Enhancing RISC-V Vector Extension for Efficient Application of Post-quantum Cryptography

Yifan Zhao*, Honglin Kuang*, Yi Sun*, Zhen Yang*
Chen Chen†, Jianyi Meng†, Jun Han*
*State Key Laboratory of Integrated Chips and Systems, Fudan University
†The T-Head Division, Alibaba Cloud
Shanghai, China

***Abstract*—We present a cryptography extension built on RISC-V Vector Extension for efficient application of lattice-based post-quantum cryptography, offering custom instructions that can perform vectorized operations on polynomials of variable length and data width. We use micro-operation architecture to simplify the execution of variable-latency vector instructions and propose fracturable modular arithmetic units to support operations on variable coefficient width. On this basis, a vector unit is designed, achieving significant speed-up compared to the state-of-the-art counterparts for number-theoretic-transform-based polynomial multiplication. This cryptography extension is further integrated into the gem5 simulator to evaluate CRYSTALS-Kyber and CRYSTALS-Dilithium; results outperform the state-of-the-art implementations with more than $2.3\times$ improvement in cycle count.**

***Index Terms*—Post-quantum cryptography, RISC-V, Vector Architecture, ISA Extension**

I. Introduction

Modern information society relies on Public-key Cryptography (PKC) as a reliable security infrastructure. However, the emergence of the quantum computer, combined with Shor's algorithm [1], enables attackers to quickly crack traditional PKC algorithms. Therefore, it is necessary to develop quantum-resistant PKC algorithms as the next generation of cryptosystems.

In this regard, the National Institute of Standards and Technology (NIST) launched the Post-Quantum Cryptography (PQC) Standardization process. In July 2022, NIST announced four candidate algorithms for standardization, with Kyber key encapsulation mechanism [2] and Dilithium digital signature scheme [3], two lattice-based PQC algorithms, selected as two primary algorithms to be implemented because of their strong security and excellent performance [4].

As the standard PQC algorithms have been selected, the cryptographic implementations on processors call for faster migration to the post-quantum cryptosystem. However, empowering general-purpose processors with quantum resistance is challenged by complex polynomial operations over finite fields with variable data width. Under such circumstances, the instruction set extension (ISE) strategy can play an instrumental role in deploying PQC on processors due to its fine balance between achieving performance advantages and enabling software flexibility. Moreover, this strategy is further facilitated by open-source instruction set architectures (ISA) like RISC-V [5].

Corresponding author: Jun Han, State Key Laboratory of Integrated Chips and Systems, Fudan University, junhan@fudan.edu.cn

There have been plenty of hardware implementations based on RISC-V ISE for accelerating PQC. The designs in [6]–[8] focus on lightweight scalar extensions. Although these works are suitable for small IoT devices, the scalar nature hinders the potential to exploit data-level parallelism (DLP). Nevertheless, massive DLP is inherent in polynomial operations of PQC algorithms, such as number-theoretic transform (NTT) and sampling. This suggests that these operations can be vectorized through single-instruction-multiple-data (SIMD) architecture to pursue higher performance. Xin *et al.* [9] propose a vector coprocessor targeting these kernel operations. However, the performance gain of this work is limited by Amdahl's law due to the lack of a general-purpose SIMD ISA to accelerate the remaining non-kernel operations. Besides, this work cannot well adapt to the variable data width across different PQC algorithms. In light of this, the design of custom vector extensions are preferably based on a general-purpose SIMD ISA to achieve better performance improvement and good adaptability.

The RISC-V vector extension (RVV), currently in stable version 1.0 [10], can serve as an optimal platform for further vector extensions targeting PQC algorithms. Moreover, with the renewed interest in vector processing, RISC-V vector processors implementing RVV 1.0 are emerging in both academia and industry, such as FPGA-based Vicuna [11] designed by TU Wien and ASIC-based Xuantie C908 [12] presented by Alibaba Cloud. Built on RVV 1.0, the RISC-V vector cryptography extension [13] is currently in draft stage. However, this ISE is designed to support traditional symmetric cryptography algorithms and does not involve modern public-key cryptography including PQC algorithms. Concerning RVV-based extensions for PQC algorithms, the implementation in [14] can be seen as the first practice. However, this extension is incomplete as it relies on specialized register files to perform computation rather than reusing the vector register file (VRF) of RVV, which introduces unnecessary communication overhead and resource consumption. Moreover, this work is limited to optimizing the polynomial operations in Kyber without the generality to support multiple parameter sets in different PQC algorithms. Consequently, the full potential of RVV-based ISE

DOI 10.1109/ASAP57973.2023.00014

for PQC algorithms remains to be explored.

Building on this momentum, this paper extends RVV with a cryptography ISE to efficiently compute and program lattice-based PQC applications. The proposed RVV-based ISE can act as a reference for future additions to the developing RISC-V vector cryptography extension. The main contributions of this paper are highlighted as follows:

- We investigate the vectorization of kernel polynomial operations in PQC algorithms and propose a custom cryptography ISE built on RVV. Utilizing the rich SIMD infrastructure of RVV, such as dynamic register typing and variable vector length, the proposed custom instructions can perform vectorized operations on polynomials of diverse length and coefficient data width and closely cooperate with RVV to enable efficient programming.
- We utilize micro-operation architecture to support the pipelined execution of variable-latency vector instructions. Fracturable modular arithmetic units are proposed to well handle operations on variable coefficient widths. On this basis, we design a custom vector unit that outperforms the state-of-the-art works for polynomial multiplication.
- We integrate the proposed cryptography vector extension into an RVV-enabled gem5 architectural simulator to evaluate the performance of Kyber and Dilithium through the enhanced RVV, and achieve more than 2.3× speedup in cycle count compared to the state-of-the-art implementations.

