Towards ML-KEM & ML-DSA on OpenTitan

Amin Abdulrahman[1], Felix Oberhansl[2], Hoang Nguyen Hien Pham[3,4], Jade Philipoom[5], Peter Schwabe[1,6], Tobias Stelzer[2] and Andreas Zankl[2,7]

[1] Max Planck Institute for Security and Privacy (MPI-SP), Bochum, Germany,
amin@abdulrahman.de, peter@cryptojedi.org
[2] Fraunhofer Institute for Applied and Integrated Security (AISEC), Garching, Germany,
firstname.lastname@aisec.fraunhofer.de
[3] BULL SAS, Les Clayes-sous-Bois, France
[4] Université Grenoble Alpes, CNRS, IF, Grenoble, France,
hoang-nguyen-hien.pham{@eviden.com, @univ-grenoble-alpes.fr},
nguyenhien.phamhoang@gmail.com
[5] zeroRISC, Boston, USA, jadep{@zerorisc.com, @opentitan.org}
[6] Radboud University, Nijmegen, The Netherlands
[7] 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 primary algorithms selected by NIST for standardization—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 12% in OTBN, which corresponds to an increase of less than 2% for the full Earlgrey OpenTitan core.

Keywords: Post-quantum cryptography · ML-KEM · ML-DSA · OpenTitan · instruction set extension · HW/SW co-design

1 Introduction

In July 2022, the NIST post-quantum standardization effort produced as first output a selection of four primitives for standardization: the signature schemes CRYSTALS-Dilithium [LDK+22], Falcon [PFH+22], and SPHINCS+ [HBD+22], and the key encapsulation mechanism CRYSTALS-Kyber [SAB+22]. Out of the three signature schemes, *"NIST selected Dilithium as the primary signature algorithm that it will recommend for general use"* [AAA+20, Sec. 1]. Draft standards for three of those schemes—Dilithium, SPHINCS+, and Kyber—were published in August 2023 [NIS23b, NIS23c, NIS23a] and the final standards for these three schemes are expected to be ready later this year. The standards will use different naming of the algorithms: Dilithium will be standardized as ML-DSA, SPHINCS+ as SLH-DSA, and Kyber as ML-KEM.

Already now, before the standards are finalized, several applications have started using the new primitives, in particular Kyber. The most notable examples are integration

into TLS by Google, Cloudflare, and Mozilla[1] [ABBO24, EWP+23], integration into the Signal secure messenger [KS24], into Apple's iMessage protocol [App24], and into the Zoom end-to-end encrypted video-conferencing protocol [BBC+24].

While these prominent examples of early adoption certainly inspire hope for a speedy migration of at least parts of our digital infrastructure to post-quantum cryptography (PQC), it is also worth noting that all these examples share characteristics that simplify quick deployment: all end points of communication are controlled by one or at most a few entities, they do not require protections against local (e.g., power or EM analysis) side-channel attacks, deployment can be achieved through already in-place update infrastructure, and, most importantly, implementations of the new schemes are entirely in software.

Applications that rely on hardware acceleration for cryptography will naturally take more time to migrate, but significant effort has already been invested into the implementation of post-quantum cryptography—mostly lattice-based cryptography—on embedded platforms with hardware accelerators. These works can roughly be grouped into two categories. The first category studies how *existing* hardware accelerators for big-integer arithmetic can be used to speed up the polynomial arithmetic underlying structured-lattice-based schemes [AHH+18, GMR21]. The second category aims at building dedicated accelerators for PQC [BUC19].

We argue that for the foreseeable future, neither of these approaches is fully satisfactory. The attempt to utilize existing hardware is certainly highly relevant to deploy PQC on legacy devices that are already in the field. However, it is also rather clear that future generations of security chips will want to take acceleration of PQC into account in the design phase. Also, the two primary algorithms selected by NIST for general use, ML-DSA and ML-KEM, tightly integrate multiplication through the number-theoretic transform (NTT) into the algorithm specification. This makes it hard to gain performance when employing a different multiplication algorithm that is amenable to acceleration through fast big-integer arithmetic. For example, the "Kyber" implementation described in [AHH+18] is incompatible to the actual Kyber specification for exactly this reason.

Post-quantum-only security chips are most likely what we will want in the very long run, but 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. For example, all of the early-adopter applications listed above, employ such hybrid solutions. It would certainly be possible to deploy dedicated PQC accelerators *in addition to* ECC and RSA accelerators, but as we show in this paper, the resulting increase in hardware resources is unnecessary.

Contributions and organization of this paper. We show that small modifications and extensions to the hardware design and ISA of existing cryptographic hardware, designed to accelerate ECC and RSA, yields highly efficient accelerators for both traditional asymmetric cryptography, and novel post-quantum schemes. More specifically, we ready the OpenTitan hardware root of trust (RoT) for the lattice-based algorithms ML-DSA and ML-KEM. 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, build system, etc. 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, which aims at accelerating only (modular)

[1] https://hg.mozilla.org/releases/mozilla-release/file/d3c71a6fc9a1aecf1fe04f8de2fc0b816588e677/security/manager/ssl/nsNSSIOLayer.cpp#l1439

big-integer arithmetic in hardware; higher-level routines like ECC point operations or exponentiation are implemented in software. In our upgrade to the OpenTitan platform, we proceed as follows:

- We start by carefully optimizing ML-DSA and ML-KEM on OTBN in a software-only approach, i.e., without requiring any modifications to the OpenTitan hardware design, except for an increase in data memory. This implementation serves as a baseline for our performance evaluation and a starting point for profiling. It is described in detail in Section 3.
- Unsurprisingly, we identify Keccak permutations as a major bottleneck in our software-only implementation. We resolve this by adding an interface from OTBN to the Keccak accelerator. This interface and the resulting increase in performance are described in Section 4.
- The main remaining bottleneck is polynomial arithmetic. In order to speed up this part, we propose extensions to the OTBN ISA, which let us operate on the existing 256-bit-wide registers as vectors of small integers. This ISA extension is intentionally designed as a generic vector instruction set, rather than a set of highly specific instructions targeting only ML-DSA and ML-KEM. This decision is partly motivated by the fact that more specific instructions would not result in a dramatic gain in performance, partly by the idea that the instructions will also be useful for the implementation of other cryptographic schemes (for a discussion, see Section 8), and partly as such generic extensions are similar in spirit to the existing generic extensions for big-integer arithmetic. The ISA extensions and their estimated performance increase are discussed in Section 5.
- We then present our modifications to the OpenTitan hardware that implement the interface from OTBN to the Keccak core and our ISA extensions. In fact, we present two different approaches, one that aims at sharing as much hardware as possible between big-number arithmetic and vector arithmetic and one that investigates possible gains in performance of ML-DSA and ML-KEM at the expense of larger investment in circuitry. The hardware implementations are described in Section 6.
- Overall, our final evaluation shows that with an increase in circuit area of only 11.40% (plus the required increase in data memory) in the OTBN core, we achieve a speed-up of up to a factor of 9.14 in ML-DSA and up to 8.87 in ML-KEM. This increase in OTBN hardware size corresponds to an increase of only 1.79% of the full OpenTitan *Earl Grey* top-level design. We present details of these results and compare them to related work in Section 7 and conclude the paper with a discussion in Section 8.

Artifact. We will make all software and hardware described in this paper publicly available under permissive licenses compatible to the OpenTitan license as soon as possible.

Related work. Research in PQC implementations has seen a long series of work with a wide spectrum, ranging from pure software to pure hardware designs, across multiple platforms. Extensive studies have been conducted on software implementations of Kyber and Dilithium for the Arm Cortex-M4 [HZZ+22, HAZ+24, BRS22, AHKS22, GKS21, ABCG20, BKS19]. Highly optimized implementations for single instruction multiple data (SIMD)-architecture have been presented for, e.g., the Intel AVX2 [Dil23] and Arm Neon [BHK+22] platforms. To enhance PQC performance on resource-constrained devices, hardware/software co-designs have been explored, where compute-intensive operations are offloaded to hardware, which yields efficient performance while maintaining flexibility for

future security updates. A notable related work is a configurable post-quantum arithmetic logic unit (ALU) for the OTBN unit [SOSK23], accelerating polynomial arithmetic of Dilithium, Kyber, and Falcon, with the Dilithium verification procedure as a case study. Other tightly coupled accelerators for post-quantum cryptography, targeting different performance/resource trade-offs, have been presented in [KSFS24, NDMZ+21, FSS20, LTQ+24, LQYW24]. Among which, [KSFS24, FSS20, LTQ+24] 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 [ZXXH22], targeting high-speed implementation on edge nodes, 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 [FBR+22].

2 Preliminaries

2.1 Notation

We mainly follow the conventions of the National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 203 [NIS23a] and 204 [NIS23b].

Intervals of integers are denoted by double square brackets, e.g., $[\![a, b]\!] = \{a, a+1, \ldots, b\}$. An outwards-facing double square bracket declares the interval excluding the respective endpoint, e.g., $[\![a, b[\![$, denotes the set $\{a, a+1, \ldots, b-1\}$.

We denote polynomials by lowercase letters, e.g., a, vectors of polynomials by lowercase boldface letters, e.g., $\boldsymbol{a}$ and matrices of polynomials by uppercase boldface letters, e.g., $\boldsymbol{A}$. The polynomial ring R_q is defined as $\mathbb{F}_q[X]/\langle X^n + 1\rangle$, where $\mathbb{F}_q = \mathbb{Z}/q\mathbb{Z}$, q is a prime number and $n \in \mathbb{N}$. If not stated otherwise, $n = 256$ is the polynomial size in the remainder of this paper. A polynomial $a = a_0 + a_1 X + \cdots + a_{n-1}X^{n-1} \in R_q$ is represented as a vector $(a_0, \ldots, a_{n-1}) \in \mathbb{F}_q^n$.

For congruences, we follow the notation from FIPS 204 [NIS23b]: For odd (respectively even) q, the centralized reduction $r' = r \bmod^{\pm} q$ is defined as the unique number in $[\![-\frac{q-1}{2}, \frac{q-1}{2}]\!]$ (respectively $[\![-\frac{q}{2}, \frac{q}{2}]\!]$) that fulfills $r' \equiv r \bmod q$. Similarly, $r' = r \bmod^{+} q$ denotes the unique number in $[\![0, q[\![$ that fulfills $r' \equiv r \bmod q$. We also denote $r \bmod^{+} 2^d$ as $[r]_d$ and $\lfloor \frac{r}{d} \rfloor$ as $[r]^d$, with $d \in \mathbb{N}$.

Let $\mathbb{B}$ denote the set of 8-bit integers $\{0, \ldots, 255\}$. For a byte-array $B \in \mathbb{B}^m$, $B[i]$ denotes the entry at index i, while $B[i : j]$ denotes the subarray from index i to j of B, where $i < j$.

2.2 ML-DSA

The draft of FIPS 204, available since August 2023 [NIS23b], specifies the digital signature scheme Dilithium [DKL+18, LDK+22] under the name module-lattice-based digital signature algorithm (ML-DSA).

ML-DSA is believed to fulfill the strong existential unforgeability under chosen-message attack (SUF-CMA) security property, even in the presence of powerful quantum computers [NIS23b]. Its security is based on the hardness of finding short vectors in a lattice [NIS23b]. In particular, the problems ML-DSA relies on are the module learning with errors (MLWE) and a variant of the module short integer solution (MSIS) problem. ML-DSA is constructed following the Fiat-Shamir with aborts pattern [Lyu09].

ML-DSA operates over the polynomial ring $R_q = \mathbb{F}_q[X]/\langle X^n + 1\rangle$ where $q = 8380417$. The scheme offers three different security levels called ML-DSA-44, ML-DSA-65, ML-DSA-87, which vary in their lattice dimension and in a number of further parameters, as shown in Table 1. For a description of the algorithms refer to Algorithms 2.1 to 2.3.

Inside Algorithms 2.1 to 2.3, a number of supporting functions are used: There are several functions for encoding data from a polynomial into a byte array and vice versa, also called bit-packing functions. The routines ExpandA and ExpandMask are responsible for sampling polynomials from a seed expanded using SHAKE128 as an extended output function (XOF), while H is instantiated with SHAKE256. The functions Power2Round, Decompose, LowBits, HighBits, MakeHint, UseHint are related to the key compression for ML-DSA. More details on the subroutines used in ML-DSA are provided by the draft of FIPS 204 [NIS23b].

Table 1: Overview of ML-DSA's parameter sets [NIS23b].

Scheme (NIST level)	$\mid pk \mid$	$\mid sig \mid$	(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

Algorithm 2.1: ML-DSA: Key generation, following [NIS23b]

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

Algorithm 2.2: ML-DSA: Verification, following [NIS23b]

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

Algorithm 2.3: ML-DSA: Signing, following [NIS23b]

Input : Secret key $sk \in \mathbb{B}^{128+32((\ell+k)\cdot \mathsf{bitlen}(2\eta)+13k)}$
Input : Message $M \in \{0,1\}^*$
Output: Signature $\sigma \in \mathbb{B}^{32+\ell\cdot 32(1+\mathsf{bitlen}(\gamma_1-1))+\omega+k}$

$(\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0) \leftarrow \mathsf{skDecode}(sk)$
$\hat{\boldsymbol{s}}_1 \leftarrow \mathsf{NTT}(\boldsymbol{s}_1)$
$\hat{\boldsymbol{s}}_2 \leftarrow \mathsf{NTT}(\boldsymbol{s}_2)$
$\hat{\boldsymbol{t}}_0 \leftarrow \mathsf{NTT}(\boldsymbol{t}_0)$
$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k\times\ell} \leftarrow \mathsf{ExpandA}(\rho)$
$\mu \leftarrow \mathrm{H}(tr \| M, 2n)$
$rnd \leftarrow\$ \{0,1\}^n$
$\rho' \leftarrow \mathrm{H}(K \| rnd \| \mu, 2n)$
$\kappa \leftarrow 0$
$(\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
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

2.3 ML-KEM

Similar to ML-DSA, the module-lattice-based key-encapsulation mechanism (ML-KEM), coined in FIPS 203 [NIS23a], is based on Kyber [SAB+22]. 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 [FO99] 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. Concretely, ML-KEM uses $k = 2$ for ML-KEM-512, $k = 3$ for ML-KEM-768 and $k = 4$ for ML-KEM-1024. For more details on K-PKE, see Algorithms 2.4 to 2.6. We refer to [SAB+22] for detailed specifications of ML-KEM and underlying supporting routines in K-PKE. The polynomial ring used in ML-KEM is also $R_q = \mathbb{F}_q/\langle X^n + 1\rangle$ but with $q = 3329$. Table 2 lists other relevant parameters of ML-KEM with different security levels.

The symmetric cryptographic functions G, XOF, PRF are instantiated with SHA3-512, SHAKE128 and SHAKE256, respectively. The first draft of FIPS 203 [NIS23a] included several modifications of KYBER, including the addition of certain input-validation steps in ML-KEM. Due to an ongoing discussion to remove the input check steps, we will not include them in our ML-KEM implementations. The same approach has recently been taken in [AOB+24].

