IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 632–655. DOI:10.46586/tches.v2025.i1.632-655

Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}

Jipeng Zhang[1], Yuxing Yan[2], Junhao Huang[3,4], Çetin Kaya Koç[1,5,6]

[1] Nanjing University of Aeronautics and Astronautics, Nanjing, China jp-zhang@outlook.com
[2] Shanghai Aerospace Electronic Technology Institute, Shanghai, China yanyuxing7408@163.com
[3] BNU-HKBU United International College, Zhuhai, China huangjunhao@uic.edu.cn
[4] Hong Kong Baptist University, Hong Kong, China
[5] Iğdır University, Iğdır, Turkey
[6] University of California Santa Barbara, Santa Barbara, USA cetinkoc@ucsb.edu

Abstract. With the standardization of NIST post-quantum cryptographic (PQC) schemes, optimizing these PQC schemes across various platforms presents significant research value. While most existing software implementation efforts have concentrated on ARM platforms, research on PQC implementations utilizing various RISC-V instruction set architectures (ISAs) remains limited. In light of this gap, this paper proposes comprehensive and efficient optimizations of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. We thoroughly optimize these implementations for dual-issue CPUs, believing that our work on various RISC-V ISAs will provide valuable insights for future PQC deployments.

Specifically, for Keccak, we revisit a range of optimization techniques, including bit interleaving, lane complementing, in-place processing, and hybrid vector/scalar implementations. We construct an optimal combination of methods aimed at achieving peak performance on dual-issue CPUs for various RISC-V ISAs. For the NTT implementations of Kyber and Dilithium, we deliver optimized solutions based on Plantard and Montgomery arithmetic for diverse RISC-V ISAs, incorporating extensive dual-issue enhancements. Additionally, we improve the signed Plantard multiplication algorithm proposed by Akoi et al. Ultimately, our testing demonstrates that our implementations of Keccak and NTT across various ISAs achieve new performance records. More importantly, they significantly enrich the PQC software ecosystem for RISC-V.

Keywords: SHA-3 · Keccak · Kyber · Dilithium · RISC-V · RISC-V Vector · Plantard Arithmetic · NTT

1 Introduction

In August 2024, NIST released three PQC standards: FIPS 203 [NIS24a], FIPS 204 [NIS24b], and FIPS 205 [NIS24c]. These standards respectively standardize ML-KEM, ML-DSA, and SLH-DSA, which are based on Kyber [ABD+21], Dilithium [DKL+21], and SPHINCS+ [ABB+22]. The deployment of these PQC schemes across various platforms holds significant research value.

Kyber and Dilithium are two popular Lattice-based Cryptographic (LBC) schemes mainly because of their modular design, high security, and excellent performance. The time-consuming operations of Kyber and Dilithium are SHA-3 [NIS15] and polynomial multiplication. Because both of them are operated on the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$, in which their moduli q and the degree n are carefully designed so that they can leverage the efficient Number Theoretic Transform (NTT) with $O(n \log n)$ time complexity to speed up the polynomial multiplication. After years of research by the

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

cryptographic community, polynomial multiplication has been throughout optimized across various platforms and architectures, especially AVX2 [Sei18], ARM Cortex-M [ABCG20, CHK+21, GKS20, ACC+22, AHKS22, HZZ+22, HZZ+24, HAZ+24], and ARM Cortex-A platforms [BHK+22, KSYS22, NG23]. Despite these optimizations focusing on the polynomial multiplication, another core operation, SHA-3, in LBC schemes can account for over 70% of the running time for an optimized implementation on ARM Cortex-M4, according to PQM4 [KRSS19]. Therefore, any speed-ups in SHA-3 would bring significant performance improvements to the entire LBC scheme. Recently, several works have optimized SHA-3 on ARMv8-A [BK22] and ARMv7-M [HAZ+24] architectures, significantly improving the efficiency of hash-based SPHINCS+ and LBC schemes like Kyber and Dilithium.

1.1 Motivations and Contributions

The motivations of the paper can be summarized as follows.

Limited Software Implementations on RISC-V. Despite the numerous PQC implementations available on various platforms, software implementations of Keccak and other PQC schemes on RISC-V have not received as much attention as those on ARM platforms. To our knowledge, there are only two publicly available Keccak implementations for RISC-V: one by [Sto19] for RV32I and another open-source implementation for RV64IB[1]. Regarding PQC implementations on RISC-V, previous works [ZHLR22, HZZ+24] primarily focus on low-end 32-bit RISC-V platforms with RV32IMAC ISAs.

Hardware Implementations Lack Efficient Baseline. The flexibility and customization of RISC-V instruction extensions have spurred research on PQC accelerators [AEL+20, FSS20, YSZ+24, WJW+19, FSMG+19, BUC19, WTJ+20, KSFS24] from a hardware perspective. However, the absence of optimized PQC software implementations on RISC-V means that many existing hardware designs primarily rely on the C reference implementation as their evaluation baseline. We believe that providing optimized PQC software implementations for a wide range of RISC-V ISAs will effectively fill this gap and offer a more reliable baseline for evaluating hardware implementations.

Challenges of Optimizing PQC for Different RISC-V ISAs. Achieving this goal, however, is non-trivial. The RISC-V ISA is modular, with each instruction set supporting specific operations. Consequently, different ISA combinations lead to significant variations in optimization strategies for cryptographic primitives. For example, the Keccak implementation on RV{32,64}I{B}{V} presents eight distinct ISA combinations, each potentially requiring a unique optimization strategy.

Contributions. Facing these challenges, we propose efficient software implementations of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. By proposing different optimization strategies for various ISA combinations, our implementations cater to a range of devices from low-end IoT to high-end mobile devices.

- **Optimized Keccak implementations.** We thoroughly revisit various optimization techniques for Keccak and construct an optimal combination of methods aimed at achieving peak performance on the dual-issue XuanTie C908 core. Our implementations cover four scalar instruction sets, specifically RV{32,64}I{B}. Additionally, we explore the use of vector extension (RVV) to optimize Keccak and successfully apply hybrid vector/scalar techniques to C908 RV32I{B}V. Our implementations set new performance records for Keccak across various ISAs. More importantly, our comprehensive support for different ISAs enhances the software ecosystem of Keccak on the RISC-V architecture.

[1] https://github.com/riscv/riscv-crypto/blob/main/benchmarks/sha3/zscrypto_rv64/Keccak.c

- **Optimized NTT implementations for Kyber and Dilithium.** Different ISAs require distinct optimization techniques for NTT implementations, especially for the time-consuming modular multiplication algorithms within NTT. Our target ISAs include RV{32,64}IM and RVV. We explore how to achieve optimal performance for Kyber's and Dilithium's NTT on the dual-issue XuanTie C908 core across these ISAs. Our implementations not only benefit the PQC software ecosystem but also serve as benchmarks for RISC-V-based hardware implementations.
- We integrate optimized Keccak and NTT implementations into Kyber and Dilithium, setting new performance records. The contributions of this paper extend beyond LBC schemes, providing value for hash-based signature schemes that rely on Keccak.
- As a secondary outcome, we also improve the signed Plantard multiplication algorithm proposed by Akoi et al. [AMOT22]. Although its practicality in LBC schemes is less than the variant proposed by Huang et al. [HZZ+22], we present it here for potential future research or applications.

Our implementations are publicly available at https://github.com/Ji-Peng/PQRV/tree/ches2025 under the MIT license.

2 Preliminaries

In this section, we give a brief introduction of Kyber, Dilithium, Keccak, and the target platform used in this paper.

2.1 Kyber

Kyber is an IND-CCA2-secure KEM scheme constructed from the IND-CPA-secure public key encryption (PKE) using the Fujisaki-Okamoto (FO) transformation [FO99]. The security of Kyber is based on the Module-LWE (MLWE) problem, which suggests that it is hard to distinguish between $(\mathbf{A}, \mathbf{t} = \mathbf{As} + \mathbf{e})$ and the uniform random sampling $(\mathbf{A}, \mathbf{u})$. Kyber features a modular design that uses a k-dimensional matrix $\mathbf{A}$, where $k = 2, 3, 4$ corresponds to Kyber512, Kyber768, and Kyber1024, respectively. This design provides flexibility in adjusting security levels and allows for the reuse of core implementations. Kyber employs SHA-3 primitives to generate the polynomial matrices and vectors. Due to the page limit, we refer to [ABD+21] for detailed specifications of Kyber PKE and KEM.

2.2 Dilithium

The security of Dilithium relies on two hard lattice problems: MLWE and SelfTargetMSIS. Like Kyber, Dilithium also adopts a modular design, providing flexibility in adjusting security levels. The SelfTargetMSIS problem [KLS18] involves finding a vector $[\mathbf{z}, c, \mathbf{v}]^T$ with small coefficients and a message μ such that $\mathrm{H}\left(\mu \| [\mathbf{A}|\mathbf{t}|\mathbf{I}] \cdot [\mathbf{z}, c, \mathbf{v},]^T\right) = c$. Here, $\mathbf{A}$ and $\mathbf{t}$ are uniformly random polynomial matrices and vectors, respectively, and $\mathbf{I}$ is the identity matrix. It is suggested that one can get a non-tight security reduction from the standard MSIS problem to the SelfTargetMSIS problem in the Random Oracle Model (ROM) [DKL+21]. Similar to Kyber, Dilithium requires extensive SHA-3 computations to generate polynomial matrices and vectors. Matrix-vector and polynomial multiplications are among its core operations.

