HORCRUX

A Lightweight PQC-RISC-V eXtension Architecture

Alessandra Dolmeta[1], Valeria Piscopo[1], Guido Masera[1], Maurizio Martina[1] and Michael Hutter[2,3]

[1] Politecnico di Torino, Italy name.surname@polito.it
[2] University of the Bundeswehr Munich, Germany michael.hutter@unibw.de
[3] PQShield, Austria michael.hutter@pqshield.com

Abstract. This work presents a RISC-V extension for Post-Quantum Cryptography (PQC) called HORCRUX, which provides a unified Instruction-Set Extension (ISE) supporting all NIST-approved PQC algorithms. HORCRUX addresses the current fragmentation in hardware support, where existing extensions typically focus on individual algorithms or limited subsets of PQC schemes, and targets the common kernels shared across ML-KEM, ML-DSA, SLH-DSA, and HQC. To address the primary computational bottlenecks of all these algorithms (namely, modular arithmetic, matrix multiplication, and hash transformations), the extension introduces new RISC-V instructions executed by a tightly coupled coprocessor. This coprocessor requires only minimal hardware resources, making the architecture efficient for constrained devices while still providing substantial acceleration. An experimental evaluation on a Zynq UltraScale+ FPGA demonstrates speedups of up to 3× for lattice and hash-based schemes, and over 30× for code-based schemes, while adding less than 2,900 LUTs and 400 FFs to the design. The extension's modular structure maintains backward compatibility with standard RISC-V cores, offering a scalable solution for deploying PQC on highly constrained embedded systems.

Keywords: Post-Quantum Cryptography · RISC-V · Instruction Set Extension · Hardware Security · ML-KEM · ML-DSA · HQC · SLH-DSA

1 Introduction

The advent of large-scale quantum computers poses a fundamental threat to the security of widely deployed public-key primitives such as RSA and ECC, which are vulnerable to Shor's algorithm. In response, the National Institute of Standards and Technology (NIST) initiated a multi-year standardization effort to identify quantum-resistant alternatives [MPST+16]. The resulting Post-Quantum Cryptographic (PQC) schemes address authenticity and confidentiality through digital signatures and encryption primitives, respectively. For encryption, the community has adopted Key Encapsulation Mechanisms (KEMs) as efficient alternatives to traditional Public-Key Encryption (PKE) and key exchange protocols. Following years of evaluation, NIST selected a first set of PQC standards in 2024 [MPST+16]: ML-KEM (FIPS 203) [Nat24b] for public-key encryption and key establishment, ML-DSA (FIPS 204) [Nat24a] and SLH-DSA (FIPS 205) [Nat24c] for digital signatures (lattice-based and hash-based, respectively). More recently, HQC (Hamming Quasi-Cyclic), a compact and efficient code-based KEM, was also selected for standardization [Nat25b], reinforcing the evolving PQC landscape.

In practical deployments, particularly on resource-constrained embedded systems, many PQC algorithms introduce new performance challenges not typically encountered with classical primitives. Software-only implementations of lattice-, code-, and hash-based schemes often exhibit higher computational latency and increased memory or code size compared to traditional algorithms, like AES or RSA. These overheads can make certain PQC schemes less suitable for low-end platforms, such as tiny microprocessors, without dedicated hardware acceleration or careful optimization. To overcome these barriers, dedicated hardware support is crucial for meeting the throughput, latency, and memory constraints typically required in embedded applications.

In this paper, we present HORCRUX — a RISC-V Instruction Set Extension (ISE) for PQC. Leveraging RISC-V's modular ISA, HORCRUX invokes a coprocessor through custom instructions that work with unmodified toolchains. Using the Core-V-eXtension InterFace (`CV-X-IF`) [Ope24a], the coprocessor attaches with minimal microarchitectural changes, avoiding both the complexity of standalone accelerators and the toolchain modifications typical of classical ISA extensions. We detail the design and integration, and show that a small set of opcodes delivers practical PQC acceleration without disrupting existing software flows.

Contributions. Our primary contributions are as follows:

- **First PQC-RISC-V ISE:** We introduce, to our knowledge, the first RISC-V extension designed *across* PQC families—not a single scheme—capturing the shared arithmetic and bit-level kernels spanning lattice-, hash-, and code-based algorithms.
- **Broad-spectrum PQC coverage:** HORCRUX natively supports two key encapsulation mechanisms (ML-KEM and HQC) and two digital signature schemes (ML-DSA and SLH-DSA). This broad coverage makes it particularly well-suited for applications that require fallback solutions (crypto agility), ensuring robustness and continuity amid the evolving PQC standardization landscape.
- **Seamless integration and prototype:** HORCRUX is integrated into a standard RISC-V pipeline with minimal impact on legacy control and datapaths.
- **Fine-grained operation profiling:** We provide cycle-accurate profiling of PQC workloads across key stages and sub-modules, based on NIST's Known Answer Tests (KATs). These standardized test vectors ensure correctness and reproducibility, making them a reliable foundation for benchmarking and comparing hardware implementations.
- **Comprehensive performance evaluation:** We demonstrate up to 30$\times$ reductions in key-generation, encapsulation, and signature latencies, as well as notable decreases in code size, while maintaining a minimal hardware footprint of under 2,900 LUTs and 400 FFs evaluated on a Zynq UltraScale+ FPGA.

To align with NIST's PQC evaluation framework, which mandates characterization on 8-, 16-, and 32-bit microcontrollers, we focus our design and experiments on a 32-bit core to balance resource constraints, compatibility, and efficient arithmetic execution [NIS25]. We do not incorporate masking or active fault-injection defenses, consistent with the protection profile defined for the RISC-V `Zk` extensions [RIS22].

Organization. The rest of the paper is organized as follows: Section 2 introduces the PQC algorithms targeted in this work and reviews ISEs along with related literature. Section 3 presents our custom instructions, and Section 4 details their hardware implementation. Section 5 reports code size, performance, and area results. Finally, Section 6 and Section 7 discuss future directions and conclude the paper.

Source code. The source code for all our implementations is available open-source.[1]

[1]GitHub Repository: `https://github.com/vlsi-lab/HORCRUX`

2 Background

This section provides the foundational context for our work. First, Subsection 2.1 describes the standardized PQC algorithms we aim to accelerate, highlighting their mathematical structures and key computational bottlenecks. Then, Subsection 2.2 introduces ISEs as a mechanism for domain-specific acceleration and an overview of related literature.

2.1 PQC Algorithms

This work considers PQC algorithms from three main algebraic families: lattice-based, hash-based, and code-based. Each offers different trade-offs in performance, complexity, and security against quantum adversaries.

In the lattice-based category, **FIPS 203** [Nat24b] defines the ML-KEM (Module-Lattice-Based Key-Encapsulation Mechanism), based on CRYSTALS-Kyber, which offers IND-CCA security and relies on the hardness of Learning with Errors (LWE) over polynomial rings. Similarly, **FIPS 204** [Nat24a] specifies the ML-DSA (Module-Lattice-Based Digital Signature Algorithm), built on CRYSTALS-Dilithium, a digital signature scheme with EU-CMA security. Both ML-KEM and ML-DSA depend on polynomial arithmetic, including multiplications of large polynomials and small-error polynomial sampling.

In the hash-based category, **FIPS 205** [Nat24c] defines the SLH-DSA (Stateless Hash-Based Digital Signature Algorithm), instantiated as SPHINCS+. This scheme is constructed from hypertrees of one-time Winternitz signatures and Merkle authentication paths, with security grounded in the collision resistance and pseudorandomness of cryptographic hash functions such as SHA-256, SHAKE-256, and Haraka.

In the code-based category, HQC [Nat25b] is a leading candidate selected by NIST for standardization, though not yet assigned an official FIPS number at the time of writing. HQC is based on the hardness of syndrome decoding in Moderate-Density Parity-Check (MDPC) codes. Notably, HQC was chosen over McEliece-style schemes, which, despite their strong security foundations, suffer from large public key sizes.

A general overview of each algorithm follows below. For additional details, readers are referred to the original NIST submissions and corresponding FIPS drafts. The key arithmetic kernels that dominate performance are analyzed in Section 3, where we present our RISC-V-based extensions designed to accelerate these critical operations.

ML-KEM. ML-KEM is a lattice-based key encapsulation mechanism (KEM) built upon the Module-Learning With Errors (Module-LWE) problem over the ring $\mathbb{Z}_q[x]/(x^n+1)$ with $n = 256$ and (12-bit) modulus $q = 3329$. It offers three security levels, which correspond approximately to NIST levels 1, 3, and 5, by adjusting the module rank and error distribution parameters to achieve quantum-resistant hardness. Polynomial multiplications and convolutions in dimension $n = 256$ are accelerated via the Number Theoretic Transform (NTT) and its inverse (INTT), which constitute one of the primary computational hotspots. Noise sampling uses the center-binomial distribution (CBD) to produce small error polynomials with the precise statistical properties required for Module-LWE security. All pseudorandomness, including matrix and error generation, is derived from Keccak-based extendable-output functions (e.g., SHAKE128/256), representing another key performance bottleneck. Through this combination of compact key/ciphertext sizes, rigorous security proofs, and optimized arithmetic, ML-KEM achieves a practical balance for post-quantum-secure KEM deployment.

ML-DSA. ML-DSA is a lattice-based digital signature scheme grounded in the hardness of the Module-SIS and Module-LWE problems over the same ring structure as ML-KEM but with different (23-bit) modulus $q = 8380417$. It offers three security tiers—corresponding to NIST levels 2, 3, and 5 by tuning the module rank, norm bounds, and noise parameters. Signatures are generated via a Fiat–Shamir transform applied to an identification protocol:

secret key vectors are sampled from a centered-binomial distribution (CBD), and a sparse challenge polynomial is derived from a Keccak-based hash (SHAKE256) of the message and commitment. As in ML-KEM, polynomial multiplications and inner products in ML-DSA are accelerated using the NTT and its inverse, which constitute two of the main computational bottlenecks. Rejection sampling, required to enforce bound constraints on the signature vector norm, introduces variable timing and extra hashing rounds, making it another key performance bottleneck.

HQC. HQC is a code-based KEM whose security relies on the difficulty of decoding random quasi-cyclic codes in the Hamming metric over $\mathbb{F}_2[x]/(x^n + 1)$ with $n \approx 17689$. It defines three security levels—targeting NIST levels 1, 3, and 5—by varying the code length, error weight, and repetition parameters. Core encryption and syndrome computation involve polynomial multiplications in dimension n over $\mathbb{F}_2$, which has been shown to dominate the runtime profile of HQC. All random seed expansion and error vector sampling are driven by a Keccak-based XOF, representing a secondary performance bottleneck. Decryption requires concatenated decoding of an outer Reed–Solomon code (via Berlekamp–Massey) and an inner Reed–Muller code (via majority logic), each introducing nontrivial computational overhead. A revised HQC specification was released on August 22, 2025, introducing editorial clarifications and several technical adjustments to KEM construction and key formats [HQC25].

SLH-DSA. SLH-DSA is a stateless hash-based signature scheme built from a D-layer hypertree of few-time FORS trees and Winternitz One-Time Signature (WOTS+) instances, operating over a chosen hash function H (e.g., SHA-256, SHAKE-256, or Haraka). It provides three security levels—128, 192, and 256 bits—by adjusting the hypertree height D, the FORS tree parameters (k, a), and the WOTS+ Winternitz parameter w to balance signature size and security. Each signature entails computing $D \cdot h$ Merkle authentication paths and k small FORS trees, as well as n-length WOTS+ chain evaluations per leaf, all of which rely on repeated evaluations of H. The dominant performance bottleneck is thus the sheer volume of hash function calls—across WOTS+ chains, FORS tree hashing, Merkle path computations, and the PRF-based seed expansion—making H the primary cost driver. Parallelism and vectorized implementations of H can mitigate but not eliminate this overhead, which remains the key challenge in achieving high-speed SLH-DSA signatures. In the parameter names, the suffix "**f**" denotes the *fast* profile, while "**s**" targets smaller sizes at reduced speed, and "**r**" *robust* variants.

