H matrix in QC-MDPC codes, is both circulant and sparse. In particular, it is sufficient to store the positions of the ones in the first column of A.

To this end, each T value in input to the dual-memory module represents the position of a one in the first column of the A matrix. For each T value, the dual-memory module performs a shift-rotate of the v vector by T positions before adding the result to the accumulator by means of the compute stage ($calc$). We note that the generic $\circ$ operation performed by the compute stage $calc$, can be specialized depending on the actually required operation.

For example, Equation 4.1 shows the vector-matrix multiplication between a 4-bit row-vector (b) and a 4×4, binary, circulant matrix (C).

$$
\begin{aligned}
r = b \cdot C = \\
&= \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \\
&= \begin{bmatrix} b_2 + b_3 & b_0 + b_3 & b_0 + b_1 & b_1 + b_2 \end{bmatrix}
\end{aligned} \tag{4.1}
$$

In particular, the sparse representation of the C matrix, i.e., C^{sp}, that is made of the positions of the 1s in its leftmost column, is defined in Equation 4.2.

$$C^{sp} = \begin{bmatrix} 2 & 3 \end{bmatrix} \tag{4.2}$$

To this end, the vector-matrix multiplication between b and C^{sp} can be computed as the sum of the dense vector shift-rotated to the left by amounts equal to the elements of C^{sp}. This is shown in Equation 4.3, where b is the dense vector and the sparse-represented positions of the 1s in C are identified as C_i^{sp}. The $x <<< y$ notation specifies a left shift-rotate of vector x by y positions.

$$\begin{aligned} r = b \cdot C &= \sum_{i=0}^{v-1} (b <<< C_i^{sp}) = \\ &= (b <<< 2) + (b <<< 3) = \\ &= \begin{bmatrix} b_2 & b_3 & b_0 & b_1 \end{bmatrix} + \begin{bmatrix} b_3 & b_0 & b_1 & b_2 \end{bmatrix} = \\ &= \begin{bmatrix} b_2 + b_3 & b_3 + b_0 & b_0 + b_1 & b_1 + b_2 \end{bmatrix} \end{aligned} \tag{4.3}$$

From the computational viewpoint, the dual-memory module updates the accumulator's memory with a sequence of five steps for each T value in input. At time $T1$ a new T position is received by the `vOp` module that performs the readout from the M_v memory. We note that the T position can be misaligned with respect to the BW-bit size of each line in the M_v memory, thus the `AlignBuf` in the `vOp` module is used to store the trail of the first readout line at time $T2$, e.g., AAA in Figure 4.4. After the first clock cycle used to align the reads from the M_v memory, the `vOp` module produces BW bits of data for the successive sets of clock cycles required to completely readout the v vector. Each line produced by the `vOp` module is obtained by concatenating the content of the `AlignBuf` with the initial part of the next readout line from M_v. In particular, each readout line from M_v has the first part used to compose the BW-bit output, while the remaining part is stored in the `AlignBuf` buffer to be concatenated in the next clock cycle. Considering the scenario depicted in Figure 4.4, the `vOp` module outputs the first BW bits, i.e., $AAABBBBB$, at time $T3$. At the same time, the M_{res} memory is completely read starting from line 0. In particular, the content of each line q of M_{res} is combined with the output from the `vOp` module at time $T4$, before being stored at the same q-th line of M_res at time $T5$.

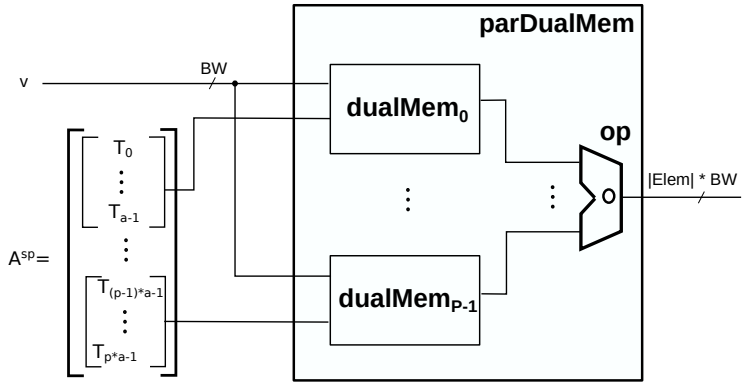


Figure 4.5: *Detailed view of the proposed parallel dual-memory architecture.*

Divide-and-conquer approach

Figure 4.5 shows the parallel architecture (`parDualMem`) as composed of a parametric number of dual-memory modules that are combined to perform the vector-matrix multiplication efficiently. The `parDualMem` module takes the vector v and the sparse representation of the binary circulant matrix A (A^{sp}) in input and produces the vector-matrix product in output. Depending on the actual operator implemented in place of the generic $\circ$ one, the size of each element of the output can vary. To this end, the bandwidth of the `parDualMem` module allows to output BW elements of the result at once, while $|Elem|$ identifies the size of the generic element of the result vector.

Starting from the sparse representation of the binary circulant matrix A, each dual-memory module receives a subset of positions and, for each of them, it accumulates the shift-rotate of the v vector. The final result is obtained by combining the outputs from all the implemented dual-memory modules operated by the op computing block (see Figure 4.5). We note that the `parDualMem` architecture allows a design-time configurable parallelism ranging from 1 to the number of ones in the first column of the A matrix.

Parallel architecture

Within the proposed decoder, the `parDualMem` architecture is employed to efficiently perform the UPC computation in the `calcUpc` module (see $UPC_i = s \cdot H_i$ in Figure 4.3) and the generation of the syndrome bit-flip vectors in the `calcBf` module (see $sBf_i = eBf_i \cdot H_i^T$ in Figure 4.3). In particular, the UPC computation, i.e., $upc_i = s \cdot H_i$, corresponds to a

vector-matrix multiplication in the integer domain. Thus, the configurable $calc$ stage in the `dualMem` module is customized to perform the integer addition. Differently, the syndrome bit-flip computation, i.e., $s_i^{bf} = e_i^{bf} \odot H_i^T$, corresponds to a vector-matrix multiplication computed in the binary domain, or equivalently, since circulant matrices are isomorphic to polynomials modulo $x^p - 1$, to a binary polynomial (or carry-less) multiplication. To this end, the configurable $calc$ stage in the `dualMem` module is customized to implement the bitwise XOR operator.

Notably, each instance of the `parDualMem` module allows to independently configure the level of parallelism between 1 and v, i.e., the number of ones in each column of the parity-check matrix, thus providing a flexible decoding architecture that can exploit different performance-area trade-offs.

4.2.2 Complexity analysis

This section discusses the complexity analysis of the proposed bit-flipping decoding architecture in terms of both time and space. The goal is to highlight the architectural optimizations that allow to implement a family of decoders for large QC-MDPC codes across a wide range of resource-performance trade-offs.

Time complexity

Equation (4.4) is a 6-parameter equation that defines the time required to perform a complete decoding procedure (T_{dec}), expressed in terms of number of clock cycles. The parameter $iter_{max}$ represents the maximum number of decoding iterations, p is the number of syndrome bits, n_0 is the number of circulant blocks that compose the parity-check matrix H, v is the weight of each column of the H matrix, BW is the bandwidth of the decoder datapath in bits, and Par_{Dec} is the parallelism in the UPC and syndrome bit-flips computation. Note that we do not have control over the $iter_{max}$, n_0, p and v parameters, since they are parameters of the QC-MDPC code. In contrast, the purpose of the proposed architecture is to provide an efficient and scalable hardware decoder to support the implementation of any QC-MDPC code-based cryptosystem.

$$T_{dec} = iter_{max} \cdot (n_0 + 1) \cdot \left\lceil \frac{p}{BW} \right\rceil \cdot \left\lceil \frac{v}{Par_{Dec}} \right\rceil \tag{4.4}$$

More in detail, $iter_{max}$ is a parameter of the decoding algorithm, p, n_0, and v are parameters of the considered QC-MDPC code, while BW and par are configurable parameters of the proposed decoding architecture that can be

Table 4.2: *Temporal evolution of the pipelined execution of one iteration of the decoding procedure, when the parity-check matrix H is composed of three blocks ($n_0 = 3$).*

Time epoch	1	2	3	4
$\mathbf{H_0}$	calcUpc	calcBf		
$\mathbf{H_1}$		calcUpc	calcBf	
$\mathbf{H_2}$			calcUpc	calcBf

tuned to explore different resource-performance trade-offs.
Equation (4.4) is the product of four terms. Once the parameters of the code, i.e., p, n_o, and v, are set, the first term, i.e., $iter_{max}$, defines the maximum number of iterations in the bit-flipping decoding procedure to achieve the required Decoding Failure Rate (DFR). The second term, i.e., $(n_0 + 1)$, accounts for the `calcUpc` and `calcBf` operations across the entire set of blocks in the H matrix. In particular, the decoding architecture is optimized to perform such processing in a pipelined fashion by leveraging two computational aspects. First, the computation on each block of H is independent from all the others. Second, for each block of H, the computations within the `calcUpc` and `calcBf` can be performed independently. Table 4.2 shows the pipelined execution of the decoder to perform a single iteration when the underlying code features a parity-check matrix H made of three circulant blocks, i.e., the n_0 code parameter is equal to 3. Time is expressed in time epochs, i.e., 1, 2, 3 and 4, where the duration of each epoch depends on the computational time required by the slowest of the `calcUpc` and `calcBf` stages. The pipelined execution allows to reduce the computational time from $2 \cdot n_0$, if the `calcUpc` and `calcBf` stages were completely serialized, to $(n_0 + 1)$.

To optimize the performance of the pipelined architecture, the Par_{Dec} and BW parameters are set to the same values for both stages. The stages are thus balanced, i.e, they have the same execution time. The third term, i.e., $\lceil \frac{p}{BW} \rceil$, represents the number of memory lines to be read and written for each 1 position in the H matrix. As shown by Equation (4.4), the computational time decreases when the bandwidth BW increases. Last, the fourth term, i.e., $\lceil \frac{v}{Par_{Dec}} \rceil$, accounts for the parallel computation of the ones of the H matrix. Indeed, for each block in the H matrix, our decoding architecture allows to configure how many 1 positions of H are processed in parallel in the `calcUpc` and `calcBf` stages.

Space complexity

Equation (4.5) defines the memory requirement (M_{dec}) of the proposed bit-flipping decoding architecture, expressed as the cumulative memory required by the `calcUpc` and `calcBf` stages, i.e., $M_{calcUpc}$ and M_{calcBf}, respectively. The memory requirement is provided in terms of number of BRAM memories, that are the de-facto storage memory in the FPGA. We note that the flip-flops, that represent the other type of memory resource in FPGAs, are not accounted for in the rest of the analysis for two reasons. First, their storage capacity is only a tiny fraction of the capacity offered by BRAM memories. Second, flip-flops are usually employed to store partial, i.e., temporary, results within a computational stage, thus minimally affecting the memory space requirements.

$$\begin{aligned} M_{dec} &= M_{calcUpc} + M_{calcBf} = \\ &= \Big(M_H + Par_{Dec} \cdot (M_s + M_{upc})\Big) + \\ &+ \Big(M_H + Par_{Dec} \cdot (M_e^{bf} + M_s^{bf}) + M_e\Big) \end{aligned} \tag{4.5}$$

According to Equation (4.5), the `calcUpc` stage requires to store the H matrix (M_H) the syndrome (M_s), and the computed UPCs (M_{upc}). We note that the term $(M_s + M_{upc})$ defines the memory requirement of a single dual-memory component within the `calcUpc` stage. In particular, the architectural parameter Par_{Dec} is the multiplier to the cumulative memory requirement, that accounts for the possibility of implementing a parallel set of dual-memory blocks to compute the UPCs in the `calcUpc` stage.

In a similar manner, the `calcBf` stage requires to store the H matrix (M_H), the error bit-flips (M_e^{bf}), the syndrome bit-flips (M_s^{bf}) and the error vector (M_e). In particular, the term $(M_e^{bf} + M_s^{bf})$ defines the memory requirement of a single dual-memory component within the `calcBf` stage. As for the `calcUpc` stage, the architectural parameter Par_{Dec} represents the multiplier to the cumulative memory requirement, that accounts for the possibility of implementing a parallel set of dual-memory blocks to compute the bit-flips in the `calcBf` stage.

Given the bandwidth (BW_{BRAM}) and the size in bits (S_{BRAM}) of a single FPGA BRAM memory, Equations (4.6)- (4.9) define the detailed memory requirements to store each of these matrices and vectors. In particular, Equation (4.6) defines the number of BRAM memories required to store the syndrome (M_s), error bit-flips (M_e^{bf}) and syndrome bit-flips (M_s^{bf}) vectors.

We note that all of them share the same size of p bits.

$$M_s = M_e^{bf} = M_s^{bf} = \left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.6}$$

In a similar manner, Equation (4.7) defines the number of BRAM memories necessary to store the error vector (M_e).

$$M_e = n_0 \cdot \left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.7}$$

Last, the number of BRAMs required to store the UPCs (M_{upc}) and the positions of the ones in the H matrix (M_H) are defined by Equation (4.8) and Equation (4.9), respectively.

$$M_{upc} = \left\lceil \frac{p \cdot log(v)}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.8}$$

$$M_H = n_0 \cdot \left\lceil \frac{v \cdot log(p)}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.9}$$

We note that the term $\lceil \frac{BW}{BW_{BRAM}} \rceil$ is common to Equations (4.6)- (4.9), and it defines the integer number of BRAM memories as a function of the bandwidth parameter (BW). In particular, a BW value exceeding the available BRAM bandwidth imposes an integer increase in the number of BRAMs, regardless of the actual occupation in bits of the stored element. Given the code parameters, the space complexity highlights that the actual memory requirement to implement the decoder is a function of the two configurable architectural parameters, i.e., Par_{Dec} and BW, that allow to regulate the resource-performance trade-off.

Considering Equation (4.6) and Equation (4.7), the term $\lceil \frac{p}{BRAM_{size}} \rceil$ defines the number of BRAM elements as a function of the size of p with respect to the storage capacity of a single BRAM, i.e., $BRAM_{size}$. The additional n_0 multiplier in Equation (4.7) highlights that the size of the error vector is n_0 times bigger than the syndrome. Considering Equation (4.8), the term $p \cdot log(v)$ accounts for the need to store p UPCs, each of which is the sum of v syndrome bits. In a similar manner, the term $v \cdot log(p)$ in Equation (4.9) accounts for the need to store the v positions of the ones for a block of the H matrix, where each of the v positions requires $log(p)$ bits.

4.2.3 Modifications to implement Black-Gray-Flip decoding

The QC-MDPC bit-flipping decoding architecture detailed in the previous parts was the subject of few minor modifications in order to implement

the Black-Gray-Flip (BGF) decoding algorithm, which is employed in the BIKE cryptoscheme. The adoption of BGF decoding makes therefore our decoding hardware implementation fully compliant with the latest official NIST-submitted specification of BIKE [6].

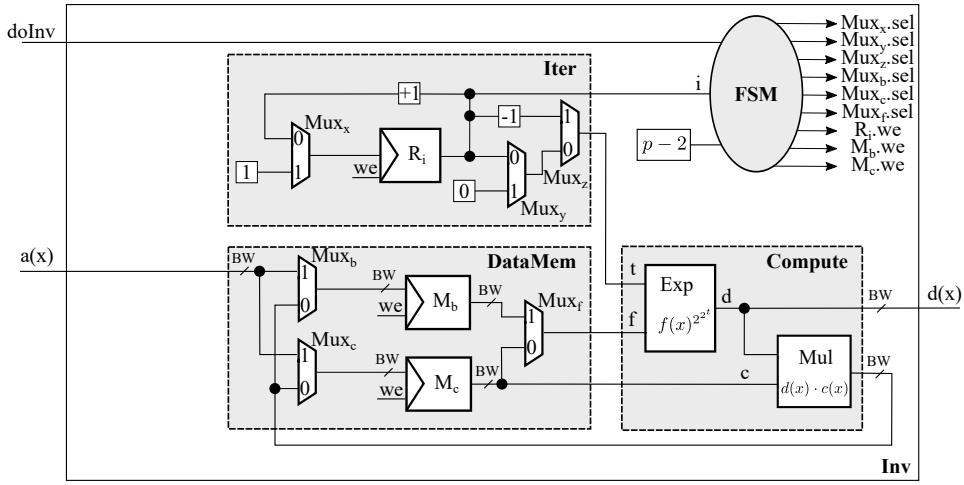
As detailed in the theoretical background discussion in Section 2.5.4, few changes must be applied to the generic QC-MDPC bit-flipping decoding architecture. The logic for the bit-flipping iterations remains the same as in the baseline algorithm, except for simple comparisons between the UPC values and two threshold values in the first decoding iteration, which allow obtaining the black and gray bitmasks. Such comparison components are already part of the original baseline architecture. The main addition to the updated BGF-compliant architecture consists in two n-bit BRAM memories to store the black and gray bitmasks, which are then used in two separate decoding iterations, i.e., the second and the third ones, by XORing them with the error bits corresponding to UPC values greater than or equal to a fixed threshold, which is equal to half the Hamming weight of a circulant block of the H matrix.

Overall, the aforementioned architectural changes do not significantly modify the previously detailed time and space complexity analysis. The execution time can be computed according to the same formula, accounting for the addition of the two black and gray decoding iterations within the $iter_{max}$ parameter, while the memory occupation must account only for additional memories storing the black and gray bitmasks, which correspond to a total of four p-bit memories.

4.3 Inversion architecture

The binary polynomial inversion architecture implements the inversion procedure, based on Fermat's inversion algorithm, detailed in Algorithm 4. Such procedure consists of the iterated computation of binary polynomial multiplications and exponentiations, which are performed by dedicated components whose architecture is detailed later in Section 4.4 and Section 4.5, respectively.

This section overviews the binary polynomial inversion component introduced in [43] discussing its architectural and algorithmic aspects and the optimized scheduling of the hardware operations that enables an efficient use of the multiplication and exponentiation subcomponents. Finally, it outlines the time and space complexity of the proposed inversion architecture as functions of both the architecture and code parameters.



(a) *Inversion architecture*

		FSM output control signals								
		Mux.sel						**M/R.we**		
1:	**function** $[d(x)]$ INVERSION$(a(x))$	***x***	***y***	***z***	***b***	***c***	***f***	***i***	***b***	***c***
2:	$b(x) = a(x);$	1	–	–	1	1	–	1	1	1
3:	$c(x) = a(x);$									
4:	**for** $i \in 1 : (\lceil \log_2 (p-2) \rceil - 1)$ **do**	–	–	–	–	–	–	0	0	0
5:	$d(x) = c(x)^{2^{2^i - 1}};$	–	–	1	–	–	0	0	0	0
6:	$c(x) = d(x) \cdot c(x);$	–	–	–	–	0	–	0	0	1
7:	**if** $(p-2)_2[i] == 1_2$ **then**	–	–	–	–	–	–	0	0	0
8:	$d(x) = b(x)^{2^{2^i}};$	–	0	0	–	–	1	0	0	0
9:	$b(x) = d(x) \cdot c(x);$	–	–	–	0	–	–	0	1	0
10:	**end if**	0	–	–	–	–	–	1	0	0
11:	**end for**	–	–	–	–	–	–	0	0	0
12:	$d(x) = b(x)^2;$	–	1	0	–	–	1	0	0	0
13:	**return** $d(x);$									
14:	**end function**									

(b) *FSM control signals associated to the inversion algorithm*

Figure 4.6: *Top-view architecture of the inversion module, composed of the computational datapath and of the finite state machine that drives the control signals according to the execution of the inversion algorithm (Algorithm 4).*

4.3.1 Architectural view

The architecture of the proposed inversion module (`Inv`) is shown in Figure 4.6a. The module takes as inputs the binary polynomial $a(x)$ to invert and the control signal `doInv` that starts the computation, and outputs the binary polynomial $d(x)$ that is the multiplicative inverse of $a(x)$. The proposed architecture, and in particular its FSM logic, is built upon the inversion algorithm described in Algorithm 4, as shown in Figure 4.6b.

The `Inv` module consists of four submodules, i.e., `Compute`, `DataMem`, `Iter`, and `FSM`. The computational unit (`Compute`) implements the optimized architectures to perform the binary polynomial exponentiation (`Exp`) and multiplication (`Mul`). The memory module (`DataMem`) is meant to efficiently store the input polynomial as well as the intermediate results of the computation. The iteration module (`Iter`) produces the values of the iterator i according to the implemented inversion algorithm (see Figure 4.6b). Finally, the finite state machine controller (`FSM`) generates the control signals that drive the multiplexers of the datapath and the write enable signals of the registers and memories, depending on the values of the iterator `i`, the code parameter p, and the `doInv` input.

4.3.2 Algorithmic view

The proposed architecture is built upon the inversion procedure described in Algorithm 4.6b.

The input phase starts when the `doInv` input signal is set to 1, storing the binary polynomial `a(x)` received as an input to the `Inv` module in the two memories of the `DataMem` submodule, i.e., $M_{b(x)}$ and $M_{c(x)}$. Such hardware phase corresponds to the execution of the lines 2-3 in Figure 4.6b.

At the end of the input phase, the `Inv` module starts computing the polynomial inverse by iteratively executing the hardware operations corresponding to the instructions at lines 4-11 in Figure 4.6b. The FSM selectively asserts the selectors of the multiplexers and the write-enable, i.e., `we`, control signals to ensure the correct execution of the inversion procedure. By observing that the value of p is a fixed parameter of the cryptosystem, we note that the FSM only requires the value of the i counter at each iteration to correctly generate the values of the control signals, thus mimicking the execution of the control instructions, i.e., the *for* loop and the *if* conditional statement at lines 4 and 7 of the inversion algorithm. Figure 4.6b highlights the values of the control signals within the proposed architecture during the hardware execution of the inversion algorithm, where the "$-$" symbol identifies *don't care* values.

