Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1

Hanno Becker[1], Vincent Hwang[2,3], Matthias J. Kannwischer[3], Bo-Yin Yang[3] and Shang-Yi Yang[4]

[1] Arm Research, Cambridge, UK
hanno.becker@arm.com
[2] National Taiwan University, Taipei, Taiwan
vincentvbh7@gmail.com
[3] Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu, by@crypto.tw
[4] Chelpis Co. Ltd., Taipei, Taiwan
nick.yang@chelpis.com

Abstract. We present new speed records on the Arm-v8A architecture for the lattice-based schemes Dilithium, Kyber, and Saber. The core novelty in this paper is the combination of Montgomery multiplication and Barrett reduction resulting in "Barrett multiplication" which allows particularly efficient modular one-known-factor multiplication using the Arm-v8A Neon vector instructions. These novel techniques combined with fast two-unknown-factor Montgomery multiplication, Barrett reduction sequences, and interleaved multi-stage butterflies result in significantly faster code. We also introduce "asymmetric multiplication" which is an improved technique for caching the results of the incomplete NTT, used e.g. for matrix-to-vector polynomial multiplication. Our implementations target the Arm Cortex-A72 CPU, on which our speed is 1.7× that of the state-of-the-art matrix-to-vector polynomial multiplication in Kyber [Nguyen–Gaj 2021]. For Saber, NTTs are far superior to Toom–Cook multiplication on the Arm-v8A architecture, outrunning the matrix-to-vector polynomial multiplication by 2.1×. On the Apple M1, our matrix-vector products run 2.1× and 1.9× faster for Kyber and Saber respectively.

Keywords: NIST PQC · Arm-v8A · Neon · Dilithium · Kyber · Saber

1 Introduction

When large quantum computers arrive, Shor's algorithm [Sho97] will break almost all currently deployed public-key cryptography by solving the integer factorization and the discrete logarithms problems. Preparing for this, the U.S. National Institute of Standards and Technology (NIST) has initiated a process to select new cryptosystems that withstand the increased capabilities of quantum computers – an area known as Post-Quantum Cryptography (PQC). This process naturally divides into categories of digital signatures and key encapsulation mechanisms (KEMs) [NIS] and is currently in the third round, where 7 finalists and 8 alternate candidates still compete [AASA+20].

Undoubtedly, the scheme(s) selected at the end of the NIST PQC competition will become prominent computational workloads of the future. It is therefore important to understand their performance and resource characteristics on the hugely diverse spectrum of today's and upcoming computing platforms, ranging from low-power IoT devices over desktops to high-end data center cores, to name some. Representative of the former, the focus of performance analysis of PQC on embedded devices has so far been the Arm®

Cortex™-M4 CPU. Representative of the latter, the focus of performance analysis on high-end cores has so far been the AVX2-capable Intel® CPUs Haswell and Skylake.

In this article, we contribute to the performance analysis of prominent PQC candidates on CPUs implementing the application profile / A-profile of the Arm architecture[1] – an area which, despite its importance, has been comparatively little studied so far. CPUs implementing the A-profile are ubiquitous, and form a wide spectrum in themselves: It contains power-efficient CPUs like the Cortex-A7 processor for Linux-capable embedded IoT devices, cores for the mobile and desktop market like the Cortex-A78 processor, as well as the Arm® Neoverse™ IP for infrastructure applications, for example, the Arm Neoverse-based AWS Graviton processors. The Fugaku supercomputer ranked as the fastest supercomputer in the world in 2020 and 2021 [TOP20, TOP21], is also based on a core implementing the A-profile. Considering the breadth of use and availability of A-profile cores in the computing ecosystem, it is therefore important to include them in the performance evaluation for PQC.

Another axis of distinction within the A-profile is the specific version of the architecture, such as Arm-v7A, Arm-v8A, and, as of late, Arm-v9A, and their respective sets of extensions. In this article, we focus on implementations of PQC on CPUs based on the 64-bit Arm-v8A architecture, leveraging the availability of the Arm-v8A version of the Arm® Neon™ Single Instruction Multiple Data (SIMD) instruction set. We do not study implementations based on the Neon instruction set for Arm-v7A or implementations based on the Scalable Vector Instructions SVE and SVE2 here – this is left for future work.

Returning to the nature of the PQC workloads themselves: Many of the remaining NISTPQC candidate schemes are based on so-called structured lattices, for which the central arithmetic operation is modular polynomial multiplication. A central implementation technique for such polynomial multiplication is the Number-Theoretic Transform (NTT), an integer-analog of the Fast Fourier Transform.

In this work, we explore the use of NTTs in implementing the NISTPQC structured lattice finalist candidates. It has always been a point of interest to determine the realms of applicability for various advanced multiplication techniques, and in addition to finding out how well NTTs do, we also compare them to other approaches towards polynomial multiplication, in particular Toom-Cook/Karatsuba.

Contributions. We exhibit NTT-based implementations of NISTPQC cyclotomic-ring lattice candidates for CPUs implementing the A-profile of the Arm architecture, leveraging the Neon vector extension. We optimized mainly for the common Cortex-A72 CPU (used e.g. in the Raspberry Pi 4), and somewhat for the Apple M1, a high-end desktop core.

We improve on old and discover new implementation techniques, including a Barrett-reduction based one-known-factor multiplication, which we show to be roughly equivalent to the Montgomery multiplication technique of [Sei18], but particularly suitable for Neon.

We also introduce the trick of "asymmetric base multiplication" which is applicable whenever we are caching incomplete NTT results (i.e., Kyber/Saber). Furthermore, we improve on the best-known two-unknown-factors multiplications and Barrett reduction sequences in the literature.

Code. Our code is available at https://github.com/neon-ntt/neon-ntt.

Related Work. The most recent work on lattice-based cryptography on the Arm-v8A architecture using the Neon vector extension is by Nguyen and Gaj [NG21] and Sanal et al. [SKS+21]. NTT is frequently used in the context of polynomial multiplications

[1] The Arm architecture has three profiles – application (A), real-time (R) and embedded (M) – and each Arm-based CPU implements a version of one of those profiles. The well-studied Cortex-M4 processor, for example, belongs to the M-profile.

Table 1: Kyber and Saber Parameter Sets

name	l	(d_1, d_2)	$\eta(s\|s')$	$\eta(e\|e'\|e")$	name	l	$T = 2^{e_T}$	η
`Kyber512`	2	$(10, 4)$	6	4	`LightSaber`	2	2^3	10
`Kyber768`	3	$(10, 4)$	4	4	`Saber`	3	2^4	8
`Kyber1024`	4	$(11, 5)$	4	4	`FireSaber`	4	2^6	6

which form the basis of almost all lattice-based PQC. Recently, NISTPQC third-round candidates have been implemented for the Arm Cortex-M3 and Cortex-M4 using NTTs. The most relevant works are Botros et al. [BKS19] and Alkim et al. [ABCG20] on Kyber, Greconici et al. [GKS21] on Dilithium, Chung et al. [CHK+21] on Saber and NTRU, and Alkim et al. [ACC+21] on NTRU Prime.

Structure of the paper. This paper is structured as follows: Section 2 introduces the schemes we implement, the Arm-v8A microarchitecture, and the mathematical background on NTTs. In Section 3, we present more mathematics in the form of reductions for the implementations used in this paper. In Section 4, we go through the implementation details of the different schemes. In Section 5, we show performance numbers and conclude.

2 Preliminaries

2.1 Kyber

Kyber [ABD+20b] is a NISTPQC finalist candidate lattice-based key encapsulation mechanism based on the Module Learning With Errors (M-LWE) problem. The module is of dimension $\ell \times \ell$ over the ring $R_q = \mathbb{F}_q[x]/\langle x^n + 1\rangle$, with $q = 3329$ and $n = 256$. The Kyber KEM is derived *a la* [HHK17] from a CPA-secure Public-Key Encryption (PKE) .

Please refer to algorithmic descriptions of the PKE in in Appendix A. In the CPA-secure key generation and encryption the rate-determining operations are $(\ell \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplications $A^T \cdot s$ and As' (`MatrixVectorMul`). In decryption, it is the $\ell \times 1$ inner product of polynomials $b'^T \cdot s$ (`InnerProd`). Note that [ABD+20b] specifies that *we do all multiplications via incomplete NTT, and NTT results are in bit-reversed order*. The public matrix A is sampled in (incomplete) NTT domain by expanding a seed using the extendable-output function (XOF) `SHAKE128`. There is 1 matrix-to-vector polynomial multiplication and 0, 1, and 2 inner products of polynomials in each of key generation, encapsulation, and decapsulation respectively.

Parameters. The module dimension ℓ, the rounding parameters (d_1, d_2), and the width of the centered binomial distribution η vary according to the parameter sets `Kyber-512`, `-768`, and `-1024` (targeting the NIST security levels 1, 3, and 5 respectively, cf. Table 1).

2.2 Saber

Saber [DKRV20] is a NISTPQC finalist candidate lattice-based key encapsulation mechanism based on the Module Learning With Rounding (M-LWR) problem. The module is of dimension $\ell \times \ell$ over the ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, with $q = 2^{13}$ and $n = 256$. Similar to Kyber, the Saber KEM is built on top of a CPA-secure PKE via the CCA-transform *a la* [HHK17]. For algorithmic descriptions of the Saber PKE see Appendix B.

In CPA-secure key generation and encryption, the rate-determining operations are $(\ell \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplications $A^T \cdot s$ and As' (`MatrixVectorMul`). In decryption, it is the $\ell \times 1$ inner product of polynomials $b'^T \cdot s$ (`InnerProd`). There is 1 `MatrixVectorMul` in key generation; 1 `MatrixVectorMul` + 1 `InnerProd` in encapsulation;

and 1 `MatrixVectorMul` + 2 `InnerProd` in decapsulation, as decapsulation needs a full re-encryption.

Note that Saber's base ring $\mathbb{Z}_{2^{13}}$ is not a field and thus not directly amenable for application of the NTT. Accordingly, the specification samples the public matrix A in polynomial domain.

Parameters. The module dimension l, the rounding parameter T, and the secret distribution parameter η vary according to the parameter sets `Lightsaber`, `Saber`, and `Firesaber` (respectively targeting the NIST security levels 1, 3, and 5, cf. Table 1).

2.3 Dilithium

Dilithium [ABD+20a] is a NISTPQC [NIS] finalist digital signature scheme based on the M-SIS (Module Small Integer Solutions) and M-LWE problems. The module is of dimension $k \times \ell$ over the ring $R_q = \mathbb{F}_q[x]/\langle x^{256}+1\rangle$ with $q = 2^{23} - 2^{13} + 1 = 8380417$.

For algorithmic descriptions see Appendix C. The core operation of key generation, signature generation, and signature verification is the $(k \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplications As_1, Ay, and Az (resp.) (`MatrixVectorMul`). In signature generation, this operation is particularly time-consuming since it is executed in a loop. Similar to Kyber, Dilithium builds a (complete) NTT into the specification, i.e., A is sampled in NTT domain using the XOF `SHAKE256`.

Table 2: Dilithium parameter sets

Name	NIST level	(k,ℓ)	η	β	ω	$\lvert pk\rvert$	$\lvert sig\rvert$	exp. iterations
`Dilithium2`	2	$(4,4)$	2	78	80	1312	2420	4.25
`Dilithium3`	3	$(6,5)$	4	196	55	1952	3293	5.1
`Dilithium4`	5	$(8,7)$	2	120	75	2592	4595	3.85

Parameters. Dilithium has parameter sets `Dilithium2`, `Dilithium3`, and `Dilithium5` targeting the corresponding NIST security levels. For all parameter sets $\gamma_1 = (q-1)/16 = 523776$ and $\gamma_2 = \gamma_1/2 = 261888$. For each parameter set, the remaining parameters are given in Table 2. The parameters consist of the matrix dimension (k,ℓ), the sampling bounds of the secret η, and the rejection thresholds β and ω.

2.4 Modular arithmetic

In this section, we establish some basic facts and notation about modular arithmetic which we will use throughout the paper.

Notation. We will denote $[\![\ ]\!]$ any "integer approximation", by which we mean $[\![\ ]\!] : \mathbb{Q} \to \mathbb{Z}$ with $z - [\![z]\!] \leq 1$ for all $z \in \mathbb{Z}$. Examples include the rounding functions $\lfloor\ \rfloor, \lceil\ \rceil, \lfloor\ \rceil$, but also e.g. the $2\mathbb{Z}$-valued $\lfloor z \rceil_2 := 2\left\lfloor \frac{z}{2} \right\rceil$, which will be of special interest later. Note that we do not require $[\![z]\!] = z$ for all $z \in \mathbb{Z}$, and it does in fact not hold for $\lfloor\ \rceil_2$.

Let $N \in \mathbb{N}$, henceforth called the *modulus*; in our application we will have either $N = 2^k$ or $N = q$ an odd prime. We denote $\mathbb{Z}_N := \mathbb{Z}/N\mathbb{Z}$ (or $\mathbb{F}_q$ if $N = q$ is a prime) the quotient ring of $\mathbb{Z}$ by the equivalence relation $x \equiv_N y$, identifying two integers which leave the same residue after division by N. For $z \in \mathbb{Z}$, we denote $\underline{z} \in \mathbb{Z}_N$ its residue class; in the case of a 2-power N, we use the notation $\overline{z}$ instead.

Both the signed interval $S_N := \left\{-\left\lfloor \frac{N}{2} \right\rfloor, -\left\lfloor \frac{N}{2} \right\rfloor + 1, \ldots, \left\lfloor \frac{N-1}{2} \right\rfloor - 1, \left\lfloor \frac{N-1}{2} \right\rfloor\right\}$ and the unsigned interval $U_N := \{0, 1, \ldots, N-1\}$ are fundamental domains for $\equiv_N$; for $z \in \mathbb{Z}$, we denote $z \bmod^{\pm} N \in S_N$ and $z \bmod^{+} N \in U_N$ the unique representatives of z in S_n and

U_N and call them *canonical signed representative* and *canonical unsigned representative*, respectively. For example, $-15 \bmod^{\pm} 13 = -2$ and $-15 \bmod^{+} 13 = 11$.

The canonical signed and unsigned representatives are related to the integer approximations $\lfloor \ \rfloor$ and $\lceil \ \rceil$ via $z \bmod^{\pm} N = z - N \left\lfloor \frac{z}{N} \right\rceil$ and $z \bmod^{+} N = z - N \left\lfloor \frac{z}{N} \right\rfloor$. More generally, we can define $z \bmod^{[\![\ ]\!]} N := z - N \left[\!\left[\frac{z}{N} \right]\!\right]$ for any integer approximation $[\![\ ]\!]$.

Example 1. We consider $z \bmod^{\lfloor \rceil_2} N$ associated with $\lfloor z \rceil_2 = 2 \left\lfloor \frac{z}{2} \right\rceil$. In this case, $z \bmod^{\lfloor \rceil_2} N$ is a representative of z modulo N within $\{-N, \ldots, N\}$ of the same parity as z. For example, if z is even, we have $z \bmod^{\lfloor \rceil_2} N = z \bmod^{\pm} N$ if $z \bmod^{\pm} N$ is even, and $z \bmod^{\lfloor \rceil_2} N = z \bmod^{\pm} N - \mathrm{sign}(z \bmod^{\pm} N)N$ otherwise.

