# Accelerating FrodoKEM in Hardware

Sanjay Deshpande[1], Patrick Longa[2] and Jakub Szefer[1]

[1] Northwestern University, Evanston, USA,
sanjay.deshpande1@northwestern.edu,jakub.szefer@northwestern.edu
[2] Microsoft Research, Redmond, USA, plonga@microsoft.com,

**Abstract.** FrodoKEM, a conservative post-quantum key encapsulation mechanism based on the plain Learning with Errors (LWE) problem, has been recommended for use by several government cybersecurity agencies and is currently undergoing standardization by the International Organization for Standardization (ISO). Despite its robust security guarantees, FrodoKEM's performance remains one of the main challenges to its widespread adoption. This work addresses this concern by presenting a fully standard-compliant, high-performance hardware implementation of FrodoKEM targeting both FPGA and ASIC platforms. The design introduces a scalable parallelization architecture that supports run-time configurability across all twelve parameter sets, covering three security levels (L1, L3, L5), two PRNG variants (SHAKE-based and AES-based), and both standard and ephemeral modes, alongside synthesis-time tunability through a configurable performance parameter to balance throughput and resource utilization. For security level L1 on Xilinx Ultrascale+ FPGA, the implementation achieves 3,164, 2,846, and 2,614 operations per second for key generation, encapsulation, and decapsulation, respectively, representing the fastest standard-compliant performance reported to date while consuming only 27.8K LUTs, 64 DSPs, and 8.1K flip-flops. These results significantly outperform all prior specification-compliant implementations and even surpass non-compliant designs that sacrifice specification adherence for speed. Furthermore, we present the first ASIC evaluation of FrodoKEM using the NANGATE45 45 nm technology library, achieving 7,194, 6,471, and 5,943 operations per second for key generation, encapsulation, and decapsulation, respectively, with logic area of 0.235 mm$^2$. The ASIC implementation exhibits favorable sub-linear area scaling and competitive energy efficiency across different performance parameter configurations, establishing a baseline for future comparative studies. The results validate FrodoKEM's practical viability for deployment in high-throughput, resource-constrained, and power-sensitive cryptographic applications, demonstrating that conservative post-quantum security can be achieved without compromising performance.

**Keywords:** PQC · FrodoKEM · Hardware · FPGA · ASIC

## 1 Introduction

The advent of quantum computing poses a significant threat to classical public-key cryptographic schemes, such as RSA and ECC, which underpin the security of today's digital infrastructure [Sho94, DS13, KBF$^+$15, Mos18]. In anticipation of the emergence of cryptographically relevant quantum computers (CRQCs), the cryptographic community has been engaged in a major global effort to develop and standardize post-quantum cryptographic (PQC) algorithms to protect against quantum computer attacks. Over the past decade, significant progress has been made in this direction, most notably through initiatives such as the NIST PQC standardization process [Nat17] and parallel efforts by international bodies like the International Organization for Standardization (ISO) and the Internet Engineering Task Force (IETF).

Among the various families of PQC algorithms, lattice-based schemes have emerged as particularly promising due to their strong security foundation and relatively efficient implementations. FrodoKEM [ABD+25a], a conservative lattice-based key encapsulation mechanism (KEM) rooted in the hardness of the plain Learning with Errors (LWE) problem, has garnered considerable attention in the community. While FrodoKEM was not selected as a finalist in the NIST competition, it continues to be a strong alternative partly due to the notable support by several government agencies, including the Dutch NLNCSA and AIVD [Gen24, Gen22], the French ANSSI [Nat23], and the German BSI [Fed24], which have recommended it for use as a conservative option. Moreover, FrodoKEM is undergoing standardization by ISO [Int24].

As PQC transitions from theory to practice, hardware evaluation becomes a critical aspect of readiness for real-world deployment. Efficient and secure hardware implementations are essential for integrating PQC into constrained and high-performance environments. Although FrodoKEM attracted some hardware research during its time in the NIST process [HOKG18, HMOR21], prior designs in the literature primarily focused on low-area implementations or did not fully adhere to the official specification, often compromising on performance and completeness.

**Contributions.** In this work, we present a high-performance, fully standard-compliant[1] hardware implementation of FrodoKEM for both FPGA and ASIC platforms. Our design features comprehensive configurability: run-time switching between all twelve parameter sets, covering three security levels (L1, L3, L5), two PRNG variants (SHAKE-based and AES-based), and both standard and ephemeral modes, plus synthesis-time tunability through a parameterizable performance factor (called `"T"`) to balance throughput and resource utilization. This unified multi-parameter approach is particularly valuable for ASIC deployments, where post-fabrication flexibility is not possible: a single chip can adapt to different security requirements and applications without requiring separate silicon implementations. For FPGA platforms, this design enables dynamic security level negotiation and protocol adaptation at run-time. For example, for security level L1, our FPGA implementation achieves 3,164, 2,846, and 2,614 operations per second for key generation, encapsulation, and decapsulation, respectively, making them the fastest standard-compliant results reported to date while consuming only 27.8K LUTs, 64 DSPs, and 8.1K flip-flops. Furthermore, we present the first ASIC evaluation of FrodoKEM. Notably, our implementation demonstrates sub-linear area scaling, achieving (for security level L1) 7,194, 6,471, and 5,943 operations per second for key generation, encapsulation, and decapsulation, respectively, with logic area ranging from 0.128 to 0.235 mm$^2$ across different performance (`T`) configurations. Refer to Section 7 for complete results.

## 2 FrodoKEM: Background

This section briefly describes FrodoKEM and its associated algorithms. For complete details, readers are referred to [GLN+25, ABD+25b].

FrodoKEM is a post-quantum, IND-CCA secure KEM [ABD+25a] that is based on the plain LWE problem [Reg05], a fundamental hard problem in lattice-based cryptography. Unlike many other lattice-based cryptographic schemes that rely on structured lattices such as Ring-LWE or Module-LWE variants (e.g., NewHope [ADPS16] and CRYSTALS-Kyber [Nat24]), FrodoKEM is built on plain, unstructured lattices, making it a conservative choice in terms of security since this design decision eliminates potential vulnerabilities arising from algebraic structures.

FrodoKEM builds upon FrodoPKE, a public-key encryption scheme whose IND-CPA security is tightly related to the hardness of the LWE problem. By applying a variant of

[1]Our implementation follows the FrodoKEM specification described in its Internet-Draft [LBES25] and ISO submission [ABD+25b].

**Algorithm 1** FrodoKEM Key Generation (KeyGen)

```
Input: –
Output: Public key $pk = (seed_A||b)$, secret key $sk = (s||seed_A||b||S^T||pkh)$
1: Choose uniformly random seeds $s$, $seed_{SE}$, and $z$ of bitlengths $len_{sec}$, $len_{SE}$, and $len_A$, respectively
2: Generate pseudorandom seed $seed_A \leftarrow \text{SHAKE}(z, len_A)$
3: Generate the matrix $A \leftarrow \text{Gen}(seed_A)$
4: Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \ldots, r^{(2n\bar{n}-1)}) \leftarrow \text{SHAKE}(0\text{x}5F||seed_{SE}, 32n\bar{n})$
5: Sample error matrix $S^T \leftarrow \text{SampleMatrix}((r^{(0)}, r^{(1)}, \ldots, r^{(n\bar{n}-1)}), \bar{n}, n)$
6: Sample error matrix $E \leftarrow \text{SampleMatrix}((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \ldots, r^{(2n\bar{n}-1)}), n, \bar{n})$
7: Compute $B \leftarrow AS + E$
8: Compute $b \leftarrow \text{Pack}(B)$
9: Compute $pkh \leftarrow \text{SHAKE}(seed_A||b, len_{sec})$
10: Return public key $pk \leftarrow seed_A||b$ and secret key $sk \leftarrow s||seed_A||b||S^T||pkh$
```

**Algorithm 2** FrodoKEM Encapsulation (Encaps)

```
Input: Public key $pk = seed_A||b$
Output: Ciphertext $c = c_1||c_2||salt$ and shared secret $ss$
1: Choose uniformly random values $u$ and $salt$ of bitlengths $len_{sec}$ and $len_{salt}$, respectively
2: Compute $pkh \leftarrow \text{SHAKE}(pk, len_{sec})$
3: Generate pseudorandom values $seed_{SE}||k \leftarrow \text{SHAKE}(pkh||u||salt, len_{SE} + len_{sec})$
4: Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \ldots, r^{2\bar{n}n+\bar{n}^2-1}) \leftarrow \text{SHAKE}(0\text{x}96||seed_{SE}, 16(2\bar{n}n + \bar{n}^2))$
5: Sample error matrix $S' \leftarrow \text{SampleMatrix}((r^{(0)}, r^{(1)}, \ldots, r^{(\bar{n}n-1)}), \bar{n}, n)$
6: Sample error matrix $E' \leftarrow \text{SampleMatrix}((r^{(\bar{n}n)}, r^{(\bar{n}n+1)}, \ldots, r^{(2\bar{n}n-1)}), \bar{n}, n)$
7: Generate the matrix $A \leftarrow \text{Gen}(seed_A)$
8: Compute $B' \leftarrow S'A + E'$
9: Compute $c_1 \leftarrow \text{Pack}(B')$
10: Sample error matrix $E'' \leftarrow \text{SampleMatrix}((r^{(2\bar{n}n)}, r^{(2\bar{n}n+1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}), \bar{n}, \bar{n})$
11: Compute $B \leftarrow \text{Unpack}(b, n, \bar{n})$
12: Compute $V \leftarrow S'B + E''$
13: Compute $C \leftarrow V + \text{Encode}(u)$
14: Compute $c_2 \leftarrow \text{Pack}(C)$
15: Compute $ss \leftarrow \text{SHAKE}(c_1||c_2||salt||k, len_{sec})$
16: Return ciphertext $c \leftarrow c_1||c_2||salt$ and shared secret $ss$
```

the Fujisaki–Okamoto (FO) transform [FO99], FrodoKEM upgrades FrodoPKE into an IND-CCA-secure KEM.

There are two variants of FrodoKEM, which are determined by the pseudorandom number generator (PRNG) that is used for the generation of a public matrix called $A$. The first variant uses AES128 [Nat01], and the other SHAKE128 [Nat15]. Likewise, FrodoKEM consists of two main variants: 1) ephemeral FrodoKEM (or eFrodoKEM) that is intended for scenarios in which key pairs are reused a fairly small number of times, and 2) standard FrodoKEM, which includes countermeasures against multi-ciphertext attacks and, hence, does not impose any restriction on the reuse of key pairs. Finally, the FrodoKEM specification provides three parameter sets, namely FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344, which correspond to NIST security levels L1, L3 and L5, respectively. Overall, considering the different variants above and their combinations, there are twelve (12) parameter sets. See [GLN+25, Section 6.2] for more details.

Like other KEM schemes, FrodoKEM consists of three main primitives: key generation, encapsulation, and decapsulation. The algorithms corresponding to these primitives are presented in Algorithm 1, Algorithm 2, and Algorithm 3, respectively. All the essential parameters used in FrodoKEM which we will refer to in subsequent sections, are tabulated in Table 1[2]. Section 5 includes a brief description of the main building blocks that are targeted in the implementation. For full details on notation and the internal algorithms, refer to [GLN+25, Sections 3 and 5].

