Barrett Multiplication for Dilithium on Embedded Devices

Studies on the impact of available multiplication instructions

Vincent Hwang[1], YoungBeom Kim[2] and Seog Chung Seo[2]

[1] Max Planck Institute for Security and Privacy, Bochum, Germany
vincentvbh7@gmail.com
[2] Kookmin University, Seoul, Korea
darania@kookmin.ac.kr, scseo@kookmin.ac.kr

Abstract. We optimize the number-theoretic transforms (NTTs) in Dilithium — a digital signature scheme recently standardized by the National Institute of Standards and Technology (NIST) — on Cortex-M3 and 8-bit AVR. The core novelty is the exploration of micro-architectural insights for modular multiplications. Recent work [Becker, Hwang, Kannwischer, Yang and Yang, Volume 2022 (1), Transactions on Cryptographic Hardware and Embedded Systems, 2022] found a correspondence between Montgomery and Barrett multiplications by relating modular reductions to integer approximations and demonstrated that Barrett multiplication is more favorable than Montgomery multiplication by absorbing the subtraction to the low multiplication. We first point out the benefit of Barrett multiplication when long and high multiplication instructions are unavailable, unusable, or slow. We then generalize the notion of integer approximations and improve the emulation of high multiplications used in Barrett multiplication.

Compared to the state-of-the-art assembly-optimized implementations on Cortex-M3, our constant-time NTT/iNTT are $1.38 - 1.51$ times faster and our variable-time NTT/iNTT are 1.10–1.21 times faster. On our 8-bit AVR, we outperform Montgomery-based C implementations of NTT/iNTT by 6.37–7.27 times by simply switching to the proposed Barrett-based implementation. We additionally implement Barrett-based NTT/iNTT in assembly and obtain $14.10 -$ 14.42 times faster code.

For the overall scheme, we provide speed-optimized implementations for Dilithium parameter sets `dilithium2 and dilithium3` on Cortex-M3, and stack-optimized implementations for all parameter sets on Cortex-M3 and 8-bit AVR. We briefly compare the performance of speed-optimized `dilithium3`. Compared to the state-of-the-art assembly implementation on Cortex-M3, our assembly implementation reduces the key generation, signature generation, and signature verification cycles by 2.30%, 23.29%, and 0.69%. In the 8-bit AVR environment, our Barrett-based C implementation reduces the key generation, signature generation, and signature verification cycles by 45.09%, 56.80%, and 50.40%, respectively, and our assembly-optimized implementation reduces the cycles of each operation by 48.85%, 61.70%, and 55.08%, respectively.

Keywords: Modular multiplication · Barrett multiplication · Lattice-based cryptography · Dilithium · Microcontroller · Cortex-M3 · 8-bit AVR

1 Introduction

We optimize the number-theoretic transforms (NTTs) in the digital signature scheme Dilithium, recently standardized by the Nation Institute of Standards and Technology on embedded systems. Our core novelty boils down to efficient instantiations of modular multiplications. Let q be an odd positive integer and $\mathtt{R}$ be a power of two, with exponent a power of two. In cryptography, modular multiplication modulo q is one of the core computations. For efficiency reasons, people commonly instantiate arithmetic modulo q with arithmetic modulo $\mathtt{R}$. In this paper, we focus on the signed arithmetic and define $\mathbb{Z}_n := [-\frac{n}{2}, \frac{n}{2}) \cap \mathbb{Z}$ for a positive integer n. Furthermore, we denote $\bmod^{\pm} n$ the function sending an integer a to the unique integer $a \bmod^{\pm} n$ in $\mathbb{Z}_n$ satisfying $a \equiv a \bmod^{\pm} n \pmod{n}$[1]. We call a multiplication instruction a multiply-low instruction if it multiplies two numbers a, b drawn from $\mathbb{Z}_{\mathtt{R}}$ and computes a value sufficiently close to $ab \bmod^{\pm} \mathtt{R}$. Similarly, we call a multiplication instruction a multiply-high instruction if it computes a value sufficiently close to the upper $\log_2 \mathtt{R}$ bits of ab, and a multiply-long instruction if the result is sufficiently close to ab. For simplicity, we also call the corresponding accumulative/subtractive variants multiply-low, multiply-high, or multiply-long instructions.

For two integers $a, b \in \mathbb{Z}_{\mathtt{R}}$, Montgomery multiplication (accumulative variant) computes ab with a multiply-long and finds a $(2\log_2 \mathtt{R})$-bit integer c with $c \equiv 0 \pmod{\mathtt{R}}$ and $c \equiv ab \pmod{q}$ via the Chinese remainder theorem, which can be implemented by one multiply-low and one multiply-long by applying the divided difference form. Since c is a multiple of $\mathtt{R}$, we compute $\frac{c}{\mathtt{R}}$ by extracting the upper $\log_2 \mathtt{R}$ bits and find $\frac{c}{\mathtt{R}} \equiv ab\mathtt{R}^{-1} \pmod{q}$. If b is a constant known beforehand, we replace b with $b\mathtt{R} \bmod^{\pm} q$ and compute a representative of $ab \bmod^{\pm} q$ [Mon85]. Recently, [Sei18] proposed a subtractive variant of Montgomery multiplication and replaced the multiply-longs by multiply-highs. Barrett multiplication is an alternative approach. Essentially, we compute a "quotient" $\tilde{q}$ with a multiply-high such that $ab - \tilde{q}q$ falls into $\mathbb{Z}_{\mathtt{B}}$ for $q \le \mathtt{B} \le \mathtt{R}$. We then compute the products ab and $\tilde{q}q$ by two multiply-lows and subtract them. In summary, we need two multiply-high/longs and one multiply-low for Montgomery multiplication, and one multiply-high and two multiply-lows for Barrett multiplication. See Table 1 for an overview.

Table 1: Overview of the number of multiplication instructions for each type used in Montgomery and Barrett multiplications. Montgomery multiplication (acc.) stands for the accumulative variant and similarly for the subtractive variant.

	Multiply-low	Multiply-high	Multiply-long
Montgomery multiplication (acc.)	1	0	2
Montgomery multiplication (sub.)	1	2	0
Barrett multiplication	2	1	0

From the micro-architectural point of view, multiply-low instructions are fairly common, but multiply-high and multiply-long instructions might be unavailable, incomplete, or unusable on some platforms. For example, on an ARM Cortex-M3 processor implementing Armv7-M, multiply-long instructions take $3-7$ cycles, depending on the magnitude of the result [ARM10, Table 18-1] and cannot be used in computing secret data. In this case, one usually emulates multiply-high and multiply-long instructions with multiply-low, incurring a significant performance penalty. See Table 2 for the available instructions in the instruction set architecture (ISA) Armv7E-M where Armv7E-M stands for Armv7-M with the DSP extension, and See Table 3 for an overview of the timing of multiplication instructions on Cortex-M3 and Cortex-M4. Additionally, in the AVR environment with an 8-bit register size, our signed implementation of 16-bit multiply-long takes 18 cycles, while our emulation of signed 32-bit multiply-long (using 8 accumulator registers) takes 102 cycles. Therefore, when implementing modular multiplications, avoiding 32-bit multiply-long can lead to significant performance improvements. In other words, Barrett multiplication is obviously more favorable than Montgomery multiplication in both the Cortex-M3 and

[1]The notation $\bmod^{\pm}$ specifies that signed representation is used for defining $\mathbb{Z}_n$.

