RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography

Tim Fritzmann[1], Georg Sigl[1] and Johanna Sepúlveda[2]

[1] Technical University Munich, Germany, `{tim.fritzmann,sigl}@tum.de`
[2] AIRBUS Defence and Space GmbH, Taufkirchen, Germany `johanna.sepulveda@airbus.com`

Abstract. Empowering electronic devices to support Post-Quantum Cryptography (PQC) is a challenging task. Compared with traditional cryptography, PQC introduces new mathematical elements and operations which are usually not easy to implement on standard CPU architectures. Especially for low cost and resource constraint devices, hardware acceleration is absolutely required. In addition, as the standardization process of PQC is still ongoing, a focus on maintaining crypto-agility is mandatory. To cope with such requirements, Hardware/Software Co-Design techniques have been recently used for developing complex and highly customized PQC solutions. However, while most of the previous works have developed loosely coupled PQC accelerators, the design of tightly coupled accelerators and Instruction Set Architecture (ISA) extensions for PQC have been barely explored. To this end, we present RISQ-V, an enhanced RISC-V architecture that integrates a set of powerful tightly coupled accelerators to speed up lattice-based PQC. RISQ-V efficiently reuses processor resources and reduces the amount of memory accesses. This significantly increases the performance while keeping the silicon area overhead low. We present three contributions. First, we propose a set of powerful hardware accelerators deeply integrated into the RISC-V pipeline. Second, we extended the RISC-V ISA with 28 new instructions to efficiently perform operations for lattice-based cryptography. Third, we implemented our RISQ-V in ASIC technology and on FPGA. We evaluated the performance of NewHope, Kyber, and Saber on RISQ-V. Compared to the pure software implementation on RISC-V, our Co-Design implementations show a speed up factor of up to 10.5 for NewHope, 9.6 for Kyber, and 2.7 for Saber. For the ASIC implementation, the energy consumption was reduced by factors of up to 8.8 for NewHope, 7.7 for Kyber, and 2.1 for Saber. The cell count of the CPU was increased by a factor of 1.6 compared to the original RISC-V design, which can be considered as a moderate increase for the achieved performance gain.

Keywords: Post-Quantum Cryptography · RISC-V · Instruction Set Extension · Lattice-based Cryptography

1 Introduction

Public-key cryptography (PKC) provides the basis for establishing secured communication channels between multiple parties. The security of the PKC in use today is mainly based on the hardness of two mathematical problems, the factorization of large numbers (e.g., RSA) and the calculation of discrete logarithms (e.g., ECC). However, both problems can be solved in polynomial time when a large-scale quantum computer is built. The foreseeable breakthrough of quantum computers therefore represents a risk for all communication systems. In order to ensure PKC security, cryptographic algorithms based on new hard mathematical problems are required. Post-Quantum Cryptography refers to a set of algorithms based on different hard mathematical problems that are considered secure against

traditional and quantum computer attacks. It can be deployed on both classical computers and today's communication infrastructures. Among the different Post-Quantum algorithms, proposals based on the hardness of lattice problems are one of the main alternatives for practical implementations of Post-Quantum PKC. Lattice-based cryptographic constructions hold strong security proofs based on the worst-case hardness and offer a good trade-off between security and efficiency [LPR10]. Among the 26 candidates of the NIST second round Post-Quantum standardization process, 12 algorithms (9 KEM/encryption and 3 signature algorithms) belong to the category of lattice-based cryptography [Nat19]. As the Post-Quantum standard is not ready yet, a focus on maintaining crypto-agility is mandatory. That is, the Post-Quantum implementations should be as flexible as possible in order to meet the possible future developments.

Empowering electronic devices with strong security poses several design challenges due to limited resources, strict performance and flexibility requirements, the vulnerability to implementation attacks, and short time-to-market pressure. To cope with such challenges, Hardware/Software Co-Design strategies are usually employed. For a given application, their goal is to find an optimal assignment of tasks between software running on microprocessors and hardware implemented on an ASIC or an FPGA. By splitting the tasks of an algorithm into hardware and software elements, high speed and flexible implementations can be developed.

Co-Design is further fueled by the early success of open-source hardware initiatives (e.g., Sun OpenSPARC [Ora19], IBM OpenPOWER [Ope19], RISC-V [RIS19b], Linux Foundation CHIPS Alliance Project [Lin19]), allowing to shorten the design cycle for complex and highly customized embedded systems. As a result, the final hardware architecture may include a single or multiple processors, communication structure (bus or Network-on-Chip), memory, peripherals, and application-specific hardware components, also known as hardware accelerators. Two main models of hardware accelerators can be identified: i) *Loosely coupled accelerators*, which are implemented outside the processor cores. In order to reduce the communication overhead, these coprocessors usually run the complete cryptographic operation in hardware; and ii) *Tightly coupled accelerators*, which are deeply embedded within the processor architecture as application-specific functional units. While the loosely coupled alternative is easy to integrate into the system, usually it involves poor flexibility and high communication overhead. In contrast, the tightly alternative reduces the communication overhead and presents a low silicon area overhead. However, the Instruction Set Architecture (ISA) of the processor must be extended.

RISC-V is a free and open ISA, which enables a new era of processor innovation. An ISA defines a hardware/software interface as a collection of basic operations the processor must support, enabling software portability between different implementations of a particular ISA. RISC-V allows the development of open-source hardware with hardware security extensions and secure coprocessors.

A Post-Quantum ISA extension is recognized as an important task by the Cryptographic Extensions Task Group of the RISC-V Security Standing Committee [RIS19a]. However, up to now this committee has not addressed a Post-Quantum ISA extension. Overall, while many efforts towards the design and implementation of Post-Quantum loosely coupled accelerators for RISC-V architectures have gathered great attention, only a single work deals with tightly coupled accelerators through a RISC-V ISA extension.

Related Work. Hardware/Software Co-Design for accelerating Post-Quantum Cryptography has been the target of an increasing research effort. Loosely coupled Post-Quantum accelerators are proposed in [FSF+18, FDNG19, FSMG+19, BUC19, WTJ+20, MTK+20]. For the best of our knowledge, only the work of [AEL+20] proposed a tightly coupled Post-Quantum accelerator.

The work of [FSF+18] was the first initiative of Post-Quantum acceleration through

Co-Design. The authors implemented the IEEE-1363.1 standardized version of NTRUEncrypt. The software tasks were executed on an ARM Cortex-A53 and a loosely coupled coprocessor was used to accelerate the polynomial multiplication. A similar approach and platform was used in [FDNG19] for implementing several NTRU variants submitted to the NIST standardization process. In [MTK+20], Saber was implemented. The software tasks were executed on an ARM Cortex-A9 and the polynomial multiplication, based on Toom-Cook, was accelerated through a hardware coprocessor.

Several Post-Quantum implementations based on RISC-V and loosely coupled hardware accelerators have been proposed. In [WJW+18], XMSS was accelerated through the integration of a SHA-256 hardware core. Almost simultaneously in [FSMG+19], the authors implemented the CPA version of NewHope accelerated through an NTT and Keccak low-power coprocessor. In [BUC19], a flexible crypto-processor, able to support several lattice-based algorithms, was proposed. This work supports Frodo, NewHope, qTESLA and CRYSTALS-Kyber/Dilithium. Low-power modular arithmetic, NTT, and Keccak cores were used to accelerate polynomial computations. In [WTJ+20], the authors developed hardware accelerators for cSHAKE, Gaussian sampling, NTT, and sparse polynomial multiplication to accelerate qTESLA.

These previous RISC-V implementations that use loosely coupled accelerators for Post-Quantum Cryptography present three disadvantages:

- **High data transfer overhead:** caused by the intense communication among cores. Most designs use memory-mapped accelerators, which are connected through a bus, e.g., an AXI bus, to the processing element. Data that is sent to the accelerator has to be first loaded from the main memory into the processing element, i.e., into the register file, from where it is forwarded to the memory of the accelerator. When the accelerator finished the execution, the processed data has to be sent from the processing element back to the main memory. A Direct Memory Access (DMA) controller could speed up the data transfer, but will also increase the complexity of the design.

- **High amount of hardware resources:** they require huge buffers to store the input and output data of the accelerator. In particular, the NTT module usually requires a huge amount of memory for storing input data, output data, and LUTs for the Twiddle factors of the forward and inverse NTT. Previously proposed works use up to four separate memory blocks to execute the NTT transform. While this excessive use of resources might be acceptable for high performance devices, it is questionable if it is suitable for small embedded devices. The usage of multiple memories unnecessarily increases the design complexity and cost.

- **Low flexibility:** since standalone hardware accelerators usually run a complete cryptographic algorithm, they are inflexible and hard to update.

For the best of our knowledge, only a single work has exploited the potential of Hardware/Software Co-Design to create tightly coupled accelerators for Post-Quantum Cryptography. In [AEL+20], the authors integrated a finite field multiplier into the RISC-V architecture in order to accelerate polynomial multiplications in NewHope and Kyber. Their proposed finite field multiplier already lead to a good performance improvement. In contrast to the work of [AEL+20], in this work we present a set of more powerful hardware accelerators capable of performing vectorized modulo arithmetic, NTT computations, and efficient generation of random polynomials. Polynomial generation has been pointed by several previous works [KRSS19, AEL+20, ABCG20] as one of the main reasons for performance degradation. This bottleneck is usually even more critical than the polynomial arithmetic.

Contributions. In this work, we propose **RISQ-V**, an enhanced RISC-V architecture that embeds a set of powerful tightly coupled accelerators to speed up lattice-based Post-Quantum Cryptography. Their implementation nature, directly integrated into the RISC-V processing element, eliminates the drawbacks of loosely coupled accelerators by sharing the existing processor resources. Our results show that our proposal combines a good performance with flexibility and has a small silicon footprint. Thus, our solution is suitable for a wide range of applications, from constrained devices, such as smartcards or IoT platforms, to a powerful Multi-Processor System-on-Chip (MPSoC). In summary, this paper presents three contributions:

First, we propose a set of tightly coupled hardware accelerators capable of performing: i) two butterfly operations in parallel; ii) on-the-fly Twiddle factor generation; iii) vectorized modulo arithmetic; and iv) efficient generation of random polynomials. These accelerators are deeply integrated into the RISC-V pipeline.

Second, we create 28 new instructions for performing packed modular arithmetic (addition, subtraction, multiplication, multiply-accumulate), butterfly operation, update of Twiddle factors, update/multiplication with scaling factors, bit-reversal, hash computations, and binomial sampling.

Third, we developed a RISQ-V FPGA prototype and an ASIC implementation to evaluate its performance and cost. We exploit the RISQ-V capabilities to implement the Post-Quantum algorithms NewHope, Kyber and Saber with different security levels.

Organization. The remainder of this article is structured as follows: In Section 2, lattice-based cryptography and its performance bottlenecks are described. In Section 3, we present the design decisions and hardware architectures of the developed Post-Quantum accelerators. In Section 4, the integration of these accelerators into the RISC-V platform and the new Post-Quantum ISA extension are discussed. In Section 5, evaluation results of RISQ-V for an FPGA prototype and an ASIC implementation are presented. A conclusion is given in Section 6.

2   Background

2.1   Lattice-based Cryptography

Most lattice-based cryptosystems are built upon the Learning With Errors (LWE) problem. Among the different alternatives, the algebraically structured variants Ring-LWE (R-LWE) and Module-LWE (M-LWE); and the deterministic variant Module-Learning With Rounding (M-LWR), are very attractive since they offer performance advantages and smaller key sizes over the plain LWE problem [AD17, BPR12].

The R-LWE problem, was introduced in [LPR10]. In contrast to the plain LWE problem, R-LWE replaces n-dimensional vectors by polynomials of degree smaller than n. These polynomials are an element of the ring $\mathcal{R} = \mathbb{Z}_q/\langle\phi(x)\rangle$, with integers n and q, and the cyclotomic polynomial $\phi(x)$ that is usually chosen to be $x^n + 1$. This substitution allows to increase the computational performance and to decrease key and ciphertext sizes. The basic R-LWE instance can be written as in Eq. (1), where $\boldsymbol{a}$ is a public polynomial, $\boldsymbol{s}$ a secret polynomial and $\boldsymbol{e}$ an error polynomial.

$$\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{e} \tag{1}$$

All arithmetic operations are performed in the ring $\mathcal{R}$. The hardness of the R-LWE problem is based on the difficulty to recover $\boldsymbol{s}$ when $\boldsymbol{a}$ and the result $\boldsymbol{b}$ are given (search problem). Moreover, it is known to be a hard problem to distinguish between the pair $(\boldsymbol{a}, \boldsymbol{b})$ and a truly uniform and independent sample pair (decision problem). Usually, $\boldsymbol{a}$ is sampled from a uniform distribution U_q whose outcome ranges between 0 and $q - 1$. The

polynomials s and e are sampled from an error distribution Ψ_k, which is usually a binomial or Gaussian distribution. Among the candidates of the second round of the NIST Post-Quantum standardization process, NewHope [AAB$^+$19] is the only Key Encapsulation Mechanism (KEM), that is built upon the R-LWE problem.

The M-LWE problem was introduced in [LS15]. In contrast to R-LWE, the M-LWE problem replaces the single ring elements with module elements over the same ring. Instead of using a single large ring element, it uses matrices of smaller ring elements for constructing the public element, and vectors of smaller ring elements for the secret and error elements. R-LWE can be considered to be a special case of the M-LWE problem, where the width of the matrix A over the ring $\mathcal{R}$ is always 1 [ABD$^+$19]. To increase the security level, it is sufficient to increase the dimension of the matrices and vectors, while the ring is always the same. That is, the underlying ring operations do not change, instead the amount that these operations are performed changes (more or fewer). One of the main advantages of M-LWE is that the concrete-security/efficiency trade-off is highly tunable [AD17]. Kyber [ABD$^+$19] is a second round Post-Quantum NIST candidate built upon M-LWE.

The LWR problem was introduced in [BPR12] as a derandomized version of the LWE problem. It uses the rounding operator $\lfloor x \rceil_p$ with respect to some modulus $p < q$, which scales x by p/q and then rounds the result to the nearest integer modulo p. Therefore, in contrast to the LWE problem, the noise in the LWR problem is generated deterministically. LWR replaces $a \cdot s + e$ (given in Eq. 1) by the rounding function $\lfloor a \cdot s \rceil_p$ with respect to some modulus p. The module version M-LWR is a straightforward generalization of M-LWE. As a result, the schemes based on M-LWR are characterized by reduced bandwidth for keys and ciphertexts. Saber [DKRV19] is a second round Post-Quantum NIST candidate based on M-LWR.

The tightly coupled hardware accelerators developed in this work were used for enhancing the performance of the three promising NIST Post-Quantum candidates: NewHope (R-LWE), Kyber (M-LWE) and Saber (M-LWR). The developed hardware accelerators are able to support multiple security levels (from I to V). For the performance evaluation, always the lowest and highest NIST security level was chosen. To be precise, the following instances were evaluated: NewHope-512 (Level I), NewHope-1024 (Level V), Kyber-512 (Level I), Kyber-1024 (Level V), Lightsaber (Level I), and Firesaber (Level V). The following chapters describe the performance bottlenecks of these schemes and the proposed hardware accelerators.