II. Preliminaries

A. Symbol Definition

We define $\mathcal{R}_q$ to represent polynomial rings $\mathbb{Z}_q/\phi(x)$. In both Kyber and Dilithium, the cyclotomic ring $\mathcal{R}_q$ is used with $\phi(x) = X^n + 1$, modulo q a prime and degree $n = 256$. The polynomial elements over the ring are denoted as lowercase symbols like $a \in \mathcal{R}_q$. In contrast, we use $\hat{a}$ to represent the polynomial in the NTT domain. We use $\odot$ to denote pointwise multiplication (PWM) $\hat{c} = \hat{a} \odot \hat{b}$ with $\hat{c}_i = \hat{a}_i \hat{b}_i$. Besides, we use $\circ$ to denote basecase multiplication (BM) $\hat{c} = \hat{a} \circ \hat{b}$ over degree two polynomial ring $\mathbb{Z}_q[x]/\left(X^2 - \omega^i\right)$ with $\hat{c}_{2i} + \hat{c}_{2i+1}X = (\hat{a}_{2i} + \hat{a}_{2i+1}X)(\hat{b}_{2i} + \hat{b}_{2i+1}X) \bmod (X^2 - \omega_n^i)$.

B. Vectorized Lattice-based PQC

In PQC algorithms, a polynomial over ring $\mathcal{R}_q$ can be represented as a vector of coefficients, $a = [a_0, a_1, \ldots, a_{n-1}]$, with $a = \sum_{i=0}^{n-1} a_i X^i$. Any polynomial operation essentially performs on vectors of coefficients. Thus, most polynomial operations can be vectorized to improve throughput.

The polynomial multiplication (PMUL) is the main performance bottleneck in PQC algorithms, which makes it the focus of vectorization. Typically, the polynomial multiplication $c = a \cdot b$ is performed in a NTT-based method $c = NTT^{-1}(NTT(a) \odot NTT(b))$ to reduce complexity from $O(n^2)$ to $O(nlog(n))$. Furthermore, NTT can be optimized through negative wrapped convolution (NWC) [15] to avoid zero padding in classic NTT. The NWC version of NTT

Algorithm 1 Vectorized n-point complete NTT

Input: $a \in \mathcal{R}_q$, precomputed ψ_{2n}^k for $k \in [0, n-1]$ stored in bit-reversed order of k in array **zetas**, vector lane number=$vlan$
Output: $\hat{a} = NTT(a)$
Require: The NTT point n is a power of 2

```
r ← 1
for stage = 1 to log2 n do
    m ← n / 2^(stage−1)
    if stage ≤ log2 n − log2(vlan) then
        for i = 0 until 2^(stage−1) do
            ω ← zetas[r]
            for j = 0 until m / (2vlan) do
                for k = 0 until vlan do
                    idx ← i·m + j·vlan + k
                    u ← a_idx
                    v ← a_{idx+m/2}
                    a_idx ← u + v·ω mod q
                    a_{idx+m/2} ← u − v·ω mod q
                end for
            end for
            r ← r + 1
        end for
    else
        for i = 0 until n / (2vlan) do
            for j = 0 until 2vlan / m do
                ω ← zetas[r]
                for k = 0 to m/2 do
                    idx ← i·2vlan + j·m + k
                    u ← a_idx
                    v ← a_{idx+m/2}
                    a_idx ← u + v·ω mod q
                    a_{idx+m/2} ← u − v·ω mod q
                end for
                r ← r + 1
            end for
        end for
    end if
end for
return a
```

$\hat{a} = NTT(a)$ and inverse NTT (INTT) $a = INTT(\hat{a})$, used in both Kyber and Dilithium, are formulated as $\hat{a}_i = \sum_{j=0}^{n-1} \psi_{2n}^j \cdot a_j \cdot \omega_n^{ij}$ and $a_i = n^{-1} \cdot \psi_{2n}^{-i} \sum_{j=0}^{n-1} \hat{a}_j \cdot \omega_n^{-ij}$, with ω_n and $\psi_{2n} = \sqrt{\omega_n} \bmod q$ the n-th and $2n$-th root of unity.

In [9], the authors propose a vectorized algorithm for NTT computation. However, it requires some redundant operations, including bit reversing, pre-processing of multiplying by ψ_{2n}^j and post-processing of multiplying by $n^{-1}\psi_{2n}^{-i}$. Instead, we propose a vectorized version with these redundant operations omitted, as shown in Algorithm 1. To avoid bit reversing, we follow the approach proposed in [6] and [16] that firstly the Cooley-Tukey (CT) butterfly is used to compute NTT, then the Gentleman-Sande (GS) butterfly is used to perform