8-bit AVR.

Table 2: Summary of multiplication instructions for $\texttt{R} = 2^{32}$ in Armv7E-M. Instructions followed by (E) are part of the DSP extension and do not exist in Armv7-M.

Multiply-low	Multiply-high	Multiply-long
`mul`, `mla`, `mla`	`smmul` (E), `smmulr` (E)	`smull`, `smlal`, `umull`, `umlal`

Table 3: Summary of multiplication instruction timings on Cortex-M3 (Armv7-M) and Cortex-M4 (Armv7E-M).

	Cortex-M3	Cortex-M4
`mul`	1	1
`mla/mls`	2	1
`{s, u}{mul, mla, mls }l`	3 − 7	1

Our second observation relies on the approximation nature of Barrett multiplication. Recall that Barrett multiplication computes $ab - \tilde{q}q \in \mathbb{Z}_{\texttt{B}}$. In the literature, people chose $\tilde{q} = \left\lfloor \frac{a \left\lfloor \frac{b\texttt{R}}{q} \right\rceil}{\texttt{R}} \right\rceil$ implementing $\texttt{B} = 2q$ and $\tilde{q} = \left\lfloor \frac{a \left\lfloor \frac{2^k b\texttt{R}}{q} \right\rceil}{2^k \texttt{R}} \right\rceil$ for sufficiently large k implementing $\texttt{B} = q$. We instead relax the bound $\texttt{B}$ and relate the degree relaxation to the quality of approximation implemented by $\tilde{q}$. Conceptually, we throw away the lower-limb computation while computing $\tilde{q}$ by emulating multiply-high and show that the resulting bound $\texttt{B}$ is still good enough for our use case.

Contributions. We summarize our contributions as follows.

- We show that Barrett multiplication is more favorable than Montgomery multiplication when multiply-high and multiply-long instructions are unavailable or unusable.
- We generalize the notion of integer approximations and show that the approximation nature of Barrett multiplication enables efficient emulation of multiply-highs. Along with this generalization, we find that Barrett multiplication performs the same as a long multiplication on Cortex-M3. This shows that any modular multiplication calling at least one long multiplication with non-zero preprocessing/postprocessing cost performs worse than our Barrett multiplication on Cortex-M3.
- We apply our ideas to the post-quantum digital signature Dilithium [ABD+20] recently standardized by NIST [NIS] on Cortex-M3 and 8-bit AVR. Compared to the state-of-the-art optimized Cortex-M3 implementation by [GKS21], our constant-time Barrett-based NTT/iNTT is 1.38 − 1.51 times faster than their Montgomery-based implementation, and our variable-time Barrett-based NTT/iNTT is 1.10 − 1.21 times faster than their variable-time Montgomery-based implementations. In an 8-bit AVR environment, our C implementation of Barrett-based NTT/iNTT are 6.37–7.27 times faster than the reference implementation [ABD+20] using Montgomery arithmetic. Additionally, our assembly implementation maximizes performance improvements (14.10 − 14.42 times faster).
- For the overall scheme, we provide speed-optimized implementations for Dilithium parameter sets `dilithium2 and dilithium3` on Cortex-M3, and stack-optimized implementations for all parameter sets on Cortex-M3 and 8-bit AVR. We briefly compare the performance of `dilithium3`. Compared to the state-of-the-art assembly implementation on Cortex-M3 by [GKS21], our speed-optimized assembly implementation reduces the key generation, signature generation, and signature verification

cycles by 2.30%, 23.29%, and 0.69%. As for the stack-optimized implementation, we compare our C and assembly implementations since there are no publicly available implementations. Our assembly-optimized implementation reduces the cycles of key generation, signature generation, and signature verification of our C implementation by 12.96%, 27.00%, and 22.72%, respectively. For the 8-bit AVR environment, we compare our implementations with the reference C Montgomery-based implementation, since there are no prior works. Our Barrett-based `dilithium3` C implementation reduces the cycles of key generation, signature generation, and signature verification by 45.09%, 56.80%, and 50.40%, respectively, and our assembly-optimized Barrett-based implementation reduces the cycles of key generation, signature generation, and signature verification by 48.85%, 61.70%, and 55.08%, respectively.

Source code. Our source code will be publicly available at `https://github.com/vincentvbh/Barrett_Dilithium_Embedded`.

Structure of this paper. This paper is structured as follows. Section 2 goes through the preliminaries, Section 3 describes our insights on modular multiplications, and Section 4 details our implementations of NTT/iNTT on Cortex-M3 and 8-bit AVR. Finally, Section 5 shows the performance numbers of NTT/iNTT and the overall impact on Dilithium.

2 Preliminiaries

Section 2.1 describes Dilithium [ABD+20], Section 2.2 reviews number-theoretic transform, Section 2.3 reviews integer approximations from [BHK+22], Section 2.4 reviews modular multiplications. Section 2.5 describes Cortex-M3, and Section 2.6 describes 8-bit AVR.

2.1 Dilithium

Dilithium [ABD+20] is a digital signature recently standardized by NIST [NIS]. It is based on the Module Small Integer Solutions (M-SIS) and the Module Learning With Errors (M-LWE) problems. The module is a $k \times \ell$ matrix over the polynomial ring $\mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ where $q = 2^{23} - 2^{13} + 1$ is a prime and $(k, \ell) = (4, 5), (6, 5), (8, 7)$, depending on the security level. Please see Table 4 for an overview of parameter sets and [ABD+20] for algorithm description.

Table 4: Dilithium parameter sets.

Parameter set	NIST security level	(k, ℓ)
`dilithium2`	II	$(4, 4)$
`dilithium3`	III	$(6, 5)$
`dilithium5`	V	$(8, 7)$

The core operation of key generation, signature generation, and signature verification is the $(k \times \ell) \times (\ell \times 1)$ matrix-to-vector multiplications. Dilithium builds NTT into the specification – the matrix is sampled as if all the entries are ready in the domain of NTT.

2.2 Number Theoretic Transform

Let R be a commutative ring with identity. For an n-th root of unity $\omega_n \in R$, we call it principal if $\forall j = 1, \ldots, n-1, \sum_{i=0}^{n-1} \omega_n^{ij} = 0$. For an invertible element $\zeta \in R$, we have the following isomorphism by the Chinese remainder theorem for polynomial rings:

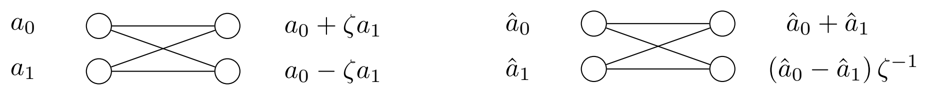
$$\frac{R[x]}{\langle x^n - \zeta^n \rangle} \cong \prod_i \frac{R[x]}{\langle x - \zeta\omega_n^i \rangle}.$$

If $n = 2^h$ is a power of two, we have

$$\frac{R[x]}{\langle x^n - \zeta^n \rangle} \cong \prod \frac{R[x]}{\left\langle x^{\frac{n}{2}} \pm \zeta^{\frac{n}{2}} \right\rangle} \cong \cdots \cong \prod_{i_0,\dots,i_{h-1}=0,1} \frac{R[x]}{\left\langle x - \zeta\omega_n^{\sum_j i_j 2^j} \right\rangle}.$$

This is the radix-2 Cooley–Tukey FFT for a discrete weighted transform [CT65, CF94]. In this paper, we specialize it to $\zeta = \omega_{2n}$ so $\zeta^n = -1$.

