Enabling Lattice-Based Post-Quantum Cryptography on the OpenTitan Platform

Tobias Stelzer
Fraunhofer AISEC
Garching, Germany
tobias.stelzer@aisec.fraunhofer.de

Felix Oberhansl
Fraunhofer AISEC
Garching, Germany
felix.oberhansl@aisec.fraunhofer.de

Jonas Schupp
Technical University Munich, Germany
TUM School of Computation, Information and Technology
Munich, Germany
jonas.schupp@tum.de

Patrick Karl
Technical University Munich, Germany
TUM School of Computation, Information and Technology
Munich, Germany
patrick.karl@tum.de

ABSTRACT

The first generation of post-quantum cryptography (PQC) standards by the National Institute of Standards and Technology (NIST) is just around the corner. The need for secure implementations is therefore increasing. In this work, we address this need and investigate the integration of lattice-based PQC into an open-source silicon root of trust (RoT), the OpenTitan. RoTs are important security building blocks that need to be future-proofed with PQC. The OpenTitan features multiple cryptographic hardware accelerators and countermeasures against physical attacks, but does not offer dedicated support for lattice-based PQC. Thus, we propose instruction set extensions for the OpenTitan Big Number Accelerator (OTBN) to improve the efficiency of polynomial arithmetic and sampling. As a case study we analyze the performance of signature verification of digital signature scheme Dilithium. Our implementation verifies signatures within 997,722 cycles for security level II, pushing this RoT functionality below 10 ms for the OpenTitan's target frequency of 100 MHz. With an overhead of 242 kGE, our hardware extensions make up only about 5 % of the total RoT area. All our extensions integrate seamlessly with countermeasures against physical attacks and comply with the adversary model chosen by the OpenTitan project.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Digital signatures**.

KEYWORDS

post-quantum cryptography, lattice-based cryptography, digital signatures, hardware/software co-design

ASHES '23, November 30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0262-4/23/11.
https://doi.org/10.1145/3605769.3623993

ACM Reference Format:
Tobias Stelzer, Felix Oberhansl, Jonas Schupp, and Patrick Karl. 2023. Enabling Lattice-Based Post-Quantum Cryptography on the OpenTitan Platform. In *Proceedings of the 2023 Workshop on Attacks and Solutions in Hardware Security (ASHES '23), November 30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605769.3623993

1 INTRODUCTION

In recent years, research on post-quantum cryptography (PQC) gained significant momentum. Since Shor's work on prime factorization and discrete logarithms [37], the threat of quantum computers looms over the security assumption of contemporary asymmetric cryptosystems like RSA and ECC. Consequently, quantum secure alternatives must be employed to prepare for potential breakthroughs of large-scale quantum computers. Efficient implementations and integration of various PQC schemes have come more and more into the focus of researchers. Recently, these efforts were focused on the schemes the National Institute of Standards and Technology (NIST) selected in 2022 as part of its 5-year effort to standardize PQC schemes. For digital signatures, the two lattice-based schemes CRYSTALS-Dilithium [14] and Falcon [15], and the hash-based scheme SPHINCS+ [7] were selected. As key encapsulation mechanism (KEM), the lattice-based scheme CRYSTALS-Kyber [9] was selected. NIST asked researchers to focus their efforts on a few platforms to ensure comparability of results, e.g. the ARM Cortex M4 microcontroller [1, 20, 21] for software implementations and Artix-7 FPGAs [13, 23, 36, 38] for hardware designs. As the standardization process produces the first standards, the integration into more heterogeneous platforms needs to be evaluated to assist the transition to PQC schemes. In particular, the investigation of hardware/software co-designs is important as they combine the flexibility of software with the efficiency of hardware. Interesting efforts in that direction were made in [4, 16, 18, 22]. The authors' approach is to extend the open RISC-V instruction set architecture (ISA) to accelerate PQC operations. Hardware extensions are then coupled tightly into the processor pipeline and reuse processor resources such as the internal registers. In contrast to connecting external accelerators over a generic bus interface, this approach saves area and time needed for data transfers. The authors of [4] extend the ISA for Kyber's and NewHope's finite-field arithmetic. Fritzmann et al. [18] integrate a complete set of PQC hardware

accelerators into the RISC-V pipeline. Besides accelerating Number Theoretic Transform (NTT)-based polynomial multiplication for Kyber, NewHope and Saber, they also target polynomial generation in this work. In contrast to [4] and [18], the authors of [16] target masked hardware/software co-design extensions and provide a more generic implementation which supports a variety of lattice-based schemes. Karl et al. [22] re-use some of these accelerators for PQC signatures.

In this work, we undertake the next logical step in this direction by integrating hardware extensions for Dilithium into a heterogeneous System-on-Chip (SoC) platform, namely the OpenTitan[27]. The OpenTitan is a RISC-V based open-source silicon root of trust (RoT). As such it can be responsible for booting a system and updating firmware securely, mitigating any attempts of an adversary to execute malicious code on the system. Chips that are designed today and have an approximate lifetime of more than 15 years must support PQC signatures to meet such security guarantees. Further, RoTs must continue to support contemporary signatures based on RSA or ECC, as e.g. the French and German Offices for Information Security recommend using hybrid signatures built from a contemporary and a PQC signature scheme [5, 11]. Finally, a RoT should incorporate countermeasures against physical attacks such as fault injections that could potentially be used by an adversary to circumvent a secure boot process [10]. Currently, the OpenTitan fulfills two of the three listed requirements: support for contemporary signatures and countermeasures against physical attacks. We close this gap by extending the OpenTitan to support lattice-based PQC. More specifically, we extend the OpenTitan Big Number Accelerator (OTBN), the OpenTitan's dedicated Public-Key Cryptography (PKC) co-processor, by dedicated hardware accelerators for polynomial arithmetic and sampling. By extending the instruction set of this co-processor we make our hardware accelerators accessible. As a result, the OTBN is able to accelerate the schemes Dilithium, Kyber and Falcon in addition to traditional PKC. We choose to focus on Dilithium in this work, due to its popularity, close relation to the Kyber KEM, and overall efficiency. In summary, we provide the following contributions:

- Design of a configurable post-quantum arithmetic logic unit (PQ-ALU) for the OTBN that implements polynomial arithmetic used within Dilithium, Kyber and Falcon.
- Design of Keccak instruction set extensions for the OTBN following the approach from [34] to improve polynomial sampling in Dilithium, Kyber and Falcon.
- A complete implementation of the Dilithium verify operation on the OpenTitan that is suitable to provide secure boot and firmware update functionality needed by a silicon RoT.
- A detailed evaluation of our hardware design, demonstrating that Dilithium's signature verification can be achieved with a latency below 10 ms. Furthermore, we provide a comparison with related work on PQC RISC-V instruction set extensions and signature implementations.

Our hardware extensions as well as the Dilithium implementations are publicly available under https://github.com/Fraunhofer-AISEC/otbn-pq.