[2]As noted in Table 1, in contrast to eFrodoKEM, standard FrodoKEM includes the use of a salt, *salt*, during encapsulation and decapsulation, and an enlarged seed $seed_{SE}$.

**Algorithm 3** FrodoKEM Decapsulation (Decaps)

```
Input: Ciphertext $c = c_1||c_2||salt$, secret key $sk = s||seed_A||b||S^T||pkh$
Output: Shared secret $ss$
1: Compute $B' \leftarrow \text{Unpack}(c_1, \bar{n}, n)$
2: Compute $C \leftarrow \text{Unpack}(c_2, \bar{n}, \bar{n})$
3: Compute $M \leftarrow C - B'S$
4: Compute $u' \leftarrow \text{Decode}(M)$
5: Generate pseudorandom values $seed'_{SE}||k' \leftarrow \text{SHAKE}(pkh||u'||salt, len_{SE} + len_{sec})$
6: Generate pseudorandom bit string $(r^{(0)}, r^{(1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}) \leftarrow \text{SHAKE}(0x96||seed'_{SE}, 16(2\bar{n}n + \bar{n}^2))$
7: Sample error matrix $S' \leftarrow \text{SampleMatrix}((r^{(0)}, r^{(1)}, \ldots, r^{(\bar{n}n-1)}), \bar{n}, n)$
8: Sample error matrix $E' \leftarrow \text{SampleMatrix}((r^{(n\bar{n})}, r^{(n\bar{n}+1)}, \ldots, r^{(2n\bar{n}-1)}), \bar{n}, n)$
9: Generate the matrix $A \leftarrow \text{Gen}(seed_A)$
10: Compute $B'' \leftarrow S'A + E'$
11: Sample error matrix $E'' \leftarrow \text{SampleMatrix}((r^{(2\bar{n}n)}, r^{(2\bar{n}n+1)}, \ldots, r^{(2\bar{n}n+\bar{n}^2-1)}), \bar{n}, \bar{n})$
12: Compute $B \leftarrow \text{Unpack}(b, n, \bar{n})$
13: Compute $V \leftarrow S'B + E''$
14: Compute $C' \leftarrow V + \text{Encode}(u')$
15: (In constant time) $\bar{k} \leftarrow k'$ if $B'||C = B''||C'$ else $\bar{k} \leftarrow s$
16: $ss \leftarrow \text{SHAKE}(c_1||c_2||salt||\bar{k}, len_{sec})$
17: Return shared secret $ss$
```

Table 1: Parameter sets of FrodoKEM [LBES25]. For ephemeral mode, $len_{salt} = 0$ and $len_{SE}$ is halved to 128, 192 and 256 for FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344, respectively.

| **Parameter Set** | $q$ | $\chi$ | $n$ | $\bar{n}$ | $B$ (bits) | $len_A$ (bits) | $len_{sec}$ (bits) | $len_{SE}$ (bits) | $len_{salt}$ (bits) | SHAKE |
|---|---|---|---|---|---|---|---|---|---|---|
| FrodoKEM-640 | 32,768 | 5 | 640 | 8 | 2 | 128 | 128 | 256 | 256 | SHAKE128 |
| FrodoKEM-976 | 65,536 | 5 | 976 | 8 | 3 | 128 | 192 | 384 | 384 | SHAKE256 |
| FrodoKEM-1344 | 65,536 | 4 | 1,344 | 8 | 4 | 128 | 256 | 512 | 512 | SHAKE256 |

## 3 Related work

FrodoKEM has seen relatively limited exploration in terms of full hardware implementation. To the best of our knowledge, only three such implementations exist in the literature to date [HOKG18, HMOR21, DGA25]. The work by Howe et al. [HOKG18] introduced the first hardware implementation of FrodoKEM targeting resource-constrained devices. Their FPGA-based design follows a low-area architecture, utilizing a single multiply-and-accumulate (MAC) unit (implemented as a DSP) within the matrix multiplication unit to sequentially perform all required multiplications and additions. While this design does not provide tunable performance parameters, it includes a synthesis-time parameter that allows switching between two parameter sets, FrodoKEM-640 and FrodoKEM-976. The implementation adheres to the original FrodoKEM Round 1 specification [ABD+17], employing cSHAKE and AES for PRNG tasks, such as public matrix generation and error distribution matrix construction. Given the memory bottleneck of storing the public matrix $A$, Howe et al. opted not to store $A$ in its entirety.

A subsequent work [HMOR21] builds upon [HOKG18] by addressing this memory bottleneck through on-the-fly generation of the public matrix $A$. Additionally, it introduces a synthesis-time parameter that allows increasing the number of MAC units within the matrix multiplication unit to increase parallelization and, thus, reduce the total number of clock cycles. However, in contrast to the FrodoKEM specification, Howe et al. [HMOR21] replace cSHAKE and AES with the hardware-friendly Trivium cipher for PRNG operations. This design, like its predecessor, also includes a synthesis-time parameter to switch between FrodoKEM-640 and FrodoKEM-976. Additionally, it also implements first-order masking to the decapsulation operation $M = C - B'S$ by reusing the parallelized matrix multiplier.

More recently, Düzyol et al. [DGA25] presented a specification-compliant FrodoKEM hardware implementation that employs operation-specific parallelization strategies to accelerate large matrix multiplication. Their design uses a $3 \times 8$ architecture for key generation, which parallelizes the $AS$ computation using 24 multipliers operating on 3 elements per row of $A$ and all 8 columns of $S$ simultaneously. For encapsulation and decapsulation, they adopt an $8 \times 2 \times 2$ parallelization scheme for the $S'A$ computation, that is, utilizing 32 multipliers to process 2 elements per row across 2 rows of $A$ and all 8 rows of $S'$ concurrently. Their memory architecture stores each column of $S/S'$ in dedicated BRAM blocks to support parallel access patterns. Like our design, they generate matrix $A$ on-the-fly during computation to minimize memory footprint; however, their approach maintains dual copies of each row in double-buffered storage to sustain the required throughput for their parallelization scheme.

In addition to full hardware implementations, alternative hardware-based approaches for FrodoKEM have been explored, including HW/SW co-design [DFA⁺20, KFS22] and high-level synthesis (HLS) implementations [Bus21]. The HW/SW co-design approach proposed in [DFA⁺20] and [KFS22] focuses on optimizing the computationally intensive matrix multiplication by offloading it to dedicated hardware while handling all other operations on the processor. In contrast, the HLS-based implementation from [Bus21] transforms the software implementation accompanying FrodoKEM's NIST submission [ABD⁺25a], into a hardware design using HLS techniques. This work shows that the results based on HLS are competitive with the full hardware implementation from [HOKG18]. In addition to that, more recently [UMVM25] extended the HLS approach to FrodoKEM's ISO submission [ABD⁺25b] by employing profiling-guided optimization in which they demonstrated 34% reduction in clock cycles for key generation compared to the reference software implementation.

Besides hardware implementations, FrodoKEM has also been implemented on embedded devices. A notable example is given by [BBC⁺23], which focuses on optimizing execution on resource-constrained platforms. Broadly, these embedded implementations address the following challenges: storing the large public matrix $A$ can be impractical in constrained environments. Hence, generating $A$ on-the-fly (when performing the matrix multiplications $AS$ and $S'A$) instead of storing it explicitly is preferred. This significantly reduces the memory footprint and helps make FrodoKEM feasible on devices with limited storage. Additionally, evaluation of generating (comparatively smaller error distribution matrices) $S$, $E$, $S'$, and $E'$ on-the-fly is explored. However, the generation of all the required matrices needs careful computation scheduling and the use of efficient memory access patterns which can lead to performance bottlenecks. To address this concern, two different matrix multiplication techniques are discussed in [BBC⁺23] depending on where matrix $A$ appears in the matrix multiplication: 1) When $A$ appears on the left (in $AS$), schoolbook matrix multiplication is used; 2) when $A$ appears on the right (in $S'A$), then a row-by-chunk method is applied.

In our work, we integrate different suitable techniques from the literature to develop a full standard-compliant, high-speed FrodoKEM implementation. Our design adheres to the specification by utilizing SHAKE and AES for all the PRNG requirements. Additionally, we adopt the on-the-fly generation of the public matrix $A$ and optimize our matrix multiplication accordingly to improve efficiency. Unlike existing hardware implementations that handle key generation, encapsulation, and decapsulation as separate modules, our approach is to unify these operations into a single design by efficiently sharing resources. Furthermore, our implementation supports run-time configurability across all twelve parameter sets, covering three security levels (L1, L3, L5), two PRNG variants (SHAKE-based and AES-based), and both standard and ephemeral modes, and offers synthesis-time configurability to optimize performance based on specific design constraints.

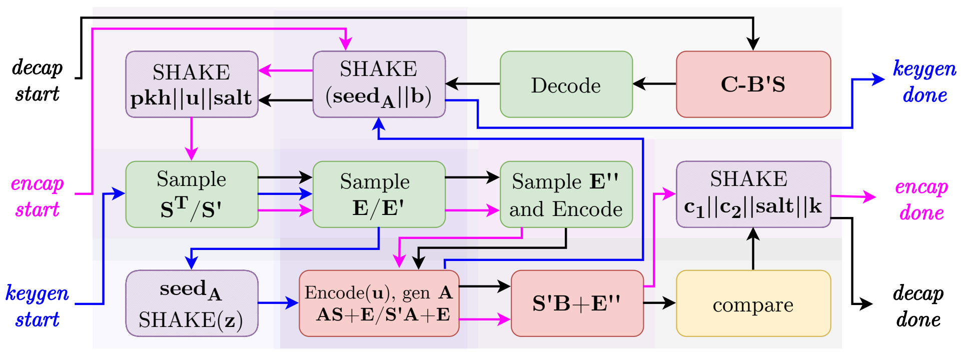

Figure 1: Overview of the operational flow of our combined keygen, encap, and decap module, blue lines for keygen, magenta lines for encap, and black lines for decap.

## 4 Proposed architecture design

This section presents a top-down overview of the architecture and operation of the FrodoKEM hardware implementation. A key feature of our design is the efficient integration of key generation, encapsulation and decapsulation into a unified architecture; Figure 1 gives an overview of the operational flow. The datapath for key generation is shown in blue, for encapsulation in magenta, and for decapsulation in black. As can be seen, several modules such as the matrix arithmetic, sampling, and hashing, are efficiently shared by all three primitives.

Our unified construction implements the following modules: `Matrix Arithmetic` module (described in Section 5.2), which performs matrix multiplication, addition, and subtraction; `encode` and `decode` modules (described in Section 5.3) for performing encoding and decoding operations; `sampler` module for generating the error distribution matrices (described in Section 5.1); and `mem compare` module, which performs comparison of two memory contents for ciphertext validation during decapsulation.

In addition, our architecture exhibits the following features.