Algorithm 2.4: K-PKE.KeyGen(), following [NIS23a]

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

$z \xleftarrow{\$} \mathbb{B}^{32}$
$(\rho, \sigma) \leftarrow G(z)$
$N \leftarrow 0$
for $(i \leftarrow 0; i < k; i++)$ **do**
  for $(j \leftarrow 0; j < k; j++)$ **do**
    $\hat{\boldsymbol{A}}[i,j] \leftarrow \mathsf{SampleNTT}(\mathsf{XOF}(\rho, i, j))$
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**
  $\boldsymbol{e}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
$\hat{\boldsymbol{s}} \leftarrow \mathsf{NTT}(\boldsymbol{s})$
$\hat{\boldsymbol{e}} \leftarrow \mathsf{NTT}(\boldsymbol{e})$
$\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{A}} \circ \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}$
$\mathsf{ek}_{\mathrm{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\boldsymbol{t}}) || \rho$
$\mathsf{dk}_{\mathrm{PKE}} \leftarrow \mathsf{ByteEncode}_{12}(\hat{\boldsymbol{s}})$
return $(\mathsf{ek}_{\mathrm{PKE}}, \mathsf{dk}_{\mathrm{PKE}})$

Algorithm 2.5: K-PKE.Decrypt($\mathsf{dk}_{\mathrm{PKE}}, c$), following [NIS23a]

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

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

Table 2: Overview of ML-KEM's parameter sets and sizes (in bytes) of keys and ciphertext [NIS23a].

Scheme (NIST level)	$\lvert \mathsf{ek} \rvert$	$\lvert \mathsf{dk} \rvert$	$\lvert c \rvert$	$\lvert K \rvert$	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)

Algorithm 2.6: K-PKE.Encrypt($\mathsf{ek}_{\text{PKE}}, m, r$), following [NIS23a]

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 $c \leftarrow (c_1 || c_2)$

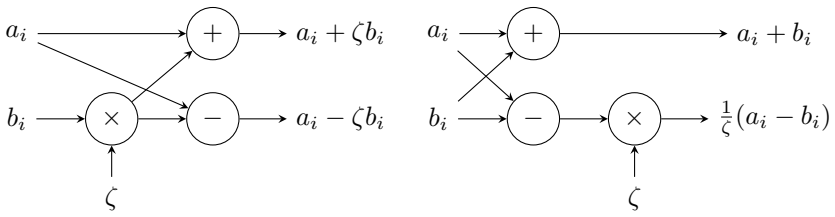
2.4 Number Theoretic Transform

The NTT is the discrete Fourier transform (DFT) on finite fields. Thanks to the divide-and-conquer pattern enabled by the Chinese remainder theorem (CRT) proposed in the work of Cooley-Tukey [CT65] and Gentleman-Sande [GS66], also referred to as fast Fourier transform (FFT), polynomial multiplication using NTT can be implemented efficiently in $\mathcal{O}(n \log n)$ instead of in $\mathcal{O}(n^2)$ using the general schoolbook method in the polynomial ring $R_q = \mathbb{F}_q[X]/\langle X^n + 1\rangle$, $q = 8380417$ in ML-DSA or $q = 3329$ in ML-KEM.

Assume that a primitive $2n$th root of unity ζ exists. Then the set of all primitive $2n$th roots of unity is $\left\{\zeta^{2i+1} | i \in [\![0, n-1]\!]\right\}$. As the $2n$th cyclotomic polynomial $\Phi_{2n}(X) = X^n + 1$ is factored into n pairwise co-prime linear polynomials $(X - \zeta^{2i+1})$ for $i \in [\![0, n-1]\!]$, 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). A polynomial $a \in R_q$ is thus transformed into its "NTT representation" $(\hat{a}_0, \ldots, \hat{a}_{n-1}) \in \mathbb{F}_q^n$. The roots of unity are called "twiddle factors". We remark that in our actual implementation of the NTT, the coefficients of the output vector are not in the normal order $\hat{a}_i = a(\zeta^{2i+1})$ for $i \in [\![0, n-1]\!]$ as described above, but they will rather be in bit-reversed order $\hat{a}_{2i} = a(\zeta^{\mathsf{br}_8(128+2i)})$ and $\hat{a}_{2i+1} = a(-\zeta^{\mathsf{br}_8(128+2i)})$ where $i \in [\![0, 127]\!]$ and $\mathsf{br}_8(x)$ is the bit reversal of a $\log n = 8$-bit integer x.

Given two polynomials $a, b \in R_q$, to compute $a \cdot b$, we first transform a, b into their NTT representation $\hat{a}$, $\hat{b} \in \mathbb{F}_q^n$, i.e., $\hat{a} = \mathsf{NTT}(a) = (\hat{a}_0, \ldots, \hat{a}_{n-1})$ $\hat{b} = \mathsf{NTT}(b) = (\hat{b}_0, \ldots, \hat{b}_{n-1})$. Then we compute the "pointwise" multiplication $\hat{a} \circ \hat{b}$ in "NTT domain": $\hat{a} \circ \hat{b} = (\hat{a}_0\hat{b}_0, \ldots, \hat{a}_{n-1}\hat{b}_{n-1})$. Finally, the result is transformed back to R_q by applying INTT, i.e., $a \cdot b = \mathsf{INTT}(\hat{a} \circ \hat{b}) = \mathsf{INTT}(\mathsf{NTT}(a) \circ \mathsf{NTT}(b))$.

While it is not a necessity [Sei18, Section 2.1], most commonly, the Cooley–Tukey (CT) algorithm is used for the forward NTT and the Gentleman–Sande (GS) algorithm for the INTT. A visualization of the "butterfly" operations used in the CT and GS algorithms is shown in Figure 1.



(a) Cooley–Tukey butterfly. (b) Gentleman–Sande butterfly.

Figure 1: NTT butterfly operations.

NTT in ML-DSA. In the case of ML-DSA, $2n \mid (q-1)$ holds and a $2n$th root of unity $\zeta = 1753$ exists. Therefore, we can compute $\log n = 8$ layers of NTT which amounts to a full splitting of the ring. This is also sometimes referred to as a "complete" NTT. The polynomial multiplication can be computed as mentioned above.

NTT in ML-KEM. As $n \mid (q-1)$ and $(2n) \nmid (q-1)$, R_q does not have a $2n$th but only an nth root of unity for ML-KEM. Let $\zeta = 17$ be the first primitive nth root of unity of R_q. Then the set of all primitive nth roots of unity of R_q is $\left\{\zeta^{2i+1} | i \in [\![0, \frac{n}{2} - 1]\!]\right\}$. As $\Phi_{2n}(X) = X^n + 1$ is factored into $n/2$ pairwise co-prime irreducible quadratic polynomials of the form $(X^2 - \zeta^{2i+1})$ for $i \in [\![0, \frac{n}{2} - 1]\!]$, we have the ring isomorphism $R_q \cong \prod_{i=0}^{\frac{n}{2}-1} \mathbb{F}_q[X]/\langle X^2 - \zeta^{2i+1}\rangle$. Thus, the NTT representation of a in bit-reversed order is $\hat{a} = (\hat{a}_0 + \hat{a}_1 X, \ldots, \hat{a}_{n-2} + \hat{a}_{n-1} X)$ where $\hat{a}_{2i} + \hat{a}_{2i+1} X = a\left(\zeta^{2\mathsf{br}_7(i)+1}\right)$, which now consists of $n/2$ linear polynomials over $\mathbb{F}_q$. In this case, the NTT is called "incomplete", and we compute only seven instead of $\log(n) = 8$ layers. As a result, the multiplication of two polynomials $a, b \in R_q$ is similar to that in ML-DSA, except for the pointwise multiplication in the NTT representations $\hat{a}$ and $\hat{b}$. Specifically, for $i \in [\![0, \frac{n}{2} - 1]\!]$,

$$(\hat{a}_{2i} + \hat{a}_{2i+1} X)\,(\hat{b}_{2i} + \hat{b}_{2i+1} X) = (\hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}) + (\hat{a}_{2i}\hat{b}_{2i+1} + \hat{a}_{2i+1}\hat{b}_{2i}) X.$$

This special multiplication is sometimes referred to as "pair-pointwise" multiplication. In the remainder of this paper, we will use the term *pair-pointwise* for multiplication in the NTT domain in ML-KEM and the term *pointwise* in the context of ML-DSA.

2.5 Modular Multiplications

A popular choice for modular arithmetic in cryptographic schemes is the Montgomery multiplication [Mon85], which has been optimized and extended for signed inputs of larger range in [Sei18]. This signed version is officially used in the reference implementations of Kyber and Dilithium [Dil23, Kyb23]. In the scope of our work, we only use the original unsigned Montgomery multiplication (cf. Algorithm 2.7).

Algorithm 2.7: Montgomery multiplication [Mon85].

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

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

In 2021, Plantard [Pla21] introduced a new modular multiplication inherited from that of Montgomery with a similarly dedicated "Plantard representation". It accepts as input non-negative integers $[\![0, q[\![$ and outputs also integers in the same range as there is a correction step at the end of the algorithm where the modulus q is conditionally subtracted from the result if they are equaled. Algorithm 2.8 shows the Plantard multiplication without the final correction, resulting in the output range $[\![0, q]\!]$. The efficiency of the Plantard multiplication over the Montgomery one lies in the fact that the former can use one multiplication less than the latter in case the multiplication $b \cdot R$ in Algorithm 2.8 is pre-computed. In exchange, this pre-multiplication doubles the size of the second input to $2d$-bit instead of d-bit as in Algorithm 2.7. The work of Huang et al. [HZZ+22] extends the input range of the Plantard multiplication to $[\![-q2^\alpha, q2^\alpha]\!]$ by adding a rounding constant α while shifting the output range to $[\![-(q-1)/2, (q-1)/2]\!]$. The larger input range for even signed integers together with the centralized representation of the output have proven to be very effective in some optimized implementations of NTT and INTT where layer merging and lazy reduction are applied.

Having considered all three options for our baseline implementations of ML-KEM and ML-DSA on OTBN where multiplications are completely implemented in software, we choose the original Plantard multiplication without the final correction step (cf. Algorithm 2.8). 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. Furthermore, due to the specific register and instruction sets of OTBN, which are explained in detail in Section 3, whether the output equals to q does not affect the implementation and lazy reduction is not applicable rendering the improved Plantard multiplication useless in this case. As for the implementations of ML-KEM and ML-DSA with our proposed instructions, Montgomery multiplication is chosen for the hardware multipliers because it accepts inputs of the same size, which fits the context of the vectorized multiplication instruction perfectly. This is further discussed in Section 5.

While ML-KEM uses only Algorithm 2.8 as the modular multiplication for its baseline implementation, ML-DSA also employs another efficient reduction called reduce32 (cf. Algorithm 2.9) for single-word reduction.

Algorithm 2.8: Plantard multiplication [Pla21]

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

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

Algorithm 2.9: Specialized reduction for ML-DSA [Dil23] reduce32

Input : $0 \leq a \leq 2^{31} - 2^{22} - 1$,
$q = 8380417$
Output : $r = a \mod q$,
$a \in [\![-6283009, 6283007]\!]$

$t = \left\lfloor \frac{a+2^{22}}{2^{23}} \right\rfloor$
return $r = a - tq$

2.6 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 [Ope23a]. It consists of several hardware intellectual property (IP) blocks, together with a main 32-bit Ibex RISC-V core and a big-number co-processor, called OTBN, accelerating asymmetric cryptography, such as RSA [RSA78] and elliptic curve cryptography [Mil86, Kob87], making up the Earl Grey microcontroller [Ope23b]. The majority of IP blocks are dedicated to cryptographic operations, including Keccak message authentication code (KMAC) supporting SHA-3 and (c)SHAKE [SHA15], an HMAC block supporting SHA2, AES [AES01] for encryption/decryption used in OpenTitan protocols, and a cryptographically secure random number generator (CSRNG) together with an entropy source IP block enabling the generation of (non)deterministic or true random numbers compliant to, e.g., NIST standards.

2.6.1 OTBN

The OTBN [Ope23a] co-processor is designed to securely accelerate classical asymmetric cryptography such as RSA and elliptic curve cryptography. Specifically, conditional jump or branch instructions always cause a stall until the branch condition is resolved. This eliminates the possibility of any kind of Spectre BHB [KHF+19] vulnerabilities [Ope23a, Section 8.2.2]. In addition, loads from and stores to data memory are not cached, which prevents cache-timing attacks [Ope23a, Section 8.2.2]. Memory scrambling and register blanking are deployed to further counteract side-channel leakage [Ope23a, Section 7.2.2] Finally, a checksum for instructions and data memory accesses as well as an instruction counter to detect skips from the Ibex core are deployed against fault injection [Ope23a, Section 8.2.2].

Besides its enhanced security, the appealing feature of the OTBN co-processor is its instruction set architecture (ISA). Part of the ISA is 32-bit RISC-V-based, offering 32 general-purpose registers (GPRs) `x0` to `x31` used by the "base instruction subset" for the control flow of an OTBN application. The other part is a custom big-number ("bn") instruction set 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 containing all zeros 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 (the carry C, most significant bit (MSB) M, least significant bit (LSB) L, zero flag Z), key material (accessed via the key manager), a special "modulus register" `MOD` and an "accumulate register" `ACC`, used in some of the big-number instructions [Ope23a, Section 8.2].

We want to note that the big-number instruction set is geared towards arithmetic on (unsigned) 256-bit numbers and really not for performing (signed) arithmetic on small 32-bit or 16-bit integers. For example, widening multiplications require sign-extension of the inputs up to a length that is as long as the number of correct bits desired in the low part of the output. Due to the lack of an instruction for sign extension, this is a rather intricate operation. Further, when performing addition and subtraction on signed 16- or 32-bit integers, the top bits will not be cut off (as they normally would), therefore we need to handle these remaining bits manually. Finally, the `bn.addm` and `bn.subm` instructions only perform the correcting subtraction or addition either for results $>$ `MOD` or respectively < 0, not both. This also shows that signed modular arithmetic is not a target functionality.

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 the OTBN in this work is written in assembly.

2.6.2 Python Simulator

OTBN comes with a cycle-accurate Python simulator for software development purposes. The simulator takes `.elf` files that have been built for OTBN as input, executes the code, and dumps the register contents. It offers detailed statistical data on the execution, such as cycle counts and stalls, as well as an instruction histogram and numbers of function calls. We make extensive use of this simulator for testing our proposals of the extended OTBN big-number instructions, testing implementations of ML-KEM/ML-DSA and obtaining benchmark results. This serves as a pre-test for the real hardware modifications of the OTBN core later on.

2.6.3 KMAC Block

The KMAC core can be used to compute Keccak-based message authentication codes (MACs), as well as unauthenticated SHA-3, including its XOF operation mode called SHAKE [Ope23a, Section 9.13]. The latter is especially of interest for the implementation of PQC schemes such as ML-KEM or ML-DSA. The hardware design offers a compile-time choice between a version with first-order masking enabled and a version without masking. The technique applied is called domain-oriented masking (DOM) and increases the area required by the logic design by a factor of greater than two [Ope23a, Section 9.13.1]. Computing the masked permutation Keccak-f for a state of $b = 1600$ bits takes four cycles per round for a number of $(12 + 2 \log b) = 24$ rounds, resulting in 96 cycles in total [Ope23a, Section 9.13]. For the unmasked implementation, one round of the Keccak-f permutation takes just one cycle. However, in this work, we will assume the version with first-order masking enabled, as we expect it to be more popular in practice and in order to offer a more conservative performance estimation. The KMAC core obtains the randomness needed for masking directly from the OpenTitan entropy distribution network. The core is accessible to the Ibex core via the system bus and to other OpenTitan peripherals via three application interfaces.

