PQCUARK: A Scalar RISC-V ISA Extension for ML-KEM and ML-DSA

Xavier Carril[1], Alicia Manuel Pasoot[1], Emanuele Parisi[1], Carlos Andrés Lara-Niño[2], Oriol Farràs[2], Miquel Moretó[1]
[1]*Barcelona Supercomputing Center, Barcelona, Spain*
[2]*Universitat Rovira i Virgili, Tarragona, Spain*
xavier.carril@bsc.es

***Abstract*—Recent advances in quantum computing pose a threat to the security of digital communications, as large-scale quantum machines can break commonly used public-key cryptographic algorithms. To mitigate this risk, post-quantum cryptography (PQC) schemes are being standardized, with recent NIST recommendations selecting two lattice-based schemes, ML-KEM for key encapsulation and ML-DSA for digital signatures, alongside other schemes. Two computationally intensive kernels dominate the execution of these schemes: the Number-Theoretic Transform (NTT) for polynomial multiplication and the `Keccak-f1600` permutation function for polynomial sampling and hashing. This paper presents PQCUARK, a scalar RISC-V ISA extension that accelerates these key operations. PQCUARK integrates two novel accelerators within the core pipeline: (i) a packed SIMD butterfly unit capable of performing NTT butterfly operations on 2×32bit or 4×16bit polynomial coefficients, and (ii) a permutation engine that delivers two Keccak rounds per cycle, hosting a private state and a direct interface to the core Load Store Unit, eliminating the need for a custom register file interface. We have integrated PQCUARK into an RV64 core and deployed it on an FPGA. Experimental results demonstrate that PQCUARK provides up to 10.1× speedup over the NIST baselines and 2.3× over the optimized software, and it outperforms similar state-of-the-art approaches between 1.4-12.3× in performance. ASIC synthesis in GF22-FDSOI technology shows a moderate core area increase of 8% at 1.2 GHz, with PQCUARK units being outside the critical path.**

***Index Terms*—Post-Quantum Cryptography, Lattice-Based Cryptography, Instruction Set Extension, RISC-V**

I. Introduction

With the proliferation of embedded computing devices deployed across safety and security-critical domains, ensuring secure digital communication has become a fundamental requirement. For decades, public-key cryptography has served as the foundation of digital communication security, with schemes whose security rely on the hardness of the RSA and discrete logarithm problems. However, the advent of quantum computing poses an existential threat to these schemes, as quantum algorithms can efficiently solve the hard mathematical problems on which their security is based [1]. This imminent risk has driven a global effort to design post-quantum cryptographic (PQC) methods that are resistant against both quantum and classical computers. This effort has reached a significant milestone with the standardization of the PQC schemes: Module-Lattice based Key Encapsulation Mechanism (ML-KEM) and Module-Lattice based Digital Signature Algorithm (ML-DSA) [2], [3]. Benchmarking studies of these PQC schemes have identified two primary computational bottlenecks: the Number Theory Transform (NTT), used to multiply polynomials [4] efficiently, and the `Keccak-f1600` permutation, which is at the core of hashing and pseudo-random number generation routines [5]. Consequently, both NTT and `Keccak-f1600` have been identified as prime candidates for hardware acceleration, as their optimization can substantially reduce the latency of lattice-based PQC schemes [6], [7].

This work introduces **P**ost-**Q**uantum **C**ryptography **U**nit for **A**rithmetic **R**ISC-V C(**K**)rystals (PQCUARK), a slim scalar RISC-V ISA extension that targets the computational bottlenecks of ML-KEM and ML-DSA. PQCUARK consists of two tightly coupled accelerators: a fully pipelined Butterfly Unit (BFU) for the NTT and a Keccak unit. The BFU uses a packed-SIMD datapath that can operate on either 2×32-bit or 4×16-bit polynomial coefficients, allowing it to fully benefit from 64-bit registers while exploiting the arithmetic properties of ML-KEM polynomials, whose coefficients are 16-bit wide. Moreover, its pipelined design enables multiple modular multiplications to overlap in time without affecting the processor's critical path. The Keccak unit contains a series of combinatorial blocks, each of which implements a single Keccak round with a private state decoupled from the register file, eliminating the need for a custom accelerator-register file interface in application-class pipelines with register renaming, where custom logic keeps track of the association between physical and architectural registers. Additionally, the Keccak unit interfaces directly with the Load-Store Unit and reuses the core's external data channel width, which is potentially larger than the architectural register width. It also leverages the core's memory translation hardware, allowing for virtual memory access without requiring a dedicated MMU. In summary, we

