ISA Extensions for Finite Field Arithmetic

Accelerating Kyber and NewHope on RISC-V

Erdem Alkim[1], Hülya Evkan[1], Norman Lahr[1], Ruben Niederhagen[1] and Richard Petri[1]

Fraunhofer SIT, Darmstadt, Germany
erdemalkim@gmail.com, hevkan@gmail.com, norman.lahr@lahr.email,
ruben@polycephaly.org, rp@rpls.de

Abstract. We present and evaluate a custom extension to the RISC-V instruction set for finite field arithmetic. The result serves as a very compact approach to software-hardware co-design of PQC implementations in the context of small embedded processors such as smartcards. The extension provides instructions that implement finite field operations with subsequent reduction of the result. As small finite fields are used in various PQC schemes, such instructions can provide a considerable speedup for an otherwise software-based implementation. Furthermore, we create a prototype implementation of the presented instructions for the extendable VexRiscv core, integrate the result into a chip design, and evaluate the design on two different FPGA platforms. The effectiveness of the extension is evaluated by using the instructions to optimize the Kyber and NewHope key-encapsulation schemes. To that end, we also present an optimized software implementation for the standard RISC-V instruction set for the polynomial arithmetic underlying those schemes, which serves as basis for comparison. Both variants are tuned on an assembler level to optimally use the processor pipelines of contemporary RISC-V CPUs. The result shows a speedup for the polynomial arithmetic of up to 85% over the basic software implementation. Using the custom instructions drastically reduces the code and data size of the implementation without introducing runtime-performance penalties at a small cost in circuit size. When used in the selected schemes, the custom instructions can be used to replace a full general purpose multiplier to achieve very compact implementations.

Keywords: PQC · lattice-based crypto · NewHope · Kyber · RISC-V · ISA extension

1 Introduction

Since the beginning of the 21st century, the developments in quantum computing have been building up momentum. Given the current state of the art with the first small but promising quantum computers in the research labs of Google, IBM, and Microsoft, it is general consensus that we can expect larger quantum computers to be available and operational in the next decades. The most prominent quantum algorithms are Grover's algorithm, which gives a square-root speedup on brute-force search problems, and Shor's algorithm, which solves the integer factorization and the discrete logarithm problem in polynomial time. Both algorithms threaten information security: While the impact of Grover's algorithm generally can be mitigated by tuning security parameters (e.g., by doubling key sizes of symmetric ciphers), Shor's algorithm renders most of the currently used asymmetric cryptography like RSA, DSA, and DH as well as Elliptic Curve Cryptography (ECC) insecure.

These developments in quantum computing and the devastating impact of Shor's algorithm on our current IT security has spawned active research into alternative cryptographic systems that are secure against attacks with the aid of quantum computers. This field of research is called Post-Quantum Cryptography (PQC). There are five popular families of PQC schemes: code-based, hash-based, lattice-based, multivariate, and isogeny-based cryptography. The research in PQC has culminated in a NIST standardization process that has started in December 2016 and is expected to be finished in 2022 to 2024.

An important aspect of PQC research is the efficient and secure implementation of PQC algorithms. In contrast to current algorithms, which require efficient arithmetic on medium to large integers of a few hundred bits (e.g., modular arithmetic for ECC) to a few thousand bits (e.g. RSA-2048), many PQC schemes require operations over small finite fields with a size of typically less than 20 bits. In this paper, we investigate the impact of providing finite field extensions to an Instruction Set Architecture (ISA) on the performance of PQC schemes with the example of the lattice-based key-encapsulation mechanisms Kyber and NewHope (see Section 2.1). Our metrics are:

- cycle count: What speedup can we gain?
- issued instructions: How efficiently are the cycles used?
- clock frequency: What is the impact on the longest path?
- wall-clock time: Can we exploit improvements in cycle count?
- area: What is the cost in FPGA resources?
- time-area product: What is the trade-off between time and area?

In order to perform these measurements, we require a prototype implementation of a CPU with our experimental ISA extensions and optimized software using this extension. To build such a platform, we need an open ISA, which enables us to add extensions, and an open implementation of this ISA for FPGAs, which we can easily be used as basis for our new finite-field instruction-set extensions. The RISC-V ISA and its VexRiscv implementation (see Section 2.2) are an ideal platform for these requirements.

Related work. For lattice-based schemes, there exist three classes of implementations: pure software implementations, dedicated hardware modules, and software-hardware codesigns. In the survey paper [NDR+19] the spectrum of existing implementation variants for lattice-based schemes is summarized. It covers implementations on general purpose processors as well as implementations on resource constraint microcontrollers and FPGAs. Some implementations also utilize graphics cards or Digital Signal Processors (DSPs) for acceleration. Furthermore, the authors identified discrete noise sampling (Gaussian or Binomial) and matrix or polynomial multiplications, which also include the finite field arithmetic, as the main bottlenecks over all lattice-based schemes. The most common approach used for polynomial multiplication is the Number-Theoretic Transform (NTT). Among others, in software implementations the modular arithmetic is optimized by platform-specific Single-Instruction-Multiple-Data (SIMD) instructions (e.g. SSE, AVX2, or ARM NEON) and by applying optimizations on the assembly level.

There are several projects implementing software libraries for general purpose processors that include the Kyber and NewHope schemes: PQClean[1], liboqs[2], libpqcrypto[3], and SUPERCOP[4]. On constrained devices, the authors in [ABCG20] presented an optimized software implementation of Kyber and NewHope for an ARM Cortex-M4 platform with a small memory footprint. They reached a speedup of ca. 10% for both Kyber and NewHope implementation compared to the ARM Cortex-M4 implementations in [BKS19]

[1] https://github.com/PQClean/PQClean
[2] https://openquantumsafe.org/#liboqs
[3] https://libpqcrypto.org/
[4] https://bench.cr.yp.to/supercop.html

and [KRSS]. They achieved this speedup by optimizing finite field arithmetic and NTT operations. The results were integrated to the pqm4[5] project, which is a post-quantum crypto library for the ARM Cortex-M4. We refer to this work in more detail in Section 3 and Section 5.

In [AHH+19] Albrecht et al. show an adapted variant of Kyber and NewHope on a commercially available smartcard (Infineon SLE 78). They utilize the existing RSA/ECC co-processor to accelerate finite field operations and the AES and SHA-256 co-processor to speed up the pseudo random number generation and the hash computation of their Kyber and NewHope adaption.

Other software-based works on constrained devices in the area of lattice-based schemes are an implementation of NTRU [BSJ15] on a smart card microprocessor, of BLISS on ARM Cortex-M4F microcontroller [OPG14] and on Atmel ATxmega128 [LPO+17]. Furthermore, there are implementations of Ring-LWE schemes on Atmel ATxmega128 [LPO+17], on Texas Instruments MSP430 as well as on ARM Cortex-A9 using the NEON engine to vectorize the NTT [LAKS18], and on ARM Cortex-M4F [dCRVV15]. For the scheme NewHope, an implementation on ARM Cortex-M0 and M4 is presented in [AJS16]. Also for ARM Cortex-M4 the authors of [KMRV18] present an implementation of the module lattice-based scheme Saber.

To the best of our knowledge there are no dedicated hardware implementations for Kyber or other module lattices-based schemes. However, there are hardware implementations for standard and ideal lattices that share the common requirement for polynomial multiplication and finite field arithmetic with our work: In [GFS+12], the first hardware implementation of Ring-LWE-based PKE was presented instantiating a high-throughput NTT with the cost of high area consumption. The smallest Ring-LWE implementation was proposed in [PG14] and a high-speed implementation using DSPs on the FPGA in [PG13]. Also the NewHope implementation presented in [OG17] utilizes DSPs for the NTT. A performance improvement is further reached by the authors of [KLC+17] because they are using four butterfly units for the NTT.

For the RISC-V there already exist a few software-hardware co-designs for PQC schemes: In [FSM+19] the authors present an implementation of the lattice-based scheme NewHope-CPA-KEM. They are using the RISC-V processor-core variant from the Pulpino distribution (RI5CY, RV32I ISA with RV32M multiplier, four stage in-order pipeline) and accelerate the NTT and the hash operations with distinct co-processors. The co-processors are connected to the RISC-V core by memory mapped IO via the AHB data bus. Furthermore, the signature scheme XMSS is implemented on RISC-V with a memory mapped acceleration core by the authors of [WJW+19]. They are using the Murax SoC setup of the VexRiscv ecosystem[6]. Finally, in [KZDN18] the authors presented a realization of a memory mapped and formally verified AES co-processor for RISC-V. Outside of the context of RISC-V, a small instruction set extension is proposed in [GKP04], which introduces a few specialized multiply-accumulate instructions. The goal was to accelerate the reduction operation of the subfield in the context of elliptic curves with optimal extension fields.

In July 2018, the RISC-V Foundation announced a security standing committee that, among other topics, intends to extend the RISC-V ISA with instructions to accelerate cryptographic operations[7]. In June 2019, the project XCrypto[8] was presented at the RISC-V Workshop Zurich [MPP19]. The authors define an extension of the RISC-V instruction set with the focus on classical cryptography. There does not yet seem to be comparable work regarding the requirements for post-quantum schemes.