Algorithm 2 Vectorized n-point polynomial basecase multiplication

Input: $\hat{a} = NTT(a), \hat{b} = NTT(b)$, precomputed ψ_{2n}^{k} for $k \in [0, n-1]$ stored in bit-reversed order of k in array **zetas**, vector lane number = $vlan$
Output: $\hat{c} = \hat{a} \circ \hat{b}$

function $bmul((u_0, u_1, v_0, v_1, \omega))$
  $s_0 \leftarrow u_0 + u_1,\ s_1 \leftarrow v_0 + v_1$ ▷ Phase One
  $m_0 \leftarrow u_0 \cdot v_0,\ m_1 \leftarrow u_1 \cdot v_1$ ▷ Phase One
  $z_0 \leftarrow m_0 + m1 \cdot \omega$ ▷ Phase Two
  $z_1 \leftarrow s_0 \cdot s_1 - (m_0 + m_1)$ ▷ Phase Two
  return (z_0, z_1)
end function
for $i = 0$ to $\frac{n}{2l}$ **do** ▷ Use l to represent $vlan$ for simplicity
  for $j = 0$ to $\frac{l}{2}$ **do**
    $\omega \leftarrow \mathbf{zetas}[\frac{n}{2} + i \cdot \frac{l}{2} + j]$
    $u_0, u_1 \leftarrow \hat{a}_{2il+4j}, \hat{a}_{2il+4j+1}$
    $v_0, v_1 \leftarrow \hat{b}_{2il+4j}, \hat{b}_{2il+4j+1}$
    $(\hat{c}_{2il+4j}, \hat{c}_{2il+4j+1}) = bmul(u_0, u_1, v_0, v_1, \omega)$
    $u_2, u_3 \leftarrow \hat{a}_{2il+4j+2}, \hat{a}_{2il+4j+3}$
    $v_2, v_3 \leftarrow \hat{b}_{2il+4j+2}, \hat{b}_{2il+4j+3}$
    $(\hat{c}_{2il+4j+2}, \hat{c}_{2il+4j+3}) = bmul(u_2, u_3, v_2, v_3, -\omega)$
  end for
end for
return $\hat{c}$

INTT. Additionally, we integrate pre- and post-processing into the butterfly operation by merging the scaling factors ψ_{2n}^{j} and $n^{-1}\psi_{2n}^{-i}$ with the twiddle factors ω_n^{ij} and ω_n^{-ij}. Vector lane number $vlan$ defines the number of butterfly operations executed in parallel. After $\log_2 n$ stages of butterfly operations, complete NTT is accomplished, then PWM operation is required to multiply two polynomials in the NTT domain. The PWM operation can be easily vectorized by multiplying multiple polynomial coefficients in parallel. Nevertheless, instead of PWM, the complex BM operation is used for multiplying polynomial elements in Kyber because the NTT in Kyber is incomplete. We split the BM operation into two phases to reduce its complexity and propose the parallel version in Algorithm 2.

Apart from polynomial arithmetic operations, another part of kernel operations in PQC algorithms is polynomial generation, which consists of the pseudo-random number generation (PRNG) and sampling process. PRNG is accomplished through hash functions specified in the SHA3 standard [17], which provide a sequence of random bits for sampling. The vectorized sampling processes are proposed in [10], which take chunks of random bits and use parallel samplers to generate multiple polynomial coefficients concurrently according to a specific distribution.

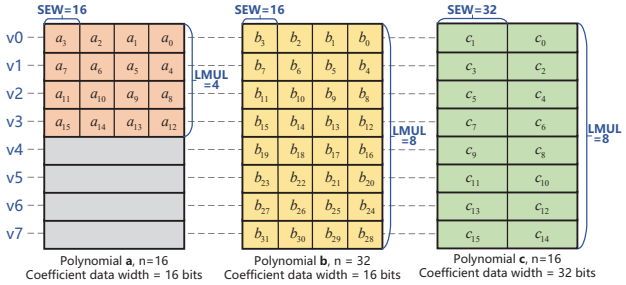

Fig. 1. Simplified logical view of an 8-entry VRF with *VLEN*=64.

III. Design Principle of RVV-based Custom Instructions

RVV provides a comprehensive SIMD infrastructure, from dynamic register typing to Intrinsic API, to support efficient computation and concise vectorized programming. Thus, the design of RVV-based custom instructions shall fully consider and utilize these advantages to integrate into the RVV environment with high efficiency and programmability for PQC applications.

A. Utilize Dynamic Register Typing of RVV

The vector register type in RVV can be dynamically configured in two dimensions. One is the selected element width (*SEW*), which determines the number of elements *VLEN*/*SEW* in a *VLEN* bit-long vector register. The other is the vector length multiplier (*LMUL*), which represents the number of vector registers combined to form a vector register group. Given an *SEW* and *LMUL* setting, a single vector instruction can operate on a register group with vector length *vl* up to *VLMAX*=*LMUL*×*VLEN*/*SEW*.

This mechanism can be utilized to accommodate a whole polynomial in one vector register group through proper register type configuration. With *vl* configured to the polynomial length, and *SEW* configured to the coefficient data width, as depicted in Fig. 1, such a VRF setting allows a single custom instruction to process a whole polynomial, which leads to high DLP exploitation and programming simplicity.

B. Utilize RVV Intrinsic to Closely Cooperate

RVV intrinsic[1] is the interface to low level RVV assembly in high level programming language C/C++, which contains a rich type system and intrinsic APIs. For example, the data type $vint16m4_t\ vx$ represents a register group vx with *SEW*=16 and *LMUL*=4. These parameters are configured through the following configuration-setting intrinsic $vsetvl_e16m4$, which also returns the active vector length *vl* according to the given application vector length (*AVL*). Then, the register group type vx and vector length *vl* can be used in intrinsic APIs to perform vector operations.

