FrodoKEM Hardware Implementation for Post-Quantum Cryptography

Fernando Aparicio Urbano-Molano, *Senior Member, IEEE,* and Jaime Velasco-Medina, *Senior Member, IEEE*

***Abstract*—FrodoKEM, a key encapsulation mechanism (KEM) based on the learning with errors (LWE) problem, would be included for standardization by the International Organization for Standardization (ISO) and recommended for post-quantum cryptography (PQC) migration by the BSI (German Federal Office for Information Security) and the ANSSI (French Cybersecurity Agency). It is closely related to the challenging time-computational problem inherent to algebraically unstructured lattices. However, hardware implementations of this scheme are required to verify its effectiveness in real-world applications. To the best of our knowledge, this is the first hardware implementation of FrodoKEM using High-Level Synthesis (HLS), which meets all requirements of the version submitted for standardization to ISO. The proposed design started with the profiling of the reference C software implementation using Valgrind software tools, to identify the functions that are the most time-consuming. The advantages of the proposed implementation include a 34% improvement in the speed metric of the Key Generation module in comparison with the reference software implementation. The results show that the key generation, encapsulation, and decapsulation use 26%, 39%, and 32%, respectively, of the total area utilization on the Artix-7.**

Link to graphical and video abstracts, and to code: https://latamt.ieeer9.org/index.php/transactions/article/view/9651

***Index Terms*—Lattice-Based Cryptography, Post-Quantum Cryptography, FrodoKEM, High-Level Synthesis (HLS), FPGA.**

I. Introduction

ONE of the biggest challenges of current cryptography is how not to lose its security when the quantum computer arrives and thus not generate fear among users. Peter Shor demonstrated that computational problems such as discrete logarithms and bigger integer factoring can be solved faster in a hypothetical quantum computer making traditional public key cryptography ineffective [1]. Then, to address the above issue, post-quantum cryptosystems have been proposed such as Hash-based, code-based, and Lattice-based cryptography among others [2].

According to NIST (National Institute of Standards and Technology) from the U.S. Department of Commerce, actual public key cryptography infrastructure took almost 20 years to implement and it will necessary a great effort to guarantee a secure migration from current cryptosystems widely used to the new quantum computing resistant. Therefore, NIST in December 2016 requested nominations for Public-Key post-quantum cryptographic Algorithms and its goal is to standardize PQC systems that achieve the highest security against attacks from both classical and quantum computers. Then, the proposed NIST evaluation parameters were security analysis, performance, and implementation characteristics.

Agencies such as BSI and ANSSI have decided to recommend for PQC migration, in addition to the algorithms standardized by NIST, more conservative ones like FrodoKEM, a key-encapsulation mechanism based on algebraically unstructured lattices that would be included for standardization by ISO, and was part of the third round of the NIST process. Thus, in this work, we have implemented the ephemeral FrodoKEM-640 -SHAKE128 algorithm on an FPGA (Field Programmable Gate Array) using High-Level Synthesis and compared its performance with similar works. Using this language, we carried out one first hardware implementation for the later design of this algorithm using a traditional HDL (Hardware Description Language) to achieve better performance.

This paper makes the following contributions:

1) A profile of the reference C software implementation submitted to ISO standardization is developed to identify the most time-consuming functions, prioritizing their implementations in hardware.
2) Parallel design of the Gaussian samplers S and E for the FrodoKEM-640 considering a constant time.
3) Improvements to the reference C software implementation for speeding up the generation of large public matrix (A) using SHAKE128, securing protection against side-channel attacks (SCA) by fly generating one row of A.
4) Hardware design of the key generation, encapsulation, and decapsulation modules of FrodoKEM-640 using HLS to reduce time-consuming design, achieving a good trade-off between speed and area.
5) An analysis of clock cycles and the area of the key generation implementation to determine which blocks required improvement in future versions.

The rest of this paper is organized as follows. Section II introduces related research works. Section III explains the background. Section IV describes the methodology. In Section V, we explain the experimental results and discussions in

The associate editor coordinating the review of this manuscript and approving it for publication was Carolina Del-Valle-Soto (*Corresponding author: Fernando Aparicio Urbano-Molano*).

This work was supported by the Ministry of Science and Technology and the SGR of Colombia.

Fernando Aparicio Urbano-Molano, and J. Velasco-Medina are with the School of Electrical and Electronics Engineering, Universidad del Valle, Cali, Colombia (e-mails: fernando.urbano@correounivalle.edu.co, and jaime.velasco@correounivalle.edu.co).

detail. Finally, conclusions are given in Section VI.

II. Related Works

Since the request for the nomination of PQC algorithms by NIST, several have been presented; however, some have not been implemented in hardware due to their complexity. For hardware implementation, the NIST team suggested the use of Cortex-M4 for microcontrollers and AMD Xilinx Artix-7 for FPGA. This decision was to facilitate comparison between hardware performance data.

In [3], the authors presented the first hardware implementation of an area-optimized hardware architecture of a lattice-based cryptographic scheme, implementing the standard-LWE encryption on a Spartan-6 FPGA. It incorporates a fast-discrete Gaussian sampler, designed to generate samples in parallel and without delay to the critical path. The implementation results show that this encryption scheme can achieve competitive performance compared to ring-LWE hardware designs. Finally, this work discusses the security advantages of standard lattices over ideal lattices, emphasizing that standard-LWE does not require the additional security assumptions associated with structured ideal lattices. This aspect highlights the robustness of the proposed scheme against potential attacks.

In [4], the authors implemented the version of FrodoKEM submitted to the first round of NIST on microcontrollers and FPGA. For hardware implementation, they used cSHAKE (Secure Hash Algorithm-3) [5], and AES (Advanced Encryption Standard) as a pseudo-random number generator (PRNG). The design used a vector-matrix multiplication operation for carrying out the matrix-matrix multiplication (called matrix multiplication) to save area. The FPGA implementation achieves output rates of 51 results per second for key generation and encapsulation, and 49 results per second for decapsulation. However, this hardware implementation required more than three million cycles to generate the result. The ARM (Advanced RISC Machine) implementation also shows optimized memory allocation, fitting well within the constraints of embedded microcontrollers.

To achieve parallelization of some operations, particularly vector-matrix multiplication, in [6] the authors used Trivium instead of cSHAKE, which does not comply with the requirements of the proposal submitted to NIST. The authors also explore the integration of first-order masking into the decapsulation module, which is essential for countering side-channel attacks. This addition improves security and complements the overall performance enhancements achieved through parallelization. The use of Trivium improves both area and speed compared to the previous works, and they use a directive not to generate BRAM (Block RAM); however, it does not specify the number of clock cycles required to generate the result.

