An Optimized FrodoKEM Implementation on Reconfigurable Hardware

Giuseppe Manzoni[1 [0009−0007−6339−7139]], Shekoufeh Neisarian[1 [0000−0002−5408−0494]], and Elif Bilge Kavun[1,2 [0000−0003−3193−8440]]

[1] Barkhausen Institut, Schweriner Straße 1, 01067 Dresden, Germany
{giuseppe.manzoni,shekoufeh.neisarian,elif.kavun}@barkhauseninstitut.org
[2] TU Dresden, Nöthnitzer Straße 46, 01187 Dresden, Germany
elif_bilge.kavun@tu-dresden.de

Abstract. FrodoKEM is a Post-Quantum (PQ) Key Encapsulation Mechanism (KEM) built on the Learning with Errors (LWE) problem. Unlike other lattice-based approaches, it avoids using structured lattices to enhance its resilience against attacks. FrodoKEM is selected as a Round 3 alternate candidate in the US National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) Standardization competition, recommended/accepted by several information security agencies in the world, and currently being reviewed for adoption by the International Organization for Standardization (ISO), which calls for efficient real-world implementations of the algorithm. This paper introduces an optimized Field Programmable Gate Array (FPGA)-based architecture for FrodoKEM that achieves up to a factor of 3.5 reduction in resource utilization compared to existing studies, eliminates the need for Digital Signal Processing (DSP) blocks in FPGA implementations, reduces the number of required BRAMs, and delivers up to 9.7 times the throughput. The architecture benefits from parallelization, which results in faster performance, and it integrates key generation, encapsulation, and decapsulation into a single unified implementation that supports all three parameter sets; FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344.

Keywords: Post-Quantum Cryptography (PQC) · Lattice-based · FrodoKEM · Multiplication · Reconfigurable Hardware · Field Programmable Gate Array (FPGA).

1 Introduction

Public-key cryptography relies heavily on mathematical problems such as factoring large integers or computing discrete logarithms, which are computationally difficult to solve in a reasonable time using classical computers [14]. The security of conventional cryptographic algorithms is fundamentally based on the assumption that these problems remain unsolvable. However, the emergence of quantum computers threatens this assumption. With their ability to solve certain problems, such as factoring in polynomial time, quantum computers pose a

significant risk to the security of traditional cryptographic systems [15, 16]. This breakthrough would allow attackers to break public-key encryption schemes and compute secret keys. Since the arrival of quantum computers is inevitable, new ways of securing data are necessary.

This has led agencies and scientists all over the world to invest in research projects to prevent quantum computers from becoming a threat to the data that are currently transmitted on conventional computers [11]. The developments for this are grouped under the term Post-Quantum Cryptography (PQC) [10]. The goal of PQC is to develop systems that are secure on normal computers and quantum computers while simultaneously working on existing communication protocols and networks [12]. Since the mathematical problems of discrete logarithm and integer factorization can be feasibly broken by quantum computers, other problem classes have to be used as a means of achieving security. Some examples currently being researched are code-based cryptography, lattice-based cryptography, and hash-based cryptography [4].

To standardize PQC algorithms, the US National Institute of Standards and Technology (NIST) started its PQC project in February 2016 [13]. In addition to NIST, there are other national institutions that recommend different PQC algorithms to achieve security. Although not solely focused on the task of PQC, the German Federal Office for Information Security (BSI), without influencing the NIST process, gives its recommendation on algorithms to use to secure against quantum attacks. While the standardized algorithms presented by NIST are also part of the BSI's recommendations, their exact recommendations differ. BSI has published its own recommendations for cryptographic mechanisms in the 2024 technical guideline, called "Cryptographic Mechanisms: Recommendations and Key Lengths" [3]. In this guideline, BSI outlines recommendations in several areas, including asymmetric encryption schemes and key agreement, symmetric encryption schemes, hash functions, data authentication, instance authentication, random number generators, and secret sharing. Even though PQC is not the only focus of the guideline, it includes recommendations for PQC algorithms and emphasizes the importance of starting the migration process as early as possible. Although no specific evaluation criteria are provided for PQC algorithms, the guideline sets a minimum security level of 120 bits as the basis for its recommendations.

FrodoKEM [1] is the first Key Encapsulation Mechanism (KEM) recommended by BSI. It is based on the Learning with Errors (LWE) problem using unstructured grids. Similarly to Module-LWE (ML)-KEM, this LWE problem is utilized to construct a Public-Key Encryption (PKE) scheme, with the Fujisaki–Okamoto (FO) transform applied to ensure Indistinguishability under Chosen-Ciphertext Attack (IND-CCA) security. FrodoKEM is considered the more conservative option, as unstructured grids are generally better understood than modules [1]. While NIST decided not to standardize FrodoKEM due to ML-KEM being more efficient, the BSI recommends using FrodoKEM with the parameter sets FrodoKEM-976 and FrodoKEM-1344. This recommendation reflects BSI's preference for schemes with fewer algebraic assumptions and a stronger se-

curity margin, particularly for long-term security use cases. These parameter sets correspond to security levels 3 and 5 of the NIST security levels, respectively, and result in the private key size, public key size, and signature size as shown in Table 1.