RVV intrinsic should be extended to custom vector instructions to facilitate the programming of PQC applications. To

[1]https://github.com/riscv-non-isa/rvv-intrinsic-doc

achieve this goal, the binutil of the RVV GCC compiler[2] can be modified to add opcodes of custom vector instructions. On this basis, we can conveniently use RVV intrinsic to perform vectorized polynomial operations via close cooperation of standard and custom RVV instructions. The following codes demonstrate the processing of polynomial a with the VRF setting in Fig. 1.

```
/*enable RVV intrinsic*/
#include <riscv_vector.h>
// define two vector register group vx and vy
vint16m4_t vx, vy;
// set LMUL, SEW and get VL
size_t vl = vsetvl_e16m4(n);
// load polynomial a to vx
vx = vle16_v_i16m4((int16_t*)a, vl);
// perform polynomial operation
vy = v_poly_operation(vx, vl);
// store results back to polynomial a
vse16_v_i16m4((int16_t*)a, vy, vl);
```

Listing 1. Utilize RVV Intrinsic to program PQC applications. The Intrinsics in blue and teal respectively correspond to standard and custom RVV instructions.

C. Utilize micro-ops architecture to simplify execution

Custom vector instructions perform complex operations on a variable number of vector registers, leading to a variable latency, which makes it complicated to execute down the pipeline. In this case, the micro-operation architecture can be utilized to treat vector instructions as macro-operations (macro-ops), which would be further decomposed into multiple simple micro-operations (micro-ops) to execute, as shown in Fig. 3. Micro-ops perform a simpler function but contain more control signals and thus are much easier to implement in the pipeline than macro-ops. Moreover, through adjusting the number of decoded micro-ops according to the current VRF setting, the custom instructions can smoothly operate on a variable vector length. In this way, we can design complex vector instructions while maintaining a compact and efficient execution pipeline.

IV. Custom Instructions And Hardware Primitives Design

A. Custom Instruction Design

Following the principles described in Section III, we extend RVV with a set of custom vector instructions, as shown in TABLE I, to efficiently compute and program PQC applications. Besides, we extend the standard VRF $v[\,]$ with an additional VRF $av[\,]$ containing nine vector registers. $av[0]-av[7]$ are used to store the twiddle factors ψ_{2n}^{k} in Algorithm 1 and 2, while $av[8]$ is used to store the Keccak state used in PRNG.

1) Macro-ops to micro-ops: In TABLE I, the instructions in bold are defined as macro-ops that require further decoding into multiple micro-ops to execute. Each micro-op processes an element group of a vector register in parallel, and the processing parallelism depends on the vector lane number $vlan$.

As a demonstration, Fig. 2 shows the execution of micro-ops decoded from macro-op **vbfct.vv**. This macro-op performs the

[2]https://github.com/riscv-collab/riscv-gnu-toolchain/tree/rvv-next

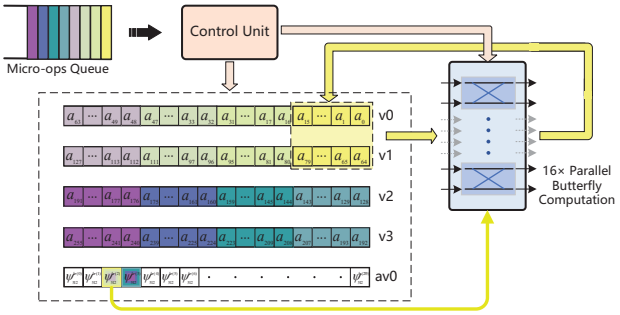

Fig. 2. Execution Detail of micro-ops decoded from macro-op **vbfct.vv**.

first stage of butterfly operation within an NTT computation of a *vl*=256 polynomial a in vector register group v0 with *LMUL*=4, *SEW*=32 and *VLEN*=2048. Each micro-op reads 2×16 coefficients from vector registers and the matching twiddle factors from additional vector registers, performing $vlan = 16$ sets of butterfly operations with $16\times$ parallel Butterfly Units (BFU). Thus total $256/(2\times 16) = 8$ micro-ops would be decoded from one macro-op **vbfct.vv** in such a condition. The macro-to-micro decoding would attach additional control fields to the micro-ops, such as the starting index of elements to be accessed in the standard and additional vector registers and the corresponding access length.

2) Support for variable vector register typing: Custom vector instructions support variable vector register typing. The settings of vector registers such as *SEW*, *LMUL*, and *vl* should be configured in advance through the RVV configuration-setting instructions, as these settings regulate the generation and execution of micro-ops.

For polynomial arithmetic instructions like **vbfct.vv** and **vbfgs.vv**, the number of micro-ops decoded from macro-ops is calculated as $\frac{(VLEN/SEW)\times LMUL}{2vlan}$. The executing parallelism $vlan_{SEW=16}$ is twice as large as $vlan_{SEW=32}$, as we propose a BFU architecture composed of fracturable modular arithmetic units that can double the throughput for lower-width coefficients. As RVV specifies the upper limit of $LMUL_{max} = 8$, the proposed instructions can operate on polynomials with up to $n = \frac{VLEN\times 8}{SEW}$ coefficients.