**Synthesis-time performance configurability.** Our design objectives prioritize both high throughput and area efficiency, targeting practical deployment scenarios across multiple hardware platforms. Central to achieving these goals is the configurable performance parameter `T`, which determines the degree of parallelism in `Matrix Arithmetic` module (described in Section 5.2) by specifying how many dot products are computed simultaneously per clock cycle. By adjusting `T` at synthesis time, we can flexibly trade area for throughput: larger `T` values deploy more multiply-and-accumulate (MAC) blocks and require proportionally more memory bandwidth, achieving higher operations per second at the cost of increased resource utilization.

**Dynamic security level and parameter selection.** Our hardware implementation features a unified design that supports, at run-time, all the parameter sets (and, hence, all the security levels) specified in FrodoKEM's Internet-Draft [LBES25]. This approach enables dynamic parameter selection, providing full flexibility in practical deployments. The motivation behind this design choice is to develop a flexible hardware architecture suitable for both FPGA and ASIC implementations. Since ASIC designs are inherently fixed, integrating multiple parameter sets within a single module enhances its applicability across various use cases.

### 4.1 Operational flow and scheduling

Next, we describe the operational flow and scheduling of each primitive controller.

- Key generation. In key generation (Algorithm 1), the `sampler` module utilizes $seed_{SE}$ to produce pseudorandom bits. These bits undergo inversion sampling based on a fixed distribution table (detailed in Section 5.1) to generate elements of the matrix $S^T$. The same procedure is applied to generate the matrix $E$. Next, $z$ is input into SHAKE to derive $seed_A$, which is then fed into SHAKE or AES modules depending on the choice of parameter set. The generation of matrix $A$ and the computation of $b = AS + E$ occur simultaneously, as outlined in Section 5.2. Finally, $seed_A||b$ is hashed using a SHAKE module to produce $pkh$.

- Encapsulation. In encapsulation (Algorithm 2), the public key input ($seed_A||b$) is hashed using SHAKE to derive $pkh$, which along with uniformly random values $u$ and $salt$, are then hashed to generate the pseudorandom values $seed_{SE}$ and $k$ (note that this $seed_{SE}$ differs from the one generated during key generation). Next, pseudorandom bits are generated from $0x96||seed_{SE}$, and the matrices $S', E', E''$ are sampled using the `sampler` module. Then, the operation $encoded_u = Encode(u)$ (described in Section 5.3), the generation of matrix $A$, and the computation of $c_1 = S'A + E'$ are all executed in parallel. Then, the operation $V = S'B + E''$ is performed using the `Matrix Arithmetic` module. And finally, $c_2 = V + encoded_u$ is computed, and hashing of $ss = c_1||c_2||salt||k$ is performed using the SHAKE module to generate the shared secret ($ss$).

- Decapsulation. In decapsulation (Algorithm 3), the process begins with matrix multiplication and subtraction, computing $M = C - B'S$, followed by the operation $u' = Decode(M)$. After this step, all encapsulation operations are repeated. However, before the final hashing step of $ss = c_1||c_2||salt||\bar{k}$, the input ciphertext is validated by comparing it against the recomputed ciphertext.

Our hardware design assumes that there is an external source of randomness (e.g., a true random number generator) that provides the required random values, i.e., $s$, $seed_{SE}$, $z$, $u$, and $salt$ in Line 1 of Algorithm 1 and Algorithm 2. We also note that in our design, we do not need to perform any special Pack or Unpack operation shown in Algorithm 1, Algorithm 2, and Algorithm 3. As the data is loaded from memory (e.g., BRAM), we arrange the bits in packed form as per the specification.

## 5 Implementation

This section describes the different building blocks and sub-modules in the architecture, including pseudorandom number generation and hashing, matrix arithmetic, and encoding and decoding. We note that all the arithmetic operations in FrodoKEM happen in $\mathbb{Z}_q$, with $q$ chosen according to the parameter set (see Table 1). For our unified design, we set $q$ to its largest bitlength, i.e., 16 bits. For the operations involving a smaller $q$ (i.e., in the case of FrodoKEM-640), the MSB is unused and fixed to value 0.

For evaluating the different modules, and later on to evaluate the full implementation, we use the following platforms.

**FPGA implementation.** For FPGA evaluation, we target the AMD Zynq Ultrascale+ `xczu7ev` device, a high-performance platform fabricated using 16 nm FinFET technology. We synthesize, implement, and analyze our design using the AMD Vivado Design Suite (version 2023.2). All area, timing, and power statistics reported in Tables 4 and 5 are extracted from post-implementation reports, ensuring accurate resource utilization and timing closure. The design operates at frequencies ranging from 168 to 193 MHz across different `T` configurations.

**ASIC implementation.** To evaluate our design's viability for custom silicon implementations and establish a baseline for future ASIC-based deployments, we perform a complete ASIC synthesis and place-and-route flow using open-source tools and libraries. We select the NANGATE45[3] Open Cell Library [Nan11], a widely adopted 45 nm technology library that provides comprehensive standard cell characterization.The complete RTL-to-GDSII flow is executed using the OpenROAD [ABC+19] toolchain, an open-source platform that integrates synthesis, floorplanning, placement, clock tree synthesis, routing, and timing analysis. For on-chip memory generation, we utilize the Bespoke Silicon Group's FakeRAM generator [Bes24], which produces timing-accurate SRAM models suitable for evaluating the physical design.

## 5.1 Pseudorandom number generation and hashing

For the deterministic generation of random bit sequences, as required by the hashing operations and sampling of error distribution matrices, FrodoKEM uses the SHA-3-based extendable output function SHAKE [Nat15]. The specific SHAKE variant that is used per parameter set is outlined in Table 1. As shown in the Figure 3, the top-level FrodoKEM design interfaces with an external SHAKE module, which follows a standard ready-valid protocol, ensuring efficient data exchange.

**Generation of the public matrix $A$, function Gen().** FrodoKEM also requires a PRNG to generate the public matrix $A$. The public matrix $A$ consists of pseudorandom elements generated from the PRNG using a seed, $seed_A$, and placed in a row-major fashion. The FrodoKEM specification provides two PRNG options: one based on AES128 and another using SHAKE128. The method for generating $A$ differs between these two approaches, and the exact algorithm can be found in [GLN+25, Section 3.2]. In our hardware design, we support both PRNG variants which can be selected at run-time, providing flexibility based on performance and implementation constraints (we call the respective protocols SHAKE-based and AES-based FrodoKEM). In our implementation, the interface for the AES128 module is similar to the one provided for SHAKE.

**Sampling of error distribution matrices, function SampleMatrix().** The process for generating the sampled error matrices ($S, E, S', E', S'', E''$ in Algorithm 1, Algorithm 2 and Algorithm 3) is as follows. The SHAKE module is initialized with a constant 8-bit value prepended to $seed_{SE}$ as shown in line 4 of Algorithm 1 and Algorithm 2, and line 6 of Algorithm 3. Once the pseudorandom bits from SHAKE are received, they must be post-processed to align with the required error distribution. This is achieved through a sample function that maps each 16-bit random string $r^{(i)} = \{r_0, r_1, \ ..., r_{15}\}$ to a discrete error value to sample a matrix coefficient, as described in [GLN+25, Sections 2.2.4 and 2.2.5]. For this, inversion sampling is applied using a predefined distribution table $T_\chi$ that represents FrodoKEM's error distribution (the tables $T_\chi$ for each parameter set can be found in [ABD+25b, Table A.4]). In our hardware design, we implement a `sampler` module using a table lookup method since the selection range is limited, from 0 to 15, making it an efficient and resource-effective approach. We also provide a performance parameter $\mathtt{T}_{sample}$, which determines the number of elements that are sampled in parallel. As shown in Figure 3, the sampled outputs are stored in a single port memory.

**AES128 and SHAKE implementations.** While open-source implementations of AES-128 [New24] and SHAKE [DLK+25] exist in the literature, we developed custom implementations tailored to our architecture's specific throughput requirements. The primary

[3]We select the NANGATE45 (45 nm) [Nan11] over more advanced libraries, ASAP7 (7 nm) [CVS+16] and FreePDK3 (3 nm) [NC 23] due to its robust OpenROAD support and compatibility with open-source SRAM generators (BSG FakeRAM [Bes24]). Since FrodoKEM requires substantial on-chip memory, accurate SRAM modeling is essential; neither advanced library provides this capability. While NANGATE45 yields conservative estimates, it enables a fully reproducible flow and establishes a credible baseline.

Table 2: Time and area estimates for our AES and SHAKE modules used in the FrodoKEM implementation. `Unroll` refers to the number of rounds unrolled from the round function. *Full+Sub means fully unrolled and pipelined; additionally, each AES round is sub-pipelined with 4 registers. +Reported numbers are in terms of slices instead of LUTs. *Supports four SHA3 modes, SHAKE128, and SHAKE256.

| `Unroll` | **PRNG** | Area | | **Cycles** | **Freq** | **Time** | **Throughput** |
|---|---|---|---|---|---|---|---|
| | | **LUT** | **FF** | | (MHz) | ($\mu s$) | (Gbps) |
| | | | **FPGA Design** | | | | |
| | | **Device: AMD Zynq Ultrascale+ (`xczu7ev`)** | | | | | |
| 1 | SHAKE128/SHAKE256 | 4,086 | 1,621 | 25 | 500 | 0.050 | 26.88/21.76 |
| 2 | SHAKE128/SHAKE256 | 7,725 | 1,621 | 13 | 454 | 0.029 | 46.94/38.00 |
| 3 | SHAKE128/SHAKE256 | 11,157 | 1,621 | 9 | 364 | 0.025 | 54.36/44.00 |
| Full | AES128 | 7,569 | 2,698 | 1 | 450 | 0.002 | 57.60 |
| | | **Device: AMD Artix 7 (`xc7a200t`)** | | | | | |
| 1 | SHAKE256 [DLK+25] | 4,383 | 2,708 | 24 | 250 | 0.096 | 11.33 |
| | | **Device: Xilinx Virtex 7 (`xc7v2000t`)** | | | | | |
| 1 | SHAKE128/SHAKE256* [CS25] | 1,364+ | | 26 | 437 | – | 19.35/18.28 |
| | | **Device: Xilinx Virtex 7** | | | | | |
| 1 | SHAKE256 [SHTW23] | 1,421+ | | 26 | 294 | – | 12.80 |
| | | **Device: Xilinx Virtex 6 (`xc6vlx240t`)** | | | | | |
| Full + Sub* | AES128 [SS15] | 28,520+ | | – | 804 | – | 102.91 |
| | | **Device: Zynq UltraScale+ (`xczu9eg`)** | | | | | |
| Full | AES128 [VCV+20] | 15,029 | 4,296 | 1 | 220 | – | 28.16 |

**ASIC Design**

| **Unroll** | **PRNG** | Area | | **Cycles** | **Freq** | **Time** | **Throughput** |
|---|---|---|---|---|---|---|---|
| | | **LUT** | **FF** | | (MHz) | ($\mu s$) | (Gbps) |
| | | **Library: NANGATE45** | | | | | |
| 1 | SHAKE128/SHAKE256 | 0.055 | | 25 | 1,050 | 0.024 | 56.45/45.70 |
| 2 | SHAKE128/SHAKE256 | 0.093 | | 13 | 960 | 0.014 | 99.25/80.34 |
| 3 | SHAKE128/SHAKE256 | 0.145 | | 9 | 660 | 0.014 | 98.56/79.79 |
| Full | AES128 | 0.114 | | 1 | 980 | 0.001 | 125.44 |
| 1 | Keccak [GOW+24] | 0.045 | | 26 | 1,316 | 0.020 | 80.98 |
| 1 | AES128 [PdCKN16] | 0.062 | | – | 1,550 | 0.065 | 4.84 |

motivation is to satisfy the pseudorandom bit generation bandwidth needed for on-the-fly matrix $A$ generation during matrix multiplication (computations $AS$ and $S'A$), as detailed in Section 5.2. To eliminate stalls in the parallelized matrix multiplication unit, our AES128 module employs a fully unrolled and pipelined architecture across all `T` configurations, producing one complete block per clock cycle. For SHAKE, we implement variable unrolling factors matched to the parallelism level: `T = 16` uses a round-based iterative architecture (i.e., one Keccak round per cycle), `T = 32` employs `Unroll = 2` (i.e., two rounds per cycle), and `T = 64` uses `Unroll = 3` (i.e., three rounds per cycle). This design ensures that the PRNG throughput scales proportionally with the matrix multiplication parallelism, preventing the pseudorandom generation from becoming a performance bottleneck.

