IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 130–153. DOI:10.46586/tches.v2024.i2.130-153

A Highly-efficient Lattice-based Post-Quantum Cryptography Processor for IoT Applications

Zewen Ye[1,3], Ruibing Song[1], Hao Zhang[1], Donglong Chen[2], Ray Chak-Chung Cheung[3] and Kejie Huang[1]

[1] Zhejiang University, Hangzhou, China
{lucas.zw.ye,songruibing,floyd.haozhang,huangkejie}@zju.edu.cn
[2] BNU-HKBU United International College, Zhuhai, China donglongchen@uic.edu.cn
[3] City University of Hong Kong, Hong Kong, China r.cheung@cityu.edu.hk

Abstract. Lattice-Based Cryptography (LBC) schemes, like CRYSTALS-Kyber and CRYSTALS-Dilithium, have been selected to be standardized in the NIST Post-Quantum Cryptography standard. However, implementing these schemes in resource-constrained Internet-of-Things (IoT) devices is challenging, considering efficiency, power consumption, area overhead, and flexibility to support various operations and parameter settings. Some existing ASIC designs that prioritize lower power and area can not achieve optimal performance efficiency, which are not practical for battery-powered devices. Custom hardware accelerators in prior co-processor and processor designs have limited applications and flexibility, incurring significant area and power overheads for IoT devices. To address these challenges, this paper presents an efficient lattice-based cryptography processor with customized Single-Instruction-Multiple-Data (SIMD) instruction. First, our proposed SIMD architecture supports efficient parallel execution of various polynomial operations in 256-bit mode and acceleration of Keccak in 320-bit mode, both utilizing efficiently reused resources. Additionally, we introduce data shuffling hardware units to resolve data dependencies within SIMD data. To further enhance performance, we design a dual-issue path for memory accesses and corresponding software design methodologies to reduce the impact of data load/store blocking. Through a hardware/software co-design approach, our proposed processor achieves high efficiency in supporting all operations in lattice-based cryptography schemes. Evaluations of Kyber and Dilithium show our proposed processor achieves over **10**$\times$ speedup compared with the baseline RISC-V processor and over **5**$\times$ speedup versus ARM Cortex M4 implementations, making it a promising solution for securing IoT communications and storage. Moreover, Silicon synthesis results show our design can run at **200 MHz** with **2.01 mW** for Kyber KEM 512 and **2.13 mW** for Dilithium 2, which outperforms state-of-the-art works in terms of PPAP (Performance $\times$ Power $\times$ Area).

Keywords: Post-quantum Cryptography · RISC-V · Single-Instruction-Multiple-Data · Lattice-Based Cryptography · Internet-of-Things

1 Introduction

Shor's algorithm [Sho99] undermines the cryptographic strength of traditional Public Key Cryptography (PKC) schemes such as Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC) algorithms, making them vulnerable to quantum computer threats. In addition, the IBM Quantum Development Roadmap[1] predicts that a quantum computer with over 4000 qubits will be attainable by 2025. To defend against the potential

[1]https://www.ibm.com/quantum/roadmap

Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

attacks from quantum computers, the National Institute of Standards and Technology (NIST) has organized four rounds of competitions since 2017 to standardize Post-Quantum Cryptography (PQC) as an alternative to traditional PKC. In 2022, NIST announced the standardization of four Round 3 candidates and introduced four candidates in Round 4 for further consideration. Based on the underlying hard mathematical problems, these PQC schemes can be categorized into three classes: lattice-based, code-based, and hash-based cryptography. Among the four standardization schemes of Round 3 and four candidates of Round 4, three algorithms (CRYSTALS-Kyber [BDK+18], CRYSTALS-Dilithium [DLL+18], and Falcon [FHK+18]) are Lattice-Based Cryptography (LBC) schemes, which have better performance and smaller key sizes. However, it is important to note that the critical operations vary across different PQC schemes and secure-level parameters. To accommodate more than one PQC scheme, it is crucial for the hardware design to exhibit sufficient flexibility. Though LBC schemes offer the advantage of smaller key sizes in PQC candidates, they are still about 5× larger than traditional PKC schemes. For instance, the public key sizes of CRYSTALS-Kyber and CRYSTALS-Dilithium typically are in the range of 800-1184 bytes and 1312-2592 bytes, respectively. In comparison, the public key sizes of RSA typically range from 128-512 bytes [BR96]. However, the demand for stringent security measures in IoT devices has made efficient solutions tailored to such devices increasingly critical. In the IoT domain, where many devices are interconnected and communicate with each other, they are more vulnerable to attacks. Consequently, protecting them against quantum computers' attacks is challenging due to limited hardware resources, low power consumption requirements, and the need for flexibility in a wide range of applications.

Low power consumption, high efficiency, small chip area, and a certain level of flexibility are the focuses for IoT applications. However, current state-of-the-art works (including FPGA, ASIC, co-processor, processor designs) have not successfully combined all these aspects into a single design solution. The limitation of existing FPGA and ASIC works lies in their large hardware consumption and lack of flexibility to accommodate diverse schemes. Although several FPGA and ASIC designs [KLC+17, STCZ18, ZZZ+22] prioritize speed, they are typically limited to specific algorithms with high power and resource consumption. State-of-the-art co-processor and processor designs [BUC19, FSS20, XHY+20, AMI+22, KSFS23, ZXXH22] accelerated LBC schemes by designing customized hardware accelerators for critical operations with highly parallel architectures and some specific parameter settings. However, these hardware accelerators usually have limited running frequency and large power consumption, posing significant overhead for battery-powered IoT devices. For instance, [XHY+20] and [ZZZ+22] proposed high-efficiency and flexible designs; however, they come at the cost of larger power consumption and chip area requirements. On the other hand, these customized hardware accelerators are often unable to efficiently accelerate fine-grained operations such as polynomial compress/decompress, logic shiftings, and more. [BUC19] introduced a low-power design, but it lacks the desired level of flexibility and cannot support the acceleration of polynomial compress/decompress. To address the efficiency, low costs, and flexibility requirements, this paper presents a solution using instruction set architecture level and algorithmic-level co-optimization.

Our Contributions. In this paper, we present an efficient Lattice-based Cryptography processor based on a customized RISC-V Single-Instruction-Multiple-Data (SIMD) architecture. We focus on two lattice-based schemes CRYSTALS-Kyber and CRYSTALS-Dilithium. To address the requirements of not only power consumption but also computing speed, chip area, and flexibility, we propose a novel SIMD hardware architecture for PQC in IoT applications, including dedicated designs and optimizations for efficient parallel computations, resource-reusable design, and memory access. Moreover, we propose algorithmic-level and architectural-level optimizations with our proposed fine-grained SIMD instructions for efficient polynomial multiplications. The detailed technical aspects of our work are as follows:

- Flexibility through fine-grained resource reuse. Our proposed processor is designed with two SIMD working modes (320-bit and 256-bit) and resource-sharing hardware units for various arithmetic and logic operations, including additions, exclusive-or, multiplications, etc.

- Efficient computing with parallelism. Our proposed fine-grained SIMD instructions enable efficient parallel computations of various arithmetic and logic operations using 256-bit mode and accelerate Keccak computation using 320-bit mode. Small operations, including polynomial compression, pack, logic shifting, modular reduction, etc. that can be vectorized are accelerated with our SIMD instructions.

- Efficient computing with efficient memory access. To resolve data dependency issues within SIMD data, data shuffling hardware units are designed and integrated into the proposed processor. Moreover, a dual-issue path is proposed to achieve efficient memory access by reducing data blocking during computation. Computation instructions are executed simultaneously with memory access instructions. With the proposed dual-issue design, we further optimize the performance at the algorithm level.

2 Background

This section introduces the LBC schemes, including CRYSTALS-Kyber and CRYSTALS-Dilithium, which belong to Key Encapsulation Mechanism (KEM) and Digital Signature (DS), respectively. Furthermore, we give a brief introduction of the bottlenecks operations in these two schemes, including hashing, polynomial multiplications, and polynomial sampling. Polynomial vectors are denoted in bold lower-case, and polynomial matrices are denoted in bold upper-case. The quotient ring of integers modulo q is denoted as $\mathbb{Z}_q$, and the polynomial ring with degree n and modulo q is denoted as $\mathbb{Z}_q[x]/(x^n + 1)$.

2.1 CRYSTALS-Kyber

CRYSTALS-Kyber [BDK+18] is an IND-CCA2-secure KEM constructed from Kyber.CPAPKE (IND-CPA-secure Public Key Encryption (PKE) scheme) using tweaked Fujisaki-Okamoto (FO) transform [FO99]. FO transform is achieved by using a random oracle (hash function) to generate a shared secret, encapsulating and decapsulating the shared secret, and deriving the symmetric key and final shared secret using a Key Derivation Function (KDF). There are three security levels for CRYSTALS-Kyber, and the corresponding parameters are shown in Table 1. n is the polynomial length, k is the dimension of the polynomial vector, q is the modulo, η_1 and η_2 are used for Centered Binomial Distribution (CBD) sampling. For more details of CRYSTALS-Kyber, readers can refer to [BDK+18].

Table 1: Parameter sets for CRYSTALS-Kyber.