2.2 PQC Instruction Set Extensions and Related Work

Instruction-Set Extensions augment a general-purpose base ISA with domain-specific functionality, offering a middle ground between software libraries and full hardware accelerators. By embedding carefully selected instructions, ISEs accelerate critical algorithms with minimal area and power overhead. In domains like PQC, this approach significantly reduces code size and execution time compared to software-only solutions, while avoiding the complexity and inflexibility of custom accelerators. A well-designed ISE offers strong performance gains with lightweight integration. In recent years, numerous hardware accelerators and ISEs have been proposed to enhance the efficiency of PQC primitives, as surveyed in [OBS23]. [AEL+20] proposes ISA extensions targeting finite field arithmetic to accelerate Kyber and NewHope on RISC-V, focusing on efficient modular multiplication and reduction operations using the VexRiscv core. [BUC19] introduces Sapphire, a configurable crypto-processor for lattice-based post-quantum protocols, but combining programmable instruction extensions with loosely coupled hardware accelerators. [DDMV+25] implements a coprocessor exploiting the `CV-X-IF`, accelerating HQC bottlenecks with a series of tightly-coupled accelerators. [FSS20] proposes RISQ-V, a tightly coupled RISC-V co-processor with dedicated accelerators for Kyber and New-Hope. [GLM24] proposes hardware acceleration of Kyber symmetric primitives on a 32-bit RISC-V core using official (*Zkne*, *Zknh*) and custom-ISEs (*Xkyber*). [JDH+25] presents one of the first highly optimized assembly software implementations to deploy Kyber and Dilithium on 64-bit RISC-V ISA, but purely software. [KA20] provides an efficient and flexible NTT solution through application-specific dynamic instruction scheduling, memory dependence prediction, and datapath optimizations. [KGHRM23] implements an instruction set capable of accelerating any algorithm based on finite field arithmetic $\mathbb{F}_{2^m}$. [LQYW24] implements a compact ISE for Kyber, implementing k^2-reduction into an instruction for the butterfly transformation, and auxiliary instructions to facilitate the rearranging of NTT's coefficients. [LVC24] focuses on SPHINCS+: a tightly coupled round-based design is used to accelerate the Keccak and Haraka hash functions. [MBB+23] implements a lightweight custom Arithmetic Logic Unit (ALU), used in parallel to the regular ALU, to accelerate the NTT computations for both Kyber and Dilithium, extending the RISC-V ISA by five new custom instructions for each algorithm. [NDMZ+21] proposes a dedicated PQC ALU, embedded directly in the pipeline of a RISC-V processor, to accelerate NTT and its inverse for both Kyber and Dilithium. [Saa24] presents SLotH, accelerating all SLH-DSA parameters and features with a SHAKE and a SHA-2 unit. [SOSK23] proposes an ISE to accelerate the polynomial arithmetic and sampling of the OpenTitan Big Number Accelerator. The hardware is then tested on the Dilithium verify operation. [WZZ+24] proposes an optimized hardware-software co-design for Kyber and Dilithium, accelerating specific parts with a customized accelerator, utilizing assembly instructions for others, and leveraging a multi-core acceleration scheme for certain tasks. In [YLW+25], a fine-grained, tightly coupled component accelerates the Keccak operation of SHA-3 to optimize SLH-DSA. A customized SIMD ALU, along with a SIMD register file, is used to parallelize the execution of Keccak steps, avoiding bottlenecks driven by data dependencies. [YSZ+24] presents a hardware processor for lattice-based PQC, implementing key primitives to enable practical deployment on resource-constrained devices. [ZZYH24] implements a coprocessor that includes a SHA-3 accelerator, adding 7 custom instructions.

The study most closely aligned with our work is the coprocessor design in [LKKK22], which introduces RISC-V ISA extensions for NIST PQC standards via the CV-X-IF interface. Although no performance figures are reported in that paper, the architecture is further detailed and evaluated in [LKK23]. Their RISC-V extension covers Keccak-*f* permutations, NTT with Montgomery-based modulo reductions, binomial and rejection sampling routines, finite-field and conditional arithmetic operations, and direct access to dedicated coprocessor registers, supporting both the finalized post-quantum algorithms and fourth-round PQC candidates. However, their coprocessor only reports metrics for individual submodules—no end-to-end algorithm results, and still lacks integrated benchmarks across all supported schemes.

Table 1 summarizes the main contributions of the most relevant related works. For each paper, it is indicated which cryptographic primitives or components are targeted by the proposed hardware accelerators, grouped by scheme. The classification includes four categories: symmetric primitives such as (Keccak)-based hashing (Sym.Prim.), polynomial operations such as NTT/INTT multiplications or modular reductions (Poly), sampling mechanisms such as CBD or rejection sampling (Sampler), and data compression routines for reducing the size of ciphertexts and public keys (Compress). This organization enables a direct comparison of the covered functional blocks and highlights the completeness of our proposed architecture across ML-KEM, ML-DSA, SLH-DSA, and HQC.

Additional related works include designs targeting different primitives or HW/SW codesign approaches [AMI+23, ALCZ20, BDC20, FSMG+19, MSS24, RLC+25, XHY+20]. Lightweight cryptography accelerators are discussed in [CGM+22, TGMD20], masked HW/SW implementations in [FBR+21], and cryptographic ISA extensions with side-channel protections in [MNP+20, MP21]. Moreover, a recent article outlines lowRISC's

Table 1: Comparison of related works.

	ML-KEM				ML-DSA			SLH-DSA	HQC	
Work	Sym. Prim.	Poly	Sampler	Compress	Sym. Prim.	Poly	Sampler	Sym. Prim.	Sym. Prim.	Poly
[AEL+20]		X								
[DDMV+25]									X	X
[FSS20]	X	X								
[GLM24]	X	X	X	X						
[KA20]		X				X				
[LKK23]	X	X	X		X	X	X		X	
[LQYW24]		X								
[LVC24]								X		
[MBB+23]		X				X				
[NDMZ+21]		X				X				
[Saa24]								X		
[SOSK23]	X	X			X	X	X			
[YLW+25]								X		
[YSZ+24]	X	X	X		X	X				
[ZZYH24]								X		
This work	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

efforts to support PQC in open-source hardware [low24], highlighting collaborative research involving OpenTitan and OTBN.

To the best of our knowledge, our work is the first to propose a RISC-V compliant ISE that simultaneously supports all four NIST-selected post-quantum finalists (ML-KEM, ML-DSA, SLH-DSA, and HQC) within a unified architectural framework. This holistic approach enables broad algorithmic coverage and fosters extensibility for secure, future-proof cryptographic implementations.

3 Design Methodology

In this section, we detail the custom instructions introduced in our ISE, examining both their functional role within the target PQC algorithms and the specific integration strategies adopted for each. Before delving into the implementation details of each of them, it is important to outline the guiding principles behind our design.

As a preliminary step, we conducted a comprehensive profiling of the target PQC schemes on an FPGA platform. This analysis provides fine-grained insight into performance-critical bottlenecks across various schemes [2]. By identifying recurring operations that dominate execution time or exhibit poor hardware efficiency, we derived the quantitative foundation for selecting candidates for ISE. However, unlike many of the works discussed in Subsection 2.2, our primary objective is not to achieve the highest possible speedup through full hardware acceleration. Had that been our goal, we would have opted for loosely coupled accelerators or dedicated coprocessors capable of offloading entire cryptographic subroutines. Our aim is more foundational: we seek to identify a minimal yet effective set of instructions that, much like the standard RISC-V cryptographic extension for classical algorithms [RIS22], can serve as the basis for a general-purpose ISE targeting PQC.
To this end, our methodology is guided by three key constraints:

Constraint 1: Instructions must be RISC-V compliant, adhering to standard encoding formats with two (`R-type`) or three (`R4-type`) source and a single destination register.

Constraint 2: The hardware design must limit area overhead, favoring resource sharing and streamlined datapaths to reduce silicon footprint.

Constraint 3: The extension must be compatible with RISC-V context-switching. Temporary internal state is allowed if confined to instruction execution or explicit instruction sequences, with deterministic behavior on re-execution.

[2] Tables 10 - 13.

This principled approach allows us to design an extensible architectural foundation capable of supporting a wide range of PQC algorithms without sacrificing efficiency, especially with a really low area overhead. HORCRUX is a small-area design intended for integration with embedded devices, rather than as a SIMD/vector-capable main application core, further reinforcing its lightweight and efficient nature. This work represents a prototype implementation, with the entire design specifically tailored to the four PQC algorithms discussed earlier. The custom instructions have been directly integrated into the corresponding software implementations, as our primary focus is on demonstrating hardware-side advantages. Since the design targets a 32-bit SoC, all source and destination registers are 32-bit wide. When more than 32 bits are required, multiple source registers are used; when fewer bits are needed, inputs remain 32-bit, but the datapath operates only on the significant bits. This avoids redundant hardware and maintains efficiency. The same principle applies to outputs: if more than 32 bits are needed, the result is produced over multiple instructions, which are grouped in a way that consistently satisfies Constraint 3. Moreover, the proposed ISE is inherently extensible and can be adapted to support additional functions or algorithms with minimal modifications, offering a foundation toward crypto-agile architectures, as discussed in Section 6. Once the approach is established, the instructions can be progressively generalized, trading off a small area increase to achieve greater versatility and broader software compatibility.

The resulting instruction set is organized into five functional groups: (i) Keccak ISE, (ii) polynomial coefficient compression ISE, (iii) modular reduction ISE, (iv) sampling ISE, and (v) Galois Field arithmetic ISE. Each group targets performance-critical operations identified during profiling, providing hardware support across diverse PQC algorithms.

Keccak ISE. Keccak is a family of sponge-based cryptographic primitives that serves as the foundation for multiple standardized hash and eXtendable-Output Functions (XOFs), such as SHA-3 and SHAKE [BDP+15]. Its flexibility and strong security properties make it a core building block in several PQC schemes, where it is employed for hashing, pseudo-random number generation, and key derivation. These include ML-KEM, ML-DSA, SLH-DSA, and HQC, each using various Keccak-based functions (e.g., SHA3-256, SHAKE-128/256) depending on the specific role within the algorithm, such as Cryptographically Secure Pseudorandom Number Generator (CSPRNG), key derivation, or Merkle tree construction. These Keccak-based functions constitute the primary bottleneck of ML-KEM, ML-DSA, and SLH-DSA. All these different functions are implemented as a mode of the same Keccak function, Keccak-f[1600] in Algorithm 1, so the same instance can be used for all. To facilitate hardware/software co-design and compatibility, the Keccak software implementation has been refactored to operate on 32-bit platforms.

Algorithm 1: Keccak-f[1600]: State permutation.

```
  Input: $A \in \mathbb{Z}_{2^{64}}^{25}$, RC Round Constants, Output: $A \in \mathbb{Z}_{2^{64}}^{25}$
1 for $rnd \leftarrow 0$ to 23 do
2     for $x \leftarrow 0$ to 4 do
3         $C[x] \leftarrow A[x] \oplus A[x+5] \oplus A[x+10] \oplus A[x+15] \oplus A[x+20]$
4     for $x \leftarrow 0$ to 4 do
5         $D[x] \leftarrow C[(x+4) \bmod 5] \oplus \texttt{ROL}(C[(x+1) \bmod 5], 1)$
6     for $i \leftarrow 0$ to 24 do
7         $A[i] \leftarrow A[i] \oplus D[i \bmod 5]$
8     $A[0] \leftarrow A[0] \oplus \mathrm{RC}[\mathrm{rnd}]$
```

To accelerate the Keccak permutation, we first target the ROL operation—a bitwise left rotation used extensively in the θ, ρ, and π steps. The pseudocode in Algorithm 1 illustrates the round logic, where ROL is used to rotate intermediate values. We implement a dedicated hardware block that takes a 64-bit word split into two 32-bit inputs, performs a left rotation by a variable offset (provided through a third source register), and outputs the rotated result as two 32-bit words. Hence, two custom instructions are introduced: `rol32_h` and `rol32_l`. In addition to the rotation, we also introduce the `bcop32` custom instruction to accelerate the χ step of the KECCAK-$f[1600]$ permutation, which is responsible for introducing non-linearity. This instruction replaces the common bitwise operation of the form $A = B \oplus (\neg C \wedge D)$. It is a single clock cycle using an `R4`-type format (three sources, one destination). This reduces both instruction count and critical path in the heavily repeated χ transformation. While it would be possible to accelerate the entire KECCAK-$f[1600]$ permutation by operating on the full state concurrently, such an approach would violate our design constraints. Specifically, it would not be RISC-V compliant (Constraint 1), as it would require non-standard wide-state operations; it would incur significant area overhead (Constraint 2) due to the complexity of the datapath and the huge dimension of the entire KECCAK state; and it would necessitate a dedicated architectural state (Constraint 3), such as a custom register file to hold the 1600-bit state. Instead, our fine-grained instruction-level acceleration preserves compliance, efficiency, and integration simplicity, satisfying all three constraints mentioned.

