Towards ML-KEM & ML-DSA on OpenTitan

Amin Abdulrahman*, Felix Oberhansl†, Hoang Nguyen Hien Pham‡§, Jade Philipoom¶,
Peter Schwabe*‖, Tobias Stelzer† and Andreas Zankl†**
*Max Planck Institute for Security and Privacy (MPI-SP), Bochum, Germany
amin@abdulrahman.de, peter@cryptojedi.org
†Fraunhofer Institute for Applied and Integrated Security (AISEC), Garching, Germany
firstname.lastname@aisec.fraunhofer.de
‡BULL SAS, Les Clayes-sous-Bois, France
§Université Grenoble Alpes, CNRS, IF, Grenoble, France
hoang-nguyen-hien.pham@{eviden.com, univ-grenoble-alpes.fr}, nguyenhien.phamhoang@gmail.com
¶zeroRISC, Boston, USA
jadep@{zerorisc.com, opentitan.org}
‖Radboud University, Nijmegen, The Netherlands
**Technical University of Munich (TUM), Munich, Germany

***Abstract*—This paper presents extensions to the OpenTitan hardware root of trust that aim at enabling high-performance lattice-based cryptography. We start by carefully optimizing ML-KEM and ML-DSA—the two algorithms primarily recommended and standardized by NIST—in software targeting the OpenTitan Big Number (OTBN) accelerator. Based on profiling results of these implementations, we propose tightly integrated extensions to OTBN, specifically an interface from OTBN to OpenTitan's Keccak accelerator (KMAC core) and extensions to the OTBN ISA to support operations on 256-bit vectors. We implement these extensions in hardware and show that we achieve a speedup by a factor between 6 and 9 for different operations and parameter sets of ML-KEM and ML-DSA compared to our baseline implementation on unmodified OTBN. This speedup is achieved with an increase in cell count of less than 17% in OTBN, which corresponds to an increase of less than 3% for the full Earl Grey OpenTitan core.**

1. Introduction

In August 2024, NIST published final standards for three post-quantum cryptography (PQC) schemes: the signature schemes ML-DSA and SLH-DSA, and the key encapsulation mechanism ML-KEM. The publication of these standards marks an inflection point: large production systems are now beginning to migrate so that they can support PQC. As with classical cryptography, hardware specialization is a vital tool for increasing performance, and will have a large impact on the timeline and success rate of PQC migrations. The questions of how to optimize PQC on general purpose hardware and how to design PQC-specific accelerators are already very active research topics (e.g., [1], [2], [3]), so we choose to focus on a slightly different question: *how efficient can we make PQC on an existing pre-quantum platform with minor hardware changes?*

This question targets hardware/software co-designs for a secure microcontroller which cares about efficiently executing PQC. We focus on adding multipurpose processor instructions to accelerate small, generalizable mathematical operations that are useful across PQC schemes – even beyond the standardized ones – and also for pre-quantum cryptography. This preserves much more flexibility than a dedicated hardware accelerator because software using these instructions is updatable. If a scheme changes slightly or a new scheme is standardized, it will be possible to roll out those changes in software to run on the same hardware, whereas a pure-hardware solution would require a new silicon manufacturing run. Migration efforts that value support for hybrid cryptographic schemes or the possibility of adopting future changes of the PQC landscape will benefit from this flexibility in particular.

Further, we expect that any hardware deployed in the next decade will still require support for pre-quantum asymmetric cryptography, i.e., ECC and RSA. One reason is to support legacy applications, but a much more important reason is that sensible deployment of post-quantum cryptography today uses hybrid schemes that combine the novel algorithms with established pre-quantum algorithms. Applications that have already started integrating PQC, such as KYBER-enabled TLS by Google [4], Cloudflare [5], and Mozilla [6], the Signal secure messenger [7], and Apple's iMessage protocol [8], all use (pure-software) hybrid solutions. This is because PQC schemes are still relatively young and not recommended as a replacement for established classical schemes, by, e.g., ANSSI [9] and BSI [10]. This motivates us to implement NIST PQC standards on a target that *also* accelerates ECC and RSA.

We chose OpenTitan [11] as the target platform because of its accessibility and relevance. OpenTitan is a fully open-source silicon root of trust (RoT) that comprises a main 32-bit Ibex RISC-V core and a coprocessor called

OpenTitan Big Number (OTBN) with its own RISC-V-based instructions. It is set to soon replace the security chip of Chromebooks[1], highlighting the growing trend of deploying open-source security chips in real-world commercial products. OpenTitan today computes classical cryptography on the OTBN coprocessor, which is equipped with wide registers and various countermeasures against side-channel attacks. Therefore, we chose to explore the design space of modifying OTBN to efficiently support a wider range of cryptographic schemes, with the lattice-based ML-DSA and ML-KEM as a case-study.

Contribution. We ready the OpenTitan hardware RoT for ML-DSA and ML-KEM, and show that small modifications and extensions to the hardware design and instruction set architecture (ISA) of existing cryptographic hardware, designed to accelerate ECC and RSA, yields highly efficient accelerators for both traditional asymmetric and novel post-quantum cryptography. While ensuring this, we consider the generality of the proposed extensions as an additional goal to provide a flexible solution for cryptographic schemes beyond the two studied in this work. Our approach leverages multiple features of the OpenTitan platform in general and the OTBN unit in particular: First and foremost our research is made possible by the fact that OpenTitan is an open platform with the hardware implementation, software, and build system being publicly available under permissive licenses. Furthermore, OpenTitan already features a high-performance hardware implementation of Keccak, a central building block of both ML-DSA and ML-KEM. Also, the hardware/software co-design for ECC and RSA on OpenTitan uses a rather low-level ISA accelerating only (modular) big-integer arithmetic in hardware; higher-level routines like ECC point operations or exponentiation are implemented in software.

Outline. Section 2 recalls the necessary background on ML-DSA and ML-KEM, as well as the OpenTitan platform and its OTBN coprocessor.

Section 3 presents software-only implementations of ML-DSA and ML-KEM on OTBN with only an increase in OTBN's data memory (DMEM) and instruction memory (IMEM). It serves as a baseline for our performance evaluation and a starting point for profiling.

Section 4 tackles Keccak as it is the major bottleneck identified in our baseline. We resolve this by adding an interface from OTBN to the Keccak accelerator.

Section 5 addresses polynomial arithmetic as the remaining bottleneck, which we take on by proposing extensions to the OTBN ISA, allowing to operate on the 256-bit-wide registers as vectors of small integers. The extensions are designed as a generic vector instruction set rather than being specific to only ML-{KEM,DSA}, aiming to benefit other cryptographic schemes as well. This approach follows the design philosophy of the big-integer arithmetic instructions.

Section 6 presents the actual hardware implementation of our proposals introduced in the previous sections. For accelerating the polynomial arithmetic, we take two different approaches: one that maximizes resource sharing between big-number and vector arithmetic and one that investigates possible gains in performance of ML-{KEM,DSA} at the expense of larger investment in circuitry.

Our final evaluation shows that with an increase of OTBN's circuit area of 15.22%, we achieve a speedup of up to a factor of 9.14 in ML-DSA and 8.82 in ML-KEM. This increase corresponds to an increase of only 2.25% of the full OpenTitan *Earl Grey* top-level design. Details of these results and comparisons with related work is shown in Section 7 followed by a discussion in Section 8.

Artifact. All software and hardware described in this paper are publicly available under permissive licenses compatible to the OpenTitan license at https://github.com/PQC-OpenTitan/towards-ml-kem-and-ml-dsa-on-opentitan.

Related Work. Research in PQC implementations spans from pure software to pure hardware designs, across multiple platforms. Extensive studies have focused on software implementations of KYBER and DILITHIUM on Arm Cortex-M4 [12], [13], [14], [15], [16], [17], [18]. Highly optimized single-instruction-multiple-data (SIMD) implementations have been presented for, e.g., the Intel AVX2 [19], [20], Arm Neon [21], and RISC-V [22] platforms. Hardware/software co-design approaches offload compute-intensive operations to hardware, balancing between high performance and flexibility. An approach somewhat similar to ours on OTBN was explored in independent concurrent work [23]. The authors focus on accelerating the number-theoretic transform (NTT) of KYBER only and present results that are about $4\times$ slower while likely slightly better in terms of hardware cost when compared to ours. We provide a more detailed analysis in Section 7. Another notable work is a configurable post-quantum arithmetic logic unit (ALU) for OTBN [24], accelerating polynomial arithmetic of DILITHIUM, KYBER, and FALCON. An implementation of DILITHIUM's polynomial arithmetic on OTBN using the Kronecker+ method [25] is provided in [26].

Further tightly coupled accelerators for PQC, targeting different performance/resource trade-offs, have been presented in [27], [28], [29], [30], [31]. Among which, [27], [29], [30] provide hardware acceleration for polynomial generation using Keccak, while the others solely focus on speeding up the NTT-based polynomial multiplication and modular arithmetic. A less lightweight work [32] proposes a domain-specific processor optimized for module lattices. All of these designs extend the RISC-V ISA with scheme-specific instructions. A more generic approach involving masked accelerators is introduced in [33]. We will provide a detailed comparison to the most relevant literature, highlighting the differences to our work in Section 7.

2. Preliminaries

For notation, we mainly follow the conventions of NIST FIPS 203 [34] and 204 [35]. In addition, we define $[r]_d$ to be the unique element r' in the range $[0, 2^d)$ such that $r' \equiv r \bmod 2^d$ and denote $\lfloor \frac{r}{d} \rfloor$ as $[r]^d$, with $d \in \mathbb{N}$.

1. https://www.design-reuse.com/news/56335/nuvoton-lowrisc-opentitan-security-chip-chromebooks.html

2.1. NIST PQC Standards

2.1.1. ML-DSA. FIPS 204 [35], specifies the digital signature scheme DILITHIUM [36], [37] under the name module-lattice-based digital signature algorithm (ML-DSA). It is believed to fulfill the strong existential unforgeability under chosen-message attack (SUF-CMA) security property, even in the presence of powerful quantum computers [35]. Its security is based on the hardness of finding short vectors in a lattice [35]. More specifically, the problems it relies on are the module learning with errors (MLWE) and a variant of the module short integer solution (MSIS) problems. ML-DSA is constructed following the Fiat-Shamir with aborts pattern [38]. ML-DSA operates over the polynomial ring $R_q = \mathbb{F}_q[X]/\langle X^n + 1\rangle$ with $n = 256$ and $q = 8380417$. The scheme offers three security levels called ML-DSA-44, ML-DSA-65, ML-DSA-87, which vary in, e.g., their lattice dimension, as shown in Table A.1. The scheme makes heavy use of symmetric cryptographic primitives in the form of SHAKE128 and SHAKE256 as extendable output functions (XOFs) [39]. For a detailed description of the algorithms, we refer to Section B and [35]. For this work, we only consider the "internal" functions of ML-DSA, as they make up for the vast majority of the runtime and contain all relevant primitives.

2.1.2. ML-KEM. Similar to ML-DSA, the module-lattice-based key-encapsulation mechanism (ML-KEM), coined in FIPS 203 [34], is based on KYBER [40]. It is an indistinguishability under adaptive chosen ciphertext attack (IND-CCA2)-secure key encapsulation mechanism (KEM) obtained by applying a slightly tweaked Fujisaki-Okamoto transform [41] to the underlying indistinguishability under chosen plaintext attack (IND-CPA)-secure public-key encryption (PKE) scheme, denoted as K-PKE. Its security is based on the MLWE problem scaled for different parameter sets through the rank k of the module. For more details on K-PKE and ML-KEM we refer to [40]. The polynomial ring used in ML-KEM is also of the form $R_q = \mathbb{F}_q/\langle X^n + 1\rangle$ with $n = 256$ but uses $q = 3329$. Table A.2 lists other relevant parameters of ML-KEM with different security levels. Next to SHAKE128 and SHAKE256, the symmetric primitives at use in ML-KEM also include SHA3-$\{256, 512\}$ [39]. Just as with ML-DSA, we only consider the internal functions of ML-KEM in the following. A description of the algorithms can be obtained from Section B.

2.2. Number Theoretic Transform

The NTT is the discrete Fourier transform (DFT) on finite fields. Using algorithms by Cooley–Tukey (CT) [42] and Gentleman–Sande (GS) [43], also referred to as fast Fourier transform (FFT) algorithms, polynomial multiplication using the NTT can be implemented efficiently in $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$ using the general schoolbook method for $R_q = \mathbb{F}_q[X]/\langle X^n + 1\rangle$.

Assume that a primitive $2n$-th root of unity ζ exists. Then we have the ring isomorphism $R_q \cong \prod_{i=0}^{n-1} \mathbb{F}_q[X]/\langle X - \zeta^{2i+1}\rangle$. The forward and backward mapping are denoted as NTT and INTT respectively, where the latter stands for inverse number-theoretic transform (INTT). The roots of unity are called "twiddle factors". Multiplication of two polynomials $a, b \in R_q$ can be computed as "pointwise" multiplication of their "NTT representations":

$$a \cdot b = \mathsf{INTT}(\hat{a} \circ \hat{b}) = \mathsf{INTT}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$$

This approach can straightforwardly be applied to ML-DSA, computing $\log n = 8$ layers of NTT which amounts to a full splitting of the ring. For ML-KEM, we can only compute seven layers of NTT due to the absence of a $2n$-th root of unity. This implies that the multiplication inside NTT-domain is not pointwise, but on linear polynomials – thus also referred to as "pair-pointwise".

Algorithm 2.1: Montgomery Multiplication [45].

Input : $a, b \in [0, q)$, $q \in \left(0, 2^d\right)$, $R = \left[-q^{-1}\right]_d$
Output: $r = ab(2^{-d}) \mod q$ and $r \in [0, q)$

$c = ab$
$r = \left[c + \left[[c]_d\, R\right]_d\, q\right]^d$
if $r \geq q$ **then return** $r - q$
return r

Algorithm 2.2: Plantard Multiplication [44].

Input : $a, b \in [0, q]$, $q < \frac{2^{d+1}}{1+\sqrt{5}}$, $R = \left[q^{-1}\right]_{2d}$
Output: $r = ab(-2^{-2d}) \mod q$ and $r \in [0, q]$

return $r = \left[\left(\left[[abR]_{2d}\right]^d + 1\right) q\right]^d$

2.3. Modular Multiplications

The previously discussed polynomial arithmetic is based on modular arithmetic as the lowest-level building block. Algorithm 2.2 shows the Plantard multiplication [44] but omitting a final correction step, resulting in the output range $[0, q]$. Plantard's approach is usually faster than Montgomery's (cf. Algorithm 2.1) because the former requires one multiplication less in case $b \cdot R$ in Algorithm 2.2 is precomputed. In exchange, this doubles the size of the second input to $2d$-bit instead of d-bit as in Algorithm 2.1. The work from [12] improves the algorithm by extending the input range of the Plantard multiplication while introducing signed arithmetic.

2.4. OpenTitan

OpenTitan is a project building a RISC-V-based open-source silicon RoT stewarded by lowRISC, with collaborative engineering from ETH Zürich, Google, G+D Mobile Security, Nuvoton Technology, Western Digital, and zeroRISC to develop and maintain the open-source silicon design [11]. It consists of several hardware intellectual property (IP) blocks, together with a main 32-bit Ibex

RISC-V core and a big-number coprocessor, called OTBN, accelerating classical asymmetric cryptography, making up the Earl Grey microcontroller [46]. The majority of IP blocks are dedicated to cryptographic operations, including Keccak message authentication code (KMAC), HMAC-SHA2, AES [47], and a cryptographically secure random number generator (CSRNG) together with an entropy source IP block enabling the generation of deterministic or true random numbers compliant to, e.g., NIST standards.

2.4.1. OTBN. OTBN [11] is designed to securely accelerate classical asymmetric cryptography such as RSA [48] and elliptic-curve cryptography [49], [50]. It is equipped with enhanced countermeasures against Spectre BHB [51] vulnerabilities [11, Section 8.2.2], cache-timing attacks [11, Section 8.2.2], side-channel leakage [11, Section 7.2.2] and fault injection [11, Section 8.2.2]. These countermeasures include data-path blanking, Hsiao integrity-protection code, and secure wiping of OTBN's internal states and memories [52]. Data-path blanking forces unused data paths to zero, preventing sensitive data from producing power leakage over those paths. To protect data from modifications through physical attacks, Hsiao protection code [53] adds seven parity bits to each 32-bit data chunk, allowing detection of at least three errors. Upon detecting an integrity violation, OTBN halts all processing immediately. The secure wipe mechanism clears all stored state from register files, instruction and data memory, to prevent leftover data leakage.