Table 1. Resulting key and signature sizes for the different parameter sets for FrodoKEM [1]

Parameter Set	Private Key Size	Public Key Size	Ciphertext Size	Security Level
FrodoKEM-640	19888	9616	9752	1
FrodoKEM-976	31296	15632	15792	3
FrodoKEM-1344	43088	21520	21696	5

The contributions of the paper are as follows. [3]

- We developed an optimized FPGA-based architecture for FrodoKEM, a BSI-recommended Post-Quantum (PQ) PKE scheme.
- Our architecture eliminates the need for Digital Signal Processing (DSP) units by utilizing Look-Up Tables (LUTs) for 16× 5-bit multiplications to avoid the inefficiencies of DSP blocks that are typically designed for 25× 18-bit operations. As we only use a fraction of DSPs' full capabilities, they would be under-utilized resources. If the FrodoKEM module is used as part of a larger design, we would leave the DSPs free to be utilized by another module that needs their full capabilities. Additionally, not all FPGAs have the same number of DSPs; for instance, some FPGAs, such as Lattice Semiconductor's iCE40, do not include DSPs at all. Furthermore, DSPs are large and area-consuming units that could be better utilized for other purposes or saved to reduce costs by opting for a more affordable FPGA.
- Our implementation benefits from 32-way parallel multiplication, which enhances the speed of matrix operations.
- A unified architecture is designed that combines key generation, encapsulation, and decapsulation, to simplify integration and reduce design overhead. This module can simultaneously configure all three functions, with shared resources such as the Keccak module, multiplier, and BRAMs.

This paper is organized as follows. Section 2 provides a detailed overview of FrodoKEM and FPGAs. Section 3 reviews related work in the literature. Section 4 details the optimized architecture and its implementation in FPGAs. Section 5 presents the results. Finally, Section 6 concludes the paper with a summary of the findings.

[3] It's possible to find the verilog implementation described in this paper at https://github.com/bi-tud-sds/lightsec_25_frodokem

2 Background

This section presents details of the lattice-based FrodoKEM scheme and introduces reconfigurable hardware specifics.

2.1 FrodoKEM

FrodoKEM [1] is a lattice-based PQC algorithm based on the LWE problem. Unlike other schemes of this type, FrodoKEM avoids using structured lattices, which provides a more conservative security against quantum attacks, as FrodoKEM will remain secure even if there is a new cryptanalytic development against more structured lattices. The scheme is designed to provide different levels of security, in line with the standards set by NIST for PQC: FrodoKEM-640 offers a level of security similar to AES-128, FrodoKEM-976 provides security comparable to AES-192, and FrodoKEM-1344 has a security level similar to AES-256 [1]. Key generation, encapsulation, and decapsulation are presented in Algorithm 1, Algorithm 2, and Algorithm 3, respectively. The key generation function outputs the keypair $(pk, sk) = (seed_A \parallel b, s \parallel seed_A \parallel b \parallel S^T \parallel pkh)$. The Encapsulation takes as input a public key $pk = seed_A \parallel b$ and outputs a ciphertext $c = c_1 \parallel c_2 \parallel salt$ and a shared secret ss. The decapsulation function takes as input a ciphertext $c = c_1 \parallel c_2 \parallel salt$ and a secret key $sk = s \parallel seed_A \parallel b \parallel S^T \parallel pkh$, and outputs a shared secret ss. A key component in all three algorithms is the generation of the pseudorandom matrix $A \in \mathbb{Z}_q^{n \times n}$. As FrodoKEM avoids algebraic structures, it requires larger matrices, which makes directly storing and transmitting the public matrix A inefficient and resource-heavy. To solve this problem, a seed is included in the key pair and used to generate matrix A at the beginning of each algorithm execution. The function `Gen(`$seed_A$`)` is used to generate this matrix from a seed s of length l_A, which can be derived using either AES-128 or SHAKE-128. AES-128 is typically preferred for software-based implementations, while SHAKE-128 is more efficient and thus commonly used in hardware. It also employs the function `SampleMatrix` to obtain a matrix in $\mathbb{Z}_q^{n \times \bar{n}}$ whose values have a quantized Gaussian sampling distribution, using a pseudorandom array of 16-bit integers r, one per element of the matrix [1].

2.2 Field Programmable Gate Arrays (FPGAs)

FPGAs are reprogrammable logic devices that combine hardware performance with software flexibility. These electronic circuits, composed of an array of programmable logic gates, can be dynamically reshaped to perform various tasks, which distinguishes them from conventional computer chips, which have static functionality. To configure these devices, hardware description languages, such as Verilog, VHDL, or SystemVerilog, are used. These languages allow for the precise definition and implementation of digital circuits by specifying the interactions and functions of the logic gates within the FPGA. Furthermore, design

Algorithm 1 Key Generation for FrodoKEM

Input: None (the random seeds $s, seed_{SE}, z$ are chosen internally)
Output: Public key $pk = (seed_A \parallel b)$, Secret key $sk = (s \parallel seed_A \parallel b \parallel S^T \parallel pkh)$
Choose uniformly random seeds $s, seed_{SE}$ and z with bit lengths $len_{sec}, len_{SE}, len_A$ (respectively)
Generate pseudorandom seed $seed_A \leftarrow \text{SHAKE}(z, len_A)$
Generate matrix $A \leftarrow \text{Gen}(seed_A)$
Generate bit string $(r^{(0)}, r^{(1)}, \ldots, r^{(2n\bar{n}-1)}) \leftarrow \text{SHAKE}(0x5F \parallel seed_{SE}, 32n\bar{n})$
Sample error matrix $S^T \leftarrow \text{SampleMatrix}((r^{(0)}, r^{(1)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n)$
Sample error matrix $E \leftarrow \text{SampleMatrix}((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \ldots, r^{(2n\bar{n}-1)}), n, \bar{n})$
Compute $B \leftarrow A \cdot S + E$
Compute $b \leftarrow \text{Pack}(B)$
Compute $pkh \leftarrow \text{SHAKE}(seed_A \parallel b, len_{sec})$
Return: Public key $pk = seed_A \parallel b$, Secret key $sk = s \parallel seed_A \parallel b \parallel S^T \parallel pkh$
Note: The matrix S^T is encoded row-by-row from $S^T_{0,0}$ to $S^T_{\bar{n}-1,n-1}$, where each matrix coefficient $S^T_{i,j}$ is a signed integer encoded as a 15 or 16-bit string in little-endian byte order. The encoding of $S^T_{i,j}$ is given by:

$$(s_0, s_1, \ldots, s_{15}) \leftarrow S^T_{i,j} = -s_{15} \cdot 2^{15} + \sum_{k=0}^{14} s_k \cdot 2^k$$