2 PRELIMINARIES

2.1 Notation

Let $\mathcal{R}_q = \mathbb{Z}_q[x]/\phi(x)$ denote a polynomial ring with modulus q and cyclotomic polynomial $\phi(x)$. Let lower case letters denote polynomials i.e. $a \in \mathcal{R}_q$, bold lower-case letters denote vectors of polynomials i.e. $\boldsymbol{b} \in \mathcal{R}_q^k$ and bold upper-case letters denote matrices of polynomials $\boldsymbol{A} \in \mathcal{R}_q^{k \times l}$ for $k, l \in \mathbb{Z}$.

2.2 Lattice-based Cryptography and the Learning with Error Problem

Lattice-based cryptography [2] is a promising class of algorithms for post-quantum PKC. Its security relies on the hardness of mathematical problems defined over lattices. A very popular lattice problem which is used to construct cryptographic primitives is the Learning with Error (LWE) problem [35] and its algebraic variants, namely the Ring-Learning with Error (RLWE) problem and the Module-Learning with Error (MLWE) problem. They are highly relevant for practical applications as they offer higher efficiency and smaller key sizes compared to the plain LWE problem [3, 29]. Therefore, these two variants are briefly introduced in the following.

2.2.1 Ring-Learning with Error Problem. In [29] Lyubashevsky et al. introduced the RLWE problem using ideal lattices. The RLWE distribution $D_{q,s,\chi}$ over $\mathcal{R}_q$ is defined by the tuple $(a_i(x), r_i(x))$, where the elements of $a_i(x)$ are uniformly sampled over $\mathbb{Z}_q$ and $r_i(x)$ is calculated in the polynomial ring $\mathcal{R}_q$ according to Equation 1.

$$r_i(x) = a_i(x) \cdot s(x) + e(x) \mod \phi(x) \tag{1}$$

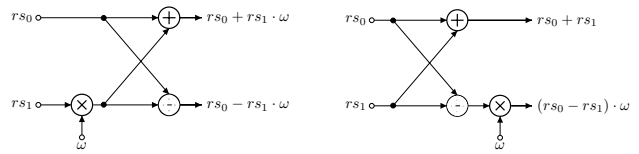
2.2.2 Module-Learning with Error Problem. The MLWE introduced in [24] is a generalization of the LWE and RLWE problem. Given the secret polynomial $\boldsymbol{s}(x) \in \mathcal{R}_q^{k_2}$ and the error polynomial $\boldsymbol{e} \in \mathcal{R}_q^{k_1}$ the MLWE problem is defined as in Equation 2.

$$\boldsymbol{r}(x) = \boldsymbol{A}(x) \cdot \boldsymbol{s}(x) + \boldsymbol{e}(x) \mod \phi(x) \tag{2}$$

Compared to RLWE, the public polynomial $a_i(x)$ of the polynomial ring $\mathcal{R}_q$ is replaced by a public module $\boldsymbol{A}(x) \in \mathcal{R}_q^{k_1 \times k_2}$ with dimension $k_1 \times k_2$, which is sampled from a uniform distribution.

2.3 Performance Bottlenecks in Lattice-based PQC Schemes

2.3.1 Polynomial Sampling. The aforementioned variants of the LWE problem require sampling of random polynomials. On that account most lattice-based PQC schemes - including Kyber, Dilithium and Falcon - rely on the hash functions defined in the SHA-3 standard [32]. The SHA-3 family is based on the Keccak [8] primitive and includes two extendable-output functions (XOFs), namely SHAKE128 and SHAKE256. These XOFs are popular choices for pseudo-random number generation and suitable for polynomial sampling. Both, SHAKE128 and SHAKE256, operate on a 1600-bit Keccak state A which can be represented by a three-dimensional cube. This array contains 5×5 words, each with a length of 64 bits. These words are called lanes. Each bit within this cube can be addressed over $A[x, y, z]$, where $0 \leq x < 5$, $0 \leq y < 5$ and $0 \leq z < 64$. Similarly, each lane can be addressed over $A[x, y]$. A plane is defined as $A[0, y]||A[1, y]||A[2, y]||A[3, y]||A[4, y]$. The



(a) CT Butterfly (b) GS Butterfly

Figure 1: NTT butterfly operations

state permutation consists of 24 rounds of the $f - 1600$ function. This function itself is split into five consecutive steps, namely θ, ρ, π, χ and ι which manipulate the state array A. For a detailed explanation of Keccak we refer the reader to [8].

2.3.2 Polynomial Multiplication. In lattice-based cryptography, polynomial arithmetic is performed over polynomial rings $\mathcal{R}_q$. Typically, $\phi(x)$ is chosen as $x^n + 1$. Computing the product $c(x)$ of two polynomials $a(x)$ and $b(x)$ within this polynomial ring using the naive schoolbook multiplication results in a computational complexity of $O(n^2)$.

There are several methods to reduce the complexity of polynomial multiplication. In particular the NTT-based approach has been a popular choice among cryptographers, as it reduces the complexity from $O(n^2)$ to $O(n \log(n))$. Two approaches to compute the NTT are the Cooley-Tukey (CT) [12] and the Gentleman-Sande (GS) [19] algorithms. The core elements of both algorithms are the so-called butterfly operations, which consist of simple arithmetic in $\mathbb{Z}_q$. Figure 1 illustrates these butterfly operations. The factor ω corresponds to a root-of-unity for the polynomial and coefficient ring and is often also referred to as twiddle factor. While the CT butterfly operation decimates the input in time domain the GS butterfly decimates the input in frequency domain. As discussed by Pöppelmann et al. in [33], there are different variants of NTT algorithms using the CT and GS butterfly operations. To avoid additional bit-reversal steps, it is a common procedure to use CT butterflies for transformation into the NTT domain and GS butterflies for reverse transformation.

2.4 OpenTitan Big Number Accelerator

The OTBN [28] is a co-processor within the OpenTitan SoC, specialized for traditional PKC. As the OTBN operates on sensitive data like secret keys, the whole design centers around security. Consequently, there is a rich set of security features implemented in the OTBN. Moreover, the control flow is strictly separated from the data flow by using two distinct instructions sets.

The architecture of the OTBN is based on the Ibex core [25] and consists of two stages, namely the Instruction Fetch (IF) stage and the Instruction Decode (ID) stage. The OTBN has two distinct sets of registers, namely the General Purpose Registers (GPRs) and Wide Data Registers (WDRs). The 32-bit wide GPRs are accessed over the RV32I base instruction subset and are connected to a basic Arithmetic Logic Unit (ALU). The base instruction subset is not used for the data flow, but for the control flow only. To support the wide-integer arithmetic for ECC and RSA, the OTBN contains the Big Number (BN) instruction subset which has a 256-bit wide data path. The 256-bit WDRs are accessed over the BN instruction subset and are used for the data flow. To operate on this register set, the OTBN deploys the Big Number Arithmetic Logic Unit (BN-ALU) with a 256-bit data path and the Big Number Multiply-Accumulate (BN-MAC) module. The OTBN features various generic countermeasures against physical attacks, such as one-hot encoding of signals and integrity codes for register contents. For detailed information about the technical specification and the ISA of the OTBN the reader is referred to [28].