There are several cryptographic instruction set extensions for modern CPUs, e.g., the

[5] https://github.com/mupq/pqm4

[6] https://github.com/SpinalHDL/VexRiscv

[7] https://riscv.org/2018/07/risc-v-foundation-announces-security-standing-committee-calls-industry-to-join-in-efforts/

[8] https://github.com/scarv/xcrypto

AES-NI extension and the Intel SHA Extensions for x86 processors. To the best of our knowledge, there is no previous work providing instruction set extensions for finite field arithmetic or other operations required for lattice-based cryptography.

Our contribution. This paper offers two contributions: First, we present optimized RISC-V implementations of the polynomial arithmetic used by the Kyber and NewHope schemes. These implementations target the standardized RISC-V ISA, adapting recent developments of optimizations for the ARM instruction set to the features of the RISC-V instruction set to achieve fast implementations with optimal usage of the processor pipeline. We integrate the results into implementations of the Kyber and NewHope schemes and evaluate their performance. The second contribution is a custom instruction set extension for finite field arithmetic for small fields. We introduce four new instructions and create a prototype implementation of an extended RISC-V processor for two FPGA platforms. We evaluate Kyber and NewHope implementations that utilize these custom instructions and compare the results to our first contribution. The result serves as a suitable alternative to larger accelerators, especially in size constrained applications.

Structure. We provide information about the lattice-based PQC schemes NewHope and Kyber as well as the RISC-V ISA and the VexRiscv implementation in Section 2. In Section 3 we describe how we port and optimize NewHope and Kyber to the official RISC-V ISA in order to obtain a base line for our performance measurements. Section 4 gives details about our ISA extensions and the implementation and the integration of the extensions into VexRiscv. Finally, we provide an evaluation of our work in Section 5.

We will make our implementation publicly available under an open source license.

2 Background

In this section, we introduce the lattice-based PQC schemes Kyber and NewHope as well as the RISC-V ISA and the RISC-V implementation VexRiscv that we use as base for our implementation and experiments.

2.1 Lattice-based Cryptography

The construction of lattice-based cryptography is based on the assumed hardness of lattice problems. The Shortest Vector Problem (SVP) is the most basic example of such problems. The goal of the SVP is to find the shortest nonzero vector in a lattice represented by an arbitrary basis. The first lattice-based cryptographic constructions were proposed by Ajtai in 1996 [Ajt96] using the Short Integer Solution (SIS) problem. The goal of the SIS problem is to find a small solution for a linear system over a finite field, which Ajtai proved in the average case to be as hard as the SVP in the worst case.

In 2005, Regev introduced a lattice-based public-key scheme [Reg05] that relies on the learning with error problem (LWE), which Regev showed to be as hard as several worst-case lattice problems. The goal in LWE is to find an n-based linear function over a finite field ring from a given sample where some of the function samples may be incorrect. For example, let $(\mathbf{x_i}, y_i)$ be a given sample where each $\mathbf{x_i} \in {\mathbb{Z}_q}^n$ is a vector of n integers modulo q, and $y_i \in \mathbb{Z}_q$ is an integer modulo q. The goal is to find the linear function $f : {\mathbb{Z}_q}^n \longmapsto \mathbb{Z}_q$ where $y_i = f(\mathbf{x_i})$ for any given sample. Solving this problem is not hard, since in this case one needs only n different samples to define f using linear algebra. However, if some of the samples are erroneous, the LWE problem becomes a hard problem requiring many samples for a solution.

There are two variants of the LWE problem that are as hard as the original problem. Both variants are used in some of the NIST PQC submissions. The first variant is the

Table 1: NewHope parameter sets (see [AAB+19]). The failure probability is given as δ.

	n	q	η	δ	NIST level
NewHope512	512	12289	8	2^{-213}	I
NewHope1024	1024	12289	8	2^{-216}	V

learning with error over rings (Ring-LWE) problem, which uses polynomial rings over finite fields as domain for the LWE. It was proposed by Lyubashovsky, Peikert, and Regev in 2010 [LPR10]. Basically, the secrets and errors are not integer vectors but polynomials coming from a polynomial ring. The other variation of LWE is the learning with error on modules (Module-LWE) problem, introduced, e.g., by Langlois and Stelé in 2012 [LS15]. Module-LWE is based on the Ring-LWE but replaces ring elements with module elements over the same ring. While Ring-LWE uses a polynomial ring, Module-LWE uses a matrix of these polynomials on the same ring. The secrets and errors turn into vectors of polynomials. Comparing Ring-LWE and Module-LWE at an implementation level, using modules makes it possible to choose the dimensions of the module for a given ring, which gives more freedom to balance security and efficiency.

Although NewHope and Kyber use different polynomial rings, their structure is the same, $\mathbb{Z}_q[X]/(X^n+1)$, denoted as $\mathcal{R}_q$. Moreover, both schemes use a centered binomial distributions (B_η) for generating noise.

2.1.1 NewHope

NewHope is a key-encapsulation mechanism based on the Ring-LWE problem. It has two variants named NewHope-CPA-KEM and NewHope-CCA-KEM for semantic security under adaptive chosen plaintext (CPA) and adaptive chosen ciphertext (CCA) attacks respectively. The authors use an CPA-secure public-key encryption (CPA-PKE) scheme inside both versions with a simple transformation from CPA to CCA security using the Fujisaki Okamoto (FO) transformation [FO99].

The NewHope submission to the NIST standardization process provides two security levels [AAB+19]. While the degree of polynomial changes ($n = 512$ or $n = 1024$), other parameters, e.g., the modulus ($q = 12289$) and the parameter of the noise distribution ($\eta = 8$), remain same for both security levels. Table 1 gives an overview on the proposed parameters and their failure probabilities.

NewHope-CPA-PKE. For *key generation*, a random bit string $seed_a$ is selected and a uniformly random polynomial $a \in \mathcal{R}_q$, is generated using $seed_a$. Then, coefficients of $s, e \in \mathcal{R}_q$ are sampled from a centered binomial distribution with parameter $\eta = 8$. After all the polynomials have been generated, the public key is computed as $b = as + e$ and packed together with $seed_a$, while the secret key contains the polynomial s.

For the *encryption* of a message μ, a secret $s' \in \mathcal{R}_q$ and an error $e'' \in \mathcal{R}_q$ are sampled from the centered binomial distribution. Then, the message is encrypted by computing $v = bs' + e'' + \text{Encode}(\mu)$. In addition, a corresponding R-LWE sample is generated using the same a and s' with a freshly generated $e' \in \mathcal{R}_q$ as $u = as' + e'$. Finally, u and v are packed as ciphertext.

For *decryption*, u and v are decoded and the message μ is deciphered using the secret key s by computing $\mu = \text{Decode}(v - us)$. (The functions Encode() and Decode() map between the message space and $\mathcal{R}_q$.)

Key encapsulation. The *KEM key generation* performs exactly same computations as NewHope-CPA-PKE key generation except for the packing of the secret key: While for NewHope-CPA-KEM the secret key only contains the polynomial s in a packed form, NewHope-CCA-KEM requires the public key to be added to secret key to perform additional checks that are required by the FO transform.

The *encapsulation* algorithm selects a random message μ and uses the CPA-PKE encryption algorithm to encrypt it.

The *decapsulation* algorithm uses the CPA-PKE decryption to obtain μ. The CPA secure version of the scheme directly uses a hash of μ as shared key, while the CCA secure version re-computes the entire encryption process (using μ instead of a randomly selected message) to ensure u and v have been generated using the encryption of the same message. This additional check is called FO transform. The decapsulation returns a random shared secret if the check fails and a hash of μ otherwise.

NewHope utilizes NTT as a part of the scheme. This means that is a generated in the NTT domain by default. Both key and ciphertext (b, s, u) are packed in the NTT domain except for the polynomial v with the added message. Keeping objects in the NTT domain allows to use less inverse NTT operations and therefore to increase the efficiency.

2.1.2 CRYSTALS-Kyber

Kyber [ABD+19] is a KEM based on the Module-LWE problem. At its core, Kyber is using an IND-CPA secure public key encryption scheme (Kyber.CPAPKE) that is converted to a IND-CCA2 secure KEM (Kyber.CCAKEM) using an adapted FO transform. Kyber is part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS)[9] together with its sibling signature scheme Dilithium [DKL+19]. Both CRYSTALS schemes have been submitted to the NIST standardization process. For the following brief and simplified description of Kyber.CPAPKE and Kyber.CCAKEM (from the round 2 version), we closely follow the notation of the Kyber specification [ABD+19]; please refer to [ABD+19] for further details. Vectors are written as bold lower-case letters, matrices as bold upper-case letters.

Kyber.CPAPKE. The underlying public-key encryption scheme of Kyber has several integer parameters: $n = 256$, $q = 3329$, $\eta = 2$, and k, d_u, and d_v with integer values depending on the security level. Parameter $k \in \{2, 3, 4\}$ is the main parameter for controlling the security level by setting the lattice dimension nk; d_u, and d_v are chosen accordingly for balancing security, ciphertext size, and failure probability.