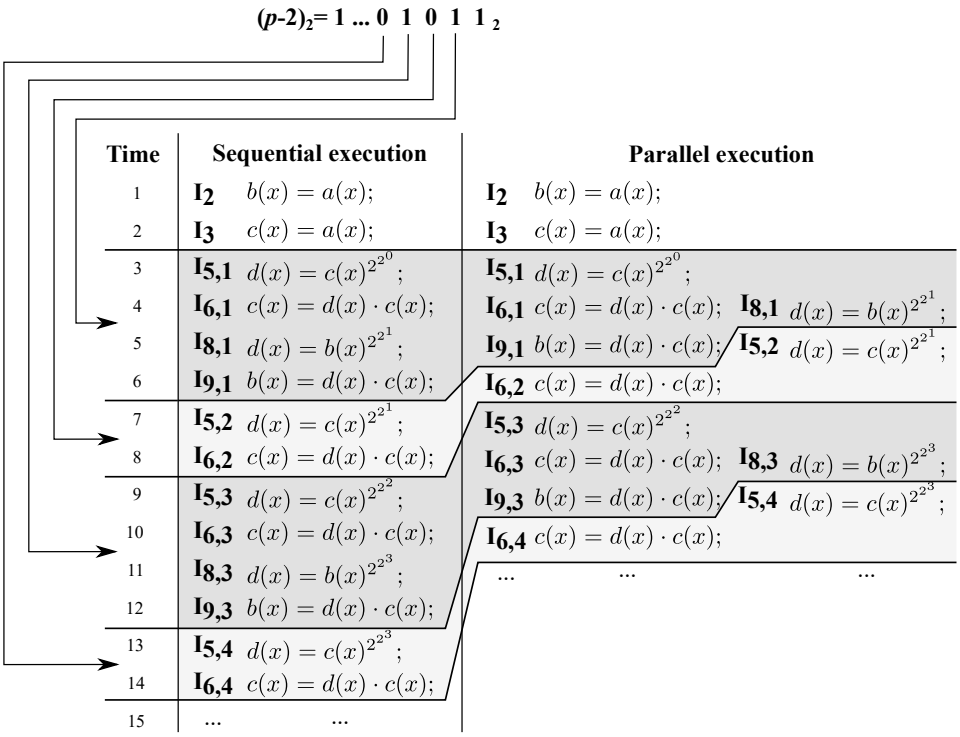


Figure 4.7: *Temporal evolution of the sequential and optimized executions of the inversion algorithm for* $(p-2) = 459_{10} = 111001011_2$. $I_{x(,y)}$ *represents the* x*-th instruction of the inversion algorithm at the* y*-th iteration, where* $x \in \{1 \dots 14\}$ *and* $y \in \{1 \dots 4\}$.

Once all the iterations have been executed, the FSM forces the final squaring of the $c(x)$ polynomial (see line 12) and subsequently outputs the obtained result $d(x) = a(x)^{-1}$ (see line 13).

4.3.3 Optimized hardware scheduling

The proposed inversion architecture is designed to schedule the exponentiations and multiplications to always use the `Exp` and `Mul` modules concurrently whenever possible, thus maximizing performance without duplicating the instances of the computational resources. Starting from the analysis of the inversion algorithm in Figure 4.6b, we identified two pairs of instructions for which the computation can be optimized by means of a concurrent execution, since each pair of instructions shows no data dependence. Considering the i-th iteration of the inversion algorithm (see lines 4-11 Figure 4.6b), the multiplication and the exponentiation instructions at line 6 and 8, respectively, can be concurrently executed on two separate functional units. In a

similar manner, the instructions at line 9 of the i-th iteration and at line 5 of the $(i+1)$-th iteration can also be computed at the same time. We note that the concurrent execution of the two pairs of instructions is constrained to the validity of the condition at line 7 of the inversion procedure in Figure 4.6b, i.e., $(p-2)_2[i] == 1$.

To demonstrate the effectiveness of the implemented hardware scheduling, Figure 4.7 shows an example of the execution of the first four iterations of the inversion algorithm, i.e., $i \in \{1, 2, 3, 4\}$, considering $(p-2) = 459_{10} = 111001011_2$. To better highlight the execution speedup due to the proposed optimized hardware scheduling, Figure 4.7 unrolls the considered *for* loop iterations. In particular, $\mathrm{I}_{x(,y)}$ identifies the instruction at line x of the inversion procedure that is executed during the y-th iteration of the *for* loop. The execution of the inversion algorithm takes advantage of the optimized hardware scheduling for each i-th iteration of the *for* loop such that $(p-2)[i]$ is equal to 1, since the validity of the condition at line 7 (see Figure 4.6b) allows the concurrent execution of the two identified pairs of multiplication-exponentiation instructions. Considering the example in Figure 4.7, the optimized hardware scheduling and the non-optimized sequential scheduling execute the four considered iterations in 10 and 14 time units, respectively.

The performance speedup of the proposed hardware scheduling is due to the concurrent executions at iterations 1, i.e., $I_{6,1}$-$I_{8,1}$ and $I_{9,1}$-$I_{5,2}$, and 3, i.e., $I_{6,3}$-$I_{8,3}$ and $I_{9,3}$-$I_{5,4}$, respectively (see timesteps 4, 5, 8, and 9 in Figure 4.7).

It is important to note that the actual performance speedup due to the optimized hardware scheduling is a function of the number of ones in the binary encoding of $(p-2)$ (see line 7 in Figure 4.6b), where p is a parameter of the cryptosystem. However, the selection of the value of p is subject to a set of contrasting requirements to balance the decode failure rate, the performance, and the security of the cryptosystem, thus preventing a choice of its value that only favors the performance of inversion as also highlighted in [6, 12].

4.3.4 Complexity analysis

This section discusses the time and space complexity of the proposed inversion architecture, highlighting the design choices that allow its implementation across a wide range of resource-performance trade-offs.

Time complexity

The time complexity of the inversion procedure (T_{inv}) can be expressed as a function of only the polynomial length p and the execution times of the exponentiation (T_{exp}) and multiplication (T_{mul}). Without considering the proposed scheduling optimization, the inversion procedure requires one exponentiation and one multiplication at each iteration of the *for* loop, and, in addition, one more exponentiation and one more multiplication at each i-th iteration corresponding to an i-th bit of $(p-2)$ that is equal to 1. The number of executed iterations is equal to $\lceil \log_2(p-2) - 1 \rceil$. In addition, one final exponentiation at the power of 2 is performed.

The proposed scheduling optimization reduces the number of operations that are required in the i-th iterations for which $(p-2)_2[i]$ is equal to 1. In such case, an iteration requires two times the execution time of the operation taking the longest between exponentiation and multiplication, instead of the execution time of two exponentiation and two multiplications. The resulting time complexity can therefore be expressed in clock cycles as in Equation 4.10.

$$\begin{aligned} T_{inv} = & ((2 \cdot (hw(p-2) - 1)) - 1) \cdot \max\{T_{exp}, T_{mul}\} \\ & + (zeros(p-2) + 1) \cdot (T_{exp} + T_{mul}) \\ & + T_{exp} \end{aligned} \tag{4.10}$$

Notably, $hw(p-2)$ corresponds to the number of bits of $(p-2)$ set to 1, while $zeros(p-2)$ corresponds to the number of zeros of the binary representation of $(p-2)$, that is equal to $(\lceil \log_2(p-2) \rceil - hw(p-2))$.

Space complexity

The area occupied by the binary polynomial inversion architecture can be expressed as the sum of the resources employed by the `Mul` and `Exp` components, plus two memories storing the p-bit binary polynomials $b(x)$ and $c(x)$ that are employed throughout the inversion procedure. Equation (4.11) defines the number of BRAMs of the inversion module (M_{inv}).

$$M_{inv} = M_{exp} + M_{mul} + 2 \cdot \left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.11}$$

It has four parameters. Other than p and BW, S_{BRAM} represents the size of a BRAM, that may be either 16Kb or 32Kb in Artix-7 FPGAs, while BW_{BRAM} represents the data bandwidth of a BRAM, that may be either 32 bits for 16Kb memories or 64 bits for 32Kb memories.

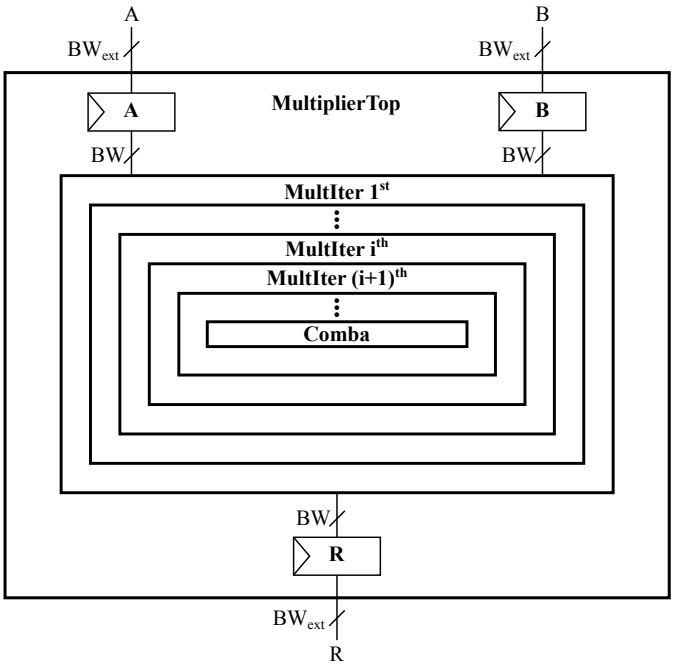


Figure 4.8: *Top view of the dense-dense binary polynomial multiplication achitecture.*

Equation (4.11) is the sum of three terms. M_{exp} and M_{mul} refer to the space complexity of the exponentiation and multiplication modules, while the third term corresponds to the two p-bit memories that store temporary variables employed by the inversion algorithm.

The first factor of the latter, i.e., 2, represents the number of p-bit memories. The second factor $\lceil \frac{p}{S_{BRAM}} \rceil$ accounts for the number of BRAM memories required to store a p-bit polynomial. The third factor $\lceil \frac{BW}{BW_{BRAM}} \rceil$ accounts for the number of BRAM memories necessary to provide the required BW data bandwidth.

The complexity of the exponentiation and multiplication components are instead discussed in detail in the following of this chapter, respectively in Section 4.5.3 and Section 4.4.3.

4.4 Dense-dense multiplication architecture

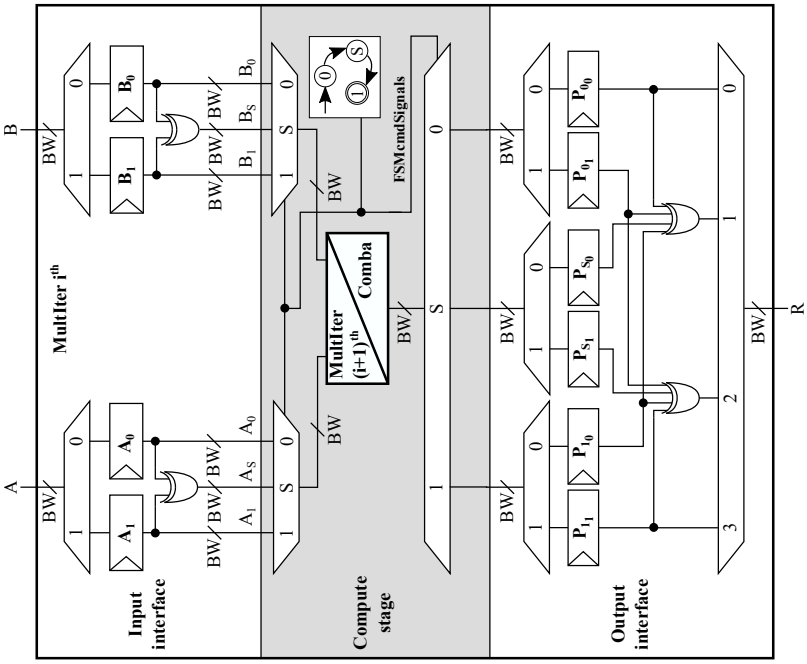
The binary polynomial multiplication architecture introduced in [111] combines, in a recursive fashion, the usage of the Karatsuba, Comba, and schoolbook multiplication algorithms discussed in Section 2.4.2. Figure 4.8 depicts the architectural top view of the proposed `MultiplierTop` multi-

plier component, which receives as inputs two binary polynomial operands A and B and outputs their product R. Both the operands and the result of their multiplication are represented in a dense way, i.e., as p-bit strings corresponding to $GF(2^m)$ polynomials, thus the multiplication operation is also referred to as dense-dense multiplication.

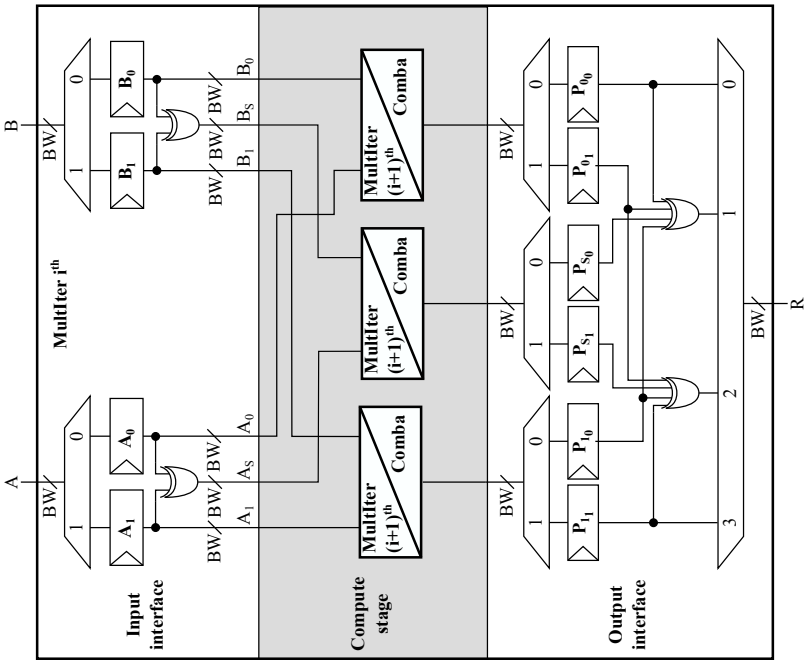
To ease the integration of the proposed component in real designs, the input and output interfaces offer a configurable bandwidth, BW_{ext}, as well as input and output memory layers to store the inputs and the produced output, respectively. Such design completely decouples the bandwidth of the internal multiplier datapath (BW) from the available external bandwidth (BW_{ext}). In particular, the former has no externally imposed constraints, while the latter can be constrained by the pin count or the data channel width of the system-on-chip that integrates the multiplier. The input and output memory layers are crucial components to operate on large polynomials, since no physical interface can accommodate a datapath width of dozens of thousands of bits.

The architecture of the `MultiplierTop` module allows to implement a configurable number of iterations of the Karatsuba algorithm, as depicted by the nested `MultIter` blocks in Figure 4.8, thus aggressively reducing the number of required partial products. At the end of the recursive application of the Karatsuba algorithm, the Comba multiplication algorithm performs the actual computation of the partial products (see `Comba` in Figure 4.8). We note that the use of the Comba multiplication algorithm at the end of the Karatsuba iterations allows to optimally schedule the computation of each partial product, also considering that the size of the operands after the recursive application of the Karatsuba iterations is still too large to fit into the combinational $BW \times BW$ multiplier, which performs the carry-less multiplication between two BW-bit digits.

The rest of this section is organized in three parts. Section 4.4.1 details the architecture that allows recursively applying the Karatsuba algorithm for a predefined number of times. Such a structure is meant to minimize the number of required partial products and maximize the parallelism level to compute the remaining partial products. Section 4.4.2 discusses the architecture to actually compute the partial products. Depending on the required performance-resources trade-off, such configurable computing architecture can implement either a single Comba multiplier, which computes the partial products in a serial way, i.e., one at a time, or a set of parallel Comba multipliers, which compute multiple partial products simultaneously. Finally, Section 4.4.3 discusses the time and space complexity of the multiplication component as functions of both the architecture and code parameters.



(a) *Serial architecture of nested i^{th} and $(i+1)^{th}$ Karatsuba iterations*



(b) *Parallel architecture of nested i^{th} and $(i+1)^{th}$ Karatsuba iterations*

Figure 4.9: *Architecture of the proposed Karatsuba multiplier, implementing a configurable number of nested Karatsuba algorithm iterations. A_i, B_i and P_{j_k} are BW-bit bandwidth memories.*

4.4.1 Karatsuba multiplier architecture

The proposed architecture is based on a hybrid approach which leverages the recursive application of the Karatsuba algorithm, to minimize the number of partial products, and of the Comba algorithm, used as the leaf node of the recursion, to optimally schedule the operations to compute each partial product. Such design approach allows to separately optimize the modules implementing the Karatsuba and Comba algorithms.

Figure 4.9 depicts the architecture of two nested Karatsuba iterations, i^{th} and $(i+1)^{th}$, which is at the core of the iterative application of the Karatsuba algorithm. In particular, the inner Karatsuba iteration can implement either the serial (see Figure 4.9a) or parallel (see Figure 4.9b) computation of the three partial products, thus allowing an additional level of flexibility to trade the performance with the resource utilization. In the serial case, the three partial products are computed by the same component in a sequential way. On the contrary, in the parallel case, each partial product is assigned to a dedicated multiplication component.

Regardless of its serial or parallel implementation, each Karatsuba iteration (`MultIter`) receives two polynomials in input and it outputs the result of their carry-less multiplication. The input interface splits each one of the two polynomials in two halves, according to the Karatsuba algorithm. Each half of each polynomial, i.e., A_1, A_0, B_1 and B_0, is stored in a separate memory element. In a similar manner, the output interface delivers the final multiplication result by composing the computed partial products according to the Karatsuba algorithm. We note that the proposed multiplier is parametric with respect to the implemented channel width, i.e., BW, that is used as an additional configuration option to trade performance with resource utilization.

The compute stage receives the operands from the input interface and delivers the computed partial products to the output interface. The compute stage implements the logic to perform the computation of the three partial products required by the current Karatsuba iteration. We note that, instead of directly computing the three partial products by means of either one (serial) or three (parallel) Comba multipliers (see `Comba` in Figure 4.9a and Figure 4.9b), a nested application of the Karatsuba algorithm can be performed. In this scenario, the `MultIter` block represents the key element to implement the recursive application of the Karatsuba algorithm. In contrast, the Comba module represents the leaf node at the end of the recursive application of the Karatsuba algorithm.

From the architectural viewpoint, the use of either a parallel or serial

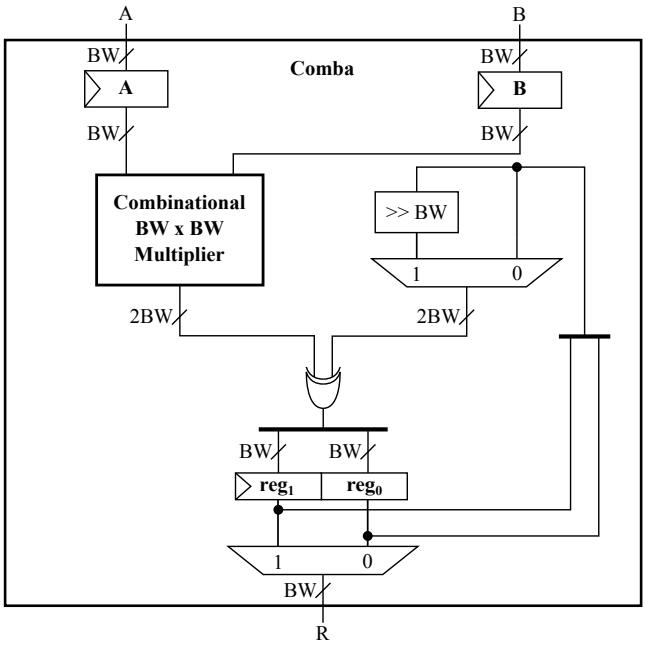


Figure 4.10: *Architecture of the proposed Comba multiplier. A and B are memories with a BW-bit bandwidth, reg_1 and reg_0 are BW-bit registers.*

implementation of the compute stage represents a configuration parameter of the proposed dense-dense multiplier. The parallel implementation of the compute stage only requires a proper connection of the input and the output signals to the nested `MultIter/Comba` modules (see Figure 4.9b). The serial implementation of the compute stage must orchestrate the computation of the three partial products by leveraging the single, i.e., shared, computing block (`MultIter/Comba`) (see Figure 4.9a). To this purpose, a simple finite-state-machine drives the multiplexing infrastructures to forward the correct part of the operands from the storage elements of the i^{th} `MultIter` module to the single compute unit, i.e., $(i+1)^{th}$ `MultIter/Comba`.

In summary, the proposed dense-dense multiplier architecture allows to flexibly configure *i)* the number of Karatsuba iterations to be implemented, *ii)* either the parallel or the serial computation for each of them, and *iii)* the internal channel width `BW`. The `MultIter` module implements an iteration of the Karatsuba algorithm, also offering the possibility to iterate the procedure by nesting parallel or serial instances of the same module.

Algorithm 8 Bit-level combinational multiplication. A and B are BW-bit digits, R is $(2BW-1)$-bit long. $A[i]$, $B[i]$ and $R[i]$ indicate single bits.

```
1: function [R] CombinationalMul(A, B)
2:     for i ∈ 0 : BW − 1 do
3:         for j ∈ 0 : BW − 1 do
4:             R[i + j] = R[i + j] ⊕ (A[i] · B[j]);
5:         end for
6:     end for
7: end function
```

4.4.2 Comba multiplier architecture

The Comba multiplier (see `Comba` in Figure 4.9a and Figure 4.9b) is tasked with the computation of each partial product in the innermost Karatsuba multiplication modules. To this end, the *Comba Multiplier* module represents the terminal block, i.e., the leaf node, of the recursive application of the Karatsuba algorithm.