3 HARDWARE ACCELERATORS

This section provides a detailed description of the hardware elements which are integrated into the OTBN. A PQ-ALU and Keccak-Unit (KU) which accelerate polynomial arithmetic and sampling, respectively. A novelty that separates our design from previous works [4, 16, 18, 22] is the integration of a twiddle and round counter update unit (TRCU) and register address unit (RAU). The RAU and TRCU update and store necessary constants for data and control flow in a set of registers, namely the Post-Quantum Special Purpose Register Set (PQ-SPR). This inherent parallelism enhances the efficiency of our instructions while avoiding the necessity to artificially increase the number of register operands. Using the dedicated PQ-SPR prevents interference with the standard register sets and thereby, saves additional stack operations.

3.1 Post-Quantum Arithmetic Logic Unit

As mentioned in Section 2.3, polynomial arithmetic, and particularly multiplication, is a major performance bottleneck for lattice-based cryptography. Typically, polynomial multiplication is realized in the NTT domain, which makes the NTT the major bottleneck. Previous implementations [6, 17, 18] use NTT algorithms which rearrange the input polynomial in bit-reverse order as this allows a simple on-the-fly calculation of twiddle factors. On-the-fly calculation is desirable for designs that support multiple schemes as it avoids the necessity to store $O(n)$ pre-computed twiddle factors for each scheme. In [39], the authors raised the concern that loads and stores during re-ordering might cause trivially exploitable side-channel leakage of secret polynomials and proposed an algorithm to avoid the bitreversal operation, while still allowing the computation of most twiddle factors on-the-fly. In this work, we extend the NTT algorithm presented in [39].

3.1.1 In-place NTT with on-the-fly Twiddle Factor Generation. Algorithm 1 describes a $\text{NTT}^{CT}_{no \rightarrow br}$ in-place variant, based on the algorithm from [39] with an extension to also support incomplete NTTs. For this purpose, Algorithm 1 takes the additional input l which describes in how many terms the input polynomial should be split and determines when the transformations should be aborted. The inverse NTT is adapted analogously from [39]. The values of ω_m and ζ are derived from the few pre-computed twiddle factors, depending on the current NTT layer p. The inner loop (line 8-13) corresponds to the CT butterfly operation as discussed in Section 2.3.2. The outer loop (line 6 -15) handles the coefficient offsets for the respective layer, such that a dedicated bit-reversal is obsolete. Algorithm 1 shows that only $O(\log n)$ twiddle factors need to be pre-computed and stored in memory. The remaining values for ζ are computed on-the-fly. For a more detailed explanation of

Data: a_i for $i = 0, \ldots, n-1$; (ω_n^k, ψ_n^k) for $k = \frac{l}{2}, \frac{l}{4}, \ldots, 1$; n; l
Result: a_i for $i = 0, \ldots, n-1$

$m := n >> 1$; $p := l >> 1$;
$stop := n/l$; $j_2 := 1$;
while $m \geq stop$ **do**
  $\omega_m := \omega_n^p \mod q$;
  $\zeta = \psi_n^p \mod q$;
  for $(j := 0; j < j_2; j++)$ **do**
    $j_{br} :=$ BitReverse(j);
    for $(k := 0; k < m; k++)$ **do**
      $rs_0 := a_{k+j_{br}}$;
      $rs_1 := a_{k+j_{br}+m}$;
      $a_{k+j_{br}} := (rs_0 + \zeta \cdot rs_1) \mod q$;
      $a_{k+j_{br}+m} := (rs_0 - \zeta \cdot rs_1) \mod q$;
    end
    $\zeta := \zeta \cdot \omega_m \mod q$;
  end
  $m := m >> 1$;
  $p := p >> 1$;
  $j_2 := j_2 << 1$;
end
return a_i for $i = 0, \ldots, n-1$

Algorithm 1: Iterative Forward NTT based on [39]

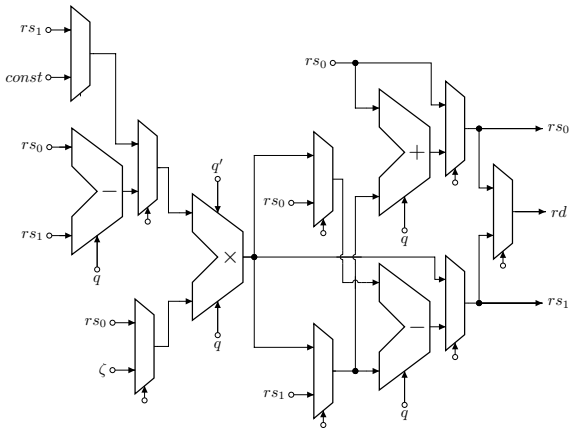


Figure 2: Architecture of PQ-ALU

the NTT algorithm, the reader is referred to [39]. In the following, we will describe instruction set extensions that can handle this task efficiently. This keeps the code size compact and allows flexibility in switching from one PQC scheme to another.

3.1.2 PQ-ALU Hardware Design. To accelerate the NTT, previous works extend the instruction set to support finite field arithmetic [4] and the butterfly operations directly [18, 31]. In this work, we follow the approach of [18, 31] and offer direct support for the butterfly operations. In contrast to previous works, we choose to implement a 32-bit wide datapath to provide better support for lattice-based schemes which need larger primes. Furthermore, our PQ-ALU provides full configurability of the finite-field arithmetic as the prime q can be chosen arbitrarily within 32 bit. The architecture of our PQ-ALU is depicted in Figure 2 and implements modular arithmetic in $\mathbb{Z}_q$. It consists of two subtractors, one adder and one multiplier along with the respective modular reduction logic. Modular reduction for the adder and the subtractors is implemented using conditional subtractions and additions. For the modular multiplier, the Montgomery reduction algorithm [30] is implemented. The signals q and $q' = -q^{-1} \mod 2^{32}$ configure the arithmetic blocks. The proposed PQ-ALU supports the execution of both butterfly operations, the CT and the GS butterfly operation. Besides these butterfly operations, the PQ-ALU supports finite-field addition, subtraction, multiplication and scaling by constants. For finite field addition, subtraction and multiplication, as well as for the butterfly operations, the PQ-ALU consumes the two operands rs_0 and rs_1. For the in-place butterfly operations the PQ-ALU consumes the twiddle factor ζ as additional operand. While the finite-field operations store a single result in rd, the butterfly operations generate two results which are written back to rs_0 and rs_1. For scaling by constants, the PQ-ALU uses rs_0 as well as the constant $const$ as operands. The values of ζ, $const$, q and q' originate from the PQ-SPR located within the TRCU (see Section 3.3).

3.2 Keccak Unit

In almost all lattice-based schemes Keccak-based algorithms are used for polynomial sampling, which constitutes another major performance bottleneck in lattice-based cryptography (Section 2.3). Keccak's internal structure makes it highly suitable for vectorized implementations. Depending on the available chip area, the instruction set can be extended for different use cases. The authors of [18] propose a highly customized instruction to process a complete round permutation of the whole state within a single cycle. On that account, the Keccak accelerator is required to access 50 32-bit registers concurrently. Regarding the OTBN, such a highly customized instruction is hard to integrate in compliance with the OTBN's register protection. Therefore, we base our instructions on the more generic vector-processing unit proposed in [34]. Although the work of [34] targets a 128-bit architecture, the design patterns can be applied to other architectures, e.g. the OTBN and its 256-bit wide WDRs. The architecture of the KU is illustrated in Figure 3 and consists of two distinct modules, namely the Keccak-Lane-Unit (KLU) and the Keccak-Plane-Unit (KPU).

