A Fast and Configurable FPGA Implementation of FrodoKEM with DSP-Free Strategies

Jianchi Zhu †, Yaodong Wei †, Wenbo Kong, Xingrui Gu and Jing Tian (✉)

School of Integrated Circuit, Nanjing University, Suzhou, China
{231880339,yaodongwei,231880456,231880441}@smail.nju.edu.cn,tianjing@nju.edu.cn

Abstract. FrodoKEM is an unstructured LWE-based key encapsulation mechanism (KEM), widely recognized for its conservativeness and robustness. However, there remains a performance gap of approximately one order of magnitude between FrodoKEM and structured LWE-based KEMs, which severely limits its competitiveness during the standardization process. This work proposes the fastest hardware implementation of FrodoKEM to date, which is also configurable, achieving up to 2.5× speedup over prior hardware designs. Specifically, for the most time-consuming module, modular matrix multiplication, a block matrix processing approach is adopted, where the computation is reformulated into block-level operations and further mapped to vector inner-product computations. Based on this formulation, two optimized architectures are proposed for vector inner-product computation using a carry-save-adder (CSA)-based Wallace tree structure, achieving improvements in resource efficiency while eliminating the use of DSP resources. By fixing the block size to 4×4, the data generation rate using parallel SHAKE-based random number generators is precisely matched with the data consumption rate of matrix multiplication, thereby improving hardware utilization efficiency. The proposed design is implemented in Verilog and verified on both Xilinx Virtex-7 and Artix-7 FPGA platforms, as well as ASIC implementation.

Keywords: Post-Quantum Cryptography · FrodoKEM · Unstructured LWE · FPGA · ASIC · Block Matrix

1 Introduction

Since the proposal of Shor's quantum algorithm [Sho94], traditional public-key cryptographic systems based on the difficulty of large integer factorization and the discrete logarithm problem have faced fundamental security challenges, making the evolution of cryptographic systems toward Post-Quantum Cryptography (PQC) a widespread industry consensus. With continuous breakthroughs in quantum computing technology, the likelihood of realizing large-scale general-purpose quantum computers within the next decade has steadily increased, further accelerating the research and standardization process of PQC algorithms. In 2016, the National Institute of Standards and Technology (NIST) [CCJ+16] initiated the evaluation and standardization of PQC algorithms.

Among the numerous candidate schemes, Lattice-Based PQC (LB-PQC) schemes based on the Learning With Errors (LWE) problem have become a focal point of research and application, owing to the favorable trade-off they exhibit among security, computational efficiency, and implementation resource consumption. Depending on whether additional algebraic structures are introduced, these schemes are typically classified into two major branches: structured LWE schemes and unstructured LWE schemes.

† These authors contributed equally.
✉ Corresponding author.

As a mainstream unstructured LWE scheme, FrodoKEM, despite not being selected by NIST, has been recognized for its conservative security properties by the International Organization for Standardization (ISO) [ISO25], the German Federal Office for Information Security (BSI) [Fed25], and the Dutch National Cyber Security Agency (NLNCSA) [Net23].

It is worth noting that, despite its security advantages, FrodoKEM suffers from high computational cost and large storage overhead compared with structured lattice schemes. Its performance bottleneck is primarily concentrated in large-scale matrix operations under $\mathbb{Z}_q$, particularly the matrix multiplications $\mathbf{AS}$ and $\mathbf{S'A}$. In software implementations, these operations account for more than 98% of the execution time. Furthermore, as the security level increases from FrodoKEM-640 (Level 1) to FrodoKEM-1344 (Level 5), the matrix dimension n more than doubles, posing significant challenges to memory bandwidth and hardware resources.

In existing hardware implementations, accelerating the matrix multiplications is also the main job. Howe *et al.* [HOKG18] pioneered the hardware implementation of FrodoKEM on FPGA in 2018. By reusing computational units, they successfully reduced hardware resource consumption, but the serialized design resulted in high latency. In 2021, Buschkowski *et al.* [Bus21] investigated High-Level Synthesis (HLS) for FrodoKEM, demonstrating that optimized HLS could match the performance of early hardware designs in [HOKG18] with significantly reduced engineering effort. Meanwhile, Howe *et al.* [HMOR21] identified that the primary performance bottleneck in FrodoKEM hardware implementations stems from the inherently serial structure of the SHAKE random number generator. They proposed substituting the standard SHAKE with the Trivium [DCP08] stream cipher to alleviate the bottleneck in matrix generation. Although this approach is effective in terms of performance, it renders the implementation incompatible with NIST and ISO standards, deviating from the original specification of FrodoKEM. In 2025, Düzyol *et al.* [DGA25] presented a standard-compliant FPGA implementation that achieves millisecond-level latency by enhancing hardware architectural parallelism and performing deep optimization on the pipelined data flow between SHAKE and the multiplier, with a cost of abundant DSP multipliers. Recently, [LLY+26] exploited aggressive pipeline insertion and supported configurability, achieving high operating frequency and reduced execution latency compared with prior designs. However, the design is limited to FPGA-only evaluation and lacks a unified architecture for left- and right-side matrix multiplications.

Note that most existing pure hardware designs [HOKG18, Bus21, HMOR21, DGA25] are not configurable across different security levels and execution stages. Moreover, even the fastest existing implementation [LLY+26] remains significantly slower than the representative Kyber hardware design, a structured-lattice-based KEM scheme [XL21]. To overcome these limitations, this paper proposes a high-performance and configurable hardware accelerator for FrodoKEM. Through deep optimization of processes such as matrix multiplication and random bits generation, this work achieves lower computation latency than the fastest FrodoKEM hardware implementation reported to date, while simultaneously supporting configurable integration of multiple functionalities and parameter sets.

1.1 Our Contributions

In this work, we present the fastest hardware implementation of FrodoKEM to date, which is also configurable, achieving up to 2.5× speedup over the fastest existing implementation [LLY+26]. Our main contributions are summarized as follows:

- We present the fastest FrodoKEM hardware with a unified controller, which is configurable, allowing runtime stage selection without re-programming, greatly enhancing flexibility and practicality.

- We propose a dynamically configurable 4×4 block multiplication strategy that enables bidirectional matrix multiplication, maximizing parallelism, with increased hardware resource utilization as a trade-off.
- A DSP-free architecture is introduced by leveraging operand compression and modulo arithmetic, significantly reducing hardware overhead and making it ASIC-friendly, and is implemented on both FPGA and ASIC platforms.
- We eliminate the random data generation bottleneck by designing a quad-parallel SHAKE engine, which quadruples the output rate to perfectly feed a 4×4 block multiplier per cycle, removing pipeline stalls.

1.2 Outline

The rest of the paper is organized as follows. Section 2 introduces the notation and parameters used and provides an overview of the FrodoKEM algorithm. Section 3 details the proposed hardware architectures for FrodoKEM. In Section 4, the FPGA and ASIC implementation results are provided and compared with previous works. Finally, Section 5 concludes our work.

2 Preliminaries

2.1 Notations and Parameters

The security of FrodoKEM relies on the difficulty of the LWE problem [Reg09] over generic integer lattices. This scheme involves arithmetic operations in the integer ring modulo q. Matrices are denoted by uppercase bold letters (e.g., $\mathbf{A}$), and vectors are denoted by lowercase bold letters.

The involved notations in FrodoKEM are presented below [ABD+25].

$q = 2^D$	a power-of-two integer modulus with $D \leq 16$.
$n, \bar{n}$	integer matrix dimensions with $n, \bar{n} \equiv 0 \pmod 8$.
D	the number of bits encoded in each matrix entry.
$\mathbb{Z}_q$	the set of integers modulo q.
χ	the discrete, symmetric error distribution on $\mathbb{Z}$ used by FrodoKEM.
T_χ	a table $(T_\chi(0), T_\chi(1), ..., T_\chi(d))$ with $(d+1)$ positive integers based on the cumulative distribution function for χ, with $d \in \mathbb{Z}^+$.

The summary of parameters for FrodoKEM is shown in Table 1, where all three security levels are included.

2.2 FrodoKEM

The complete key encapsulation mechanism (KEM) comprises three algorithms. Below, we outline the standard procedure, with particular emphasis on matrix multiplication as the primary computational bottleneck.