Xavier Carril is supported by the predoctoral programme AGAUR-FI Joan Oró grant (2024 FI-1 00520), funded by the Generalitat de Catalunya (Department of Research and Universities) and the European Social Fund Plus. Emanuele Parisi is supported by AI4S fellowships from the "Generación D" initiative (Red.es, Ministerio para la Transformación Digital y de la Función Pública, C005/24-ED CV1), funded by EU NextGenerationEU funds through PRTR, partially funded by Generalitat de Catalunya [2021-SGR-00763], and by Spanish MCIU/AEI project PID2023-146511NB-I00 co-funded by EU ERDF. O. Farràs and C.A. Lara-Nino are supported by project HERMES funded by European Union NextGenerationEU/PRTR via INCIBE and the Spanish Ministry of Science and Innovation MCIN/AEI 10.13039/501100011033 (Grants ACITHEC PID2021-124928NB-I00, MATSE PID2024-156636NB-C22). This work is part of the project PID2023-146511NB-I00 funded by the Spanish Ministry of Science, Innovation and Universities MCIU /AEI /10.13039/501100011033 and EU ERDF.

TABLE I: Security and key or signature/ciphertext sizes for ML-KEM and ML-DSA according to FIPS-203 and FIPS-204

Algorithm	NIST Category	Sizes (in bytes)		
		Public key	Private key	Ciphertext/ Signature
ML-KEM-512	1	800	1632	768
ML-KEM-768	3	1184	2400	1088
ML-KEM-1024	5	1568	3168	1568
ML-DSA-44	2	1312	2560	2420
ML-DSA-65	3	1952	4032	3309
ML-DSA-87	5	2592	4896	4627

propose the following contributions:

- **Design.** We present PQCUARK, a scalar RISC-V ISA extension for ML-KEM and ML-DSA, featuring two accelerators: a packed-SIMD BFU and a Keccak unit.
- **Integration.** We integrate the designed accelerators into the Sargantana [8] RISC-V core and deploy the system on FPGA.
- **Characterization.** PQCUARK provides up to 10.1× speedup over the NIST baselines [2], [3] and 2.3× over the optimized software [7] with a 8% area increase for GF22-FDSOI technology at 1.2GHz. It outperforms the runtime of similar approaches [9], [10] by up to 1.4-12.3×.

To promote reproducibility and adoption of our research, we provide an open-source release of PQCUARK[1].

II. Background

A. Module Lattice PQC Schemes

Both ML-KEM and ML-DSA are defined on the polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$, where n is a power of 2 ($n = 256$) and q is a small prime satisfying $q \equiv 1 \mod 2n$ ($q = 3329$ for ML-KEM and $q = 8380417$ for ML-DSA). This means that their operands can be represented as polynomials of degree up to $n-1$ with $\log_2 q$-bit coefficients. The underlying operations of ML-KEM and ML-DSA include multiplications between these operands, which can be solved as polynomial products with complexity $O(n^2)$. However, given the special relationship between q and n, it is possible to use the NTT to reduce this complexity to $O(n \log n)$. In this case, applying the NTT to a polynomial product is similar to using the FFT to a convolution. The other computationally expensive operations of ML-KEM and ML-DSA involve what is called $sampling$. Their security claims primarily rely on the randomness quality for the selection of polynomials from the vector space. This process must follow the appropriate distributions used in the security proofs. In practice, this is achieved through repeated calls to a pseudo-random function, such as the Keccak permutation. Therefore, optimizing this operation is the second great challenge of these algorithms. Table I summarizes the security categories and key or ciphertext/signature sizes for the standardized variants of ML-KEM and ML-DSA.

[1]https://github.com/bsc-loca/PQCUARK

B. Scalar ISA Extensions for ML-KEM and ML-DSA

Since the submission of the first lattice-based PQC schemes for standardization, several works have proposed RISC-V scalar ISA extensions to support their execution. There exist two major families of approaches, depending on the computational primitives selected for optimization.

The first approach focuses on minimizing the resource utilization of the ISA extension by optimizing only the polynomial multiplication [10]–[12]. Alkim et al. [11] enhance the RV32 VexRiscv microcontroller with four instructions to accelerate finite field arithmetic operations. Nannipieri et al. [10] enhance the CVA6 RV64 core architecture with a custom post-quantum arithmetic unit, introducing novel opcodes to compute the butterfly operations for forward and inverse NTT. Li et al. [12] replace the traditional Montgomery and Barrett reductions in the butterfly unit with the k^2-red reduction method, demonstrating lower resource utilization. These approaches result in low-overhead extensions, but not tackling the acceleration of Keccak permutations limits the maximum performance gain achievable on end-to-end ML-KEM and ML-DSA workloads.