B. Scalable and Efficient Custom Instruction Programming

Benefiting from the micro-ops architecture and variable vector register typing, the proposed custom instructions demonstrate good support for polynomial operations of different parameter settings. We further modify the RVV GCC compiler to enable programming with RVV Intrinsic API, which facilitates the cooperation of RVV and custom instructions. Using the proposed cryptography extension, we can easily develop PQC programs with high scalability and efficiency.

1) Polynomial Multiplication: We present the RVV Intrinsic programming of the polynomial multiplication in Kyber and Dilithium in List 2 and 3 respectively, with the same *VLEN* as the VRF in Fig. 2. The difference of modulus q leads to different vector group settings. Besides, the NTT/INTT in

TABLE I
BRIEF INTRODUCTION OF THE CUSTOM INSTRUCTIONS

Instructions	Function[a]	Description
vsetmodulusq	csrModulusq$\leftarrow r[rs1]$	Write the modulus q to a custom additional control and status register (CSR), csrModulusq
vbfct.vv	$v[vd] \leftarrow$ CT butterfly$(vs[vs1], imm)$	Perform the imm-th stage of CT butterfly operation to the polynomial in vector register group $vs1$
vbfgs.vv	$v[vd] \leftarrow$ GS butterfly$(vs[vs1], imm)$	Perform the imm-th stage of GS butterfly operation to the polynomial in vector register group $vs1$
vbasemul.vv	$v[vd] \leftarrow$ Basecase multiplication$(v[vs1], v[vs2])$	Perform basecase multiplication to the polynomials in vector register group $vs1$ and $vs2$
vmodadd.vv	$v[vd] \leftarrow (v[vs1] + v[vs2])\%\ q$	Perform modular addition to the polynomial coefficients in vector register group $vs1$ and $vs2$
vmodsub.vv	$v[vd] \leftarrow (v[vs1] - v[vs2])\%\ q$	Perform modular subtraction to the polynomial coefficients in vector register group $vs1$ and $vs2$
vmodmul.vv	$v[vd] \leftarrow (v[vs1] \times v[vs2])\%\ q$	Perform modular multiplication to the polynomial coefficients in vector register group $vs1$ and $vs2$
vmod.v	$v[vd] \leftarrow (v[vs1]\%\ q)$	Perform modular reduction to the polynomial coefficients in vector register group $vs1$
vkeccakinit.v	$av[8] \leftarrow 0$	Initializes the Keccak state in the additional vector register $av[8]$.
vkeccakabsorb.v	$v[vd] \leftarrow$ Keccak absorb$(v[vs1], mode)$	Absorb the input bytes in $v[vs2]$ into the Keccak state according to SHA3 function $mode$.
vkeccaksqueeze.v	$v[vd] \leftarrow$ Keccak squeeze$(mode)$	Squeeze one block of rate bytes according to SHA3 function $mode$ from Keccak state to $v[vd]$.
vhammweight.v	$v[vd] \leftarrow$ Hamming weight$(v[vs1], imm)$	Compute the hamming weight of the first imm bits of the register elements of vector register group $v[vs1]$
vale.v	$av[avd] \leftarrow$ memory$(r[rs1])$	Perform vector unit-stride load to additional vector register avd with base memory address $r[rs1]$
vase.v	memory$(r[rs1]) \leftarrow av[avs2]$	Perform vector unit-stride store to additional vector register $avs2$ with base memory address $r[rs1]$

[a] $r[\ ]$ is scalar register file, while $v[\ ]$ and $av[\ ]$ are VRF and additional VRF, respectively. vd is the index of the destination vector register group, and $(vs1, vs2)$ are indexes of the two source vector register groups; the same goes for avd, $avs1$, $avs2$, and $rs1$.

Kyber is one butterfly stage less than that in Dilithium, as incomplete NTT/INTT is used in Kyber.

```
void poly_mul_Kyber(const int16_t a[256], const int16_t b
     [256], int16_t c[256]) {
  size_t i, vl;
  vint16m2_t vx, vy, vz;
  int16_t modulusq = 3329;
  vsetmodulusq(modulusq);
  vl = vsetvl_e16m2(256);
  vx = vle16_v_i16m2(a, vl);
  vy = vle16_v_i16m2(b, vl);
  for(i = 1; i <= 7; i++) {     // perform incomplete NTT
    vx = vbfct_vv(vx, i, vl);
    vy = vbfct_vv(vy, i, vl);
  }
  vz = vbasemul_vv(vx, vy, vl);
  for(i = 2; i <= 8; i++) {     // perform incomplete INTT
    vz = vbfgs_vv(vz, i, vl);
  }
  vse16_v_i16m2(c, vz, vl);
}
```

Listing 2. RVV Intrinsic Programming of Polynomial Multiplication in Kyber.

```
void poly_mul_Dilithium(const int16_t a[256], const int16_t
     b[256], int16_t c[256]) {
  size_t i, vl;
  vint32m4_t vx, vy, vz;
  int32_t modulusq = 8380417;
  vsetmodulusq(modulusq);
  vl = vsetvl_e32m4(256);
  vx = vle32_v_i32m4(a, vl);
  vy = vle32_v_i32m4(b, vl);
  for(i = 1; i <= 8; i++) {     // perform complete NTT
    vx = vbfct_vv(vx, i, vl);
    vy = vbfct_vv(vy, i, vl);
  }
  vz = vmodmul_vv(vx, vy, vl);
  for(i = 1; i <= 8; i++) {     // perform complete NTT
    vz = vbfgs_vv(vz, i, vl);
  }
  vse32_v_i32m4(c, vz, vl);
}
```

Listing 3. RVV Intrinsic Programming of Polynomial Multiplication in Dilithium.