Table 2 presents area and timing results for our custom SHAKE and AES implementations, in comparison with some designs from the literature. Our implementations achieve competitive or superior performance across both area and time metrics. For example, our SHAKE implementation, which employs an architecture similar to [DLK+25], demonstrates improved area efficiency and higher throughput despite using the same round-based iterative approach. Similarly, our fully pipelined AES128 module achieves single-cycle

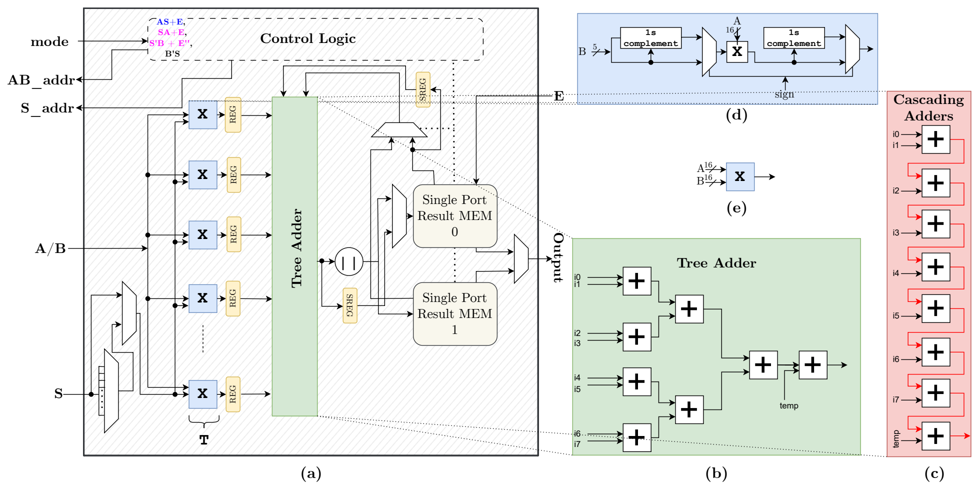

Figure 2: a) Hardware design of the `Matrix Arithmetic` module; b) Tree Adder structure for accumulating the partial products; c) Cascading Adders which were eventually replaced by Tree Adder for frequency optimization; d) optimized dot product multiplier for our ASIC design; e) DSP-based multiplier design used in our FPGA Design.

block generation while maintaining comparable or lower resource utilization than prior fully unrolled implementations. These optimizations ensure that PRNG throughput scales appropriately with our parameterized `Matrix Arithmetic` module.

## 5.2 Matrix arithmetic

### 5.2.1 Matrix multiplication

Together with the generation of matrix $A$, the matrix multiplications with $A$ are the most expensive operations in FrodoKEM, dominating execution time in key generation ($AS+E$), encapsulation ($S'A+E'$), and decapsulation ($S'A+E'$). The design of an efficient matrix multiplication architecture requires careful consideration of memory organization and data flow.

As described in Section 4, the error distribution matrices $S$, $S'$, $E$, and $E'$ are sampled once per operation and stored in on-chip memory (BRAM for FPGA, SRAM for ASIC), enabling repeated random access during computation. In contrast, the public matrix $A \in Z_q^{n\times n}$ is generated dynamically on-the-fly. This design choice is driven by memory constraints: e.g., for FrodoKEM-640, storing $A$ would require $640 \times 640 \times 15 = 6.14$ Mbit, far exceeding practical on-chip memory budgets. Instead, we generate $A$ row-by-row using a pseudorandom generator seeded with $seed_A$, consuming each row during matrix multiplication before generating the next one. To prevent PRNG generation latency from stalling the arithmetic pipeline, we implement a dual-memory scheme with two row-sized memory units: `MEM A Matrix Row 0` and `MEM A Matrix Row 1` (shown in Figure 3).

**Dual-memory scheme flow.** When the PRNG generates the first row of $A$, it is written to `MEM A Matrix Row 0`, triggering the matrix multiplication unit to start the computation. Concurrently, while the arithmetic unit consumes data from `MEM A Matrix Row 0`, the PRNG generates the second row and writes it to `MEM A Matrix Row 1`. Upon completing the first row's computation, the multiplier seamlessly switches to `MEM A Matrix Row 1` and continues processing. The memories then alternate roles: as `MEM A Matrix Row 1` feeds the multiplier, the PRNG refills `MEM A Matrix Row 0` with the third row. This ping-pong continues throughout the $n$ rows of $A$, completely hiding PRNG latency provided generation completes within the multiplication time for one row. Communication between the matrix multiplication unit and PRNG module employs a standard ready-valid handshake protocol, ensuring correct synchronization and preventing memory overwrites or starvation. The multiplier asserts `ready` when prepared to consume a new row, and the PRNG asserts `valid` upon completing row generation, with data transfer occurring when both signals are high.

**Modes of operation.** Our matrix multiplication unit includes a mode selector port, allowing the operation to switch between different multiplication variants. This flexibility is necessary because $A$ is generated on-the-fly, affecting how the multiplication is performed. The following cases are covered.

1. Key generation ($AS$): $A$ appears on the left-hand side of the multiplication and is generated in row-major fashion. Accordingly, we rely on a conventional matrix multiplication method that multiplies the rows of $A$ with the columns of $S$, accumulating the sum to form the resultant matrix. Since $A$ is generated dynamically, each row of $A$ is processed sequentially.
2. Encapsulation and decapsulation ($S'A$): Having $A$ on the right-hand side of the expression is more challenging, as we still need to generate $A$ in a row-major format. To address this, we implement a row-wise memory-friendly method, where our multiplication unit computes partial sums for each location in the resultant matrix using each row of $A$ as it is generated. Instead of computing the entire multiplication at once, we maintain a rolling accumulation of intermediate results. A similar technique has been discussed in [BBC+23], and they call it the row-by-chunk matrix multiplication method.
3. Other small matrix multiplications ($BS + E$, $CB - E$): For smaller multiplications, matrices are available fully in memory, as mentioned before for the case of error matrices $S$ and $E$. Accordingly, we follow the conventional matrix multiplication method that multiplies the rows of the matrix on the left with the columns on the right and accumulates the sum sequentially.

**Hardware architecture.** Direct implementation of the matrix operations would require computing all dot products simultaneously, resulting in significant hardware costs. For instance, computing $B = AS$ for FrodoKEM-640 would necessitate $640 \times 8 = 5,120$ parallel multiply-accumulate operations, far exceeding practical resource budgets. To achieve a scalable area-throughput trade-off, we introduce a parameterized architecture that performs `T` parallel dot products per cycle, where `T` is a synthesis-time configurable parameter. Larger `T` values increase throughput proportionally but require additional DSP blocks and memory bandwidth.

Partial results during accumulation are stored in `RESULT RAM 0` and `RESULT RAM 1` (refer to Figure 2a). We employ two single-port RAMs rather than one dual-port RAM to ensure design portability across FPGA and ASIC platforms. On FPGAs, single-port and dual-port BRAMs have similar area footprints, as both configurations utilize the same underlying BRAM primitives. However, on ASICs, single-port SRAM compilers generate significantly smaller[4] and lower-power memories compared to dual-port equivalents. Since our design requires only one read and one write per cycle to different banks, two single-port RAMs provide equivalent functionality without a performance penalty while maintaining optimal area across both platforms.

[4]Multi-port capabilities in SRAM bitcell designs quadratically increase the bitcell size with the number of access ports, and it also contributes to an increase in the power consumption due to an increase in circuitry. [SMP14, Chapter 4]

**Computational complexity.** The matrix multiplication $B = AS + E$ involves matrices $A \in \mathbb{Z}_q^{n \times n}$ and $S \in \mathbb{Z}_q^{n \times \bar{n}}$, where, e.g., $n = 640$ and $\bar{n} = 8$ for FrodoKEM-640. The total number of multiply-accumulate operations is $n^2 \times \bar{n}$. With `T` parallel multipliers, the computation requires $\lceil n^2 \times \bar{n}/\texttt{T} \rceil$ clock cycles. Including one initialization cycle to load operands and one finalization cycle to signal completion to the control FSM, the total latency is:

$$\text{cycles}_{\text{mat\_mul}} = 2 + \left\lceil \frac{n^2 \times \bar{n}}{\texttt{T}} \right\rceil \tag{1}$$

For example, FrodoKEM-640 key generation with $\texttt{T} = 16$ requires $2 + \lceil 3,276,800/16 \rceil = 204,802$ cycles (see Table 3). This latency dominates the overall operation time as shown in Tables 5 and 6.

**PRNG throughput requirement.** To achieve seamless on-the-fly generation of matrix $A$ without stalling the matrix multiplication pipeline, the PRNG must generate each row within the time required to process the previous row. Formally, let $C_{\text{row\_gen}}$ denote the cycles to generate one row of $A$, and $C_{\text{row\_mult}}$ denote the cycles to multiply one row of $A$ with $S$ or $S'$. The no-stall condition requires:

$$C_{\text{row\_gen}} \leq C_{\text{row\_mult}} = \frac{n \times \bar{n}}{\texttt{T}} \tag{2}$$

Considering SHAKE-based FrodoKEM, generating one row of $A$ requires $n$ elements of 16 bits each. SHAKE outputs 1344 bits (for SHAKE128) and 1088 bits (for SHAKE256) per squeeze operation. Therefore, using a configurable number of Keccak rounds, the generation latency is given by:

$$C_{\text{row\_gen}}^{\text{SHAKE}} = \left\lceil \frac{n \times 16}{\text{SHAKE\_OUTPUT\_SIZE}} \right\rceil \times \left(\frac{24}{\texttt{Unroll}} + 1\right) \tag{3}$$

where $\texttt{Unroll} \in \{1, 2, 3\}$ determines how many Keccak rounds are executed per cycle. The latency of our SHAKE module is given in Table 2. Please note that the PRNG latency for SHAKE is considered as $(\frac{24}{\texttt{Unroll}} + 1)$ because it includes registered output.

