A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-kyber

Li, Huimin; Mentens, Nele; Picek, Stjepan

DOI
10.1145/3489517.3530552

Publication date
2022

Document Version
Final published version

Published in
Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC 2022

A Scalable SIMD RISC-V based Processor with Customized Vector Extensions for CRYSTALS-Kyber

Huimin Li
Delft University of Technology
The Netherlands

Nele Mentens
Leiden University, The Netherlands
KU Leuven, Belgium

Stjepan Picek
Radboud University and
Delft University of Technology, The Netherlands

ABSTRACT

This paper uses RISC-V vector extensions to speed up lattice-based operations in architectures based on HW/SW co-design. We analyze the structure of the number-theoretic transform (NTT), inverse NTT (INTT), and coefficient-wise multiplication (CWM) in CRYSTALS-Kyber, a lattice-based key encapsulation mechanism. We propose 12 vector extensions for CRYSTALS-Kyber multiplication and four for finite field operations in combination with two optimizations of the HW/SW interface. This results in a speed-up of 141.7, 168.7, and 245.5 times for NTT, INTT, and CWM, respectively, compared with the baseline implementation, and a speed-up of over four times compared with the state-of-the-art HW/SW co-design using RV32IMC.

KEYWORDS

Lattice-based Cryptography, Polynomial Operation, Vector Instruction, SIMD Processor, RISC-V, ISA Extension

ACM Reference Format:
Huimin Li, Nele Mentens, and Stjepan Picek. 2022. A Scalable SIMD RISC-V based Processor with Customized Vector Extensions for CRYSTALS-Kyber. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530552

1 INTRODUCTION

Currently, the confidentiality and integrity of communication channels between multiple parties are greatly supported by public-key cryptography (PKC) algorithms. However, with the arrival of quantum computers, these PKC algorithms are not secure anymore. The main mathematical problems they rely on, the factorization of big integers and the calculation of discrete logarithms, can be solved in polynomial time using Shor's algorithm [12].

Therefore, post-quantum cryptography (PQC) algorithms, which are resistant to traditional and quantum computer attacks, are proposed. The National Institute of Standards and Technology (NIST) has initiated a post-quantum cryptography standardization process worldwide since 2016 [10]. On July 22, 2020, NIST announced the 15 candidates for Round three [10]. Among these PQC algorithms, lattice-based algorithms occupy seven places. Thanks to its security and efficiency, lattice-based cryptography can be used for many security applications such as key-encapsulation mechanisms (KEMs), identity-based encryption (IBE) [20], and Fully Homomorphic Encryption (FHE) [9]. The implementation of lattice-based algorithms is a prominent research area. There are three types of strategies: pure hardware (HW) design, pure software (SW) design, and hardware/software (HW/SW) co-design [1]. Among those, HW/SW co-design combines the advantages of the other two, which are high-speed and flexibility, by partitioning the whole design into two parts: the hardware part implemented on FPGA or ASIC, and the software part in one or more processors that can be embedded in the FPGA or ASIC.

Lattice-based algorithms work with many costly polynomial operations with a high degree. Especially polynomial multiplication is believed to be one of the bottlenecks in lattice-based implementations [1]. The number-theoretic transform (NTT), a specialized form of the Discrete Fourier Transform (DFT) [7], is used by some lattice-based algorithms such as CRYSTALS-Kyber, CRYSTALS-Dilithium, and Fully Homomorphic Encryption. Even though NTT can reduce the time complexity from $\mathbf{O(n^2)}$ (for traditional DFT algorithms) to $\mathbf{O(nlog(n))}$, the algorithm is still very time-consuming.

Polynomial operations are suitable for working in a data-parallel operation mode through vector architectures, also called Single-Instruction-Multiple-Data (SIMD) architectures. One crucial requirement to implement SIMD processors is to have a vector instruction set architecture (ISA) that is preferably free and open-source. Fortunately, vector extensions for the RISC-V ISA are available. The most recent version is RVV1.0, the 1.0 version of the RISC-V vector extensions (RVV). To our knowledge, there is only one work [13] that adopts RVV for the implementation of PQC. In [13], the authors use RVV in Classic McEliece, a PQC algorithm based on code-based cryptography. For lattice-based cryptography, performance improvements using RVV are still unexplored.

To fill in the gap, we use RISC-V vector extensions to improve the efficiency of lattice-based operations based on HW/SW co-design. We first realize a scalable SIMD processor written in SystemVerilog to support RVV1.0. Then, we analyze the structure of the number-theoretic transform (NTT), inverse NTT (INTT), and coefficient-wise multiplication (CWM) in CRYSTALS-Kyber, a lattice-based key encapsulation mechanism. Later, we propose two optimizations of the HW/SW interface and 16 vector extensions: 12 for CRYSTALS-Kyber multiplication and four for finite field operations. Our contributions are the following:

- We realize a scalable SIMD processor supporting RISC-V vector extensions and implement it on a Xilinx Alveo U250 accelerator card.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530552

- We propose two HW/SW interface optimizations and 16 vector extensions for CRYSTALS-Kyber multiplication and finite field operations. Our results show a speed-up of 141.7, 168.7, and 245.5 times for NTT, INTT, and CWM, respectively, compared with the baseline implementation, and a speed-up of over four times compared with the state-of-the-art HW/SW co-design using the RV32IMC ISA.