3.2.1 Keccak Plane Unit. The KPU operates on planes (5×64-bit) and implements the χ and parts of the θ step of a Keccak round. More specifically, the KPU consumes one plane and outputs one plane. The χ block depicted in Figure 3 computes

$$A'[x, y] = A[x, y] \oplus (\neg A[x+1, y] \,\&\, A[x+2, y])$$

for each lane within one plane. The θ block implements the parity computation within the θ step as

$$A'[x, y, z] = A[x-1, y, z] \oplus A[x+1, y, z-1].$$

3.2.2 Keccak Lane Unit. The KLU operates on lanes (64-bit) and implements the χ step and parts of the θ step of a Keccak round. The KLU takes two lanes as inputs and outputs one lane. This block implements different variants of 64-bit XOR operations. For the

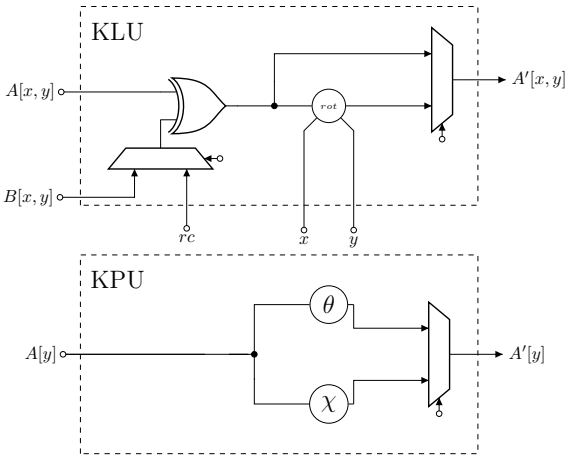


Figure 3: KU consisting of KLU and KPU

ι step it implements the XOR operation between one lane and a round counter value rc which originates from the PQ-SPR located within the TRCU as

$$A'[x, y] = A[x, y] \oplus rc.$$

The ρ step can be formulated as

$$A'[x, y] = rot(A[x, y] \oplus B[x, y]), (x, y)).$$

It implements a XOR operation where the bits of both input lanes are rotated depending on x and y. The values of x and y originate from the RAU.

3.3 Twiddle and Round Counter Update Unit

The TRCU consists of a PQ-SPR with corresponding arithmetic units to update the values within these registers accordingly as depicted in Figure 4. This module is responsible to provide both processing units, PQ-ALU and KU with the necessary constants as stated in Section 3.1 and Section 3.2. Moreover, it computes the twiddle factors ζ on-the-fly. To this end, the TRCU updates ζ as $\zeta = \zeta \cdot \omega_m$. The value of ω_m depends on the current NTT layer and is a pre-computed constant which is stored within the ω-table. This table can hold up to eight pre-calculated values. One entry within this table is selected through the idx_ω. The ψ-table and the rc-table for Keccak work analogously. For the inverse NTT one pre-computed ω_m value is sufficient as the remaining values can also be calculated on-the-fly as $\omega_m = \omega_m^2$. Similarly, ψ can be updated on-the-fly for the inverse NTT as $\psi_m = \omega_m$. Therefore, storing one pre-computed value is sufficient in this case. To reduce the memory requirements when computing a basecase multiplication for Kyber (incomplete NTT) the TRCU exploits the symmetry $\omega_n^{k+n/2} = -\omega_n^k$ and updates ζ as $\zeta = q - \zeta$.

3.4 Register Address Unit

The RAU automatically computes WDRs-addresses and word indices for the NTT. Additionally, it holds and updates the values of x and y for the offset selection within the KLU as described in Section 3.2. Figure 5 shows the architecture of the RAU. The registers idx_0 and idx_1 correspond to the values of $k + j_{br}$ and

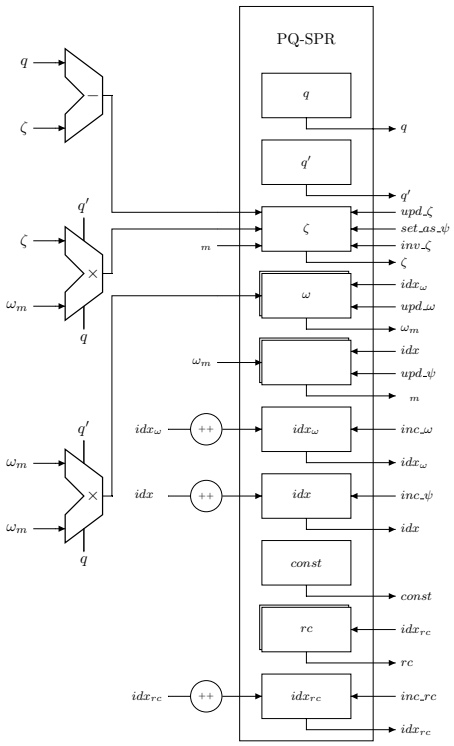


Figure 4: TRCU and PQ-SPR

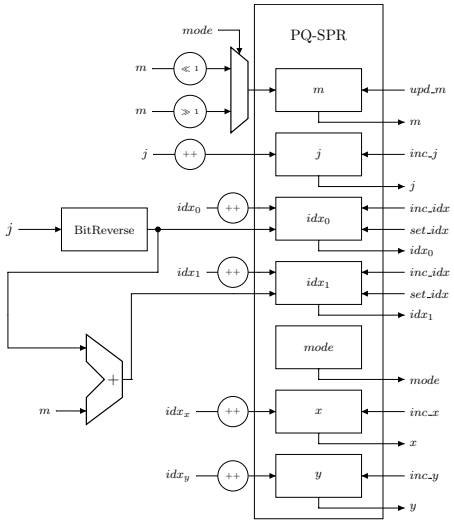


Figure 5: RAU and PQ-SPR for control flow operations

$k + j_{br} + m$, respectively. Depending on these values the respective WDRs-addresses and word indices can be derived. Its integration into the ISA is discussed in the next section.

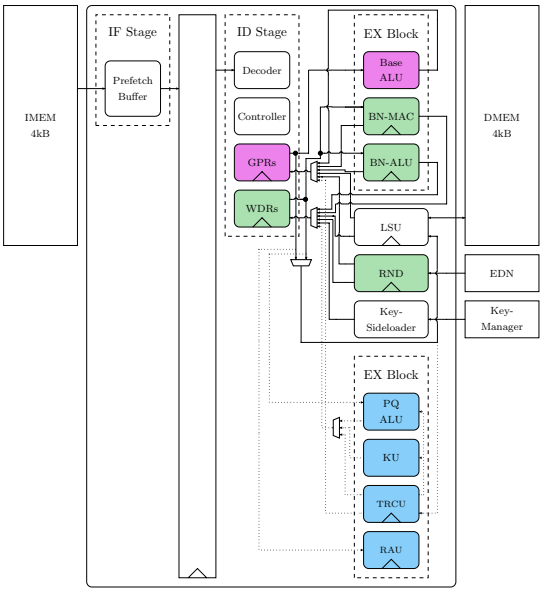


Figure 6: Simplified architecture of the OTBN with PQC extension. Elements of the BN, Base and PQC subset are highlighted in green, pink and blue, respectively. Paths involved in the integration of the PQC modules are dotted.