NIST Security Level	n	k	q	η_1	η_2
1	256	2	3329	3	2
3	256	3	3329	2	2
5	256	4	3329	2	2

2.2 CRYSTALS-Dilithium

As a lattice-based digital signature scheme, CRYSTALS-Dilithium [DLL+18] uses polynomials with the same length 256 as CRYSTALS-Kyber while the modulo $q = 8380417$ is much larger. Table 2 summarizes the parameters used in Dilithium for different security levels. The letters k and ℓ are the dimensions of polynomial matrices, and η is the secret key polynomial coefficients range. In digital signature protocol, there are two parties (signer and verifier) and three main operations, including key generation, signing, and verification. Key generation creates a public and secret key pair in polynomial form. Signing combines a message hash, private key, and public key to create a signature with a witness and hint polynomial; Verification checks the signature's correctness using the signer's public key and the message hash. For more details of CRYSTALS-Dilithium, readers can refer to [DLL+18].

Table 2: Parameter sets for CRYSTALS-Dilithium.

NIST Security Level	q	(k, ℓ)	η
1	8380417	(4, 4)	2
3	8380417	(6, 5)	4
5	8380417	(8, 7)	2

2.3 Bottlenecks Operations

Hash Function. In the FO transform of KEM and the computation of DS, hash functions play a crucial role, and it has been shown that the hash function occupies more than 40% of the running cycles in Kyber, Dilithium, and many other schemes in the ARM platform [KRSS19]. Currently, Secure Hash Algorithm 3 (SHA-3) is widely used as a hash function, which was selected as the winner of the NIST hash function competition in 2012 [CPB+12]. SHA-3 uses a sponge construction - Keccak, where the input message is padded and then absorbed into a state array. The state array is then processed using a series of permutation rounds to provide diffusion and confusion to the input message. After processing all inputs, the state array is squeezed to produce the output hash value. To accelerate SHA-3, the most critical part is the Keccak-f[1600], which accepts 1600 bits as inputs and generates 1600 bits as outputs. Keccak-f[1600] consists of 24 rounds with five steps $(\theta, \rho, \pi, \chi, \iota)$ in each round, as described in Algorithm 1.

Accelerating Keccak using 256-bit SIMD instructions is challenging due to data dependencies in the ρ and π steps as a Keccak state matrix involves $5 \times$ 64-bit data. The presence of an extra 64-bit data requires additional instructions to handle it. The work [BDPVA13] utilized Intel AVX instructions to accelerate Keccak. Another work [RS16] proposed customized SIMD instructions for Keccak with 128 and 256 bits that are combined with multiple operations. However, their efficiency is hindered when using 128 or 256-bit registers due to the data dependencies.

Polynomial Multiplication and Number Theoretic Transform. Polynomial multiplication is another bottleneck operation in KEM and DS. For example, the key generation and verification process of Dilithium 2 involves more than 16 polynomial multiplications. In polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$, the complexity can be reduced from $O(n^2)$ to $O(n \log n)$ by using Number Theoretic Transform (NTT), polynomial multiplication still accounts for approximately 50% of running cycles [BNAMK21].

NTT is a Discrete Fourier Transform (DFT) variant where all numbers are integers, and modular reduction is applied in all operations. NTT takes a polynomial $a(x) = \sum_{i=0}^{n-1} a_i x^i$ as input and returns the output $A(x) = \sum_{i=0}^{n-1} A_i x^i$, where $A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \pmod q$ and ω is the primitive n-th root-of-unity modulo q. For INTT, the twiddle factor ω is

Algorithm 1 Plane-per-Plane processing of Keccak-f [BDPVA13]

```
Require: State matrix $A$ ($5 \times 5 \times 64$ bits)
Ensure: State matrix $E$ ($5 \times 5 \times 64$ bits)
1: for $x = 0$ to 4 do
2:     $C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4]$ // $\theta$ step
3: end for
4: for $x = 0$ to 4 do
5:     $D[x] = C[x-1] \oplus \mathrm{ROT}(C[x+1], 1)$ // $\theta$ step
6: end for
7: for $y = 0$ to 4 do
8:     for $x = 0$ to 4 do
9:         $B[x] = \mathrm{ROT}((A[x',y'] \oplus D[x']), r[x',y'])$, with $\begin{pmatrix} x' \\ y' \end{pmatrix} = M^{-1} \begin{pmatrix} x \\ y \end{pmatrix}$ // $\rho$ and $\pi$ step
10:    end for
11:    for $x = 0$ to 4 do
12:        $E[x,y] = B[x] \oplus ((\mathrm{NOT}\, B[x+1])\, \mathrm{AND}\, B[x+2])$ // $\chi$ step
13:    end for
14: end for
15: $E[0,0] = E[0,0] \oplus \mathrm{RC}[i]$ // $\iota$ step
```

replaced with the multiplicative inverse ω^{-1}, and additional final scaling n^{-1} is required: $a_i = N^{-1} \sum_{j=0}^{n-1} A_j \omega^{-ij} \pmod{q}$. Similar to DFT, a divide-and-conquer method can be used to reduce the complexity of NTT to $O(n \cdot \log(n))$.

Polynomial Sampling. Two types of random sampling are involved in LBC algorithms: rejection sampling and binomial sampling. In polynomial samplings, additions, subtractions, and other logic operations are performed on random uniform numbers generated by a Pseudo Random Number Generator (PRNG), which involves output data from a hash function. Though these operations can be performed efficiently in ASIC and FPGA designs, they require several logic and arithmetic operations in a processor, resulting in critical running cycles.

In the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$, rejection sampling is used to ensure that the coefficients are within the range $[0, q-1]$. The process of rejection sampling can be described as follows:

- Generate a uniformly sampled data point x from $[0, 2^k - 1]$, where $k > \log q$.
- Check if x falls within the desired range of $[0, q-1]$. It is accepted as a valid sample if x is within this range.
- If x is outside the desired range, it is rejected. In this case, the above steps are repeated until a valid sample within $[0, q-1]$ is obtained.

On the other hand, centered binomial distribution sampling is used for secret and error polynomials where the coefficients are within a small range $(-\eta, \eta)$. The process of CBD sampling is shown as follows:

- Sample 1-bit data: $(a_1, ..., a_\eta, b_1, ..., b_\eta) \leftarrow \{0,1\}^{2\eta}$
- Output: $\sum_{i=1}^{\eta}(a_i - b_i)$

3 Hardware Architecture

In this section, our proposed SIMD architecture is presented. The proposed processor is based on CV32E40P[2], which is a 32-bit, in-order RISC-V core with a 4-stage pipeline and

[2]https://github.com/openhwgroup/cv32e40p

RV32IMC instruction set architecture. To optimize the performance of LBC algorithms, we propose some customized SIMD instructions and corresponding hardware designs. The data width of SIMD can be configured to 256-bit mode (eight parallel 32-bit arithmetic operations) and 320-bit mode (ten parallel 32-bit Keccak operations).

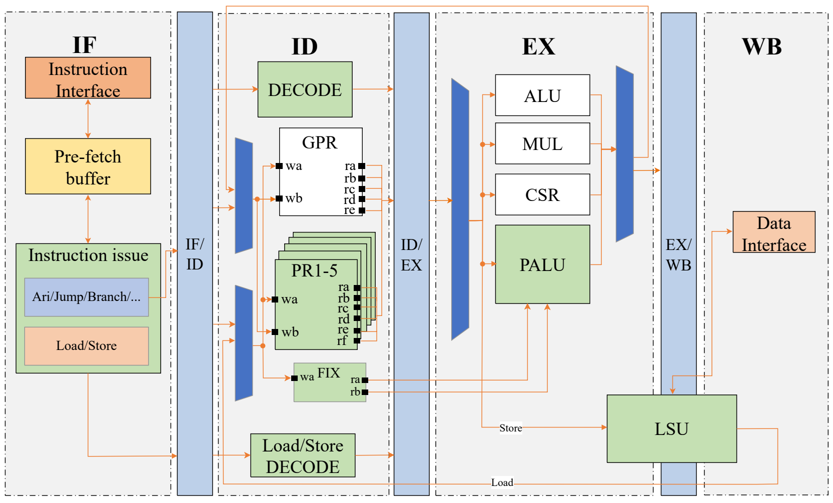


Figure 1: Proposed SIMD Processor Architecture.

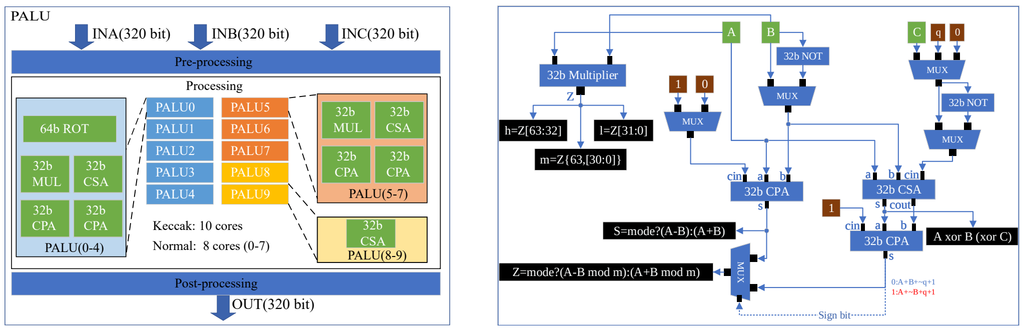
3.1 Proposed Instruction Set Architecture

As illustrated in Figure 1, the processor consists of four pipeline stages, including Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), and Write-Back (WB). To accelerate the LBC algorithms, we apply several optimizations in our hardware designs:

SIMD Register File. 5 SIMD register files (**PR1** to **PR5**) consisting of 16 rows of 64 bits are added in the ID stage. 8×32-bit or 10×32-bit data are stored in the register files PR1 to PR5. Moreover, a small register file (**FIX**) consisting of 2 rows of 32 bits is added to store frequently used constant numbers, such as parameters q and q^{-1} in Kyber and Dilithium.