Polynomial Coefficient Compression ISE. In ML-KEM, polynomial coefficients belong to the ring $\mathbb{Z}_q$ with $q = 3329$. To reduce the size of ciphertexts and public keys, these coefficients are compressed to fewer bits. Compression is a lossy quantization that introduces limited noise but is designed to preserve decryption correctness. The mapping converts each input coefficient $x \in \mathbb{Z}_q$ to a d-bit integer as follows:

$$\text{Compress}_d(x) = \left\lfloor \frac{2^d}{q} \cdot x \right\rceil \bmod 2^d$$

where d is the target bit-width, depending on the specific context in ML-KEM (e.g., $d \in \{4, 5, 10, 11\}$). A direct implementation of this formula would require high-precision division, which is expensive in hardware. Instead, the expression is approximated using fixed-point arithmetic and precomputed constants:

$$\text{Compress}_d(x) \approx \left(\left(x \cdot 2^d + \left\lfloor \frac{q}{2} \right\rfloor\right) \cdot C_d\right) \gg S_d \bmod 2^d$$

where $C_d = \left\lfloor \frac{2^{S_d}}{q} \right\rfloor$ is a fixed-point constant encoding the reciprocal of q scaled by 2^{S_d}. The addition of $q/2$ enables rounding, while the product with C_d followed by a right shift of S_d approximates division by q. The result is then masked to retain the d least significant bits. For efficiency, the multiplication by C_d is realized through shift-and-add operations, avoiding general-purpose multipliers. Four dedicated instructions (`compress1-4`) are provided for $d \in 4, 5, 10, 11$, each hard-wired with its constants and bit width to minimize area and latency while preserving compliance with Constraints 1–3. The overall procedure is summarized in Algorithm 2.

Algorithm 2: HORCRUX'S GENERIC COMPRESS

```
Input: $x \in \mathbb{Z}_q$, Output: compressed coefficient $z$
1 $x \leftarrow x + ((x \gg 15) \wedge q)$
2 $y \leftarrow (x \ll d) + \lfloor q/2 \rfloor$
3 $p \leftarrow y \cdot C_d$
4 $z \leftarrow (p \gg S_d) \bmod 2^d$
```

Modular Reduction ISE. Modular arithmetic over $\mathbb{Z}_q$ is essential in lattice-based schemes, which rely on efficient modular reduction to support frequent multiplications by the public modulus Q. While addition and subtraction can use simple conditional subtraction, modular multiplication demands fast, division-free reductions into $[0, Q)$ or $[-Q/2, Q/2)$. Two widely used techniques are Montgomery [Mon85] and Barrett [Bar86] reduction. The reported contributions of reduction, NTT, and INTT operations, demonstrate the importance of introducing a dedicated custom ISE in ML-KEM and ML-DSA schemes. While Montgomery and Barrett reductions support arbitrary primes, they rely on multipliers, which are hardware-intensive. To improve efficiency in a unified hardware design for ML-KEM and ML-DSA schemes, we adopt special-prime reduction techniques. Inspired by the work of Solinas [Sol99], we design reduction algorithms tailored to primes with special structure, enabling reductions using only bit shifts and additions. Algorithm 3, Algorithm 4, Algorithm 5, and Algorithm 6 show the proposed modular-reduction algorithms implemented as HORCRUX ISEs.

Algorithm 3: HORCRUX's KEM-fast-reduce

```
Input: $x \in \mathbb{Z}$ (16-bit), Output: $z \in \left[-\frac{Q-1}{2}, \frac{Q-1}{2}\right]$ (16-bit)
1 $t_1 \leftarrow x$, $t_2 \leftarrow x \ll 1$, $t_3 \leftarrow x \ll 2$, $t_4 \leftarrow x \ll 3$, $t_5 \leftarrow x \ll 5$, $t_6 \leftarrow x \ll 7$, $t_7 \leftarrow x \ll 8$,
    $t_8 \leftarrow x \ll 9$, $t_9 \leftarrow x \ll 12$
2 $t \leftarrow (\sum_{i=1}^{9} t_i) \gg 24$
3 $m \leftarrow (t \ll 11) + (t \ll 10) + (t \ll 8) + t$
4 $z \leftarrow (x - m < Q)\ ?\ x - m : x - m - Q$
```

Algorithm 4: HORCRUX's KEM-reduce16

```
Input: $x \in \mathbb{Z}$ (32-bit), with $x \in [-Q \cdot 2^{15},\ Q \cdot 2^{15} - 1]$, Output: $r \in (-Q,\ Q)$ (16-bit)
1 $t_1 \leftarrow x$, $t_2 \leftarrow x \ll 8$, $t_3 \leftarrow x \ll 9$, $t_4 \leftarrow x \ll 12$, $t_5 \leftarrow x \ll 13$, $t_6 \leftarrow x \ll 14$, $t_7 \leftarrow x \ll 15$
2 $t \leftarrow \sum_{i=1}^{7} t_i$
3 $m \leftarrow (t \ll 11) + (t \ll 10) + (t \ll 8) + t$
4 $r \leftarrow (x - m) \gg 16$
```

Algorithm 5: HORCRUX's DSA-fast-reduction

```
Input: $x \in \mathbb{Z}$ (32-bit), with $x \leq 2^{31} - 2^{22} - 1$, Output: $z \in \left[-\frac{Q}{2}, \frac{Q}{2}\right]$ (32-bit)
1 $hi \leftarrow x \gg 23$, $lo \leftarrow (x \gg 22) \wedge 1$
2 $t \leftarrow hi + lo$
3 $m \leftarrow (t \ll 23) - (t \ll 13) + t$
4 $z \leftarrow x - m$
5 if $z \geq Q$ then
6     $z \leftarrow z - Q$
```

Algorithm 6: HORCRUX DSA-reduce32

```
Input: $a \in \mathbb{Z}$ (64-bit), with $a \in [-Q \cdot 2^{31},\ Q \cdot 2^{31}]$, Output: $r \in (-Q,\ Q)$ (32-bit)
1 $lo \leftarrow x \wedge (2^{32} - 1)$
2 $t_1 \leftarrow lo$, $t_2 \leftarrow lo \ll 13$, $t_3 \leftarrow lo \ll 23$, $t_4 \leftarrow lo \ll 24$, $t_5 \leftarrow lo \ll 25$
3 $t \leftarrow \sum_{i=1}^{5} t_i$
4 $m \leftarrow (t \ll 23) - (t \ll 13) + t$
5 $r \leftarrow (x - m) \gg 32$
```

All routines share one shifter–adder network, using only fixed shift–add patterns to minimize the critical path and avoid generic multipliers. Constants like Q and Q_{inv} are expressed as sums or differences of powers of two. This enables cheap, multiplier-free reductions: the quotient is estimated using a right shift, the product is reconstructed via shifts and additions, and then subtracted from the input. The shifter–adder network is made possible by the pseudo-Mersenne structure of the ML-KEM and ML-DSA primes. By factoring out these identical shift-and-add micro-kernels, we not only accelerate the code but also ensure maximal compliance with simple RISC-V instruction constraints, while maintaining constant-time execution. This results in several key advantages. First, high performance is enabled as shifts and additions correspond to single-cycle instructions with short critical paths (Constraint 1), thereby maximizing throughput. Second, hardware simplification is achieved, eliminating the need for multipliers and reducing the area footprint (Constraint 2). Moreover, being in hardware, this approach avoids the need for branching or comparisons, ensuring faster and constant runtime execution.

Sampler ISE. Sampling is a key operation in PQC, used to generate noise, secrets, and other structured randomness. Common methods include Centered Binomial Distribution (CBD) sampling for small bounded coefficients, Gaussian sampling for precise noise in signatures, and rejection sampling for uniform values over modular domains. CBD sampling is a core component of lattice-based cryptosystems such as ML-KEM, used for generating polynomials with small random coefficients (get-noise-η). In this work, both CBD samplers for $\eta = 2$ (CBD_2) and $\eta = 3$ (CBD_3) have been implemented using custom hardware instructions to accelerate sampling. In ML-KEM, the parameter η controls the variance of the binomial distribution: for each coefficient, η pairs of random bits are sampled, summed, and subtracted to produce centered values in the range $[-\eta, \eta]$.

The computation is organized in two stages: a lightweight preprocessing phase that combines and partially accumulates input bits to simplify coefficient extraction, and a hardware-accelerated stage where coefficients are computed directly using dedicated instructions, avoiding iterative shifts and masking. Each sampler uses a set of custom instructions operating on preprocessed input words, reducing instruction count and latency. Specifically, twelve opcodes are defined—four for CBD_3 and eight for CBD_2—each extracting a coefficient from a 32-bit word in a single cycle, eliminating variable shifting logic and minimizing area overhead. The high-level algorithm is illustrated in Algorithm 7.

Algorithm 7: HORCRUX's CBD Sampling

```
Input: $d$ (32-bit pre-processed input), $j$ (index selector), $insn\_i$ (instruction)
Output: $o$ (8-bit sample output)
1 if $insn\_i = CBD_3\_gen$ then
2     $a \leftarrow (d \gg (6 \cdot j)) \wedge 0x7$
3     $b \leftarrow (d \gg (6 \cdot j + 3)) \wedge 0x7$
4 else if $insn\_i = CBD_2\_gen$ then
5     $a \leftarrow (d \gg (4 \cdot j)) \wedge 0x3$
6     $b \leftarrow (d \gg (4 \cdot j + 2)) \wedge 0x3$
7 else
8     $o \leftarrow a - b$
```

In addition to CBD sampling, we introduce a custom instruction to accelerate the rejection sampling used in ML-DSA (matrix-expand), where uniformly random integers are generated by parsing a stream of random bytes and discarding those outside the valid range $[0, Q)$. This operation, referred to as `rej_uniform`, reconstructs 24-bit integers from three consecutive bytes, applies a mask to keep only the lower 23 bits, and compares the result against the modulus Q. In software, this involves multiple shifts, OR, and mask operations. To reduce instruction count and latency, we replace this logic with a dedicated instruction that takes the three input bytes as operands, performs the bit assembly and masking in hardware, and returns the candidate value in a single operation. This accelerates uniform polynomial generation and simplifies the software loop for in-line sampling.

Galois-Field ISE. Efficient arithmetic in binary finite fields is critical to the performance of many post-quantum schemes, including HQC, where operations such as large-integer multiplications and byte-wise carry-less products are extensively used. As demonstrated during profiling, the operation of multiplication between two polynomials (vec-mul) is the most computationally intensive step in the HQC. This intensity has increased since the scheme transitioned from a sparse-dense multiplicative approach to a dense-dense approach for security reasons. To accelerate this core routine, we introduce a custom ISE supporting both Karatsuba multiplication and $\mathbb{F}_{2^8}$ carry-less multiplication, used in polynomial and binary vector operations for code-based cryptography. Multiplying two large integers of bit-length n by the Schoolbook method requires $\mathcal{O}(n^2)$ single-bit multiplications. In 1962, Karatsuba discovered a divide-and-conquer approach that reduces the number of necessary multiplications to three subproblems of half the size, yielding a time complexity of $\mathcal{O}\big(n^{\log_2 3}\big) \approx \mathcal{O}(n^{1.585})$. Concretely, to multiply two $2m$–bit numbers $A = A_1 \cdot 2^m + A_0$ and $B = B_1 \cdot 2^m + B_0$, where each of A_0, A_1, B_0, B_1 is m bits, one computes: $P_0 = A_0 \times B_0$, $P_2 = A_1 \times B_1$ and $P_1 = (A_1 + A_0) \times (B_1 + B_0) - P_2 - P_0$.