4 INTEGRATION OF HARDWARE ACCELERATORS INTO THE OTBN

This section discusses the integration of the proposed hardware accelerators into the OTBN pipeline and presents the ISA extension.

4.1 Integration of Hardware Accelerators

All hardware accelerators are integrated into the ID stage of the OTBN. The resulting pipeline of the OTBN is depicted in Figure 6. Both, the PQ-ALU and the KU operate on the WDRs. For the integration of the PQ-ALU we modify the WDR set to support 32-bit word addressing. Furthermore, we extend the WDR by an additional write port as needed for the two results of a butterfly operation. The PQ-SPR within TRCU is accessible through both the WDR-set and GPR-set. The RAU is accessible only through the GPR set. As described in Section 3.4, this module generates WDR addresses as well as indices for the respective 32-bit word within one WDR. This functionality is used for instructions with indirect register addressing, which are introduced below.

Integration with Countermeasures. As already stated in Section 2.4, the OTBN features a set of generic countermeasures against physical attacks (e.g. fault injections) such as integrity-coded registers and signal encodings with high hamming distance. By re-using the WDR and GPR and integrating our accelerators into the ID stage, all these countermeasures also apply to our extensions.

4.2 Instruction Set Extension

Our ISA extensions can be grouped into two categories: instructions with direct register addressing and instructions with indirect register addressing, where the RAU selects the operands from the WDR.

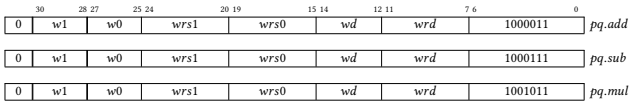


Figure 7: Custom instructions for arithmetic instructions

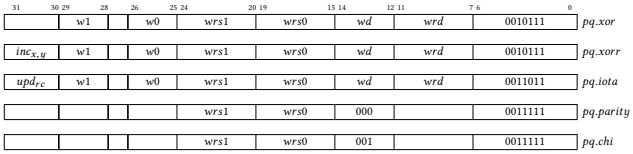


Figure 8: Custom instructions for Keccak instructions

4.2.1 Custom Instructions with Direct Register Addressing. The arithmetic instructions with direct register addressing encode the address of the WDR and the index of the 32-bit word within the WDR directly into the instruction. Figure 7 lists the customized arithmetic instructions and shows the respective encoding. The following instructions use the PQ-ALU for finite field arithmetic: *pq.add*, *pq.sub* and *pq.mul*. The customized instruction formats use the $w0$, $w1$ and wd fields to index the respective 32-bit subwords within one WDR. More specifically, the bit fields $w0$ and $w1$ are used to select the 32-bit subwords within the source WDRs addressed by $wrs0$ and $wrs1$, respectively. Similarly, wd is used to select the 32-bit subword within the destination WDR addressed by wrd.

Figure 8 lists customized Keccak instructions and shows the respective encoding. For the KU, these instructions include two lane-wise XOR operations, namely *pq.xor* and *pq.xorr*. Both instructions compute a XOR operation between two 64-bit lanes specified over $wrs0$, $w0$, $wrs1$ and $w1$. While for *pq.xor* the result is written directly into the 64-bit word addressed by wrd and wd, the *pq.xorr* instruction rotates the result before writing it to the destination register. The rotation is dependent on the x and y values inside the RAU registers. To increment these values the *pq.xorr* includes the $inc_{x,y}$ field, which encodes inc_x and inc_y. Similarly, the *pq.iota* instruction computes a lane-wise XOR of the lane specified by $wrs0$ and $w0$ and the round counter value from the rc register inside the TRCU. The value of the round counter can be updated over the upd_{rc} field, which encodes inc_rc. Finally, there are two plane-wise instructions *pq.parity* and *pq.chi*. Both are in-place instructions and operate on a 320-bit plane consisting of all 256 bits of $wrs0$ and the 64 LSBs of $wrs1$. While *pq.parity* computes a parity plane, *pq.chi* performs the χ step as described in Section 3.2.

4.2.2 Custom Instructions with Indirect Register Addressing. It is possible to minimize memory accesses by exploiting the full capacity of the WDRs. However, if instructions for a butterfly operation were to be realized with direct register addressing, the code size would be disproportionately large when implementing the NTT. This is due to the fact that address and index would have to be

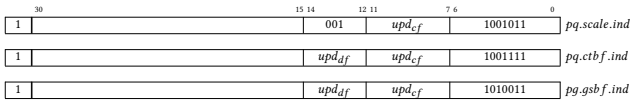


Figure 9: Custom instructions for arithmetic instructions with indirect register addressing

hard-coded into these instructions leading to one line of code for each butterfly operation to be performed. However, when addressing registers and words indirectly via the RAU, the hardware loop functionality of the OTBN can be leveraged and simple butterfly instructions can be re-used for all NTT layers to avoid a massive increase of the code size. The role of the $w0$, $w1$ as well as the $wrs0$ and $wrs1$ fields are adopted by the signals idx_0 and idx_1 from the RAU. Figure 9 lists all arithmetic instructions with indirect register addressing and shows the respective encoding. These instructions include finite field scaling with a constant *pq.scale.ind*, as well as the CT butterfly operation *pq.ctbf.ind* and the GS butterfly operation *pq.gsbf.ind*. The butterfly operations operate in-place, i.e., source and destination are identical. For *pq.scale.ind*, the result is written into the subword within the WDR addressed by $w0$ and $wrs0$. Furthermore, it does not need the idx_1 input from the RAU, as only one coefficient is processed. The factor used for scaling is taken from the *scale* signal from the TRCU.

To update NTT-specific variables, all instructions with indirect register addressing use the upd_{df} bit string to decode the $\{upd\_\zeta, upd\_\omega, upd\_\psi\}$ signals. These signals update the ζ, ω_m and ψ_m values within the TRCU. This approach offers the advantage of executing butterfly operations and updating a twiddle factor simultaneously within one clock cycle. The butterfly instructions include the bit field upd_{cf} to drive the $\{upd\_m, inc\_j, set\_idx, inc\_idx\}$ signals in the RAU. These signals are used to update the idx_0 and idx_1 signals, as explained in Section 3.4. As the butterfly instructions operate in-place, the *wd* and *wrd* is used for the upd_{cf} field.

4.3 Software Integration

The described ISA extension enables the OTBN to perform NTT-based polynomial multiplications and polynomial sampling in an efficient manner. We give a detailed explanation on how the proposed ISA extension can be leveraged for this purpose.