Proposed ALU Design. A **PALU** is added in EX stage to support 8/10 parallel operations of 32-bit data. Inside the EX stage, there are pre-processing and post-processing hardware units for data shuffling. Figure 2a shows our proposed 10-core PALU design, which is capable of receiving up to three 320-bit data from PRs and outputting up to one 320-bit data. These cores can be classified into three types with different hardware resources of Carry Propagate Adders (CPA), Carry Save Adder (CSA), multiplier, and rotator. When executing SIMD Keccak instructions, all ten cores in PALU can be utilized, whereas when executing SIMD arithmetic instructions, only PALU0 to PALU7 are utilized.

Load/Store Extensions. To minimize the overhead of data transmission between the processor and main memory, we opt for a 128-bit data path in the Load-Store Unit (LSU) because a 256-bit LSU will reduce the flexibility of data organization and consume much more resources. When working on load/store instructions, data will be loaded to or stored from (PR1, PR2) or (PR3, PR4) or PR5.



(a) Proposed PALU design.   **(b)** Data path of PALU0 to PALU7.

Figure 2: Proposed PALU design and data path.

Dual-Issue. Since our PALU can process 256/320-bit data, and the LSU only supports at most 128 bits, loading data from and storing data back to the main memory may become a bottleneck. To enhance the performance, the IF data path is extended from 32 bits to 64 bits. With this, we propose a dual-issue design where two 32-bit instructions (one load/store and the other non-load/store) are fetched and executed simultaneously. If there are data dependencies, more than two load/store instructions, more than two non-load/store instructions, or a jump/branch instruction, only one instruction will be issued and executed. Consequently, both the program and compiler must be meticulously designed to leverage this feature to optimize performance. It is noted that our proposed dual-issue design works for both SIMD instructions and RV32 instructions.

3.2 Proposed SIMD Instructions

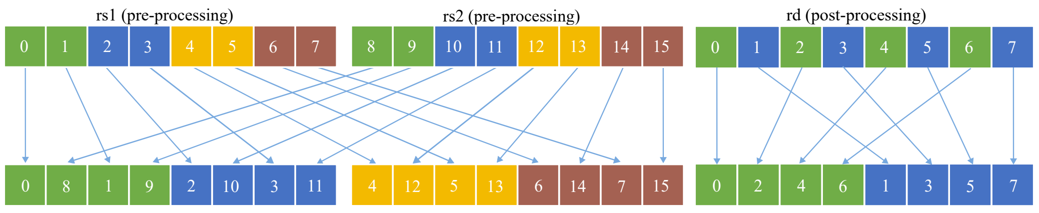
Our proposed SIMD instructions include arithmetic instructions, Keccak instructions, and load/store instructions, as described in Table 10.

3.2.1 SIMD Arithmetic Instructions

Our PALU supports a variety of arithmetic operations since it includes multipliers, adders, and rotators. For logic operations such as AND and XOR, adders are utilized. Additionally, 32-bit logic left-shift and 32-bit arithmetic right-shift are supported using rotators. Modular additions and subtractions require one more addition/subtraction than ordinary addition and subtraction because we assume the inputs are in the range $(-q, q)$. The detailed data path of PALU executing these instructions is illustrated in Figure 2b.

Data Shuffling. Data shuffling units are integrated into the pre-processing and post-processing stages of PALU to rearrange sets of 32-bit data for efficient operations. As depicted in Figure 3, 16 sets of 32-bit data are reordered during pre-processing (**input shuffling**), and 8 sets of data from the PALU (**output shuffling**) are reordered during post-processing. This approach helps to reduce data dependencies' impact on SIMD operations and improve overall performance. In addition to these two shuffling patterns, it is also possible for two 256-bit data to exchange their halves by utilizing load/store instructions. This can be achieved by loading the upper and lower halves of both 256-bit data into separate registers, exchanging the lower halves using a store instruction followed by a load instruction, and then similarly swapping the upper halves.

Rejection Sampling. To support rejection sampling for 8 sets of 32-bit data, we design a specific instruction called `bgeuv`. This instruction compares each set of 32-bit data in `rs1`



(a) Input shuffling **(b)** Output shuffling

Figure 3: Data shuffling.

to the corresponding set of 32-bit data in `rs2` and provides a branch indication if any of the comparisons show that one of the 32-bit data of `rs1` is greater than that of `rs2`.

Centered Binomial Distribution Sampling. `cbd2` and `cbd3` are designed for efficient parallel CBD sampling for $\eta = 2$ and $\eta = 3$, respectively. The data path of `cbd2` and `cbd3` are illustrated in Figure 4, where the lower 32/48 bits of `rs1` are utilized. In the case of `cbd2`, the lower 32 bits of `rs1` are split into 32 parts, and each part is extended to 4 bits by padding 3 bits of 0. To avoid introducing additional adders in PALU, these extended bits are further grouped into four 32-bit numbers: A0, A1, B0, and B1. Additions of A = A0 + A1 and B = B0 + B1 are performed in the CPAs of PALU0 and PALU1 without overflow because the result range is $[0, 2]$, which reduces the number of required adders from 16 to 2. The results A and B are then split into eight parts and sent to the CPAs of PALU0 to PALU7. Finally, the subtractions A − B in 8 CPAs are performed, and the results are stored in 256 bits. The instruction `cbd3` is similar to `cbd2` except that the lower 48 bits of `rs1` are used and split into A0 to A2 and B0 to B2.

To perform CBD sampling with $\eta > 3$, one can combine `cbd2` and `cbd3` instructions. For instance, the sum of `cbd2` and `cbd3` can form an $\eta = 5$ sampling process, and the sum of `cbd2` and `cbd2` can form an $\eta = 4$ sampling process.

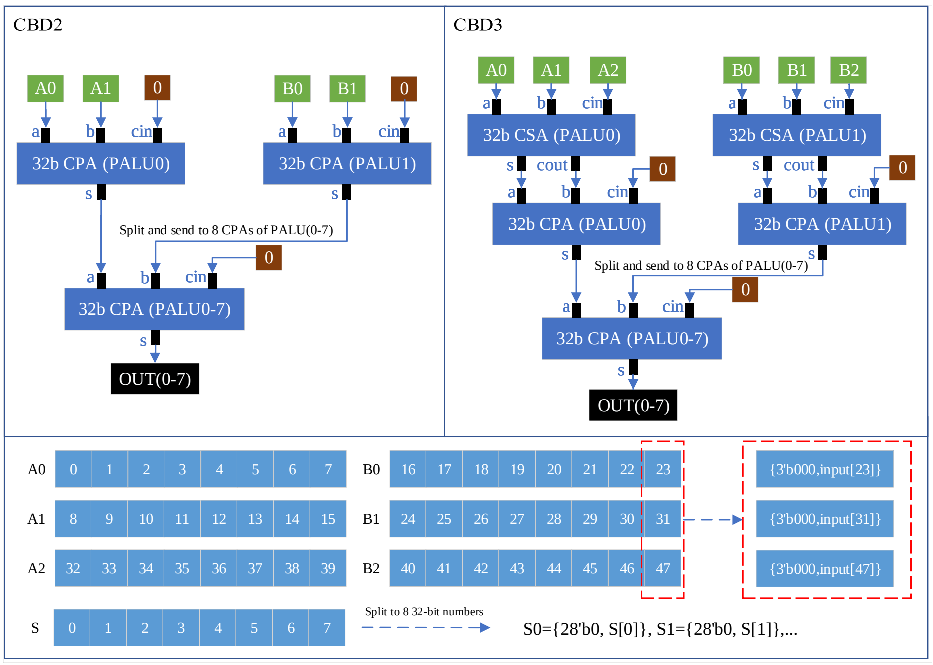


Figure 4: Datapath of `cbd2` and `cbd3`.

3.2.2 SIMD Keccak Instructions

According to our experimental results, it takes about 13,000 cycles to perform one Keccak operation on CV32E40P. To perform Keccak efficiently, we extend the datapath from 256 bits to 320 bits and allow more read ports in the SIMD register files, enabling the processing of five 64-bit data simultaneously. Following the steps of Algorithm 1, we design a set of SIMD Keccak instructions combining XOR, 64-bit rotations, AND and data shuffling to accelerate Keccak (Table 10). The detailed operations of our proposed SIMD Keccak instructions are illustrated in Figure 5.

To reduce the number of instructions and cycles of Keccak, some instructions are designed to allow three-operand processing. As supporting `rs3` directly would increase hardware complexity and power consumption, we restrict the address of `rs3` to be the address of `rs1` with a left-shift of 1 bit, i.e., $addr[rs3] = addr[rs1] << 1$. `xorv3` computes the XOR result of `rs1`, `rs2`, and `rs3`, while `xorv2` computes the XOR result of `rs1` and `rs2`. `xorrv` broadcasts the data `C` of Algorithm 1 to PALU0 to PALU9 and performs an 1-bit rotation.

Instructions `rxorv0` to `rxorv4` are used to compute B in Algorithm 1. They compute the XOR result of `rs1` and `rs2` and then perform 64-bit rotations. There are a total of 25 rotations in the ρ step of Keccak. To support the rotations with different numbers, the rotators are configured to specific rotation numbers during the executions of `rxorv0` to `rxorv4`, as shown in Table 3.