OTBN offers a 32-bit RISC-V-based ISA with 32 general-purpose registers (GPRs) `x0` to `x31` used by the "base instruction subset" for the control flow of an OTBN application, and a custom big-number ("bn") ISA providing 32 256-bit wide data registers (WDRs) `w0` to `w31` used by the "big number instruction subset" for data processing. By convention, we refer to a WDR zero as `bn0`. In addition, there are control and status registers (CSRs) and wide special-purpose registers (WSRs) that give access to randomness sources, arithmetic flags, key material, a special "modulus register" `MOD`, and an "accumulate register" `ACC` [11, Section 8.2]. Note that even though the GPRs and their relating instructions are inspired by the RISC-V integer extension RV32I, compilers and toolchains for RISC-V are not compatible with OTBN and no dedicated toolchains for higher-level languages are available. Consequently, all code for OTBN in this work is written in assembly and run with its cycle-accurate Python simulator.

2.4.2. KMAC Block. The KMAC core supports Keccak-based message authentication codes (MACs), unauthenticated SHA-3 and (c)SHAKE [11, Section 9.13]. The hardware design offers a synthesis-time option for enabling or disabling first-order masking [11, Section 9.13.1], including Domain-Oriented Masking [54] and masking data before loading into Keccak. The masked Keccak-f[1600] takes four cycles per round for a number of 24 rounds, resulting in 96 cycles in total [11, Section 9.13], while the unmasked one takes just one cycle per round. However, in this work, we will assume that the version with first-order masking has been chosen, as it is likely to be more popular in practice and provides a more conservative performance estimate.

3. Implementation on Plain OTBN

This section describes an implementation of ML-{KEM,DSA} on an essentially unmodified OTBN, serving as a performance baseline and a starting point for profiling. The only change required is to increase the DMEM size to 128 kB and the IMEM size to 32 kB, from their original sizes of 4 kB each. We describe optimization techniques for the most important parts of ML-{KEM,DSA}: modular arithmetic, NTT/INTT, and multiplication in NTT domain. We also provide a pure software implementation of Keccak-f for OTBN.

The reasons we chose to implement Keccak in software are twofold: (1) For leveraging the KMAC core, we need to transfer data out of OTBN to the Ibex core, to KMAC, and back, which exposes secrets over the less protected transit bus twice. We believe this setup increases the attack surface unnecessarily, making it more vulnerable compared to executing the schemes entirely within OTBN, providing robust protection mechanisms. (2) OTBN is not designed for interrupting the operation, yielding to Ibex, and continuing later. For example, neither the register state, nor the call stack is retained upon exit of OTBN.

We provide more in-depth explanation for many of our optimization strategies in Section C. This includes more details on the descriptions here, as well as on the sampling and bit-packing aspects which we omit here.

3.1. NTT and Multiplication in NTT Domain

3.1.1. Modular Multiplication. For modular multiplications, we choose the original Plantard multiplication without the final correction step (cf. Algorithm 2.2) for our baseline implementations of ML-{KEM,DSA} on OTBN over the improved version from [12]. This is because computation with unsigned integers on OTBN is less complex compared to signed numbers, which require more effort for a correct sign extension. In addition, since none of the improvements in [12] are relevant to our implementation on OTBN, we opt for the original.

Let d be 32 for ML-DSA and 16 for ML-KEM. A polynomial in either scheme is represented by a vector of n coefficients with d bits each and polynomial arithmetic breaks down to modular arithmetic on d-bit unsigned integers. To multiply two elements a and b in $\mathbb{F}_q$ using Plantard multiplication [44] we require three multiplication instructions, one right shift, and one addition in the case of ML-DSA, while for ML-KEM an additional logical AND is required. This amounts to 5 cycles per modular multiplication in the case of ML-DSA, and 6 cycles for ML-KEM. In case the second factor b is pre-multiplied by R, which normally happens in the NTT, we need one instruction less for both schemes (cf. Listing A.1, Line 6 to 9).

3.1.2. NTT. For the NTT, we apply common optimizations, e.g., merging the multiplication with n^{-1} into the last twiddle factor in the INTT [55, Sec. 3], making up for the omitted transformation into Plantard representation during the multiplication with n^{-1} [56, Sec. 5.3] as well as transforming the twiddle factors into the proper domain for the deployed modular multiplication strategy ahead of time [57, Sec. 7.2].

CT and GS Butterfly. We follow the original approach from [19], [20] to use the CT butterfly for the NTT and the GS butterfly for the INTT. A butterfly consists of a Plantard multiplication (cf. Section 3.1.1) between a coefficient and a twiddle factor, a modular addition `bn.addm`, and a modular subtraction `bn.subm`. As inputs and outputs of these two instructions are in $[0, q]$, which is due to the Plantard multiplication (cf. Algorithm 2.2), the outputs of each layer are certain to be in $[0, q]$ as well, inhibiting growth throughout the computation. Therefore, we do not require lazy reductions. The twiddle factors are already stored in Plantard representation, saving one multiplication (cf. Algorithm 2.2). Subsequently, a CT butterfly takes six and seven cycles for ML-DSA and ML-KEM, respectively. Listing A.1 shows how to extract data for a CT butterfly in ML-DSA and store the results back to the registers – incurring a large overhead for data movement.

Layer Merge. In this work, we also adopt a 4–4 layer merge for ML-DSA and 4–3 for ML-KEM, making use of OTBN's WDRs for reducing the memory accesses.

3.1.3. Multiplication in NTT Domain. The pointwise multiplication in ML-DSA consists of n modular multiplications in $\mathbb{F}_q$ ($q = 8380417$), which is equivalent to n Plantard multiplications as described in Section 3.1.1. For ML-KEM, a pair-pointwise multiplication consists of multiplications of 128 pairs of linear polynomials, each of which requires five Plantard multiplications.

Pseudo Vectorization. Pointwise addition of any two vectors requires extracting individual coefficients into two separate WDRs for performing addition, which causes a disproportionately high overhead for data movement. "Pseudo vectorization" means adding two vectors of n/d coefficients using the non-vectorized addition instruction `bn.add` and obtain the result of a vectorized one which is possible since the sum of two $\log(q)$-bit integers will not exceed d bits. This is used in the accumulation of the matrix-vector product in ML-{KEM,DSA}. While the accumulated outputs must be manually reduced to $[0, q]$, this technique greatly improves performance.

3.2. Keccak on OTBN

For our software implementation of Keccak on OTBN, we aimed to make use of the available resources as efficiently as possible. The fact that Keccak-f relies on logical AND, XOR, and NOT operations allows for pseudo vectorization, leveraging the wide registers of OTBN for improved efficiency. To achieve this, we carefully arrange 25 lanes within seven WDRs. An exception to this is the $\rho - \pi$ step, where we need to compute on individual lanes.

3.3. Profiling

Figure 1a and Figure 1b present a heatmap table illustrating the cycle count percentages for ML-{KEM,DSA}. Hashing emerges as the most time-consuming operation in both schemes, aligning with the profiling results in previous works [12, Table 6] [58]. This observation prompts us to leverage OpenTitan's KMAC core for potential optimization.

Percent	Poly.	Samp.	Hash	Pack	Round	Red.
K	8	12	77	1	1	1
S	29	8	55	1	3	3
V	12	10	74	1	2	1

(a) ML-DSA-65

Percent	Poly.	Samp.	Hash	Pack
K	18	11	70	1
E	21	11	66	2
D	27	10	60	3

(b) ML-KEM-768

Figure 1: Cycle count profiling on OTBN, median values over 10 000 iterations. Groups with less than 1% not displayed. May not add up to 100% due to rounding.

3.4. Reflection

In this section, we reflect on the implementation process of ML-{KEM,DSA} on OTBN from an ISA-perspective.

A clear benefit of the big-integer arithmetic capabilities is the ability to load large amounts of data, i.e., 256 bits, within just two clock cycles. Further, many "bitwise" operations can profit from the wide registers as it is possible to compute on lots of data in parallel. Even simple arithmetic operations can profit from this in certain instances, as we have shown with the application of pseudo-vectorization in Section 3.1.3.

However, more complex operations like multiplications are not amenable to this strategy. As mentioned in Section 3.1.1, at least four instructions are required for a modular multiplication of two coefficients – not counting the four instructions required to extract the individual coefficients from the WDRs they had been initially loaded to. This implies two things: (1) The same number of cycles required for the modular multiplication itself is additionally spent on data movement, and (2) for many of the operations, the WDRs are barely utilized, only using the first 16 or 32 bits out of 256. In contrast, common microcontroller architectures are equipped with digital signal processor (DSP) extensions that allow parallelism over the data inside a register and simply more flexible ISAs which oftentimes help to express a desired computation in a shorter instruction sequence. For example, the Arm Cortex-M4 can compute one Plantard multiplication with just two instructions, taking only two cycles [12]. As we will see in Section 7, this also leads to our plain implementation presented in this section remaining behind the Cortex-M4's performance.

Summing up, OTBN does generally offer powerful hardware components, but the ISA is not well suited for the kind of operations that are required for ML-{KEM,DSA}.

4. Implementation on OTBN With Keccak Acceleration

Based on the profiling results from Section 3.3, we decided to explore the hardware/software co-design approach by interfacing between OTBN and OpenTitan's KMAC core, referred to as OTBN$^{\text{KMAC}}$ in the following.

KMAC Interface. The KMAC core in OpenTitan is accessible via the main TL-UL bus and application interfaces for the key manager, life-cycle controller, and ROM controller hardware blocks. The simplicity and high throughput of the application interface make it an ideal choice for connecting KMAC with OTBN. Therefore, we introduce an additional one to enable this direct connection. All interfaces use a 64-bit data path with simple control logic and status signals. The KMAC core outputs the digest as two boolean shares on a parallel data path.

On the KMAC side, only minor modifications are required, such as enabling dynamic configuration of the hash algorithms to support SHA3-{256,512}, SHAKE{128,256}. We did not implement side-channel protections on top of the already existing ones mentioned in Section 2.4.2 as our changes to KMAC do not require them. On OTBN side, special-purpose registers for KMAC configuration, message, status, and digest are added. The status register, controlled by KMAC signals, allows OTBN to determine if KMAC is ready for operation. The configuration register contains the hash function and the length of the data to be processed. These registers can be written and read with already present instructions (`bn.wsrr`, `bn.wsrw`, `csrrw`). Data is sent to KMAC by writing to the 256-bit message register, which is connected to a small FIFO that outputs 64-bit words to KMAC's application interface. If the FIFO has not yet consumed all contents of the message register while a new instruction is being fetched and decoded, the pipeline stalls. The input width and depth of the FIFO can be optimized for transfer efficiency. For our case study, we selected a small FIFO which is capable of holding four 64-bit words but can consume a complete 256-bit word within a single cycle. Since these modifications are related to OTBN, we implemented them in accordance with the standard OTBN countermeasures mentioned in Section 2.4.1. These include blanking of unused data paths, as well as a secure wipe mechanism and integrity-protection codes for the additional special-purpose registers. Figure 5 illustrates that the KMAC interface connects OTBN's BN-ALU module, which handles special registers, to the application interface of the KMAC core.

Python Simulator of KMAC Interface. We extend the OTBN Python simulator to implement said interface and to match the performance characteristics of the actual hardware, integrating the special purpose registers for configuration, status, message and digest introduced above.

4.1. Profiling

We now examine the profiling results assuming that OTBN interfaces directly with KMAC. As shown in Figures 2a and 2b, the time spent on hashing is drastically reduced thanks to the powerful KMAC core. In ML-DSA, most of the runtime is now spent on polynomial arithmetic. For ML-KEM, polynomial arithmetic constitutes an even larger portion. This indicates that we could achieve a further substantial reduction in runtime by accelerating polynomial arithmetic on OTBN$^{\text{KMAC}}$.

Percent	Poly.	Samp.	Hash	Pack	Round	Red.	Other
K	41	42	5	6	3	3	0
S	68	13	2	3	7	8	0
V	51	26	4	3	10	4	1

(a) ML-DSA-65

Percent	Poly.	Samp.	Hash	Pack
K	64	30	3	2
E	67	25	3	5
D	70	19	2	8

(b) ML-KEM-768

Figure 2: Cycle count profiling on OTBN$^{\text{KMAC}}$, median values over 10 000 iterations. Groups with less than 1% not displayed. May not add up to 100% due to rounding.

5. Extending the OTBN ISA

This section introduces the changes to the OTBN ISA that we propose based on our observations from Section 3. We will refer to OTBN including the KMAC interface and our proposed instructions as OTBN$^{\text{KMAC}}_{\text{Ext.}}$.

Goal of the ISA Extensions. The main goal of the ISA extensions is to broaden the range of cryptographic primitives that can be accelerated using OTBN, with a focus on the lattice-based ML-{KEM,DSA} in this work. More specifically, we aim to reduce the time spent on polynomial arithmetic identified as the predominant remaining bottleneck in Section 3 after hashing was addressed. We work towards that goal by providing instructions that speed up common, elementary arithmetic and bit manipulation operations. For a discussion of further, possible applications see Section 8. This is in contrast to hash functions, which require parallel processing of large bit vectors which make dedicated coprocessors more suitable as long as transfer latency does not become an issue [59]. An additional goal we set is to keep the number of new instructions to a minimum as OTBN is a RISC-based architecture and the 32-bit instruction encoding makes opcodes scarce. The primary metric for evaluating the extensions is the performance in terms of cycle count – while we also comment on the memory usage and code size, we did not specifically optimize for them. For a discussion on the impact of memory optimizations, we refer to Section 8.

5.1. Proposed Instructions

In the following, we argue why we deem the addition of SIMD instructions as a promising approach to circumvent some of the previously identified bottlenecks and to improve the performance of the polynomial arithmetic. First, we have noticed in Section 3 how much time is spent on data movement and also how much performance gain could be achieved through the pseudo-vectorization strategy (cf. Section 3.1.3). Second, the NTT and INTT naturally lend themselves to parallelization because of the independence of the individual butterfly operations on each layer. Lastly, prior work has shown that the performance of polynomial multiplication can be greatly improved with SIMD instructions, e.g., using Intel AVX2 [37], Arm Neon [21], or the RISC-V vector extension [22].

We propose five new instructions, each with multiple subvariants. Following the reasoning above, the first three are `bn.addv`, `bn.subv`, and `bn.mulv`, offering SIMD (modular) addition, subtraction, and multiplication, respectively. Note that although `bn.mulv` is highly similar to the one in [60], it was developed independently. The fourth instruction interleaves data inside two WDRs when interpreting them as vectors of multiple elements. While it is a staple in SIMD instruction sets, it is especially useful for NTT and INTT in our case (see Section 5.2). Lastly, we propose a bit-shifting instruction, a core operation in most (SIMD) instruction sets, beneficial for various functions of ML-{KEM,DSA}. For example, it allows fully vectorizing decomposition in ML-DSA and facilitates sampling coefficients in $[-\eta, \eta]$. In the more detailed description below, `<type>` defines the subvariants of the instruction, including the operation on vector elements of different sizes, e.g., `.8S` for a 32-bit or `.16H` for a 16-bit element view. Furthermore, the `m` suffix indicates a variant that includes (pseudo) modular reduction.

- `bn.addv<type> <wrd>, <wrs1>, <wrs2>`: Vectorized addition with optional conditional subtraction. `<type>` can be `(m){.8S,.16H}`. Each pair of d-bit elements in the source registers `<wrs1>` and `<wrs2>` is added together and stored to the respective element in `<wrd>`. The result is truncated in case of an overflow. If `m` is set in `<type>`, value defined in the `MOD` register is subtracted from the result in case it is greater than or equal to `MOD`.
- `bn.subv<type> <wrd>, <wrs1>, <wrs2>`: Vectorized subtraction with optional conditional addition. `<type>` can be `(m){.8S,.16H}`. This instruction functions similarly to `bn.addv`, but with subtraction. `MOD` is added to the subtraction result in case it is negative.
- `bn.mulv<type> <wrd>, <wrs1>, <wrs2>[, <lane>]`: Vectorized multiplication with optional modular reduction. `<type>` can be `(m)(.l){.8S,.16H}`. `l` specifies a lane-wise mode of operation, meaning that instead of the element-wise multiplication, all elements of `<wrs1>` are multiplied with a fixed element of `<wrs2>` at index `<lane>` in $\left[0, \frac{n}{d} - 1\right]$. Next, the result is either truncated or reduced $\bmod^{+}$ `MOD` in case `m` is set in `<type>`.
- `bn.trn1`/`bn.trn2<type> <wrd>, <wrs1>, <wrs2>`: Interleaving of even/odd indexed vector elements. For this instruction, `<type>` can also be `.4D` (for 64-bit elements) and `.2Q` (for 128-bit elements), alongside with `.8S` and `.16H`.
- `bn.shv<type> <wrd>, <wrs> <shift_type> <shift_bits>`: Bitwise logical shift operation of individual vector elements. `<type>` can be `.8S` or `.16H`. `<shift_type>` defines whether to perform a left (<<) or right (>>) shift. `<shift_bits>` is the number of bits to shift each element.