2) Polynomial Generation: Instead of implementing the entire sampling process directly, we use RVV arithmetic instructions cooperated with custom instructions for sampling to maximize the utilization of existing RVV specifications.

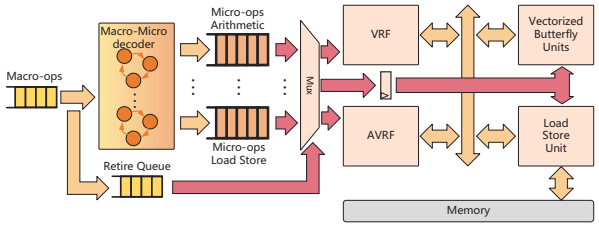

Fig. 3. The architecture of the proposed custom vector unit using micro-ops architecture.

For example, Listing 4 demonstrates the rejection sampling in Kyber. We first squeeze the random bits from the keccak state to the vector register vx. Then, we use the vector compare instruction **vmsltu.vx** to accept numbers less than $q = 3329$ Finally, the number of valid samples is counted through the vector count population instruction **vcpop.m**.

For binomial sampling, we use the **vhammweight.v** instruction to compute the Hamming weight of each register element from two vector register groups; then the sampling results are obtained by the **vsub.vv** instruction performing vector subtract.

```
size_t vl, valid_num=0;
vuint16m1_t vx;
vbool_16 vm;
vl = vsetvl_e16m1(SHAKE128_RATE/2);
vx = vkeccaksqueeze_v(vx, SHAKE128_MODE);
vm = vmsltu_vx_u16m1_b16(vx, KYBER_Q, vl);
valid_num = vcpop_m_b16(vm, vl);
```

Listing 4. RVV Intrinsic Programming of Rejection Sampling in Kyber.

C. Custom Vector Unit Design

We further implement a custom vector unit using micro-ops architecture, as shown in Fig. 3, which supports the proposed polynomial arithmetic instructions and basic RVV settings.

In order to well support operations on variable coefficient data width, we design a set of fracturable modular arithmetic

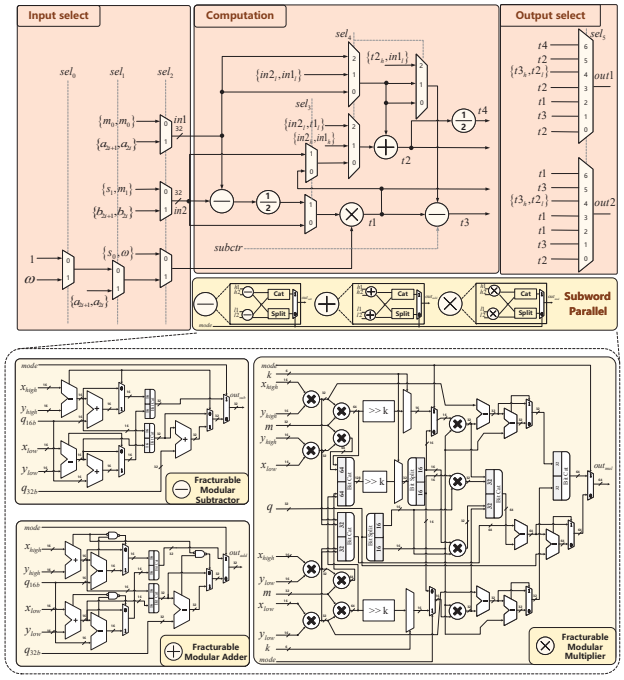

Fig. 4. BFU architecture featuring fracturable modular arithmetic units and configurable functionality.

units, as the bottom half of Fig. 4 shows, that can exploit the subword parallelism across different PQC algorithms. For example, 32-bit data width is used in Dilithium to store a polynomial coefficient, while the required data width is halved in Kyber. Thus, two polynomial coefficients in Kyber can be packed into a single 32-bit word through register type configuration and processed in parallel. According to the coefficient data width, the fracturable modular arithmetic units can be configured in Comp or Split mode. In Comp mode, the carry chain manipulation logic is activated, and 32-bit modular operations are performed by combining 16-bit partial results into 32-bit outputs. In Split mode, the 32-bit data paths are split into two separate 16-bit width ones to double computation throughput for low-width coefficients.

By assembling the fracturable modular arithmetic units, we propose a novel BFU architecture as the computing workhorse, shown in the top half of Fig. 4. The $\{,\}$ symbol represents concatenating two 16-bit numbers to a 32-bit number. Each BFU can perform the CT/GS butterfly operation on one pair of 32-bit coefficients or two pairs of 16-bit ones. Besides, the BFU is designed to be highly functionally configurable via sophisticated data path organization. In addition to simple modular operations, the complex BM operation is also supported by cascading two BFUs together. The first BFU performs the phase 1 BM operation that $out1 = \{s_1, s_0\}$ and $out2 = \{m_1, m_0\}$, while the second BFU accomplishes the phase 2 BM operation that $out1 = \{c_{2i+1}, c_{2i}\}$.