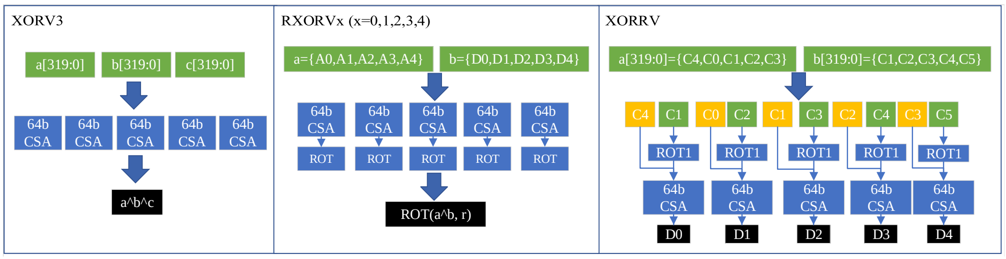


Figure 5: Keccak Instructions Data path. A 64-bit CSA is two 32-bit CSAs in the PALU.

Table 3: Rotators configurations in different cores.

	PALU0	PALU1	PALU2	PALU3	PALU4
rxorv0	27	28	62	1	0
rxorv1	20	55	6	44	36
rxorv2	39	25	43	10	3
rxorv3	8	21	15	45	41
rxorv4	14	56	61	2	18

`shufflev` is designed to allow internal data shuffling which accepts `rs1` and `imm` as inputs. For example, when imm=1, the data input as [`rs1`[0], `rs1`[1], `rs1`[2], `rs1`[3], `rs1`[4]] will give output `rd`=[`rs1`[4], `rs1`[0], `rs1`[1], `rs1`[2], `rs1`[3]]. `xornavi` is used to compute `E` in Algorithm 1, which combines logic NOT, AND, and XOR operations. To handle the data dependencies problem, we introduce `rs4` and `rs5`, whose addresses are fixed in all cases. Additionally, `xornavi` requires a 3-bit immediate value `imm` where $0 \leq \text{imm} \leq 5$. Although there is no `rs2` in the RISC-V I type instruction, we still use $instr[24:20]$ to represent `rs2`, and $instr[27:25]$ is used as the 3-bit immediate value in hardware design.

3.2.3 SIMD Load/Store Instructions

The proposed SIMD load/store instructions are designed for 32-bit register file FIX and 64-bit register files PRs:

- `lv` and `sv` are used for SIMD PR1 to PR4.
- `lw64` and `sw64` are used for SIMD PR5.
- `lwf` is used for SIMD FIX.

Note that when dealing with load/store instructions, `rs1` comes from the General Purpose Register (GPR).

4 Design Methodology

In this section, the design methodologies using our proposed hardware and corresponding customized SIMD instructions are proposed. To enhance performance, our proposed dual-issue feature is utilized, which provides significant performance improvements in programs with fewer data dependencies. Moreover, algorithmic-level and architectural-level co-optimizations for polynomial multiplications are presented to further enhance the performance.

4.1 Keccak Acceleration

The assembly code for Keccak using the proposed instructions is described in Listing 1, following the steps of plane-per-plane processing (Algorithm 1). For simplicity, the data loading and storing are ignored here. Note that registers `x0` to `x31` can function as either GPR or SIMD PR, which are distinguished by hardware decoder.

Listing 1: Keccak assembly code.

```
// load data to x0, x1, x2, x3, x6
// using lv, lw64 and shufflev ...

// Round 0
// compute C
xorv3 x5, x1, x0
xorv3 x5, x3, x5
// compute D
xorrv x5, x5, x5
// compute B
rxorv1 x10, x1, x5
rxorv2 x11, x2, x5
rxorv3 x12, x3, x5
rxorv4 x13, x6, x5
rxorv0  x9, x0, x5
// compute E part 1
xornavi x15, 0b001010(x9)
xornavi  x4, 0b1101010(x9)
xornavi  x8, 0b101010(x9)
// compute RC
lw64 x5, 0(%[round_consts])
xorv2rc x15, x15, x5
// compute E part 2
xornavi  x7, 0b10001010(x9)
xornavi x14, 0b1001010(x9)
// Round 1
```

```
// compute C
xorv3 x5, x4, x15
xorv3 x5, x7, x5
// compute D
xorrv x5, x5, x5
// compute B
rxorv1 x10,  x4, x5
rxorv2 x11,  x8, x5
rxorv3 x12,  x7, x5
rxorv4 x13, x14, x5
rxorv0  x9, x15, x5
// compute E part 1
xornavi x0, 0b001010(x9)
xornavi x1, 0b1101010(x9)
xornavi x2, 0b101010(x9)
// compute RC
lw64 x5, 8(%[round_consts])
xorv2rc x0, x0, x5
// compute E part 2
xornavi x3, 0b10001010(x9)
xornavi x6, 0b1001010(x9)

// 22 rounds ...

// store data back
// using sv, sw64 and shufflev ...
```

As Algorithm 1 is not in-place, the input and output are stored in different memory addresses. To optimize memory, after every two rounds, the output of the second round is stored back in the same address as the input of the first round. As 5×5 64-bit elements of the Keccak state matrix are assumed to be stored in continuous memory address by columns, some elements that are aligned to 8 bytes are loaded using `lw64` instructions, and elements that are aligned to 16 bytes are loaded using `lv` instructions. To ensure that the elements in the PRs have consistent indices, `shufflev` instructions are used after data loading, as shown in Figure 6. The process of storing the state matrix in PRs back to the main memory is the reversal of the loading process. There is no other load/store of the state matrix inside the 24 rounds operations because the memory space of PRs is sufficient. The operations of Round 0 and Round 1 are shown in lines 4 to 64 of Listing 1. To reduce

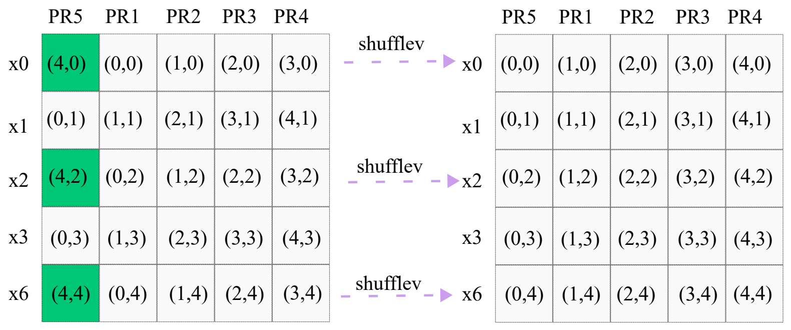


Figure 6: The organization of Keccak state matrix in PRs.

memory consumption, the output `E` of Round 1 is stored in the address of the input `A` of Round 0. Similarly, the following 22 rounds repeat this process. Note that instead of unrolling all 24 rounds, we implement a loop to handle the 24 rounds to reduce the code size. Benefiting from our proposed design, the number of cycles of one Keccak operation is 404, which is about 32× faster than that of the CV32E40P processor (Table 5).

4.2 Modular Arithmetic

Our proposed processor supports modular additions and subtractions of two operands within range $(-q, q)$ using the `addvm` and `subvm` instructions. However, modular multiplications are more complex and require several instructions. The most commonly used algorithms for modular multiplication and reduction are the Barrett algorithm [Bar86] and the Montgomery algorithm [Mon85]. However, if the modulus $q > 2^{16}$, the subtraction of [Bar86] would be more than 32 bits, which introduces additional computational complexity. On the other hand, both of these algorithms are incompatible with signed number arithmetic, which poses challenges when the CBD sampling generates signed numbers.

To address these issues and provide more flexibility, signed arithmetic is supported in our proposed hardware, and signed Montgomery algorithm [Sei18] is implemented for modular reduction. As shown Algorithm 2, parameters β, β^{-1}, q, and q^{-1} are required and we choose $\beta = 2^{32}$. The input range of a is $[-\frac{\beta}{2}q, \frac{\beta}{2}q)$, and the output r' is in the range $(-q, q)$. Because this algorithm can not compute $a \cdot b \bmod q$ directly, a or b should be multiplied with β before signed Montgomery reduction. With our proposed instruction, modular multiplication can be implemented in 5 instructions, as shown in Listing 2.

Listing 2: Assembly code of modular multiplication.

```
// load data to x0 and x1 using lv ...
// load constant data q and q^{-1} to FIX using lwf ...
// modular multiplication
mulv   x14,  x0,  x1 // lower 32-bit result a0
mulvh  x13,  x0,  x1 // higher 32-bit result a1
mulvm  x14, x14, x14 // a0 * q^{-1} mod beta
mulvhf x14, x14, x14 // m * q / beta
subv    x2, x13, x14 // a1 - t1
// store back using sv ...
```

Algorithm 2 Signed Montgomery Reduction [Sei18]

```
Require: $0 < q < \frac{\beta}{2}, -\frac{\beta}{2}q \le a = a_1\beta + a_0 < \frac{\beta}{2}q$ where $\beta \cdot \beta^{-1} \equiv 1 \pmod q, 0 \le a_0 < \beta$
Ensure: $r' = \beta^{-1}a \pmod q, -q < r' < q$
  1: $m \leftarrow a_0 q^{-1} \bmod {}^{\pm}\beta$
  2: $t_1 \leftarrow \lfloor \frac{mq}{\beta} \rfloor$
  3: $r' \leftarrow a_1 - t_1$
```

4.3 Number Theoretic Transform

NTT is a bottleneck operation with the complexity of $O(n \cdot \log(n))$ that is commonly used to accelerate polynomial multiplications. Gentleman-Sande NTT and Cooley-Tukey INTT [CT65] are optimized and implemented with the proposed instructions in our design. The Gentleman-Sande scheme performs modular multiplication after additions, whereas the Cooley-Tukey scheme performs modular multiplication at first. With our proposed SIMD instructions, eight butterfly operations are performed simultaneously. To deal with data dependencies during different layers, input shuffling and output shuffling schemes are utilized. To explain how NTT and INTT are performed, we take a 32-point polynomial