5.2. Impact on ML-{KEM,DSA} Implementations

This section discusses how our proposed extensions influence the implementation of ML-{KEM,DSA} and illustrates the most important subroutines.

Just as for OTBN, we provide additional descriptions on our implementation in Section D.

5.2.1. Polynomial Addition & Subtraction. A significant impact of our proposed instructions can be observed in functions related to polynomial arithmetic. The cumbersome extraction of individual coefficients from the WDRs can be replaced by a simple sequence of a load, the SIMD addition, and a store. This reduces both arithmetic costs and data movement overhead.

5.2.2. NTT & INTT. For the NTT, the code drastically simplifies as well. As we provide a dedicated instruction for modular multiplication, the need for applying Plantard's algorithm is eliminated.

ML-DSA. Assuming the computation of the forward NTT in ML-DSA, the code for the CT-butterfly can be reduced down to three instructions for computing eight butterfly operations in parallel. The implementation of the GS-butterfly in the INTT is similar. Regarding the layer merge, we proceed with the same 4–4 merge as in the plain implementation of ML-DSA. The approach to vectorization we take is closely related to the one taken in [21]. To apply the data transposition in order to adjust for the coefficient's stride-requirement on later layers, we use a sequence of `bn.trn1` and `bn.trn2` instructions, following the strategy from [21].

ML-KEM. For ML-KEM, the butterfly simplifies as for ML-DSA, except that the `.16H` variants are used. Thanks to the availability of 32 256-bit WDRs, all seven layers of the ML-KEM NTT can be merged, reducing the memory operations to a minimum. However, a similar "transposition" with `.8S` variant as with ML-DSA is required in order to compute the last three layers.

5.2.3. Multiplication in NTT Domain. Thanks to `bn.mulv(m)`, the base multiplication for ML-{KEM,DSA} becomes directly vectorizable.

ML-DSA. As the base multiplication in ML-DSA is a pointwise multiplication, its computation using `bn.mulvm` is trivial: we load one WDR of each input polynomial, multiply them via `bn.mulvm`, and store the result into the result polynomial.

ML-KEM. In ML-KEM, the pair-pointwise multiplication adds slight complexity due to multiplying even/odd indexed coefficients of the linear polynomials, which do not "align" inside the vectors. To handle this, we make use of our `bn.trn` instructions to re-order the coefficients accordingly. See Listing D.3 for a concrete implementation of this strategy.

5.2.4. Profiling. The profiling data on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, as shown in Figures 3a and 3b, demonstrates that our ISA extensions fulfill their task by reducing the cycle count for polynomial arithmetic by up to 46 percentage points.

5.2.5. Suitability for Masking of ML-{KEM,DSA}. Recognizing the importance of protections against side-channel attacks, we briefly touch on the applicability of our ISA extensions for a possible masked implementation of ML-KEM and ML-DSA. Naturally, our vectorized instructions are advantageous for masking linear operations, such as the NTT. While additional instructions may be useful to further accelerate non-linear parts, it is beyond the scope of this work and left as future research. Regarding a sensible masking strategy for ML-{KEM,DSA} on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, we believe that shares of the same value should not be combined within a single WDR, but rather shares of different values as recommended in [61]. Since these shares have no data dependency, this approach does not increase the leakage and may even introduce something similar to additional switching noise.

6. Hardware Implementation

This section describes our hardware implementations and the modifications we applied to the OTBN architecture and its components. All our modifications were implemented with consideration for the countermeasures mentioned in Section 2.4.1, namely data-path blanking, secure wipe of registers and Hsiao integrity-protection codes.

Percent	Poly.	Samp.	Hash	Pack	Round	Red.	Other
K	19	66	8	6	1	0	0
S	52	27	5	6	7	2	1
V	27	48	8	5	11	0	1

(a) ML-DSA-65

Percent	Poly.	Samp.	Hash	Pack	Other
K	19	67	8	5	1
E	21	60	7	12	0
D	24	50	6	19	1

(b) ML-KEM-768

Figure 3: Cycle count profiling on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, median values over 10 000 iterations. Groups with less than 1% not displayed. May not add up to 100% due to rounding.

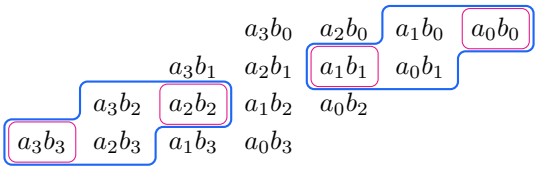

Figure 4: Configurable vectorized multiplication.

6.1. Basic Building Blocks

To enable vectorized 32-bit and 16-bit operations with minimal resource usage, we design basic building blocks capable of performing both.

6.1.1. Configurable Vectorized Adder. We adapt the existing 256-bit adders in the big number arithmetic logic unit (BN-ALU) for OTBN to enable 32-bit and 16-bit vectorized (modular) addition and subtraction, along with 256-bit (modular) addition/subtraction. We split a 256-bit addition into 16 $\times$ 16-bit additions, adding multiplexers for carry inputs. This allows for variable adder sizes based on carry propagation. For the 256-bit addition (used in `bn.add`), each 16-bit adder's carry input connects to the previous adder's output carry. For 16-bit vectorized addition, the carry inputs are set to the input carry of the 256-bit adder. For 32-bit vectorized addition, two 16-bit adders are linked, connecting the first adder's output carry to the second's input carry. Depending on the size of the operations, input carries to either one 256-bit adder, one 32-bit adder (.i.e, two connected 16-bit adders) or one 16-bit adder are used to differentiate between addition and subtraction.

6.1.2. Configurable Vectorized Multiplier. While our multiplier accepts two 64-bit operands as inputs and outputs a 128-bit result, it can compute either one 64-bit, two 32-bit or four 16-bit multiplications. Our approach can be generalized for arbitrary-width multiplications. We achieve this by splitting one 64$\times$64-bit product into several partial products, where each is the product of two 16-bit chunks. When adding all partial products together, the result of the 64-bit multiplication is obtained. By only considering certain partial products and zeroing irrelevant ones, we achieve vectorized {16,32}-bit multiplications. Figure 4 shows that each 64-bit operand is split into 16-bit chunks: $a = (a_3||a_2||a_1||a_0)$ and $b = (b_3||b_2||b_1||b_0)$. For 64-bit multiplication, each 16-bit chunk of a is multiplied by that of b, and the resulting partial products are summed to produce the final 128-bit result c. For vectorized 32-bit multiplication, two adjacent 16-bit chunks form the 32-bit operands, i.e., $a = a'_1||a'_0$, where $a'_1 = (a_3||a_2)$ and $a'_0 = (a_1||a_0)$. The result is then calculated as $c = (a'_1 \times b'_1||a'_0 \times b'_0)$, ignoring products from $a'_1 \times b'_0$ and $a'_0 \times b'_1$ as shown in the blue frames. Similarly, $c = (a_3 \times b_3||a_2 \times b_2||a_1 \times b_1||a_0 \times b_0)$,

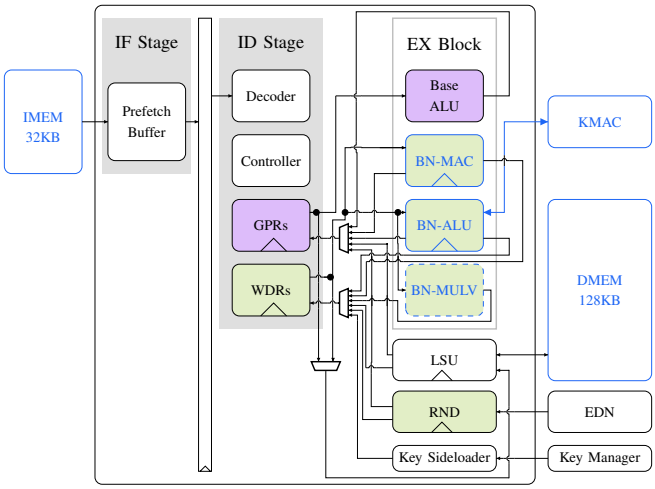

Figure 5: Simplified diagram of the modified OTBN pipeline. Light gray blocks represent the pipeline stages and their components. Lilac and green blocks correspond to GPR- and WDR-related components, respectively. Our modifications are highlighted in blue, while the dashed border indicates the newly added optional module.

for vectorized 16-bit multiplication, considering only the partial products within the pink frames.

6.2. Integration Into OTBN Architecture

This section outlines the modifications necessary for integrating our proposed modules, aiming at maximizing resource sharing while maintaining architectural simplicity for effective decoding and control. Figure 5 provides a high-level overview of where our changes are added within OTBN.

Our new configurable vectorized adder (cf. Section 6.1.1) will replace the current 256-bit adders in BN-ALU, enabling the `bn.addv`, `bn.addvm`, `bn.subv`, `bn.subvm` instructions. Similarly, we integrate the functionalities for `bn.shv` and `bn.trn1`/`bn.trn2` instructions also into the existing BN-ALU as it already contains functional units for shifting.

For integrating the `bn.mulv(m)` instruction, two options exist: reusing existing resources in the big number multiply accumulate unit (BN-MAC) at the cost of latency versus integrating a new module into OTBN's pipeline leading to increased resource consumption but also enhanced performance. We refer to implementations of ML-{KEM,DSA} using the modified BN-MAC as $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, and the latter high-end approach as $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$.

For the first option, we replace BN-MAC's 64-bit multiplier and 256-bit adder with our basic building blocks (see Sections 6.1.1 and 6.1.2) to enable vectorized multiplications and additions while maintaining support for the original `bn.mulqacc` instruction. To minimize the resource overhead, we split `bn.mulv(m)` into several cycles and reuse existing computational resources, requiring additional control logic within the BN-MAC, decoder, and controller to manage the pipeline stalls and keep all redundancy checks throughout the pipeline in sync. This approach keeps the control path relatively simple and integrates well with OTBN's architecture. While according to the OTBN design rationale, all instructions should complete within a single cycle, some mechanisms stall the pipeline for loads or if the internal randomness register does not contain fresh randomness. The KMAC interface may also stall the pipeline (see Section 4). Although one could add an instruction for each execution stage of `bn.mulv(m)` (see Section 6.2.2), we do not explore this further since this is a trade-off between code and hardware complexity. Modifications to the BN-MAC and details on the multi-cycle approach are outlined in Section 6.2.2.

For the second option, we outsource this operation into a new separate module capable of executing `bn.mulv(m)` in a single cycle, ensuring clean and straightforward integration, particularly regarding decoding and control logic. The big number vector multiplier (BN-MULV) module and its integration is described in Section 6.2.3.

6.2.1. Modified Big Number ALU. We replace the two 256-bit adders in BN-ALU, namely `Adder X` and `Adder Y`, with two configurable vectorized adders from Section 6.1.1, allowing 16-bit, 32-bit, and the original 256-bit operations with minimal overhead. For `bn.addv` and `bn.subv`, only `Adder X` computes either $x = a + b$ or $x = a - b$. For `bn.addvm`, `Adder X` computes $x = a + b$ and `Adder Y` performs $y = x - q$. Depending on $x < q$, either output from `Adder X` or `Adder Y` is selected based on carry propagation. Modular subtraction with `bn.subvm` is handled similarly, selecting outputs based on $x < 0$. Carry propagation varies, depending on whether 256-bit, 32-bit, or 16-bit operations are desired. Note that basically every carry bit which is not propagated to the next 16-bit adder is effectively an output carry and, therefore, responsible to select the results from either `Adder X` or `Adder Y`.

We also integrate transpose functionality for `bn.trn1`/`bn.trn2` and vectorized shifts for `bn.shv` into BN-ALU. Both operations require minimal additional hardware. Lastly, four special-purpose registers for Keccak integration are added as detailed in Section 4.

6.2.2. Modified Big Number MAC. We replace the 64-bit multiplier within the BN-MAC with our configurable vectorized multiplier (cf. Section 6.1.2). Similarly, we replace the original adder with our configurable vectorized adder (cf. Section 6.1.1). As explained above, one `bn.mulv(m)` is split into several clock cycles. In the following, we describe the different execution stages for the different `bn.mulv(m)` variants. For `.8S` variants, $d = 32$ and for `.16H` variants, $d = 16$. The resulting architecture of our modified BN-MAC unit is depicted in Figure 6. As for `bn.mulvm` specifically, Montgomery multiplication (cf. Algorithm 2.1) is implemented, instead of Plantard, because it does not expand the input size, which fits the context of the vectorized multiplication instruction perfectly. To configure q and R,

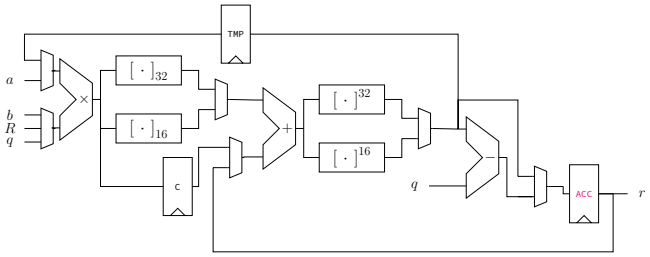

Figure 6: Architecture of our modified BN-MAC module.

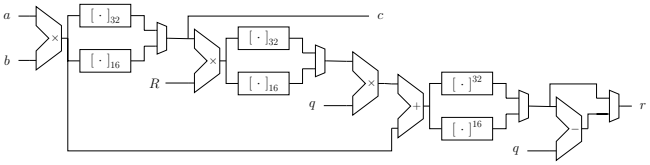

Figure 7: Architecture of our proposed BN-MULV module.

respectively, we added a dedicated connection from the MOD register within the BN-ALU to the BN-MAC module and use sub-words of this register.

Execution Stages for bn.mulvm. This instruction takes two 256-bit WDRs as operands, and the multiplier operates on them quarter-word-wise (64-bit-wise). For each quarter word, the first cycle computes $c = ab$ and $[c]_d$. In the second cycle, $m = [cR]_d$ is calculated. The TMP register is then added to store the intermediate results from the first two clock cycles. Lastly, $r = [m \times q + c]^d$ and a conditional subtraction of r (if $r \geq q$) are done in one clock cycle. To achieve this, we integrate an additional subtractor into the BN-MAC, saving one clock cycle per quarter word with minimal area overhead. The partial results from each quarter word operation are stored and concatenated in the accumulator register ACC. Additionally, $c = a \times b$ from the first cycle must be stored for reuse in the third cycle. Therefore, we integrated the register C into the BN-MAC unit. This design requires 12 clock cycles per bn.mulvm instruction.

Execution Stages for bn.mulv. Similarly to bn.mulvm, the modified BN-MAC also takes 256-bit operands and processes them quarter-word-wise for the bn.mulv instruction. However, it requires only one clock cycle per quarter word, computing $c = a \times b$ and $[c]_d$, with partial results concatenated in the accumulator register. This results in 4 clock cycles per bn.mulv instruction.

6.2.3. Big Number MULV Module. For the single-cycle bn.mulv(m) approach, we integrate the new BN-MULV module into the OTBN pipeline. This design aims for high performance and minimizes changes to OTBN's control logic. The architecture of the BN-MULV module is illustrated in Figure 7.

For the bn.mulv instruction (vectorized multiplication without modular reduction), c is selected as the output. For bn.mulvm, the BN-MULV module handles vectorized modular multiplication using the Montgomery algorithm (Algorithm 2.1), supporting either 16-bit (.16H) or 32-bit (.8S) input vectors. It executes 16 16-bit or 8 32-bit operations concurrently, sharing the same multiplier for both, as outlined in Section 6.1.2. The adder and subtractor are also shared, following the configurable adder approach from Section 6.1.1. The vectorized addition on line 2 followed by the vectorized conditional subtraction on lines 3-5 of Algorithm 2.1, is implemented similarly to the pseudo-modulo reduction within the BN-ALU for the bn.addv instruction. Additional routing is required to efficiently implement the selection of the lower or upper d bits ($[\ .\ ]_d$ or $[\ .\ ]^d$ respectively) with additional multiplexers distinguishing between .8S and .16H variants. The values of q and R are configured analogously to our BN-MAC extension through a dedicated connection from the BN-ALU's MOD register to the BN-MULV module.