2 NOTATION AND BACKGROUND INFORMATION

We use lower-case italic letters like p to denote polynomials, while lower-case bold letters like $\mathbf{p}$ are used to denote vectors of polynomials, and upper-case bold letters like $\mathbf{P}$ denote matrices of polynomials. Furthermore, we use $\hat{p}$, $\hat{\mathbf{p}}$, and $\hat{\mathbf{P}}$ to represent these variables in the corresponding NTT domain. Further, let $\mathbf{v}^T$ be the transpose of the vector $\mathbf{v}$ and $\mathbf{A}^T$ be the transpose of the matrix $\mathbf{A}$. We define $\mathbf{v}[i]$ to denote a vector $\mathbf{v}$'s i-th entry (where i starts from zero), and $\mathbf{A}[i][j]$ to denote the entry in row i and column j in a matrix $\mathbf{A}$. We define polynomial rings R_q as $\mathbb{Z}_q[X]/\phi(x)$. Here, $\phi(x)$ is $(X^n + 1)$, q is a prime, and n is a power of two. We use $\mathbf{NTT}$, $\mathbf{NTT^{-1}}$, and $\mathbf{CWM}$ for the corresponding functions. We use $\cdot$ to denote integer and polynomial multiplication, and use $\circ$ to denote coefficient-wise multiplication. For two vectors of polynomials, $\mathbf{f}$ and $\mathbf{g}$, the product $\mathbf{f} \cdot \mathbf{g}$ can be computed efficiently as $\mathbf{NTT^{-1}}(\mathbf{NTT}(\mathbf{f}) \circ \mathbf{NTT}(\mathbf{g}))$. Finally, we denote messages as m, ciphertexts as ct, public keys as pk, and secret keys as sk.

2.1 CRYSTALS-Kyber

CRYSTALS-Kyber is a lattice-based cryptosystem of which the security is based on the hardness of the Module Learning With Errors (MLWE) problem, with q equal to 3 329 and n 256 [2]. Its public-key encryption scheme (Kyber.CPAPKE) features indistinguishability under chosen plaintext attack (IND-CPA) and includes three steps: key generation (KeyGen), encryption (Enc), and decryption (Dec) [2]. These three steps can be summarized as follows, assuming that $\hat{\mathbf{A}} \in R_q^{k \times k}$ is generated through uniform sampling, and $\mathbf{s} \in R_q^k$, $\mathbf{e} \in R_q^k$, $\mathbf{r} \in R_q^k$, $\mathbf{e}_1 \in R_q^k$ and $e_2 \in R_q$ are generated through centered-binomial-distribution sampling [2, 3]:

KeyGen: $pk := \hat{\mathbf{A}} \circ \mathbf{NTT}(\mathbf{s}) + \mathbf{NTT}(\mathbf{e}), sk := \mathbf{NTT}(\mathbf{s})$.

Enc: $ct := (\mathbf{u}, v)$, with $\mathbf{u} = (\mathbf{NTT^{-1}}(\hat{\mathbf{A}}^T \circ \mathbf{NTT}(\mathbf{r})) + \mathbf{e}_1$ and $v = \mathbf{NTT^{-1}}(pk^T \circ \mathbf{NTT}(\mathbf{r})) + e_2 + m$.

Dec: $m := v - \mathbf{NTT^{-1}}(\hat{\mathbf{s}}^{\mathbf{T}} \circ \mathbf{NTT}(\mathbf{u}))$.

2.2 Number-theoretic Transform

For the number-theoretic transform (NTT), when $\phi(x)$ is of form x^n+1, the negative wrapped convolution [14] is used to directly compute polynomial multiplication with coefficients in R_q. For a vector $f = \sum_{i=0}^{n-1} f_i x^i$, its NTT operation transform is $\hat{f} = \mathbf{NTT}(f) = \sum_{i=0}^{n-1} \hat{f}_i X^i$ with $\hat{f}_i = \sum_{j=0}^{n-1} \psi^j f_j \omega^{ij} (\bmod\ q)$ for $i = 0, 1, \ldots, n-1$. ω is the twiddle factor, defined as the n-th root of unity, with the conditions that $\forall i < n, \omega^i \neq 1 (\bmod\ q)$ and $\omega^n \equiv 1 (\bmod\ q)$. $\psi = \sqrt{\omega}$. Similarly, the corresponding inverse (INTT) operation is defined as $f = \mathbf{NTT^{-1}}(\hat{f}) = \sum_{i=0}^{n-1} f_i X^i$ with $f_i = n^{-1}\psi^{-i} \sum_{j=0}^{n-1} \hat{f}_j \omega^{-ij} (\bmod\ q)$ for $i = 0, 1, \ldots, n-1$.