**Example with FrodoKEM-640 with $\texttt{T} = 16$.** For $n = 640$, $\bar{n} = 8$, and $\texttt{T} = 16$:

- $C_{\text{row\_mult}} = 640 \times 8/16 = 320$ cycles
- $C_{\text{row\_gen}}^{\text{SHAKE}} = \lceil 640 \times 16/1344 \rceil \times 25 = 8 \times 25 = 200$ cycles (with Unroll = 1)

Since $200 < 320$, the condition is satisfied and no stalls occur. This holds for all security levels at $\texttt{T} = 16$ with Unroll = 1. For $\texttt{T} = 32$ and $\texttt{T} = 64$, we require Unroll = 2 and Unroll = 3, respectively, to maintain $C_{\text{row\_gen}} \leq C_{\text{row\_mult}}$.

**AES-128 bandwidth constraints.** For AES-based FrodoKEM, the small block size of AES128 (128-bits) necessitates high-frequency block generation to sustain matrix multiplication throughput. For $\texttt{T} = 16$, we employ a fully unrolled and pipelined AES128 implementation that produces one block per cycle, which satisfies the throughput requirements. However, as `T` increases to 32 or 64, a single AES128 module becomes insufficient to maintain the required bit generation rate, causing the matrix multiplier to stall. This bottleneck is evident in the performance results reported in Tables 5 and 6, where throughput scaling becomes sublinear for higher `T` values. Deploying multiple fully unrolled and pipelined AES128 modules in parallel would eliminate this bottleneck and restore linear performance scaling. For consistency in our evaluation, we restrict our design to a single AES128 module to enable fair comparison across all variants.

**Optimizing the dot product multiplier for ASIC design.** We note that the error distribution matrices ($S$, $S'$, $E$, $E'$, $E''$) contain elements from $[-d, d]$ where $d \in \{12, 10, 6\}$ for FrodoKEM-640/976/1344, respectively. This bounded range enables multiplier optimization. On Ultrascale+ FPGAs, DSP blocks perform fixed $27 \times 18$-bit multiplication regardless of operand width, offering no area benefit from specialization. However, on ASICs with synthesized multipliers, we exploit the narrow range by implementing $16 \times 5$-bit signed multipliers rather than full $16 \times 16$-bit units (shown in Figure 2d). Sign handling is separated from magnitude multiplication to avoid extension overhead. This optimization achieves approximately 20% area reduction in the ASIC matrix multiplication unit across different `T` values (refer to Table 3), plus some frequency improvement as well.

**Critical path optimization via tree-based adder.** We observe that the bottleneck in achieving high operating frequency arises within the matrix multiplication unit, specifically in the accumulation stage following parallel multiply-accumulate operations. Our initial implementation employed a cascading (ripple-carry) adder chain (shown in Figure 2c) to sum the `T` partial products, placing all `T` additions sequentially in the critical path. This linear accumulation structure created a timing bottleneck that limited maximum clock frequency, particularly for larger `T` values.

To mitigate this issue, we replaced the cascading adder with a logarithmic-depth tree adder (shown in Figure 2b). This architectural modification reduces the critical path from `T` sequential additions to $\lceil \log_2(\mathtt{T}) \rceil$ adder levels. For example, with $\mathtt{T} = 8$, the critical path shrinks from 8 cascaded additions to only 3 logarithmic stages, significantly improving timing slack and enabling higher operating frequencies. This optimization is particularly effective as `T` scales, allowing our design to maintain competitive clock rates even at high parallelism levels (shown in Table 3).

**A note on why we did not use Strassen's algorithm for matrix multiplication.** Although Strassen's algorithm [Str21] offers better asymptotic complexity over traditional matrix multiplication ($O(n^{2.807})$ vs. $O(n^3)$), it is impractical for hardware implementation for the matrix sizes used in FrodoKEM. The recursive decomposition in Strassen's algorithm requires complex control logic and produces irregular memory access patterns that conflict with efficient BRAM/SRAM utilization. Additionally, it is incompatible with our on-the-fly streaming generation of matrix $A$, which relies on row-wise sequential processing to minimize memory footprint. In contrast, our direct multiplication approach provides simpler control logic, predictable timing for constant-time execution, and straightforward parallelization through the configurable `T` parameter.

**Alternative design considerations.** We explored two alternative architectural approaches to further optimize memory utilization. First, we evaluated storing only `T` elements of each matrix $A$ row in the dual-memory scheme, rather than buffering complete rows. While this approach would reduce BRAM/SRAM consumption from storing $n$ elements to just `T` elements per buffer, it introduces substantial synchronization overhead: the matrix multiplication unit must stall and wait for the PRNG to generate the next `T`-element chunk within each row, resulting in frequent pipeline bubbles that negate the memory savings.

Second, we considered generating matrix $S$ on-the-fly similarly to matrix $A$, which would eliminate the need to store $S$ in memory. However, this approach requires deploying two independent PRNG instances, one for $A$ and one for $S$, since both matrices must be accessed simultaneously during multiplication. Given that PRNG modules have a high area consumption (as shown in Table 2), doubling the PRNG resources would increase the total area significantly, exceeding the BRAM savings achieved by eliminating the storage of $S$, while also degrading throughput due to frequent PRNG synchronization. Therefore, we

Table 3: Time and area estimates for the Matrix Arithmetic module for performing a single selected operation among $A_{nxn}S_{nx\bar{n}} + E_{nx\bar{n}}$ or $S'_{\bar{n}xn}A_{nxn} + E_{\bar{n}xn}$, targeting AMD Ultrascale+ for FPGA design and NANGATE45 technology library for ASIC design.

$\ell_1 : n = 640, \bar{n} = 8$ / $\ell_3 : n = 976, \bar{n} = 8$ / $\ell_5 : n = 1,344, \bar{n} = 8$

| **FPGA Design, Target Device: Zynq Ultrascale+ (`xazu7ev`)** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Matrix** | | **Area** | | | | **Timing** | | |
| **Dimension** | `T` | **Logic** | | **Memory** | | **Cycles** | **Freq.** | **Time** |
| | | **LUT** | **DSP** | **FF** | **BRAM** | $(\times 10^3)$ | (MHz) | $(ms)$ |
| $\ell_1/\ell_3/\ell_5$ | 16 | 1,372 | 16 | 662 | 15 | 205/476/903 | 200 | 1.02/2.38/4.51 |
| $\ell_1/\ell_3/\ell_5$ | 32 | 2,382 | 32 | 1,170 | 15 | 102/238/451 | 185 | 0.56/1.29/2.45 |
| $\ell_1/\ell_3/\ell_5$ | 64 | 4,910 | 64 | 2,191 | 29 | 51/119/226 | 169 | 0.30/0.70/1.34 |

| **ASIC Design, Technology: NANGATE45** | | | | | | |
|---|---|---|---|---|---|---|
| **Matrix Dimension** | `T` | **Logic** $(mm^2)$ | **Memory** $(mm^2)$ | **Cycles** $(\times 10^3)$ | **Freq.** (MHz) | **Time** $(ms)$ |
| **Similar Design as FPGA (using widths 16×16 for dot products)** | | | | | | |
| $\ell_1/\ell_3/\ell_5$ | 16 | 0.028 | 0.397 | 205/476/903 | 439 | 0.47/1.08/2.06 |
| $\ell_1/\ell_3/\ell_5$ | 32 | 0.057 | 0.554 | 102/238/451 | 418 | 0.24/0.57/1.08 |
| $\ell_1/\ell_3/\ell_5$ | 64 | 0.111 | 1.018 | 51/119/226 | 403 | 0.13/0.30/0.56 |
| **Optimized for ASIC (using widths 16×5 for dot products)** | | | | | | |
| $\ell_1/\ell_3/\ell_5$ | 16 | 0.023 | 0.397 | 205/476/903 | 452 | 0.45/1.05/2.00 |
| $\ell_1/\ell_3/\ell_5$ | 32 | 0.044 | 0.554 | 102/238/451 | 425 | 0.24/0.56/1.06 |
| $\ell_1/\ell_3/\ell_5$ | 64 | 0.089 | 1.018 | 51/119/226 | 409 | 0.13/0.29/0.55 |

adopted the current architecture with full-row buffering for $A$ and pre-stored $S$ matrices as the optimal area-performance trade-off.

### 5.2.2 Matrix addition and subtraction

For matrix addition and subtraction operations such as $AS + E$, $SA + E$, and $C - BS$, we preload all the required values into the result memory `RESULT RAM 0` before initiating a matrix multiplication, and then proceed to perform the addition and subtraction in parallel to the multiplication itself, effectively achieving these operations without additional clock cycle overhead and with minimal extra hardware resources.

## 5.3 Encode and decode

As detailed in [GLN+25, Appendix B], the encode operation encodes bit strings of length $l = B \cdot \bar{n}^2$ as a $\bar{n} \times \bar{n}$ matrix, with each matrix element consisting of $B$-bit values in $\mathbb{Z}_q$. Specifically, each $B$-bit string in the input is encoded as a matrix coefficient by first interpreting its integer value and then multiplying it by $q/2^B$, with the bits read from least significant to most significant. In our hardware design, we implement this `encode` module using a shift register and a table lookup. We chose this approach over direct multiplication with $q/2^B$ because the value of $B$ in all parameter sets is small (see Table 1), which allows for a smaller multiplexer size and a more efficient hardware design. Additionally, we parameterize the number of matrix elements processed in parallel by the module.

The decode function performs the reverse operation of the encode operation. It takes $\bar{n} \times \bar{n}$ matrix (with each element consisting of $B$-bits in $\mathbb{Z}_q$ as input and generates a $B \cdot \bar{n}^2$ bit string. Specifically, each matrix coefficient is read in row-major format and decoded by dividing its integer value by $q/2^B$ and then rounding it to the nearest integer modulo

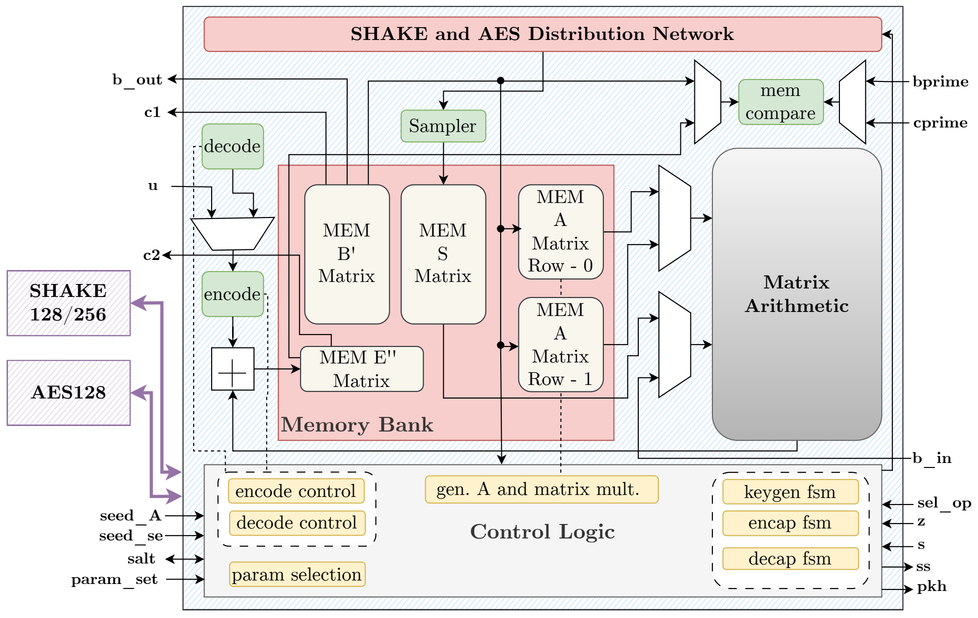

Figure 3: Top Level block design of our unified FrodoKEM hardware design.