In [7], the authors studied how to improve the matrix multiplication using the Strassen algorithm achieving improvements up to 30% over the straightforward FrodoKEM approach. However, according to the authors, this approach only applies to AES128 and matrices must be stored entirely. This technique may be useful for high-speed devices with an AES accelerator and no area constraints, but it is not suitable for low-area.

Authors in [8] introduce a discrete Gaussian sampling hardware design that is both efficient and flexible. The design can support various sampling parameters, making it adaptable for different cryptographic applications. A critical feature of the proposed design is its constant-time behavior, which eliminates timing side-channel attacks. This is achieved by ensuring that all memory elements are accessed simultaneously, thus preventing any timing discrepancies that could be exploited. The design employs a Cumulative Distribution Table (CDT) approach, reduces table size through Gaussian convolutions, and utilizes a fusion tree search algorithm. This combination results in a compact and fast sampling technique, marking the first hardware implementation of the fusion tree search algorithm. The design has an advantage because can support the discrete Gaussian distributions used in Falcon and FrodoKEM.

Because of the complexity of PQC algorithms, it is essential to explore methodologies like HLS and co-design to speed up hardware implementation time. For example, authors in [9] presented the implementation of FrodoKEM, Round5, and Saber. The matrix multiplication and SHAKE128/256 hash functions were implemented in RTL (Register Transfer Level) and the rest of the scheme in the ARM core. This analysis provides valuable insights into the performance characteristics of these schemes under this new benchmarking framework. The results show that the total speed-up for all analyzed schemes exceeds a factor of 7, with the highest speed-up reaching 28.4 for the FrodoKEM scheme. This indicates that the proposed SW/HW co-design approach can lead to substantial performance improvements in PQC implementations.

In [10], the authors presented the hardware implementation of FrodoKEM, using a similar methodology to the previous work. This implementation is specifically designed for hardware-constrained environments like smart meters, addressing the need for quantum-secure cryptographic solutions in these applications. A novel SoC (System-on-Chip) - FPGA architecture is proposed to accelerate the most time-consuming operations of the FrodoKEM scheme. This approach allows for a more efficient execution of the cryptographic routines, which is crucial for the performance of smart meters. The work provides a detailed analysis of the execution time and hardware resource usage of the proposed SoC implementation. Experimental results indicate that the execution time is reduced to one-third compared to the benchmark software implementation, demonstrating the effectiveness of the proposed solution in enhancing performance for smart meter applications. However, there is no information concerning the total area of the design and the number of clock cycles used.

In [11], the authors use an HLS-based design methodology for extensive design-space exploration. This methodology is useful for generating RTL descriptions from high-level specifications, allowing design optimizations for specific constraints like area and latency. This work discusses optimizations like loop *unrolling* and loop pipelining that improve the latency of PQC implementations. These optimizations are crucial for enhancing the efficiency of hardware designs. The FrodoKEM implementation results using a Virtex-7 FPGA are 128031

LUT (Lookup Table) and 117736 clock cycles for encapsulation, and 179290 LUT and 335891 clock cycles for decapsulation.

The authors in [12] deployed the FrodoKEM (version 2021) scheme over a RISC-V (Reduced Instruction Set Computer) using hardware/software co-design that supports all three parameter sets of the scheme. To enhance performance, this work introduces tightly coupled hardware accelerators specifically designed for the computationally intensive tasks involved in FrodoKEM. The authors provide a detailed performance analysis, demonstrating speedup factors of up to 8.13 when comparing the accelerated design to plain software implementations. This shows the effectiveness of the proposed hardware acceleration.

In [13], the authors claim to be the first to implement GPU (Graphics Processing Unit) acceleration for three post-quantum key exchange algorithms: FrodoKEM, NewHope, and Kyber. This is a notable advancement as it explores the potential of using CUDA (Compute Unified Device Architecture) for processing these algorithms, which are critical in the context of quantum resistance in cryptography. The work presents two distinct implementation strategies: batch mode and single mode processing on multiple GPUs. This dual approach allows flexibility in executing, catering to different computational needs and environments. For FrodoKEM-976 a significant speedup of 50.6x was achieved for KeyGen, 44.2x for Encaps, and 36.9x for Decaps compared to the reference C software implementation.

In [14], the authors introduced optimized techniques for parallel matrix multiplication specifically for FrodoKEM-640 - AES. This implementation leverages the vector registers and vector instructions available in ARMv8 processors, allowing efficient computation of matrix operations. The proposed method can compute 80 elements of the output matrix simultaneously, significantly enhancing performance. The authors also used the build-in AES accelerator for AES encryption on the ARM processor, and applied these techniques to the FrodoKEM-640 scheme, further enhancing performance. In the best-case scenario, the FrodoKEM implementation with the AES accelerator demonstrated a performance increase of 3.33x compared to the reference C software implementation.

Authors in [15] introduced a new implementation of FrodoKEM-AES using the ARM Neon instruction set as a baseline, which already set speed records on Apple's M1 and M3 processors. The work presents an AMX (Apple Matrix eXtensions) instruction implementation that improves upon the ARM Neon, showcasing the potential of AMX for enhancing cryptographic performance. The authors explored various matrix multiplication strategies and introduced a novel technique for generating matrix A in FrodoKEM-AES. They utilized AMX-unique *genlut* instruction for Gaussian sampling, achieving improvements of up to 418% compared to the ARM Neon implementation.

As described above, most of the previous hardware implementation were for the version of FrodoKEM submitted to the first round of NIST. However, the actual version of FrodoKEM (2024) uses SHAKE instead of cSHAKE; the Gaussian parameter σ was modified from 2.75 to 2.8; the generation of pseudo-random bits for error matrices S and E in KeyGen and S', E', and E'' in Encaps were changed, and to reduce the risk of batch attacks targeting multiple keys, it includes the hash value of the public key pkh in the calculation of the random bit-string r. This last part modifies the scheme by adding the SHAKE function to the key generation and encapsulation algorithms. In this work, we contribute to the state-of-the-art of PQC hardware implementation, presenting the design and synthesis results of the up-to-date version of the FrodoKEM key-generation, encapsulation, and decapsulation modules and exploring the capabilities of the profiling tools and the HLS design flow.