We give an example for $n = 2$. Let's define $\mathrm{CT}(-,-,\zeta)$ as the function $(a_0, a_1) \mapsto (a_0 + \zeta a_1, a_0 - \zeta a_1)$. Clearly, we can implement the isomorphism $R[x]/\langle x^2 - \zeta^2 \rangle \cong \prod R[x]/\langle x \pm \zeta \rangle$ as $a_0 + a_1 x \mapsto \mathrm{CT}(a_0, a_1, \zeta)$. If we further define $\mathrm{GS}(-,-,\zeta) \coloneqq (\hat{a}_0, \hat{a}_1) \mapsto (\hat{a}_0 + \hat{a}_1, (\hat{a}_0 - \hat{a}_1)\,\zeta)$, then $\mathrm{GS}(-,-,\zeta^{-1}) \circ \mathrm{CT}(-,-,\zeta) = \mathrm{CT}(-,-,\zeta) \circ \mathrm{GS}(-,-,\zeta^{-1}) = (a_0, a_1) \mapsto 2(a_0, a_1)$ where $\circ$ stands for function composition. Generally, we call $\mathrm{CT}(-,-,-)$ a Cooley-Tukey butterfly (CT butterfly) and $\mathrm{GS}(-,-,-)$ a Gentleman-Sande butterfly (GS butterfly). See Figure 1 for illustrations.



(a) Cooley–Tukey butterfly $\mathrm{CT}(a_0, a_1, \zeta)$. (b) Gentleman–Sande butterfly $\mathrm{GS}(\hat{a}_0, \hat{a}_1, \zeta^{-1})$.

Figure 1: Radix-2 butterflies, adapted from [AHY22].

Since a size-n Cooley–Tukey FFT can be implemented entirely with CT butterflies, we can invert the computation by replacing all CT butterflies with GS butterflies and canceling out the scaling at the end. Gentleman–Sande FFT [GS66] proceeds in a different way – we first convert $R[x]/\langle x^m - \psi^m \rangle$ into $R[y]/\langle y^m - 1 \rangle$ whenever $\psi^m \neq 1$ and split with $R[y]/\langle y^m - 1 \rangle \cong \prod R[y]/\left\langle y^{\frac{m}{2}} \pm 1 \right\rangle$. Since the result of Cooley-Tukey FFT for $R[x]/\langle x^n - \zeta^n \rangle$ can be implemented entirely with GS butterflies, we can also invert it with CT butterflies. The benefit for inverting with CT butterflies instead of GS butterflies when $\zeta^n \neq 1$ is that while multiplying by n^{-1} for canceling out the scaling, we can merge $n - 1$ of them with multiplications by ζ^{-i} if we implement with CT butterflies whereas implementing with GS butterflies only allow us to merge $\frac{n}{2}$ of them with ζ^{-i}. We refer to [ACC⁺22, Figure 1] for illustrations.

2.3 Integer Approximation

For a function $[\![\,]\!] : \mathbb{R} \to \mathbb{Z}$, [BHK+22] call it an integer approximation if

$$\forall r \in \mathbb{R}, |r - [\![r]\!]| \leq 1.$$

Common examples are the floor function $\lfloor\rfloor$, ceiling function $\lceil\rceil$, and rounding-half-up function $\lfloor\rceil$. [BHK+22] chose $\lfloor\rceil_2 \coloneqq r \mapsto 2\left\lfloor\frac{r}{2}\right\rceil$ and demonstrated its benefit for the vector instruction set Neon in Armv8-A. It is easily seen that $\lfloor\rfloor, \lceil\rceil, \lfloor\rceil, \lfloor\rceil_2$ are all integer approximations.

2.4 Modular Multiplications

Throughout this paper, we consider $\mathtt{R} = 2^{32}$ and $q < \frac{\mathtt{R}}{2}$ an odd number, and focus on signed arithmetic. For an integer approximation $[\![\,]\!]$, we define the corresponding modular reduction $\mathrm{mod}^{[\![\,]\!]} q : \mathbb{Z} \to \mathbb{Z}$ as

$$\mathrm{mod}^{[\![\,]\!]} q \coloneqq z \mapsto z - \left[\!\!\left[\frac{z}{q} \right]\!\!\right] q.$$

Furthermore, we define $\left|\mathrm{mod}^{[\![\,]\!]} q\right| \coloneqq \max_{z \in \mathbb{Z}} \left| z \bmod^{[\![\,]\!]} q \right|$. If $[\![\,]\!] = \lfloor\,\rceil$, we denote $\mathrm{mod}^{\lfloor\,\rceil}$ as $\mathrm{mod}^{\pm}$.

Montgomery multiplication. Let $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right)$ be two integers. Montgomery multiplication [Mon85, Sei18] computes a representative of $ab\mathtt{R}^{-1} \bmod q$ as

$$\frac{ab + \left(ab(-q^{-1}) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \equiv ab\mathtt{R}^{-1} \pmod{q}.$$

If b is known in prior, we replace b with $b\mathtt{R} \bmod^{\pm} q$ and compute

$$\frac{a\left(b\mathtt{R} \bmod^{\pm} q\right) + \left(a\left(b\mathtt{R} \bmod^{\pm} q\right)\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \equiv ab \pmod{q}.$$

Since $\left|\frac{a\left(b\mathtt{R} \bmod^{\pm} q\right) + \left(a\left(b\mathtt{R} \bmod^{\pm} q\right)\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}\right| \leq \frac{|a|\left|\bmod^{\pm} q\right| + \left|\bmod^{\pm} \mathtt{R}\right| q}{\mathtt{R}} = \frac{q}{2}\left(1 + \frac{|a|}{\mathtt{R}}\right)$, the result is a 32-bit value.

In [Sei18], they proposed the following subtractive variant for vector arithmetic:

$$\left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \right\rfloor.$$

This sometimes improves the overall performance since we don't need to keep track of the lower $\log_2 \mathtt{R}$ bits of the products.

Barrett multiplication. Barrett multiplication was first introduced only for the reduction form [Bar86, Sei18] – For reducing a value a, we compute with

$$a - \left\lfloor \frac{a \left\lfloor \frac{\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil q.$$

[Sho] proposed the multiplicative form for unsigned arithmetic, and [BHK+22] proposed the signed multiplication with integer approximations. [BHK+22] computed a representative of $ab \bmod q$ as

$$ab - \left\lfloor \frac{a \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil q.$$

[BHK+22] showed that the result is a 32-bit value by establishing a correspondence between Barrett and Montgomery multiplications.