The negative wrapped convolution technique dramatically improves the work efficiency of NTT and INTT by eliminating the doubling of the sizes of inputs with zero padding and a separate polynomial reduction operation by $\phi(x)$ [14]. However, it adds pre-processing and post-processing by multiplying with ψ^j or ψ^{-i}. Since Round 2 of the NIST PQC competition, the parameter q in CRYSTALS-Kyber has been reduced from 7 681 to 3 329, eliminating the need for pre-processing and post-processing operations. The new NTT operation requires an early termination and generates 128 polynomials with a degree of two. Analogously, the INTT operation processes 128 degree-2 polynomials, and an extra coefficient-wise multiplication (CWM) is required to multiply two degree-2 polynomials in $\mathbb{Z}_q[x]/(x^2 - \omega^i)$. In [19] and [17], the authors use a technique, named DIVby2, to eliminate the multiplication with $n^{-1}(\bmod\ q)$ after the butterfly structure of the INTT operation. That is, when x is even, $x/2(\bmod\ q)$ equals $(x \gg 1)$, while when x is odd, $x/2(\bmod\ q) = (x \gg 1) + x[0] \times ((q+1)/2)$. The three algorithms are shown in Algorithms 1, 2, and 3, respectively, where $br_{l-1}(\cdot)$ is the bit-reversal operation for a word size of $l - 1$ [16, 17].

Algorithm 1 NTT Algorithm in CRYSTALS-Kyber

```
Input: $f(x) \in R_q$, $\omega_n \in \mathbb{Z}_q$, $n = 2^l$.
Output: $\hat{f}(x) \in R_q$
$k \leftarrow 1$
for $i$ from 1 by 1 to $l - 1$ do
    $m \leftarrow 2^{l-i}$
    for $s$ from 0 by $m$ to $n$ do
        for $j$ from $s$ by 1 to $s + m$ do
            $\mathbf{a}, \mathbf{b}, \mathbf{w} \leftarrow f[j], f[m+j], \omega^{br_{l-1}(k)} \bmod q$
            $\mathbf{t} \leftarrow (\mathbf{w} \cdot \mathbf{b}) \bmod q$
            $\mathbf{e}, \mathbf{o} \leftarrow (\mathbf{a} + \mathbf{t}) \bmod q, (\mathbf{a} - \mathbf{t}) \bmod q$
        end for
        $k \leftarrow k + 1$
    end for
end for
```

Algorithm 2 INTT Algorithm in CRYSTALS-Kyber

```
Input: $\hat{f}(x) \in R_q$, $\omega_n^{-1} \in \mathbb{Z}_q$, $n = 2^l$
Output: $f(x) \in R_q$
$k \leftarrow 0$
for $i$ from $l - 1$ by $-1$ to 1 do
    $m \leftarrow 2^{l-i}$
    for $s$ from 0 by $m$ to $2^l$ do
        for $j$ from $s$ by 1 to $s + m$ do
            $\mathbf{a}, \mathbf{b}, \mathbf{w} \leftarrow \hat{f}[j], \hat{f}[j+m], \omega^{br_{l-1}(k)+1} \bmod q$
            $\mathbf{e}, \mathbf{o} \leftarrow (\mathbf{a} + \mathbf{b}) \bmod q, (\mathbf{a} - \mathbf{b}) \cdot \mathbf{w} \bmod q$
            $\hat{f}[j], \hat{f}[j+m] \leftarrow$ DIVby2 $(\mathbf{e})$, DIVby 2$(\mathbf{o})$
        end for
        $k \leftarrow k + 1$
    end for
end for
```

Algorithm 3 CWM Algorithm in CRYSTALS-Kyber

```
Input: $\hat{f}(x), \hat{g}(x) \in R_q$, $\omega \in \mathbb{Z}_q$
Output: $\hat{c}(x) \in R_q$
for $i$ from 0 by 1 to $2^{l-1}$ do
    $\mathbf{w} \leftarrow \omega^{br_{l-1}(i)+1} \bmod q$
    $\mathbf{a_0}, \mathbf{a_1} \leftarrow \hat{f}[2i], \hat{f}[2i+1]$
    $\mathbf{b_0}, \mathbf{b_1} \leftarrow \hat{g}[2i], \hat{g}[2i+1]$
    $\hat{c}[2i] \leftarrow (\mathbf{a_0} \cdot \mathbf{b_1} + \mathbf{a_1} \cdot \mathbf{b_0}) \bmod q$
    $\hat{c}[2i+1] \leftarrow (\mathbf{a_1} \cdot \mathbf{b_1} \cdot \mathbf{w} + \mathbf{a_0} \cdot \mathbf{b_0}) \bmod q$
end for
```

3 SYSTEM DESIGN

3.1 SIMD Processor Design

Similar to [11] and [18], the proposed SIMD processor in our paper contains two parts, as illustrated in Figure 1: a scalar core (top) and a vector processing unit (bottom). To accelerate the design process, we use the existing RISC-V core, Ibex [8], as the scalar core. Ibex is a two-stage, 32-bit open-source core, written in SystemVerilog [8]. The two parts interface with each other through vector instructions, scalar registers, and memory data.