2.2.1 Key Generation

Algorithm 1 generates the public key (pk) and secret key (sk). The main steps are: using input information z and SeedSE, generate SeedA, matrix $\mathbf{S}$, and $\mathbf{E}$ through the SHAKE

Table 1: Parameters for (e)FrodoKEM-640, (e)FrodoKEM-976 and (e)FrodoKEM-1344.

Parameters	(e)FrodoKEM-640	(e)FrodoKEM-976	(e)FrodoKEM-1344
D	15	16	16
q	32768	65536	65536
n	640	976	1344
$\bar{n}$	8	8	8
d	12	10	6
SHAKE	SHAKE128	SHAKE128&SHAKE256	SHAKE128&SHAKE256

module; then generate matrix $\mathbf{A}$ row by row; perform the operation "$\mathbf{B} = \mathbf{AS} + \mathbf{E}$". Feed Pack ($\mathbf{B}$) and SeedA as input to the SHAKE module to compute the public key hash (pkh). After concatenating pk and sk, transmit them via the I/O port. The core operation is calculating "$\mathbf{B} = \mathbf{AS} + \mathbf{E}$".

Algorithm 1: FrodoKEM.KeyGen

Input: Random Seeds s, $seed_{\mathbf{SE}}$ and z
Output: Public Key pk, Secret Key sk

$seed_{\mathbf{A}} \leftarrow \text{SHAKE}(z)$;
$\mathbf{A} \leftarrow \text{Frodo.Gen}(seed_{\mathbf{A}})$;
$\mathbf{S^T}, \mathbf{E} \leftarrow \text{Frodo.Sample}(seed_{\mathbf{SE}})$;
$\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E} \pmod q$;
$b \leftarrow \text{Frodo.Pack}(\mathbf{B})$;
$pk \leftarrow seed_{\mathbf{A}} || b$;
$pkh \leftarrow \text{SHAKE}(pk)$;
$sk \leftarrow (s || seed_{\mathbf{A}} || b, \mathbf{S}, pkh)$;
return (pk, sk);

Algorithm 2: FrodoKEM.Encaps

Input: Public Key $pk = seed_{\mathbf{A}} || b$ and μ
Output: Ciphertext c, Shared Secret ss

$seed_{\mathbf{SE}} || k \leftarrow \text{SHAKE}(\text{SHAKE}(pk) || \mu)$;
$\mathbf{S'}, \mathbf{E'}, \mathbf{E''} \leftarrow \text{Frodo.Sample}(seed_{\mathbf{SE}})$;
$\mathbf{A} \leftarrow \text{Frodo.Gen}(seed_{\mathbf{A}})$;
$\mathbf{B} \leftarrow \text{Frodo.Unpack}(b)$;
$\mathbf{B'} \leftarrow \mathbf{S'A} + \mathbf{E'} \pmod q$;
$\mathbf{V} \leftarrow \mathbf{S'B} + \mathbf{E''} \pmod q$;
$\mathbf{C} \leftarrow \mathbf{V} + \text{Frodo.Encode}(\mu)$;
$c_1 \leftarrow \text{Frodo.Pack}(\mathbf{B'})$; $c_2 \leftarrow \text{Frodo.Pack}(\mathbf{C})$;
$ss \leftarrow \text{SHAKE}(c_1 || c_2 || k)$;
return $(c_1 || c_2, ss)$;

2.2.2 Encapsulation

Algorithm 2 generates a shared key (ss). Similar to Key Generation, after generating matrices $\mathbf{A}$, $\mathbf{S'}$, $\mathbf{E'}$, and $\mathbf{E''}$ via SHAKE, it involves two matrix multiplications: $\mathbf{B} = \mathbf{S'A} + \mathbf{E}$ and $\mathbf{V} = \mathbf{S'B} + \mathbf{E''}$. Following a Pack operation and processing through

the SHAKE module, the shared key ss is obtained. Computing $\mathbf{B}$ poses a challenge in hardware because it requires multiplying the rows of $\mathbf{S}'$ by the columns of $\mathbf{A}$, which is generated row-wise. The block matrix architecture proposed in Section 3.3 resolves this issue.

2.2.3 Decapsulation

The decapsulation algorithm is quite similar to the encapsulation algorithm, incorporating a constant-time check in the later stages to ensure CCA security.

Algorithm 3: FrodoKEM.Decaps

Input: Ciphertext $c = c_1||c_2$, Secret Key sk
Output: Shared Secret ss

$\mathbf{B}' \leftarrow$ Frodo.Unpack(c_1); $\mathbf{C} \leftarrow$ Frodo.Unpack(c_2);
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S} \pmod q$;
$\mu' \leftarrow$ Frodo.Decode($\mathbf{M}$);
$seed_{\mathbf{E}'}||k' \leftarrow$ SHAKE($pkh||\mu'$);
$\mathbf{S}', \mathbf{E}', \mathbf{E}'' \leftarrow$ Frodo.Sample($seed_{\mathbf{E}'}$);
$\mathbf{A} \leftarrow$ Frodo.Gen($seed_{\mathbf{A}}$);
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$;
$\mathbf{V}' \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$;
$\mathbf{C}' \leftarrow \mathbf{V}' +$ Frodo.Encode(μ');
if $\mathbf{B}'' = \mathbf{B}'$ ***and*** $\mathbf{C}' = \mathbf{C}$ **then**
  $ss \leftarrow$ SHAKE($c||k'$);
end
else
  $ss \leftarrow$ SHAKE($c||s$);
end
return ss;

2.3 Matrix Generation

The generation and processing of matrix $\mathbf{A}$ constitute the most time-consuming operations in FrodoKEM. Matrix $\mathbf{A}$ is generated by concatenating the index of the current row with SeedA as the seed and feeding it into a pseudorandom number generator to produce one row of $\mathbf{A}$ (Algorithm 4). The pseudorandom number generators available include SHAKE and AES. Since SHAKE is extensively used in FrodoKEM beyond matrix $\mathbf{A}$ generation, while AES is limited solely to matrix $\mathbf{A}$ generation, and SHAKE offers greater flexibility, this work adopts SHAKE as the generator for matrix $\mathbf{A}$.

Algorithm 4: Frodo.Gen (SHAKE)

Input: Seed $seed_{\mathbf{A}}$, Dimension n
Output: Matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$

for $i \leftarrow 0$ **to** $n-1$ **do**
  $b \leftarrow i||seed_{\mathbf{A}}$;
  $c_{i,0}||c_{i,1}||\ldots||c_{i,n-1} \leftarrow$ SHAKE(b);
  `/* Each` $c_{i,j}$ `is 16 bits. */`
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $\mathbf{A}_{i,j} \leftarrow c_{i,j} \pmod q$;
  end
end
return $\mathbf{A}$;

3 Proposed Hardware Architecture

This work designs a unified, resource-efficient, and high-speed hardware architecture supporting all FrodoKEM security levels (Levels 1, 3, and 5) and all execution stages.

3.1 Top-Level Architecture

The top-level architecture of FrodoKEM is shown in Fig. 1, primarily consisting of the following components: 1) Parallel Vector Dot Product Unit, 2) Matrix Multiplier, 3) Random bits Generator, 4) Memory, and 5) Control Unit.

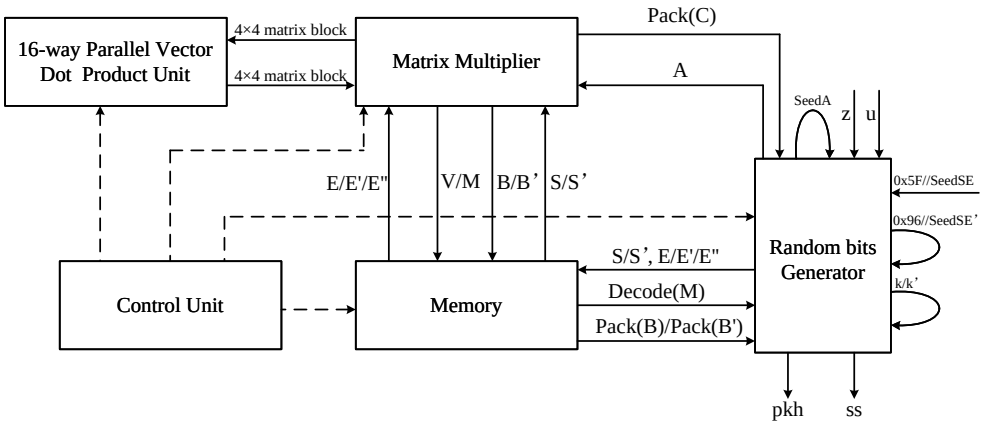


Figure 1: Proposed top-level architecture of FrodoKEM, illustrating the five principal modules along with the main data flows between them.