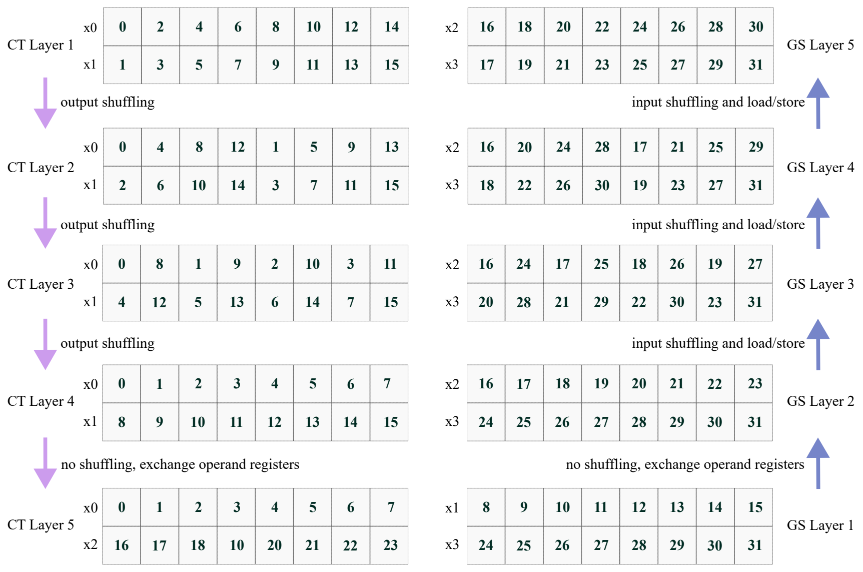

CT Layer 1
x0 0 2 4 6 8 10 12 14
x1 1 3 5 7 9 11 13 15
x2 16 18 20 22 24 26 28 30
x3 17 19 21 23 25 27 29 31
GS Layer 5
output shuffling
input shuffling and load/store
CT Layer 2
x0 0 4 8 12 1 5 9 13
x1 2 6 10 14 3 7 11 15
x2 16 20 24 28 17 21 25 29
x3 18 22 26 30 19 23 27 31
GS Layer 4
output shuffling
input shuffling and load/store
CT Layer 3
x0 0 8 1 9 2 10 3 11
x1 4 12 5 13 6 14 7 15
x2 16 24 17 25 18 26 19 27
x3 20 28 21 29 22 30 23 31
GS Layer 3
output shuffling
input shuffling and load/store
CT Layer 4
x0 0 1 2 3 4 5 6 7
x1 8 9 10 11 12 13 14 15
x2 16 17 18 19 20 21 22 23
x3 24 25 26 27 28 29 30 31
GS Layer 2
no shuffling, exchange operand registers
no shuffling, exchange operand registers
CT Layer 5
x0 0 1 2 3 4 5 6 7
x2 16 17 18 10 20 21 22 23
x1 8 9 10 11 12 13 14 15
x3 24 25 26 27 28 29 30 31
GS Layer 1

Figure 7: 32 points example of NTT and INTT.

as an example, as shown in Figure 7. Note that we have `addvmt`, `subvmt`, `addvti`, and `subvti` instructions to perform data shuffling. The numbers in the boxes are the indices of polynomial coefficients where 32 coefficients are stored in 4 SIMD registers. In CT Layer 1 to 4 and GS Layer 2 to 5, `x0` and `x1` are the operands of the butterfly operation; in CT Layer 5 and GS Layer 1, `x0` and `x2` are the operands of the butterfly operation. Data shufflings are applied between Layer 1 and 2, Layer 2 and 3, and Layer 3 and 4. The data shuffling of Cooley-Tukey data flow is performed by output shuffling (as shown in Figure 3) and load/store instructions, while the data shuffling of Gentleman-Sande data flow is performed by input shuffling. By utilizing our proposed SIMD instructions, NTT can be accelerated by about 8×. To further enhance the performance, dual-issue, lazy reduction, and twiddle factor sharing are utilized.

Dual-issue. There are many load/store during NTT computation because the SIMD PR can only store 16 × 256-bit data, and twiddle factors also need to be loaded into

SIMD PRs. To improve performance, we leverage the limited data dependencies within a layer of NTT to maximize the benefits of our dual-issue scheme. One SIMD butterfly operation requires 7 instructions, including 5 instructions for modular multiplication, 1 for modular addition, and 1 for modular subtraction. On the other hand, the data for one SIMD butterfly operation requires 6 load instructions (butterfly operands and twiddle factors) and 2 store instructions. In total, 7 instructions are used for computation, and 8 instructions are used for load/store. The number of load/store instructions can be reduced to 6 when the update of the twiddle factors is not necessary.

To utilize dual-issue, we apply interleaving of butterfly instructions and load/store instructions in programming. The current butterfly operations are mixed with proceeding data storing or succeeding data loading. In this case, NTT/INTT can be executed efficiently. The assembly codes of 256-pt Gentleman-Sande NTT are shown in Listing 3. The required data for the first butterfly operation is loaded (lines 1 to 6), and dual-issue is applied in lines 8 to 27. Due to the advantage of dual-issue, the number of cycles for lines 8 to 27 is 10 rather than 17.

Lazy Reduction. Lazy reduction is commonly used to reduce the number of instructions required to perform modular additions and subtractions [HZZ+22]. In our design, we have instructions `addvm` and `subvm` that support modular additions and subtractions directly, and our proposed instructions `addvmt` and `subvmt` support modular additions and subtractions with output shuffling. To minimize the number of new instructions, we do not introduce instructions that support modular additions and subtractions with input shuffling. In our design, we utilize lazy reduction in all layers of Gentleman-Sande NTT for both Kyber and Dilithium. Lazy reduction can be applied because Algorithm 2 can accept data in the range $[-\frac{\beta}{2}q, \frac{\beta}{2}q)$ and limit the output to the range $(-q, q)$. As the multiplicands used in the modular multiplications of NTT/INTT are pre-computed twiddle factors, their values can be constrained to the range of $[-\frac{q}{2}, \frac{q}{2})$. Consequently, the range of the other operands is limited to $(-\beta, \beta)$. After performing one lazy reduction, the output range increases by the value of q [Sei18]. Since the polynomial coefficients are in the range $(-q, q)$ at first, after operations of 8 layers with lazy reduction, the range becomes $(-9q, 9q)$, which is still a subset of $[-2^{31}, 2^{31} - 1]$ for both Kyber and Dilithium.

Listing 3: Assembly code of Gentleman-Sande NTT.

```
1  lv   x0,   0(%[tws])     // load twiddle factors
2  lv  x16,  16(%[tws])
3  lv   x1, 512(%[coeffs]) // load coeffs[128-131] for BF1
4  lv  x17, 528(%[coeffs]) // load coeffs[132-135] for BF1
5  lv   x2,   0(%[coeffs]) // load coeffs[0-3] for BF1
6  lv  x18,  16(%[coeffs]) // load coeffs[4-7] for BF1
7
8  // Dual-issue: interleaving of load/store and arithmetic instructions
9  lv   x6, 544(%[coeffs]) // load coeffs[136-139] for BF2
10 mulv x15, x0, x1
11 lv  x22, 560(%[coeffs]) // load coeffs[140-143] for BF2
12 mulvh x3, x0, x1
13 // twiddle factors sharing: only update when necessary
14 lv   x0,  32(%[tws])
15 lv  x16,  48(%[tws])
16
17 lv   x7,  32(%[coeffs]) // load coeffs[8-11] for BF2
18 mulvm  x14, x15, x0
19 lv  x23,  48(%[coeffs]) // load coeffs[12-15] for BF2
20 mulvhf x13, x14, x0
21 lv   x1, 576(%[coeffs]) // load coeffs[144-147] for BF3
22 subv x12, x3, x13
23 lv  x17, 592(%[coeffs]) // load coeffs[148-151] for BF3
24 addvm x4, x2, x12
25 sv   x4,   0(%[coeffs]) // save coeffs[0-3] for BF1
26 subvm x5, x2, x12
27 sv  x20,  16(%[coeffs]) // save coeffs[4-7] for BF1
28 // ...
```

Twiddle Factors Sharing. To further enhance performance, we reduce the number of twiddle factors loading by observing that the number of different twiddle factors varies across different layers. Although the number of required twiddle factors for one layer is $\frac{n}{2}$, some twiddle factors are the same in some layers. In the case of Gentleman-Sande, the number of different twiddle factors decreases as the layer increases, while in Cooley-Tukey INTT, the number of different twiddle factors increases with the layer. Therefore, the twiddle factors are loaded into dedicated SIMD registers at first and will be updated when different twiddle factors are used. In layers with fewer unique twiddle factors, the same factors can be shared multiple times, which eliminates the need to continuously reload them from the main memory.

By applying the methodologies described above, our proposed design achieves a cycle count of 1750 for one 256-point NTT and 1925 for one 256-point INTT. These results represent a significant improvement (25× faster) over the cycle counts achieved by CV32E40P processor, as shown in Table 5. Some NTT-only designs [RVM+14, FLX19, LTHW22, YCH22] that can not perform Kyber KEM and Dilithium could have an NTT cycle count of 256, while our NTT cycle count is 1.7k because only some simple SIMD arithmetic instructions are added to our proposed processor. Only the proposed shuffling hardware which is composed of some MUXs is designed specifically for NTT. The other proposed hardware designs (PALU, dual-issue, etc.) are not only optimized for NTT but also for many other operations including Keccak, polynomial samplings, etc.