The vector unit consists of four modules: Vector Instruction Interface (**VecISAInterface**), Vector Load and Store Unit (**VecLSU**), Vector Register File (**VecRegfile**), and Vector Operation Execution (**VecOpExec**), as shown in Figure 1. The **VecISAInterface** module decodes the vector instructions, which are fetched and transferred from the scalar core. It decouples these instructions into configuration-setting instructions, memory instructions, and vector arithmetic instructions. Then, the configuration-setting instructions are processed inside the **VecISAInterface** module; the memory instructions are sent to the **VecLSU** module, and vector arithmetic instructions are sent to the **VecOpExec** module. In the **VecLSU** module, the memory instructions are decoupled into vector load instructions and vector store instructions. In the **VecOpExec** module, the vector arithmetic instructions are decoded further into different operations by the Arithmetic Operation Pre-Processing (**ArithOpPrepro**) submodule, according to the two fields of funct3 and funct6 in the instruction. And then, the exact instructions are sent to the execution modules, all of which are in the same Execution Lane (**ExLane**) sub-module. The lane number (*LaneNum*) parameter defines the number of **ExLane** sub-modules instantiated in the SIMD architecture.

3.1.1 *Vector Register File.*

Besides the scalar register file inside the Ibex core, another vector register file is foreseen inside the vector processing unit. According to the RVV1.0 specification, there should be in total 32 vector registers [15]. In each vector register, there are several vector elements. The width of every element is defined by the parameter *ELEN*. To be compatible with the Ibex core, *ELEN* is fixed to 32-bit in this work. The width of the vector registers is defined by the parameter *VLEN*. Consequently, *LaneNum* is determined by *VLEN*/*ELEN*. That is, a vector register is viewed as being divided into *VLEN*/*ELEN* elements [15]. *VL* and *LMUL* are two other important parameters in RVV1.0. *VL* is the vector length and specifies the number of elements to be operated on in parallel within a vector extension. It can be less or greater than *LaneNum*. When *VL* is less than *LaneNum*, all elements are put in the same vector register. When *VL* is greater than *LaneNum*, several vector registers are grouped. RVV1.0 [15] defines the parameter *LMUL*, the vector length multiplier, to specify the number of vector registers that are grouped.

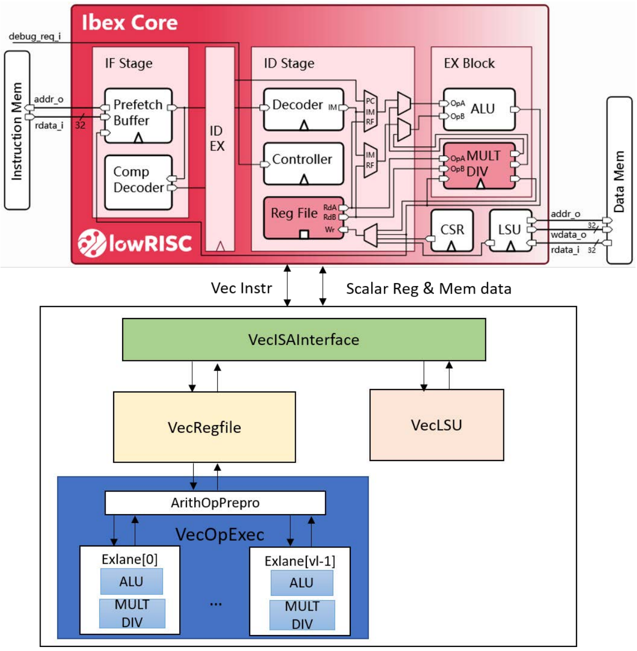


Figure 1: The architecture of the SIMD RISC-V based Processor.

Figure 2 shows an example with *LaneNum* = 4 and *VL* = 8. In this case, *LMUL* should be set to 2. As defined in the RVV specification, the maximum value of *LMUL* is 8. When *LMUL* is greater than one, the base address of each vector that uses the vector register file changes. For the instruction: {vadd.vv v0,v0,v1}, the base address of v0 is zero, while the base address of v1 is two. The **VecISAInterface** module takes care of the address allocation.