For *key generation*, a matrix $\mathbf{A} \in \mathcal{R}_q^{k\times k}$ is generated randomly. The vectors $\mathbf{s}, \mathbf{e} \in \mathcal{R}_q^k$ are sampled from distribution B_η using a pseudo-random function (PRF). The vector $\mathbf{t} \in \mathcal{R}_q^k$ is computed as $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. The public key pk is an encoding of $\mathbf{t}$ and the secret key sk is an encoding of $\mathbf{s}$.

For *encryption* of a message $m \in \mathcal{R}_q$, first $\mathbf{t}$ is decoded from pk. Then, a matrix $\mathbf{A} \in \mathcal{R}_q^{k\times k}$ is generated. Vectors $\mathbf{r}, \mathbf{e_1} \in \mathcal{R}_q^k$ and $e_2 \in \mathcal{R}_q$ are sampled from B_η using the PRF. Vector $\mathbf{u} \in \mathcal{R}_q^n$ is computed as $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e_1}$ and $v \in \mathcal{R}_q$ is computed as $v = \mathbf{t}^T\mathbf{r} + e_2 + m$. The ciphertext c is a concatenation of encodings of $\mathbf{u}$ and v.

For *decryption* of a ciphertext c, c is decoded back to $\mathbf{u}$ and v and the secret key sk is decoded to $\mathbf{s}$. Then the plaintext message is simply obtained by computing $m = v - \mathbf{s}^T\mathbf{u}$.

One important optimization is built into the Kyber scheme: Vector-vector and matrix-vector multiplication can efficiently be performed in the NTT domain. This allows the Kyber team to exploit trade-offs between ciphertext size (using compression) and computing time. In order to avoid conversions into and out of NTT domain, the Kyber specification requires the public key to be stored and transmitted in the NTT domain. Furthermore, performing operations in the NTT domain allows the matrix $\mathbf{A}$ to be directly sampled in the NTT domain, which gives further speedups.

[9] https://pq-crystals.org/

Table 2: Kyber parameter sets (see [ABD+19]). The failure probability is given as δ.

	n	k	q	η	(d_u, d_v)	δ	NIST level
Kyber512	256	2	3329	2	$(10, 3)$	2^{-178}	I
Kyber768	256	3	3329	2	$(10, 4)$	2^{-164}	III
Kyber1024	256	4	3329	2	$(10, 5)$	2^{-174}	V

Kyber.CCAKEM. The public-key encryption scheme Kyber.CPAPKE is transformed into a KEM scheme using an FO transform. This requires several hash operations, e.g., for computing a message digest, and a key-derivation function (KDF). The hash functions are instantiated with SHA-3 or SHA-2 for a "90s" variant and the PRF and KDF with SHAKE-256. The cost of these symmetric operations typically exceeds the cost of finite field and polynomial arithmetic in Kyber.CPAPKE.

There are three parameter sets Kyber512, Kyber768, and Kyber1024 of Kyber that match to NIST levels I, III, and V and vary mainly in lattice dimension and compression parameters. Table 2 gives an overview over the parameter sets.

2.2 RISC-V

The RISC-V project was started in 2010 by the University of California, Berkley. Since 2015 it is organized in the RISC-V Foundation[10] with more than 275 memberships from academia and industry. The goal for the RISC-V project is to provide a free and open ISA suitable for processor designs (in contrast to simulation or binary translation) without dictating a specific micro-architecture style or implementation technology. The ISA itself is designed to satisfy the reduced instruction set computing (RISC) principles and is a small base integer instruction set with several sets of modular extensions (e.g., for integer multiplication, floating point operations, etc.) as well as designated space for future or custom extensions. There are several implementations of the RISC-V ISA, for example the Rocket Chip[11], the PicoRV32[12], and the VexRiscv[13]. In the following, we provide a short introduction into several implementations and a rationale for our choice of a specific implementation as basis for the work in this paper.

Extendable RISC-V implementations. One of the earliest implementations of the RISC-V ISA is the Rocket Chip with UC Berkeley as one of the main contributors. It is not a monolithic design but rather a system-on-chip (SoC) generator written in the hardware construction language Chisel[14]. The Rocket Chip design is also the basis of the SiFive Freedom E300 chip[15]. Chips generated with the Rocket Chip generator implement the RISC-V ISA including some standard extensions with a five-stage pipeline. The designs can be extended via a simple co-processor interface called "Rocket Custom Coprocessor" (RoCC). This interface can be used to implement custom instructions using one of the unused opcode spaces of the RISC-V ISA. While this interface is easy to use for development, it is also fairly inflexible since it does not allow the extension of the existing pipeline stages for creating multi-stage instructions. The much smaller PicoRV32 implementation follows a similar approach for implementing extensions. The core itself is tailored to very small designs and is also fully parametrizable. For instruction set extensions, the core offers the

[10] https://riscv.org/
[11] https://github.com/freechipsproject/rocket-chip
[12] https://github.com/cliffordwolf/picorv32
[13] https://github.com/SpinalHDL/VexRiscv
[14] https://chisel.eecs.berkeley.edu/
[15] https://github.com/sifive/freedom

"Pico Co-Processor Interface" (PCPI), following a similar approach as the Rocket Chip generator.

VexRiscv. The VexRiscv project aims at providing a highly modular implementation of the RISC-V ISA. Instead of using a fixed extension interface for co-processor cores, the VexRiscv core itself is extendable by design. The design is based on a flexible pipeline that is extended by plugins. VexRiscv is written in the SpinalHDL hardware description language[16], which is a domain specific language for designing digital circuits. SpinalHDL is built on top the Scala programming language. Hardware is described using a software oriented approach, however, *without* using high-level synthesis. The fundamental building blocks of the VexRiscv core are the pipeline stages defined by registers between the stages called *stageables*. During generation of the hardware, plugins are adding processing logic to stages to processes input stageables and insert output stageables. VexRiscv provides a RISC-V CPU implemented with five pipeline stages: (1) fetch, (2) decode, (3) execute, (4) memory, and (5) writeback. All functionality of the CPU is implemented using plugins as described above. The following describes some of the plugins, which make up the basic functionality of the RISC-V implementation.

The fetch stage is primarily built up by the `IBusSimplePlugin`, which has a simple memory interface from which it will fetch instructions starting at a configurable reset vector. This plugin also features an interface for other plugins to access the program counter, e.g., to implement jump instructions. The instructions fetched from memory are passed to the decode stage, which is primarily implemented by the `DecoderSimplePlugin`. The decoder itself is not tailored to RISC-V but built as a simple detector for bit patterns in a given bit string, i.e., the instruction. It offers a simple interface for other plugins to describe patterns that are to be detected and associated actions, i.e., values that the decoder needs to insert into specific stageables. For the most part, this is used to inject flags that control the behavior of the logic defined by each plugin. The register file is implemented as a plugin as well and injects the register values into pipeline stageables. If a flag in the pipeline indicates that an instruction produces a result, the register-file plugin writes the value to the register file in the final writeback stage. The `HazardSimplePlugin` tracks the usage of registers and delays an instruction in the decoding stage whenever a dependency to a preceding unretired instruction exists. If the result of an instruction is available in earlier stages, this plugin can also be configured to bypass the result directly to the depending instruction instead of delaying it. Register values are rarely used directly by plugins; instead the values are processed by the `SrcPlugin`. Depending on flags from the decoder, this plugin detects instruction-variants with immediate values and injects these values instead of register values to the following stages as source operands.

The described plugins are used to build up the core framework of the processor as shown in Figure 1. Most other plugins then build on top of this to implement the basic integer ALU, shifters, or multipliers. Note that a plugin is not necessarily restricted to a single stage. For example, a multiplication plugin can spread its logic over multiple stages to avoid long unregistered signal paths. A branching plugin interprets branch instructions and indicates jumps to the `IBusSimplePlugin`. The `DBusSimplePlugin` implements the load- and store instructions and offers a memory interface similar to the instruction bus. Further plugins implement the special machine registers, such as cycle counters or trap vectors, and a debug plugin offers a JTAG interface to control the core.

Due to this flexible architecture, we chose the VexRiscv core as a basis for our experiments. In particular, we are making use of its ability to implement instruction set extensions that utilize the entire pipeline instead of providing a restricted coprocessor interface.