A correspondence between Barrett and Montgomery multiplications. [BHK+22] showed that for an integer approximation $[\![\,]\!]$, we have

$$ab - \left\lfloor \frac{a \left[\!\!\left[\frac{b\mathtt{R}}{q} \right]\!\!\right]}{\mathtt{R}} \right\rceil q = \frac{a\left(b\mathtt{R} \bmod^{[\![\,]\!]} q\right) + \left(a\left(b\mathtt{R} \bmod^{[\![\,]\!]} q\right)\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}.$$

Their proof clearly transfers to the following generalization: For integer approximations $\llbracket\rrbracket_0, \llbracket\rrbracket_1$, we have

$$ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q = \frac{a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right) + \left(a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right)\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}}.$$

We leave the justification to readers since it is completely routine by unfolding the definitions. The correspondence allows us to argue the output range as follows:

$$\left| ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \right| \leq \frac{|a| \left|\bmod^{\llbracket\rrbracket_0} q\right| + \left|\bmod^{\llbracket\rrbracket_1} \mathtt{R}\right| q}{\mathtt{R}}.$$

If $\llbracket\rrbracket_0 = \llbracket\rrbracket_1 = \lfloor\rceil$, we have $\left| ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \right| \leq \frac{q}{2}\left(1 + \frac{|a|}{\mathtt{R}}\right)$.

2.5 Cortex-M3

Cortex-M3 implements the instruction set architecture Armv7-M. We briefly describe relevant instructions in Armv7-M [ARM21] and their timing on Cortex-M3 [ARM10]. `add` adds up two 32-bit values and `sub` subtracts them. `adc` and `sbc` add and subtract the values with carry. `lsl` and `lsr` logically shift a 32-bit value left and right by the specified constant/register value. `asr` performs an arithmetic right-shift. `ubfx` extracts certain consecutive bits and unsigned-extends the result to a 32-bit value. `sbfx` signed-extends the result to a 32-bit value. Each of the above instructions takes one cycle (we exclude the instruction timing involving PC operands). `mul` multiplies two 32-bit values, `mla` accumulates the product to the accumulator, and `mls` subtracts the product from the accumulator. `mul` takes one cycle and `mla/mls` takes two cycles. `{u, s}mull` computes the 64-bit unsigned/signed product of two 32-bit values, and `{u, s}mlal` accumulates the product to an accumulator. `{u, s}{mul, mla}l` takes 3 to 7 cycles and is input-dependent [ARM10, Table 18-1].

2.6 8-Bit AVR

The 8-bit AVR microcontroller architecture employs a straightforward two-stage pipeline. Most of its instructions execute in a single cycle. Internally, it is equipped with 32 general-purpose 8-bit registers, designated as `[r0:r31]`. Due to this, basic arithmetic operations, including bit operations, are performed on 8-bit units. We briefly describe relevant instructions in 8-bit AVR environment [Atm16]. Analogous to the fundamental Cortex-M3, it supports 8-bit unit operations like `add`, `sub`, `adc`, and `sbc`. `lsl` and `lsr` logically shift an 8-bit value one bit to the left and right, respectively. `asr` performs an arithmetic 1-bit right-shift. Each of the above instructions takes one cycle. Excluding the early AVR architectures like the ATtiny series, which possess byte-sized Static RAM (SRAM), the AVR microcontrollers primarily accommodate multiplication instructions via a dedicated hardware multiplication unit. The product of the multiplication is always returned in `[r0:r1]`. `mul` multiplies two unsigned 8-bit values, while `muls` multiplies two signed 8-bit values. `mulsu` multiplies 8-bit signed and unsigned value. These multiplication instructions take two cycles. Unlike `mul`, which allows all registers as operands, both `muls` and `mulsu` mandate the use of registers within the `[r16:r31]` range as operands.

3 Barrett Multiplication: Revisited

3.1 Integer Approximation: Revisited

We generalize the notion of integer approximations. For a function $\llbracket\rrbracket : \mathbb{R} \to \mathbb{Z}$, we call it an integer approximation if

$$\exists \delta \in \mathbb{R}_{>0}, \forall r \in \mathbb{R}, |r - \llbracket r \rrbracket| \leq \delta.$$

When δ is known, we call $\llbracket\rrbracket$ a δ-integer-approximation. The generalizations of $\bmod^{\llbracket\rrbracket}$ and $\left|\bmod^{\llbracket\rrbracket} q\right|$ are defined in the same way.

3.2 Barrett Multiplication with Approximated High Products

Let $a \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right), b \in \left[-\frac{q}{2}, \frac{q}{2}\right)$ be integers. Recall that Barrett multiplication computes a representative of $ab \bmod q$ as

$$ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q$$

for integer approximations $\llbracket\rrbracket_0, \llbracket\rrbracket_1$. We choose $\llbracket\rrbracket_0 = \lfloor\rceil$ and $\llbracket\rrbracket_1$ the following integer approximation:

$$\forall r \in \mathbb{R}, \llbracket r \rrbracket_1 := \left\lfloor \frac{a_l b_h}{\sqrt{\mathtt{R}}} \right\rfloor + \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor + a_h b_h$$

where $a_l + a_h\sqrt{\mathtt{R}} = \frac{r\mathtt{R}}{\left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}$, $b_l + b_h\sqrt{\mathtt{R}} = \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0$ and $a_l, b_l \in [0, \sqrt{\mathtt{R}})$. Observe that

$$\forall r \in \mathbb{R}, |\llbracket r \rrbracket_1 - \lfloor r \rceil| \leq 3,$$

we have

$$\left| ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \right| \leq \left| ab - \left\lfloor \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rceil q \right| + 3q \leq \frac{q}{2}\left(7 + \frac{|a|}{\mathtt{R}}\right).$$

Therefore, computing with $ab - \left\llbracket \frac{a \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q$ is tolerable as long as $\frac{q}{2}\left(7 + \frac{|a|}{\mathtt{R}}\right) < \frac{\mathtt{R}}{2}$. In Dilithium, this is the case since $q = 2^{23} - 2^{13} + 1$ and $\mathtt{R} = 2^{32}$. The benefit is that $\llbracket\rrbracket_1$ is faster than $\lfloor\rceil$ if we have to emulate them with $\frac{\log_2 \mathtt{R}}{2} = \frac{\log_2 \mathtt{R}}{2} \times \frac{\log_2 \mathtt{R}}{2}$ multiply-low instructions. The same argument holds if we only have $\frac{\log_2 \mathtt{R}}{4} = \frac{\log_2 \mathtt{R}}{4} \times \frac{\log_2 \mathtt{R}}{4}$ multiply-low instructions.

4 Implementations

Section 4.1 details our implementations of modular multiplications and Section 4.2 describes our layer-merging strategies.

4.1 Modular Multiplications

We first describe our implementations of modular multiplications. Section 4.1.1 describes our Cortex-M3 implementations and Section 4.1.2 describes our AVR implementations.