4.3.1 NTT-based Multiplication. To implement the control flow of forward and inverse NTT, the special purpose registers inside the RAU can be accessed over dedicated instructions. Additionally, these registers can be updated automatically either by dedicated update instructions or through the upd_{cf} field of arithmetic instructions with indirect register addressing. The data flow of both forward and inverse NTT mainly consists of CT and GS butterfly operations which are executed by the PQ-ALU. These operations are implemented via indirect register addressing. The pre-computed values of ω_n^p and ψ_n^p for the forward NTT (and ω_n^{-1} and ψ_n^{-1} for the inverse NTT) can be loaded into the ψ-register and ω-register within the TRCU by using dedicated instructions. The currently selected twiddle factor ζ as well as the values of ω_m and ψ_m can be updated automatically, either by a dedicated instruction or over the upd_{df} field of the butterfly instructions. To demonstrate the performance of our proposed solution we provide the clock cycle count for forward and inverse NTT for Kyber, Dilithium and Falcon polynomials in Table 1. It includes the reference implementations executed on the main processor of the OpenTitan, the Ibex, as a baseline. The measured performance improvement of our implementation can be explained mainly by three reasons. First, the minimization of memory accesses due to the enormous capacities of the WDRs within the OTBN. Second, the direct hardware support for the butterfly operations. Third, the concurrent execution of a butterfly operation and a twiddle factor update. After the NTT, the coefficients are converted into Montgomery domain via *pq.scale.ind* instructions and then multiplied pointwise via *pq.mul* instructions.

Table 1: NTT clock cycles for tightly coupled approaches

	Platform	NTT	NTT^{-1}	$\odot$ / $\circ$
Kyber [18]	RISC-V (PULPino)	1,935	1,930	N/A
Kyber [4]	RISC-V (VexRiscv)	6,868	6,367	2,395
Kyber [31]	RISC-V (CV6A)	18,488	18,488	N/A
Kyber baseline	RISC-V (Ibex)	37,975	54,189	16,417
Kyber [This Work]	OTBN	1,454	1,726	1,448
Dilithium [31]	RISC-V (CV6A)	18,554	21,375	N/A
Dilithium baseline	RISC-V (Ibex)	40,002	46,649	7,455
Dilithium [This Work]	OTBN	1,972	2,244	768
Falcon-512 [18]	RISC-V (PULPino)	8,169	8,684	N/A
Falcon-512 [4]	RISC-V (VexRiscv)	14,787	14,893	2,783
Falcon-512 baseline	RISC-V (Ibex)	134,047	137,727	14,875
Falcon-512 [This Work]	OTBN	5.172	5.712	1.512
Falcon-1024 [18]	RISC-V (PULPino)	18,537	20,171	N/A
Falcon-1024 [4]	RISC-V (VexRiscv)	31,295	31,735	5,472
Falcon-1024 baseline	RISC-V (Ibex)	296,109	301,334	29,723
Falcon-1024 [This Work]	OTBN	13,598	14,652	2,992

4.3.2 Keccak Round Function. For the representation of the Keccak state two WDRs are combined to represent one plane in the following manner: Even WDRs store four lanes while odd WDRs store only one lane within its 64 LSBs. In this way, ten WDRs are necessary to store the whole 1600-bit state. Although 192 bits are unused in each odd WDR this data layout makes state manipulations, especially plane-wise computations very simple. To implement the θ, ρ, π, χ and ι step we use the *bn.xor* instruction together with our custom Keccak instructions. More specifically, we use the *bn.xor* instruction and the *pq.parity* instruction to generate the parity plane in the θ step. The remaining part of the θ step is merged with the ρ and π step by exploiting the new *pq.xorr* instruction. For the χ and ι step our extension includes dedicated instructions, respectively. To reduce the number of necessary clock cycles, the *pq.xorr* instruction allows to update the values of x and y simultaneously. Furthermore, the *pq.iota* instruction allows to increment the idx_{rc} value concurrently to the execution of the ι step. Using our extensions together with the state representation mentioned above, leads to a clock cycle count of 40 for one Keccak round function.

5 RESULTS

This section discusses the results of this work which include the implementation of the Dilithium signature verification algorithm as well as synthesis results for FPGA and ASIC targets.

Table 2: Parameters of Dilithium (taken from [14])

NIST Security Level		2	3	5
Parameters				
q	modulus	8380417		
n	polynomial degree	256		
d	dropped bits from t	13		
τ	# of ± 1's in c	39	49	60
γ_1	y coefficient range	2^{17}	2^{19}	
γ_2	low-order rounding range	$(q-1)/88$	$(q-1)/32$	
(k,l)	dimensions of A	$(4,4)$	$(6,5)$	$(8,7)$
β	$\tau \cdot \eta$	78	196	120
ω	max # of 1's in h	80	55	75

5.1 Dilithium Signature Verification on the OTBN

Digital signatures are used to sign and assure the authenticity of a message. It is required that only the creator of a message is capable of producing a valid signature. PKC provides the means to construct such digital signature algorithms. The main idea is to use the private key to sign a message and to use the public key to verify the generated signature. In the context of securely booting a system, digital signatures are adopted to verify the authenticity of the system's firmware. To achieve this even with quantum computers around, quantum secure digital signature algorithms such as Dilithium must be adopted.

5.1.1 Verification Procedure. Dilithium supports the three NIST security levels 2, 3 and 5, for which the parameters slightly change. These parameters are summarized in Table 2.

The signature verification procedure of Dilithium is shown in Algorithm 2. In a first step, it generates the public module $\boldsymbol{A}$ via the ExpandA function. This function samples uniformly distributed polynomials using SHAKE128. Next, the digest μ is generated by means of the SHAKE256 function. The SampleInBall function regenerates the challenge polynomial c by invoking SHAKE128 again. The signature verifies correctly and is thus accepted only if the number of 1's in the hint $\boldsymbol{h}$ is smaller than ω and the challenge $\tilde{c}$ can be regenerated from the message M and $\boldsymbol{w_1'}$. Furthermore, the verifier checks if the signature is valid in the first place by checking that all coefficients of $\boldsymbol{z}$ have values less than $\gamma_1 - \beta$. For detailed information about Dilithium and the corresponding subfunctions the reader is referred to [14].

Data: pk, M, σ
1 $\boldsymbol{A} \in \mathbb{Z}_q^{k \times l}[x]/(x^n+1) :=$ ExpandA(ρ) ;
2 $\mu \in \{0,1\}^{512} :=$ SHAKE-256(SHAKE-256$(\rho || \boldsymbol{t_1}) || M)$;
3 $c \in B_\tau :=$SampleInBall$(\tilde{c})$;
4 $\boldsymbol{w_1'} :=$ UseHints$_q(\boldsymbol{h}, \boldsymbol{A} \cdot \boldsymbol{z} - c\boldsymbol{t_1}2^d, 2\gamma_2)$;
5 **return** $(||\boldsymbol{z}||_\infty < \gamma_1 - \beta)$ & $(\tilde{c} ==$SHAKE-256$(\mu || \boldsymbol{w_1'}))$ & (# of 1's in $\boldsymbol{h} \leq \omega$)

Algorithm 2: Dilithium verification (taken from [14])

5.1.2 Implementation. As a case study, we implemented Dilithium's signature verification procedure on the OTBN with our extensions in assembly language, benchmarked its performance and compared our results with state-of-the-art implementations.