4.4 Polynomial Multiplication

In Dilithium, polynomial multiplication can be accelerated by Negacyclic Convolution, where two NTT, one point-wise multiplication, and one INTT are performed [CMV+14]. However, due to the limitation of parameter settings in Kyber, the polynomial multiplication is different from Dilithium, and in-completed NTT [XL21] is utilized. Unlike the approach in [XL21], we split Kyber's 256 polynomial coefficients into even and odd parts and perform NTT/INTT transformations independently. In this case, the design methodologies for general NTT are utilized in Kyber 128-pt NTT. Moreover, we propose a method to reduce the number of modular reductions in Kyber polynomial multiplications and polynomial matrix-vector multiplications. The details of our optimized Kyber polynomial multiplication are depicted in Algorithm 3 where `BR` is the bit-reversal operation. To further enhance the performance of polynomial multiplications, operations merging and dual-issue are applied.

Optimization for Modular Multiplication. In Kyber, there are 5 modular multiplications in one basecase multiplication (lines 5 and 6 in Algorithm 3), which requires 25 instructions. To reduce the complexity, we replace some modular multiplications with ordinary multiplications.

Since we use lazy reduction in Gentleman-Sande NTT, the output coefficients are in the range of $(-8q, 8q)$. When performing the operations in line 6 using ordinary arithmetic, the resulting values fall within the range of $(-65q^2, 65q^2)$, which remains within the limits of a 32-bit representation. Thus, instead of performing two modular multiplications and one modular addition as required in line 6, we just perform two ordinary multiplications, one ordinary addition, and one additional modular reduction. For line 5, the modular multiplication of $B'_o(x) \odot D$ is performed first, followed by similar processing as line 6. In this case, the operations of line 5 can be reduced to one modular multiplication, two ordinary multiplications, one ordinary addition, and one modular reduction.

Additionally, this technique can also be used in matrix multiplications where elements are polynomials in Kyber and Dilithium. When the outputs of ordinary operations are in the 32-bit range, matrix multiplications can use ordinary multiplications first. A modular reduction is performed after the summation of all polynomials in the same row/column.

This technique significantly reduces the number of modular multiplications and improves the overall performance.

Operations Merging. When using SIMD instructions for computation, data needs to be loaded from the main memory to SIMD PR and then stored back to the main memory, resulting in a significant overhead in memory accesses, particularly when there are large amounts of data. To address this issue, we propose an operation merging technique that reduces the number of data load/store by reorganizing more operations into one SIMD computation.

In polynomial multiplications, we take advantage of the fact that the twiddle factors of the last layer in Gentleman-Sande NTT and the first layer in Cooley-Tukey INTT are ones, making modular multiplication free. This reduces the number of instructions for one butterfly operation to only 2. Therefore, we merge more operations into the last layer of Gentleman-Sande NTT and the first layer of Cooley-Tukey INTT to fully utilize the dual-issue feature and reduce the overhead of load/store. In our design, the final scaling of INTT and post-processing are merged into the first layer of Cooley-Tukey INTT, and the point-wise or basecase multiplications are merged into the last layer of Gentleman-Sande NTT, which significantly reduces the load/store overhead.

Algorithm 3 Proposed Kyber polynomial multiplication

```
Require: $A(x), B(x) \in Z_{3329}[x]/(x^{256}+1)$
Require: Primitive 256-th root of unity $\zeta = 17$, $C = (\zeta^0, \zeta^1, ..., \zeta^{127})$
Require: $D = (\zeta^{2\cdot\mathbf{BR}(0)+1}, \zeta^{2\cdot\mathbf{BR}(1)+1} ..., \zeta^{2\cdot\mathbf{BR}(127)+1})$
Ensure: $H(x) = A(x)B(x), H(x) \in Z_{3329}[x]/(x^{256}+1)$
1: // pre-processing and NTT
2: $A'_e(x) \leftarrow \mathbf{NTT}(A_e(x) \odot C)$, $A'_o(x) \leftarrow \mathbf{NTT}(A_o(x) \odot C)$
3: $B'_e(x) \leftarrow \mathbf{NTT}(B_e(x) \odot C)$, $B'_o(x) \leftarrow \mathbf{NTT}(B_o(x) \odot C)$
4: // basecase multiplication
5: $H'_e(x) \leftarrow A'_e(x) \odot B'_e(x) + A'_o(x) \odot B'_o(x) \odot D$ // two modular reductions
6: $H'_o(x) \leftarrow A'_e(x) \odot B'_o(x) + A'_o(x) \odot B'_e(x)$ // one modular reduction
7: // INTT and post-processing
8: $H_e(x) \leftarrow \mathbf{INTT}(H'_e(x)) \odot (\zeta^0, \zeta^{-1}, ..., \zeta^{-127})$
9: $H_o(x) \leftarrow \mathbf{INTT}(H'_o(x)) \odot (\zeta^0, \zeta^{-1}, ..., \zeta^{-127})$
```

Dual-issue. The technique of utilizing dual-issue is similar to that of NTT/INTT because the point-wise and basecase multiplications operate on polynomials, which have fewer data dependencies. To take advantage of dual-issue, we divide the entire operation into multiple batches, each with some load/store instructions. Then, one operation is combined with the load/store of a preceding or succeeding operation, allowing overlap computation with memory access and reducing the overhead of load/store blockings.

The above optimizations enable our proposed design to achieve a remarkable 25.5× and 30.3× acceleration for Kyber 2×2 matrix multiplication and Dilithium 4×4 matrix multiplication (Table 5), respectively.

4.5 Polynomial Rejection Sampling

Rejection sampling is performed by checking whether a given $\lceil log(q) \rceil$-bit random number is in $[0, q-1]$. The rejection probability is $(1 - 3329/2^{12}) = 18.726\%$ for Kyber and $(1 - 8380417/2^{23}) = 0.098\%$ for Dilithium. However, when sampling 8 numbers in parallel, the rejection probability for Kyber is $(1-(3329/2^{12})^8) = 80.961\%$, and $(1-(8380417/2^{23})^8) = 0.779\%$ for Dilithium. In this case, parallel rejection sampling of Kyber would take a long time due to the high rejection probability.

To reduce the rejection probability of Kyber, we extend the range from $[0, q-1]$ to $[0, k \cdot q - 1]$ where k is an integer, and then perform modular reduction to reduce the range back to $[0, q-1]$ [BUC19]. By searching all possible k that satisfy $k \cdot q < 2^{31}$ and calculating the corresponding rejection probability, k with the minimum rejection probability can be found. For Kyber, the minimum rejection probability of 8 parallel samplings is 2×10^{-5}, where $k = 645082$. For Dilithium, the probability is 0.779%, where $k = 1$. Therefore, Kyber rejection sampling requires additional modular reduction.

In our design, the `bgeuv` instruction is used to check whether the given eight values are all in $[0, k \cdot q - 1]$. Instruction jump occurs when one of them is out of range. An example program for Kyber rejection sampling is shown in Listing 4.

Listing 4: Assembly code of Kyber rejection sampling.

```
addi %[cnt_data], %[addr_a], 0
addi %[cnt_buf], %[buf], 0
lv x0,  0(%[addr_b]) // load kq
lv x16, 0(%[addr_b])
lv x1,  0(%[mask]) // load mask=2^31-1
lv x17, 0(%[mask])
lv x15, 0(%[beta]) // load (beta mod q)
lv x31, 0(%[beta])
rej_loop:
  lv x2,   0(%[cnt_buf])
  lv x18, 16(%[cnt_buf])
  andv x3, x2, x1 // reduce to 31 bit
  addi %[cnt_buf], %[cnt_buf], 32
  bgeuv x3, x0, rej_loop
rej_modular: // modular multiplication
  mulv   x14,  x3, x15
  mulvh  x13,  x3, x15
  mulvm  x14, x14, x14
  mulvhf x14, x14, x14
  subv    x3, x13, x14
  sv x3,   0(%[cnt_data]) // store back
  sv x19, 16(%[cnt_data])
  // check if 256 numbers are sampled
  addi %[cnt_data], %[cnt_data], 32
  addi %[index], %[index], 1
  bltu %[index], %[data_256], rej_loop
```

4.6 Other Polynomial Operations

In addition to our proposed optimizations, other operations (polynomial compressions, packings, Dilithium Power2round, Decompose, etc.) are optimized with the proposed SIMD instructions whenever parallelism can be easily achieved. Our implementations are based on PQClean [KSSW22], where Power2round and Decompose are computed by additions, shiftings, and several logic operations. Operations that are difficult to parallelize are not optimized using the proposed SIMD instructions.

5 Experimental Results

The experimental results of our proposed processor for Kyber and Dilithium with different security levels are presented in this section. We compare the cycle counts of the accelerated bottleneck operations of our proposed processor with those of the baseline CV32E40P processor. In addition, we compare the cycle counts of Kyber and Dilithium with some state-of-the-art designs to demonstrate the advantages of our design. Furthermore, we present the implementation results of our proposed hardware design on FPGA and Silicon synthesis results with 28 *nm* process technology.

Table 4: Code sizes of Kyber KEM and Dilithium.

	Code size (kB)
Kyber KEM 512	24.4
Kyber KEM 768	26.1
Kyber KEM 1024	28.0
Dilithium 2	36.9
Dilithium 3	37.3
Dilithium 5	39.8

Our proposed processor is thoroughly tested and verified using the tools provided by CV32E40P. The supports for FPU and other customized instructions are turned off to ensure fair comparisons. To support assembly programming of our customized

SIMD instructions, we modified the RISC-V assembler by using riscv-opcodes tool[3]. The bottleneck operations and our optimizations are re-implemented by assembly code.