Algorithm 2 Encapsulation for FrodoKEM

Input: Public key $pk = seed_A \parallel b$
Output: Ciphertext $c = c_1 \parallel c_2 \parallel salt$, Shared secret ss
Choose uniformly random values u and $salt$ with bit lengths len_{sec} and len_{salt} (respectively)
Compute $pkh \leftarrow \text{SHAKE}(pk, len_{sec})$
Generate values $seed_{SE} \parallel k \leftarrow \text{SHAKE}(pkh \parallel u \parallel salt, len_{SE} + len_{sec})$
Generate $(r^{(0)}, r^{(1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}) \leftarrow \text{SHAKE}(0x96 \parallel seed_{SE}, 16(2\bar{n}n + \bar{n}^2))$
Sample error matrix $S' \leftarrow \text{SampleMatrix}((r^{(0)}, r^{(1)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n)$
Sample error matrix $E' \leftarrow \text{SampleMatrix}((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \ldots, r^{(2n\bar{n}-1)}), \bar{n}, n)$
Generate the matrix $A \leftarrow \text{Gen}(seed_A)$
Compute $B' \leftarrow S' \cdot A + E'$
Compute $c_1 \leftarrow \text{Pack}(B')$
Sample error matrix $E'' \leftarrow \text{SampleMatrix}((r^{(2\bar{n}n)}, r^{(2\bar{n}n+1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}), \bar{n}, \bar{n})$
Compute $B \leftarrow \text{Unpack}(b, n, \bar{n})$
Compute $V \leftarrow S' \cdot B + E''$
Compute $C \leftarrow V + \text{Encode}(u)$
Compute $c_2 \leftarrow \text{Pack}(C)$
Compute $ss \leftarrow \text{SHAKE}(c_1 \parallel c_2 \parallel salt \parallel k, len_{sec})$
Return: Ciphertext $c = c_1 \parallel c_2 \parallel salt$, Shared secret ss

Algorithm 3 Decapsulation for FrodoKEM

Input: Ciphertext $c = c_1 \parallel c_2 \parallel salt$, Secret key $sk = s \parallel seed_A \parallel b \parallel S^T \parallel pkh$
Output: Shared secret ss
Compute $B' \leftarrow \text{Unpack}(c_1, \bar{n}, n)$
Compute $C \leftarrow \text{Unpack}(c_2, \bar{n}, \bar{n})$
Compute $M \leftarrow C - B' \cdot S$
Compute $u' \leftarrow \text{Decode}(M)$
Generate values $seed'_{SE} \parallel k' \leftarrow \text{SHAKE}(pkh \parallel u' \parallel salt, len_{SE} + len_{sec})$
Generate $(r^{(0)}, r^{(1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}) \leftarrow \text{SHAKE}(0x96 \parallel seed'_{SE}, 16(2\bar{n}n + \bar{n}^2))$
Sample error matrix $S' \leftarrow \text{SampleMatrix}((r^{(0)}, r^{(1)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n)$
Sample error matrix $E' \leftarrow \text{SampleMatrix}((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \ldots, r^{(2n\bar{n}-1)}), \bar{n}, n)$
Generate the matrix $A \leftarrow \text{Gen}(seed_A)$
Compute $B'' \leftarrow S' \cdot A + E'$
Sample error matrix $E'' \leftarrow \text{SampleMatrix}((r^{(2\bar{n}n)}, r^{(2\bar{n}n+1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}), \bar{n}, \bar{n})$
Compute $B \leftarrow \text{Unpack}(b, n, \bar{n})$
Compute $V \leftarrow S' \cdot B + E''$
Compute $C' \leftarrow V + \text{Encode}(u')$
`(In constant time)` $\bar{k} \leftarrow k'$ if $B' \parallel C = B'' \parallel C'$ else $k' \leftarrow s$
Compute $ss \leftarrow \text{SHAKE}(c_1 \parallel c_2 \parallel salt \parallel \bar{k}, len_{sec})$
Return: Shared secret ss

tools/suites accompanying FPGAs offer software that is used to design and program FPGA tasks sequentially.

Their reprogrammable nature allows FPGAs to adapt to different tasks without replacement, which makes them different from Application-Specific Integrated Circuits (ASICs), which are designed for specific purposes. Inside an FPGA, there is a grid-like structure comprised of Configurable Logic Blocks (CLBs). These CLBs contain basic building blocks, such as Lookup Tables (LUTs), Multiplexors (MUXs), Full Adders (FAs), and D-Flip Flops (D-FFs). They can be thought as individual puzzle pieces that can be rearranged to create different circuits. The interconnections between these blocks enable communication and cooperation.

LUTs are employed in FPGAs as an alternative to conventional logic gates. LUTs serve as highly configurable, programmable memory units capable of executing a wide range of logical functions. In particular, FPGAs such as Zynq UltraScale+ and Artix-7 typically support configurations with either five input bits and two output bits or six input bits and one output bit [2]. Each LUT can be programmed to store a truth table, mapping each of the $2^5 = 32$ input combinations to one of the $2^2 = 4$ possible output states in a 5-input, 2-output configuration. This capability enables a LUT to implement up to 4^{32} different functions based on various input and output configurations. The architecture of an LUT comprises memory cells equal to 2^n, where n is the number of inputs. The exponential growth in memory capacity as the number of inputs increases generally limits practical FPGA applications to LUTs with two to five inputs. These configurations allow FPGAs to handle significantly more complex digital

designs using fewer resources. In addition, LUTs can be interconnected to create small blocks of Random Access Memory (RAM) and execute complex functions or algorithms, which highlights their essential role in supporting customizable digital circuits within FPGAs.