In our implementation, the main processor of the OpenTitan unpacks the signature σ and the public key pk. Furthermore, μ is computed outside of the OTBN in the OpenTitan's HMAC core [26], as messages are allowed to be of arbitrary size and the Data Memory (DMEM) of the OTBN is limited. Afterwards, $\boldsymbol{z}$, $\boldsymbol{h}$, $\tilde{c}$, ρ, $\boldsymbol{t_1}$ and μ are transferred to the OTBN. The OTBN iteratively computes $\boldsymbol{w_1'}$ and derives the challenge $\tilde{c}'$ of it via SHAKE-256$(\mu || \boldsymbol{w_1'})$. If $\tilde{c}$ is equal to $\tilde{c}'$ then the signature verifies correctly. While the OTBN only checks if $\tilde{c}$ is equal to $\tilde{c}'$, the main processor of the OpenTitan is responsible for checking if the signature was valid in the first place when unpacking the signature.

We implemented Dilithium's verification procedure for all three NIST security levels. For that purpose, we extended the OTBN's Instruction Memory (IMEM) and DMEM to 8kB and 32kB, respectively. Table 3 compares our performance results with a baseline implementation for the Ibex and with previous works in terms of clock cycle count. The clock cycle count for the full implementation on the OpenTitan includes all pre-computation done outside of the OTBN, as well as all data transfers. In addition to that, we state the cycle counts of the accelerated computations within the OTBN separately. From Table 3 it can be seen that our implementation is consistently faster than optimized implementations for the Cortex-M4 [1] and the previous hardware/software co-design in [31]. While a certain performance speed-up compared to SW implementations can be expected, the advantage compared to [31] is most likely due to the fact that our design not only accelerates NTT-based polynomial arithmetic as in [31], but also the Keccak-based sampling operations. Nevertheless, the hardware/software co-designs in [22, 40] show better performance than our implementation. The design in [40] is implemented on a 64 bit high-performance core with matrix extensions and thus, cannot be directly compared. Comparing our results with [22], two aspects must be considered: First, our implementation unpacks the signature and regenerates μ outside the OTBN, while the work in [22] performs all the operations on a single processor with hardware acceleration. It can be seen that the computational part on the OTBN is less than half of the total cycle count of the whole signature verification and therefore, this preprocessing poses a major bottleneck in our implementation. Second, the PULPino integrates a 4-stage pipelined RISC-V processor, already yielding higher baseline performance than the 2-stage Ibex core in the OpenTitan.

The hardware extensions proposed in this paper reduce the cycle count to 35.0 %, 31.5 % and 27.6 % for NIST security level II, III and V, respectively, when compared to the baseline implementation on the Ibex. The improvement factors between the different security levels also underline the benefit of hardware acceleration for higher security levels, which is due to the increased computational effort that benefits from the accelerators.

5.2 Synthesis Results

The OpenTitan supports both an FPGA build flow as well as an ASIC build flow. Therefore, this section examines the synthesis

Table 3: Clock cycle count for Dilithium Verify

	Platform	CC
Dilithium-II [1]	Cortex-M4 (SW)	1,572,000
Dilithium-II [31]	CVA6 SoC	1,700,679
Dilithium-II [22]	PULPino	651,217
Dilithium-II [40]	RocketCore	89,800
Dilithium-II [Baseline]	Ibex (SW)	2,847,938
Dilithium-II [This Work]	OpenTitan	997,722
	⌞(OTBN)	(403,940)
Dilithium-III [1]	Cortex-M4 (SW)	2,692,000
Dilithium-III [31]	CVA6 SoC	2,963,936
Dilithium-III [22]	PULPino	1,126,938
Dilithium-III [40]	RocketCore	110,300
Dilithium-III [Baseline]	Ibex (SW)	4,721,794
Dilithium-III [This Work]	OpenTitan	1,488,526
	⌞(OTBN)	(684,270)
Dilithium-V [1]	Cortex-M4 (SW)	4,707,000
Dilithium-V [31]	CVA6 SoC	5,132,776
Dilithium-V [22]	PULPino	1,848,324
Dilithium-V [40]	RocketCore	160,200
Dilithium-V [Baseline]	Ibex (SW)	8,057,313
Dilithium-V [This Work]	OpenTitan	2,223,143
	⌞(OTBN)	(1,195,244)

Table 4: ASIC Synthesis - Area Results 22nm (Globalfoundries 22FDSOI)

	Total Area [μm^2]	Memory [μm^2]	Total Area NAND GE
Unmodified	1,567,683	511,207	4,710,587
Extended [This Work]	1,648,193	533,664	4,952,504

results for the Xilinx 7k410tfbg676-1 FPGA, the official OpenTitan evaluation platform, and for a 22nm ASIC design process. *

5.2.1 ASIC Results. For the ASIC design synthesis, this work uses the 22 nm Globalfoundries 22FDSOI technology. Table 4 shows the area for the original OpenTitan and the OpenTitan with the PQC extension for the OTBN. Compared to the original design only a slight increase in total area can be observed. Furthermore, the target clock frequency of 100 MHz for the OpenTitan is not influenced by the proposed OTBN extension.

5.2.2 FPGA Results. Table 5 shows the resource utilization of the OpenTitan and the single accelerators for the Xilinx 7k410tfbg676-1 FPGA. Furthermore, Table 5 provides a comparison with previous works. Table 5 shows that integrating the hardware accelerators into the OTBN increase the resource utilization of the OpenTitan only slightly. Compared to previous ISA extensions, our NTT acceleration offers a wider spectrum of twiddle update functionality and full configurability at runtime. This accounts for the slightly higher resource consumption. Compared to the Keccak accelerator presented in [18], our KU utilizes fewer resources. This is due to the fact that our KU does not implement a whole Keccak round in one clock cycle, but shares resources among the hardware elements implementing the single steps of a Keccak round function.

Table 5: FPGA Synthesis - Resource Utilization on Xilinx 7k410tfbg676-1 FPGA

Design	LUT	FF	DSP	BRAM
Unmodified	228,617	125,601	29	422
Extended [This Work]	234,954	126,607	62	427
PQ-ALU	1,830	0	11	0
TRCU	1,939	904	22	0
RAU	118	47	0	0
KU	1,312	0	0	0
NTT accelerator [18]	2,908	170	9	0
Keccak accelerator [18]	3,847	0	0	0
Finite field extension [4]	1,907	1,658	7	34

6 CONCLUSION

To tackle the design challenges associated with the adoption of PQC, recent works have applied different hardware/software co-design strategies. To this end, we provide a novel contribution by integrating hardware extensions for lattice-based PQC into a complete silicon RoT SoC. We used the OpenTitan platform and extended the OTBN, a dedicated PKC co-processor, with tightly coupled hardware accelerators for polynomial arithmetic and sampling. The resulting design comes at rather low resource overhead considering the overall size of the OpenTitan platform. Furthermore, we presented a case study which analyses our implementation with regards to post-quantum signature verification with Dilithium, one of the most critical tasks of a silicon RoT. Exploiting our OTBN extension leads to a significant reduction in clock cycle count compared to the baseline software implementation for Dilithium-II, -III and -V. Our work demonstrates that - while not trivial - it is possible to integrate lattice-based cryptography efficiently into complex systems such as a silicon RoT alongside to contemporary PKC standards. The resulting design can be used for hybrid signature verification as required by RoT functionalities to enable a secure transition to PQC.