TABLE 1: Resource Utilization on Xilinx 7-Series FPGAs.

Design	LUT	FF	DSP	BRAM
BN-MAC	2,141	312	16	0
BN-MAC$_{\text{Ext.}}$	4,450	508	16	0
BN-MULV$_{\text{Ext.++}}$	10,472	0	96	0
BN-ALU	6,321	320	0	0
BN-ALU$^{\text{KMAC}}$	8,982	1,286	0	0
BN-ALU$_{\text{Ext.}}$	8,604	320	0	0
BN-ALU$^{\text{KMAC}}_{\text{Ext.}}$	11,649	1,286	0	0
Butterfly (KYBER, DILITHIUM) [24]	3,887	951	33	0
Butterfly (KYBER, NewHope) [29]	2,908	170	9	0
Mod. arith. (NewHope) [62]	1,907	1,658	7	34
Mod. arith. (KYBER) [28]	178	0	5	0.5
Mod. arith. (DILITHIUM) [28]	377	0	10	0.5
Mod. arith. (KYBER) [31]	93	0	1	0
Mod. arith. (DILITHIUM) [30]	312	0	4	0
Keccak [24]	1,312	0	0	0
Keccak [30]	3,622	1,605	0	0
Keccak [29]	3,847	0	0	0

TABLE 2: Resource Utilization for 7nm ASIC. Area in μm^2.

Design	Cell Count	Cell Area	Net Area	Total Area
BN-MAC	13,376	1,623	822	2,446
BN-MAC$_{\text{Ext.}}$	22,637	2,510	1,307	3,817
BN-MULV$_{\text{Ext.++}}$	100,090	10,280	5,419	15,699
BN-ALU	20,377	2,150	1,264	3,414
BN-ALU$^{\text{KMAC}}$	29,317	3,344	1,710	5,054
BN-ALU$_{\text{Ext.}}$	22,313	2,269	1,434	3,702
BN-ALU$^{\text{KMAC}}_{\text{Ext.}}$	32,271	3,592	1,965	5,557

6.2.4. Synthesis Results for Single Extensions. Table 1 and Table 2 present synthesis results for Xilinx 7-Series devices and ASIC results for the ASAP7 PDK [63] respectively. The tables include four variants: BN-ALU, the reference design without extensions; BN-ALU$^{\text{KMAC}}$, with the KMAC interface (see Section 4); BN-ALU$_{\text{Ext.}}$, with vector extensions only; and BN-ALU$^{\text{KMAC}}_{\text{Ext.}}$, which includes both the KMAC interface and vector extensions. Our results show that BN-ALU$_{\text{Ext.}}$ does not introduce a significant overhead,

with most resources reused. However, the KMAC interface induces additional overhead due to the need for additional flip-flops, extended read/write ports for new special-purpose registers, and countermeasures including blanking, wiping, and integrity protection (cf. Section 2.4.1). Moreover, the KMAC interface also includes a small FIFO.

For our vectorized multiplication approach, Table 1 and Table 2 show synthesis results for the original BN-MAC, its extended version BN-MAC$_{\text{Ext.}}$ (cf. Section 6.2.2), and the proposed BN-MULV$_{\text{Ext.++}}$ module (cf. Section 6.2.3). Our results show that BN-MAC$_{\text{Ext.}}$ leads to a moderate increase in resources, while BN-MULV$_{\text{Ext.++}}$ is several times larger than BN-MAC, which aligns with the resource requirements for parallel Montgomery multiplication discussed in Section 6.2.3. BN-MAC$_{\text{Ext.}}$ includes additional registers with integrity protections, and a new subtractor (cf. Section 6.2.2) with corresponding blanking mechanism, all of which contribute to the increase in resource consumption. Table 1 compares our extensions to other tightly coupled accelerators revealing that the others are more compact, as they only focus on specific extensions and do not incorporate generic big number arithmetic for contemporary cryptography. Most of the works cited in Table 1 use 32-bit architectures, while ours operate on 64-bit or 256-bit. The Keccak designs require similar or fewer resources than the KMAC interface; however, as we will show later, the external KMAC offers massive performance improvements. Furthermore, none of the cited works account for side-channel compliance with OTBN design requirements, A more detailed comparison considering processor specifications can be found in Section 7.3.

7. Results

This section presents the results of our work, including cycle counts, memory usage, code size, and FPGA/ASIC synthesis outcomes. We separately report cycle counts for polynomial multiplication functions and full schemes, along with comparisons to related work and other common implementation targets.

Testing & Benchmarking Setup. We test our ML-{KEM,DSA} implementations using the OpenTitan Python simulator for OTBN, OTBN$^{\text{KMAC}}$, OTBN$^{\text{KMAC}}_{\text{Ext.}}$, and OTBN$^{\text{KMAC}}_{\text{Ext.++}}$ with the added KMAC interface and new instructions. The simulator incorporates cycle count estimates for KMAC along with the communication overhead from OTBN to KMAC. For correctness verification, we compared our implementations to Pope's Python versions of ML-KEM and ML-DSA [64], [65], which have been evaluated using known answer test (KAT) vectors derived from the schemes' official C reference implementations [19], [20]. We evaluated our code on over 10 000 random inputs.

We compare our software implementations to commonly known platforms that share some characteristics with OTBN/OTBN$^{\text{KMAC}}_{\text{Ext.}}$. For Intel AVX2, we reference draft-standard compliant implementations from the pq-crystals team[2], and run benchmarks on an Intel Core i7-6700K Skylake processor with hyperthreading and Turbo Boost disabled. We compile using `gcc` version 12.2.0 on Debian 11. For RISC-V vector (RVV) extension, we compare to the numbers for `RV64IMBV` from [22], which use a XuanTie C908 64-bit CPU (CanMV-K230 dev. board). For Cortex-M4, we employ the pqm4 framework [58] using code for KYBER from [12] and for DILITHIUM from [13], adapted to NIST draft standards by Kannwischer and compiled with `arm-none-eabi-gcc` version 13.2.1.

Although OTBN serves as a co-processor to the main Ibex processor, we find it fair to compare cycle counts of the cryptographic scheme directly on OTBN. While Ibex may need to configure OTBN by loading firmware in some cases, this step can typically be prepared at boot-up. Data transfers between the two processors are as fast as normal memory accesses since they share certain memory sections. In fact, the OpenTitan architecture even allows shielding secrets from Ibex via its key manager, potentially reducing the need for some data transfers. For this paper, however, we only consider plain cycle counts on OTBN.

7.1. Software Benchmarks

7.1.1. Polynomial Multiplication. Table 3 shows the cycle counts for the polynomial multiplication related functions of ML-{KEM,DSA}. Our OTBN$^{\text{KMAC}}_{\text{Ext.}}$ implementation outperforms the implementations on plain OTBN with speedups up to a factor of eight. Additionally, the OTBN$^{\text{KMAC}}_{\text{Ext.++}}$ implementation is two to three times faster than that of OTBN$^{\text{KMAC}}_{\text{Ext.}}$.

Comparing our OTBN$^{\text{KMAC}}_{\text{Ext.}}$ results with the closely related work from [23], we can see that our implementation is more than $4\times$ faster, while making use of similar vectorization techniques. A key difference here is that we provide vectorized *and* modular addition, subtraction, and multiplication instructions, while [23] replicates these operations in software. We believe that our approach aligns better with OTBN's design philosophy, as it is geared towards cryptographic operations – where modular operations are a staple – while still remaining rather general. Compared to [24], we see an 18% slowdown in the transformations but a 30% speedup in pointwise multiplication in the case of ML-DSA. In contrast, our implementation is up to twice as fast for ML-KEM, attributed to vectorization enabling greater parallelization on OTBN$^{\text{KMAC}}_{\text{Ext.}}$, while [24] does not consider a SIMD approach. Results from [26] suggest that applying the Kronecker+ technique from [25] may not be suitable for OTBN, as our baseline implementation performs better. The AVX2 implementation on Intel Skylake, offering equally-sized registers, outperforms our work on OTBN$^{\text{KMAC}}_{\text{Ext.}}$ due to its super-scalar architecture and out-of-order (OoO) execution capabilities. Our implementation on OTBN$^{\text{KMAC}}_{\text{Ext.}}$ outperforms the one on the C908 due to

2. https://github.com/pq-crystals/kyber/tree/11d00ff1f20cfca1f72d819e5a45165c1e0a2816, https://github.com/pq-crystals/dilithium/tree/e7bed6258b9a3703ce78d4ec38021c86382ce31c

OTBN's WDRs being 2× as wide as the C908's registers. In addition to that, our ISA extensions being tailored for modular arithmetic contributes to our speedup: `bn.mulvm` takes only 12 cycles on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, whereas the RVV implementation requires explicit Montgomery multiplication using four instructions with a latency of 4–5 cycles each [22]. However, our gains remain below a factor of 2 due to the C908's superscalar nature. Compared to the work from [28], we achieve speedups up to a factor of 18 on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, mainly attributed to our vectorized approach. Despite the less general approach in [29], we still manage to obtain a speedup of nearly 2×.

TABLE 3: Cycle Counts for Polynomial Multiplication Related Functions of ML-{KEM,DSA}.

	Platform	NTT	INTT	Base Mul.
ML-DSA	$\text{OTBN}_{\text{Ext.++}}^{\text{KMAC}}$	996	1003	230
	$\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$	2404	2587	582
	$\text{OTBN}^{(\text{KMAC})}$	8206	8701	2552
	OTBN [24][a]	1972	2244	768
	OTBN [26]	10 763	13 943	9714
	Skylake [37]	840	800	150
	C908 [22]	3395	3540	759
	Cortex-M4 [15]	8066	8388	1931
	[28]	18 554	21 375	—
ML-KEM	$\text{OTBN}_{\text{Ext.++}}^{\text{KMAC}}$	384	392	284
	$\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$	1000	1096	724
	$\text{OTBN}^{(\text{KMAC})}$	8133	8771	4604
	OTBN [23][a]	4356	—	—
	OTBN [24][a]	1454	1726	1448
	Skylake [40]	208	228	86
	C908 [22]	1575	1840	753
	Cortex-M4 [12]	4474	4684[b]	2422
	[28]	18 488	18 488	—
	[29]	1935	1930	—
	[31]	4189	3481	3257

[a] Modified variant of OTBN. [b] For ML-KEM-512.

7.1.2. Full-Scheme Benchmarks. We present the benchmark results for all three parameter sets and all three operations of ML-DSA and ML-KEM in Tables 4 and 5.

As shown in Table 4, our $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ implementation achieves performance gains of six to nine times compared to plain OTBN, largely due to the KMAC interface, as evident from the numbers for $\text{OTBN}^{\text{KMAC}}$. The $\text{OTBN}_{\text{Ext.++}}^{\text{KMAC}}$ implementation is again up to 32% faster than $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$.

Comparing our work for $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ to the implementations for the verification from [24], we are around five to six times faster, which shows that the faster Keccak acceleration and pointwise multiplication makes up for the slightly slower (inverse) NTT. The KMAC core just taking 96 cycles per Keccak permutation compared to the 1770 the C908 needs is an important reason for the speedup we achieve over the implementation from [22] – in addition to the factors mentioned in Section 7.1.1. However, our performance on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ remains behind the AVX2 optimized implementation on Intel Skylake. In terms of hardware/software co-designs, our cycle counts are lower than all compared works. The compact implementation from [30] is the closest to our work on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, although it relies on specifically tailored extensions for DILITHIUM. A comparison of respective hardware overheads will follow in Section 7.3.

The performance of ML-KEM on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ mirrors that of ML-DSA, with significant speedups attributed to the KMAC interface. Our ISA extensions yield an even greater performance gain for ML-KEM, tracing back to the higher parallelism for 16-bit elements. We outperform the plain OTBN implementation by nearly a factor of nine. Again, $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ outperforms the C908 implementation, but cannot keep up with AVX2. The hardware/software co-design with the most comparable performance is that of [29], which also uses a vectorized approach to modular arithmetic and a Keccak accelerator. However, it differs in specificity and accelerator capabilities, resulting in speedups of up to a factor of 4 for our work. As the work from [31] picks a highly resource-constrained approach without making use of any form of Keccak acceleration, it is no surprise that the speedups on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ are as high as a factor of 17.

As mentioned in Section 2.4.2, we only provide masked KMAC numbers as it is a more conservative choice and likely to be manufactured in practice. However, if unmasked KMAC is used, ML-{KEM,DSA} performance can be derived from the profiling tables in Sections 3 to 5 and Tables 4 and 5. For instance, we estimate cycle savings of 4–7% for ML-DSA-65 based on Figure 3a and Table 4 with unmasked Keccak.

Memory & Code Size. Memory usage and code size were not optimized in our work but remain comparable to those of other microcontroller architectures. For example on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, our stack usage is very close to that of Arm Cortex-M4, being 24.9–39.9% below the results on M4 for ML-KEM and 1.1–3.6% for ML-DSA key generation and verification. For signing, ours is 2.4–13.6% larger than that of M4. We follow state-of-the-art (SotA) implementation patterns and refrain from applying optimization techniques that trade memory or code size for performance. More detailed data on these metrics can be obtained from Section E.

7.2. PQC on OpenTitan: System Impact

Previously, we only compare the performance of ML-{KEM,DSA} with and without our extensions isolated on OTBN. Thus, in the following, we discuss the practicality of them with respect to OpenTitan as a whole. The two main questions arising from this are: (1) How does running PQC in addition to classical cryptography on OTBN affect the system-performance? (2) Will locking KMAC with OTBN slow down common applications relying on KMAC and how will it impact OpenTitan in general?

With respect to (1), it is true that an application requiring, e.g., an ECDSA-P256 signature would have to wait in case, e.g., an ML-DSA-44 signature is being computed on OTBN. First, it has to be said that the impact of such effects is highly dependent on the use-case of OpenTitan and the applications running on it. To approach this question, we compare the performance of PQC using our extensions

TABLE 4: ML-DSA Full-Scheme Cycle Counts. Median Result Was Selected, if Given. 10 000 Iterations.

	Platform	Key Gen.	Sign	Verify
ML-DSA-44	OTBN	1 242 491	2 578 069	1 231 623
	$\text{OTBN}^{\text{KMAC}}$	270 910	1 116 688	318 933
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext}}$	149 889	409 892	158 382
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	130 753	286 544	131 179
	OpenTitan [24][b,c]	—	—	997 722
	Skylake [37][a]	92 062	210 670	96 102
	C908 [22][b]	—	—	—
	Cortex-M4 [13][a]	1 353 035	2 887 554	1 351 519
	[27][b]	593 403	1 905 872	651 217
	[28][b]	1 592 325	5 884 266	1 700 679
	[30][b]	541 869	845 005	563 385
ML-DSA-65	OTBN	2 190 308	4 499 356	2 113 136
	$\text{OTBN}^{\text{KMAC}}$	438 173	1 926 881	493 458
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext}}$	261 003	699 303	256 486
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	233 910	476 280	215 778
	OpenTitan [24][b,c]	—	—	1 488 526
	Skylake [37][a]	154 764	348 512	155 208
	C908 [22][b]	645 000	2 139 000	646 000
	Cortex-M4 [13][a]	2 390 147	4 871 469	2 294 454
	[27][b]	1 067 824	3 253 378	1 126 938
	[28][b]	2 974 897	10 211 677	2 963 936
	[30][b]	902 273	1 329 844	918 863
ML-DSA-87	OTBN	3 752 752	6 191 376	3 682 307
	$\text{OTBN}^{\text{KMAC}}$	691 143	2 354 620	769 669
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext}}$	410 615	915 838	421 653
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	365 501	657 262	361 715
	OpenTitan [24][b,c]	—	—	2 223 143
	Skylake [37][a]	242 186	430 212	241 930
	C908 [22][b]	—	—	—
	Cortex-M4 [13][a]	4 071 682	6 672 924	4 000 721
	[27][b]	1 784 767	4 357 249	1 848 324
	[28][b]	5 001 302	13 339 255	5 132 776
	[30][b]	1 533 230	2 065 456	1 561 021

[a] Own benchmarks. [b] Round 3 DILITHIUM.
[c] Including modified variant of OTBN, parts of the execution on Ibex Core.

TABLE 5: ML-KEM Full-Scheme Cycle Counts. Median Result Was Selected, if Given. 10 000 Iterations.