Figure 4.10 depicts the architecture of the `Comba` module, which performs the multiplication of the input operands according to the schedule of the Comba algorithm [30]. We note that the iterative application of the Karatsuba algorithm minimizes the number of required partial products, while also halving the size of the operands at each iteration. However, the size of the operands in input to the Comba multiplier module is still in the order of thousands of bits, thus far too large to perform a single combinational multiplication. In contrast, the Comba multiplier assumes that each operand is made of a set of BW-bit digits and performs the multiplication, according to the Comba algorithm, in a digit-by-digit processing fashion. At the core of the `Comba` module, the `Combinational BW x BW Multiplier` performs the multiplication between two digits (see Figure 4.10). In particular, Algorithm 8 details the steps to perform the bit-level combinational multiplication of the two BW-bit digits according to the schoolbook multiplication algorithm.

The Comba multiplier schedules the $BW \times BW$ multiplications according to the strategy proposed by Comba, i.e., producing a single BW-bit digit of the result at a time, by computing all the partial products contributing to it. This approach minimizes the number of bits required to maintain in memory the sum of the partial products. To implement this strategy, two BW-bit registers, reg_1 and reg_0, are employed to store the sum of all the contributions to the said portion of the result. reg_1 and reg_0 store respectively the BW most and least significant bits of the XOR of partial products computed by the combinational multiplier. When the computation of the

sum is completed, the least significant BW bits, i.e., the BW bits stored in reg_0, are committed to the output of the Comba Multiplier, while the most significant ones, i.e., the BW bits stored in reg_1, are copied over in reg_0.

4.4.3 Complexity analysis

This section discusses the time and space complexity of the proposed dense-dense multiplication architecture, highlighting the design choices that allow its implementation across a wide range of resource-performance trade-offs. For the sake of simplicity, the complexity discussion only considers the case in which each Karatsuba multiplier module computes its three partial products by means of either three Karatsuba or Comba multipliers in a parallel fashion, as depicted in Figure 4.9b.

Time complexity

Let Par_{DDMul} be the parallelism parameter that expresses how many times the Karatsuba recursion formula is applied, and BW be the bandwidth of the multiplier datapath, then its time complexity can be expressed as in Equation 4.12.

$$T_{mul} = \left(\sum_{i=0}^{Par_{DDMul}} \frac{2}{2^i} \right) \cdot \left\lceil \frac{p}{BW} \right\rceil + \left\lceil \frac{\left\lceil \frac{p}{2^{Par_{DDMul}}} \right\rceil}{BW} \right\rceil^2 \tag{4.12}$$

The first term refers to the data movement between the different layers of Karatsuba recursion, while the second term refers to the execution time required by the $2^{Par_{DDMul}}$ innermost Comba multipliers, each concurrently computing one partial product of the Karatsuba formula.

Space complexity

The area occupied by the dense-dense binary polynomial multiplication architecture can be expressed as shown in Equation (4.13), which defines the number of BRAMs occupied by the multiplication module (M_{mul}).

$$M_{mul} = \left(3^{Par_{DDMul}} \cdot 6 - 4\right) \cdot \left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.13}$$

It has five parameters. Other than Par_{DDMul}, p, and BW, S_{BRAM} represents the size of a BRAM, that may be either 16Kb or 32Kb in Artix-7 FPGAs, while BW_{BRAM} represents the data bandwidth of a BRAM, that may be either 32 bits for 16Kb memories or 64 bits for 32Kb memories.

$f(x) = x^{10} + x^9 + x^3 + x^1 + x^0 = 11000001011_2$

$g(x) = f(x)^k = f(x)^4 = \mathbf{00010011011}$

Time	$f(x)$	$g_1(x)$	$g_0(x)$
0	11000001011	00000000000	00000000000
1	11000001011	00000010000	00000000001
2	11000001011	00000010010	00000000001
3	11000001011	00000010010	00000000001
4	11000001011	00000010010	00000000001
5	11000001011	00000011010	00000000001
6	11000001011	00000011010	00010000001
7	$g(x) = g_1(x) \oplus g_0(x) = \mathbf{00010011011}$		

Figure 4.11: *Example of parallelized exponentiation.*

Equation (4.13) is the product of three factors. The first one, i.e., $(3^{Par_{DDMul}} \cdot 6 - 4)$, represents the number of p-bit memories. The second factor $\lceil \frac{p}{S_{BRAM}} \rceil$ accounts for the number of BRAM memories required to store a p-bit polynomial. The third factor $\lceil \frac{BW}{BW_{BRAM}} \rceil$ accounts for the number of BRAM memories necessary to provide the required BW data bandwidth. Most notably, the Par_{DDMul} parallelism parameter has an exponential impact on the occupied memory resources.

4.5 Exponentiation architecture

The binary polynomial exponentiation is a critical operation within the inversion algorithm. The implementation of the exponentiation component must therefore be carefully designed to optimize the area-performance trade-off in order to enable the efficient computation of binary polynomial inversion.

Starting from the the exponentiation procedure detailed in Algorithm 5, the exponentiation architecture proposed in [43] leverages the possibility to independently compute each bit of the result polynomial $g(x)$ to deliver a parallel architecture that allows the concurrent computation of Par_{Exp} bits of $g(x)$. The parallel architecture is achieved by employing Par_{Exp} separate hardware memories. In particular, each memory manages the writing of one of the Par_{Exp} bits of $g(x)$. Once all p bits of $f(x)$ have been processed and written to the corresponding Par_{Exp} memories, their bit-wise XOR produces the final $g(x)$ polynomial.

To demonstrate the performance speedup due to the use of the proposed

parallel exponentiation architecture, Figure 4.11 details the computation of the $g(x)$ polynomial as the 4-th power of the f(x) polynomial using a parallelism of 2, i.e., $Par_{Exp} = 2$. Notably, aside from the parallel computation, the example in Figure 4.11 performs the computation previously discussed in Section 2.4.3 (see Figure 2.3). At timestep 0, $f(x)$ holds the input polynomial, while the Par_{Exp} $g_i(x)$ polynomials, $g_0(x)$ and $g_1(x)$, are set at 0. At each subsequent timestep, Par_{Exp} adjacent bits are read from the $f(x)$ polynomial, and each of them is written to the corresponding $g_i(x)$ polynomial. Blue and red colors to highlight the bits processed at each timestep as well as their positions in the $g_i(x)$ polynomials, where $i \in \{0, 1\}$. Once all p bits of the $f(x)$ polynomial have been read and written in the correct position of the Par_{Exp} $g_i(x)$ polynomials, the $g_i(x)$ polynomials are bit-wise XORed to produce the $g(x)$ result polynomial, which is the 4-th power of $f(x)$.

The rest of this section first overviews the binary polynomial exponentiation component introduced in [43] from the architectural and algorithmic point of views and then outlines the time and space complexity of the proposed exponentiation architecture as functions of both the architecture and code parameters.

4.5.1 Architectural view

The `Exp` module in Figure 4.12 represents our architecture for polynomial exponentiation. It has a `BW`-bit input `f` and an input `t`, corresponding to the base polynomial $f(x)$ and to the exponent 2^{2^t}, respectively, and a `BW`-bit output `g` that corresponds to the resulting polynomial $g(x) = f(x)^{2^{2^t}}$. `BW` is a design-time parameter that defines the datapath bandwidth of the exponentiation module.

The `Exp` module is designed as a two-stage architecture, composed of the `Stage1` and `Stage2` modules. They contain a memory, composed of FPGA BRAMs, that can hold p bits and has a `BW`-bit read/write data bandwidth, and they respectively store the $f(x)$ and $g_i(x)$ polynomials. The PAR_E design-time parameter defines the degree of parallelism within the exponentiation module, i.e., the number of `Stage2` replicas that are instantiated to parallelize the computation. To further improve the efficiency of the proposed architecture, two lookup tables `AddrIncr` and `AddrStart` are populated at compile time to provide the address increment and start values for $g_i(x)$ memories. `AddrIncr` contains $log_2(p-2)$ entries, indexed from 0 to $(log_2(p-2)-1)$, each containing the $(Par_{Exp} \cdot 2^{2^t}) \bmod p$ value, where t is the index of the entry. `AddrStart` contains $log_2(p-2)$ sets of

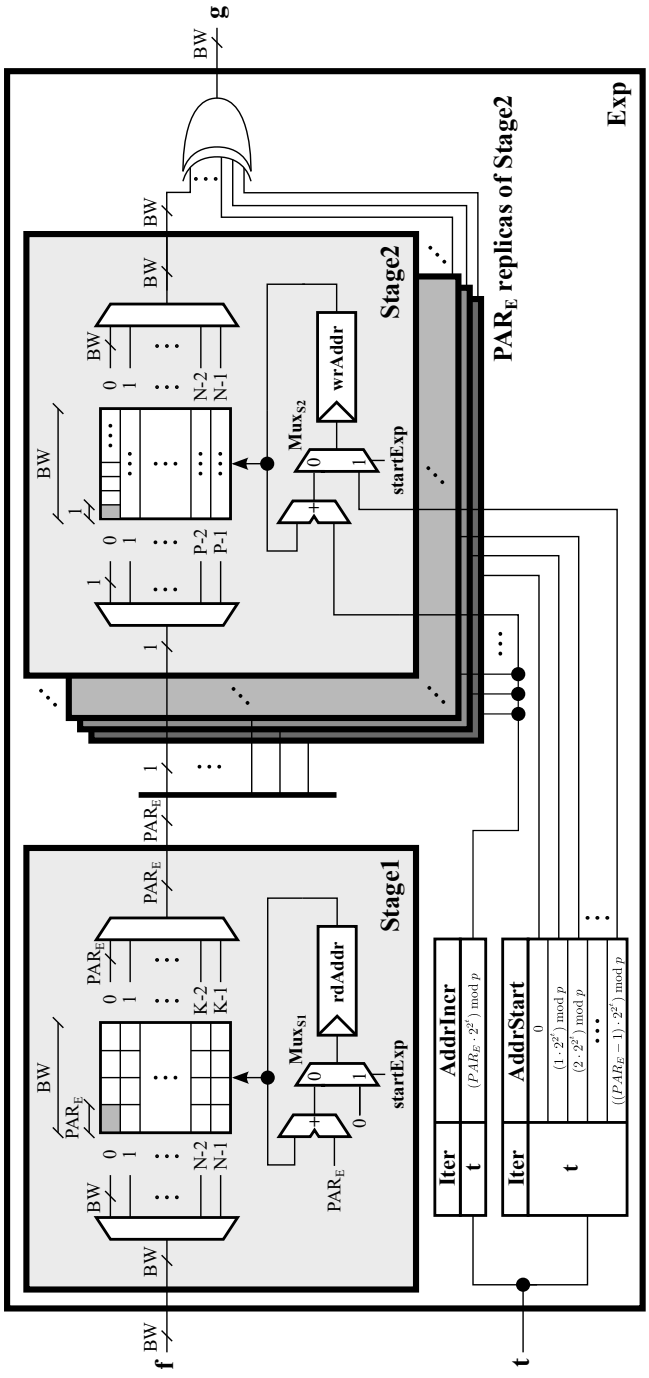


Figure 4.12: *Detailed view of the proposed exponentiation architecture. Legend:* $N = \left\lceil \frac{P}{BW} \right\rceil$, $K = \left\lceil \frac{P}{Par_{Exp}} \right\rceil$, $PAR_E = Par_{Exp}$.

entries, indexed from 0 to $(log_2(p-2)-1)$. Each set of entries contains Par_{Exp} values equal to $(s \cdot 2^{2^t}) \bmod p$ value, where s holds all integer values comprised between 0 and $(Par_{Exp}-1)$, referring to the corresponding $g_i(x)$ memory, and t is the index of the set of Par_{Exp} entries.

4.5.2 Algorithmic view

The execution of the exponentiation can be seen as organized in three phases, i.e., the *Input*, *Computation*, and *Output* ones. During the *Input* phase, the `Exp` module stores the p-bit $f(x)$ polynomial into the memory component of the `Stage1` module, passing `BW` bits per clock cycle through the `f` input, while the `Stage2` memory is reset to contain all 0 bits. At the same time, the t value fed through the `t` input is used to index the `AddrIncr` and the Par_{Exp} `AddrStart` values within the two respective lookup tables. The `Stage2` modules share the same `AddrIncr` value, while the `AddrStart` values are correctly dispatched to the instances of the `Stage2` module. Thereafter, the *Computation* phase takes place. At each clock cycle, Par_{Exp} bits are read and output from the memory of the `Stage1` module, from the least to the most significant bits of the p-bit $f(x)$ polynomial. These Par_{Exp} bits are split and each of them is fed as a single-bit signal to one of the replicas of the `Stage2` module. Each single-bit input to a `Stage2` module is written, one per clock cycle, into the `Stage2` memory at a position that starts from the `AddrStart` value and that is incremented (modulo p) at each clock cycle by the `AddrIncr` value. The *Computation* phase ends when all p bits read from the `Stage1` memory have been written to their corresponding positions in the Par_{Exp} `Stage2` memories. Finally, during the *Output* phase, the content of the `Stage2` memories is output, `BW` bits per clock cycle, and the Par_{Exp} `BW`-bit outputs are XORed. We note that p and 2^{2^t} are coprime, i.e., their GCD is 1, thus, it is guaranteed that there can not be any bits set to 1 in two or more different `Stage2` memories, i.e., we cannot have any cancellations due to the XOR operation. The result of the XOR operation corresponds to the actual $g(x)$ polynomial, which is output `BW` bits per clock cycle through the `g` port.

4.5.3 Complexity analysis

This section discusses the time and space complexity of the proposed exponentiation architecture, highlighting the design choices that allow its implementation across a wide range of resource-performance trade-offs.

Time complexity

Equation (4.14) defines the time required to execute an exponentiation (T_{exp}), expressed in terms of clock cycles.

$$T_{exp} = \left\lceil \frac{p}{BW} \right\rceil \cdot \left\lceil \frac{BW}{Par_{Exp}} \right\rceil \tag{4.14}$$

It has three parameters. p corresponds to the polynomial length. It is a parameter of the QC-MDPC code and, thus, it can not be controlled by the hardware designer. BW is the bandwidth of the exponentiation datapath expressed in bits and Par_{Exp} is the parallelism implemented in the exponentiation module. Both are configurable parameters of the proposed architecture and can be tuned to explore different area-performance trade-offs.

Equation (4.14) is the product of two terms. The first term $\lceil \frac{p}{BW} \rceil$ represents the number of memory lines to be read from the input polynomial and written into the output polynomial. The second term $\lceil \frac{BW}{Par_{Exp}} \rceil$ accounts for the parallel writing on separate BRAMs for the output polynomial. Equation (4.14) is fully independent from the input polynomial and depends instead exclusively on the p code parameter and on the BW and Par_{Exp} architectural parameters. Since the execution time of the multiplication module is also independent from its input values, and the same holds for the top inversion module, then our implementation guarantees constant-time execution of binary polynomial inversion.

Space complexity

Experimental results showed empirically that LUT and BRAM relative utilization of the available FPGA resources are similar to each other across all hardware instances on the whole Artix-7 family and for all polynomial lengths, with the LUT utilization being slightly larger than the BRAM one on average. At the same time, flip-flops are mostly unused in the proposed architecture. The number of BRAMs is therefore deemed a good metric for the space complexity of the exponentiation module.

Our architecture requires one p-bit memory for the `Stage1` module and one p-bit memory for the `Stage2` module. Due to the parameterized replication of `Stage2` modules, the overall exponentiation module requires $(Par_{Exp}+1)$ p-bit memories. Equation (4.15) defines the number of BRAMs of the exponentiation module (M_{exp}).

$$M_{exp} = (Par_{Exp} + 1) \cdot \left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \tag{4.15}$$

It has five parameters. Other than p, BW, and Par_{Exp}, S_{BRAM} represents the size of a BRAM, that may be either 16Kb or 32Kb in Artix-7 FPGAs, while BW_{BRAM} represents the data bandwidth of a BRAM, that may be either 32 bits for 16Kb memories or 64 bits for 32Kb memories.

Equation (4.15) is the product of three terms. The first term $(Par_{Exp} + 1)$ represents the number of p-bit memories. The second term $\lceil \frac{p}{S_{BRAM}} \rceil$ accounts for the number of BRAM memories required to store a p-bit polynomial. The third term $\lceil \frac{BW}{BW_{BRAM}} \rceil$ accounts for the number of BRAM memories necessary to provide the required BW data bandwidth.

4.6 Dense-sparse multiplication architecture

The dense-sparse multiplication architecture introduced in [13] implements a similar structure to the `parDualMem` parallel architecture employed in the *calcBf* stage of the QC-MDPC bit-flipping decoder described previously in Section 4.2.1. In particular, the underlying operations are performed in the binary polynomial arithmetic. As in the case of the decoder architecture, the configurable parallelism allows replicating the `dualMem` component to speed up the computation.

The proposed dense-sparse multiplication architecture is designed to outperform the dense-dense multiplication architecture in cases when one of the two operands has a low Hamming weight, and can thus be represented in a sparse form. Examples of such binary polynomials in the BIKE KEM are the n-bit error vector, with Hamming weight $t \approx \sqrt{n}$, and the p-bit binary polynomials corresponding to the two H_i blocks of the parity-check matrix, each with Hamming weight $v \approx \sqrt{n}$. Instances of multiplications involving sparse polynomials appear in all three BIKE KEM primitives.

4.6.1 Complexity analysis

This section discusses the time and space complexity of the proposed dense-sparse multiplication architecture as functions of both the architecture and code parameters, highlighting the design choices that allow its implementation across a wide range of resource-performance trade-offs.

Time complexity

Equation (4.14) defines the time required to execute a multiplication (T_{dsmul}), expressed in terms of clock cycles.

$$T_{dec} = \left\lceil \frac{p}{BW} \right\rceil \cdot \left\lceil \frac{HW}{Par_{DSMul}} \right\rceil \tag{4.16}$$

It has three parameters, namely the p polynomial length paramater of the QC-MDPC code, the HW Hamming weight of the sparse operand, the BW bandwidth of the dense-sparse multiplication datapath, and the Par_{DSMul} parallelism implemented in the dense-sparse multiplication module. The latter two are configurable parameters of the proposed architecture and can be tuned to explore different area-performance trade-offs.

Equation (4.16) is fully independent from the actual values of the input polynomials, thus our implementation of the dense-sparse multiplication provides constant-time execution of binary polynomial inversion.

Space complexity

The dense-sparse multiplication architecture occupies a number of BRAM memories (M_{dsmul}) as defined in Equation (4.17).

$$M_{dsmul} = \left(\left\lceil \frac{HW \cdot log(p)}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \right) + \\ + 2 \cdot Par_{DSMul} \cdot \left(\left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \right) \tag{4.17}$$

Equation (4.17) is the sum of two terms. The first one corresponds to the memory required to store the sparse binary polynomial operand, which has Hamming weight HW, while the second term of the addition refers to the two memories of each of the Par_{DSMul} parallel `dualMem` components.

4.7 Other components

The execution of the BIKE KEM primitives requires performing two more main operations, namely, SHA-3 cryptographic hash and pseudorandom number generation based on SHAKE. Notably, SHA-3 and SHAKE belong to the same family of cryptographic functions.

The two operations are computed by dedicated accelerators that are however not parametric and configurable. Their architecture makes use of on already publicly available accelerators for the Keccak sponge function, that is the core building block of both SHA-3 and SHAKE, applying the proper modifications to satisfy the SHA-3 NIST standard [35] and to implement the PRNG logic surrounding the SHAKE component. The implemented hardware components are designed to provide effective support for the two operations, even though they are not a critical target for optimization within the shope of this PhD thesis.

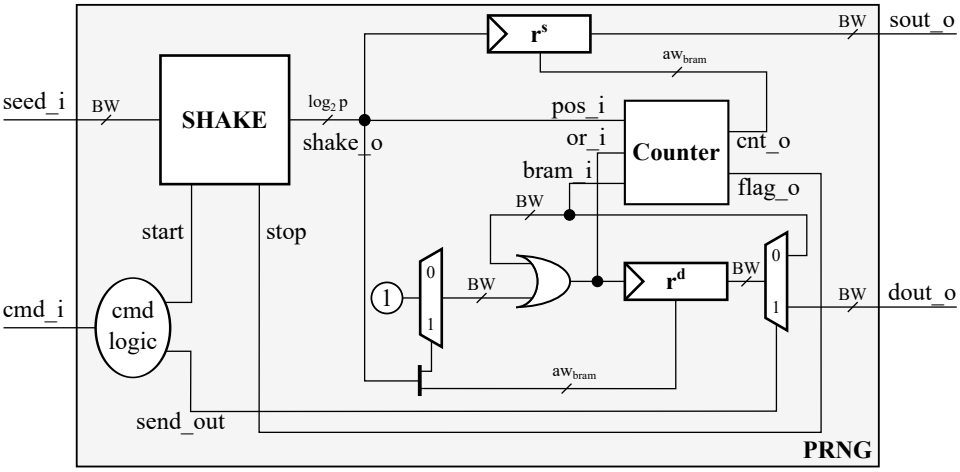


Figure 4.13: *Architecture of SHAKE-based PRNG.*

The rest of this section briefly discusses the architectures of the SHA-3 component and of the SHAKE-based uniform pseudorandom number generation one.

4.7.1 SHA-3 architecture

The `SHA-3` module implements the SHA3-384 cryptographic hash function [35]. It computes the 384-bit digest of the SHA3-384 cryptographic function of the input message according to an architecture similar to the high-speed core detailed in [19], which was modified to support the standard SHA-3 cryptographic hash functions in place of the original, pre-standard Keccak functions. The `SHA-3` module takes as input the input message msg padded according to the `01‖10*1` SHA-3 padding scheme, and outputs the 384-bit hash digest dig, that is computed by executing one Keccak-f round function per each clock cycle. The I/O operations are carried out through the BW-bit data input and output ports of the module.