If $[\![\ ]\!]$ satisfies $[\![z+1]\!] = [\![z]\!] + 1$, then $z \bmod^{[\![\ ]\!]} N$ descends to a function $\mathbb{Z}_N \to \mathbb{Z}$; this is the case for $\lfloor \ \rfloor$ and $\lfloor \ \rceil$, and we will also use $\underline{z} \bmod^{\pm} N$ and $\underline{z} \bmod^{+} N$ for $\underline{z} \in \mathbb{Z}_N$.

We will need the following fact:

Fact 1. Let $R = 2^n$ and $\overline{x} \in \mathbb{Z}_R$. Then the following holds:

1. If $x \neq \overline{2^{n-1}} \in \mathbb{Z}_R$, then $(-x) \bmod^{\pm} R = -(x \bmod^{\pm} R)$.
2. If $x = \overline{2^{n-1}} \in \mathbb{Z}_R$, then $(-x) \bmod^{\pm} R = -(x \bmod^{\pm} R) - R$.

Beyond those canonical representatives, we will also be interested in non-unique representatives from intervals $\{-N', -N'+1, \ldots, N'-1, N'\}$, where N' is suitably bounded with respect to N. By "algorithms for arithmetic modulo N" we mean algorithms which compute addition and multiplication in $\mathbb{Z}_N$ in terms of "small" representatives within those sets. The primary objective in designing such algorithms is the avoidance of generic integer division; a secondary goal is to avoid unnecessary reductions. Our main techniques are Montgomery and Barrett reduction, which we recall briefly now.

2.4.1 Barrett reduction

We have $z \bmod^{\pm} N = z - N \cdot \left\lfloor \frac{z}{N} \right\rceil$. The idea behind Barrett reduction [Bar86] is to approximate $\frac{z}{N} = z\frac{R}{N}/R \approx z \left[\!\left[\frac{R}{N} \right]\!\right] / R$, so $z \bmod^{\pm} N \approx z - N \cdot \left\lfloor z \left[\!\left[\frac{R}{N} \right]\!\right] / R \right\rceil$, where $[\![-]\!]$ is a choice of integer approximation and $R = 2^n > N$ is fixed. Since $\left[\!\left[\frac{R}{N} \right]\!\right]$ can be precomputed, this provides an approximation to $z \bmod^{\pm} N$ relying solely on division by $R = 2^n$, which common hardware can realize as a bitshift. We denote the resulting approximation to $z \bmod^{\pm} N$ by $\mathbf{bar}_N^{[\![\ ]\!]}(z)$, or $\mathbf{bar}_N^{\pm}(z)$ if $[\![\ ]\!] = \lfloor \ \rceil$. See summary in Algorithm 1.

Algorithm 1 Barrett reduction

Require: N modulus, $R = 2^n > N$
Require: $\left[\!\left[\frac{R}{N} \right]\!\right] \in \mathbb{Z}$ approx. of $\frac{R}{N}$.
Require: $z \in \mathbb{Z}$, $|z| \leq R$, to be reduced
Ensure: $z' \equiv z \pmod N$, $|z'| < \frac{3}{2}N$.
1: $t \leftarrow \left\lfloor z \left[\!\left[\frac{R}{N} \right]\!\right] / R \right\rceil$
2: $c \leftarrow Nt$
3: $\mathbf{bar}_N^{[\![\ ]\!]}(z) := z' \leftarrow z - c$

Algorithm 2 Montgomery reduction

Require: N odd, $R = 2^n > N$
Require: $T \in \mathbb{Z}$, $T \equiv \pm N^{-1} \pmod R$
Require: $z \in \mathbb{Z}$, to be reduced
Ensure: $z' \equiv zR^{-1} \pmod N$, $|z'| \leq \frac{|z|}{2^n} + \frac{N}{2}$.
1: $k \leftarrow zT \bmod^{\pm} R$
2: $c \leftarrow kN$
3: $\mathbf{mont}_N^{\pm}(z) := z' \leftarrow \frac{z \mp c}{R}$

Fact 2. For any choice of integer approximation $[\![\ ]\!]$, we have

$$|z \bmod^{\pm} N - \mathbf{bar}_N^{[\![\ ]\!]}(z)| \leq N \left\lceil \frac{|z|}{R} \right\rceil .$$

In particular, if $|z| \leq R$, we have $|\mathbf{bar}_N^{[\![\ ]\!]}(z)| < \frac{3}{2}N$.

Implementation considerations. In the context of Algorithm 1, the result has absolute value $|c-t| < \frac{3}{2}N$. If $N < \frac{R}{3}$, $c-t$ is a signed canonical representative modulo $R = 2^n$, and thus uniquely determined by its residue modulo 2^n. This observation allows us to perform steps 2 and 3 in Algorithm 1 in single-width arithmetic, leading to the presentation of Barrett reduction in terms of single-width operations alone (Algorithm 3).

Algorithm 3 Barrett reduction, implementation-view

Require: N modulus, $R = 2^n$ s.t. $N < \frac{R}{3}$.
Require: $\left[\!\left[\frac{R}{N}\right]\!\right] \in \mathbb{Z}$ precomputed integer approximation of $\frac{R}{N}$.
Require: $z \in \mathbb{Z}$ representative mod N with $|z| \leq R$, to be reduced
Ensure: $\mathbf{bar}_N^{[\![\,]\!]}(z)$ representative of z with $|\mathbf{bar}_N^{[\![\,]\!]}(z)| < \frac{3}{2}N < \frac{R}{2}$.

1: $t \leftarrow \left\lfloor z \left[\!\left[\frac{R}{N}\right]\!\right] / R \right\rceil$ ▷ Signed multiply-high with rounding
2: $\overline{c} \leftarrow \overline{N} \cdot \overline{t}$ ▷ Unsigned single-width multiply
3: $\overline{z} \leftarrow \overline{z} - \overline{c}$ ▷ Unsigned single-width subtract
4: $\mathbf{bar}_N^{[\![\,]\!]}(z) \leftarrow \overline{z} \bmod^{\pm} R$ ▷ Canonical signed representative

2.4.2 Montgomery reduction

Like Barrett reduction, Montgomery reduction [Mon85] provides a way to trade expensive division by N for cheap division by a power of two.

The idea is simple: Assume we would like to reduce $z \in \mathbb{Z}$ with respect to the odd modulus N, and that z happens to be a multiple of $R = 2^n \in \mathbb{Z}$. Then the (cheaply computed) integer division $\frac{z}{R}$ is a representative of $\underline{z} \cdot \underline{R}^{-1} \in \mathbb{Z}_N$ which is shorter than z by r bits. In other words, if we accept "twisting" our target residue $\underline{z}$ by some factor $\underline{R}^{-1} \in \mathbb{Z}_N$, we can shorten its representative.[2]

If the given representative $z \in \mathbb{Z}$ is not evenly divisible by R, we can turn it into one through the following correction step: We need to find some k such that $z - kN$ is divisible by R, that is, $\overline{z} = \overline{kN}$ in $\mathbb{Z}_R$. Since N and R are coprime, this is achieved by taking k to be a small representative of $\overline{z} \cdot \overline{N}^{-1}$ in $\mathbb{Z}_R$; we will always choose $\overline{z}\overline{N}^{-1} \bmod^{\pm} R$.

The so obtained "Montgomery reduction" is summarized in Algorithm 2 and denoted $\mathbf{mont}_N^+(z)$ – with R always being implicit. Algorithm 2 also shows a close variant $\mathbf{mont}_N^-(z)$ which implements the correction step through an addition $z + kN$ instead of a subtraction: In this case, we want k to be a small representative of $-\overline{z} \cdot \overline{N}^{-1}$ in $\mathbb{Z}_R$, for which we choose $-\overline{z}\overline{N}^{-1} \bmod^{\pm} R$. We will call this variant "negative" Montgomery reduction. Their relation is as follows:

[2] Note that $\mathbb{Z}_N$ is *unordered*. We only talk about smallness of *representatives* of elements of $\mathbb{Z}_N$, not the elements of $\mathbb{Z}_N$ themselves. In particular, one should not consider the multiplication by $\underline{R}^{-1}$ as some form of scaling, but instead as an abstract permutation of $\mathbb{Z}_N$.

Algorithm 4 Montgomery reduction, implementation-view

Require: N odd modulus, $R = 2^n > N$, $T \in \mathbb{Z}$ representative of $\overline{N}^{-1} \in \mathbb{Z}_R$
Require: $z \in \mathbb{Z}$ representative mod N, to be reduced
Ensure: $\mathbf{mont}_N^+(z)$ representative of $\underline{zR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^+(z)| \leq \frac{|z|}{2^n} + \frac{N}{2}$.

1: $\overline{k} \leftarrow \overline{z}\overline{T}$ ▷ Unsigned single-width multiply
2: $k \leftarrow \overline{k} \bmod^{\pm} R$ ▷ Canonical signed representative
3: $c \leftarrow \left\lfloor \frac{kN}{R} \right\rfloor$ ▷ Signed multiply-high with truncation
4: $z_{\text{high}} = \left\lfloor \frac{z}{R} \right\rfloor$ ▷ High part extraction
5: $\mathbf{mont}_N^+(z) \leftarrow z_{\text{high}} - c$ ▷ Signed subtraction/addition

Algorithm 5 Montgomery multiplication, implementation-view

Require: N odd modulus, $R = 2^n > N$
Require: $a, b \in \mathbb{Z}$ representative mod N with $|a|, |b| < R$.
Require: Precomputed $\overline{b_{\text{tw}}} = \overline{b} \cdot \overline{N}^{-1} \in \mathbb{Z}_R$.
Ensure: $\mathbf{mont}_N^+(a, b)$ representative of $\underline{abR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^+(a, b)| \leq \frac{|a||b|}{2^n} + \frac{N}{2}$.

1: $\overline{k} \leftarrow \overline{a} \cdot \overline{b_{\text{tw}}}$ ▷ Unsigned single-width multiply
2: $k \leftarrow \overline{k} \bmod^{\pm} R$ ▷ Canonical signed representative
3: $c \leftarrow \left\lfloor \frac{kN}{R} \right\rfloor$ ▷ Signed multiply-high with truncation
4: $z_{\text{high}} = \left\lfloor \frac{ab}{R} \right\rfloor$ ▷ Multiply-high
5: $\mathbf{mont}_N^+(a, b) \leftarrow z_{\text{high}} - c$ ▷ Signed subtraction

Fact 3. Assume the context of Algorithm 2. If $\overline{z} \neq \overline{2^{n-1}} \in \mathbb{Z}_R$, then $\mathbf{mont}_N^+(z) = \mathbf{mont}_N^-(z)$. Otherwise, $\mathbf{mont}_N^-(z) = \mathbf{mont}_N^+(z) - N$.

Proof. This follows from Fact 1 and the fact that $\overline{2^{n-1}} \cdot \overline{x} = \overline{2^{n-1}}$ for odd x. □

Remark 1. The exceptional case $\overline{z} = \overline{2^{n-1}}$ can be made explicit: $\mathbf{mont}_N^{\pm}(z) = \frac{z \pm 2^{n-1}N}{2^n}$.

Implementation considerations. We briefly summarize known implementation aspects of Montgomery reduction.

Firstly, in the context of Algorithm 2 for $\mathbf{mont}_N^+(z)$, $z - c$ is divisible by R, and hence $\frac{z-c}{R} = \left\lfloor \frac{z}{R} \right\rfloor - \left\lfloor \frac{c}{R} \right\rfloor$. This allows to rewrite the algorithm in terms of single-width operations, as detailed in Algorithm 4. This description does *not* apply to $\mathbf{mont}_N^-(z)$: This is because $\frac{z+c}{R}$ will usually introduce a carry-in from low-half to high-half as part of $z + c$, and this carry is lost in $\left\lfloor \frac{z}{R} \right\rfloor + \left\lfloor \frac{c}{R} \right\rfloor$. We will revisit this later.

Secondly, consider the use of Montgomery reduction for modular multiplication by constants: If $z = ab$ for single-width values $a, b \in \mathbb{Z}$, and b is known in advance, then $\overline{b} \cdot \overline{N}^{-1} \in \mathbb{Z}_R$ can be precomputed and leads to Algorithm 5, the Montgomery reduction for products (a.k.a., "Montgomery Multiplication") with one known factor.

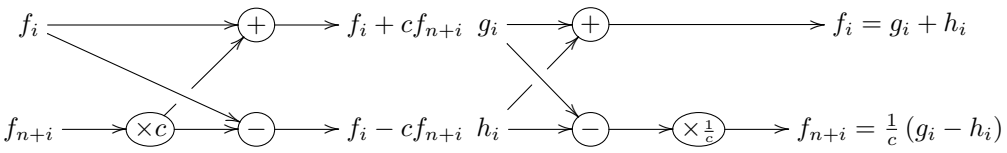
Finally, for microarchitectures with length-doubling ("long") and non-doubling products, the optimal implementation varies depending on what instructions are available. For the Neon instruction set, please refer to Section 3.2.5 and Algorithm 14.

2.5 Fixed point arithmetic

Let $n \in \{16, 32\}$. The fixed-point interpretation of a single-width signed integer $a \in \{-2^{n-1}, -2^{n-1} + 1, \ldots, 2^{n-1} - 1\}$ is the single-precision fractional value $\frac{a}{2^{n-1}} \in [-1, +1)$. Likewise, the fixed-point interpretation of a double-width signed integer a is the double-precision fractional value $\frac{a}{2^{2n-1}}$.

In this interpretation, the double-precision product of two single-precision fractional values corresponds to a *doubling long multiplication* on the corresponding signed integers: $\frac{a}{2^{n-1}} \cdot \frac{b}{2^{n-1}} = \frac{ab}{2^{2n-2}} = \frac{2ab}{2^{2n-1}}$. For this reason, doubling long multiplications are found in some ISAs supporting fixed point arithmetic: Neon offers `SQDMULL`, Helium offers `VQDMULL`, and SVE 2 offers `QDMULL`.

Similarly, single-precision approximations to the product of two single-precision fractional values correspond to a *doubling long multiplication returning high half* $\frac{a}{2^{n-1}} \cdot \frac{b}{2^{n-1}} = \frac{2ab}{2^{2n-1}} \approx \frac{[\![\frac{2ab}{2^n}]\!]}{2^{n-1}}$, with flexibility in terms of the choice of approximation $[\![-]\!]$; common choices are *truncation* via $\lfloor - \rfloor$ and *rounding* to the nearest integer $\lfloor - \rceil$. For this reason, ISAs supporting fixed point arithmetic offer instructions for doubling long multiplication returning high halves, often with separate variants for truncation and rounding: For example,



(a) Cooley–Tukey (CT) Butterfly (b) Gentleman–Sande (GS) Butterfly

Figure 1: The "Butterflies" of Fast Fourier Transforms

Neon offers `SQDMULH` (truncating) and `SQRDMULH` (rounding), Helium offers `VQ[R]DMULH`, and SVE2 offers `Q[R]DMULH`. We note that Intel's AVX2 includes `VPMULHRSW` but not a non-rounding variant.

Finally, ISAs supporting fixed-point arithmetic often offer multiply-accumulate-add or even multiply-accumulate-subtract variants for their single-precision fixed-point multiplications: Neon from ARMv8.1-A onwards offers `VQRDMLAH` and `VQRDMLSH`, Helium offers `VQ[R]DMLAH`, and SVE2 offers `QRDMLAH` and `QRDMLSH`. Interpreted as integer operations, those correspond to doubling long multiplications returning (rounded/truncated) high half with accumulate add/subtract.