[16] https://github.com/SpinalHDL/SpinalHDL

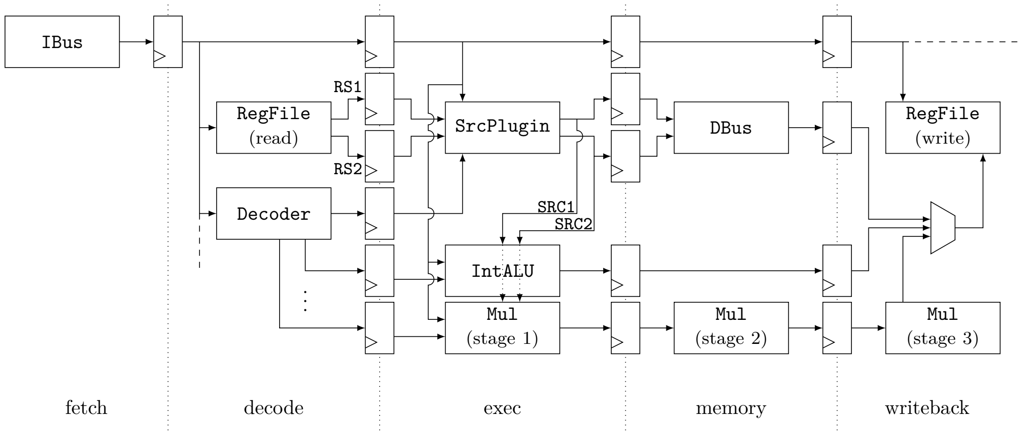

Figure 1: The simplified architecture of the VexRiscv core.

3 Kyber and NewHope on RISC-V

This section covers the implementation of the polynomial arithmetic used by NewHope and Kyber, using the standardized RISC-V ISA. To the best of our knowledge, there are no previous optimization attempts of these schemes for the RISC-V platform. Accordingly, we follow the approach for optimization in the reference implementations as well as previous publications for platforms similar to RISC-V like the ARM Cortex-M4. The optimization target is the RISC-V integer base instruction set with the standardized multiplication extensions, designated `rv32im` by the GCC compiler. While the multiplication extensions feature dedicated division and remainder instructions, we decided not to use them based on the assumption that smaller embedded devices, e.g., smartcards, would not feature a single (or few) cycle logic for division due to its prohibitive cost in terms of circuit size. Similarly, the ARM Cortex-M4 also does not feature a single cycle division instruction. In the following, we describe the implementation of the finite field used for the coefficients of the polynomials in Section 3.1. Section 3.2 follows with details on the NTT implementation and related operations on polynomials.

3.1 Finite Field Arithmetic

In the reference implementations of both Kyber and NewHope, the authors used two different reduction algorithms for multiplication and addition/subtraction to be able to make use of specific 16-bit operations, e.g., to perform 16 operations in parallel using AVX2 extensions or to use specific DSP instructions defined in some Cortex-M4 cores. Because the targeted RISC-V ISA does not support 16-bit arithmetic operations, we implement a 32-bit reduction algorithm. The following paragraphs describe our RISC-V-specific design choices.

Polynomial representation. Our representation of the polynomials and their coefficients follows the C reference implementation by the NewHope and Kyber authors as well as some of the improvements described in [BKS19]. The polynomials have 256, 512, or 1024 coefficients, each of which being an element in $\mathbb{F}_q$ with either a 14-bit ($q = 12289$) or a 12-bit ($q = 3329$) prime. The reference implementations uses a signed 16-bit integer representation, which avoids the need for handling underflows during the Montgomery reduction step. Without slight modification, this approach does not lend itself well to the RISC-V ISA, as the RISC-V does not feature a sign-extension operation as, e.g., the ARM

Algorithm 1: Barrett reduction on 32-bit inputs.

```
Input: Integer $a < 2^{32}$                          (register a0)
Input: Prime $q$                                     (register a1)
Input: Integer $\lfloor \frac{2^{32}}{q} \rfloor$    (register a2)
Output: reduced $a < 2^{14}$                         (register a0)
mulh t0, a0, a2 // $t \leftarrow a \cdot \lfloor \frac{2^{32}}{q} \rfloor$
mul t0, t0, a1 // $t \leftarrow t \cdot q$
sub a0, a0, t0 // $a \leftarrow a - q$
```

Algorithm 2: 4× interleaved Barrett reduction on 32-bit inputs.

```
Input: Integer $a_i < 2^{32}$, with $1 \leq i \leq 4$
                                       (registers a0 to a3)
Input: Prime $q$                                     (register a4)
Input: Integer $\lfloor \frac{2^{32}}{q} \rfloor$    (register a5)
Output: reduced $a < 2^{14}$                         (register a0)
mulh t0, a0, a5 // first batch
mulh t1, a1, a5 // second batch
mulh t2, a2, a5 // third batch
mulh t3, a3, a5 // forth batch
mul t0, t0, a4 // first batch
mul t1, t1, a4 // second batch
...
sub a0, a0, t0 // first batch
...
```

ISA with its `sxth` instruction.

The multiplication extension for RISC-V features an instruction that returns the high 32-bit of a 32 × 32-bit multiplication, which can be used to remove shifting operations during the modular reduction. Thus, we decided to implement a 32-bit Barrett reduction for a better fit to the target ISA, as shown in Algorithm 1. This optimization approach only requires three instructions, which is less than the approach for unsigned Montgomery reduction with four instructions as used in [AJS16]. This and similar other approaches used for the ARM instruction set cannot be used here, since RISC-V does not feature a fused multiply-and-add instruction. In contrast to a Montgomery reduction, we can perform modular reduction after both multiplication and addition/subtraction without the need for any base transformation of the coefficients.

Instruction interleaving. The processor pipeline of the VexRiscv features five pipeline stages. The instructions of the reduction may block this pipeline considerably, as each instruction depends on the result of its predecessor (see Algorithm 1). Therefore, each instruction blocks the following dependent one from progressing beyond the decoding stage until the instruction is retired. The processor can bypass the later stages for some instructions, which shortens these stalls. For example, the result of the addition and subtraction instructions (`sub` in Algorithm 1) become ready during the execute stage (stage three of five) and can be bypassed directly to the following instruction in the decoding stage. This however is not possible for multiplication (`mul` and `mulh` in Algorithm 1), as the result of this instruction is only readied during the final fifth stage.

Since the RISC-V features a large number of registers, this problem can be alleviated considerably by reducing multiple results in an interleaved manner. Algorithm 2 shows (in abbreviated form) how we interleave the execution of four independent reductions to avoid pipeline stalls due to interdependence of consecutive instructions. While this optimization likely is not as effective for RISC-V CPUs with shorter pipelines, it does not harm the performance, as the number of instructions remains the same as with four consecutive reductions. A more than four-fold interleave is not necessary for the VexRiscv, as the fifth instruction is only just fetched, as the first makes its result ready. We also apply the same optimization to the polynomial multiplication, addition, and subtraction.

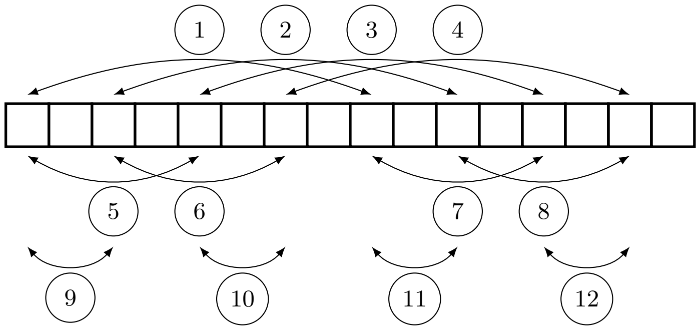

Figure 2: The order of butterfly operations for an NTT with three merged layers.

3.2 Number Theoretic Transform

At the core of NewHope and Kyber is the NTT, which is used to speed up polynomial multiplication. Previous results from [GOPS13], [AJS16], [BKS19], and [ABCG20] reduce the number of memory loads and stores by merging several layers of the NTT. The NTT transformation requires powers of the primitive n-th root of unity during butterfly operations, which are called twiddle factors. Because the implementation might require the twiddle factors to be in some special order, these factors are often precomputed and kept in memory.

Unmerged implementations load one pair of coefficients from memory, load the appropriate twiddle factor from memory, perform the butterfly operation, and store the result. When merging layers, as illustrated in Figure 2, the number of loaded pairs and butterfly operations per layer is doubled, i.e., for n layers, 2^n coefficients and $2^n - 1$ twiddle factors are loaded and $2^{n-1} \cdot n$ butterfly operations can be performed before storing the results.

The RISC-V ISA features a sufficient amount of registers to merge up to three layers, i.e., eight coefficients are processed with twelve butterfly operations before the next set is loaded. Each four of the twelve butterfly operations are then interleaved to further avoid pipeline stalls as described above. The NTT used in Kyber features seven layers. Thus, we can process the first three layers in one block and the next three in eight sequential blocks. The last layer cannot be merged and is processed with four interleaved butterfly operations. Although there are not enough registers to merge four layers, the last layer can be merged with the previous chunk by loading only half of the coefficients. This idea was used in [AJS16] to combine the 10th layer of NewHope1024 with merged 7th, 8th, and 9th layers.

While this technique can be helpful in NewHope, Kyber has a slightly different structure that makes it hard to get improvements using this technique. Kyber implementations are terminating the NTT computation at the 7th layer, thus the last layer is performed on degree-two polynomials. This means that at the merged 4th, 5th, and 6th layers should be performed on each coefficients of the small polynomials, i.e., performing a scalar multiplication, addition, and subtraction of degree-two polynomials, while the above technique loads new values into the registers that were used for twiddle factors. Thus, the technique can save 2^n loads per merged butterfly operation, while bringing the need of loading $2^n - 1$ twiddle factors to use them for the other coefficient of degree-two polynomials. Thus we decided to use this technique only for our NewHope implementations.