FPGAs also feature Input-Output (I/O) blocks that act as gateways for signals entering and exiting the FPGA. Additionally, there are Fixed Functional Logic Blocks (FFLBs) like multipliers or Digital Signal Processing (DSP) blocks, as well as Block Random Access Memory (BRAM) for data storage. BRAMs are distinct components within FPGAs, separate from the individual LUTs that are typically used for logic operations. Unlike LUTs, BRAMs serve as dedicated memory blocks designed to store large amounts of data. This includes read-only data, temporary data, and data read from external devices. DSPs are more resource-intensive than LUTs and consume more power and area, especially for simple logic tasks. Additionally, LUTs are better in parallel designs, whereas DSPs are optimized for specific tasks and may not handle parallelism as effectively.

3 Related Work

Several works have explored FrodoKEM from both hardware and software perspectives, each with distinct trade-offs in performance, area, and design strategy. The first hardware implementation of FrodoKEM was introduced by Howe et al. [8], targeting the FrodoKEM-640 and FrodoKEM-976 parameter sets. Although it demonstrated the feasibility of hardware acceleration for lattice-based cryptography, the design was limited in terms of parallelism and resource efficiency. A subsequent hardware implementation was proposed by Howe et al. [7], also focusing on FrodoKEM-640 and 976. However, this design used Trivium to generate the large matrix A instead of the SHAKE128 or AES functions recommended by NIST. This deviation from the standard FrodoKEM specification reduces the comparability and applicability of the implementation in standardized settings.

Gu et al. [6] accelerated FrodoKEM on the Zynq Ultrascale+ FPGA platform by proposing a new architecture for the SHAKE128 function. They designed a high-throughput SHA-3 structure by optimizing combinational logic calculations and utilizing pipelining techniques, but the different FPGA platform makes it difficult to compare with existing results.

In the software domain, the work from Fiho et al. [5] investigated matrix multiplication optimizations and introduced strategies similar to the AMX accelerator found in Apple processors. Notably, our Mode 2 multiplier architecture closely aligns with the approach described in this work, although we implement it fully on hardware instead of using an accelerator.

4 Implementation

This section describes the architecture of our implementation, which integrates all three key generation, encapsulation, and decapsulation of FrodoKEM into a single module. First, we detail the sub-modules responsible for data handling, followed by those that manage command processing. Finally, we discuss general design considerations and optimization techniques.

4.1 Data Modules

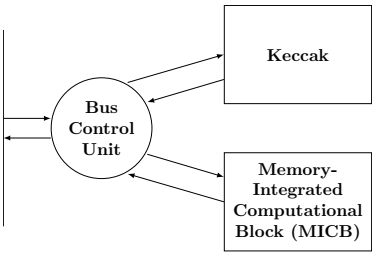

Fig. 1. High-level design architecture.

From a high-level perspective, the architecture consists of two primary sub-modules: the `Keccak` module, which handles the SHAKE128 and SHAKE256 computations, and the `Memory-Integrated Computational Block (MICB)` module, which stores the remaining variables and performs all other necessary operations. Each of these sub-modules interfaces with the peripherals through two ports, one for input and one for output. The connections between the sub-modules and the peripherals are managed by a `Bus Control Unit`, as illustrated in Figure 1.

The ports have 64-bit data wires for unidirectional data transmission. Additionally, there is a `canReceive` wire, which operates in the opposite direction of the data and indicates whether the destination port is capable of receiving data. An `isReady` wire signals when data has been sent, and it can only be set if the `canReceive` wire is also active. Furthermore, a `isLast` wire is used to specify whether the current element is the final item in the data stream. This `isLast` wire may be driven either by the data source or the destination, with the goal of minimizing the need for additional counters and reducing the size of the control buses. With the exception of the `isLast` wire, this is the same port that our implementation uses to communicate with the peripherals. For these two ports, the `isLast` wire is managed by two small sub-modules located between the peripherals and the `Bus Control Unit`.

The `Bus Control Unit` is responsible for connecting three pairs of ports and routing the data between them, and it supports one multicast communication at

a time. The unit uses the `isLast` wire of the ports to determine when a data flow has ended and when it can schedule a new connection between a set of ports.

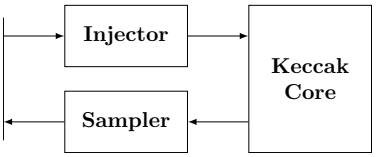

Fig. 2. Architecture of the `Keccak` module.

Keccak Module As shown in Figure 2, the `Keccak` module can be divided into three parts: the `Keccak Core` module, which performs the main operation; the `Sampler` module, which optionally performs FrodoKEM's sampling on the output received from the `Keccak Core`; and the `Injector` module, which can inject into the data fed into the `Keccak Core` either an arbitrary number of 64-bit zeros, an arbitrary byte, or the $Seed_A$, which it stores internally. To allow sending bytes, the input port of the `Keccak Core` includes an additional `isByte` wire, which differentiates between 8-bit and 64-bit values.

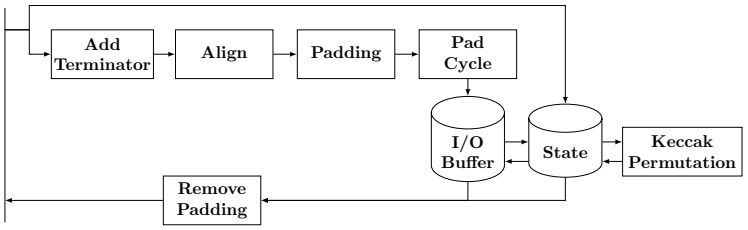

Fig. 3. Architecture of the `Keccak Core` module.