	Platform	Key Gen.	Encaps	Decaps
ML-KEM-512	OTBN	322 911	351 037	396 188
	$\text{OTBN}^{\text{KMAC}}$	87 072	117 890	162 124
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext}}$	36 599	46 199	57 717
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	32 375	40 119	48 485
	Skylake [40][a]	29 450	30 998	30 518
	C908 [22][b]	—	—	—
	Cortex-M4 [12][a]	369 449	373 051	408 633
	[28][b]	419 597	438 280	100 796
	[29][c]	150 106	193 076	204 843
	[31][b]	622 000	785 000	713 000
ML-KEM-768	OTBN	562 392	609 781	668 548
	$\text{OTBN}^{\text{KMAC}}$	156 969	194 602	253 998
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext}}$	69 625	81 615	97 048
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	61 967	71 585	82 698
	Skylake [40][a]	47 598	46 730	47 420
	C908 [22][b]	165 000	197 000	207 000
	Cortex-M4 [12][a]	603 262	626 531	673 369
	[28][b]	694 504	731 597	130 348
	[29][c]	273 370	325 888	340 418
	[31][b]	988 000	1 237 000	1 133 000
ML-KEM-1024	OTBN	910 496	964 913	1 041 230
	$\text{OTBN}^{\text{KMAC}}$	245 747	290 406	365 049
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext}}$	113 777	127 924	148 035
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	101 807	113 044	127 656
	Skylake [40][a]	64 606	65 326	67 776
	C908 [22][b]	—	—	—
	Cortex-M4 [12][a]	959 632	981 334	1 041 133
	[28][b]	1 090 458	1 126 462	159 639
	[29][c]	349 673	405 477	424 682
	[31][b]	1 543 000	1 851 000	1 719 000

[a] Own benchmarks. [b] Round 3 KYBER. [c] Round 2 KYBER.

from Table 4 to the performance of ECC and RSA as it is currently provided for OTBN by the OpenTitan team from Table 6. We can observe that at the same security level, ML-DSA-44 signing is 41% faster than ECDSA-P256, and even 150× faster than RSA-3072. We deem this as an indicator for the acceptability of the additional strain on the overall system.

Regarding (2), locking KMAC with OTBN will potentially stall non-OTBN applications that require KMAC. However, in scenarios where both KMAC and OTBN are needed alongside other IPs, this approach reduces the overall runtime. Indeed, by comparing the results for OTBN and $\text{OTBN}^{\text{KMAC}}$ from Tables 4 and 5, we can see that the cycle savings from using the KMAC interface are *always* significantly larger than the time KMAC is locked for OTBN. Thus, we found it compelling from a design perspective to lock two blocks for short time rather than one for several times as long, leading to the overall time-saving. Another way to avoid locking KMAC is to access it through Ibex. Yet, this is not desirable for the reasons given in Section 3.

Note that due to the asynchronous nature of OpenTitan's cryptolib, it is still possible for Ibex to issue cryptographic operations to other IP blocks, i.e., AES, HMAC, etc., while OTBN and KMAC are busy with a PQC operation.

As this work strives for a low-overhead solution for accelerating classical cryptography and PQC simultaneously, we do not consider the alternative approach of adding a second KMAC core and PQC accelerator. This solution would come at a significant hardware cost which we aim to avoid – although mitigating any potential performance degradation.

TABLE 6: ECC/RSA Benchmarks [66].

Operation	Cycles
ECDSA-P256 Sign	704 126
ECDSA-P384 Sign	1 697 985
RSA-2048 Sign	18 889 021
RSA-3072 Sign	61 303 537

7.3. Hardware Utilization

As demonstrated in previous sections, we outperform most existing RISC-V-based ISA extensions [27], [28], [29], [30], [31] in terms of cycle counts and latency. This can be

attributed to three main factors. First, our approach utilizes the 256-bit WDRs of OTBN for SIMD operations, providing a significant advantage over {32,64}-bit architectures. While [24] also leverages OTBN's WDRs, they only compute one 32-bit butterfly operation per cycle and does not fully exploit WDRs for SIMD. Second, thanks to the availability of OTBN's WDRs, the cost of memory accesses is minimized. Third, we implemented a dedicated interface to KMAC, which computes a Keccak round in just 4 cycles, compared to 40 cycles with the Keccak ISA extensions in [24]. Although the Keccak accelerator in [27], [29] can compute one round per cycle, it requires additional floating-point registers and accesses multiple registers at once. We found that not integrating such a powerful accelerator into the processor pipeline itself, but providing a dedicated interface offers similar performance and even allows a cleaner integration.

Table 8 presents the ASIC synthesis results using the ASAP7 PDK [63]. We synthesized the Top-Earlgrey design rather than the Chip-Earlgrey-ASIC design (which is built on top of Top-Earlgrey with some additional modules) due to missing standard cells in the PDK. The table also shows synthesis results for OTBN with different extension variants. For both designs, we treated memory as a black box and only targeted logic overhead as memory requirements are similar for all variants. These numbers highlight that the performance improvement of our $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ implementation comes at a low cost. In contrast, the significant performance gains of our $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ are relatively expensive, nearly doubling the size of OTBN. However, considering OpenTitan's overall area, this remains a reasonable approach in case highest performance is desired.

We analyzed the impact of our extensions on the critical path by evaluating out-of-context ASIC synthesis results for OTBN and the ASAP7 PDK. In the original OTBN and $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, the critical path is within the BN-MAC. For $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$, the critical path moves to our new BN-MULV module. These shifts are expected due to the complexity of the implemented operations. The maximum clock frequency decreases from 819 MHz for the original OTBN to 490 MHz for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ and 350 MHz for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$. The drop in maximum frequency for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ can be explained by the fact that our extension adds hardware to the most critical path within the original OTBN. Considering the complexity of our BN-MULV module, it appears evident that the most critical path is within the BN-MULV module for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$. Considering the drop in maximum clock frequency, our $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ still achieves a $3.5\times$ – $4.4\times$ net speedup, while $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ reaches $3.8\times$ – $5.5\times$, meaning the more resource-efficient $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ can outperform $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ in some cases at maximum frequency. This highlights the potential for design space exploration. However, the OpenTitan Earl Grey ASIC implementation aims for a moderate frequency of 100 MHz which is easily achieved for both, $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ and $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$.

Table 7 provides more insights into the hardware utilization of related work. Our FPGA results target Xilinx 7-Series devices, including those for OTBN and the Chip-Earlgrey-CW310 design. In [28], the authors choose the more powerful application-level processor CVA6. The works in [27], [29] utilize PULPino, a microcontroller with slightly more features than Ibex. Meanwhile, [30], [31] employs the very compact Hummingbird E203 core. In general, comparisons with other studies except [24] are not straightforward, as OTBN is a very specific target. While it includes big-number arithmetic modules and countermeasures, it lacks features present in other platforms. Furthermore, OTBN's fault injection and side-channel countermeasures imply that all extensions must consider the same protections. The extensions in [24], [28], [30], [31] are considerably more compact yet reduce performance. For [27], [29], [30], the relative overhead is larger, but both the base and extended platform are more compact than our extended OTBN. The OTBN extension proposed in [23] is not included in Table 7 as they do not provide an actual hardware implementation. A resource-efficient integration of their vectorized shift/addition/subtraction could involve reconfiguring the BN-ALU, similar to our method. While both approaches to vectorized multiplication reconfigures the BN-MAC, they focus on 32-bit multiplications, as opposed to our design which supports both 16-bit and 32-bit multiplications, offering greater flexibility at the cost of more multiplexing logic. A further difference is that our implementation incorporates an additional subtractor to speed up the modular multiplication as discussed in Section 6.2.2. The hardware cost for their concatenate-and-left-shift-immediate and broadcast instruction is minimal due to simple signal rewiring and multiplexing. As a result, their approach trades a slightly lower hardware overhead for less flexibility and significantly reduced performance (cf. Section 7.1.1).

TABLE 7: Comparison With SotA HW/SW Co-designs.

Design	ASIC	FPGA		
	Cell Count	LUT	FF	DSP
$\text{Top-Earlgrey}^{\text{KMAC}}_{\text{Ext.}}$	757 640	246 616	121 722	22
$\text{Top-Earlgrey}^{\text{KMAC}}_{\text{Ext.++}}$	841 418	254 625	121 516	118
$\text{OTBN}^{\text{KMAC}}$	141 052	35 061	16 585	16
$\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$	156 354	40 150	16 806	16
$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	254 655	48 158	16 641	112
[24]	—	55 409	16 575	49
[29]	57 413	24 306	10 837	18
[27]	65 968	22 356	13 181	13
[28]	—	64 855	60 349	29
[31]	13 573	9614	6669	5
[30]	22 936	15 258	12 934	7

In summary, while existing designs may better suit specific use cases requiring compact platforms, our extensions for OpenTitan – being the first industry-grade open-source secure element – offer seamless micro-architectural integration, flexibility, and high performance with minimal hardware overhead. Our comparison with SotA designs indicates that the hardware costs of our extensions are acceptable, relative to both related work and the overall OpenTitan.

TABLE 8: ASIC Synthesis – Area Consumption for 7 nm Process Without Memories. Area Is Given in μm^2.

Design	Cell Count	Cell Area	Net Area	Total Area
Top-Earlgrey	740 101	106 885	41 763	148 647
Top-Earlgrey$^{\text{KMAC}}_{\text{Ext.}}$	757 640	109 004	42 991	151 995
Top-Earlgrey$^{\text{KMAC}}_{\text{Ext.++}}$	841 418	117 325	47 564	164 889
OTBN	134 258	16 264	7471	23 735
OTBN$^{\text{KMAC}}$	141 052	17 202	7863	25 064
OTBN$^{\text{KMAC}}_{\text{Ext.}}$	156 354	18 741	8608	27 349
OTBN$^{\text{KMAC}}_{\text{Ext.++}}$	254 655	28 788	13 872	42 660

8. Discussion and Future Work

As noted in Section 5, we take a different approach from most related work by offering a more generic ISA extension for vector arithmetic, rather than scheme-specific instructions. Our extensions accelerate schemes requiring modular arithmetic with moduli fitting in a 32-bit word, including many lattice-based schemes such as Falcon [67], FrodoKEM [68], or NTRU Prime [69]. Also the MQ-based scheme MQOM [70] and the hybrid homomorphic encryption "Pasta" [71] rely on such arithmetic. Almost all aforementioned schemes also benefit from the KMAC interface. For symmetric cryptography, our instructions can optimize ARX-ciphers ChaCha20/Salsa20 [72], [73] and SHA256 [74] hashing with vectorized additions modulo 2^{32} and shifts(/rotations).

A potential follow-up could involve applying memory reduction techniques from [14], [16] and exploring how trade-offs on OTBN change with access to the fast KMAC block for hashing. In this regard, extending the ISA with a bit-mask-based permutation instruction for vectorized rejection sampling, as in [75], could be considered, given that most stack optimizations shift runtime towards sampling. Since OpenTitan already offers a masked KMAC core, extending our work to masked implementations of ML-{KEM,DSA} whilst re-evaluating the adequacy of our proposed extensions could be worthwhile. Additionally, the applicability of our ISA extension to, e.g., the Falcon verification, signature schemes from NIST's on-ramp process, or fully homomorphic encryption could also be studied. As OpenTitan targets high security standards, a formally verified re-implementation of ML-{KEM,DSA} on OTBN would be a logical next step. OTBN support for the Jasmin language [76] is a current work-in-progress by Arranz Olmos[3].

Another possibility for future work is to explore design optimizations. For instance, our multiplier in Section 6.1.2 currently uses only four of its sixteen 16-bit multipliers for ML-KEM. Since ML-KEM's 16-bit multiplications do not require carry-save-adders for partial product combination, increasing the number of parallel 16-bit multiplications could be a cost-effective improvement.

3. https://github.com/sarranz/jasmin/tree/demo1

Acknowledgements

This research was supported by Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; by the European Commission through the ERC Starting Grant 805031 (EPOQUE); by the German Federal Ministry of Education and Research (BMBF) in the framework of the 6GEM research hub under grant number 16KISK038; by the Bavarian Ministry of Economic Affairs, Regional Development and Energy in the context of the project Trusted Electronics Bavaria (TrEB); by the SALTO strategic exchange program between the Centre National de la Recherche Scientifique (CNRS) and the Max-Planck-Gesellschaft (MPG); and by the French National Research Agency in the course of the "Investissements d'avenir" program (ANR-15-IDEX-02). We thank Andrew "bunnie" Huang for providing insight on the performance characteristics of our hardware design.

References

[1] M. R. Albrecht, C. Hanser, A. Hoeller, T. Pöppelmann, F. Virdia, and A. Wallner, "Implementing RLWE-based schemes using an RSA co-processor," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 1, pp. 169–208, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7338 1

[2] A. Greuet, S. Montoya, and G. Renault, "On using RSA/ECC coprocessor for ideal lattice-based key exchange," in *COSADE 2021: 12th International Workshop on Constructive Side-Channel Analysis and Secure Design*, ser. Lecture Notes in Computer Science, S. Bhasin and F. De Santis, Eds., vol. 12910. Springer, Cham, Oct. 2021, pp. 205–227. [Online]. Available: https://doi.org/10.1007/978-3-030-89915-8_10 1

[3] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 4, pp. 17–61, 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8344 1

[4] D. Adrian, B. Beck, D. Benjamin, and D. O'Brien, "Advancing our amazing bet on asymmetric cryptography," Post on the Chromium Blog, 2024. [Online]. Available: https://blog.chromium.org/2024/05/advancing-our-amazing-bet-on-asymmetric.html 1

[5] W. Evans, B. Westerbaan, C. Patton, P. Wu, and V. Gonçalves, "Post-quantum cryptography goes GA," Post on the Cloudflare Blog, 2023. [Online]. Available: https://blog.cloudflare.com/post-quantum-cryptography-ga/ 1

[6] Mozilla, "Mozilla Firefox Mercurial," 2023. [Online]. Available: https://hg.mozilla.org/releases/mozilla-release/file/d3c71a6fc9a1aecf1fe04f8de2fc0b816588e677/security/manager/ssl/nsNSSIOLayer.cpp#l1439 1

[7] E. Kret and R. Schmidt, "The PQXDH key agreement protocol," Signal, Tech. Rep., 2024. [Online]. Available: https://signal.org/docs/specifications/pqxdh/pqxdh.pdf 1

[8] Apple Security Engineering and Architecture (SEAR), "iMessage with PQ3: The new state of the art in quantum-secure messaging at scale," 2024. [Online]. Available: https://security.apple.com/blog/imessage-pq3/ 1

[9] "ANSSI views on the post-quantum cryptography transition (2023 follow up)," French Cybersecurity Agency (ANSSI), Paris, France, Position Paper, 2023. [Online]. Available: https://cyber.gouv.fr/sites/default/files/document/follow_up_position_paper_on_post_quantum_cryptography.pdf 1

[10] "BSI TR-02102-1: Cryptographic mechanisms: Recommendations and key lengths, version: 2024-1," Federal Office for Information Security (BSI), Bonn, Germany, Technical Guideline, 2024. [Online]. Available: https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Technische-Richtlinien/TR-nach-Thema-sortiert/tr02102/tr02102_node.html 1

[11] OpenTitan Team, "Datasheet - OpenTitan documentation," 2023. [Online]. Available: https://opentitan.org/book/doc/introduction.html 1, 3, 4

[12] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen, "Improved Plantard arithmetic for lattice-based cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 614–636, 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9833 2, 3, 4, 5, 11, 12, 13, 25

[13] J. Huang, A. Adomnicai, J. Zhang, W. Dai, Y. Liu, R. C. C. Cheung, Ç. K. Koç, and D. Chen, "Revisiting Keccak and Dilithium implementations on ARMv7-M," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 1–24, 2024. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11419 2, 11, 13

[14] J. W. Bos, J. Renes, and A. Sprenkels, "Dilithium for memory constrained devices," in *AFRICACRYPT 22: 13th International Conference on Cryptology in Africa*, ser. Lecture Notes in Computer Science, L. Batina and J. Daemen, Eds., vol. 2022. Springer, Cham, Jul. 2022, pp. 217–235. [Online]. Available: https://eprint.iacr.org/2022/323 2, 15

[15] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels, "Faster Kyber and Dilithium on the Cortex-M4," in *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, G. Ateniese and D. Venturi, Eds., vol. 13269. Springer, Cham, Jun. 2022, pp. 853–871. [Online]. Available: https://eprint.iacr.org/2022/112 2, 12