Then the final product is reconstructed as $A \times B = P_2 \cdot 2^{2m} + P_1 \cdot 2^m + P_0$, reducing the number of multiplications from four to three. Our custom Karatsuba accelerator is optimized for a 32-bit processing system and implements the three-way Karatsuba decomposition over two 64-bit operands by splitting each into 32-bit halves: $A = A_1 \parallel A_0, \quad B = B_1 \parallel B_0$, where $A_i, B_i \in \{0, \dots, 2^{32} - 1\}$. Because the full product is 128 bits wide, we stage the computation across four custom instructions (`karats1`–`karats4`), each handling a 32-bit chunk of the final result, as shown in Algorithm 8. ISE `karats1` computes the least-significant 32 bits ($P_0[31:0]$) and writes them directly to the result register. `karats2` computes the most-significant 32 bits ($P_2[127:95]$) and stores them. `karats3` computes the middle 64 bits, breaking them into two 32-bit words, and `karats4` issues an extra store to write the second half of those middle bits.

Algorithm 8: HORCRUX KARATSUBA

```
Input  : $A = A_1 \parallel A_0$, $B = B_1 \parallel B_0$ (64-bit operands as two 32-bit halves)
Output : $P$ (128-bit, as four 32-bit values)
1 karats1: $P_0 \leftarrow A_0 \cdot B_0$
2 karats2: $P_3 \leftarrow A_1 \cdot B_1$
3 karats3: $P_1 \leftarrow MSB((A_0 \oplus A_1) \cdot (B_0 \oplus B_1)) \oplus P_0 \oplus P_3$
4 karats4: $P_2 \leftarrow LSB((A_0 \oplus A_1) \cdot (B_0 \oplus B_1)) \oplus P_0 \oplus P_3$
```

For byte-wise $\mathbb{F}_{2^8}$ multiplication, we implement a constant-time carry-less algorithm using four additional instructions (`gf_carryless_mul_1`–`gf_carryless_mul_4`), each computing one partial contribution to the final 16-bit product. This is shown in Algorithm 9. The instruction semantics emulate a bit-sliced selection of precalculated values based on the input operands, using masked XOR operations in hardware.

All instructions adhere to RISC-V standard formats (Constraint 1), using two or three source operands and one destination register. The hardware minimizes area overhead (Constraint 2) through datapath sharing and the avoidance of multipliers, and it introduces no persistent architectural state (Constraint 3); internal state is transient, confined to each instruction's execution window or sequence, ensuring deterministic behavior.

4 Hardware Architecture

Building upon this design and instruction set described in the previous section (Section 3), this section describes the underlying hardware architecture that realizes the proposed PQC extension. To provide a clearer overview of the custom instructions implemented, Table 2 summarizes all instructions by class, each corresponding to a dedicated hardware module that accelerates critical operations in the PQC algorithms discussed above.

Algorithm 9: HORCRUX's GF-CARRYLESS

```
Input   : Bytes $a$, $b \in \mathbb{F}_{2^8}$ (8-bit, 8-bit)
Output  : $P = a \cdot b$, split as $(P_L, P_H)$ (16-bit)
1  $g_0 \leftarrow$ gf_carryless_mul_1$(a, b)$;
2  $g_1 \leftarrow$ gf_carryless_mul_2$(a, b)$;
3  $g_2 \leftarrow$ gf_carryless_mul_3$(a, b)$;
4  $g_3 \leftarrow$ gf_carryless_mul_4$(a, b)$;
5  $P_L \leftarrow g_0 \oplus (g_1 \ll 2) \oplus (g_2 \ll 4) \oplus (g_3 \ll 6)$;
6  $P_H \leftarrow (g_1 \gg 6) \oplus (g_2 \gg 4) \oplus (g_3 \gg 2)$;
7  if $b[7] = 1$ then
8      $P_L \leftarrow P_L \oplus (a \ll 7)$;
9      $P_H \leftarrow P_H \oplus (a \gg 1)$;
```

Table 2: List of HORCRUX custom instructions grouped by class.

Class	Instruction	Type	Description
KECCAK	`rol32_h`	R4	Performs a left rotation by a variable offset (as explained in Algorithm 1)
	`rol32_l`	R4	
	`bcop32`	R4	Computes $A = B \oplus (\neg C \wedge D)$
COMPRESS	`compress1`	R	Compress polynomial coefficient in ML-KEM (Algorithm 2)
	`compress2`	R	
	`compress3`	R	
	`compress4`	R	
REDUCTION	`KEM-fast-reduce`	R	Reduction for ML-KEM (Algorithm 3)
	`KEM-reduce16`	R	Reduction for ML-KEM (Algorithm 4)
	`DSA-fast-reduce`	R	Reduction for ML-DSA (Algorithm 5)
	`DSA-reduce32`	R	Reduction for ML-DSA (Algorithm 6)
CBD	`cbd3_1`, `cbd3_2`, `cbd3_3`, `cbd3_4`	R	Computes CBD operations using bitwise shifts and masks on input (Algorithm 7)
	`cbd2_1`, `cbd2_2`, `cbd2_3`, `cbd2_4` `cbd2_5`, `cbd2_6`, `cbd2_7`, `cbd2_8`	R	
	`rej_uniform`	R	Accelerate modular rejection sampling
GF	`karats_1`, `karats_2` `karats_3`, `karats_4`	R	Compute 32-bit chunks of 64×64 Karatsuba product (Algorithm 8)
	`gf_carryless_mul_1`, `gf_carryless_mul_2` `gf_carryless_mul_3`, `gf_carryless_mul_4`	R	Compute partials of carry-less $\mathbb{F}_{2^8}$ byte multiplication (Algorithm 9)

The hardware modules associated with the custom instructions in Table 2 are instantiated within a unified and modular framework that facilitates their integration into the core. In the following, we describe how these modules are organized and interconnected, outlining the architectural principles and implementation choices that enable low-latency execution and efficient hardware utilization. To validate the approach, HORCRUX has been integrated into X-HEEP (eXtendable Heterogeneous Energy-Efficient Platform), an open-source RISC-V microcontroller described in System Verilog [MSM$^+$24]. The complete system is shown in Figure 1, and all the code is publicly available on our GitHub.[3]

All input and output signals of HORCRUX are handled by the Core-V eXtension Interface (`CV-X-IF`) [Ope24a]. The `CV-X-IF`, developed by the OpenHW Group, enables seamless connectivity to the CPU core without modifying its pipeline or toolchain. It supports efficient offloading of custom instructions to external accelerators through structured signals that coordinate instruction dispatch and result write-back. Originally introduced in the `CV32E40X` core [Ope24b], `CV-X-IF` has since been adopted by the `CV32E40PX` variant [EE24], where a dispatcher routes essential control and data signals to the interface.

The internal structure of HORCRUX is depicted in Figure 2. It essentially consists of three modules: the decoding block (`id-stage`), the accelerators in `HORCRUX-core`, and the commit block (`commit-stage`). Generally, each module receives its inputs through the `xif-issue-if` package, from which two key signals are extracted: the source register

[3] GitHub Repository: https://github.com/vlsi-lab/HORCRUX

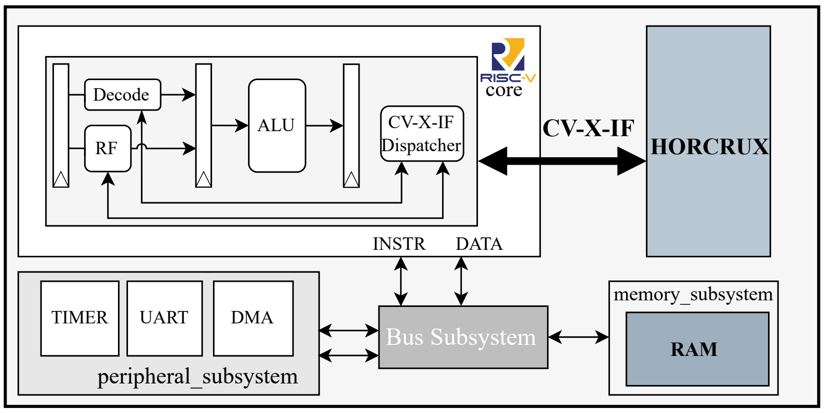

Figure 1: Overview of the X-HEEP architecture and the integration of HORCRUX.

values (`rs1, rs2, rs3`) and the instruction encoding (`insn`) of Figure 2. The `id-stage` is fully generic: it does not hard-code opcodes, but predecodes against the descriptor table in `horcrux_pkg` (e.g., `CoproInstr`, `NbInstr`, `horcrux_insn`). Therefore, adding a new custom instruction only requires (i) updating the package with its mask/match and response fields, and (ii) instantiating the corresponding accelerator in `horcrux-core`; the `id-stage` automatically recognizes it and forwards operands/enable signals without modification. Symmetrically, the `commit-stage` is format-agnostic: it simply writes back the 32-bit result and bookkeeping (`rd`, `id`) when `done` is asserted, so no changes are needed when new modules are added.

In short, extending HORCRUX means editing the SystemVerilog package and dropping in a new module—the decode and commit blocks remain unchanged. This modular organization enables low-latency instruction offloading, parallel operation of independent accelerators, and seamless reintegration of results into the CPU pipeline without disrupting standard RISC-V execution flow.

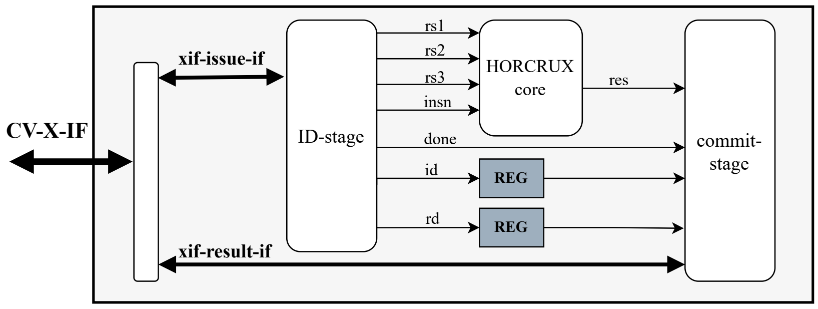

Figure 2: Block diagram of the HORCRUX system module.

4.1 HORCRUX Accelerators

In the following, we provide additional implementation details for each instruction class and compare our accelerator with existing designs.

The testbench was developed through a quantitative analysis of the algorithms, using a SystemVerilog environment for benchmark execution and profiling via performance counters. These profiling tools enabled a precise and efficient assessment of the performance of each operation. Area utilization was assessed by synthesizing selected configurations with

Vivado 2022.2 on the AMD Zynq UltraScale+ ZCU102 platform. To evaluate hardware cost, multiple syntheses were performed: one for each instruction class and one for the complete HORCRUX coprocessor. The partial syntheses, reported in Table 3, Table 4, and Table 5, were conducted solely to compare individual units with state-of-the-art designs. Naturally, in these reduced configurations, the relative area of the `id-stage` is more pronounced, while it becomes negligible in the full integration reported in Table 8 and discussed in Section 5.

All baseline software implementations are taken from the official NIST submission website [Nat20, Nat25c]. These comparisons highlight the effectiveness of the instruction-level extensions in reducing runtime. It is essential to note that the same simulation conditions, including profiling and performance counters, apply to all evaluations presented here, ensuring consistency across the analysis in Section 5.

Keccak. The KECCAK-specific custom instructions are a direct hardware realization of the operations described in Section 3. Both `rol32_h`, `rol32_l`, and `bcop32` are implemented as `R4`-type instructions, each operating on three source registers and producing a single 32-bit result. This design is fully compliant with the RISC-V scalar ISA, which supports three source registers through specialized formats like those used in floating-point instructions (e.g., `fmadd.s` [WA19]). Using more than three source operands requires extensions like RVV [Int21], which rely on a separate register file and encoding. By staying within the standard 32-bit format, our implementation remains compatible with existing toolchains and avoids non-standard encodings. Results are shown in Table 3.

Table 3: Single-block KECCAK clock cycles and implementation results.