In Figure 3, the structure of the `Keccak Core` is shown. This module can perform both the SHAKE128 and SHAKE256 operations, with the option to save and load the internal state. The `State` is implemented as a circular buffer, allowing for state I/O, and is used by the `Keccak Permutation` module every clock cycle to perform one round of the Keccak permutation on the 1600-bit state. Each round takes one clock cycle, and a total of 24 cycles are required to complete the full permutation. Additionally, the module includes a `I/O buffer`, which is another circular buffer for a 1344-bit value, corresponding to the data rate of SHAKE128. This buffer is kept separate from the `State` because reading and writing the I/O of the SHAKE128 function with a 64-bit bus takes 21 clock cycles. Keeping the buffers separate allows the Keccak permutation to be applied while the I/O operations are ongoing. The input to the `I/O buffer` is processed

in multiple stages. First, the `Add Terminator` module appends the $0x1F$ SHAKE terminator at the end of the input (indicated by the `isLast` wire), by injecting a single byte into the bus and delaying the `isLast` signal. The `Align` bus converts the 8-bit or 64-bit bus into a 64-bit-only bus. The `Padding` module applies the Keccak padding to fill the input block to either 1088 or 1344 bits, and adds the last terminator bit required by Keccak's padding scheme. Finally, the `Pad Cycle` module adds zero words to fill the `I/O buffer` in the case of using SHAKE256, which has a smaller data rate than SHAKE128. The final sub-module of the `Keccak` module is the `Remove Padding` module, which removes the data from the `I/O buffer` that should not be sent as output. This occurs either when the data rate is smaller than the full buffer size or when the destination of the data flow sets the `isLast` wire.

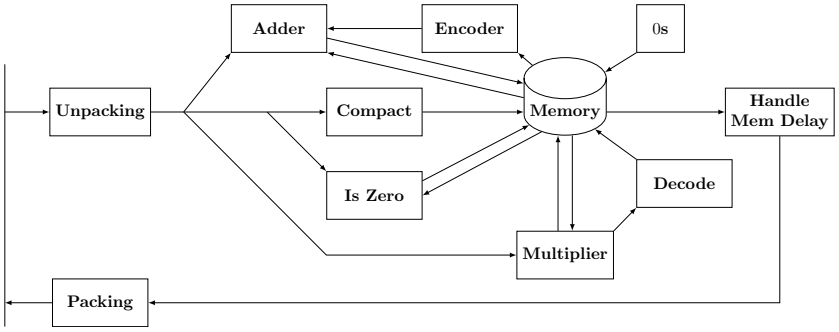

Fig. 4. Architecture of the `MICB` module.

MICB Module The internal architecture of the `MICB` module is shown in Figure 4. It consists of two main sub-modules: the `Memory` and `Multiplier` sub-modules, along with several smaller operations and multiplexers. The buses in the module are composed solely of data wires of varying sizes, and the entire module is centrally controlled. The `0s` produces zeros to erase secret information from memory. The `Is Zero` verifies whether the input consists of zeros and then outputs either a value from the memory or the input. The `Encode`, `Decode`, `Pack`, and `Unpack` modules perform the corresponding FrodoKEM operations on buses of 4, 8, 4, and 4 values in parallel, respectively. The `Adder` operates on 4 pairs of values in parallel and is capable of both addition and subtraction. The `Handle Mem Delay` compensates for the 2-cycle delay of the BRAM's read operation. Finally, the `Compact` module either leaves the input unchanged or converts the 16-bit standard encoding of matrix S into a compact format with 4-bit values plus a sign, 4 values at a time. This transformation is possible because the values of S fall within the range $[-12, 12]$.

The `Multiplier` consists of 32 multipliers that operate on 5-bit and 16-bit operands, along with internal storage for temporary values and two modes of

operation. In the first mode, it multiplies an input matrix of size 8×4 containing 5-bit values by an input vector of size 4×1 containing 16-bit values, accumulating the resulting 8×1 values in the internal storage. In the second mode, it multiplies a vector of size 8×1 containing 5-bit values from the internal storage by an input vector of size 1×4 containing 16-bit values, producing an output matrix of size 8×4.

Lastly, the `Memory` module consists of 8 BRAMs, organized as an 8×512 matrix of 64-bit words, and stores all variables of the FrodoKEM algorithms, except for $seed_A$. The non-matrix values are stored column-wise across the BRAMs. The 16-bit matrices are stored with 4 values in each word, while S is stored differently depending on the FrodoKEM variant: for FrodoKEM-1344, eight 4-bit values are stored per word, and for other variants, four 5-bit values are stored per word, depending on the range of the sampling distribution. Lastly, matrices with asymmetric dimensions may be stored in a transposed format to optimize memory usage.

4.2 Control Modules

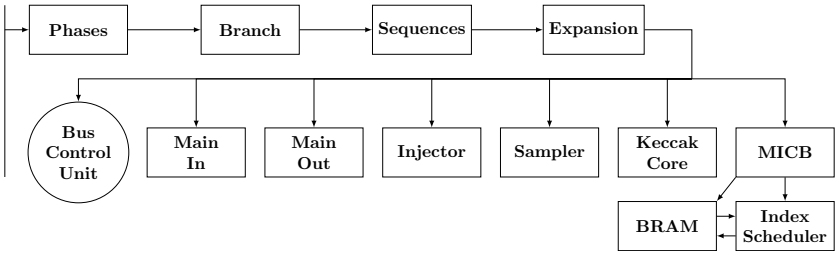

Fig. 5. Architecture of the control modules.