2.3 Number Theoretic Transform

Matrix-vector multiplication, a computationally intensive operation in both Kyber and Dilithium, is primarily composed of numerous polynomial multiplications. In both Kyber

and Dilithium, polynomial multiplications are performed on the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, with the polynomial order fixed at $n = 256$. The moduli used in Kyber and Dilithium are $q = 3329$ and $q = 8380417$, respectively. In Kyber, the modulus satisfies $q \equiv 1 \bmod n$, rather than $q \equiv 1 \bmod 2n$, which allows only primitive 256th roots of unity. Conversely, in Dilithium, the modulus satisfies $q \equiv 1 \bmod 2n$, resulting in primitive 512th roots of unity. As a result, polynomial multiplication in Kyber can be accelerated using an incomplete 7-layer NTT, while Dilithium employs a fully-splitting 8-layer NTT.

In both Kyber and Dilithium, the 7-layer and 8-layer NTTs decompose the polynomial $X^n + 1$. For Kyber, it is factored into 128 degree-1 polynomials modulo $X^2 - \zeta^{2i+1}$ for $i \in [0, 127]$. In Dilithium, it is factored into 256 degree-0 polynomials modulo $X - \zeta^{2i+1}$ for $i \in [0, 255]$, where ζ represents the corresponding primitive root. This factorization is expressed as

$$X^{256} + 1 = \prod_{i=0}^{127} \left(X^2 - \zeta^{2i+1}\right) \text{ and } X^{256} + 1 = \prod_{i=0}^{255} \left(X - \zeta^{2i+1}\right)$$

for Kyber and Dilithium, respectively. Following the degree-1 and degree-0 pointwise multiplications in Kyber and Dilithium, respectively, the inverse NTT (INTT) is applied to transform the NTT-domain values back to the standard domain. Additionally, the 12-bit and 23-bit moduli used in Kyber and Dilithium result in their respective NTT operations typically employing 16-bit and 32-bit implementations. The butterfly unit forms the core component of both NTT and INTT. Two distinct butterfly algorithms are used: the Cooley-Tukey (CT) algorithm [CT65] and the Gentleman-Sande (GS) algorithm [GS66].

2.4 Keccak

Apart from the polynomial multiplication, both Kyber and Dilithium require extensive SHA-3 computations to generate polynomial matrices and vectors. The SHA-3 family, based on the sponge construction, includes four hash functions: SHA3-{224,256,384,512}, where the number indicates the output digest length, and two extendable-output functions (XOFs): SHAKE{128,256}, where the number indicates the security strength. All SHA-3 functions are constructed using the Keccak-f1600 permutation. Keccak-f1600 consists of 24 rounds, each composed of five transformations: θ, ρ, π, χ, and ι. Algorithm 1 provides an overview of a round of the Keccak-f1600, where $r[x, y]$ and RC are fixed constants, $\oplus$ denotes the `XOR` operation, and `ROT` refers to a 64-bit left rotation operation.

The state of Keccak-f1600 comprises 1600 bits, organized in three dimensions, with bits identified by coordinates $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_{64}$. The bit index $i = z + 64(5y + x)$ corresponds to coordinates (x, y, z). This 1600-bit state can also be viewed as a 5×5 matrix $A[x, y]$, with 64-bit values.

Several optimization techniques for implementing Keccak are discussed in [BDH+12]; a brief overview follows.

Bit interleaving, primarily used on 32-bit CPUs, maps 64-bit rotations to 32-bit rotations by adjusting the state's encoding. For instance, $L[z] = A[x, y, z]$ is mapped to U_0 and U_1 with $U_0[j] = L[2j]$ and $U_1[j] = L[2j + 1]$ for $j \in [0, 31]$, which divides each 64-bit lane by odd and even positions. Rotating L by 2τ bits corresponds to separately rotating the two 32-bit words by τ bits. Rotating L by $2\tau + 1$ bits corresponds to $U_0 \leftarrow \texttt{ROT}_{32}(U_1, \tau + 1)$ and $U_1 \leftarrow \texttt{ROT}_{32}(U_0, \tau)$. Notably, when the rotation offset is 1, only one 32-bit rotation is required, which applies to 6 out of the 29 64-bit rotations in Keccak-f1600.

We refer to the encoding and decoding methods from XKCP[1] to analyze the overhead introduced by this technique. Encoding two 32-bit values to be absorbed and XORing

[1] `toBitInterleavingAndAND` and `fromBitInterleaving` macros at https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/plain-32bits-inplace/KeccakP-1600-inplace32BI.c

Algorithm 1 A round of Keccak-f1600

Input: The 1600-bit state A; A constant RC
Output: A

1: θ step:
2: $C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], \quad \forall x \text{ in } 0 \dots 4$
3: $D[x] = C[x-1] \oplus \texttt{ROT}(C[x+1], 1), \quad \forall x \text{ in } 0 \dots 4$
4: $A[x,y] = A[x,y] \oplus D[x], \quad \forall (x,y) \text{ in } (0 \dots 4, 0 \dots 4)$
5: ρ and π steps:
6: $B[y, 2x+3y] = \texttt{ROT}(A[x,y], r[x,y]), \quad \forall (x,y) \text{ in } (0 \dots 4, 0 \dots 4)$
7: χ step:
8: $A[x,y] = B[x,y] \oplus ((\texttt{NOT}\ B[x+1,y])\ \texttt{AND}\ B[x+2,y]), \quad \forall (x,y) \text{ in } (0 \dots 4, 0 \dots 4)$
9: ι step:
10: $A[0,0] = A[0,0] \oplus \text{RC}$
11: **return** A

them with the corresponding Keccak state consumes 56 logical instructions. For Kyber and Dilithium, this process typically requires fewer than $56 \times 5 = 280$ instructions. During the squeezing phase, decoding two 32-bit states consumes 48 logical instructions. For SHA3-{256,512} and SHAKE{128,256}, the decoding overhead amounts to $\{17, 9, 21, 17\} \times 48 = \{816, 432, 1008, 816\}$ instructions, respectively. Usually, encoding and decoding happen at the API level (like in SHA3 or SHAKE) and not inside the Keccak process itself.

Lane complementing technique reduces the number of `NOT` operations in the χ mapping by representing some states as their complements, converting some `AND` operations into `OR` operations. For more details, we refer to [BDH+12, Sec 2.2]. This technique benefits architectures that lack a single instruction to perform both `AND` and `NOT` operations. For our Keccak-f1600 implementation on RV64IM, we use this technique by performing `NOT` operations on 6 states before storing and after loading states, reducing the number of `NOT` operations in the χ mapping of a Keccak-f1600 round from 25 to 8.

In-place implementation. As shown in Algorithm 1, the ρ and π steps rotate 25 states and store the results into the temporary $B[]$. Then, the χ step uses $B[]$ as input and stores the results into $A[]$. The in-place technique aims to align the positions (register or memory resources) of $B[]$ and $A[]$, ensuring that the positions of $A[x,y]$ remain the same before and after n rounds of computation. Bertoni et al. [BDH+12, Sec 2.5] describes an in-place method for $n = 4$, which requires unrolling four rounds with different position orders for each. In contrast, [BK22, Sec 3.1] describes an in-place method for $n = 1$, which is adopted in this paper. More details will be discussed in Section 4.

Lazy rotations. Becker et al. [BK22] introduced the lazy rotation technique for the ARMv8-A architecture, leveraging the Barrel shifter feature to eliminate explicit rotations. Later, Huang et al. [HAZ+24] applied this technique to the ARMv7-M architecture. However, this approach is not applicable to RISC-V due to the absence of the Barrel shifter feature.

The canonical implementation of Keccak-f1600 incurs 76 `XORs`, 25 `ANDs`, 25 `NOTs`, and 29 64-bit rotations per round. The number of instructions required for specific architectures using various optimization techniques will be discussed in Section 4.

2.5 Target platform: CanMV-K230 with C908 core

We use the CanMV-K230 development board [Can24], equipped with a XuanTie C908 core [TH23] operating at 1.6 GHz. The CPU features 32KB each of L1 instruction and data cache, and 256KB L2 cache. The C908 supports the RISC-V {32,64}GCBV instruction set, where the vector extension version is 1.0 [RIS21b] with `VLEN`=128 (Vector register bit length), and the B extension version is 1.0.0 [RIS21a]. It does not implement

Table 1: Latency and (cycles per instruction) CPI of commonly-used instructions on Canaan K230 C908 core. `SEW`: selected element width.

Instruction	Latency	CPI
RV{32,64}I		
`arith/logic/compare`	1	0.5
`lh/lw/ld`	3	1
`sh/sw/sd`	1	1
RV{32,64}M		
`mulw` on RV64M	3	1
`mul{h}` on RV64M	4	2
`mul{h}` on RV32M	3	1
RV{32,64}B		
`rori/andn`	1	0.5
RV{32,64}V		
`vadd/vsub`	4	1
`vmul{h}` with `SEW`$\leq$16	4	1
`vmul{h}` with `SEW`>16	5	1
`logic/vmerge`	4	2
`vrgather`	5	4
`vle`	≥ 3	2
`vse`	1	2.3

the vector cryptography bit-manipulation extension (Zvkb [RIS21c]). It supports both RV64 and RV32 execution modes. The C908 has several notable features: a 9-stage pipelining architecture; in-order dual-issue for scalar instructions and single-issue for vector instructions; in-order fetch, dispatch, execute, and retire; supporting for write combining; concurrent bus access for memory read/write operations up to 8-way/12-way.