The parallel vector multiplier computes 4×4 matrix multiplication by instantiating 16 specially optimized vector dot product units. The specific optimization techniques and the rationale for selecting 4×4 matrix multiplication are discussed in Section 3.4. The Matrix Multiplier orchestrates the Parallel Vector Dot Product Unit to compute large-scale matrix multiplications, such as $\mathbf{AS}$, $\mathbf{S'A}$, $\mathbf{S'B}$, and $\mathbf{B'S}$.

The Random bits generator module instantiates four SHAKE modules internally. Depending on the operating mode, either all four SHAKE modules operate in parallel or only a single SHAKE module is activated. The basic core of SHAKE module is Keccak [D+15] core, where random bits can be output each cycle after the initialization. In our design, each SHAKE module outputs 64 bits per cycle, corresponding to four matrix elements.

The storage module is logically designed for maximal resource reuse to reduce hardware overhead. It consists of registers and BRAMs. Registers are mainly used to store small matrix blocks, short messages, and pipeline registers for timing optimization. The BRAMs store matrices $\mathbf{S}/\mathbf{S'}$, $\mathbf{E}/\mathbf{E'}$, and $\mathbf{B}/\mathbf{B'}$, as well as their packed representations. The detailed BRAM reuse strategy is discussed in Section 3.5. Notably, due to its large dimension, matrix $\mathbf{A}$ is not stored in BRAM; instead, it is generated and processed on the fly, as described in Section 3.2 and Section 3.3.

Finally, the control module is responsible for generating control signals to coordinate and manage the operation of all the aforementioned modules.

3.2 Random Bits Generator

In FrodoKEM, SHAKE128 and SHAKE256 are employed as the pseudorandom number generators (PRNGs). SHAKE is an extendable-output function (XOF) based on the Keccak core, which can accept inputs of arbitrary length and generate pseudorandom bitstreams of any required length. In this work, we design a configurable SHAKE module that, through external control signals, can absorb and squeeze messages of arbitrary length and invoke either SHAKE128 or SHAKE256 to process the input data.

As shown in Fig. 2, the top-level architecture of SHAKE module consists of a control unit, a Keccak-p[1600, 24] core, a padding logic unit, and a 21 × 64-bit hash buffer. This modular design enables flexible configuration and efficient data processing.

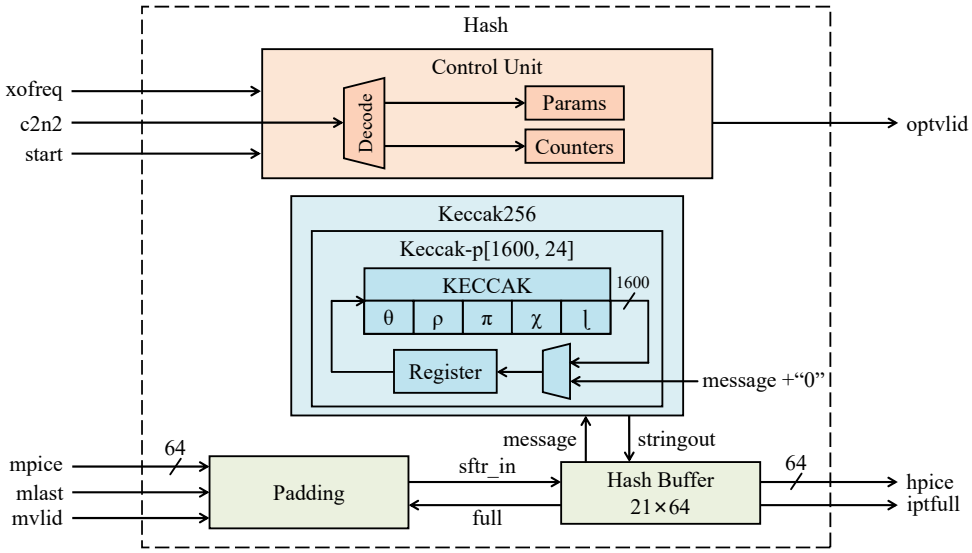


Figure 2: Top-level Architecture of the SHAKE Module.

The top-level interfaces and their functions are summarized in Table 2.

Table 2: Signal Definitions for the Configurable SHAKE Core Interface.

Signal	Direction	Width	Description
`start`	Input	1	Global enable signal to activate the core.
`c2n2`	Input	5	Keccak capacity (2 bits)//Padding start byte (3 bits).
`xofreq`	Input	1	Request signal for extendable output generation.
`mlast`	Input	1	Indicates the last segment of the input message.
`mpice`	Input	64	Input message segment (Message Piece).
`mvlid`	Input	1	Valid indicator for the input message segment.
`hpice`	Output	64	Output hash segment (Hash Piece).
`optvlid`	Output	1	Valid indicator for the output hash segment.
`iptfull`	Output	1	Status flag indicating the internal hash buffer is full.

In FrodoKEM, the SHAKE module is used to generate random seeds, matrices $\mathbf{S}/\mathbf{E}$, and matrix $\mathbf{A}$. Due to the constraints imposed by the FrodoKEM protocol and the operating mechanism of SHAKE, all data except matrix $\mathbf{A}$ must be generated sequentially

by a single SHAKE module. For matrix $\mathbf{A}$, since each row is generated from a fixed and mutually distinct seed, multiple SHAKE modules can be employed in parallel to generate different rows simultaneously.

A key consideration of our design is the selection of the SHAKE parallelism and the corresponding matrix block size. A single SHAKE module achieves an output throughput of 64 bits per cycle, corresponding to four matrix elements. To fully utilize this throughput while maintaining architectural regularity, square block partitioning is adopted. This ensures computational consistency between left and right matrix multiplications and balanced data generation across both row and column dimensions.

Based on this, the parallelism of SHAKE is set to four, enabling the generation of a 4×4 matrix block per cycle. Alternative parallelism degrees, such as 2 or 8, were also considered. However, these configurations cannot produce square matrix blocks within a single cycle, leading to additional buffering requirements, timing overhead, and increased control complexity. Therefore, the 4-parallel configuration provides the most efficient balance between throughput and implementation cost.

The generated matrix blocks are streamed into the Matrix Multiplier unit described in Section 3.3, where block-level computations are efficiently scheduled. This design allows the generation of matrix $\mathbf{A}$ to overlap with subsequent matrix multiplication, enabling concurrent execution and improving overall system throughput.

3.3 Matrix Multiplier

Matrix multiplication over $\mathbb{Z}_q$ dominates the execution time of FrodoKEM. The most time-consuming matrix multiplication mainly involves two forms: $\mathbf{AS}$ and $\mathbf{S'A}$. Due to its large dimension, matrix $\mathbf{A}$ cannot be pre-stored; therefore, a *generate–consume* architectural approach is required.