Our implementation employs a moderate amount of code unrolling, e.g., to enable interleaving. Due to the high number of registers, unrolling is not necessary to free registers otherwise used for counters. Accordingly, we are using a looped approach for the iteration through the merged blocks to reduce the code-size. The inner twelve butterfly operations are, however, fully unrolled and reused by each of the blocks.

3.3 On-The-Fly Computation of Twiddle Factors

In this section, we describe a technique to reduce memory consumption during the NTT computation. This version of the algorithm is often called `Iterative` NTT [LRCS01]. The order of the twiddle factors used during the NTT varies depending on the choice of the butterfly operation, e.g., Gentelemen-Sande [GS66] or Cooley-Tukey [CT65], and the order of the input polynomial such as bit-reversed order or normal order, and also the order in which the coefficients are processed. Our first design decision was not to use the bit-reversal process with the goal to reduce memory instructions. Thus, we implemented a forward NTT for normal order to bit-reversed order as well as an inverse NTT for bit-reversed order to normal order.

Although on-the-fly computation of twiddle factors reduces the memory usage, multiplication with reduction is more expensive than a single memory operation on most platforms. Therefore, this version of the algorithm is often used in FPGA implementations, where implementers can design efficient modular arithmetic circuits [RVM+14]. Software implementations usually opt for precomputation of all twiddle factors [BKS19], or at least some part of them [AJS16] to reduce the running time.

While the RISC-V ISA only contains general purpose computing instructions, the instruction code space also reserves opcodes for custom instructions. Hence both FPGA implementation tricks and the tricks used in implementations on constrained microcontrollers can be used on the RISC-V by providing instruction set extensions for modular arithmetic. In Section 4, we describe a single cycle multiplication operation in $\mathbb{Z}_q$ that makes the on-the-fly computation of twiddle factors possible without any performance penalty. Furthermore, when interleaved, this multiplication does not stall the pipeline of the processor, unlike a memory load instruction.

Changing the order of twiddle factors. To implement `Iterative` NTT, one needs to arrange the order of the twiddle factors in a way that can be computed on-the-fly. This can be done by changing the order of the input polynomial and the distance between the coefficients that are processed in the butterfly operation. Two different implementations for three layer butterfly operations are given in Figure 3. Note that two implementations compute same result, but the order of the twiddle factors that are being processed in each layer is different.

In Figure 3, w^{i_r}, w^{j_r} and w^{k_r}, where $r = 0, 1, 2, 3$, are some arbitrary powers of n-th root of unity for a selected chunk of the NTT computation. Both implementations use the Gentlemen-Sande butterfly. In case that the order of the input polynomial should not be changed, the same result can be achieved by changing the butterfly operation used, i.e., Cooley-Tukey butterfly can be used instead of Gentlemen-Sande or vice-versa.

Early termination of NTT computation. For the second round of the NIST process, Kyber proposed a new and smaller q. However, to be able to perform NTT for polynomial multiplication, they needed to use a technique by Moenck [Moe76] that will be called "early termination of NTT computation" in this document. For the second round parameters of Kyber, the authors performed NTT for only 7 layers, thus their polynomials are written as $n' = 128$ polynomials in $\mathbb{Z}_q/(X^2 - w^i)$, where w^i is a i-th power of the n'-th root of unity. The order of these powers is same with the order of the output of the NTT. Thus if we use a NTT variant that takes polynomials in normal order and returns them in the bit reversed order, we need to perform degree-2 polynomial arithmetic with some power of w that is in the bit-reversed order. As a result, we still need to precompute some values. However, in bit-reversed order, each quadruple has powers in the form of $w^i, w^{i+\frac{n'}{2}}, w^{i+\frac{n'}{4}}, w^{i+\frac{3n'}{4}}$, where $i = 0, .., \frac{n'}{4}$ in bitreversed order, and this can be written as $w^i, -w^i, w^{i+\frac{n'}{4}}, -w^{i+\frac{n'}{4}}$. Thus these four powers of w can be computed using only w^i and $w^{\frac{n'}{4}}$ and we can reduce

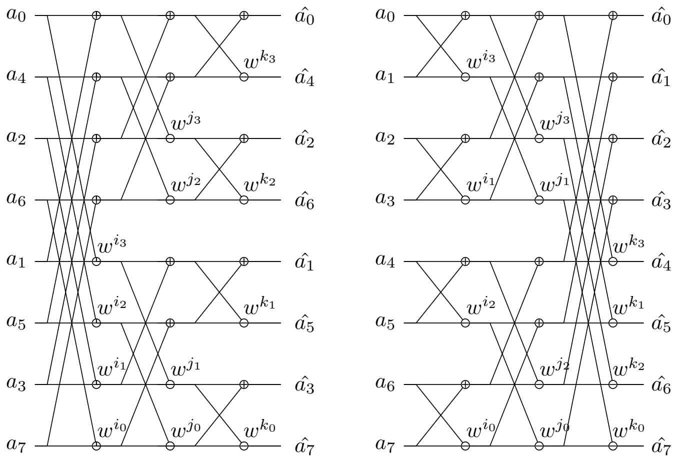

Figure 3: Butterfly operation with bit-reversed twiddle factors.

the size of the precomputed table by only using first $\frac{n'}{4}$ powers of w in bit-reversed order and multiply them with $w^{\frac{n'}{4}}$ on-the-fly when needed. Note that if the starting polynomial is $\mathbb{Z}_q/(X^n + 1)$, NTT requires to be able to represent it as $\mathbb{Z}_q/(X^n - \gamma^{n'})$ where γ in n'-th root of -1. Thus, all precomputed values should be multiplied with γ to complete Chinese remainder representation.

4 Instruction Set Extension for Finite Fields

As introduced in Section 2.2, we chose the VexRiscv core as a baseline RISC-V implementation, due to its flexible design. In the following we describe the general architecture of the designed system, as well as the design of the instruction set extension for small finite fields.

4.1 General Architecture

In addition to the RISC-V implementation, the SpinalHDL and VexRiscv projects also provide plugins for buses, memory, and IO. These plugins can be freely combined as needed. The VexRiscv project provides some examples of stand-alone SoC instances. We used the most simple example, the Murax SoC, as baseline for our implementation. The core configuration is tailored towards small FPGAs, with many features deactivated. Our goal is to create a chip with some key features similar to the ARM Cortex-M4 (cf. [Yiu14]). Since the VexRiscv core features two more pipeline stages than the ARM Cortex-M4, instructions generally need more cycles to be retired. To ameliorate this disadvantage of the VexRiscv core, we enabled the hazard management plugin with the pipeline bypass feature in the configuration, which makes the results of many instructions available already in the execute stage.

Furthermore, we used two separate plugins for integer multiplication: a full multiplier, and a simpler multi-cycle multiplier. While the multiplier of the Cortex-M4 operates in a single cycle, the full VexRiscv multiplier is spread across the three pipeline stages exec, memory, and writeback. None of the stages in the full multiplier takes more than one cycle to complete, so several multiplication instructions can be handled in quick succession, as long as they do not have register interdependencies. The simplified multi-cycle integer multiplier remains in the memory stage for 32 cycles, blocking the pipeline until the result is ready. Which of the two multiplier plugins is used, is chosen when the hardware is generated. The VexRiscv division instruction plugin also requires multiple cycles to complete and stalls the pipeline, similar to the Cortex-M4. While we do not use the division instruction in any of our implementations, we included it to match the feature set of the Cortex-M4 in order to get a comparable area consumption. As final modification to the default core configuration, we enabled several machine state registers, e.g., the cycle and the instruction counter for later performance evaluations. The design also includes the JTAG debug core plugin by default, which in real-world scenarios may be left out.

The memory architecture of the Murax SoC combines the data and the instruction bus in order to use a simple single-master bus architecture. To mimic the bus architecture of a Cortex-M4 core, our modifications to the Murax SoC include a full multi-master bus matrix with both the instruction and data bus ports of the core as masters. The memory bus itself is a simple pipelined bus, i.e. using independent command and response buses. Our design also features two SRAM blocks instead of one as bus slaves. This is to accommodate the two masters, as well as to mimic the existence of a separate fast on-chip storage for code and an SRAM block for stack and data, as present in most Cortex-M4 chips. We linked all evaluated programs accordingly to map code into the lower, and stack and other data to the upper block. This avoids most pipeline stalls, which otherwise occur in case the instruction and the data bus compete for the same bus slave at the same time. Apart from memory, the data bus of the core also includes a peripheral bus slave component with a memory mapped UART controller for communication.