4.7.2 Uniform pseudorandom number generation architecture

The `PRNG` module, which is depicted in Figure 4.13, performs the generation of a pseudorandom sequence of bits with fixed Hamming weight by making use of an internal SHAKE256 component (`SHAKE`), which implements an architecture that is similar to the one employed by the `SHA-3` module discussed in Section 4.7.1, albeit producing a variable-length output according to the needs of the surrounding pseudorandom generation logic. SHAKE256 is indeed an extendable output function, i.e., a function that outputs a digest

of any desired length, that is part of the SHA-3 family [35]. The digest output by the SHAKE256 component is broken up in $(\log_2 p)$-bit chunks, each possibly representing the position of a bit set to 1 within a p-bit vector. The extracted values are evaluated to discard the values which have already been previously extracted, thus avoiding cancellations and therefore enabling the generation of a vector with the desired Hamming weight. Moreover, values larger than or equal to p are also discarded, providing a uniform distribution of bits set to 1 within the random-generated bitvector. Indeed, operating on values larger than or equal to p to make them modulo-p values, e.g., through the modulo operator, would provide a biased distribution instead of a uniform one.

The `PRNG` is constant-time with respect to the generated bitvector, meaning that the execution time does not depend on the generated positions of 1s within the bitvector, but on the number of values rejected due to repetition or due to being greater than or equal to p. Such information would not be exploitable by an attacker to retrieve the generated bitvector, i.e., the private key and the error vector within the BIKE key generation and encapsulation primitives [89].

4.8 Design space exploration

In order to provide the best hardware support, the proposed client and server architectures leverage a set of state-of-the-art configurable accelerators for the most complex operations employed within the KEM primitives. However, such flexibility comes at the cost of a broad design space, which imposes the use of an efficient search strategy to minimize the exploration time.

Therefore, a four-step complexity-oriented heuristic proposed in [41] drives the design space exploration according to the time and space complexity of the most computationally intensive operations in the three KEM primitives. Notably, the overall computation time on the client side can be considered as the sum of the execution times of the key generation and decapsulation KEM primitives [6], while encapsulation represents the sole server-side functionality. Moreover, the configurable components employed to implement multiplication [13], inversion [43], and decoding [110] highlight block RAM (BRAM) as the scarcest resource thus their space complexity can be approximated as the sum of BRAMs used for key generation and decapsulation, on the client side, and encapsulation, on the server side.

Remarkably, the application of the heuristic to the proposed BIKE architecture makes use of the time- and space-complexity formulas expressed for

the configurable components, i.e., bit-flipping decoding, binary polynomial inversion, dense-dense binary polynomial multiplication, binary polynomial exponentiation, and dense-sparse binary polynomial multiplication. Such complexity formulas were previously discussed in Section 4.2.2, Section 4.3.4, Section 4.4.3, Section 4.5.3, and Section 4.6.1, respectively. Other possible quantitative or qualitative metrics, e.g., estimating energy or power consumption or providing a measure of information leakage against side-channel attacks, are not instead considered in the design space exploration.

The complexity-oriented heuristic is composed of the following steps.

Step 1 - Starting from the computational time of the AVX2 implementation of BIKE, the heuristic computes the fraction of time spent executing each primitive in the server and the client. Intel AVX2 data in Table 4.1 shows that the fraction of time for key generation and decapsulation is around 20% and 80%, respectively, on the client side, while the encapsulation represents the entire server time. Such ratios are used to assign the amount of resources devoted to each KEM primitive module in the client and server architectures.

Step 2 - For each KEM primitive module, the heuristic identifies the operations executed by parametric components that require the largest fraction of execution time. In particular, the heuristic considers the set of parametric operations for which the execution time is at least 90% of the total execution time of the primitive or the entire set of configurable components otherwise. For example, Table 4.1 shows that multiplication, which is the sole parametric operation in our hardware implementation of encapsulation, accounts for up to 23% of its execution time on Intel AVX2 platforms. In contrast, decoding, which is also computed in hardware by a configurable component, accounts for more than 90% of the execution time in the AVX2 implementation of decapsulation.

Step 3 - For each component or group of components, the heuristic explores time- and space-complexity formulas to identify the combination of parameters that allows maximizing performance within the assigned resource budget. The exhaustive search in the parameter space to find out the best parameter configurations for each module is very fast since it leverages the configurable components' time- and space-complexity formulas without involving any time-consuming hardware synthesis and place-and-route tasks.

Step 4 - The heuristic implements the client and server designs employing the configurations obtained at **Step 3**. Notably, our algorithm is robust and conservative to account for *i)* the non-predictability of the synthesis and implementation of EDA tools, and *ii)* the fact that a small change in the

parameters can severely affect the performance and resource utilization. Therefore, the heuristic could land to an unfeasible configuration or to a configuration for which not all the available resources can be used since small increments in the parameters would make it unfeasible within the resource budget. In the former case, the heuristic iteratively re-implements the failed design by lowering the values of the parameters for which the time-complexity formulas show the smallest performance degradation, and this process keeps on until the design becomes feasible. In the latter case, the heuristic iteratively re-implements the non-optimal design by increasing the values of the parameters for which the space-complexity formulas highlight the smallest resource utilization increase, until either the performance improvement is lower than a certain threshold or the design saturates the available hardware resources.

CHAPTER 5

Experimental results

This chapter discusses the experimental evaluation of the proposed architecture implementing the BIKE cryptoscheme. First, it details the performance of existing state-of-the-art software and hardware implementations of BIKE, which act as a benchmark for the performance of the proposed design. Then, it discusses the experimental results for the components implementing the most complex operations of BIKE, comparing their area and performance to state-of-the-art ones and evaluating how they scale by varying the configurable architectural and code parameters. Finally, it compares the area and performance of the proposed implementation of the whole BIKE cryptoscheme against software and hardware ones from literature.

Parts of this chapter are derived from previously published works co-authored by the author of this thesis. The experimental results for QC-MDPC bit-flipping decoding discussed in Section 5.4 were obtained from [110], while those for the dense-dense binary polynomial multiplication in Section 5.5 and the binary polynomial inversion in Section 5.7 were derived from [111] and [43], respectively. Finally, the experimental analysis of the area and performance of the whole KEM client and server nodes provided in Section 5.9 were adapted from the work in [41]. More details about the referenced publications are provided in Appendix A.

Table 5.1: *Breakdown of the execution times of BIKE, expressed in milliseconds, for different security levels, architectures, and software implementations. Legend: Kg key generation, En encapsulation, De decapsulation primitive, PRNG and* ***H*** *pseudorandom generation, Inv. inversion, Mult. multiplication, Dec. decoding, Other other operations executed in the KEM primitives,* ***K*** *and* ***L*** *SHA-3 hash function.*

		Target CPU, software version and NIST security level							
		ARM32		**ARM64**		**Intel**		**Intel**	
KEM		**C99 [5]**		**C99 [3]**		**C99 [3]**		**AVX2 [3]**	
prim.	**Op.**	**SL1**	**SL3**	**SL1**	**SL3**	**SL1**	**SL3**	**SL1**	**SL3**
Kg	PRNG	0.88	1.24	0.44	1.20	0.04	0.10	0.01	0.02
	Inv.	319.08	883.05	19.66	62.66	3.46	11.24	0.18	0.53
	Mult.	12.77	36.64	1.00	3.05	0.17	0.56	0.01	0.02
	Other	0.01	0.02	0.05	0.06	0.01	0.01	0.00	0.00
		332.74	920.95	21.15	66.97	3.68	11.91	0.20	0.57
En	**H** func.	0.86	1.24	0.79	2.24	0.07	0.18	0.02	0.03
	Mult.	12.66	37.21	1.00	3.05	0.18	0.55	0.01	0.02
	L func.	0.63	1.11	0.08	0.15	0.01	0.02	0.01	0.02
	K func.	0.17	0.33	0.04	0.08	0.01	0.01	0.01	0.01
	Other	0.50	1.05	0.08	0.08	0.00	0.01	0.00	0.00
		14.83	40.94	1.99	5.60	0.27	0.77	0.05	0.09
De	Dec.	463.15	1185.65	32.12	99.80	3.90	12.24	0.75	2.41
	L func.	0.63	1.12	0.08	0.15	0.01	0.02	0.01	0.03
	H func.	0.86	1.16	0.79	2.24	0.06	0.18	0.01	0.03
	K func.	0.17	0.34	0.05	0.08	0.01	0.01	0.01	0.01
	Other	0.00	0.00	0.89	2.39	0.08	0.21	0.03	0.06
		464.82	1188.27	33.93	104.65	4.07	12.67	0.81	2.55
	Total	812.38	2150.16	57.06	177.23	8.02	25.35	1.06	3.21

5.1 Benchmark software performance

Software performance of BIKE, collected on a range of computing platforms, provides a benchmark for the quality of the proposed FPGA-based hardware architecture. We consider computing platforms ranging from low-end ARM-based embedded systems to desktop-class Intel CPUs. Moreover, different computing platforms can exploit different versions of BIKE software implementations. Our software performance analysis includes 32- and 64-bit architectures, ARM and x86 ISAs, embedded- and desktop-class processors, and plain-C99 and AVX2-optimized software.

The results of this analysis are detailed in Table 5.1. For each CPU and software implementation, we executed BIKE 100 times and average the collected execution times. Performance data for software execution was collected on a 32-bit ARM Cortex-A9 CPU, on a 64-bit ARM Cortex-

A53 CPU, and on a Intel Core i5-10310U CPU, which represent different platform types across the computing spectrum.

ARM Cortex-A9 is an embedded-class 32-bit processor implementing the ARMv7-A ISA. We execute BIKE on a ARM Cortex-A9 dual-core processor featured on a Xilinx Zynq-7000 heterogeneous SoC, which couples the ARM processor with programmable FPGA logic. The ARM CPU part has a clock frequency up to 667MHz, and the external memory mounted on the employed Digilent Zedboard development board, which houses the Zynq-7000 chip, is a 512MB DDR3. The BIKE software [5] is executed on top of the Xilinx Petalinux operating system.

ARM Cortex-A53 is an embedded-class 64-bit processor implementing the ARMv8-A ISA. In particular, we consider the RP3A0 system-in-package mounted on a Raspberry Pi Zero 2 W, that features a quad-core 64-bit ARM Cortex-A53 processor clocked up to 1GHz and 512MB of SDRAM. We executed the 64-bit portable C99 implementation of BIKE [5] on the Raspberry Pi running the 64-bit Raspberry Pi OS Lite operating system, that is based on Debian 11, and setting a fixed 1GHz clock frequency through Linux *cpupower* tools.

Intel Core i5-10310U is a desktop-class 64-bit processor implementing the x86-64 ISA and providing support for the Intel AVX2 extension. The PC mounting the Intel CPU ran the Ubuntu 20.04.3 LTS operating system. We executed the 64-bit portable C99 implementation and the AVX2-optimized version [5]. The non-AVX2 executed at a 4.2GHz average clock frequency, while the AVX2 one ran at 4GHz.

The range of computing platforms considered in the software benchmarking phase resulted in significant differences in terms of absolute performance when executing the BIKE software, as shown by data provided in Table 5.1.

On the lower end, the ARM Cortex-A9 platform, a 32-bit CPU running at 667MHz, provided execution times of 812ms and 2150ms, i.e., in the order of seconds, for BIKE instances with NIST security levels 1 and 3, respectively.

Moving to a more efficient code that made use of 64-bit instructions, as well as to a more modern and 64-bit ARMv8-A architecture, provided a speedup of more than 10×. The performance on the ARM Cortex-A53 64-bit CPU, also running at a higher 1GHz clock frequency, measured at 57ms and 177ms for AES-128 and -192 security instances of BIKE, respectively.

Executing the same software implementation of BIKE on the Intel CPU resulted in a further speedup of around 7×. The different architecture and the higher clock frequency, in the order of 4GHz, allowed executing BIKE instances with security levels 1 and 3 in 8ms and 25ms, respectively.

Table 5.2: *Breakdown of the execution times of AES-128 security instances of BIKE, expressed in milliseconds, for different state-of-the-art FPGA-based implementations. Legend:* ***LW*** *lightweight,* ***TO*** *trade-off,* ***HS*** *high-speed,* ***HLS*** *high-level synthesis.*

KEM	**Reference implementation**					
primitive	**LW [89]**	**HS [89]**	**LW [88]**	**TO [88]**	**HS [88]**	**HLS [74]**
Key generation	21.90	2.69	3.79	1.87	1.67	137.84
Encapsulation	1.25	0.13	0.44	0.28	0.13	6.33
Decapsulation	13.35	1.97	6.90	4.21	1.89	135.48
Total	36.50	4.79	11.14	6.36	3.70	279.65

Finally, we evaluated the execution, on the same Intel CPU, of a software implementation that made use of instructions from the Intel AVX2 extension. The execution times of 1.1ms and 3.2ms are around 8× smaller than those obtained by the plain-C99 software, which highlights the effectiveness of those dedicated instructions in a software making wide use of binary polynomial arithmetic.

5.2 Benchmark hardware performance

We evaluate the performance of state-of-the-art hardware implementations of BIKE as a further benchmark for the quality of the proposed FPGA-based hardware architecture. We consider both human-designed and HLS-generated ones, all of them targeting FPGA architectures. The results of this analysis are detailed in Table 5.2.

All the hardware state-of-the-art implementations target either Xilinx Artix-7 FPGAs or Xilinx Zynq-7000 heterogeneous SoCs, which pair a ARM CPU with programmable FPGA logic equivalent to the Artix-7 one. Moreover, the Xilinx Artix-7 platform is the target for hardware implementations within the NIST PQC standardization process.

The lightweight instance of [89] is faster than 64-bit ARM software execution, while the high-speed instance takes less than the Intel CPU executing plain-C99 software, taking less than 5ms compared to just above 8ms.

The lightweight, trade-off, and high-speed FPGA-based implementations of [88] range from 11.14ms to 3.70ms, further improving performance but still slower than the AVX2 software executed on a Intel CPU with 4GHz average clock frequency, which takes instead 1.06ms. The high-speed instance is anyway more than two times faster than plain-C99 software execution on the same Intel CPU.

Finally, the HLS-designed hardware implementation of BIKE [74] pro-

vides a speedup up to 2.9× over the software execution on a ARM 32-bit CPU, however requiring more FPGA resources than those available on the target Zynq-7020 chip. A HW/SW approach executing encapsulation on the CPU rather than implementing it on the FPGA still provides a 2.78× speedup while satisfying the resources constraints.

The orders of magnitude of difference in the performance between human-designed hardware implementations and HLS-generated ones highlight the difficulty of HLS tools to make an efficient use of FPGA resources, in particular for applications as complex as the BIKE cryptosystem.

5.3 Experimental setup

This section details the experimental setup for the overall architectures implementing the client and server functionalities of the BIKE KEM. Further details about the validation and evaluation of the single components that are part of the whole KEM architecture are provided later in their corresponding sections. Remarkably, the adoption of the LEDAcrypt PKC and KEM as use cases for the evaluation of some components produces results that are fully comparable to those that would be obtained by targeting the BIKE cryptosystems. The LEDAcrypt and BIKE schemes are indeed QC-MDPC code-based cryptoschemes, employ therefore the same underlying arithmetic, and the code parameters are in the same orders of magnitude. The specific code parameters are explicitly detailed whenever not targeting the BIKE cryptoscheme.

5.3.1 BIKE code parameters

The proposed client and server architectures target the security levels 1 and 3 of the BIKE KEM, which correspond to AES-128- and AES-192-equivalent security and each with a different underlying QC-MDPC code. Such two security levels are also targeted by the reference software [5] and hardware [25] implementations. The employed QC-MDPC codes have a $2p$-bit code word length and a p-bit information word length. For each BIKE code *Bj*, where *j* corresponds to the security level, Table 2.1 reports the size p of h_i blocks of H, the Hamming weight v of the rows of h_i blocks, the Hamming weight t of e, and the number of decoding iterations $iter$.

5.3.2 LEDAcrypt code parameters

As previously mentioned, part of the experimental evaluation of the parametric components employed within the proposed BIKE client and server

cores targeted cryptoschemes from the LEDAcrypt cryptography suite. The LEDAcrypt suite consists of QC-MDPC code-based post-quantum KEM and PKC schemes, similar to the BIKE scheme. In particular, the proposed architectures for QC-MDPC bit-flipping decoding, dense-dense binary polynomial multiplication, and binary polynomial exponentiation were evaluated considering the QC-MDPC codes employed by the KEM and PKC schemes from the LEDAcrypt suite. Experimental results for the inversion module target instead both LEDAcrypt and BIKE codes. This section discusses, for each component evaluated targeting LEDAcrypt schemes, how the corresponding code parameters relate to the BIKE ones.

The evaluation of the QC-MDPC bit-flipping decoder was carried out considering QC-MDPC code parameters for the LEDAcrypt-KEM-CPA scheme, which are reported later in Table 5.4. In particular, the $C1$ and $C4$ codes are the most similar to those employed within AES-128- and AES-192-equivalent security BIKE instances, i.e., $B1$ and $B3$ in Table 2.1. Those codes share the same number n_0 of blocks in the parity-check matrix H, equal to 2, while the Hamming weight v of a row of the H matrix of the $C1$ and $C4$ codes also corresponds to the v value of the $B1$ and $B3$ codes, respectively. The p code parameters are in the same order of magnitude, with LEDAcrypt-KEM-CPA codes having values of 10883 and 21011 compared to the 12323 and 24659 polynomial lengths of the BIKE ones. Finally, the number of decoding iterations is also comparable, with LEDAcrypt-KEM-CPA and BIKE schemes only differing by one iteration, i.e., 6 and 5, respectively. Moreover, LEDAcrypt and BIKE schemes employ similar bit-flipping decoding algorithms, with the only difference of the additional black and gray iterations employed in the BGF decoding variant implemented by the latter.

The dense-dense binary polynomial multiplier was evaluated considering the polynomial lengths employed by the nine configurations of the LEDAcrypt PKC scheme. The range of values held by the polynomial degree p across the different configurations of the LEDAcrypt PKC scheme is similar to the values of p employed in BIKE. As listed later in Table 5.6, the nine values of p considered for the experimental evaluation range from 8467 ($P1$) to 37619 ($P9$), whereas the BIKE $B1$ and $B3$ codes have values of p equal to 12323 and 24659, respectively. In particular, the $P2$ and $P3$ codes, with polynomial lengths p of 9643 and 14717, respectively, can be considered lower and upper bounds for BIKE's $B1$ code, which has a p value of 12323. Similarly, the $P7$ and $P8$ codes, with polynomial lengths p of 24533 and 28477, respectively, can be considered lower and upper bounds for BIKE's $B3$ code, which has a p value of 24659.

Finally, the binary polynomial exponentiation results are reported for polynomials with length specified as in the $C1$, $C4$, and $C7$ LEDAcrypt-KEM-CPA codes used for the QC-MDPC bit-flipping decoding experimental evaluation, while binary polynomial inversion was evaluated against both the nine LEDAcrypt-KEM-CPA polynomial lengths reported in Table 5.4 and the BIKE $B1$ and $B3$ codes listed in Table 2.1 and targeted by the proposed BIKE client and server architectures.

5.3.3 Software setup

The software implementation of BIKE considered as the software reference is the open source version freely available online [5]. It provides both a baseline C99 portable software implementation and an optimized code that employs the Intel AVX2 extension, thus providing higher performance. The two software versions were executed on an Intel Core i5-10310U desktop-class CPU, forcing a fixed operating frequency of 4.4 GHz to avoid performance variability due to power management. For each BIKE code configuration, the execution times of key generation, encapsulation, and decapsulation for the C99 and AVX2 software have been obtained as the average of 30 executions.

5.3.4 Hardware setup

The architectures discussed in Section 4.1 have been described in SystemVerilog and implemented in Xilinx Vivado 2020.2, targeting Artix-7 FPGAs and a clock frequency of 91 MHz.

The Xilinx Artix-7 FPGA family was selected as the target for the experimental evaluation of the proposed architectures for multiple reasons. On the one hand, the Xilinx Artix-7 family is the de-facto standard in academic research, including the cryptography field, due to its wide availability and to having the best price-performance ratio among FPGAs. On the other hand, NIST targets specifically Artix-7 FPGAs for the hardware implementations of cryptoschemes in its post-quantum standardization process, in order to provide a fair comparison environment for all proposals by avoiding differences due to the usage of different FPGA technologies or technology nodes of ASIC implementations.

All the identified instances of the proposed architectures satisfy the area constraints given by the available resources on the target Xilinx Artix-7 FPGAs, which are listed in Table 5.3, and the timing requirements, i.e., a 91 MHz clock frequency. For each considered FPGA and code configuration, in the following only the best hardware implementations, i.e., those

Table 5.3: *Available resources on FPGAs from the Xilinx Artix-7 family, expressed in terms of look-up tables (LUT), flip-flops (FF), and block RAM (BRAM).*

FPGA	LUT	FF	BRAM
Artix-7 12	8000	16000	20
Artix-7 15	10400	20800	25
Artix-7 25	14600	29200	45
Artix-7 35	20800	41600	50
Artix-7 50	32600	65200	75
Artix-7 75	47200	94400	105
Artix-7 100	63400	126800	135
Artix-7 200	134600	269200	365

with the smallest execution time and which satisfy the area and timing constraints, are reported. Such instances have been identified after a design space exploration that employed the four-step, complexity-oriented heuristic described in Section 4.8 considering the configurable parameters of the architecture, which are the parallelism of the dense-sparse multiplier (Par_{DSMul}), of the bit-flipping decoder (Par_{Dec}), and of the dense-dense multiplication (Par_{DDMul}) and exponentiation components (Par_{Exp}) within the inversion module. For simplicity, we fixed the bandwidths of the dense-sparse multiplier (BW_{DSMul}), bit-flipping decoder (BW_{Dec}), and inversion module (BW_{Inv}) to the same value as the bandwidth of the client and server top modules (BW).

The performance of the proposed architectures was collected by averaging the execution times of 10000 operations of each analyzed component, i.e., the client and server cores and the decoding and arithmetic modules, while their area consumption was obtained as the FPGA resource utilization after their implementation, i.e., place and route, in Xilinx Vivado 2020.2.

5.3.5 Functional validation

The proposed client and server architectures, as well as the components implementing decoding and arithmetic operations, have been functionally validated through both post-implementation timing simulation and board prototype execution, checking the correctness of the obtained results against the reference software implementation of BIKE [5].