2.6 Number Theoretic Transform

In Kyber, Saber, and Dilithium, we need arithmetic in $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^{256}+1\rangle$, a polynomial ring. Here $q = 3329 = 13{\cdot}2^8{+}1$ for Kyber, $q = 2^{13}$ for Saber, and $q = 2^{23}{-}2^{13}{+}1 = 8380417$ for Dilithium. More specifically, we need to do a matrix-to-vector multiplication and inner products. All three will be developed in parallel after we switch to a modulus q' for Saber [CHK+21] so that the results are the same as for a computation over $\mathbb{Z}[x]/\langle x^{256}+1\rangle$.

Let $\mathbb{F} = \mathbb{F}_q$ for Dilithium and Kyber and $\mathbb{F} = \mathbb{F}_{q'}$ for Saber. From the CRT (Chinese Remainder Theorem) we have the ring isomorphism (Figure 1):

$$\mathbb{F}[x]/\langle x^{2n} - c^2\rangle \cong \mathbb{F}[x]/\langle x^n - c\rangle \times \mathbb{F}[x]/\langle x^n + c\rangle;$$
$$\sum_{i=0}^{2n-1} f_i x^i \leftrightarrow \left(\sum_{i=0}^{n-1} (f_i + cf_{n+i})\, x^i, \sum_{i=0}^{n-1} (f_i - cf_{n+i})\, x^i\right).$$

To split a $2n$-long polynomial, we perform CT butterflies between n pairs of coefficients, each from one half of the polynomial. Everything can be done in-place, and the CT butterfly has an obvious inverse — the GS butterfly, except for a factor of 2. For even n and $\pm c$ with square roots in $\mathbb{F}$ we can recurse this process, each stage comprising the same number of butterflies. Let ζ be a principal root of order 2^k, so that $\zeta^{2^{k-1}} = -1$. An *(incomplete) NTT* is defined as the series of ring isomorphisms computed as follows:

$$\begin{aligned}
&\frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^k} - c^{2^k}\rangle} \cong \frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^{k-1}} - c^{2^{k-1}}\rangle} \times \frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^{k-1}} - \zeta^{2^{k-1}} c^{2^{k-1}}\rangle} \\
\cong\ & \frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^{k-2}} - c^{2^{k-2}}\rangle} \times \frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^{k-2}} - \zeta^{2^{k-1}} c^{2^{k-2}}\rangle} \times \\
& \frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^{k-2}} - \zeta^{2^{k-2}} c^{2^{k-2}}\rangle} \times \frac{\mathbb{F}[x]}{\langle x^{d\cdot 2^{k-2}} - \zeta^{3\cdot 2^{k-2}} c^{2^{k-2}}\rangle} \\
\cong\ & \cdots \cong \prod_{i=0}^{2^k-1} \frac{\mathbb{F}[x]}{\langle x^d - c\cdot\zeta^{\mathrm{br}_k(i)}\rangle}, \\
& \text{where } \mathrm{br}_k\left((b_{k-1}\cdots b_1 b_0)_2\right) := (b_0 b_1 \cdots b_{k-1})_2 \ \text{(“}k\text{-bit reversal”).}
\end{aligned}$$

Here $k = 7$, $d = 2$, $c^{128} = -1$ for Kyber, $k = 8$, $d = 1$, $c^{256} = -1$ for Dilithium, and $\zeta = c^2$ for both, as the NTT is specified. Saber is flexible and only requires $d \cdot 2^k = 256$ and $c^{2^{k-1}} = -1$. The image under the ring isomorphism of a polynomial f is called "the NTT" of f and denoted $NTT(f)$. A polynomial product fg in the ring is computed as $fg = NTT^{-1}(NTT(f) \circ NTT(g))$, where $\circ$ is the `base_mul` giving products in each of $\frac{\mathbb{F}[x]}{\left\langle x^d - c \cdot \zeta^{\mathrm{br}_k(i)} \right\rangle}$.

3 Modular Multiplication

In this section, we present improvements to Barrett reduction and Montgomery multiplication from a theoretical and implementation perspective: In Section 3.1, we present a relation between Barrett reduction and Montgomery reduction and expand it to a relation between Montgomery multiplication and a new variant of modular multiplication via Barrett reduction which we call "Barrett multiplication". We also introduce two new variants of Montgomery multiplication using fixed-point arithmetic. In Section 3.2, we study implementations on SIMD extensions, focusing on the Neon instruction set. Concretely, we find a 3-instruction sequence for SIMD modular multiplication by known constants, improving on the 4-instruction sequence introduced by [Sei18, LS19], and a 5-instruction sequence for modular multiplication with unknown inputs on the Neon instruction set, improving the sequence used in [NG21]. We also describe a 4-instruction sequence for two unknown inputs; it does impose a parity constraint on the input, however, which may or may not be satisfiable depending on the application context.

3.1 Theory

3.1.1 Barrett reduction vs. Montgomery reduction

In this section, we compare Barrett and Montgomery reduction for single-width values.

Proposition 1. Let N be odd, $R = 2^n > N$ and $z \in \mathbb{Z}$. Then Barrett reduction and negative Montgomery reduction satisfy the following relation:

$$\mathbf{bar}_N^{\pm}(z) = \mathbf{mont}_N^{-}(z(R \bmod^{\pm} N)). \tag{1}$$

More generally, for an arbitrary approximation $[\![\ ]\!]$, we have

$$\mathbf{bar}_N^{[\![\ ]\!]}(z) = \mathbf{mont}_N^{-}(z(R \bmod^{[\![\ ]\!]} N)) \tag{2}$$

Remark 2. Since $\mathbf{mont}_N^{-}(x)$ computes a representative of $\underline{x} \cdot \underline{R}^{-1}$ in $\mathbb{Z}_N$, the factor $R \bmod^{\pm} N$ in (4) does not come as a surprise. It also explains why $\mathbf{bar}_N^{\pm}(z)$ and $\mathbf{mont}_N^{-}(zR \bmod^{\pm} N)$ can only differ by a multiple of N. The value in Proposition 1 is in working with explicit representatives.

Remark 3. It follows from Fact 3 that Proposition 1 also holds for $\mathbf{mont}_N^{+}(z)$ except for the case $\overline{z} \cdot \overline{R \bmod^{\pm} N} = \overline{2^{n-1}}$, in which case $\mathbf{bar}_N^{\pm}(z) = \mathbf{mont}_N^{+}(z(R \bmod^{\pm} N)) - N$.

Lemma 1. In the context of Proposition 1, $\overline{[\![\frac{R}{N}]\!]} = -\overline{R \bmod^{[\![\ ]\!]} N} \cdot \overline{N}^{-1}$ in $\mathbb{Z}_R$, hence

$$[\![\tfrac{R}{N}]\!] \bmod^{\pm} R = \left(-(R \bmod^{[\![\ ]\!]} N) \cdot (\overline{N}^{-1} \bmod^{\pm} R)\right) \bmod^{\pm} R. \tag{3}$$

Proof of Lemma 1. We have $N\,[\![\frac{R}{N}]\!] = R - R \bmod^{[\![\ ]\!]} N$ in $\mathbb{Z}$. Passing to $\mathbb{Z}_R$ and multiplying by $\overline{N}^{-1}$ proves the claim. Equation 3 follows by applying $\_ \bmod^{\pm} N : \mathbb{Z}_R \to \mathbb{Z}$. □

Proof of Proposition 1. The proof is mostly careful unraveling of definitions.

$$\begin{aligned}\mathbf{bar}_N^{[\![\,]\!]}(z) &\stackrel{\text{def}}{=} z - N \cdot \left\lfloor \frac{z \left[\!\left[\frac{R}{N}\right]\!\right]}{R} \right\rceil \stackrel{\text{def}}{=} z - N \frac{z \left[\!\left[\frac{R}{N}\right]\!\right] - (z \left[\!\left[\frac{R}{N}\right]\!\right]) \bmod^{\pm} R}{R} \\ &\stackrel{\text{Lemma 1}}{=} z - \frac{zN \left[\!\left[\frac{R}{N}\right]\!\right] - N \left[\left(-z(R \bmod^{[\![\,]\!]} N)(\overline{N}^{-1} \bmod^{\pm} R)\right) \bmod^{\pm} R\right]}{R}\end{aligned}$$

Note that the right hand side of this expression already matches the correction term occurring in $\mathbf{mont}_R^-(z(R \bmod^{[\![\,]\!]} N))$. Next, we replace $\left[\!\left[\frac{R}{N}\right]\!\right] = \frac{R-(R \bmod^{[\![\,]\!]} N)}{N}$ and obtain

$$\begin{aligned}\ldots &= z - \frac{zN \frac{R-(R \bmod^{[\![\,]\!]} N)}{N} - N \left[\left(-z(R \bmod^{[\![\,]\!]} N)(\overline{N}^{-1} \bmod^{\pm} R)\right) \bmod^{\pm} R\right]}{R} \\ &= \frac{z(R \bmod^{[\![\,]\!]} N) + N \left[-z(R \bmod^{[\![\,]\!]} N)(\overline{N}^{-1} \bmod^{\pm} R) \bmod^{\pm} R\right]}{R} \\ &= \mathbf{mont}_N^-(z(R \bmod^{[\![\,]\!]} N)).\end{aligned}$$

□

Corollary 1. In the context of Proposition 1,

$$|\mathbf{bar}_N^{[\![\,]\!]}(z)| \leq \frac{|z| \cdot |R \bmod^{[\![\,]\!]} N|}{R} + \frac{N}{2}$$

In particular, if $|z| < N$, and $N < \frac{R}{2}$, then $|\mathbf{bar}_N^{[\![\,]\!]}(z)| < N$.

Proof. The first part follows from Proposition 1 and the bound for Montgomery reduction. For the second, recall $|\_ \bmod^{[\![\,]\!]} N| \leq N$. □

3.1.2 Barrett multiplication

The relation between Barrett reduction and Montgomery reduction exhibited in the previous section raises the question of whether there is a natural extension to Montgomery multiplication, and what the analog on the Barrett side is. The answer turns out to be a variant of Barrett reduction for multiplication with known constants, which we describe in this section. While very natural in retrospect, the definitions and results are novel to the best of our knowledge.

Recall (Section 2.4.1) the idea of Barrett reduction: To reduce $z \in \mathbb{Z}$, we approximate

$$z \bmod^{\pm} N = z - N \cdot \left\lfloor \frac{z}{N} \right\rceil = z - N \cdot \left\lfloor \frac{z\frac{R}{N}}{R} \right\rceil \approx z - N \cdot \left\lfloor \frac{z \left[\!\left[\frac{R}{N}\right]\!\right]}{R} \right\rceil,$$

replacing $z\frac{R}{N}$ by the precomputed $z \left[\!\left[\frac{R}{N}\right]\!\right]$. If $z = ab$ for $a, b \in \mathbb{Z}$ and b a known constant, we can improve the quality of the approximation by pulling b into the approximation, approximating $ab\frac{R}{N} \approx a \left[\!\left[\frac{bR}{N}\right]\!\right]$, where $\left[\!\left[\frac{bR}{N}\right]\!\right]$ can be precomputed. We call this the "Barrett multiplication" of a, b and denote it $\mathbf{bar}_N^{[\![\,]\!]}(a, b)$, or $\mathbf{bar}_N^{\pm}(a, b)$ if $[\![\,]\!] = \lfloor\,\rceil$. As before, the choice of R is implicit in the notation and to be understood from the context. We describe Barrett multiplication in Algorithm 6.

Algorithm 7 Montgomery multiplication via doubling, abtract view

Require: N odd modulus, $R = 2^n > N$
Require: $a, b \in \mathbb{Z}$ representative mod N with $|a|, |b| < R$.
Require: Precomputed $T = \overline{N}^{-1} \in \mathbb{Z}_R$.
Ensure: $\mathbf{mont}_N^+(a,b)$ representative of $\underline{abR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^+(a,b)| \leq \frac{|a||b|}{2^n} + \frac{N}{2}$.
1: $z \leftarrow \left\lfloor \frac{2ab}{R} \right\rfloor$
2: $k \leftarrow abT \bmod^{\pm} R$
3: $c \leftarrow \left\lfloor \frac{2kN}{R} \right\rfloor$
4: $\mathbf{mont}_N^{\pm}(z) \leftarrow \frac{z-c}{2}$

Algorithm 6 Barrett multiplication, abstract view

Require: N modulus, $R = 2^n > N$
Require: $b \in \mathbb{Z}$, $\left[\!\left[\frac{bR}{N}\right]\!\right] \in \mathbb{Z}$ precomputed integer approximation of $\frac{bR}{N}$.
Require: $a \in \mathbb{Z}$ representative mod N with $|a| < R$, to be reduced
Ensure: $\mathbf{bar}_N^{[\![\,]\!]}(a,b)$ representative of ab with $|\mathbf{bar}_N^{[\![\,]\!]}(a,b)| < N$.
1: $z \leftarrow ab$
2: $t \leftarrow \left\lfloor \frac{a\left[\!\left[\frac{bR}{N}\right]\!\right]}{R} \right\rfloor$
3: $c \leftarrow Nt$
4: $\mathbf{bar}_N^{[\![\,]\!]}(a,b) \leftarrow z - c$

We now obtain the desired analog of Proposition 1 for Barrett multiplication:

Proposition 2. Let N be odd, $R = 2^n > N$ and $a, b \in \mathbb{Z}$. Then Barrett multiplication and negative Montgomery multiplication satisfy the following relation:

$$\mathbf{bar}_N^{\pm}(a,b) = \mathbf{mont}_N^{-}(a, bR \bmod^{\pm} N). \tag{4}$$

More generally, for an arbitrary approximation $[\![\,]\!]$, we have

$$\mathbf{bar}_N^{[\![\,]\!]}(a,b) = \mathbf{mont}_N^{-}(a, bR \bmod^{[\![\,]\!]} N). \tag{5}$$

Proof. Replace $R \bmod^{[\![\,]\!]} N$ by $bR \bmod^{[\![\,]\!]} N$ in the proof of Proposition 1. □

Corollary 2. In the context of Proposition 2,

$$|\mathbf{bar}_N^{[\![\,]\!]}(a,b)| \leq \frac{a(bR \bmod^{[\![\,]\!]} N)}{R} + \frac{N}{2}$$

In particular, for $|a| < N$ and $N < \frac{R}{2}$ we have $|\mathbf{bar}_N^{[\![\,]\!]}(a,b)| < N$.

3.1.3 Montgomery multiplication via doubling

Because of the doubling inherent in many fixed point instructions, the variant of Montgomery multiplication described in Algorithm 7 will be useful.