3 Implementation on plain OTBN

This section describes an implementation of ML-DSA and ML-KEM on an essentially unmodified OTBN. The only change we require is an increased data-memory size. This implementation will serve as a performance baseline and as a starting point for profiling. Our description focuses on optimization techniques of the main computation blocks in ML-DSA and ML-KEM, i.e., modular arithmetic, NTT and INTT, multiplication in NTT domain, sampling, and bit packing. We also provide a pure software implementation of Keccak-f for OTBN inspired by tiny_sha3 by Saarinen[2].

3.1 Modular Multiplication and Reduction

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 of size d-bit each and polynomial arithmetic breaks down to modular arithmetic on d-bit unsigned integers.

Modular multiplication. In order to multiply two elements a and b in $\mathbb{F}_q$ using Plantard multiplication [Pla21], we prepare a WDR with all relevant constants and perform

[2] https://github.com/mjosaarinen/tiny_sha3/tree/master

the multiplication using the 64×64-bit multiplier of OTBN. In particular, we compute Algorithm 2.8 as follows:

1. Load constants into a WDR, e.g., `consts` $= (m||q||1||R)$, where $m = 2^d - 1$ is a mask and $R = q^{-1} \mod 2^{2d}$. Next, load the elements a and b into two separate WDRs, e.g., `coeffa` and `coeffb`.

2. Multiply a and b as two 64-bit integers: `bn.mulqacc.wo.z wtmp, coeffa.0, coeffb.0, 0`. The register `wtmp` now has $ab \mod 2^{2d}$ at its first quad word because we do not shift it to the left, i.e., the constant 0 at the rightmost side.

3. Compute $(ab \mod 2^{2d})R$, and keep the result modulo 2^{2d}: `bn.mulqacc.wo.z wtmp , wtmp.0, consts.0, 192`. The constant 192 is the amount of bits to shift the result to the 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 picked as a multiple of 64 `wtmp` must be masked with m to extract the correct result: `bn.and wtmp, wtmp, consts`.

4. Shift to the right by t bits and add the result with 1: `bn.add wtmp, consts, wtmp >> t`, where $t = 144$ for ML-KEM and $t = 160$ for ML-DSA. This means the result is in the second quad word of `wtmp`.

5. Multiply the result with q: `bn.mulqacc.wo.z wtmp, wtmp.1, consts.2, 0`.

6. Shift to the right by d bits: `bn.rshi wtmp, bn0, wtmp >> d`.

As we can see, for a typical modular multiplication, we need five instructions in ML-DSA and six instructions in ML-KEM. In case the second factor b is pre-multiplied by R, which normally happens in NTT, we need one instruction less for both schemes (cf. Listing 1, Line 6 to 9).

Reduction. 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.

3.2 NTT

This section describes the implementation of the NTT and INTT on OTBN. We make use of common optimizations such as merging the multiplication with n^{-1} into the last twiddle factor in the INTT [LN16, Sec. 3], making up for the omitted transformation into Plantard representation during the multiplication with n^{-1} [LS19, Sec. 5.3] as well as transforming the twiddle factors into the proper domain for the deployed modular multiplication strategy ahead of time [ADPS16, Sec. 7.2].

CT and GS butterfly. We follow the original approach from [Dil23, Kyb23] to use the CT butterfly for the NTT and the GS butterfly for the INTT. A CT or GS butterfly consists of a Plantard multiplication (cf. Section 3.1) between a coefficient and a twiddle factor, preceded or followed, respectively, by 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 2.8), the outputs of each layer are certain to be in $[\![0, q]\!]$ as well, inhibiting a 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 2.8). Subsequently, a CT butterfly takes six and seven cycles for ML-DSA and ML-KEM, respectively. Listing 1 shows how to extract data for a CT butterfly in ML-DSA and store the results back to the buffer registers.

```
/* 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 1: CT butterfly on OTBN.

Layer merge. A popular optimization technique for NTT and INTT is "layer merging". For a t-layer NTT (or INTT), a t_0-layer merge, where $t_0 \leq t$, means loading and storing 2^{t_0} coefficients from/to memory only once after processing t_0 layers of NTT on these coefficients, instead of loading and storing them t_0 times as in the traditional layer-by-layer approach. The number of layers to be merged t_0 is mostly limited by the number and size of registers available on the processor. A t_0–t_1 layer merge, where $t_0 + t_1 = t$, means that the first t_0 and the last t_1 layers are merged separately, causing each coefficient to be loaded and stored twice throughout one of the transformations. In this work, we 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. Particularly, $13(n/d)$ 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, one for holding intermediate values, and another one for holding twiddle factors (cf. Listing 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.

3.3 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 multiplication blocks as described in Section 3.1. Recall that for ML-KEM, a pair-pointwise multiplication of two polynomials in NTT domain consists of multiplications of 128 pairs of linear polynomials, each of which requires five Plantard multiplications resulting in $5 \times 128 = 640$ Plantard multiplications for a complete pair-pointwise product. Particularly, two consecutive linear polynomials in an NTT representation use some twiddle factor and its negation respectively. In order to save WDRs, we reuse this factor by negating the product $\hat{a}_{2i+1}\hat{b}_{2i+1}$ of the second linear polynomial and multiplying it with the same twiddle factor, instead of negating the twiddle factor itself. The negation $-\hat{a}_{2i+1}\hat{b}_{2i+1}$ is done by subtracting from zero using `bn.subm` so that the result is put back in the positive range $[\![0, q]\!]$ for later operations. The following additions of the form $\hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}$ and $\hat{a}_{2i}\hat{b}_{2i+1} + \hat{a}_{2i+1}\hat{b}_{2i}$ are computed with the modular addition instruction `bn.addm` of OTBN. As the output of every Plantard multiplication is in $[\![0, q]\!]$, the result of additions using `bn.addm` is ensured to be in $[\![0, q]\!]$ as well.

Pseudo vectorization. Let $a = (a_0, \ldots, a_n)$ and $b = (b_0, \ldots, b_n)$ be two input vectors for the pointwise addition, where $a_i, b_i \in [\![0, q]\!]$ for $i \in [\![0, n]\!]$ and $\log q < d$. Recall that a WDR can store n/d coefficients: let `w0` $= (a_0, \ldots, a_{n/d})$ and `w1` $= (b_0, \ldots, b_{n/d})$. Usually, we would proceed to shift a_i and b_i into separate WDRs, adding them using `bn.addm` before shifting the result $(a_i + b_i) \bmod q$ back to one of the source WDRs and shifting out the used coefficient from the other source WDR as well. This process would be repeated for every pair of coefficients a_i and b_i individually.

The idea of "pseudo vectorization" is instead of processing coefficients one by one as explained above, we can add two vectors of n/d coefficients using the non-vectorized addition instruction `bn.add` and obtain the result of a vectorized one. We apply this technique during the accumulation in the matrix-vector product in ML-DSA and ML-KEM. This is possible because the addition of two polynomials will not exceed d bits during accumulation as part of the matrix-vector product for the respective parameter sets. While the outputs of the accumulated matrix-vector product must be reduced to $[\![0, q]\!]$ manually, using either Plantard reduction for ML-KEM or a version of `reduce32` for ML-DSA (cf. Section 3.1), applying this technique greatly improves the performance.

3.4 Sampling

Rejection sampling in $[\![0, q]\!]$. Listing 2 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
    beq     outp, t0, _skip_store1      /* n coefficients are sampled?*/
    bn.and  cand, coeff_mask, shake_reg /* Extract 3-byte candidate c from shake output */

    bn.cmp  cand, mod                   /* c-q */
    csrrs   a4, 0x7C0, zero             /* Read flags Z, L, M, C*/
    andi    a4, a4, 3                   /* Extract M, C*/
    bne     a4, const_3, _skip_store1   /* Reject if M!=1 & C!=1 i.e. (q <= c) */
    bn.rshi accumulator, cand, accumulator >> 32
    addi    accumulator_count, accumulator_count, 1
    bne     accumulator_count, const_8, _skip_store1/* accumulator is full of 8 coeffs? */

    bn.sid  accumulator_idx, 0(outp++)              /* Store full accumulator to memory */

    li      accumulator_count, 0
_skip_store1:
    bn.or   shake_reg, bn0, shake_reg >> 24     /* Shift out used 3 bytes */
  ret
```

Listing 2: Inner loop of uniform sampling in ML-DSA on OTBN.

3.5 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 [Dil23], 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 [Dil23] 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 [Kyb23], 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

Percent	Poly Arith.	Sample	Hash	Pack	Round	Reduce
K-3	8	12	77	1	1	1
S-3	29	8	55	1	3	3
V-3	12	10	74	1	2	1

(a) ML-DSA-65

Percent	Poly Arith.	Sample	Hash	Pack
K-768	18	12	70	1
E-768	21	11	66	2
D-768	27	10	60	3

(b) ML-KEM-768

Figure 2: Cycle count profiling on OTBN, median values over 10 000 iterations. Groups with less than 1% not displayed. Percentages may not add up to 100% due to rounding.

$d_{\{u,v\}}$ because after the left shift of $d_{\{u,v\}}$ bits (cf. Table 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.

3.6 Keccak on OTBN

As previously mentioned, our pure software implementation of Keccak-f is based on tiny_sha3. The input to the Keccak-f permutation are 1600 bytes, equivalently 25 64-bit lanes, which will go through five steps θ, ρ, π, χ and ι in this order and for exactly 24 rounds. A detailed description of Keccak-f can be found in [NIS23c].

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, careful arrangement of lanes within seven WDRs is required. While circularly rotating multiple lanes by the same amount of bits can be done concurrently using masking and shifting, individual processing is necessary when each lane requires a different rotation, as in the $\rho - \pi$ step.

By using pseudo vectorization, we succeed in speeding up the implementation by 40%, when compared to an implementation using the "standard" approach for 64-bit architectures. Specifically, the 64-bit approach on OTBN takes 286 cycles for one round; while our approach takes 171 cycles, which results in 2760 cycles saved for 24 rounds of Keccak-f. The input and output arrangement before each permutation, while appearing intricate, outperforms the 64-bit implementation by 20 cycles, requiring only 58 cycles. However, implementing interfaces for the symmetric primitives, including SHA3-{256, 512} and SHAKE{128, 256}, presents a challenge. Depending on the exact use case, it may be required to access the memory at addresses that are neither 4-, nor 32-byte-aligned, meaning we need to explicitly extract/insert the bytes into a GPR to simulate byte-aligned memory accesses, which comes with a hit in performance.

3.7 Profiling

Figure 2a and Figure 2b present a heatmap table illustrating the cycle count percentages for ML-DSA and ML-KEM. Hashing emerges as the most time-consuming operation in both schemes, a finding that aligns with the profiling results in previous works [HZZ+22, Table 6][KRSS19]. This observation prompts us to leverage the KMAC block of OTBN for potential optimization.

3.8 Reflection

The wide registers on OTBN demonstrate their efficiency by allowing the caching of large amounts of data internally, which significantly reduces the cost of memory access compared to GPRs. The `bn.subm` instruction efficiently adjusts subtraction results to positive values at no additional cost. Similarly, `bn.rshi` proves effective for shifting, requiring only two instructions per coefficient. The `bn.add` and `bn.rshi` instructions further highlight the practicality of WDRs through pseudo-vectorized addition and shifting, as long as the coefficient size after these operations remains within the capacity of a single d-bit vector element. The efficiency gain in addition surpasses an n/d-fold improvement, primarily because previously, each coefficient addition often required more than one instruction.

The architecture of OTBN, however, presents several obstacles to efficient implementation that should be emphasized. Firstly, the need to extract coefficients to a separate WDR for computation and to repack them for storage is inefficient. While this approach reduces memory accesses, it necessitates considerable effort in data movement, which is identified as a key performance bottleneck. Secondly, the pseudo-vectorization technique cannot be effectively applied to multiplication due to the multiplier's limitation to 64×64-bit products. Moreover, the absence of implicit truncation for multiplication results creates additional overhead since it has to be done explicitly. Additionally, the immediate for `bn.mulqacc` is restricted to multiples of 64, forcing explicit shifting in Step 3 of the Plantard multiplication (cf. Section 3.1). Thirdly, other central processing units (CPUs) equipped with digital signal processor (DSP) extensions often execute the Plantard multiplication with much shorter instruction sequences. For instance, the Arm Cortex-M4 requires only two instructions ([HZZ+22]). Finally, while the 32-byte data path for memory access allows loading large amounts of data quickly, the lack of flexibility when loading smaller amounts of data, i.e., individual coefficients, from unaligned addresses oftentimes incurs performance penalties.

4 Implementation on OTBN with Keccak Acceleration

In the light of the profiling results from Section 3.7, we have decided to study the hardware/software co-design approach by interfacing to the OpenTitan's existing KMAC core. We refer to OTBN with said interface as $\text{OTBN}^{\text{KMAC}}$.

KMAC interface. The KMAC core within OpenTitan is accessible via the main TL-UL bus interface, as well as through application interfaces for the key manager, life-cycle controller, and ROM controller. To facilitate OTBN's interaction with the KMAC core, we introduce an additional application interface. All application interfaces feature a 64-bit data path and employ straightforward control logic, including simple status signals such as `ready`, `valid`, and `last data`. The KMAC outputs the digest as two boolean shares on a parallel data path.

The simplicity and high throughput of the application interface make it an attractive solution for integrating KMAC with OTBN. On the KMAC side, only minor modifications are required, such as enabling dynamic configuration of the hash algorithms to support SHA3-256, SHA3-512, SHAKE128, and SHAKE256. For OTBN, special-purpose registers for KMAC configuration, message, status, and digest are added. The status register, controlled by KMAC signals, allows OTBN to determine if the KMAC core is ready for operation. The configuration register contains the hash function and the length of the data to be processed. All registers can be written and read with big-number (`bn.wsrr`, `bn.wsrw`) and general purpose (`csrrw`) instructions for accessing special purpose registers.

Data is sent to the KMAC core by writing to the 256-bit message register, which is connected to a small FIFO that outputs 64-bit words to the KMAC application interface. If

Percent	Poly Arith.	Sample	Hash	Pack	Round	Reduce	Other
K-3	41	42	5	6	3	3	0
S-3	67	13	2	3	7	8	0
V-3	51	26	4	3	10	4	1

(a) ML-DSA-65

Percent	Poly Arith.	Sample	Hash	Pack
K-768	63	31	3	2
E-768	66	26	3	5
D-768	70	20	2	8

(b) ML-KEM-768

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

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.

Once the KMAC core completes its operation, the corresponding bit in the status register is set, and the two 256-bit digest registers contain the unmasked digest. In the future, this interface could be configured to also support masked digests.

For all modifications to OTBN and KMAC - and in general for all hardware modifications within this paper - the integration of countermeasures against fault injection and side-channel attacks was taken into account. This includes the integrity protection of registers, blanking and wiping of sensitive values, sparse encodings of control signals and redundancy checks throughout the OTBN pipeline.

Python simulator of KMAC interface. An interface to the Python simulator was first implemented by Philipoom[3]. The simulator matches the behavior of the actual hardware using hash functions from PyCryptodome[4] and integrating the special purpose registers for configuration, status, message and digest introduced above. An abstract implementation of the KMAC application interface including the FIFO within the OTBN models the fact that the OTBN can write data faster to the KMAC core than the KMAC core can process the data and accounts for potential stalls.

