

Optimized Implementations of Keccak, Kyber, and Dilithium on the MSP430 Microcontroller

DongHyun Shin^{1*}, YoungBeom Kim^{1,2*}, Ayesha Khalid², Maire O'Neill² and
Seog Chung Seo^{1,2†}

¹ Kookmin University, Seoul, Republic of Korea

dkldkl13@kookmin.ac.kr, darania@kookmin.ac.kr, scseo@kookmin.ac.kr

² Queen's University Belfast, Northern Ireland, United Kingdom

a.khalid@qub.ac.uk, m.oneill@ecit.qub.ac.uk

Abstract. Post-Quantum cryptography (PQC) typically requires more memory and computational power than conventional public-key cryptography. Until now, most active research in PQC optimization for embedded devices has focused on 32-bit and 64-bit ARM architectures, specifically Cortex-M0/M3/M4 and ARMv8. To enable a smooth migration of PQC algorithms in Internet of Things environments, optimization research is also required for devices with lower computational capabilities. To address this gap, we present the optimized implementation methodologies of CRYSTALS-Kyber and CRYSTALS-Dilithium, the National Institute of Standards and Technology (NIST) standardized key-encapsulation mechanism (KEM) and digital signature algorithm (DSA) on a widely used 16-bit MSP430 microcontroller. We review the current state-of-the-art implementation methodologies for Keccak, Kyber, and Dilithium, and carefully redesign them to suit the MSP430 architecture. For Number-Theoretic Transform (NTT)-based polynomial multiplication, we redesign optimal modular arithmetic, layer merging, and point-wise multiplication by taking full advantage of the characteristics of the MSP430. As a result, compared with the reference implementations in C, the optimized 16-bit NTT achieves performance improvements of 134%, 249%, and 210% for NTT, NTT^{-1} , and point-wise multiplication, respectively, while the optimized 32-bit NTT achieves performance improvements of 91%, 96%, and 56% for NTT, NTT^{-1} , and point-wise multiplication, respectively. Furthermore, for Keccak, we propose twisting and zig-zag techniques tailored to the MSP430, aimed at optimizing memory accesses. As a result, compared with the reference implementation in C, the optimized Keccak achieves a performance improvement of 57%. Moreover, compared with the reference implementations in C, our Kyber and Dilithium implementations achieve 46.1%–51.3%, 45.6%–60.0%, and 46.2%–62.3% for key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps), respectively, and 44.5%–48.3%, 57.5%–65.0%, and 46.1%–50.0% improvements for KeyGen, signing (Sign), and verifying (Verify), respectively.

Keywords: CRYSTALS-Kyber, CRYSTALS-Dilithium, 16-bit MSP430 microcontroller, NTT, SHA-3, modular arithmetic, lattice-based cryptography, Keccak)

1 Introduction

The advent of Shor's algorithm has raised serious concerns about the security of public-key cryptographic systems based on the integer factorisation problem and discrete logarithm problem (DLP), such as RSA and elliptic curve cryptography (ECC), as it reduces the

*These authors contributed equally to this work.

†Corresponding Author.

computational complexity of these primitives to polynomial time [Sho99]. In response, the National Institute of Standards and Technology (NIST) launched a standardization project for post-quantum cryptography (PQC) in 2016 [NIS].

In July 2022, NIST announced the algorithms selected for standardization in next-generation public-key cryptography: CRYSTALS-Kyber (Kyber) [ABD⁺20b] for key encapsulation mechanisms (KEMs), and CRYSTALS-Dilithium (Dilithium) [ABD⁺20a], Falcon [PFH⁺20], and SPHINCS+ [HBD⁺20] for digital signature algorithms (DSAs). Kyber and Dilithium have been standardized as ML-KEM [Nat24a] and ML-DSA [Nat24b], respectively, in the NIST FIPS 203 and FIPS 204 specifications. With the selection of these PQC standards, research on integrating PQC solutions into existing communication infrastructures has intensified, emerging as an urgent task to ensure long-term security. PQC research spans a broad spectrum of topics and open challenges, including algorithm design, cryptanalysis, optimized implementation, and integration into communication infrastructures. From an implementation perspective, because PQC operations are computationally demanding and require more memory than ECC-based counterparts, considerable efforts have been devoted to enabling efficient deployment on 32-bit and 64-bit CPUs, primarily focusing on ARM architectures [BKS19, ABCG20, AHKS22, HZZ⁺22, BHK⁺21, KSY⁺22, KSS22, HKS25]. On the protocol side, various approaches have been proposed to adapt PQC to different communication environments. For conventional protocols such as TLS/SSL [PST20], IPsec [BCP⁺22], and SSH [CPS19], efficient integration often requires appropriate modifications [MDDS24], while in some cases newly designed protocols are introduced to achieve efficiency [SSW20, GW22, KS25]. In wireless Sensor Networks (WSNs), in particular, practical protocol migration becomes feasible only when the execution of PQC can be realized on resource-constrained devices.

WSNs consist of numerous low-cost, low-power sensor nodes that communicate over short distances via wireless links. Each sensor node typically includes an integrated sensor, microcontroller, memory, transmitter, and energy storage (battery). Depending on the application requirements, these nodes can be realized through various hardware combinations, such as MicaZ [Cro00], Telos [Uni04], SkyMote [Sen06], AVR [Atm96], STM [STM87], and Texas Instruments [Tex30]. For example, in large-scale civilian deployments such as wildfire detection systems requiring thousands of sensor nodes, cost-efficiency is critical, leading to the widespread use of 8-bit AVR microcontrollers [KKA⁺20, KAK⁺20]. By contrast, for applications demanding extreme low-power operation, 16-bit MSP430 microcontrollers from Texas Instruments are commonly adopted [LSH⁺15, ARG16, CC17]. The MSP430 is particularly notable for its specialized features, such as the digital signal processing (DSP) module, and has been widely adopted for a broad range of embedded applications requiring ultra-low-power operation [CGMM15, ARG16].

Most prior studies on the feasibility of PQC implementation have primarily focused on mid- to high-end microcontrollers and high-performance processors such as 32-bit ARM Cortex-M3/M4 [BKS19, ABCG20, AHKS22, HZZ⁺22, HKS25, CYS24], 32/64-bit RISC-V [HZZ⁺24, ZYHK25], and 64-bit ARM Cortex-A [BHK⁺21, KSY⁺22, KSS22]. While these devices are crucial in IoTs, they are not appropriate for WSNs requiring ultra low-cost and ultra low-power sensor nodes, such as the 16-bit MSP430 and 8-bit AVR MCUs, which remain essential in specific industrial domains owing to their low-power consumption and cost-efficiency. This creates a critical security concern, particularly for low-end devices operating at the edge of WSNs. If only the rest of the system transitions to PQC algorithms such as Kyber and Dilithium, while these edge devices remain unprotected, they could become significant security vulnerabilities. An adversary could potentially recover sensitive data or commands transmitted among devices at the network edge, effectively nullifying all PQC-based protections applied to the upper layers of the WSN [KU24].

Therefore, we aim to demonstrate that Kyber and Dilithium can be efficiently implemented on the MSP430, which is widely used as an IoT device, through performance

optimizations. Software optimization of Keccak and NIST standardized PQC algorithms have not been studied on this platform compared to ARM-based architectures¹.

To the best of our knowledge, no prior research has reported an implementation of Keccak on MSP430. Since most PQC schemes rely heavily on Keccak as a core component, it is necessary to reexamine the existing implementation strategies for PQC in the context of MSP430. To this end, we analyze the state-of-the-art Keccak and PQC implementation techniques proposed for RISC-V and ARM Cortex-M/A series platforms [HZZ⁺24, ZYHK25, Sei18, BKS19, ABCG20, AHKS22, HZZ⁺22, BHK⁺21, KSY⁺22, KSS22, HKS25]. We identify the need for dedicated optimization strategies for the MSP430 architecture. This paper presents novel Keccak, Kyber, and Dilithium implementation methodologies tailored to the characteristics of the MSP430, including its 16-bit register structure, DSP module, and instruction set.