The implementation accepts multiple input commands: Key Generation, Encapsulation, Decapsulation, Add Entropy, and Setup Test. The first three commands are based on the FrodoKEM specification. The Add Entropy command is used to inject entropy into the Pseudo Random Number Generator (PRNG), for example, after startup. The Setup Test command disables the PRNG for the subsequent execution and loads values that would otherwise be generated by it. The input commands are sent to a `Phases` module, which breaks them down into a sequence of code blocks, ensuring that branches are fully contained within each phase. The output is then passed to a `Branch` module, which manages the loop to generate the matrix A and handles branches depending on whether the execution is a test or not, such as deciding whether to run the PRNG. These commands are subsequently sent to `Sequences`, which expands each input command into a series of individual instructions. Finally, `Expansion` converts each instruction into a command for the corresponding module, as illustrated in Figure 5. In

that figure, the `Main-in` and `Main-out` control the primary input and output of the implementation, while the `Bus Control Unit`, `Injector`, and `Sampler` modules serve as the controllers for the components described in the previous section. It is worth noting the `Keccak Core` controller, which includes an internal scheduler that enables partial overlapping of two consecutive executions of the SHAKE function. This feature is crucial to prevent noticeable slowdowns during the generation of the matrix A. Each row of A is generated with an independent SHAKE128 call, and a purely sequential execution would waste cycles to complete the last output before beginning the first input of the next execution. Instead, the scheduler allows the first input of the next execution to be processed during the final execution of the `Keccak Permutation` module. The last output is then computed while performing the first execution of the next `Keccak Permutation`. In this way, the `Keccak Core` continuously computes the Keccak permutation without idle cycles. Lastly, the control modules of the `MICB` module are introduced. The main control module breaks the 35 commands into multiple phases, if necessary, and determines which value or matrix is being operated on, as well as the type of operation to be performed. For each operation, the `Index Scheduler` manages the timing of the current index update, deciding which index to send data to or receive data from. The `BRAM` module handles the low-level details of storage, such as managing the offset and size of values, and rearranging the 512-bit parallel output from the 8 BRAMs into smaller buses for the output of the `Memory` module. It also performs the reverse operation for the input.

4.3 Inter Modules

In this section, we describe implementation details that are transversal to the individual modules.

Not Storing the Ciphertext in the Decapsulation The decapsulation algorithm uses the ciphertext both at the beginning and at the end, but we will show here that both the latter usages can be moved ahead of time. This allows us to avoid saving the ciphertext, which can be up to 21,696 bits in size, and storing it would require an additional BRAM, increasing the area of the implementation.

The first operation that we need to move is the comparison between the matrices generated internally (B'', C') and those received as part of the ciphertext (B', C). This use can be moved ahead by initializing B'' with $-B'$ and C' with $-C$. In this way, at the end of the decapsulation we only need to compare the matrices with zero.

The second operation is the final hash of the decapsulation, which is given by $ss = \text{SHAKE}(c \parallel \bar{k})$, where c represents the ciphertext. In this case we can not move the invocation of the hash function to an earlier point because $\bar{k}$ is available only at the end. Our solution was to store the 1,600-bit state of the `Keccak` module mid-operation in the BRAMs of the `Memory` module. This approach splits the input operation at a point that is not aligned with SHAKE's data

rate, necessitating a division of the SHAKE operation in the middle of an input block. To address this, the `Injector` module is used to append zero-padding after the ciphertext to complete the input block, and subsequently, zero-padding is prepended before $\bar{k}$ to position it correctly within the input block.

Multiplication FrodoKEM involves various multiplications in its algorithms, utilizing the two types of multiplication described previously. Specifically, in key generation, the expression $B = A \cdot S + E$ is transformed into $B^T += S^T \cdot A^T$ to employ the first multiplier mode, with B initialized as E. In the encapsulation algorithm, the multiplications $B' = S' \cdot A + E'$ use the second multiplier mode, while $V = S' \cdot B + E''$ uses the first mode. In decapsulation, this last multiplication uses the second mode of operation as the intialization of B'' with $-B'$ means we can not store B in memory. As the packing restricts how the matrix B can be accessed, we need to use the second mode of operation of the multiplier. Lastly, the decapsulation has the additional multiplication $M = C - B' \cdot S$, which utilizes the first multiplier mode.

As with most existing implementations, we generate the matrix A on the fly, and in our case, we do so row-by-row. The operations involving the multiplications with the matrix A will now be detailed, as these are the slowest parts of the FrodoKEM algorithms and, therefore, the most time-sensitive.

During the $B^T += S^T \cdot A^T$ operation, the `MICB` module manages the loading and storing of a column of B^T for each row of A, and loads an 8×4 sub-matrix of S for each block of 4×1 values of A, which are provided row-by-row from the `Keccak` module. As the processing of a row nears completion, the scheduler of the `Keccak` module facilitates the reception of the input for the next SHAKE128 invocation, signaling through the `canReceive` wire to the `Injector` to transmit the two bytes representing the row number and then the $seed_A$. Overall, this operation accesses A row-wise, while S is accessed once per row of A. Each row of B is updated once during the operation. The $B' += S' \cdot A$ operation is similar, with the matrix A being accessed row-by-row. The key difference is that S' is accessed only once, with a column vector being loaded before each row of A. The matrix B' is fully updated once for every row of A, and for each block of 4×1 values of A, a sub-matrix of 8×4 values of B' is updated.