4.1 Profiling

In this section, we consider the profiling results under the assumption that the OTBN has an interface to the KMAC core.

As it becomes clear from Figures 3a and 3b, the time spent on hashing is drastically reduced thanks to the powerful KMAC core. For ML-DSA, we can see that the majority of the time is now spent on polynomial arithmetic, followed by the sampling. The picture is similar for ML-KEM, where the polynomial arithmetic accounts for an even larger portion of the runtime, also followed by the sampling.

This result leads us to the conclusion that a second major reduction in runtime could be achieved by accelerating the polynomial arithmetic on OTBN$^{\text{KMAC}}$.

[3] https://github.com/jadephilipoom/opentitan/commit/e86be3446204f439c41c142b077a4ca8b449b1c9

[4] https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html

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 start by describing our overall goal for the extensions, before detailing the new instructions and explaining the reasoning behind them. We will refer to our implementations of ML-KEM and ML-DSA with the proposed instructions and KMAC block enabled on OTBN as $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$.

5.1 Goal of the ISA Extensions

The main goal of the ISA extensions is to accelerate the computation of lattice-based cryptography on OTBN, with the focus on ML-KEM and ML-DSA. However, we aim to offer instructions that are general enough to be useful for other cryptographic schemes, also beyond lattice-based cryptography. More specifically, we aim to reduce the time spent on polynomial arithmetic as we have identified it to be the main bottleneck (cf. Section 3), next to polynomial generation, which we already addressed in the previous section. The underlying hash functions have other use cases than asymmetric cryptography and require parallel processing of large bit vectors. Dedicated co-processors are therefore more suitable than instruction set extensions, as long as transfer latency does not become an issue [KSS24].

With OTBN being a reduced instruction set computer (RISC)-based architecture and the 32-bit instruction encoding making opcodes scarce, an additional goal is to keep the number of new instructions to a minimum.

The primary metric for evaluating the extensions is the performance in terms of the cycle count – while we also report 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.

5.2 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 3.8 how much time is spent on extracting individual coefficients from the WDRs and also how much performance gain could be achieved through the application of the pseudo-vectorization strategy as introduced in Section 3.3. 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 by making use of SIMD instructions, for example using Intel AVX2 [LDK+22] or Arm Neon [BHK+22].

We propose a total of five new instructions with multiple subvariants each. Our first three proposals immediately follow from the reasoning above: `bn.addv`, `bn.subv`, `bn.mulv` . These instructions offer SIMD (modular) addition, subtraction, and multiplication respectively. Note that although our proposal for `bn.mulv` is highly similar to the one presented in [Saa23], the approach was developed independently.

The fourth instruction we propose serves the purpose of interleaving data inside two WDRs when interpreting them as vectors of multiple elements. While such an instruction is a staple in SIMD instruction sets, in our scenario, it is particularly useful for the NTT and INTT. More details on this can be found in Section 5.3.

Lastly, we propose an instruction for bit shifting, which is useful across various functions throughout ML-KEM and ML-DSA and a basic operation present in the majority of (SIMD) instruction sets. For example, in ML-DSA the SIMD bit shifting instruction allows us

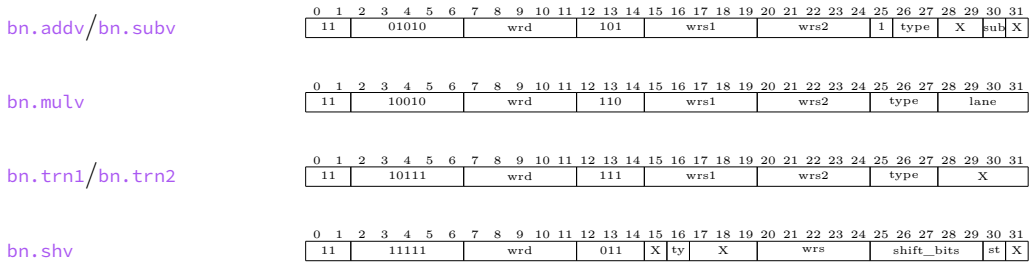


Figure 4: Instruction encoding for the proposed extensions.

to fully vectorize the decomposition. Furthermore, the sampling of coefficients in $[\![-\eta, \eta]\!]$ benefits from this operation.

A more detailed description of the instructions can be found below. In the description of the instruction `<type>` defines the subvariants of the instruction, including the operation on vector elements of different sizes, e.g., `.8S` for a 32-bit element view 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 $[\![0, \frac{n}{d} - 1]\!]$. Next, the result is either truncated or reduced mod$^+$ `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.

The encoding of the instructions is shown in Figure 4. The names of the fields are chosen in accordance with the naming of the operands of the previously introduced instructions. “ty” is short for `<type>`, and “st” for `<shift_type>`.

5.3 Impact on the Implementation of ML-KEM and ML-DSA

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

5.3.1 Polynomial Addition & Subtraction

The biggest 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. Next to the reduction of the cost for the arithmetic, a similarly impactful saving is incurred due to the reduction of the data movement overhead. Listings 3 and 4 show how the implementation of polynomial addition and subtraction changes under the application of `bn.addvm` and `bn.subvm`.

```
1 loopi 32, 4
2   bn.lid vec_1_idx, 0(src1++)
3   bn.lid vec_2_idx, 0(src2++)
4
5   bn.addvm.8S vec_1, vec_1, vec_2
6
7   bn.sid vec_1_idx, 0(dst++)
```

Listing 3: Polynomial addition on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$

```
1 loopi 32, 4
2   bn.lid vec_1_idx, 0(src1++)
3   bn.lid vec_2_idx, 0(src2++)
4
5   bn.subvm.8S vec_1, vec_1, vec_2
6
7   bn.sid vec_1_idx, 0(dst++)
```

Listing 4: Polynomial subtraction on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$

5.3.2 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. Further, the amount of memory operations can be reduced compared to the implementation detailed in Section 3 as we do not require the extraction of individual coefficients from the WDRs into separate WDRs to perform the computation on.

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. See Listing 6 for an excerpt of the implementation. The implementation of the GS-butterfly in the INTT is highly 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, e.g., in [BHK+22]. The first five layers of NTT can be computed straightforwardly, as the stride between the coefficients inside individual WDRs is sufficient. However, starting from layer 6, the elements that the butterfly would be computed on are located inside the same WDR. Therefore, we need to permute the data before we can continue with the computation. Using a sequence of multiple `bn.trn1` and `bn.trn2` instructions, we follow the strategy from [BHK+22] and "transpose" the 8×8 matrix of elements, made up of considering eight WDRs with eight coefficients each. See Figure 5 for a visualization.

ML-KEM. For ML-KEM, the butterfly simplifies just as for ML-DSA, shown in Listing 6, except that the `.16H` variants of the instructions are used. Regarding the layer merge, due to the availability of 32 256-bit WDRs, we are able to merge all seven layers of the ML-KEM NTT. This reduces 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.

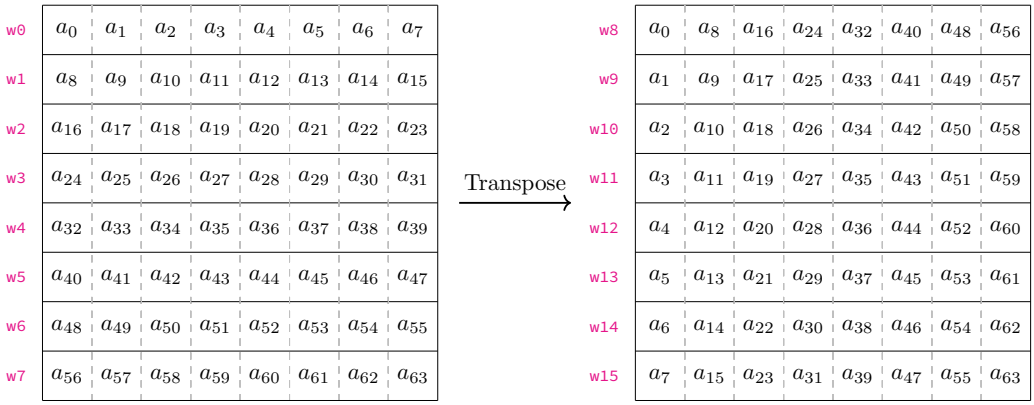


Figure 5: Visualization of the transposition.

5.3.3 Multiplication in NTT Domain

Especially thanks to the availability of the `bn.mulv(m)` instruction, the base multiplication for ML-DSA and ML-KEM 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 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 $\hat{a}_{2i} = \hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}\zeta^{2\mathsf{br}_7(i)+1}$ 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 5 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.

```
/* a1b1, a0b0 */
bn.mulvm.16H wtmp0, coeffsa, coeffsb
bn.mulvm.16H wtmp1, coeffsc, coeffsd

/* a0b1, a1b0 */
bn.rshi      wtmp, bn0, coeffsb >> 16
bn.trn1.16H  coeffsb, wtmp, coeffsb
bn.mulvm.16H coeffsb, coeffsa, coeffsb

bn.rshi      wtmp, bn0, coeffsd >> 16
bn.trn1.16H  coeffsd, wtmp, coeffsd
bn.mulvm.16H coeffsd, coeffsc, coeffsd

/* Multiply with Twiddle factors */
bn.trn2.16H  wtmp, wtmp0, wtmp1
bn.mulvm.16H wtmp, wtmp, twiddles
bn.trn1.16H  wtmp0, wtmp0, wtmp
bn.rshi      wtmp, bn0, wtmp >> 16
bn.trn1.16H  wtmp1, wtmp1, wtmp

/* a1b1+a0b0; a1b0+a0b1 */
bn.trn1.16H  coeffsa, wtmp0, coeffsb
bn.trn2.16H  coeffsb, wtmp0, coeffsb
bn.addvm.16H res0, coeffsa, coeffsb

bn.trn1.16H  coeffsc, wtmp1, coeffsd
bn.trn2.16H  coeffsd, wtmp1, coeffsd
bn.addvm.16H res1, coeffsc, coeffsd
```

Listing 5: ML-KEM pair-pointwise multiplication on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$.

```
bn.mulvm.l.8S tmp, vec8, tf1, 0
bn.subvm.8S   vec8, vec0, tmp
bn.addvm.8S   vec0, vec0, tmp
```

Listing 6: ML-DSA CT-butterfly on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$.

5.3.4 Sampling

Rejection sampling. Although it is possible to vectorize the rejection sampling routines in ML-DSA and ML-KEM as introduced in [GS16] and applied in [Dil23, Kyb23], 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. See Section 8 for a further discussion of this topic.

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 $\eta = 2$.

5.3.5 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,

Percent	Poly Arith.	Sample	Hash	Pack	Round	Reduce	Other
K-3	19	66	8	6	1	0	0
S-3	52	27	5	6	7	2	1
V-3	27	48	8	5	11	0	1

(a) ML-DSA-65

Percent	Poly Arith.	Sample	Hash	Pack	Other
K-768	19	68	8	5	1
E-768	21	59	7	13	0
D-768	24	50	6	19	1

(b) ML-KEM-768

Figure 6: Cycle count profiling on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, median values over 10 000 iterations. Groups with less than 1% not displayed. Percentages may not add up to 100% due to rounding.

however, since we decided to operate mod^{+}, we need to transform the coefficients into their centralized representatives $\text{mod}^{\pm}$ 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.

5.4 Profiling

The profiling data for our implementations on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$, as shown in Figures 6a and 6b, already indicates that our extensions to the ISA help with achieving our initial goal of reducing the cycle count for the polynomial arithmetic. We reduce the part of the runtime spent on polynomial arithmetic by up to 46 percentage points.

6 Hardware implementation

This section describes our hardware implementations and the modifications we applied to the OTBN architecture and its components.

6.1 Basic Building Blocks

To allow the execution of vectorized 32-bit and 16-bit operations while keeping the resource usage low, we design basic building blocks which are capable of both. In addition, for vectorized addition and subtraction, utilizing existing resources for 256-bit additions and subtractions is possible. Similarly, the vectorized modular multiplication of polynomial coeffcients is designed such that it merges with the existing 64-bit multiply and accumulate unit. The following sections describe the working principle behind our configurable addition and multiplication module.

6.1.1 Configurable Vectorized Adder

The big number arithmetic logic unit (BN-ALU) already contains 256-bit wide adders. We adapt these adders to enable the OTBN to execute 32-bit and 16-bit vectorized (modular) additions and subtractions besides the 256-bit (modular) addition. In the following,

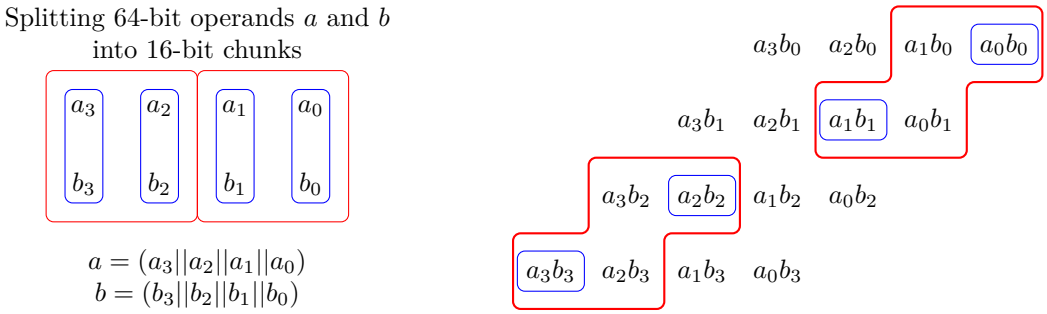


Figure 7: Configurable vectorized multiplication. Considering only certain partial products, enables the execution of vectorized 32-bit and 16-bit multiplications besides one 64-bit multiplication. For 64-bit multiplication all partial products are considered, for 32-bit multiplications only the partial products highlighted in red are considered and for 16-bit multiplications only the blue partial products are considered.

we describe our adder module, which will replace the 256-bit adders within the OTBN BN-ALU and provide the means for the `bn.addv` and `bn.subv` instructions.

The idea is to split one 256-bit addition into 16 × 16-bit additions and add multiplexers to the carry input of each adder. Depending on the carry propagation, adders of different size can be formed. Therefore, we add multiplexers for the carry propagation between these 16 adders. By changing the connections of the carry inputs and outputs, this adder is capable of executing the original 256-bit addition, as well as vectorized 32-bit and vectorized 16-bit additions. For the original 256-bit addition, as used for example in `bn.add`, the carry input of each 16-bit adder is connected to the output carry of the previous adder. For a vectorized 16-bit addition, the input carry of each 16-bit adders is set to the input carry c_0. For a vectorized 32-bit addition, two subsequent 16-bit adders are connected to form a 32-bit adder by connecting the output carry of the first adder to the input carry of the second adder. Note that extending this approach for vectorized 64-bit and 128-bit additions is straightforward.