Consider $\mathbf{B} = \mathbf{AS} + \mathbf{E}$, with all elements in $\mathbb{Z}_q$. By setting the block dimension $J = 4$, matrix $\mathbf{A}$ can be reconstructed as being composed of $M \times M$ block matrices $\mathbf{A} = (\mathbf{A}_{i,j})_{1 \le i,j \le M}$, where $M = \frac{n}{J}$, and each block matrix $\mathbf{A}_{i,j} \in \mathcal{R}^{J \times J}$. Similarly, the secret matrix $\mathbf{S} \in \mathcal{R}^{n \times \bar{n}}$ (where $\bar{n} = 8$) is decomposed using a uniform block dimension $J = 4$. Therefore, $\mathbf{S}$ is reconstructed as being composed of $M \times 2$ block matrices $\mathbf{S} = (\mathbf{S}_{i,j})_{1 \le i \le M, 1 \le j \le 2}$, where $M = \frac{n}{J}$ and $\mathbf{S}_{i,j} \in \mathcal{R}^{J \times J}$. The matrices $\mathbf{A}$ and $\mathbf{S}$ are represented as follows:

$$A = \begin{pmatrix} \mathbf{A}_{1,1} & \dots & \mathbf{A}_{1,M} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{M,1} & \dots & \mathbf{A}_{M,M} \end{pmatrix}, \quad S = \begin{pmatrix} \mathbf{S}_{1,1} & \mathbf{S}_{1,2} \\ \mathbf{S}_{2,1} & \mathbf{S}_{2,2} \\ \vdots & \vdots \\ \mathbf{S}_{M,1} & \mathbf{S}_{M,2} \end{pmatrix} \tag{1}$$

where $\mathbf{E}_{i,j}, \mathbf{B}_{i,j} \in \mathcal{R}^{J \times J}$ for $1 \le i \le M$ and $1 \le j \le 2$. The large-scale matrix multiplication is partitioned into block matrix multiplications (As illustrated in Fig. 3):

$$\mathbf{B}_{i,j} = \sum_{k=1}^{M} \mathbf{A}_{i,k} \mathbf{S}_{k,j} + \mathbf{E}_{i,j} \tag{2}$$

In this architecture, each 4×4 matrix block is treated as a basic unit, where each coordinate (i, j) refers to a specific position in the block matrix array. When generating the block $\mathbf{A}_{i,j}$ in row i and column j, two corresponding 4×4 blocks are simultaneously fetched from row j of matrices $\mathbf{S}$ and $\mathbf{E}$. These blocks are multiplied with the current $\mathbf{A}_{i,j}$, and the results are accumulated into two 4×4 blocks in row j of matrix $\mathbf{B}$.

After generating all elements of the four rows of matrix $\mathbf{A}$, matrices $\mathbf{S}$ and $\mathbf{E}$ have been fully traversed once, completing a global update of matrix $\mathbf{B}$. By repeating this process, each time four rows of matrix $\mathbf{A}$ are generated in parallel (with one row defined at the

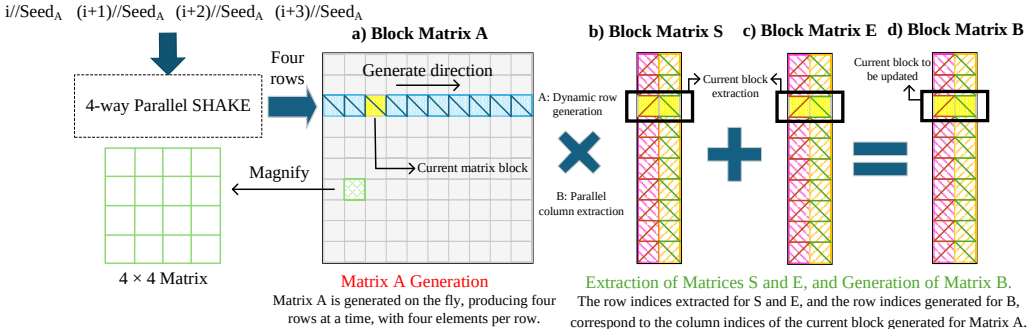


Figure 3: Block-wise Matrix Multiplication Architecture for $\mathbf{AS} + \mathbf{E}$.

block-matrix level), matrix $\mathbf{B}$ is cumulatively updated. Iterating this procedure $M = \frac{n}{j}$ times yields the final result of matrix $\mathbf{B}$.

The computation of $\mathbf{S'A} + \mathbf{E'}$ follows an analogous process and will not be discussed in detail here, and the overall scheduling is illustrated in Fig. 4. The only difference is that, when generating four consecutive rows of $\mathbf{A}$, the corresponding blocks from $\mathbf{S'}$ and $\mathbf{E'}$ remain unchanged.

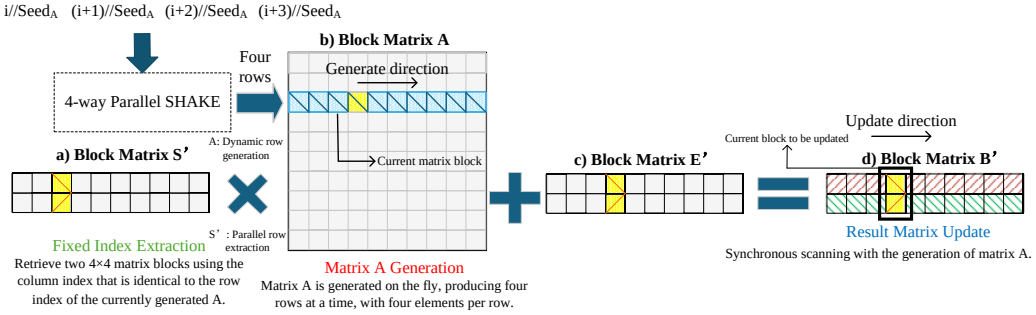


Figure 4: Block-wise Matrix Multiplication Architecture for $\mathbf{S'A} + \mathbf{E'}$.

3.4 Parallel Vector Dot Product Unit

The Parallel Vector Dot Product Unit receives block matrices from the Matrix Multiplier and computes the results of block matrix multiplication by performing vector dot product units in parallel. To shorten the critical path and reduce unnecessary hardware resource waste, this module avoids using conventional DSP multipliers. Instead, it computes vector dot products by decomposing the element-wise multiplications into partial-product summations. By applying high-order bits truncation, the algorithm effectively eliminates redundant high-order bits computations in modular arithmetic, thereby significantly reducing unnecessary hardware resource usage.

All matrix elements in FrodoKEM belong to $\mathbb{Z}_q$, and, as indicated in Section 3.4.1, the secret matrix elements are confined to a small range of $[-12, +12]$. This work exploits this favorable arithmetic property and proposes an operand-compression strategy for the vector dot product unit, reducing the bit-width of secret matrix elements involved in multiplication to 4 bits, thereby significantly decreasing the number of partial products. High-order partial products that do not contribute to the final result (above 16 bits) are discarded through truncation. In addition, a Carry-Save Adder (CSA) is used to construct

a Wallace tree [Wal64], implementing a partial-product accumulation process with minimal critical path delay. The proposed architecture ensures correct computations over $\mathbb{Z}_q$ while minimizing both hardware area and critical path latency.

3.4.1 Operand Compression

As defined in the **Sampling from the error distribution** algorithm (Algorithm 5), Elements (s, e) of matrices $\mathbf{S}$ and $\mathbf{E}$ are sampled from a 16-bit random string $r = (r_0, r_1, \ldots, r_{15})$ using a fixed distribution table T_χ:

Algorithm 5: Sampling from the Error Distribution χ

Input: 16-bit random string $r = (r_0, r_1, \ldots, r_{15})$
Output: Error value $s \in \{-d, \ldots, d\}$

$t \leftarrow \sum_{i=1}^{15} r_i \cdot 2^{i-1}$;
$e \leftarrow 0$;
for $i \leftarrow 0$ **to** $d-1$ **do**
  if $t > T_\chi(i)$ **then**
    $e \leftarrow e + 1$;
  end
end
$s \leftarrow (-1)^{r_0} \cdot e$;
return s;

According to the **Distribution Table Entries** in [ABD+25] , the maximum index d varies by parameter set: $d = 12$ for FrodoKEM-640, $d = 10$ for FrodoKEM-976, and $d = 6$ for FrodoKEM-1344. Consequently, the sampled results (s, e) are strictly confined to the range $[-d, d] \subseteq [-12, 12]$. In 16-bit two's complement representation, any value within the range $[-12, 12]$ exhibits redundancy of high-order bits (all 0s for $s \geq 0$ and all 1s for $s < 0$). Our design leverages this by extracting the sign bit of the secret matrix element s and selecting different partial-product generation strategies accordingly. Using only the effective lower 4 bits of s for multiplication, the complexity of a single multiplication is reduced from 16×16 bits to 16×4 bits, which decreases the hardware logic required for partial-product generation by approximately 75%.

The compression mechanism operates as follows:

• When the upper 12 bits of s are all zeros, the product with a 16-bit operand a is simply $a \cdot (s_3 s_2 s_1 s_0)_2$.