This paper primarily focuses on the RV{32,64}IM{B}{V} instruction sets. The C908 user manual [TH23] provides latencies for integer and multiplication instructions but does not reveal any microarchitectural details. Additionally, the latencies for B extension and V extension instructions are undocumented. Therefore, we conducted a series of microbenchmarks on the C908 core inspired by [Fog23, AR19]. There are minor discrepancies between our microbenchmark results and the latencies provided in the documentation. For detailed discrepancies, we refer to our artifact. For the purpose of this paper, we will refer to our microbenchmark results as shown in Table 1.

We highlight several noteworthy features. The load-use latency from the data of a `lh/{lw,ld}` instruction to the ALU of a dependent datapath instruction is three/two cycles. This means that in a back-to-back `lh/{lw,ld}-add` sequence the `add` instruction would stall for two/one cycle. We attribute the additional latency of the `lh` instruction to zero or sign extension. The latencies for the `vle` and `vse` instructions are relatively low, benefiting from multi-way concurrent bus access. The CPI for `vse` instruction is slightly higher than for `vle`, which we attribute to the additional overhead introduced by the write combining feature.

3 Modular Arithmetic

During the NTT computation over R_q, coefficient operations are performed modulo q. There are two widely used algorithms for modular reduction: Montgomery [Mon85] and Barrett reduction [Bar86]. Barrett reduction relies on integer approximation. The idea of Barrett reduction is to first precompute an integer approximate reciprocal $\lfloor \beta/q \rfloor$, where

Algorithm 2 Signed Montgomery multiplication [Sei18]

Input: Two signed integers a, b satisfying $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$, where $q \in (0, \frac{\beta}{2})$ and $\beta = 2^l$
Output: $r = ab\beta^{-1} \bmod^{\pm} q, r \in (-q, q)$
1: $c = c_1\beta + c_0 = a \cdot b$
2: $m = c_0 \cdot q^{-1} \bmod^{\pm} \beta$
3: $t_1 = \lfloor m \cdot q/\beta \rfloor$
4: $r = c_1 - t_1$
5: **return** r

$\beta = 2^l > q$ and l is the machine word size (e.g., 16, 32, 64). Then, it approximately computes $c \bmod q = c - q \cdot \lfloor c/q \rfloor \approx c - q \cdot \lfloor c\lfloor \beta/q \rfloor/\beta \rfloor$. Recently, Becker et al. [BHK+22] proposed an efficient Barrett multiplication for the one-known-factor multiplication using the single-width multiply-high-with-rounding instruction. Because NEON does not support the normal single-width multiply-high instruction, they tailored the Barrett multiplication to use the $\lfloor \cdot \rceil$ function and proposed an efficient Barrett multiplication implementation using the single-width NEON instructions. More discussion can be found in [BHK+22]. This work will focus on Montgomery and Plantard arithmetic.

3.1 Signed Montgomery and Plantard Arithmetic

Montgomery arithmetic [Mon85] aims to replace the expensive division by modular or division by $\beta = 2^l$. Note that the result produced by Montgomery arithmetic is not the exact product ab; instead it is given in Hensel remainder form as $ab\beta^{-1} \bmod q$. Nevertheless, this would not be an issue because we can precompute the constant $\beta \bmod q$ with the twiddle factors as in [ADPS16]; then Montgomery multiplication with these twiddle factors produces results in the normal domain. The state-of-the-art signed Montgomery multiplication is proposed by Seiler in [Sei18]. This algorithm is designed to utilize only single-width operations. As shown in Algorithm 2, this algorithm only needs to compute one high-half multiplication, one low-half multiplication, and one single-width subtraction. This is useful for vectorized implementation because handling the double-width intermediate values would sacrifice half of the parallelism.

In 2021, a novel modular arithmetic called Plantard arithmetic was proposed by Plantard [Pla21]. Since then, two versions of Plantard arithmetic have been developed [HZZ+22, AMOT22] to effectively handle signed integers in LBC schemes. Compared to Montgomery multiplication, Plantard multiplication by a constant can save one multiplication by pre-computation, making it advantageous for the multiplication by twiddle factors in NTT/INTT. As shown in Algorithm 3 and Algorithm 4, when b is a constant, such as a twiddle factor in LBC schemes, we can pre-multiply $bq' \bmod^{\pm} 2^{2l}$ to save one multiplication. Note that Algorithm 3 proposed in [HZZ+24] further extended the input range of Plantard arithmetic in [HZZ+22]. However, this algorithm requires $l \times 2l$-bit multiplication, which may not be suitable for platforms lacking efficient support for this operation, such as AVX2, NEON, and RISC-V vector extension. Therefore, Montgomery arithmetic remains the prefered choice for these architectures.

The main difference between Algorithm 3 and Algorithm 4 lies in their rounding functions: Algorithm 3 uses rounding toward negative infinity ($\lfloor \cdot \rfloor$), while Algorithm 4 rounds to the nearest integer ($\lfloor \cdot \rceil$). Division by β is typically implemented using the right shift instruction on modern CPUs. Furthermore, since most modern CPUs, such as ARM Cortex-M and RISC-V, do not support right shifts with $\lfloor \cdot \rceil$ rounding, Algorithm 4 is inefficient on platforms lacking this feature. The original paper implemented Algorithm 4 with two additional additions with 2^{l-1} to simulate these rounding operations (see Source code 1 in [AMOT22]). In this case, their Plantard arithmetic implementation consumes

Algorithm 3 Signed Plantard multiplication [HZZ+24]

Input: Two signed integers a, b such that $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha}), q < 2^{l-\alpha-1}, \alpha > 0, q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in [-\frac{q+1}{2}, \frac{q-1}{2}]$

1: $r = abq' \bmod^{\pm} 2^{2l}$
2: $r = \lfloor r/\beta \rfloor + 2^{\alpha}$
3: $r = \lfloor rq/\beta \rfloor$
4: **return** r

Algorithm 4 Signed Plantard multiplication [AMOT22]

Input: Two signed integers a, b with $|a|, |b| \leq 2^{l-1}$, the odd modulus $q < 2^{l-1}$ and $q' = q^{-1} \bmod^{\pm} 2^{2l}$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in [-\frac{q-1}{2}, \frac{q-1}{2}]$

1: $r = abq' \bmod^{\pm} 2^{2l}$
2: $r = \lfloor r/\beta \rceil$
3: $r = \lfloor rq/\beta \rceil$
4: **return** r

one more addition than Algorithm 3.

3.2 An update on rounding-based Plantard arithmetic

As a secondary outcome of this work, we propose an improvement to the signed Plantard multiplication presented in [AMOT22]. We show that the $\lfloor \cdot \rceil$ function in line 2 of Algorithm 4 can be replaced with $\lfloor \cdot \rfloor$ by slightly restricting the modulus q.

Let $p = abq^{-1} \bmod^{\pm} 2^{2l}$. The original correctness proof ensures $p_0 = p \bmod^{\pm} 2^l$ and $p_1 = \lfloor p/2^l \rceil = (p - p_0)/2^l$. Instead of using $\lfloor \cdot \rceil$, we adopt the relationship between p, p_0 and p_1 in [HZZ+22], namely $p_0 = p \bmod 2^l$ and $p_1 = \lfloor p/2^l \rfloor = (p - p_0)/2^l$. The correctness of signed Plantard multiplication in [AMOT22] relies on the inequality $|ab - p_0 q| < 2^{2l-1}$. After this modification, since the maximum value of p_0 increases from 2^{l-1} to 2^l, we need to reduce the range of q from 2^{l-1} to 2^{l-2} to maintain the inequality. Notably, both Kyber and Dilithium have $q < 2^{l-2}$; thus this stricter constraint on the modulus does not impact practicability in these LBC schemes. Consequently, the remainder of the correctness proofs remains unchanged. Overall, we have

$$\frac{pq - ab}{2^{2l}} = \left\lfloor \frac{pq - ab}{2^{2l}} + \frac{ab - p_0 q}{2^{2l}} \right\rceil = \left\lfloor \frac{p_1 q}{2^l} \right\rceil = \left\lfloor \frac{\left\lfloor \frac{abq^{-1} \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor q}{2^l} \right\rceil. \tag{1}$$

In sum, we replace the $\lfloor \cdot \rceil$ function in line 2 of Algorithm 4 with the $\lfloor \cdot \rfloor$ function, which is commonly supported by modern CPUs. This improvement eliminates one addition needed to simulate the $\lfloor \cdot \rceil$ function while maintaining the same instruction count as Algorithm 3. Nevertheless, we will use the signed Plantard arithmetic presented in [HZZ+22, HZZ+24] for the remainder of this paper.

4 Optimized Keccak Implementations

4.1 Keccak on RV64I and RV64IB

The 64-bit left rotation on RV64I. The θ and $\rho\pi$ steps require 64-bit left rotation operations. Due to the absence of a dedicated rotation instruction, we use the instruction sequence `slli t,a,n; srli b,a,64-n; xor b,b,t` to implement `b ← ROT(a,n)`. Given that most intermediate values during a Keccak-f1600 round cannot be overwritten directly, this sequence commonly requires two temporary registers—one for holding the intermediate value and one for the result.

Lane complementing on RV64I. Given that the RV64I instruction set lacks a direct `a AND NOT(b)` instruction, the lane complementing technique proves beneficial. By

employing this technique, we perform NOT operations on 6 out of 25 64-bit state lanes after loading and before saving them. This reduces the number of NOT operations required in the χ step of a Keccak-f1600 round from 25 to 8. Additionally, some of the AND operations in the χ step are transformed into OR operations, resulting in new computation patterns.