4.2 Instruction Set Extension

To examine the effectiveness of small instruction set extensions, for example in the scenario of a small specialized processor of a smartcard, TPM, or other HSMs, we target the underlying finite field of the Kyber and NewHope ciphers. However, this technique can also be applied to other cryptographic primitives that require finite fields arithmetic with elements that can be represented with fewer bits than a machine word. There are plenty such primitives in the PQC domain, e.g., other lattice-based schemes, code-based schemes, or multivariate schemes. The idea is to implement all arithmetic concerning a small finite field with a dedicated set of instructions, which perform the respective operation and the subsequent reduction, instead of using several standard instructions for the operation and the reduction. Even though implementing a reduction on a standard ISA only takes a couple of instructions, the time savings can accumulate. Furthermore, the code size of the implementation can be reduced, which may save valuable space in case of small embedded implementations. Depending on existing size constraints, such a custom finite field instruction may be a valuable alternative to much larger dedicated accelerators as used in, e.g., [OG17] and [WJW+19].

We decided to use the Barrett reduction algorithm as basis for our custom instruction, as shown in Algorithm 3. Compared to Montgomery reduction, this avoids the need to convert the elements to the Montgomery domain and back. This would either be necessary for each single instruction or the programmer would need to keep track of which values or results are in Montgomery domain. Barrett reduction is generally less efficient than Montgomery reduction in many scenarios. However, here it simplifies data handling, which makes a computational overhead worthwhile. Furthermore, depending on the choice of the prime, the low Hamming weight of the constants can be used to avoid large multipliers. We use the low Hamming weight of $q = 3329$ in Kyber (and $q = 12289$ of NewHope) to implement the required multiplication with q as four shifts and additions.

Figure 4 illustrates the integrated circuit of the custom instructions for addition, subtraction, and multiplication in $\mathbb{F}_q$. The calculation begins with the inputs from the source registers in the execute stage. Then the field operations, i.e. addition, subtraction, or multiplication, are performed and the desired operation is chosen according to two of the bits in the instruction word. The output of this stage is the unreduced result of

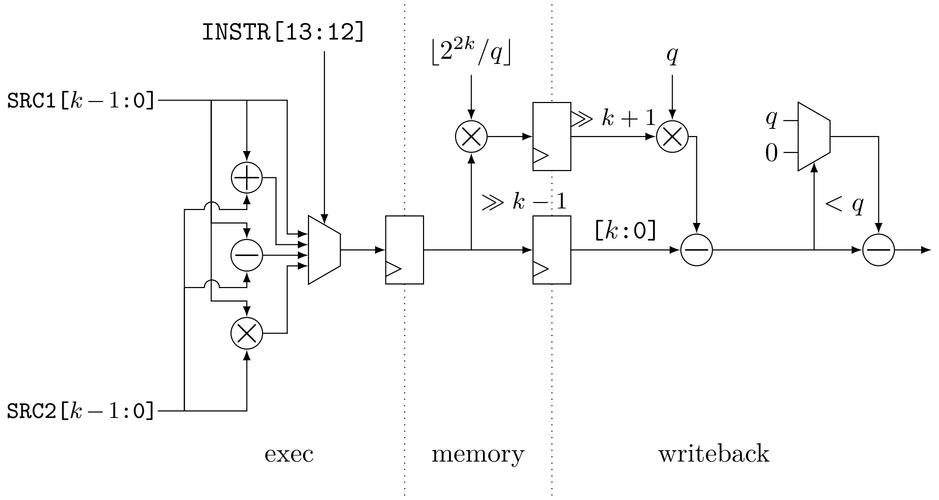

Figure 4: Circuit diagram for the custom instructions in $\mathbb{F}_q$ with $q < 2^k$.

Algorithm 3: Barrett reduction algorithm. [MVO96]

```
Input: Positive integers $x < 2^{2k}$, $2^{k-1} < q < 2^k$ and $\mu = \lfloor \frac{2^{2k}}{m} \rfloor$
Output: Reduced integer $r = x \pmod{q}$
$y_1 \leftarrow x \gg (k-1)$
$y_2 \leftarrow (y_1 \cdot \mu) \gg (k+1)$
$r \leftarrow x - y_2 \cdot q \pmod{2^k}$
if $r \geq q$ do: $r \leftarrow r - q$
```

the intended operation, i.e., the input x of Algorithm 3. In the memory stage, the upper bits of the unreduced result, i.e. value y_1 in Algorithm 3, is multiplied with the Barrett constant μ and passed on to the writeback stage. Effectively, the result calculated in the memory stage of the processor is y_2 of Algorithm 3. In the writeback stage, the calculated value y_2 is multiplied with the prime q. This multiple of the prime is subtracted from the unreduced result, which is then guaranteed to be smaller than $2q$. A subsequent conditional subtraction of q then fully reduces the value.

We implemented two variants of the custom instruction: a variant which operates with a set of primes fixed when the CPU is synthesized, as well as a flexible variant, capable of setting the prime at run time. The difference between those variants is simply the origin of the values of k, q, and $\lfloor 2^{2k}/q \rfloor$ as used in Figure 4. In the case of the fixed variant, these values are provided by a mux which selects the constants accordingly. The flexible variant instead sources these values from internal registers. These registers are then part of the CPU state and execution context. Overall three finite field operations are introduced by the plugin: finite field addition `ffadd`, finite field subtraction `ffsub`, and finite field multiplication `ffmul`. A fourth instruction `ffred` does not perform finite field arithmetic, but it simply reduces the value stored in the source register modulo q. Two further instructions `ffset` and `ffget` provide access to the internal state of the flexible variant and are not implemented by the fixed variant. Figure 5 shows the bit format of the six instructions. We use standard instruction formats, as specified by the RISC-V ISA. Bit 0 to 6 designate the custom extension, while bit 12 to 14 pick the finite field operation. The bit position for the registers are defined by the RISC-V ISA. The higher seven bits are used as an index by the fixed variant of the custom instruction to select the used prime.

Bit	31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
`ffadd`	`q_idx`	`rs2`	`rs1`	011_{bin}	`rd`	0001011_{bin}
`ffsub`	`q_idx`	`rs2`	`rs1`	001_{bin}	`rd`	0001011_{bin}
`ffmul`	`q_idx`	`rs2`	`rs1`	010_{bin}	`rd`	0001011_{bin}
`ffred`	`q_idx`	unused	`rs1`	011_{bin}	`rd`	0001011_{bin}
`ffset`	0000000_{bin}	`rs2`	`rs1`	100_{bin}	unused	0001011_{bin}
`ffget`	0000000_{bin}	unused	unused	100_{bin}	`rd`	0001011_{bin}

Figure 5: The instruction format for the custom finite field arithmetic instructions.

5 Evaluation

In this section we evaluate the performance of our optimized RISC-V software implementation presented in Section 3, as well as the performance of our implementation that utilizes the custom instructions presented in Section 4. Furthermore, the size and speed of the chip design on FPGA target platforms is discussed.

NTT and polynomial operations. The cycle count and number of issued instructions for the polynomial arithmetic is presented in Table 3. We evaluated the C reference implementation[17] by the Kyber authors (C-Ref.), our optimized software implementation (`rv32im`), as well as a variant using the custom instructions (`custom`). The targeted platform for C-Ref. and `rv32im` is our RISC-V core (see Section 4.1) with a full multiplier enabled. The `custom` implementation does not use the general purpose multiplier within the implementation of the polynomial arithmetic. Therefore, the multiplier plugin could be removed from the RISC-V CPU if not needed otherwise. To the best of our knowledge, no other optimized RISC-V software implementation of Kyber is available. Therefore, we added recent results from Alkim, Bilgin, Cenk, and Gerard [ABCG20] on an ARM Cortex-M4 microcontroller as reference.

The optimized software implementation requires 53% to 85% fewer cycles than the C reference implementation for all polynomial operations. In the optimized implementations, the average number of cycles per instruction ranges from 1.25 in the case of the polynomial multiplication down to 1.05 in the case of the inverse NTT suggesting that the utilization of the processor pipeline is at a near optimum. The use of the custom instructions reduces the number of cycles by another 20% to 33% compared to the `rv32im` version (up to 89% fewer cycles than the C reference).

A much more significant advantage of the custom instruction becomes apparent when the size of the code is considered. Table 3 shows the size of the code and related precomputed values for the polynomial arithmetic operations. Note, that the Kyber cipher always uses the same polynomial arithmetic for all variants. Due to the looped implementation, the C reference remains the smallest. For the Kyber ciphers, the impact of the custom instruction is a moderate 24% size decrease, as the precomputed values are fairly moderate in size (two 128x16 bit tables). The difference becomes much more noticeable for the NewHope ciphers, with a 46% and 61% decrease in size. Furthermore, the size does not increase as much when the degree of the polynomial is raised. Here, the size increase of the `custom` variant only stems from additional NTT layers, as the code is unrolled. In principle,

[17] `https://github.com/pq-crystals/kyber`, commit `46e283ab575ec92dfe82fb12229ae2d9d6246682`

Table 3: Number of cycles and executed instructions per cycle (in parenthesis) for a single polynomial operation of addition, subtraction, multiplication (element-wise in NTT domain), and `NTT` as well as overall code size in byte.