• When the upper 12 bits of s are all ones, the product with a 16-bit operand a can be decomposed as

$$\begin{aligned} a \cdot s &= a \cdot (1111_1111_1111_0000)_2 + a \cdot (s_3 s_2 s_1 s_0)_2 \\ &= a \cdot (2^{16} - 2^4) + a \cdot (s_3 s_2 s_1 s_0)_2 \\ &= a \cdot \left((s_3 s_2 s_1 s_0)_2 - 2^4\right) \pmod{2^{16}}. \end{aligned} \tag{3}$$

To further simplify the computation, define $b = 2^4 - (s_3 s_2 s_1 s_0)_2 = (\acute{s_3}\acute{s_2}\acute{s_1}\acute{s_0})_2 + 1$. Because $(s_3 s_2 s_1 s_0)_2 \in (0100, 1111)_2$ for negative samples drawn from the error distribution, b is always a positive 4-bit integer. Substituting this definition yields

$$a \cdot s = -(a \cdot b) \pmod{2^{16}} = (\overline{(a \cdot b)} + 1) \bmod 2^{16}. \tag{4}$$

That is, when the upper bits of s are all ones, it suffices to extract its lower 4 bits and apply a bitwise inversion followed by an increment by one to form a 4-bit operand for multiplication with a; the final result is then obtained by performing bitwise inversion and adding one to the product. This reformulation leads to a significant reduction in hardware resource consumption.

3.4.2 Truncation of High-Order Bits Beyond the Effective Modulo Range

The generation of multiplication partial products follows a scheme similar to long multiplication. In the context of the FrodoKEM encryption algorithm, a denotes a 16-bit operand with all bits being valid, while s denotes a sampled secret value. Each operand bit $s[j]$ $(j = 0, 1, 2, 3)$ is logically ANDed with every bit of a(a[i]) to generate a partial product vector PP_j. Specifically, each element of the partial-product vector is given by

$$\mathrm{PP}_j[i] = a[i] \cdot s[j], \quad i = 0, 1, \ldots, 15,\ j = 0, 1, 2, 3, \tag{5}$$

which corresponds to the value $a \times s[j]$. Since the weight of $s[j]$ is 2^j, the partial product PP_j is left-shifted by j bits before accumulation.

Since all operations are performed modulo 2^{16}, any bit in PP_j with a weight greater than 2^{15} does not affect the final result. Therefore, a truncated partial-product scheme is adopted, in which all bits at position 16 and above are directly discarded, and only the remaining lower bits are processed. This effectively avoids unnecessary hardware and latency overhead caused by invalid high-order computations. The resulting *low-bit valid window* employed in this design is illustrated in Fig. 5.

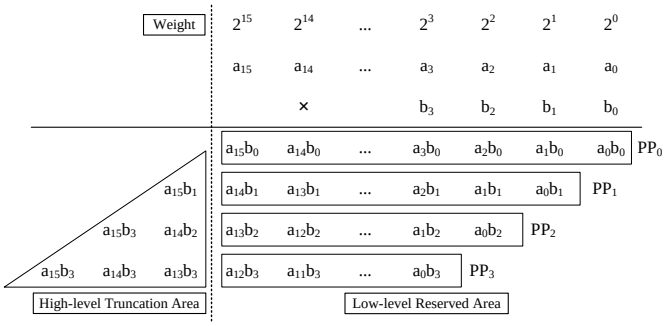


Figure 5: Implicit Modulo 2^q Truncation. High-order bits are discarded.

For each multiplication, four partial products with decreasing bit widths (16, 15, 14, and 13 bits, respectively) are generated, with their higher-order weights already aligned according to the corresponding shifts. To ensure uniform bit width during subsequent accumulation, zeros are padded into the vacant lower-bit positions introduced by shifting.

3.4.3 Wallace Tree Summation

For a 4×4 block matrix multiplication, a total of 16 vector dot products are involved, with each dot product comprising four element-wise multiplications. To correctly combine all partial products generated in a vector dot product and obtain the right result, two summation strategies can be employed. This section presents the optimized architecture adopted in this work, while the other summation strategy is discussed in Section 3.4.4.

In the design adopted in this work, a vector dot product is computed in a two-stage manner: the partial products within each element-wise multiplication are first reduced to obtain individual multiplication results, which are then accumulated across all element pairs to produce the final dot product result. To optimize the summation process, a CSA Wallace tree is employed. As illustrated in Fig. 6(a), two layers of Wallace trees are used to compress four partial-product vectors into a sum–carry pair, which is then resolved by a carry-propagate adder (CPA). Subsequently, a conditional two's-complement operation (bitwise inversion followed by addition of one) is applied to the result based on whether

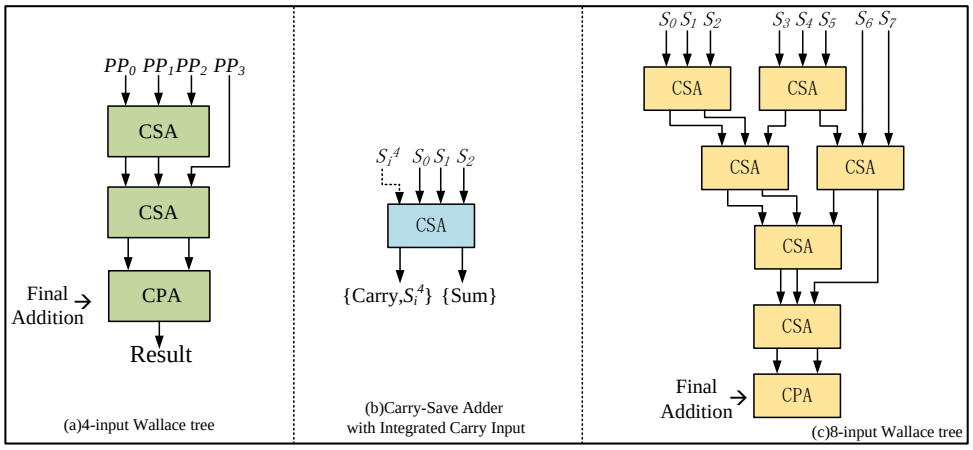


Figure 6: Multiplication Internal Microstructure.(a) Four-input Wallace tree.(b) CSA adder with fused external carry input.(c) Eight-input Wallace tree.

the most significant bit (MSB) of s is equal to one, yielding the product of a_i and s_i. The corresponding hardware architecture is shown in Fig. 7.

Finally, the same CSA Wallace tree structure as that used in the multiplier is employed to compute the vector dot product. At this stage, the four scalar multiplication results are provided as inputs and compressed to produce the final dot product result. This "white-box" arithmetic structure provides full transparency of the multiplication process. It allows designers to freely insert registers within the multiplication datapath to enable fine-grained pipelining, effectively breaking critical paths and achieving higher clock frequencies.

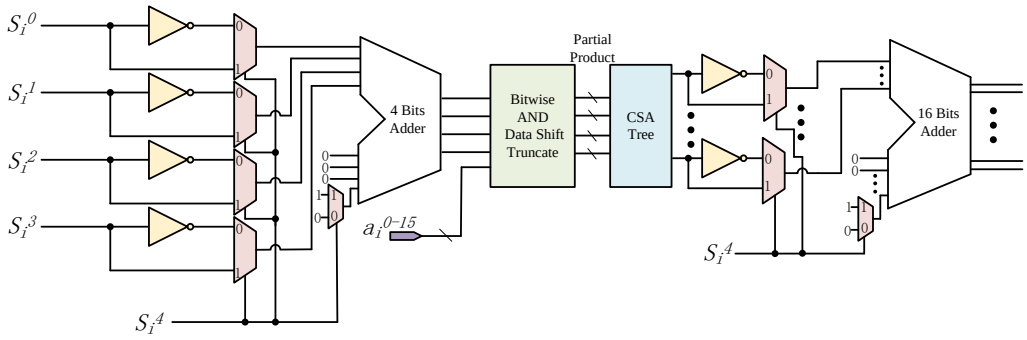


Figure 7: $a \times s$ multiplier circuit architecture with 4-bit s involved in multiplication.

3.4.4 Sign-Embedded Dot Product Architecture

Another summation approach increases the operand width to 5 bits and still applies a truncation structure similar to that shown in Fig. 5, while avoiding separate bitwise inversion and add-one operations at the input and output. In this case, s has five effective bits, with weights of -2^4, 2^3, 2^2, 2^1, and 2^0, respectively. The lower four bits are processed in the same manner as in the previous method. However, the partial-product generation associated with the sign bit differs from that of the lower bits. Specifically, if the sign bit is zero, the corresponding partial product is zero; otherwise, the partial product equals the bitwise inversion of a plus one within the effective truncation range, where the optimized add-one operation is introduced as described in the following paragraph.