1.1 Contributions

- **Optimized Keccak implementation methodology.** Despite Keccak's widespread use in cryptographic primitives, no publicly available implementation exists for the 16-bit MSP430 architecture, not even in the official eXtended Keccak Code Package (XKCP) [BDPVA22]. This gap underscores the challenge of realizing standardized algorithms on ultra-constrained platforms. After thoroughly reviewing existing optimization techniques for Keccak, we conclude that existing methods are not well-suited for MSP430. To address this, we propose two novel techniques: the twisting technique, which minimizes memory access cost, and the zig-zag technique, which improves computational efficiency. Our Keccak implementation, which is the first reported implementation on MSP430, achieves approximately 57% performance improvement over the C reference, setting a new speed record.
- **Optimized NTT implementation methodologies.** We redefine the NTT implementation strategy to align with the register structure and instruction set of the 16-bit MSP430 architecture. To maximize the use of the memory-mapped dedicated hardware multiplier, we propose a schoolbook-based lazy reduction technique for the point-wise multiplication step. Additionally, we revisit the merging strategy and modular multiplication methods to better suit the characteristics of the MSP430 platform. As a result, compared with the reference implementations in C, the optimized 16-bit NTT achieves performance improvements of 134%, 249%, and 210% for NTT, NTT^{-1} , and point-wise multiplication, respectively, while the optimized 32-bit NTT achieves performance improvements of 91%, 96%, and 56% for NTT, NTT^{-1} , and point-wise multiplication, respectively.
- **Kyber and Dilithium implementations on MSP430.** We present the MSP430 implementations of Kyber and Dilithium supporting all security levels, enabled by our integrated Keccak and NTT optimizations. As a result, compared to the C reference Kyber implementation, we achieve performance improvements of 46.1–51.3%, 45.6–60.0%, and 46.2–62.3% for the Keygen, Encaps, and Decaps of Kyber at all security levels, while for Dilithium we achieve performance improvements of 44.5–48.3%, 57.5–65.0%, and 46.1–50.0% for Keygen, Sign, and Verify, respectively. Our code is available at https://github.com/KMU-CSE/PJT_MSP430_KECCAK_KYBER_DILITHIUM.

¹Even though, in 2017, Zhe Liu et al. presented an optimized Ring-LWE algorithm on 16-bit MSP430 by optimizing the modular reduction with their "SWAMS2" method [LAKS17], there is a limitation because the target algorithm is not a standard algorithm. In 2025, YB Kim and SC Seo proposed an optimized Kyber instantiation on 8-bit AVR MCU [KS25] by optimizing its performance with Lookup Table (LUT)-based modular multiplications. However, their LUT-based methods are only applicable to 8-bit AVR MCUs.

Table 1: Profiling: Kyber768 (Ref) on MSP430 **Table 2:** Profiling: Dilithium3 (Ref) on MSP430

Kyber768	All	Keccak	NTT	Dilithium3	All	Keccak	NTT
KeyGen	5 480 k (100%)	4 057 k (74.1%)	1 217 k (22.2%)	KeyGen	24 638 k (100%)	21 274 k (86.4%)	2 986 k (12.1%)
Encaps	7 204 k (100%)	5 172 k (71.8%)	1 736 k (24.1%)	Sign	35 363 k (100%)	23 729 k (67.1%)	9 853 k (27.9%)
Decaps	7 070 k (100%)	4 250 k (60.1%)	2 400 k (34.0%)	Verify	24 480 k (100%)	19 665 k (80.3%)	4 279 k (17.5%)

2 Preliminaries

2.1 CRYSTALS-Kyber

Kyber is a Module-Learning With Error (LWE)-based KEM scheme and was standardized by NIST in 2024 as Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) [Nat24a]. It achieves IND-CPA security for PKE and constructs KEM through the Fujisaki-Okamoto transformation, ultimately achieving IND-CCA2 security. Kyber involves three main stages: KeyGen, Encaps, and Decaps. The primary operation involves polynomial multiplication in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where $n = 256$ and $q = 3329$. The key components are a square matrix $\mathbf{A}$ of size k and polynomial vectors $\mathbf{s}$ and $\mathbf{e}$ of size k (2, 3, and 4). The essential computation of $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$ is performed, representing the core of Module-LWE. The KeyGen process generates $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{e}$ and computes $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. Here, $\mathbf{A}$ and $\mathbf{t}$ are public information, while the secret vector $\mathbf{s}$ and noise $\mathbf{e}$ are kept confidential. For efficiency in polynomial multiplication, NTT is employed. Starting from the NIST competition Round 2, both the public key (pk) and secret key (sk) maintain coefficient representations in the NTT domain. Polynomial multiplication for ciphertext and decryption is the main operation in the Encryption and Decryption processes. The GenMatrixA and SampleVec procedures in all processes use the SHAKE-based pseudo-random number generator. Additionally, in the KEM process, SHA-3 is invoked for the Fujisaki-Okamoto transformation.

Table 1 presents the profiling results for each key technical element of Kyber768 in the MSP430 environment. k denotes kilo cycles. SHAKE and SHA-3 account for approximately 60.1%–74.1% clock cycles. Additionally, NTT-based polynomial multiplication accounts for about 22.2%–34.0% clock cycles. Therefore, the appropriate targets for implementation optimization are Keccak, the core function of SHA-3 and SHAKE, and NTT-based polynomial multiplication.

2.2 CRYSTALS-Dilithium

Dilithium is a Module-Learning With Error (LWE)-based DSA scheme and was standardized by NIST in 2024 as Module-Lattice-Based Digital Signature (ML-DSA) scheme [Nat24b]. Its security is based on the Module-LWE and Module Short Integer Solution (MSIS) problems. Dilithium involves three main stages: KeyGen, Sign, and Verify. The polynomial ring used by Dilithium is $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ where $n = 256$ and $q = 2^{23} - 2^{13} + 1$. Similar to Kyber, matrix-vector multiplication and SHA-3 are the core operations. Dilithium is constructed using the Fiat-Shamir with aborts paradigm. Dilithium has three security levels, but unlike Kyber (NIST security 1, 3, and 5), it supports NIST security 2, 3, and 5. The dimensions of the public matrix for these levels are (4, 4), (6, 5), and (8, 7), respectively, forming a non-square structure that differentiates Dilithium from Kyber and affects both its arithmetic and memory behavior in constrained environments.

Table 2 presents the profiling results for each key technical element of Dilithium3 in the MSP430 environment. For signature generation, we report a single-run measurement that excludes aborts. SHAKE and SHA-3 account for approximately 60.1%–74.1% clock cycles. Additionally, NTT-based polynomial multiplication accounts for about 22.2%–34.0% clock cycles. Therefore, as in Kyber, the appropriate targets for implementation optimization are Keccak, the core function of SHA-3 and SHAKE, and NTT-based polynomial multiplication.

2.3 Number Theoretic Transform

Polynomial multiplication typically has a complexity of $\mathcal{O}(n^2)$ when using the schoolbook method. However, lattice-based cryptographic schemes prefer faster algorithms such as the Number Theoretic Transform (NTT), a variation of the Discrete Fourier Transform (DFT) over finite fields, which reduces complexity to $\mathcal{O}(n \log n)$. NTT can be viewed as evaluating a polynomial at n points in the ring $\mathbb{Z}_q[X]/(X^n \pm 1)$, requiring primitive roots of unity. Cyclic NTT uses $X^n - 1$ and requires a primitive n -th root of unity ζ_n in $\mathbb{Z}_q$, while Negacyclic NTT, based on $X^n + 1$, needs a primitive $2n$ -th root ζ_{2n} . Efficient NTT implementations leverage the Chinese Remainder Theorem (CRT) and FFT techniques. Through CRT, Negacyclic NTT maps $\mathbb{Z}_q[X]/(X^n + 1)$ to $\prod_i \mathbb{Z}_q[X]/(X - \zeta_{2n}^{2i+1})$ via $\log n$ layers, each reducing the polynomial degree [LZ22].