6.1.2 Configurable Vectorized Multiplier

The second basic building block is a configurable vectorized multiplier. In the following, we explain the principle behind our 64-bit operand and 128-bit result multiplier, but our approach is applicable to arbitrary-width multiplications. Our multiplier is able to compute either one 64-bit multiplication, two 32-bit multiplications or four 16-bit multiplications. We achieve this by splitting one 64-bit × 64-bit product into several partial products, where each partial product 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 setting irrelevant ones to zero, our multiplier is able to compute vectorized 32-bit or 16-bit multiplications. This is illustrated in Figure 7. In this example, each 64-bit operand is split into 16-bit chunks. Consequently, $a = (a_3||a_2||a_1||a_0)$ and $b = (b_3||b_2||b_1||b_0)$. For a 64-bit multiplication, each 16-bit chunk of a must be multiplied with each 16-bit chunk of b, and the resulting partial products must be added up to obtain the final 128-bit result c. For a vectorized 32-bit multiplication two subsequent 16-bit chunks build the respective 32-bit operand, i.e. $a = a'_1||a'_0$, where $a'_1 = (a_3||a_2)$ and $a'_0 = (a_1||a_0)$. Then, the final result is calculated by $c = (a'_1 \times b'_1||a'_0 \times b'_0)$. Consequently, partial products which are obtained by $a'_1 \times b'_0$ or $a'_0 \times b'_1$ are not considered. This approach is illustrated by the red frame in Figure 7. Similarly, for a vectorized 16-bit multiplication, the final result is obtained by $c = (a_3 \times b_3||a_2 \times b_2||a_1 \times b_1||a_0 \times b_0)$. Therefore, only the

partial products within the blue frames are considered, and all others are ignored.

6.2 Integration into OTBN Architecture

This section describes the modification needed to integrate our newly proposed modules. As mentioned above, we want to share as many resources as possible while keeping the architecture as simple as possible. This includes ensuring meaningful decoding and control architectures. As described above, our vectorized addition unit for `bn.addv` and `bn.subv` instruction was designed as a drop-in replacement for the existing adder using the same resources for vectorized and big-number additions and subtractions. Similiarly, as the BN-ALU already contains funtional units for the shifting, we integrate the functionality for our proposed `bn.shv` and `bn.trn1`/`bn.trn2` instruction into the BN-ALU.

The `bn.mulv` extension offers a larger reasonable design space to consider. For this work, we examined two trade-offs. First, we try to reuse as much of the existing resources available in the big number multiply accumulate unit (BN-MAC) unit as possible at the cost of latency. Second, we integrate a completely new module into the OTBN pipeline at the cost of a significantly increased resource consumption but a higher performance improvement. Accordingly, we refer to the implementations of ML-KEM and ML-DSA using the modified BN-MAC as $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$; while $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ refers to the latter high-end approach.

For the first trade-off, we found that the BN-MAC already provides resources which can be reused for vectorized arithmetic. Specifically, the BN-MAC contains a 64-bit multiplier and a 256-bit adder. Replacing both with our proposed basic building blocks, described in Section 6.1.1 and Section 6.1.2, enables vectorized multiplications and additions while still supporting the original `bn.mulqacc` operations. To keep the resource overhead as low as possible, we split the `bn.mulv` instructions into several cycles and reuse the already existing computational resources. This requires additional control logic within the BN-MAC, as well as in the decoder and controller to stall the pipeline and keep all redundancy checks throughout the pipeline in sync. Even with this approach, the control path is kept relatively simple and changes integrate well into the simple architecture of OTBN. It should be noted that according to the OTBN design rationale, all instructions should complete within a single cycle. However, mechanisms exist that stall the pipeline for loads or if the internal randomness register does not contain fresh randomness. In fact, the KMAC interface also needs to potentially stall the pipeline (see Section 4). One could consider adding an instruction for every execution stage of the `bn.mulv` instruction (see Section 6.2.2 for the execution stages). Since this is a code complexity versus hardware complexity trade-off, we do not further explore this here. The modifications to the BN-MAC and details on the multi-cycle approach are described in Section 6.2.2.

For the second trade-off, we found that it is appropriate to outsource this operation into a new separate module that is capable of executing `bn.mulv` in a single cycle. This approach provides a clean and straightforward integration, especially when considering the decoding and control logic. The alternatives, namely integrate the one-cycle `bn.mulv` either into the BN-MAC or into the BN-ALU have the following disadvantages: Both approaches would still add a significant resource overhead. This is mainly due to the fact the `bn.mulv` requires 4×64-bit multipliers. While the BN-ALU provides the means for the additions and subtractions within `bn.mulv`, this requires to add 4×64-bit multipliers to the architecture. This would also increase the internal complexity of the BN-ALU and the control and decode logic drastically. The BN-MAC already contains one 64-bit multiplier and one 256-bit adder. Therefore, extending the BN-MAC for our proposed `bn.mulv` instruction would require adding three 64-bit multipliers and a subtractor. Similiarly, as for the BN-ALU, this would also increase the internal complexity of the BN-MAC, decoder and controller drastically. As the hardware resources saved are disproportionate to the increased complexity of the control logic, we are pursuing an independent approach.

Adding a standalone big number vector multiplier (BN-MULV) module for the bn.mulv instruction keeps the control path simple and ensures a clean integration. This comes at the cost of a significant increase in hardware utilization. The BN-MULV module and its integration is described in Section 6.2.3.

In summary, when integrating the bn.mulv instruction, one must choose between a compact hardware implementation with a more complex control path and a simple control path with single cycle vector multiplication and a significant increase in hardware resources. The exact hardware costs are explored later on in this chapter.

6.2.1 Modified Big Number ALU

As descibed above, we extend the BN-ALU for bn.addv, bn.subv, bn.shv and bn.trn1 /bn.trn2 instructions.

To integrate support for the bn.addv and bn.subv instructions, we replace the adders within the BN-ALU with our configurable vectorized adders. To be more specific, we replaced the original 256-bit adder within Adder X and Adder Y with 16 16-bit adders, respectively. This enables the OTBN to execute the original 256-bit wide operations as required by bn.add/bn.addm and bn.sub/bn.subm instructions, as well as 16-bit and 32-bit vector operations for the bn.addv and bn.subv while keeping the resource overhead at a minimum.

For the simple bn.addv and bn.subv instructions, only Adder X is used to compute $x = a+b$ and $y = a-b$, respectively. For the bn.addvm instruction, Adder X is responsible to execute $x = a+b$ while Adder Y calculates the pseudo-modulo reduction $y = x \mod q$. Depending on, whether $x < q$ or $x \geq q$, either the results of Adder X or Adder Y are selected as outputs. This check is achieved by evaluating the carry propagation of Adder X and Adder Y . More specifically, if the carry bits c_o^i for $i \in [\![0, 15]\!]$ of Adder X or Adder Y are set, then the output of Adder Y is selected as output, otherwise the results of Adder X are taken as result.

Similiarly, for the bn.subvm instruction, Adder X is responsible to execute $x = a - b$ while Adder Y calculates $y = x + q$. Depending on, whether $x < 0$ or $x \geq 0$, either the results of Adder X or Adder Y are selected as ouputs. If Adder X generates a carry, then the respective output of Adder X is selected as result, otherwise the respective output of Adder Y is taken as result.

The carry propagation is different for 256-bit, 32-bit and 16-bit operations. In case of a 256-bit operation, only the carries of the 16th adders within Adder X and Adder Y are considered. This carry bit is used to select all 16 adder results. For 16-bit (.16H) operations on the other hand, the carry of each adder is considered to select the respective output either from Adder X or Adder Y. For 32-bit operation (.8S), only every second carry is considered and selects the result for two subsequent 16-bit results. An illustration of this mechanism is given in Figure 8. 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.

Furthermore, we integrate the transpose functionality for bn.trn1/bn.trn2, as well as the vectorized shifts for bn.shv into BN-ALU. The shift operation can only partially be merged into the existing shifter and the transpose functionality is entirely new. However, both operations do not require a significant amount of hardware resources compared to existing modules within the BN-ALU.

Additionally, for the integration of Keccak, we integrated four special-purpose registers into the BN-ALU. Their functionality and how the Keccak core is interfaced trough these registers is described in detail in Section 4.

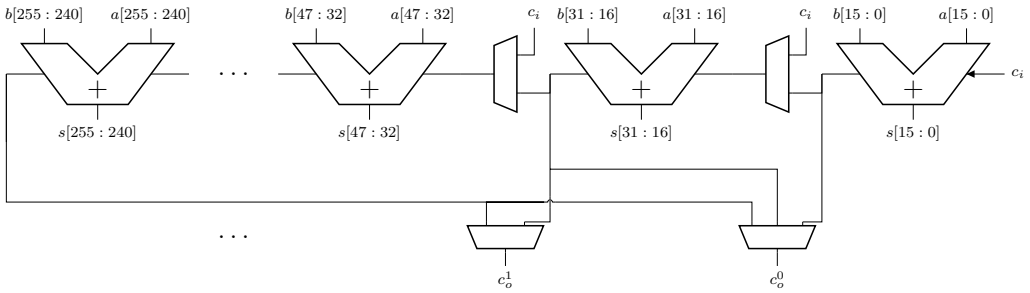


Figure 8: Output carry generation of 256-bit operations and vectorized 16-bit/32-bit operations in `Adder X` or `Adder Y`.

6.2.2 Modified Big Number MAC

We choose to replace the 64-bit multiplier within the BN-MAC with our configurable vectorized multiplier, introduced in Section 6.1.2. This enables the OTBN to execute 16-bit and 32-bit vectorized multiplications in addition to the original 64-bit multiplications. This comes at minimum cost, as all resources used for multiplication are reused. Only some multiplexers need to be added. Similarly, we replaced the original adder with our configurable vectorized adder as described in Section 6.1.1. It supports either 16-bit (`.16H`) or 32-bit (`.8S`) vectors as input operands. As explained above, we split the execution of one `bn.mulv` into several clock cycles. In the following, we describe the different execution stages for the different `bn.mulv` variants. Depending on the option, d is either set to 16 or 32. For `.8S` variants, $d = 32$ and for `.16H` variants, $d = 16$. The resulting architecture of our modified BN-MAC unit is depicted in Figure 9. To configure q and R, respectively, we added a dedicated connection from the `MOD` register within the BN-ALU to the BN-MAC module and use the content of this register. Specifically, for `.16H` variants, we use `MOD`[15:0] as q and `MOD`[47:32] as R. For `.8S` variants, we use `MOD`[31:0] as q and `MOD`[63:32] as R.

Execution stages for `bn.mulvm`. For the `bn.mulvm` instructions, our modified BN-MAC implements vectorized modular multiplication by leveraging the Montgomery multiplication algorithm, given in Algorithm 2.7. In particular, it takes 256-bit WDRs as operands, and operates on them quarter-word-wise (64-bit-wise). For each quarter word, in the first cycle $c = a \times b$ and $[c]_d$ is performed. In the second cycle, $m = [c \times R]_d$ is computed. To store the intermediate results computed in these first two clock cycles, we added the `TMP` register. Lastly, $r = [m \times q + c]^d$ and a conditional subtraction of r (if $r \geq q$) are left. We merge these two steps into one clock cycle. To achieve that, we integrate an additional subtractor into the BN-MAC. This additional subtractor saves one clock cycle per quarter word and only introduces a small area overhead. Furthermore, it does not seem reasonable to compute the conditional subtraction as a separate step with the BN-MAC. The partial results obtained per quarter word operation are stored and concatenated in the accumulator register `ACC`. Furthermore, $c = a \times b$, which is computed in the first cycle must be stored, as it is used in the third cycle again. Therefore, we integrated the register `C` into the BN-MAC unit. This approach requires 12 clock cycles per `bn.mulvm` instruction.

Execution stages for `bn.mulv`. Similar to the `bn.mulvm` instruction, for the `bn.mulv` instruction our modified BN-MAC takes 256-bit WDRs as operands, and operates on them quarter-word-wise. For the `bn.mulv` variants only one clock cycle per quarter word is required. Specifically, for each quarter word $c = a \times b$ and $[c]_d$ is computed and partial results are concatenated in the accumulator register.

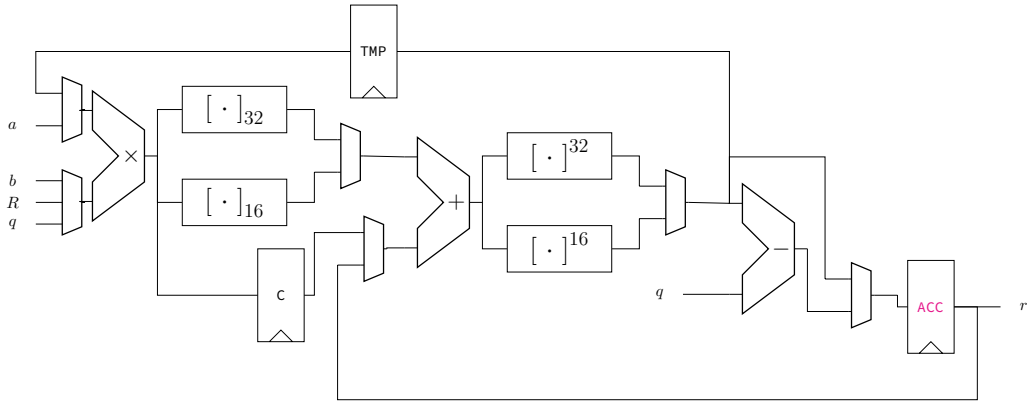


Figure 9: Architecture of our modified BN-MAC module.

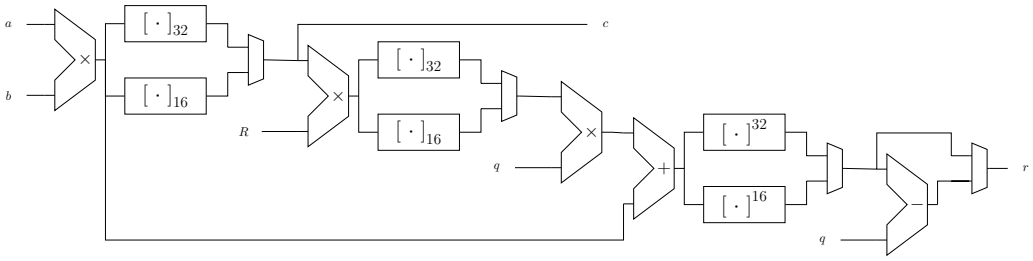


Figure 10: Architecture of our proposed BN-MULV module.

6.2.3 Big Number MULV Module

As described above, for the `bn.mulv` single-cycle approach, we integrate a complete new module into the OTBN pipeline, namely the BN-MULV module. The rationale for this module was laid out above. It enables high performance and keeps modifications to the OTBN's control logic to a minimum. The architecture of our newly proposed BN-MULV module is illustrated in Figure 10.