5.1 Code Sizes

We measured the code sizes by using `riscv32-unknown-elf-readelf` tools. As shown in Table 4, a 40 kB instruction memory is enough to support all the evaluation schemes. On the other hand, a 64 kB data memory is enough for running all these programs in our proposed processor. Therefore, our design utilizes a total of 104 kB SRAM, comprising a 40 kB instruction SRAM and a 64 kB data SRAM.

Table 5: Cycle count of bottleneck operations.

	CV32E40P (Baseline)	**Proposed Design**	**Acceleration Rate**
Keccak	12,924	404	32.0
256-pt NTT	45,270	1,750	25.9
256-pt INTT	49,738	1,925	25.8
Kyber-512 2×2 Matrix Multiplication	445,072	17,426	25.5
Dilithium-2 4×4 Matrix Multiplication	564,028	18,621	30.3
Kyber Rejection Sampling	60,303	4,657	12.9
Dilithium Rejection Sampling	92,717	5,500	16.9
CBD2 Sampling	24,305	1,951	12.5
CBD3 Sampling	40,791	2,508	16.3

5.2 Comparisons of Cycle Count

Table 6: Cycle count comparisons of Kyber KEM.

Work	**Security Level**	**Platform**	**Key Gen.**	**Encaps.**	**Decaps.**	**Total**
Kyber KEM Ref. [BDK+18]	1	x86-64	122,684	154,524	288,912	566,120
PQM4 [KRSS19]		Cortex-M4	514,291	652,769	621,245	1,788,305
VPQC [XHY+20]		Co-processor	18,556	45,886	79,989	144,431
[ZXXH22]		Co-processor	9,400	19,000	43,800	72,200
RISQ-V [FSS20]		RISC-V	150,106	193,076	204,843	548,025
Baseline		RISC-V	917,579	1,234,102	1,372,150	3,523,831
Proposed		RISC-V	**88,550**	**89,080**	**107,549**	**285,179**
Kyber KEM Ref. [BDK+18]	3	x86-64	199,408	235,260	425,492	860,160
PQM4 [KRSS19]		Cortex-M4	976,757	1,146,556	1,094,849	3,218,162
[ZXXH22]		Co-processor	14,200	26,200	59,100	99,500
RISQ-V [FSS20]		RISC-V	273,370	325,888	340,418	939,676
Baseline		RISC-V	1,908,045	2,297,127	2,420,253	6,625,425
Proposed		RISC-V	**164,053**	**166,322**	**196,794**	**527,169**
Kyber KEM Ref. [BDK+18]	5	x86-64	307,148	346,648	617,848	1,271,644
PQM4 [KRSS19]		Cortex-M4	1,575,052	1,779,848	1,709,348	5,064,248
VPQC [XHY+20]		Co-processor	39,689	81,569	136,475	257,733
[ZXXH22]		Co-processor	18,500	33,700	77,500	129,700
RISQ-V [FSS20]		RISC-V	349,673	405,477	424,682	1,179,832
Baseline		RISC-V	2,223,879	2,699,323	2,849,316	7,772,518
Proposed		RISC-V	**194,523**	**197,808**	**244,168**	**636,499**

The cycle counts in all our tests are measured by the `rdcycle` instruction and the cycle count of bottleneck operations in Kyber KEM and Dilithium are summarized in Table 5, which achieve more than 10× acceleration. The average results under 1000 tests of Kyber KEM and Dilithium with different security levels are presented in Table 6 and Table 7, respectively. The cycle counts are compared with state-of-the-art processor implementations including reference implementation (Intel x86-64), Cortex-M4, RISC-V co-processor and RISC-V. It should be noted that our decapsulation results include one encapsulation operation.

[3]https://github.com/riscv/riscv-opcodes

Table 7: Cycle count comparisons of Dilithium.

Work	Security Level	Platform	Key Gen.	Sign	Verify	Total
Dilithium Ref. [DLL+18]	1	x86-64	300,751	1,355,434	327,362	1,983,547
PQM4 [KRSS19]		Cortex-M4	1,400,412	6,157,001	1,461,284	9,018,697
[ZXXH22]		Co-processor	45,800	175,100	89,800	310,700
[KSFS23]		RISC-V	593,403	1,905,872	651,217	3,150,492
Baseline		RISC-V	2,558,152	4,239,156	2,889,910	9,687,218
Proposed		RISC-V	**177,148**	**594,477**	**207,970**	**979,595**
Dilithium Ref. [DLL+18]	3	x86-64	544,232	2,348,703	522,267	3,415,202
PQM4 [KRSS19]		Cortex-M4	2,282,485	9,289,499	2,228,898	13,800,882
[ZXXH22]		Co-processor	68,400	224,600	110,300	403,300
[KSFS23]		RISC-V	1,067,824	3,253,378	1,126,938	5,448,140
Baseline		RISC-V	4,446,507	6,355,525	4,648,733	15,450,765
Proposed		RISC-V	**283,866**	**745,810**	**343,555**	**1,373,231**
Dilithium Ref. [DLL+18]	5	x86-64	819,475	2,856,803	871,609	4,547,887
PQM4 [KRSS19]		Cortex-M4	3,097,421	8,468,805	3,173,500	14,739,726
[ZXXH22]		Co-processor	94,900	313,200	160,000	568,100
[KSFS23]		RISC-V	1,784,767	4,357,249	1,848,324	7,990,340
Baseline		RISC-V	7,444,308	10,114,960	7,771,766	25,331,034
Proposed		RISC-V	**467,453**	**1,172,521**	**536,543**	**2,176,517**

Our proposed designs exhibit approximately 10× faster than the baseline implementations and about 5× faster than the PQM4 implementations in cycle counts. Furthermore, our proposed processor achieves a remarkable 2× acceleration compared with the reference implementations on Intel x86-64 platform, despite its greater complexity with more pipeline stages and issue paths.

5.3 Hardware Implementation Results

Table 8: Resource utilization in FPGA.

	Platform	LUTs	Registers	BRAMs	DSPs	Freq.
Baseline	Zynq-7000	4,852	2,142	20	5	150
[FSS20]	Zynq-7000	24,306	10,837	32	18	-
Proposed	Zynq-7000	24,177	8,859	24	37	125

FPGA. We utilized Vivado 2017.4 with default settings to synthesize and implement our proposed processor and the CV32E40P processor. The results are presented in Table 8. Our proposed processor uses a total of 104 kB of memory and the baseline uses a total of 96 kB of memory (32 kB for instruction and 64 kB for data). Our resource usage is slightly lower compared to RISQ-V [FSS20], while our cycle counts are only half of those in RISQ-V.

Silicon. We synthesized our proposed processor using 28 nm technology. Both High Voltage Threshold (HVT) and Low Voltage Threshold (LVT) cells were used under 25 °C and 0.9 V. Our proposed processor operates at a frequency of 500 MHz with LVT cells and 200 MHz with HVT cells. Latch-based register files and clock-gating technique are used to optimize power consumption further. From our experiments, about 70% of registers are gated in our design.

The technology, frequency, logic gate count, SRAM requirements, performance, and power of different works are presented in Table 9. To make fair comparisons, the corresponding power consumption for Kyber KEM 512 and Dilithium 2 are normalized to the 28 nm process by the methods of [SB17]. As [FSS20] specifically focuses on very low-frequency targets, we also synthesized our design under 10 MHz for comparisons. To take into consideration all relevant factors, the Performance, Power, and Area Product (PPAP) for Kyber KEM 512 and Dilithium 2 are computed.

As can be seen, our proposed design achieves minimal PPAP, highlighting its significant advantages over other designs in terms of power, performance, and area. Furthermore,

Table 9: Silicon synthesis results.

				Logic		Kyber KEM 512				Dilithium 2			
	Tech. *(nm)*	**Volt.** *(V)*	**Freq.** *(MHz)*	**Gates** *(kGE)*	**SRAM** *(kB)*	**Perf.** *(ms)*	**Power** *(mW)*	**Norm. Power**	**PPAP**[*]	**Perf.** *(ms)*	**Power** *(mW)*	**Norm. Power**	**PPAP**
[FSS20]	65	1.2	10	170	146	54.80	2.57	1.49	**13,881**	-	-	-	-
This (HVT)	28	0.9	**10**	140.3	104	28.52	1.54	1.54	**6,548**	97.96	1.60	1.60	**23,369**
[KRSS19]	-	-	168	-	196	10.64	-	-	-	53.68	-	-	-
[BUC19]	40	1.1	72	106	40.25	4.84	5.49	2.69	**1,380**	11.52	7.57	3.70	**4,517**
[XHY$^+$20]	28	0.9	300	979	12	0.48	32.12	32.12	**15,139**	-	-	-	-
[ZZZ$^+$22]	28	0.9	500	1,900	484	0.02	163	163	**6,462**	0.09	237	237	**39,064**
[AMI$^+$22]	28	-	1,000	747	34.82	0.02	366.98	366.98	**4,155**	0.06	367.10	367.10	**16,913**
[ZXXH22]	28	-	540	581	24.75	0.13	83.23	83.23	**6,465**	0.58	85.29	85.29	**28,512**
[KSFS23]	22	0.8	800	244	>156	-	-	-	-	3.94	6.94	11.25	**10,810**
Baseline (HVT)	28	0.9	200	30.5	96	17.62	1.50	1.50	**806**	48.44	1.56	1.56	**2,304**
This (HVT)	28	0.9	**200**	149.1	104	1.43	2.01	2.01	**427**	4.90	2.13	2.13	**1,556**
This (LVT)	28	0.9	**500**	166.6	104	0.57	6.50	6.50	**618**	1.96	6.72	6.72	**2,193**

[*] PPAP = Perf. × Norm. Power × Logic Gate. $(ms \cdot W \cdot GE)$

our proposed processor provides support for all necessary operations of Kyber KEM and Dilithium presented in PQClean. As our design provides flexible fine-grained acceleration, various small polynomial operations including polynomial compress/decompress, reduction, etc are efficiently executed in our proposed processor. In contrast, [BUC19] can not support polynomial encode/decode, compress/decompress directly. On the other hand, [FSS20, XHY$^+$20, KSFS23] can only support one of the two LBC schemes. Through hardware/software co-design, our proposed processor can efficiently support all operations required by Kyber KEM and Dilithium.