Kyber operates over $\mathbb{Z}_q[X]/(X^{256} + 1)$ with $q = 3329$. Since 3329 lacks a 512-th root of unity, Kyber employs an incomplete Negacyclic NTT using the 256-th root $\zeta = 17$, achieving 7 NTT layers. The transformation yields [Nat24a]:

$$X^{256} + 1 = \prod_{i=0}^{127} (X^2 - \zeta_{2n}^{2i+1}) = \prod_{i=0}^{127} (X^2 - \zeta_{2n}^{2br_7(i)+1})$$

Here, $br_7(i)$ denotes 7-bit bit-reversal of i , and any $f \in R_q$ is decomposed as:

$$f \bmod (X^2 - \zeta_{2n}^{2br_7(0)+1}), \dots, f \bmod (X^2 - \zeta_{2n}^{2br_7(127)+1})$$

NTT computations use FFT-style butterfly operations: Cooley-Tukey (CT) for NTT and Gentleman-Sande (GS) for NTT^{-1} . CT naturally produces bit-reversed output, and GS restores the normal order, avoiding explicit reordering. Dilithium, in contrast, operates over $\mathbb{Z}_q[X]/(X^{256} + 1)$ with $q = 8380417$, which admits a primitive 512-th root of unity. Hence, Dilithium supports a complete 256-point Negacyclic NTT (Complete NTT) [Nat24b].

2.4 Keccak

In 1993, NIST introduced SHA-0 (Secure Hash Algorithm 0). Subsequently, two standard hash functions, SHA-1 and SHA-2, were proposed. However, security concerns arose with SHA-1 owing to collision pair attacks based on cryptanalysis. To address this, NIST organized a competition to establish a new hash function standard. Keccak emerged as the winning algorithm and was subsequently standardized by NIST as SHA-3 [Nat15]. Unlike the SHA-1 and SHA-2 families, SHA-3 employs a new type of structure called the sponge construction. SHA-3 produces a fixed-length output, while SHAKE, a variant of SHA-3, can generate outputs of arbitrary lengths. Both algorithms rely on the Keccak permutation function, which updates the internal state.

The Keccak-f1600 function is a permutation consisting of six steps: pre- θ , θ , π , ρ , χ , and ι , as specified in Algorithm 1. The state $\mathbf{A}[]$ is organized as a 5×5 array of lanes, each with a size of 64-bit. The internal state is represented using the x , y , and z axes, where the x axis corresponds to rows, the y axis to columns, and the z axis to lanes [Dwo15]. All index operations are performed in modulo 5. The permutation state array is denoted as $\mathbf{A}[]$, and a particular lane within the state is represented as $\mathbf{A}[\mathbf{x}, \mathbf{y}]$.

Algorithm 1 Keccak-f1600 [BDH⁺12]**Input:** 1600-bit $A[x,y]$, all indexes are calculated mod 5**Output:** 1600-bit $A[x,y]$

```

1: //r[x,y], RC[i] are constants fixed in the specification
2: for i in 0 to 23:
3:   //pre- $\theta$  step, for x in 0 to 4
4:    $C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4]$ 
5:    $D[x] = C[x-1] \oplus \text{rot}(C[x+1], 1)$ 
6:   // $\theta$  step, for x,y in 0 to 4
7:    $A[x,y] = A[x,y] \oplus D[x]$ 
8:   // $\rho + \pi$  steps, for x,y in 0 to 4
9:    $B[y, 2x+3y] = \text{rot}(A[x,y], r[x,y])$ 
10:  // $\chi$  step, for x,y in 0 to 4
11:   $A[x,y] = ((\sim B[x+1,y]) \& B[x+2,y]) \oplus B[x,y]$ 
12:  // $\iota$  step
13:   $A[0,0] = A[0,0] \oplus RC[i]$ 
14: end for

```

The function $\text{rot}(W, r)$, which performs bitwise rotation on lane W by r , is employed, and $B[x, y]$ serves as an intermediate variable. In the **GenMatrixA** and **SampleVec** processes of Kyber, the SHAKE algorithm is employed to generate random polynomial coefficients, and in the Fujisaki-Okamoto transformation, SHA-3 is utilized.

2.5 16-bit MSP430

MSP430, a low-power MCU family developed by Texas Instruments, is tailored for applications demanding low-power consumption[Ins]. Typically, RAM ranges from 16 KB to 128 KB, and frequencies span from 1 MHz to 25 MHz. The MCU operates on a 16-bit architecture and features 12 general registers, labeled **r4-r15**, each 16 bits in size. Notably, MSP430 lacks a multiplication instruction and instead relies on a memory-mapped hardware multiplier (referred to as the multiplier). Performing a 16-bit multiplication requires four multiplier accesses (two writes for the operand words and two reads for the resulting words). This incurs higher costs compared to devices equipped with a built-in multiplication instruction [Tex18]. The 16-bit multiplier features dedicated registers: **MPY**, **MAC**, **OP2**, **RESL0**, and **RESH1**. The **MPY**, **MAC**, and **OP2** registers hold the operands, while the multiplication results are written to **RESL0** and **RESH1**. Writing operands to the **MAC** register instead of the **MPY** enables an accumulation operation rather than a simple multiplication. Since these registers are memory-mapped, accessing them incurs the same costs as accessing memory. Consequently, using the multiplier's registers as general-purpose registers is inefficient. However, this characteristic can sometimes be exploited to improve efficiency by writing directly from memory to the multiplier. The instruction set of the MSP430 is documented in [Tex].

3 Optimized Keccak Implementation Methodologies on 16-bit MSP430

In this section, we present several optimization strategies to improve the performance of the Keccak-f1600 operation, the core function of SHA-3, on the MSP430. First, we review various optimization techniques that have been presented so far in the literature.

Table 3: Comparison of Keccak Implementation Techniques on Various Platforms. The studies on RISC-V, Cortex-M4, and Armv8-A were conducted in [ZYHK25], [HAZ⁺24], and [BK22], respectively.

Methods[BDH ⁺ 12]	Platforms	MSP430
Lane complementing	RISC-V [†]	X
Lazy rotations	Cortex-M4, Armv8-A	X
Bit-interleaving	RISC-V [‡] , Cortex-M4	$\triangle$
In-place processing	RISC-V ^{†‡} , Cortex-M4, Armv8-A	X

[†]: RV{32,64}I, [‡]: RV{32,64}IB, **X**: Not applicable to MSP430, $\triangle$: Feasible but suboptimal

3.1 Existing Optimization Implementation Methods

Table 3 summarizes the Keccak optimization techniques proposed in [BDH⁺12]. These techniques have been adapted to a variety of architectures, each leveraging its own characteristics such as register size, availability of a barrel shifter, and supported instruction set. However, these techniques are not well-suited for the MSP430 architecture.

3.1.1 Lane complementing

The χ step in Keccak consists of five XOR, five AND, and five NOT operations. The lane complementing technique introduced in [BDH⁺12] reduces the number of NOT operations by transforming AND operations into OR operations using De Morgan's laws. This technique is beneficial in environments such as RISC-V [ZYHK25], which lack a single instruction to perform AND and NOT simultaneously. We implement the χ step using the **bic** instruction, which computes AND and NOT simultaneously. This approach is similar to the ARMv8-A implementation methodology in [BK22]. Therefore, the lane complementing technique is not adopted.

3.1.2 Lazy rotations

The ρ step rotates each lane to the left. The lazy rotations technique introduced in [BDH⁺12], delays these rotations and processes them simultaneously with the θ step in the next round. However, as noted in [ZYHK25], this technique cannot be applied in environments without an inline barrel shifter. Unfortunately, since the MSP430 does not support a barrel shifter, this technique cannot be used. Keccak-f1600 performs 29 left rotations per round. Efficiently implementing these rotations in the 16-bit MSP430 environment is discussed in Section 3.1.3.

3.1.3 Bit-interleaving

On a 64-bit architecture, each lane is allocated to a single register. However, for platforms that do not support this, [BDH⁺12] recommends using the bit-interleaving technique. On a 32-bit architecture, odd-positioned bits are placed in one register, while even-positioned bits are placed in another, representing an entire lane. This technique replaces 64-bit rotations with 32-bit rotations. However, this approach requires additional encoding and decoding operations to rearrange the lanes at the beginning and end of the permutation. Referring to XKCP [XKCCP], we evaluated this approach on MSP430 and found that separating 16-bit data into odd and even bits requires the memory accesses (load and store), 3×2 AND, 7×2 SHIFT, 3×2 OR, and 3×2 MOVE operations.

Table 4: Rotation mapping for efficient ρ step implementation on MSP430. Rotations that are multiples of 16 are free and do not require actual execution. ‘+’ denotes a left rotation, and ‘-’ denotes a right rotation. A rotation of -8 costs 16 cycles, while rotations of ± 1 each requires 5 cycles.