4.1.1 Cortex-M3

For our Armv7-M implementation on Cortex-M3, we detail our variable-time and constant-time Barrett multiplications and compare them with the state-of-the-art Montgomery multiplications by [GKS21]. Table 5 summarizes the performance cycles of various multiplication operations. Since our Barrett multiplication, along with our observation of the approximation nature, performs the same as the emulated long multiplication, we find that: **Our Barrett multiplication outperforms any modular multiplication algorithm calling an emulated long multiplication followed by a reduction subroutine with non-zero cost on Cortex-M3.**

Table 5: Overview of multiplication operations on Cortex-M3. All implementations are constant-time unless stated otherwise. We assume that inputs are 32-bit register values. Further optimizations may be applied by merging the decomposition with the memory operations. However, this complicates the comparisons when the layer-merging technique is applied on Cortex-M3.

Multiplication operation	Work	Cycle
Long (variable-time)	[ARM10]	3 − 5
Long (constant-time)	[GKS21]	12
Montgomery multiplication (variable-time)	[GKS21]	7 − 11
Montgomery multiplication (constant-time)	[GKS21]	23
Barrett multiplication (variable-time)	This work	6 − 8
Barrett multiplication (constant-time)	This work	12

Variable-time Barrett multiplication. For 32-bit values a, b with $b \in \left\{-\frac{q}{2}, \dots, \frac{q}{2}\right\}$ known, we precompute $\left\lfloor \frac{2^{32}b}{q} \right\rceil$ and compute

$$ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rfloor q$$

as a representative of $ab \bmod q$. This is our variable-time Barrett multiplication and Algorithm 1 is an illustration. Since the images of $\lfloor\rfloor$ and $\lfloor\rceil$ differ by at most 1, we have

$$\left| ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rfloor q \right| \le \left| ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rceil q \right| + q \le \frac{q}{2}\left(3 + \frac{|a|}{2^{32}}\right).$$

Comparing to the state-of-the-art variable-time Montgomery multiplication [GKS21, ACC+21] (cf. Algorithm 2), Barrett multiplication turns the `smlal` into an `mls`. Since `smlal` takes 3 to 7 cycles and `mls` takes only two cycles, Barrett multiplication is obviously faster.

Algorithm 1 Variable-time Barrett multiplication on Cortex-M3.

Inputs: $\texttt{a} = a, \texttt{b} = b, \texttt{bhi} = \left\lfloor \frac{2^{32}b}{q} \right\rceil$.

Outputs: $\texttt{c} = ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rfloor q$.

```
1: smull lo, hi, a, bhi      ▷ $\texttt{lo} + \texttt{hi} \cdot 2^{32} = a \left\lfloor \frac{2^{32}b}{q} \right\rceil$.
2: mul   c,  a, b            ▷ $\texttt{c} = ab \bmod {}^{\pm}2^{32}$.
3: mls   c, hi, $q$, c       ▷ $\texttt{c} = ab - \left\lfloor \frac{a \left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right\rfloor q$.
```

Algorithm 2 Variable-time Montgomery multiplication on Cortex-M3 [GKS21, ACC+21].

```
Inputs: a = $a$, b = $b$.
Outputs: hi = $\frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}}$.
1: smull lo, hi, a, b                        ▷ lo + hi · $2^{32} = ab$.
2: mul   lo, lo, $-q^{-1} \bmod^{\pm} 2^{32}$        ▷ lo = $-abq^{-1} \bmod^{\pm} 2^{32}$.
3: smlal lo, hi, lo, $q$                     ▷ hi = $\frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}}$.
```

Constant-time Barrett multiplication. For the constant-time Barrett multiplication, we again precompute $\left\lfloor \frac{2^{32} b}{q} \right\rceil$ if b is known. Then, we compute

$$ab - \left[\!\left[\frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right]\!\right]_1 q$$

as a representative of $ab \bmod q$. Algorithm 3 is an illustration. For computing $\left[\!\left[\frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right]\!\right]_1$, we implement the macro `mulhi_split` as shown in Algorithm 7. In summary, our constant-time Barrett multiplication takes 10 cycles whereas constant-time Montgomery multiplication takes 19 cycles (cf. Algorithm 4).

Algorithm 3 Constant-time Barrett multiplication on Cortex-M3.

```
Inputs: a = $a$, b = $b$, blo + bhi · $2^{16} = \left\lfloor \frac{2^{32} b}{q} \right\rceil$.
Outputs: t3 = $ab - \left[\!\left[ \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right]\!\right]_1 q$.
1: mul         t3,  a,   b                    ▷ t3 = $ab \bmod^{\pm} 2^{32}$.
2: ubfx        t0,  a,  #0, #16
3: asr          a,  a, #16                    ▷ t0 + a · $2^{16} = a$.
4: mulhi_split t1,  a, bhi, t0, blo,  t2      ▷ t1 = $\left[\!\left[ \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right]\!\right]_1$.
5: mls         t3, t1,   $q$,  t3               ▷ t3 = $ab - \left[\!\left[ \frac{a \left\lfloor \frac{2^{32} b}{q} \right\rceil}{2^{32}} \right]\!\right]_1 q$.
```

Algorithm 4 Constant-time Montgomery multiplication on Cortex-M3 [GKS21].

```
Inputs: al + ah · $2^{16} = a$, bl + bh · $2^{16} = b$.
Outputs: res = $\frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}}$.
1: sbsmull al, res,   al,  ah, bl, bh, tmp0
                        ▷ al + res · $2^{32} = ab$, 7 cycles [GKS21, Listing 5].
2: mul     bh,  al, $-q^{-1} \bmod^{\pm} 2^{32}$            ▷ bh = $-abq^{-1} \bmod^{\pm} 2^{32}$.
3: ubfx    bl,  bh,   #0, #16
4: asr     bh,  bh,  #16                      ▷ bl + bh · $2^{16} = -abq^{-1} \bmod^{\pm} 2^{32}$.
5: sbsmlal al, res,   bl,  bh, ql, qh, tmp0
                        ▷ res = $\frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}}$, 9 cycles [GKS21, Listing 6].
```

4.1.2 8-Bit AVR

Since there is no research for Dilithium in an 8-bit AVR environment, we directly implement and compare Montgomery and Barrett multiplications. Specifically, our work encompasses the full range of 16/32-bit multiply-low/high/long macros operating at the granular level of 8-bit words. In summary, our constant-time Barrett multiplication takes 129 cycles, whereas constant-time Montgomery multiplication takes 184 cycles on an 8-bit AVR environment. Table 6 gives an overview of the multiplication operations.

Table 6: Overview of multiplication operations on 8-bit AVR.