6 Discussion and Future Work

In our designs, we have implemented all the schemes following PQClean, where many operations are implemented with constant time. However, a more detailed analysis of protection against possible attacks including side-channel, timing, and EM attacks is our future work. One significant advantage of our design is its ability to implement protection countermeasures in both software and hardware domains. We have the capability to incorporate various countermeasures, such as random masking, electromagnetic shielding, and noise addition, to enhance the security of our system. Moving forward, we are committed to further exploring these countermeasures and conducting in-depth research to identify additional techniques that can strengthen the security of our design even further. Moreover, to further bolster our design for code-based and hash-based schemes, we are also actively exploring additional techniques and conducting optimizations in both hardware and software.

7 Conclusion

In this paper, we have presented a highly efficient LBC processor that offers significant advantages over state-of-the-art designs. Instead of developing customized hardware accelerators for the critical operations presented in LBC algorithms, we have designed a customized SIMD RISC-V processor. This approach provides greater flexibility and potential applications for IoT devices. With our hardware and software co-design, our proposed design can implement all the operations presented in LBC algorithms, not just some critical operations. The hardware architecture is designed to efficiently reuse resources for various arithmetic and logic operations. To further enhance the efficiency of memory accesses, our proposed processor has been accommodated with a dual-issue path for parallel execution of load/store instructions. Our hardware and software co-design has allowed us to propose several algorithmic-level and architectural-level optimizations for polynomial multiplications. Additionally, we have achieved a significant performance improvement of Keccak with our proposed design due to the reduced overhead of memory accesses. We have performed FPGA and Silicon synthesis for our proposed design and compared it with the baseline RISC-V processor. Our design has achieved a significant performance improvement for the LBC algorithms. Moreover, our proposed processor outperforms state-of-the-art designs in terms of performance, power, and area, while offering greater flexibility. Our proposed design is particularly well-suited for IoT applications, where it can significantly reduce power and resource consumption while enhancing security against quantum computer attacks.

Acknowledgments

This research has been supported by the China National Key R&D Program (2022YFB4400704), the Basic Public Welfare Research Program of Zhejiang Province (LGG22F020017), and National Natural Science Foundation of China (62002023).

References

[AMI$^+$22] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. Kali: A crystal for post-quantum security using kyber and dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(2):747–758, 2022.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[BDK$^+$18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BDPVA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 313–314. Springer, 2013.

[BNAMK21] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. A monolithic hardware implementation of kyber: Comparing apples to apples in pqc candidates. In *Progress in Cryptology–LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America, Bogotá, Colombia, October 6–8, 2021, Proceedings 7*, pages 108–126. Springer, 2021.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures-how to sign with rsa and rabin. In *International conference on the theory and applications of cryptographic techniques*, pages 399–416. Springer, 1996.

[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.

[CMV$^+$14] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, R. C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(1):157–166, 2014.

[CPB$^+$12] Shu-jen Chang, Ray Perlner, William E Burr, Meltem Sönmez Turan, John M Kelsey, Souradyuti Paul, and Lawrence E Bassham. Third-round report of the sha-3 cryptographic hash algorithm competition. *NIST Interagency Report*, 7896:121, 2012.

[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.

[DLL$^+$18] Léo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals–dilithium: Digital signatures from module lattices. 2018.

[FHK$^+$18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5), 2018.

[FLX19] Xiang Feng, Shuguo Li, and Sufen Xu. RLWE-oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(3):556–559, 2019.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*, pages 537–554. Springer, 1999.

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

[HZZ$^+$22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022.

[KLC$^+$17] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High performance post-quantum key exchange on fpgas. *Cryptology ePrint Archive*, 2017.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. 2019.

[KSFS23] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on risc-v with hardware acceleration. *ACM Transactions on Embedded Computing Systems*, 2023.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&amp;P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[LTHW22] Minghao Li, Jing Tian, Xiao Hu, and Zhongfeng Wang. Reconfigurable and High-Efficiency Polynomial Multiplication Accelerator for CRYSTALS-Kyber. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

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

[RS16] Hemendra K Rawat and Patrick Schaumont. SIMD instruction set extensions for keccak with applications to SHA-3, keyak and ketje. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–8. 2016.

[RVM$^+$14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-LWE cryptoprocessor. In *International workshop on cryptographic hardware and embedded systems*, pages 371–391. Springer, 2014.

[SB17] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *Cryptology ePrint Archive*, 2018.

[Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[STCZ18] Shiming Song, Wei Tang, Thomas Chen, and Zhengya Zhang. LEIA: A 2.05mm ^{2} 140mW lattice encryption instruction accelerator in 40nm CMOS. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, San Diego, CA, 2018. IEEE.

[XHY$^+$20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture. *IEEE transactions on circuits and systems I: regular papers*, 67(8):2672–2684, 2020.

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

[YCH22] Zewen Ye, Ray CC Cheung, and Kejie Huang. PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(10):4068–4072, 2022.

[ZXXH22] Yifan Zhao, Ruiqi Xie, Guozhu Xin, and Jun Han. A high-performance domain-specific processor with matrix extension of risc-v for module-lwe applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2871–2884, 2022.

[ZZZ$^+$22] Yihong Zhu, Wenping Zhu, Min Zhu, Chongyang Li, Chenchen Deng, Chen Chen, Shuying Yin, Shouyi Yin, Shaojun Wei, and Leibo Liu. A 28nm 48kops 3.4 μj/op agile crypto-processor for post-quantum cryptography on multi-mathematical problems. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 514–516. IEEE, 2022.

A Proposed SIMD Instructions

Table 10: Proposed SIMD Instructions.

Instr.	**Type**	**rs1**	**rs2**	**rd**	**Description**
					SIMD Arithmetic Instructions
addv	R	8×32	8×32	8×32	8 Parallel additions
subv	R	8×32	8×32	8×32	8 Parallel subtractions
andv	R	8×32	8×32	8×32	8 Parallel logic AND
xorv	R	8×32	8×32	8×32	8 Parallel logic XOR
addvm	R	8×32	8×32	8×32	8 Parallel modular additions with FIX[0]
subvm	R	8×32	8×32	8×32	8 Parallel modular subtractions with FIX[0]
addvmt	R	8×32	8×32	8×32	The same as addvm but with output shuffling
subvmt	R	8×32	8×32	8×32	The same as subvm but with output shuffling
addvti	R	8×32	8×32	8×32	The same as addv but with input shuffling
subvti	R	8×32	8×32	8×32	The same as subv but with input shuffling
mulv	R	8×32	8×32	8×32	8 Parallel multiplications of lower 32 bits
mulvh	R	8×32	8×32	8×32	8 Parallel mul. of higher 32 bits
mulvm	R	8×32	-	8×32	8 Parallel mul. of `rs1` and FIX[1], yielding lower 32 bits
mulvhf	R	8×32	-	8×32	8 Parallel mul. of `rs1` and FIX[0], yielding higher 32 bits
cbd2	R	8×32	-	8×32	Centered binomial distribution $[-2, 2]$, sampling 8 numbers
cbd3	R	8×32	-	8×32	Centered binomial distribution $[-3, 3]$, sampling 8 numbers
sllvi	I	8×32	-	8×32	Logic left shift of each 32-bit number
sravi	I	8×32	-	8×32	Arithmetic right shift of each 32-bit number
bgeuv	B	8×32	8×32	-	Branch if one of `rs1` >`rs2`, for rejection sampling
					SIMD Keccak Instructions
xorv3	R	10×32	10×32	10×32	10 Parallel XOR with 3 operands (one is read from fixed address)
xorrv	R	10×32	-	10×32	Compute D of Algorithm 1
rxorv0	R	10×32	10×32	10×32	XOR and rotate specific amounts for computing B of Algorithm 1
rxorv1	R	10×32	10×32	10×32	The same as rxorv0 but different rotation amounts
rxorv2	R	10×32	10×32	10×32	The same as rxorv0 but different rotation amounts
rxorv3	R	10×32	10×32	10×32	The same as rxorv0 but different rotation amounts
rxorv4	R	10×32	10×32	10×32	The same as rxorv0 but different rotation amounts
xorv2	R	10×32	10×32	10×32	10 Parallel logic XOR
xorv2rc	R	10×32	10×32	10×32	2 Parallel logic XOR (compute lower 64-bit XOR)
xornavi	I	10×32	-	10×32	Compute E of Algorithm 1
shufflev	I	10×32	-	10×32	Data shuffling of five 64-bit numbers
					SIMD Load/Store Instructions
lv	I	32	-	4×32	Load 128-bit data to SIMD PR$i(1 \le i \le 4)$ `rd`
lwf	I	32	-	32	Load 32-bit data to FIX `rd`
lw64	I	32	-	2×32	Load 64-bit data to SIMD PR5 `rd`
sv	S	32	4×32	-	Store 128-bit data in SIMD PR$i(1 \le i \le 4)$ `rs2` back to memory
sw64	S	32	2×32	-	Store 64-bit data in SIMD PR5 `rs2` back to memory