$y \backslash x$	3	4	0	1	2
2	$32 - 8 + 1$ (25)	$48 - 8 - 1$ (39)	$+3$ (3)	$16 - 8 + 2$ (10)	$48 - 5$ (43)
1	$64 - 8 - 1$ (55)	$16 + 4$ (20)	$32 + 4$ (36)	$48 - 4$ (44)	$16 - 8 - 2$ (6)
0	$32 - 4$ (28)	$32 - 8 + 3$ (27)	$+0$ (0)	$+1$ (1)	$64 - 2$ (62)
4	$64 - 8$ (56)	$16 - 2$ (14)	$16 + 2$ (18)	$+2$ (2)	$64 - 3$ (61)
3	$16 + 5$ (21)	$16 - 8$ (8)	$48 - 8 + 1$ (41)	$48 - 3$ (45)	$16 - 1$ (15)

As a result, encoding and decoding the 1600-bit internal state of Keccak-f1600 incurs approximately 7600 cycles. Furthermore, to suit a 16-bit environment, one can use four registers and extend the bit-interleaving approach by mapping bit positions modulo 4. With this approach, only a single register rotation is required, and the same computation can be realized by permuting the registers. However, as before, encoding and decoding incur substantial overhead. Without using bit-interleaving, computing the rotation of a 64-bit operand requires chaining four registers so that bits propagate through the register chain. Rotations by multiples of 16 bits can be performed at no additional cost by permuting registers, as introduced in [FG21]. The bit-interleaving approach is less efficient than the optimization method proposed in [FG21], for the following reasons.

First, the MSP430 lacks a barrel shifter. Environments in which conventional bit-interleaving is efficient typically provide a barrel shifter, and the bit-interleaving layout is intended to exploit it. On the MSP430, only the following single-bit shift or rotate instructions are available: `rla`, `rlc`, `rra`, and `rrc`.

Second, both the 16-bit and 32-bit bit-interleaving approaches require two instructions per single rotation. Because `rlc` and `rrc` use the carry flag, additional instructions are needed to stage and preserve it. This can be advantageous for small rotation amounts, but the benefit diminishes for larger rotations and can even reverse. For example, for a rotation by 24, the 32-bit bit-interleaving approach requires two 12-bit rotations on 32-bit operands, whereas the 16-bit bit-interleaving approach requires four 6-bit rotations on 16-bit operands. Since both approaches require 48 rotation instructions, the total cost is 48 cycles. In contrast, without bit interleaving, only a single right 8-bit rotation on 64-bit operands is required, and a left 32-bit rotation can be implemented at no additional cost via a permutation of registers. The right 8-bit rotation on 64-bit operands was introduced in [FG21] and requires 16 cycles.

Third, the encoding and decoding overheads are substantial. On the 16-bit MSP430, which provides only a limited number of general-purpose registers and lacks efficient bitwise manipulation instructions, the bit-interleaving operations impose excessive register pressure. This architectural constraint makes the approach impractical for MSP430.

[FG21] proposed an optimized rotation of 64-bit operand method for MSP430, where 1-bit right, 1-bit left, and 8-bit right rotations are performed in 5, 5, and 16 cycles, respectively. We appropriately utilize these three implementation methodologies to compute rotations of sizes that are multiples of 16 at no additional cost. Therefore, the required rotation sizes in the ρ step are shown in Table 4.

3.1.4 In-place processing

In general, when implementing Keccak-f1600, two internal states, A and B, are used, as specified in Algorithm 1. The in-place technique introduced in [BDH⁺12] reduces memory usage by using only a single state. This technique integrates four rounds and stores all data at the same memory location during each round, enabling an efficient implementation of Keccak-f1600. By applying this technique, the memory requirement for a internal state is reduced to 200 bytes of memory, saving a total of 200 bytes.

Additionally, [ZYHK25] extended this technique by applying the in-place method to a single round in the RISC-V environment. However, this approach is only applicable to architectures where the entire internal state can be allocated to registers. Since the MSP430 provides a maximum of 12 16-bit registers, allowing only 192 bits of data to be placed, this technique cannot be applied. Therefore, instead of using the in-place technique introduced in [BDH⁺12], we propose a twisting technique that optimizes memory access in the MSP430 environment while maintaining two internal states, as in the standard implementation. The details of this technique are provided in Section 3.2.1.

3.2 Efficient Memory Access Technique for the 16-bit MSP430

In addition to the aforementioned techniques, other optimizations have been proposed that leverage specific architectural characteristics. For example, [BK22] proposed a SIMD-based implementation methodology utilizing vector registers on the ARMv8-A architecture, while [HAZ⁺24] introduced a memory access optimization technique tailored to the pipeline structure on the ARMv7-M architecture. [ZYHK25] proposed minimizing memory accesses by utilizing sufficient register resources. However, these methods are not suitable for MSP430, a resource-constrained embedded device. MSP430 does not support vector registers, has a simple pipeline structure, and suffers from high register pressure, allowing at most three lanes to be placed in registers. Therefore, we explore alternative approaches to optimize Keccak on MSP430.

Keccak operations mainly consist of bitwise operations, which are computationally inexpensive. However, since the MSP430 can allocate at most three lanes to registers, each step of a round requires frequent memory accesses to the internal state. A single Keccak round consists of six steps: $\text{pre-}\theta$, θ , ρ , π , χ , and ι , with 24 rounds performed in total. Memory access instructions are more costly than bitwise operations, and the MSP430's 16-bit architecture requires twice as many instructions as a 32-bit architecture to perform the same operations. This indicates that the primary performance bottleneck in implementing Keccak on MSP430 arises from the memory access overhead. To reduce the number of memory accesses throughout the entire round, we restructured the original six steps— $\text{pre-}\theta$, θ , ρ , π , χ , and ι —into three merged stages: $\text{pre-}\theta$, $\theta + \rho + \pi$, and $\chi + \iota$, based on the data size and linearity required in each step. Furthermore, we propose a twisting technique to reduce the memory access cost of each step on MSP430 and a zig-zag technique to minimize the number of memory accesses.

3.2.1 Twisting technique without additional overhead

MSP430 employs the `mov` instruction for both load and store operations. It primarily supports two addressing modes for memory access: the *indexed mode* and the *indirect autoincrement mode* [Tex18]. In the indexed mode, expressed as $\text{X}(\text{Rn})$, the constant offset X is added to the base register Rn , enabling access to the memory address $\text{Rn} + \text{X}$. This mode applies to both source (load) and destination (store) operands and typically requires three clock cycles. The indirect autoincrement mode, denoted as @Rn+ , accesses the address contained in Rn and then automatically increments Rn after the operation, allowing efficient sequential memory traversal.

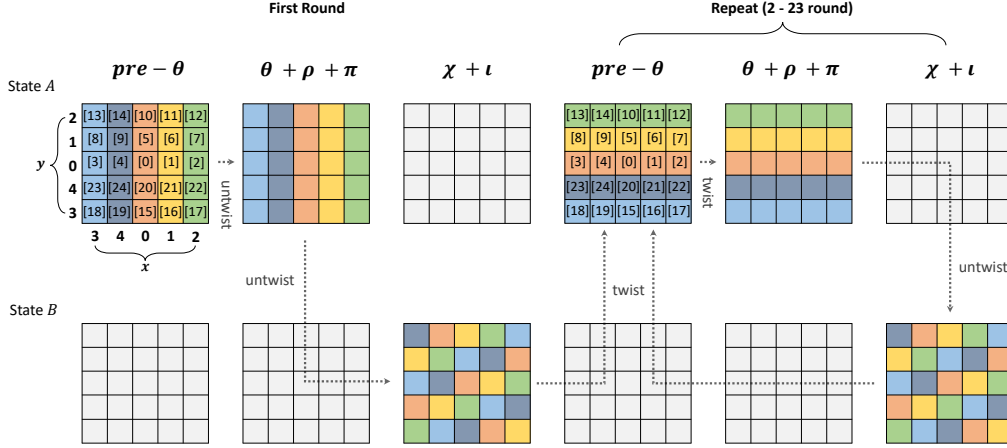


Figure 1: The process of computing Keccak over 23 rounds by twisting the internal state. The 24th round is computed without applying twisting.

The increment is by one for `.b` instructions and by two for `.w` instructions (defaulted to `.w` if omitted). Indirect autoincrement mode executes in two cycles but can only be used for `src` (load) operations, not for `dst` (store) operations. Therefore, only the indexed mode can be used for store operations. When loading the aligned data from memory, the indirect autoincrement mode is more efficient.