TABLE I
SIZES IN BITS FOR $seed_{SE}$ AND $seed_{salt}$ FOR FRODOKEM

	FrodoKEM-640	**FrodoKEM-976**	**FrodoKEM-1344**
len_{SE}	256	384	512
len_{salt}	256	384	512
	eFrodoKEM-640	**eFrodoKEM-976**	**eFrodoKEM-1344**
len_{SE}	128	192	256
len_{salt}	0	0	0

III. FRODOKEM KEY-ENCAPSULATION MECHANISM

A. Learning with Errors

In 2005, Regev introduced the learning with errors problem [16]. This mathematical problem has become fundamental in PQC due to its assumed resistance to attacks from quantum computers.

LWE can be defined as follows: Let n, $q \in \mathbb{Z}^+$, and an 'error' probability distribution χ on $\mathbb{Z}_q$. Let $A_{s,\chi}$ be the distribution, $e \in \mathbb{Z}_q$ been an error sampled from χ, $s \in \mathbb{Z}_q^n$ been the secret, $a \in \mathbb{Z}_q$ being uniformly chosen at random, and generating as output:

$$(a, b = <a, s> + e \mod q) \in \mathbb{Z}_q^n, \mathbb{Z}_q \tag{1}$$

The challenge in the LWE problem is to find out the secret vector *s* given many pairs of (a_i, b_i) where each pair follows the equation above. Without the error (i.e., $e = 0$), it would be relatively easy to find the secret vector *s*. However, with the small error *(e)* added to each equation, the problem becomes more difficult. So far, no quantum algorithm is known that can solve this problem in polynomial time.

B. FrodoKEM

FrodoKEM is a post-quantum key-encapsulation mechanism, which means that it is designed to exchange keys securely between two parties. It uses the LWE problem involving noisy linear equations to achieve cryptographic security [16]. It is part of the family of lattice-based cryptographic schemes that rely on unstructured lattices, making it more conservative in terms of security assumptions.

It is designed for IND-CCA (Indistinguishability under Chosen Ciphertext Attack) and has three versions, FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344 for NIST security levels 1, 3 and 5. Each security level matches or exceedes

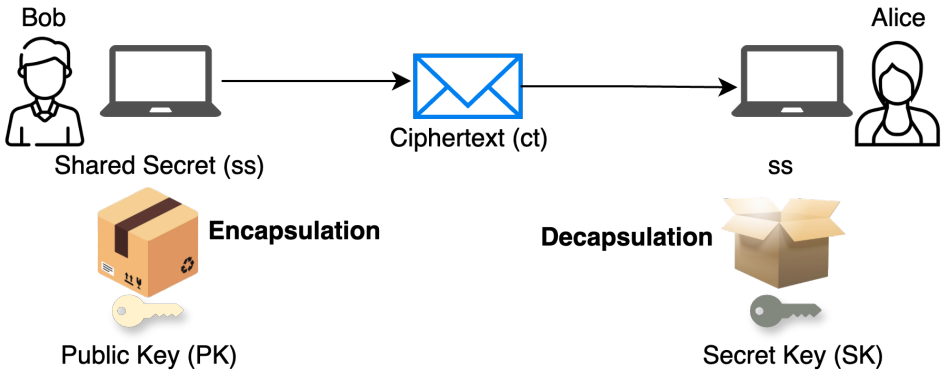

Fig. 1. Key Encapsulation Mechanism.

the brute-force security of AES-128, AES-192, and AES-256. To pseudo-randomly generate a large public matrix (called A) there are two approaches. The first one is to use AES128 for achieving hardware acceleration, and the second one is SHAKE128 for providing better performance in the absence of AES hardware acceleration.

Ephemeral instances of FrodoKEM (eFrodoKEM-640, eFrodoKEM-976, and eFrodoKEM-1344) are only meant for applications that ensure a small number of ciphertexts (e.g., 2^8), optimal for resource-constrained environments such as those used for the Internet of Things environment. Table I shows the values of $seed_{SE}$ and $seed_{salt}$ for each one of the versions.

Fig. 1 shows the diagram block of the Key Encapsulation Mechanism, which has three main components:

1) Key generation: Alice generates a public and secret key pair, and sends the public key to Bob. The public key contains a matrix **A** and some noisy values related to a secret vector **s**.
2) Encapsulation: Bob uses the public key to encapsulate a shared secret key. He sends the ciphertext to Alice and keeps the shared secret.
3) Decapsulation: Alice receives the ciphertext and uses her secret key to decapsulate it, recovering the same shared secret.

Algorithm 1 FrodoKEM.KeyGen

Input: None.
Output: (pk, sk).

1: $s||seed_{SE}||z \xleftarrow{\$} U(\{0,1\}^{len_s+len_{seed_{SE}}+len_z})$
2: $seed_A \leftarrow SHAKE(z, len_A)$
3: $A \leftarrow Frodo.Gen(seed_A)$
4: $bit\_string \leftarrow SHAKE(0X5F||seed_{SE}, 32n\bar{n})$
$bit\_string \leftarrow (r^{(0)}, r^{(1)}, ..., r^{(2n\bar{n}-1)})$
5: $S^T \leftarrow Frodo.Sample(r^{(0)}, r^{(1)}, ..., r^{(n\bar{n}-1)}, \bar{n}, n)$
6: $E \leftarrow Frodo.Sample(r^{(n\bar{n})}, r^{(n\bar{n}+1)}, ..., r^{(2n\bar{n}-1)}, n, \bar{n})$
7: $B \leftarrow AS + E$
8: $b \leftarrow Frodo.Pack(B)$
9: $pkh \leftarrow SHAKE(seed_A||b, len_s)$
10: **return** $pk \leftarrow (seed_A||b), sk \leftarrow (s||pk||S^T||pkh)$

The LWE problem is used to generate the key-pair as the matrix $B = AS + E$, this generates multiple tuples (a_i, b_i). According to this principle, the matrix operations are the most time-consuming. The three components are carried out by the algorithms KeyGen, Encaps, and Decaps [17].