Multiplication operation	Work	Cycle
`mulsu_16x16_32`	[Ret21]	17
`muls_16x16_32`	[Ret21]	18
`muls32xQ_lo32`	This work	23
`muls32x32_lo32`	[Ret21]	36
`muls32xQinv_lo32`	This work	27
`muls32xQ_hi32`	This work	51
`muls32x32_64`	[Ret21]	102
Montgomery multiplication	This work	184
Barrett multiplication	This work	129

For the Barrett multiplication (cf. Algorithm 5), we implement two 16-bit multiply-long macros (`muls16x16_32` and `mulsu16x16_32`) and two 32-bit multiply-low macros (`muls32xQ_lo32` and `muls32x32_lo32`). We implement the multiply instructions by leveraging the "Move-and-Add" (MA) technique, as proposed in [LSSR+15] and referenced in [Ret21]. The macro `muls16x16_32` multiplies two signed 16-bit values, and macro `mulsu16x16_32` multiplies 16-bit signed and unsigned value. If one of the operands of 16-bit multiply-long is unsigned, we can save one `sbc` instruction. Thus, `mulsu_16x16_32` (17 cycles) is 1 cycle faster than `muls16x16_32` (18 cycles). We further optimize the 32-bit multiply-low when q is one of the operands by observing that the least significant 8-bit of q are 1's. Since the least significant word is 1, there's no need to operate carry propagation due to the signed representation. As a result, we implement solely using the `mul` commands (no `muls` and `mulsu`). Please see Algorithm 8 for an illustration. Our `muls32xQ_lo32` (23 cycles) is 13 cycles faster than the generic `muls32x32_lo32` (36 cycles).

Algorithm 5 Constant-time Barrett multiplication on 8-bit AVR.

```
Inputs: (a3‖ ··· ‖a0) = $a$, (b3‖ ··· ‖b0) = $b$, (bp3‖ ··· ‖bp0) = $\left\lfloor \frac{2^{32}b}{q} \right\rceil$.
        ((a3‖a2)= $a_h$, (a1‖a0)= $a_l$, (bp3‖bp2)= $b_h$, (bp1‖bp0)= $b_l$)
Outputs: (c3‖ ··· ‖c0) = $c = ab - \left[\!\left[ \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_1 q$.
 1: muls16x16_32   a2, a3, bp2, bp3, c0, ···, c3        ▷ $c = a_h b_h$.
 2: mulsu16x16_32  bp2, bp3, a0, a1, t0, ···, t3        ▷ $t = a_l b_h$.
 3: mov r0, t3     lsl r0       sbc r0, r0              ▷ r0 = SignExtend(t3[7 : 7]).
 4: add c0, t2     adc c1, t3   adc c2, r0   adc c3, r0 ▷ $c = a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor$.
 5: mulsu16x16_32  a2, a3, bp0, bp1, t0, ···, t3        ▷ $t = a_h b_l$.
 6: mov r0, t3     lsl r0       sbc r0, r0              ▷ r0 = SignExtend(t3[7 : 7]).
 7: add c0, t2     adc c1, t3   adc c2, r0   adc c3, r0
        ▷ $c = a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor + \left\lfloor \frac{a_h b_l}{2^{16}} \right\rfloor = \left[\!\left[ \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_1$.
 8: muls32xQ_lo32  c0, ···, c3, qimm, t0, ···, t3       ▷ $t = \left[\!\left[ \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_1 q$, 23 cycles
 9: muls32x32_lo32 a0,···,a3, b0,···,b3, c0,···,c3      ▷ $r = ab \bmod {}^{\pm}2^{32}$.
10: sub c0, t0     sbc c1, t1   sbc c2, t2   sbc c3, t3
        ▷ (c3‖ ··· ‖c0) = $c = ab - \left[\!\left[ \frac{a\left\lfloor \frac{2^{32}b}{q} \right\rceil}{2^{32}} \right]\!\right]_1 q$.
```

Conversely, Montgomery multiplication based on [Sei18] (cf. Algorithm 6) requires one 32-bit multiply-low, one 32-bit multiply-high, and one 32-bit multiply-long macros. Since the least significant word of q and $q^{-1} \bmod {}^{\pm}2^{32}$ in Dilithium is 1, we implement the 32-bit multiply-low/high instructions (`muls32xQinv_lo32` and `muls32xQ_hi32`) similarly to Algorithm 8. Macros `muls32xQinv_lo32` and `muls32xQ_hi32` take 27 and 51 cycles, respectively. As the 64-bit ab needs to hold be in the register, we use row-wise multiplication [GPW+04] technique for 32-bit multiply-long instructions. The macro `muls32x32_64` takes 102 cycles.

Algorithm 6 Our constant-time Montgomery multiplication on 8-bit AVR adapted from [Sei18].

```
Inputs: (a3‖ ··· ‖a0) = $a$, (b3‖ ··· ‖b0) = $b$
Outputs: (r7‖ ··· ‖r4) = $\frac{ab-\left(abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
 1: muls32x32_64    a0, ···, a3, b0, ···, b3, r0, ···, r8    ▷ $r = ab$.
 2: muls32xQinv_lo32  r0, ···, r3, qiimm, t0, ···, t3
        ▷ $t = abq^{-1} \bmod {}^{\pm}2^{32}$, 27 cycles.
 3: muls32xQ_hi32   t0, ···, t3, qimm, t4, ···, t7
        ▷ $t = \frac{\left(abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$, 51 cycles.
 4: sub r4, t4      sbc r5, t5   sbc r6, t6   sbc r7, t7
        ▷ (r7‖ ··· ‖r4) = $\frac{ab-\left(abq^{-1} \bmod {}^{\pm}2^{32}\right)q}{2^{32}}$.
```

4.2 Number Theoretic Transforms

Section 4.2.1 describes our Cortex-M3 NTT/iNTT implementations and Section 4.2.2 describes our AVR NTT/iNTT implementations.

4.2.1 Cortex-M3

We employ the same 2-2-2-2 layer merging strategy for variable-time NTT/iNTT as [GKS21]. As for the constant-time NTT, we compute one layer at a time due to the high register pressure. For the constant-time iNTT, the first three layers are merged as follows: we load four coefficients, apply all butterflies, load the other four coefficients, and apply all butterflies with twiddle factors 1 or ω_4. The butterflies with twiddle factors ω_8 and ω_8^3 are computed separately. For the remaining layers, we compute one layer at a time.

4.2.2 8-Bit AVR

Unlike the ARM architecture, in the 8-bit AVR environment, the displacement of the load indirect (`ldd`) instruction is limited to [0, 63]. Thus, one can access data up to 64 bytes from a base address. Accessing addresses beyond this range requires an additional 2 cycles (`adiw`). Furthermore, given the 32 registers, excluding the address register, there isn't sufficient space to store 4 coefficients and temporary values for merging 2 layers. As a result, in the AVR implementation, we do not employ a merging strategy and compute one layer at a time. For both NTT and iNTT, we utilize the CT butterfly. The Twiddle factors required for NTT, iNTT, and twisting are all stored in the flash memory.

5 Results

We present the performance numbers in this section. Section 5.1 describes our benchmarking environment, Section 5.2 shows the performance of NTTs/iNTTs, and Section 5.3 summarizes the overall performance of Dilithium.

5.1 Benchmarking Environment

Cortex-M3. We benchmark our Armv7-M implementations on a `nucleo-f207zg` board containing a `stm32f207zg` core with 128 KiB of SRAM and 1 MB of flash memory. According to [STM20, Sections 3.2 and 3.6], `stm32f207zg` provides accesses to SRAM and flash memory with 0 wait state up to the frequency 120 MHz. Nevetheless, we follow the literature [ACC+22] and benchmark at frequency 30 MHz for consistency. We compile our code with the cross-compiler `arm-none-eabi-gcc` version `10.3.1`.