The internal state `A` of Keccak-f1600 is defined as a one-dimensional array, `uint64_t* A`, with the lanes of Keccak-f1600 structured as shown in Figure 1 [BDH⁺12]. For example, in the pre- θ step, for each `x`, `C[x] = A[x,0] xor ... xor A[x,4]` must be computed, requiring data from lanes in the same column. At this point, the distance between operand lanes is $8 \cdot 5$ bytes. Using the indexed mode, computing `C[x]` requires memory access to five lanes, resulting in a total of 60 cycles. Although the distance between lanes is $8 \cdot 5$ bytes, the data within each lane is aligned, allowing indirect autoincrement mode to be applied to a single lane. By using the indirect autoincrement mode for each lane and computing the address only when moving to the next lane, the computation can be completed in approximately $40 + 5$ cycles. However, this approach still incurs address calculation overhead. To eliminate this overhead and fully utilize the indirect autoincrement mode, we propose the twisting technique. Specifically, before computing the pre- θ step, the lanes in the same column are sequentially arranged in memory without additional computation. This allows the pre- θ step to utilize the indirect autoincrement mode without requiring additional address calculations. We define the rearranged internal state as the twisted state, while the original state is referred to as the untwisted state. Since the internal state is rearranged during the storing process at the end of each step, the twisting and untwisting operations can be performed without additional computational cost.

Figure 1 shows the process of calculating the 23 rounds of Keccak using the proposed twisting technique. We use two states, `A[]` and `B[]`. State `A[]` represents the input state, state `B[]` is used as a substate. In the first round, the pre- θ , ..., π steps are performed, and the results are stored in `B[]` using the untwisted state offset. After computing the $\chi + \iota$ step, the lane offsets are adjusted according to the twisted state and stored in `A[]`. From the second round onward, as shown in Figure 1, twisting and untwisting operations are performed after each step. Starting with the second round, the pre- θ step operates in the twisted state, where the required lanes are already aligned. This alignment allows the pre- θ step to utilize the indirect autoincrement mode for loading lanes without requiring additional address calculations (as shown in Algorithm 2).

Algorithm 2 First pre- θ step in twisted state**Input:** ptrA**Output:** r4, ..., r7 = C[0]₀, ..., C[0]₃, with C[0] = A[0,0] ^ ... ^ A[0,4]

1: mov @ptrA+, r4; mov @ptrA+, r5;	
2: mov @ptrA+, r6; mov @ptrA+, r7;	▷ r4...r7 = A[0,0] _{0:3}
3: xor @ptrA+, r4; xor @ptrA+, r5;	
4: xor @ptrA+, r6; xor @ptrA+, r7;	▷ r4...r7 ^ = A[0,1] _{0:3}
5: xor @ptrA+, r4; xor @ptrA+, r5;	
6: xor @ptrA+, r6; xor @ptrA+, r7;	▷ r4...r7 ^ = A[0,2] _{0:3}
7: xor @ptrA+, r4; xor @ptrA+, r5;	
8: xor @ptrA+, r6; xor @ptrA+, r7;	▷ r4...r7 ^ = A[0,3] _{0:3}
9: xor @ptrA+, r4; xor @ptrA+, r5;	
10: xor @ptrA+, r6; xor @ptrA+, r7;	▷ r4...r7 ^ = A[0,4] _{0:3}

As previously described, the pre- θ step benefits from the twisted state by eliminating the need to compute the addresses for the next lane. However, owing to high register pressure, the computed D[] values cannot be allocated in registers and must instead be stored in memory. In the θ step, for each x, D[x] is XORed with the lanes in the same column. Without the twisted state, this would require either reloading D[x] for each lane, using the indexed mode to access A[], or computing the address of A[] to use the indirect autoincrement mode, all of which introduce significant overhead. By using the twisted state, these overheads are completely avoided. The $\rho + \pi$ step, which operates independently on each lane, is unaffected by the twisting layout. Therefore, the $\theta + \rho + \pi$ steps are computed using the twisted state. Unlike the pre- θ step, the $\chi + \iota$ step requires lanes from the same row. Therefore, using the untwisted state enables the use of the indirect autoincrement mode. Once the $\chi + \iota$ step is completed, the data is stored in A using the twisted offset. This approach enables data alignment without incurring additional computational costs as the offset adjustment is performed during the store operation. Consequently, the proposed twisting technique enables the efficient use of the indirect autoincrement mode in all steps without incurring any additional overhead.

3.2.2 Efficient zig-zag processing in pre- θ and $\chi + \iota$ steps

All MSP430 instructions consist of a source (src) and a destination (dst). For example, instruction **xor r4, r5** performs an XOR operation between r4 and r5, updating the result in r5. As a result, the original value in dst is overwritten. These instruction characteristics can sometimes lead to inefficiencies in implementation. Additionally, owing to the high register pressure of MSP430, only three lanes can be allocated to registers at a time. Therefore, when implementing Keccak on the MSP430, it is crucial to carefully manage the registers and efficiently determine the execution order of the operations. Taking into account these constraints, we propose an optimized implementation methodology to efficiently calculate the pre- θ and $\chi + \iota$ steps. This approach maximizes the utilization of the values placed in registers while minimizing unnecessary overhead.

In the pre- θ step, computing each C[x] requires five lanes as operands. Since each lane is needed only once for all C[x] computations, we avoid explicitly loading them into registers. Instead, we use the **xor @ptr+, reg** instruction, which performs a direct XOR operation between memory and a register. This instruction operates in two cycles, the same as **mov @ptr+, reg**, saving 1 cycle compared to executing separate load and XOR instructions. Furthermore, each C[x] is used twice in the computation of all D[x]. Due to the high register pressure of the MSP430, it is not possible to allocate all C[x] values to the registers. We adopt an on-the-fly computation approach, where D[] is computed immediately after each C[] value is calculated.

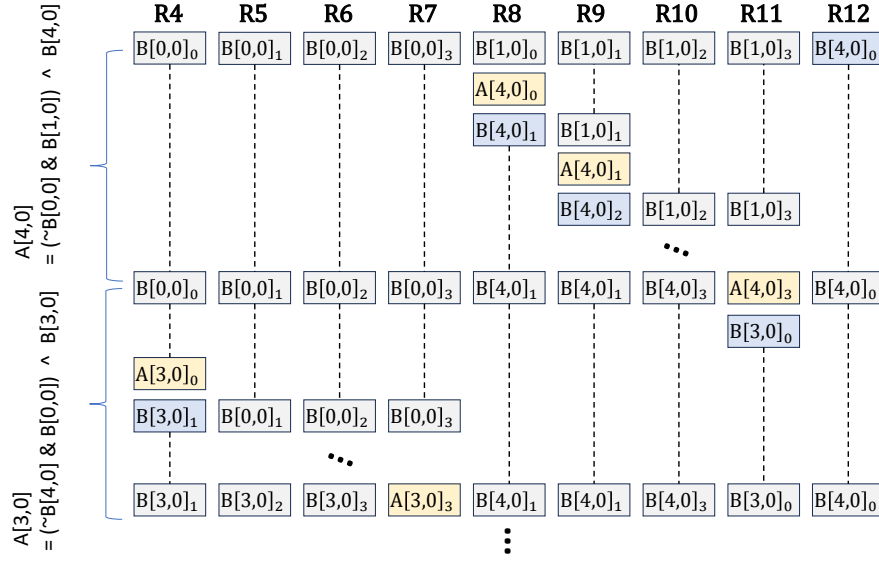


Figure 2: Efficient implementation of the χ step using a zig-zag approach. R4-R12 refer to the register numbers. The figure illustrates the computation sequence over time from top to bottom. The blue boxes indicate data being loaded into the registers, while the yellow boxes represent the computation of the χ step followed by storing the results into memory.

Therefore, instead of storing the computed $C[x-1]$ and $C[x+1]$ values in memory, we directly use them to compute $D[x]$. During this process, since the operand value for the rotate operation changes during computation, we assign it as the destination register for the XOR operation. This ensures that the $C[x-1]$ value, which serves as the source, remains unchanged and can be reused for subsequent computations, such as when calculating $D[x-2]$. We compute in the following order to reuse the left-hand operand values: $D[0] \rightarrow D[3] \rightarrow D[1] \rightarrow D[4] \rightarrow D[2]$. In the memory-storing approach, five $C[x]$ values are computed, but a total of 40 memory accesses are required to store and reload them for later use. By contrast, the on-the-fly approach computes six $C[x]$ values without storing them, ultimately reducing the total number of memory accesses by 20.