A CSA produces two outputs, namely the carry and the sum. Since the least significant bit (LSB) of the carry output is zero, s_4 can be inserted at this position as an implicit "add-one" component without incurring any additional hardware cost (Fig. 6(b)). For each vector dot product, partial products of similar bit widths from the four multiplications are paired and fed into the Wallace tree (e.g., eight partial-product pairs of 16/15 bits and eight pairs of 14/13 bits). This pairing strategy reduces resource overhead caused by operand width expansion due to zero-padding at the lower bits. With its eight-input structure, the Wallace tree minimizes bit-level redundancy, thereby improving hardware efficiency (Fig. 6(c)).

However, the result indicates that the second method is not cost-effective: although it reduces related logic, it introduces an additional 1-bit effective operand width. For the FrodoKEM design, the two versions differ by only 968 LUTs, indicating a small but still non-negligible resource gap. Therefore, the first method is adopted to reduce hardware cost. Interestingly, when the operands are not constrained to a small numerical range (unlike FrodoKEM secret matrix elements), the second method becomes the more general and scalable approach. It is considered the more widely applicable solution for signed multiplications.

3.5 Memory

In this implementation of FrodoKEM, BRAMs are utilized to store the $8 \times n$ and $n \times 8$ matrices. The 4×4 matrix blocks of matrix $\mathbf{A}$, which are generated on-the-fly, and the 8×8 small matrices used during the Encapsulation and Decapsulation processes are stored in registers. Specifically, the memory module is primarily composed of two parts:

RAM_S_E_B: This component utilizes a simple_dual_port configuration. Each BRAM block is capable of storing a matrix of size $n \times 8 \times 16$ bits. The BRAM depth is configured as $n/4$, while the width is configured as either 64 bits or 512 bits(as shown in Fig. 8).

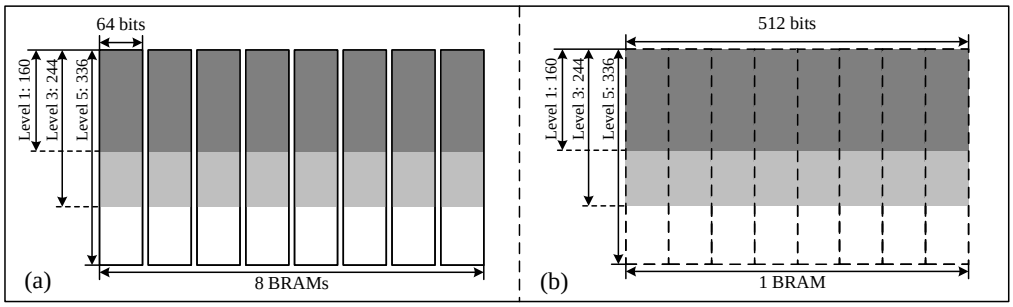


Figure 8: (a) Eight parallel BRAM groups for storing matrices **S** and **E**. (b) Entire BRAM block for storing matrix operation results.

For the 64-bit-width configuration, eight parallel BRAMs are grouped together as one memory block, which is used to store the matrices **S** and **E** generated by the SHAKE and sampling modules. During the write process, the write ports are connected to the sampling module, and the eight BRAMs are written sequentially. During the read process, however, all eight BRAMs are accessed in parallel. The amount of data read out in a single cycle is $8 \times 4 \times 16$ bits, corresponding to two 4×4 matrix blocks.

The 512-bit-width BRAMs are used to store intermediate matrices generated during computation. At the logical level, each BRAM is divided into eight equally sized partitions, where each partition has a capacity of $n \times 1 \times 16$ bits and is used to store either one column of an $n \times 8$ matrix or one row of an $8 \times n$ matrix. Physically, however, an entire matrix is

stored within a single BRAM block. The amount of data read from or written to each address is $4 \times 8 \times 16$ bits, which also corresponds to two 4×4 matrix blocks.

Therefore, both BRAM configurations are compatible with the previously described 4×4 block matrix multiplication module. Taking the write and read operations of the Secret Matrix $\mathbf{S}$ as an example, Fig. 9 illustrates its specific storage scheme. When generating the $8 \times n$ matrix via sampling, the matrix is generated row by row, and the elements of each row are stored sequentially into the 8 BRAMs from top to bottom. During the matrix multiplication process, to obtain two 4×4 block matrices, data is read in parallel from the corresponding address of the 8 BRAMs, thereby retrieving the corresponding four rows of the Secret Matrix $\mathbf{S}$.

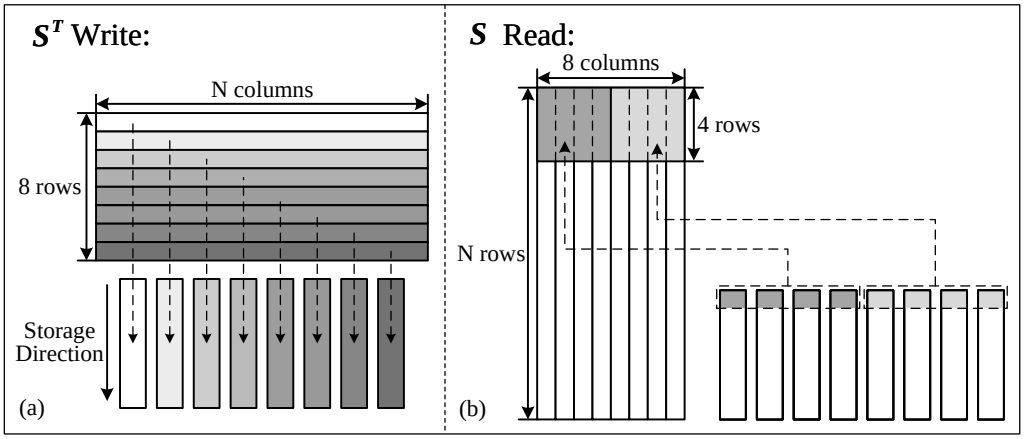


Figure 9: The Write And Read of the Secret Matrix $\mathbf{S}$.

REG_V_M_C: This part is used to store the 8×8 matrices that frequently appear in the FrodoKEM process, as well as their data obtained after the Pack operation. Given the small data volume, registers are adopted for memory to provide better flexibility in data access and modification.

To minimize memory resource consumption, we maximize the reuse of BRAM resources, ultimately compressing the memory requirement to accommodate four $8 \times n$ matrices, hereinafter referred to as four "memory blocks". Considering the Decapsulation stage, which imposes the highest demand on memory resources: the input c_1 occupies one memory block, and its unpacked form $\mathbf{B}'$ remains stored in this block. The input Secret Matrix $\mathbf{S}$ occupies one memory block; after executing the computation $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$, the subsequently generated matrix $\mathbf{S}'$ replaces the previously stored $\mathbf{S}$ in that block. Simultaneously, a new memory block is allocated for the generated $\mathbf{E}'$. During the computation of $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$, the result $\mathbf{B}''$ is stored in the block allocated to $\mathbf{E}'$. Finally, a memory block is provided for the input b, and the matrix $\mathbf{B}$ generated by Unpack(b) replaces b in that memory block. In summary, the entire FrodoKEM process can be completed using only four memory blocks.

Considering that the 18k and 36k BRAMs in FPGA possess maximum physical bit-width limits of 36 bits and 72 bits, respectively, in simple dual-port mode, while the proposed architecture employs both 64-bit and 512-bit BRAM configurations, the system automatically maps each memory block to eight RAMB36 units. When $n = 640$, the actual depth utilization of the BRAM is relatively low; however, when $n = 1344$, the depth utilization increases significantly while the data width remains constant. Consequently, the quantity of BRAM resources consumed remains unchanged. This characteristic of being "width-bound rather than capacity-bound" endows the proposed architecture with superior parameter scalability.