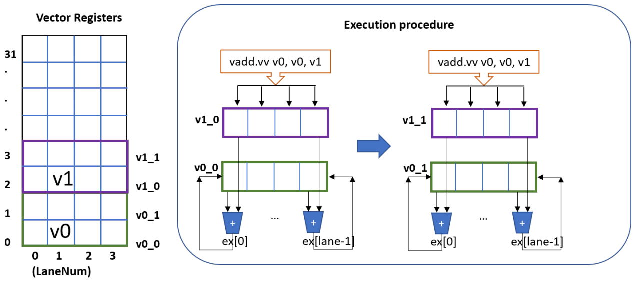


Figure 2: Vector register file and address allocation.

3.1.2 *Vector Load and Store.*

The vector unit shares the same datapath as the scalar core to read and write scalar registers and load and store memory data. The data from the two read ports in the scalar register file are always ready. The writing to the scalar register file is only enabled when a configuration-setting instruction is processed.

For vector load and store instructions, our SIMD processor supports all three different types of address modes as specified in the RVV1.0, including vector unit-stride mode, vector constant-strided mode, and vector indexed mode [15]. Vector unit-stride mode accesses contiguous elements in memory, starting from the base address. Vector constant-strided mode accesses memory elements with a constant address space bigger than the width of one element in memory, starting from the base address. The vector indexed mode accesses several elements with their address offset value given by a vector and the base address provided by a scalar register.

The vector load and store instructions are the only ones that cannot do data-parallel implementations because only one RAM address can be set, and only one RAM element can be accessed. Thus, these memory instructions are the most time-consuming in the SIMD processor. When the store instruction is triggered, the **VecLSU** module first reads all elements from the first vector register address. It then sends these elements to Data RAM one by one corresponding to the lane order from zero to {*LaneNum* - 1}. Then, all elements from the following vector data registers belonging to the same vector will be fetched in sequence and sent to RAM through the procedure mentioned above. The process of the load instruction is the reverse process of the store instruction. The required data will be fetched from RAM with the address order defined by the three different modes. All readout data will be sent directly to the vector register file.

3.1.3 Vector Execution. In the **VecOpExec** module, the **ArithOp-Prepro** sub-module further decodes the vector arithmetic instructions based on the three-bit funct3 and six-bit funct6 fields. The funct3 field is to specify sub-categories of arithmetic instructions: whether the two operands are vector-vector (.vv), vector-immediate (.vi), or vector-scalar (.vx), and whether the corresponding operations are integer operations, multiply/division (MULT/DIV) operations, or fixed-point operations. The funct6 field specifies the operation type, for example, whether the operations are addition, shift, multiplication, etc.

As shown in Figure 2, after the instruction {vadd.vv v0,v0,v1} is sent to the **VecOpExec** module, it is recognized as an integer operation with two operands to be vector-vector (.vv), and the operation code to be an addition. Then the two vectors: v0 and v1, will be read from the vector register file, with the vectors' base addresses set to zero and two, respectively. All elements from the first vector register are read out at the same time. Two elements from the vector v0 and v1 in Figure 2, with the same index number (or lane order), will be sent to the same **ExLane** sub-module for the addition operation. After the addition operation finishes, the result from every **ExLane** sub-module will be sent to vector v0 according to the index number. Then, all elements from the following vector registers belonging to the same vector will be fetched in sequence. Again, two elements with the same index number will be sent to their corresponding **ExLane** sub-module, and the result from every **ExLane** sub-module will be written back to vector v0. The parameter *LMUL* defines the total number of operations.

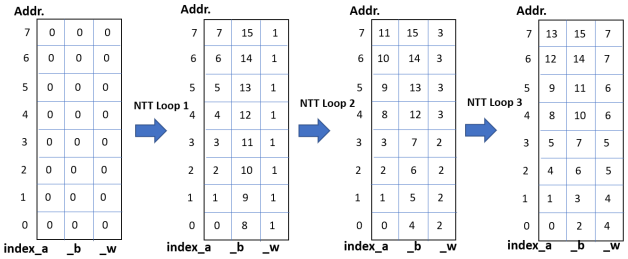
3.2 NTT Design

We propose two HW/SW interface optimizations to improve the performance of our architecture: register pooling and automatic index generation. Further, we propose custom vector instructions for NTT and for finite field arithmetic operations.

3.2.1 Register Pooling. We use the term register pool for multiple registers doing the same job. Unlike RAM, where there is often only one address that can be set, the data in the same register pool operate independently, and multiple data can be read and written simultaneously. The purpose of applying register pooling is to increase the loading and storing throughput in every loop of NTT, INTT, and CWM and eliminate the time lost when exchanging data with the Data RAM. Three types of register pools are proposed in this design to support the parallel computation of the NTT, INTT, and CWM algorithms in CRYSTALS-Kyber.