The χ step operates on three lanes in the same row as operands, with each lane being used three times during the χ step computation. Like the pre- θ step, it is more efficient to retain and reuse these values in registers. To achieve this, one lane is used as dst, while the other two lanes are preserved. Figure 2 shows the process of computing $A[4]$, ..., $A[0]$ in the χ step using this approach, focusing on the first two lanes ($A[4]$, $A[3]$). The blue boxes indicate data being loaded into the registers, while the yellow boxes represent the computation of the χ step followed by storing the results into memory. To compute $A[x,y]$, the values $B[x,y]$, $B[x+1,y]$, and $B[x+2,y]$ are required. By using $B[x+2,y]$ as the destination, $B[x+1,y]$ can be reused when computing $A[x+1,y]$, and $B[x,y]$ can be reused when computing both $A[x+1,y]$ and $A[x+2,y]$. With this approach, all lanes, except the first and last, can be reused three times. Therefore, we compute in the following order to reuse the two left-hand operands: $A[4,y] \rightarrow A[3,y] \rightarrow A[2,y] \rightarrow A[1,y] \rightarrow A[0,y]$. As a result, instead of requiring 60 `mov` instructions to perform 15 memory accesses for 5 lanes, this method uses only 28 `mov` instructions by accessing those 5 lanes just 7 times in total.

Algorithm 3 Montgomery reduction [Sei18]	Algorithm 4 Barrett reduction [Sei18]
Input: $c = a \cdot b$ such that $c = c_1\beta + c_0$ and $c \in (-\frac{\beta}{2}q, \frac{\beta}{2}q)$, where $\beta = 2^{16}$ if $q < 2^{16}$, the odd modulus $q \in (0, \frac{\beta}{2})$ Output: $r \equiv ab\beta^{-1} \bmod q$, $r \in (-q, q)$ 1: $m = c_0 \cdot q^{-1} \bmod^\pm \beta$ 2: $t_1 = \lfloor m \cdot q / \beta \rfloor$ 3: $r = c_1 - t_1$ 4: return r	Input: q with $0 < q < \frac{\beta}{2}$, and a with $a \in [-\frac{\beta}{2}, \frac{\beta}{2})$, where $\beta = 2^{16}$ Output: $r \equiv a \bmod q$, $r \in [0, q]$ 1: $v \leftarrow \lfloor \frac{2^{\log(q)-1} \cdot \beta}{q} \rfloor$ 2: $t \leftarrow \lfloor \frac{av}{2^{\log(q)-1} \cdot \beta} \rfloor$ 3: $t \leftarrow tq \bmod \beta$ 4: return $r \leftarrow a - t$

Table 5: Cycle counts for optimized 16-bit modular multiplication (for Kyber) on MSP430

16-bit Modular Multiplication	Work	Target platform	Cycles on the MSP430
LookUp-Table	[KS25]	8-bit AVR	37
Barrett (Approximate)	[HKS25]	32-bit Cortex-M3	35
Barrett	[BHK ⁺ 21]	64-bit Cortex-A	35
Plantard	[HZZ ⁺ 22]	32-bit Cortex-M4	28
	[HZZ ⁺ 24]	32-bit RISC-V	
Montgomery	[Sei18]	AVX2/AVX512	30

4 Modular Arithmetic on MSP430

Kyber and Dilithium’s 16/32-bit polynomial coefficient requires 16/32-bit multiplications, which necessitate modular reduction by q . However, using nonconstant-time division operations for this reduction introduces vulnerabilities to timing-based side-channel attacks. To address this, efficient constant-time reduction methods are essential. Montgomery reduction [Mon85] and Barrett reduction [Bar86] are widely used constant-time algorithms for modular reduction in public-key cryptosystems. For lattice-based cryptography using small prime q , signed Montgomery reduction (Algorithm 3) and signed Barrett reduction (Algorithm 4) were proposed [Sei18] and have been applied to the implementations of Kyber and Dilithium. In addition, an improved Plantard reduction has been recently proposed and applied to Kyber [HZZ⁺22]. The core principle behind these reduction methods is to replace $/q$ with shift operations that can be efficiently computed on computing devices.

4.1 Existing Modular Multiplication and Selection for MSP430

To accelerate lattice-based cryptographic schemes, recent studies have proposed optimized implementation methodologies of 16-bit and 32-bit modular multiplication. This section reviews the state-of-the-art multiplication techniques across different architectures and identifies the most efficient method for MSP430. Table 5 presents the number of clock cycles required when applying optimized 16-bit modular multiplication methods to the MSP430.

For 8-bit AVR environments, which have fewer resources than the 16-bit MSP430 core and only 8-bit registers, a signed look-up table (LUT)-based modular multiplication has been proposed [KS25]. For Kyber’s modulus $q = 3329$, this method uses two 8-bit LUTs for reduction. However, this approach is feasible only on AVR platforms. Although MSP430 devices also typically lack cache memory, they require 16-bit LUT access and include a DSP, making the AVR-style LUT-based method less suitable due to increased memory footprint and instruction overhead when emulating table lookups on a 16-bit datapath.

In addition, recent studies [BHK⁺21, HKS25] have proposed Barrett-based multiplication. The Barrett arithmetic proposed in [BHK⁺21] leverages advanced NEON instructions, such as `sqdmulh` and `srshr`, available on ARMv8 architectures to achieve efficient modular multiplication. However, MSP430 lacks rounding and advanced DSP instructions, rendering this method inefficient. Similarly, the approximate Barrett arithmetic introduced in [HKS25] enables Kyber’s modular multiplication using 8-bit multiplications. Nevertheless, on MSP430, 8-bit and 16-bit multiplications incur similar costs, thus, making this approach impractical.

Plantard multiplication, as described in [HZZ⁺22] and [HZZ⁺24], introduces another approach. The key difference between these two methods lies in the expansion of the input range. While [HZZ⁺22] uses 32×16 bit operations, [HZZ⁺24] extends the input range to 32×32 -bit to utilize unused register space in 32-bit architectures. Both methods follow the same computational procedure, but the extended input range in [HZZ⁺24] is supported by additional mathematical proofs. Montgomery multiplication [Sei18] requires three 16×16 -bit multiplications, whereas Plantard multiplication [HZZ⁺22] uses one 32×16 -bit and one 16×16 -bit multiplication. On 32-bit architectures, 32×16 -bit and 16×16 -bit multiplications require the same number of registers and execution time, giving Plantard multiplication a performance advantage [HZZ⁺22, HZZ⁺24].

Kyber For implementing Kyber on MSP430, both Plantard and Montgomery arithmetic can be considered. On the MSP430, 32×16 -bit multiplication is more costly than 16×16 -bit due to the additional `mov` instructions required. Nevertheless, Plantard multiplication is faster than Montgomery multiplication because it requires fewer multiplication operations. However, we still choose Montgomery multiplication for the following reasons. First, when considering the results in Table 5, we assume that the multiplication operands are allocated in registers. When accounting for operand loading, Plantard multiplication using 32-bit operands becomes slower. Second, since Plantard multiplication involves 32-bit data, it results in higher register pressure compared to Montgomery multiplication. This makes it more difficult to apply the NTT merging technique effectively. In addition, using Plantard multiplication increases the size of each twiddle factor from 16 bits to 32 bits, which doubles the stack usage. Therefore, we conclude that Montgomery multiplication (see Algorithm 3) is more suitable for the MSP430 architecture. Similarly, we select the Barrett reduction (see Algorithm 4) for the 16-bit modular reduction in the MSP430.

Dilithium In Dilithium, 32-bit modular multiplication is required for polynomial arithmetic. The MSP430 supports both 16-bit and 32-bit multipliers, and the latter requires twice as many cycles. Plantard multiplication requires 64-bit multiplication, which leads to inefficient implementation. Other modular multiplications can be simply extended by increasing the word size, that is, by doubling the number of instructions. Therefore, consistent with the 16-bit modular multiplication, we adopt Montgomery multiplication and Barrett reduction.