Post-implementation simulation targeted the Artix-7 35 (*xc7a35tcpg236-1*), Artix-7 50 (*xc7a50tcpg236-1*), and Artix-7 200 (*xc7a200tsbg484-1*) Xilinx FPGAs, while board prototype execution targeted the Digilent Nexys 4 DDR board, that features an Artix-7 100 (*xc7a100tcsg324-1*) FPGA. In both cases, we implemented instances of the proposed architectures for

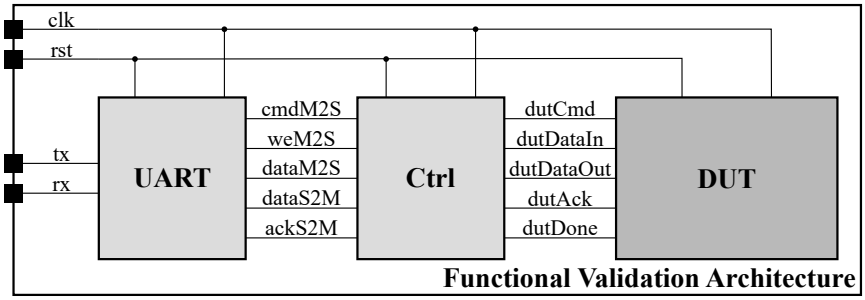


Figure 5.1: *Hardware setup for the functional validation.*

each code configuration and for each target FPGA. Each hardware instance executed 10000 operations, depending on the specific design under test, and their results were compared with the corresponding outputs of software execution to check their correctness.

Figure 5.1 describes the functional validation architecture used for both post-implementation simulation and prototype execution. The functional validation architecture is made of three parts. The FPGA controller (`Ctrl`) communicates with the host computer to collect the input and return the output, the `UART` module creates a communication channel between the FPGA controller and the host computer, and the `DUT` block represents an instance of the design under test. The `DUT` block corresponds indeed to either a BIKE client or server core or to a QC-MDPC bit-flipping decoding or binary polynomial inversion, multiplication, or exponentiation component. To perform a DUT computation, the `Ctrl` module drives the `cmdM2S` and `weM2S` signals to collect the inputs from the `UART` module. The FPGA controller waits until the UART has sent the required data before closing the communication, which implements a blocking protocol. Once the input has been collected, the `dutCmd` signal is used to load the inputs into `DUT` and to start the computation. BW bits per clock cycle of the input data are passed to the DUT module through the `dutDataIn` signal. The `DUT` module signals the end of the computation through the `dutDone` control signal while BW bits of $c(x)$ per clock cycle are loaded into `Ctrl` through the `c` signal. The `DUT` and `Ctrl` modules exchange data through an acknowledged protocol (see `cmdDut` and `dutAck` signals). Finally, the `Ctrl` module sends the result back to the `UART` module through the `dataM2S` signal. The `Ctrl` and `UART` modules also exchange data through an acknowledged protocol (see `cmdM2S` and `ackS2M` signals).

Table 5.4: *Code parameters of the LEDAcrypt-KEM-CPA configurations [11].*

Security level	LEDAcrypt-KEM-CPA configuration	Code parameters n_0	p	v	$iter$
AES-128	C1	2	10883	71	6
	C2	3	8237	79	5
	C3	4	7187	83	4
AES-192	C4	2	21011	103	6
	C5	3	15373	117	5
	C6	4	13109	123	4
AES-256	C7	2	35339	137	4
	C8	3	25603	155	4
	C9	4	21611	163	4

5.4 QC-MDPC bit-flipping decoding

The QC-MDPC bit-flipping decoding architecture discussed in Section 4.2 was evaluated with respect to its resource utilization on Xilinx Artix-7 FPGAs and to its performance compared to a reference software execution.

The proposed decoding architecture was instantiated for each of the nine Ci LEDAcrypt-KEM-CPA code configurations listed in Table 5.4 and compared to the corresponding software execution of two reference software implementations of QC-MDPC bit-flipping decoding extracted from the official implementation of the LEDAcrypt-KEM-CPA cryptoscheme [11]. The baseline C11 and AVX2-optimized software implementations of the bit-flipping decoder were adapted to always execute $iter$ iterations, i.e., early termination when obtaining a syndrome with a null Hamming weight was disabled to obtain a constant number of decoding iterations. The software-implemented decoding was executed on an Intel Core i7-6700HQ processor, which supports the Intel AVX2 extension and runs at a clock frequency up to 3.5GHz.

The architecture for QC-MDPC bit-flipping decoding was implemented on the Artix-7 12 (*xc7a12tcsg325-1*) and Artix-7 200 (*xc7a200tsbg484-1*) FPGAs, respectively the lowest and highest end of the Xilinx Artix-7 family, targeting a 100 MHz operating frequency, i.e., a 10 ns clock period.

The experimental results detailed in the rest of this section are reported for only the best combination of parameters of each code configuration, i.e., the set of parameters that produces the feasible decoder instance which provides the best performance in terms of decoding execution time. Such configurations have been identified after an extensive design space exploration that explored four bandwidths (BW_{Dec}), i.e., 32, 64, 128 and 256

Table 5.5: *Configuration parameters for the decoder instances on Artix-7 FPGAs.*

LEDAcrypt-KEM-CPA	**Artix-7 12**		**Artix-7 200**	
Configuration	BW_{Dec}	Par_{Dec}	BW_{Dec}	Par_{Dec}
C1	32	4	128	24
C2	32	4	128	32
C3	32	4	128	32
C4	32	2	128	24
C5	32	4	128	24
C6	32	4	128	24
C7	32	1	128	24
C8	32	2	128	24
C9	32	1	128	24

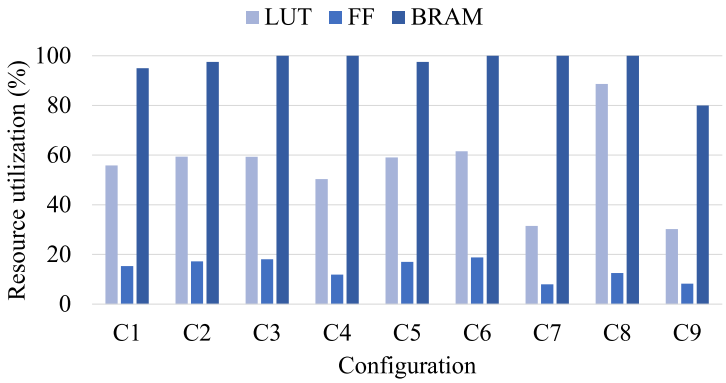
bits, and a set of values comprised between 1 and 32, i.e., 1, 2, 4, 8, 16, 24 and 32 bits, for the parallelism degree (Par_{Dec}) in the UPC and syndrome update computation. The hardware decoding instances and their identified architectural parameters are listed in Table 5.5.

5.4.1 Area results

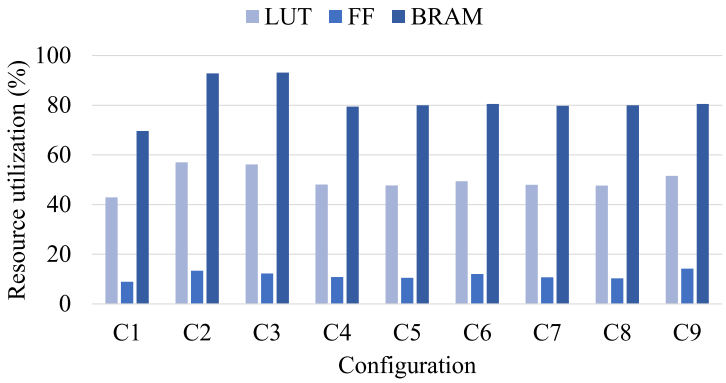
The proposed decoder makes use of the BRAMs of the FPGA as the primary means of storage for the inputs, the intermediate values and the result, allowing the decoder to fit on tiny FPGAs even for codes with a large block size p. In such a way, the maximum allowed dimension of the dense vectors that store the syndrome, the error and the UPCs is not a function of the available amount of flip-flops, that easily become the scarcest resources on small FPGAs, but it is instead a function of the available BRAM storage capacity. We note that a single BRAM can store up to 36kb and the smallest considered FPGA features 20 BRAMs.

For each configuration of the LEDAcrypt cryptosystem, considering the Xilinx Artix-7 12 and Artix-7 200 FPGAs, Figure 5.2 reports the normalized resource utilization of the look-up table (LUT), flip-flops (FF), and block RAM (BRAM) elements, as a percentage on the total available.

As expected, the use of BRAM resources dominates each design on both the Xilinx Artix-7 12 and Artix-7 200 thus minimizing the use of flip-flops, which are therefore never the scarcest resource. We note that even if the flip-flop utilization is low, the unused flip-flop resources can not be exploited to further improve the design. For example, on average the FF utilization on the Xilinx Artix-7 12 is below 15%, while the BRAM utilization is above 95% (see Figure 5.2a). However, the entire Xilinx Artix-7 12 features 16,000 FFs, thus their contribution is lower that the storage capacity of a single



(a) *Xilinx Artix-7 12*



(b) *Xilinx Artix-7 200*

Figure 5.2: *Resource utilization of the proposed QC-MDPC bit-flipping decoder implemented on the Xilinx Artix-7 12 and Artix-7 200 FPGAs. The utilization for LUT, FF, and BRAM resources is expressed as a percentage of the available resources on the target FPGA.*

BRAM. In a similar manner, the FF utilization on the Xilinx Artix-7 200 is lower than 15% for each LEDAcrypt configuration. Even in such scenario, it is impossible to improve the design by leveraging on the FF resources. Instead, the average BRAM resource utilization is 82%, thus the storage capacity is never the bottleneck of the implemented designs. In contrast, the limiting factor to a better resource utilization is the timing, which becomes the bottleneck on the decoder implementations on the Xilinx Artix-7 200, due to the massive parallelism that is achieved thanks to the vast amount of available resources. However, a complete performance discussion is left to the performance evaluation part in the following of this section.

Considering the LUT resources, we note an average utilization of 55% and 50% on the Xilinx Artix-7 12 and Xilinx Artix-7 200, respectively.

Although such resource type never becomes the scarcest one across the entire set of considered decoder implementations, its utilization varies depending on the actual level of parallelism of each implemented decoder, since LUTs are used to implement the combinational logic of the decoder. Table 5.5 reports the level of parallelism for the two design-time knobs of our decoding architecture for each combination of FPGA and LEDAcrypt configuration. The 32-bit bandwidth (BW_{Dec}) is found to be the optimal value to implement each LEDAcrypt configuration on the Xilinx Artix-7 12 FPGA, while the optimal level of parallelism Par_{Dec} to maximize the computation of the two vector-matrix multiplications in the bit-flipping procedure, i.e., $UPC = H \cdot s$ and $s = H \cdot e$, ranges between 1 and 4, thus determining a variability in the used LUTs depending on the implemented LEDAcrypt configurations. For example, LUT utilization is around 60% for configurations in the range $C1 - C6$, while it drops down to 30% for $C7$ and $C9$ and it peaks to almost 90% for $C8$ (see Figure 5.2a). We note that Par_{Dec} is equal to 4 for $C1 - C6$ configurations, thus determining a higher use of LUTs with respect to $C7$ and $C9$ configurations, while the latter decoder instances targeting larger QC-MDPC codes are implemented with a Par_{Dec} value of 1. Similarly, the large use of LUTs for $C8$ is motivated by both the larger QC-MDPC code compared to the one of $C1 - C6$ and the possibility of using a Par_{Dec} parallelism of 2 without exceeding the available hardware resources of the FPGA. Considering the implemented decoders on the Xilinx Artix-7 200, the average LUT utilization is 50% with small variations between different configurations (see Figure 5.2b). The 128-bit bandwidth has been found optimal for the entire set of LEDAcrypt configurations, while a Par_{Dec} value of 24 is employed for all the configurations but $C2$ and $C3$, for which 32 is used instead (see Table 5.5). Such increase in the parallelism for $C2$ and $C3$ impacts the used LUTs, for which a value slightly below 60% is reported (see LUT for $C2$ and $C3$ in Figure 5.2b). We note one more time that the impossibility of resorting to either a more aggressive level of parallelism or a larger bandwidth for the decoders implemented on the Artix-7 200 FPGA, is due to the imposed timing constraints equal to 10ns and not to the available resources on the board. However, as explained in the following, such decoder implementations still allow overcoming the performance of optimized software-implemented decoders employing the Intel AVX2 extension by 5 times, on average.

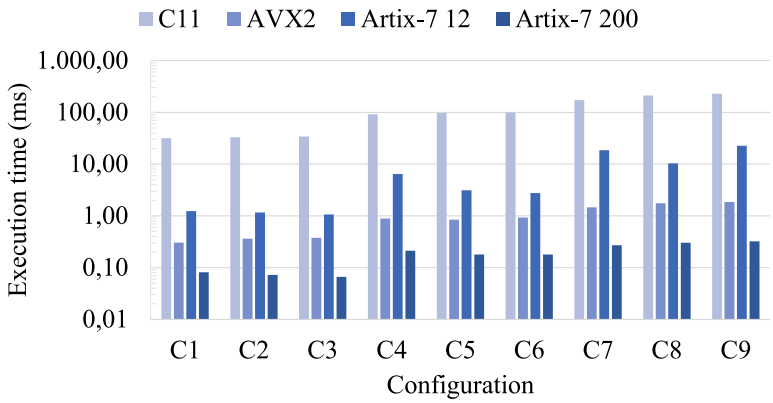


Figure 5.3: *Execution time (in milliseconds) of QC-MDPC bit-flipping decoding. Results are shown for software decoding on the Intel i7 processor and for hardware decoding on the Artix-7 12 and Artix-7 200 FPGAs.*

5.4.2 Performance results

Figure 5.3 reports the performance results expressed as the execution time to complete the bit-flipping decoding procedure for all the LEDAcrypt configurations. The results are reported for each configuration considering the two software implementations, i.e., `C11` and `AVX2`, and the two hardware implementations, which targets the Xilinx Artix-7 12 and Artix-7 200 FPGAs, respectively. As expected, the execution time increases with the size and the weight of the QC-MDPC code for all the implementations. For example, `C11` takes 32 ms and 229 ms to complete the decoding of $C1$ and $C9$, respectively (see Figure 5.3).

In order to highlight the actual performance speedup across the different implementations of the decoding procedure, Figure 5.4 reports the performance speedup of the AVX2 software and of the two hardware implementations, normalized with respect to the *C11* software version. The decoders targeting the low-end Xilinx Artix-7 12 FPGA show an execution time comprised between 1 and 25 milliseconds, with a corresponding performance improvement between 9 and 36 times (23 times on average) with respect to the `C11` software implementation.

We note that the optimized software implementation employing the Intel AVX2 extension (`AVX2`) shows an average performance speedup of 108× compared to the *C11* reference software version. However, our decoders targeting the Xilinx Artix-7 200 FPGA show a further average 5× speedup against the performance-optimized software employing the Intel AVX2 extension (see Figure 5.4). Such results demonstrate the superior performance

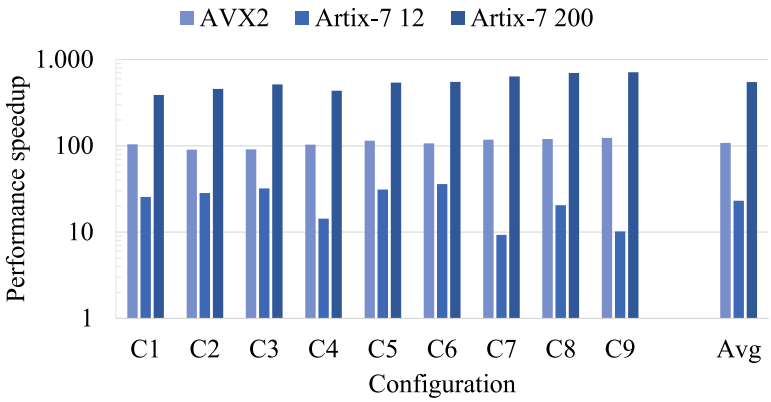


Figure 5.4: *Performance improvement with respect to C11 software decoding executed on the Intel i7 processor. Results are shown for AVX2 software decoding on the Intel i7 processor and for hardware decoding on the Artix-7 12 and Artix-7 200 FPGAs.*

and scalability of our decoding architecture against optimized software solutions exploiting custom and hardware-accelerated instructions offered by recent high-end Intel processors.

5.5 Dense-dense binary polynomial multiplication

The dense-dense binary polynomial architecture discussed in Section 4.4 was evaluated with respect to its resource utilization on Xilinx Artix-7 FPGAs and to its performance compared to a reference software execution.

The proposed architecture was instantiated for each of the nine Pi LEDAcrypt PKC configurations listed in Table 5.6 and compared to the corresponding software execution of the binary polynomial multiplication by exploiting the *gf2x* C library [21], in particular the 1.3.0 version. The *gf2x* C library implements the Karatsuba, Toom-Cook and FFT multiplication algorithms and represents the state-of-the-art for software-implemented large binary polynomial multiplications. The software-implemented multiplication was executed on an Intel Core i7-6700HQ processor, which supports the Intel AVX2 extension and runs at a clock frequency up to 3.5GHz.

The architecture for dense-dense binary polynomial multiplication was implemented on the Artix-7 12 (*xc7a12tcsg325-1*) and 200 (*xc7a200tsbg484-1*) FPGAs, respectively the lowest and highest end of the Xilinx Artix-7 family, targeting a 143 MHz operating frequency, i.e., a 7 ns clock period.

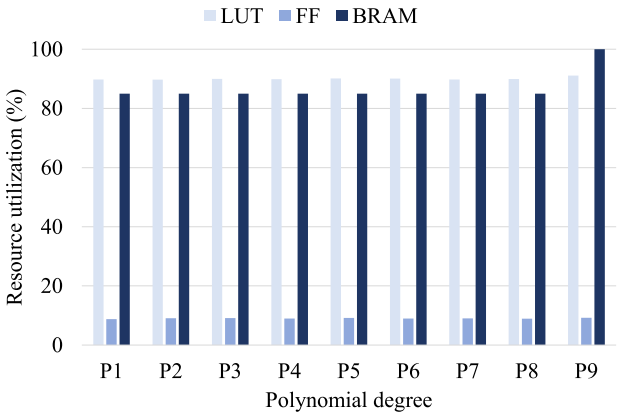
Table 5.6: *Code parameters of the LEDAcrypt-PKC configurations [11].*

Security level	LEDAcrypt PKC configuration	Code parameters n_0	p
AES-128	P4	2	15013
	P2	3	9643
	P1	4	8467
AES-192	P7	2	24533
	P5	3	17827
	P3	4	14717
AES-256	P9	2	37619
	P8	3	28477
	P6	4	22853

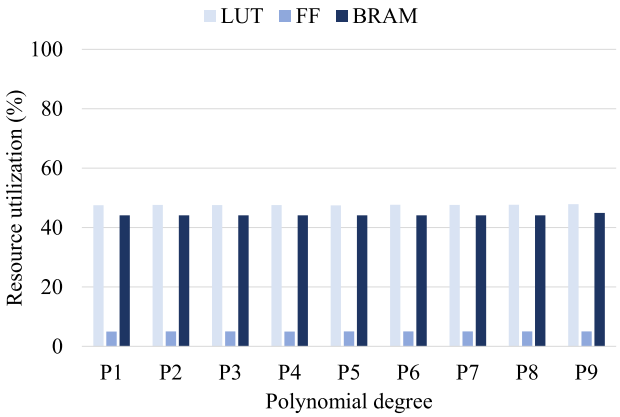
5.5.1 Area results

The proposed dense-dense multiplier exploits a massive BRAM utilization to store partial products and the final result with two positive side-effects. First, the multiplier can be implemented on tiny FPGAs even for the multiplication of large operands. In particular, the maximum allowed dimension of the operand in bits is not function of the available amount of flip-flops, that easily become the scarcest resources on small FPGAs, but it is function of the available BRAM storage capacity. We note that a single BRAM can store up to 36kbit and that the smallest considered FPGA features 20 BRAMs. Second, the use of a nested structure to implement the multiplier where storage elements surround the compute stage, optimizes the critical path by construction. In particular, the critical path, which remains independent from the number of implemented Karatsuba iterations, depends on the width (BW) of the combinational multiplier. This, in turn, determines the critical path of the proposed multiplier, that, however, cannot be improved by a reduction of the value of BW. In fact, any reduction of the value of BW aiming to optimize the critical path of the combinational multiplier generates a much more severe overall performance degradation due to the underutilization of the BRAM data-transfer bandwidth.

Figure 5.5 reports the normalized resource utilization for each polynomial size of the LEDAcrypt cryptosystem considering the Xilinx Artix-7 12 and the Xilinx Artix-7 200 FPGAs. In particular, for each polynomial size, the percentage utilization of LUT, flip-flops and BRAM elements is reported. We note that the massive use of BRAM resources minimizes the use of flip-flops, which are therefore never the scarcest resource, while LUT and BRAM utilization are almost aligned even if the reported utilization greatly differs between the two considered FPGAs. For each of the 9



(a) *Xilinx Artix-7 12*



(b) *Xilinx Artix-7 200*

Figure 5.5: *Resource utilization of the dense-dense multiplier implemented on Xilinx Artix-7 12 and Artix-7 200 FPGAs. Relative utilization of LUT, FF, and BRAM resources is expressed as the percentage of the available resources on the target FPGA.*

LEDAcrypt configurations, both the Artix-7 12 and 200 FPGAs have the internal bandwidth of the multiplier set to 64 bits, which corresponds to the bandwidth of the BRAM memories available on the Artix-7 family, making it an optimal choice. For Xilinx Artix-7 12, all the 9 considered polynomials have their optimal hardware configuration with 1 Karatsuba recursion and 3 Comba multipliers, i.e., 3 partial products are computed in parallel. All configurations almost saturate the FPGA resources in terms of LUT and BRAM, while the low FF utilization, i.e., 8% on average, is due to the massive use of BRAM to store intermediate values. We note that the unused flip-flops cannot be efficiently employed, since even all together they cannot contain the information stored in a single BRAM.