The first register pool, named **coeff_data**, is used to store coefficient data. There are two register sub-pools in **coeff_data**, called $\textbf{coeff\_data}_\textbf{0}$ and $\textbf{coeff\_data}_\textbf{1}$, respectively, in which there are 256 12-bit registers to store all polynomial coefficients in one NTT vector. $\textbf{coeff\_data}_\textbf{0}$ serves as temporary storage for the coefficient data of the NTT and INTT algorithms, and for the first coefficient data in the CWM algorithm. $\textbf{coeff\_data}_\textbf{1}$ serves as temporary storage for the second coefficient data in the CWM algorithm. The second register pool, called **poly_index**, is used to store the index number for each loop. There are three register sub-pools in **poly_index**, called $\textbf{poly\_index}_\textbf{a}$ and $\textbf{poly\_index}_\textbf{b}$, and $\textbf{poly\_index}_\textbf{w}$, respectively, in which there are 128 7-bit registers to store the index number of **a**, **b** and **w** in Algorithms 1, 2, and 3. The third register pool, named **tw**, has 128 12-bit registers to store the twiddle factors. The initial value of all twiddle factors is pre-calculated and stored in bit-reversal order, and updated to different values according to the type of algorithms.

3.2.2 Automatic Index Generation. Before the three algorithms get started, all polynomials in one vector are stored in the register pool **coeff_data**. That is, the result of the previous operation is not sent back to the Data RAM but stored here in **coeff_data**. Our design keeps the outer loop structure and unloops the inner two loop structures (Algorithms 1 and 2). Customized vector extensions control the loop number of the outermost layer. The register pool **poly_index** changes automatically according to the loop number. In Figure 3, we illustrate the processing of a vector in NTT with the polynomial number, the index and the loop number equal to 16, 8 and 3, respectively.



Addr.	index_a	_b	_w	Addr.	index_a	_b	_w	Addr.	index_a	_b	_w	Addr.	index_a	_b	_w
7	0	0	0	7	7	15	1	7	11	15	3	7	13	15	7
6	0	0	0	6	6	14	1	6	10	14	3	6	12	14	7
5	0	0	0	5	5	13	1	5	9	13	3	5	9	11	6
4	0	0	0	4	4	12	1	4	8	12	3	4	8	10	6
3	0	0	0	3	3	11	1	3	3	7	2	3	5	7	5
2	0	0	0	2	2	10	1	2	2	6	2	2	4	6	5
1	0	0	0	1	1	9	1	1	1	5	2	1	1	3	4
0	0	0	0	0	0	8	1	0	0	4	2	0	0	2	4

Figure 3: Automatic index generation for a, b, and w in NTT

In each loop, vector **a** and vector **b** are read from register pool **coeff_data**, and vector **w** is read from register pool **tw**. Their polynomial order is changed according to register pool $\textbf{poly\_index}_\textbf{a}$,

Table 1: Customized vector extensions in the SIMD processor.

Instruction Type	Instructions	Description	Latency
Customized Polynomials Load	*vlpolye8/16/32*	Load Polynomials from Data RAM to **coeff_data** with element width of 8-bit/16-bit/32-bit.	1+*VL*
Customized Polynomials Store	*vspolye8/16/32*	Store Polynomials **coeff_data** to Data RAM with element width of 8-bit/16-bit/32-bit.	1+*VL*
Customized Multiplication Configuration	*vnttcfg*	Configure multiplication type to be NTT, and set loop number.	2
	vinttcfg	Configure multiplication type to be INTT, and set loop number.	2
	vcwcfg	Configure multiplication type to be CWM, and set loop number.	2
Customized Polynomials Read	*vreadpoly*	Read polynomial from **coeff_data** to Vector Register File.	1+*LMUL*
Customized Polynomials Write	*vwritepoly*	Write polynomial from Vector Register File to **coeff_data**.	
Customized Twiddle Factor Read	*vreadtw*	Read twiddle factors from **tw** to Vector Register File.	
Finite Field Addition	*vaddmod*	Finite Field Addition	
Finite Field Subtraction	*vsubmod*	Finite Field Subtraction	
Modular Reduction	*vmod*	Modular Reduction	
Finite Field Division by Two	*vdivby2*	Divide the butterfly output by two in the Finite Field	

poly_index$_b$, and **poly_index$_w$**, respectively. Later, the re-ordered vectors **a**, **b** and **w** are stored in the destination vector registers for the consecutive arithmetic operations. After all operations in one loop are finished, the order of polynomials in vectors **a** and **b** will be changed back to their initial order according to **poly_index$_a$** and **poly_index$_b$**, and written back to register pool **coeff_data**. Note that vector **w** is not sent to **tw** because it does not change with the loop number. The whole process is illustrated in Figure 4, where all parameters are the same as in Figure 3.