2.2 Performance bottlenecks

The matrix/polynomial generation, addition and multiplication are the basic operations required to encrypt, decrypt, sign, and verify information in lattice-based cryptosystems.

Polynomial Generation. Structured LWE-based cryptography requires the sampling of large random polynomials. The standard way to create the public, secret and error polynomials is to use uniformly distributed random numbers and model each coefficient according to the desired distribution. The coefficients of the public polynomial can be obtained through the rejection sampling process. This technique selects (accepts) the uniform samples that fit to the target distribution (desired range), otherwise the sample is rejected. The coefficients of the secret and error polynomials must be binomially distributed. Although turning uniform random numbers into binomial samples just requires few computational cycles for a single polynomial coefficient, this operation does not scale well. Usually, to achieve the amount of randomness required to implement these two types of sampling processes, a Pseudo Random Number Generator (PRNG), e.g., a hash function, is included. This PRNG expands a small random seed extracted from a physical source of entropy in an electronic device. As reported in [KRSS19], the amount of time

spent in the execution of hash functions for lattice-based Post-Quantum schemes, like NewHope and Kyber, is in the range of 50% of the total execution time.

Polynomial Multiplication using the NTT.   Besides the generation of random polynomials, polynomial multiplication has been identified as the other most computationally intensive operation [PG12, DB16]. While polynomial addition can be performed with $\mathcal{O}(n)$ operations, a naive polynomial multiplication requires $\mathcal{O}(n^2)$ operations [FS19]. Thus, it becomes specially prohibitive for high dimensional polynomials (larger values of n). However, when polynomials are expressed in the spectral domain, the multiplication operation between a pair of polynomials can be performed coefficient-wise.

The Discrete Fourier Transform (DFT) is a general method to transform a polynomial $\boldsymbol{a}$ into the spectral domain. The coefficients of $\boldsymbol{a}$ are transformed as in Eq. 2, where $n-1$ is the degree of the polynomial and ω_n the n-th root of unity.

$$\hat{a}_i = \sum_{j=0}^{n-1} \omega_n^{ij} \cdot a_j \tag{2}$$

To transform the result vector from the spectral domain into the normal domain, the inverse must be applied as in Eq. 3.

$$a_i = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{-ij} \cdot \hat{a}_j \tag{3}$$

The Number Theoretic Transform (NTT) is a specialized version of the DFT where ω_n is an element of the finite ring $\mathbb{Z}_q$ instead of a complex number. For the NTT computation, the coefficient ring must contain primitive roots of unity. The n-th root of unity $\omega_n \in \mathbb{Z}_q$, while $\omega_n^n = 1 \mod q$ and $\omega_n^i \neq 1 \mod q$ for $\forall i \in [0, n-1]$. The polynomial multiplication $\boldsymbol{c}$ between a pair of polynomials $\boldsymbol{a}$ and $\boldsymbol{s}$ through the NTT is calculated as in Eq. 4.

$$\boldsymbol{c} = \text{NTT}^{-1}(\text{NTT}(\boldsymbol{a}) \odot \text{NTT}(\boldsymbol{s})) \tag{4}$$

Where $\odot$ denotes a coefficient-wise multiplication and NTT^{-1} is the inverse NTT.

The product polynomial is reduced after the polynomial multiplication by the cyclotomic polynomial $\phi(x) = x^n + 1$. In this way, the $2n$-length product polynomial is reduced to a n-length polynomial. When the $2n$-th root of unity $\gamma_n = \sqrt{\omega_n}$ exists, the reduction in the spectral domain simply corresponds to scaling the i-th coefficient of the polynomials with the factor γ_n^i before the NTT, and with γ_n^{-i} after the NTT^{-1}. Thus, it can be avoided to pad the polynomials $\boldsymbol{a}$ and $\boldsymbol{s}$ in Eq. 4 with zeros. The $2n$-th root of unity exits when $q \equiv 1 \mod 2n$ holds. The NTT (forward) and NTT^{-1} (inverse) reduced by $\phi(x)$ can be written as in Eq. 5 and Eq. 6, respectively.

$$\hat{a}_i = \sum_{j=0}^{n-1} \gamma_n^j \cdot \omega_n^{ij} \cdot a_j \quad , \tag{5}$$

$$a_i = \frac{1}{n} \cdot \gamma_n^{-i} \sum_{j=0}^{n-1} \omega_n^{-ij} \cdot \hat{a}_j \quad . \tag{6}$$

To speed up the DFT from a complexity of $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot log(n))$, the Fast Fourier Transform (FFT) can be used. This is also true for the NTT, while the name remains in this case. The FFT exploits the symmetric structure of the Fourier transform by splitting one DFT into smaller DFTs. The computation that breaks a larger DFT into smaller DFTs (subtransforms) is called butterfly operation. The butterfly operation is also used to combine the smaller DFTs into a larger DFT. Two of the most used algorithms to

realize the fast NTT and NTT^{-1} operations are the Cooley-Tukey (CT) [CT65] and the Gentleman-Sande (GS) [GS66] algorithms. While the arithmetic of both algorithms is similar, they mainly differ in the way they store and access data and in the way how the butterfly operation is performed. The butterfly operation consists of a multiplication by a power of ω_n, an addition, and a subtraction in $\mathbb{Z}_q$. The powers of ω_n are called *Twiddle factors*. While the Cooley-Tukey algorithm follows the decimation-in-time approach, i.e., $x' \leftarrow x + y\omega_n$ and $y' \leftarrow x - y\omega_n$ with $\omega_n, x, y \in \mathbb{Z}_q$, the Gentleman-Sande algorithm follows the decimation-in-frequency approach, i.e., $x' \leftarrow x + y$ and $y' \leftarrow (x - y)\omega_n$.

Polynomial Multiplication using Karatsuba and Toom-Cook. Polynomial multiplications using the NTT are very efficient. However, the NTT can only be applied on a restricted parameter set. The prime should be 'NTT-friendly', i.e., it should be chosen such that $x^n - 1$ can be expressed as a product of linear factors. This, for instance, is the case when $q \equiv 1 \mod 2n$. Not all lattice-based algorithms have NTT-friendly primes. For example Saber does not use a prime for its modulo reductions and sets $q = 2^{13}$. This excludes the usage of the NTT but simplifies the modulo arithmetic. There are other alternatives to efficiently multiply polynomials. Two of the most used algorithms for performing flexible and efficient multiplications are Karatsuba and Toom-Cook. These algorithms iteratively split a large polynomial multiplication into several smaller polynomial multiplications. Karatsuba outperforms Toom-Cook for low to medium degree polynomials. Otherwise, Toom-Cook performs more efficient.

Karatsuba Multiplication: The Karatsuba algorithm reduces the quadratic runtime of the polynomial multiplication to a complexity of $\mathcal{O}(n^{log_2(3)}) \approx \mathcal{O}(n^{1.58})$. The length-$m$ polynomials $\boldsymbol{a}$ and $\boldsymbol{s}$ are split into two length-$m/2$ polynomials: i) lower part ($\boldsymbol{a}^l$, $\boldsymbol{s}^l$); and ii) higher part ($\boldsymbol{a}^h$, $\boldsymbol{s}^h$). Instead of four polynomial multiplications of these length-half polynomials, Karatsuba's tweak requires only three different multiplications as in Eq. 7

$$\boldsymbol{c} = \boldsymbol{a}\boldsymbol{s} = \boldsymbol{a}^l\boldsymbol{s}^l + ((\boldsymbol{a}^h + \boldsymbol{a}^l)(\boldsymbol{b}^h + \boldsymbol{b}^l) - \boldsymbol{a}^h\boldsymbol{b}^h - \boldsymbol{a}^l\boldsymbol{b}^l)x^{m/2} + \boldsymbol{a}^h\boldsymbol{s}^h x^m \ . \tag{7}$$

Toom-Cook Multiplication: The Toom-Cook multiplication is a generalization of the Karatsuba algorithm. Instead of splitting polynomials into two smaller polynomials, the polynomial is split into k parts. For the case $k = 2$, the k-way Toom-Cook algorithm is identical to the Karatsuba algorithm.

3 Hardware Accelerators

In order to enhance the performance of lattice-based cryptography, in this work the following hardware accelerators were designed: i) an NTT and modular arithmetic accelerator; ii) an accelerator used for Karatsuba/Toom-Cook multiplications; iii) a Keccak accelerator for the pseudo random number generation; and iv) a binomial sampling accelerator. These modules reduce the performance bottlenecks discussed in Section 2 and speed up the overall performance.

3.1 Number Theoretic Transform (NTT) Design

In this subsection, the optimization techniques and design choices for our NTT architecture are presented. The performance and size of the architecture for the NTT described in Section 2 are highly influenced by: i) the calculation of Twiddle factors; ii) the bit-reversal computation; and iii) the memory access strategy.

Our NTT architecture is mainly designed for the NewHope instances NewHope-512 and NewHope-1024, but it also supports the Kyber instances Kyber-512, Kyber-768, and Kyber-1024. As Saber chooses the modulo $q = 2^{13}$, it is not suitable for the NTT. While

NewHope-512 and NewHope-1024 have a polynomial length of $n = 512$ and $n = 1024$, respectively, Kyber has the same polynomial length of $n = 256$ for all instances. As the polynomial length highly affects the NTT costs, different design decisions were made for NewHope and Kyber.

Calculation of Twiddle factors (powers of ω_n):   Most of the optimized software implementations for performing the NTT precompute the Twiddle factors and store them into a separated memory location. As the generated tables for storing these Twiddle factors are very large for high-degree polynomials, previous hardware architectures devised an approach to compute these values (powers of ω_n) on-the-fly [RVM$^+$14, FS19]. Due to the high memory access latency of long tables, calculating the Twiddle factors on-the-fly can be even faster than loading them from memory. In order to combine the advantages of precomputation and on-the-fly Twiddle factors computation, in this work we propose a hybrid approach. For larger polynomials, such as in NewHope-512 and NewHope-1024, on-the-fly computation is used. Otherwise, precomputed tables are used, i.e., for all Kyber instances.

Bit-reversal computation: Performing a reorder or bit-reversal step before or after the NTT computation can be an expensive operation. There are several in-place variants of the Cooley-Tukey and Gentleman-Sande algorithms, in the following referred to as $NTT_{br \to no}^{CT}$, $NTT_{no \to br}^{CT}$, $NTT_{br \to no}^{GS}$, and $NTT_{no \to br}^{GS}$. The difference between these variants is the order of the input and output data. At the bit-reversed to normal order $(br \to no)$ variant, the input coefficients must be loaded in bit-reversed order and the output is given in normal order. For the $(no \to br)$ variant, the input is in normal order and the output in bit-reversed order. Previous works avoid the bit-reversal step by using a combination of different algorithms for the forward and inverse transform, e.g., $NTT_{no \to br}^{CT}$ and $INV\text{-}NTT_{br \to no}^{GS}$ [POG15]. However, this combination requires that the Twiddle factors are loaded during the NTT computation in bit-reversed order, which increases the complexity of the on-the-fly computation. In order to avoid large memories, we decided to use the basic $NTT_{br \to no}^{CT}$ algorithm for the forward as well as the inverse transform of NewHope. The input coefficients of the forward transform are randomly sampled from the error distribution, consequently no bit-reversal is required for the forward transform. As for Kyber, the precomputation of the Twiddle factors is due to the smaller polynomial length less expensive, we use the combination of $NTT_{no \to br}^{CT}$ and $INV\text{-}NTT_{br \to no}^{GS}$ to completely avoid the bit-reversal step. This approach follows the reference implementation of Kyber, which uses these two NTT variants.

Memory Access Strategy: The main performance bottlenecks of the NTT are the load and store operations to transfer the coefficients between the main memory and the processing element. A non-optimized NTT architecture always loads two coefficients from the memory into the register file, performs the butterfly operation, and stores the result. The NTT can be divided into $log_2(n)$ layers, with n being the polynomial length. A naive approach requires to load and store all n coefficients in each layer, resulting in total in $n \cdot log_2(n)$ load and store operations, respectively. To decrease the memory access overhead two methods can be used: merging of NTT layers or storing two coefficients in one memory word.

Merging the NTT layers was already discussed in [AJS16, BKS19, ABCG20]. The goal is to keep a certain amount of coefficients as long as possible within the register file. Let l denote the total amount of coefficients that can be stored within the register file. After computing the butterfly operations for the l coefficients within the first layer, the results are not written back to the main memory. Instead of completing the first layer, the next

layer is already processed. This method saves the memory accesses for the next layer. When l coefficients can be loaded into the registers, up to $log_2(l)$ layers can be merged. Previous works loaded up to 16 coefficients into the registers to process up to four NTT levels without reloading coefficients between the layers. In [ABCG20], for NewHope-512 $3 + 3 + 3$ layers of in total 9 layers and for Kyber $3 + 3 + 1$ layers of in total 7 layers were merged. In this work, we use the complete floating point register set of the RISC-V core (32×32 bit) to store $l = 2 \times 32 = 64$ coefficients (each 16 bit) for NewHope. An address controller loads the coefficients directly from the floating point register set into two butterfly units such that the calculation of multiple NTT levels can be performed. As Kyber requires different algorithms for the forward and inverse transforms, no dedicated address controllers are developed. This means, for Kyber, the general purpose register set was used to store $l = 16$ coefficients (each 16 bit) allowing to merge $3 + 3 + 1$ layers.

To further reduce the amount of load and store operations, multiple coefficients can be stored in a single memory word (32 bit). As the polynomial coefficients of NewHope and Kyber can be expressed with at most 16 bit, two coefficients are stored in a single word. After performing the butterfly operation, the intermediate results of the coefficients are swapped in order to prepare them for the next layer. Note that precalculated Twiddle factors can be stored in any desired order. Such particularity enhances the traverse possibilities through the NTT structure and allows to avoid the swapping operation.