The second approach privileges performance gain in the complete ML-KEM and ML-DSA workload, implementing novel instructions to optimize both polynomial multiplication and Keccak permutations [9], [13], [14]. Fritzmann et al. [9] propose a comprehensive ISA extension that includes custom opcodes to accelerate Keccak permutations, polynomial sampling, and polynomial multiplication. The proposed implementation enhances the RI5CY [15], a 32-bit RISC-V core targeting low-power microcontroller-class applications, introducing novel NTT, Keccak, and polynomial sampling units between the decode and execute stages. In this work, the maximum frequency reachable by the system is limited by the NTT accelerator, since the authors do not adopt any pipelining in its architecture. The Keccak unit executes a single round of the Keccak permutation function using the floating-point register files and a fraction of the integer register files to store the pseudo-random function state between permutations. Lee et al. [13] implement a tightly-coupled cryptographic co-processor that accelerates Montgomery reduction (a computational step in NTT) and certain bitwise operations in the `Keccak-f1600` functions. Although it is more lightweight than alternative approaches, the solution proposed in [13] has a relatively low performance boost compared to all other methods because it optimizes only some of the computations involved in the NTT and Keccak permutations. Furthermore, the authors do not present any experimental results concerning the end-to-end ML-KEM and ML-DSA workloads, which makes comparison challenging. Dolmeta et al. [14] propose an interesting hybrid approach, where two loosely-coupled memory-mapped accelerators support NTT and Keccak execution. In contrast, a slim ISA extension supports Montgomery and Barrett reductions and polynomial sampling. While hybrid approaches are appealing, we focus on scalar-ISA-extension approaches, which rely on tightly-coupled accelerators [9], [10] to avoid the inconvenience of handling memory translation for out-of-core accelerators in application-class processors.

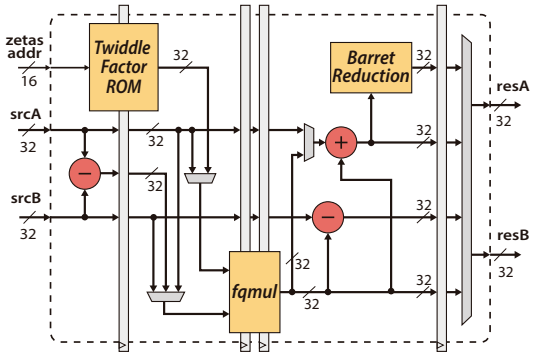

Fig. 1: Architecture of the Butterfly Unit (BFU)

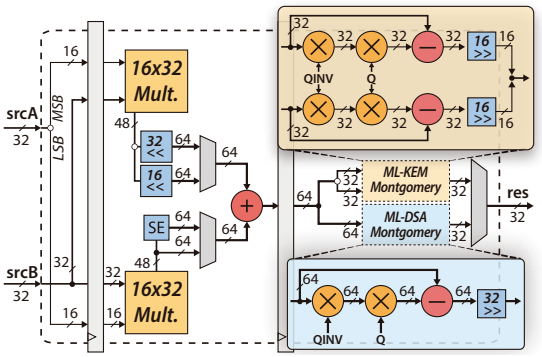

Fig. 2: Montgomery Reduction Multiplication (`fqmul`)

C. Software Implementations

We compare the PQCUARK performance enhancement against two software implementations of ML-KEM and ML-DSA, which are freely available. First, we consider the reference code distributed as part of the NIST submission package [2], [3]. We also consider PQRV [7], a recently proposed set of hand-optimized RISC-V assembly implementations of the NTT and Keccak permutation routines. To the best of our knowledge, this constitutes the highest-performing software implementation of ML-KEM and ML-DSA to date. As PQCUARK implements a scalar ISA extension, the vectorized PQRV routines are beyond the scope of this work.

III. Number Theoretic Transform Extension

To accelerate the computation of the NTT, we adopt a synergistic approach that combines hardware-level enhancements with algorithmic optimizations. At the hardware level, we introduce a tightly coupled BFU, optimized explicitly for high-throughput modular arithmetic. This unit serves as the core accelerator for forward and inverse NTT operations, as well as for modular polynomial multiplications. The BFU achieves reduced latency and increased throughput through pipelining and the parallel execution of two butterfly operations for 16-bit ML-KEM coefficients. On the algorithmic side, we build upon the Montgomery-based implementation by Zhang et al. [7]. Our extension implements the 4+3 layer-merging strategy for ML-KEM and the 4+4 layer-merging strategy for ML-DSA, thereby reducing the number of memory accesses required throughout the computation of NTT and inverse NTT (INTT).