The rori and andn instructions on RV64IB. The B extension provides the 64-bit rotation instruction rori for directly computing the ROT operation in Algorithm 1 and the andn instruction for computing (NOT $B[x+1,y]$) AND $B[x+2,y]$ in the χ step.

Firstly, compared to the 3-instruction rotation sequence on RV64I, using the rori instruction significantly reduces register allocation pressure and minimizes Read After Write (RAW) hazards. Secondly, the technique of lane complementing, used to reduce the number of NOT operations, is no longer necessary.

Register allocation and in-place implementation. We use 25 registers to maintain the 1600-bit state, leaving five registers for other purposes. We adopt Becker et al.'s in-place method, which will be described briefly, with further details in [BK22, Sec 3.1]. Recall that the $\rho\pi$ steps of Algorithm 1 store their results in $B[]$. The goal is to slightly offset loc $B[]$ from loc $A[]$ for the computation of $\rho\pi$ and to move entries back to their original places in χ. The loc X denotes the register used by X. We set loc $B[x,y]$ = loc $A[x,y]$ for $x \notin \{0,1\}$ and loc $B[x,y]$ = loc $A[x,(y+1)\%5]$ for $x \in \{0,1\}$ and $y \in \{1,2,3,4\}$, while using fresh registers for $B[0,0]$ and $B[1,0]$. This in-place method eliminates the need for mov instructions and stack spilling in a round of Keccak-f1600.

Optimizations for dual issue on RV64I. Compared to the implementation by Becker et al. on ARMv8-A [BK22], our dual-issue optimization is more complex. ARMv8-A has 31 programmable registers, whereas we have only 30. Additionally, ARMv8-A supports the bic instruction for computing c $\leftarrow$ a AND NOT(b), and Becker et al. utilized the Barrel shifter to eliminate explicit rotations. This provided them with sufficient registers to alternate instructions and avoid pipeline stalls.

In contrast, our three-instruction rotation operation requires an additional temporary register to store intermediate values, increasing register allocation pressure. This instruction sequences also have RAW hazards, which, if not properly interleaved, can lead to pipeline stalls.

To highlight the importance of carefully designing the register allocation scheme, consider this example: If one adopts Becker et al.'s allocation scheme, which consumes five registers to hold $C[]$ during the computation of the θ step (as shown in [BK22, Fig 4]), there will be no available registers to compute $D[]$, leading to multiple instances of stack spilling.

Considering that the θ step is the most register-intensive part of a round computation, we focus on optimizing this step. Our calculation sequence is as follows: (1) $C[0], C[2] \rightarrow D[1]$. This sequence uses three temporary registers and can be interleaved to execute without stalls; (2) $C[1] \rightarrow A[1,*] \oplus D[1] \rightarrow C[4]$. Similarly, this sequence can also be interleaved to stall-free execution. After this sequence, $D[1]$ can be released. At this point, four out of five temporary registers are occupied by $C[0], C[1], C[2]$, and $C[4]$. (3) $D[3] \rightarrow C[3] \rightarrow A[3,*] \oplus D[3]$. During the calculation of $D[3]$, one register freed from the previous steps is used, but subsequent calculations await the release of the register occupied by $C[2]$. As a result, the two shifts and two XOR operations required for computing $D[3]$ cannot be fully interleaved to eliminate RAW hazards. We thus use stack spilling to obtain an additional temporary register, avoiding stalls. (4) $D[4] \rightarrow A[4,*] \oplus D[4] \rightarrow D[2] \rightarrow A[2,*] \oplus D[2] \rightarrow D[0] \rightarrow A[0,*] \oplus D[0]$. By utilizing the additional temporary register obtained through the above-mentioned stack spilling, this sequence can also execute without stalls.

To illustrate the impact of the aforementioned stack spilling on performance, we report the performance of two versions of Keccak-f1600. We use the triplet (cycle, instruction, CPI) to measure performance, representing the CPU cycles consumed, the number of

retired instructions, and cycles per instruction (CPI), respectively. When transitioning from a version without stack spilling but with minor RAW hazards to a version using stack spilling to eliminate these hazards and enable stall-free execution, the metrics improved from $(2662, 5000, 0.53)$ to $(2591, 5048, 0.51)$ on C908. The second version requires one `ld` and one `sd` instruction per round, resulting in 48 additional instructions.

Optimizations for dual issue on RV64IB. We employ a similar dual-issue optimization method and computation sequence as on RV64I. The only difference is that the implementation on RV64IB does not require stack spilling, allowing instructions to be alternated effectively for stall-free execution.

Statistics. Our Keccak-f1600 implementation on RV64I uses 202 instructions per round, including 196 logical instructions, 2 `ld/sd` instructions for stack spilling, and 4 instructions for loading constants needed by the ι step and maintaining related addresses. The implementation on RV64IB uses 76 `xor`, 25 `andn`, and 29 `rori` instructions per round, totaling 130 logical instructions, plus 4 instructions for constants needed by the ι step.

4.2 Keccak on RV32I and RV32IB

Compared to the RV64 implementation, the RV32 implementation is more complex due to the insufficient number of registers to maintain the 1600-bit state. Therefore, we designed a register allocation scheme aimed at pipeline friendliness for the RV32I and RV32IB implementations. Both RV32I and RV32IB implementations use the in-place method by Becker et al. Similar to the RV64 implementation, the RV32I employs lane complementing, while the RV32IB does not; the relevant analysis for RV64 also applies to RV32, and will not be repeated here. The RV32I does not use bit interleaving, while the RV32IB does, as explained below.

Stoffelen [Sto19] utilized a combination of plane-by-plane processing [BDH+12, Sec 2.4] and 4-round unrolling [BDH+12, Sec 2.5] to reduce memory access. We did not adopt these methods in our implementation for several reasons: (1) Our goal is to reuse the RV32 implementation to create hybrid vector/scalar implementations, which will be discussed later. This requires the vector and scalar implementations to have similar execution flows, prompting us to use Becker et al.'s 1-round in-place method. (2) Our design prioritizes pipeline efficiency. Stoffelen's implementation suffers from numerous load-use hazards, as reflected by its CPI of 0.69, indicating performance degradation. (3) The code size of the 4-round unrolling in-place method is nearly four times that of the 1-round implementation. As noted in [ZHLR22, Sec 5.1], a larger code size can lead to performance degradation on the 32-bit E31 RISC-V core.

Register allocation and optimizations for dual issue. The in-place method used here is similar to that in Subsection 4.1, with the only difference being that `loc` X indicates the register or memory location occupied by X. Considering that RV32 has only $30 \times 32 = 960$ bits of register resources, it cannot fully accommodate the entire 1600-bit Keccak-f1600 state. Consequently, frequent memory access is unavoidable, leading to load-use hazards and potential performance degradation. Our primary goal is to improve pipeline friendliness.

Our basic strategy is to keep some states permanently resident in registers, so they can directly participate in computations and these computations are alternated with memory access instructions to mitigate load-use hazards. In the implementation on RV64, the `a0` register is used for computation because accessing memory to retrieve the Keccak-f1600 state is not required during each round. However, on RV32, due to frequent `a0`-related memory accesses, `a0` is not used for the round computations.

We first determine the number of registers needed to store temporary values. By examining the first six lines of Listing 2, we can see our basic pipeline optimization strategy. This sequence can execute without stalling and requires six temporary registers. Considering that 64-bit rotation operations also require two additional registers, the minimum number

```
1  .macro xor5 dst,b,g,k,m,s,tmp
2      lw    \dst, \b(a0)
3      lw    \tmp, \g(a0)
4      xor   \dst, \dst, \tmp
5      lw    \tmp, \k(a0)
6      xor   \dst, \dst, \tmp
7      lw    \tmp, \m(a0)
8      xor   \dst, \dst, \tmp
9      lw    \tmp, \s(a0)
10     xor   \dst, \dst, \tmp
11 .endm
```

Listing 1: RV32I assembly code from [Sto19] to compute half a parity lane. Load-use hazards prevent optimal pipelining. The xor instruction of the 4th line incurs at least a 1-cycle stall.

```
1  lw   tmp0l, 0*8+0(a0) # A[0,0]l
2  lw   tmp0h, 0*8+4(a0) # A[0,0]h
3  xor  tmp1l, A[0,1]l, A[0,2]l
4  xor  tmp1h, A[0,1]h, A[0,2]h
5  lw   tmp2l, 15*8+0(a0) # A[0,3]l
6  lw   tmp2h, 15*8+4(a0) # A[0,3]h
7  xor  tmp1l, \tmp1l, \tmp0l
8  xor  tmp1h, \tmp1h, \tmp0h
9  lw   tmp0l, 20*8+0(a0) # A[0,4]l
10 lw   tmp0h, 20*8+4(a0) # A[0,4]h
11 xor  tmp1l, \tmp1l, \tmp2l
12 xor  tmp1h, \tmp1h, \tmp2h
13 xor  tmp1l, \tmp1l, \tmp0l
14 xor  tmp1h, \tmp1h, \tmp0h
```

Listing 2: Our optimized RV32I assembly code to compute a parity lane. Part of the state resides permanently in register. Load-use hazards are eliminated by alternating lw and xor instruction.

of temporary registers needed is eight. Additionally, we allocate one more temporary register for flexible purposes, such as the vector/scalar hybrid implementation.