Proposition 3. *Algorithm 7 is correct.*

Proof. By construction, $ab - kN$ is divisible by R. Hence, so is $2ab - 2kN$, and we get $\left\lfloor \frac{2ab}{R} \right\rfloor - \left\lfloor \frac{2kN}{R} \right\rfloor = \frac{2(ab-kN)}{R} = 2\left(\left\lfloor \frac{ab}{R} \right\rfloor - \left\lfloor \frac{kN}{R} \right\rfloor\right)$. Algorithm 7 thus yields the same result as Montgomery multiplication. □

Algorithm 8 Montgomery multiplication via rounding, abtract view

Require: N odd modulus, $R = 2^n > N$
Require: $a, b \in \mathbb{Z}$ representative mod N with $|a|, |b| < \frac{R}{2}$, s.t. a or b is odd.
Require: Precomputed $T = -\overline{N}^{-1} \in \mathbb{Z}_R$.
Ensure: $\mathbf{mont}_N^+(a, b)$ representative of $\underline{abR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^+(a, b)| \leq \frac{|a||b|}{2^n} + \frac{N}{2}$.
1: $z \leftarrow \left\lfloor \frac{2ab}{R} \right\rceil$
2: $k \leftarrow abT \bmod^{\pm} R$
3: $c \leftarrow \left\lfloor \frac{2kN}{R} \right\rceil$
4: $\mathbf{mont}_N^{\pm}(z) \leftarrow \frac{z+c}{2}$

3.1.4 Montgomery multiplication via rounding

As first pointed out by Seiler in [Sei18] and mentioned in Section 2.4.2, single-width Montgomery multiplication Algorithm 5 breaks when trying to use addition instead of subtraction in the final step, because of a carry-in from low half to high half. Algorithm 8 presents a remedy using rounding. Note the assumption that one of the inputs is odd.

Proposition 4. *Algorithm 8 is correct.*

Proof. Since we use $-\overline{N}^{-1} \in \mathbb{Z}_R$ instead of $\overline{N}^{-1} \in \mathbb{Z}_R$, we have $\overline{ab} = -\overline{kN}$ in $\mathbb{Z}_R$. Assuming $\overline{ab} \neq 2^{n-1}$, Fact 1 therefore implies $ab \bmod^{\pm} R = -(kN \bmod^{\pm} R)$, and hence

$$\left\lfloor \frac{2ab}{R} \right\rceil + \left\lfloor \frac{2kN}{R} \right\rceil = \frac{2ab - (ab \bmod^{\pm} R) + 2kN - (kN \bmod^{\pm} R)}{R} = \frac{2(ab - kN)}{R}.$$

The result of Algorithm 8 therefore equals that of ordinary Montgomery multiplication.

It remains to be justified why $\overline{ab} \neq 2^{n-1}$, which involves the assumption that either a or b are odd: If, say, b is odd, then $\overline{ab} = 2^{n-1}$ implies $\overline{a} = 2^{n-1}$, hence $|a| \leq 2^{n-1} = \frac{R}{2}$, contradicting the assumption that $|a|, |b| < \frac{R}{2}$. □

3.2 Implementation

In this section, we look at Barrett and Montgomery multiplication from an implementation perspective, focusing on the Arm-v8A version of the Neon SIMD instructions.

3.2.1 Barrett multiplication in 3-instructions

Analogously to Algorithm 3, Barrett multiplication can be described in terms of single-width operations. The details are spelled out in Algorithm 9. We see that Barrett multiplication can be expressed in terms of 3 single-width operations: 1× unsigned multiply-low, 1× unsigned multiply-low-accumulate, 1× multiply-high-with-rounding.

Algorithm 9 Barrett multiplication

Require: N odd, $R = 2^n$ s.t. $N < \frac{R}{3}$.
Require: $b \in \mathbb{Z}$, $\left[\!\left[\frac{bR}{N}\right]\!\right] \in \mathbb{Z}$ approx. $\frac{bR}{N}$.
Require: $a \in \mathbb{Z}$, $|a| \leq R$, to be multiplied
Ensure: $z \equiv ab \pmod N$, $|z| < \frac{3}{2}N < \frac{R}{2}$.
1: $\overline{z} \leftarrow \overline{a} \cdot \overline{b}$
2: $t \leftarrow \left\lfloor a \left[\!\left[\frac{bR}{N}\right]\!\right] / R \right\rceil$
3: $\overline{z} \leftarrow \overline{z} + \overline{-N} \cdot \overline{t}$
4: $\mathbf{bar}_N^{[\![\,]\!]}(a, b) = z \leftarrow \overline{z} \bmod^{\pm} R$

Algorithm 10 Barrett multiplication in Neon

Require: N odd, $R = 2^n$ s.t. $N < \frac{R}{3}$.
Require: $b \in \mathbb{Z}$, $\frac{\left\lfloor \frac{bR}{N} \right\rceil_2}{2}$ precomputed
Require: $a \in \mathbb{Z}$, $|a| \leq R$, to be multiplied
Ensure: $z \equiv ab \pmod N$, $|z| < \frac{3}{2}N < \frac{R}{2}$.
1: `mul` z, $\overline{a}$, $\overline{b}$
2: `sqrdmulh` t, a, $\frac{\left\lfloor \frac{bR}{N} \right\rceil_2}{2}$
3: `mls` z, t, $\overline{N}$

In Neon, unsigned multiply-low and multiply-low-accumulate are implemented via `MUL` and `MLA`, respectively. The multiply-high-with-rounding operations $(a, b) \mapsto \left\lfloor \frac{ab}{2^n} \right\rceil$ does not have an exact match in Neon, but as explained in Section 2.5, there is `SQRDMULH` which computes $(a, b) \mapsto \left\lfloor \frac{2ab}{2^n} \right\rceil$ instead. We work around this difference by choosing the even integer approximation $\lfloor \ \rceil_2$ as the basis for Barrett multiplication: In this case, we have $\left\lfloor \frac{a\lfloor bR/N \rceil_2}{R} \right\rceil = \left\lfloor \frac{2a(\lfloor bR/N \rceil_2/2)}{R} \right\rceil$, which can be implemented through `SQRDMULH` since $\left\lfloor \frac{bR}{N} \right\rceil_2/2$ can be computed upfront. We show the resulting Neon sequence in Algorithm 10.

Note, however, that Barrett multiplication requires one factor to be known upfront. It does not apply for "point multiplication" of two unknown values.

Remark 4. This strategy works for 16- and 32-bit moduli, and any SIMD ISA which offers a double-multiply-high-with-rounding instruction. This includes the M-Profile Vector Extension (MVE), the Scalable Vector Extension 2 (SVE2) and, (16-bit lanes only), AVX2.

3.2.2 Barrett reduction

We comment on the implementation of Barrett reduction (Algorithm 3) in the Neon instruction set. For Barrett reduction of a single-width value, we choose R as large as possible in Algorithm 3 so that $\left\lfloor \frac{R}{N} \right\rceil$ is still a single-width signed value, increasing precision of the approximation. To compute $\left\lfloor \frac{z\left\lfloor \frac{R}{N} \right\rceil}{R} \right\rceil$, however, we can no longer use `SQRDMULH` as in Algorithm 10 but have to split it into `SQDMULH` and a rounding right shift `SRSHR`. We show our implementation of Barrett reduction in the Neon instruction set in Algorithm 11.

Old Algorithm 11 Barrett Reduction (Vectorized) [NG21, Algorithm 13]

Require: Odd modulus `N`, i bits long.
Require: Radix $R = 2^{16}$ or 2^{32}
Require: Rounding constant `r` $= 2^{i-3}$.
Require: Multiplier `V` $= \lceil 2^{i-2}R/\mathtt{N} \rfloor$
Ensure: $z \equiv B \pmod N$, $\frac{-N}{2} \leq z < \frac{N}{2}$

```
1: smull T0, B, V
2: smull2 T1, B, V
3: uzp2 T0, T0, T1
4: add T1, r, T0
5: sshr T1, T1, #(i − 2)
6: mls z, T1, N
```

Algorithm 11 Barrett Reduction for Neon

Require: Odd modulus `N`, i bits long.
Require: Radix $R = 2^{16}$ or 2^{32}
Require: Multiplier `V` $= \lceil 2^{i-2}R/\mathtt{N} \rfloor$
Ensure: $z \equiv B \pmod N$, $\frac{-N}{2} \leq z < \frac{N}{2}$

```
1: sqdmulh T0, B, V
2: srshr T1, T1, #(i − 1)
3: mls z, T1, N
```

`sqdmulh` = doubling multiplication, high half
`sshr` = signed shift right
`srshr` = signed shift right with rounding
Note: `N,V` can be one lane of a Neon register

3.2.3 Montgomery multiplication via doubling

Ordinary Montgomery multiplication does not lend itself to a straightforward implementation in Neon because Neon does not offer a multiply-high instruction. This is the reason why e.g. [NG21] use the long multiply `UMULL` to implement Montgomery multiplication.

We propose to use the doubling multiply-high instruction `QDMULH` instead to implement Algorithm 7. Moreover, the final step $\mathbf{mont}_N^{\pm}(z) \leftarrow \frac{z-c}{2}$ can be implemented via the halving subtract instruction `SHSUB`.

The resulting Neon sequence is shown in Algorithm 12. It provides a 5-instruction sequence for Montgomery multiplication of two unknown values, and a 4-instruction sequence if one factor is a constant.

Algorithm 12 Montgomery multiplication with doubling.

Require: N odd modulus, $R = 2^n > N$
Require: $a, b \in \mathbb{Z}$ representative mod N with $|a|, |b| < R$.
Require: Precomputed $T = \overline{N}^{-1} \in \mathbb{Z}_R$.
Ensure: $\mathbf{mont}_N^+(a,b)$ representative of $\underline{abR}^{-1} \in \mathbb{Z}_N$ satisfying $|\mathbf{mont}_N^+(a,b)| \leq \frac{|a||b|}{2^n} + \frac{N}{2}$.

```
sqdmulh z, a, b
mul k, a, bT mod± R
sqdmulh c, k, N
shsub z, z, c
```

Algorithm 13 Montgomery multiplication with rounding.

```
sqrdmulh z, a, b
mul k, a, −bN^{-1} mod± R
sqrdmlah z, k, N
```

3.2.4 Montgomery multiplication via rounding

It is natural to try to merge the multiply-high and subtract step in Montgomery multiplication to shorten the modular multiplication sequence further. However, there is no plain multiply-high-accumulate instruction. Instead, MVE, SVE2 and the Arm-v8.1A extension to Neon provide a multiply-high-accumulate-with-rounding instruction, which lends itself to an implementation of Algorithm 8 as shown in Algorithm 13. Like Barrett multiplication, this provides a 3-instruction sequence for modular multiplication with a known constant; unlike Barrett multiplication, however, it also applies to two unknown factors, provided one of them is known to be odd. While this is a strong condition, one has some leverage to reason about parity since the result of Algorithm 13 is always even. Note also that since `sqrdmlah` does not preform halving, Algorithm 13 computes a representative of $\underline{2abR}^{-1}$ instead of $\underline{2abR}$.

3.2.5 Montgomery multiplication in long arithmetic

Old Algorithm 14 Neon Montgomery multiplication [NG21, Alg. 12][SKS+21].

Inputs: $(\mathtt{a}, \mathtt{b}) = (a, b)$
Outputs: $\mathtt{c} \equiv abR^{-1} \pmod N$, $|c| < N$

```
1: smull l, a, b
2: smull2 h, a, b
3: uzp1 t0, l, h
4: uzp2 t1, l, h
5: mul c, t0, N̄^{-1} mod± R
6: smull l, c, N
7: smull2 h, c, N
8: uzp2 t0, l, h
9: sub c, t1, t0
```

Algorithm 14 Our Neon Montgomery

Inputs: $(\mathtt{a}, \mathtt{b}) = (a, b)$
Outputs: $\mathtt{c} = abR^{-1} \pmod N$, $|c| < N$

```
1: smull l, a, b
2: smull2 h, a, b
3: uzp1 t, l, h
4: mul t, t, −N̄^{-1} mod± R
5: smlal l, t, N
6: smlal2 h, t, N
7: uzp2 c, l, h
```

Steps 3–7 do a better `l,h`→`c` Montgomery reduction than [NG21, Alg. 12, Steps 3–9].

We have so far focused on single-width implementations of Montgomery and Barrett multiplication. Those implementations, however, do not lend themselves well to applications which need to compute *sums* of modular multiplications: In this case, it is natural to reduce only once after the accumulation, rather than once after every product. Single-width

modular multiplication cannot achieve this because it misses the carry-in between low and high parts. Instead, we need Montgomery multiplication using long products. We show an implementation of Montgomery multiplication in long arithmetic in Algorithm 14.
After long pairwise products via `smull, smull2` from two vectors, we perform a Montgomery reduction by taking the lower halves (conveniently, with `uzp1`). This we multiply by the inverse of the modulus with `mul`. We multiply the modulus to the result and accumulate the long products with `smlal, smlal2`. Now we collate the Montgomery results in the top half happily with `uzp2`. The result is exactly right as in Algorithm 14. As desired, we can accumulate several products before the Montgomery reduction (cf. Section 4.2).

4 Implementation

In this section, we fix $\mathbb{F} = \mathbb{F}_q$ for Dilithium and Kyber, and $\mathbb{F} = \mathbb{F}_{q'}$ for Saber. Section 4 is organized as follows: In Section 4.1, we describe our choices of butterflies for NTTs. Section 4.2 introduces the asymmetric multiplication applicable to the `MatrixVectorMul` in Kyber and Saber. Section 4.3 decribes our findings on how interleaving can be applied for radix-2 NTTs on Cortex-A72.

4.1 Butterflies

In this section, we adopt a more architectural viewpoint of radix-2 NTTs.

The need for permutations in the NTTs. For vectorized implementations, an important consideration is the overhead of permuting data within vector registers. This is required when the distance between the inputs of a butterfly is less than the size of a SIMD register. First, we notice that since each SIMD register in Neon is holding 16 bytes, any butterfly taking inputs at distance larger than 16 bytes doesn't require any permutation. Now we carefully look at the NTTs at each layer and number the layers from 0. After each layer, the distance for the butterfly inputs is halved. For Dilithium and Saber, we compute 32-bit NTTs for degree-255 polynomials. At the kth layer, we are computing butterflies with inputs distancing by $4 \cdot 2^{7-k} = \frac{512}{2^k}$ bytes. Therefore, after the 5th layer, any follow-up NTTs require permutations. As for Kyber with 16-bit NTTs for degree-255 polynomials, it is after the 4th layer that one needs permutations for the follow-up NTTs.

`{ld, st}{1, 2, 3, 4}` and `trn{1, 2}`. `ld{2, 3, 4}` are loading with the indicated degree of interleaving. An array of structures with 2 (3, 4) elements are loaded to 2 (3, 4) SIMD registers where the first lane of each register is holding the first structure and so on. On the other hand, `ld1` is simply loading consecutively to 1 to 4 SIMD registers. `st{1, 2, 3, 4}` are their counterparts for storing structures. Besides shuffling with memory operations, we have `trn{1, 2}` for permuting the elements of SIMD registers. `trn1` (`trn2`) moves the even (odd) indices of the first source to the even indices of the destination and the even (odd) indices of the second source to the odd indices of the destination.

Saber. First of all, since Arm-v8A provides instructions for both 16-bit and 32-bit arithmetic, we find no reasons to employ 16-bit NTTs as in [NG21]. We decide to implement 32-bit NTT as implemented for Cortex-M4 in [CHK+21]. For `NTT`, we compute for $\mathbb{F}[x]/\langle x^{256}+1\rangle$ with 6 layers of CT butterflies. On the other hand, we compute $\texttt{NTT}^{-1}$ for $\mathbb{F}[x]/\langle x^{256}-1\rangle$ with 6 layers of CT butterflies and then map $\mathbb{F}[x]/\langle x^{256}-1\rangle$ to $\mathbb{F}[x]/\langle x^{256}+1\rangle$. Recall that at the end of $\texttt{NTT}^{-1}$, we have to multiply each coefficient with 2^{-k}, so we can multiply each chunk of 4 coefficients with the precomputed $(2^{-k}, 2^{-k}\zeta^{-1}, 2^{-k}, \ldots, 2^{-k}\zeta^{-63})$.