A. Butterfly Unit Design

We engineered the BFU to efficiently execute the core modular operations required by the NTT and the INTT. As depicted in Figure 1, the BFU comprises dedicated arithmetic submodules capable of performing modular additions, multiplications, and reductions. It supports both Cooley–Tukey (CT) and Gentleman–Sande (GS) butterfly operations, corresponding to NTT and INTT computations, respectively. The tight integration with the processor pipeline enables the BFU to issue multiple such operations with minimal control overhead.

Drawing inspiration from the design methodology of [16], the BFU supports parallel execution of two butterfly operations per instruction for 16-bit ML-KEM polynomial coefficients, and single-operation execution for 32-bit ML-DSA polynomial coefficients. The Montgomery reduction multiplication (`fqmul`, Figure 2) employs two dedicated 16x32-bit multipliers. Each operation consumes one NTT operand and a precomputed twiddle factor fetched from a small ROM. The partial products are aligned via bit-level shifts and sign extensions, merged, and then forwarded to the Montgomery reduction stage. The `fqmul` datapath adapts per scheme: ML-KEM processes two 32-bit coefficient products in parallel, whereas ML-DSA handles a single 64-bit coefficient product. The reduction uses the fixed modulus (`Q`) and its precomputed inverse (`QINV`), implementing an efficient shift-and-add Montgomery multiplication reduction procedure.

The modular adders and subtractors (in red in Figures 1 and 2) are optimized following the same principles as in [16], ensuring constant-time execution and resistance to timing side-channel attacks. The BFU achieves a one-cycle throughput and a four-cycle latency for butterfly operations, operating at a peak frequency of 2 GHz in GF22-FDSOI technology. Additionally, the BFU supports standalone modular multiplication for both 16-bit and 32-bit operands, making it suitable for both base and point-wise polynomial multiplications. This design enables unified hardware support for both ML-KEM and ML-DSA.

B. Extended ISA Implementation for NTT

To exploit the capabilities of the BFU, we extend the RISC-V ISA with custom instructions for modular arithmetic, butterfly operations, and data packing and unpacking to support efficient BFU input and output handling.

- [**`pqcuark.bfnttk`**, **`pqcuark.bfinttk`**]: Implement two parallel butterfly operations for the forward (CT) and inverse (GS) NTT, respectively. Each 64-bit `rs1` register encodes four 16-bit input coefficients (two per operation). The `rs2` register acts as an index to memory-resident twiddle factors (up to 128 values). The two 2x16-bit butterfly results are stored in the destination register `rd`.
- [**`pqcuark.bfnttd`**, **`pqcuark.bfinttd`**]: Execute a single butterfly operation on 32-bit coefficients for the forward and inverse NTT, respectively. The 64-bit `rs1` register holds two 32-bit operands, and `rs2` indexes up to 256

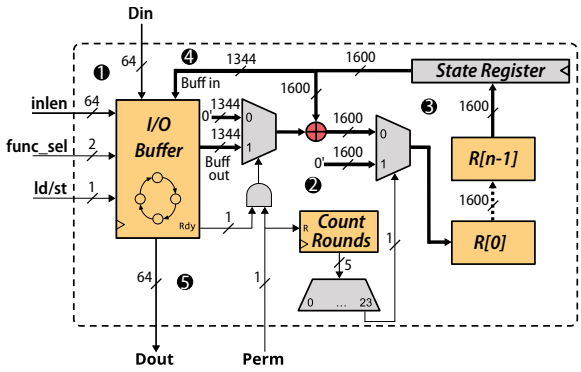

Fig. 3: Architecture of the Keccak unit

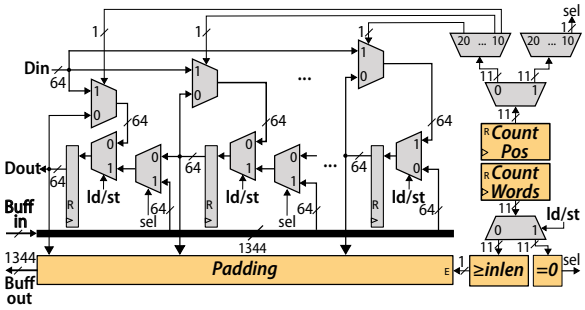

Fig. 4: Keccak unit I/O buffer

twiddle factors. The 2x32-bit butterfly result is stored in the destination register `rd`.