We have 20 remaining registers available for storing the 640-bit state. The chosen state lanes are $A[0,1], A[0,2], A[1,3], A[1,4], A[2,0], A[2,3], A[3,1], A[3,4], A[4,0], A[4,2]$. Notably, in line 2 of Algorithm 1, each $C[x]$ relies on five lanes, specifically $A[x,*]$. Our selection ensures that each $C[x]$ computation can directly access two of the required 64-bit lanes from the registers, allowing the computation instructions to alternate with memory accesses. This approach helps mitigate load-use hazards; see Listing 1 and Listing 2.

The 64-bit rotation and bit interleaving on RV32I. We use four shift instructions and two `xor` instructions to achieve a 64-bit rotation operation. Alternating these six instructions allows for stall-free execution. A 32-bit rotation requires two shift and one `xor` instruction. Referencing the data mentioned in Subsection 2.4, the bit interleaving technique indicates that six out of 29 64-bit rotations have an offset of one, reducing the instruction count by $24 \times 6 \times 3 = 432$. In the context of Kyber and Dilithium, SHAKE128 is used to generate polynomial matrices. Using SHAKE128 as an example, even when ignoring the encoding overhead of the `absorbing` phase, a single `squeeze` operation requires 1008 instructions for decoding. Therefore, from an instruction count perspective, this technique does not provide direct benefits for Keccak-f1600 on RV32I.

Bit interleaving on RV32IB. A 64-bit rotation still requires four shift instructions and two `xor` instructions. With bit interleaving, six of the 29 rotations only consume one `rori` instruction, while the remaining 23 consume two `rori` instructions each. The saved instruction count is $24 \times (29 \times 6 - 6 - 23 \times 2) = 2928$. Therefore, this technique is advantageous for implementations on RV32IB. Additionally, using `rori` instructions slightly alleviates register allocation pressure.

Statistics. In the RV32I implementation, each round consumes 344 logical instructions, 115 `lw` instructions, and 101 `sw` instructions. The implementation on RV32IB uses 258 logical instructions, 115 `lw` instructions, and 101 `sw` instructions per round. Most of the memory access instructions are used for loading and storing the Keccak-f1600 state, with a smaller portion allocated for stack spilling.

4.3 Keccak on RVV

The implementation strategy for Keccak-f1600 using the vector extension is nearly identical to the implementation on RV64I. All logical instructions used in the RV64I implementation

have vector equivalents. Considering our target platform has a `VLEN` of 128, the vector implementation can execute two Keccak-f1600 instances in parallel.

There are a few minor differences between the implementations on RVV and RV64I: (1) RVV has 32 vector registers, which allows for simpler instruction scheduling to optimize pipelining without causing stack spilling. (2) The `vi` version of shift instructions, such as `vsll.vi vd,vs2,uimm`, use a 5-bit immediate value to specify the shift amount. Since the shift amounts in Keccak-f1600 may exceed what a 5-bit immediate can represent, we have to use the `vx` versions of these instructions, where a scalar register specifies the shift amount. As a result, the RVV implementation includes a few `li` instructions and occasionally occupies one scalar register.

Statistics. Each round uses 196 vector logical instructions, one `vle64` for loading constants used in the ι step, and 26 scalar instructions.

4.4 Hybrid Implementations on C908 RV{32,64}I{B}{V}

The hybrid vector/scalar implementation approach has been used to accelerate Salsa20 on ARMv7-A [BS12], X25519 on ARMv8-A [Len19], and more recently, Keccak on ARMv8-A and ARMv8.4-A [BK22]. As discussed in [BK22, Sec 4.4], the hybrid idea involves interleaving code paths A and B, which use different execution units, to promote parallel execution. One reason the hybrid idea is effective is that these cores often have independent scalar and SIMD execution units. Our target platform shares this characteristic.

Next, we analyze the performance of Keccak-f1600 on RVV, which is key to understanding our hybrid implementation. We will then explain why the hybrid idea is not suitable for C908 RV64I{B}V and demonstrate its successful application on C908 RV32I{B}V.

Performance of RVV implementation. Recall that all vector instructions of C908 are single-issue, with logical vector instructions having a CPI of 2. Ideally, the CPI of Keccak-f1600 on RVV should be 2. Table 2 reports the performance, showing a CPI slightly below 2 due to the inclusion of some scalar instructions. The average cost of one-way Keccak-f1600 on RVV is 4827 cycles, which is significantly higher than the counts on C908 RV64I{B}. This performance discrepancy is primarily due to the limited capability of the vector backend in handling logical instructions.

Hybrid implementation on C908 RV64I{B}V. The hybrid approach does not yield benefits for C908 RV64I{B}V. Our implementation on C908 RV64I{B} already achieves near-ideal CPI, meaning our programs fully utilize the CPU's front and back end. When combining RVV with RV64I{B} for a hybrid implementation, vector instructions compete with scalar instructions for the front-end capacity. Despite a vector instruction can execute two 64-bit logical operations in parallel, its computational power is still inferior to that of scalar implementation. We included various RV64I{B}V hybrid implementations in our artifact as they may be suitable for future hardware.

Hybrid implementation on C908 RV32I{B}V. Our RVV implementation computes one-way Keccak-f1600 faster than RV32I{B} on C908, which is a premise for the hybrid approach's applicability here. Consider the execution of three consecutive 4-cycle vector logical instructions on C908. Even with two execution units available, the third instruction would stall for 2 cycles, waiting for available units. Thus, for Keccak-f1600 implementation on C908 RVV, even if the front end can issue only one instruction per cycle, the backend logical units remain a bottleneck, causing pipeline stalls. Our microbenchmarks corroborate this, showing that sequences of `vand;vand` and `vandx2;andx4` both take 4 cycles. For an ideal hybrid implementation on C908, the ratio of vector to scalar instructions should be 1:2. Our RVV implementation uses 197 vector and 26 scalar instructions per round, while RV32I and RV32IB implementations use 560 and 474 scalar instructions per round, respectively. When constructing a hybrid implementation with one RVV and one RV32I{B}, the ratios are 1:3 and 1:2.5, respectively. To achieve the ideal ratio, one could use three RVV implementations with two RV32I implementations, closely

Algorithm 5 Plantard multiplication by a constant for Kyber on RV32IM [HZZ+24]

Input: 32-bit signed integer $a \in [-137q, 230q]$; $\alpha = 3$; precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{2l}$; $q2^l = q \times 2^l$; $l = 16$
Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in (-\frac{q}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **mul** r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
3: **srai** $r, r, \#16$
4: **addi** $r, r, 2^{\alpha}$ ▷ $r \leftarrow ([r]^l + 2^{\alpha})$
5: **mulh** $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$
6: **return** r

Algorithm 6 Montgomery multiplication for Dilithium on RV32IM

Input: Two signed integers a, b satisfying $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$, where $q \in (0, \frac{\beta}{2})$ and $\beta = 2^{32}$; precomputed integer bq' where b is a constant and $b \in (0, q)$; $q' = q^{-1} \bmod \beta$
Output: $r = ab\beta^{-1} \bmod^{\pm} q, r \in (-q, q)$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} \beta$ ▷ precomputed
2: **mul** r, a, bq'
3: **mulh** t, a, b
4: **mulh** r, r, q
5: **sub** r, t, r
6: **return** r

```
.macro ct_bfu_x2 a0_0,a0_1,a1_0,a1_1,zeta0,zeta1,q16,t0,t1
    mul  \t0, \a0_1, \zeta0;    mul  \t1, \a1_1, \zeta1
    srai \t0, \t0, 16;          srai \t1, \t1, 16
    addi \t0, \t0, 8;           addi \t1, \t1, 8
    mulh \t0, \t0, \q16;        mulh \t1, \t1, \q16
    sub  \a0_1, \a0_0, \t0;     sub  \a1_1, \a1_0, \t1
    add  \a0_0, \a0_0, \t0;     add  \a1_0, \a1_0, \t1
.endm
```

Listing 3: 2-way alternation of Plantard-based CT butterfly unit.

approximating 1:2. This would result in an 8-way parallel Keccak-f1600 implementation, significantly increasing programming complexity and making it difficult to construct a unified loop execution flow. Therefore, we decided to start with one RVV and one RV32I{B} to construct the hybrid implementation. We also experimented with combining one RVV and two or three scalar implementations, resulting in 3-way, 4-way, and 5-way parallel Keccak-f1600. Among these, the 3-way version performed best on C908, as shown in Table 2.

5 NTT Optimizations for Kyber and Dilithium

Dual-issue optimization and CPI of NTT. Given that our various NTT implementations rely on dual-issue optimization, we illustrate the basic principles using Huang et al.'s Plantard multiplication on RV32 [HZZ+24] as an example. In Algorithm 5, each instruction depends on the result of the previous one, which is manageable for single-issue CPUs but introduces RAW hazards on dual-issue CPUs. The optimization strategy involves alternating multiple instances to mitigate these hazards. Listing 3 demonstrates the alternating execution of two Plantard-based CT butterfly units. Considering the 3 or 4-cycle delay of multiplication instructions on C908, six-way or eight-way alternation yields optimal performance. However, in NTT implementations, we can achieve at most four-way alternation due to register constraints. Each Plantard-based CT unit consumes two multiplication instructions and four logical/arithmetic instructions, ideally achieving a CPI of $(1 \times 2 + 0.5 \times 4)/6 = 0.67$ on C908 RV32IM. Our microbenchmarks indicate that four-way CT alternation achieves CPIs of 0.88 and 1.08 on C908 RV32IM and RV64IM, respectively. The pipeline optimization for the RVV implementation is similar. By utilizing scalar registers to load certain constants and employing the `vmul{h}.vx` instruction, we can achieve up to 8-way interleaving. This results in an ideal CPI close to 1.