Dilithium. We implement `NTT` with CT butterflies and $\texttt{NTT}^{-1}$ with GS butterflies. For `NTT`, after 4 radix-2 splits, the distance of the butterfly inputs for the next layer is 32 bytes. For the next 4 layers of NTTs, we first load with `ld1` and apply two layers of radix-2 splits on four SIMD registers as usual. Next, we transpose the four registers with Algorithm 28. At the end of butterflies, we can store with `st4` to cancel out the transpose. For the $\texttt{NTT}^{-1}$, we invert the entire process: for the initial 4 layers of $\texttt{NTT}^{-1}$, we load with `ld4`, compute two layers with GS, invert Algorithm 28, compute two layers with GS, and store with `st1`. For the last 4 layers, we invert CT butterflies with GS butterflies and merge half of the multiplications by 2^{-8} with ζ^{-128}.

Kyber. We also implement `NTT` with CT butterflies and $\texttt{NTT}^{-1}$ with GS butterflies for Kyber. For the `NTT`, after 4 radix-2 splits, the distance of the butterfly inputs for the next layer is 16 bytes. Now we proceed for the bottom 3 layers with a different permutation. First, we `ld4` each 4-byte to 8 SIMD registers. Next, we use `trn1` and `trn2` to separate the upper 4-bytes from the lower 4-bytes for butterflies as shown in Algorithm 29. In this way, we can then apply 3 layers of butterflies without intermediate permutations, which prohibits an aggressive interleaving of instructions. We will discuss the interleaving in Section 4.3. For $\texttt{NTT}^{-1}$, we invert the entire `NTT` and also merge half of the multiplications by 2^{-7} with ζ^{-64}.

Table 3 is the summary of butterflies for `NTT` and $\texttt{NTT}^{-1}$.

4.2 Asymmetric Multiplication

We introduce an idea that we call *asymmetric multiplication* for improving the efficiency of matrix-to-vector polynomial multiplication based on incomplete NTTs, and in general whenever incomplete NTTs are cached. It is thus applicable to Kyber and Saber, but not to Dilithium.

Recall that for Kyber and Saber, we can compute As' as $\texttt{NTT}^{-1}(\texttt{NTT}(A) \circ \texttt{NTT}(s'))$, where $k = 2$ and 4 for Kyber and Saber, respectively. As with any matrix-to-vector product, every entry of $\texttt{NTT}(s')$ will be multiplied with ℓ entries from $\texttt{NTT}(A)$. It is therefore beneficial to cache $\texttt{NTT}(s')$, which is indeed a known and common optimization technique for NTT-based multiplication. Asymmetric multiplication extends this observation by caching more computations on the multiplicands of one side.

Specifically, note that when computing the product of $a = \sum_i a_i x^i$ and $s = \sum_i s_i x^i$ in $\mathbb{F}[x]/\langle x^k - \omega \rangle$, we need to compute and sum $a_i s_j$ for $i + j < k$ and $\omega a_i s_j = a_i(\omega s_j)$ for $i + j \geq k$. It is therefore beneficial to precompute and cache the scaled ωs along with s. After the precomputation, the arithmetic cost for products in $\mathbb{F}[x]/\langle x^k - \omega \rangle$ is then effectively reduced to that of products in $\mathbb{F}[x]/\langle x^k - 1 \rangle$. Note that this is different from the isomorphism $\mathbb{F}[x]/\langle x^k - \omega \rangle \cong \mathbb{F}[y]/\langle y^k - 1 \rangle$, which doesn't work for Kyber as there is no k-th roots for ω. We call the multiplication strategy "asymmetric" because it requires the s-input in expanded form $(s, \omega s)$, while the a-input in the usual form. Algorithm 16 is an illustration. Note again the arithmetical similarity between Algorithm 15 and Algorithm 16.

We denote `NTT_heavy` the composition of the incomplete NTT with the computation $s \mapsto (s, \omega s)$, and `asymmetric_mul` implementing $\texttt{asymmetric\_mul}\,(\texttt{NTT}(a), \texttt{NTT\_heavy}(s)) = \texttt{NTT}(a) \circ \texttt{NTT}(s)$ in the asymmetric fashion. Then As' can be computed as

$$As' = \texttt{NTT}^{-1}\left(\texttt{asymmetric\_mul}(\texttt{NTT}(A), \texttt{NTT\_heavy}(s'))\right).$$

Lastly, we extend the idea of *better accumulation for schoolbook multiplication* in [CHK+21] to *better accumulation for asymmetric multiplication*, giving the 64-bit results and reducing them to 32-bit after computing all the corresponding asymmetric multiplications. See Algorithm 32.

Algorithm 15 4×4 convolution.

Inputs:
`c`, `l`, `h`, `a0`, ..., `a3`, `T0`, ..., `T3`
Output:
$c_0 + c_1x + c_2x^2 + c_3x^3 =$
$\big(\boldsymbol{a} * \boldsymbol{b} \bmod (x^4 - 1)\big)\,\mathtt{R}^{-1}$

1: Name $\mathtt{c} = c_0$,
$(\mathtt{a0}, \ldots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$,
$(\mathtt{T0}, \ldots, \mathtt{T3}) = (b_0, b_1, b_2, b_3)$
2: `smull l, a0, T0`
3: `smull2 h, a0, T0`
4: `smlal l, a1, T3`
5: `smlal2 h, a1, T3`
6: `smlal l, a2, T2`
7: `smlal2 h, a2, T2`
8: `smlal l, a3, T1`
9: `smlal2 h, a3, T1`
10: `q_montgomery c, l, h`
11: Rename $(\mathtt{T0\text{-}3}) = (b_1, b_2, b_3, b_0)$, $\mathtt{c} = c_1$ and repeat lines 4–10
12: Rename $(\mathtt{T0\text{-}3}) = (b_2, b_3, b_0, b_1)$, $\mathtt{c} = c_2$ and repeat lines 4–10
13: Rename $(\mathtt{T0\text{-}3}) = (b_3, b_0, b_1, b_2)$, $\mathtt{c} = c_3$ and repeat lines 4–10

Algorithm 16 4×4 asymmetric multiplication.

Inputs:
`c`, `l`, `h`, `a0`, ..., `a3`, `T0`, ..., `T3`
Output:
$c_0 + c_1x + c_2x^2 + c_3x^3 =$
$\big(\boldsymbol{a} * \boldsymbol{b} \bmod (x^4 - \omega)\big)\,\mathtt{R}^{-1}$

1: Name $\mathtt{c} = c_0$,
$(\mathtt{a0}, \ldots, \mathtt{a3}) = (a_0, a_1, a_2, a_3)$,
$(\mathtt{T0}, \ldots, \mathtt{T3}) = (b_0, \omega b_1, \omega b_2, \omega b_3)$
2: `smull l, a0, T0`
3: `smull2 h, a0, T0`
4: `smlal l, a1, T3`
5: `smlal2 h, a1, T3`
6: `smlal l, a2, T2`
7: `smlal2 h, a2, T2`
8: `smlal l, a3, T1`
9: `smlal2 h, a3, T1`
10: `q_montgomery c, l, h`
11: Rename $(\mathtt{T0\text{-}3}) = (b_1, \omega b_2, \omega b_3, b_0)$, $\mathtt{c} = c_1$ and repeat lines 4–10
12: Rename $(\mathtt{T0\text{-}3}) = (b_2, \omega b_3, b_0, b_1)$, $\mathtt{c} = c_2$ and repeat lines 4–10
13: Rename $(\mathtt{T0\text{-}3}) = (b_3, b_0, b_1, b_2)$, $\mathtt{c} = c_3$ and repeat lines 4–10

`q_montgomery` is Step 3–7 in Algorithm 14.

4.3 Interleaving for Multi-Layer Butterflies on Cortex-A72

This section describes our findings on how to implement radix-2 NTTs for the CPU Cortex-A72. We follow the software optimization guide of Cortex-A72 [ARM].

In-order frontend and out-of-order backend. In the in-order frontend of the pipeline, instructions are fetched and decoded into internal micro-operations (μops). After renaming the registers, μops are dispatched to the out-of-order backend. In the backend, there is one branch pipeline `B`, two integer pipelines `I0` and `I1`, one integer multi-cycle pipeline `M`, two FP/ASIMD pipelines `F0` and `F1`, one load pipeline `L`, and one store pipeline `S`. While up to three μops can be dispatched per cycle, there are limitations on the number of each type of μops that can be dispatched simultaneously. The following are the numbers for each type in a single cycle: one μop using `B`, up to two μops using `I0`/`I1`, up to two μops using `M`, one μop using `F0`, one μop using `F1`, and up to two μops using `L`/`S`. Furthermore, μops are dispatched in the oldest-to-youngest age order.

Algorithm 17 CT butterflies.

Inputs:

$\mathtt{a}, \mathtt{b}, \mathtt{t}, (\mathtt{z[l]}, \mathtt{z[h]}) = (\bar{\omega}, \frac{\lfloor \frac{\omega R}{N} \rceil_2}{2})$

Outputs:

$(\mathtt{a}, \mathtt{b}) = (\mathtt{a} + \omega\mathtt{b}, \mathtt{a} - \omega\mathtt{b})$

```
1: mul t, b, z[h]
2: sqrdmulh b, b, z[l]
3: mls t, b, N
4: sub b, a, t
5: add a, a, t
```

Algorithm 18 GS butterflies.

Inputs:

$\mathtt{a}, \mathtt{b}, \mathtt{t}, (\mathtt{z[l]}, \mathtt{z[h]}) = (\bar{\omega}, \frac{\lfloor \frac{\omega R}{N} \rceil_2}{2})$

Outputs:

$(\mathtt{a}, \mathtt{b}) = (\mathtt{a} + \mathtt{b}, (\mathtt{a} - \mathtt{b})\,\omega)$

```
1: sub t, a, b
2: add a, a, b
3: mul b, t, z[h]
4: sqrdmulh t, t, z[l]
5: mls b, t, N
```

`F0` and `F1` for butterflies. We focus on the pipelines `F0` and `F1` for our vectorized implementation. The basic building blocks for butterflies are Algorithm 17 and Algorithm 18. Instructions `mul`, `mls`, `sqrdmulh` can only go to `F0`, while `sub` and `add` go to `F0` or `F1`. We interleave the instructions so `sub` and `add` have a better chance to be dispatched to `F1`. We believe `F0` is the bottleneck of `NTT`s, and hence, reducing the loading of `F0` speeds up the computation.

`qq_butterfly_{top, bot}` and `qq_butterfly_mixed{, _rev}`. To facilitate the development of assembly implementation of radix-2 NTTs with interleaving, we split the computation of each layer into two `qq_butterfly_top` and two `qq_butterfly_bot`. `qq_butterfly_top` computes 4 Montgomery multiplications and `qq_butterfly_bot` computes 4 sub-add pairsas shown in Algorithm 33 and Algorithm 34. They are designed in the way that if we pass the same arguments to them, calling `qq_butterfly_top` followed by `qq_butterfly_bot` implements CT butterflies and reversing the order implements GS butterflies. Algorithm 37 is a straightforward implementation of a 4-layer-CT-butterfly. If there are no dependencies, we can interleave `qq_butterfly_bot` with `qq_butterfly_top` giving `qq_butterfly_mixed` for CT butterflies and `qq_butterfly_mixed_rev` for GS butterflies as shown in Algorithm 35 and Algorithm 36.

Multi-layer butterflies. A common approach for reducing the number of memory operations is to compute several layers of NTTs at a time with multi-layer butterflies. To avoid the dependencies when interleaving instructions, we find that for radix-2 NTTs, computing coefficients distributed over 16 SIMD registers is the most beneficial approach. This suggests that 4 layers at a time are possible since $2^4 = 16$.

Interleaving for multi-layer butterflies. At first glance, it seems natural to compute the butterflies in the order $(\mathtt{v0}, \mathtt{v8})$, …, $(\mathtt{v6}, \mathtt{v14})$, $(\mathtt{v1}, \mathtt{v9})$, …, $(\mathtt{v7}, \mathtt{v15})$ for the 0th layer, $(\mathtt{v0}, \mathtt{v4})$, $(\mathtt{v2}, \mathtt{v6})$, $(\mathtt{v8}, \mathtt{v12})$, $(\mathtt{v10}, \mathtt{v14})$, $(\mathtt{v1}, \mathtt{v5})$, $(\mathtt{v3}, \mathtt{v7})$, $(\mathtt{v9}, \mathtt{v13})$, $(\mathtt{v11}, \mathtt{v15})$ for the 1st layer, and so on. Since the independencies of the first three layers are exploited, the computation can be interleaved as follows: interleave the first half of sub-add pairs of layer 0 with the second half of multiplications of layer 0, the second half of sub-add pairs of layer 0 with the first half of multiplications of layer 1, and so on. However, we cannot interleave layers 2 with 3 at all. To overcome this, we compute the butterflies in the following order: $(\mathtt{v1}, \mathtt{v9}), \ldots, (\mathtt{v7}, \mathtt{v15})$, $(\mathtt{v0}, \mathtt{v8}), \ldots, (\mathtt{v6}, \mathtt{v14})$ for the 0th layer, $(\mathtt{v1}, \mathtt{v5})$, $(\mathtt{v3}, \mathtt{v7})$, $(\mathtt{v9}, \mathtt{v13})$, $(\mathtt{v11}, \mathtt{v15})$, $(\mathtt{v0}, \mathtt{v4})$, $(\mathtt{v2}, \mathtt{v6})$, $(\mathtt{v8}, \mathtt{v12})$, $(\mathtt{v10}, \mathtt{v14})$ for the 1st layer, $(\mathtt{v1}, \mathtt{v3})$, $(\mathtt{v5}, \mathtt{v7})$, $(\mathtt{v9}, \mathtt{v11})$, $(\mathtt{v13}, \mathtt{v15})$, $(\mathtt{v0}, \mathtt{v2})$, $(\mathtt{v4}, \mathtt{v6})$, $(\mathtt{v8}, \mathtt{v10})$, $(\mathtt{v12}, \mathtt{v14})$ for the 2nd layer, and finally $(\mathtt{v0}, \mathtt{v1})$, $(\mathtt{v2}, \mathtt{v3})$, $(\mathtt{v4}, \mathtt{v5})$, $(\mathtt{v6}, \mathtt{v7})$, $(\mathtt{v8}, \mathtt{v9})$, $(\mathtt{v10}, \mathtt{v11})$, $(\mathtt{v12}, \mathtt{v13})$, $(\mathtt{v14}, \mathtt{v15})$

for the 3rd layer. In this order, we are exploiting the independencies among the entire 4-layer computation as shown in Algorithm 38.

5 Result

We provide benchmarking results on the Arm Cortex-A72 processor and the Apple M1.

Arm Cortex-A72. The Arm Cortex-A72 CPU implements the Arm-v8A architecture and has a triple-issue out-of-order pipeline. Specifically, we use the Raspberry Pi 4 Model B featuring the quad-core Broadcom BCM2711 chipset. It comes with a 32 kB L1 data cache, a 48 kB L1 instruction cache, and a 1 MB L2 cache and runs at 1.5 GHz. For hashing, we use the SHA-3/SHAKE Neon implementation by Nguyen et al. [NG21]. For benchmarking individual functions we make use of the cycle counter of the PMU. For benchmarking the full cryptographic schemes, we use SUPERCOP.[3] We use gcc version 10.3.0 with `-O3`.