In this paper, we present the hardware implementation of ephemeral FrodoKEM-640 - SHAKE128, which focuses on accelerating the key generation described in Algorithm 1. It uses uniformly random values to generate the required seeds. FrodoKEM uses a large $n \times n$ matrix with coefficients in $\mathbb{Z}_q$ denoted as A and generated with the function Frodo.Gen as shown in line 3 and using SHAKE128. To perform the sampling, a random bit string of length 16-bit denoted by $r^{(\cdot)}$ is required for each coefficient of the matrix. The random bit strings are initially generated using either SHAKE128 (in FrodoKEM-640) or SHAKE256 (in FrodoKEM-976 and FrodoKEM-1344), using a fixed prefix and a newly generated random seed as input shown in line 4. The secret (S) and error (E) matrices are sampled from a Gaussian distribution as shown in lines 5 and 6 through the Frodo.Sample() function. It takes a seed that is expanded to uniformly distributed values and then redistributed to Gaussian samples that form the matrices [12]. Then, the generated matrices are processed according to line 7 to generate matrix B, which is processed using the function Frodo.Pack() (line 8). Finally, a SHAKE is required to generate pkh as shown in line 9. The public key pk is composed of the $seed_A$ and b, and the private (secret) key is derived from the public key, the hashed public key (*pkh*), a uniform vector (s), and the secret matrix (S^T).

Algorithm 2 describes the key encapsulation component of FrodoKEM. It shares some functions with Algorithm 1, such as the Gaussian sampling Frodo.Sample to create the matrices S', E', and E'' as seen in lines 4, 5, and 6. Frodo.Gen manages the generation of A according to line 7, and Frodo.Pack in lines 9 and 13 transforms a $n_1 \times n_2$ matrix C in $\mathbb{Z}_q$ into a byte array. The remaining functions are Frodo.Unpack in line 10 does the reverse process by transforming a byte array to an $n_1 \times n_2$ matrix C in $\mathbb{Z}_q$, and Frodo.Encode encodes bit strings of length $l = B \cdot \overline{n}^2$ as $\overline{n} \times \overline{n}$ matrices in $\mathbb{Z}_q$ as shown in line 12. The decapsulation algorithm is not shown because it shares almost the same fundamental operations as encapsulation and the only different function is Frodo.Decode which performs

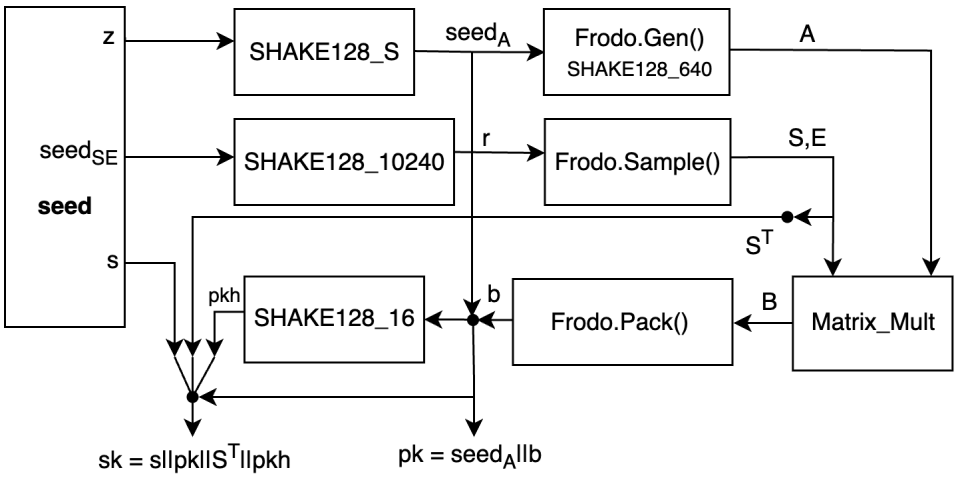

Fig. 2. Functional blocks of the FrodoKEM Key Generation.

the reverse operation of Frodo.Encode.

Algorithm 2 FrodoKEM.Encaps

Input: $(pk = seed_A||b)$, μ.
Output: (ct, ss).
1: $pkh \leftarrow SHAKE(pk, len_{sec})$
2: $seed_{SE}||k \leftarrow SHAKE(pkh||\mu||salt, len_{SE} + len_{sec})$
3: $r^{(0)}, r^{(1)}, ..., r^{(2\overline{n}n+\overline{n}\overline{n}-1)} \leftarrow SHAKE(0X96||seed_{SE}, 16(\overline{n}n + \overline{n}\overline{n})$
4: $S' \leftarrow Frodo.Sample(r^{(0)}, r^{(1)}, ..., r^{(n\overline{n}-1)}, \overline{n}, n)$
5: $E' \leftarrow Frodo.Sample(r^{(n\overline{n})}, r^{(n\overline{n}+1)}, ..., r^{(2n\overline{n}-1)}, n, \overline{n})$
6: $E'' \leftarrow Frodo.Sample(r^{(2\overline{n}n)}, r^{(2\overline{n}n+1)}, ..., r^{(2\overline{n}n+\overline{n}^2-1)}, \overline{n}, \overline{n})$
7: $A \leftarrow Frodo.Gen(seed_A)$
8: $B' \leftarrow S'A + E'$
9: $C_1 \leftarrow Frodo.Pack(B')$
10: $B \leftarrow Frodo.Unpack(b, n, \overline{n})$
11: $V \leftarrow S'B + E''$
12: $C \leftarrow V + Frodo.Encode(\mu)$
13: $C_2 \leftarrow Frodo.Pack(C)$
14: $ct = C_1||C_2||salt$
15: $ss \leftarrow SHAKE(ct||k, len_{sec})$
16: **return** ct, ss

IV. Methodology

A. Profiling

The first step of the proposed methodology is to carry out a software profiling of the C code implementation. This process analyzes the code to obtain the most time-consuming functions, including the number of calls and the percentage of the total execution time. With this tool, we can understand the behavior of the code, analyze its functions based on the time results, and focus on which ones could be accelerated in hardware.

B. Design of FrodoKEM Key Generation with HLS

The second step is to determine the suitability of the C code for HLS synthesis. It is important to mention, that in most cases some modifications are required. The most important functions are improved after evaluating the code with the profiling process. Fig. 2. shows the functional blocks of KeyGen, where all the buses have 16-bit.

Fig. 3 shows the functional blocks of the encapsulation component. The buses are 16-bit unless otherwise specified.

TABLE II
Definition of SHAKE128 Functions

NAME	INPUT (bytes)	OUTPUT (bytes)
SHAKE128_S	16	16
SHAKE128_10240	18	20480
SHAKE128_640	18	1280
SHAKE128_16	9616	32

We can see that key generation and encapsulation share some blocks. The decapsulation functional block is not included because it shares similar blocks to encapsulation.