Asymmetric multiplication [AHKS22, BHK+22] optimizes the computation of Kyber's matrix-vector multiplication **As**. During this process, the NTT-domain secret vector $\hat{\mathbf{s}}$ is used k times, and the term $\hat{s}_{2i+1}\zeta^{2br_7(i)+1}$ is repeatedly computed. To reduce the computational load, these terms are cached. For Kyber NTT implementation, better accumulation [AHKS22, ACC+22, BHK+22] involves accumulating results into 32-bit values during vector inner product calculations, followed by reduction. For Kyber NTT on RV32IM and RV64IM, we employed both techniques. However, our experiments on RVV showed negligible benefits from them, so we did not adopt these two techniques on RVV.

All our implementations use CT units for NTT and GS units for INTT. Utilizing CT units for INTT does not provide performance benefits, as detailed in [HZZ+24, Sec 5].

Kyber NTT on RV32IM. Our implementation is based on the speed-version presented in [HZZ+24], which uses a 4+3 layer merging strategy. Thanks to Plantard arithmetic, no modular reductions are needed during the NTT and INTT computations. The Plantard multiplication employed is described in Algorithm 5. While they also proposed several memory optimization strategies, we did not adopt these as our focus was not on memory optimization. As will be discussed below, by precomputing bq', Montgomery multiplication requires only four instructions: three multiplications and one subtraction. However, this method has the drawback of increased register consumption due to loading bq'. Consequently, we cannot implement a 4+3 layer merging strategy and must instead use a 3+3+1 layer merging strategy. In addition, when using Montgomery arithmetic, the INTT computation process requires extra modular reduction.

Kyber NTT on RV64IM. For the RV64IM implementation, only minor modifications to Algorithm 5 are required. Specifically, in line 2, the `mulw` instruction should be used instead of `mul`, as `mulw` is faster on C908 RV64IM. Additionally, the constant $q2^{16}$ in line 5 should be changed to $q2^{48}$. The rest of the implementation remains consistent with the RV32IM version.

Dilithium NTT on RV32IM. Using Plantard arithmetic requires the 32×64-bit multiplier, which is why we use Montgomery arithmetic instead, as described in Algorithm 6. We precompute $bq' \leftarrow bq^{-1} \mod 2^l$, thus Algorithm 6 uses three `mul{h}` instructions and one `sub` instruction. This technique is also employed in the Kyber AVX2 implementation[2] and the NTT implementations on RVV discussed below utilize this technique. Loading bq' increases register usage, which limits us to merging a maximum of three layers at a time, resulting in a 3+3+2 layer merging strategy. Both NTT and INTT calculations do not require modular reduction. For example, in INTT, the input range is $(-q, q)$, and after 8 layers of INTT, the range becomes $(-2^8q, 2^8q)$, which avoids overflow.

Dilithium NTT on RV64IM. The 64-bit multiplier on RV64IM makes Plantard arithmetic feasible. By making simple modifications to Algorithm 5, Plantard arithmetic can be applied. Specifically, l is set to 32, so in line 5, $q2^l$ becomes $q2^{32}$; α is set to 8; and the shift offset in line 3 changes from 16 to 32. According to [HZZ+24, Sec 3], the maximum and minimum allowable values of a are $a_{max} < (2^{64} - 8380417 \times 2^{40})/8380417 \approx 131457q$ and $a_{min} > (8380417 \times 2^{32} - 8380417 \times 2^{40})/8380417 \approx -130686q$, respectively. The layer merging strategy is 4+4. Both NTT and INTT calculations do not require modular reduction.

Kyber NTT on RVV. Due to the absence of 16×32-bit multiplication instructions on RVV, using 32×32-bit multiplications for Plantard arithmetic would reduce parallelism by half. Consequently, we adopt Montgomery arithmetic, adjusting β in Algorithm 6 to 2^{16}. The instructions used in Algorithm 6 have corresponding counterparts in the vector extension. A key trick is to use the `vmul{h}.vx` instruction whenever possible, where one operand is specified by a scalar register. This approach reduces the allocation pressure on vector registers and takes advantage of the faster scalar load instructions compared to

[2] https://github.com/pq-crystals/kyber/blob/8e390f7152cf66f27cb39f164a3b2a8256bf863c/avx2/ntt.S

vector loads. Similar to the Kyber NTT AVX2 implementation, our layer merging strategy is 1+6. No additional modular reduction is required during the NTT computation. In the INTT process, before computing the 0th layer, each coefficient is first modularly multiplied by the constant $\mathsf{mont}^2/128$ to normalize the coefficient range to $(-q, q)$. The mont^2 term is involved in the Montgomery domain conversion. After the first three levels, coefficients in 16 vector registers approach overflow boundaries, necessitating modular reduction. After two more levels, two additional registers require reduction. In total, the INTT process requires 18 extra modular reductions. Additionally, the `vrgather.vv` and `vmerge.vvm` instructions are used to rearrange polynomial coefficients within the registers to construct the desired execution flow.

Dilithium NTT on RVV. Similar to the Kyber NTT on RVV, we utilize Montgomery arithmetic, specifically the vectorized version of Algorithm 6. The layer merging strategy is 4+4, and both NTT and INTT computations do not require additional modular reduction.

LMUL settings on RVV. We experimented with LMUL > 1 and found that the `ntt` and `intt` subroutines did not benefit from these setting. However, the `poly_reduce`, `poly_tomont`, and some subroutines related to `poly_basemul` showed improvements.

Hybrid implementation on C908. Our experiments show that hybrid idea do not improve NTT performance on C908. For C908, `vmul;vmul;vmul;mul` and `vmul;vmul;vmul` consume the same cycles, indicating that a 4:1 vector-to-scalar instruction ratio is ideal. Multi-way implementations should be considered, meaning we would need to construct {33,17,9}-way hybrid NTT implementations for {16,32,64}-bit NTTs. The Raccoon signature scheme [dPKPR24], which relies on 64-bit NTT, is a potential application. For a core with a smaller vector unit, a 2:1 vector-to-scalar instruction ratio might suffice, making 9-way 32-bit and 5-way 64-bit hybrid NTTs applicable to Dilithium and Raccoon, respectively.

6 Results and Comparisons

We obtain the experimental results using the CanMV-K230 development board [Can24], equipped with a XuanTie C908 core [TH23], as introduced in Subsection 2.5. We use the `Xuantie-900 linux-5.10.4 glibc gcc toolchain v2.8.0`, with all programs compiled using the `-O3` optimization level. All instructions utilized in this work have constant execution time, and we avoid any branches and memory accesses related to secret keys.

6.1 Keccak

Table 2 presents the performance and comparisons of our various implementations. The C reference implementation was obtained from the PQClean library[3].

Given the limited research on the implementation of Keccak across various RISC-V instruction sets, we not only compare our work with the C reference implementation and available RISC-V implementations but also provide performance comparisons between our RV64 and RVV implementations and the ARM Cortex-A55, as well as our RV32 implementations against the ARM Cortex-M4. These comparisons are clearly unfair and are provided for reference only.

The Cortex-A55 is an in-order dual-issue CPU. The optimal CPI for the scalar and NEON instructions used to implement Keccak are 0.5 and 1, respectively. The Cortex-M4 is a single-issue CPU with a CPI of 1 for both logical and arithmetic instructions. Both the A55 scalar and the Cortex-M4 support rotate and `bic` instructions. The A55 NEON supports `bic` instructions but does not support rotate instructions, requiring two instructions to achieve rotation. Additionally, the A55 scalar and the Cortex-M4 support barrel shifter features, enabling the use of lazy rotation techniques.

[3] https://github.com/PQClean/PQClean/blob/master/common/fips202.c at commit 05df469.

Table 2: Performance comparison of Keccak-f1600 on C908 RV{32,64}I{B}{V}. For our implementation, the number of cycles and retired instructions are directly obtained by reading the corresponding performance counters, with cycle counts determined as the median over 10000 iterations. CPI means cycles per instruction. The numbers in parentheses are normalized based on the degree of parallelism. The notation 'x2' indicates a 2-way implementation.

Implementation	Method	Cycles	Instructions	CPI
C on RV32I	ref.	15779	24487	0.64
[Sto19] on RV32I	lane compl. & bit interl. & 4-round in-place & plane-by-plane	8734	12740	0.69
Our RV32I	lane compl. & 1-round in-place & dual-issue opt.	7808	14890	0.52
C on RV32IB	ref.	12341	20460	0.6
[HAZ+24] on Cortex-M4	lazy rotation & bit interl. & 4-round in-place & plane-by-plane	9218	~9156[2]	1.01
Our RV32IB	bit interl. & 1-round in-place & dual-issue opt.	6222	11554	0.54
C on RV64I	ref.	4926	8585	0.57
Our RV64I	lane compl. & 1-round in-place & dual-issue opt.	2591	5049	0.51
C on RV64IB	ref.	2412	4296	0.56
RISCV-Crypto[1] on RV64IB	inline asm for `rori/andn`	2563	4649	0.55
[BK22] on A55-Scalar	lazy rotation & 1-round in-place & multi-issue opt.	1418	2747	0.52
Our RV64IB	1-round in-place & dual-issue opt.	1770	3405	0.52
Our RVVx2	lane compl. & 1-round in-place & dual-issue opt.	9655 (4827)	5462	1.77
[BK22] on A55-NEON	1-round in-place	4560 (2280)	3840	1.19
NEON/Scalar x5 [BK22]	A55-Scalar & A55-NEON	8960 (1792)	-	-
Our RV32IVx3	RV32I & RVVx2	11850 (3950)	20273	0.58
Our RV32IVx4		20374 (5093)	35190	0.58
Our RV32IVx5		30544 (6108)	50285	0.61
Our RV32IBVx3	RV32IB & RVVx2	10527 (3509)	17012	0.62
Our RV32IBVx4		16670 (4167)	28472	0.59
Our RV32IBVx5		24299 (4859)	39991	0.61

[1] https://github.com/riscv/riscv-crypto/blob/main/benchmarks/sha3/zscrypto_rv64/Keccak.c at commit efb77a9.