[16] D. O. C. Greconici, M. J. Kannwischer, and A. Sprenkels, "Compact Dilithium implementations on Cortex-M3 and Cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 1, pp. 1–24, 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8725 2, 15

[17] E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard, "Cortex-M4 optimizations for {R,M}LWE schemes," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 3, pp. 336–357, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8593 2

[18] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of Kyber on Cortex-M4," in *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*, ser. Lecture Notes in Computer Science, J. Buchmann, A. Nitaj, and T. eddine Rachidi, Eds., vol. 11627. Springer, Cham, Jul. 2019, pp. 209–228. [Online]. Available: https://doi.org/10.1007/978-3-030-23696-0_11 2

[19] pq-crystals, "Dilithium reference implementation," GitHub, Nov. 2023. [Online]. Available: https://github.com/pq-crystals/dilithium/ 2, 5, 11, 22, 23

[20] ——, "Kyber reference implementation," GitHub, Dec. 2023. [Online]. Available: https://github.com/pq-crystals/kyber 2, 5, 11, 22, 23

[21] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, pp. 221–244, 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9295 2, 7

[22] J. Zhang, Y. Yan, J. Huang, and Ç. K. Koç, "Optimized software implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, pp. 632–655, Dec. 2024. [Online]. Available: https://eprint.iacr.org/2024/1515 2, 7, 11, 12, 13

[23] E. Urquhart and F. Stajano, "Acceleration of core post-quantum cryptography primitive on open-source silicon platform through hardware/software co-design," in *Cryptology and Network Security*. Singapore: Springer Nature Singapore, 2025, pp. 144–161. [Online]. Available: https://www.cl.cam.ac.uk/~fms27/papers/2024-UrquhartStajano-acceleration.pdf 2, 11, 12, 14

[24] T. Stelzer, F. Oberhansl, J. Schupp, and P. Karl, "Enabling Lattice-Based Post-Quantum Cryptography on the OpenTitan Platform," in *Proceedings of the 2023 Workshop on Attacks and Solutions in Hardware Security*, ser. ASHES '23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 51–60. [Online]. Available: https://doi.org/10.1145/3605769.3623993 2, 10, 11, 12, 13, 14, 25

[25] J. W. Bos, J. Renes, and C. van Vredendaal, "Post-quantum cryptography with contemporary co-processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer," in *USENIX Security 2022: 31st USENIX Security Symposium*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, Aug. 2022, pp. 3683–3697. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bos 2, 11

[26] H. Turcuman, "Speeding-up post-quantum cryptography on an RSA co-processor," Master's thesis, Technical University of Munich, Munich, Sep. 2023. [Online]. Available: https://github.com/horiaionut/kroneker-plus-on-otbn/blob/5576d7b035f5fe55a7199987ea05613d4aa913e7/paper.pdf 2, 11, 12

[27] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "Post-quantum signatures on RISC-V with hardware acceleration," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 2, pp. 30:1–30:23, 2024. [Online]. Available: https://eprint.iacr.org/2022/538 2, 13, 14, 25

[28] P. Nannipieri, S. Di Matteo, L. Zulberti, F. Albicocchi, S. Saponara, and L. Fanucci, "A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms," *IEEE Access*, vol. 9, pp. 150 798–150 808, 2021. [Online]. Available: https://arpi.unipi.it/retrieve/e0d6c931-5a0c-fcf8-e053-d805fe0aa794/A_RISC-V_Post_Quantum_Cryptography_Instruction_Set_Extension_for_Number_Theoretic_Transform_to_Speed-Up_CRYSTALS_Algorithms.pdf 2, 10, 12, 13, 14

[29] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 239–280, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8683 2, 10, 12, 13, 14, 25

[30] L. Li, Q. Tian, G. Qin, S. Chen, and W. Wang, "Compact instruction set extensions for Dilithium," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 2, pp. 23:1–23:21, 2024. [Online]. Available: https://doi.org/10.1145/3643826 2, 10, 12, 13, 14

[31] L. Li, G. Qin, Y. Yu, and W. Wang, "Compact instruction set extensions for Kyber," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 3, pp. 756–760, 2024. [Online]. Available: https://doi.org/10.1109/TCAD.2023.3327104 2, 10, 12, 13, 14

[32] Y. Zhao, R. Xie, G. Xin, and J. Han, "A high-performance domain-specific processor with matrix extension of RISC-V for module-LWE applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 7, pp. 2871–2884, Jul. 2022. [Online]. Available: https://doi.org/10.1109/TCSI.2022.3162593 2

[33] T. Fritzmann, M. Van Beirendonck, D. B. Roy, P. Karl, T. Schamberger, I. Verbauwhede, and G. Sigl, "Masked accelerators and instruction set extensions for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, pp. 414–460, 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9303 2

[34] National Institute of Standards and Technology, "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard," 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203 2, 3, 18, 20

[35] ——, "FIPS 204: Module-Lattice-Based Digital Signature Standard," 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.204 2, 3, 18, 19, 20

[36] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 238–268, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/839 3

[37] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "CRYSTALS-DILITHIUM," National Institute of Standards and Technology, Tech. Rep., 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 3, 7, 12, 13, 25

[38] V. Lyubashevsky, "Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures," in *Advances in Cryptology – ASIACRYPT 2009*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 5912. Springer, Berlin, Heidelberg, Dec. 2009, pp. 598–616. [Online]. Available: https://www.iacr.org/archive/asiacrypt2009/59120596/59120596.pdf 3

[39] "Secure Hash Algorithm-3," National Institute of Standards and Technology, NIST FIPS PUB 202, U.S. Department of Commerce, Aug. 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.202 3

[40] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, D. Stehlé, and J. Ding, "CRYSTALS-KYBER," National Institute of Standards and Technology, Tech. Rep., 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 3, 12, 13

[41] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Advances in Cryptology – CRYPTO'99*, ser. Lecture Notes in Computer Science, M. J. Wiener, Ed., vol. 1666. Springer, Berlin, Heidelberg, Aug. 1999, pp. 537–554. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_34 3

[42] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf 3

[43] 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*, 1966, pp. 563–578. [Online]. Available: https://doi.org/10.1145/1464291.1464352 3

[44] T. Plantard, "Efficient word size modular arithmetic," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1506–1518, Jul. 2021. [Online]. Available: https://thomas-plantard.github.io/pdf/Plantard21.pdf 3, 4, 21

[45] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985. [Online]. Available: https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf 3

[46] OpenTitan Team, "OpenTitan's RTL freeze - leveraging transparency to create trustworthy computing · lowRISC: Collaborative open silicon engineering," Jun. 2023. [Online]. Available: https://lowrisc.org/news/opentitans-rtl-freeze-leveraging-transparency-to-create-trustworthy-computing/ 4

[47] "Advanced Encryption Standard (AES)," National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, Nov. 2001. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.197-upd1 4

[48] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the Association for Computing Machinery*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: https://doi.org/10.1145/359340.359342 4

[49] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology – CRYPTO'85*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed., vol. 218. Springer, Berlin, Heidelberg, Aug. 1986, pp. 417–426. [Online]. Available: https://doi.org/10.1007/3-540-39799-X_31 4

[50] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987. [Online]. Available: https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf 4

[51] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 1–19. [Online]. Available: https://www.spectreattack.com/spectre.pdf 4

[52] OpenTitan, "Opentitan big number accelerator (OTBN) technical specification," Jan. 2024. [Online]. Available: https://opentitan.org/book/hw/ip/otbn/index.html#security-features 4

[53] L. Chen, "Hsiao-code check matrices and recursively balanced matrices," *CoRR*, vol. abs/0803.1217, 2008. [Online]. Available: http://arxiv.org/abs/0803.1217 4

[54] H. Gross, D. Schaffenrath, and S. Mangard, "Higher-order side-channel protected implementations of Keccak," Cryptology ePrint Archive, Report 2017/395, 2017. [Online]. Available: https://eprint.iacr.org/2017/395 4

[55] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *CANS 16: 15th International Conference on Cryptology and Network Security*, ser. Lecture Notes in Computer Science, S. Foresti and G. Persiano, Eds., vol. 10052. Springer, Cham, Nov. 2016, pp. 124–139. [Online]. Available: https://eprint.iacr.org/2016/504 5

[56] V. Lyubashevsky and G. Seiler, "NTTRU: Truly fast NTRU using NTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 3, pp. 180–201, 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8293 5

[57] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A new hope," in *USENIX Security 2016: 25th USENIX Security Symposium*, T. Holz and S. Savage, Eds. USENIX Association, Aug. 2016, pp. 327–343. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim 5

[58] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4," Cryptology ePrint Archive, Report 2019/844, 2019. [Online]. Available: https://eprint.iacr.org/2019/844 5, 11

[59] P. Karl, J. Schupp, and G. Sigl, "The impact of hash primitives and communication overhead for hardware-accelerated SPHINCS+," in *Constructive Side-Channel Analysis and Secure Design*, R. Wacquez and N. Homma, Eds. Cham: Springer Nature Switzerland, 2024, pp. 221–239. [Online]. Available: https://eprint.iacr.org/2023/1767 6

[60] M.-J. O. Saarinen, "Benchmarking RISC-V post-quantum crypto," Nov. 2023. [Online]. Available: https://mjos.fi/doc/20231108-rvsummit-pqc.pdf 7

[61] S. Gao, B. Marshall, D. Page, and E. Oswald, "Share-slicing: Friend or foe?" *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, pp. 152–174, 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8396 8

[62] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 3, pp. 219–242, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8589 10

[63] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016. [Online]. Available: https://doi.org/10.1016/j.mejo.2016.04.006 10, 14

[64] G. Pope, "kyber-py," GitHub, Aug. 2024. [Online]. Available: https://github.com/GiacomoPope/kyber-py 11

[65] ——, "dilithium-py," GitHub, Aug. 2024. [Online]. Available: https://github.com/GiacomoPope/dilithium-py 11

[66] OpenTitan, "Introduction to OTBN," Jan. 2024. [Online]. Available: https://opentitan.org/book/hw/ip/otbn/doc/otbn_intro.html#performance 13

[67] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "FALCON," National Institute of Standards and Technology, Tech. Rep., 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 15

[68] M. Naehrig, E. Alkim, J. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila, "FrodoKEM," National Institute of Standards and Technology, Tech. Rep., 2020. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions 15

[69] D. J. Bernstein, B. B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. van Vredendaal, and B.-Y. Yang, "NTRU Prime," National Institute of Standards and Technology, Tech. Rep., 2020. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions 15

[70] T. Feneuil and M. Rivain, "MQOM — MQ on my Mind," National Institute of Standards and Technology, Tech. Rep., 2023. [Online]. Available: https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures 15

[71] C. Dobraunig, L. Grassi, L. Helminger, C. Rechberger, M. Schofnegger, and R. Walch, "Pasta: A case for hybrid homomorphic encryption," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 3, pp. 30–73, 2023. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/10956 15

[72] D. J. Bernstein *et al.*, "Chacha, a variant of Salsa20," in *Workshop record of SASC*, vol. 8, no. 1, 2008, pp. 3–5. [Online]. Available: https://cr.yp.to/chacha/chacha-20080120.pdf 15

[73] D. J. Bernstein, *The Salsa20 Family of Stream Ciphers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97. [Online]. Available: https://cr.yp.to/snuffle/salsafamily-20071225.pdf 15

[74] "Secure hash standard," National Institute of Standards and Technology, NIST FIPS PUB 180, U.S. Department of Commerce, May 1993. 15

[75] S. Gueron and F. Schlieker, "Speeding up R-LWE post-quantum key exchange," in *Secure IT Systems*, ser. Lecture Notes in Computer Science, B. B. Brumley and J. Röning, Eds. Cham: Springer International Publishing, 2016, pp. 187–198. [Online]. Available: https://eprint.iacr.org/2016/467 15, 23

[76] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *ACM CCS 2017: 24th Conference on Computer and Communications Security*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 1807–1823. [Online]. Available: https://acmccs.github.io/papers/p1807-almeidaA.pdf 15

Appendix A.
Auxiliary Material

TABLE A.1: Overview of ML-DSA's Parameter Sets [35].

Name (NIST level)	$\lvert pk \rvert$	$\lvert sig \rvert$	(k, ℓ)	η	τ	γ_1	γ_2	#reps
ML-DSA-44 (2)	1312 B	2420 B	$(4, 4)$	2	39	2^{17}	$(q-1)/88$	4.25
ML-DSA-65 (3)	1952 B	3293 B	$(6, 5)$	4	49	2^{19}	$(q-1)/32$	5.1
ML-DSA-87 (5)	2592 B	4595 B	$(8, 7)$	2	60	2^{19}	$(q-1)/32$	3.85

TABLE A.2: Overview of ML-KEM's Parameter Sets [34].

Name (NIST level)	$\lvert$ ek $\rvert$	$\lvert$ dk $\rvert$	$\lvert c \rvert$	$\lvert K \rvert$	k	(η_1, η_2)	(d_u, d_v)
ML-KEM-512 (1)	800 B	1632 B	768 B	32 B	2	$(3, 2)$	$(10, 4)$
ML-KEM-768 (3)	1184 B	2400 B	1088 B	32 B	3	$(2, 2)$	$(10, 4)$
ML-KEM-1024 (5)	1568 B	3168 B	1088 B	32 B	4	$(2, 2)$	$(11, 5)$

```
/* Mask out coefficients from buffer*/
bn.and coeffa, coeffsa, consts >> 192
bn.and coeffb, coeffsb, consts >> 192

/* Plantard multiplication: Twiddle * coeffb */
bn.mulqacc.wo.z coeffb, coeffb.0, twiddle.0, 192 /*
    (coeffb*R) mod 2^2d */
bn.add          coeffb, consts, coeffb >> 160 /* +1 */
bn.mulqacc.wo.z coeffb, coeffb.1, consts.2, 0 /* *q */
bn.rshi         wtmp, consts, coeffb >> 32 /* >> d
    */
/* Butterfly */
bn.subm coeffb, coeffa, wtmp
bn.addm coeffa, coeffa, wtmp

/* Shift results back to buffer and shift out used
    coefficients */
bn.rshi coeffsa, coeffa, coeffsa >> 32
bn.rshi coeffsb, coeffb, coeffsb >> 32
```

Listing A.1: CT butterfly on OTBN.

Appendix B.
ML-KEM and ML-DSA Algorithms

Algorithm B.1: ML-DSA: Key Generation, Following [35].

Output: Public key $pk \in \mathbb{B}^{32+32k(\mathsf{bitlen}(q-1)-13)}$
Output: Secret key $sk \in \mathbb{B}^{128+32((\ell+k)\cdot\mathsf{bitlen}(2\eta)+13k)}$
$\xi \leftarrow\!\!\$\ \{0,1\}^n$
$(\rho, \rho', K) \in \{0,1\}^n \times \{0,1\}^{2n} \times \{0,1\}^n \leftarrow \mathrm{H}(\xi \mid k \mid \ell, 4n)$
$(\boldsymbol{s}_1, \boldsymbol{s}_2) \in S_\eta^\ell \times S_\eta^k \leftarrow \mathsf{ExpandS}(\rho')$
$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k\times\ell} \leftarrow \mathsf{ExpandA}(\rho)$
$\boldsymbol{t} \leftarrow \mathsf{INTT}(\hat{\boldsymbol{A}} \circ \mathsf{NTT}(\boldsymbol{s}_1)) + \boldsymbol{s}_2$
$(\boldsymbol{t}_1, \boldsymbol{t}_0) \leftarrow \mathsf{Power2Round}(\boldsymbol{t}, 13)$
$pk \leftarrow \mathsf{pkEncode}(\rho, \boldsymbol{t}_1)$
$tr \in \{0,1\}^{2n} \leftarrow \mathrm{H}(\mathsf{BytesToBits}(pk), 2n)$
$sk \leftarrow \mathsf{skEncode}(\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0)$
return (pk, sk)

Algorithm B.2: ML-DSA: Signing, Following [35].