For the `bn.mulv` instructions without modular reduction, c is selected as output. For the `bn.mulvm` instructions, our BN-MULV module implements vectorized modular multiplication by also leveraging the Montgomery multiplication algorithm (Algorithm 2.7). It supports either 16-bit (`.16H`) or 32-bit (`.8S`) vectors as input operands. For the `.16H` variants it executes 16 16-bit in parallel, and for the `.8S` variants it executes 8 32-bit concurrently. It uses the same multiplier for both, `.16H` or 32-bit `.8S`, by following the configurable vectorized multiplication approach presented in Section 6.1.2. Similiarly, the adder and subtractor are shared for both variants. To achieve that, we follow the configurable adder approach presented in Section 6.1.1. The vectorized addition in line 2 followed by the vectorized conditional subtraction in line 3-5 of Algorithm 2.7, is implemented similar to the pseudo-modulo reduction within the BN-ALU for the `bn.addv` instruction. Selecting the lower d bits ($[\,.\,]_d$) or the upper d bits ($[\,.\,]^d$) is implemented efficiently in hardware as only additional routing resources are required. However, to differentiate between `.8S` and `.16H` variants, multiplexers need to be added. 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 3: FPGA synthesis - resource utilization on Xilinx 7-Series FPGAs

Design	LUT	FF	DSP	BRAM
BN-MAC	2,141	312	16	0
BN-MAC$_{\text{Ext.}}$	4,287	508	16	0
BN-MULV$_{\text{Ext.++}}$	10,474	0	96	0
BN-ALU	6,321	320	0	0
BN-ALU$^{\text{KMAC}}$	9,516	1,595	0	0
BN-ALU$_{\text{Ext.}}$	8,604	320	0	0
BN-ALU$^{\text{KMAC}}_{\text{Ext.}}$	12,442	1,595	0	0
Butterfly (Kyber, Dilithium) [SOSK23]	3,887	951	33	0
Butterfly (Kyber, NewHope) [FSS20]	2,908	170	9	0
Mod. arith. (NewHope) [AEL+20]	1,907	1,658	7	34
Mod. arith. (Kyber) [NDMZ+21]	178	0	5	0.5
Mod. arith. (Dilithium) [NDMZ+21]	377	0	10	0.5
Mod. arith. (Kyber) [LQYW24]	93	0	1	0
Mod. arith. (Dilithium) [LTQ+24]	312	0	4	0
Keccak [SOSK23]	1,312	0	0	0
Keccak [LTQ+24]	3,622	1,605	0	0
Keccak [FSS20]	3,847	0	0	0

Table 4: ASIC synthesis - resource utilization for 7nm process. Area is given in μm^2.

Design	Cell Count	Cell Area	Net Area	Total Area
BN-MAC	13,376	1,623	822	2,446
BN-MAC$_{\text{Ext.}}$	22,583	2,496	1,300	3,796
BN-MULV$_{\text{Ext.++}}$	100,123	10,530	5,389	15,918
BN-ALU	20,377	2,150	1,264	3,414
BN-ALU$^{\text{KMAC}}$	27,053	3,195	1,652	4,846
BN-ALU$_{\text{Ext.}}$	22,313	2,269	1,434	3,702
BN-ALU$^{\text{KMAC}}_{\text{Ext.}}$	34,967	4,044	2,096	6,140

6.2.4 Synthesis Results for Single Extensions

Table 3 and Table 4 present the synthesis results for Xilinx 7-Series devices and ASIC results for the ASAP7 PDK [CVS+16], respectively. For the BN-ALU these tables contain four different variants. BN-ALU is the reference implementation without any extension. BN-ALU$^{\text{KMAC}}$ includes the interface to the KMAC as described above and in Section 4. BN-ALU$_{\text{Ext.}}$ presents the results for the BN-ALU with our vector extensions only. Finally, BN-ALU$^{\text{KMAC}}_{\text{Ext.}}$ contains both, the KMAC interface and the vector extension. Our results indicate that the BN-ALU$_{\text{Ext.}}$ implementation does not introduce a significant overhead and most resources can be reused. However, the KMAC interface induces an overhead as we are not able to reuse existing hardware but need to add additional flip-flops due to the new special-purpose registers, extend the read and write ports for special purpose registers and introduce blanking, wiping and integrity protection countermeasures for the registers. Moreover, the interface to the KMAC contains a small FIFO. For our vectorized multiplication approach, Table 3 and Table 4 contain synthesis results for the original BN-MAC, its extended version BN-MAC$_{\text{Ext.}}$ which is described in Section 6.2.2 and the newly

proposed BN-MULV$_{\text{Ext.++}}$ module which is presented in Section 6.2.3. Our results indicate that the BN-MAC$_{\text{Ext.}}$ leads to a moderate increase in resources while our BN-MULV$_{\text{Ext.++}}$ implementation is several times larger than the BN-MAC. Considering the amount of required resources for parallel Montgomery multiplication pointed out in Section 6.2.3, this is expected. For the BN-MAC$_{\text{Ext.}}$, we integrated additional registers with corresponding integrity protections and checks. Furthermore, as explained in Section 6.2.2, we added an additional subtractor and a corresponding blanking mechanism. In combination with the required control logic these components make up the increase in resource consumption induced by the BN-MAC$_{\text{Ext.}}$ extension. Table 3 also offers a comparison with other tightly coupled accelerators from literature. We observe that extensions from literature are more compact. Mainly this is due to the fact that these numbers only account for very specific extensions and none of the modules include generic big number arithmetic for contemporary cryptography. Most of the works cited in Table 3 use 32-bit architectures, while our extensions operate on 64-bit or 256-bit. The Keccak numbers from literature show that integrating instruction set extensions for Keccak requires as many resources as or fewer resources than the KMAC interface. But as we will show later, massive performance gains can be achieved with the external KMAC. Further, none of the works consider features as needed to be compliant with OTBN's design rationale such as integrity protection, blanking and secure wipes. A more thorough comparison of our work with designs from literature that also consider the respective processors can be found in Section 7.5.

7 Results

In this section, we present the results of our work in terms of cycle counts, memory usage, code size, as well as field programmable gate array (FPGA) and application-specific integrated circuit (ASIC) synthesis results. For the cycle count, we consider the polynomial multiplication related functions and the full schemes separately. We also give comparisons to related work and other common implementation targets.

7.1 Testing & Benchmarking Setup

We test our implementations of ML-DSA and ML-KEM using the Python simulator for OTBN as provided by the OpenTitan team and for OTBN$^{\text{KMAC}}$, OTBN$^{\text{KMAC}}_{\text{Ext.}}$, and OTBN$^{\text{KMAC}}_{\text{Ext.++}}$ using the same simulator with additions of the KMAC interface and our new instructions, respectively. In order to evaluate functional correctness, we compare our implementations against open source Python implementations of Kyber and Dilithium by Pope[5], modified to match the ML-KEM and ML-DSA draft standards on more than 10 000 random inputs.

For obtaining the cycle counts on OTBN, OTBN$^{\text{KMAC}}$, OTBN$^{\text{KMAC}}_{\text{Ext.}}$, and OTBN$^{\text{KMAC}}_{\text{Ext.++}}$, we again make use of the cycle-accurate Python simulator in which we estimate the cycles for the KMAC interface based on the available data in the documentation and measurements evaluating the KMAC core. Computing one round of masked Keccak permutation takes 4 cycles; some additional overhead is incurred due to the interfacing to KMAC which we also account for in the simulation.

For our comparison to software implementations, we select ones updated to the NIST draft standards which are based on previously published work – if available. This is the case for the implementations using Intel AVX2, Arm Neon, and the implementations on the Cortex-M4. We redo most of the benchmarks ourselves and give more details on the exact setups in the following.

[5] https://github.com/GiacomoPope

- AVX2: We use the draft-standard compliant implementations provided by the KYBER and DILITHIUM teams[6] and compile it using `gcc` version 12.2.0. We obtain the benchmark results on an Intel Core i7-6700K Skylake processor with hyperthreading and Turbo Boost disabled running Debian 12.

- Neon: We make use of the implementation presented in [BHK+22], with adaptations to the NIST draft and further optimizations that have been made by the authors since the original publication[7] and compile it using `gcc` version 12.2.0. For benchmarking, we use a Raspberry Pi 4 with a Cortex-A72 processor running Debian 12.

- Cortex-M4: The benchmarking setup we use is based on pqm4 [KRSS19], including code from [HZZ+22] for KYBER, and [HAZ+24] for DILITHIUM, adapted to the NIST draft standards by Kannwischer. We compile the software using `arm-none-eabi-gcc` version 13.2.1 from the Arm GNU toolchain.

While the OTBN is a co-processor to the main Ibex processor, we consider a simple comparison of cycle counts of the cryptographic scheme on the OTBN to be fair. Granted, there might be scenarios in which the Ibex first has to configure the OTBN by loading its firmware. In most scenarios, however, this step can be prepared at boot-up. Writing data to the OTBN and reading back results is as fast as normal memory accesses as the two processors share certain memory sections. In fact, the OpenTitan architecture even allows to shield secrets such as the secret key from the Ibex via its key manager, which might be advantageous in certain scenarios and would make some data transfers unnecessary. For this paper, however, we only consider plain cycle counts on the OTBN.

7.2 Polynomial Multiplication

Table 5 shows the cycle counts for the polynomial multiplication related functions of ML-DSA and ML-KEM. From the table we can see that our implementation for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ outperforms the implementations on plain OTBN with speed-ups up to a factor of eight. The implementation on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ in turn is two to three times faster than the one on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$.

Comparing our results on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ to the closely related work from [SOSK23], we can observe slow-downs for the transformations of up to 18% in the case of ML-DSA, while we manage to speed up the pointwise multiplication by 30%. In contrast to this, we are up to two times faster in the case of ML-KEM. This can be traced back to the vectorization allowing an even higher degree of parallelization on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, while [SOSK23] does not consider a SIMD approach. The results from [Tur23] may suggest that applying the Kronecker+ technique from [BRv22] might not be suitable on OTBN, as our baseline implementation using Plantard arithmetic yields better results.

One may wonder why the AVX2 implementation on Intel Skylake (which has a similar register size) outperforms our work on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$. This is due to its super-scalar architecture and out-of-order (OoO) execution capabilities. We observe a similar trend for the likewise super-scalar Cortex-A72 using Arm Neon.

Compared to the work from [NDMZ+21], we achieve speed-ups up to a factor of 18 on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, which we mainly attribute to our vectorized approach. Despite the less general approach in [FSS20], we still manage to obtain a speed-up of nearly 2×.

[6] https://github.com/pq-crystals/kyber/commit/11d00ff1f20cfca1f72d819e5a45165c1e0a2816, https://github.com/pq-crystals/dilithium/commit/e7bed6258b9a3703ce78d4ec38021c86382ce31c

[7] https://github.com/neon-ntt/neon-ntt/commit/0de97e07f69002ed3219828d35ee438f3802bb34

Table 5: Benchmarks for polynomial multiplication related functions of ML-DSA and ML-KEM. All numbers given refer to cycles.

	Platform	NTT		INTT		Base Mul.	
ML-DSA	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ (This work)	996	(×0.41)	1003	(×0.39)	230	(×0.40)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ (This work)	2404	(×1.00)	2587	(×1.00)	582	(×1.00)
	$\text{OTBN/OTBN}^{\text{KMAC}}$ (This work)	8206	(×3.41)	8701	(×3.36)	2552	(×4.38)
	OTBN [SOSK23][a]	1972	(×0.82)	2244	(×0.87)	768	(×1.32)
	OTBN [Tur23]	10 763	(×4.48)	13 943	(×5.39)	9714	(×16.69)
	Skylake [LDK+22]	848	(×0.35)	806	(×0.31)	156	(×0.27)
	Cortex-A72 [BHK+22]	1802	(×0.75)	2535	(×0.98)	—	—
	Cortex-M4 [AHKS22]	8066	(×3.36)	8388	(×3.24)	1931	(×3.32)
	[NDMZ+21]	18 554	(×7.72)	21 375	(×8.26)	—	—
ML-KEM	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ (This work)	384	(×0.38)	392	(×0.36)	284	(×0.39)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ (This work)	1000	(×1.00)	1096	(×1.00)	724	(×1.00)
	$\text{OTBN/OTBN}^{\text{KMAC}}$ (This work)	8133	(×8.13)	8771	(×8.00)	4605	(×6.36)
	OTBN [SOSK23][a]	1454	(×1.45)	1726	(×1.57)	1448	(×2.00)
	Skylake [SAB+22]	218	(×0.22)	234	(×0.21)	86	(×0.12)
	Cortex-A72 [BHK+22]	955	(×0.96)	1128	(×1.03)	—	—
	Cortex-M4 [HZZ+22]	4474	(×4.47)	4684[b]	(×4.27)	2422	(×3.35)
	[NDMZ+21]	18 488	(×18.49)	18 488	(×16.87)	—	—
	[FSS20]	1935	(×1.94)	1930	(×1.76)	—	—
	[LQYW24]	4189	(×4.19)	3481	(×3.18)	3257	(×4.50)

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

7.3 Full Scheme Benchmarks

We present the benchmark results for all three parameter sets and all three algorithms of ML-DSA and ML-KEM in Tables 6 and 7.

As shown in Table 6, we achieve performance gains of a factor of six to nine, when comparing our implementation on plain OTBN with our implementation for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$. As expected, a large contribution to this is due to the KMAC interface which becomes apparent when considering the numbers for $\text{OTBN}^{\text{KMAC}}$. The implementation on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ is again up to 32% faster than the one on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$.

Comparing our work for $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ to the implementations for the verification from [SOSK23], 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.

Due to the fast Keccak accelerator, we even manage to outperform the super-scalar Cortex-A72 with Arm Neon. However, our performance on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ remains behind the AVX2 optimized implementation on Intel Skylake.

With respect to hardware/software co-designs, we achieve lower cycle counts than all the works in the comparison. From a performance perspective, the very compact implementation from [LTQ+24] is the closest to our work on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, while relying on specifically tailored extensions for Dilithium. A comparison of the respective hardware overheads will follow in Section 7.5.

When considering the performance of ML-KEM on $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, the situation is similar as for ML-DSA: We achieve significant speed-ups through the KMAC interface, with the overall performance gain due to our ISA extensions being larger than for ML-DSA. This can be traced back to the higher degree of parallelism for 16-bit elements. We outperform the implementation on plain OTBN by almost a factor of nine.

Again, $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ outperforms the Arm Neon implementation, but cannot keep up with the highly super-scalar AVX2 implementation.

The hardware/software co-design offering the most comparable performance is the one

presented in [FSS20]. While the work is similar due to the vectorized approach to modular arithmetic and the fact that it also employs a Keccak accelerator, it differs in the degree of specificity and the capabilities of the aforementioned accelerator. We achieve speed-ups of up to a factor of 4. As the work from [LQYW24] picks a highly resource-constrained approach without making use of any form of Keccak acceleration, it is no surprise that the speed-ups on $\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$ are as high as a factor of 17.

Our benchmark results also reflect a key-difference between the round 3 version of Kyber and the draft standard for ML-KEM, namely the encapsulation usually taking longer than the decapsulation for Kyber, while it is the other way around for ML-KEM. This is due to the draft standard omitting parts of the hashing originally present in Kyber [NIS23a].

Table 6: ML-DSA full scheme benchmarks. All numbers given refer to cycles. Median result was selected, if given. 10 000 iterations for our measurements.