Figure 1 illustrates an example of the applied NTT layer merging and swapping techniques for the $NTT_{br \leftarrow no}^{CT}$ variant that we use for NewHope. In the small example for $n = 16$, eight coefficients can be loaded from the main memory into the registers ($l = 8$). Moreover, two butterfly operations can be processed in parallel within the units *BF0* and *BF1*. The red rectangles show the order in which the coefficient pairs are processed by the two butterfly units. At the beginning, the first eight coefficients $a_0, a_8, a_4, a_{12}, a_2, a_{10}, a_6, a_{14}$ are loaded pairwise into the register file. In the following, we use the term *distance* to describe the difference between the positions of two coefficients in an array. In the first layer, the distance between the coefficients that are processed together is equal to one, i.e., they are located in the same register. The coefficient pairs (a_0, a_8), (a_4, a_{12}) are processed first using the two butterfly units and (a_2, a_{10}), (a_6, a_{14}) are processed next. Instead of storing the result back to the main memory, the second layer is already processed. In the second layer, the distance between the coefficients that are processed together is equal to two. It requires the results of the first layer for the coefficient pairs (a_0, a_4), (a_2, a_6) and (a_8, a_{12}), (a_{10}, a_{14}). As these coefficient pairs are not next to each other, i.e., they are not within the same register, a swap is performed by the butterfly units indicated in Figure 1 by the blue arrows. This swapping operation, which is an exchange of 16 bit values in two registers, can be performed in hardware almost for free. In the third layer, the distance between the coefficients that are processed together is four, which means that some required coefficients would still be within the register set. However, not all coefficients required for the swapping operation to prepare the content of the registers for the next layer are located in the register set. Therefore, the amount of layers that can be merged is $log_2(l) - 1$ and in the example shown in Figure 1 two. However, the swapping technique allows to have always the right coefficients in the same memory word, which significantly decreases the access to the main memory. Table 1 illustrates the register content and the input values for the butterfly units *BF0* and *BF1* for the small example in Figure 1. An unoptimized version of the NTT diagram is shown in Figure 9 (Appendix A). This unoptimized version only loads two coefficients at the same time, has only one butterfly unit, and processes layer after layer. While in this small example the unoptimized version shown in Figure 9 requires $n \cdot log_2(n) = 16 \cdot 4 = 64$ memory accesses, the optimized version shown in Figure 1 ($n = 16$, $l = 8$) requires $n/2 \cdot (log_2(n) + 1 - (log_2(l) - 1)) = 24$ memory accesses.
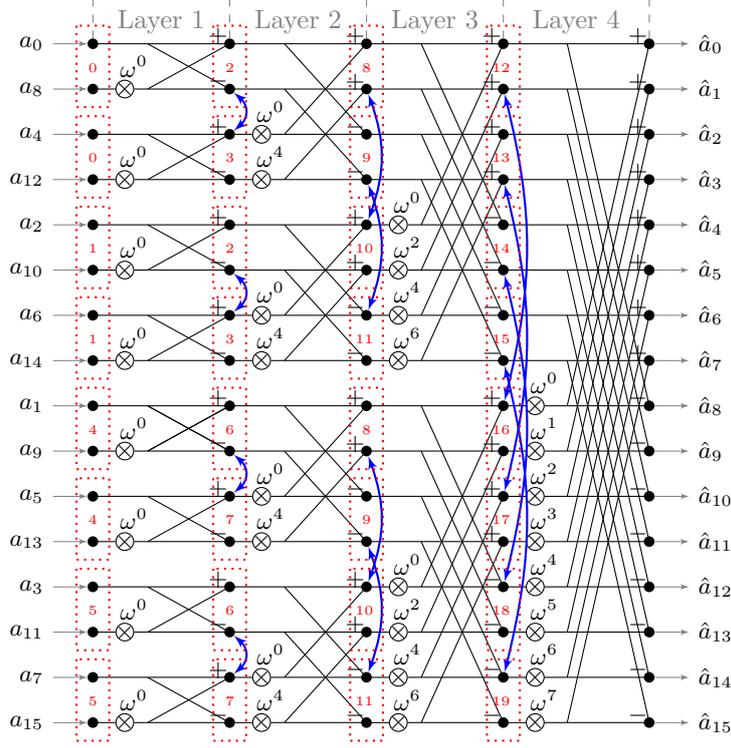
Figure 1: Example $NTT_{br \leftarrow no}^{CT}$ with $n = 16$. In this case, two coefficients are stored in a single word, eight coefficients can be loaded to the register file ($l = 8$), and two pairs of coefficients can be executed in parallel. The red boxes indicate which coefficients are stored together in one word and in which order the coefficients are processed by the two butterfly units. The blue arrows show the coefficients which are swapped after the butterfly operations. For $l = 8$, $log_2(l) - 1 = 2$ layers were merged.

Table 1: Register content and input for the two butterfly units BF0 and BF1 for the example $n = 16$, $l = 8$.

Register Content	Load coeffs.	Step 0		Step 1		Step 2		Step 3		Store Coeffs./ Load Coeffs.	
		BF0	BF1	BF0	BF1	BF0	BF1	BF0	BF1		...
R0	a_0, a_8	a_0, a_8				a_0, a_4				a_1, a_9	
R1	a_4, a_{12}		a_4, a_{12}					a_8, a_{12}		a_5, a_{13}	
R2	a_2, a_{10}			a_2, a_{10}			a_2, a_6			a_3, a_{11}	
R3	a_6, a_{14}				a_6, a_{14}				a_{10}, a_{14}	a_7, a_{15}	

Algorithm 1 shows the $NTT_{br \leftarrow no}^{CT}$ algorithm with the discussed optimizations: on-the-fly Twiddle factor calculation, storing two coefficients in one memory word, and the swapping operation. The merging technique is not directly included in this algorithm, but will be discussed as well. The algorithm can be divided into three steps: input preparation, calculation of the first NTT layers, and calculations of the last NTT layer.

In the first step, the coefficients of polynomial $\boldsymbol{a}$ are stored in a bit-reversed order in the main memory (line 1). The input coefficients are stored in a way that two coefficients are always stored in a single word.

In the second step (line 2-19), the first $log_2(n) - 1$ NTT layers are calculated. The Twiddle factor ω is initialized in line 4, and always updated during runtime by a modulo multiplication with ω_m (line 17). The value of ω_m depends on the current NTT layer indicated by the variable m. In practice, the values for ω_m still need to be precomputed.

Algorithm 1: NTT transform

Input: Coefficients a_i, with $i = 0, 1, \ldots, n-1$, precalculated values of ω_m
Result: Coefficients $\hat{a}_i$, with $i = 0, 1, \ldots, n-1$

1 $a \leftarrow \text{BitReversal}(a)$
2 **for** $m = 2$ **to** $n/2$ **by** $m = 2m$ **do**
3 $\quad \omega_m \leftarrow \omega_n^{n/m}$
4 $\quad \omega \leftarrow \omega_n^{n/(2m)}$ or 1 for NTT^{-1}
5 $\quad$ **for** $j = 0$ **to** $m/2 - 1$ **by** 1 **do**
6 $\quad\quad$ **for** $k = 0$ **to** $n/2 - 1$ **by** m **do**
7 $\quad\quad\quad a_{k+j+m/2}, a_{k+j} \leftarrow \text{MEM}_{k+j}$ `// Load first coefficient pair`
8 $\quad\quad\quad H_1, L_1 \leftarrow a_{k+j+m/2}, a_{k+j}$
9 $\quad\quad\quad a_{k+j+3m/2}, a_{k+m+j} \leftarrow \text{MEM}_{k+j+m/2}$ `// Load second coefficient pair`
10 $\quad\quad\quad H_2, L_2 \leftarrow a_{k+j+3m/2}, a_{k+m+j}$
11 $\quad\quad\quad H_1, H_2 \leftarrow (H_1 \cdot \omega) \mod q, \quad (H_2 \cdot \omega) \mod q$
12 $\quad\quad\quad a_{k+j+m/2}, a_{k+j} \leftarrow (L_1 - H_1) \mod q, \quad (L_1 + H_1) \mod q$
13 $\quad\quad\quad a_{k+j+3m/2}, a_{k+m+j} \leftarrow (L_2 - H_2) \mod q, \quad (L_2 + H_2) \mod q$
14 $\quad\quad\quad \text{MEM}_{k+j} \leftarrow a_{k+j+m}, a_{k+j}$ `// Store and swap coefficients`
15 $\quad\quad\quad \text{MEM}_{k+j+m/2} \leftarrow a_{k+j+3m/2}, a_{k+j+m/2}$ `// Store and swap coefficients`
16 $\quad\quad$ **end**
17 $\quad\quad \omega \leftarrow (\omega \cdot \omega_m) \mod q$
18 $\quad$ **end**
19 **end**
20 $m \leftarrow n$ `// Prepare last round`
21 $\omega_m \leftarrow \omega_n^{n/m}$
22 $\omega \leftarrow \omega_n^{n/(2m)}$ or 1 for NTT^{-1}
23 **for** $j = 0$ **to** $m/2 - 1$ **by** 1 **do**
24 $\quad a_{j+m/2}, a_j \leftarrow \text{MEM}_j$ `// Load coefficient pair`
25 $\quad H_1, L_1 \leftarrow a_{j+m/2}, a_j$
26 $\quad H_1 \leftarrow (H_1 \cdot \omega) \mod q$
27 $\quad a_{j+m/2}, a_j \leftarrow (L_1 - H_1) \mod q, \quad (L_1 + H_1) \mod q$
28 $\quad \text{MEM}_j \leftarrow a_{j+m/2}, a_j$ `// Store without swap`
29 $\quad \omega \leftarrow (\omega \cdot \omega_m) \mod q$
30 **end**

However, only one entry for each layer is required, resulting in $log_2(n)$ precomputations. To merge the multiplication by powers of γ, ω is initialized with the square root of ω_m (line 4). The same precomputations as for ω_m can be used because $\omega_m^{1/2}$ and ω_m from the previous layer have the same value. Only for the first layer ($m = 2$), the value $\omega_m^{1/2}$ has to be computed.

The most important part of Algorithm 1 are the butterfly operations, described in the inner loop (line 7-15). At the beginning, two coefficient pairs are loaded from the main memory locations MEM_{k+j} and $MEM_{k+j+m/2}$ and are assigned to two temporary variables each consisting of a lower halfword L_1/L_2 and a higher halfword H_1/H_2 (line 7-10). Then, two butterfly operations are performed (line 11-13). Finally, the result is swapped and stored in the respective memory location (line 14-15).

The third and last step is the computation of the last NTT layer (line 20-30). The operations of the last layer are similar to the operations of the other layers. The main difference is that no swapping operation is required.

The algorithmic modifications required for the merging technique include the manipulation of the start and end values of the outer and inner loop (lines 2 and 6). For the hardware architecture, we consider 32 registers available from the floating point register set for storing $l = 64$ coefficients. In this way, five layers can be merged. The nested loops can be split into $n/64$ parts. The outer loop, which indicates the current layer, starts for each part from 2^1 and terminates at the value 2^5, i.e., $m = \{2, 4, 8, 16, 32\}$. The inner loop iterates for the i-th part from $k = 32i$ to $k = 32i + 31$, where $i \in [0, \ldots, n/64 - 1]$. Instead of loading and storing the coefficients from the main memory, the coefficients are kept within the register set and are only refreshed between the $n/64$ parts.

3.2 Number Theoretic Transform (NTT) Architecture

Our proposed hardware architecture for the *NTT and Modular Arithmetic Unit* is shown in Figure 2. It is composed of three main modules: *Address Unit*, *Twiddle Update Unit* and *Modular Arithmetic Unit*. The architecture is optimized for the $NTT_{br \leftarrow no}^{CT}$ algorithm, which is used in NewHope. However, it also supports different NTT algorithms. The input of the *NTT and Modular Arithmetic Unit* is the content of two registers from the processor's register bank (with lower halfwords L_1/L_2 and higher halfwords H_1/H_2), and the control signals from the instruction decoder. The output consists of the processed input values and the control signals for the register bank.

Address Unit: This module controls the merging operation by setting the two read and write addresses for the register bank according to Algorithm 1 (with the modified loop start and end values) in order to merge five NTT layers. In this work, the automatic address calculation is only supported for $NTT_{br \leftarrow no}^{CT}$ and therefore only for NewHope. Supporting different NTT variants would be possible but would also lead to a non-negligible increase of area. The address calculation is triggered by the *multiple_bf* signal. At each clock cycle three sets of signals are updated: the read addresses *raddr*, the write addresses *waddr*, and the write enable signal *wen*. The *Address Unit* is also responsible for selecting the correct index for the small LUT of the precalculated values of ω_m and for triggering the Twiddle factor update. When the *NTT and Modular Arithmetic Unit* is in single operation mode, the *Address Unit* remains in *idle* state.

Twiddle Update Unit: This module calculates the Twiddle factors ω on-the-fly. The update of ω is always triggered by the *update_w* signal, which is either set by the *Address Unit* or by the corresponding signal from the instruction decoder. The current Twiddle factor $\omega = \omega \cdot \omega_m \mod q$ is updated as described in lines 17 and 29 of Algorithm 1. The value of ω is initialized according to line 4 and line 22. The value for ω_m is determined through the current NTT layer. All $log_2(n)$ possible values for ω_m are precalculated and
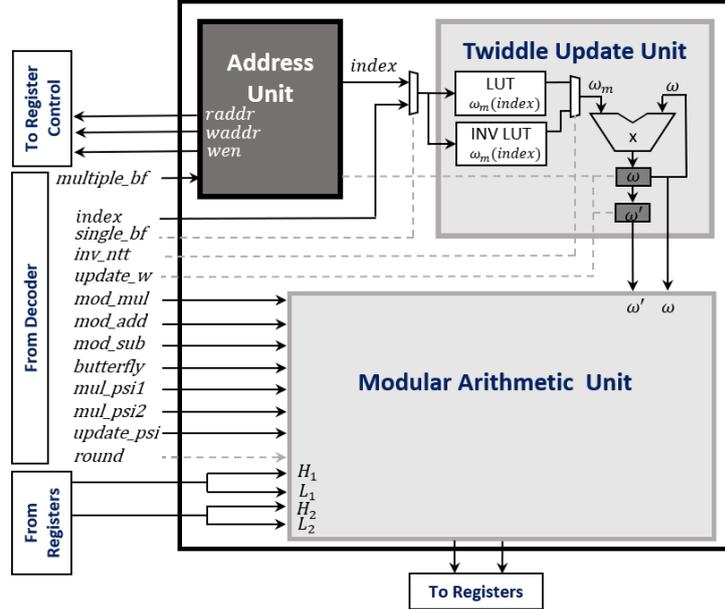
Figure 2: NTT and Modular Arithmetic Unit

stored in a LUT within the hardware accelerator. The *index* signal is used to select the correct value from the LUT. Depending on the value of the *inv_ntt* signal, either the LUT for the forward NTT or the inverse NTT is selected.

Modular Arithmetic Unit: This module performs the arithmetic for the following operations: *Butterfly Operation* (decimation-in-time and decimation-in-frequency), *Post-Processing Operation* (*mul_gamma*1, *mul_gamma*2, *update_gamma*), and *Vectorized Modulo Arithmetic* (*mod_mul, mod_add, mod_sub*). It is composed of a set of arithmetic elements, registers, and a *Reordering MUX*. A detailed microarchitecture of the *Modular Arithmetic Unit* in the *Butterfly Operation* mode is shown in Figure 3. The architectures of the *Modular Arithmetic Unit* in the *Post-Processing Operation* and *Vectorized Modulo Arithmetic* operation mode look very similar and are shown in Figure 10 and Figure 11 (Appendix B). The only difference between these figures are the signals which are forwarded by the multiplexers.