- [**`pqcuark.fqmul16.l`**, **`pqcuark.fqmul16.h`**]: Perform two parallel modular multiplications on 16-bit coefficients. The suffixes `l` and `h` determine whether the operands are extracted from the lower or upper halves of the source registers, respectively. The resulting 2×16-bit product is stored in the corresponding half of the `rd` register.
- [**`pqcuark.fqmul32.l`**, **`pqcuark.fqmul32.h`**]: Execute a single modular multiplication on 32-bit coefficients. As above, the `l` and `h` suffixes determine operand extraction and result placement within `rd`.
- **`pqcuark.packu`**: An extension of the `pack` bit manipulation instruction included in the RISC-V Bit-Manipulation (`B`) extension. It packs the 32-bit most significant bits of `rs1` and `rs2` into `rd` to pack data before feeding it into the BFU.

IV. Keccak Permutation Extension

The Keccak algorithm underlies the various hashing and polynomial sampling algorithms employed in ML-KEM and ML-DSA, including SHAKE, SHA-3, binomial, and uniform sampling. Keccak is based on the sponge construction, in application of the `Keccak-f1600` permutation and multirate padding. For SHA-3, the construction initially resets the 1600-bit permutation state register to zero ($r + c$ bits), implements the padding rule on the input string, and subsequently divides it into blocks of r bits, where the exact values of r and c are specified by the FIPS202 [5] standard, depending on the function to execute. The sponge construction then proceeds in two phases: (i) absorb, and (ii) squeeze. In the absorb phase, each input block is XORed with the first r bits of the state, followed by the multirate padding and the application of the `Keccak-f1600` permutation. Once all blocks are absorbed, the squeeze phase begins, where the algorithm outputs r-bit blocks from the state, applying the permutation between each output step. The process finishes when the `Keccak-f1600` function produces the desired number of bytes.

While alternative solutions in the literature accelerate Keccak by integrating a loosely-coupled accelerator in the system-on-chip [14], this approach introduces hardware complexity because it requires a custom memory translation mechanism for the accelerator to access virtual memory [17]. To address this, we integrate our Keccak acceleration unit within the core's pipeline and we implement a custom I/O buffer, capable of interfacing the core Load Store Unit and autonomously handling r-bit blocks loading and storing, exploiting the capabilities of the core memory interface.

A. Keccak Permutation Unit Design

The design of the Keccak unit is inspired by the high-speed core architecture presented by the Keccak Team[2]. We have enhanced the original design to integrate the permutation engine into the core's pipeline, interfacing it with the Load Store Unit for data gathering and with the pipeline core circuitry to handle data dependencies. As shown in Figure 3, the accelerator includes an I/O buffer, a state register, a round counter, and a sequence of n round modules (in this case $n = 2$), each performing one of the 24 permutation rounds required by `Keccak-f1600`. The module begins in ❶ by reading `inlen` and `func_sel` from two novel control and status registers (CSR), `CSR_KECCAK_INLEN` and `CSR_KECCAK_FSEL`, which specify the length of the input message and the function to execute. Additionally, the `ld/st` signals place the I/O buffer in either load or store mode. The I/O buffer, depicted in Figure 4, is implemented with 64-bit registers sized to the maximum processing rate among the four algorithms, which is 1344 bits. Data transfers occur in 64-bit blocks. Two counters coordinate this process: (*Count Pos*) determines the position of each block within the buffer during load and store operations, resetting upon reaching the buffer size; (*Count Words*) tracks the number of bits transferred. In input mode, the Keccak unit pads the input message when the counter reaches the input length, and the result is written to *Buff out*. In output mode, the output result *Buff in* is registered when the counter is zero; otherwise, all registers act as a shift register to return the data in 64-bit blocks through `Dout`. Both counters handle the case when the input data word size is not a multiple of 64 bits:

$$count \mathrel{+}= \begin{cases} |inlen - count_words| & \text{if } count < 64, \\ 64 & \text{if } count \geq 64. \end{cases} \quad (1)$$

In ❷, the *ready* signal is asserted to integrate the input block into the state. During both absorption and squeezing, the *perm* signal activates the round counter and the bank

[2]https://keccak.team/index.html

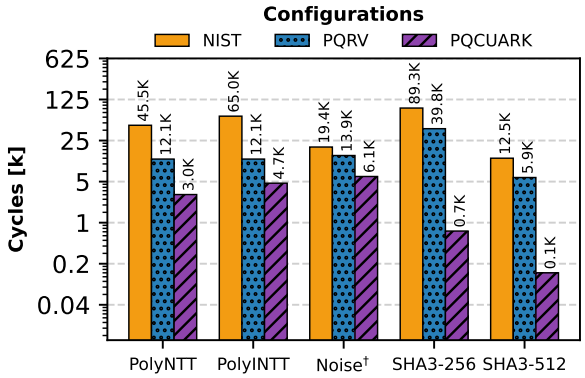

Fig. 5: ML-KEM-768 relevant routines extracted from NIST [2], PQRV [7] and PQCUARK simulation executions.†Refers to `poly_getnoise_eta2`.