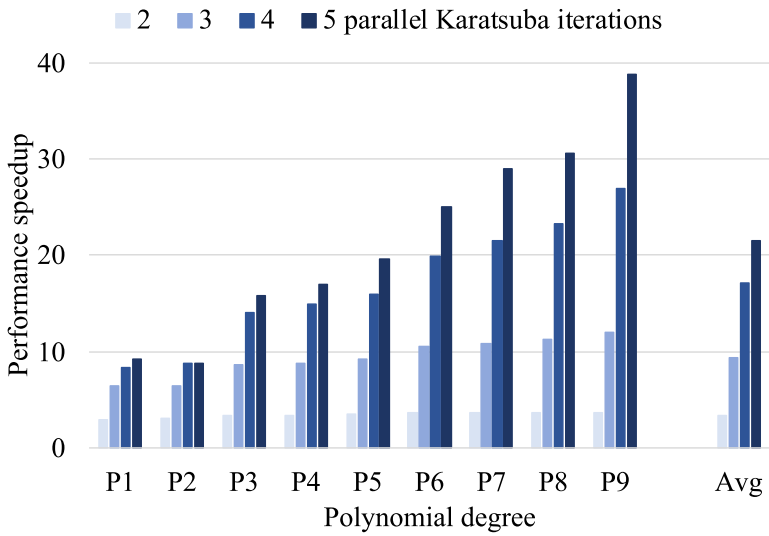


Figure 5.6: *Performance speedup with respect to the hardware multiplication performed with 1 parallel Karatsuba iteration. Results are shown for hardware multipliers with a number of parallel Karatsuba iterations varying between 2 and 5.*

For Xilinx Artix-7 200, all the 9 considered polynomials have their optimal hardware configuration with 3 Karatsuba recursion and 27 Comba multipliers. However, the resource utilization for both the LUT and the BRAM is limited to 50%. To better understand this supposedly low resource utilization, we need to analyze the Karatsuba algorithm. In particular, such algorithm allows to substitute a partial product computation with a few binary additions in $\mathbb{Z}_2[x]$. Considering the proposed architecture, a new set of BRAM is used at each iteration of the Karatsuba and additional LUTs are used to perform the additional operations and to compose the intermediate results into the final partial product. Moreover, the use of too many nested Karatsuba iterations can negatively affect the performance since the time spent to split the operands becomes bigger than the time spent to actually perform the Comba multiplication, i.e., Comba operands are too small. To this extent, only the use of a parallel Karatsuba iteration offers a significant performance speedup with, however, a non-negligible cost in terms of resource utilization. For each LEDAcrypt configuration, the performance speedup due to the nested implementation of parallel Karatsuba iteration is reported in Figure 5.6. The performance speedup is defined as the ratio between the execution times on a hardware multiplier with only 1 parallel Karatsuba iteration and on hardware multipliers with a number of parallel Karatsuba iterations comprised between 2 and 5. We note that

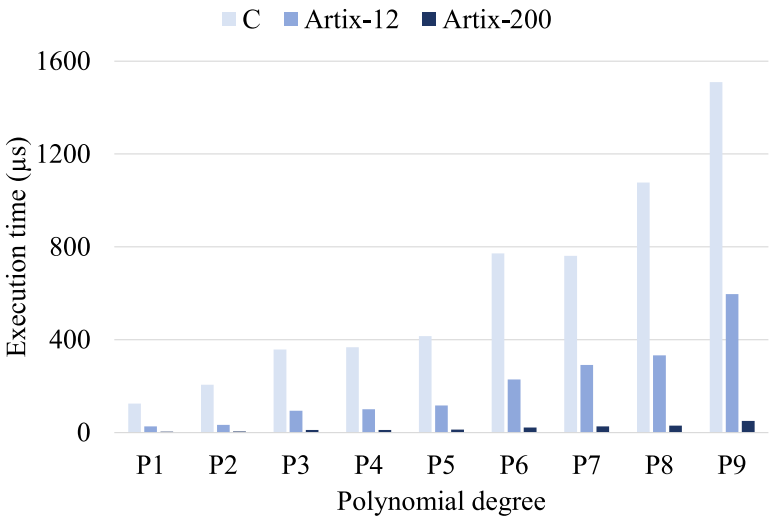


Figure 5.7: *Execution time (in microseconds) of a multiplication. Results are shown for software multiplication on the Intel i7 core and hardware multiplication on the Artix-7 12 and Artix-7 200 FPGAs.*

its value is always significantly positive, while the number of required BRAMs and LUTs grows $3\times$ for each parallel Karatsuba iteration. To this extent, the resource utilization on the Xilinx Artix-7 200 is motivated by the impossibility to add another Karatsuba iteration due to resource limitation, while by using a larger FPGA, such as those of the Xilinx Virtex-7 family, the proposed multiplier can further improve its offered performance.

5.5.2 Performance results

Figure 5.7 reports the performance, i.e., the execution time, for all the 9 considered polynomial degrees, thus covering all the LEDAcrypt cryptoscheme configurations. For each polynomial degree, results are reported for the hardware implementations targeting the Artix-7 12 and 200 FPGAs and for the software reference. Considering the LEDAcrypt PKC use case, a multiplication executed with the *gf2x* library takes between 124 and 1510 microseconds, while our hardware multipliers implemented on the Artix-7 12 and 200 FPGAs take respectively between 27 and 597 and between 4 and 50 microseconds.

We define the performance improvement metric as the ratio between the execution times of a single multiplication on the software reference implementation and on our hardware multipliers.

The Artix-7 200 implementation of the proposed multiplier, as shown

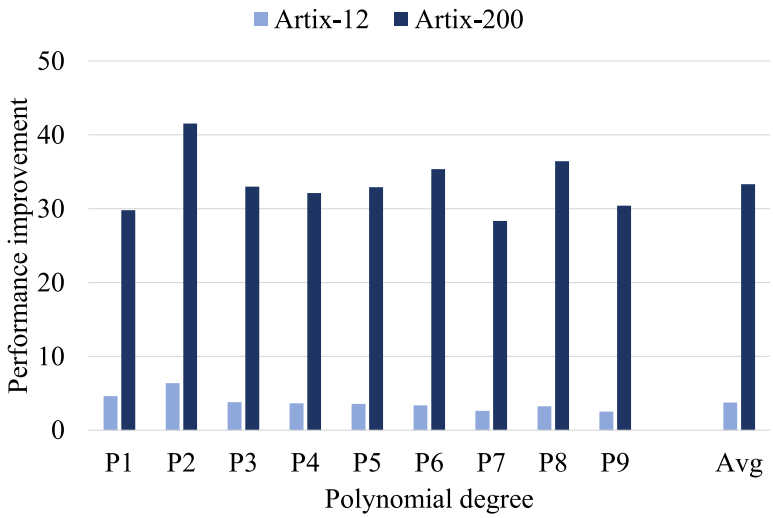


Figure 5.8: *Performance improvement with respect to C11 software multiplication executed on the Intel i7 processor. Results are shown for hardware multiplication on the Artix-7 12 and Artix-7 200 FPGAs.*

in Figure 5.8, offers a performance speedup between 28.3 and 41.5 times (33.3 times faster on average) compared to the software implementation. Similarly, the Artix-7 12 implementation of the proposed multiplier offers a performance speedup between 2.5 and 6.4 times (3.6 times faster on average) compared to the software implementation. It is worth noticing that, despite the ad-hoc hardware microarchitecture, the FPGA implementation works at 143 MHz while the software multiplication executed on an Intel i7 processor clocked at 3.5 GHz.

5.6 Binary polynomial exponentiation

This section briefly discusses the resource utilization and performance of the exponentiation architecture described in Section 4.5. Table 5.7 lists the area and performance metrics for instances of the exponentiation module targeting polynomials with length p values of 10883, 21011, 35339, as in the $L1.2$, $L3.2$, and $L5.2$ LEDAcrypt KEM-CPA instances, respectively (see $C1$, $C4$, $C7$ in Table 5.4). The experimental results are provided for different values of the configurable parameters BW and Par_{Exp} to highlight how their variations impact the area and performance metrics. The listed hardware instances are synthesized and implemented on Xilinx Artix-7 FPGAs targeting a 133MHz clock frequency, i.e., a 7.5ns clock period.

Results in Table 5.7 highlight a mostly linear impact of both the BW

Table 5.7: *Resource utilization and performance of the proposed exponentiation architecture for different configurations of code and architectural parameters.*

p	BW	Par_{Exp}	**LUT**	**FF**	**BRAM**	**Latency [us]**
10883	32	1	312	163	1	84.4
		2	599	309	1.5	43.5
		4	981	567	2.5	23.0
		8	1891	1065	4.5	12.8
		16	3501	2006	8.5	7.7
		32	6451	3425	16.5	5.1
	64	1	420	191	2	83.4
		32	9743	4937	33	3.8
		64	19291	8838	65	2.6
21011	32	1	341	177	2	162.6
		32	8574	3732	33	9.9
	64	1	426	203	2	160.4
		32	9940	5337	33	7.4
		64	17783	9331	65	4.9
35339	32	1	433	187	3	273.5
		32	7982	4009	49.5	16.6
	64	1	489	215	4	269.6
		32	12410	5381	66	12.4
		64	23812	9840	130	8.3

and Par_{Exp} configurable architectural parameters on the FPGA resource utilization and on the execution time. Moreover, the p code parameter, i.e., the polynomial length, also shows a linear impact on both the area and performance metrics. The specific inputs to the exponentiation, i.e., the base polynomial and the integer exponent, do not impact instead in any way the execution time of an exponentiation operation, which is computed in a constant time for a given set of p, BW, and Par_{Exp} parameters.

Notably, the smallest instances for each polynomial length, i.e., instances with BW and Par_{Exp} parameters set to 1 and 32, respectively, fit widely even on the smallest chip from the Artix-7 family. The smallest instance for polynomials with length 35339 occupies indeed 433 LUT, 187 FF, and 3 BRAM resources.

5.7 Binary polynomial inversion

This section discusses the area and the performance of the binary polynomial inversion architecture described in Section 4.3 to demonstrate its efficiency and scalability across the entire Xilinx Artix-7 family of mid-range FPGAs.

We adopted the LEDAcrypt-KEM-CPA [11] and BIKE [5] key encapsu-

Table 5.8: *Architectural parameters for hardware instances of the proposed inversion architecture on Artix-7 12 and 200 FPGAs.*

		Artix-7 12			Artix-7 200		
Code	p	BW	Par_{Exp}	Par_{DDMul}	BW	Par_{Exp}	Par_{DDMul}
$L1.4$	7187	64	1	1	64	32	3
$L1.3$	8237	64	1	1	64	64	3
$L1.2$	10883	64	1	1	64	64	3
$B1$	12323	64	1	1	64	64	3
$L3.4$	13109	64	1	1	64	32	3
$L3.3$	15373	32	16	1	64	64	3
$L3.2$	21011	64	1	1	64	64	3
$B3$	24659	64	1	1	64	64	3
$L5.4$	21611	64	1	1	64	32	3
$L5.3$	25603	64	1	1	64	64	3
$L5.2$	35339	32	1	1	64	32	3
$B5$	40973	32	1	1	64	32	3

lation mechanisms as representative use cases to demonstrate the validity of the proposed architecture, implementing the inversion module on all FPGAs of the mid-range Xilinx Artix-7 family. Table 5.8 reports all the p code parameters for BIKE and LEDAcrypt-KEM-CPA, ranging from 7187 to 40973.

The proposed architecture's performance was compared against two state-of-the-art software implementations running on an Intel Core i7 processor [69] and against a state-of-the-art hardware implementation targeting the Artix-7 FPGA family [25].

Each design instance of the proposed inversion architecture was implemented at a 133MHz operating frequency,i.e., a 7.5ns clock period. For each considered FPGA and code configuration, Table 5.8 lists the architectural paramters of the best hardware implementation, i.e., the feasible one providing the shortest execution time for a binary polynomial inversion inversion. Such instances were identified after exploring two bandwidths BW, 32 and 64 bits, three levels of multiplication parallelism Par_{DDMul}, with 1, 2, and 3 Karatsuba recursions computed in parallel, and a large set of levels of exponentiation parallelism Par_{Exp}, with values equal to the powers of 2 between 1 and BW.

The proposed architecture was compared to the reference inversion module extracted from the hardware implementation of BIKE, that targets FPGAs and is freely available online [25]. The state-of-the-art reference was implemented and simulated targeting Artix-7 FPGAs and using the same synthesis and implementation directives used for the proposed inversion

Table 5.9: *Resource utilization, timing, and performance of the reference inversion hardware instances [25].*

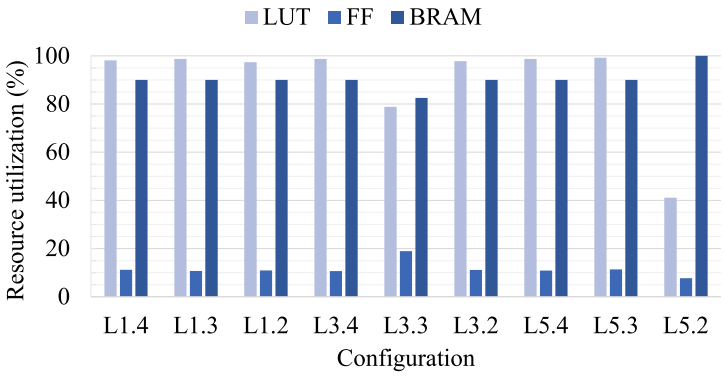
Code	BW	LUT	FF	BRAM	Freq. [MHz]	Latency [ms]
B1	32	1776	342	3	100	25.20
	64	4162	427	3	80	8.88
	128	11721	733	6	74	3.36
B3	32	1585	311	3	100	110.02
	64	4366	493	3	83	35.26
	128	12025	660	6	74	12.04

architecture. We considered only the reference instances implementing the third exponentiation strategy (see Section 3.2), since they show a lower or equal area and higher or equal performance than the other two [89]. The hardware reference implementation of BIKE is available in three bandwidths, i.e., 32, 64, and 128 bits, for the security levels 1 and 3 of BIKE, while no hardware support is available for security level 5. The proposed architecture is hence compared to the 32-, 64-, and 128-bit bandwidth configurations with exponentiation strategy 3 of the BIKE hardware implementation. The instances with 32- and 64-bit bandwidth can be instantiated on an Artix-7 12 FPGA, i.e., the smallest Artix-7 chip, while the 128-bit instances must target an Artix-7 25 or larger FPGA due to the required LUT resources. Resource utilization, maximum clock frequency, and execution time for the reference hardware instances of BIKE are detailed in Table 5.9.

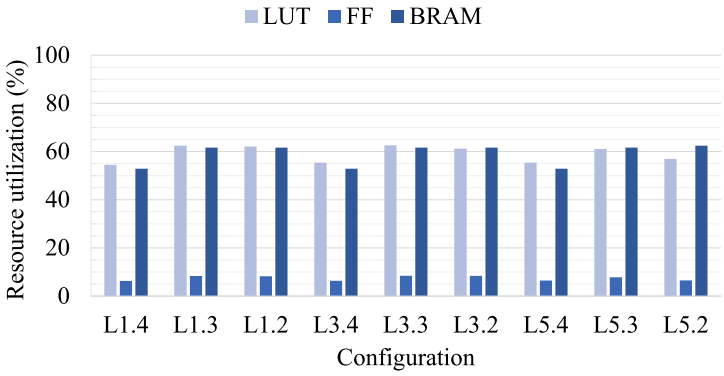
On the software side, we compared the performance against two reference software versions of binary polynomial inversion extracted from the implementation of LEDAcrypt-KEM-CPA. The `C11` version was used as the baseline reference design for performance evaluation, while the optimized software implementation employing the Intel `AVX2` extension was considered as the top-notch reference from the point of view of performance. Both are freely available online [69], and they were executed on an Intel Core i7-6700HQ desktop-class CPU running at 3.5GHz.

5.7.1 Area results

The proposed architecture makes use of the BRAMs of the FPGA as the primary means of storage, allowing the inversion module to fit on tiny FPGAs even for codes with a large block size p. In such a way, the maximum allowed dimension of the dense vectors that store the input, intermediate, and output polynomials is not a function of the available amount of flip-flops, that easily become the scarcest resources on small FPGAs, but it is



(a) *Xilinx Artix-7 12*



(b) *Xilinx Artix-7 200*

Figure 5.9: *Resource utilization of the proposed inversion architecture implemented on the Xilinx Artix-7 12 and 200 FPGAs. The utilization for each resource type is expressed as a percentage of the available resources on the target FPGA.*

instead a function of the available BRAM storage capacity. We note that a single BRAM can store up to 36kb and the smallest Artix-7 FPGA features 20 BRAMs and 16000 flip-flops, while the considered polynomial lengths range from 7187 to 40973 bits.

Figure 5.9 reports the utilization of the LUT, flip-flop, and BRAM resources as a percentage of the total available resources on the Artix-7 12 and 200 FPGAs, for polynomial lengths that suit the nine LEDAcrypt-KEM-CPA cryptosystem configurations. Look-up tables are the most used FPGA resource in smaller designs fitting on Artix-7 12 FPGAs. Indeed, most best-performing designs that are still suitable for the smallest Artix-7 FPGA require up to 99% of available LUT resources, while used BRAMs are around 90-95%. Similarly, the majority of Artix-7 200 instances show a slightly higher utilization of LUTs than BRAMs. Regardless of the dif-

ferences in used FPGA resources, all designs targeting the whole range of Artix-7 FPGAs are characterized by a wide usage of BRAMs, thus significantly minimizing the use of flip-flops. Even if the flip-flop utilization is low, it must be noted that the unused FF resources can not be exploited to further improve the design. For example, on average the FF utilization on the Artix-7 12 is below 15%, while the BRAM utilization is above 90% (see Figure 5.9a). However, an Artix-7 12 chip features 16000 FFs, thus its storage capacity is lower than a single BRAM and insufficient to store p-bit polynomials. In a similar manner, the FF utilization on Artix-7 200 is lower than 10% for each LEDAcrypt configuration. Even in such scenario, it is impossible to improve the design by leveraging the FF resources.

In contrast, we identified two main limiting factors to a higher grade of parallelism. On the dense-dense multiplier side, increasing the Par_{DDMul} parallelism, i.e., implementing parallel computation of 4 or more Karatsuba recursions, demands a number of LUTs and BRAMs that is not available on any FPGA from the Artix-7 family. On the exponentiation side, a high level of Par_{Exp} parallelism, which is nonetheless bounded by the BW bandwidth parameter, may cause timing closure at implementation time to fail, thus requiring to resort to instances with lower Par_{Exp} that work at the target 133 MHz clock frequency. As shown in Table 5.8, configurations such as $L5.4$ have a Par_{Exp} value equal to 32, while other ones such as $L5.3$ have a Par_{Exp} value equal to 64, which is the maximum allowed value. A lower exponentiation parallelism results in around 4% and 8% lower LUT and BRAM utilization, respectively, on Artix-7 200 implementations with Par_{Exp} 32 instead of 64. Notably, Par_{Exp} values comprised between 33 and 63, when coupled with a 64-bit BW bandwidth, would still require the same exact amount of clock cycles as required by a Par_{Exp} value equal to 32, hence providing no actual advantage.

5.7.2 Performance results

The performance assessment is achieved by comparing the execution time of the proposed inversion procedure to those of the software implementation of LEDAcrypt and the hardware implementation of BIKE.

In particular, Table 5.10 reports the performance results for all LEDAcrypt-KEM-CPA configurations, considering the two software references, i.e., `C11` and `AVX2`, and the two hardware instances of the proposed architecture that target the Artix-7 12 and 200 FPGAs. For example, `C11` takes between 1.80 ms and 49.95 ms to complete the inversion, with the two extremes corresponding to the $L1.4$ and $L5.2$ code configurations, respectively.

Table 5.10: *Execution times, expressed in milliseconds, of C11 and AVX2 software inversion [69] run on a i7-6700HQ CPU and of hardware instances of the proposed architecture on Artix-7 FPGAs.*

	Software [69]		Proposed architecture	
Code	**C11**	**AVX2**	**Artix-7 12**	**Artix-7 200**
$L1.4$	1.80	0.20	1.18	0.10
$L1.3$	2.53	0.24	1.49	0.11
$L1.2$	4.46	0.35	2.10	0.16
$L3.4$	6.25	0.50	2.71	0.24
$L3.3$	8.11	0.78	8.56	0.28
$L3.2$	16.79	0.95	5.73	0.44
$L5.4$	19.58	1.24	6.09	0.51
$L5.3$	22.69	1.06	7.85	0.57
$L5.2$	49.95	2.43	47.61	1.11

Figure 5.10 reports the performance speedup of the AVX2 software and the two hardware implementations, normalized with respect to the `C11` software, highlighting the actual performance improvement across the different implementations of the inversion procedure. The performance speedup of the x implementation is defined as the ratio between the execution time of the C11 software (T_{C11}) and the execution time of x (T_x), where $x \in \{\texttt{AVX2}, \texttt{Artix-7 12}, \texttt{Artix-7 200}\}$, as shown in Equation 5.1.

$$speedup_x = \frac{T_{C11}}{T_x} \tag{5.1}$$

The inversion modules targeting the low-end Artix-7 12 FPGA show an execution time comprised between 1.18 and 47.61 milliseconds, with a performance speedup between 0.95 and 3.22 (2.08 on average). Notably, the only LEDAcrypt-KEM-CPA configuration for which Artix-7 12 performance is worse than C11 performance is $L3.3$, because of the reduced bandwidth BW due to area constraints (specifically, LUTs). The optimized `AVX2` software implementation shows a performance speedup ranging between 8.9 and 21.4 (14.5 on average), while our inversion modules targeting the Artix-7 200 FPGA show a performance speedup ranging between 18.3 and 45.2 (31.7 on average), compared to the `C11` reference. Moreover, our solution overcomes the `AVX2` software implementation by 2.2 times on average, thus demonstrating the superior capability compared to optimized software solutions that exploit custom instructions offered by recent high-end Intel processors.