The *Butterfly Operation* calculates two butterfly operations $\{L_1 - H_1 \cdot \omega, \ L_1 + H_1 \cdot \omega\}$ and $\{L_2 - H_2 \cdot \omega, \ L_2 + H_2 \cdot \omega\}$ in parallel. The calculation is triggered by the *butterfly* signal. In order to calculate two butterfly operations in parallel within the last NTT layer (Algorithm 1, line 23-30), the previous Twiddle factor ω' is also forwarded to the *Modular Arithmetic Unit*. As a result, the utilization of the *Modular Arithmetic Unit* can be increased and the execution time of the inverse NTT can be decreased. The *Reordering MUX* is responsible for preparing the coefficients to the next round (layer) by swapping the coefficients after the butterfly operation for all rounds, except for the last one. The architecture was extended in order to support the calculation of the decimation-in-frequency butterfly operation, i.e., the calculation of $\{(L_1 - H_1) \cdot \omega, \ L_1 + H_1\}$ and $\{(L_2 - H_2) \cdot \omega, \ L_2 + H_2\}$. For reasons of better visibility, this extension is not shown in Figure 3. The calculation of $L_1 + H_1$ and $L_2 + H_2$ can be done by the existing modulo adders. To avoid any combinatorial loop in the circuit, either two additional modulo multipliers or modulo subtractors are required. As the modulo subtractors are more compact (less area consuming) than the modulo multipliers, they were used in this work to enhance the design. These subtractors were placed on top of the existing multipliers. The
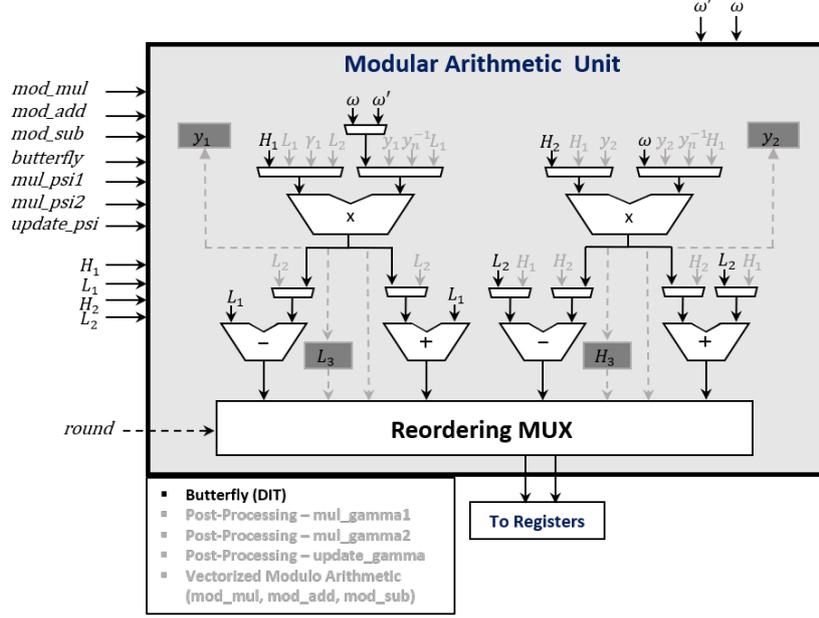
Figure 3: Modular Arithmetic Unit – Butterfly Operation decimation-in-time (DIT)

input of the additional subtractors are (L_1, H_1) and (L_2, H_2). The output is connected to the multiplexers of the left input of the existing multipliers.

The *Post-Processing Operation* (*mul_gamma*1, *mul_gamma*2, *update_gamma*) is used to calculate the multiplications with n^{-1} and γ^{-i} at the inverse NTT operation. These multiplications can be merged with the last layer of the inverse NTT. Therefore, first the control signal *mul_gamma1* ensures that the coefficients a_i and $a_{i+n/2}$ of the first input (L_1, H_1) are multiplied with $\gamma_1 = n^{-1}\gamma^{-i}$ and $\gamma_2 = n^{-1}\gamma^{-i-n/2}$, respectively. Before the multiplication with the first coefficients a_0 and $a_{n/2}$ starts, γ_1 is initialized with $n^{-1}\gamma_n^{-1} \mod q$ and γ_2 with $n^{-1}\gamma_n^{-n/2} \mod q$. The *mul_gamma2* control signal is used to perform the multiplications with the next coefficient pair a_{i+1} and $a_{i+1+n/2}$. The *Reordering MUX* brings the results of the *mul_gamma1* and *mul_gamma2* operation in the desired order $\{a_{i+1}, a_i\}$ and $\{a_{i+1+n/2}, a_{i+n/2}\}$ and assigns the result to the output. The *update_gamma* signal is used to update $\gamma_1 = \gamma_1\gamma_n^{-1}$ and $\gamma_2 = \gamma_2\gamma_n^{-1}$ after each *mul_gamma*1 and *mul_gamma*2 operation.

The *Vectorized Modulo Arithmetic* (*mod_mul, mod_add, mod_sub*) operation uses the *Modular Arithmetic Unit* to calculate vectorized modulo multiplications ($L_1 \cdot L_2 \mod q$ and $H_1 \cdot H_2 \mod q$); additions ($L_1 + L_2 \mod q$ and $H_1 + H_2 \mod q$); and subtractions ($L_1 - L_2 \mod q$ and $H_1 - H_2 \mod q$). Where L_1 and L_2 are the lower 16 bit of the two forwarded registers from the register set and H_1 and H_2 are the higher 16 bit. The two results from the modulo multipliers, adders, or subtractors are combined in the *Reordering MUX* and assigned to the output signal. The corresponding control signals *mod_mul*, *mod_add*, and *mod_sub* are used to perform the vectorized modulo arithmetic and to output the desired result. Similar to the *Butterfly Operation*, the *Vectorized Modulo Arithmetic* belongs to the category of packed (vectorized) arithmetic and follows the Single Instruction Multiple Data (SIMD) principle. This means a single instruction is used to process multiple data elements in parallel.

3.3 Bit-Reversal

The bit-reversal operation is a particular permutation of a sequence of elements. As discussed in Section 3.1, it is a key part of the NTT. Consider an array of n elements, the index of the i-th element a_i can also be represented in binary notation $i = \{b_0, b_1, \ldots, b_{log_2(n)-1}\}$. The bit-reversal operation swaps the i-th element with the j-th element which has the bit-reversed index $j = \{b_{log_2(n)-1}, b_{log_2(n)-2}, \ldots, b_0\}$. Reversing the bits is an expensive software task. A straightforward approach with a runtime of $\mathcal{O}(m)$ is looping through all $m = log_2(n)$ bits of an integer. The fastest solution is to use a LUT. This LUT requires n entries, each consisting of m bits. In particular, for large arrays, i.e., for large polynomial lengths, this approach will lead to a high memory footprint (large LUT). In order to have a better trade-off between memory footprint and performance, we extend the RISC-V ISA and develop special instructions for the bit-reversal operation. These instructions are derived from the store word/halfword operations *sw, sh*. In addition to the value to be stored and the destination address, the new instructions also take an offset. This offset is added in a bit-reversed order to the destination address. The functionality of the new instructions can be expressed as MEM[$rs1$+bitrev($rs2$)]$\leftarrow rd$, where $rs1$ contains the destination address, $rs2$ the offset, and rd the value that is stored. Reversing the offset in hardware can be efficiently solved through a rewiring process. The bit-reversal of a whole polynomial can be performed by loading in a loop each coefficient, and storing the coefficient with the new instruction. In this case, the offset simply corresponds to the loop counter.

3.4 Results for the NTT Accelerator

In this paper, we use the PULP RISC-V Toolchain[1] (version 7.1.1) with the optimization flag 'O3' (optimization for speed) to compile the code. A description of the used RISC-V platform is given in Section 4. Table 2 summarizes the clock cycle count required for performing the NTT, inverse NTT, and bit-reversal step in NewHope and Kyber. While in NewHope different polynomial lengths are used for different security levels ($n = 512$ and 1024), Kyber uses the same polynomial length for all security levels ($n = 256$). In all implementations mentioned in Table 2, the bit-reversal step was eliminated for Kyber by using two different NTT variants. The baseline for our optimized implementations are the clean C-code versions of the PQ-M4 project in [KRSS19], which are based on the reference implementations in [AAB+19] and [ABD+19].

In comparison to the baseline implementation on RISC-V, the optimized NewHope implementation achieves a speedup factor of 13.18/12.40 (NTT/NTT^{-1}) for length-512 polynomials and a speed up factor of 13.01/11.95 (NTT/NTT^{-1}) for length-1024 polynomials. The integrated optimization techniques (on-the-fly Twiddle factor calculation, merge of five NTT layers with direct access of the *NTT and Modular Arithmetic Unit* to the processor's floating point register set, and the storage of two coefficients in one word) result in a significant performance improvement when compared to the baseline implementation. The new bit-reversal instruction does not only eliminate the LUT for this step (1024 bytes for NewHope-512 and 2048 bytes for NewHope-1024), but also leads to an improvement in speed. Although the architecture of the *NTT and Modular Arithmetic Unit* is not optimized for the $NTT^{CT}_{no \rightarrow br}$ and $INV\text{-}NTT^{GS}_{br \rightarrow no}$ variants, Kyber also achieves a considerable speedup factor of 17.93/27.79 (NTT/NTT^{-1}) when compared to the baseline implementation. In fact, only the parallel decimation-in-frequency butterfly operations and the vectorized modulo multiplication of the *NTT and Modular Arithmetic Unit* are used.

Our results show that our architecture beats the clock cycle count of the latest assembler optimized ARM Cortex-M4 implementations in [ABCG20], the RISC-V imple-

---

[1] https://github.com/pulp-platform/pulp-riscv-gnu-toolchain

mentation in [AEL+20], which uses a finite field multiplier to accelerate the NTT, and the Hardware/Software Co-Design architecture of NewHope in [FSMG+19], which uses a loosely coupled NTT accelerator. In [AEL+20], the bit-reversal step was also eliminated for NewHope. However, two different NTT variants have to be used, implying an increase of code size. In [FSMG+19], the bit-reversal costs were hidden by a re-wiring during the transfer of the coefficients to the memory of the NTT accelerator.

In addition, in contrast to the NewHope baseline implementation and the implementations presented in [ABCG20, AEL+20], our proposed architecture does not require large LUTs. While the NewHope reference implementation in [AAB+19] requires $7n$-bytes (n denotes the polynomial length) for storing LUTs for the bit-reversal step, Twiddle factors and pre-/post processing, our implementation only requires $4 \cdot log_2(n) + 4$ bytes. In concrete numbers, the LUTs were reduced from 7168 to 44 bytes for NewHope-1024 and from 3584 to 40 bytes for NewHope-512. In contrast to the design in [FSMG+19], no additional data memory is required for the accelerator. A summary of the required hardware costs for the NTT as well as the whole NewHope and Kyber implementations is presented in Section 5.

Table 2: Clock cycle count for the NTT

	Device	**NTT**	**NTT^{-1}**	**BR**
NewHope-512 [ABCG20]	ARM Cortex-M4	$31,217$	$23,439$	$-$
NewHope-512 [AEL+20]	RISC-V (VexRiscv)	$14,787$	$14,893$	0
NewHope-512 baseline	RISC-V (PULPino)	$107,666$	$107,668$	$6,623$
NewHope-512 opt.	**RISC-V (PULPino)**	**$8,169$**	**$8,684$**	**$2,056$**
NewHope-1024 [FSMG+19]	RISC-V (PULPino)	$24,609$	$24,609$	0
NewHope-1024 [ABCG20]	ARM Cortex-M4	$68,131$	$51,231$	$-$
NewHope-1024 [AEL+20]	RISC-V (VexRiscv)	$31,295$	$31,735$	0
NewHope-1024 baseline	RISC-V (PULPino)	$241,121$	$241,123$	$13,279$
NewHope-1024 opt.	**RISC-V (PULPino)**	**$18,537$**	**$20,171$**	**$4,105$**
Kyber [ABCG20]	ARM Cortex-M4	$6,855$	$6,983$	0
Kyber [AEL+20]	RISC-V (VexRiscv)	$6,868$	$6,367$	0
Kyber baseline	RISC-V (PULPino)	$34,703$	$53,636$	0
Kyber opt.	**RISC-V (PULPino)**	**$1,935$**	**$1,930$**	0

3.5 Hardware Accelerator for Karatsuba/Toom-Cook Polynomial Multiplication

Recent works propose a combination of the Karatsuba and Toom-Cook methods to perform the polynomial multiplication in Saber [DKRV18, DKRV19, MKV20]. Using the *four-way* Toom-Cook method, the product of a pair of 256-coefficient polynomials is split into seven polynomial multiplications with polynomials of length 64. These 64×64-coefficient multiplications are then further split into 16×16-coefficient multiplications using two levels of Karatsuba. After performing the recursive splittings, the polynomial length is small enough to efficiently perform the schoolbook multiplication. At a certain point, further splitting the polynomials does not bring any performance advantage since the savings derived from the multiplication do not outweigh the increasing number of additions. Similar to [DKRV18] and the reference implementation provided to NIST, in this work we stop the recursive splitting at a polynomial length of 16.