$2^B$. Again, in our hardware design, we perform this using shift register, table look-up, and comparators (for rounding to the nearest integer).

## 6 Top-level datapath and controller design

The top-level hardware architecture of our FrodoKEM module is depicted in Figure 3. This module integrates all the underlying components detailed in Section 5, producing a unified design capable of executing all three FrodoKEM operations, key generation, encapsulation, and decapsulation, at run-time. It supports twelve different configurations corresponding to three security levels for both SHAKE-based and AES-based FrodoKEM and the standard and ephemeral variants. The selection of the FrodoKEM operation is managed via the `sel_op` port, while the parameter set configuration is chosen through the `param_set` port. Additionally, a synthesis-time performance parameter, `T`, determines the number of multiply-and-accumulate (MAC) units within the `Matrix Arithmetic` module, also determining the word width of BRAMs accordingly.

Each element of the public matrix $A$ belongs to $\mathbb{Z}_q$ and consists of $\log_2(q)$ bits, whereas elements of the error distribution matrices $S$, $E$, $S'$, $E'$, and $E''$ are represented using $\chi$ bits, as specified in Table 1. The module supports full-width interfaces for both SHAKE and AES operations through a typical ready-valid protocol, where SHAKE has 1344-bit input/output widths, and AES has 128-bit input/output widths. Since the word size of BRAMs and consequently all other operations depend on the performance parameter `T`, incoming PRNG bits need to be sequentially shifted into appropriate BRAM locations. To facilitate this, we employ a `SHAKE and AES Distribution Network` which consists of a variable shifter and a shift register `SREG`, as illustrated in Figure 3.

The datapath operation for each primitive is outlined in Figure 1 and further elaborated in Section 4. The `Control Logic` module controls the overall execution flow of the datapath. It consists of micro finite state machines (FSMs) for each operation, which facilitates the module-level parallelism. For example, as described in Section 5.2, the generation of the public matrix $A$ and the matrix multiplication operations ($AS$ during key generation and $S'A$ during encapsulation/decapsulation) occur in parallel. To support this parallel execution, the `Control Logic` module coordinates with an auxiliary `"gen A and matrix mult"` FSM, which manages the communication between the matrix multiplication unit and the external choice of PRNG (AES and SHAKE).

**A note on side-channel security.** FrodoKEM's ISO submission [ABD+25b, Section 10.3.1] explicitly addresses constant-time implementation requirements, and describes how the algorithm's design inherently facilitates timing-attack resistance. All loop bounds and control flow decisions are determined by public parameters rather than secret-dependent values, ensuring that execution paths remain independent of sensitive data. Furthermore, the choice of modulus $q$ as a power of two enables efficient and secure modular reduction. These design features ensure constant-time execution across all operations in our hardware implementation, providing protection against timing-based side-channel attacks.

In addition to that, we also implement a first-order masking scheme for the critical decapsulation operation $M = C - B'S$, where the secret key is used. Our architectural design enables this masking countermeasure to be deployed without additional area overhead or performance degradation. This efficiency is achieved by leveraging the unused multipliers in the `Matrix Arithmetic` module already present in our design (see Section 5.2). The masking technique we employ follows the approach described by Howe et al. [HMOR21], adapted to our parallel architecture.

We selectively protect only the `Matrix Arithmetic` module because our architecture enables this protection at effectively no area overhead, as the existing computational resources naturally accommodate the masked operations. However, we acknowledge that comprehensive side-channel protection requires additional countermeasures. Specifically, full protection would involve masking the decode operation and secret key generation process as well. Furthermore, while our implementation incorporates structural protections against first-order side-channel attacks during the matrix operations in decapsulation, rigorous validation through Test Vector Leakage Assessment (TVLA) or similar empirical analysis would be required to certify the effectiveness of these countermeasures in practice. We leave comprehensive side-channel evaluation and protection of the complete FrodoKEM implementation as future work.

## 7 Evaluation

This section describes the evaluation results of our implementation on the targeted FPGA and ASIC platforms described in Section 5. We observed negligible performance differences between the standard and ephemeral FrodoKEM variants; therefore, we only report results for the former.

**FPGA design evaluation.** Tables 4 and 5 present the area and timing results for our FrodoKEM hardware design targeting the AMD Zynq Ultrascale+ (`xczu7ev`) FPGA, compared to other full hardware implementations from the literature [HOKG18, HMOR21, DGA25]. It is important to note that the design in [HMOR21] does not fully comply with the FrodoKEM specification [NAB+20]. Specifically, the authors use Trivium [Int12] instead of SHAKE128 or AES128 for generating $A$, citing Trivium's lightweight and hardware-friendly characteristics as their motivation. As discussed in Section 3, the design in [HMOR21] offers configurable performance parameters similar to our `T` parameter. We report only their highest-performance configuration, which is equivalent to our `T = 16` design. Additionally, their implementation provides synthesis-time parameters to switch between FrodoKEM-640 and FrodoKEM-976.

A specification-compliant hardware design of FrodoKEM is presented in [DGA25]. This work employs different optimization strategies for each FrodoKEM primitive. For key generation, they optimize the large matrix multiplication ($AS$) by deploying 24 parallel multipliers to perform internal dot products (using a 3x8 optimization strategy), and for encapsulation and decapsulation, the matrix multiplication ($S'A$) is optimized using an $8 \times 2 \times 2$ parallelization strategy with 32 parallel multipliers, as discussed in Section 3. To support both optimization strategies, each column of the $S/S'$ matrices is stored in a separate BRAM block. Similar to our approach described in Section 5.2, matrix $A$ is generated on-the-fly during matrix multiplication. However, [DGA25] stores two copies of each row of $A$ to support their optimization techniques.

The area estimates provided in [HMOR21, HOKG18, DGA25] are reported separately for key generation, encapsulation, and decapsulation. For brevity, we aggregated these values for both parameter sets presented by the authors and report the combined totals. Additionally, we note that the designs in [HMOR21, HOKG18] reported in Table 4 do not account for PRNG area, while [DGA25] does not explicitly state whether PRNG area is included, though we assume SHAKE area is incorporated in their estimates.

In our design, we utilize SHAKE and AES as PRNGs, as required in the specification. The area estimates in Table 4 for our design exclude PRNG contributions, as these modules can be interfaced externally. The area and timing results for the PRNG modules are provided separately in Table 2. As described in Section 5.2, the `T` parameter controls performance scaling, and we present results for `T = 16, 32, 64`. While the design supports arbitrary `T` values, selecting a `T` that does not divide parameter $n$ requires padding with zeros. To enable on-the-fly generation of matrix $A$ without overhead or wait times, the SHAKE module must be configured with `Unroll = 1, 2, 3` for `T = 16, 32, 64`, respectively, in SHAKE-based FrodoKEM mode. Similarly, the AES module is fully unrolled and pipelined for AES-based FrodoKEM.

Selecting large `T` values can impact the maximum operating frequency, as the critical path resides within the matrix multiplication unit (detailed in Section 5.2). Specifically, the critical path is dominated by the long addition chain following dot product computations. To mitigate this bottleneck, we adopt a tree-based adder architecture rather than a linear addition chain, thereby reducing the critical path length (described in Section 5.2).

We observe that doubling the `T` value does not significantly increase area overhead. This is because the most resource-intensive operations, dot product computations within matrix multiplications, are implemented using DSP blocks. Consequently, the area increase is primarily reflected in DSP utilization rather than logic resources.

The area results in Table 4 demonstrate that our implementation achieves the highest area efficiency while supporting run-time configuration across all twelve parameter sets. The timing results in Table 5 compare our SHAKE-based FrodoKEM implementation against the Trivium-based design from [HMOR21] and SHAKE-based implementations from [HOKG18, DGA25]. Performance for our AES-based variant is comparable to the SHAKE-based numbers. However, for `T = 32` and `T = 64`, the AES-based design experiences a slowdown because a single AES module cannot provide sufficient bandwidth to generate matrix $A$ on-the-fly while sustaining parallel matrix multiplication. In these configurations, the matrix multiplication unit completes computation on the current row and stalls until the next row becomes available. This limitation could be addressed by deploying multiple AES128 modules in parallel. However, for fair comparison across variants, we restrict our design to a single AES128 module.

The reported operations per second (Ops/sec) metrics demonstrate that our `T = 32` and `T = 64` designs significantly outperform both specification-compliant implementations [HOKG18, DGA25] and non-compliant designs [HMOR21] from the literature.

**ASIC design evaluation.** To demonstrate the versatility and scalability of our FrodoKEM architecture, we extended our evaluation beyond FPGA implementation to ASIC design using the NANGATE45 technology library and OpenRoad toolchain. Table 6 presents comprehensive results for area, power, and timing characteristics. To the best of our knowledge, this represents the first FrodoKEM ASIC implementation reported in the

Table 4: Area comparison of our FrodoKEM hardware design ported on Zynq Ultrascale+ FPGA with the related work.

| **FPGA DESIGN** | | | | | | | |
|---|---|---|---|---|---|---|---|
| T | **Parameter Selection** | **Area** | | | | **Power** | **Freq.** |
| | | **Logic** | | **Memory** | | (W) | (MHz) |
| | | **LUT** | **DSP** | **FF** | **BRAM** | **Log.+Mem.+IO** | |
| **Our Work (AMD Ultrascale+ xazu7ev)** | | | | | | | |
| 16 | All | 14,469 | 16 | 5,793 | 38 | 3.02 | 193 |
| 32 | All | 20,012 | 32 | 6,561 | 46 | 4.29 | 178 |
| 64 | All | 27,835 | 64 | 8,085 | 87 | 6.93 | 168 |
| **[HMOR21] (AMD Artix 7 xc7a35t)** | | | | | | | |
| 16 | FrodoKEM-640† | 22,911 | 48 | 13,023 | 0 | – | 160 |
| | FrodoKEM-976† | 28,017 | 48 | 12,963 | 0 | – | 157 |
| | FrodoKEM-640† | 15,264 | 48 | 12,769 | 12.5 | – | 149 |
| | FrodoKEM-976† | 16,270 | 48 | 12,765 | 19.0 | – | 148 |
| **[HOKG18] (AMD Artix 7 xc7a35t)** | | | | | | | |
| 1 | FrodoKEM-640† | 20,586 | 3 | 10,588 | 33 | – | 162 |
| | FrodoKEM-976† | 22,137 | 3 | 10,624 | 48 | – | 162 |
| **[DGA25] (AMD Artix 7)** | | | | | | | |
| 24/32/32 | FrodoKEM-640† | 45,220 | 88 | 33,838 | 58 | – | 147 |

†LUT, DSP, FF, and BRAM results aggregated for key generation, encapsulation, and decapsulation.

literature, establishing a valuable baseline for future comparative studies.