Operation	Platform	Key Gen.		Sign		Verify	
ML-DSA-44	OTBN	1 242 455	(×8.29)	2 574 222	(×6.28)	1 226 370	(×7.75)
	$\text{OTBN}^{\text{KMAC}}$	270 888	(×1.81)	1 115 320	(×2.72)	318 783	(×2.01)
	$\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$	149 867	(×1.00)	410 186	(×1.00)	158 226	(×1.00)
	$\text{OTBN}_{\text{Ext.++}}^{\text{KMAC}}$	130 730	(×0.87)	287 122	(×0.70)	131 023	(×0.83)
	OpenTitan [SOSK23][b,c]	—	—	—	—	997 722	(×6.31)
	Skylake [LDK+22][a]	91 924	(×0.61)	207 014	(×0.50)	97 082	(×0.61)
	Cortex-A72 [BHK+22][a]	266 767	(×1.78)	632 345	(×1.54)	264 349	(×1.67)
	Cortex-M4 [HAZ+24][a]	1 352 958	(×9.03)	2 854 917	(×6.96)	1 343 288	(×8.49)
	[KSFS24][c]	593 403	(×3.96)	1 905 872	(×4.65)	651 217	(×4.12)
	[NDMZ+21][c]	1 592 325	(×10.62)	5 884 266	(×14.35)	1 700 679	(×10.75)
	[LTQ+24][c]	541 869	(×3.62)	845 005	(×2.06)	563 385	(×3.56)
ML-DSA-65	OTBN	2 190 278	(×8.39)	4 490 766	(×6.44)	2 107 440	(×8.22)
	$\text{OTBN}^{\text{KMAC}}$	438 154	(×1.68)	1 842 696	(×2.64)	493 307	(×1.92)
	$\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$	261 000	(×1.00)	697 203	(×1.00)	256 327	(×1.00)
	$\text{OTBN}_{\text{Ext.++}}^{\text{KMAC}}$	233 893	(×0.90)	477 322	(×0.68)	215 627	(×0.84)
	OpenTitan [SOSK23][b,c]	—	—	—	—	1 488 526	(×5.81)
	Skylake [LDK+22][a]	154 308	(×0.59)	342 708	(×0.49)	154 622	(×0.60)
	Cortex-A72 [BHK+22][a]	510 197	(×1.95)	1 053 606	(×1.51)	440 317	(×1.72)
	Cortex-M4 [HAZ+24][a]	2 390 080	(×9.16)	4 878 759	(×7.00)	2 289 269	(×8.93)
	[KSFS24][c]	1 067 824	(×4.09)	3 253 378	(×4.67)	1 126 938	(×4.40)
	[NDMZ+21][c]	2 974 897	(×11.40)	10 211 677	(×14.65)	2 963 936	(×11.56)
	[LTQ+24][c]	902 273	(×3.46)	1 329 844	(×1.91)	918 863	(×3.58)
ML-DSA-87	OTBN	3 752 708	(×9.14)	6 193 418	(×6.78)	3 676 261	(×8.72)
	$\text{OTBN}^{\text{KMAC}}$	691 121	(×1.68)	2 358 194	(×2.58)	769 517	(×1.83)
	$\text{OTBN}_{\text{Ext.}}^{\text{KMAC}}$	410 599	(×1.00)	913 609	(×1.00)	421 498	(×1.00)
	$\text{OTBN}_{\text{Ext.++}}^{\text{KMAC}}$	365 484	(×0.89)	656 032	(×0.72)	361 557	(×0.86)
	OpenTitan [SOSK23][b,c]	—	—	—	—	2 223 143	(×5.27)
	Skylake [LDK+22][a]	244 128	(×0.59)	430 214	(×0.47)	242 666	(×0.58)
	Cortex-A72 [BHK+22][a]	776 238	(×1.89)	1 408 686	(×1.54)	753 514	(×1.79)
	Cortex-M4 [HAZ+24][a]	4 071 579	(×9.92)	6 638 503	(×7.27)	3 986 607	(×9.46)
	[KSFS24][c]	1 784 767	(×4.35)	4 357 249	(×4.77)	1 848 324	(×4.39)
	[NDMZ+21][c]	5 001 302	(×12.18)	13 339 255	(×14.60)	5 132 776	(×12.18)
	[LTQ+24][c]	1 533 230	(×3.73)	2 065 456	(×2.26)	1 561 021	(×3.70)

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

Table 7: ML-KEM full scheme benchmarks. All numbers given refer to cycles. Median result was selected, if given. 10 000 iterations for our measurements.

Operation	Platform	Key Gen.		Encaps		Decaps	
ML-KEM-512	OTBN	324 075	(×8.87)	352 716	(×7.56)	399 128	(×6.86)
	$\text{OTBN}^{\text{KMAC}}$	88 918	(×2.43)	120 212	(×2.58)	165 718	(×2.85)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$	36 554	(×1.00)	46 649	(×1.00)	58 160	(×1.00)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	32 330	(×0.88)	40 567	(×0.87)	48 930	(×0.84)
	Skylake [SAB+22][a]	29 624	(×0.81)	31 084	(×0.67)	30 464	(×0.52)
	Cortex-A72 [BHK+22][a]	59 567	(×1.63)	63 576	(×1.36)	73 354	(×1.26)
	Cortex-M4 [HZZ+22][a]	369 323	(×10.10)	368 577	(×7.90)	404 159	(×6.95)
	[NDMZ+21][b]	419 597	(×11.48)	438 280	(×9.40)	100 796	(×1.73)
	[FSS20][c]	150 106	(×4.11)	193 076	(×4.14)	204 843	(×3.52)
	[LQYW24][b]	622 000	(×17.02)	785 000	(×16.83)	713 000	(×12.26)
ML-KEM-768	OTBN	563 731	(×8.10)	611 598	(×7.45)	671 625	(×6.89)
	$\text{OTBN}^{\text{KMAC}}$	159 774	(×2.30)	197 884	(×2.41)	258 545	(×2.65)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$	69 565	(×1.00)	82 055	(×1.00)	97 471	(×1.00)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	61 909	(×0.89)	72 009	(×0.88)	83 129	(×0.85)
	Skylake [SAB+22][a]	47 768	(×0.69)	46 858	(×0.57)	47 474	(×0.49)
	Cortex-A72 [BHK+22][a]	95 875	(×1.38)	105 436	(×1.28)	117 905	(×1.21)
	Cortex-M4 [HZZ+22][a]	603 140	(×8.67)	622 059	(×7.58)	668 899	(×6.86)
	[NDMZ+21][b]	694 504	(×9.98)	731 597	(×8.92)	130 348	(×1.34)
	[FSS20][c]	273 370	(×3.93)	325 888	(×3.97)	340 418	(×3.49)
	[LQYW24][b]	988 000	(×14.20)	1 237 000	(×15.08)	1 133 000	(×11.62)
ML-KEM-1024	OTBN	911 648	(×8.02)	966 529	(×7.53)	1 044 112	(×7.03)
	$\text{OTBN}^{\text{KMAC}}$	249 490	(×2.19)	294 623	(×2.30)	370 528	(×2.50)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$	113 689	(×1.00)	128 339	(×1.00)	148 439	(×1.00)
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	101 716	(×0.89)	113 453	(×0.88)	128 059	(×0.86)
	Skylake [SAB+22][a]	64 608	(×0.57)	65 536	(×0.51)	67 870	(×0.46)
	Cortex-A72 [BHK+22][a]	150 581	(×1.32)	161 850	(×1.26)	184 320	(×1.24)
	Cortex-M4 [HZZ+22][a]	959 511	(×8.44)	976 865	(×7.61)	1 036 665	(×6.98)
	[NDMZ+21][b]	1 090 458	(×9.59)	1 126 462	(×8.78)	159 639	(×1.08)
	[FSS20][c]	349 673	(×3.08)	405 477	(×3.16)	424 682	(×2.86)
	[LQYW24][b]	1 543 000	(×13.57)	1 851 000	(×14.42)	1 719 000	(×11.58)

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

7.4 Memory & Code Size

Tables 8 and 9 present the stack usage and code size for our ML-KEM and ML-DSA implementations, including the OTBN, $\text{OTBN}^{\text{KMAC}}$, and $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ variants. The $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ variant shares code with $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$, resulting in identical memory consumption and code size. Notably, $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ generally uses less memory, and thanks to vectorization, has smaller code sizes compared to their baseline counterparts. While memory and code size optimization were not our primary focus, the stack usage for key generation and verification in ML-DSA remains comparable to that of [HZZ+22]. Signing in ML-DSA slightly exceeds Cortex-M4 usage but remains within reasonable limits. Our three verifications are also only slightly larger than [SOSK23]. Our test structure provides separate figures for the three routines, unlike [KSFS24], hindering direct comparison. ML-KEM's stack usage is considerably higher, more than doubled compared to Cortex-M4. However, since we have not optimized for stack size and ML-DSA already demands significant memory, this is not a concern for our consideration. Our code sizes also approximate those on Cortex-M4, slightly higher for ML-DSA and slightly smaller for ML-KEM with $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ variant.

Table 8: 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	5776	8336	8400	37 740	56 028	36 156
	$\text{OTBN}^{\text{KMAC}}$	5600	8224	8224	37 328	55 712	35 840
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	5536	8160	8160	37 248	50 624	34 720
	OpenTitan [SOSK23][a,b]	—	—	—	—	—	$\leq$ 32 000
	Skylake [LDK+22]	100	100	100	100	100	100
	Cortex-M4 [HZZ+22]	4364	5436	5412	38 296	49 416	36 184
	[KSFS24]	—	—	—		61 216[c]	
ML-KEM-768 ML-DSA-65	OTBN	9808	12 880	12 944	60 268	85 724	57 692
	$\text{OTBN}^{\text{KMAC}}$	9632	12 768	12 768	59 856	85 416	57 376
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	9568	12 704	12 704	59 776	78 272	56 384
	OpenTitan [SOSK23][a,b]	—	—	—	—	—	$\leq$ 32 000
	Skylake [LDK+22]	100	100	100	100	100	100
	Cortex-M4 [HZZ+22]	5396	6468	6452	60 824	68 864	57 720
	[KSFS24]	—	—	—		92 720[c]	
ML-KEM-1024 ML-DSA-87	OTBN	14 928	18 512	18 576	97 132	123 612	92 764
	$\text{OTBN}^{\text{KMAC}}$	14 752	18 400	18 400	96 720	123 304	92 448
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	14 688	18 336	18 336	96 640	121 280	91 456
	OpenTitan [SOSK23][a,b]	—	—	—	—	—	$\leq$ 32 000
	Skylake [LDK+22]	100	100	100	100	100	100
	Cortex-M4 [HZZ+22]	6436	7500	7484	97 688	115 968	92 824
	[KSFS24]	—	—	—		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 9: 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	18 160	3744	3360	21 904	25 392	5632	9696	31 024
	$\text{OTBN}^{\text{KMAC}}$	15 300	2688	3360	17 988	20 136	4800	9696	24 936
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	9620	1536	3360	11 156	17 636	2592	9696	20 228
	[FSS20][b]	—	—	—	12 532	—	—	—	—
	[KSFS24]	—	—	—	—	20 624	—	—	—
	Cortex-M4 [HZZ+22]	—	—	—	15 824	—	—	—	18 596
ML-KEM-768 ML-DSA-65	OTBN	18 660	3744	4832	22 404	26 168	5632	12 704	31 800
	$\text{OTBN}^{\text{KMAC}}$	15 800	2688	4832	18 488	20 112	4800	12 704	24 912
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	10 072	1536	4832	11 608	17 524	2592	12 704	20 116
	[FSS20][b]	—	—	—	11 658	—	—	—	—
	[KSFS24]	—	—	—	—	20 052	—	—	—
	Cortex-M4 [HZZ+22]	—	—	—	15 992	—	—	—	18 588
ML-KEM-1024 ML-DSA-87	OTBN	21 716	3744	6464	25 460	27 052	5632	15 520	32 684
	$\text{OTBN}^{\text{KMAC}}$	18 856	2688	6464	21 544	20 956	4800	15 520	25 756
	$\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$	13 524	1536	6464	15 060	18 484	2592	15 520	21 076
	[FSS20][b]	—	—	—	12 874	—	—	—	—
	[KSFS24]	—	—	—	—	20 324	—	—	—
	Cortex-M4 [HZZ+22]	—	—	—	16 912	—	—	—	18 468

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

7.5 Hardware Utilization and Comparison to other HW/SW Co-designs

As shown in the previous sections, in terms of cycle counts and latency, we outperform most existing relevant RISC-V-based ISA extensions from literature [FSS20, NDMZ+21, KSFS24, LTQ+24, LQYW24]. In summary, this can be attributed mainly to three points, some of which were already discussed above. First, our approach exploits the 256-bit WDRs of the OTBN to perform operations in a SIMD manner. This provides a significant advantage compared to 32-bit or 64-bit RISC-V architectures. Another work [SOSK23] also leverages the OTBN and its WDRs. However, their extensions compute only one 32-bit butterfly operation per clock cycle and does not exploit the WDRs for SIMD operations. Second, the enormous capacities of the OTBN's WDRs allow us to reduce memory accesses to a minimum. Third, we implemented a dedicated interface to the OpenTitan KMAC core. This module is able to compute a Keccak round within 4 cycles. The work presented in [SOSK23] implements ISA extensions for Keccak. Their approach takes 40 cycles per Keccak round. This is another reason for the performance improvement of our work over [SOSK23]. Although the Keccak extension of [FSS20, KSFS24] is able to compute one round per clock cycle, their Keccak accelerator needs additional floating point registers and accesses all of them, together with some general purpose 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 as their approach and allows a cleaner integration.

Table 11 presents the ASIC synthesis results using the ASAP7 PDK[CVS+16]. We synthesized the Top-Earlgrey design rather than the Chip-Earlgrey-ASIC design due to missing standard cells in the PDK. The Chip-Earlgrey-ASIC is built on top of Top-Earlgrey and contains additional module such as an analog-sensor interface and pads. Furthermore, the table contains synthesis results for the OTBN with different variants of our extensions. For both designs, we applied a memory as black box synthesis and only targeted logic overhead as the memory requirements for all different variants are similar.

These numbers highlight that the performance improvement of our $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ implementation comes at rather low cost. On the other hand, the enormous performance improvement of our $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$ is relatively costly and nearly doubles the size of the OTBN. However, considering the OpenTitan's overall area, it is still a reasonable approach and does not have a significant impact.

We analyzed the effect of our extensions on the critical path by evaluating out-of-context FPGA synthesis results for the OTBN and Xilinx 7-Series devices as target. For the original OTBN, the critical path is located within the BN-MAC. For $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$, the critical path changes and is located within the BN-ALU, going through `Adder X` and `Adder Y`. For $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$, the critical path changes as well and is located within our newly proposed BN-MULV module. Considering the complexity of the implemented operations, these changes of the critical path are expected. Compared to the original OTBN, the maximum clock frequency is decreased from 39.3 MHz to 21.3 MHz for the $\text{OTBN}^{\text{KMAC}}_{\text{Ext.}}$ implementation and to 18.6 MHz for the $\text{OTBN}^{\text{KMAC}}_{\text{Ext.++}}$. Note that these numbers are limited in their significance and only few conclusions can be drawn from them. First, our hardware extensions affect the critical path and this must be considered when building an OpenTitan-based ASIC. Second, the purpose of the CW310 FPGA implementation is prototyping only and it's not intended to be a final product. Hence, the target frequency for the OTBN on the FPGA is only 10 MHz, meaning that our hardware extensions have zero impact on the maximum clock frequency in this case. Third, the impact of our hardware extensions on the maximum clock frequency of an ASIC design is difficult to quantify precisely, because the maximum clock frequency is highly dependent on the target platform and technology node. When implemented on an ASIC, the OpenTitan has a moderate target frequency of 100 MHz. In our opinion, the critical path in our hardware extensions still allows to achieve this frequency for many manufacturing processes.

Table 10: Comparison of hardware utilization of this work with state-of-the-art HW/SW Co-designs. The overhead with relation to the base platform the ISA extension was applied on are given as factors next to the absolute values.