Input : Secret key $sk \in \mathbb{B}^{128+32((\ell+k)\cdot\mathsf{bitlen}(2\eta)+13k)}$
Input : Formatted message $M' \in \{0,1\}^*$
Input : Random or dummy variable $rnd \in \{0,1\}^n$
Output: Signature $\sigma \in \mathbb{B}^{32+\ell\cdot 32(1+\mathsf{bitlen}(\gamma_1-1))+\omega+k}$
$(\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0) \leftarrow \mathsf{skDecode}(sk)$
$\hat{\boldsymbol{s}}_1 \leftarrow \mathsf{NTT}(\boldsymbol{s}_1)$
$\hat{\boldsymbol{s}}_2 \leftarrow \mathsf{NTT}(\boldsymbol{s}_2)$
$\hat{\boldsymbol{t}}_0 \leftarrow \mathsf{NTT}(\boldsymbol{t}_0)$
$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k\times\ell} \leftarrow \mathsf{ExpandA}(\rho)$
$\mu \leftarrow \mathrm{H}(tr \| M', 2n)$
$\rho'' \leftarrow \mathrm{H}(K \| rnd \| \mu, 2n)$
$\kappa \leftarrow 0$
$(\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
while $(\boldsymbol{z}, \boldsymbol{h}) = \perp$ **do**
  $\boldsymbol{y} \leftarrow \mathsf{ExpandMask}(\rho'', \kappa)$
  $\boldsymbol{w} \leftarrow \mathsf{INTT}(\hat{\boldsymbol{A}} \circ \mathsf{NTT}(\boldsymbol{y}))$
  $\boldsymbol{w}_1 \leftarrow \mathsf{HighBits}(\boldsymbol{w})$
  $\tilde{c} \in \{0,1\}^{2\lambda} \leftarrow \mathrm{H}(\mu \| \mathsf{w1Encode}(\boldsymbol{w}_1), 2\lambda)$
  $(\tilde{c}_1, \tilde{c}_2) \in \{0,1\}^n \times \{0,1\}^{2\lambda-n} \leftarrow \tilde{c}$
  $c \leftarrow \mathsf{SampleInBall}(\tilde{c}_1)$
  $\hat{c} \leftarrow \mathsf{NTT}(c)$
  $\langle\langle c\boldsymbol{s}_1 \rangle\rangle \leftarrow \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{s}}_1)$
  $\langle\langle c\boldsymbol{s}_2 \rangle\rangle \leftarrow \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{s}}_2)$
  $\boldsymbol{z} \leftarrow \boldsymbol{y} + \langle\langle c\boldsymbol{s}_1 \rangle\rangle$
  $\boldsymbol{r}_0 \leftarrow \mathsf{LowBits}(\boldsymbol{w} - \langle\langle c\boldsymbol{s}_2 \rangle\rangle)$
  if $\|\boldsymbol{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\boldsymbol{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
  else
    $\langle\langle c\boldsymbol{t}_0 \rangle\rangle \leftarrow \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{t}}_0)$
    $\boldsymbol{h} \leftarrow \mathsf{MakeHint}(-\langle\langle c\boldsymbol{t}_0 \rangle\rangle, \langle\langle c\boldsymbol{s}_2 \rangle\rangle + \langle\langle c\boldsymbol{t}_0 \rangle\rangle)$
    if $\|\langle\langle c\boldsymbol{t}_0 \rangle\rangle\|_\infty \geq \gamma_2$ or # of 1's in $\boldsymbol{h} > \omega$ **then**
      $(\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
  $\kappa \leftarrow \kappa + \ell$
$\sigma \leftarrow \mathsf{sigEncode}(\tilde{c}, \boldsymbol{z} \bmod^{\pm} q, \boldsymbol{h})$
return σ

Algorithm B.3: ML-DSA: Verification, Following [35].

Input : Public key $pk \in \mathbb{B}^{32+32k(\mathsf{bitlen}(q-1)-13)}$
Input : Message $M' \in \{0,1\}^*$
Input : Signature $\sigma \in \mathbb{B}^{32+\ell\cdot 32(1+\mathsf{bitlen}(\gamma_1-1))+\omega+k}$
Output: Boolean

$(\rho, \boldsymbol{t}_1) \leftarrow \mathsf{pkDecode}(pk)$
$(\tilde{c}, \boldsymbol{z}, \boldsymbol{h}) \leftarrow \mathsf{sigDecode}(\sigma)$
if $\boldsymbol{h} = \perp$ **then**
  return false
$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k\times\ell} \leftarrow \mathsf{ExpandA}(\rho)$
$tr \leftarrow \mathrm{H}(\mathsf{BytesToBits}(pk), 2n)$
$\mu \in \{0,1\}^{2n} \leftarrow \mathrm{H}(tr \| M', 2n)$
$(\tilde{c}_1, \tilde{c}_2) \in \{0,1\}^n \times \{0,1\}^{2\lambda-n} \leftarrow \tilde{c}$
$c \leftarrow \mathsf{SampleInBall}(\tilde{c}_1)$
$\boldsymbol{w}'_{\text{Approx}} \leftarrow$
  $\mathsf{INTT}(\hat{\boldsymbol{A}} \circ \mathsf{NTT}(\boldsymbol{z}) - \mathsf{NTT}(c) \circ \mathsf{NTT}(2^{13} \cdot \boldsymbol{t}_1))$
$\boldsymbol{w}'_1 \leftarrow \mathsf{UseHint}(\boldsymbol{h}, \boldsymbol{w}'_{\text{Approx}})$
$\tilde{c}' \leftarrow \mathrm{H}(\mu \| \mathsf{w1Encode}(\boldsymbol{w}'_1), 2\lambda)$
return $[[\|\boldsymbol{z}\|_\infty < \gamma_1]] \wedge [[\tilde{c} = \tilde{c}']] \wedge [[\text{number of 1's in } \boldsymbol{h} \le \omega]]$

Algorithm B.4: K-PKE.KeyGen(d), Following [34].

Input : Randomness $d \in \mathbb{B}$
Output: Encryption key $\mathsf{ek}_{\text{PKE}} \in \mathbb{B}^{384k+32}$
Output: Decryption key $\mathsf{dk}_{\text{PKE}} \in \mathbb{B}^{384k}$

$(\rho, \sigma) \leftarrow G(d \| k)$
$N \leftarrow 0$
for $(i \leftarrow 0; i < k; i++)$ **do**
  for $(j \leftarrow 0; j < k; j++)$ **do**
    $\hat{\boldsymbol{A}}[i,j] \leftarrow \mathsf{SampleNTT}(\mathsf{XOF}(\rho, j, i))$
for $(i \leftarrow 0; i < k; i++)$ **do**
  $\boldsymbol{s}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
for $(i \leftarrow 0; i < k; i++)$ **do**
  $\boldsymbol{e}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
$\hat{\boldsymbol{s}} \leftarrow \mathsf{NTT}(\boldsymbol{s})$
$\hat{\boldsymbol{e}} \leftarrow \mathsf{NTT}(\boldsymbol{e})$
$\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{A}} \circ \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}$
$\mathsf{ek}_{\text{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\boldsymbol{t}}) \| \rho$
$\mathsf{dk}_{\text{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\boldsymbol{s}})$
return $(\mathsf{ek}_{\text{PKE}}, \mathsf{dk}_{\text{PKE}})$

Algorithm B.5: K-PKE.Encrypt($\mathsf{ek}_{\text{PKE}}, m, r$), Following [34].

Input : Encryption key $\mathsf{ek}_{\text{PKE}} \in \mathbb{B}^{384k+32}$
Input : Message $m \in \mathbb{B}^{32}$
Input : Random $r \in \mathbb{B}^{32}$
Output: Ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$

$N \leftarrow 0$
$\hat{\boldsymbol{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\text{PKE}}[0 : 384k])$
$\rho \leftarrow \mathsf{ek}_{\text{PKE}}[384k : 384k + 32]$
for $(i \leftarrow 0; i < k; i++)$ **do**
  for $(j \leftarrow 0; j < k; j++)$ **do**
    $\hat{\boldsymbol{A}}[i,j] \leftarrow \mathsf{SampleNTT}(\mathsf{XOF}(\rho, i, j))$
for $(i \leftarrow 0; i < k; i++)$ **do**
  $\boldsymbol{r}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(r, N))$
  $N \leftarrow N + 1$
for $(i \leftarrow 0; i < k; i++)$ **do**
  $\boldsymbol{e_1}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$
  $N \leftarrow N + 1$
$e_2 \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$
$\hat{\boldsymbol{r}} \leftarrow \mathsf{NTT}(\boldsymbol{r})$
$\boldsymbol{u} \leftarrow \mathsf{INTT}(\hat{\boldsymbol{A}}^\intercal \circ \hat{\boldsymbol{r}}) + \hat{\boldsymbol{e}}_{\boldsymbol{1}}$
$\mu \leftarrow \mathsf{Decompress}_1(\mathsf{ByteDecode}_1(\mathsf{m}))$
$v \leftarrow \mathsf{INTT}(\hat{\boldsymbol{t}}^\intercal \circ \hat{\boldsymbol{r}}) + e_2 + \mu$
$c_1 \leftarrow \mathsf{ByteEncode}_{d_u}(\mathsf{Compress}_{d_u}(\boldsymbol{u}))$
$c_2 \leftarrow \mathsf{ByteEncode}_{d_v}(\mathsf{Compress}_{d_v}(\boldsymbol{v}))$
return $c \leftarrow (c_1 || c_2)$

Algorithm B.6: K-PKE.Decrypt($\mathsf{dk}_{\text{PKE}}, c$), Following [34].

Input : Decryption key $\mathsf{dk}_{\text{PKE}} \in \mathbb{B}^{384k}$
Input : Ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$
Output: Message $m \in \mathbb{B}^{32}$

$c_1 \leftarrow c[0 : 32 d_u k]$
$c_2 \leftarrow c[32 d_u k : 32(d_u k + d_v)]$
$\boldsymbol{u} \leftarrow \mathsf{Decompress}_{d_u}(\mathsf{ByteDecode}_{d_u}(c_1))$
$v \leftarrow \mathsf{Decompress}_{d_v}(\mathsf{ByteDecode}_{d_v}(c_2))$
$\hat{\boldsymbol{s}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{dk}_{\text{PKE}})$
$w \leftarrow v - \mathsf{INTT}(\hat{\boldsymbol{s}}^\intercal \circ \mathsf{NTT}(\boldsymbol{u}))$
$m \leftarrow \mathsf{ByteEncode}_1(\mathsf{Compress}_1(w))$
return m

Appendix C.
Details of OTBN Implementations

This section gives additional details on the implementations on plain OTBN that go beyond the ones provided in Section 3.

C.1. NTT and Multiplication in NTT Domain

C.1.1. Plantard Multiplication. In addition to the description in Section 3.1.1, the exact sequence of operations for multiplying two elements a and b in $\mathbb{F}_q$ using Plantard multiplication [44] is given in the following:

1) Load constants into WDR `consts` = $(m||q||1||R)$, where $m = 2^d - 1$ and $R = q^{-1} \mod 2^{2d}$. Load a and b into WDRs `coeffa` and `coeffb`.
2) Multiply a and b as 64-bit integers: `bn.mulqacc.wo.z wtmp, coeffa.0, coeffb.0, 0`. WDR `wtmp` now has $ab \mod 2^{2d}$ at its first quad word.
3) Compute $(ab \mod 2^{2d})R$, and keep the result modulo 2^{2d}: `bn.mulqacc.wo.z wtmp, wtmp.0, consts.0, 192`. 192 is the amount of bits the result is shifted left, meaning, by dropping some top-bits, the final result $\mod 2^{2d}$ will be in the fourth quad word of `wtmp` for ML-DSA. However, for ML-KEM, $2d$ is only 32 and since the shift amount can only be multiples of 64, `wtmp` must be masked with m: `bn.and wtmp, wtmp, consts`.
4) Shift right by t bits and add 1: `bn.add wtmp, consts, wtmp >> t`, where $t = 144$ for ML-KEM and $t = 160$ for ML-DSA. The result will be in the second quad word of `wtmp`.
5) Multiply by q: `bn.mulqacc.wo.z wtmp, wtmp.1, consts.2, 0`.
6) Shift right by d bits: `bn.rshi wtmp, bn0, wtmp >> d`.

C.1.2. Reduction. Next to modular multiplications, there are two places throughout our implementations where we require some form of explicit modular reductions:

1) Before checking the norm bound in ML-DSA: the centralized representative in $[\![\frac{-q-1}{2}, \frac{q-1}{2}]\!]$ of coefficients is required in this step as $\|w\|_\infty$ is defined as $|w \mod^{\pm} q|$. We use (variants) of the reduce32 function as also used in the reference implementation, as well as constant-time conditional subtractions to achieve this goal.
2) After the application of (pair-)pointwise multiplication with pseudo-vector accumulation, where the values can grow beyond q before the INTT. Inputs to the INTT must be in $[\![0, q]\!]$ to avoid getting negative results that cannot be reduced back into the positive domain implicitly using `bn.subm`. We perform this reduction using a variant of reduce32 for ML-DSA and using the Plantard multiplication with the constant $((-2^{2d}) \mod q)R \mod 2^{2d}$ for ML-KEM.

C.1.3. NTT. As the implementation of a 4-layer merge is not straight forward, we outline some more details on our approach here.

Layer merge. In our implementation, $13(n/d)$ input coefficients are loaded on 13 WDRs, called "buffer registers". The rest is loaded directly from the memory during the transformation with help of the GPRs as we do not have enough WDRs for storing all input data and doing the computation simultaneously. Since coefficients indexed $(16i|i \in [\![0, 15]\!])$ are needed for the first 4-layer merge, the required coefficients are masked out and moved to another set of 16 WDRs, called the "working state"; while the unused ones are still kept in the buffer registers. In addition, we need one register for storing constants in Plantard multiplication as explained in Section 3.1.1, one for holding intermediate values, and another one for holding twiddle factors (cf. Listing A.1), summing up to 32 registers for an NTT or INTT invocation. As OTBN only has a 64×64-bit multiplier, it does not make sense to load more than four twiddle factors into a WDR – regardless of whether they are 32 or 64-bit in size – as it would incur additional overhead for data movement. This fits perfectly for ML-DSA, because the size of the twiddle factors are doubled to 64-bit due to Plantard representation, but not for ML-KEM. Due to our register allocation strategy, for each iteration of a 4-layer merge, two loads of twiddle factors are needed, and they are reloaded in every iteration to enable the buffering strategy mentioned above.

C.2. Sampling

Rejection Sampling in $[\![0, q[\![$. Listing C.1 shows how we implement the rejection sampling on the output bytes of SHAKE256. We check if one coefficient candidate in the case of ML-DSA or two in the case of ML-KEM c (i.e., `cand`), made up of three bytes of SHAKE256 output read from `shake_reg`, is less than q. If this is the case, the candidate is shifted into the result register `accumulator`. In case the candidate is rejected, the corresponding three bytes (for ML-DSA) or 12 bits (for ML-KEM) are shifted out of `shake_reg` and we sample the next candidate(s). By bundling the accepted candidates into a WDR before storing, we can reduce the memory-access cost. Also note that even though we cannot early-exit from the hardware loop `loopi`, it is still used in our implementation because it costs only a single cycle and does not require either additional instructions or registers to handle the loop logic in comparison to a traditional while-loop, which would be less efficient overall.

Sampling in $[\![-\eta, \eta]\!]$. Both the binomial sampling in ML-KEM and the rejection sampling in ML-DSA yield integers that fall within a signed range. For efficiency and compatibility with unsigned integer calculations in subsequent routines, we employ modular subtraction `bn.subm` in both sampling methods of ML-KEM and ML-DSA, replacing standard subtraction `bn.sub`. Pseudo vectorization is also applied to enhance bitwise addition in binomial sampling of ML-KEM.

```
_poly_uniform_base_inner_loop:
  loopi 10, 12
    /* n coefficients are sampled?*/
    beq     outp, t0, _skip_store1
    /* Extract 3-byte candidate c from shake output */
    bn.and  cand, coeff_mask, shake_reg
    /* c-q */
    bn.cmp  cand, mod
    /* Read flags Z, L, M, C*/
    csrrs   a4, 0x7C0, zero
    /* Extract M, C */
    andi    a4, a4, 3
    /* Reject if M!=1 & C!=1 i.e. (q <= c) */
    bne     a4, const_3, _skip_store1
    bn.rshi accumulator, cand, accumulator >> 32
    addi    accumulator_count, accumulator_count, 1
    /* Accumulator is full of 8 coeffs? */
    bne     accumulator_count, const_8, _skip_store1
    /* Store full accumulator to memory */
    bn.sid  accumulator_idx, 0(outp++)
    li      accumulator_count, 0
_skip_store1:
    /* Shift out used 3 bytes */
    bn.or   shake_reg, bn0, shake_reg >> 24
  ret
```

Listing C.1: Inner loop of uniform sampling in ML-DSA on OTBN.