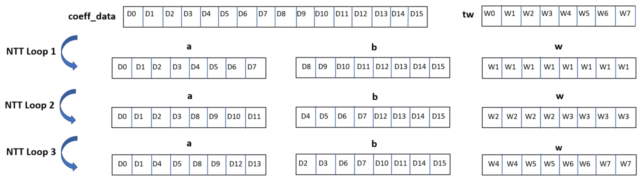

coeff_data
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15
tw
W0 W1 W2 W3 W4 W5 W6 W7
NTT Loop 1
a
D0 D1 D2 D3 D4 D5 D6 D7
b
D8 D9 D10 D11 D12 D13 D14 D15
w
W1 W1 W1 W1 W1 W1 W1 W1
NTT Loop 2
a
D0 D1 D2 D3 D8 D9 D10 D11
b
D4 D5 D6 D7 D12 D13 D14 D15
w
W2 W2 W2 W2 W3 W3 W3 W3
NTT Loop 3
a
D0 D1 D4 D5 D8 D9 D12 D13
b
D2 D3 D6 D7 D10 D11 D14 D15
w
W4 W4 W5 W5 W6 W6 W7 W7

Figure 4: The polynomial order changes according to the loop number in NTT.

3.2.3 *Customized Vector Instructions for NTT.*

There are usually three methods to extend instructions in RISC-V: 1) using custom instructions; 2) modifying the compiler; 3) rewriting unused existing instructions. In [1] and [5], the authors use the first method, while the second method is adopted in [4]. Modifying the compiler is often too time-consuming and inflexible because the toolchain needs to be configured whenever one instruction changes. In this paper, we use the first and third methods.

To realize the above mentioned operations in 3.2, we use custom instructions, including custom_0 and custom_1, to extend the specific vector extensions for multiplication in CRYSTALS-Kyber, see Table 1. In our design, all these vector extensions are R-Type [1, 5]. The two source operands and the destination operand can be scalar registers or vector registers. We design 12 customized Vector extensions for NTT, which belong to six categories.

Polynomial Load Extensions include *vlpolye8*, *vlpolye16*, and *vlpolye32*. They are used to load data from Data RAM to the vector register file with a data width of 8-bit, 16-bit, and 32-bit, respectively.

Polynomial Store Extensions include *vspolye8*, *vspolye16*, and *vspolye32*. They are used to store data from the vector register file to Data RAM with a data width of 8-bit, 16-bit, and 32-bit, respectively.

Multiplication Configuration Extensions include *vnttcfg*, *vinttcfg*, and *vcwcfg*. They configure the multiplication to NTT, INTT, and CWM, respectively. They also set the loop number.

Polynomial Read Extension includes *vreadpoly*. It is used to read a polynomial from **coeff_data** to the vector register file.

Polynomial Write Extension includes *vwritepoly*. It is used to write polynomials from the vector register file to **coeff_data**.

Twiddle Factor Read Extension includes *vreadtw*. It is used to read twiddle factors from **tw** to the vector register file.

Table 2: Resource usage for SIMD Processor supporting CRYSTAL-Kyber multiplication.

Lane Num	LUT	LUTRAM	FF	BRAM	DSP
0	2.4K	48	890	16	4
4	45.5K	48	13.3K	16	26
8	93.2K	48	17.9K	16	42
16	166.1K	48	27.2K	16	74
32	318.2K	48	46.0K	16	138

3.2.4 *Optimization for finite field arithmetic operations.*

In this work, we also extend four vector extensions for finite field operations using the third method mentioned in 3.2.3, including *vaddmod*, *vsubmod*, *vmod*, *vdivby2*, as listed in Table 1. We define *vaddmod* for finite field addition, *vsubmod* for finite field subtraction, *vmod* for modular reduction, and *vdivby2* for $x/2 \bmod q$ after the INTT operation, as described in Section 2.2. What is worth mentioning here is the modular reduction operation, *vmod*. We adopt the technique proposed in [17] to reduce the latency to one clock cycle by utilizing the property that $2^{12} \equiv 2^9 + 2^8 - 1 (\bmod 3329)$.

4 EXPERIMENTAL RESULTS

We first develop the scalable SIMD processor using SystemVerilog and select a Xilinx Alveo U250 Data Center accelerator card for FPGA evaluation. The Alveo U250 has rich resources to support multiple lanes, within a total of 1 728K LUTs, 791K LUTRAM, 3 456K flip-flops, 2 688 BRAM, 12 288 DSP, 676 IO, 1 344 BUFG, and 32 PLL. After completing the behavioral simulations using Vivado 2019.2, we set *LaneNum* to 4, 8, 16, and 32, respectively. The four different architectures and the original IBex core (zero lanes) are synthesized and implemented through Vivado 2019.2 using the Alveo U250 card. The resource usage is shown in Table 2, where the LUT, LUTRAM, FF, BRAM, and DSP usage is compared.

The next step is to optimize the NTT, INTT, and CWM algorithms. We use the RISC-V GNU Compiler Toolchain (version rvv-intrinsic)[1]. Similar to [4], we set the optimization flag to 'O3' to compile the code and the baseline implementations to the clean

[1] https://github.com/riscv-collab/riscv-gnu-toolchain/tree/rvv-intrinsic

Table 3: Execution time for different values of *LaneNum* in our SIMD processor and comparisons with the baseline implementation and [4].