We focus on the hardware implementation of the SHAKE128 instances (including the one used to generate the matrix A) and the matrix multiplication. SHAKE128/256 are extendable-output functions (XOFs) from the SHA-3 algorithm above mentioned. The XOFs functions are based on the Keccak algorithm, which is based on the Sponge construction. The Sponge Keccak algorithm is carried out as follows: initially, the input is padded, then absorbed and concatenated with zeros to increase its security, then the Keccak function makes some random permutations and rounds, and finally, the squeezing stage generates the output. In the next section, we will see that most of the design area is used by the SHAKE128 function, so it is important to improve it [18]. In Table II we show all the SHAKE128 functions for the KeyGen algorithm with its input and output sizes in bytes.

The E and S matrices are sampled from the distribution χ. The process starts with the expansion of the seed value coming from a SHAKE128_10240 to a set of uniformly distributed values. These values are then redistributed into Gaussian samples, which establish the matrices. The values of S and E matrices are stored as a vector in different RAMs.

The other time-consuming operation is the matrix multiplication, it has 640×640 elements or coefficients of 16-bit each. The rows of A are generated on the fly by SHAKE128_640 function and stored in a RAM, then they are multiplied by the columns of S, and the addition with E is performed sequentially in the same clock cycle and the result is stored

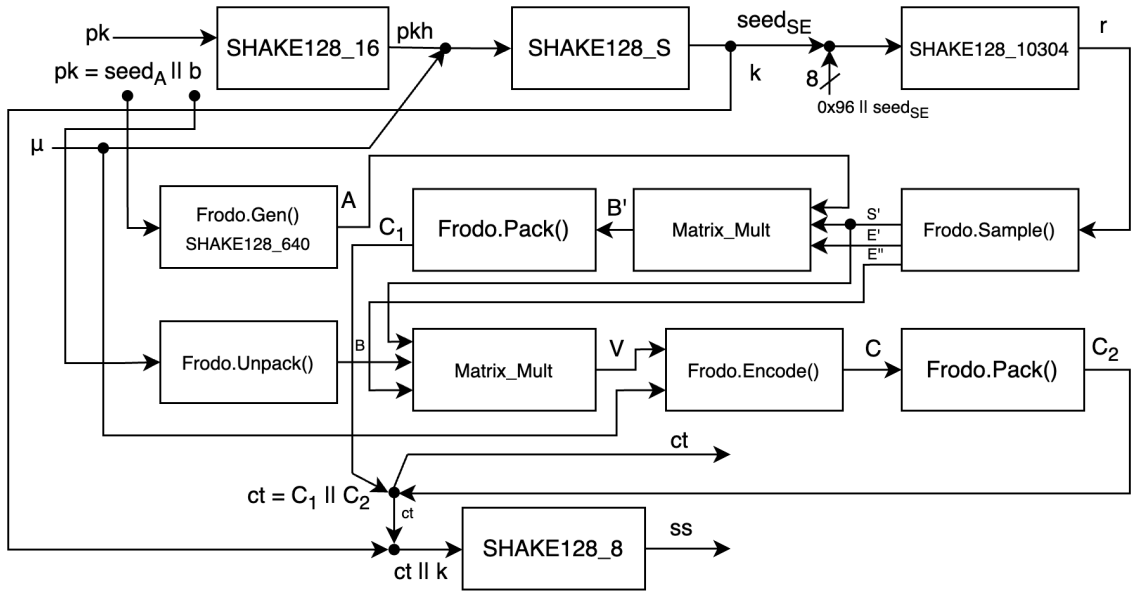

Fig. 3. Functional blocks of the FrodoKEM Encapsulation.

back.

The Frodo.Pack() function is employed to transform a matrix of dimensions $n_1 \times n_2$, represented by B, into a byte matrix of dimension $\mathbb{Z}_q$. This is achieved by reading the matrix row by row, from the least significant coefficient to the most one, and then concatenating the resulting values to produce a bit string.

Finally, the hardware was optimized using HLS-specific C directives. To improve the performance in the Matrix Multiplication block, we have used pragmas (directives) such as *array_partition* to split large arrays of 16-bit variables that can be accessed in parallel. Similarly, *unroll* allows some loop iterations to run in parallel as well and reduce iterations. For the other functions, we use *pipeline*, which facilitates the execution of operations within a loop, allowing the iterations to be performed simultaneously. Additionally, an interface with *ap_memory* is used to specify that a variable should be treated as a memory interface. This pragma allows control of the access and storage of the data, influencing the behavior of the synthesis tool for memory usage.

In order to verify the design, we adapted the Python script provided by the FrodoKEM team to generate the test vectors, stored them in a text file, and then imported them into the HLS C code for co-simulation purposes. We used AMD Xilinx Vivado HLS 2020.1 tools and an FPGA Artix-7 xc7a200t-ffg1156-3. The generated VHDL code was synthesized on the Artix-7 using Vivado 2020.1 and on the Stratix IV EP4SGX230KF40C2 using Intel Quartus Prime 21.1 Standard Edition, which was possible because no specific libraries or Megafunctions were used.

V. Results

A. Profiling Results

This section presents the profiling and synthesis results. To achieve the profile of FrodoKEM, it is essential to know from the software implementation, the time required to perform the functions KeyGen, encapsulation, and decapsulation. This time allows us to know the percentages of the execution times for each function.

TABLE III
Results of the Required Cycles for Each Algorithm from the FrodoKEM-640 Verification

Algorithm	Cycles
Key Generation	3028723
Encapsulation	3141530
Decapsulation	3114443

The profiling of the original benchmark code was obtained using Valgrind on a desktop computer with the following specifications: an AMD Ryzen 9 5900X processor, a clock speed of 3.7 GHz, 32 GB of RAM, and Ubuntu 20.04.

The reference C software implementation has a Known Answer Test (KAT) file including one hundred values with tuples containing the *seed*, *secret key* (sk), *public key* (pk), *ciphertext* (ct), and *shared secret* (ss) values. The KAT file and the FrodoKEM-640 scheme are simultaneously executed for one second to perform the verification, doing 1222 iterations for the key generation, 1178 for encapsulation, and 1188 for decapsulation. It also provides a mean of the required cycles for each algorithm as shown in Table III.

Profiling was performed by running the verification process but simultaneously with Valgrind. The following results were obtained: 27.80% of the execution time is for key generation, 28.53% for encapsulation, and 43.25% for decapsulation.