Since the coefficients in Saber are, similar to NewHope and Kyber, smaller than 16 bit, they are suitable for packed (vectorized) arithmetic. Although the ISA extension of the RISC-V specification[2] for packed arithmetic is still in draft mode, the RISC-V core used

---

[2]Document Version 20191213: https://riscv.org/specifications/isa-spec-pdf/

in this work already supports some packed operations [TGS19]. In the following, useful instructions for the polynomial multiplication in Saber are listed:

$pv.add.h$: $rd[15:0] = (rs1[15:0] + rs2[15:0]) \& 0xffff$
                        $rd[31:16] = (rs1[31:16] + rs2[31:16]) \& 0xffff$
$p.mulu$: $rd[31:0] = rs1[15:0] \cdot rs2[15:0]$
$p.mulhhu$: $rd[31:0] = rs1[31:15] \cdot rs2[31:15]$
$p.macuN$: $rd[31:0] = rs1[15:0] \cdot rs2[15:0] + rd) >> Is3$
$p.machhuN$: $rd[31:0] = rs1[31:16] \cdot rs2[31:16] + rd) >> Is3$

These instructions are comparable to the Cortex ARM-M4. Saber and also other Post-Quantum NIST candidates, such as some NTRU variants, use a modulo that is a power of two and $q \leq 2^{16}$. For this reason, it is possible to calculate and store two multiplications in parallel. Moreover, the schoolbook multiplication as well as the Karatsuba step in Eq. 7 can benefit from a Multiply Accumulate (MAC) function. Therefore, we develop a vectorized modulo multiply accumulate function, in the following denoted as **$pq.mac$**. The functionality of $pq.mac$ can be expressed with:

$pq.mac$: $rd[15:0] = (rs1[15:0] \cdot rs2[15:0] + rd[15:0]) \mod q'$
                $rd[31:16] = (rs1[31:16] \cdot rs2[31:16] + rd[31:16]) \mod q'$

In this work, the parameter q' in $pq.mac$ was set to 2^{16}. In this way, it is suitable for all schemes that use a power of two modulo smaller or equal to 2^{16}. After performing the polynomial multiplication, the result can be reduced with the original modulus q (simple masking operation) because $(a \mod q') \mod q \equiv a \mod q$ as long as both moduli are a power of two and $q' \geq q$. The hardware architecture for the $pq.mac$ operation is shown in Figure 4. Using the $pq.mac$ operation, the amount of clock cycles for the polynomial multiplication in Saber was reduced from $104,074$ to $71,349$.
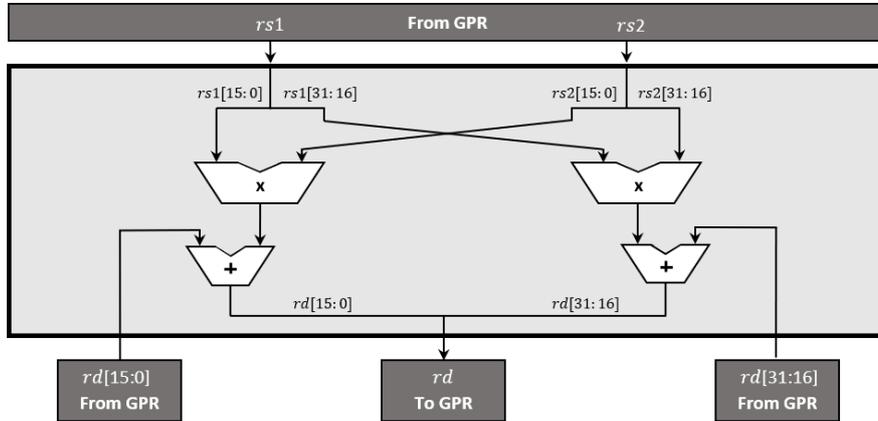

Figure 4: Vectorized Modulo Multiply Accumulate ($pq.mac$)

3.6   Pseudo Random Number Generation

Most Post-Quantum cryptosystems require a huge amount of randomness. This is particularly true for the generation of random polynomials in lattice-based cryptography. To produce this large amount of randomness, usually a small seed is expanded using a PRNG. Three primitives are specially suitable for this task: SHA-3, AES and ChaCha20. Among these three alternatives, SHA-3 is the most energy-efficient option because it generates the highest amount of pseudo-random bits per round [BUC19]. SHA-3 is a subset of the Keccak family standardized by NIST. The standard lists four specific instances of SHA-3

and two extendable-output functions (SHAKE128 and SHAKE256). While the SHA-3 functions have a specified output length, the two SHAKE variants permit extracting a variable length of output data, which makes it a suitable candidate for the pseudorandom bit generation. All instances are based on a 1600-bit state. This state can be represented in a three-dimensional array containing 25 words, each with a length of 64 bits. These words can be structured in a cube with x and y coordinates that are indexed from $0 \le x < 5$ and $0 \le y < 5$. Each bit of this cube can be addressed with $A[x, y, z]$. In order to facilitate the description of the applied functions, the following conventions are used: the part of the state which presents the word is also called lane; a two-dimensional part of the state with fixed z is called a slice; and all lanes with the same x-coordinate form a sheet.

The most important part of the SHA-3 and SHAKE primitives is the Keccak permutation function, which calls in each of in total 24 rounds the *f-1600* function. Each round is characterized by the five consecutive steps θ, ρ, π, χ and ι. These steps have a state array A as input and output B, a processed new state array.

Theta Step (θ): This step first computes the parity of each sheet of the state $C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$ for x in $[0, 4]$. The output of the θ step can then be computed with $B[x, y] = A[x, y] \oplus (C[x-1] \oplus rot(C[x+1], 1))$ for x and y in $[0, 4]$, where *rot* defines the rotation.

Rho/Pi Step (ρ/π): The ρ and π steps work on the lanes of the state $A[x, y]$. That is, they are usually processed together. The ρ step rotates the lane by a constant offset $r[x, y]$ which depends on the x and y positions. This can be formulated as $B[x, y] = rot(A[x, y], r[x, y])$, for x and y in $[0, 4]$. The π step swaps the complete lanes of the state according to $B[y, 2x + 3y] = A[x, y]$ for x and y in $[0,4]$.

Chi/Iota Step (χ/ι): The non-linear χ step can be computed according to $B[x, y] = A[x, y] \oplus (\overline{A}[x+1, y] \& A[x+2, y])$ for x and y in $[0,4]$. At the ι step, a round constant R_c is XORed to the lane $A[0, 0]$, i.e., $B[0, 0] = A[0, 0] \oplus R_c[i]$. This round constant depends on the current round i. Also the χ and ι step can be combined.

3.7   Hardware Accelerator − Pseudo Random Number Generation

The *Keccak team* presented three different optimized hardware implementations of Keccak: a high speed core, a mid-range core (trade-off between speed and area), and a low area coprocessor [BDH+20]. A description of these implementations can be found on their website[3]. The *high speed core* operates in a standalone fashion and requires no further resources for the Keccak calculations. Chunks of a message can be sent to the accelerator and the core will output the hash value. This approach was also used in [FSMG+19]. The authors connected a loosely coupled high performance Keccak accelerator to the AHB in order to accelerate the Keccak operations in NewHope. The *low area* solution (of the Keccak team) is a coprocessor which uses the system memory as data storage instead of storing the Keccak state internally. Only temporary results are kept internally in registers. As a trade-off between the high speed and low area solution, the Keccak team presented the *mid-range core* which is based on some of the ideas presented in [JA11]. This core rearranges the order of the permutation function such that the two slice oriented steps χ and θ are processed closer together. In order to reduce the required area, the core only works on a subset of the slices that compose the state. This approach requires a different initial and final round. However, as the ρ, π and ι steps work on lanes and the χ and θ steps on slices, it is not the optimal solution to split the state as the state must be loaded

---

[3]https://keccak.team/files/Keccak-implementation-3.2.pdf

multiple times from the memory.

In this work, we develop an alternative solution for Keccak which presents a trade-off between performance and area. It can be classified between the high speed and low area Keccak solutions of the *Keccak team*. To avoid a high access rate to the main memory, the state is not split into multiple parts. However, instead of a standalone loosely coupled Keccak accelerator, we design a hardware accelerator for a single round of the Keccak permutation. To reuse existing resources from the RISC-V core, the complete Floating Point Register Set (FPR) with 32×32 bit registers and a part of the General Purpose Register Set (GPR) with 18×32 bits are used. In the following, this combination of FPR and GPR is denoted as Post-Quantum Register Set (PQR). To be precise, the temporary registers $t0$ to $t6$ and the saved registers $s1$ to $s11$ are used from the GPR. In this way, enough remaining registers exist in the GPR to guarantee a normal operation of the RISC-V core and also enough registers are used to store the complete Keccak state in the PQR. The saved registers have to be stored on the stack before the Keccak function starts. Similar to the *NTT and Modular Arithmetic Unit*, the registers can be accessed in parallel.

Figure 5 illustrates our Keccak hardware accelerator. The input of this accelerator is the content of the PQR, the current round for determining the round constant, and a start and reset signal. Triggered by the start signal, the accelerator will perform one round of the *f-1600* function, where the round signal selects the corresponding round constant. The result of the processed state is written back to the PQR. This step can be repeated for all 24 rounds. The permuted state can then be stored from the PQR to the main memory.

To reduce the memory access during the generation of random polynomials, we keep the state in the registers as long as possible. The following steps are used for the generation of random polynomials. First, the state is set to zero with the *keccak_rst* signal and written into the PQR, which also resets the related registers. After setting all registers to zero, the Keccak absorption phase begins. During this phase, the input message or a message block is written into a subset of the state. The state permutation will then transform this state. Depending on the rate, a certain number of bits is squeezed out while a part of the state, the capacity, remains untouched. The squeezed output is then processed in order to generate the desired random polynomials. Thereby, the state registers must remain untouched, when the state is not written back to the memory. To obtain fresh randomness, the state is permuted and squeezed again. Keeping the state for the whole polynomial sampling process within the PQR leads to a significant performance improvement as the access to the main memory is eliminated in this case.
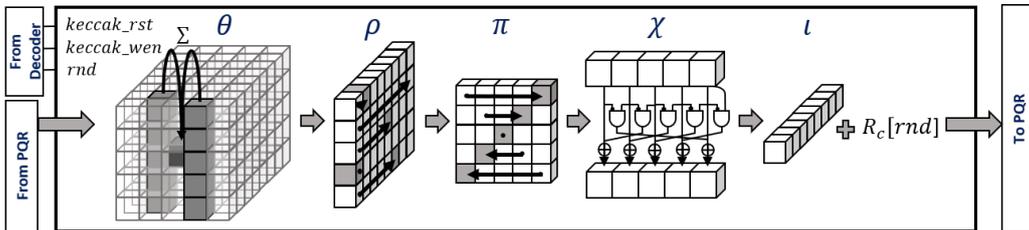

Figure 5: Keccak Accelerator

3.8 Binomial Sampling

Many LWE-based schemes, such as NewHope, Kyber and Saber, replaced the discrete Gaussian error distribution by a centered binomial distribution. This significantly increases the efficiency, avoids complex arithmetic or large LUTs, and offers better protec-

tion against side-channel attacks. Let Ψ_k be a binomial distribution which is centered at zero and has a standard deviation of $\sigma = \sqrt{k/2}$. The distribution is determined by $\Psi_k = \sum_{i=0}^{k-1}(b_i - b_i') \mod q$, where $b_i, b_i' \in \{0,1\}$ are uniform independent bits. This is similar to taking two k-bit integers b and b', calculating the respective Hamming weight and subtracting one Hamming weight from the other (modulo subtraction).

Figure 6 shows the developed hardware architecture of the *Binomial Sampling Unit*, which turns uniformly distributed samples into binomially distributed ones. The *Binomial Sampling Unit* reads the uniform samples contained in the two registers $rs1$ and $rs2$ from the *GPR* and calculates the result rd. As the modulus is smaller than 16 bit for the considered schemes, the output register can concatenate two samples. The *Binomial Sampling Unit* supports different parameters of k: for NewHope $k = 8$ [AAB+19], for Kyber $k = 2$ [ABD+19], for Lightsaber $k = 5$, and for Firesaber $k = 3$ [DKRV19]. Depending on the *mode* signal, the multiplexers forward two sums of k bits to the modulo subtractors. The two output samples of the subtractors will be combined and written back to the *GPR*.
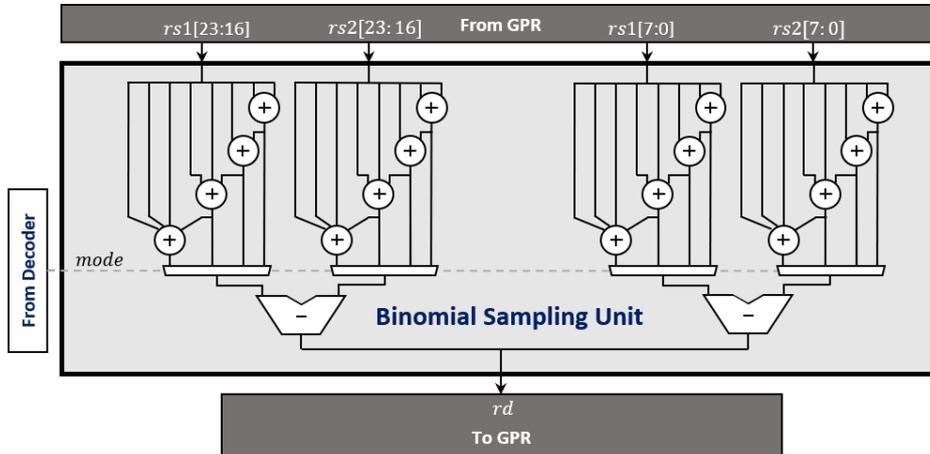

Figure 6: Binomial Sampling Unit

3.9 Results for Keccak and Polynomial Sampling

Table 3 summarizes the clock cycle count of the SHAKE256 implementation, the uniform sampling, and the binomial sampling. While the uniform rejection sampling is used in LWE-based schemes to generate the random public polynomial, the binomial sampling is used to generate the secret and error polynomials for R-LWE/M-LWE schemes. For M-LWR schemes, the binomial sampling is only used for generating the secret polynomials. The results show that the SHAKE256 function was accelerated by a factor of 103.59 in comparison to the software baseline implementation. Also compared to the baseline implementation, the uniform sampling was accelerated by a factor of 26.90 for NewHope-512 ($n = 512$), 27.12 for NewHope-1024 ($n = 1024$), 22.37 for Kyber-512 ($n = 256$ for all Kyber security levels), and 31.74 for Lightsaber ($n = 256$ for all Saber security levels). The optimized binomial sampling makes use of the fast *Keccak Accelerator* described in Section 3.7 as well as the *Binomial Sampling Unit* described in Section 3.8. With these optimizations, we achieve a speedup factor of 35.69, 35.80, 14.27, and 40.15 for NewHope-512, NewHope-1024, Kyber-512, and Lightsaber, respectively. The speedup factor for Kyber is smaller when compared to NewHope or Lightsaber due to the low variance of the error distribution.

Although, we aimed for a mid-range Keccak solution, the results show that our design outperforms the uniform and binomial sampling of [FSMG+19], which uses a loosely

coupled standalone high performance Keccak implementation. Their architecture is able to calculate two Keccak rounds in one clock cycle. However, the large communication overhead poses a substantial drawback of their architecture. In this work, the communication overhead was nearly eliminated. During the Keccak absorption phase the message initializes the state. Afterwards, the Keccak state is hold in the registers for the complete sampling process. The required hardware costs are summarized in Section 5.

Table 3: Cycle Count of SHAKE256 (32 Byte input/output length), uniform sampling and binomial sampling. Kyber and Saber have for all security strengths the same polynomial length. The results for *Poly Uniform* and *Poly Binomial* are given with respect to generating one polynomial.

	SHAKE256	Poly Uniform	Poly Binomial
NewHope-512 cca baseline	$31,907$	$272,615$	$280,079$
NewHope-512 cca opt.	**308**	**10,136**	**7,847**
NewHope-1024 cpa [FSMG+19]	–	$42,050$	$75,682$
NewHope-1024 cca baseline	$31,907$	$548,019$	$560,058$
NewHope-1024 cca opt.	**308**	**20,205**	**15,643**
Kyber-512 cca baseline ($n=256$)	$31,907$	$501,300$	$33,902$
Kyber-512 cca opt. ($n=256$)	**308**	**22,414**	**2,375**
Lightsaber cca baseline ($n=256$)	–	$306,387$	$130,896$
Lightsaber cca opt. ($n=256$)	–	**9,652**	**3,260**

4 Integration of Accelerators into RISC-V Platform and ISA Extension

RISC-V is an open Instruction Set Architecture (ISA) based on the concepts of the Reduced Instruction Set Computer (RISC) principles. The RISC-V initiative started in 2010 by the University of California, Berkley and has meanwhile grown to a large non-profit corporation. RISC-V provides a free, open, flexible, and extensible ISA usable for embedded systems as well as high performance computers. Several open-source hardware implementations supporting the RISC-V ISA exist. Implementations which have drawn particular attention are Rocket Chip[4], VexRiscv[5], and the RISC-V cores from PULP[6].