Ref.	**Scheme**	**Baseline**	**Optimized**		**FPGA**	**LUTs/FFs/DSPs/BRAMs**
[LKK23]	KECCAK	11,722	1,632	[×7.18]	-	-
[GLM24]	SHA3-512	30,197	22,355	[×1.35]	-	-
[HDT+20]	KECCAK	-	-	[×3]	Xilinx VC707	5,466 / 2,791 / 0 / 0
[FSS20]	SHAKE-256	31,907	308	[×103]	Zynq-7000	3,847 / 0 / 0 / 0
This work	KECCAK	24,787	9,061	[×**2.7**]	ZCU-102	603 / 103 / 0 / 0

[LKK23] proposes custom instructions for the KECCAK-f permutation requiring five source registers—beyond the RISC-V scalar ISA limit of three—achieving a ×7.18 speed-up but with high area overhead from an added register buffer, and targeting an ASIC platform, making it not directly comparable to HORCRUX. [AMI+23] introduces a Keccak-based sampling unit for ML-KEM and ML-DSA, occupying 12,326 LUTs and 3,560 FFs on ZCU-102—over ×20 and ×34 our area, respectively. [HDT+20] presents a SHA-3 accelerator on VC707 (Rocket Chip) using secure write-only memory, with 5,466 LUTs and 2,791 FFs—over nine times our area—and similar speed-up. [GLM24] achieves a modest ×1.35 speed-up using only the *Zknh/zKne* extensions, without hardware optimization. [FSS20] exploits both FPR and GPR through non–RISC-V-compliant instructions invoking a PQ-ALU, reaching ×100 speed-up but consuming 3,847 LUTs on Zynq-7000 (excluding FPR). [Saa24] implements a memory-mapped KECCAK-f[1600] unit on Artix-7 with 5,582 LUTs (over nine times our area). [LVC24] integrates a tightly coupled KECCAK-f[1600] engine with a PQC register (PQCR), reporting 2,528 LUTs on Spartan-7 (excluding PQCR). [SOSK23] combines custom and bit-manipulation instructions with Wide Data Registers (WDR), achieving 40 cycles per KECCAK round and using 1,312 LUTs on Kintex-7.

Beyond standalone benchmarks, the KECCAK ISE demonstrates consistent performance gains across all integrated PQC schemes. In ML-KEM, the hardware-accelerated `hash-h` and `hash-g` functions achieve speedups of ×2.34 and ×2.31, while the SHAKE-based `rkprf` reaches ×2.27. For HQC, the `prng-get-bytes` and `xof-get-bytes` functions both gain ×2.62, and the hashing primitives (`hash-i`, `hash-j`, `hash-g`, `hash-h`) achieve speedups between ×2.04 and ×2.24. Similarly, in ML-DSA, the `uniform-`η and `uniform-`γ samplers see ×2 improvements, while SHAKE-based hashing accelerates by ×2.16. These results confirm that the proposed instruction set delivers uniform benefits across diverse PQC workloads without algorithm-specific tuning. For SLH-DSA (simple/robust), all Keccak-based functions halve their cycles across sign and verify.

Polynomial Coefficient Compression. The hardware module supports multiple compression formats selected via the instruction `insn_i`. It first applies: $u = a + ((a \gg 15) \wedge q)$ to handle negative coefficients correctly. The value of u is then left-shifted by d and offset by either 1664 or 1665, depending on the compression mode, before being entered into the shift-and-add multiplier. This unified architecture minimizes area and latency while maintaining constant-time operation. The compress-ISE is utilized in the pack-cp function, which results in a speed-up of $\times 3.42$. Unlike Gewehr et al. [GLM24], who use constant multipliers and dedicated instructions, our approach avoids multipliers, reducing area and simplifying the datapath while supporting all ML-KEM compression modes. While their implementation achieves up to 78 % speedup, our design reaches 62 %, still delivering significant acceleration. A direct area comparison is not feasible, as they report only ASIC-level results without isolating the compression unit.

Modular Reduction. We implement a single modular reduction unit that supports all four optimized reduction routines described in Section 3. Most of the implementations cited in Table 4 target full NTT, INTT, or PWM acceleration, and typically include dedicated memory blocks to handle polynomial coefficients during computation. While effective for improving throughput, such designs are not RISC-V compliant, as they require custom memory management and modifications to the execution model. [AEL+20] adds instruction to `rv32im` ISE, exploiting a BRAM on the Artix-7 FPGA. [FSS20] implement NTT and modular reduction unit exploiting on-the-fly twiddle factor calculation, accessing FPRs, and storing two coefficients in one word. They implement it on a Zynq-7000 FPGA, occupying more than 4× our resources, and 6 DSPs. [KA20] proposes a domain-specific NTT architecture that uses application-specific dynamic instruction scheduling and memory dependence prediction. NTT has a speed-up of $\times 3.7$ and $\times 2.8$, respectively, for Kyber and Dilithium. It occupies just 417 LUTs when synthesized on Virtex-7, but INTT and PWM are not considered. [LQYW24] adopts k^2-reduction to the design of modulo operations, which extremely reduces the area overhead. [MBB+23] implements a very compact modular reduction unit, using a unified multiplier for both Kyber and Dilithium. It is implemented on Zynq-7000 FPGA, having half of our LUTs, but uses 2 DSPs, and a speed-up between $\times 3.4$ and $\times 3.6$. [NDMZ+21] has higher speed-up, but two different structures are implemented on ZCU-106 FPGA, losing the versatility between Kyber and Dilithium. Lastly, [SOSK23] implements a PQ-ALU, with a Twiddle and Round Counter Update unit (TRCU) and a Register Address Unit (RAU), achieving a high speed-up, but at the expense of very high area occupancy on the Kintex-7.

Table 4: Comparison of NTT/INTT/PWM implementations for ML-KEM (K) and ML-DSA (D) on various FPGAs: ▲ Artix-7 35T, ■ Zynq-7000, ◊ ZCU-106, ★ ZCU-102, ▽ Kintex-7.

Ref.		**Baseline**			**Optimized**						**FPGA**	**LUTs/FFs/ DSPs/BRAMs**
		NTT	**INTT**	**PWM**	**NTT**		**INTT**		**PWM**			
[AEL+20]	K	34,538	56,040	13,634	6,868	[×**5.0**]	6,367	[×**8.8**]	2,395	[×**5.7**]	[▲]	104 / 35 / 0 / 1
[FSS20]	K	34,703	53,636	-	1,930	[×**17.9**]	1,930	[×**27.8**]	-		[■]	2,908 / 170 / 9 / 0
[LQYW24]	K	24,670	34,911	9,263	4,189	[×**5.9**]	3,481	[×**10**]	3,257	[×**2.8**]	[▲]	93 / 0 / 1 / 0
[MBB+23]	D	17,041	20,371	4,346	4,753	[×**3.6**]	6,027	[×**3.4**]	1,274	[×**3.4**]	[■]	360 / 16 / 2 / 0
[NDMZ+21]	K	50,355	76,912	-	18,488	[×**2.7**]	18,488	[×**4.2**]	-		[◊]	178 / 0 / 5 / 0.5
	D	38,043	46,266	-	18,554	[×**2.0**]	21,375	[×**2.2**]	-			
[SOSK23]	K	37,975	53,189	16,417	1,454	[×**26.1**]	1,726	[×**30.8**]	1,448	[×**11.3**]	[▽]	3,387 / 951 / 33 / 0
	D	40,002	46,649	7,455	1,972	[×**20.3**]	2,244	[×**20.8**]	768	[×**9.7**]		
This work	K	46,923	91,626	73,948	21,693	[×**2.2**]	18,757	[×**4.88**]	11,321	[×**6.53**]	[★]	632 / 76 / 0 / 0
	D	46,920	55,911	191,771	23,896	[×**1.96**]	28,233	[×**1.98**]	109,698	[×**1.75**]		

Our approach, in contrast, focuses on small, ISA-compliant extensions that integrate tightly into the existing RISC-V pipeline, avoiding changes to memory hierarchy or instruction flow. Importantly, these modular reductions are not limited to NTT/INTT operations. In ML-KEM, reductions are also invoked during polynomial compression, decompression, and serialization steps, as well as during key generation and encapsulation, where modular products must be brought back into canonical form. In ML-DSA, reduction routines are used extensively during the matrix expansion, polynomial accumulation, and signature verification phases to reduce wide intermediate values back into the valid range modulo Q. Despite tighter constraints of our design, we achieve performance improvements of up to $\times$**6.53** for ML-KEM and $\times$**1.98** for ML-DSA, as reported in Table 4. These results validate the efficiency of our approach in achieving high-performance, low-area PQC support while maintaining full RISC-V compliance.

Sampler. The CBD unit is a straightforward implementation of Algorithm 7. [YSZ+24] splits input into sub-words, performs additions with multiple CPAs in parallel, and computes $a - b$ using dedicated adders per extraction window. They report an acceleration rate of 12.5 for CBD_2 and of 16.3 for CBD_3. In [GLM24], CBD_2 and CBD_3 sampling are accelerated via dedicated *Xkyber* instructions processing two coefficients in parallel, achieving approximately a speed-up factor of $\times 2.3/\times 3.6$ in cycle count compared to the software baseline. [XHY+20], [BUC19], and [MBB+23] integrate coefficient sampling into KECCAK hardware without isolating or optimizing CBD. Our approach utilizes indexed extraction with custom instructions for per-sample processing, resulting in a speed-up factor of $\times 2.11$ for η_1 and $\times 1.90$ for η_2.

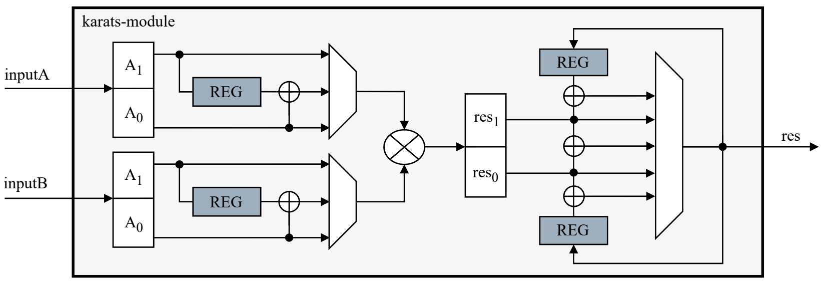

Figure 3: Architecture of the HORCRUX binary-field Karatsuba module.

Galois-Field. The high-level datapath of the binary-field Karatsuba-multiplication accelerator is shown in Figure 3. The two inputs are passed 32-bit values at a time. A single 32×32 multiplier unit (shown in the middle of the figure) implements carry-less multiplication, enabling rapid computation of the terms $A_0B_0, A_1B_1, (A_1 \oplus A_0)(B_1 \oplus B_0)$ in hardware with minimal depth. Each part of the raw product is generated in a different clock cycle and then sent to the output, 32 bits at a time. Each custom instruction handles differently how to feed inputs to the multiplier, gate the reconstruction logic, and strobe the stores. From the programmer's view, the entire 64×64-bit Karatsuba multiplication is performed by invoking these four instructions in order.

As explained in Section 3, we also implement byte-wise multiplication in $\mathbb{F}_{2^8}$. Each instruction extracts a 2-bit slice of the operand and uses bit mask-based logic to select the corresponding partial product from a precomputed set. This approach avoids conditional branches, employing bitwise masking and XOR operations to combine contributions from each candidate. To assess the performance of our Galois-field hardware extensions,

Table 5: HQC-related $\mathbb{F}_2$ arithmetic implementation results on the ZCU-102.