In ASIC implementations, the advantages of the proposed memory architecture become even more pronounced. The use of four memory blocks represents the minimum storage resources required to support a fully configurable FrodoKEM, achieving maximal memory reuse by storing the Pack and Unpack information within the same memory block. While this storage advantage is less evident on FPGAs with fixed-size memory resources, the proposed architecture minimizes the overall storage requirement through aggressive memory reuse. Moreover, the memory organization mainly consists of large contiguous memory blocks and requires only two groups of eight parallel BRAMs, thereby reducing memory fragmentation and simplifying the overall memory organization for ASIC implementation.

3.6 Control Unit

This module is designed to integrate the aforementioned modules to implement different algorithms in FrodoKEM. Following a bottom-up processing flow, it constructs the design architecture shown in Fig. 10. From bottom to top, and from concrete to abstract, it progressively realizes a unified algorithmic architecture that integrates the three major processes of FrodoKEM.

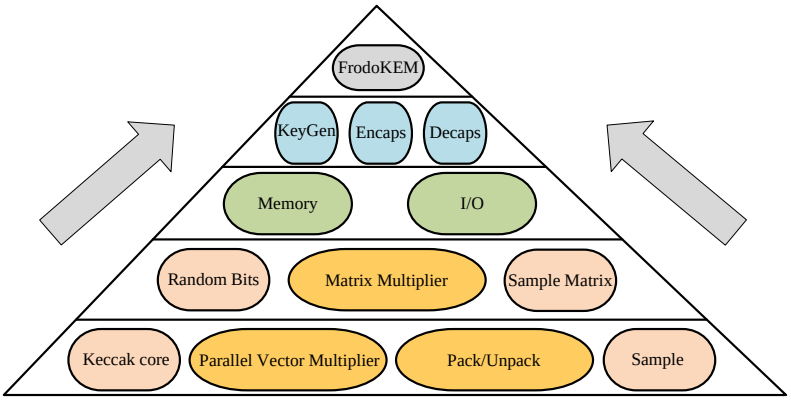


Figure 10: Hardware Control Hierarchy of FrodoKEM.

The most fundamental modules include the Keccak core, the Parallel Vector Dot Product module, the Pack/Unpack module, and the Sampling module. Building upon these four modules, we construct the Random Bits Generator, the Matrix Multiplier, and the matrix sampling generation algorithm for matrices **S** and **E**. Subsequently, by integrating storage and external input/output capabilities, we implement the three major functions of FrodoKEM: KeyGen, Encaps, and Decaps. These components are integrated onto a single development board and can be selected via control signals.

4 Implementation Results and Comparison

The proposed architecture is implemented on Xilinx Virtex-7 XC7V200 and Artix-7 XC7A200 FPGAs using the Vivado 2023.2 design suite, while an ASIC implementation is obtained using the SMIC 40 nm LL process (TT, 1.1 V, 25°C) to provide a more comprehensive evaluation of its hardware efficiency.

4.1 Resource and Performance Results

This work designs and implements a unified hardware architecture integrating KeyGen, Encaps, and Decaps for all FrodoKEM security levels. The architecture switches between different functionalities through mode selection signals, enabling the execution of multiple FrodoKEM algorithm flows on the same platform without re-synthesis. As a result, hardware resource reuse efficiency and system deployment flexibility are significantly improved, demonstrating strong engineering practicality and applicability.

FPGA Results: Table 3 summarizes the resource consumption of the whole design and the major modules implemented on both the Xilinx Artix-7 and the Xilinx Virtex-7 platforms. As indicated in the table, the Random Bits Generator accounts for approximately 40% of the total LUTs and 54% of the FFs, constituting the primary resource overhead. This is attributed to the deployment of four parallel SHAKE modules to match the generation rate of matrix $\mathbf{A}$ with its consumption rate. The Vector Dot Product Unit occupies 29% of the LUTs and 8% of the FFs. Although the Vector Dot Product Unit itself has been significantly optimized, the instantiation of two 16-way parallel DSP-free Vector Dot Product Units inevitably results in considerable resource consumption. The Pack and Unpack modules account for 1% and 4% of the total LUTs and FFs, respectively, possessing relatively simple logic. Finally, as discussed in Section 3.5, when upgraded to higher security levels (FrodoKEM-976 and FrodoKEM-1344), the architecture can directly utilize the depth redundancy of existing BRAMs to store larger matrices, which means that the BRAM resource usage can still be maintained at a constant 32 units. This design allows the architecture to achieve high performance while offering exceptional engineering scalability and hardware resource efficiency.

Table 3: Resource usage of different modules on the FPGA platform.

FPGA Model	Artix-7				Virtex-7			
Module	LUTs	FF	DSPs	BRAMs	LUTs	FF	DSPs	BRAMs
RBG[1]	22175	13496	0	0	22182	13859	0	0
VDPU[2]	17434	2049	0	0	16457	2049	0	0
Pack & Unpack	480	960	0	0	512	1024	0	0
BRAM Array	0	0	0	32	0	0	0	32
Control Unit	16329	8619	0	0	17075	8666	0	0
Total	56418	25124	0	32	56226	25598	0	32

[1] RBG denotes Random Bits Generator.
[2] VDPU denotes Vector Dot Product Unit.

ASIC Results: To further demonstrate the efficiency and ASIC portability of our FrodoKEM architecture, we performed logic synthesis using Synopsys Design Compiler based on the SMIC 40-nm LL process (TT, 1.1 V, 25°C) to obtain the ASIC synthesis results. Table 4 presents the area performance of our FrodoKEM architecture. Meanwhile, to verify the area efficiency of the optimized DSP-free multiplier in ASIC implementations, we conducted additional experiments targeting a single 4×4 multiplication module. The results show that a single optimized 4×4 multiplier architecture requires 30.4K equivalent gates, compared to 49.4K for the unoptimized version. This directly indicates that the proposed architecture reduces the ASIC area by approximately 40%.

Table 4 reports the area of the proposed FrodoKEM architecture under ASIC synthesis evaluation. The synthesized logic area occupies only 0.29 mm^2, demonstrating promising hardware efficiency for ASIC implementation. In addition, To minimize storage overhead, the proposed architecture aggressively reuses memory resources throughout the entire FrodoKEM flow. As a result, the total matrix storage requirement is reduced to only 698 kb, including all intermediate matrices and temporary storage.

Table 4: Area performance of different modules on ASIC platform.

	Area(μm^2)	**Logic Gates (10^3 NAND2 equiv)**	**Percentage**
Logic	290,486	455.0	100%
RBG	124,313	194.7	42.795%
VDPU	69,930	109.5	24.932%
Pack & Unpack	10,460	16.4	3.601%
Control Unit	85,783	134.4	29.531%

4.2 Comparison

This section compares the proposed hardware architecture with existing mainstream FrodoKEM FPGA implementations. Prior works mainly include resource-reuse-based embedded implementations [HOKG18], software–hardware co-design architectures based on RISC-V instruction set extensions [FSS20], high-level synthesis (HLS)-based implementations [Bus21], as well as recently proposed high-speed hardware architectures [HMOR21, DGA25, LLY+26]. Compared with these approaches, the proposed architecture achieves significant reductions in FrodoKEM execution time and delivers competitive Area-Time Product (ATP) performance, processing strong potential for efficient memory utilization and high-performance acceleration on ASIC platforms. What's more, this work presents a configurable hardware implementation of FrodoKEM. We aggregate the reported resource utilization and timing overheads of existing designs with separately implemented functionalities, and compare them with our integrated implementation. The detailed comparison results are summarized in Table 5.

To date, the work by [LLY+26] represents the fastest existing design, but this work requires only about 40% (Artix-7) of the runtime of the implementation by [LLY+26] for all three functions.

Regarding timing optimization, this design does not rely on DSP multipliers; instead, an efficient custom hardware multiplier is developed by exploiting the properties of modular arithmetic, thereby eliminating the inefficiency of DSP bit-width utilization caused by modular operations. Furthermore, due to the transparent "white-box" structure of the proposed architecture, which allows for fine-grained pipelining of the multiplication module, we have successfully reduced the critical path and increased the clock frequency. Even with the integration of the three core functions, the proposed architecture is still able to maintain an operating frequency of 182 MHz on Virtex-7 and 138 MHz on Artix-7, and the operating frequency can be further increased by introducing more pipeline stages into the fully transparent architecture.