PRNG The key generation and encoding algorithms of FrodoKEM require pseudo-random inputs. To generate these, the `Keccak` module is used in conjunction with a pool of 512-bit values stored in the `Memory` module. The entropy in the pool can be increased through an explicit command from the main module of the implementation. Additionally, the pool is automatically updated using the inputs processed by other commands. At the beginning of the key generation, the required pseudorandom variables are generated using SHAKE256($0x00 \parallel$ pool), and the entropy is subsequently updated with pool = SHAKE256($0x01 \parallel$ pool). Near the start of the encapsulation process, the required pseudorandom values are generated using SHAKE256($0x02 \parallel$ pool), and the pool is updated with SHAKE256($0x03 \parallel$ pool $\parallel$ pkh). Here, pkh refers to the hash of the public key,

which serves to introduce additional entropy. The hash typically contains entropy from an independent entropy pool, and if an adversary misses a single public key, they would need to guess the entire hash in order to derive the next state of the pool. Finally, while decapsulation does not require any pseudorandom values, the entropy is still updated with SHAKE256($0x04 \parallel \text{pool} \parallel ss$), where ss represents the shared secret. This value is inaccessible to an adversary who either failed to intercept the incoming message or did not possess the private key, making it a valuable source of entropy. The explicit command for adding entropy executes pool = SHAKE256($0x05 \parallel \text{pool} \parallel \text{IN}$), where the input IN is 512 bits.

5 Results

This section presents the results of our proposed implementation and compares them with existing FPGA-based implementations, more specifically with Howe et al.'s work from 2018 [8]. We believe that Howe et al.'s work from 2021 [7] is not a fair comparison because its goal is to help the standardization process explore different alternatives, and so they generate the matrix A with Trivium, a lightweight cryptosystem, instead of the standard Shake128 or AES. Comparing the two would be like comparing the time of an implementation that uses RSA and one that uses ECC, the result is meaningful to compare variations of the standard, but it is not a fair comparison between implementations. This deviation from the standard is particularly relevant when in Section 1, paragraph 5, the authors say "To be parallelised, however, the matrix multiplication requires the use of a smaller and more performant pseudo-random number generator. We propose to achieve the performance required for the randomness generation by using Trivium" which we believe hints that their architecture would not be feasible with the regular Shake128 function.

We benchmark our results using the Xilinx Artix-7 XC7A35T FPGA using Vivado Design Suite 2024.2, while Howe et al. [8] uses the same FPGA but with Vivado Design Suite 2019.1. Also, we measured the overall area consumption in equivalent slices, which we estimate using the formula [9, 2]:

$$\text{AREA} = 0.25 * \text{LUT} + 0.125 * \text{FF} + 102.4 * \text{DSP} + 116.2 * \text{BRAM} \quad (1)$$

In Figure 6 and Table 2 we show the comparison of our results with [8]'s. More specifically, we show both a comparison with the individual modules, and with a set of modules to achieve a given functionality.

If we compare a single algorithm of a single parameter set of FrodoKEM, we see that our module has 9.0× to 9.7× the throughput of [8]. On the other hand, comparing the area is more nuanced as our module does more than any individual module of [8]. The biggest module of [8] that is functionally included in ours is the FrodoKEM-976 decapsulation, and our implementation is 21% bigger, but the area to throughput ratio is reduced by a factor of 8. We want to point out that while we do have an increase in size, we also support the higher 1344 parameter set, which Howe et al.'s work [8] does not as it had not been standardized yet.

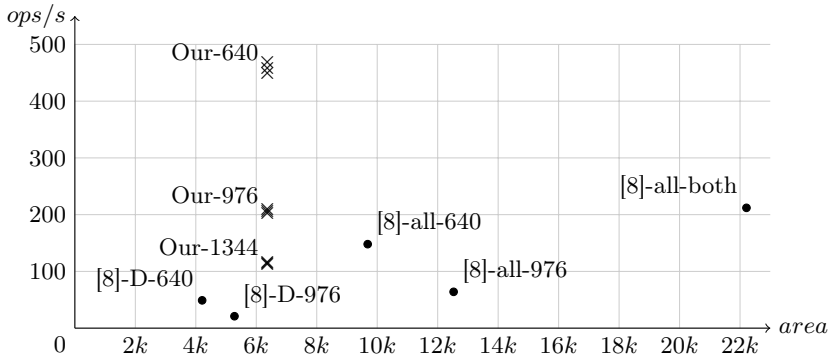

Fig. 6. Comparisons of area and throughput. For our results, we only use one label for each parameter set as the three data points are really close. Legend: Decapsulation (-D-), All three algorithms (-all-), both parameter sets supported by [8] (-both).

The other case is if the user needs a full implementation of FrodoKEM. In ASICs, it is common practice to create a single mask that contains all the functionalities to reduce costs. Yet this is still relevant for FPGAs, as a common use case is a server, which needs to be able to generate keys and perform the decapsulation to allow clients to connect to it, but it also needs to support the encapsulation too for whenever it needs to access external resources, for example, a database. Lastly, a server should support multiple parameter sets as it does not know what the other endpoint will support. In this case, we need to consider the set of all modules from [8]. The area of our module is a factor of 3.5 smaller than that of this set. The throughput depends on which operations we need to execute, as the set of [8]'s 6 modules can execute six operations at a time only for different operations. In the best case, their throughput is of 212 operations per second. For the same distribution of operations, our module computes 335 operations per second, which is 58% higher. Of course, if the operations are all of the same type, for example due to a Denial of Service attack, then our throughput is 798% to 868% higher, which shows the advantages of the higher flexibility of our unified module. This flexibility is further enhanced by our module's ability to execute operations for the 1344 parameter set.

6 Conclusion

This paper presents a novel hardware architecture optimized for FrodoKEM, a lattice-based PQC algorithm that is included in the BSI's recommended cryptographic algorithms. The proposed FPGA-based implementation has a novel architecture that allows it to reduce the number of BRAMs, eliminate the need for DSP blocks, and increase throughput and flexibility. The evaluation results, conducted on an Artix-7 FPGA, demonstrate a significant throughput improvement of nearly an order of magnitude over existing state-of-the-art FrodoKEM

Table 2. Comparison of the implementations. Legend: Key generation (K), Encapsulation (E), Decapsulation (D). 'Both' and 'all' mean multiple modules together. A/T is the Area to Throughput Ratio. All works use the XC7A35T FPGA.