Rocket Chip was constructed using the hardware construction language Chisel[7]. The implementation offers a dedicated interface, called Rocket Custom Coprocessor (RCC), to extend the system with hardware accelerators. However, the presented design does not allow to easily change the pipeline stage and is therefore not suitable for the development of tightly coupled accelerators. VexRiscv was developed using another high level hardware description language called SpinalHDL[8]. The VexRiscv project allows modifications of the pipeline stage. This platform was used in [AEL+20] to integrate a tightly coupled finite field multiplier and in [WTJ+20] for developing loosely coupled qTESLA accelerators. The PULP project features three different RISC-V cores designed using the hardware description language SystemVerilog. The Ariane core is a 6-stages 64-bit solution. For smaller embedded devices, the PULP team offers the 2-stages 32-bit solution Ibex (formerly Zero-riscy) and the 4-stages 32-bit solution CV32E40P (formerly RI5CY). The RISC-V cores CV32E40P and Ibex can be integrated into the single-core

---

[4] https://github.com/chipsalliance/rocket-chip
[5] https://github.com/SpinalHDL/VexRiscv
[6] https://github.com/pulp-platform
[7] https://www.chisel-lang.org
[8] https://github.com/SpinalHDL

micro-controller platform PULPino[9] offering a rich set of peripherals such as I2C, SPI, UART, and GPIO.

Similar to [FSMG$^+$19], we decided to use the CV32E40P core and integrated this core into the PULPino platform. As this core and platform is written in SytemVerilog, we have the full control over the whole architecture, which makes it ideally suitable for our modifications of the core and pipeline stage. The CV32E40P core has a comparable performance to the widely deployed ARM-Cortex M4, but it is a little slower.

4.1   Integration of PQ Hardware Accelerators

Figure 7 shows the architecture of the CV32E40P core with Post-Quantum extensions, which we name **RISQ-V**. The CV32E40P processor has a four stages in-order pipeline. The main components are a Prefetch Buffer, an Instruction Decoder, a General Purpose Register Bank (GPR), a Floating Point Register Bank (FPR), an Arithmetic Logic Unit (ALU), a Multiplication Unit (MULT), a Control and Status Register Unit (CSR), and a Load-Store Unit (LSU). The core is extended by two completely new components: the *PQR-ALU* and the *PQ-ALU*.

The *PQR-ALU* consists of the *NTT and Modular Arithmetic Unit* and the *Keccak Accelerator*. These two modules require parallel access to the register banks. Therefore, the *PQR-ALU* is directly located within the *Decoding Stage*. This avoids to route the register signals to the *Execution Stage*. The *PQ-ALU* contains the *Binomial Sampling Unit*. This accelerator requires only two input and one output register and has a similar construction like the *MULT* unit and the *ALU*. To reuse existing hardware resources, we integrate the *pq.mac* operation, described in Section 3.5, directly into the *MULT* unit. The hardware resources for performing the multiplications are already available in the *MULT* unit and an extension for the *pq.mac* support comes with a negligible overhead of multiplexers and two additions. Enhancing or downsizing our developed *PQR-ALU* and *PQ-ALU* is straightforward. All accelerators are added as modules and can be selected using dedicated define directives. Thus the accelerators can be selected according to the application requirements.
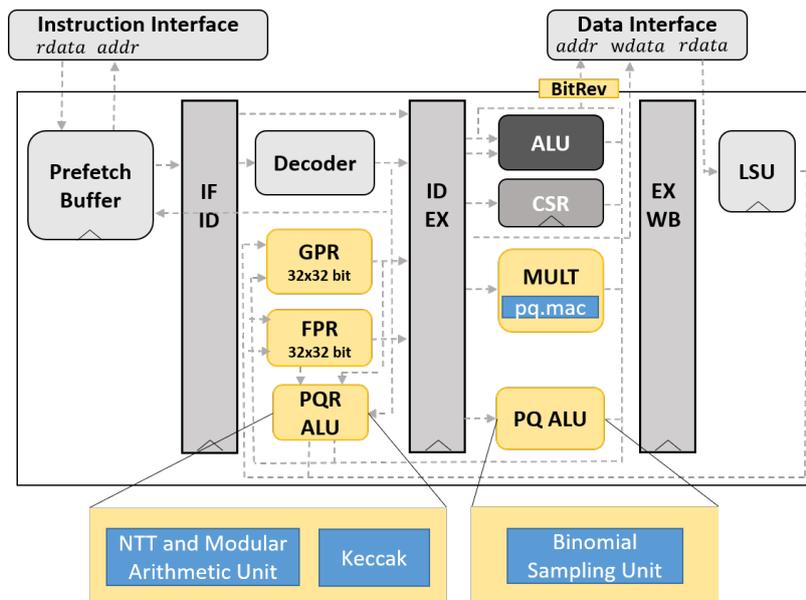

Figure 7: RISQ-V

---

[9]https://github.com/pulp-platform/pulpino

4.2 RISC-V ISA Extension

To enhance the basic integer instruction set (**I**), RISC-V defines several standard extensions, including the extensions for multiplication/division (**M**); single, double, and quad precision floating point operations (**F**, **D**, **Q**); atomic operations (**A**); and compressed instructions (**C**).

The CV32E40P core fully supports the **I** instruction set and the extensions **M**, **F**, and **C**. In addition, this core provides the PULP specific extension **Xpulp**, which includes hardware loops, SIMD extensions, bit manipulation and post-increment instructions. In this work, we develop the **PQ** extension. Figure 8 shows the RISC-V base instruction format types *R*, *I*, *S*, and *U*. Depending on the instruction type, the instruction structure consists of an *opcode*, function fields, immediate values, the source registers $rs1$ and $rs2$, and the destination register rd. We decided to use the *R-type* for all Post-Quantum instructions. This allows the use of multiple operations with only a single opcode. The *opcode* chosen in this work is the unused value $0x77$.
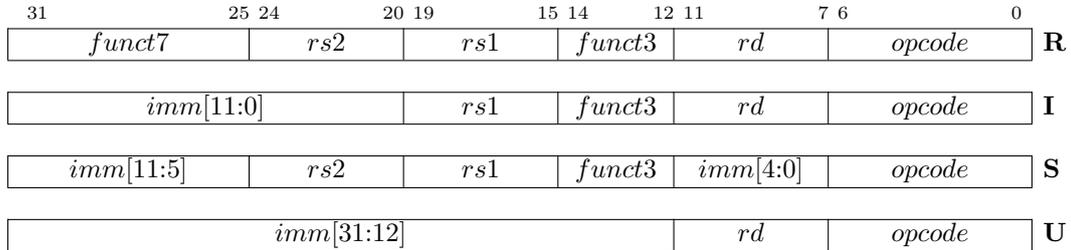
31	25	24	20	19	15	14	12	11	7	6	0	
$funct7$		$rs2$		$rs1$		$funct3$		rd		$opcode$		**R**

$imm[11:0]$	$rs1$	$funct3$	rd	$opcode$	**I**

$imm[11:5]$	$rs2$	$rs1$	$funct3$	$imm[4:0]$	$opcode$	**S**

$imm[31:12]$	rd	$opcode$	**U**

Figure 8: Base RISC-V instruction formats

The developed Post-Quantum instructions can be divided into seven main classes: *NTT Configuration*, *NTT Operation*, *Modular Arithmetic Unit*, *Bit-Reversal*, *PQ-MAC*, *Keccak operation*, and *Binomial Sampling*. All instructions which do not require $rs1$, $rs2$, or rd can use the value zero for the unused position. A full list of all Post-Quantum instructions is provided in Table 9 (Appendix C).

NTT Configuration Class: This class performs the following functionalities: i) setting the selected scheme: NewHope-512, NewHope-1024, or Kyber (all security categories); ii) setting the forward or inverse NTT; and iii) setting either the first rounds or the last round of the NTT. When NewHope-512 or NewHope-1024 is selected, the parameters for the optimized $NTT_{br\leftarrow no}^{CT}$ calculation will be set accordingly. Moreover, selecting one of the NewHope variants configures the modulus to $q = 12289$ and the Montgomery parameter for the modulo multipliers to $-q^{-1} \mod R = 12287$ with $R = 2^{18}$ for all schemes. When Kyber is selected the modulo parameters are configured to $q = 3329$ and $-q^{-1} \mod R = 199935$. Setting the forward or inverse NTT will select the corresponding precomputed values for ω_m of the *Twiddle Update Unit*. Finally, the selection of the first rounds or last round will affect the *Reordering MUX* of the *Modular Arithmetic Unit*, more specifically, the swapping of the coefficients.

NTT Operation Class: This class contains all instructions for the optimized $NTT_{br\leftarrow no}^{CT}$ calculation. The *pq.ntt_multiple_bf* instruction triggers the automatic calculation of the first five merged NTT rounds. The *pq.ntt_single_bf* instruction is used to calculate two parallel decimation-in-time butterfly operations. The instructions *pq.update_m* ($rs1$ is used for the *index*) and *pq.update_omega* are used to control the *Twiddle Update Unit*. The *pq.mul_gamma*1, *pq.mul_gamma*2, and *pq.update_gamma* instructions are used

for calculating the scaling by n^{-1} and γ^{-i} within the inverse NTT.

Modulo Arithmetic Operation Class: This class is used for the vectorized modulo multiplication, addition, subtraction, and the butterfly operations decimation-in-time and decimation-in-frequency. In this context, $rs1$ and $rs2$ are used as source registers and rd as destination register (for the butterfly instructions $rs1$ is also used for the second output). In contrast to the $pq.ntt\_single\_bf$ operation, which is used for the optimized $NTT_{br\leftarrow no}^{CT}$, the $pq.bf\_dit$ operation does not use the registers from the *FPR* but from the *GPR*.

Bit-Reversal Class: This class contains the bit-reversal instructions for the polynomial lengths $n = 256$, $n = 512$, and $n = 1024$. The register rd contains the value that has to be stored, $rs1$ the base address of the store location, and $rs2$ the value of the offset. The offset value will be added to the base address in bit-reversed order. The *Load Store Unit* will then store the coefficient to the desired location.

PQ MAC Class: This class contains the vectorized modulo multiply accumulate function described in Section 2.2.

Keccak Operation Class: This class is used to perform one round of the Keccak permutation. The $keccak\_wen$ signal, described in Section 3.7 is set to one when the $pq.keccak\_f1600$ instruction is executed. In this context, $rs1$ selects the current Keccak round and $rs2$ is used to reset the state.

Binomial Sampling Class: This class is used to turn uniform samples in $rs1$ and $rs2$ into binomially distributed coefficients in rd.

5 Experimental Results

This section presents our experimental results for NewHope, Kyber and Saber. For all measurements, the Chosen Ciphertext Attacks (CCA) variant of the schemes was chosen and for all schemes the lowest and highest NIST security category was evaluated. The software baseline measurements use the clean C-code versions of the PQ-M4 project in [KRSS19], which are based on the reference implementations in [AAB+19], [ABD+19], and [DKRV19].

5.1 Cycle Count

The results of our clock cycle benchmarks are summarized in Table 4. In comparison to the software baseline implementation, a speedup factor of 10.07 (NewHope-512), 10.47 (NewHope-1024), 7.68 (Kyber-512), 9.62 (Kyber-1024), 2.65 (Lightsaber), and 2.48 (Firesaber) for a complete algorithm run was achieved. Although the *NTT and Modular Arithmetic Unit* was not particularly designed for the NTT types used in Kyber, a significant performance improvement was measured. The second round Kyber submission chooses a prime q for which the condition $q \equiv 1 \mod 2n$ does not hold and an early termination of the NTT is required. This reduces the cost for the NTT, but a so called basecase multiplication consisting of 128 products is necessary. For this basecase multiplication, the vectorized modulo arithmetic of the *Modular Arithmetic Unit* was exploited.

Besides that, our work beats the cycle count of the latest assembler optimized ARM Cortex-M4 implementations of NewHope and Kyber in [ABCG20, KRSS19] and the RISC-V VexRiscv implementation in [AEL+20]. Using more powerful accelerators than only the basic finite field multiplier as in [AEL+20] is paying off: in comparison to their work a

speedup factor of 6.51 (NewHope-512), 6.53 (NewHope-1024), 4.65 (Kyber-512), and 6.15 (Kyber-1024) was achieved. Furthermore, our design is faster than the CPA version of the NewHope-1024 implementation in [FSMG$^+$19], which uses loosely coupled accelerators, although the CPA version has no costly re-encryption step ($1,078,695$ vs. $1,113,984$ cycles). The better performance can be explained by the reduced communication overhead due to the tight coupling of the NTT and Keccak accelerators to the register banks and by the additional usage the new accelerators developed in this work (*Modular Arithmetic Unit* and *Binomial Sampling Unit*). For the Saber instances the polynomial multiplication remains the performance bottleneck, although the *pq.mac* operation already brings a considerable improvement. Just recently, the authors in [MKV20] proposed a technique called *lazy interpolation* in order to accelerate the evaluation and interpolation phase during the Toom-Cook multiplication. The integration of this optimization technique into our approach would probably result in a further improvement. This has been left as future work.

Just recently, the authors of NewHope announced in the NIST PQC Forum[10] that a new reference code is available. This new version appeared as a response of the work presented in [BDG20], which exploits the incorrect oracle cloning of some NIST KEM PQC candidates to perform key-recovery attacks. The new version of NewHope includes a domain separation for the SHAKE calls in order to make each hash call independent. That is, all hash calls with the same input size use a domain separator label (e.g., a nonce). It is estimated by the *NewHope team* that this modification will have a negligible influence on the performance results of NewHope. RISQ-V is able to support efficiently the new version of NewHope. The absorption of the input message is completely controlled by software and can be easily modified. The integration of a complete domain separation has been left as future work.

Table 5 summarizes the measured code size for both the baseline and the optimized implementations. In particular for the optimized NewHope implementations, the code size was significantly decreased compared to the baseline implementation. This is mainly due to the fact that the large LUTs for the Twiddle factors and bit-reversal were eliminated. Moreover, the ISA extensions have the side effect that fewer instructions for complex operations are required. It should also be noted that the Saber implementations in [KRSS19] have a significantly larger memory consumption. This is caused by the fact that the authors developed a tool for the automatic generation of an optimized assembler code for the polynomial multiplication in Saber. While the optimized assembler code leads to a very fast polynomial multiplication, the code size is large (nearly 10,000 lines[11]).

5.2 FPGA Results

RISQ-V can be synthesized for FPGAs as well as ASICs. For the FPGA evaluation, the Xilinx Zynq-7000 programmable SoC was chosen. The resource utilization of the complete *RISQ-V* implementation and the costs for the single accelerators are provided in Table 6. In total, the circuit size of RISQ-V is 9,210 LUTs and 1,261 registers larger than the circuit size of the original PULPino platform. For this comparison, we omitted the *FPU* of the original PULPino platform as it is not necessarily required for the considered Post-Quantum algorithms.