As analyzed above, this work achieves a substantial acceleration and a significant reduction in execution latency for FrodoKEM. As shown in Table 6, the proposed design consistently outperforms existing FrodoKEM FPGA implementations in terms of execution latency, demonstrating clear advantages in both architectural efficiency and overall performance.

Furthermore, we must acknowledge that the deliberate avoidance of dedicated DSP blocks and the parallel deployment of SHAKE modules in this work lead to an increase in LUT usage, resulting in no advantage in terms of LUT consumption compared to existing implementations. However, under this design choice, the proposed architecture is expected to exhibit significant advantages on ASIC platforms: The design for operand compression and high-order truncation can drastically reduce the usage of logic gates, thus significantly decreasing chip area.

Table 5: Resource and Execution Time Comparison of FrodoKEM FPGA Implementations. For each design, the three rows of time and resource results correspond to three stages: KeyGen, Encaps, and Decaps.

Design	Security Level	Area					Frequency (MHz)	Platform	Time(ms)/ATP[2]		
		LUT	FF	DSP	BRAM	Slice[1]			KeyGen	Encaps	Decaps
This work	**FrodoKEM-640**								**0.256/5.5**	**0.275/5.9**	**0.278/6.0**
	FrodoKEM-976	**56226**	**25598**	**0**	**32**	**21493**	**182**	**Virtex-7**	**0.544/11.7**	**0.577/12.4**	**0.581/12.5**
	FrodoKEM-1344								**0.963/20.7**	**1.002/21.5**	**1.010/21.7**
This work	**FrodoKEM-640**								**0.338/7.3**	**0.363/7.8**	**0.367/7.9**
	FrodoKEM-976	**56418**	**25124**	**0**	**32**	**21541**	**138**	**Artix-7**	**0.717/15.5**	**0.761/16.4**	**0.766/16.5**
	FrodoKEM-1344								**1.270/27.4**	**1.321/28.5**	**1.332/28.7**
[Bus21]	FrodoKEM-640[3]	12687	6719	1	6	4669	167	Artix-7	19.74/92.2	–	–
		11396	7024	1	10	5275	167		–	19.96/105	–
		11792	7451	1	10	5374	162		–	-	20.76/111.6
[HOKG18]	FrodoKEM-640[3]	3771	1800	1	6	2440	167	Artix-7	19.65/47.9	–	–
		6745	3528	1	11	4345	167		–	19.89/86	–
		7220	3549	1	16	5626	162		–	–	20.72/116.6
	FrodoKEM-976[3]	7139	1800	1	8	3746	167	Artix-7	45.63/170.9	–	–
		7209	3537	1	16	5623	167		–	46.00/258.7	–
		7773	3559	1	24	7623	162		–	–	47.81/364.5
[HMOR21]	FrodoKEM-640[3]	971	433	1	0	345	191	Artix-7	16.95/5.8	–	–
		4246	2131	1	0	1164	190		–	17.24/20.1	–
		10518	2299	1	0	2732	190		–	–	17.54/47.9
[DGA25]	FrodoKEM-640[3]	12246	8859	24	18	9702	147	Artix-7	1.024/9.9	–	–
		15908	12111	32	20	11902	136.98		–	0.928/11.0	–
		17066	12868	32	20	12191	133.15		–	–	0.95/11.6
[LLY+26]	FrodoKEM-640								0.859/5.6	0.876/5.7	0.883/5.8
	FrodoKEM-976	13423	6042	0	14	6609	207	Artix-7	1.788/11.8	1.815/12.0	1.826/12.1
	FrodoKEM-1344								3.160/20.9	3.197/21.1	3.212/21.2

[1] Slice is calculated as Slice = LUTs/4 + 232.4 × BRAMs + 102.4 × DSPs for 7-series FPGAs. [DNKYL22]
[2] Area and Time Product (ATP) = Slice × time(s).
[3] The three processes of KeyGen, Encaps and Decaps cannot be configured and implemented on the same FPGA platform simultaneously.

Table 6: Comparison of Clock Cycles required by different FrodoKEM hardware implementations.

Design	Security Level	Clock Cycles		
		KeyGen	Encapsulation	Decapsulation
[HOKG18]	1	3277k	3318k	3359k
[HOKG18]	3	7921k	7683k	7746k
[Bus21]	1	3291k	3329k	3366k
[HMOR21]	1	3240k	3275k	3334k
[DGA25]	1	150k	127k	128k
[LLY+26]	1	178k	181k	183k
	3	370k	376k	378k
	5	654k	662k	665k
This work	1	47k	50k	51k
	3	99k	105k	106k
	5	175k	182k	184k

Moreover, the proposed design is further evaluated from the perspective of time–resource co-optimization by introducing the Area-Time Product (ATP), whose calculation method is shown in Table 5 and the corresponding footnote. ATP provides a comprehensive metric for characterizing the overall performance of a hardware architecture under different optimization strategies by jointly considering resource consumption and execution time, where a smaller ATP value indicates better overall efficiency.

According to the results listed in Table 5, after excluding the implementation in [HMOR21], which does not comply with the official FrodoKEM specification, the configurable architecture proposed in this work significantly outperforms the vast majority of existing implementations. It is worth noting that [LLY+26] reaches a better ATP via intensive resource optimization. Our design, by contrast, focuses more on runtime performance. It delivers the minimal execution latency across all benchmarks and runs up to 2.5× faster than the implementation in [LLY+26]. Other than that, even when the non-compliant design in [HMOR21] is included in the comparison, the ATP achieved by our design in the encapsulation and decapsulation stages remains superior. In addition, extending the architecture in [HMOR21] to support configurability would inevitably result in a total Slice count exceeding that of any individual stage, while the operating frequency would also degrade due to increased integration complexity, leading to longer execution time. In summary, this configurable hardware architecture delivers a substantial speedup and achieves competitive ATP performance.

5 Conclusion

In this work, we present an efficient hardware architecture for FrodoKEM. The key advantage of our design lies in a low-complexity block multiplier and a high-speed architecture. This implementation achieves the fastest performance for FrodoKEM and we expect these advances to strengthen the competitiveness of FrodoKEM within the PQC landscape.

References

[ABD+25] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM: Learning with errors key encapsulation — preliminary standardization proposal. `https://frodokem.org/files/FrodoKEM_standard_proposal_20250929.pdf`, 2025.

[Bus21] Fabian Buschkowski. Efficient hardware implementation of post-quantum cryptography using high-level synthesis. 2021.

[CCJ+16] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray A Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology . . . , 2016.

[D+15] Morris J Dworkin et al. SHA-3 standard: Permutation-based hash and extendable-output functions. 2015.

[DCP08] Christophe De Cannière and Bart Preneel. Trivium. In *New Stream Cipher Designs: The eSTREAM Finalists*, pages 244–266. Springer, 2008.

[DGA25] Gökçe Düzyol, Muhammed Said Gündoğan, and Atakan Arslan. Can FrodoKEM run in a millisecond? FPGA says yes! *Cryptology ePrint Archive*, 2025.

[DNKYL22] Phap Duong-Ngoc, Sunmin Kwon, Donghoon Yoo, and Hanho Lee. Area-efficient number theoretic transform architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(3):1270–1283, 2022.

[Fed25] Federal Office for Information Security, Germany. Cryptographic mechanisms: Recommendations and key lengths. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf`, January 2025.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 239–280, 2020.

[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(4):317–327, 2021.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *Cryptology ePrint Archive*, 2018.

[ISO25] ISO/IEC JTC 1/SC 27. ISO/IEC 18033-2:2006/FDAmd 2 – information technology – security techniques – encryption algorithms – part 2: Asymmetric ciphers. ISO/IEC Final Draft Amendment, 2025. Status: Under development.

[LLY+26] Kai Li, Jiahao Lu, Fu Yao, Guang Zeng, Dongsheng Liu, Shengfei Gu, Zhengpeng Zhao, and Jiachen Wang. A high performance and efficient post-quantum crypto-processor for frodokem. *arXiv preprint arXiv:2601.16500*, 2026.

[Net23] Netherlands National Communications Security Agency. The PQC migration handbook. `https://publications.tno.nl/publication/34641918/oicFLj/attema-2023-pqc.pdf`, December 2023.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[Wal64] Christopher S Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, 13(1):14–17, 1964.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 328–356, 2021.