[2] Huang et al. did not report the number of instructions directly. Therefore, we disassembled their executable file, identified the main loop of Keccak, and multiplied the number of instructions in the main loop by six to estimate this data.

Our RV64I implementation is 90% faster than the C reference implementation. The C implementation also performs well on RV64IB because the compiler successfully utilizes the B extension's rotate and `andn` instructions. These instructions reduce register allocation pressure and, consequently, stack spilling. Our RV64IB implementation is 36% faster than the C implementation. When compared to the A55-scalar implementation from [BK22], our disadvantage mainly stems from a higher instruction count. The A55-scalar implementation uses lazy rotation techniques to reduce the number of instructions.

Stoffelen's implementation on RV32I [Sto19] is based on the Keccak implementation for ARMv7-M from XKCP[4]. This approach incorporates techniques such as bit interleaving, lane complementing, and the 4-round unrolling in-place method. Additionally, the plane-per-plane processing combined with 4-round unrolling reduces memory accesses. Stoffelen did not explain the use of bit interleaving, which we speculate is due to his direct adaptation from ARMv7-M (which supports rotation instructions) implementation without thorough discussion. The main advantage of his implementation lies in the plane-per-plane processing method, which significantly reduces memory accesses but at the cost of 4-round unrolling.

Our RV32I implementation is twice as fast as the C reference implementation and 12% faster than Stoffelen's implementation. It is important to note that Stoffelen's implementation employs bit interleaving, which, for example, adds 1008 extra instructions for each squeeze operation in SHAKE128. Our RV32IB implementation is 98% faster than the C implementation, although this comparison is slightly unfair due to the use of bit interleaving in our implementation. When compared to [HAZ+24] on Cortex-M4, their implementation benefits from fewer instructions due to plane-by-plane processing and lazy rotation. Nevertheless, our implementation still consumes fewer cycles than theirs.

As mentioned in Subsection 4.4, our target platform's vector units are relatively weak at handling logical instructions, resulting in a higher CPI for the RVV implementation. When comparing our RVV implementation to Becker et al.'s A55-NEON implementation [BK22], we note that NEON benefits from a richer instruction set, which reduces the number of instructions.

The hybrid implementation by Becker et al. [BK22] differs slightly from ours, making a direct comparison challenging. The main advantage of our hybrid approach is that the insertion of scalar instructions alleviates pipeline stalls. We combine vector and RV32 implementations for the hybrid, while they combine NEON and 64-bit scalar implementations.

Our RV32IVx3 implementation shows a 22% performance improvement over the RVVx2 implementation. Our RV32IBVx3 and RV32IBVx4 implementations achieve 38% and 16% performance improvements over the RVVx2 implementation, respectively.

6.2 NTT

Table 3 presents the performance comparison of the NTT-related subroutines.

Kyber NTT. Comparing our RV32IM implementation with [HZZ+24] reveals that our dual-issue optimizations accelerate performance by a factor of 1.8 to 2.3. Our RV32IM implementation consumes more cycles than the Cortex-M4 implementation in [HZZ+22]. This difference arises because the M4's SIMD capabilities allow it to complete two 16-bit CT butterflies in just 7 1-cycle instructions. Our implementation requires 12 instructions, with 4 3-cycle multiplication instructions and facing unavoidable RAW hazards.

The RV64IM implementation is slower than RV32IM, which is expected; the `mul` instruction consumes one additional cycle on C908 RV64IM. Furthermore, our RVV implementation consumes more cycles than the A72-NEON implementation from [BHK+22]. This comparison is challenging to assess fairly due to differing implementation strategies

[4] https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/ARM/KeccakP-1600-inplace-32bi-armv7m-le-gcc.s

Table 3: Performance and comparison of NTT, INTT, and base multiplication on C908 RV{32,64}IM{V}, with cycle counts determined as the median over 10000 iterations. Each set of three lines represents the metrics for NTT, INTT, and base multiplication, respectively. For [HZZ+24] on RV32IM, we derived a version based on their implementation, preserving the core code and aligning the interfaces, and tested it on our platform. There are two metrics for the base multiplication due to the use of asymmetric multiplication techniques, which have led to multiple versions. In our implementation, there are actually more versions, but we will only report two here. The first metric involves retrieving the cached value $\hat{s}_{2i+1}\zeta^{2br_7(i)+1}$ and accumulating the multiplication result into 32-bit intermediate values. The second metric, in addition to performing the operation described in the first, also includes modular reduction.

	Impl.	Method	Cycles	Inst.	CPI
Kyber	[HZZ+24] on RV32IM	Plant & 4+3 & single-issue opt.& CT+GS	13218	6774	1.95
			11427	7317	1.56
			2891/5900	2572/3663	1.12/1.61
	Our on RV32IM	Plant & 4+3 & dual-issue opt.& CT+GS	5714	6870	0.83
			6005	7398	0.81
			1673/2668	2412/3536	0.69/0.75
	[HZZ+22] on Cortex-M4	Plant & 4+3& CT+GS	4474	-	-
			4684	-	-
			2422	-	-
	Our on RV64IM	Plant & 4+3 & dual-issue opt.& CT+GS	6609	6871	0.96
			6996	7399	0.95
			2122/3245	2413/3537	0.88/0.92
	Our on RVV	Mont & 1+6 & dual-issue opt& CT+GS	1575	1347	1.17
			1840	1592	1.16
			753	738	1.02
	[BHK+22] on A72-NEON	Barrett & 4+3& CT+GS	1200	-	-
			1338	-	-
			952	-	-
Dilithium	Our on RV32IM	Mont & 3+3+2 & dual-issue opt.& CT+GS	7054	8692	0.81
			7561	9206	0.82
			2026	2349	0.86
	Our on RV64IM	Plant & 4+4& CT+GS	8258	7765	1.06
			8484	8293	1.02
			2320	2227	1.04
	[AHKS22] on Cortex-M4	Mont & 3+3+2& CT+CT	8066	-	-
			8388	-	-
			1931	-	-
	Our on RVV	Mont & 4+4 & dual-issue opt.& CT+GS	3395	2899	1.17
			3540	3106	1.14
			668	118	5.66
	[BHK+22] on A72-NEON	Barrett & 4+4& CT+GS	2241	-	-
			2821	-	-
			1378	-	-

Table 4: Performance of Kyber768 and Dilithium3 on C908 RV{32,64}IM{B}{V}. Most of the cycle counts ($k = 1000$) are determined as the median over 10000 iterations, except that Dilithium `Sign` is obtained as the average over 10000 iterations.

Impl.	Kyber768			Dilithium3		
	KeyGen	Encaps	Decaps	KeyGen	Sign	Verify
[HZZ+24] RV32IM	1052k	1261k	1179k	-	-	-
[HAZ+24] Cortex-M4	604k	732k	674k	2394k	5575k	2302k
Ref RV32IM[1]	1222k	1602k	1691k	4123k	13671k	4145k
Ref RV32IMB[1]	1048k	1377k	1481k	3422k	12635k	3504k
Our RV32IM	496k	606k	578k	1934k	5069k	1889k
Our RV32IMB	447k	550k	532k	1752k	4746k	1720k
Our RV32IMV	312k	419k	371k	1165k	3193k	1165k
Our RV32IMBV	284k	382k	346k	1087k	3054k	1091k
Ref RV64IM[2]	742k	986k	1185k	1841k	8232k	1958k
Ref RV64IMB[2]	603k	803k	1011k	1328k	7217k	1474k
Our RV64IM	278k	326k	357k	920k	3333k	939k
Our RV64IMB	237k	275k	316k	753k	3085k	791k
Our RV64IMV	204k	243k	248k	810k	2406k	800k
Our RV64IMBV	165k	197k	207k	645k	2139k	646k
[BHK+22] A72	99k	127k	120k	515k	1089k	447k

[1] https://github.com/pq-crystals/kyber/tree/main/ref at commit 441c051.
[2] https://github.com/pq-crystals/dilithium/tree/master/ref at commit f1f8085.

and the A72 is an out-of-order CPU. In simple terms, one CT butterfly on NEON requires only five instructions, while we require six.

Dilithium NTT. Our RV32IM implementation consumes slightly fewer cycles compared to the Cortex-M4 implementation from [AHKS22]. They use CT butterflies to construct both NTT and INTT; however, their implementation requires five 1-cycle instructions for a CT butterfly, compared to our six instructions, which include three 3-cycle multiplication instructions. Examining the instruction count for the RV64IM and RV32IM implementations highlights the advantages of Plantard arithmetic, where the slower performance of RV64IM is still attributed to the additional cycle consumed by the `mul` instruction. Similarly, our RVV implementation consumes more cycles than the A72-NEON implementation, for the same reasons noted in the Kyber NTT comparison.