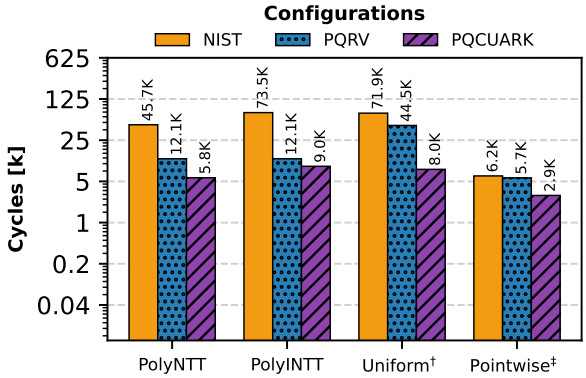

Fig. 6: ML-DSA-65 relevant routines extracted from NIST [3], PQRV [7] and PQCUARK simulation executions.†Refers to `poly_uniform`.‡Refers to `poly_pointwise`

of combinational round modules. In ❸, each combinational module `R[i]` computes one round per cycle, with the counter stepping by n. Then, a 1600-bit register stores intermediate results. The input to the combinational logic varies by phase: during absorption, it is the XOR of the state and input block; during squeezing, it is the saved state from the register. Finally, in ❹ and ❺, the resulting state is buffered and written to memory in 64-bit words. The Keccak unit achieves a maximum peak frequency of 2 GHz in GF22-FDSOI technology by using two combinatorial round modules `R[i]`. Thus, each `Keccak-f1600` permutation has a latency of 12 cycles.

B. Extended ISA Implementation for Keccak

We implement dedicated instructions to handle state initialization, input loading, output storing, and state permutation.

- **`pqcuark.keccak.init`:** Reset the Keccak state and initialize the accelerator function selection and input length.
- **`pqcuark.keccak.ld`:** Receive the address of a 64-bit data chunk through `rs1` and load it into the I/O buffer.
- **`pqcuark.keccak.f1600`:** Perform 24 rounds of the `Keccak-f1600` permutation.
- **`pqcuark.keccak.st`:** Store a 64-bit word from the I/O buffer into the memory address indicated in `rs1`.

V. Experimental Results

The BFU, along with the Keccak accelerator, creates the PQCUARK functional unit, which is integrated as a tightly coupled accelerator within the execute stage of the Sargantana RV64 core [8]. Sargantana is an in-order RISC-V single-core processor that features a 7-stage pipeline and non-blocking caches, consisting of a 32 KB L1 cache and a 512 KB L2 cache. The overall design has been deployed on the AMD-Xilinx U55C FPGA platform for emulation. Moreover, we present ASIC results running logic synthesis with Cadence Genus v21.15 targeting GF22-FDSOI technology.

A. Performance Results

Table II shows the execution results of ML-KEM and ML-DSA across security levels. All results correspond to the following software running on Sargantana [8]: the NIST baseline, the PQRV software optimized version, and PQCUARK ISA extension. Runtime is estimated at 1.2 GHz ASIC, and speedups are measured against the NIST baseline. For ML-DSA experiments, signatures are generated over 64-byte messages using a fixed seed to remove randomness and ensure deterministic results. Results show that PQCUARK consistently achieves 4.3–5.2× speedups for ML-KEM and 4.3×–10.1× for ML-DSA, while sustaining similar IPC values across different operations and security levels. Moreover, the PQCUARK extension delivers approximately double the performance of PQRV [7], the most optimized software implementation of ML-KEM and ML-DSA in the state-of-the-art. Figures 5 and 6 break down the execution of ML-KEM-768 and ML-DSA-65 with the most computationally demanding directives. The PQCUARK consistently reduces the cycle counts compared to both the NIST baselines [2], [3] and the PQRV optimized software [7], with the most significant gains in NTT, sampling, and hashing.

B. Area and Power Consumption Results

The FPGA post-implementation resource utilization shows that the Sargantana core requires 327.8K LUTs, 95.9K FFs, 25 DSPs, and no BRAMs. The PQCUARK extension adds 129.9K LUTs (39.6% overhead), 5.76K FFs (6%), and no additional DSPs or BRAMs. Concerning the ASIC synthesis results, Table III reports the ASIC synthesis results in GF22-FDSOI at 1.2 GHz. In the presented module hierarchy, the first row covers the Sargantana core pipeline together with the L1 instruction and data caches. The PQCUARK extension accounts for 8% of the final core area (0.22 mm^2, 1120 kGE), which corresponds an area overhead of 8.7% over the core area and 34.4% over the datapath. This represents an acceptable cost for application-class cores compared to the execution benefits shown in Table II. Cadence Genus also reports 55.2 mW of static power for PQCUARK, which corresponds to 9.13% of the final core power. This translates into a 10.05% power overhead over the baseline core and a 47.6% overhead over the datapath.