Figure 5.11 shows the breakdown of execution times for the macro-operations scheduled within the inversion procedure, highlighting the time

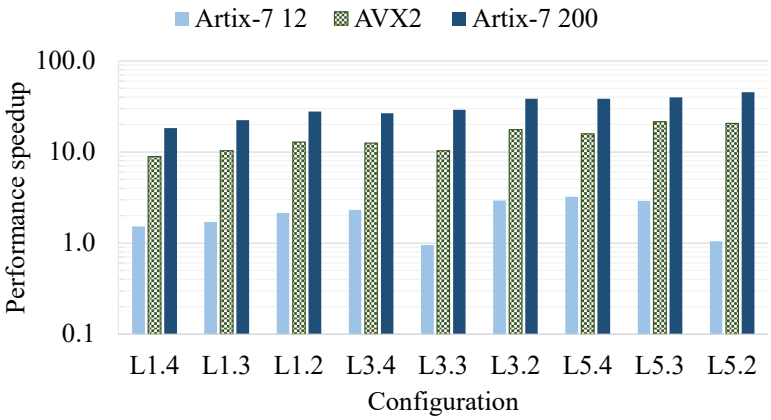


Figure 5.10: *Performance speedup with respect to C11 software inversion. Software inversion is executed on the i7-6700HQ CPU, while hardware inversion is implemented on the Artix-7 12 and 200 FPGAs.*

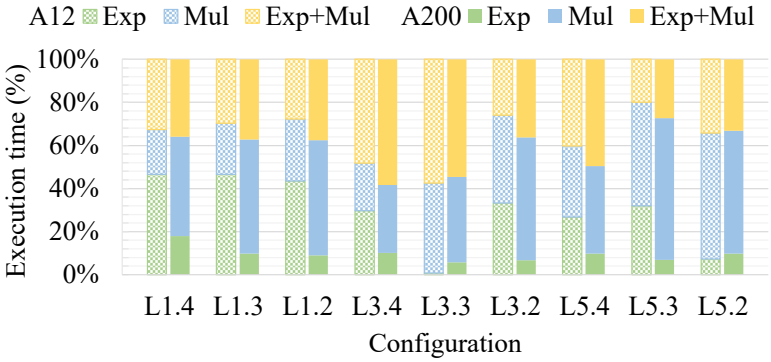


Figure 5.11: *Breakdown of the execution times of the inversion macro-operations, for instances of the proposed architecture targeting the Artix-7 12 and 200 FPGAs.*

spent computing exponentiations, multiplications, and concurrent exponentiations and multiplications. For each configuration of the LEDAcrypt code, the left and right columns specify the breakdown of the execution times for instances of the inversion targeting the Artix-7 12 and 200 FPGAs, respectively. We note that, for each reported result, the corresponding architectural parameters and performance results are reported in Table 5.8 and Table 5.10, respectively. Figure 5.11 highlights a large fraction of the execution time spent in performing the concurrent execution of the multiplication and the exponentiation. Such value is comprised between 20% and 57% (35% on average) on Artix-7 12 and between 27% and 60% (41% on average) on Artix-7 200 instances. Considering the performance benefit due to the op-

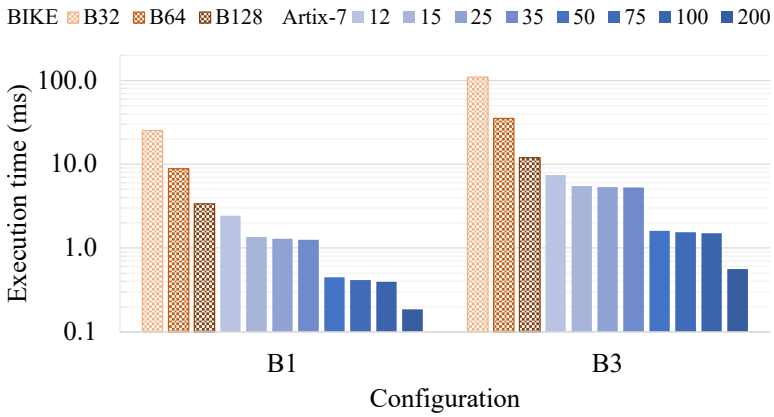


Figure 5.12: *Execution time of inversion. Results are shown for the reference hardware implementation and for instances of our inversion architecture on all Artix-7 FPGAs.*

timized hardware scheduling as well as its theoretical analysis detailed in Section 4.3.4, we note that the fraction of the execution time spent performing concurrent exponentiations and multiplications grows higher according to two factors, i.e., the Hamming weight of the binary encoding of the $(p-2)$ value and the difference in execution time between exponentiations and multiplications, which depends on the level of parallelism for each module.

The performance achieved by the proposed architecture is also compared to the BIKE reference hardware. Figure 5.12 reports the execution time to complete the inversion procedure of BIKE for polynomial lengths required to implement AES-128 and AES-192 security, i.e., $B1$ and $B3$ in Table 2.1. We note that the reference hardware does not support the BIKE configuration with AES-256 security, thus we could not compare our architecture performance with respect to the $B5$ polynomial length. Results are reported for state-of-the-art hardware accelerators with 32-, 64-, and 128-bit bandwidth, and for instances of the proposed architecture targeting each FPGA of the Artix-7 family. We remark that the reference hardware instances of BIKE with 128-bit bandwidth only fit the Artix-7 25 and larger FPGAs. In contrast, each of our inversion instances was chosen to provide the best possible performance while satisfying the resource availability of all the target FPGAs using a 133 MHz operating frequency. Compared to the BIKE reference hardware, our solution provides a speedup ranging from 1.4 to 18.1 for $B1$ and between 1.6 and 21.5 for $B3$. The minimum and maximum speedup are achieved on the Artix-7 12 and 200 FPGAs, respectively, while the other instances of our scalable architecture provide a

Table 5.11: *Architectural parameters, resources, and performance of inversion instances that target the $B3$ code.*

Artix-7	BW	Par_{Exp}	Par_{DDMul}	LUT	FF	BRAM	Latency [ms]
12	64	1	1	7954	1782	18	7.44
15	64	8	1	10180	2952	25	5.50
25	64	16	1	12568	4274	33	5.36
35	64	32	1	17291	6899	49	5.29
50	64	16	2	26787	7310	69	1.61
75	64	32	2	31547	9935	85	1.54
100	64	64	2	39319	14945	117	1.50
200	64	64	3	81928	24626	225	0.56

range of intermediate speedup values.

To further investigate the performance improvements, Table 5.11 reports the architectural parameters, resource utilization, and performance of the inversion instances on the Artix-7 FPGAs, considering the $B3$ polynomial length (see Table 2.1). The experimental results confirm that the higher time complexity of the multiplication with respect to the exponentiation (see Section 4.4.3 and Section 4.5.3) may suggest favoring the optimization of the former over the latter. For example, the execution time decreases from 1.50ms to 0.56ms by increasing the multiplication parallelism Par_{DDMul} from 2 to 3 (see lines Artix-7 100 and 200 in Table 5.11). However, results in Table 5.11 also highlight the critical contribution to the overall performance of inversion given by optimizing the exponentiation component. For instance, the execution time drops from 7.44ms to 5.29ms by increasing the exponentiation parallelism Par_{Exp} from 1 to 32 (see lines Artix-7 12 and 35 in Table 5.11).

5.8 Dense-sparse binary polynomial multiplication

This section briefly discusses the resource utilization and performance of the dense-sparse multiplication architecture described in Section 4.6. Table 5.12 lists the area and performance metrics for instances of the dense-sparse multiplication module targeting polynomials with length p values of 12323 and 24659, as in the SL1 and SL3 instances of BIKE, respectively (see $B1$ and $B3$ in Table 2.1). In addition, the SL1 case considers two different Hamming weights HW of the sparse operand, which correspond to the v and t parameters of the BIKE code. The experimental results are provided for different values of the configurable architectural parameters BW and Par_{DSMul} to highlight how their variations impact the area and performance

Table 5.12: *Resource utilization and performance of the proposed dense-sparse multiplication architecture for different configurations of code and architectural parameters.*

Code parameters		Arch. parameters					
p	HW	BW	Par_{DSMul}	**LUT**	**FF**	**BRAM**	**Latency [us]**
12323	71	32	1	435	237	1.5	278.34
			4	1442	782	5.0	70.58
			16	5274	2976	19.5	19.62
		64	1	643	294	2.5	141.31
			16	8604	3935	35.5	9.97
			64	34235	15608	140.5	4.00
		128	64	73480	27998	268.5	2.08
	134	32	16	5274	2976	19.5	35.30
		64	16	8604	3935	35.5	17.93
			64	34235	15608	140.5	5.99
		128	64	73480	27998	268.5	3.11
24659	103	32	16	6378	2865	35.5	54.41
		64	16	8550	4306	35.5	27.46
			64	35044	17098	141.5	7.86
		128	64	72346	25376	269.5	4.00

metrics. The listed hardware instances are synthesized and implemented on Artix-7 FPGAs targeting a 100MHz clock frequency, i.e., a 10ns period.

Results in Table 5.12 highlight the impact of the BW and Par_{DSMul} architectural parameters on the resource utilization and latency, with the former increasing and the latter reducing, respectively, in a linear way as the two parameters grow larger. Moreover, an increase in the Hamming weight HW of the sparse operand is shown to result in a linearly proportional increase in the multiplication latency, whereas the area occupation is unmodified. Finally, the p code parameter also demonstrates a linear increase impact on the execution time of the operation. Notably, the specific inputs to the dense-sparse multiplication, i.e., both the dense and sparse operands, do not impact instead in any way the execution time of a multiplication, which is computed in a constant time for a given set of p, HW, BW, and Par_{DSMul} parameters.

5.9 KEM primitives

The complexity-oriented heuristic detailed in Section 4.8 drove the design space exploration and led to the identification of the configurable parameters for the various client and server core instances. The parameters identified from the design space exploration are listed in Table 5.13. They refer to

Table 5.13: *Architectural parameters for the client and server instances of the proposed architecture identified through the complexity-based heuristic. − denotes non-instantiated components. Refer to Table 2.1 for the related BIKE code parameters.*

	Target		Architectural parameters				
Core	**Artix-7**	**Code**	BW	Par_{Dec}	Par_{DSMul}	Par_{DDMul}	Par_{Exp}
Client	**50**	*B1*	64	2	2	1	4
		B3	64	2	2	1	2
	200	*B1*	64	24	12	3	32
		B3	64	20	12	3	32
Server	**35**	*B1*	64	–	12	–	–
		B3	64	–	10	–	–
	200	*B1*	64	–	136	–	–
		B3	64	–	104	–	–

both client and server cores implementing NIST security level 1 and 3 BIKE instances on the smallest feasible and largest Artix-7 FPGAs.

The rest of this section discusses the area and performance of such eight proposed implementations while also comparing them to the state-of-the-art reference ones.

5.9.1 Area results

The proposed architecture makes extensive use of the FPGA BRAM for storage purposes, allowing the cryptographic core to fit on smaller FPGAs even for codes with a large block size p. Flip-flops would otherwise quickly become the scarcest resources on small FPGAs, due to the need to store multiple p-bit vectors, where p ranges between 12323 and 24659. Indeed, the smallest considered FPGA, i.e., Artix-7 35, features just 41600 flip-flops while, instead, packing 50 36kb BRAM memories, that can store overall up to 1843200 bits.

However, we identified a few factors that concurred to limit the maximum degree of parallelism. For the multiplier component `Mul`, the Par_{DSMul} parallelism is bounded by the values of v and t. Concerning the inversion component `Inv`, increasing the Par_{DDMul} parallelism over 3, i.e., implementing parallel computation of 4 or more Karatsuba recursions within the multiplier functional unit, requires a number of LUTs and BRAMs that is not available on any FPGA from the Artix-7 family. The Par_{Exp} parallelism is instead bounded by the value of the bandwidth BW. Finally, the degree of parallelism Par_{Exp} of the decoding component `Dec` is limited by the imposed timing constraint of a 91MHz clock frequency.

The resource utilization, in terms of LUTs, flip-flops, and blocks of

Table 5.14: *Area results for the proposed client cores, expressed in terms of look-up tables (LUT), flip-flops (FF), and block RAM (BRAM), and relative resource utilization, expressed as a percentage within round brackets.*

	Artix-7 50			Artix-7 200		
Code	**LUT**	**FF**	**BRAM**	**LUT**	**FF**	**BRAM**
B1	31792	17805	43.5	126510	51492	357
	(98%)	(27%)	(58%)	(94%)	(19%)	(98%)
B3	31411	20181	45.5	124891	53067	360
	(96%)	(31%)	(61%)	(93%)	(20%)	(99%)
Available	32600	65200	75	134600	269200	365

Table 5.15: *Area results for the proposed server cores, expressed in terms of look-up tables (LUT), flip-flops (FF), and block RAM (BRAM), and relative resource utilization, expressed as a percentage within round brackets.*

	Artix-7 35			Artix-7 200		
Code	**LUT**	**FF**	**BRAM**	**LUT**	**FF**	**BRAM**
B1	19804	11401	30	91422	46208	275.5
	(95%)	(27%)	(60%)	(68%)	(17%)	(75%)
B3	19979	12282	28	72725	37795	235.5
	(96%)	(30%)	(56%)	(54%)	(14%)	(65%)
Available	20800	41600	50	134600	269200	365

BRAM, for the instances targeting the Artix-7 35 and 200 FPGAs of the proposed client and server architectures is detailed in Table 5.14 and Table 5.15, respectively.

The reported results demonstrate how the proposed cryptographic cores can scale from the smaller Artix-7 35 FPGA up to the larger Artix-7 200 FPGA. Moreover, they show that BRAM memories are the most used resources, relatively to the ones available on the target chip, on the larger Artix-7 200 FPGAs, while instances targeting the smaller chips are bounded by the LUT utilization. The proposed architectures usually employ a large fraction of the available look-up tables, while requiring a smaller fraction of flip-flops.

On the contrary, as shown in Table 5.16, the state-of-the-art implementation [89] chosen as the hardware reference can not effectively use all the resources available on larger FPGAs, since the high-speed instance employs only 32%, 2%, and 11% of the LUT, FF, and BRAM resources available on the largest Artix-7 FPGA, respectively.

Table 5.16: *Area results for the reference hardware implementation [89] split into client and server cores, expressed in terms of look-up tables (LUT), flip-flops (FF), and block RAM (BRAM), and relative resource utilization, expressed as a percentage within round brackets.*

		Lightweight (LW) [89]			High-speed (HS) [89]		
Core	**Code**	**LUT**	**FF**	**BRAM**	**LUT**	**FF**	**BRAM**
Client	*B1*	11454 (55%)	4602 (11%)	14 (28%)	43084 (32%)	610 (2%)	39 (11%)
	B3	–	–	–	–	–	–
Server	*B1*	6730 (32%)	3298 (8%)	3 (6%)	14829 (6%)	3471 (1%)	10 (3%)
	B3	–	–	–	–	–	–
	Available	20800	41600	50	134600	269200	365

5.9.2 Performance results

The performance of the proposed architectures is assessed by comparing the execution times of client-side and server-side computations to those of the C99 and AVX2 reference software and hardware implementations of BIKE. To better evaluate the performance compared to software execution, we define the speedup as the ratio between the execution time of the AVX2 software and the one resulting from the execution on a specific software or hardware instance. A speedup value greater than 1 indicates a performance improvement over the AVX2 software while a value below 1 corresponds to worse performance.

Table 5.17 reports the performance results for the two software references, i.e., C99 and AVX2, the instances of the proposed client architecture that target the Artix-7 50 and 200 FPGAs, and the lightweight and high-speed instances of the reference hardware implementations [88, 89]. Performance data are reported as the execution time, expressed in milliseconds, and as the corresponding speedup over AVX2 software execution, reported within round brackets. The instance of the proposed client architecture targeting the Artix-7 50 FPGA provides a hardware support that is around six times slower than the reference AVX2 software execution, as shown by the speedup of $0.18\times$ for the *B1* and *B3* BIKE configurations. On the contrary, instantiating the client module on the Artix-7 200 FPGA results in a significant performance improvement over the AVX2 reference, with speedups of $1.78\times$ for *B1* and $1.91\times$ for *B3*. Referring to the $B1$ use case in the client scenario, our Artix-7 50 implementation is around six times

Table 5.17: *Client-side execution times, expressed in milliseconds, and speedup over AVX2 software.*

	CPU [5]		FPGA Our		FPGA [89]		FPGA [88]	
Code	**C99**	**AVX2**	**A7 50**	**A7 200**	**LW**	**HS**	**LW**	**HS**
B1	8.56	1.03	5.71	0.58	35.25	4.66	10.69	3.56
	0.12×		0.18×	1.78×	0.03×	0.22×	0.10×	0.29×
B3	27.65	3.40	19.27	1.71	–	–	35.75	11.32
	0.12×		0.18×	1.91×			0.09×	0.30×

Table 5.18: *Server-side execution times, expressed in milliseconds, and speedup over AVX2 software.*

	CPU [5]		FPGA Our		FPGA [89]		FPGA [88]	
Code	**C99**	**AVX2**	**A7 35**	**A7 200**	**LW**	**HS**	**LW**	**HS**
B1	0.28	0.05	0.03	0.03	1.25	0.13	0.44	0.13
	0.18×		1.70×	1.70×	0.04×	0.38×	0.11×	0.38×
B3	0.92	0.11	0.08	0.06	–	–	1.35	0.37
	0.12×		1.38×	1.83×			0.08×	0.30×

faster than the lightweight instance of [89], while our Artix-7 200 client implementation is around eight times faster than the high-speed instance of [89]. The proposed Artix-7 50 and Artix-7 200 client modules also outperform the state-of-the-art hardware implementations in [88], as shown in Table 5.17. Considering both security level 1 and 3 instances of BIKE, the lightweight and high-speed instances of [88] provide indeed speedups up to 0.1× and 0.3×, compared to the speedups as low as 0.18× and 1.78× of the smallest and largest instances of the proposed architecture.

Table 5.18 provides instead performance data for the server-side hardware support. In the server scenario, both the Artix-7 35 and Artix-7 200 instances improve performance over AVX2. Artix-7 35 provides speedups of 1.70× and 1.38× for *B1* and *B3*, while Artix-7 200 is 1.70× and 1.83× faster than AVX2, respectively. Moreover, both the Artix-7 35 and 200 implementations of our server architectures outperform the high-speed instances of both state-of-the-art [89] and [88] hardware implementations. As shown in Table 5.18, considering both security level 1 and 3 instances of BIKE, the lightweight and high-speed instances of [88] provide indeed speedups up to 0.11× and 0.38×, compared to the speedups as low as 1.38× and 1.70× of the smallest and largest instances of the proposed architecture.

CHAPTER 6

Conclusions

The definition of novel cryptography solutions is paramount in the wake of the advancements of quantum computing. The threat posed by quantum computers, which are expected to make traditional public-key cryptography solutions obsolete, is driving the research effort in the field of post-quantum computing.

BIKE is a post-quantum KEM that is a candidate for standardization within the NIST post-quantum cryptography standardization process. Newly-defined standards for cryptography must not only provide the desired security against both traditional and quantum attacks, but they must also provide a performance that enables a satisfactory quality of service when deployed in real-world use cases. While there already exist hardware solutions that can support QC-MDPC code-based cryptography, most of them are tailored to codes with dimensions in the order of hundreds of bits, while others either do not provide sufficient performance or do not make an efficient use of all the resources available on the target FPGA chips.

This thesis presented a configurable FPGA-based hardware architecture that implements the BIKE QC-MDPC code-based cryptosystem, with the aim of improving performance over the existing state-of-the-art software and hardware solutions. The proposed architecture provides an effective

FPGA-based hardware support for QC-MDPC codes that are suitable to post-quantum cryptography applications. It exploits a set of configurable code and architectural parameters that allow using a single design to support different QC-MDPC codes underlying the PQC cryptosystems and to target any FPGA chip from the Xilinx Artix-7 family. Different area-performance trade-offs can be explored through the parametric configurability to satisfy the performance requirements and the area constraints set for the overall system that integrates the BIKE hardware support.

The proposed architecture implements two modules that support the KEM primitives to be executed on the client and server nodes of the key exchange, repsectively. The experimental evaluation carried out on the proposed architecture highlight significant improvements over the state-of-the-art software and hardware implementations of BIKE from the literature.

On the one hand, compared to the reference software implementation, which exploits the Intel AVX2 extension on desktop-class CPUs, the client and server instances provide a performance speedup up to 1.91× and 1.83×, respectively. In addition, the instances of the proposed architecture targeting the smaller Artix-7 50 FPGAs chips still outperform the software execution of baseline, non-AVX2-optimized code on the same CPU.

On the other hand, the proposed FPGA-based BIKE architecture also outperforms the other hardware implementations available from literature, including both HLS-generated and human-designed ones, and provides a speedup over the fastest state-of-the-art FPGA-based instance of up to 6×.

Future extensions of the work presented in this thesis include the development of an ISA extension that allows an effective acceleration of QC-MDPC code-based post-quantum cryptography on RISC-V CPUs.

APPENDIX $\mathcal{A}$

List of publications

This appendix lists in detail the scientific publications resulting from the research carried out during the PhD period.

Section A.1 lists the main papers related to this thesis, whose content was presented in the previous chapters. The author of this thesis contributed to the papers listed in Section A.1 for both the theoretical and experimental parts, also in works where he was not the first author. Section A.2 lists works covering other topics and resulting from the collaboration with other researchers.

A.1 Main publications

- **Title**: Flexible and Scalable FPGA-Oriented Design of Multipliers for Large Binary Polynomials.
  Authors: Davide Zoni, Andrea Galimberti, and William Fornaciari.
  Journal: IEEE Access.
  Year: 2020.
  Volume: 8.
  Pages: 75809–75821.
  DOI: 10.1109/ACCESS.2020.2989423.
  Cited as: [111].
  Personal contribution: Contributed to the implementation and verification. Also contributed to the writing of the paper.