V. Experimental Results

To obtain the precise operation latency and resource overhead, we implement the proposed custom vector unit using Chisel and evaluate the proposed design on the Xilinx UltraScale+ ZCU111 platform with Xilinx Vivado 2019.2.

Table II reports the evaluated performance results. With a *VLEN*=2048 VRF, this implementation can support NTT and PMUL up to $n = 1024$. Compared to the scalar-based implementations [7] [8], our work achieves more than two orders of magnitude improvement in cycle count of NTT. Compared to the RVV-based implementation [14], the speedup of 12.5 is attained when computing the PMUL in Kyber, with vector lane number both set to 32. Moreover, our work computes the NTT in Kyber $1.32\times$ faster than the state-of-the-art vector-based implementation [9] under the same lane number; this performance gain mainly benefits from the fracturable butterfly unit design, which achieves 39% reduction in cycle count computing the NTT in Kyber with a lower coefficient width, compared to the NTT in Dilithium. Table IV shows the evaluated area utilization. To make a fair comparison with the FPGA-based related works, we sum the area usage of our hardware implementation and the FPGA-based RISC-V vector processor Vicuna to calculate the Area $\times$ Cycle Ratio of the complete processor system. With lane number set to 32, our work achieves better Area $\times$ Cycle metrics in terms of both LUT and DSP consumption.

To evaluate the performance gain of the overall PQC algorithm, we integrate the proposed RVV-based cryptography extension into the RVV-enabled gem5 micro-architecture simulator[3]. We use micro-ops architecture to define the instruction functionality in gem5's ISA description files. The Minor CPU model is selected to provide an in-order single-issue core for workload scheduling and non-vectorizable computing tasks. Based on the official C-code of PQC algorithms Kyber [2] and Dilithium [3], we accelerate the kernel operations through the cooperation of RVV and the proposed custom vector instructions. In addition, we also optimize the non-kernel operations, such as polynomial compression and decompression, using the SIMD instructions provided by RVV. To improve simulation accuracy, we model the latency of RVV instructions and custom vector instructions, respectively, based on the real latency of the vector engine in the commercial processor Xuantie C908 and the 32-lane custom vector unit. With this simulation model, we evaluate the performance of PQC algorithms Kyber and Dilithium. The evaluation results are presented in TABLE III and compared with the state-of-the-art counterpart implementations. Besides, we also list the CoreMark benchmark [19] scores of each CPU core to provide a reference for their performance, because the core performance has a great impact on the overall cycle count of the algorithm, even though this factor is basically irrelevant to the optimization schemes proposed in each work.

As can be seen from TABLE III, on the basis of similar CoreMark/MHz scores of the used CPU models, our work

[3]https://github.com/plctlab/plct-gem5

TABLE II
PERFORMANCE COMPARISON OF NTT AND PMUL WITH THE COUNTERPART IMPLEMENTATIONS

	RANTT [7]	RISQ-V [8]	VPQC [9]	DAC'22 [14]	**This work**
Supported Lattice Parameter	Modulus $q < 2^{32}$ Dimension $n \leq 2048$	Modulus $q < 2^{16}$ Dimension $n \leq 1024$	Modulus $q < 2^{16}$ Dimension $n \leq 2048$	Modulus $q = 3329$ Dimension $n = 256$	Modulus $q < 2^{32}$ Dimension $n \leq 1024$
Lane Number	1	2	32	32	32
NTT and PMUL[a]Performance in Kyber ($q = 3329$, $log_2 q = 12$, $n = 256$)					
Cycles(NTT/INTT/PMUL)	–	1935/1930/-	41/-/-	383/453/1334	31/31/107
NTT and PMUL[b]Performance in Dilithium ($q = 8380417$, $log_2 q = 23$, $n = 256$)					
Cycles(NTT/INTT/PMUL)	43756/-/-	–	–	–	51/51/167

[a] In Kyber, PMUL(a,b)=INTT(NTT(a) $\circ$ NTT(b))).
[b] In Dilithium, PMUL(a,b)=INTT(NTT(a) $\odot$ NTT(b))).

TABLE III
MESURED PERFORMANCE OF PQC ALGORITHMS, KYBER AND DILITHIUM

Design	CPU Model	CoreMark/MHz[a]	Instruction Set Architecture	Cycle Count (kCCs)							
				CRYSTALS-Kyber-512-CCA-KEM				CRYSTALS-Dilithium-II			
				KeyGen	Encaps	Decaps	Total Cycles[b]Ratio	KeyGen	Sign	Verify	Total Cycles[b]Ratio
This Work	Minor CPU	2.34	RISC-V+Enhanced RVV	22.8	23.8	31.1	1	82.7	159.2	84.9	1
RISQ-V [8]	CV32E40P	2.74	RISC-V + Scalar ISE	150.1	193.1	204.9	7.2	–	–	–	–
Sapphire [6]	Unknown	2.33[c]	RISC-V + Scalar ISE	74.5	131.7	142.3	5.0	167.4	634.8	229.5	3.5
VPQC [9]	SCR1	2.24	RISC-V + Vector ISE	18.6	45.9	80	2.3	–	–	–	–
TCHES'22 [18]	Cortex-A72	5.4	Armv8 + Neon	62.5	80.7	76.4	2.9	269.7	649.2	272.8	3.8

[a] Under GCC Compiler Optimization Option -O2.
[b] Total Cycles refer to the cycles required for an encapsulation and decapsulation, or a signature signing and verification, as the key generation can be done offline.
[c] Sapphire [6] does not present the CPU model, but mentions that the CPU performance is similar to Arm Cortex M0; thus the CoreMark score of Arm Cortex M0 is used.