C.3. Bit Packing

The general idea of bit packing in ML-KEM and ML-DSA is to arrange coefficients tightly next to each other such that there are no free bits between any two of them to save space for data transfer. This mostly boils down to shifting coefficients with `bn.rshi` and an extensive use of WDRs for caching data on OTBN. The unpacking is implemented using the same principal. While the packing is similar for all functions in both ML-KEM and ML-DSA, the data processing step before or after it varies.

(Un)packing Coefficients in Negative Input Range in ML-DSA. As an example, we consider the function for packing coefficients that are in $[\![-\eta, \eta]\!]$ in the case of ML-DSA-44, where $\eta = 2$. In the C reference implementation [19], the coefficient to be packed is a signed integer, and thus, it is subtracted from η in order to retrieve an unsigned result in $[\![0, 2\eta]\!]$. As we made the choice to operate on unsigned integers, we cannot simply perform this subtraction, as, e.g., -1 maps to $q-1$ in our case, and $\eta - (q-1)$ is certainly not in the desired range. All we need to do is to apply `bn.subm` instead of the regular `bn.sub`, which will move the result of the subtraction back into the positive domain, yielding values in $[\![0, 2\eta]\!]$.

Encoding and Decoding of Hint Vector in ML-DSA. The encoding and decoding in the C reference implementation [19] uses a lot of control logic based on the signature data, as well as unaligned memory accesses, both of which are weaknesses of OTBN. Thus, we decided to implement both using the base instruction set operating on 32-bit GPRs, which is more useful for managing the control flow and less restricted regarding memory access. The reason why this operation still costs many cycles is the manual 4-byte alignment of addresses and the subsequent extraction of the desired byte, based on the lower two bits of the unaligned address, to simulate byte-aligned memory access.

Compression and Decompression of Ciphertext in ML-KEM. In the current C reference implementation [20], the compression of an element $x \in \mathbb{F}_q$ to $d_{\{u,v\}}$ bits replaces the division by q by an addition followed by a multiplication and a right shift for it to be constant-time. Without question, multiplication must be done individually. For $d_v = 4$ in ML-KEM-512, addition and shifting can be pseudo-vectorized, but not for other cases of $d_{\{u,v\}}$ because after the left shift of $d_{\{u,v\}}$ bits (cf. Table A.2), the size of integers is at least 17-bit, exceeding a 16-bit vector element. We certainly can arrange the coefficients into 32-bit vector elements and still perform a pseudo shift/addition. Nevertheless, after this costly arrangement, coefficients must be extracted again for the multiplication, neutralizing the saving from the pseudo vectorization.

Appendix D. Details of OTBN$^{\text{KMAC}}_{\text{Ext.}}$ Implementations

In the following, we provide supplementary details to the explanations given in Section 5.

D.1. Polynomial Addition and Subtraction

To demonstrate the simplicity of the code using our new extensions, we present two simple examples for polynomial addition and subtraction in Listings D.1 and D.2.

D.2. NTT & INTT

A visualization for the transposition operation mentioned in Section 5.2.2 is given in Figure 8: We transpose an 8×8 matrix of elements that is obtained by considering the wide registers as rows of a matrix in order to achieve an appropriate stride for the following computation.

```
loopi 32, 4
  bn.lid vec_1_idx, 0(src1++)
  bn.lid vec_2_idx, 0(src2++)

  bn.addvm.8S vec_1, vec_1, vec_2

  bn.sid vec_1_idx, 0(dst++)
```

Listing D.1: Vectorized addition on OTBN$^{\text{KMAC}}_{\text{Ext.}}$.

```
loopi 32, 4
  bn.lid vec_1_idx, 0(src1++)
  bn.lid vec_2_idx, 0(src2++)

  bn.subvm.8S vec_1, vec_1, vec_2

  bn.sid vec_1_idx, 0(dst++)
```

Listing D.2: Vectorized subtraction on OTBN$^{\text{KMAC}}_{\text{Ext.}}$.

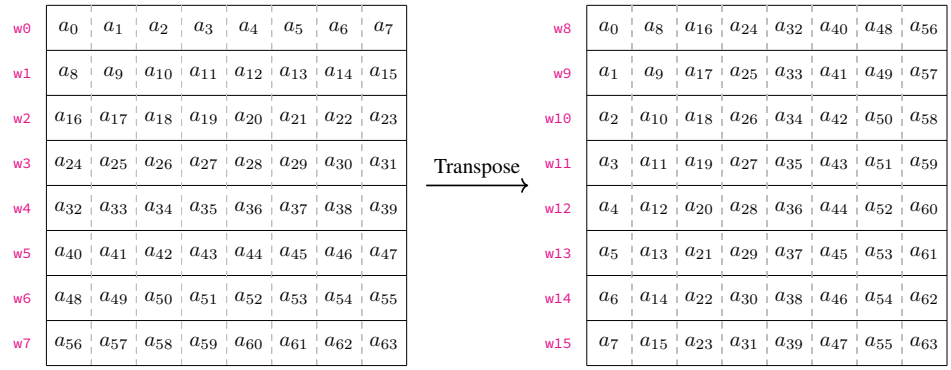

Figure 8: Visualization of the transposition.

D.3. Base Multiplication in NTT Domain in ML-KEM

ML-KEM. In ML-KEM, the need for a 2×2 schoolbook multiplication makes the implementation slightly more involved while still remaining elegant compared to the plain implementation. For computing the product $\hat{c} = \hat{c}_{2i} + \hat{c}_{2i+1}X$ between two linear polynomials $\hat{a} = \hat{a}_{2i} + \hat{a}_{2i+1}X$, $\hat{b} = \hat{b}_{2i} + \hat{b}_{2i+1}X$, we compute $\hat{a}_{2i} = \hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}\zeta^{2\text{br}_7(i)+1}$ and $\hat{c}_{2i+1} = \hat{a}_{2i}\hat{b}_{2i+1} + \hat{b}_{2i}\hat{a}_{2i+1}$. For this, we need to multiply two coefficients of each polynomial that are not located at the same index in their respective WDRs. Listing D.3 shows how the pair-pointwise multiplication is done in ML-KEM thanks to the transpose instructions `bn.trn1` and `bn.trn2`. Specifically, `coeffsa` = $(a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$ and `coeffsb` = $(b_{n-1}, b_{n-2}, \ldots, b_1, b_0)$ are loaded from the memory. The multiplication $a_i b_i$ is obvious with `bn.mulvm` (Line 2). Directly vectorizing the multiplication with roots of unity requires an additional $n/2 = 128$ multiplications of $a_{2i}b_{2i}$ with 1. However, to save $128 \times 16 = 2048$ multiplications per pair-pointwise operation, we compute a second input vector `coeffsd`, pack all coefficients to be multiplied from `coeffsb` and `coeffsd` in `wtmp` and perform the vectorized multiplication. The result is then unpacked with one `bn.rshi` and two `bn.trn1` (Line 17, 19). To compute the multiplication $\hat{a}_{2i}\hat{b}_{2i+1}$ and $\hat{b}_{2i}\hat{a}_{2i+1}$, we right-shift `coeffsb` by 16 bits (Line 6) and use `bn.trn1` to reorder `coeffsb` to be $(b_{n-2}, b_{n-1}, \ldots, b_2, b_3, b_0, b_1)$ (Line 7). In the end, we have the result vectors `wtmp0` = $(a_{n-1}b_{n-1}, \ldots, a_1b_1, a_0b_0)$ and `coeffsb` = $(a_{n-1}b_{n-2}, a_{n-2}b_{n-1}, \ldots, a_3b_2, a_2b_3, a_1b_0, a_0b_1)$. For the additions, we only need to use one `bn.trn1` on `wtmp0` and `coeffsb` to make $(a_{n-2}b_{n-1}, a_{n-2}b_{n-2}, \ldots, a_0b_1, a_0b_0)$ (Line 22) and one `bn.trn2` to make $(a_{n-1}b_{n-2}, a_{n-1}b_{n-1}, \ldots, a_1b_0, a_1b_1)$ (Line 23). The final result is obtained by adding the two vectors `coeffsa` and `coeffsb` together.

```
1  /* a1b1, a0b0 */
2  bn.mulvm.16H wtmp0, coeffsa, coeffsb
3  bn.mulvm.16H wtmp1, coeffsc, coeffsd
4
5  /* a0b1, a1b0 */
6  bn.rshi      wtmp, bn0, coeffsb >> 16
7  bn.trn1.16H  coeffsb, wtmp, coeffsb
8  bn.mulvm.16H coeffsb, coeffsa, coeffsb
9
10 bn.rshi      wtmp, bn0, coeffsd >> 16
11 bn.trn1.16H  coeffsd, wtmp, coeffsd
12 bn.mulvm.16H coeffsd, coeffsc, coeffsd
13
14 /* Multiply with Twiddle factors */
15 bn.trn2.16H  wtmp, wtmp0, wtmp1
16 bn.mulvm.16H wtmp, wtmp, twiddles
17 bn.trn1.16H  wtmp0, wtmp0, wtmp
18 bn.rshi      wtmp, bn0, wtmp >> 16
19 bn.trn1.16H  wtmp1, wtmp1, wtmp
20
21 /* a1b1+a0b0; a1b0+a0b1 */
22 bn.trn1.16H  coeffsa, wtmp0, coeffsb
23 bn.trn2.16H  coeffsb, wtmp0, coeffsb
24 bn.addvm.16H res0, coeffsa, coeffsb
25
26 bn.trn1.16H  coeffsc, wtmp1, coeffsd
27 bn.trn2.16H  coeffsd, wtmp1, coeffsd
28 bn.addvm.16H res1, coeffsc, coeffsd
```

Listing D.3: ML-KEM pair-pointwise multiplication on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$.

D.4. Sampling

Rejection Sampling. Although it is possible to vectorize the rejection sampling routines in ML-DSA and ML-KEM as introduced in [75] and applied in [19], [20], our ISA extensions are not tailored to apply this optimization. The lack of a bit-mask-based permutation instruction inhibits the application of the technique in our case.

Sampling in $[\![-\eta, \eta]\!]$ & Binomial Sampling. As opposed to the general uniform sampling, the sampling of coefficients in $[\![-\eta, \eta]\!]$ for ML-DSA clearly benefits from our proposed instructions. This is due to a sequence of arithmetic operations that are applied on each sampled coefficient after it passes the rejection step. Instead of applying these operations on each coefficient individually, we "collect" the coefficients in a WDR until it is filled up and then compute in a vectorized fashion. In the binomial sampling routine of ML-KEM, we apply a similar trick. This saves one of seven

instructions inside the innermost loop which amounts to about 15% of the overall runtime of the binomial sampling for the case of $\eta = 2$.

D.5. Further Applications

Bit Packing. The bit-packing functions profit from the availability of the WDRs in the baseline implementation already. However, in instances where the coefficients need to be subtracted from a constant value for transforming between the representation on the wire and the representation as a coefficient, the `bn.subvm` instruction can be leveraged, instead of performing individual subtractions. Especially, `bn.subvm` can be used to implicitly unpack the coefficients into their representation mod^+.

Reductions. Throughout the implementation of ML-KEM, no explicit reductions are required as all operations implicitly reduce the processed data and therefore inhibit growth of the coefficients. In ML-DSA, also all arithmetic operations provide implicit reductions, however, since we decided to operate mod^+, we need to transform the coefficients into their centralized representatives $\mathrm{mod}^\pm$ before performing the norm bound check. This transformation can be done using the reduce32 function, which we can implement efficiently using our extensions.

Rounding. While the rounding in ML-DSA only accounts for a small fraction of the runtime, we still note that we have been able to fully vectorize the implementations of the Decompose and Power2Round functions, which highlights the universality of our extensions.

Appendix E. Additional Results

TABLE E.1: ML-KEM and ML-DSA memory usage. All numbers refer to bytes.

NIST Level	Platform	ML-KEM			ML-DSA		
		K	E	D	K	S	V
ML-KEM-512 ML-DSA-44	OTBN	3232	3712	3840	37 740	50 108	36 156
	$\text{OTBN}^{\text{KMAC}}$	3232	3712	3840	37 328	48 880	34 928
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	2784	3264	3392	37 248	48 800	34 848
	OpenTitan [24][a,b]	—	—	—	—	—	$\leq$ 32 000
	Skylake [37]	—	—	—	—	—	—
	Cortex-M4 [12]	4364	5436	5412	38 296	49 416	36 184
	[27]	—	—	—		61 216[c]	
ML-KEM-768 ML-DSA-65	OTBN	4256	4736	4864	60 268	77 628	57 692
	$\text{OTBN}^{\text{KMAC}}$	4256	4736	4864	59 856	76 400	56 464
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	3808	4288	4416	59 776	76 320	56 384
	OpenTitan [24][a,b]	—	—	—	—	—	$\leq$ 32 000
	Skylake [37]	—	—	—	—	—	—
	Cortex-M4 [12]	5396	6468	6452	60 824	68 864	57 720
	[27]	—	—	—		92 720[c]	
ML-KEM-1024 ML-DSA-87	OTBN	5280	5760	5888	97 132	119 900	92 764
	$\text{OTBN}^{\text{KMAC}}$	5280	5760	5888	96 720	118 672	91 536
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	4832	5312	5440	96 640	118 592	91 456
	OpenTitan [24][a,b]	—	—	—	—	—	$\leq$ 32 000
	Skylake [37]	—	—	—	—	—	—
	Cortex-M4 [12]	6436	7500	7484	97 688	115 968	92 824
	[27]	—	—	—		139 840[c]	

[a] Including modified variant of OTBN, parts of the execution on Ibex Core.
[b] Round 3 KYBER.
[c] Full-scheme result.

TABLE E.2: ML-KEM and ML-DSA code size. All numbers refer to bytes.

NIST Level	Platform	ML-KEM				ML-DSA			
		Text	Const	I/O	Total[a]	Text	Const	I/O	Total[a]
ML-KEM-512 ML-DSA-44	OTBN	18 412	3744	3360	22 156	25 484	5664	9696	31 148
	$\text{OTBN}^{\text{KMAC}}$	15 348	2688	3360	18 036	20 792	4832	9696	25 624
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	9740	1568	3360	11 308	18 496	2624	9696	21 120
	[29][b]	—	—	—	12 532	—	—	—	—
	[27]	—	—	—	—	20 624	—	—	—
	Cortex-M4 [12]	—	—	—	15 824	—	—	—	18 596
ML-KEM-768 ML-DSA-65	OTBN	18 952	3744	4832	22 696	25 748	5664	12 704	31 412
	$\text{OTBN}^{\text{KMAC}}$	15 888	2688	4832	18 576	20 780	4832	12 704	25 612
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	10 232	1568	4832	11 800	18 384	2624	12 704	21 008
	[29][b]	—	—	—	11 658	—	—	—	—
	[27]	—	—	—	—	20 052	—	—	—
	Cortex-M4 [12]	—	—	—	15 992	—	—	—	18 588
ML-KEM-1024 ML-DSA-87	OTBN	22 056	3744	6464	25 800	26 568	5664	15 520	32 232
	$\text{OTBN}^{\text{KMAC}}$	18 992	2688	6464	21 680	21 596	4832	15 520	26 428
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	13 732	1568	6464	15 300	19 348	2624	15 520	21 972
	[29][b]	—	—	—	12 874	—	—	—	—
	[27]	—	—	—	—	20 324	—	—	—
	Cortex-M4 [12]	—	—	—	16 912	—	—	—	18 468

[a] Sum of Text and Const.
[b] Round 2 KYBER.

Appendix F.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

This paper addresses the problem of accelerating lattice-based cryptography in hardware without loosing generality and flexibility. It extends the OpenTitan root of trust, achieving significant speedups for ML-KEM and ML-DSA, at the price of a slight increase in hardware complexity. The first extension accelerates the communication between the OBTN and KMAC core, the other introduces new instructions to the OBTN's ISA to speedup polynomial arithmetic.

F.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

F.3. Reasons for Acceptance

1) The paper creates a new tool to enable future science. The paper brings valuable extensions to the OpenTitan silicon root of trust, an open-source project with real-world applications. The engineering effort is significant. All artifacts will be made publicly available with permissive licenses. Hence, the contributions of this paper have the potential to make positive practical impact and enable future research.
2) Provides a Valuable Step Forward in an Established Field. The paper identifies the bottlenecks for the execution of PQ cryptography on a relevant root-of-trust design (OpenTitan), extends the design to avoid these bottlenecks, and thoroughly evaluates it. A specific contribution of the design proposed in the paper is that it focuses on modular expansions rather than dedicated accelerators, providing benefits for both traditional and PQ cryptography.