Fig. 4 shows the profiling results for key generation. As expected, matrix multiplication was the most time-consuming function with 80.50%. The next function was SHAKE128 including its variants with 7.94% followed by Frodo.Pack with 5.22%. For the Matrix_ Mult, SHAKE128 is the most time consuming with 92%.

TABLE IV
HARDWARE IMPLEMENTATION RESULTS FROM FRODOKEM-640 KEY GENERATION

Algorithm	LUTs	FFs	BRAM	DSP	Fmax (MHz)	Clock Cycles
Original	133564	87891	25.0	640	122[a]	7428950
Improved	35398	20648	10.5	1	135	2009322

[a]Frequency is estimated by Vivado HLS.

Fig. 5 shows the profiling results for encapsulation. As expected, matrix multiplication was the most time-consuming function with 74.00%. The next function was SHAKE128 including its variants with 9.9%, Frodo.Pack with 5.0%, and Frodo.Unpack with 4.8%. Analyzing the Matrix-Mult, we observed that the SHAKE128 function is the most expensive, with an execution time of 94%.

Fig. 6 shows the profiling results for decapsulation. As expected, matrix multiplication was the most time-consuming function with 75.50%. The next function was Frodo.Unpack with 9.8% followed by SHAKE128 including all its variants with 7.6%. For the Matrix_ Mult, SHAKE128 has an execution time of 93.80%.

B. Synthesis Results

Considering the above results, we carried out the HLS design of the three components, beginning with KeyGen and focusing on the SHAKE128, Matrix_Mult, and Frodo.Pack functions. Table IV shows the results from the reference C software implementation with only a few modifications. The area was 99% LUTs, 33% FFs, 86% DSPs, and 3.5% BRAM of the total available resources on the Artix-7. Moreover, the number of clock cycles was approximately twice as high as the software implementation. We modified some functions and found specific features of HLS that allow improvements to the code. The pragmas' implementation allowed us to achieve the best results.

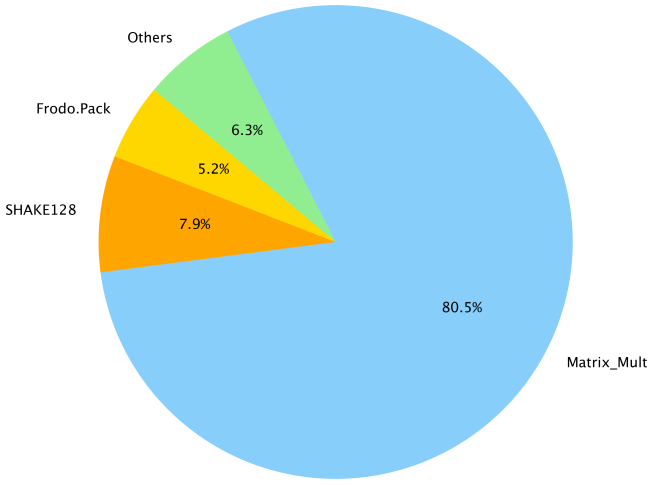

Fig. 4. Profiling results for FrodoKEM Key Generation.

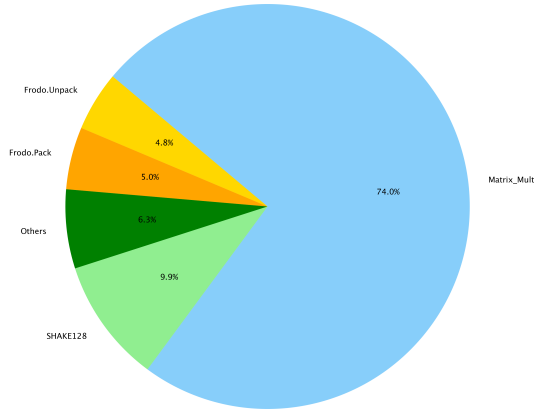

Fig. 5. Profiling results for FrodoKEM Encapsulation.

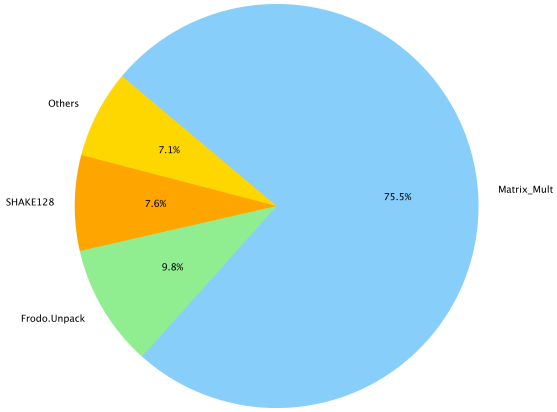

Fig. 6. Profiling results for FrodoKEM Decapsulation.

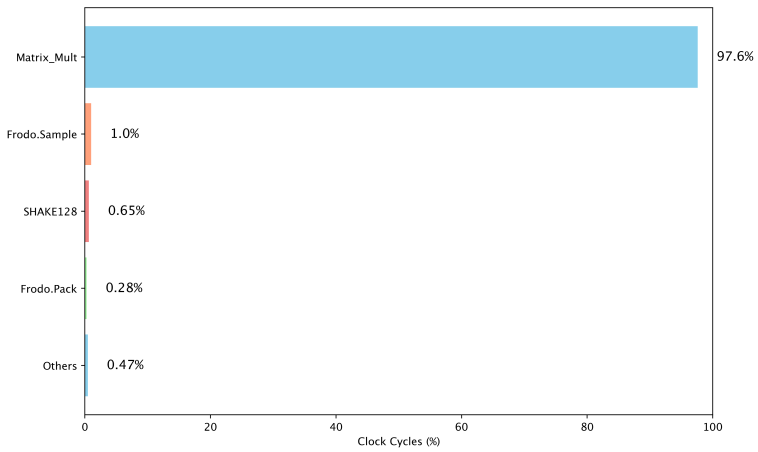

Fig. 7. Visualizing clock cycle distribution by block in FrodoKEM-640 KeyGen.