Compared to the loosely coupled NTT accelerator in [FSMG$^+$19], our accelerator has a higher number of LUTs, but a lower number of registers. The higher number of LUTs can be explained by the higher flexibility of the *NTT and Modular Arithmetic Unit*. Instead of only being capable of calculating the decimation-in-time butterfly operation, the proposed architecture also supports parallel calculations of the decimation-in-frequency butterfly

---

[10]https://groups.google.com/a/list.nist.gov/forum/#!forum/pqc-forum
[11]https://github.com/mupq/pqm4/blob/master/crypto_kem/lightsaber/m4

Table 4: Cycle count (O3, riscv32-unknown-elf-gcc 7.1.1 20170509, PULPino)

Algorithm	Device	Key gen.	Encaps.	Decaps.
NewHope-512 cca [ABCG20]	ARM Cortex-M4	578, 890	858, 982	806, 300
NewHope-512 cca [AEL+20]	RISC-V (VexRiscv)	904, 000	1, 424, 000	1, 302, 000
NewHope-512 cca baseline	RISC-V (PULPino)	1, 370, 735	2, 153, 075	2, 091, 823
NewHope-512 cca opt.	**RISC-V (PULPino)**	**123, 860**	**207, 299**	**226, 742**
NewHope-1024 cpa [FSMG+19]	RISC-V (PULPino)	357, 052 [a]	589, 285 [a]	167, 647 [a]
NewHope-1024 cca [KRSS19]	ARM Cortex-M4	1, 219, 908	1, 903, 231	1, 927, 505
NewHope-1024 cca [ABCG20]	ARM Cortex-M4	1, 157, 222	1, 674, 899	1, 587, 107
NewHope-1024 cca [AEL+20]	RISC-V (VexRiscv)	1, 776, 000	2, 742, 000	2, 528, 000
NewHope-1024 cca baseline	RISC-V (PULPino)	2, 767, 270	4, 282, 504	4, 239, 534
NewHope-1024 cca opt.	**RISC-V (PULPino)**	**235, 420**	**392, 734**	**450, 541**
Kyber-512 cca [KRSS19]	ARM Cortex-M4	514, 291	652, 769	621, 245
Kyber-512 cca [ABCG20]	ARM Cortex-M4	455, 191	586, 334	543, 500
Kyber-512 cca [AEL+20]	RISC-V (VexRiscv)	710, 000	971, 000	870, 000
Kyber-512 cca baseline	RISC-V (PULPino)	1, 137, 052	1, 547, 789	1, 525, 621
Kyber-512 cca opt.	**RISC-V (PULPino)**	**150, 106**	**193, 076**	**204, 843**
Kyber-1024 cca [KRSS19]	ARM Cortex-M4	1, 575, 052	1, 779, 848	1, 709, 348
Kyber-1024 cca [ABCG20]	ARM Cortex-M4	1, 404, 695	1, 605, 707	1, 525, 805
Kyber-1024 cca [AEL+20]	RISC-V (VexRiscv)	2, 203, 000	2, 619, 000	2, 429, 000
Kyber-1024 cca baseline	RISC-V (PULPino)	3, 378, 603	4, 024, 887	3, 949, 039
Kyber-1024 cca opt.	**RISC-V (PULPino)**	**349, 673**	**405, 477**	**424, 682**
Lightsaber cca [MKV20]	ARM Cortex-M4	466, 000	653, 000	678, 000
Lightsaber cca [KRSS19]	ARM Cortex-M4	459, 965	651, 273	678, 810
Lightsaber cca baseline	RISC-V (PULPino)	1, 071, 836	1, 503, 594	1, 537, 939
Lightsaber cca opt.	**RISC-V (PULPino)**	**366, 837**	**526, 496**	**657, 583**
Firesaber cca [MKV20]	ARM Cortex-M4	1, 340, 000	1, 642, 000	1, 679, 000
Firesaber cca [KRSS19]	ARM Cortex-M4	1, 448, 776	1, 786, 930	1, 853, 339
Firesaber cca baseline	RISC-V (PULPino)	3, 427, 099	4, 215, 630	4, 328, 885
Firesaber cca opt.	**RISC-V (PULPino)**	**1, 300, 272**	**1, 622, 818**	**1, 898, 051**

[a] Cycle count was only reported for the CPA-secure version. Due to the missing re-encryption step, the CPA version is significantly faster during the decapsulation compared to the CCA-secure versions.

Table 5: Code size in bytes

Algorithm	Device	Code Size
NewHope-512 cca [KRSS19]	ARM Cortex-M4	11, 000
NewHope-512 cca baseline	RISC-V (PULPino)	17, 658
NewHope-512 cca opt.	**RISC-V (PULPino)**	**9, 626**
NewHope-1024 cca [KRSS19]	ARM Cortex-M4	12, 176
NewHope-1024 cca baseline	RISC-V (PULPino)	21, 548
NewHope-1024 cca opt	**RISC-V (PULPino)**	**10, 816**
Kyber-512 cca [KRSS19]	ARM Cortex-M4	11, 000
Kyber-512 cca baseline	RISC-V (PULPino)	16, 928
Kyber-512 cca opt.	**RISC-V (PULPino)**	**12, 532**
Kyber-1024 cca [KRSS19]	ARM Cortex-M4	12, 424
Kyber-1024 cca baseline	RISC-V (PULPino)	17, 670
Kyber-1024 cca opt.	**RISC-V (PULPino)**	**12, 874**
Lightsaber cca [KRSS19]	ARM Cortex-M4	44, 916
Lightsaber cca baseline	RISC-V (PULPino)	18, 772
Lightsaber cca opt.	**RISC-V (PULPino)**	**12, 544**
Firesaber cca [KRSS19]	ARM Cortex-M4	44, 184
Firesaber cca baseline	RISC-V (PULPino)	17, 794
Firesaber cca opt.	**RISC-V (PULPino)**	**11, 680**

operation and packed modular arithmetic. It also has to be noted that our design does not need any further BRAM block for storing the input and output data as well as storing the Twiddle factors. Instead of hiding the post-processing step (scaling by $n^{-1}\gamma^{-i}$) by using extra multipliers, we use the same multipliers for this operation as for the butterfly calculation.

The tightly coupled Keccak implementation only uses combinatorial logic. No further registers are used in this design as the state is stored in the *FPR* and *GPR*. The tight coupling saves all logic and registers that are used for buffering the input and output data, the Keccak absorption and squeezing phase. In contrast to the Keccak implementation in [FSMG+19], we decided to use a variant which calculates one round of the permutation function per clock cycle instead of two. This is significantly faster than the low-area solution of the *Keccak Team* and is comparable to their high speed solution [BDH+20].

The two hardware accelerators of the *Execution Stage* have a negligible hardware overhead. While the *Binomial Sampling Unit* only requires 106 LUTs, the overhead for supporting the *pq.mac* operation was nearly eliminated by reusing the resources of the *MULT* unit.

Table 6: Resource Utilization FPGA

Complete Cores				
	LUTs	**Registers**	**BRAMs**	**DSPs**
PULPino orig. (w/o FPU)	14,715	9,583	32	8
RISQ-V	**23,925**	**10,844**	**32**	**21**

Single Accelerators				
	LUTs	**Registers**	**BRAMs**	**DSPs**
NTT accelerator [FSMG+19]	886	618	1	26
NTT and Modular Arithmetic Unit (PQR-ALU)	**3,098**	**173**	**0**	**9**
Keccak accelerator [FSMG+19], 0.5 cycle/round	10,435	4,225	0	0
Keccak low area[a)] [BDH+20], 375 cycle/round	1,159	236	0	0
Keccak high speed[a)] [BDH+20], 1 cycle/round	4,189	2,641	0	0
Keccak (PQR-ALU), 1 cycle/round	**3,847**	**0**	**0**	**0**
Binom. Sampling Unit (PQ-ALU)	**106**	**0**	**0**	**0**
Mult. Unit with PQ-MAC	**312**	**6**	**0**	**5**

[a)] VHDL-Design of [BDH+20] was synthesized for the same FPGA (Xilinx Zynq-7000). Resources do not include the costs for the connection to the system bus.

5.3 ASIC Results

We synthesized the ASIC design with the UMC 65 nm technology. The main objective of this work was to achieve a low energy design. Therefore, a low leakage standard cell library with high threshold voltage was chosen. Table 7 shows the amount of logic cells and the area consumption of the original PULPino and *RISQ-V*. When both designs are compared, an increase of logic cells for *RISQ-V* can be observed. The cell area was increased by $64,108\,\mu m^2$ for the combinatorial logic and by $9,927\,\mu m^2$ for the sequential logic. However, this increase does not have a large impact on the overall area because the memory is by far the largest part for both designs. The maximum clock frequency was reduced from 79.66 MHz to 45.26 MHz as the *Modular Arithmetic Unit* has a relatively long critical path. A solution to break this critical path could be to add pipeline registers within the modulo multipliers. However, this would increase the latency by one cycle. As the achieved frequency is acceptable for most embedded applications, we omit these registers.

Table 7: Area ASIC Synthesis (UMC 65 nm)

	Cell Count	Cell Area Combinatorial $[\mu m^2]$	Cell Area Sequential $[\mu m^2]$	Cell Area Memory $[\mu m^2]$
PULPino orig. (w/o FPU)	36,173	78,676	92,304	669,346
RISQ-V	**57,428**	**142,784**	**102,231**	**669,346**

Our simulated measurements for the power and energy consumption were performed at a frequency of 10 MHz, a nominal supply voltage of 1.2 V, and a temperature of 25°C. To obtain realistic results, we extracted the dynamic post-synthesis power consumption by means of gate level switching activity files. The Switching Activity Interchange Format (SAIF) file for the dynamic power calculation was generated using Cadence's Incisive Enterprise Simulator and the power consumption was calculated using the Cadence power analyzer Joules. The results regarding the power measurements of the original PULPino and our developed RISQ-V implementation are summarized in Table 8. The leakage power belongs to the category of static power consumption. Due to the higher area consumption, the leakage power is marginally higher for RISQ-V compared to the original PULPino. An interesting aspect is that the total power consumption is mainly dependent on the applied scheme but not on the security level. For instance, the optimized NewHope-512 implementation has a power consumption of $2.42\,mW$ whereas the optimized NewHope-1024 implementation has a power consumption of $2.41\,mW$. This can be explained by the fact that both instances basically use the same operations and only the execution time differs. Due to the shorter execution time, the energy consumption is significantly lower when using our tightly coupled accelerators.

Table 8: Power and energy results for the ASIC synthesis (UMC 65 nm)

	Leackage	Internal	Switching	Tot. Power	Cycles	Energy $[\mu J]$
NewHope-512 cca (baseline)	3.00e-04 W	1.35e-03 W	3.87e-04 W	2.04e-03 W	5,615,725	1,145.61
NewHope-512 cca (opt.)	3.08e-04 W	1.50e-03 W	6.08e-04 W	2.42e-03 W	557,995	**135.03**
NewHope-1024 cca (baseline)	3.00e-04 W	1.34e-03 W	3.85e-04 W	2.02e-03 W	11,289,402	2,280.46
NewHope-1024 cca (opt.)	3.10e-04 W	1.49e-03 W	6.10e-04 W	2.41e-03 W	1,078,787	**259.98**
KYBER-512 cca (baseline)	2.96e-04 W	1.38e-03 W	3.96e-04 W	2.07e-03 W	4,210,556	871.59
Kyber-512 cca (opt.)	2.98e-04 W	1.59e-03 W	6.93e-04 W	2.58e-03 W	548,119	**141.41**
KYBER-1024 cca (baseline)	2.90e-04 W	1.39e-03 W	3.95e-04 W	2.08e-03 W	11,352,622	2,361.35
Kyber-1024 cca (opt.)	2.93e-04 W	1.60e-03 W	7.09e-04 W	2.60e-03 W	1,183,372	**307.68**
Lightsaber cca (baseline)	2.98e-04 W	1.48e-03 W	3.85e-04 W	2.16e-03 W	4,113,463	888.51
Lightsaber cca (opt.)	3.09e-04 W	1.78e-03 W	6.89e-04 W	2.78e-03 W	1,551,010	**431.18**
Firesaber cca (baseline)	2.94e-04 W	1.48e-03 W	3.83e-04 W	2.16e-03 W	11,971,708	2,585.80
Firesaber cca (opt.)	3.01e-04 W	1.79e-03 W	6.83e-04 W	2.77e-03 W	4,821,233	**1,335.48**

6   Conclusion

The generation of uniformly and binomially distributed random polynomials and the polynomial arithmetic are the performance bottlenecks of lattice-based cryptography. Previous works developed loosely coupled accelerators to improve the performance characteristics of several lattice-based schemes. However, loosely coupled accelerators usually have a high data transfer overhead, require a high amount of hardware resources, and suffer from a low flexibility. In this work we developed RISQ-V, an enhanced RISC-V architecture that integrates powerful tightly coupled accelerators directly into the processing pipeline to speed up lattice-based cryptography. The accelerators include an arithmetic unit for vectorized modulo arithmetic and NTT operations, a vectorized modulo multiply accumulate unit, a Keccak accelerator for the pseudo-random bit generation, and a binomial sampling unit for the generation of binomially distributed samples. To control the tightly coupled accelerators, we extended the RISC-V ISA and developed 28 Post-Quantum instructions. The design strategy of this work was to reuse existing hardware resources

of the system processor, such as the register banks and multipliers, in order to keep the area footprint small. Moreover, we developed design strategies to decrease the access rates to the system memory during the NTT and Keccak computations. Holding data as long as possible within the processor's registers resulted in a significant performance improvement. RISQ-V was synthesized for an FPGA prototype and an ASIC. Compared to the baseline software implementation, the performance evaluation has shown that the developed tightly coupled accelerators lead to a significant reduction of the clock cycle count and energy consumption for NewHope, Kyber, and Saber.

References

[AAB+19]   Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. NewHope: Algorithm Specifcations and Supporting Documentation, 2019. https://newhopecrypto.org/data/NewHope_2019_07_10.pdf.

[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: Algorithm Specifcations and Supporting Documentation (v2), 2019. https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf.

[AD17]   Martin R. Albrecht and Amit Deo. Large modulus ring-lwe $\geq$ module-lwe. Cryptology ePrint Archive, Report 2017/612, 2017. https://eprint.iacr.org/2017/612.

[AEL+20]   Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. Isa extensions for finite field arithmetic - accelerating kyber and newhope on risc-v. Cryptology ePrint Archive, Report 2020/049, 2020. https://eprint.iacr.org/2020/049.

[AJS16]   Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. Newhope on arm cortex-m. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 332–349. Springer, 2016.

[BDG20]   Mihir Bellare, Hannah Davis, and Felix Günther. Separate your domains: Nist pqc kems, oracle cloning and read-only indifferentiability. Cryptology ePrint Archive, Report 2020/241, 2020. https://eprint.iacr.org/2020/241.

[BDH+20]   Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, and Gilles Van Assche. Keccak in VHDL, 2020. https://keccak.team/hardware.html.