TABLE IV
FPGA-BASED AREA CONSUMPTION AND COMPARISON RESULTS

Work	Lane number	Resource Usage				Area × Cycle[a]Ratio	
		LUT	FF	DSP	BRAM	LUT	DSP
RANTT [7]	1	1.5k	0.5k	4	–	7.9	4.9
RISQ-V [8]	2	24.3k	10.9k	18	32	9.4	1.6
DAC'22 [14]	32	318.2k	46.0k	138	16	24.3	2.4
This Work+Vicuna	32	**79.5k**+82.4k	**6.9k**+37.7k	**576**+128	**128**	1	1

[a] Cycle count of NTT.
[b] The BRAM consumption of Vicuna is not provided in [11].

outperforms other state-of-the-art ISE implementations with at least 2.3× speedup in total cycle count, which demonstrates the performance advantage of the proposed RVV-based vector cryptography extension. Furthermore, our enhanced RVV implementation outperforms the implementation on the high-end 3-way issue, out-of-order Arm Cortex-A72 using assembly optimized Neon vector instructions, with 2.8× and 3.7× speedup in cycle count respectively in Kyber-512 and Dilithium-II.

VI. CONCLUSIONS

In this work, we enhance RVV with a cryptography ISE to efficiently deploy PQC applications on processors. Based on the proposed design principles of RVV-based extensions, we present a set of custom instructions that support vectorized polynomial operations of variable length and data width and can cooperate closely with RVV. A custom vector unit is designed featuring micro-ops architecture and fracturable modular arithmetic units, showing significant speedup for polynomial multiplication compared to the state-of-the-art counterparts. The proposed RVV-based cryptography ISE is implemented on the gem5 simulator to evaluate CRYSTALS-Kyber and CRYSTALS- Dilithium; results outperform the state-of-the-art implementations. We believe this work can promote the design of domain-specific extensions that utilize the rich SIMD infrastructure provided by RVV.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant 61934002.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," SIAM Rev., vol. 41, no. 2, pp. 303–332, 1999.
[2] R. Avanzi et al. Crystals-Kyber–Algorithm Specifications and Supporting Documentation. Accessed: Nov. 22, 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.
[3] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, and D. Stehle. (Nov. 2022). CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.
[4] Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R. and Perlner, R., 2022. Status report on the third round of the NIST post-quantum cryptography standardization process. NIST, Tech. Rep. NISTIR, 8413.
[5] A. S. Waterman, "Design of the RISC-V instruction set architecture,"Ph.D. dissertation, UC Berkeley, Berkeley, CA, USA, 2016.
[6] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," in Proc. IACR, Aug. 2019, pp. 17–61. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8344.
[7] E. Karabulut and A. Aysu, "RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform," 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), Gothenburg, Sweden, 2020, pp. 26-32, doi: 10.1109/FPL50879.2020.00016.
[8] . Fritzmann, G. Sigl, and J. Sepélveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," in Proc.IACR, Aug. 2020, pp. 239–280. [Online]. Available: https://tches.iacr. org/index.php/TCHES/article/view/8683

[9] Xin et al., “VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture,” IEEE Trans.Circuits Syst. I, Reg. Papers, vol. 67, no. 8, pp. 2672–2684, Aug. 2020.
[10] K. Asanovic et al, Vector Extension 1.0, 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.
[11] Platzer, Michael, and Peter Puschner. ”Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation.” In 33rd Euromicro Conference on Real-Time Systems (ECRTS 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
[12] Chang Liu. ”XuanTie C908: High-performance RISC-V Processor Catered to AIoT Industry.” RISC-V Community News(blog). November 3, 2022, https://riscv.org/blog/2022/11/xuantie-c908-high-performance-risc-v-processor-catered-to-aiot-industry-chang-liu-alibaba-cloud/.
[13] A. Baum, B. Ibrahim, B. Spinney et al. RISC-V Cryptography Extensions Volume II Vector Instructions. Accessed: Mar. 04, 2023. [Online]. Available: https://github.com/riscv/riscv-crypto/tree/master/doc/vector.
[14] Li, H., Mentens, N. and Picek, S., 2022, July. A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-kyber. In Proceedings of the 59th ACM/IEEE Design Automation Conference (pp. 733-738).
[15] T. Pöppelmann and T. Gâneysu, “Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware,” in Proc. Int. Conf.Cryptol. Inf. Secur. Latin Amer. Berlin, Germany: Springer, 2012, pp. 139–158.
[16] Longa, P. and Naehrig, M., 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15 (pp. 124-139). Springer International Publishing.
[17] National Institute of Standards and Technology. FIPS PUB 202 SHA-3 standard: Permutation-based hash and extendable-output functions. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf, 2015.
[18] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, “Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1”, TCHES, vol. 2022, no. 1, pp. 221–244, Nov. 2021.
[19] S. Gal-On and M. Levy, Exploring CoreMark - A Benchmark Maximizing Simplicity and Efficacy, The Embedded Microprocessor Benchmark Consortium, [Online,accessed 26-June-2020], https://www.eembc.org/techlit/articles/coremarkwhitepaper.pdf