ACKNOWLEDGMENTS

This work was partly funded by the German Federal Ministry of Education and Research (BMBF) in the project APRIORI under grant number 16KIS1390. The authors also acknowledge the financial support by the Federal Ministry for Economic Affairs and Climate Action of Germany in the project "PoQsiKom", project identification number: 13I40V010B.

REFERENCES

[1] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. 2022. Faster Kyber and Dilithium on the Cortex-M4. In *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13269)*, Giuseppe Ateniese and Daniele Venturi (Eds.). Springer, 853–871. https://doi.org/10.1007/978-3-031-09234-3_42

[2] Miklós Ajtai. 1996. Generating Hard Instances of Lattice Problems (Extended Abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 99–108. https://doi.org/10.1145/237814.237838

[3] Martin R. Albrecht and Amit Deo. 2017. Large Modulus Ring-LWE ≥ Module-LWE. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10624)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer, 267–296. https://doi.org/10.1007/978-3-319-70694-8_10

[4] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. 2020. ISA Extensions for Finite Field Arithmetic Accelerating Kyber and NewHope on RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 3 (2020), 219–242. https://doi.org/10.13154/tches.v2020.i3.219-242

[5] ANSSI. 2022. ANSSI views on the Post-Quantum Cryptography transition. https://www.ssi.gouv.fr/en/publication/anssi-views-on-the-post-quantum-cryptography-transition/.

[6] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 4 (2019), 17–61. https://doi.org/10.13154/tches.v2019.i4.17-61

[7] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. 2019. The SPHINCS+ Signature Framework. *IACR Cryptol. ePrint Arch.* (2019), 1086. https://eprint.iacr.org/2019/1086

[8] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–314.

[9] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. 2017. CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM. *IACR Cryptol. ePrint Arch.* (2017), 634. http://eprint.iacr.org/2017/634

[10] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. 2019. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 2 (2019), 199–224. https://doi.org/10.13154/tches.v2019.i2.199-224

[11] BSI. 2022. Technische Richtlinie: Kryptographische Verfahren: Empfehlungen und Schluessellaengen. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile.

[12] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[13] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. 2023. High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. *IEEE Trans. Computers* 72, 2 (2023), 306–320. https://doi.org/10.1109/TC.2022.3222954

[14] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 1 (2018), 238–268. https://doi.org/10.13154/tches.v2018.i1.238-268

[15] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2017. Falcon Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info/.

[16] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. 2022. Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 414–460. https://doi.org/10.46586/tches.v2022.i1.414-460

[17] Tim Fritzmann and Johanna Sepúlveda. 2019. Efficient and Flexible Low-Power NTT for Lattice-Based Cryptography. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*. IEEE, 141–150. https://doi.org/10.1109/HST.2019.8741027

[18] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 4 (2020), 239–280. https://doi.org/10.13154/tches.v2020.i4.239-280

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

[20] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. 2021. Verifying Post-Quantum Signatures in 8 kB of RAM. In *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20-22, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12841)*, Jung Hee Cheon and Jean-Pierre Tillich (Eds.). Springer, 215–233. https://doi.org/10.1007/978-3-030-81293-5_12

[21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. 2021. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 1 (2021), 1–24. https://doi.org/10.46586/tches.v2021.i1.1-24

[22] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. 2023. Post-Quantum Signatures on RISC-V with Hardware Acceleration. *ACM Trans. Embed. Comput. Syst.* (jan 2023). https://doi.org/10.1145/3579092 Just Accepted.

[23] Georg Land, Pascal Sasdrich, and Tim Güneysu. 2021. A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware. In *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13173)*, Vincent Grosso and Thomas Pöppelmann (Eds.). Springer, 210–230. https://doi.org/10.1007/978-3-030-97348-3_12

[24] Adeline Langlois and Damien Stehlé. 2015. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.* 75, 3 (2015), 565–599. https://doi.org/10.1007/s10623-014-9938-4

[25] lowRISC. 2018. Ibex: An embedded 32 bit RISC-V CPU core. https://ibex-core.readthedocs.io/en/latest/

[26] lowRISC. 2023. HMAC HWIP Technical Specification. https://opentitan.org/book/hw/ip/hmac/index.html

[27] lowRISC. 2023. OpenTitan. https://opentitan.org/

[28] lowRISC. 2023. OpenTitan Big Number Accelerator (OTBN) Technical Specification. https://opentitan.org/book/hw/ip/otbn/index.html

[29] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. 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, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6110)*, Henri Gilbert (Ed.). Springer, 1–23. https://doi.org/10.1007/978-3-642-13190-5_1

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

[31] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. 2021. A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. *IEEE Access* 9 (2021), 150798–150808. https://doi.org/10.1109/ACCESS.2021.3126208

[32] NIST. 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://doi.org/10.6028/nist.fips.202 10.6028/nist.fips.202.

[33] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9230)*, Kristin E. Lauter and Francisco Rodríguez-Henríquez (Eds.). Springer, 346–365. https://doi.org/10.1007/978-3-319-22174-8_19

[34] Hemendra K. Rawat and Patrick Schaumont. 2017. Vector Instruction Set Extensions for Efficient Computation of Keccak. *IEEE Trans. Computers* 66, 10 (2017), 1778–1789. https://doi.org/10.1109/TC.2017.2700795

[35] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM* 56, 6 (2009), 34:1–34:40. https://doi.org/10.1145/1568318.1568324

[36] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smékal, Jan Hajny, Peter Cíbik, Petr Dzurenda, and Patrik Dobias. 2021. Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs. In *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, Delphine Reinhardt and Tilo Müller (Eds.). ACM, 1:1–1:11. https://doi.org/10.1145/3465481.3465756

[37] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509. https://doi.org/10.1137/S0097539795293172

[38] Yufei Xing and Shuguo Li. 2021. A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 2 (2021), 328–356. https://doi.org/10.46586/tches.v2021.i2.328-356

[39] Jiming Xu, Yujian Wang, Juan Liu, and XinâĂŹan Wang. 2020. A General-Purpose Number Theoretic Transform Algorithm for Compact RLWE Cryptoprocessors. In *2020 IEEE 14th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 1–5. https://doi.org/10.1109/ASID50160.2020.9271722

[40] Yifan Zhao, Ruiqi Xie, Guozhu Xin, and Jun Han. 2022. A High-Performance Domain-Specific Processor With Matrix Extension of RISC-V for Module-LWE Applications. *IEEE Trans. Circuits Syst. I Regul. Pap.* 69, 7 (2022), 2871–2884. https://doi.org/10.1109/TCSI.2022.3162593