Scheme	**Baseline Clock Cycles**	**Optimized Clock Cycles**	**LUTs/FFs/DSPs/BRAMs**
HQC- Schoolbook-mul	2,805	13 [×**215**]	932 / 363 / 0 / 0
HQC- Decoder	8,559,360	1,938,886 [×**4.4**]	

we measured clock-cycle reductions for both binary-field multiplications and the entire decoding. Note that the binary-field multiplication refers to the inner base multiplication operation (`schoolbook_mul`), which operates on 64-bit words, not the entire polynomial. Our accelerator achieves a substantial improvement, with the HQC base multiplication clock cycles reduced by a factor of ×215 and the full HQC decoder reduced by ×4.4 (Table 5). Compared to the Kintex-7 implementation in [DDMV+25], which required 52 clock cycles for a single HQC base multiplication, our solution reduces the cycle count to just 13, yielding a ×4 speedup. While their design occupies 905 LUTs and 255 FFs, our version uses a comparable 932 LUTs and 363 FFs, with no DSP or BRAM usage in either case. However, while [DDMV+25] reports the resource cost of the Karatsuba datapath alone, our design includes the necessary `CV-X-IF` controller logic and tightly coupled instruction support. Moreover, our implementation refers to the latest HQC specification, updated in August 2025. The ASIC-based work in [LKK23] reports a reduction in Karatsuba multiplication by a factor of 7.55, and a reduction in the Reed-Solomon decoder by a factor of 2.86. In contrast, our ZCU-102 implementation achieves a much higher reduction: the overall vector multiplication for HQC is reduced by a factor of 26.26, and the entire decoding stage by a factor of 4.4. Note that [LKK23] reports performance for the full coprocessor, while our results focus on the acceleration module alone.

5 Results

This section evaluates the proposed PQC-ISA prototype in terms of code size, execution performance, and area overhead for all four supported NIST PQC algorithms.

Code Size. Table 6 compares the `.text` section size before and after optimization for all evaluated schemes. The measurements were obtained using the `riscv32-unknown-elf-size` utility from the GCC toolchain. Results show consistent reductions, with code size improvements ranging from approximately 10% to 15% (1.10×–1.15×), confirming that the proposed optimizations effectively decrease the code footprint across all schemes.

Table 6: Comparison of baseline and optimized code size (`.text`) in bytes.

KEM	Baseline	Optimized		SIGN	Baseline	Optimized	
ML-KEM-512	27,564	24,004	[**1.15**×]	ML-DSA-2	32,704	28,472	[**1.15**×]
ML-KEM-768	27,400	23,572	[**1.16**×]	ML-DSA-3	32,424	28,688	[**1.13**×]
ML-KEM-1024	28,008	24,280	[**1.15**×]	ML-DSA-5	32,072	28,416	[**1.13**×]
HQC-1	37,606	33,654	[**1.12**×]	SLH-DSA-128f-s	27,624	24,140	[**1.14**×]
HQC-3	38,838	34,870	[**1.11**×]	SLH-DSA-192f-s	27,596	24,116	[**1.14**×]
HQC-5	45,426	41,430	[**1.10**×]	SLH-DSA-256f-s	28,000	24,516	[**1.14**×]

Performance. The performance results are reported in Table 7. For each algorithm, we executed the full reference flow against the NIST KATs—generated with the official KAT framework [Nat20, Nat25c]—to validate correctness. Cycle counts in Table 7 are averages, computed over 100 independent simulation runs per operation to ensure statistical consistency and mitigate measurement variability. For Sign, cycles are measured from API entry to return on the KAT messages and seeds, considering an average-case behavior under the KAT-driven randomness. Although the accelerator supports all SLH-DSA variants, we tested all *s* variants, and just one *r* variant. The remaining variants should perform similarly, so we omitted them to reduce simulation time. The results in Table 7 demonstrate substantial performance improvements across all targeted PQC schemes using our optimized ISE-based HORCRUX implementation. For lattice-based schemes like ML-KEM and ML-DSA, we observe consistent 2× to 2.9× speed-ups, while (binary-field) code-based schemes such as HQC benefit from even greater acceleration, achieving up to 34× improvement in key generation and encapsulation. Similarly, SLH-DSA exhibits

Table 7: Kilo-Clock cycles for baseline and optimized KeyGen, Encaps/Sign, and Decaps/Verify. Optimized columns also include the corresponding speed-up.

Algorithm	**Baseline [kCycles]**			**Optimized [kCycles]**		
	KeyGen	Encaps/ Sign	Decaps/ Verify	KeyGen	Encaps/ Sign	Decaps/ Verify
ML-KEM-512	1,250	1,470	1,805	484 [×**2.58**]	509 [×**2.89**]	612 [×**2.95**]
ML-KEM-768	2,068	2,402	2,842	801 [×**2.58**]	849 [×**2.83**]	992 [×**2.86**]
ML-KEM-1024	3,274	3,657	4,208	1,260 [×**2.58**]	1,337 [×**2.74**]	1,518 [×**2.77**]
HQC-1	57,647	115,082	174,718	2,447 [×**23.6**]	4,689 [×**24.5**]	8,042 [×**21.7**]
HQC-3	174,267	348,111	524,831	6,947 [×**25.1**]	13,381 [×**26.0**]	21,246 [×**24.7**]
HQC-5	439,928	879,182	1,324,315	13,412 [×**32.8**]	25,800 [×**34.1**]	40,628 [×**32.6**]
ML-DSA-2	3,713	7,140	3,853	1,770 [×**2.09**]	3,579 [×**1.99**]	1,863 [×**2.07**]
ML-DSA-3	6,522	24,265	6,530	3,217 [×**2.03**]	13,712 [×**1.77**]	3,297 [×**1.98**]
ML-DSA-5	11,058	28,353	11,245	5,191 [×**2.13**]	14,156 [×**2.00**]	5,328 [×**2.11**]
SLH-DSA-128-fs	148,920	3,486,490	204,805	76,565 [×**1.94**]	1,793,416 [×**1.94**]	105,412 [×**1.94**]
SLH-DSA-128-fr	286,087	6,642,087	370,937	145,947 [×**1.96**]	3,387,032 [×**1.96**]	189,008 [×**1.96**]
SLH-DSA-192-fs	221,042	5,689,179	303,500	114,140 [×**1.94**]	2,957,600 [×**1.92**]	156,758 [×**1.94**]
SLH-DSA-256-fs	585,456	11,784,433	321,355	306,815 [×**1.91**]	6,188,240 [×**1.90**]	167,431 [×**1.91**]

nearly 2× gains across all operations. These results highlight the efficiency and broad applicability of our architecture across diverse post-quantum algorithm families. It is worth noting that the actual speedup achieved for each algorithm is higher than the values reported in Table 7. For instance, in the case of ML-KEM-512, the measured speedup for encryption and decryption was ×2.94 and ×3.42, respectively, compared to the reported ×2.89 and ×2.95 for encapsulation and decapsulation. This difference stems from the fact that certain operations—such as *memcpy* routines and the management of intermediate variables exchanged between software functions—were not accelerated in hardware. A similar behavior was observed across all tested algorithms. All implementations were compiled using the GCC toolchain with the `-O2` optimization flag.

Area. Table 8 shows the resource utilization of the HORCRUX modules on a Zynq UltraScale+ FPGA (ZCU102) using Vivado 2022.2. The baseline X-HEEP SoC consumes 33,642 LUTs and 27,314 registers, of which the RISC-V core alone accounts for 6,452 LUTs and 2,165 registers. HORCRUX adds 2,824 LUTs (≈ 8.4%) and 395 registers (≈ 1.4%), which is significantly smaller than the RISC-V core itself, highlighting the compactness and low overhead of the proposed architecture. The `horcrux-core` is the dominant contributor, while `id_stage` and `commit_stage` introduce minimal logic. Notably, no BRAMs or DSP blocks are instantiated in the HORCRUX design. All modules have been synthesized at 15 MHz, matching the standard synthesis frequency of X-HEEP to enable full application testing with JTAG support. However, the system is capable of being synthesized at up to 50 MHz, and HORCRUX does not introduce any additional critical path delays, as it is not part of the timing-critical logic of the system. To preserve module hierarchy, reported values are obtained with Vivado's `unflatten` option. If the `flatten` option is enabled, the HORCRUX logic is further optimized to 2,696 LUTs and 395 registers. This difference, although apparently minor, can become more significant when integrating additional modules, confirming the scalability and optimization margin of the design.

The results prove that we achieve significant cryptographic functionality with minimal logic and register footprint, while preserving memory and DSP availability. This makes it an attractive extension for lightweight PQC support on embedded RISC-V platforms.

5.1 Final Comparison

A comprehensive overview of the most relevant related works is reported in Table 9, where we collect and compare FPGA-based implementations previously analyzed in Section 4. The table highlights the area occupation and speedup results across multiple PQC schemes.

The key observation is that all the coprocessors presented in Table 9, except ours,

Table 8: Resource utilization of HORCRUX modules (ZCU102 running @ 15 MHz).

Component	LUTs	Registers
◇ `X-HEEP`	33,642	27,314
○ RISC-V core	6,452	2,165
◇ HORCRUX	**2,820**	**395**
○ `id_stage`	124	103
○ `horcrux-core`	2,693	287
□ Keccak	478	0
□ Modular Reduction	615	0
□ Polynomial Coefficient Compression	680	0
□ Sampler	28	0
□ HQC-related $\mathbb{F}_2$ arithmetic (Karatsuba multiplier and decoder)	787	287
○ `commit_stage`	3	0

lack full support for all the standardized PQC algorithms. Among them, only the work presented in [LKK23] implements all the round-4 finalist algorithms of the NIST PQC standardization process. However, that design is implemented in a 28nm CMOS technology, hence a direct comparison with our FPGA-based architecture is less meaningful. Notably, [LKK23] is reported to have a significantly larger area compared to [FSS20], [NDMZ+21], and [KA20], which are already larger tightly-coupled accelerators than HORCRUX. The same applies to [GLM24]. Regarding [2], [3], [4], and [9] in Table 9 (respectively, [AMI+23], [DDMV+25], [FSS20], and [YSZ+24]), all these works exhibit a significantly higher area than HORCRUX. [AMI+23] achieves similar speedups to ours but with a larger hardware footprint. [DDMV+25] uses nearly 4× more area, yet only accelerates HQC and shows comparable speedups to ours. [FSS20] consumes more than 8× the area, providing a speedup of only 4× in key generation, while the encapsulation and decapsulation performances are close to ours. [YSZ+24] offers higher speedups, but the area is roughly 6× that of HORCRUX and supports only the CRYSTALS family of algorithms. Conversely, the works referenced as [1], [5], [6], [7], and [8] - [AEL+20], [LQYW24], [LVC24], [MBB+23], and [NDMZ+21] - report lower area footprints than HORCRUX, but they support only one or two algorithms, and therefore lack generality. Also, [JDH+25] focuses solely on software optimization and is thus not directly comparable to our hardware-oriented implementation. Although the design is tailored for these four schemes, results demonstrate that significant gains are achievable with minimal area impact while maintaining architectural flexibility for future extensions.

6 Discussion and Future Works

While the architecture presented in this work is a prototype specifically optimized for the four selected NIST PQC algorithms (ML-KEM, ML-DSA, SLH-DSA, and HQC), crypto agility has been a guiding principle throughout the design. As highlighted in NIST's recent guidance [Nat25a], crypto agility is the ability to efficiently support and switch between multiple cryptographic schemes using shared hardware resources. Ensuring this property is essential to maintain long-term resilience as cryptographic landscapes evolve.

Although HORCRUX is tailored to specific algorithms, it remains flexible. The pipeline uses fixed decode and commit logic, so adding an instruction only requires (i) declaring it in the decoder package and (ii) instantiating its logic module. No edits are needed to the `id-stage`, `commit-stage`. Moreover, several modules already exhibit strong cryptographic agility. The Keccak accelerator implements the core permutation. It is reusable across many PQC schemes and directly supports standard Keccak-based primitives (SHA3/SHAKE and SP 800-185 variants like cSHAKE and KMAC), as well as the *new signature on-ramps*, underscoring its wide applicability. The modular reduction unit, although tuned for ML-KEM and ML-DSA, supports a wide range of power-of-two shifted inputs and can be easily adapted to new pseudo-Mersenne primes with minimal

Table 9: Performance comparison for multiple PQC schemes running on: ▲ Artix-7 35T, ■ Zynq-7000, ◊ ZCU-106, ★ ZCU-102, ▽ Kintex-7, ♦ Spartan-7.
Reference Legend: [1] [AEL+20], [2] [AMI+23], [3] [DDMV+25], [4] [FSS20], [5] [LQYW24], [6] [LVC24], [7] [MBB+23], [8] [NDMZ+21], [9] [YSZ+24].