C. State-of-the-Art Comparison

We compare the performance gains of PQCUARK with two relevant alternative scalar ISA extensions in the state-of-the-art:

TABLE II: PQCUARK performance on ML-KEM and ML-DSA vs. software baselines, executed on Sargantana RV64IMB

Algorithm		NIST [2], [3]			PQRV [7]				**PQCUARK**			
		Cycles [$\times 10^6$]	IPC	Time [ms]	Cycles [$\times 10^6$]	IPC	Time [ms]	Speedup [×]	Cycles [$\times 10^6$]	IPC	Time [ms]	Speedup [×]
ML-KEM-512	KeyGen	0.499	0.73	0.599	0.220	0.92	0.263	2.3	0.114	0.82	0.136	**4.4**
	Encapsulate	0.615	0.76	0.738	0.274	0.90	0.329	2.2	0.120	0.79	0.144	**5.1**
	Decapsulate	0.840	0.74	1.007	0.302	0.89	0.363	2.8	0.163	0.79	0.195	**5.2**
ML-KEM-768	KeyGen	0.845	0.73	1.015	0.364	0.92	0.437	2.3	0.196	0.81	0.235	**4.3**
	Encapsulate	0.980	0.76	1.176	0.434	0.91	0.521	2.3	0.196	0.80	0.235	**5.0**
	Decapsulate	1.281	0.74	1.538	0.468	0.90	0.561	2.7	0.250	0.80	0.301	**5.1**
ML-KEM-1024	KeyGen	1.264	0.75	1.517	0.560	0.92	0.672	2.3	0.297	0.81	0.356	**4.3**
	Encapsulate	1.465	0.75	1.758	0.642	0.92	0.770	2.3	0.292	0.81	0.351	**5.0**
	Decapsulate	1.807	0.75	2.169	0.684	0.90	0.820	2.6	0.359	0.80	0.431	**5.0**
ML-DSA-44	KeyGen	1.438	0.72	1.141	0.784	0.85	0.622	1.8	0.336	0.70	0.267	**4.3**
	Sign	5.002	0.65	3.969	2.290	0.79	1.817	2.2	0.910	0.69	0.722	**5.5**
	Verify	1.694	0.69	1.344	0.832	0.85	0.660	2.0	0.363	0.70	0.288	**4.7**
ML-DSA-65	KeyGen	2.506	0.70	1.989	1.337	0.86	1.061	1.9	0.549	0.69	0.436	**4.6**
	Sign	9.752	0.65	7.740	4.042	0.79	3.208	2.4	0.966	0.70	0.767	**10.1**
	Verify	2.611	0.72	2.073	1.340	0.86	1.064	1.9	0.559	0.71	0.443	**4.7**
ML-DSA-87	KeyGen	4.026	0.72	3.195	2.175	0.88	1.726	1.9	0.846	0.73	0.671	**4.8**
	Sign	12.648	0.67	10.038	4.894	0.81	3.884	2.6	1.436	0.71	1.139	**8.8**
	Verify	4.276	0.72	3.394	2.210	0.88	1.754	1.9	0.877	0.73	0.696	**4.9**

TABLE III: Synthesis results for GF22-FDSOI at 1.2GHz

Hierarchy	Area			$P_{sta.}$
	[%]	[mm^2]	[kGE]	[mW]
`Sargantana`	100.00	2.74	13712	604.6
`└─ Datapath`	31.39	0.86	4309	171.3
`└─ PQCUARK`	**8.03**	**0.22**	**1120**	**55.2**
`├─ Keccak`	7.66	0.21	1028	53.2
`└─ BFU`	0.73	0.02	92	2.1

TABLE IV: Comparison with the State-of-the-Art

Algorithm		PQCUARK	[9]		[10]*	
		[ms]	[ms]	[×]	[ms]	[×]
ML-KEM 768	KeyGen	**0.235**	0.342	1.5	0.643	2.7
	Encaps.	**0.235**	0.407	1.7	0.677	2.9
	Decaps.	**0.301**	0.426	1.4	0.798	2.7
ML-DSA 65	KeyGen	**0.436**	—	—	2.755	6.3
	Signature	**0.767**	—	—	9.455	12.3
	Verify	**0.443**	—	—	2.744	6.2