Apple M1. The Apple M1 system-on-chip is contained in Apple's 2020 MacBooks. It has four high-performance *Firestorm* cores, and four energy-efficient *Icestorm* cores. The Apple M1 has special instructions for Keccak. We use the Keccak implementation by Westerbaan [Wes] which makes use of these instructions. For obtaining cycle counts we make use of `m1cycles.c`[4] from [NG21]. Due to the large and varying overhead of obtaining cycles, we cannot reasonably benchmark a single execution of a small function. Instead, we benchmark many iterations and report the average. We use clang version 12.0.5 with `-O3`.

Results for `NTT` and $\texttt{NTT}^{-1}$. Table 5 summarizes our results for the NTT. On the Cortex-A72, we outperform the Kyber `NTT` and $\texttt{NTT}^{-1}$ by Nguyen et al. [NG21] by 19% each, even though we include Barrett reduction which is not reported by Nguyen et al. [NG21]. Compared to the implementation by Sanal et al. [SKS+21], the speed-up is even more pronounced at 1.9× for the `NTT` and 2.4× for $\texttt{NTT}^{-1}$. For Saber, our 32-bit `NTT` is 23% faster than the 16-bit `NTT` by Nguyen et al. [NG21]. However, the 16-bit `NTT` approach requires two `NTT`s followed by CRT and, consequently, our real speed-up is 3.1×. On the Apple M1, our speed-up compared to the Kyber `NTT` and $\texttt{NTT}^{-1}$ by Nguyen et al. [NG21] is 1.6× each. The 16-bit Saber `NTT`s are outperformed by our 32-bit `NTT`s by 44% and 27% on the M1. The actual speed-up when comparing the 32-bit `NTT` to two 16-bit `NTT`s followed by CRT is 4.3×. For Dilithium, we obtain a vast speed-up over the reference implementation.

Results for `MatrixVectorMul` and `InnerProd`. Table 6 presents the results for the core arithmetic operations `MatrixVectorMul` and `InnerProd`. For Kyber, our `MatrixVectorMul` is 1.7× faster on Cortex-A72 and 2.1× faster on Apple M1 than previous implementations [NG21]. For `InnerProd`, our code is 1.5× faster on Cortex-A72 and 2.1× faster on Apple M1. For Saber, we speed up `MatrixVectorMul` by 2.1× on Cortex-A72 and 1.9× on Apple M1. The speed-up for `InnerProd` is 1.6× on Cortex-A72 and 1.3× on Apple M1. Also note that for the `InnerProd` in encapsulation of Kyber and Saber, one can re-use $\texttt{NTT\_heavy}(s')$ to obtain a much faster `InnerProd`.

Results for full schemes. Table 4 shows our results for the full cryptographic schemes. Kyber runs 9 – 13% faster on Cortex-A72 and 26% – 35% faster on Apple M1 than previous work [NG21]. For Saber, the speed-ups are even more significant with 31 – 35%

[3] https://bench.cr.yp.to/supercop.html, Version 20210604
[4] https://github.com/GMUCERG/PQC_NEON/blob/main/neon/kyber/m1cycles.c

Table 3: Summary of butterflies for `NTT`s and $\texttt{NTT}^{-1}$s.

	`NTT`	$\texttt{NTT}^{-1}$
`kyber768`	4-layer-CT + 3-layer-CT	3-layer-GS + 4-layer-GS
`saber`	2× 3-layer-CT	2× 3-layer-CT
`dilithium3`	2× 4-layer-CT	2× 4-layer-GS

Table 4: Performance results for the full schemes `kyber768`, `saber`, and `dilithium3` on Cortex-A72 and Apple M1. On Cortex-A72, we report the median cycle count of 10 000 executions for Kyber and Saber, and 100 000 executions for Dilithium. On Apple M1, we report the average cycle count of 10 000 executions for Kyber and Saber, and 1 000 000 executions for Dilithium. For Dilithium, we sign 59-byte messages.

	Cortex-A72			Apple M1		
	K	**E**	**D**	**K**	**E**	**D**
`kyber768` (Ours)	100 097	128 384	121 353	23 760	36 235	30 995
`kyber768` [NG21][a]	110 784	141 312	138 984	36 300	49 200	45 700
`kyber768` [SKS+21][b]	143 791	180 687	179 085	–	–	–
`saber` (Ours)	109 584	139 234	146 536	32 812	43 355	42 524
`saber` [NG21][a,c]	158 757	206 337	226 304	51 300	59 900	58 000
	K	**S**	**V**	**K**	**S**	**V**
`dilithium3` (Ours)	534 632	1 188 460	454 186	159 320	462 281	112 072
`dilithium3` (ref)	743 166	2 308 598	728 866	358 848	1 218 027	329 187

[a] We re-benchmark the code from [NG21] in SUPERCOP.
[b] [SKS+21] targets the Arm Cortex-A75.
[c] Fastest implementation in [NG21] (Toom–Cook).

fewer cycles on Cortex-A72 and 27 – 36 % fewer cycles on Apple M1. Unsurprisingly, our Dilithium implementation performs much better than the reference implementation.

On Saber's SHA-3 performance. As we make use of a vectorized SHA-3 implementation computing two Keccak permutations at once, optimal performance can only be achieved if SHA-3/SHAKE calls can be parallelized. Unfortunately, the Saber specification mandates sampling vectors and matrices as a single call to SHAKE and can, thus, not benefit from Keccak parallelization. We believe that the Saber specification should be changed. Saber's performance on Cortex-A72 and Apple M1 will then be much closer to Kyber.

Table 5: Performance results for `NTT`, `NTT_heavy`, `base_mul`, and $\texttt{NTT}^{-1}$ for `kyber768`, `saber`, and `dilithium3` on Cortex-A72 and Apple M1. `dim` refers to the module dimension. Some implementations implement the `base_mul` for each polynomial separately, while we implement it for an entire vector such that we can better optimize the accumulation. On Cortex-A72, we report the median cycle count of 10 000 executions. On Apple M1, we report the average cycle count of 10 000 executions. We also list related work on Cortex-A72, Cortex-A75, and Apple M1 for comparison.

Arm Cortex-A72 (Arm Cortex-A75 for [SKS+21])						
		`NTT`	`NTT_heavy`	`dim` × `base_mul`	$\texttt{NTT}^{-1}$	`CRT`
`kyber768` (Ours)	A72	1 200	1 434	952	1 338	–
`kyber768` [NG21]	A72	1 473[a]	–	3 040[d]	1 661	–
`kyber768` [SKS+21]	A75	2 332	–	3 × 1 313	3 209	–
`saber` 32-bit (Ours)	A72	1 529	2 031	2 689	1 896	–
`saber` 16-bit [NG21]	A72	1 991[b]	–	1 500[d]	1 893	813[d]
`dilithium3` (Ours)	A72	2 241	–	1 378	2 821	–
`dilithium3` (ref)	A72	9 302	–	5 × 2 325	11 633	–
Apple M1						
		`NTT`	`NTT_heavy`	`dim` × `base_mul`	$\texttt{NTT}^{-1}$	`CRT`
`kyber768` (Ours)	M1	263	309	198	262	–
`kyber768` [NG21]	M1	413[a]	–	753[d]	428	–
`saber` (Ours)	M1	301	411	790	389	–
`saber` 16-bit [NG21]	M1	539[c]	–	380[d]	531	206[d]
`dilithium3` (Ours)	M1	479	–	258	582	–
`dilithium3` (ref)	M1	2 865	–	5 × 794	3 749	–

[a] [NG21] reports cycles without a final reduction, while our implementation includes a reduction.
[b] A fair comparison to our 32-bit `NTT` is 2× 16-bit `NTT` + CRT = 4 795
[c] A fair comparison to our 32-bit `NTT` is 2× 16-bit `NTT` + CRT = 1 284
[d] Our own benchmarks; not reported in [NG21].

Table 6: Cycle counts for `MatrixVectorMul` (`MV`), and `InnerProd` (`IP`) for `kyber768`, `saber`, and `dilithium3` on Cortex-A72 and Apple M1. For `kyber768` and `saber` the `InnerProd` in encapsulation (E) can re-use intermediate results from the `MatrixVectorMul`; in decapsulation (D) a full `InnerProd` is needed. `dilithium3` does not use an `InnerProd`. On Cortex-A72, we report median cycle count of 10 000 executions. On Apple M1, we report the average cycle count of 10 000 executions.

	Arm Cortex-A72			Apple M1		
	`MV`	`IP`(E)	`IP` (D)	`MV`	`IP` (E)	`IP` (D)
`kyber768` (Ours)	11 132	2 271	6 538	2 291	461	1 232
`kyber768` [NG21]	19 300	–	9 900	4 910	–	2 545
`saber` 32-bit NTT (Ours)	33 696	9 161	15 285	7 475	2 081	3 312
`saber` 16-bit NTT [NG21]	71 300	–	31 500	18 931	–	8 470
`saber` Toom–Cook [NG21]	81 000	–	25 000	14 029	–	4 345
`dilithium3` (Ours)	37 492	–	–	7 945	–	–
`dilithium3` (ref)	215 503	–	–	72 516	–	–

References

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. NISTIR8309 – status report on the second round of the nist post-quantum cryptography standardization process, July 2020. https://doi.org/10.6028/NIST.IR.8309.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020. https://doi.org/10.13154/tches.v2020.i3.336-357.

[ABD+20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/dilithium/.

[ABD+20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://pq-crystals.org/kyber/.

[ACC+21] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial multiplication in NTRU prime comparison of optimization strategies on Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):217–238, 2021. https://doi.org/10.46586/tches.v2021.i1.217-238.

[ARM] ARM. Cortex-a72 software optimization guide. https://developer.arm.com/documentation/uan0016/a/.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *CRYPTO 1986*, LNCS, pages 311–323. Springer, 1986.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *Progress in Cryptology - AFRICACRYPT 2019*, volume 11627 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2019. https://doi.org/10.1007/978-3-030-23696-0_11.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings new speed records for saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):159–188, 2021. https://doi.org/10.46586/tches.v2021.i2.159-188.

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, 2021. https://doi.org/10.46586/tches.v2021.i1.1-24.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography*, volume 10677, pages 341–371, 2017. https://eprint.iacr.org/2017/604.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly Fast NTRU Using NTT. Cryptology ePrint Archive, Report 2019/040, 2019. https://eprint.iacr.org/2019/040.

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

[NG21] Duc Tri Nguyen and Kris Gaj. Optimized software implementations of CRYSTALS-kyber, NTRU, and saber using NEON-based special instructions of ARMv8. https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/nguyen-optimized-software-gmu-pqc2021.pdf, 2021. Third PQC Standardization Conference.

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography.

[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.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[SKS+21] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Kyber on ARM64: compact implementations of kyber on 64-bit ARM Cortex-A processors. Cryptology ePrint Archive, Report 2021/561, 2021. https://eprint.iacr.org/2021/561.

[TOP20] TOP500. Top500 November 2020, 2020. https://www.top500.org/lists/top500/2020/11/.

[TOP21] TOP500. Top500 June 2021, 2021. https://www.top500.org/lists/top500/2021/06/.

[Wes] Bas Westerbaan. Keccak-f1600 on aarch64. https://github.com/bwesterb/armed-keccak.

Acknowledgments

The authors in Taiwan thank Ministry of Science and Technology for grants 109-2221-E-001-009-MY3 and 109-2923-E-001-001-MY3, the Sinica Investigator Award AS-IA-109-M01, Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

A Kyber CPA PKE

Algorithms 19–21 ([ABD+20b]), are the CPA-secure key generation, encryption, and decryption for the Kyber PKE.

Here the module is of dimension ℓ over the ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, with $q = 3329$ and $n = 256$; $\texttt{Sample}_U$ samples from the uniform distribution; $\texttt{Sample}_B$ samples from a centered binomial (η fair coins); and `Expand` samples a seed into a uniform matrix of polynomials. Notice that A is sampled in (incomplete) NTT domain directly.

Algorithm 19 Kyber Key Generation

Output: $pk = (\text{seed}_A, \text{NTT}(b)), sk = (\text{NTT}(s))$

1: $\text{seed}_A \leftarrow \texttt{Sample}_U()$
2: $\text{NTT}(A) \in R_q^{\ell\times\ell} \leftarrow \texttt{Expand}(\text{seed}_A)$
3: $s, e \in R_q^{\ell} \leftarrow \texttt{Sample}_B()$
4: $b \leftarrow A^T \cdot s + e$

Algorithm 20 Kyber CPA Encryption

Input: $m, r, pk = (\text{seed}_A, \text{NTT}(b))$
Output: $ct = (c', b'')$

1: $\text{NTT}(A) \in R_q^{\ell\times\ell} \leftarrow \texttt{Expand}(\text{seed}_A)$
2: $s', e' \in R_q^{\ell}, e'' \in R_q \leftarrow \texttt{Sample}_B(r)$
3: $c \leftarrow As' + e'$
4: $b' \leftarrow b^T s' + e'' + m$
5: $ct \leftarrow \texttt{Compress}(c, d_1), \texttt{Compress}(b', d_2))$

Algorithm 21 Kyber CPA Decryption

Input: $ct = (c', b''), sk = (\text{NTT}(s))$
Output: m

1: $c \leftarrow \texttt{Decompress}(c', d_1)$
2: $b' \leftarrow \texttt{Decompress}(b'', d_2)$.
3: $m \leftarrow b'' - c^T s$

$$\texttt{Compress}(x, d) = \lceil (2^d/q)x \rfloor \bmod 2^d$$
$$\texttt{Decompress}(x, d) = \lceil (q/2^d)x \rfloor$$

B Saber CPA PKE

Algorithms 22–24 ([DKRV20]), are CPA-secure key generation, encryption, and decryption for the Saber PKE. Here, the module is of dimension ℓ over the $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, with $q = 2^{13}$ and $n = 256$; $\texttt{Sample}_U$ samples from the uniform distribution; $\texttt{Sample}_B$ samples from a centered binomial (η fair coins); and `Expand` samples a seed into a uniform matrix of polynomials.

Algorithm 22 Saber Key Generation

Output: $pk = (\text{seed}_A, b), sk = (s)$

1: $\text{seed}_A \leftarrow \texttt{Sample}_U()$
2: $A \in R_q^{\ell\times\ell} \leftarrow \texttt{Expand}(\text{seed}_A)$
3: $s \in R_q^{\ell} \leftarrow \texttt{Sample}_B()$
4: $b \leftarrow \texttt{Round}(A^T \cdot s)$

Algorithm 23 Saber CPA Encryption

Input: $m, r, pk = (\text{seed}_A, b)$
Output: $ct = (c, b')$

1: $A \in R_q^{\ell\times\ell} \leftarrow \texttt{Expand}(\text{seed}_A)$
2: $s' \in R_q^{\ell} \leftarrow \texttt{Sample}_B(r)$
3: $b' \leftarrow \texttt{Round}(As')$
4: $v' \leftarrow b^T (s' \mod p)$
5: $c \leftarrow \texttt{Round}(v' - 2^{\epsilon-1} m)$

Algorithm 24 Saber CPA Decryption

Input: $ct = (c, b'), sk = (s)$
Output: m

1: $v \leftarrow b'^T (s \mod p)$
2: $m \leftarrow \texttt{Round}(v - 2^{\epsilon_p - \epsilon_T} c \mod p)$

`Round` is to nearest multiple of T.