TABLE V
HARDWARE IMPLEMENTATION RESULTS FROM FRODOKEM-640

Ref.	Scheme	FPGA	Algorithm	LUTs	FFs	BRAM	DSP	Fmax (MHz)	Clock Cycles
This work	FrodoKEM	Artix-7	KeyGen	35398	20648	10.5	1	135	2009322
This work	FrodoKEM	Stratix IV	KeyGen	28838	20731	12.0	4	167	2009322
This work	FrodoKEM	Artix-7	Encaps	52068	31095	13.0	5	124	3739692
This work	FrodoKEM	Stratix IV	Encaps	36706	30820	18.0	8	164	3739692
This work	FrodoKEM	Artix-7	Decaps	43616	23081	18.5	5	122	3847377
This work	FrodoKEM	Stratix IV	Decaps	29216	22946	22.0	12	161	3847377
[11][a]	FrodoKEM	Virtex-7	Encaps	136998	44284	-	-	100	366609
[11][a]	FrodoKEM	Virtex-7	Decaps	82307	14461	-	-	100	220344
[11]	CRYSTALS-Kyber	Virtex-7	Encaps	1307815	11699	-	-	67	31669
[11]	CRYSTALS-Kyber	Virtex-7	Decaps	1977896	194126	-	-	67	43018
[19][b]	CRYSTALS-Kyber	ZYNQ-104	Key/Enc/Dec	31807	19820	32.0	290	208	132777
[11]	Classic McEliece	Virtex-7	Encaps	840430	60270	-	-	100	3787729
[11]	Classic McEliece	Virtex-7	Decaps	870908	79962	-	-	100	10659024
[20]	HQC	Artix-7	KeyGen	11484	8798	6.0	-	150	40247
[20]	HQC	Artix-7	Encaps	16487	13390	10.0	-	152	89110
[20]	HQC	Artix-7	Decaps	18739	15243	18.0	-	152	193082
[21]	BIKE	ZYNQ-7000	KeyGen	13567	11621	40.0	-	100	13784
[21]	BIKE	ZYNQ-7000	Encaps	23260	15571	96.0	-	100	633
[21]	BIKE	ZYNQ-7000	Decaps	37160	38118	96.0	35	100	13548

[a] Correspond to the version submitted to the first round of NIST.
[b] Clock cycles correspond to key generation. For encaps and decaps, they are 149777 and 57481, respectively

TABLE VI
ENERGY CONSUMED BY FRODOKEM-640

Algorithm	Platform	Latency (μs)	Power(mW)	Energy (mJ)
KeyGen	Artix-7	14883.9	3.63	54.0
	Stratix IV	12031.9	921.56	11088.1
Encaps	Artix-7	30158.8	4.51	135.8
	Stratix IV	23227.9	932.48	21659.6
Decaps	Artix-7	31535.9	3.94	124.2
	Stratix IV	23896.8	920.37	21993.9

Fig. 7 shows the clock cycle distribution by block in FrodoKEM-640 KeyGen. We can see that the matrix multiplication block used most of the clock cycles, followed by the Frodo.Sample and SHAKE128 blocks.

Matrix_Mult used 22.64% of the area, and SHAKE128_640 accounted for 95.50%. We can infer that the four SHAKE128 blocks utilized 89.10% of the total area. With this information, along with the clock cycle distribution, we can conclude that to speed up the algorithm, we must first enhance the SHAKE128 blocks and matrix multiplication. Table VI shows the energy consumption of each algorithm when executing one operation. This values are estimated by the time required to get the results (latency) and the power measurements given by Vivado and Quartus.

C. Comparison Results

To the best of our knowledge, this is the first work focused on an HLS hardware implementation of the eFrodoKEM-640 - SHAKE128 scheme. Although a direct comparison cannot be made due to the lack of similar HLS implementations of the same algorithms, we have chosen to present various works that involve hardware implementations of KEMs using HLS, which have been synthesized on different FPGAs. This includes CRYSTALS-Kyber and other schemes that have advanced to the fourth round of NIST.

Table V presents the synthesis results for the three components. The keyGen algorithm demonstrated a 34% reduction in clock cycles compared to the reference software implementation, achieving 67 operations per second on the Artix-7 and 83 operations per second on the Stratix-IV. A fair comparison with previous work is not feasible due to differences in the FPGA and the algorithms used, as mentioned earlier. Our results reflect more trade-offs than other schemes, as they require less RAM blocks and DSP resources. Compared to the HQC scheme, FrodoKEM requires more area and more clock cycles.

VI. CONCLUSION

In this paper, we present the hardware implementation of the FrodoKEM-640 scheme for PQC and its SHAKE128 variant using High-Level Synthesis. We improved the reference C software implementation and applied a set of directives that allowed optimizing the implementation, obtaining good synthesis results with speed improvements in the KeyGen component. The implementation results show that our design uses 26% of total area utilization on the Artix-7 for the key generation, 39% for the encapsulation, and 32% for the decapsulation. Furthermore, our results contribute to validating that FrodoKEM can be implemented in hardware. To achieve a better performance, it is necessary to implement the scheme using software/hardware co-design, optimizing the SHAKE and matrix multiplication blocks. Additionally, we consider that an RTL implementation of the scheme is required for future development. Furthermore, although our design has followed the standard's recommendations to ensure a protection against SCA, a detailed study of these and other attacks is essential.

One of our main concerns was that developing a pure RTL implementation of a PQC scheme could be time-consuming

and require a hardware development team. Once HLS optimization is complete, a full design can be achieved in a relatively short time. This proves to be an excellent design tool for hardware validation of PQC schemes in the current NIST call for additional digital signature proposals, where development time presents a challenge.

ACKNOWLEDGMENTS

F. A. Urbano-Molano would like to express his gratitude to the Ministry of Science, Technology, and Innovation, and SGR of Colombia for the "Becas de excelencia doctoral del Bicentenario" scholarship, as well as the AMD University program, the Intel FPGA Academic program, the *Universidad del Valle*, and Dr. Kris Gaj, co-director of the CERG (Cryptographic Engineering Research Group) at George Mason University, USA, for the training in software profiling.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999, doi: 10.1137/S0036144598347011, publisher: Society for Industrial and Applied Mathematics.

[2] *Post-quantum cryptography*. Springer, 2009, doi: 10.1007/978-3-540-88702-7.

[3] J. Howe, C. Moore, M. O'Neill, F. Regazzoni, T. Güneysu, and K. Beeden, "Lattice-based encryption over standard lattices in hardware," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. Association for Computing Machinery, 2016, doi: 10.1145/2897937.2898037, pp. 1–6.

[4] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard lattice-based key encapsulation on embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 372–393, 2018, doi: 10.13154/tches.v2018.i3.372-393.

[5] N. I. of Standards and Technology, "SHA-3 standard: Permutation-based hash and extendable-output functions," 2015.