6.3 Kyber and Dilithium

To facilitate comparison with related work, our development is based on the Kyber[5] and Dilithium[6] codebases. For the V extension, we integrated multiple-way Keccak implementations, such as RV32IVx3, RV32IBVx3, and RV32IBVx4, as much as possible into Kyber and Dilithium for accelerating the generation of polynomial matrices and vectors. Where parallelization is not possible, we directly use the scalar implementation. Additionally, we also vectorized the `rej_uniform`, `cbd2`, and `cbd3` subroutines in Kyber, as well as the `rej_uniform` and `rej_eta` subroutines in Dilithium. The vectorization of `packing`/`unpacking`-related subroutines for public and secret keys can further enhance performance; we consider this as part of our future work. We only present results for Kyber768 and Dilithium3 on C908 RV{32,64}IM{B}{V}, as shown in Table 4, due to the page limit.

When compared to the reference C implementation on the same architecture, our

[5] https://github.com/pq-crystals/kyber/tree/main/ref at commit 441c051.
[6] https://github.com/pq-crystals/dilithium/tree/master/ref at commit f1f8085.

optimized implementations often achieve speedups of nearly 2× or more. Our Kyber768 implementations on RV32IM and RV32IMB show 14% ∼ 18% and 18% ∼ 26% cycles reduction, respectively, than the state-of-the-art implementations on Cortex-M4 [HAZ+24]. Our Dilithium3 implementations on RV32IM and RV32IMB show 9% ∼ 19% and 15% ∼ 27% cycles reduction, respectively, than the state-of-the-art implementations on Cortex-M4 [HAZ+24]. This mainly comes from the fact that our platform is dual-issued and our optimizations are dual-issue friendly. While our implementations of Kyber and Dilithium consume more cycles than A72-NEON implementations in [BHK+22], we consider this expected given that the A72 is an out-of-order CPU with a more powerful instruction set.

6.4 Future work

Our work extends beyond Kyber and Dilithium; our Keccak implementations can be integrated into hash-based signatures such as XMSS [HBG+18], LMS [MCF19], and SPHINCS+. SLOTHY [ABKK24] is an advanced framework that automates instruction scheduling, register allocation, and loop optimization for ARMv8.1-M and AArch64 architectures, and future efforts should focus on extending SLOTHY to RISC-V. Additionally, it would be valuable to formally verify our NTT implementations using CryptoLine [PTWY18] on RISC-V.

For cores with `VLEN`≠128. The implementations presented in this work are primarily designed for RVV cores with `VLEN`=128. We encourage future research to develop Kyber and Dilithium implementations compatible with various `VLEN` configurations, following the RISC-V Vector programming model. These implementations should aim to ensure compatibility while also optimizing performance for specific microarchitectures. Notably, the hybrid SHA-3 implementations introduced in this work are highly dependent on the underlying microarchitecture, making the development of compatible versions a significant challenge.

Using `VLEN`=256 as an example, we describe the necessary changes and the expected performance impact: (1) For the subroutines related to `rej_uniform`, `rej_eta`, and `cbd`: The index arrays used for the `vrgather` instruction need to be redesigned, and the number of iterations should be halved. (2) For the NTT-related subroutines: The layer merging strategy can remain unchanged, but the rearrangement of polynomial coefficients needs to be redesigned, which is used to construct the required execution flow. The precomputed table of twiddle factors also needs to be redesigned to accommodate longer vectors. (3) For the Keccak implementation: A 256-bit vector width enables a 4-way Keccak implementation. The hybrid implementation needs to be carefully re-analyzed based on the specific microarchitecture. If the CPI of the relevant instructions is the same as that of the C908 core, then the cycles consumed by these subroutines should theoretically be halved, as the parallelism is doubled.

Acknowledgments

This work is supported by the Jiangsu Province 100 Foreign Experts Introduction Plan (BX2022012) and TÜBİTAK Projects (2232-118C332 and 1001-121F348). We would like to express our gratitude to Hao Cheng and anonymous reviewers for their thoughtful discussions and constructive feedback.

References

[ABB+22] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, and Maria Eichlseder et al. SPHINCS+ (version 3.1) — Submission

to round 3 of the NIST post-quantum project. https://sphincs.org/data/sphincs+-r3.1-specification.pdf, 2022.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020.

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 3.02) -– Submission to round 3 of the NIST post-quantum project. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf, 2021.

[ABKK24] Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, and Fabien Klein. Fast and clean: Auditable high-performance assembly via constraint solving. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):87–132, 2024.

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):127–151, 2022.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange — a new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, 2016.

[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. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):219–242, 2020.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *ACNS 2022*, volume 13269, pages 853–871. Springer, 2022.

[AMOT22] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi. Efficient Word Size Modular Multiplication over Signed Integers. In *ARITH 2022*, pages 94–101. IEEE, 2022.

[AR19] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS '19*, pages 673–686, New York, NY, USA, 2019. ACM.

[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*, volume 263, pages 311–323. Springer, 1986.

[BDH+12] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf, 2012.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):221–244, 2022.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, pages 272–293, Cham, 2022. Springer International Publishing. https://eprint.iacr.org/2022/1243.

[BS12] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.

[BUC19] Utsav Banerjee, Tenzin S Ukyab, and Anantha P Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *arXiv preprint arXiv:1910.07557*, 2019.

[Can24] Canaan Inc. Canaan Kendryte K230 documentation. https://github.com/kendryte/k230_docs/blob/main/README_en.md, 2024. Accessed: 2024-07.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[DKL+21] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium (version 3.1) — Submission to round 3 of the NIST post-quantum project. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf, 2021.

[dPKPR24] Rafaël del Pino, Shuichi Katsumata, Thomas Prest, and Mélissa Rossi. Raccoon: A masking-friendly signature proven in the probing model. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024*, volume 14920, pages 409–444. Springer, 2024.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.

[Fog23] Agner Fog. Test programs for measuring clock cycles and performance monitoring. https://agner.org/optimize/testp.zip, 2023.

[FSMG+19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepúlveda. 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. IEEE, 2019.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 239–280, 2020.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, Dec. 2020.

[GS66] W. Morven Gentleman and G. Sande. Fast fourier transforms: for fun and profit. volume 29 of *AFIPS Conference Proceedings*, pages 563–578, 1966.

[HAZ+24] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):1–24, 2024.

[HBG+18] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, 2018.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):614–636, 2022.

[HZZ+24] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. *IEEE Transactions on Information Forensics and Security*, 2024.

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In *Advances in Cryptology–EUROCRYPT 2018*, pages 552–586. Springer, 2018.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. https://eprint.iacr.org/2019/844, 2019.

[KSFS24] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on RISC-V with hardware acceleration. *ACM Transactions on Embedded Computing Systems*, 23(2):1–23, 2024.

[KSYS22] Youngbeom Kim, Jingyo Song, Taek-Young Youn, and Seog Chung Seo. CRYSTALS-Dilithium on ARMv8. *Security and Communication Networks*, 2022(1):5226390, 2022.

[Len19] Emil Lenngren. AArch64 optimized implementaton for X25519. https://github.com/Emill/X25519-AArch64, 2019.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-Micali Hash-Based Signatures. RFC 8554, April 2019.

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

[NG23] Duc Tri Nguyen and Kris Gaj. Fast Falcon signature generation and verification using ARMv8 NEON instructions. In *International Conference on Cryptology in Africa*, pages 417–441. Springer, 2023.

[NIS15] NIST. FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://csrc.nist.gov/pubs/fips/202/final, 2015.

[NIS24a] NIST. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/final, 2024.

[NIS24b] NIST. FIPS 204: Module-Lattice-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/204/final, 2024.

[NIS24c] NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/205/final, 2024.

[Pla21] Thomas Plantard. Efficient Word Size Modular Arithmetic. *IEEE Trans. Emerg. Top. Comput.*, 9(3):1506–1518, 2021.

[PTWY18] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic assembly programs in cryptographic primitives. In *29th International Conference on Concurrency Theory (CONCUR 2018)*, 2018.

[RIS21a] RISC-V Foundation. RISC-V Bit-Manipulation ISA-extensions Version 1.0.0-2021-06-12. https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf, 2021.

[RIS21b] RISC-V Foundation. RISC-V V Vector Extension Version 1.0-rc1-20210608. https://github.com/riscv/riscv-v-spec/releases/download/v1.0-rc1/riscv-v-spec-1.0-rc1.pdf, 2021.

[RIS21c] RISC-V Foundation. RISC-V Zvkb - Vector Cryptography Bit-manipulation. https://github.com/riscv/riscv-crypto/blob/main/doc/vector/riscv-crypto-vector-zvkb.adoc, 2021. Commit: 3d9dff8.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *IACR Cryptol. ePrint Arch.*, page 39, 2018.

[Sto19] Ko Stoffelen. Efficient cryptography on the RISC-V architecture. In *LATINCRYPT 2019*, volume 11774, pages 323–340. Springer, 2019.

[TH23] T-Head. XuanTie-C908-UserManual. https://www.xrvm.com/product/xuantie/C908, 2023. Accessed: 2023-10-01.

[WJW+19] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems: XMSS hardware accelerators for RISC-V. In *International Conference on Selected Areas in Cryptography*, pages 523–550. Springer, 2019.

[WTJ+20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the HW/SW co-design of qTESLA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3), 2020.

[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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):130–153, 2024.

[ZHLR22] Jipeng Zhang, Junhao Huang, Zhe Liu, and Sujoy Sinha Roy. Time-memory trade-offs for Saber+ on memory-constrained RISC-V platform. *IEEE Transactions on Computers*, 2022.