8-bit AVR. We benchmark our 8-bit AVR implementation using the `IAR Embedded Workbench`. We simulate them on the `Generic Devices -v6` option with Max 16 MB of SRAM and 8 MB of flash memory. We compile our AVR codes with the compiler of `IAR Embedded Workbench` version `8.10.1` using `High(speed)` level optimization option. Since 8-bit AVR comprise the single-pipeline structure, our simulations provide cycle counts equivalent to the benchmarks. To measure the stack size, we use the linker option (`Enable stack usage analysis`) of `IAR Embedded Workbench`, and the code size is measured through the information in the `.map` file.

5.2 Performance of Number Theoretic Transform

We first describe the performance of NTT/iNTT.

5.2.1 Cortex-M3

For Cortex-M3 implementations, we compare to the state-of-the-art assembly-optimized implementations by [GKS21]. Table 7 summarizes the performance of NTT/iNTT on

Cortex-M3. We compare our negacyclic NTT to for the NTT by [GKS21], and the sum of cyclic iNTT and twisting to the iNTT by [GKS21]. Table 7 summarizes the numbers.

Our variable-time NTT and iNTT are $\frac{19347}{15985} \approx 1.21\times$ and $\frac{21006}{14117+4950} = \frac{21006}{19067} \approx 1.10\times$ faster than [GKS21], respectively. For constant-time implementations, our NTT and iNTT are $\frac{33025}{21876} \approx 1.51\times$ and $\frac{36609}{19782+6742} = \frac{36609}{26524} \approx 1.38\times$ faster than [GKS21], respectively.

Table 7: Performance numbers of NTT/iNTT on Cortex-M3.

Function	[GKS21]		This work	
NTT				
	Variable-time	Constant-time	Variable-time	Constant-time
NTT (negacyclic)	19 347	33 025	15 985	21 876
iNTT				
	Variable-time	Constant-time	Variable-time	Constant-time
iNTT (negacyclic)	21 006	36 609	-	-
iNTT (cyclic)	-	-	14 117	19 782
Point mul.	-	-	4 950	6 742

5.2.2 8-Bit AVR

Since our implementation is the first work of Dilithium in the AVR environment, we compare it with the NIST reference implementation [ABD+20]. Table 8 shows the comparison of the reference implementation using Montgomery arithmetic and our implementation using Barrett arithmetic. The results for our iNTT include ring twisting cycles. Our AVR Assembly implementation is a Constant-time implementation.

Our C-based NTT and iNTT are $\frac{2860881}{449457} \approx 6.37\times$ and $\frac{3402491}{468207} \approx 7.27\times$ faster than [ABD+20], respectively. For our hand-written assembly implementation, our NTT and iNTT are $\frac{2860881}{202917} \approx 14.10\times$ and $\frac{3402491}{236028} \approx 14.42\times$ faster than [ABD+20].

Table 8: Performance numbers of NTT/iNTT on 8-bit AVR.

Function	[ABD+20]		This work	
NTT				
	C	ASM	C	ASM
NTT (negacyclic)	2 860 881	-	449 457	202 917
iNTT				
	C	ASM	C	ASM
iNTT (negacyclic)	3 402 491	-	-	-
iNTT (cyclic+Point mul.)	-	-	468 207	236 028

5.3 Performance of Scheme

This section describes the overall performance of Dilithium on Cortex-M3 and 8-bit AVR.

5.3.1 Cortex-M3

We compare the overall Cortex-M3 performance of Dilithium to existing works [GKS21, BRS22]. For `dilithium2` and `dilithium3`, we provide three implementations: (i) a speed-optimized implementation using assembly Barrett-based NTTs/iNTT, (ii) a stack-optimized C implementation partially based on the stack optimizations proposed by [BRS22], and (iii) a stack-optimized implementation using assembly Barrett-based NTTs/iNTTs. For the speed-optimized implementations, we simply replace the assembly-optimized Montgomery-based NTTs/iNTTs by [GKS21] with our assembly Barrett-based NTTs/iNTTs. As for the stack-optimized implementations, we gradually apply memory optimization techniques from [BRS22] to the reference implementation until we can run the implementations on our platform. We then deploy our assembly Barrett-based NTTs/iNTTs. For `dilithium5`, we only provide stack-optimized implementations due to the large stack usage.

We first compare our speed-optimized assembly implementations to [GKS21]. For `dilithium2`, we reduce the key generation, signature generation, and signature verification cycles by 3.61%, 7.42%, and 1.5%, respectively. For `dilithium3`, we reduce the key generation, signature generation, and signature verification cycles by 2.3%, 23.29%, and 0.69%, respectively.

Next, we compare the performance of stack-optimized implementations. The most stack-optimized implementation is by [BRS22]. However, we cannot find publicly available source code. So, we start by applying memory optimization from [BRS22] to the C reference implementation until we can run the parameter set `dilithium5`. Therefore, comparisons to [BRS22] will be unfair, and we compare our own stack-optimized C and assembly implementations. For `dilithium2`, our assembly-optimized reduces the key generation, signature generation, and signature verification cycles by 14.74%, 27.45%, and 24.68%, respectively. For `dilithium3`, our assembly-optimized reduces the key generation, signature generation, and signature verification cycles by 12.96%, 27.00%, and 22.72%, respectively. For `dilithium5`, our assembly-optimized reduces the key generation, signature generation, and signature verification cycles by 11.71%, 23.75%, and 19.69%, respectively.

Table 9: Performance of Dilithium on Cortex-M3. **K** stands for key generation, **S** stands for signature generation, and **V** stands for signature verification.

Security Level	Work	Operation					
		K		**S**		**V**	
		Cycles	Stack	Cycles	Stack	Cycles	Stack
II	Ref	2 164k	38 452	9 193k	52 076	2 377k	36 364
	[GKS21]	1 913k	38 484	6 603k	52 116	1 805k	36 404
	[BRS22]*	2 927k	4.9 KiB	18 470k	5.0 KiB	4 036k	2.7 KiB
	Stack (C)	2 354k	7 700	17 538k	9 068	2 593k	9 756
	Stack (ASM)	2 007k	7 692	12 723k	9 052	1 953k	9 748
	Speed (ASM)	1 844k	38 444	6 113k	52 068	1 778k	36 356
III	Ref	3 712k	60 972	13 114k	79 716	3 895k	57 860
	[GKS21]	3 309k	60 876	11 681k	79 620	3 026k	57 772
	[BRS22]*	5 112k	6.4 KiB	36 303k	6.5 KiB	7 249k	2.7 KiB
	Stack (C)	3 757k	9 748	25 415k	11 108	3 957k	10 772
	Stack (ASM)	3 270k	9 740	18 554k	11 092	3 058k	10 764
	Speed (ASM)	3 233k	60 964	8 961k	79 708	3 005k	57 852
V	[BRS22]*	8 609k	7.9 KiB	44 332k	8.1 KiB	12 616k	2.7 KiB
	Stack (C)	6 199k	11 796	36 133k	13 148	6 501k	13 076
	Stack (ASM)	5 473k	11 788	27 553k	13 140	5 221k	13 068

* We cannot find a publicly available implementation for the work [BRS22] and the stack consumption is reported with unit KiB.