Our ASIC synthesis results reveal promising area efficiency, with logic occupying only 0.128–0.235 mm$^2$ across our evaluated `T` configurations. Notably, the area scaling exhibits sub-linear growth: transitioning from `T = 16` to `T = 64` requires less than a 2× increase in logic area. This favorable scaling is accompanied by substantial performance gains: our SHAKE-based FrodoKEM achieves an average 3× improvement in operations per second over this range. The AES-based variant demonstrates similar scaling benefits for FrodoKEM-640, though larger parameter sets encounter the previously discussed bandwidth constraints when using a single AES module, consistent with our FPGA observations.

Power consumption scales with performance, increasing from `T = 16` to `T = 64` primarily due to memory units. Interestingly, cross-platform comparison between our ASIC and FPGA implementations (Table 4) reveals complementary efficiency profiles: the ASIC designs for `T = 16` and `T = 32` achieve lower power consumption, while the `T = 64` configuration reaches comparable power levels. This convergence is noteworthy given that our target FPGA (Ultrascale+) is fabricated using 16 nm technology, suggesting that our ASIC implementation on the 45 nm NANGATE45 library demonstrates competitive energy efficiency. A migration to advanced process nodes (e.g., 16 nm or below) would likely yield substantial additional power and area improvements, making FrodoKEM increasingly attractive for resource-constrained and power-sensitive applications.

These ASIC results validate the practical viability of our architecture across multiple platforms and establish FrodoKEM as a feasible candidate for both reconfigurable and ASIC-based cryptographic accelerators.

**Conclusion** In this work, we present a fully standard-compliant, high-speed hardware implementation of FrodoKEM, featuring a unified architecture that efficiently integrates key generation, encapsulation, and decapsulation while optimizing resource sharing. Our implementation supports run-time parameterization for all three security levels, uniquely supporting run-time configuration for all twelve parameter sets (FrodoKEM-640/976/1344 with SHAKE and AES, plus standard and ephemeral variants), and offers synthesis-time configurability (through the `T` parameter) to balance performance and resource utilization based on specific application requirements.

Our FrodoKEM hardware architecture demonstrates great versatility and scalability across both FPGA and ASIC platforms. The design achieves the highest area efficiency among specification-compliant implementations. Our `T = 32` and `T = 64` configurations deliver significantly higher throughput than all prior work, both specification-compliant [HOKG18, DGA25] and non-compliant [HMOR21] designs. The ASIC implementation, representing the first reported FrodoKEM ASIC design, exhibits favorable sub-linear area scaling (less than 2× area for 4× parallelism increase) and competitive energy efficiency despite using 45 nm technology.

With the finalization of the FrodoKEM standard [Int24], real-world deployment of this algorithm is imminent. These results establish FrodoKEM as a practical solution for high-throughput, standards-compliant post-quantum cryptography in both reconfigurable and fixed-function hardware accelerators.

Table 5: Timing comparison of our SHAKE-based and AES-based FrodoKEM hardware design ported on Zynq Ultrascale+ FPGA with the related work.

| **Parameter Set** | **Freq.** | **Cycles ($\times 10^3$)** | | | **Time (ms)** | | | **Ops/sec** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | (MHz) | **KG** | **ENC** | **DEC** | **KG** | **ENC** | **DEC** | **KG** | **ENC** | **DEC** |
| **SHAKE-based FrodoKEM** | | | | | | | | | | |
| (**Our work** `T=16`, **AMD Ultrascale+** `xczu7ev`) | | | | | | | | | | |
| FrodoKEM-640 | 193 | 210 | 217 | 222 | 1.09 | 1.12 | 1.15 | 921 | 891 | 869 |
| FrodoKEM-976 | | 485 | 497 | 505 | 2.51 | 2.57 | 2.62 | 398 | 389 | 382 |
| FrodoKEM-1344 | | 914 | 927 | 938 | 4.73 | 4.8 | 4.86 | 211 | 208 | 206 |
| (**Our work** `T=32`, **AMD Ultrascale+** `xczu7ev`) | | | | | | | | | | |
| FrodoKEM-640 | 178 | 105 | 111 | 116 | 0.59 | 0.62 | 0.65 | 1,696 | 1,603 | 1,530 |
| FrodoKEM-976 | | 243 | 253 | 262 | 1.36 | 1.42 | 1.47 | 733 | 703 | 679 |
| FrodoKEM-1344 | | 458 | 471 | 482 | 2.57 | 2.65 | 2.71 | 389 | 378 | 369 |
| (**Our work** `T=64`, **AMD Ultrascale+** `xczu7ev`) | | | | | | | | | | |
| FrodoKEM-640 | 168 | 53 | 59 | 64 | 0.32 | 0.35 | 0.38 | 3,164 | 2,846 | 2,614 |
| FrodoKEM-976 | | 122 | 132 | 141 | 0.73 | 0.79 | 0.84 | 1,374 | 1,271 | 1,191 |
| FrodoKEM-1344 | | 230 | 242 | 253 | 1.37 | 1.44 | 1.51 | 730 | 693 | 663 |
| ([**HOKG18**] `T=1`, **AMD Artix 7** `xc7a35t`) | | | | | | | | | | |
| FrodoKEM-640 | 162 | 3,276 | 3,317 | 3,358 | 19.6 | 19.8 | 20.7 | 51 | 51 | 49 |
| FrodoKEM-976 | | 7,620 | 7,683 | 7,745 | 45.6 | 46.0 | 47.8 | 22 | 22 | 21 |
| ([**DGA25**] `KG`$_T$`=24, ENC`$_T$`=32, DEC`$_T$`=32, AMD Artix 7`) | | | | | | | | | | |
| FrodoKEM-640 | 147 | 151 | 127 | 128 | 1.02 | 0.93 | 0.95 | 976 | 1,077 | 1,052 |
| **AES-based FrodoKEM** | | | | | | | | | | |
| (**Our work** `T=16`, **AMD Ultrascale+** `xczu7ev`) | | | | | | | | | | |
| FrodoKEM-640 | 193 | 208 | 215 | 219 | 1.08 | 1.11 | 1.14 | 927 | 899 | 880 |
| FrodoKEM-976 | | 483 | 492 | 499 | 2.5 | 2.55 | 2.59 | 400 | 392 | 387 |
| FrodoKEM-1344 | | 912 | 925 | 935 | 4.73 | 4.79 | 4.84 | 212 | 209 | 206 |
| (**Our work** `T=32`, **AMD Ultrascale+** `xczu7ev`) | | | | | | | | | | |
| FrodoKEM-640 | 178 | 140 | 146 | 151 | 0.79 | 0.82 | 0.85 | 1,268 | 1,216 | 1,179 |
| FrodoKEM-976 | | 317 | 327 | 335 | 1.78 | 1.84 | 1.88 | 562 | 545 | 532 |
| FrodoKEM-1344 | | 577 | 590 | 599 | 3.24 | 3.31 | 3.37 | 308 | 302 | 297 |
| (**Our work** `T=64`, **AMD Ultrascale+** `xczu7ev`) | | | | | | | | | | |
| FrodoKEM-640 | 168 | 140 | 146 | 150 | 0.83 | 0.87 | 0.89 | 1,203 | 1,154 | 1,119 |
| FrodoKEM-976 | | 316 | 325 | 333 | 1.88 | 1.94 | 1.98 | 532 | 516 | 504 |
| FrodoKEM-1344 | | 576 | 588 | 598 | 3.43 | 3.5 | 3.56 | 292 | 286 | 281 |
| ([**HOKG18**] `T=1`, **AMD Artix 7** `xc7a35t`) | | | | | | | | | | |
| FrodoKEM-640 | 162 | 3,276 | 3,317 | 3,358 | 19.6 | 19.8 | 20.7 | 51 | 51 | 49 |
| FrodoKEM-976 | | 7,620 | 7,683 | 7,745 | 45.6 | 46.0 | 47.8 | 22 | 22 | 21 |
| **Trivium-based FrodoKEM** ([**HMOR21**] `T=16`, **AMD Artix 7** `xc7a35t`) | | | | | | | | | | |
| FrodoKEM-640 | 160 | Not Reported | | | | | | 840 | 825 | 763 |
| FrodoKEM-976 | | | | | | | | 355 | 350 | 325 |

KG - Key Generation, ENC: Encapsulation, DEC: Decapsulation.

Table 6: Area and Time performance numbers of our SHAKE-based and AES-based FrodoKEM hardware design using NANGATE45 technology library.

| **Area and Power Results** | | | | | |
|---|---|---|---|---|---|
| **Parameter Set** | T | **Area** ($mm^2$) | | **Power (W)** | |
| | | **Logic** | **Memory** | **Logic** | **Memory** |
| | | **FrodoKEM (our)** | | | |
| All | 16 | 0.128 | 0.842 | 0.32 | 0.98 |
| All | 32 | 0.159 | 1.208 | 0.67 | 1.97 |
| All | 64 | 0.235 | 2.463 | 1.04 | 6.22 |