Design	ASIC		FPGA					
	Cell Count		LUT		FF		DSP	
Top-Earlgrey$^{\text{KMAC}}_{\text{Ext.}}$	754 208	(×1.02)	244 599	(×1.05)	122 053	(×1.01)	22	(×1.00)
Top-Earlgrey$^{\text{KMAC}}_{\text{Ext.++}}$	839 033	(×1.13)	253 513	(×1.09)	121 871	(×1.01)	118	(×5.36)
OTBN$^{\text{KMAC}}$	160 586	(×1.07)	35 421	(×1.10)	16 874	(×1.08)	16	(×1.00)
OTBN$^{\text{KMAC}}_{\text{Ext.}}$	167 564	(×1.12)	38 236	(×1.19)	15 807	(×1.01)	16	(×1.00)
OTBN$^{\text{KMAC}}_{\text{Ext.++}}$	310 031	(×2.07)	49 128	(×1.53)	16 948	(×1.09)	112	(×7.00)
[SOSK23]	—	—	55 409	(×1.66)	16 575	(×1.06)	49	(×3.06)
[FSS20]	57 413	(×1.59)	24 306	(×1.59)	10 837	(×1.13)	18	(×3.00)
[KSFS24]	65 968	(×1.50)	22 356	(×1.48)	13 181	(×1.33)	13	(×2.17)
[NDMZ+21]	—	—	64 855	(×1.06)	60 349	(×1.00)	29	(×1.53)
[LQYW24]	13 573	(×1.04)	9614	(×1.01)	6669	(×1.00)	5	(×1.25)
[LTQ+24]	22 936	(×2.18)	15 258	(×1.35)	12 934	(×1.14)	7	(—)

In terms of hardware utilization of related work, Table 10 provides more insights. For our FPGA synthesis results, we choose the Xilinx 7-Series devices as target and Table 10 includes synthesis results for the OTBN and the Chip-Earlgrey-CW310 design.

In [NDMZ+21], a CVA6, a more powerful application level processor is chosen. The authors of [FSS20, KSFS24] use a PULPino as platform, which is a microcontroller with slightly more features than an Ibex. In [LTQ+24, LQYW24], a very compact Hummingbird E203 core was used.

In general, it must be said that all comparisons except for [SOSK23] are not straightforward, as the OTBN is a very specific target platform. Due to its big-number arithmetic modules and countermeasures it is not compact, but still missing features that other platforms already have. Further, OTBN's fault injection and side-channel countermeasures imply that all extensions must consider the same countermeasures.

The extensions in [NDMZ+21, SOSK23, LTQ+24, LQYW24] are significantly more compact, but offer less performance. For [FSS20, KSFS24, LTQ+24], the relative overhead is larger, but both the base and extended platform are more compact than our extended OTBN.

In summary, existing designs might be better suited for a few specific use-cases, where more compact base platforms are required. However, as the first industry-grade open source secure element, our claim to our extensions for the OpenTitan is a clean integration into the micro-architecture, flexibility and high performance without too much hardware overhead. Our comparison with state-of-the-art designs shows that the hardware costs of our extensions are acceptable both in relation to related work and when the entire OpenTitan is taken into consideration.

8 Discussion and Future Work

As mentioned in Section 5.1, we choose a different approach compared to most related work by trying to provide a rather generic ISA extension for vector arithmetic, rather than highly specific instructions tailored towards lattice-based cryptography or even specific schemes. We made this decision assuming that other cryptographic schemes may also profit from efficient, vectorized modular arithmetic on "small" integers. Examples for this would be code-based schemes such as Classic McEliece [ABC+22] or multivariate quadratic (MQ)-based schemes. Further, we believe that our extension could be relevant

Table 11: ASIC synthesis - area consumption for 7nm Process without Memories. Area is given in μm^2.

Design	Cell Count		Cell Area		Net Area		Total Area	
Top-Earlgrey	740 101	(×1.00)	106 885	(×1.00)	41 763	(×1.00)	148 647	(×1.00)
Top-Earlgrey$^{\text{KMAC}}_{\text{Ext.}}$	754 208	(×1.02)	108 698	(×1.02)	42 610	(×1.02)	151 308	(×1.02)
Top-Earlgrey$^{\text{KMAC}}_{\text{Ext.++}}$	839 033	(×1.13)	117 654	(×1.10)	47 308	(×1.13)	164 962	(×1.11)
OTBN	149 931	(×1.00)	19 746	(×1.00)	8196	(×1.00)	27 942	(×1.00)
OTBN$^{\text{KMAC}}$	160 586	(×1.07)	21 467	(×1.09)	8862	(×1.08)	30 329	(×1.09)
OTBN$^{\text{KMAC}}_{\text{Ext.}}$	167 564	(×1.12)	21 759	(×1.10)	9369	(×1.14)	31 128	(×1.11)
OTBN$^{\text{KMAC}}_{\text{Ext.++}}$	310 031	(×2.07)	36 144	(×1.83)	16 150	(×1.97)	52 295	(×1.87)

for accelerating symmetric schemes, especially from the domain of Add-Rotate-Xor (ARX) ciphers for which no hardware acceleration is present on OpenTitan.

A straight-forward follow-up would be to apply the techniques for reducing the memory usage presented in [GKS21, BRS22]. In this light, it would be interesting to see how the trade-offs on OTBN would differ, assuming access to the fast KMAC block for the hashing. In the same context, it could be considered whether extending the ISA with a bit-mask-based permutation instruction to allow for vectorized rejection sampling as in [GS16] would be worthwhile with most stack optimizations shifting the runtime towards the sampling.

As OpenTitan already offers a masked KMAC core, extending our work to masked implementations of ML-KEM and ML-DSA whilst re-evaluating the adequacy of our proposed extensions could be worthwhile.

Further, the suitability of our ISA extension to, e.g., the Falcon verification, signature schemes from NIST's on-ramp process, or fully homomorphic encryption could be studied.

With OpenTitan aiming to provide a product with high security standards, a formally verified re-implementation of ML-KEM and ML-DSA on OTBN would be a logical next step. OTBN-support for the Jasmin language [ABB+17] is a current work-in-progress by Arranz Olmos[8].

As future work, the design space could be further explored and different optimization could be applied. More specifically, our multiplier presented in Section 6.1.2 does only use four of its sixteen 16-bit multipliers for Kyber. However, for Kyber's 16-bit multiplications no additional carry-save-adders are necessary for partial product combination. Therefore, it would be possible to increase the number of parallel executed 16-bit multiplications in a potentially cheap way.

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 course 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 programme between the Centre National de la Recherche Scientifique (CNRS) and the Max-Planck-Gesellschaft (MPG); and by the French National Research Agency in the framework of the "Investissements d'avenir" program (ANR-15-IDEX-02).

[8] https://github.com/sarranz/jasmin/tree/demo1

References

[AAA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical report, US Department of Commerce, NIST, 2020. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf. 1

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1807–1823. ACM Press, October / November 2017. 40

[ABBO24] David Adrian, Bob Beck, David Benjamin, and Devon O'Brien. Advancing our amazing bet on asymmetric cryptography, 2024. https://blog.chromium.org/2024/05/advancing-our-amazing-bet-on-asymmetric.html. 2

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions. 39

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR TCHES*, 2020(3):336–357, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8593. 3

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 327–343. USENIX Association, August 2016. 13

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic. *IACR TCHES*, 2020(3):219–242, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8589. 31

[AES01] Advanced Encryption Standard (AES). National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, November 2001. 11

[AHH+18] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR TCHES*, 2019(1):169–208, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7338. 2

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe

Ateniese and Daniele Venturi, editors, *ACNS 22*, volume 13269 of *LNCS*, pages 853–871. Springer, Heidelberg, June 2022. 3, 34

[AOB+24] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, Fran cois Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber – episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, Lecture Notes in Computer Science, page to appear. Springer-Verlag Berlin Heidelberg, 2024. http://cryptojedi.org/papers/#hakyberv. 7

[App24] Apple Security Engineering and Architecture (SEAR). iMessage with PQ3: The new state of the art in quantum-secure messaging at scale, 2024. https://security.apple.com/blog/imessage-pq3/. 2

[BBC+24] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Armin Namavari, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, , and Alex Stamos. Zoom cryptography whitepaper. Technical report, Zoom Video Communications, Inc., 2024. version 4.5, https://raw.githubusercontent.com/zoom/zoom-e2e-whitepaper/master/zoom_e2e.pdf. 2

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES*, 2022(1):221–244, 2022. 3, 20, 22, 33, 34, 35, 36

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19*, volume 11627 of *LNCS*, pages 209–228. Springer, Heidelberg, July 2019. 3

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. Dilithium for memory constrained devices. In Lejla Batina and Joan Daemen, editors, *AFRICACRYPT 22*, volume 2022 of *LNCS*, pages 217–235. Springer Nature, July 2022. 3, 40

[BRv22] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. Post-quantum cryptography with contemporary co-processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3683–3697. USENIX Association, August 2022. 33

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR TCHES*, 2019(4):17–61, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8344. 2

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. 8

[CVS+16] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. Asap7: A

7-nm finfet predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016. 31, 38

[Dil23] Dilithium. pq-crystals, November 2023. https://github.com/pq-crystals/dilithium. 3, 9, 10, 14, 16, 24

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR TCHES*, 2018(1):238–268, 2018. https://tches.iacr.org/index.php/TCHES/article/view/839. 4

[EWP+23] Wesley Evans, Bas Westerbaan, Christopher Patton, Peter Wu, and Vânia Gonçalves. Post-quantum cryptography goes GA, 2023. https://blog.cloudflare.com/post-quantum-cryptography-ga/. 2

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR TCHES*, 2022(1):414–460, 2022. 4

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999. 6

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled accelerators for post-quantum cryptography. *IACR TCHES*, 2020(4):239–280, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8683. 4, 31, 33, 34, 35, 36, 37, 38, 39

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR TCHES*, 2021(1):1–24, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8725. 3, 40

[GMR21] Aurélien Greuet, Simon Montoya, and Guénaël Renault. On using RSA/ECC coprocessor for ideal lattice-based key exchange. In Shivam Bhasin and Fabrizio De Santis, editors, *COSADE 2021*, volume 12910 of *LNCS*, pages 205–227. Springer, Heidelberg, October 2021. 2

[GS66] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966. 8

[GS16] Shay Gueron and Fabian Schlieker. Speeding up R-LWE Post-quantum Key Exchange. In Billy Bob Brumley and Juha Röning, editors, *Secure IT Systems*, Lecture Notes in Computer Science, pages 187–198, Cham, 2016. Springer International Publishing. https://eprint.iacr.org/2016/467. 24, 40

[HAZ+24] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium implementations on ARMv7-M. *IACR TCHES*, 2024(2):1–24, 2024. 3, 33, 35

[HBD+22] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian

Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS$^+$. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 1

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR TCHES*, 2022(4):614–636, 2022. 3, 10, 17, 18, 33, 34, 36, 37

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019. 11

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. https://www.ams.org/mcom/1987-48-177/S0025-5718-1987-0866109-5/. 11

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844. 17, 33

[KS24] Ehren Kret and Rolfe Schmidt. The pqxdh key agreement protocol. Technical report, Signal, 2024. revision 3, https://signal.org/docs/specifications/pqxdh/pqxdh.pdf. 2

[KSFS24] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on RISC-V with hardware acceleration. *ACM Transactions on Embedded Computing Systems*, 23(2):30:1–30:23, 2024. 4, 35, 36, 37, 38, 39

[KSS24] Patrick Karl, Jonas Schupp, and Georg Sigl. The impact of hash primitives and communication overhead for hardware-accelerated sphincs+. In Romain Wacquez and Naofumi Homma, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 221–239, Cham, 2024. Springer Nature Switzerland. 20

[Kyb23] Kyber. https://github.com/pq-crystals/kyber/tree/a621b8dde405cc507cbcfc5f794570a4f98d69cc, December 2023. 9, 14, 16, 24

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 1, 4, 20, 34, 35, 37

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16*, volume 10052 of *LNCS*, pages 124–139. Springer, Heidelberg, November 2016. 13

[LQYW24] Lu Li, Guofeng Qin, Yang Yu, and Weijia Wang. Compact instruction set extensions for kyber. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(3):756–760, 2024. 4, 31, 34, 35, 36, 38, 39

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR TCHES*, 2019(3):180–201, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8293. 13

[LTQ+24] Lu Li, Qi Tian, Guofeng Qin, Shuaiyu Chen, and Weijia Wang. Compact instruction set extensions for dilithium. *ACM Transactions on Embedded Computing Systems*, 23(2):23:1–23:21, 2024. 4, 31, 34, 35, 38, 39

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Heidelberg, December 2009. 4

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, Heidelberg, August 1986. 11

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. https://www.ams.org/mcom/1985-44-170/S0025-5718-1985-0777282-X/. 9, 10

[NDMZ+21] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A risc-v post quantum cryptography instruction set extension for number theoretic transform to speed-up crystals algorithms. *IEEE Access*, 9:150798–150808, 2021. https://ieeexplore.ieee.org/abstract/document/9605604/. 4, 31, 33, 34, 35, 36, 38, 39

[NIS23a] NIST. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (Draft). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf, August 2023. 1, 4, 6, 7, 8, 35

[NIS23b] NIST. FIPS 204: Module-Lattice-Based Digital Signature Standard (Draft). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.ipd.pdf, August 2023. 1, 4, 5, 6

[NIS23c] NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard (Draft). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.ipd.pdf, August 2023. 1, 17

[Ope23a] OpenTitan Team. Datasheet - OpenTitan Documentation. https://opentitan.org/book/hw/top_earlgrey/doc/specification.html, 2023. (accessed 2023-12-10). 11, 12

[Ope23b] OpenTitan Team. OpenTitan's RTL Freeze - Leveraging Transparency to Create Trustworthy Computing · lowRISC: Collaborative open silicon engineering. https://lowrisc.org/blog/2023/06/opentitans-rtl-freeze-leveraging-transparency-to-create-trustworthy-computing/, June 2023. (accessed 2023-11-13). 11

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute

of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 1

[Pla21] Thomas Plantard. Efficient Word Size Modular Arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, July 2021. https://thomas-plantard.github.io/pdf/Plantard21.pdf. 10, 12

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978. 11

[Saa23] Markku-Juhani O. Saarinen. Benchmarking RISC-V Post-Quantum Crypto. https://mjos.fi/doc/20231108-rvsummit-pqc.pdf, November 2023. 20

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 1, 6, 34, 36

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. https://eprint.iacr.org/2018/039. 9

[SHA15] Secure hash algorithm-3. National Institute of Standards and Technology, NIST FIPS PUB 202, U.S. Department of Commerce, August 2015. 11

[SOSK23] Tobias Stelzer, Felix Oberhansl, Jonas Schupp, and Patrick Karl. Enabling Lattice-Based Post-Quantum Cryptography on the OpenTitan Platform. In *Proceedings of the 2023 Workshop on Attacks and Solutions in Hardware Security*, ASHES '23, pages 51–60, New York, NY, USA, November 2023. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3605769.3623993. 4, 31, 33, 34, 35, 36, 37, 38, 39

[Tur23] Horia Turcuman. Speeding-up Post-Quantum Cryptography on an RSA Co-Processor. Master's thesis, Technical University of Munich, Munich, September 2023. https://github.com/horiaionut/kroneker-plus-on-otbn/blob/5576d7b035f5fe55a7199987ea05613d4aa913e7/paper.pdf. 33, 34

[ZXXH22] Yifan Zhao, Ruiqi Xie, Guozhu Xin, and Jun 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*, 69(7):2871–2884, July 2022. https://ieeexplore.ieee.org/document/9748063. 4