	`poly_`				
Implementation	`add` / `sub`	`basemul`	`ntt`	`invntt`	Size
		Kyber			
C-Ref.	3379 (0.69)	13634 (0.51)	34538 (0.55)	56040 (0.52)	1892 B
`rv32im`	1586 (0.82)	4050 (0.80)	8595 (0.88)	9427 (0.90)	3120 B
`custom`	1586 (0.82)	2395 (0.81)	6868 (0.82)	6367 (0.83)	2368 B
[ABCG20][a]	N/A	2325 (N/A)	6855 (N/A)	6983 (N/A)	N/A
		NewHope512			
C-Ref.	6712 (0.69)	19536 (0.53)	134322 (0.54)	137375 (0.56)	3284 B
`rv32im`	2774 (0.87)	4322 (0.92)	20548 (0.96)	21504 (0.96)	6048 B
`custom`	2774 (0.87)	2783 (0.87)	14787 (0.92)	14893 (0.92)	3264 B
[ABCG20][a]	N/A	3127 (N/A)	31217 (N/A)	23439 (N/A)	N/A
		NewHope1024			
C-Ref.	14391 (0.71)	40014 (0.54)	291354 (0.54)	306258 (0.56)	5352 B
`rv32im`	5462 (0.88)	8547 (0.92)	45252 (0.97)	47711 (0.96)	9568 B
`custom`	5462 (0.88)	5472 (0.88)	31295 (0.94)	31735 (0.92)	3664 B
[ABCG20][a]	N/A	6229 (N/A)	68131 (N/A)	51231 (N/A)	N/A

[a] Cortex-M4

a looped implementation should not grow at all. The size of the precomputed values used by the `rv32im` variant, however, grows linearly with the degree of the polynomial.

In comparison to the optimized Cortex-M4 implementation, the RISC-V implementation without custom instructions (`rv32im`) takes more cycles. Note that, when degree of the polynomial is high enough, RISC-V implementations become more efficient. The optimized ARM code uses (to some extent) small SIMD instructions that can handle two 16-bit halfwords at once. Furthermore, these 16-bit halfwords do not require the expensive sign extension operations, which are necessary for RISC-V, considerably shortening the number of instructions for Montgomery reduction. Although SIMD instructions lead the performance improvements, implementers should keep all inputs small enough to fit in 16-bit half words with using additional modular reduction procedures. The effect of these additional modular reductions makes the optimized Cortex-M4 implementation of `NTT` become slower for higher degree polynomials while the performance improvement on basemul operation remain. Our RISC-V implementation with custom instructions and code interleaving outperforms the Cortex-M4 in terms of cycle count.

Performance comparison. The significant speedups we obtain for the polynomial arithmetic are diminished when looking at the schemes as a whole. This is due to the time-dominating role of the hash functions used in Kyber and NewHope, e.g., of the Keccak permutation used as a PRF and KDF. This problem is aggravated since the only optimized implementation of the Keccak permutation for RISC-V uses a bit interleaved representation of the state [Sto19]. Therefore, it requires additional bit manipulation operations that would be very costly in RISC-V. To alleviate this, we evaluated the ciphers with a simple unrolled assembly implementation of the Keccak permutation. However, if the cycles spent within hash functions are subtracted from the measurements, the speedups translate reasonably well.

Table 4 shows the results for the cycles spent in total (including the hash function, marked as "-total") and cycles spent excluding the hash-function calls. The targeted platform for C-Ref. and `rv32im` is our RISC-V core with a full multiplier enabled. In the following we focus on the performance with the cycles spent in the hash function removed. The optimized RISC-V implementation (`rv32im`) then requires 58% to 62% fewer cycles than the C reference code (C-Ref.), with the larger parameter sets showing the slightly smaller performance gains.

The table also lists recent optimized results for the ARM Cortex-M4 processor ([ABCG20] (M4)). Compared to a C reference on the Cortex-M4 (C-Ref. (M4)), the performance gains are slightly less significant compared to our result for RISC-V. In this case, the optimized implementation requires 45% to 63% fewer cycles.

To evaluate the impact of the custom instructions, we targeted three different platforms with varying implementations of the general purpose multiplier (i.e., the standard RISC-V `mul` instruction): one without a general purpose multiplier (`custom-nomul`), and one with the full three-stage multiplier (`custom`). In the case of `custom-nomul`, the compiler was configured not to emit any multiplication instruction (apart from our custom instructions) but to implement general integer multiplications using a series of other assembler instructions. As the code for Kyber is mostly free of any multiplications (apart from the polynomial arithmetic), the impact of a missing multiplier is negligibly small. The remaining multiplications are mostly by the constant q during polynomial (de-)compression, or otherwise by powers of two, which a compiler usually implements as shifts.

Overall, the impact of our custom instructions is not as visible as before when looking only at polynomial arithmetic. However, the results for `custom-nomul` are still slightly faster than the `rv32im` implementation, except in case of the encapsulation. The version with the custom instruction including a full multiplier (`custom`) is consistently the fastest with 20% to 25% fewer cycles than the optimized RISC-V implementation (`rv32im`) (68% to 71% fewer cycles than the C reference).

Circuit resource utilization and speed. To evaluate the cost of our custom instructions, we determined the required resources for our entire extended Murax SoC in various configurations on two different FPGA platforms. We use the iCE40 UltraPlus FPGA by LatticeSemi[18] as a target platform in order to mimic a low power microcontroller. The FPGA features 5280 LUTs with four inputs, eight DSP multipliers and four 256Kx16-Bit SRAM blocks. We used the open-source toolchain of Project Icestorm[19] for synthesis. To mimic higher power microcontrollers, we also target the Xilinx Artix-35T platform, which is one of the reference platforms in the NIST standardization process. This FPGA features 33280 LUTs with six inputs, as well as 90 DSP multipliers and 50 SRAM blocks of 36 kbit each. For synthesis, we used the Xilinx Vivado 2019.1 toolchain. While the Artix-35T features more memory than the iCE40, we instantiate the same amount of memory to create two feature-identical FPGA RISC-V implementations.

For each of the FPGAs, we generated six different configurations: a reference core without a custom instruction (`rv32im`), a custom core for a single prime and once without and with a general purpose multiplier (`custom-nomul` and `custom`), a custom core for four different primes (`custom4`) and custom core with the flexible extension, again once without (`flex-nomul`) and with (`flex`) a multiplier. For each configuration we evaluated the resource usage, maximum frequency $F_{\mathtt{max}}$ as reported by the synthesis tools.

Table 5 and Table 6 each show the device utilization for the iCE40 UltraPlus and Xilinx Artix-35T FPGAs. The overhead of a custom instruction for single prime (here $q = 12289$) is small with a LUT size increase of about 6% in both FPGAs. Note that this variant only requires one additional DSP, as the instruction will share a DSP with the

[18] http://www.latticesemi.com/en/Products/DevelopmentBoardsAndKits/iCE40UltraPlusBreakoutBoard

[19] http://www.clifford.at/icestorm/

Table 4: Performance (in kilo cycles) of the cipher implementations for key generation (**K**), encapsulation (**E**), and decapsulation (**D**).

Cipher	Implementation	**K**-total	**K**	**E**-total	**E**	**D**-total	**D**
Kyber512	C-Ref.	928	307	1301	472	1345	676
	rv32im	738	118	1009	181	923	253
	custom	710	89	971	143	870	200
	C-Ref. (M4)	650	296	885	412	985	603
	[ABCG20] (M4)	456	110	587	124	544	169
Kyber1024	C-Ref.	2745	796	3319	1029	3356	1358
	rv32im	2274	325	2703	414	2535	537
	custom	2203	253	2619	330	2429	431
	C-Ref. (M4)	1895	782	2257	950	2409	1268
	[ABCG20] (M4)	1405	310	1606	322	1526	413
NewHope512	C-Ref.	1192	421	1863	648	1892	859
	rv32im	945	175	1477	262	1372	338
	custom	904	134	1424	209	1302	268
	C-Ref. (M4)	720	278	1134	514	1192	598
	[ABCG20] (M4)	579	145	859	172	807	218
NewHope1024	C-Ref.	2395	881	3697	1367	3816	1822
	rv32im	1862	349	2856	526	2676	682
	custom	1776	263	2742	412	2528	535
	C-Ref. (M4)	1460	592	2265	927	2411	1264
	[ABCG20] (M4)	1158	302	1675	352	1588	461

Table 5: iCE40 UltraPlus device utilization and speed for varying configurations.

Configuration	LUTs	RAM	SPRAM	DSP	$F_{\max}$
rv32im	4072	6	4	4	16.4 MHz
custom	4292	-"-	-"-	5	16.6 MHz
custom4	4366	-"-	-"-	6	15.8 MHz
custom-nomul	3300	-"-	-"-	2	16.0 MHz
flex	4584	-"-	-"-	6	15.9 MHz
flex-nomul	3588	-"-	-"-	3	16.1 MHz

Table 6: Xilinx Artix-35T device utilization and speed for varying configurations.