[6] J. Howe, M. Martinoli, E. Oswald, and F. Regazzoni, "Exploring parallelism to improve the performance of FrodoKEM in hardware," *Journal of Cryptographic Engineering*, vol. 11, no. 4, pp. 317–327, 2021, doi: 10.1007/s13389-021-00258-7.

[7] J. W. Bos, M. Ofner, J. Renes, T. Schneider, and C. Van Vredendaal, *The Matrix Reloaded: Multiplication Strategies in FrodoKEM*. Springer International Publishing, 2021, doi: 10.1007/978-3-030-92548-2_5, vol. 13099, series Title: Lecture Notes in Computer Science.

[8] E. Karabulut, E. Alkim, and A. Aysu, "Efficient, flexible, and constant-time Gaussian sampling hardware for lattice cryptography," *IEEE Transactions on Computers*, pp. 1–1, 2021, doi: 10.1109/TC.2021.3107729.

[9] V. B. Dang, F. Farahmand, M. Andrzejczak, and K. Gaj, "Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, doi: 10.1109/ICFPT47387.2019.00032, pp. 206–214.

[10] V. L. R. D. Costa, J. López, and M. V. Ribeiro, "A System-on-a-Chip implementation of a post-quantum cryptography scheme for smart meter data communications," *Sensors*, vol. 22, no. 19, p. 7214, 2022, doi: 10.3390/s22197214.

[11] K. Basu, D. Soni, M. Nabeel, and R. Karri, "NIST post-quantum cryptography- A hardware evaluation study," 2019, publication info: Preprint. MINOR revision.

[12] P. Karl, T. Fritzmann, and G. Sigl, "Hardware accelerated FrodoKEM on RISC-V," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2022, doi: 10.1109/DDECS54261.2022.9770148, pp. 154–159.

[13] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, 2021, doi: 10.1109/TPDS.2020.3025691.

[14] H. Kwon, K. Jang, H. Kim, H. Kim, M. Sim, S. Eum, W.-K. Lee, and H. Seo, "ARMed Frodo: FrodoKEM on 64-bit ARMv8 processors," in *Information Security Applications*. Springer International Publishing, 2021, doi = 10.1007/978-3-030-89432-0_17, vol. 13009, pp. 206–217.

[15] D. L. G. Filho, G. Brandão, G. Adj, A. Alblooshi, I. A. Canales-Martínez, J. Chávez-Saab, and J. López, "PQC-AMX: Accelerating Saber and FrodoKEM on the Apple M1 and M3 SoCs," in *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*. IEEE, 2024, doi: 10.1109/ARITH61463.2024.00012, pp. 9–16.

[16] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th annual ACM Symposium on Theory and Computing - STOC'05*, 2005, doi: 10.1145/1568318.15683, pp. 84 – 93.

[17] F. Team, "FrodoKEM specification," 2023. [Online]. Available: www.frodokem.org

[18] F. A. Urbano-Molano and J. Velasco-Medina, "SHA-3 implementation for post-quantum cryptography using high-level synthesis," in *XI Southern Programmable Logic Conference*, 2023, pp. 9–12.

[19] C.-H. Lee, J.-H. Lee, H. Jung, H. Lee, and H. Lee, "HLS-based HW/SW co-design and hybrid HLS-RTL design for post-quantum cryptosystem," *Journal of Semiconductor Technology and Science*, vol. 24, no. 3, pp. 191–198, 2024, doi: 10.5573/JSTS.2024.24.3.191.

[20] C. Aguilar-Melchor, J.-C. Deneuville, A. Dion, J. Howe, R. Malmain, V. Migliore, M. Nawan, and K. Nawaz, "Towards automating cryptographic hardware implementations: A case study of HQC," in *Code-Based Cryptography*. Springer Nature Switzerland, 2023, doi: 10.1007/978-3-031-29689-5_4, vol. 13839, pp. 62–76, series Title: Lecture Notes in Computer Science.

[21] G. Montanaro, A. Galimberti, E. Colizzi, and D. Zoni, "Hardware-software co-design of BIKE with HLS-generated accelerators," in *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2022, doi: 10.1109/ICECS202256217.2022.9970992, pp. 1–4.

Fernando Aparicio Urbano-Molano (SM'20) received the BSc degree in engineering physics from the *Universidad del Cauca*, Popayan, Colombia in 2005, the MSc degree in engineering (electronics), and the PhD in electrical and electronics engineering from the *Universidad del Valle*, Cali, Colombia in 2012 and 2025, respectively. He was a Visiting Research Scholar at Cryptographic Engineering Research Group (CERG) from George Mason University under the supervision of Dr. Kris Gaj from January to August 2024. From November 2006 to April 2020, he was an associate professor with the Department of Telematics at the college of electronics and telecommunication engineering at *Universidad del Cauca*, Colombia. His research interest are cryptographic engineering, digital systems design, reconfigurable hardware, embedded systems, Internet of Things, and Data Analysis.

Jaime Velasco-Medina (SM'96) received the B.S. degree in electrical engineering from the Universidad del Valle, Cali, Colombia, in 1985, and the M.Sc. and Ph.D. degrees in microelectronics from the Institute National Polytechnic of Grenoble, Joseph Fourier University, Grenoble, France, in 1995 and 1999, respectively. In 1988, he joined the AT&T Bell Laboratory, Allentown, PA, USA, as a Technical Staff Member for six months. He was the pioneer of the current-based testing for analog and mixed signal circuits, and online testing of operational amplifiers. He is currently a Faculty Professor with the School of Electrical and Electronics Engineering, Universidad del Valle, and the Director of the Bionanoelectronics Research Group. He has authored or co-authored more than 50 IEEE papers and 50 peer-reviewed papers in other scientific events and journals. His current research interests include digital systems design for computer arithmetic and digital signal processing; test of analog and mixed-signal integrated circuits, hardware architectures for cryptography, quantum computing, wireless communications, citocomputation, and modeling of biological systems; the design of graphene-based digital circuits for spintronics; and computational design of bionanosensors and bionanomachines for drug delivery nanosystems. Dr. Velasco-Medina is a reviewer for the JETTA, IEEE-LATW, IEEE-SPL, IBERCHIP, IEEE-LASCAS, IEEE TRANSACTIONS ON VLSI SYSTEMS, IEEE TRANSACTIONS ON SIGNAL PROCESSING, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I, and many other international publications and conferences.