Ref.	**Scheme**	**LUTs/FFs/DSPs/BRAMs**	**Optimized (kCycles)**			**Speedup**		
		FPGA	**K**	**E/S**	**D/V**	**K**	**E/S**	**D/V**
[1]	ML-KEM-512	104 / 35 / 0 / 1	710	971	870	×1.31	×1.34	×1.55
	ML-KEM-1024	[▲]	2,203	2,619	2429	×1.25	×1.27	×1.38
[2]	ML-KEM-1024	23,000 / 9,700 / 4 / 24	9	11	14	×1	×1.22	×1.56
	ML-DSA-3	[★]	40	54	47	×1	×1.35	×1.18
[3]	HQC-128	8,894 / 3,710 / 0 / 0	3,109	6,303	11,095	×21.24	×21.15	×18.80
	HQC-192	[▽]	10,848	22,579	34,895	×18.01	×17.40	×17.21
	HQC-256		20,154	41,470	65,000	×17.73	×17.34	×17.02
[4]	ML-KEM-512	24,306 / 10,837 / 37 / 32	150	706	805	×7.57	×2.15	×1.32
	ML-KEM-768	[■]	273	828	945	×6.47	×2.91	×1.76
	ML-KEM-1024		850	1,405	2,027	×4.45	×3.58	×1.62
[5]	ML-KEM-512	93 / 0 / 1 / 0	710	971	870	×1.69	×1.61	×1.78
	ML-KEM-768	[▲]	988	1,237	1,133	×1.97	×2.03	×2.18
	ML-KEM-1024		1,543	1,851	1,719	×1.99	×2.03	×2.15
[6]	SLH-DSA-128fs	2,528 / 0 / 0 / 0	8,520	199,950	13,080	×13.95	×13.92	×12.54
	SLH-DSA-128fr	[♦]	15,570	362,170	24,170	×14.74	×14.71	×13.97
[7]	ML-DSA-3	360 / 16 / 2 / 0 [■]	4,316	5,253	4,178	-	-	-
[8]	ML-KEM-512	178 / 0 / 5 / 0.5	420	438	101	×1.56	×1.87	×2.68
	ML-KEM-768	[◊]	695	732	130	×1.59	×1.81	×2.69
	ML-KEM-1024		1091	1126	160	×1.57	×1.76	×2.70
	ML-DSA-2	377 / 0 / 10 / 0.5	1,592	5884	1701	×1.18	×1.39	×1.23
	ML-DSA-3	[◊]	2,974	10,211	2,969	×1.14	×1.36	×1.19
	ML-DSA-5		5,001	13,340	5,133	×1.12	×1.32	×1.16
[9]	ML-KEM-512	19,325 / 6,717 / 32 / 4	89	89	108	×10.36	×13.85	×12.70
	ML-KEM-768	[■]	164	166	197	×11.63	×13.83	×12.29
	ML-KEM-1024		195	198	244	×11.40	×13.63	×11.68
	ML-DSA-2		467	1,173	537	×3.10	×5.42	×8.66
	ML-DSA-3		284	746	344	×3.76	×4.36	×3.28
	ML-DSA-5		177	594	208	×14.45	×7.13	×13.89
This Work	ML-KEM-512	**2,820 / 395 / 0 / 0**	484	509	612	×**2.58**	×**2.89**	×**2.95**
	ML-KEM-768	[★]	801	849	992	×**2.58**	×**2.83**	×**2.86**
	ML-KEM-1024		1,260	1,337	1,518	×**2.58**	×**2.74**	×**2.77**
	HQC-1		2,447	4,689	8,042	×**23.6**	×**24.5**	×**21.7**
	HQC-3		6,947	13,381	21,246	×**25.1**	×**26.0**	×**24.7**
	HQC-5		13,412	25,800	40,628	×**32.8**	×**34.1**	×**32.6**
	ML-DSA-2		1,770	3,579	1,863	×**2.09**	×**1.99**	×**2.07**
	ML-DSA-3		3,217	13,712	3,297	×**2.03**	×**1.77**	×**1.98**
	ML-DSA-5		5,191	14,156	5,328	×**2.13**	×**2.00**	×**2.11**
	SLH-DSA-128fs		76,565	1,793,416	105,412	×**1.94**	×**1.94**	×**1.94**
	SLH-DSA-192fs		114,140	2,957,600	156,758	×**1.94**	×**1.92**	×**1.94**
	SLH-DSA-256fs		306,815	6,188,240	167,431	×**1.91**	×**1.90**	×**1.91**

changes. For polynomial coefficient compression, new scaling behaviors can be supported by extending the instruction set with additional variants.

Overall, because instructions are designed to accelerate general subfunctions (e.g., reduction, permutation, compression), adapting to new cryptographic algorithms typically requires changes only in the HORCRUX datapath. The pipeline and software infrastructure remain untouched, enabling efficient and predictable evolution of the PQC-ISA.

7 Conclusions

This work introduced HORCRUX, a RISC-V compliant ISE designed to support all four NIST-standardized PQC algorithms, ML-KEM, ML-DSA, SLH-DSA, and HQC, within a single, lightweight coprocessor. Unlike prior efforts that typically target individual schemes or submodules, HORCRUX provides a unified and scalable architecture that balances performance, area efficiency, and design simplicity. Through the integration of minimal yet effective custom instructions, HORCRUX accelerates critical operations such as modular reduction, KECCAK permutation, polynomial compression, sampling, and Galois-field arithmetic, all while maintaining compatibility with the RISC-V toolchain. Our results show consistent improvements in execution time and code size across all algorithms, with speedups of up to 3× for lattice and hash-based schemes, and over 30× for code-based schemes, while occupying fewer resources than many state-of-the-art designs. Beyond raw performance, HORCRUX embraces a modular and crypto-agile philosophy: each instruction class is reusable across different algorithms and easily extensible to future cryptographic standards and draft-specification changes. This work serves as a practical reference for assessing implementation costs, identifying performance bottlenecks, and understanding key challenges (based on the presented profiling results) in realizing the targeted NIST PQC standards within a HW/SW co-design framework.

References

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic - accelerating kyber and NewHope on RISC-v. Cryptology ePrint Archive, Paper 2020/049, 2020.

[ALCZ20] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. FPGA-based SPHINCS+ Implementations: Mind the Glitch. *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 229–237, 2020.

[AMI+23] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(2):747–758, 2023.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Proceedings*, pages 311–323, Santa Barbara, California, USA, 1986.

[BDC20] Utsav Banerjee, Siddharth Das, and Anantha P. Chandrakasan. Accelerating Post-Quantum Cryptography using an Energy-Efficient TLS Crypto-Processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[BDP+15] Guido Bertoni, Joan Daemen, Micheal Peeters, Gilles Van Assche, and Van Keer Ronny. Fips pub 202 - sha-3 standard: Permutation-based hash and extendable-output functions, 2015.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61, Aug. 2019.

[CGM+22] Hao Cheng, Johann Großschädl, Ben Marshall, Dan Page, and Thinh Pham. RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography. Cryptology ePrint Archive, Paper 2022/1697, 2022.

[DDMV+25] Alessandra Dolmeta, Stefano Di Matteo, Emanuele Valea, Mikael Carmona, Antoine Loiseau, Maurizio Martina, and Guido Masera. TYRCA: A RISC-V Tightly-Coupled Accelerator for Code-Based Cryptography. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025.

[EE24] ESL-EPFL. cv32e40px: RISC-V Core with Extensions for Cryptography and Machine Learning. https://github.com/esl-epfl/cv32e40px, 2024. Accessed: 2024-10-11.

[FBR$^+$21] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. Cryptology ePrint Archive, Paper 2021/479, 2021.

[FSMG$^+$19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepulveda. Towards Reliable and Secure Post-Quantum Co-Processors based on RISC-V. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1148–1153, 2019.

[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*, 2020(4):239–280, Aug. 2020.

[GLM24] Carlos Gewehr, Lucas Luza, and Fernando Gehm Moraes. Hardware Acceleration of Crystals-Kyber in Low-Complexity Embedded Systems With RISC-V Instruction Set Extensions. *IEEE Access*, 12:94477–94495, 2024.

[HDT$^+$20] Trong-Thuc Hoang, Ckristian Duran, Akira Tsukamoto, Kuniyasu Suzaki, and Cong-Kha Pham. Cryptographic accelerators for trusted execution environment in risc-v processors. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2020.

[HQC25] HQC Team. Hqc specifications. https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf, 2025. Version dated 2025-08-22.

[Int21] RISC-V International. The RISC-V Vector Extension, Version 1.0. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0, 2021. Ratified by RISC-V International.

[JDH$^+$25] Xinyi Ji, Jiankuo Dong, Junhao Huang, Zhijian Yuan, Wangchen Dai, Fu Xiao, and Jingqiang Lin. ECO-CRYSTALS: Efficient Cryptography CRYSTALS on Standard RISC-V ISA. *IEEE Transactions on Computers*, 74(2):401–413, 2025.

[KA20] Emre Karabulut and Aydin Aysu. RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 26–32, 2020.

[KGHRM23] Yao-Ming Kuo, Francisco García-Herrero, Oscar Ruano, and Juan Antonio Maestro. RISC-V Galois Field ISA Extension for Non-Binary Error-Correction Codes and Classical and Post-Quantum Cryptography. *IEEE Transactions on Computers*, 72(3):682–692, 2023.

[LKK23] Jihye Lee, Whijin Kim, and Ji-Hoon Kim. A Programmable Crypto-Processor for National Institute of Standards and Technology Post-Quantum Cryptography Standardization Based on the RISC-V Architecture. *Sensors*, 23(23), 2023.

[LKKK22] Jihye Lee, Whijin Kim, Sohyeon Kim, and Ji-Hoon Kim. Post-Quantum Cryptography Coprocessor for RISC-V CPU Core. In *2022 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–2, 2022.

[low24] lowRISC. lowrisc tackles post-quantum cryptography challenges through research collaborations. https://lowrisc.org/news/lowrisc-tackles-post-quantum-cryptography-challenges-through//-research-collaborations/, 2024. Accessed: 2025-07-07.

[LQYW24] Lu Li, Guofeng Qin, Yang Yu, and Weijia Wang. Compact Instruction Set Extensions for Kyber. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(3):756–760, 2024.

[LVC24] Jonathan Lopez-Valdivieso and Rene Cumplido. Design and Implementation of Hardware-Software Architecture Based on Hashes for SPHINCS+. *ACM Trans. Reconfigurable Technol. Syst.*, 17(4), October 2024.

[MBB$^+$23] Konstantina Miteloudi, Joppe Bos, Olivier Bronchain, Björn Fay, and Joost Renes. PQ.v.ALU.e: Post-quantum RISC-v custom ALU extensions on dilithium and kyber. Cryptology ePrint Archive, Paper 2023/1505, 2023.

[MNP$^+$20] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):109–136, Dec. 2020.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[MP21] Ben Marshall and Dan Page. SME: Scalable masking extensions. Cryptology ePrint Archive, Paper 2021/1416, 2021.

[MPST$^+$16] Dustin Moody, Ray Perlner, Daniel Smith-Tone, Angela Robinson, Rene Peralta, and Lily Chen. Post-Quantum Cryptography: NIST's Plan for the Future. Technical Report NISTIR 8105, National Institute of Standards and Technology, 2016.

[MSM$^+$24] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators, 2024.

[MSS24] Stefano Di Matteo, Ivan Sarno, and Sergio Saponara. CRYPHTOR: A Memory-Unified NTT-Based Hardware Accelerator for Post-Quantum CRYSTALS Algorithms. *IEEE Access*, 12:25501–25511, 2024.

[Nat20] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization – Round 3 Submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, 2020. Accessed: 2025-07-15.

[Nat24a] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Algorithm (ML-DSA) Standard (FIPS 204). FIPS 204, NIST, 2024.

[Nat24b] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) Standard (FIPS 203). FIPS 203, NIST, 2024.

[Nat24c] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) Standard (FIPS 205). FIPS 205, NIST, 2024.

[Nat25a] National Institute of Standards and Technology. Considerations for Cryptographic Algorithm Agility. https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.39.ipd.pdf, 2025. Accessed: 2025-07-15.