5.3.2 8-Bit AVR

Table 10: Performance of Dilithium on 8-bit AVR.

Security Level	Work	Operation					
		K		**S**		**V**	
		Cycles	Stack	Cycles	Stack	Cycles	Stack
II	Ref ** (C)	73 556k	9 282	166 961k	12 059	86 860k	12 751
	Stack (C)	47 732k	9 282	72 731k	12 059	49 138k	12 751
	Stack (ASM)	44 181k	9 282	62 414k	12 059	44 081k	12 751
III	Ref ** (C)	154 028k	11 330	491 601k	14 107	169 770k	13 775
	Stack (C)	84 579k	11 330	212 376k	14 107	84 213k	13 775
	Stack (ASM)	78 786k	11 330	188 290k	14 107	76 267k	13 775
V	Ref ** (C)	255 058k	13 378	1 091 977k	16 155	276 570k	16 079
	Stack (C)	144 925k	13 378	521 106k	16 155	146 478k	16 079
	Stack (ASM)	135 525k	13 378	471 359k	16 155	134 076k	16 079

** Our stack-optimized implementation based on reference code [ABD+20].

The pure reference code, as seen in Table 9, consumes a significant amount of SRAM in the 8-bit AVR environment and cannot be simulated. Thus, we implement a stack-optimized version based on the reference code [ABD+20] and designate it as comparison code (denoted as Ref **). Table 5.3.2 compares our implementation utilizing Barrett arithmetic with Ref ** of all security levels of Dilithium. Given the limited stack space in the AVR environment, unlike Cortex-M3, we only consider the stack-optimized implementation. First, we compare the Ref ** that uses Montgomery multiplication with our implementation using Barrett multiplication. Both implementations are coded in the C language. For `dilithium2`, we reduce the key generation, signature generation, and signature verification cycles by 35.11%, 56.44%, and 43.43%, respectively. For `dilithium3`, we reduce the key generation, signature generation, and signature verification cycles by 45.09%, 56.80%, and 50.40%, respectively. For `dilithium5`, we reduce the key generation, signature generation, and signature verification cycles by 43.18%, 52.28%, and 47.04%, respectively.

Subsequently, we compare our hand-written assembly implementation with C-based Ref **. Compared with the Ref ** for all security levels of Dilithium, our assembly implementation reduces the clock cycles by nearly half. For `dilithium2`, we reduce the key generation, signature generation, and signature verification cycles by 39.94%, 62.62%, and 49.25%, respectively. For `dilithium3`, we reduce the key generation, signature generation, and signature verification cycles by 48.85%, 61.70%, and 55.08%, respectively. For `dilithium5`, we reduce the key generation, signature generation, and signature verification cycles by 46.86%, 56.83%, and 51.52%, respectively.

All implementations (Ref ** (C), Stack (C), and Stack (ASM)) of each Dilithium operation consume the same stack size, and the code size is about 50 KiB in all implementations. As can be seen from the comparison between C implementations, the 32-bit multiply-long instruction is one of the most significant bottlenecks in the 8-bit AVR environment. Especially, the significant performance improvement in the Dilithium signature generation, dominated by the rejection loop, clearly highlights the benefits of Barrett multiplication.

A Detailed Implementations

Algorithm 7 Implementation of `mulhi_split` on Cortex-M3.

Inputs: alo $= a_l$, ahi $= a_h$, blo $= b_l$, bhi $= b_h$.
Outputs: acchi $= a_h b_h + \left\lfloor \frac{a_l b_h}{2^{16}} \right\rfloor + \left\lfloor \frac{a_h b_l}{2^{16}} \right\rfloor$.

```
1: mul  acchi,   ahi,    bhi                 ▷ acchi = $a_h b_h$.
2: mul accmid,   alo,    bhi                 ▷ accmid = $a_l b_h$.
3: add  acchi, acchi, accmid, asr #16        ▷ acchi = $a_h b_h + \lfloor \frac{a_l b_h}{2^{16}} \rfloor$.
4: mul accmid,   ahi,    blo                 ▷ accmid = $a_h b_l$.
5: add  acchi, acchi, accmid, asr #16        ▷ acchi = $a_h b_h + \lfloor \frac{a_l b_h}{2^{16}} \rfloor + \lfloor \frac{a_h b_l}{2^{16}} \rfloor$.
```

Algorithm 8 Implementation of `muls32xQ_lo32` on 8-bit AVR.

Inputs: (a3$\|\cdots\|$a0) $= a$
Outputs: (c3$\|\cdots\|$c0) $= aq \bmod {}^{\pm}2^{32}$

```
1: movw c0, a0    movw c2, a2    ldi q, 0xE0     mul a0, q
2: add c1, r0     adc c2, r1     adc c3, zero    mul a2, q
3: add c3, r0     mul a1, q      add c2, r0      adc c3, r1
4: ldi q, 0x7F    mul a0, q      add c2, r0      adc c3, r1
5: mul a1, q      add c3, r0
```

References

[ABD+20] 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/. 3, 4, 14, 16

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8733. 9, 10

[ACC+22] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9292. 5, 13

[AHY22] Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):349–371, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9823. 5

[ARM10] ARM. *Cortex-M3 Technical Reference Manual*, 2010. https://developer.arm.com/documentation/ddi0337/h. 2, 7, 9

[ARM21] ARM. *Armv7-M Architecture Refernce Manual*, 2021. https://developer.arm.com/documentation/ddi0403/ed. 7

[Atm16] Atmel. *AVR Instruction Set Manual*, 2016. https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf. 7

[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. SV, 1986. https://link.springer.com/chapter/10.1007/3-540-47721-7_24. 6

[BHK$^+$22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295. 4, 5, 6

[BRS22] Joppe W Bos, Joost Renes, and Amber Sprenkels. Dilithium for Memory Constrained Devices. In *International Conference on Cryptology in Africa*, pages 217–235. Springer, 2022. 14, 15

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994. https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/?active=current. 5

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/. 5

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8725. 3, 9, 10, 13, 14, 15

[GPW$^+$04] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *Cryptographic Hardware and Embedded Systems-CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings 6*, pages 119–132. Springer, 2004. 12

[GS66] W. M. Gentleman and G. Sande. Fast Fourier Transforms: For Fun and Profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578. Association for Computing Machinery, 1966. https://doi.org/10.1145/1464291.1464352. 5

[LSSR$^+$15] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. In *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17*, pages 663–682. Springer, 2015. 11

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/?active=current. 2, 6

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography. 3, 4, 17

[Ret21] RetroDan. *AVR Assembler Site*, 2021. https://avr-asm.tripod.com/avr201.html. 11

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. 2018. https://eprint.iacr.org/2018/039. 2, 6, 12

[Sho] Victor Shoup. NTL: a Library for Doing Number Theory. http://www.shoup.net/ntl/. 6

[STM20] STMicroelectronics. *STM32F207ZG*, 2020. https://www.st.com/en/microcontrollers-microprocessors/stm32f207zg.html. 13