4.2 Optimized Montgomery Multiplication

Kyber Algorithm 5 presents the proposed optimized 16-bit Montgomery multiplication implemented in Assembly. In Kyber, the Montgomery multiplication uses constant β (with a value of 2^{16}) to ensure reduction within the range of $(-q, q)$. The original C implementation requires an explicit 16-bit shift operation for the 32-bit value m . Since the MSP430 has a 16-bit architecture, such shift operations incur no additional cost. As shown in line 7 of Algorithm 5, the explicit shift can be eliminated by directly using the lower 16 bits of the multiplication result. As a result, the optimized Montgomery reduction requires only the `sub` and `mov` instructions.

Algorithm 5 Optimized 16-bit Montgomery multiplication**Input:** two 16-bit integers a, b , with $a \cdot b \in (-q \cdot 2^{15}, q \cdot 2^{15})$ **Output:** 16-bit c_1 , with $c_1 \in (-q, q)$

```

1:  $//c(=c_1||c_0) = a \cdot b$ 
2: mov  $a$ , &MPYS; mov  $b$ , &OP2;
3: mov &RESHI,  $c_1$ ; ▷ 1 cycle saved by using  $c_0$  directly in multiplier
4:  $//m(=m_1||m_0) = c_0 \cdot q^{-1}$  ▷ Montgomery reduction begins here
5: mov &RESLO, &MPYS; mov  $\#q^{-1}$ , &OP2; ▷ 1 cycle saved if  $q^{-1}$  is in a register
6:  $//t = m_0 \cdot q$ 
7: mov &RESLO, &MPYS; mov  $\#q$ , &OP2; ▷ 1 cycle saved if  $q$  is in a register
8:  $//c_1 = c_1 - \lfloor \frac{t}{\beta} \rfloor$ 
9: sub &RESHI,  $c_1$ ;

```

Algorithm 6 Optimized 32-bit Montgomery multiplication**Input:** two 32-bit integers $a(=a_1||a_0), b(=b_1||b_0)$, with $a \cdot b \in (-q \cdot 2^{31}, q \cdot 2^{31})$ **Output:** 32-bit $c_3||c_2$, with $c_3||c_2 \in (-q, q)$

```

1:  $//c(=c_3||c_2||c_1||c_0) = a \cdot b$ 
2: mov  $a_0$ , &MPYS32L; mov  $a_1$ , &MPYS32H;
3: mov  $b_0$ , &OP2L; mov  $b$ , &OP2H;
4: mov &RES2,  $c_2$ ; mov &RES3,  $c_3$ ; ▷ 2 cycle saved by using  $c_2||c_3$  directly in multiplier
5:  $//m(=m_3||m_2||m_1||m_0) = c_1||c_0 \cdot q^{-1}$  ▷ Montgomery reduction begins here
6: mov &RES0, &MPYS32L; mov &RES1, &MPYS32H;
7: mov  $\#q_0^{-1}$ , &OP2L; mov  $\#q_1^{-1}$ , &OP2H; ▷ 2 cycle saved if  $q^{-1}$  is in a register
8:  $//t = m_1||m_0 \cdot q$ 
9: mov &RES0, &MPYS32L; mov &RES1, &MPYS32H;
10: mov  $\#q_0$ , &OP2L; mov  $\#q_1$ , &OP2H; ▷ 2 cycle saved if  $q$  is in a register
11:  $//c_3||c_2 = c_3||c_2 - \lfloor \frac{t}{\beta} \rfloor$ 
12: sub &RES2,  $c_2$ ; subc &RES3,  $c_3$ ;

```

Additionally, since m_0 is directly used as an operand to compute t , it is moved to the hardware operand register instead of a general-purpose register, saving one clock cycle.

Dilithium In Dilithium, the Montgomery multiplication uses constant β (with a value of 2^{32}) to ensure reduction within the range of $(-q, q)$. Algorithm 6 presents the proposed optimized Montgomery multiplication implemented in Assembly. The 32-bit Montgomery multiplication can be simply extended from the 16-bit Montgomery multiplication by increasing the word size. Specifically, the upper and lower bits of a 32-bit operand are placed in two registers. The operation can be extended by using the 32-bit multiplier and by doubling the number of instructions and registers. Similar to the 16-bit Montgomery multiplication, the explicit 32-bit shift operation can be omitted. In [SKS25], an optimized 32-bit Montgomery reduction for MSP430 is proposed. The authors move the entire multiplication result to general-purpose registers after the initial multiplication and then perform the Montgomery reduction. However, the lower word is immediately used as the input to the next multiplication. Therefore, we directly move the result words into the MPYS registers within the multiplier to reduce unnecessary **mov** instructions and the associated cycles. However, directly moving the values would corrupt the values in RES2 and RES3, so c_2 and c_3 must first be moved to general-purpose registers. Accessing RES2 immediately after the multiplication incurs a 5-cycle wait penalty (see p. 675 in [Tex18]). Therefore, this wait penalty can be overlapped with other independent operations by inserting them between the multiplication and the RES2 access. Further details are described in Section 5.2.

Algorithm 7 Optimized 16-bit Barrett reduction using $R = 2^{26}$

Input: 16-bit a , with $a \in [-2^{15}, 2^{15})$ **Output:** 16-bit a , with $a \in (-\frac{q}{2}, \frac{q}{2})$

```

1:  $//t(=t_1||t_0) = \lfloor \frac{av}{2^{26}} \rfloor \cdot q$ , precomputed
    $v = \lceil \frac{2^{26}}{q} \rceil$ 
2: mov  $a$ , &MPYS; mov #0x4ebf, &OP2;
3: mov &RESHI,  $a$ ; add #0x200,  $a$ ;
4: swpb  $a$ ; sxt  $a$ ;
5: rra  $a$ ; rra  $a$ ;
6: mov  $a$ , &MPYS; mov # $q$ , &OP2;
7: sub &RESLO,  $a$ ;

```

Algorithm 8 Optimized 16-bit Barrett reduction using $R = 2^{16}$

Input: 16-bit a , with $a \in [-2^{15}, 2^{15})$ **Output:** 16-bit a , with $a \in (-\frac{q}{6}, \frac{7q}{6})$

```

1:  $//t(=t_1||t_0) = \lfloor \frac{av}{2^{16}} \rfloor \cdot q$ , precomputed
    $v = \lceil \frac{2^{16}}{q} \rceil$ 
2: mov  $a$ , &MPYS; mov #0x14, &OP2;
3: mov &RESHI, &MPYS; mov # $q$ , &OP2;
4: sub &RESLO,  $a$ ;

```

Algorithm 9 Optimized 32-bit reduce32

Input: 32-bit $a(=a_1||a_0)$, with $a \in [-2^{31}, 2^{31})$ **Output:** 32-bit $a(=a_1||a_0)$, with $a \in (-\frac{3q}{4}, \frac{3q}{4})$

```

1: mov  $a_1$ ,  $tmp$ ; add #0x40,  $tmp$ ;
2: swpb  $tmp$ ; mov  $tmp$ ,  $tmp2$ ;
3: sxt  $tmp$ ; rla  $tmp2$ ;
4: rlc  $tmp$ ; mov  $tmp$ , &OP2L;
5: rla  $tmp$ ; subc  $tmp$ ,  $tmp$ ;
6: inv  $tmp$ ; mov  $tmp$ , &OP2H;
7: sub &RES0,  $a_0$ ; subc &RES1,  $a_1$ ;

```

4.3 Optimized Barrett Reduction

Kyber In general, the Barrett reduction in Kyber uses constant β with a value of 2^{16} to ensure reduction within the range of $(-\frac{q}{2}, \frac{q}{2})$. In Algorithm 4, since $R = 2^{\log(q)-1} \cdot \beta = 2^{26}$, the division of a 32-bit operand by 2^{26} can be replaced with a 26-bit shift. In this process, similarly to Algorithm 5, the 16-bit shift is omitted, and only a 10-bit shift is computed on the upper 16-bit operand. The 10-bit shift is computed in 4 cycles by utilizing **rra**, **swpb** and **sxt** (see Algorithm 7). We use this implementation for `poly_reduce`.

Furthermore, similar to Cortex-M4 [AHKS22], we consider modifying R to 2^{16} in the MSP430 environment to further accelerate the Barrett reduction. Algorithm 8 shows that changing R to 2^{16} removes the need for bit shift operations. Based on the experimental results, when using $R = 2^{16}$, the output of the reduction is 13 bits only when the 16-bit input is nearly full; otherwise, the input remains within 12 bits. Therefore, we apply this approach in the NTT^{-1} .