Configuration	LUTs	FFs	RAM	DSP	$F_{\max}$
rv32im	1738	1599	34	4	59.6 MHz
custom	1842	1634	-"-	5	59.2 MHz
custom4	1826	1624	-"-	7	57.6 MHz
custom-nomul	1496	1298	-"-	2	59.6 MHz
flex	1907	1658	-"-	7	59.4 MHz
flex-nomul	1593	1336	-"-	4	62.8 MHz

Table 7: Xilinx Artix-35T Time-Area performance of a DSP-less implementation for varying core configurations.

Configuration	LUTs	$F_{\max}$	Time	Time×Area	Relative
`rv32im`	2833	58.8 MHz	11.60 ms	32855	1.00
`custom`	3046	57.7 MHz	9.27 ms	28223	0.86
`custom4`	3255	57.2 MHz	9.36 ms	30466	0.93
`custom-nomul`	1749	55.3 MHz	10.65 ms	18634	0.57
`flex`	3524	52.4 MHz	10.21 ms	35989	1.10
`flex-nomul`	2316	55.5 MHz	10.62 ms	24595	0.75

general purpose multiplier, and another DSP can be avoided if the q has a low Hamming weight (synthesized as three shifts and additions). If more than one prime is supported (here $q = 251, 3329, 12289, 18433$), one DSP will still be shared, however the multiplication with q will use a DSP. Hence, the increase in LUTs is small (or in the case of the Artix-35T even negative), but additional DSPs will be used. The `flex` variant adds a further 5%, but no additional DSPs. The `custom-nomul` variant offers an attractive option, with a significant size decrease of 13.9 to 18.9% to the reference, without any detriment to the polynomial math performance. Even the flexible variant `flex-nomul` offers a significant decrease of 8.3 to 11.8%. We used the timing reporting tools of the FPGA toolchains to determine the maximum frequency. The maximum frequency remains mostly unaffected by the design changes, with an average 16.14 MHz for the iCE40 and 59.7 MHz for the Artix-35T.

Due to the presence of DSPs in the synthesized designs, we cannot simply determine a "time-area-product", as there is no reasonable conversion between LUTs and DSPs. To emulate a "time-area-product", we also synthesized the designs for the Artix-35T without DSPs, i.e. by forcing the toolchain to use LUTs to implement the multipliers, and evaluated the performance of the NewHope1024 decryption function. Table 7 shows the according time-area-product of the NewHope1024 cipher decapsulation function (again excluding the hash function). The implementations using the custom instructions are consistently faster, despite the slightly lower maximum frequency. Due to the low hardware overhead of the custom instruction, the time-area-product is accordingly lower, when put into relation to the standard reference core. Variants without a general purpose multiplier are especially attractive here, as the performance loss due to a missing multiplier is mostly mitigated by the custom instruction[20]. The exception here is the `flex` variant, which should, however, be considered an outlier, as synthesizing the core with DSPs does not result in lower frequency. Furthermore, the variants using the custom instruction would also use less memory, which is not part of the area estimation here.

Our results show that using instruction set extensions for finite field arithmetic gives a significant performance benefit and helps to reduce memory consumption not only of program code but also of data in situations where constants can be recomputed on-the-fly instead of storing them explicitly. This is particularly beneficial on architectures with small or slow memory. Even for an overall algorithm that performs only a small number of finite field operations alongside, e.g., hash operations, such extensions can be effective given their small area overhead.

[20] For the `*-nomul` variants, the rest of the code was compiled for the `rv32i` architecture.

References

[AAB+19] Erdem Alkim, Roberto Avanzi, Joppe Bos, Leo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. Newhope — submission to the NIST post-quantum project. Specification document (part of the submission package), 2019.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. Cryptology ePrint Archive, Report 2020/012, 2020. https://eprint.iacr.org/2020/012.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 2.0) — submission to round 2 of the NIST post-quantum project. Specification document (part of the submission package), 2019.

[AHH+19] Martin R. Albrecht, Christian Hanser, Andrea Höller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems — TCHES*, 2019(1):169–208, 2019.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. NewHope on ARM Cortex-M. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering — SPACE 2016*, volume 10076 of *LNCS*, pages 332–349. Springer, 2016.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In Gary L. Miller, editor, *ACM Symposium on the Theory of Computing*, pages 99–108. ACM, 1996.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi, editors, *Progress in Cryptology — AFRICACRYPT 2019*, volume 11627 of *LNCS*, pages 209–228. Springer, 2019.

[BSJ15] Ahmad Boorghany, Siavash Bayat Sarmadi, and Rasool Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *ACM Transactions on Embedded Computing Systems*, 14(3):42:1–42:25, April 2015.

[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.

[dCRVV15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of Ring-LWE encryption. In Wolfgang Nebel and David Atienza, editors, *Design, Automation & Test in Europe Conference & Exhibition – DATE 2015*, pages 339–344. ACM, 2015.

[DKL+19] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium — submission to round 2 of the NIST post-quantum project. Specification document (part of the submission package), 2019.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology — CRYPTO 1999*, volume 1666 of *LNCS*, pages 537–554. Springer, 1999.

[FSM+19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepúlveda. Towards reliable and secure post-quantum co-processors based on RISC-V. In *Design, Automation & Test in Europe — DATE 2019*, pages 1148–1153. IEEE, 2019.

[GFS+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes A. Buchmann, and Sorin A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 512–529. Springer, 2012.

[GKP04] Johann Groszschaedl, Sandeep S. Kumar, and Christof Paar. Architectural support for arithmetic in optimal extension fields. In *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.*, pages 111–124, Sep. 2004.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography — PQCrypto 2013*, volume 7932 of *LNCS*, pages 67–82. Springer, 2013.

[GS66] W. Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578. ACM, 1966.

[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, Report 2017/690, 2017. https://eprint.iacr.org/2017/690.

[KMRV18] Angshuman Karmakar, Jose M. Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems — TCHES*, 2018(3):243–266, 2018.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KZDN18] Joseph R. Kiniry, Daniel M. Zimmerman, Robert Dockins, and Rishiyur Nikhil. A formally verified cryptographic extension to a RISC-V processor. In *Computer Architecture Research with RISC-V – CARRV 2018*, 2018. https://carrv.github.io/2018/papers/CARRV_2018_paper_5.pdf.

[LAKS18] Zhe Liu, Reza Azarderakhsh, Howon Kim, and Hwajeong Seo. Efficient software implementation of Ring-LWE encryption on IoT processors. *IEEE Transactions on Computers*, pages 1–11, 2018.

[LPO+17] Zhe Liu, Thomas Pöppelmann, Tobias Oder, Hwajeong Seo, Sujoy Sinha Roy, Tim Güneysu, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers. *ACM Transactions on Embedded Computing Systems*, 16(4):117:1–117:24, 2017.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology — EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, 2010.

[LRCS01] Charles Eric Leiserson, Ronald L. Rivest, Thomas H. Cormen, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, MA, 2001.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designes, Codes and Cryptography*, 75(3):565–599, 2015. IACR Cryptology ePrint Archive, Report 2012/090.

[Moe76] Robert T. Moenck. Practical fast polynomial multiplication. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 136–148. ACM, 1976.

[MPP19] Ben Marshall, Daniel Page, and Thinh Pham. XCrypto: A general purpose cryptographic ISE for RISC-V. In *RISC-V Workshop Zurich 2019*, 2019. https://github.com/scarv/xcrypto/blob/master/doc/riscv-meetup-bristol-slides.pdf.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[NDR$^+$19] Hamid Nejatollahi, Nikil D. Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. Post-quantum lattice-based cryptography implementations: A survey. *ACM Computing Surveys*, 51(6):129:1–129:41, 2019.

[OG17] Tobias Oder and Tim Güneysu. Implementing the NewHope-simple key exchange on low-cost FPGAs. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology — LATINCRYPT 2017*, volume 11368 of *LNCS*, pages 128–142. Springer, 2017.

[OPG14] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In *Design Automation Conference – DAC 2014*, pages 110:1–110:6. ACM, 2014.

[PG13] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography – SAC 2013, Revised Selected Papers*, volume 8282 of *LNCS*, pages 68–85. Springer, 2013.

[PG14] Thomas Pöppelmann and Tim Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *IEEE International Symposium on Circuits and Systems – ISCAS 2014*, pages 2796–2799. IEEE, 2014.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *ACM Symposium on Theory of Computing*, pages 84–93. ACM, 2005.

[RVM$^+$14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731 of *LNCS*, pages 371–391. Springer, 2014.

[Sto19] Ko Stoffelen. Efficient cryptography on the RISC-V architecture. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology — LATINCRYPT 2019*, volume 11774 of *LNCS*, pages 323–340. Springer, 2019.

[WJW+19] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems — XMSS hardware accelerators for RISC-V. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography — SAC 2019*, volume 11959 of *LNCS*, pages 523–550. Springer, 2019.

[Yiu14] Joseph Yiu. *The Definitive Guide to ARM Cortex-M4 and ARM Cortex-M4 Processors.* Newnes, 3rd edition, 2014.