[BKS19]   Leon Botros, Matthias J Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of kyber on cortex-m4. In *International Conference on Cryptology in Africa*, pages 209–228. Springer, 2019.

[BPR12]   Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 719–737, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BUC19]   Utsav Banerjee, Tenzin Ukyab, and Anantha Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61, Aug. 2019.

[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.

[DB16]   Chaohui Du and Guoqiang Bai. Towards efficient polynomial multiplication for lattice-based cryptography. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1178–1181. IEEE, 2016.

[DKRV18]   Jan-Pieter DAnvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *International Conference on Cryptology in Africa*, pages 282–305. Springer, 2018.

[DKRV19]   Jan-Pieter DAnvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 2 Submission), 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[FDNG19]   Farnoud Farahmand, Viet B. Dang, Duc Tri Nguyen, and Kris Gaj. Evaluating the potential for hardware acceleration of four ntru-based key encapsulation mechanisms using software/hardware codesign. In *PQCrypto*, volume 11505 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2019.

[FS19]   Tim Fritzmann and Johanna Sepúlveda. Efficient and flexible low-power ntt for lattice-based cryptography. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[FSF$^+$18] Tim Fritzmann, Thomas Schamberger, Christoph Frisch, Konstantin Braun, Georg Maringer, and Johanna Sepúlveda. Efficient hardware/software codesign for ntru. In *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*, pages 257–280. Springer, 2018.

[FSMG$^+$19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepulveda. Towards reliable and secure post-quantum co-processors based on risc-v. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1148–1153. IEEE, 2019.

[GS66]   W. M. Gentleman and G. Sande. Fast fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS 66 (Fall), page 563578, New York, NY, USA, 1966. Association for Computing Machinery.

[JA11]   B. Jungk and J. Apfelbeck. Area-efficient fpga implementations of the sha-3 finalists. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 235–241, Nov 2011.

[KRSS19]   Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844.

[Lin19]   Linux Foundation. Chips: Common hardware for interfaces, processors and systems, 2019. https://chipsalliance.org/.

[LPR10]   Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23, 2010.

[LS15]   Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[MKV20]   Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 222–244, 2020.

[MTK+20]   Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism. Cryptology ePrint Archive, Report 2020/321, 2020. https://eprint.iacr.org/2020/321.

[Nat19]   National Institute of Standards and Technology. Status report on the first round of the nist post-quantum cryptography standardization process, 2019. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf.

[Ope19]   OpenPower Foundation. Openpower, 2019. https://openpowerfoundation.org.

[Ora19]   Oracle and Sun Microsystems. Opensparc overview, 2019. https://www.oracle.com/technetwork/systems/opensparc/index.html.

[PG12]   Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *International Conference on Cryptology and Information Security in Latin America*, pages 139–158. Springer, 2012.

[POG15]   Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015.

[RIS19a]   RISC-V Foundation. Cryptographic extensions task group, 2019. https://lists.riscv.org/g/tech-crypto-ext.

[RIS19b]   RISC-V Foundation. Risc-v cores and soc overview, 2019. https://riscv.org.

[RVM+14]   Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 371–391. Springer, 2014.

[TGS19]   A. Traber, M. Gautschi, and P. D. Schiavone. PULP RI5CY: User Manual. Apr. 2019., 2019. https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf.

[WJW+18]   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. *IACR Cryptology ePrint Archive*, 2018:1225, 2018.

[WTJ+20]   Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qtesla. Cryptology ePrint Archive, Report 2020/054, 2020. https://eprint.iacr.org/2020/054.
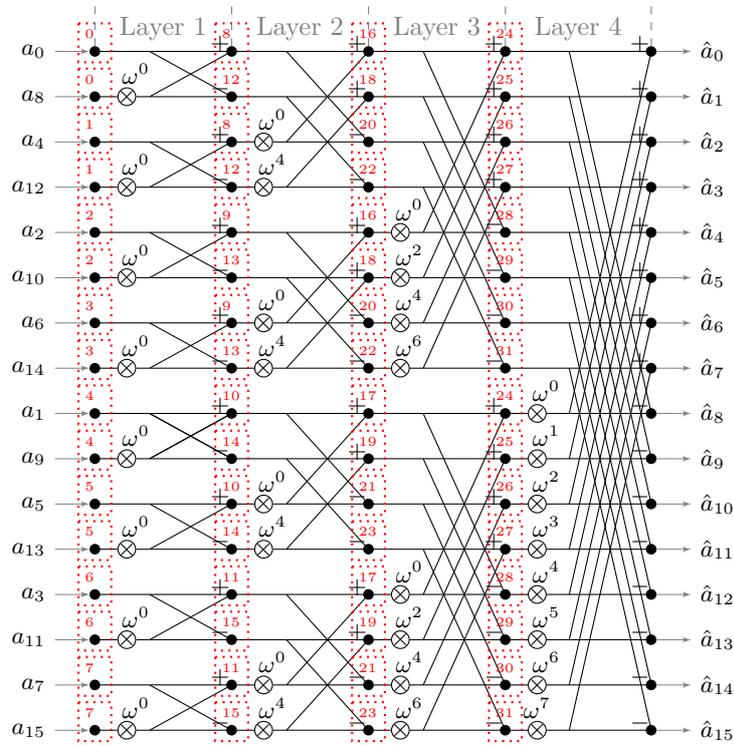
A NTT Design

Figure 9: Example $NTT_{br \leftarrow no}^{CT}$ with $n = 16$ (unoptimized). In this case, one coefficient is stored in a single memory word, two coefficients can be loaded to the register file ($l = 2$), and one pair of coefficients can be executed in parallel. The red boxes indicate which coefficients are stored together in one word (in this case they are not stored together) and in which order the coefficients are processed by a single butterfly unit.

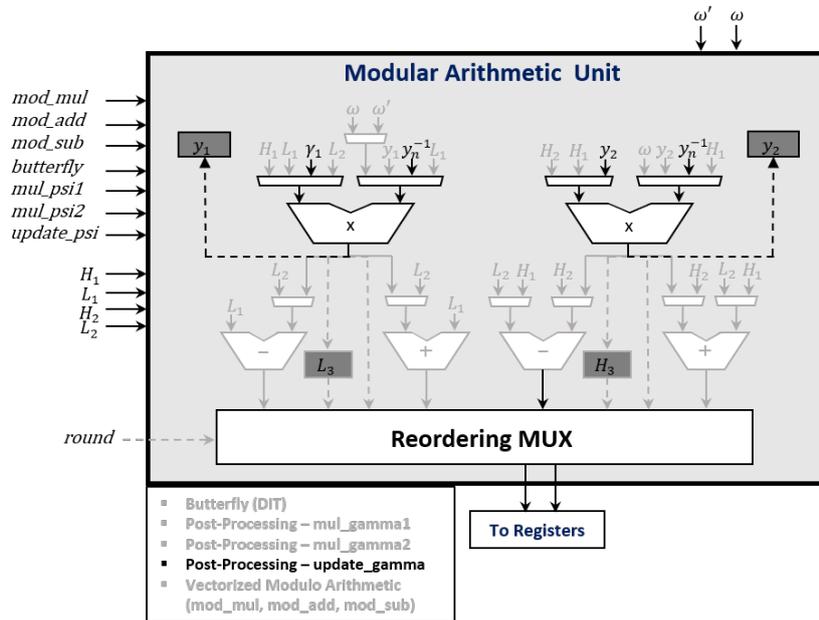# B   Hardware Accelerator – Modular Arithmetic Unit
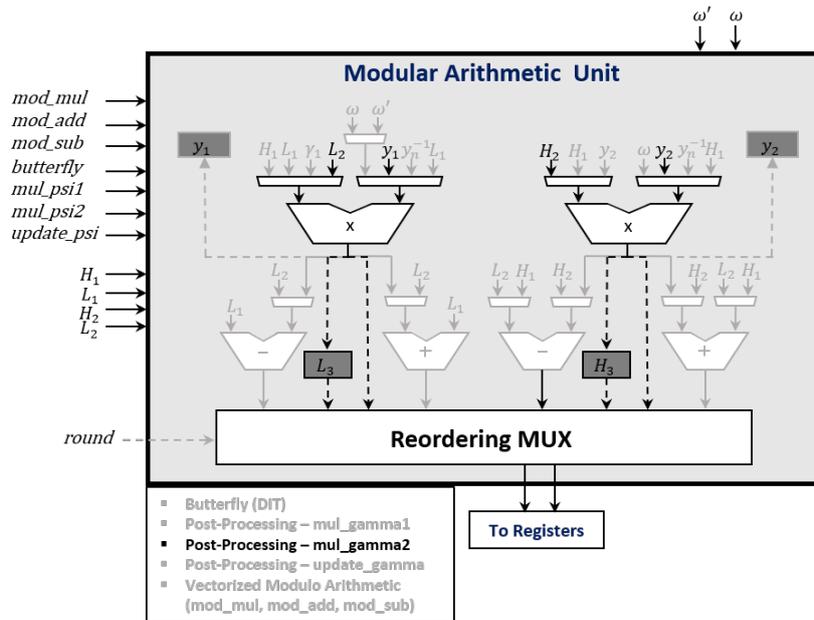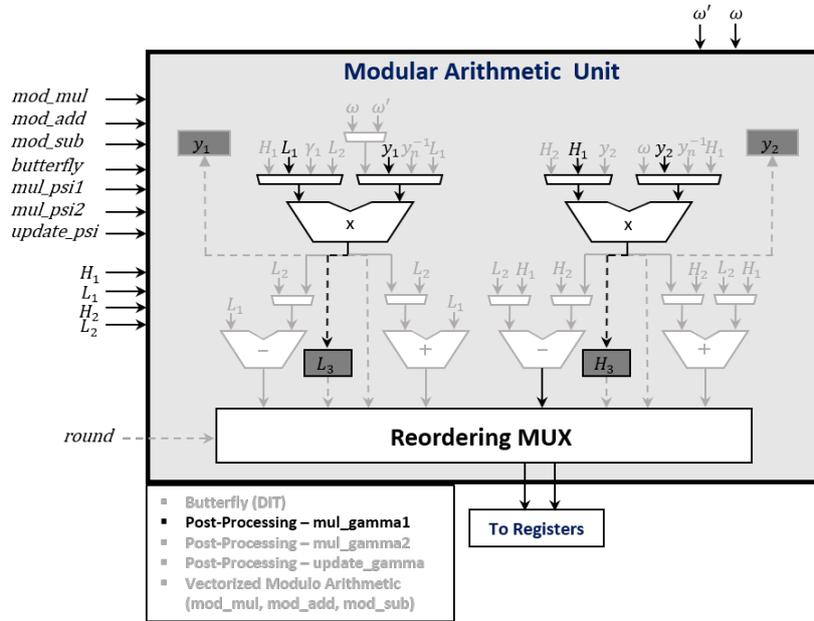
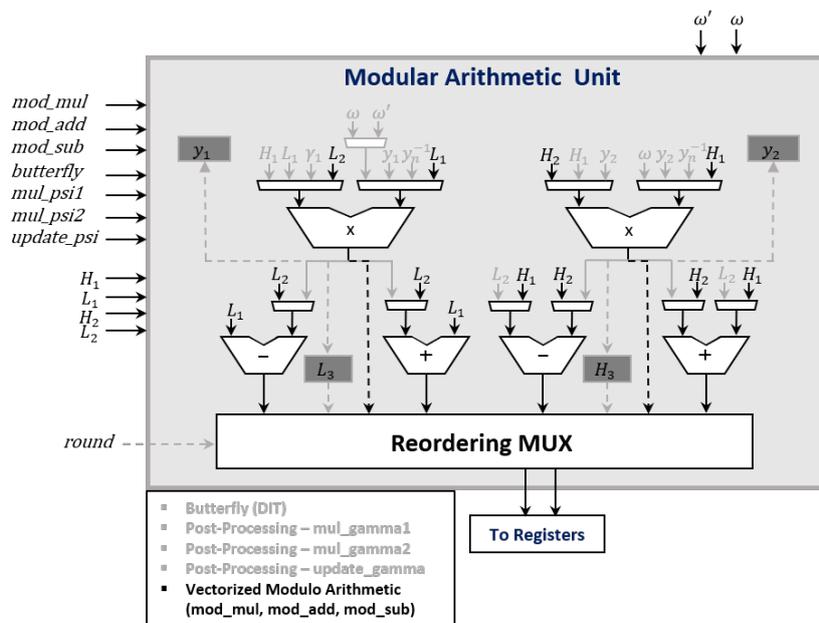**Figure 10:** Modular Arithmetic Unit – Post-Processing Operation (*mul_gamma*1, *mul_gamma*2, *update_gamma*)

Figure 11: Modular Arithmetic Unit – Vectorized Modulo Arithmetic

C   ISA Extension

Table 9: PQ ISA Extension.

Opcode	Funct3	Funct7	Operation Name
111 0111	000	0000000	NTT Configuration: *pq.set_kyber*
111 0111	001	0000000	NTT Configuration: *pq.set_newhope512*
111 0111	010	0000000	NTT Configuration: *pq.set_newhope1024*
111 0111	011	0000000	NTT Configuration: *pq.set_fwd_ntt*
111 0111	100	0000000	NTT Configuration: *pq.set_inv_ntt*
111 0111	101	0000000	NTT Configuration: *pq.set_first_rounds*
111 0111	110	0000000	NTT Configuration: *pq.set_last_round*
111 0111	000	0000001	NTT Operation: *pq.ntt_multiple_bf*
111 0111	001	0000001	NTT Operation: *pq.ntt_single_bf*
111 0111	010	0000001	NTT Operation: *pq.update_m*
111 0111	011	0000001	NTT Operation: *pq.update_omega*
111 0111	100	0000001	NTT Operation: *pq.mul_gamma1*
111 0111	101	0000001	NTT Operation: *pq.mul_gamma2*
111 0111	110	0000001	NTT Operation: *pq.update_gamma*
111 0111	000	0000010	Modular Arithmetic Operation: *pq.mod_mul*
111 0111	001	0000010	Modular Arithmetic Operation: *pq.mod_add*
111 0111	010	0000010	Modular Arithmetic Operation: *pq.mod_sub*
111 0111	011	0000010	Modular Arithmetic Operation: *pq.bf_dit*
111 0111	100	0000010	Modular Arithmetic Operation: *pq.bf_dif*
111 0111	000	0000011	Bit-reversal: *pq.br256*
111 0111	001	0000011	Bit-reversal: *pq.br512*
111 0111	010	0000011	Bit-reversal: *pq.br1024*
111 0111	000	0000100	Keccak Operation: *pq.keccak_f1600*
111 0111	000	0000101	Binomial Sampling: *pq.bs_k2*
111 0111	001	0000101	Binomial Sampling: *pq.bs_k3*
111 0111	010	0000101	Binomial Sampling: *pq.bs_k5*
111 0111	011	0000101	Binomial Sampling: *pq.bs_k8*
111 0111	001	0000110	Vectorized Modulo Multiply Accumulate: *pq.mac*