C Dilithium

Algorithm 25, Algorithm 26, and Algorithm 27 show the Dilithium key generation, signature generation, and verification (resp.). S_η is the uniform distribution $\{-\eta, -\eta+1, \ldots, +\eta\}$; $\|$ denotes concatenation; and $\|\cdot\|_\infty$ the sup-norm. For seed expansion functions `ExpandA` and `ExpandMask` are used; for details about the rounding functions `Power2Round`, `HighBits`, and `Decompose` and the hint functions `MakeHint` and `UseHint` please see [ABD+20a].

Algorithm 25 Dilithium key generation

Output: $sk = (r, K, tr, s_1, s_2, t_0)$
Output: $pk = (r, t_1)$
$r \leftarrow \{0,1\}^{256}$
$K \leftarrow \{0,1\}^{256}$
$(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$\text{NTT}(A) \in R_q^{k \times \ell} \leftarrow \texttt{ExpandA}(r)$
$t \leftarrow As_1 + s_2$
$(t_1, t_0) \leftarrow \texttt{Power2Round}(t)$
$tr \in \{0,1\}^{256} \leftarrow \mathcal{H}(r||\mathbf{t}_1)$

Algorithm 26 Dilithium signature generation

Input: $sk = (r, K, tr, s_1, s_2, t_0)$
Input: Message $M \in \{0,1\}^*$
Output: Signature $\sigma = (z, h, \tilde{c})$
$\text{NTT}(A) \in R_q^{k \times \ell} := \texttt{ExpandA}(r)$
$\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}(tr||M)$
$\kappa \leftarrow 0; (z, h) \leftarrow \perp$
$r' \in \{0,1\}^{512} \leftarrow \mathcal{H}(K||\mu)$
while $(z, h) = \perp$ **do**
  $y \in S_{\gamma_1 - 1}^\ell \leftarrow \texttt{ExpandMask}(r', \kappa)$
  $w \leftarrow Ay$; $w_1 \leftarrow \texttt{HighBits}(w)$
  $\tilde{c} \in \{0,1\}^{256} \leftarrow \mathcal{H}(\mu||w_1)$
  $\text{NTT}(c) \leftarrow \text{NTT}\left(\mathcal{H}_B(\tilde{c})\right)$
  $z \leftarrow y + cs_1$
  $r_0 \leftarrow \texttt{LowBits}(w - cs_2)$
  if $||z||_\infty \geq \gamma_1 - \beta$ **or** $||r_0||_\infty \geq \gamma_2 - \beta$ **then**
    $(z, h) = \perp$
  else
    $h \leftarrow \texttt{MakeHint}\left(-ct_0, w - cs_2 + ct_0\right))$
    if $\|ct_0\|_\infty \geq \gamma_2$ **or** # 1's in $h > \omega$ **then**
      $(z, h) = \perp$
    end if
  end if
  $\kappa \leftarrow \kappa + 1$
end while

Algorithm 27 Dilithium verification

Input: $pk = (r, \mathbf{t}_1)$
Input: Message $M \in \{0,1\}^*$
Input: Signature $\sigma = (\mathbf{z}, \mathbf{h}, \tilde{c})$
Output: `Valid` or `Invalid`

$\text{NTT}(A) \in R_q^{k \times \ell} \leftarrow \texttt{ExpandA}(r)$
$\mu \in \{0,1\}^{384} \leftarrow \mathcal{H}(\mathcal{H}(r||t_1)||M)$
$c \leftarrow \mathcal{H}_B(\tilde{c})$
$w_1' \leftarrow \texttt{UseHint}(h, Az - 2^d c t_1)$
if $\tilde{c} = \mathcal{H}(\mu||w_1')$ **and** $\|z\|_\infty < \gamma_1 - \beta$ **and** # 1's in $h \leq \omega$ **then**
    return `Valid`
else
    return `Invalid`
end if

D Permutation in Dilithium and Kyber

Algorithm 28 Permutation of bottom 4 layers in Dilithium NTT.

Inputs:

$$\texttt{a0} = a_3||a_2||a_1||a_0$$
$$\texttt{a1} = a_7||a_6||a_5||a_4$$
$$\texttt{a2} = a_{11}||a_{10}||a_9||a_8$$
$$\texttt{a3} = a_{15}||a_{14}||a_{13}||a_{12}$$

Outputs:

$$\texttt{a0} = a_{12}||a_8||a_4||a_0$$
$$\texttt{a1} = a_{13}||a_9||a_5||a_1$$
$$\texttt{a2} = a_{14}||a_{10}||a_6||a_2$$
$$\texttt{a3} = a_{15}||a_{11}||a_7||a_3$$

```
trn1 t0.4S, a0.4S, a1.4S
trn2 t1.4S, a0.4S, a1.4S
trn1 t2.4S, a2.4S, a3.4S
trn2 t3.4S, a2.4S, a3.4S
trn1 a0.2D, t0.2D, t2.2D
trn2 a2.2D, t0.2D, t2.2D
trn1 a1.2D, t1.2D, t3.2D
trn2 a3.2D, t1.2D, t3.2D
```

Algorithm 29 Permutation of bottom 3 layers of Kyber NTT.

Inputs:

$$*\texttt{src0} = (\texttt{a0a1}, \texttt{a2a3}, \ldots, \texttt{a30a31})$$
$$*\texttt{src1} = (\texttt{a64a65}, \texttt{a66a67}, \ldots, \texttt{a94a95})$$

Outputs:

$$\texttt{v24} = (\texttt{a0a1}, \texttt{a64a65}, \texttt{a16a17}, \texttt{a80a81})$$
$$\texttt{v25} = (\texttt{a2a3}, \texttt{a66a67}, \texttt{a18a19}, \texttt{a82a83})$$
$$\texttt{v26} = (\texttt{a4a5}, \texttt{a68a69}, \texttt{a20a21}, \texttt{a84a85})$$
$$\texttt{v27} = (\texttt{a6a7}, \texttt{a70a71}, \texttt{a22a23}, \texttt{a86a87})$$
$$\texttt{v28} = (\texttt{a8a9}, \texttt{a72a73}, \texttt{a24a25}, \texttt{a88a89})$$
$$\texttt{v29} = (\texttt{a10a11}, \texttt{a74a75}, \texttt{a26a27}, \texttt{a90a91})$$
$$\texttt{v30} = (\texttt{a12a13}, \texttt{a76a77}, \texttt{a28a29}, \texttt{a92a93})$$
$$\texttt{v31} = (\texttt{a14a15}, \texttt{a78a79}, \texttt{a30a31}, \texttt{a94a95})$$

```
1: ld4 {v16.4S, v17.4S, v18.4S, v19.4S}, [src0]
2: ld4 {v20.4S, v21.4S, v22.4S, v23.4S}, [src1]
3: trn1 v24.4S, v16.4S, v20.4S
4: trn2 v28.4S, v16.4S, v20.4S
5: trn1 v25.4S, v17.4S, v21.4S
6: trn2 v29.4S, v17.4S, v21.4S
7: trn1 v26.4S, v18.4S, v22.4S
8: trn2 v30.4S, v18.4S, v22.4S
9: trn1 v27.4S, v19.4S, v23.4S
10: trn2 v31.4S, v19.4S, v23.4S
```

E Assembly for Asymmetric Multiplication

Algorithm 30 `_4x4_asymmetric` as a building block for 32-bit 4×4 asymmetric multiplication in `saber`.

Inputs:
`mulacc`, `mulacc2`, `a0`, `b0`, ..., `b3`, `l0`, `h0`, ..., `l3`, `h3`, `dS`, `qS`, `dD`
Roles of the symbols:
Long multiplication with optional accumulations: `mulacc`, `mulacc2`
Coefficient of the `NTT` of matrix: `a0`
Coefficients of the `NTT_heavy` of vector: `b0`, ..., `b3`
Accumulators for double-size results: `l0`, `h0`, ..., `l3`, `h3`
Specifiers: `dS`, `qS`, `dD` = (`.2S`, `.4S`, `.2D`)
Outputs:
Case (`mulacc`, `mulacc2`) = (`smull`, `smull2`):
`l0` + `h0` « 32 = `a0` * `b0`, ..., `l3` + `h3` « 32 = `a0` * `b3`
Case (`mulacc`, `mulacc2`) = (`smlal`, `smlal2`):
`l0` + `h0` « 32+= `a0` * `b0`, ..., `l3` + `h3` « 32+= `a0` * `b3`

```
1: mulacc l0dD, a0dS, b0dS
2: mulacc2 h0dD, a0qS, b0qS
3: mulacc l1dD, a0dS, b1dS
4: mulacc2 h1dD, a0qS, b1qS
5: mulacc l2dD, a0dS, b2dS
6: mulacc2 h2dD, a0qS, b2qS
7: mulacc l3dD, a0dS, b3dS
8: mulacc2 h3dD, a0qS, b3qS
```

Algorithm 31 `qq_montgomery` for 16 parallel 32-bit Montgomery reductions.

Inputs:
`c0`, ..., `c3`, `l0`, ..., `l3`, `h0`, ..., `h3`, `t0`, ..., `t3`, `Qprime`, `Q`, `dS`, `qS`, `dD`
Roles of the symbols:
Output registers: `c0`, ..., `c3`
64-bit value to be reduced: `l0` + `h0` « 32, ..., `l3` + `h3` « 32
Auxiliary registers: `t0`, ..., `t3`
Specifiers: (`dS`, `qS`, `dD`) = (`.2S`, `.4S`, `.2D`)
Outputs:
`c0` = (`l0` + `h0` « 32)32^{-1} mod `Q`, ..., `c3` = (`l3` + `h3` « 32)32^{-1} mod `Q`

```
 1: uzp1 t0qS, l0qS, h0qS
 2: uzp1 t1qS, l1qS, h1qS
 3: uzp1 t2qS, l2qS, h2qS
 4: uzp1 t3qS, l3qS, h3qS
 5: mul t0qS, t0qS, QprimeqS
 6: mul t1qS, t1qS, QprimeqS
 7: mul t2qS, t2qS, QprimeqS
 8: mul t3qS, t3qS, QprimeqS
 9: smlal l0dD, t0dS, QdS
10: smlal2 h0dD, t0qS, QqS
11: smlal l1dD, t1dS, QdS
12: smlal2 h1dD, t1qS, QqS
13: smlal l2dD, t2dS, QdS
14: smlal2 h2dD, t2qS, QqS
15: smlal l3dD, t3dS, QdS
16: smlal2 h3dD, t3qS, QqS
17: uzp2 c0qS, l0qS, h0qS
18: uzp2 c1qS, l1qS, h1qS
19: uzp2 c2qS, l2qS, h2qS
20: uzp2 c3qS, l3qS, h3qS
```

Algorithm 32 4 parallel 32-bit better accumulations for asymmetric multiplications (for `saber`).

Inputs:
Source registers: `x0` for $\boldsymbol{a}_0$, `x1` for $\boldsymbol{b}_0$, `x2` for $\boldsymbol{b}'_0 = \omega\boldsymbol{b}_0$, `x4` for $\boldsymbol{a}_1$, `x5` for $\boldsymbol{b}_1$, `x6` for $\boldsymbol{b}'_1 = \omega\boldsymbol{b}_1$, `x8` for $\boldsymbol{a}_2$, `x9` for $\boldsymbol{b}_2$, `x10` for $\boldsymbol{b}'_2 = \omega\boldsymbol{b}_2$
Accumulators: `v16, v20, v17, v21, v18, v22, v19, v23`
Output registers: `v24, v25, v26, v27`
Symbols:
`mul_long` = `smull, smull2`
`mla_long` = `smlal, smlal2`
`acc_long` = `v16, v20, v17, v21, v18, v22, v19, v23`
`acc_long_T` = `v16, v17, v18, v19, v20, v21, v22, v23`
`T0` = `v0, v1, v2, v3`
`T1` = `v4, v5, v6, v7`
`T2` = `v8, v9, v10, v11`
`T3` = `v12, v13, v14, v15`
`C` = `v24, v25, v26, v27`
Outputs:
$\texttt{v24} + \texttt{v25}x + \texttt{v26}x^2 + \texttt{v27}x^3 = (\boldsymbol{a}_0 * \boldsymbol{b}_0 + \boldsymbol{a}_1 * \boldsymbol{b}_1 + \boldsymbol{a}_2 * \boldsymbol{b}_2)32^{-1} \bmod (x^4 - \omega)$

```
1: ld4 { T0}, [ x0]
2: ld4 { T1}, [ x1]
3: ld4 { T2}, [ x2]
4: _4x4_asymmetric mul_long, v3, v9, v10, v11, v4, acc_long
5: _4x4_asymmetric mla_long, v2, v10, v11, v4, v5, acc_long
6: _4x4_asymmetric mla_long, v1, v11, v4, v5, v6, acc_long
7: _4x4_asymmetric mla_long, v0, v4, v5, v6, v7, acc_long
8: ld4 { T3}, [ x4]
9: ld4 { C}, [ x5]
10: ld4 { T2}, [ x6]
11: _4x4_asymmetric mla_long, v15, v9, v10, v11, v24, acc_long
12: _4x4_asymmetric mla_long, v14, v10, v11, v24, v25, acc_long
13: _4x4_asymmetric mla_long, v13, v11, v24, v25, v26, acc_long
14: _4x4_asymmetric mla_long, v12, v24, v25, v26, v27, acc_long
15: ld4 { T0}, [ x8]
16: ld4 { T1}, [ x9]
17: ld4 { T2}, [x10]
18: _4x4_asymmetric mla_long, v3, v9, v10, v11, v4, acc_long
19: _4x4_asymmetric mla_long, v2, v10, v11, v4, v5, acc_long
20: _4x4_asymmetric mla_long, v1, v11, v4, v5, v6, acc_long
21: _4x4_asymmetric mla_long, v0, v4, v5, v6, v7, acc_long
22: qq_montgomery C, acc_long_T, v0, v1, v2, v3, Qprime, Q
23: st4 { C}, [x11]
```

F Assembly for Interleaved Multi-Layer Butterflies

Algorithm 33 `qq_butterfly_top` for 16 parallel 32-bit Montgomery multiplications.

Inputs:
`a0`, ..., `a3`, `b0`, ..., `b3`, `t0`, ..., `t3`, `mod`, `z0`, `l0`, `h0`, ..., `z3`, `l3`, `h3`, `qS`, `sS`
Roles of the symbols:
Operating registers: `a0`, ..., `a3`, `b0`, ..., `b3`, `t0`, ..., `t3`
Registers with twiddle factors: `z0`, ..., `z3`
Indices for twiddle factors: `l0`, ..., `l3`, `h0`, ..., `h3`
Modulus: `mod`
Specifiers: (`qS`, `sS`) = (`.4S`, `.S`)
Outputs:
$\texttt{t0} = \texttt{b0}\omega_0 \bmod^{\pm} q, \ldots, \texttt{t3} = \texttt{b3}\omega_3 \bmod^{\pm} q$

```
1: mul t0qS, b0qS, z0sS[h0]
2: mul t1qS, b1qS, z1sS[h1]
3: mul t2qS, b2qS, z2sS[h2]
4: mul t3qS, b3qS, z3sS[h3]
5: sqrdmulh b0qS, b0qS, z0sS[l0]
6: sqrdmulh b1qS, b1qS, z1sS[l1]
7: sqrdmulh b2qS, b2qS, z2sS[l2]
8: sqrdmulh b3qS, b3qS, z3sS[l3]
9: mls t0qS, b0qS, modsS[0]
10: mls t1qS, b1qS, modsS[0]
11: mls t2qS, b2qS, modsS[0]
12: mls t3qS, b3qS, modsS[0]
```

Algorithm 34 `qq_butterfly_bot` for 16 parallel 32-bit sub-add pairs.