Nannipieri et al. [10], which integrates a custom post-quantum arithmetic unit into the CVA6 RV64 core, and Fritzmann et al. [9], which extends the RI5CY RV32 core with instructions for Keccak, binomial and random sampling, and polynomial multiplication. To ensure fairness, we synthesized both CVA6 and RI5CY with Cadence Genus using the same GF22-FDSOI technology node and synthesis script as PQCUARK. The maximum achievable clock frequency for RISQ-V [9] is 800 MHz, limited by critical paths through the PQC arithmetic units introduced to optimize NTT execution. The highest frequency for [10] is 1080 MHz. Notice that the authors of [10] do not release their source code, so we assume their ISA extension does not impact the top frequency of CVA6. Based on the cycle counts reported by the authors for ML-KEM and ML-DSA at their mid-level security categories, and scaling them to these frequencies, we derived the execution latencies summarized in Table IV. Note that [10] reports encryption and decryption instead of encapsulation and decapsulation. We approximate the results of encapsulation to encryption, and decapsulation as encryption plus decryption, following the pseudocode presented in [2], neglecting the runtime contribution of Keccak routines. Table IV shows that PQCUARK achieves lower runtime across all ML-KEM operations, improving over [9] by 1.4–1.7× and reaching 2.7–2.9× speedups over [10]. For ML-DSA, PQCUARK reduces key generation and verification latency by over 6× and signing latency by 12.3× compared to [10].

VI. Conclusions

We introduce PQCUARK, a scalar RISC-V ISA extension designed to accelerate NTT and Keccak on RV64 cores. PQCUARK integrates a packed-SIMD BFU and a Keccak engine in the Sargantana RISC-V core, delivering 4.3–10.1× end-to-end speedups over the NIST baseline at the cost of around 8% area increase at 1.2 GHz. We outperform prior approaches, with improvements up to 2.9× for ML-KEM and 12.3× for ML-DSA. Future work will focus on extending PQCUARK to provide vector-based acceleration, in line with trends followed by the RVA23 profile ratification and the approach followed by the standard extensions for cryptography.

REFERENCES

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] NIST, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," Standard FIPS PUB 203, National Institute of Standards and Technology, 2024.

[3] NIST, "Module-Lattice-Based Digital Signature Standard," Standard FIPS PUB 204, National Institute of Standards and Technology, 2024.

[4] A. Satriawan *et al.*, "Conceptual Review on Number Theoretic Transform and Comprehensive Review on Its Implementations," *IEEE Access*, vol. 11, pp. 70288–70316, 2023.

[5] NIST, "SHA-3 Standard : Permutation-Based Hash and Extendable-Output Functions," Standard FIPS PUB 202, National Institute of Standards and Technology, 2015.

[6] Y. Kim *et al.*, "Vectorized Implementation of Kyber and Dilithium on 32-bit Cortex-A Series," *IEEE Access*, vol. 12, pp. 104414–104428, 2024.

[7] J. Zhang *et al.*, "Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV32,64IMBV," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, p. 632–655, Dec. 2024.

[8] M. Doblas *et al.*, "Sargantana: An Academic SoC RISC-V Processor in 22nm FDSOI Technology," in *2023 38th Conference on Design of Circuits and Integrated Systems (DCIS)*, pp. 1–6, 2023.

[9] T. Fritzmann *et al.*, "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, p. 239–280, Aug. 2020.

[10] P. Nannipieri *et al.*, "A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms," *IEEE Access*, vol. 9, pp. 150798–150808, 2021.

[11] E. Alkim *et al.*, "ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, p. 219–242, Jun. 2020.

[12] L. Li *et al.*, "Compact Instruction Set Extensions for Kyber," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 3, pp. 756–760, 2024.

[13] J. Lee *et al.*, "A Programmable Crypto-Processor for National Institute of Standards and Technology Post-Quantum Cryptography Standardization Based on the RISC-V Architecture," *Sensors*, vol. 23, no. 23, 2023.

[14] A. Dolmeta *et al.*, "ATHOS: A Hybrid Accelerator for PQC CRYSTALS-Algorithms Exploiting New CV-X-IF Interface," *IEEE Access*, vol. 12, pp. 182340–182352, 2024.

[15] M. Gautschi *et al.*, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[16] A. Aikata *et al.*, "KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 2, pp. 747–758, 2023.

[17] T. Wei *et al.*, "Cohort: Software-Oriented Acceleration for Heterogeneous SoCs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, (New York, NY, USA), p. 105–117, Association for Computing Machinery, 2023.