Ref.	Param.	Alg.	# of LUTs	# of FFs	# of BRAMs	# of DSPs	Freq. (MHz)	# of Slices	Op/s	A/T
[8]	640	K	3771	1800	6	1	167	1967	51	38.56
		E	6745	3528	11	1	167	3507	51	68.76
		D	7220	3549	16	1	162	4210	49	85.91
	976	K	7139	1800	8	1	167	3042	22	138.27
		E	7209	3537	16	1	167	4206	22	191.18
		D	7773	3559	24	1	162	5279	21	251.38
	640		17736	8877	33	3	162	9685	148	65.43
	976	all	22121	8896	48	3	162	12527	64	195.73
	both		39857	17773	81	6	162	22212	212	104.77
Ours	640	K	19082	5331	8	0	62.5	6367	469	13.59
		E							458	13.90
		D							449	14.18
	976	K							210	30.38
		E							206	30.92
		D							203	31.31
	1344	K							116	54.98
		E							114	55.71
		D							113	56.25

implementations. Additionally, the design benefits from a unified architecture that integrates key generation, encapsulation, and decapsulation into a single, streamlined module that can run either of them, and that can execute any of the three parameter sets of FrodoKEM.

Acknowledgements This work is supported in part by DFG Project No. 543352068 and the AMD University Program.

References

1. Alkim, E., Bos, J.W., Ducas, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: FrodoKEM: Learning With Errors Key Encapsulation Algorithm Specifications and Supporting Documentation. Tech. rep., FrodoKEM Project (June 2021), `https://frodokem.org/files/FrodoKEM-specification-20210604.pdf`, Accessed: 2025-04-30.
2. AMD: 7 Series FPGAs Data Sheet: Overview (DS180) (2020), `{https://docs.amd.com/v/u/en-US/ds180_7Series_Overview}`
3. Federal Office for Information Security (BSI): Cryptographic Mechanisms: Recommendations and Key Lengths. Tech. Rep. BSI TR-02102-1, Federal Office for Information Security, Bonn, Germany (February 2024), `https://www.bsi.bund.d`

e/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf, Accessed: 2025-04-30.
4. Federal Office for Information Security (BSI): Post-Quantum Cryptography (2024), https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Quantentechnologien-und-Post-Quanten-Kryptografie/Post-Quanten-Kryptografie/post-quanten-kryptografie_node.html, Accessed: 2025-04-30.
5. Filho, D.L.G., Brandão, G., Adj, G., Alblooshi, A., Canales-Martínez, I.A., Chávez-Saab, J., López, J.: Pqc-amx: Accelerating saber and frodokem on the apple m1 and m3 socs. In: 2024 IEEE 31st Symposium on Computer Arithmetic (ARITH). pp. 9–16 (2024). https://doi.org/10.1109/ARITH61463.2024.00012
6. Gu, S., Lu, J., Huang, T., Zhang, J., Li, K., Wu, C., Wang, M., Mei, X., Hu, A., Liu, D.: A low latency and high throughout hardware design of random matrix number generator for frodokem. In: 2024 IEEE 17th International Conference on Solid-State & Integrated Circuit Technology (ICSICT). pp. 1–3 (2024). https://doi.org/10.1109/ICSICT62049.2024.10831677
7. Howe, J., Martinoli, M., Oswald, E., Regazzoni, F.: Exploring Parallelism to Improve the Performance of FrodoKEM in Hardware. Journal of Cryptographic Engineering **11**(4), 317–327 (2021). https://doi.org/10.1007/s13389-021-00258-7, https://link.springer.com/article/10.1007/s13389-021-00258-7
8. Howe, J., Oder, T., Krausz, M., Güneysu, T.: Standard Lattice-Based Key Encapsulation on Embedded Devices. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 372–393 (2018). https://doi.org/10.13154/tches.v2018.i3.372-393
9. Liu, W., Fan, S., Khalid, A., Rafferty, C., O'Neill, M.: Optimized Schoolbook Polynomial Multiplication for Compact Lattice-Based Cryptography on FPGA. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(10), 2459–2463 (2019). https://doi.org/10.1109/TVLSI.2019.2922999
10. Moody, D., Perlner, R., Regenscheid, A., Robinson, A., Cooper, D.: Transition to Post-Quantum Cryptography Standards. NIST Internal Report NIST IR 8547 IPD, National Institute of Standards and Technology, Gaithersburg, MD (November 2024), https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8547.ipd.pdf, Accessed: 2025-04-30.
11. Mosca, M., Piani, M.: Quantum Threat Timeline Report 2023. Tech. rep., Global Risk Institute (December 2023), https://globalriskinstitute.org/publication/2023-quantum-threat-timeline-report/, Accessed: 2025-04-30.
12. National Institute of Standards and Technology: What Is Post-Quantum Cryptography? (August 2024), https://www.nist.gov/cybersecurity/what-post-quantum-cryptography, Last modified: 2025-06-11. Accessed: 2025-08-05.
13. NIST Information Technology Laboratory, Computer Security Resource Center: Post-Quantum Cryptography (January 2017), https://csrc.nist.gov/Projects/post-quantum-cryptography, Accessed: 2025-04-30.
14. Paar, C., Pelzl, J.: Understanding Cryptography: A Textbook for Students and Practitioners. Springer, Heidelberg, Germany (2010)
15. Shor, P.W.: Polynomial-time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Review **41**(2), 303–332 (1999). https://doi.org/10.1137/S0036144598347011
16. Trend Micro: Quantum Computing Attacks on Classical Cryptography (2023), \url{https://www.trendmicro.com/vinfo/us/security/news/security-technology/post-quantum-cryptography-quantum-computing-attacks-on-classical-cryptography}, accessed: 2025-04-30