Inputs:
`a0`, ..., `a3`, `b0`, ..., `b3`, `t0`, ..., `t3`, `mod`, `z0`, `l0`, `h0`, ..., `z3`, `l3`, `h3`, `qS`, `sS`
Roles of the symbols:
Operating registers: `a0`, ..., `a3`, `b0`, ..., `b3`, `t0`, ..., `t3`
Registers with twiddle factors: `z0`, ..., `z3`
Indices for twiddle factors: `l0`, ..., `l3`, `h0`, ..., `h3`
Modulus: `mod`
Specifiers: (`qS`, `sS`) = (`.4S`, `.S`)
Outputs:
$(\texttt{a0}, \texttt{b0}) = (\texttt{a0} + \texttt{t0}, \texttt{a0} - \texttt{t0}), \ldots, (\texttt{a3}, \texttt{b3}) = (\texttt{a3} + \texttt{t3}, \texttt{a3} - \texttt{t3})$

```
1: sub b0qS, a0qS, t0qS
2: sub b1qS, a1qS, t1qS
3: sub b2qS, a2qS, t2qS
4: sub b3qS, a3qS, t3qS
5: add a0qS, a0qS, t0qS
6: add a1qS, a1qS, t1qS
7: add a2qS, a2qS, t2qS
8: add a3qS, a3qS, t3qS
```

Algorithm 35 `qq_butterfly_mixed` for interleaving 16 parallel 32-bit CT butterflies.

Inputs:
$\mathtt{a0}, \ldots, \mathtt{a3}$, $\mathtt{b0}, \ldots, \mathtt{b3}$, $\mathtt{t0}, \ldots, \mathtt{t3}$, $\mathtt{a4}, \ldots, \mathtt{a7}$, $\mathtt{b4}, \ldots, \mathtt{b7}$, $\mathtt{t4}, \ldots, \mathtt{t7}$, $\mathtt{mod}$, $\mathtt{z0}, \mathtt{l0}, \mathtt{h0}, \ldots, \mathtt{z7}, \mathtt{l7}, \mathtt{h7}$, $\mathtt{qS}$, $\mathtt{sS}$

Roles of the symbols:
Operating registers set 1: $\mathtt{a0}, \ldots, \mathtt{a3}, \mathtt{b0}, \ldots, \mathtt{b3}, \mathtt{t0}, \ldots, \mathtt{t3}$
Operating registers set 2: $\mathtt{a4}, \ldots, \mathtt{a7}, \mathtt{b4}, \ldots, \mathtt{b7}, \mathtt{t4}, \ldots, \mathtt{t7}$
Registers with twiddle factors set 1: $\mathtt{z0}, \ldots, \mathtt{z3}$
Registers with twiddle factors set 2: $\mathtt{z4}, \ldots, \mathtt{z7}$
Indices for twiddle factors set 1: $\mathtt{l0}, \ldots, \mathtt{l3}, \mathtt{h0}, \ldots, \mathtt{h3}$
Indices for twiddle factors set 2: $\mathtt{l4}, \ldots, \mathtt{l7}, \mathtt{h4}, \ldots, \mathtt{h7}$
Modulus: $\mathtt{mod}$
Specifiers: $(\mathtt{qS}, \mathtt{sS}) = (\mathtt{.4S}, \mathtt{.S})$

Outputs:
$(\mathtt{a0}, \mathtt{b0}) = (\mathtt{a0} + \mathtt{t0}, \mathtt{a0} - \mathtt{t0}), \ldots, (\mathtt{a3}, \mathtt{b3}) = (\mathtt{a3} + \mathtt{t3}, \mathtt{a3} - \mathtt{t3})$,
$\mathtt{t4} = \mathtt{b4}\omega_4 \bmod^{\pm} q, \ldots, \mathtt{t7} = \mathtt{b7}\omega_7 \bmod^{\pm} q$

```
sub b0qS, a0qS, t0qS
mul t4qS, b4qS, z4sS[h4]
sub b1qS, a1qS, t1qS
mul t5qS, b5qS, z5sS[h5]
sub b2qS, a2qS, t2qS
mul t6qS, b6qS, z6sS[h6]
sub b3qS, a3qS, t3qS
mul t7qS, b7qS, z7sS[h7]
add a0qS, a0qS, t0qS
sqrdmulh b4qS, b4qS, z4sS[l4]
add a1qS, a1qS, t1qS
sqrdmulh b5qS, b5qS, z5sS[l5]
add a2qS, a2qS, t2qS
sqrdmulh b6qS, b6qS, z6sS[l6]
add a3qS, a3qS, t3qS
sqrdmulh b7qS, b7qS, z7sS[l7]
mls t4qS, b4qS, modsS[0]
mls t5qS, b5qS, modsS[0]
mls t6qS, b6qS, modsS[0]
mls t7qS, b7qS, modsS[0]
```

Algorithm 36 `qq_butterfly_mixed_rev` for interleaving 16 parallel 32-bit GS butterflies.

Inputs:
`a0`, ..., `a3`, `b0`, ..., `b3`, `t0`, ..., `t3`, `a4`, ..., `a7`, `b4`, ..., `b7`, `t4`, ..., `t7`, `mod`, `z0`, `l0`, `h0`, ..., `z7`, `l7`, `h7`, `qS`, `sS`
Roles of the symbols:
Operating registers set 1: `a0`, ..., `a3`, `b0`, ..., `b3`, `t0`, ..., `t3`
Operating registers set 2: `a4`, ..., `a7`, `b4`, ..., `b7`, `t4`, ..., `t7`
Registers with twiddle factors set 1: `z0`, ..., `z3`
Registers with twiddle factors set 2: `z4`, ..., `z7`
Indices for twiddle factors set 1: `l0`, ..., `l3`, `h0`, ..., `h3`
Indices for twiddle factors set 2: `l4`, ..., `l7`, `h4`, ..., `h7`
Modulus: `mod`
Specifiers: (`qS`, `sS`) = (`.4S`, `.S`)
Outputs:
(`a4`, `b4`) = (`a4` + `t4`, `a4` − `t4`), ..., (`a7`, `b7`) = (`a7` + `t7`, `a7` − `t7`),
`t0` = `b0`ω_0 mod$^{\pm}$ q, ..., `t3` = `b3`ω_3 mod$^{\pm}$ q

```
 1: mul t0qS, b0qS, z0sS[h0]
 2: sub b4qS, a4qS, t4qS
 3: mul t1qS, b1qS, z1sS[h1]
 4: sub b5qS, a5qS, t5qS
 5: mul t2qS, b2qS, z2sS[h2]
 6: sub b6qS, a6qS, t6qS
 7: mul t3qS, b3qS, z3sS[h3]
 8: sub b7qS, a7qS, t7qS
 9: sqrdmulh b0qS, b0qS, z0sS[l0]
10: add a4qS, a4qS, t4qS
11: sqrdmulh b1qS, b1qS, z1sS[l1]
12: add a5qS, a5qS, t5qS
13: sqrdmulh b2qS, b2qS, z2sS[l2]
14: add a6qS, a6qS, t6qS
15: sqrdmulh b3qS, b3qS, z3sS[l3]
16: add a7qS, a7qS, t7qS
17: mls t0qS, b0qS, modsS[0]
18: mls t1qS, b1qS, modsS[0]
19: mls t2qS, b2qS, modsS[0]
20: mls t3qS, b3qS, modsS[0]
```

Algorithm 37 4 layers of 32-bit CT butterflies over 32 SIMD registers.

Inputs:
Coefficients: $(\texttt{v0}, \ldots, \texttt{v15}) = (a_0, \ldots, a_{15})$
Auxiliary registers: $\texttt{v16}, \ldots, \texttt{v19}$, $\texttt{v28}, \ldots, \texttt{v31}$
Twiddle factors:

$$\texttt{v20} = \texttt{h0}||\texttt{l0}||0||\texttt{Q}$$
$$\texttt{v21} = \texttt{h2}||\texttt{l2}||\texttt{h1}||\texttt{l1}$$
$$\texttt{v22} = \texttt{h4}||\texttt{l4}||\texttt{h3}||\texttt{l3}$$
$$\texttt{v23} = \texttt{h6}||\texttt{l6}||\texttt{h5}||\texttt{l5}$$
$$\texttt{v24} = \texttt{h8}||\texttt{l8}||\texttt{h7}||\texttt{l7}$$
$$\texttt{v25} = \texttt{h10}||\texttt{l10}||\texttt{h9}||\texttt{l9}$$
$$\texttt{v26} = \texttt{h12}||\texttt{l12}||\texttt{h11}||\texttt{l11}$$
$$\texttt{v27} = \texttt{h14}||\texttt{l14}||\texttt{h13}||\texttt{l13}$$

Symbols:

```
L0_0 =  v0, v2,  v4,  v6,  v8, v10, v12, v14
L0_1 =  v1, v3,  v5,  v7,  v9, v11, v13, v15
L1_0 =  v0, v2,  v8,  v10, v4, v6,  v12, v14
L1_1 =  v1, v3,  v9,  v11, v5, v7,  v13, v15
L2_0 =  v0, v4,  v8,  v12, v2, v6,  v10, v14
L2_1 =  v1, v5,  v9,  v13, v3, v7,  v11, v15
L3_0 =  v0, v2,  v4,  v6,  v1, v3,  v5,  v7
L3_1 =  v8, v10, v12, v14, v9, v11, v13, v15
T0   = v16, v17, v18, v19
T1   = v28, v29, v30, v31
mod  = v20
W0   = v20, 2, 3, v20, 2, 3, v20, 2, 3, v20, 2, 3
W1   = v21, 0, 1, v21, 0, 1, v21, 2, 3, v21, 2, 3
W2   = v22, 0, 1, v22, 2, 3, v23, 0, 1, v23, 2, 3
W3_0 = v24, 0, 1, v24, 2, 3, v25, 0, 1, v25, 2, 3
W3_1 = v26, 0, 1, v26, 2, 3, v27, 0, 1, v27, 2, 3
```

Outputs: $(\texttt{v0}, \ldots, \texttt{v15}) = \texttt{NTT}(a_0, \ldots, a_{15})$

```
 1: qq_butterfly_top L0_1, T0, mod, W0
 2: qq_butterfly_bot L0_1, T0, mod, W0
 3: qq_butterfly_top L0_0, T1, mod, W0
 4: qq_butterfly_bot L0_0, T1, mod, W0
 5: qq_butterfly_top L1_1, T0, mod, W1
 6: qq_butterfly_bot L1_1, T0, mod, W1
 7: qq_butterfly_top L1_0, T1, mod, W1
 8: qq_butterfly_bot L1_0, T1, mod, W1
 9: qq_butterfly_top L2_1, T0, mod, W2
10: qq_butterfly_bot L2_1, T0, mod, W2
11: qq_butterfly_top L2_0, T1, mod, W2
12: qq_butterfly_bot L2_0, T1, mod, W2
13: qq_butterfly_top L3_0, T0, mod, W3_0
14: qq_butterfly_bot L3_0, T0, mod, W3_0
15: qq_butterfly_top L3_1, T1, mod, W3_1
16: qq_butterfly_bot L3_1, T1, mod, W3_1
```

Algorithm 38 4 layers of interleaved 32-bit CT butterflies over 32 SIMD registers.

Inputs:
Coefficients: $(\texttt{v0}, \ldots, \texttt{v15}) = (a_0, \ldots, a_{15})$
Auxiliary registers: $\texttt{v16}, \ldots, \texttt{v19}$, $\texttt{v28}, \ldots, \texttt{v31}$
Twiddle factors:

$$\texttt{v20} = \texttt{h0}||\texttt{l0}||0||\texttt{Q}$$
$$\texttt{v21} = \texttt{h2}||\texttt{l2}||\texttt{h1}||\texttt{l1}$$
$$\texttt{v22} = \texttt{h4}||\texttt{l4}||\texttt{h3}||\texttt{l3}$$
$$\texttt{v23} = \texttt{h6}||\texttt{l6}||\texttt{h5}||\texttt{l5}$$
$$\texttt{v24} = \texttt{h8}||\texttt{l8}||\texttt{h7}||\texttt{l7}$$
$$\texttt{v25} = \texttt{h10}||\texttt{l10}||\texttt{h9}||\texttt{l9}$$
$$\texttt{v26} = \texttt{h12}||\texttt{l12}||\texttt{h11}||\texttt{l11}$$
$$\texttt{v27} = \texttt{h14}||\texttt{l14}||\texttt{h13}||\texttt{l13}$$

Symbols:
```
L0_0 =  v0, v2,  v4,  v6,  v8, v10, v12, v14
L0_1 =  v1, v3,  v5,  v7,  v9, v11, v13, v15
L1_0 =  v0, v2,  v8,  v10, v4, v6,  v12, v14
L1_1 =  v1, v3,  v9,  v11, v5, v7,  v13, v15
L2_0 =  v0, v4,  v8,  v12, v2, v6,  v10, v14
L2_1 =  v1, v5,  v9,  v13, v3, v7,  v11, v15
L3_0 =  v0, v2,  v4,  v6,  v1, v3,   v5,  v7
L3_1 =  v8, v10, v12, v14, v9, v11, v13, v15
T0 = v16, v17, v18, v19
T1 = v28, v29, v30, v31
mod = v20
W0   = v20, 2, 3, v20, 2, 3, v20, 2, 3, v20, 2, 3
W1   = v21, 0, 1, v21, 0, 1, v21, 2, 3, v21, 2, 3
W2   = v22, 0, 1, v22, 2, 3, v23, 0, 1, v23, 2, 3
W3_0 = v24, 0, 1, v24, 2, 3, v25, 0, 1, v25, 2, 3
W3_1 = v26, 0, 1, v26, 2, 3, v27, 0, 1, v27, 2, 3
```

Outputs: $(\texttt{v0}, \ldots, \texttt{v15}) = \texttt{NTT}(a_0, \ldots, a_{15})$

```
1: qq_butterfly_top L0_1, T0, mod, W0
2: qq_butterfly_mixed L0_1, T0, L0_0, T1, mod, W0, W0
3: qq_butterfly_mixed L0_0, T1, L1_1, T0, mod, W0, W1
4: qq_butterfly_mixed L1_1, T0, L1_0, T1, mod, W1, W1
5: qq_butterfly_mixed L1_0, T1, L2_1, T0, mod, W1, W2
6: qq_butterfly_mixed L2_1, T0, L2_0, T1, mod, W2, W2
7: qq_butterfly_mixed L2_0, T1, L3_0, T0, mod, W2, W3_0
8: qq_butterfly_mixed L3_0, T0, L3_1, T1, mod, W3_0, W3_1
9: qq_butterfly_bot L3_1, T1, mod, W3_1
```