[Nat25b] National Institute of Standards and Technology. NIST Selects HQC as Fifth Algorithm for Post-Quantum Encryption (NIST IR 8545). Internal Report IR 8545, NIST, 2025.

[Nat25c] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization – Round 4 Submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions, 2025. Accessed: 2025-07-15.

[NDMZ$^+$21] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. *IEEE Access*, 9:150798–150808, 2021.

[NIS25] NIST. Post-Quantum Cryptography — FAQs. https://csrc.nist.gov/projects/post-quantum-cryptography/faqs, 2025. Accessed: 2025-05-09.

[OBS23] Thomas Oder, Subhadeep Banik, and Thomas Schneider. A Survey of Hardware Architectures for Post-Quantum Cryptography. *ACM Computing Surveys (CSUR)*, 55(12):1–39, 2023.

[Ope24a] OpenHW Group. *Core-V eXtension Interface (CV-X-IF) Specification*. OpenHW Group, 2024. Version f08d951d (rev. latest).

[Ope24b] OpenHW Group. *CV32E40X User Manual*, 2024. Accessed: 2024-06-11.

[RIS22] RISC-V Crypto. RISC-V Cryptographic Extension Proposals, Technical Report Volume I: Scalar & Entropy Source Instructions. Technical Report Version 1.0.1, GitHub, 2022.

[RLC$^+$25] Antonio Ras, Antoine Loiseau, Mikaël Carmona, Simon Pontié, Guénaël Renault, Benjamin Smith, and Emanuele Valea. PHOENIX: Crypto-agile hardware sharing for ML-KEM and HQC. Cryptology ePrint Archive, Paper 2025/601, 2025.

[Saa24] Markku-Juhani O. Saarinen. Accelerating SLH-DSA by Two Orders of Magnitude with a Single Hash Unit. Cryptology ePrint Archive, Paper 2024/367, 2024.

[Sol99] Jerome A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, National Security Agency, Communications Security Establishment, 1999.

[SOSK23] Tobias Stelzer, Felix Oberhansl, Jonas Schupp, and Patrick Karl. Enabling Lattice-Based Post-Quantum Cryptography on the OpenTitan Platform. In *Proceedings of the 2023 Workshop on Attacks and Solutions in Hardware Security*, ASHES '23, page 51–60, New York, NY, USA, 2023. Association for Computing Machinery.

[TGMD20] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. RISC-V Extension for Lightweight Cryptography. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 222–228, 2020.

[WA19] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213. `https://github.com/riscv/riscv-isa-manual`, 2019. Ratified by RISC-V International.

[WZZ+24] Tengfei Wang, Chi Zhang, Xiaolin Zhang, Dawu Gu, and Pei Cao. Optimized Hardware-Software Co-Design for Kyber and Dilithium on RISC-V SoC FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):99–135, Jul. 2024.

[XHY+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2672–2684, 2020.

[YLW+25] Zewen Ye, Xin Li, Chuhui Wang, Ray C. C. Cheung, and Kejie Huang. RVSLH: Acceleration of Postquantum Standard SLH-DSA With Customized RISC-V Processor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–5, 2025.

[YSZ+24] Zewen Ye, Ruibing Song, Hao Zhang, Donglong Chen, Ray Chak-Chung Cheung, and Kejie Huang. A Highly-efficient Lattice-based Post-Quantum Cryptography Processor for IoT Applications. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):130–153, Mar. 2024.

[ZZYH24] Shengnan Zhang, Yifan Zhao, Xinglong Yu, and Jun Han. RISC-V Domain-Specific Processor for Accelerating SPHINCS+ on Multi-Core Architecture. In *2024 IEEE 17th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, pages 1–3, 2024.

Appendix A

Table 10, Table 11, Table 12, and Table 13 present the profiling results for the target PQC schemes, reporting the number of clock cycles for the main stages — KeyGen, Encaps/Sign, and Decaps/Verify — as well as for their most critical submodules. Not all subfunctions of each module are shown: only those that exhibit a measurable improvement compared to the original software implementation are included. Moreover, several functions are called multiple times within the same module; however, the tables report the number of cycles for a single invocation, since the relative speed-up remains the same regardless of the total number of calls. Submodule names mirror those in the official NIST reference implementations for each scheme (ML-KEM, ML-DSA, SLH-DSA, and HQC), as distributed via the Round-3/Round-4 submission repositories [Nat20, Nat25c], ensuring a one-to-one correspondence with the canonical software baselines. The entry labeled *Others* represents all remaining operations that are either non-optimized or repeated internally, and are thus omitted for clarity.

Table 10: ML-KEM-512 cycle breakdown.

Algorithm	Module	Baseline	Optimized	Sub-Modules	Baseline	Optimized
ML-KEM-512	KeyGen	1,249,921	484,229 [×2.58]	hash-h	166,383	70,966 [×2.34]
				PKE-KeyGen	1,084,803	417,365 [×2.6]
				hash-g	28,172	12,191 [×2.31]
				gen-matrix	361,78	171,135 [×2.11]
				get-noise-η_1	60,470	26,319 [×2.3]
				vec-NTT	120,784	34,701 [×3.48]
				basemul	73,949	11,321 [×6.53]
				tomont	11,520	3,586 [×3.21]
				reduce	20,506	7,182 [×2.86]
				Others		
	Encaps	1,470,234	509,435 [×2.88]	encrypt	1,270,285	431,364 [×2.94]
				gen-matrix	361,775	171,120 [×2.11]
				get-noise-η_1	60,470	26,319 [×2.3]
				get-noise-η_2	33,588	15,800 [×2.13]
				vec-NTT	120,784	34,701 [×3.48]
				basemul	73,949	11,322 [×6.53]
				vec-INVNTT	93,700	17,617 [×5.32]
				reduce	10,256	3,342 [×3.07]
				pack-cp	19,974	5,836 [×3.42]
				Others		
	Decaps	1,804,804	612,522 [×2.95]	decrypt	319,706	93,489 [×3.42]
				vec-INVNTT	93,700	17,617 [×5.32]
				vec-NTT	120,784	34,701 [×3.48]
				basemul	73,949	11,322 [×6.53]
				reduce	10,256	3,342 [×3.07]
				encrypt	1,270,285	431,364 [×2.94]
				hash-g	28,172	12,191 [×2.31]
				rkprf	171,480	75,587 [×2.27]
				Others		

Table 11: HQC-1 cycle breakdown.

Algorithm	Module	Baseline	Optimized	Sub-Modules	Baseline	Optimized
HQC-1	KeyGen	57,646,538	2,446,504 [×23.6]	prng-get-bytes	25,849	9,867 [×2.62]
				xof-get-bytes	25,848	9,866 [×2.62]
				PKE-KeyGen	57,573,236	2,405,365 [×23.9]
				hash-i	30,421	14,424 [×2.11]
				vec-set-random	469,657	197,963 [×2.37]
				vec-mul	56,146,314	1,346,026 [×41.7]
				Others		
	Encaps	115,081,750	4,689,421 [×24.5]	hash-g	31,330	15,321 [×2.04]
				encrypt	114,511,219	4,406,135 [×26]
				vec-mul	56,146,310	1,346,022 [×41.7]
				hqc-c-kem-tostring	495,230	223,521 [×2.22]
				code-encode	175,405	54,392 [×2.22]
				Others		
	Decaps	174,717,730	8,042,555 [×21.7]	decrypt	58,599,143	2,847,163 [×20.6]
				code-decode	1,974,237	1,054,352 [×1.87]
				code-encode	175,405	54,392 [×3.22]
				vec-mul	56,146,312	1,346,024 [41.7×]
				hash-h	491,684	219,957 [×2.24]
				encrypt	114,511,219	4,406,135 [×26]
				hash-j	953,129	425,599 [×2.24]
				Others		

Table 12: ML-DSA-2 cycle breakdown.

Algorithm	Module	Baseline	Optimized	Sub-Modules	Baseline	Optimized
ML-DSA-2	KeyGen	3,713,136	1,770,487 [×2.09]	shake	29,546	13,651 [×2.16]
				matrix-expand	2,334,148	1,048,028 [×2.23]
				polyvecl-uniform-η	192,464	96,850 [×1.99]
				polyveck-uniform-η	219,791	108,200 [× 2.03]
				vec-NTT	185,791	95,670 [×1.94]
				basemul	191,771	109,698 [×1.75]
				reduce	19,489	9,249 [×2.11]
				vec-INVNTT	223,551	112,961 [×1.98]
				Others		
	Sign	7,139,645	3,579,475 [×1.99]	matrix-expand	2,334,148	1,048,028 [×2.23]
				vec-NTT	185,791	95,670 [×1.94]
				polyvecl-uniform-η	192,464	96,850 [×1.99]
				basemul	191,771	109,698 [×1.75]
				reduce	19,489	9,249 [×2.11]
				vec-INVNTT	223,551	112,961 [×1.98]
				polyveck-uniform-γ	575,009	256,022 [× 2.25]
				Others		
	Verify	3,853,489	1,862,627 [×2.07]	shake	29,546	13,651 [×2.16]
				challenge	36,727	20,786 [×1.77]
				matrix-expand	2,334,148	1,048,028 [×2.23]
				vec-NTT	185,791	95,670 [×1.94]
				Others		

Table 13: SLH-DSA-128f cycle breakdown.

Algorithm	Module	Baseline	Optimized	Sub-Modules	Baseline	Optimized
SLH-DSA simple	KeyGen	148,919,704	76,565,638 [×1.94]	merkle-gen-root	148,919,667	76,565,601 [×1.94]
				Others		
	Sign	3,486,489,776	1,793,415,539 [×1.94]	gen-message	28,520	12,575 [×2.27]
				hash-message	29,624	13,694 [×2.16]
				fors-sign	210,172,414	108,817,238 [×1.93]
				merkle-sign	148,925,396	76,571,330 [×1.94]
				Others		
	Verify	204,805,226	105,411,582 [×1.94]	hash-message	29,624	13,694 [×2.16]
				fors-pk-from-sig	7,854,309	4,081,927 [×1.92]
				wots-pk-from-sig	8,908,953	4,593,541 [×1.94]
				compute-root	100,356	52,408 [×1.91]
				Others		
SLH-DSA-f robust	KeyGen	286,087,039	145,947,103 [×1.96]	merkle-gen-root	286,064,360	145,861,146 [×1.96]
				Others		
	Sign	6,642,087,257	3,387,032,456 [×1.96]	gen-message	28,520	12,548 [×2.27]
				hash-message	29,624	13,662 [×2.17]
				fors-sign	346,284,104	177,930,011 [×1.95]
				merkle-sign	286,170,091	145,866,940 [×1.96]
				Others		
	Verify	370,937,326	189,008,344 [×1.96]	hash-message	29,624	13,665 [×2.17]
				fors-pk-from-sig	15,471,570	7,949,277 [×1.95]
				wots-pk-from-sig	16,647,498	8,995,527 [×1.85]
				compute-root	197,839	102,027 [×1.94]
				Others		

This profiling is intended solely to highlight the relative weight of the most computationally intensive functions compared to the main routines of each algorithm (KeyGen, Encaps/Sign, Decaps/Verify) and to illustrate how HORCRUX — despite its compact area and limited number of custom instructions — provides a significant acceleration. The benefit becomes even clearer when analyzing individual subfunctions rather than the complete KeyGen, Encaps/Sign, or Decaps/Verify routines, as this perspective excludes non-optimized operations such as `memcpy`, which lie outside the scope of our hardware enhancements.

It is important to note that we tested only one security level for each algorithm, as the overall behavior and performance trends are expected to be similar across different security levels. This approach simplifies the evaluation without sacrificing the generality of the results. Moreover, all the reported cycle counts represent the mean values obtained over 100 simulation runs, ensuring that the results are statistically consistent and not affected by measurement variability. Finally, for operations that depend on the message length during signing and verification, we fixed the input size to the KAT setting `MLEN_KAT` = 33 bytes for every algorithm; the reported cycles therefore reflect this configuration. Larger messages would primarily increase the hashing-bound portions proportionally, while the core arithmetic and tree- or matrix-centric kernels remain essentially message-size-independent.