| **Timing Results** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Parameter Set** | **Freq.** | **Cycles** ($\times 10^3$) | | | **Time (ms)** | | | **Ops/sec** | | |
| | (MHz) | **KG** | **ENC** | **DEC** | **KG** | **ENC** | **DEC** | **KG** | **ENC** | **DEC** |
| | | | | **SHAKE-based FrodoKEM** | | | | | | |
| | | | | (**Our work** T=16) | | | | | | |
| FrodoKEM-640 | | 210 | 217 | 222 | 0.48 | 0.49 | 0.51 | 2,090 | 2,022 | 1,972 |
| FrodoKEM-976 | 438 | 485 | 497 | 505 | 1.11 | 1.13 | 1.15 | 902 | 882 | 867 |
| FrodoKEM-1344 | | 914 | 927 | 938 | 2.09 | 2.12 | 2.14 | 479 | 472 | 467 |
| | | | | (**Our work** T=32) | | | | | | |
| FrodoKEM-640 | | 105 | 111 | 116 | 0.25 | 0.27 | 0.28 | 3,984 | 3,764 | 3,592 |
| FrodoKEM-976 | 418 | 243 | 253 | 262 | 0.58 | 0.61 | 0.63 | 1,722 | 1,651 | 1,595 |
| FrodoKEM-1344 | | 458 | 471 | 482 | 1.1 | 1.13 | 1.15 | 913 | 887 | 867 |
| | | | | (**Our work** T=64) | | | | | | |
| FrodoKEM-640 | | 53 | 59 | 64 | 0.14 | 0.15 | 0.17 | 7,194 | 6,471 | 5,943 |
| FrodoKEM-976 | 382 | 122 | 132 | 141 | 0.32 | 0.35 | 0.37 | 3,124 | 2,891 | 2,709 |
| FrodoKEM-1344 | | 230 | 242 | 253 | 0.6 | 0.63 | 0.66 | 1,660 | 1,576 | 1,508 |
| | | | | **AES-based FrodoKEM** | | | | | | |
| | | | | (**Our work** T=16) | | | | | | |
| FrodoKEM-640 | | 208 | 215 | 219 | 0.48 | 0.49 | 0.5 | 2,103 | 2,041 | 1,998 |
| FrodoKEM-976 | 438 | 483 | 492 | 499 | 1.1 | 1.12 | 1.14 | 907 | 889 | 877 |
| FrodoKEM-1344 | | 912 | 925 | 935 | 2.08 | 2.11 | 2.13 | 480 | 473 | 469 |
| | | | | (**Our work** T=32) | | | | | | |
| FrodoKEM-640 | | 140 | 146 | 151 | 0.34 | 0.35 | 0.36 | 2,978 | 2,855 | 2,768 |
| FrodoKEM-976 | 418 | 317 | 327 | 335 | 0.76 | 0.78 | 0.8 | 1,320 | 1,279 | 1,249 |
| FrodoKEM-1344 | | 577 | 590 | 599 | 1.38 | 1.41 | 1.43 | 724 | 709 | 697 |
| | | | | (**Our work** T=64) | | | | | | |
| FrodoKEM-640 | | 140 | 146 | 150 | 0.37 | 0.38 | 0.39 | 2,736 | 2,625 | 2,545 |
| FrodoKEM-976 | 382 | 316 | 325 | 333 | 0.83 | 0.85 | 0.87 | 1,211 | 1,174 | 1,146 |
| FrodoKEM-1344 | | 576 | 588 | 598 | 1.51 | 1.54 | 1.56 | 664 | 650 | 639 |

KG - Key Generation, ENC: Encapsulation, DEC: Decapsulation.

## References

[ABC+19] T Ajayi, D Blaauw, TB Chan, CK Cheng, VA Chhabria, DK Choo, M Coltella, S Dobre, R Dreslinski, M Fogaça, et al. OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain. *Proc. GOMACTECH*, pages 1105–1110, 2019.

[ABD+17] Erdem Alkim, Joppe W. Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM: Learning With Errors Key Encapsulation, 2017. FrodoKEM Round 1 specification available at https://frodokem.org/files/FrodoKEM-specification-20171130.pdf.

[ABD+25a] Erdem Alkim, Joppe W. Bos, Léo Ducas, Karen Easterbrook, Lewis Glabush, Brian LaMacchia, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, Douglas Stebila, and Fernando Virdia. FrodoKEM: Learning With Errors Key Encapsulation, 2017–2025. Specification document available at https://frodokem.org/.

[ABD+25b] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM Preliminary Standardization Proposal (submitted to ISO), updated Sept. 2025. https://frodokem.org/files/FrodoKEM_standard_proposal_20250929.pdf.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 327–343. USENIX Association, August 2016.

[BBC+23] Joppe Bos, Olivier Bronchain, Frank Custers, Joost Renes, Denise Verbakel, and Christine Vredendaal. Enabling FrodoKEM on Embedded Devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 74–96, 06 2023.

[Bes24] Bespoke Silicon Group. BSG FakeRAM. GitHub Repository, 2024. Accessed: 2025-01-15.

[Bus21] Fabian Buschkowski. Efficient Hardware Implementation of Post-Quantum Cryptography using High-Level Synthesis, 2021. Code available at https://github.com/Chair-for-Security-Engineering/FrodoKEM/ (accessed: March 21, 2025).

[CS25] Sahil Chauhan and Rahul Shrestha. Reconfigurable and Hardware-Efficient KECCAK Architecture with SHAKE Integration and Dynamic Input Processing for Post Quantum Cryptography. In *2025 International VLSI Symposium on Technology, Systems and Applications (VLSI TSA)*, pages 1–4, 2025.

[CVS+16] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[DFA+20] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and Benchmarking of Round 2

Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. Cryptology ePrint Archive, Paper 2020/795, 2020.

[DGA25] Gökçe Düzyol, Muhammed Said Gündoğan, and Atakan Arslan. Can FrodoKEM Run in a Millisecond? FPGA Says Yes! Cryptology ePrint Archive, Paper 2025/1312, 2025.

[DLK+25] Sanjay Deshpande, Yongseok Lee, Cansu Karakuzu, Jakub Szefer, and Yunheung Paek. SPHINCSLET: An Area-Efficient Accelerator for the Full SPHINCS+ Digital Signature Algorithm. *ACM Trans. Embed. Comput. Syst.*, 24(5), September 2025.

[DS13] M. H. Devoret and R. J. Schoelkopf. Superconducting circuits for quantum information: an outlook. *Science*, 339(6124):1169–1174, 2013.

[Fed24] Federal Office for Information Security (BSI). Cryptographic Mechanisms: Recommendations and Key Lengths, BSI TR-02102-1, Version: 2024-1, 2024. Available at https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=7.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[Gen22] General Intelligence and Security Service (AIVD). Prepare for the threat of quantum computers, 2022. Available at https://english.aivd.nl/publications/publications/2022/01/18/prepare-for-the-threat-of-quantumcomputers.

[Gen24] General Intelligence and Security Service (AIVD) and Centrum Wiskunde & Informatica (CWI) and Netherlands Organization for Applied Scientific Research (TNO). The PQC Migration Handbook: Guidelines for Migrating to Post-Quantum Cryptography (second edition), 2024. Available at https://publications.tno.nl/publication/34643386/fXcPVHsX/TNO-2024-pqc-en.pdf.

[GLN+25] Lewis Glabush, Patrick Longa, Michael Naehrig, Chris Peikert, Douglas Stebila, and Fernando Virdia. FrodoKEM: A CCA-secure learning with errors key encapsulation mechanism. *IACR Communications in Cryptology*, 2(3), 2025. https://eprint.iacr.org/2025/1861.

[GOW+24] Ivan Gavrilan, Felix Oberhansl, Alexander Wagner, Emanuele Strieder, and Andreas Zankl. Impeccable Keccak: Towards fault resilient SPHINCS+ implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):154–189, Mar. 2024.

[HMOR21] James Howe, Marco Martinoli, Elisabeth Oswald, and Francesco Regazzoni. Exploring parallelism to improve the performance of FrodoKEM in hardware. *Journal of Cryptographic Engineering*, 11, 11 2021.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):372–393, Aug. 2018.

[Int12] International Organization for Standardization. ISO/IEC 29192-3:2012 – Information technology — Security techniques — Lightweight cryptography — Part 3: Stream ciphers. International Standard, 2012.

[Int24] International Organization for Standardization (ISO). ISO/IEC 18033-2:2006/DAmd 2, Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers, 2024. Available at https://www.iso.org/standard/86890.html.

[KBF+15] J. Kelly, R. Barends, A. G. Fowler, A. Megrant, E. Jeffrey, T. C. White, D. Sank, J. Y. Mutus, B. Campbell, Yu Chen, Z. Chen, B. Chiaro, A. Dunsworth, I.-C. Hoi, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, A. Vainsencher, J. Wenner, A. N. Cleland, and John M. Martinis. State preservation by repetitive error detection in a superconducting quantum circuit. *Nature*, 519:66–69, 2015.

[KFS22] Patrick Karl, Tim Fritzmann, and Georg Sigl. Hardware accelerated FrodoKEM on RISC-V. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 154–159, 2022.

[LBES25] Patrick Longa, Joppe W. Bos, Stephan Ehlen, and Douglas Stebila. FrodoKEM: key encapsulation from learning with errors. Internet-Draft draft-longa-cfrg-frodokem-01, Internet Engineering Task Force, 2025. https://datatracker.ietf.org/doc/html/draft-longa-cfrg-frodokem-01.

[Mos18] Michele Mosca. Cybersecurity in an Era with Quantum Computers: Will We Be Ready?, 2018.

[NAB+20] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[Nan11] Nangate Inc. Nangate 45nm Open Cell Library. FreePDK45, NC State University, 2011. Accessed: 2025-01-15.

[Nat01] National Institute of Standards and Technology. Advanced Encryption Standard (AES). FIPS PUB 197, U.S. Department of Commerce, November 2001.

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, U.S. Department of Commerce, August 2015.

[Nat17] National Institute of Standards and Technology (NIST). Post-quantum cryptography standardization project, 2017. http://csrc.nist.gov/groups/ST/post-quantum-crypto/.

[Nat23] National Cybersecurity Agency of France (ANSSI). ANSSI views on the Post-Quantum Cryptography transition (2023 follow up), 2023. Available at https://cyber.gouv.fr/en/publications/follow-position-paper-post-quantum-cryptography.

[Nat24] National Institute of Standards and Technology. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard, 2024.

[NC 23] NC State University EDA Group. FreePDK3: An Open-Source PDK for 3-nm Gate-All-Around Technology. https://github.com/ncsu-eda/FreePDK3, 2023. Accessed: 2025-01-15.

[New24] NewAE Technology Inc. ChipWhisperer AES HDL Implementation. GitHub Repository, 2024. Part of ChipWhisperer project. Accessed: 2025-01-15.

[PdCKN16] Rodrigo Portella do Canto, Roman Korkikian, and David Naccache. *Buying AES Design Resistance with Speed and Energy*, pages 134–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.

[SHTW23] Yifeng Song, Xiao Hu, Jing Tian, and Zhongfeng Wang. A High-Speed FPGA-Based Hardware Implementation for Leighton-Micali Signature. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(1):241–252, 2023.

[SMP14] Jawar Singh, Saraju P. Mohanty, and Dhiraj Pradhan. *Robust SRAM Designs and Analysis*. Springer Publishing Company, Incorporated, 2014.

[SS15] Abolfazl Soltani and Saeed Sharifian. An ultra-high throughput and fully pipelined implementation of AES algorithm on FPGA. *Microprocessors and Microsystems*, 39(7):480–493, 2015.

[Str21] Volker Strassen. Gaussian Elimination is Not Optimal (1969). In *Ideas That Created the Future: Classic Papers of Computer Science*. The MIT Press, 02 2021.

[UMVM25] Fernando Aparicio Urbano-Molano and Jaime Velasco-Medina. FrodoKEM Hardware Implementation for Post-Quantum Cryptography. *IEEE Latin America Transactions*, 23(10):922–930, Aug. 2025.

[VCV$^+$20] Paolo Visconti, Stefano Capoccia, Eugenio Venere, Ramiro Velázquez, and Roberto de Fazio. 10 Clock-Periods Pipelined Implementation of AES-128 Encryption-Decryption Algorithm up to 28 Gbit/s Real Throughput by Xilinx Zynq UltraScale+ MPSoC ZCU102 Platform. *Electronics*, 9(10), 2020.