Test	[4]	C-baseline (Ibex)	Our SIMD Processor							
			Lane4		Lane8		Lane16		Lane32	
	Cycles	Cycles	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
NTT	1 935	54 261	3 022	18	1 538	35.3	796	68.2	383	141.7
INTT	1 930	76 413	3 582	21.3	1 818	42	936	81.6	453	168.7
CWM	—	28 228	926	30.5	466	60.6	236	119.6	115	245.5

C-code of the PQ-M4 project [6]. First, we run the baseline code on the pure IBex core, the clock cycle count for the three algorithms are 54 261, 76 414, and 28 228, respectively. Then we optimize these three algorithms using RV32IMC, RVV1.0, and customized vector extensions for CRYSTALS-Kyber multiplication and finite field operations. Again, we set the *LaneNum* to 4, 8, 16, and 32 and then count the clock cycles for the NTT, INTT, and CWM algorithms. All results are shown in Table 3. From the results, we can see that the execution time of NTT, INTT, and CWM in our design is optimized by 141.7, 168.7, and 245.5 times respectively, compared to the baseline when the *LaneNum* is set to 32. When compared with relevant related work in [4], which is a RISC-V based HW/SW co-design written in SystemVerilog using the RV32IMC ISA, the execution times of NTT and INTT are optimized by nearly 5.1 and 4.3 times, respectively.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we explore RISC-V vector extensions to improve the efficiency of lattice-based operations based on HW/SW co-design. We first realize a scalable SIMD processor written in SystemVerilog to support RVV1.0. And then, we analyze the structure of the three polynomial multiplication algorithms in CRYSTALS-Kyber, namely NTT, INTT, and CWM. We propose two techniques, called register pooling and automatic index generation, to optimize the HW/SW interface and design 12 vector extensions for CRYSTALS-Kyber multiplication and 4 for finite field operations. Our results show a speed-up of 141.7, 168.7, and 245.5 times for NTT, INTT, and CWM, respectively, compared with the baseline implementation, and a speed-up of over four times compared with state-of-the-art HW/SW co-design using RV32IMC. In future work, we will focus on the vectorization of the Keccak core and the whole CRYSTALS-Kyber cryptosystem. Additionally, we will also consider countermeasures against side-channel attacks on SIMD architectures. We will publish all our code to facilitate follow-up research.

REFERENCES

[1] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. 2020. ISA Extensions for Finite Field Arithmetic. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 219–242.

[2] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2017. CRYSTALS-Kyber algorithm specifications and supporting documentation. *NIST PQC Round* 2, 4 (2017).

[3] Zhaohui Chen, Yuan Ma, Tianyu Chen, Jingqiang Lin, and Jiwu Jing. 2020. Towards efficient Kyber on FPGAs: A processor for vector of polynomials. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 247–252.

[4] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 239–280.

[5] Elif Nur İşman, Canberk Topal, Latif Akçay, and Berna Örs. 2020. Instruction Extension of an Open Source RV32IMC Core for NTRU Cryptosystem. In *2020 European Conference on Circuit Theory and Design (ECCTD)*. IEEE, 1–5.

[6] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844. https://ia.cr/2019/844.

[7] Patrick Longa and Michael Naehrig. 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*. Springer, 124–139.

[8] lowRISC. 2021. Ibex Documentation. https://ibex-core.readthedocs.io/en/latest/01_overview/index.html.

[9] Liam Morris. 2013. Analysis of partially and fully homomorphic encryption. *Rochester Institute of Technology* (2013), 1–5.

[10] NIST. 2016. NIST Post Quantum Cryptography Standardization. https://en.wikipedia.org/wiki/NIST_Post-Quantum_Cryptography_Standardization.

[11] Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch Sirakoulis, and Giorgos Dimitrakopoulos. 2020. RISC-V 2: A Scalable RISC-V Vector Processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[12] Ray A Perlner and David A Cooper. 2009. Quantum resistant public key cryptography: a survey. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet*. 85–93.

[13] Sabine Pircher, J Geier, Alexander Zeh, and Daniel Mueller-Gritschneder. 2021. Exploring the RISC-V Vector Extension for the Classic McEliece Post-Quantum Cryptosystem. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 401–407.

[14] Thomas Pöppelmann and Tim Güneysu. 2012. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *International conference on cryptology and information security in Latin America*. Springer, 139–158.

[15] RISCVteam. 2021. riscv-v-spec-1.0. https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[16] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. 2020. 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 (2020), 2672–2684.

[17] Ferhat Yarman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2021. A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1020–1025.

[18] Jason Yu, Guy Lemieux, and Christpher Eagleston. 2008. Vector processing as a soft-core CPU accelerator. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. 222–232.

[19] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 49–72.

[20] Xiaojun Zhang, Yao Tang, Huaxiong Wang, Chunxiang Xu, Yinbin Miao, and Hang Cheng. 2019. Lattice-based proxy-oriented identity-based encryption with keyword search for cloud storage. *Information Sciences* 494 (2019), 193–207.