- **Title**: Efficient and Scalable FPGA-Oriented Design of QC-LDPC Bit-Flipping Decoders for Post-Quantum Cryptography.
  Authors: Davide Zoni, Andrea Galimberti, and William Fornaciari.
  Journal: IEEE Access.
  Year: 2020.
  Volume: 8.
  Pages: 163419–163433.
  DOI: 10.1109/ACCESS.2020.3020262.
  Cited as: [110].
  Personal contribution: Contributed to the implementation and verification. Also contributed to the writing of the paper.

- **Title**: Efficient and scalable FPGA design of GF(2m) inversion for post-quantum cryptosystems.
  Authors: Andrea Galimberti, Gabriele Montanaro, and Davide Zoni.
  Journal: IEEE Transactions on Computers.
  Year: 2022.
  DOI: 10.1109/TC.2022.3149422.
  Cited as: [43].
  Personal contribution: Design and implementation leader. Also contributed to the writing of the paper.

- **Title**: On the Use of Hardware Accelerators in QC-MDPC Code-Based Cryptography.
  Authors: Andrea Galimberti, Davide Galli, Gabriele Montanaro, William Fornaciari, and Davide Zoni.
  Book title: Proceedings of the 19th ACM International Conference on Computing Frontiers.
  Year: 2022.
  Pages: 193–194.
  DOI: 10.1145/3528416.3530243.
  Cited as: [42].
  Personal contribution: Contributed to the design, implementation, and verification. Also contributed to the writing of the paper.

- **Title**: FPGA implementation of BIKE for quantum-resistant TLS.
  Authors: Andrea Galimberti, Davide Galli, Gabriele Montanaro, William Fornaciari, and Davide Zoni.
  Book title: 2022 25th Euromicro Conference on Digital System Design (DSD).
  Year: 2022.
  Pages: 539–547.
  Cited as: [41].
  Personal contribution: Contributed to the design, implementation, and verification. Also contributed to the writing of the paper.

- **Title**: Hardware-Software Co-Design of BIKE with HLS-Generated Accelerators.
  Authors: Gabriele Montanaro, Andrea Galimberti, Ernesto Colizzi, and Davide Zoni.
  Book title: 2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS).
  Year: 2022.
  Pages: 1-4.
  DOI: 10.1109/ICECS202256217.2022.9970992.
  Cited as: [74].
  Personal contribution: Contributed to the design, implementation, and verification. Also contributed to the writing of the paper.

A.2 Other publications

- **Title**: VGM-Bench: FPU Benchmark Suite for Computer Vision, Computer Graphics and Machine Learning Applications.
  Authors: Luca Cremona, Wiliam Fornaciari, Andrea Galimberti, Andrea Romanoni, and Davide Zoni.
  Book title: Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings.
  Year: 2020.
  Pages: 323–335.
  DOI: 10.1007/978-3-030-60939-9_23.
  Personal contribution: Contributed to the verification of the project.

- **Title**: An FPU design template to optimize the accuracy-efficiency-area trade-off.
  Authors: Davide Zoni, Andrea Galimberti, and William Fornaciari.
  Journal: Sustainable Computing: Informatics and Systems.
  Volume: 29.
  Pages: 100450.
  Year: 2021.
  DOI: 10.1016/j.suscom.2020.100450.
  Personal contribution: Contributed to the design, implementation, and verification. Also contributed to the writing of the paper.

- **Title**: TEXTAROSSA: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale.
  Authors: Giovanni Agosta, Daniele Cattaneo, Wiliam Fornaciari, Andrea Galimberti et al.
  Book title: 2021 24th Euromicro Conference on Digital System Design (DSD).
  Year: 2021.
  Pages: 286–294.
  DOI: 10.1109/DSD53832.2021.00051
  Personal contribution: Team member in the TEXTAROSSA EU project on the power monitoring topic.

- **Title**: Cost-effective fixed-point hardware support for RISC-V embedded systems.
  Authors: Davide Zoni and Andrea Galimberti.
  Journal: Journal of Systems Architecture.
  Volume: 126.
  Pages: 102476.
  Year: 2022.
  DOI: 10.1016/j.sysarc.2022.102476.
  Personal contribution: Contributed to the implementation and verification. Also contributed to the writing of the paper.

- **Title**: On the Effectiveness of True Random Number Generators Implemented on FPGAs.
  Authors: Davide Galli, Andrea Galimberti, Wiliam Fornaciari, and Davide Zoni.
  Book title: Embedded Computer Systems: Architectures, Modeling, and Simulation: 22nd International Conference, SAMOS 2022, Samos, Greece, July 3-7, 2022, Proceedings.
  Year: 2022.
  Pages: 315–326.
  DOI: 10.1007/978-3-031-15074-6_20.
  Personal contribution: Contributed to the writing of the paper.

- **Title**: Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale: the TEXTAROSSA Approach.
  Authors: Giovanni Agosta, . . . , Andrea Galimberti et al.
  Journal: Microprocessors and Microsystems.
  Year: 2022.
  DOI: 10.1016/j.micpro.2022.104679.
  Personal contribution: Team member in the TEXTAROSSA EU project on the power monitoring topic.

List of Figures

List of Tables

Bibliography

[1] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérôme Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. HQC website. `https://pqc-hqc.org/`, 2017.

[2] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece website. `https://classic.mceliece.org/`, 2017.

[3] Amazon Web Services - Labs. Additional implementation of bike (bit flipping key encapsulation). `https://github.com/awslabs/bike-kem`, 2020.

[4] 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.

[5] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE website. `https://www.bikesuite.org/`, 2017.

[6] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit flipping key encapsulation - round 3 submission. `https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf`, 2021.

[7] Juan M Arrazola, Ville Bergholm, Kamil Brádler, Thomas R Bromley, Matt J Collins, Ish Dhand, Alberto Fumagalli, Thomas Gerrits, Andrey Goussev, Lukas G Helt, et al. Quantum circuits with many photons on a programmable nanophotonic chip. *Nature*, 591(7848):54–60, 2021.

[8] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[9] R. Azarderakhsh and A. Reyhani-Masoleh. Low-complexity multiplier architectures for single and hybrid-double multiplications in gaussian normal bases. *IEEE Transactions on Computers*, 62(4):744–757, April 2013.

[10] Marco Baldi. *QC-LDPC code-based cryptography*. Springer Science & Business, 2014.

[11] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt website. `https://www.ledacrypt.org/`, 2017.

[12] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt: Low-density parity-check code-based cryptographic systems. `https://www.ledacrypt.org/documents/LEDAcrypt_v3.pdf`, 2020.

[13] Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi, and Davide Zoni. Evaluating the trade-offs in the hardware design of the ledacrypt encryption functions. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 739–742, 2019.

[14] Alessandro Barenghi and Gerardo Pelosi. A comprehensive analysis of constant-time polynomial inversion for post-quantum cryptosystems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, pages 269–276, New York, NY, USA, 2020. Association for Computing Machinery.

[15] E. Berlekamp. Goppa codes. *IEEE Transactions on Information Theory*, 19(5):590–592, 1973.

[16] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[17] Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.

[18] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, May 2019.

[19] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. `https://keccak.team/obsolete/Keccak-implementation-3.1.pdf`, 2011.

[20] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. Crystals - kyber: A cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, 2018.

[21] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. gf2x website. `http://gf2x.gforge.inria.fr/`.

[22] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in gf(2)[x]. In Alfred J. van der Poorten and Andreas Stein, editors, *Algorithmic Number Theory*, pages 153–166, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[23] H. Brunner, A. Curiger, and M. Hofstetter. On computing multiplicative inverses in gf(2/sup m/). *IEEE Transactions on Computers*, 42(8):1010–1015, 1993.

[24] Wouter Castryck and Thomas Decru. An efficient key recovery attack on sidh (preliminary version). Cryptology ePrint Archive, Paper 2022/975, 2022. `https://eprint.iacr.org/2022/975`.

[25] Chair for Security Engineering @ Ruhr-Universität Bochum. Folding bike: Scalable hardware implementation for reconfigurable devices. `https://github.com/Chair-for-Security-Engineering/BIKE`, 2021.

[26] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing bike for the intel haswell and arm cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):97–124, Jul. 2021.

[27] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to bike faster. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security*, pages 833–852, Cham, 2022. Springer International Publishing.

[28] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. Experimental implementation of fast quantum searching. *Phys. Rev. Lett.*, 80:3408–3411, Apr 1998.

[29] A. Cilardo. Fast Parallel GF(2^m) Polynomial Multiplication for All Degrees. *IEEE Transactions on Computers*, 62(5):929–943, May 2013.

[30] P. G. Comba. Exponentiation cryptosystems on the ibm pc. *IBM Systems Journal*, 29(4):526–538, 1990.

[31] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[32] N. Drucker, S. Gueron, and V. Krasnov. Fast multiplication of binary polynomials with the forthcoming vectorized vpclmulqdq instruction. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 115–119, June 2018.

[33] Nir Drucker, Shay Gueron, and Dusan Kostic. Fast polynomial inversion for post quantum qc-mdpc cryptography. In Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss, editors, *Cyber Security Cryptography and Machine Learning*, pages 110–127, Cham, 2020. Springer International Publishing.

[34] Nir Drucker, Shay Gueron, and Dusan Kostic. Qc-mdpc decoders with several shades of gray. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 35–50, Cham, 2020. Springer International Publishing.

[35] Morris Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[36] Haining Fan and M. Anwar Hasan. A survey of some recent bit-parallel gf(2n) multipliers. *Finite Fields and Their Applications*, 32:5–43, 2015. Special Issue : Second Decade of FFA.

[37] G.-L. Feng. A vlsi architecture for fast inversion in gf(2/sup m/). *IEEE Transactions on Computers*, 38(10):1383–1386, 1989.

[38] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014.

[39] Richard P Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, 1982.

[40] M. G. Find and R. Peralta. Better circuits for binary polynomial multiplication. *IEEE Transactions on Computers*, 68(4):624–630, April 2019.

[41] Andrea Galimberti, Davide Galli, Gabriele Montanaro, William Fornaciari, and Davide Zoni. Fpga implementation of bike for quantum-resistant tls. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 539–547, 2022.

[42] Andrea Galimberti, Davide Galli, Gabriele Montanaro, William Fornaciari, and Davide Zoni. On the use of hardware accelerators in qc-mdpc code-based cryptography. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, page 193–194, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Andrea Galimberti, Gabriele Montanaro, and Davide Zoni. Efficient and scalable fpga design of gf(2m) inversion for post-quantum cryptosystems. *IEEE Transactions on Computers*, pages 1–1, 2022.

[44] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[45] Pierrick Gaudry, Luc Sanselme, and Emmanuel Thomé. Mpfq - a finite field library. http://mpfq.gforge.inria.fr.

[46] J. Grossschadl. A low-power bit-serial multiplier for finite fields gf(2/sup m/). In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 4, pages 37–40 vol. 4, May 2001.

[47] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[48] Shay Gueron and Michael E Kounavis. Intel® carry-less multiplication instruction and its usage for computing the gcm mode. *Intel White Paper*, pages 1–76, 2010.

[49] Jyh-Huei Guo and Chin-Liang Wang. Systolic array implementation of euclid's algorithm for inversion and division in gf(2/sup m/). *IEEE Transactions on Computers*, 47(10):1161–1167, 1998.

[50] Sindhu Hak Gupta and Bindya Virmani. Ldpc for wi-fi and wimax technologies. In *2009 International Conference on Emerging Trends in Electronic and Photonic Devices & Systems*, pages 262–265, 2009.

[51] Stefan Heyse, Ingo von Maurich, and Tim Güneysu. Smaller keys for code-based cryptography: Qc-mdpc mceliece implementations on embedded devices. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 273–292, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[52] Jeremy Hsu. Ces 2018: Intel's 49-qubit chip shoots for quantum supremacy. *IEEE Spectrum Tech Talk*, pages 1–6, 2018.

[53] J. Hu, M. Baldi, P. Santini, N. Zeng, S. Ling, and H. Wang. Lightweight key encapsulation using ldpc codes on fpgas. *IEEE Transactions on Computers*, 69(3):327–341, 2020.

[54] Jingwei Hu and Ray C.C. Cheung. Area-time efficient computation of niederreiter encryption on qc-mdpc codes for embedded hardware. *IEEE Transactions on Computers*, 66(8):1313–1325, 2017.

[55] Jingwei Hu, Wei Guo, Jizeng Wei, and Ray C. C. Cheung. Fast and generic inversion architectures over GF(2^m) using modified itoh-tsujii algorithms. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):367–371, 2015.

[56] Jingwei Hu, Wen Wang, Ray C.C. Cheung, and Huaxiong Wang. Optimized polynomial multiplier over commutative rings on fpgas: A case study on bike. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 231–234, 2019.

[57] IBM. Ibm quantum processor types. `https://quantum-computing.ibm.com/services/resources/docs/resources/manage/systems/processors`, 2017.

[58] IBM. Ibm's roadmap for scaling quantum technology. `https://research.ibm.com/blog/ibm-quantum-roadmap`, 2020.

[59] Mohamed Ismail, Imran Ahmed, Justin Coon, Simon Armour, Taskin Kocak, and Joseph McGeehan. Low latency low power bit flipping algorithms for ldpc decoding. In *21st Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 278–282, 2010.

[60] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases. *Information and Computation*, 78(3):171–177, 1988.

[61] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. SIKE website. `https://sike.org/`, 2017.

[62] Nan Jiang, Kewu Peng, Jian Song, Chanyong Pan, and Zhixing Yang. High-throughput qc-ldpc decoders. *IEEE Transactions on Broadcasting*, 55(2):251–259, 2009.

[63] Juntan Zhang and M. P. C. Fossorier. A modified weighted bit-flipping decoding of low-density parity-check codes. *IEEE Communications Letters*, 8(3):165–167, 2004.

[64] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

[65] Emanuel Knill, R Laflamme, R Martinez, and C-H Tseng. An algorithmic benchmark for quantum information processing. *Nature*, 404(6776):368–370, 2000.

[66] K. Kobayashi, N. Takagi, and K. Takagi. Fast inversion algorithm in gf(2m) suitable for implementation with a polynomial multiply instruction on gf(2). *IET Comput. Digit. Tech.*, 6:180–185, 2012.

[67] 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.

[68] Y. Kou, S. Lin, and M. P. C. Fossorier. Low-density parity-check codes based on finite geometries: a rediscovery and new results. *IEEE Transactions on Information Theory*, 47(7):2711–2736, 2001.

[69] LEDAcrypt. Ledacrypt. `https://github.com/LEDAcrypt/LEDAcrypt`, 2021.

[70] Lijuan Li and Shuguo Li. Fast inversion in gf(2m) with polynomial basis using optimal addition chains. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[71] Lars S Madsen, Fabian Laudenbach, Mohsen Falamarzi Askarani, Fabien Rortais, Trevor Vincent, Jacob FF Bulmer, Filippo M Miatto, Leonhard Neuhaus, Lukas G Helt, Matthew J Collins, et al. Quantum computational advantage with a programmable photonic processor. *Nature*, 606(7912):75–81, 2022.

[72] R. J. McEliece. A Public-Key Cryptosystem Based on Algebraic Coding Theory. *DSN Progress Report*, pages 114–116, 1978.

[73] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.

[74] Gabriele Montanaro, Andrea Galimberti, Ernesto Colizzi, and Davide Zoni. Hardware-software co-design of bike with hls-generated accelerators. In *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2022.

[75] National Institute of Standards and Technology (NIST) - U.S. Department of Commerce. Nistir 8309, status report on the second round of the nist post-quantum cryptography standardization process. `https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf`, 2020.

[76] National Institute of Standards and Technology (NIST) - U.S. Department of Commerce. Post-quantum cryptography. `https://csrc.nist.gov/projects/post-quantum-cryptography`, 2021.

[77] National Institute of Standards and Technology (NIST) - U.S. Department of Commerce. Nistir 8413, status report on the third round of the nist post-quantum cryptography standardization process. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf, 2022.

[78] National Security Agency. Commercial national security algorithm suite 2.0 (cnsa 2.0) cybersecurity advisory (csa). https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF, 2022.

[79] C. Negrevergne, T. S. Mahesh, C. A. Ryan, M. Ditty, F. Cyr-Racine, W. Power, N. Boulant, T. Havel, D. G. Cory, and R. Laflamme. Benchmarking quantum control methods on a 12-qubit system. *Phys. Rev. Lett.*, 96:170501, May 2006.

[80] Hamid Nejatollahi, Nikil Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. Post-quantum lattice-based cryptography implementations: A survey. *ACM Comput. Surv.*, 51(6), jan 2019.

[81] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Contr. Inform. Theory*, 15(2):157–166, 1986.

[82] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[83] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: Extended abstract. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 333–342, New York, NY, USA, 2009. Association for Computing Machinery.

[84] B. Rashidi. Throughput/area efficient implementation of scalable polynomial basis multiplication. In *Journal of Hardware and Systems Security*, Jan 2020.

[85] B. Rashidi, S. M. Sayedi, and R. Rezaeian Farashahi. Efficient and low-complexity hardware architecture of Gaussian normal basis multiplication over GF(2m) for elliptic curve cryptosystems. *IET Circuits, Devices Systems*, 11(2):103–112, 2017.

[86] Chester Rebeiro, Sujoy Sinha Roy, D. Sankara Reddy, and Debdeep Mukhopadhyay. Revisiting the itoh-tsujii inversion algorithm for fpga platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(8):1508–1512, 2011.

[87] Eric Rescorla et al. Rfc 8446: The transport layer security (tls) protocol version 1.3. *Internet Engineering Task Force (IETF)*, 25, 2018.

[88] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing bike: Improved polynomial multiplication and inversion in hardware. Cryptology ePrint Archive, Paper 2021/1344, 2021. https://eprint.iacr.org/2021/1344.

[89] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding bike: Scalable hardware implementation for reconfigurable devices. *IEEE Transactions on Computers*, 2021.

[90] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[91] F. Rodriguez-Henriquez, N. Cruz-Cortes, and N.A. Saqib. A fast implementation of multiplicative inversion over gf(2/sup m/). In *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, volume 1, pages 574–579 Vol. 1, 2005.

[92] Francisco Rodríguez-Henríquez, Guillermo Morales-Luna, Nazar A Saqib, and Nareli Cruz-Cortés. Parallel itoh–tsujii multiplicative inversion algorithm for a special class of trinomials. *Designs, Codes and Cryptography*, 45(1):19–37, 2007.

[93] N. Sendrier. Code-based cryptography: State of the art and perspectives. *IEEE Security Privacy*, 15(4):44–50, 2017.

[94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[95] Victor Shoup. Ntl: A library for doing number theory. https://www.shoup.net/ntl/.

[96] Victor Shoup. A proposal for an iso standard for public key encryption. Cryptology ePrint Archive, Paper 2001/112, 2001. `https://eprint.iacr.org/2001/112`.

[97] N. Takagi, J. Yoshiki, and K. Takagi. A fast algorithm for multiplicative inversion in gf(2/sup m/) using normal basis. *IEEE Transactions on Computers*, 50(5):394–398, 2001.

[98] Ioannis Tsatsaragkos and Vassilis Paliouras. A reconfigurable ldpc decoder optimized for 802.11n/ac applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(1):182–195, 2018.

[99] Sean Turner. Transport layer security. *IEEE Internet Computing*, 18(6):60–63, 2014.

[100] Ingo von Maurich and Tim Güneysu. Lightweight code-based cryptography: Qc-mdpc mceliece encryption on reconfigurable devices. In *2014 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2014.

[101] Joachim von zur Gathen and Jamshid Shokrollahi. Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F_2. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*, pages 359–369, 2005.

[102] Tadashi Wadayama, Keisuke Nakamura, Masayuki Yagita, Yuuki Funahashi, Shogo Usami, and Ichi Takumi. Gradient descent bit flipping algorithms for decoding ldpc codes. *IEEE Transactions on Communications*, 58(6):1610–1614, 2010.

[103] Wang, Troung, Shao, Deutsch, Omura, and Reed. Vlsi architectures for computing multiplications and inverses in gf(2m). *IEEE Transactions on Computers*, C-34(8):709–717, 1985.

[104] Guohui Wang, Michael Wu, Yang Sun, and Joseph R. Cavallaro. A massively parallel implementation of qc-ldpc decoder on gpu. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 82–85, 2011.

[105] Tatu Ylonen. Ssh–secure login connections over the internet. In *Proceedings of the 6th USENIX Security Symposium*, volume 37, 1996.

[106] Zhengya Zhang, Venkat Anantharam, Martin J. Wainwright, and Borivoje Nikolic. An efficient 10gbase-t ethernet ldpc decoder design with low error floors. *IEEE Journal of Solid-State Circuits*, 45(4):843–855, 2010.

[107] M. Zhao, X. Zhang, L. Zhao, and C. Lee. Design of a high-throughput qc-ldpc decoder with tdmp scheduling. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(1):56–60, 2015.

[108] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, Peng Hu, Xiao-Yan Yang, Wei-Jun Zhang, Hao Li, Yuxuan Li, Xiao Jiang, Lin Gan, Guangwen Yang, Lixing You, Zhen Wang, Li Li, Nai-Le Liu, Chao-Yang Lu, and Jian-Wei Pan. Quantum computational advantage using photons. *Science*, 370(6523):1460–1463, 2020.

[109] Qingling Zhu, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Dachao Wu, Yulin Wu, Yu Xu, Kai Yan, Weifeng Yang, Yang

Yang, Yangsen Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, and Jian-Wei Pan. Quantum computational advantage via 60-qubit 24-cycle random circuit sampling. *Science Bulletin*, 67(3):240–245, 2022.

[110] D. Zoni, A. Galimberti, and W. Fornaciari. Efficient and scalable fpga-oriented design of qc-ldpc bit-flipping decoders for post-quantum cryptography. *IEEE Access*, 8:163419–163433, 2020.

[111] 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.