Dilithium Since Dilithium uses a variant of Barrett reduction, a 23-bit shift is required. As in Kyber, we remove the explicit 16-bit shift and compute only the remaining 7-bit shift. In [SKS25], an optimized 32-bit `reduce32` is proposed. Since the MSP430 supports only single-bit shifts, the authors perform a 7-bit shift by executing **rra** seven times over seven cycles. In contrast, we compute the same operation in five cycles by using **swpb**, **sxt**, **rla**, and **rlc**. Algorithm 9 presents the proposed optimized `reduce32` implemented in Assembly. Similar to Kyber, we considered modifying the bit shift to multiples of 8 or 16, but unlike Kyber, this significantly increases the reduction range.

Algorithm 10 Optimized modular multiplication based on a Fermat prime ($q = 257$)**Input:** two 16-bit integers a, b , with $a \cdot b \in (-q \cdot 2^{15}, q \cdot 2^{15})$ **Output:** 16-bit c_0 , with $c_1 \in (-q, q)$

```

1:  $//c(=c_1||c_0) = a \cdot b$ 
2: mov  $a$ , &MPYS; mov  $b$ , &OP2;
3: mov &RESLO,  $c_0$ ; mov &RESHI,  $c_1$ ;
4: mov.b  $c_0$ ,  $tmp$ ; swpb  $c_0$ ; ▷ Fermat prime number reduction begins here
5: and.b  $c_0$ ,  $c_0$ ; sub  $c_0$ ,  $tmp$ ;
6: mov.b  $c_1$ ,  $c_0$ ; add  $tmp$ ,  $c_0$ ;
7: swpb  $c_1$ ; sxt  $c_1$ ;
8: sub  $c_1$ ,  $c_0$ ;

```

4.4 Optimized Modular Reduction Based on a Fermat Prime

In Dilithium, during the signing procedure, since c , s_1 , and s_2 are very small, the maximum sizes of cs_1 and cs_2 are also very small (see lines 18 and 19 of Algorithm 7 in FIPS 204 [Nat24b]). An approach that employs a prime q' larger than the maximum size of polynomial multiplication but smaller than the original $q = 2^{23} - 2^{13} + 1$ was proposed in [AHKS22]. In this approach, $q' = 257$ was selected for Dilithium2 and Dilithium5, while $q' = 769$ was selected for Dilithium3. Using these primes, a packing implementation methodologies based on the 16-bit NTT was proposed for the Cortex-M4 environment. Furthermore, for $q' = 257$, a light butterfly, which replaces multiplications with shift operations, was applied to the first three layers. Since these primes enable the use of a 16-bit NTT, the advantages are maximized on 16-bit architectures such as the MSP430. Following the same approach as [AHKS22, SKS25], we select $q' = 257$ with the light butterfly applied for Dilithium2 and Dilithium5, and $q' = 769$ for Dilithium3. Moreover, by exploiting the characteristics of the prime 257, we propose a 16-bit modular multiplication method on the MSP430 that is more efficient than the 16-bit Montgomery multiplication (Algorithm 5). The proposed method is presented in Algorithm 10, which shows the optimized modular multiplication based on a Fermat prime implemented in Assembly. By using $256 \equiv -1 \pmod{257}$, the expression $c = c_{0_{lo}} + c_{0_{hi}} \cdot 2^8 + c_{1_{lo}} \cdot 2^{16} + c_{1_{hi}} \cdot 2^{24}$ can be rewritten as $c = c_{0_{lo}} - c_{0_{hi}} + c_{1_{lo}} - c_{1_{hi}}$. Therefore, the computation can be performed with 8-bit additions and subtractions, and since the MSP430 supports 8-bit instructions (**.b**), it can be implemented efficiently. In particular, as multiplication operations on the MSP430 are costly, our 16-bit modular multiplication is more efficient than Montgomery multiplication.

5 NTT-based Polynomial Multiplication on 16-bit MSP430

Kyber and Dilithium employ NTT-based polynomial multiplication, which is the second most time-consuming operation in its implementation, following Keccak. On MSP430, multiplication is handled by a hardware multiplier rather than a dedicated **mul** instruction. Instead, the hardware multiplier is accessed through the **mov** instruction. Accessing the hardware multiplier incurs the same cost as memory access. Therefore, the cost of multiplication is high compared to other architectures such as 32-bit ARM and 32-bit RISC-V. In this section, we review the latest NTT optimization techniques, select the most suitable approach for the MSP430, and present the corresponding optimized implementation methodologies.

Table 6: Latest Optimization Techniques for NTT Implementation methodologies

	Techniques	Speed	Stack	MSP430
	Layer merging [GOPS13]	↑	-	✓
	Lazy reduction for pw-mul[ABCG20]	↑	-	✓
	Streaming Public matrix A [BKS19, HKS25]	↓	↓	✓
	Asymmetric multiplication[BHK ⁺ 21]	↑	↑	✗
	Better accumulation[AHKS22]	↑	↑	✗

5.1 State-of-the-Art NTT implementation methodologies

Low-power sensor nodes in WSNs are typically composed of a single-core microcontroller and separate communication and sensor modules, rather than adopting costly system-on-chip (SoC) solutions. In such environments, the MSP430 series is widely used and typically provides 2KB to 32KB of SRAM, with communication handled through simple RF modules based on SPI or UART interfaces. Although external cryptographic modules such as the CC3100 and CC3120 can be used to enable TLS/SSL communication, achieving quantum security requires alternatives beyond ECC-based cryptography. One possible approach is to port OQS-TLS, which is based on OpenSSL, to the MSP430 platform. However, experimental results show that even with embedded optimization, OpenSSL requires at least 16KB of stack memory [Ope24, KS25]. This exceeds the SRAM capacity available in most MSP430 models, making it impractical to run concurrently with real-world communication and sensor applications. Therefore, minimizing stack usage is essential when implementing Kyber and Dilithium on the MSP430.

An alternative approach is to use external PQC cryptographic co-processors such as the ATECC608A [Ins], but developing ASIC-based hardware modules is unrealistic in terms of cost and time. Moreover, integrating expensive hardware into a low-cost MCU like the MSP430 is not practical from a cost-efficiency and usability perspective. Consequently, this work aims to identify state-of-the-art implementation strategies with minimal stack usage and redesign them for the MSP430 platform to enable practical and efficient deployment.

Table 6 summarizes the state-of-the-art implementation techniques related to Kyber and Dilithium. Layer merging [GOPS13] and lazy reduction for point-wise multiplication [ABCG20] are optimization techniques that accelerate NTT-related operations without requiring additional stack space. In addition, the method of generating the public matrix **A** on-the-fly [BKS19, HKS25] significantly reduces the overall stack usage throughout the Kyber and Dilithium stream. Accordingly, we apply these three techniques to the MSP430 environment. On the other hand, asymmetric multiplication and better accumulation techniques proposed in [BHK⁺21] and [AHKS22] have been shown to improve the performance of Kyber. However, these techniques require a substantial amount of additional stack memory. According to benchmarking results from the latest PQM4 framework [KPR⁺], applying such techniques demands at least 3KB of additional stack space. Similar concerns have also been raised in implementation studies of Kyber [KS25] and Dilithium [HKS25] on 8-bit AVR platforms. Therefore, in this work, we prioritize the stable execution of all Kyber and Dilithium security levels on the MSP430, while techniques such as asymmetric multiplication and better accumulation, which incur high stack usage, are excluded from our implementation. Instead, we consider them as optional enhancements for future speed-oriented optimizations.

5.2 Butterfly Unit of NTT

In general, NTT uses the CT algorithm, while the NTT^{-1} employs the GS algorithm. An optimization strategy proposed in [AHKS22, ACC⁺21] for the Cortex-M4 architecture

Algorithm 11 16-bit CT Algorithm on MSP430**Input:** two 16-bit integers a, b , twiddle factor ζ **Output:** $a = a + b\zeta, b = a - b\zeta$

```

1: mov  $\zeta$ , &MPYS; mov  $b$ , &OP2;
2: mov &RESHI, tmp;
3: mov &RESLO, &MPYS; mov  $\#q^{-1}$ , &OP2;
4: mov &RESLO, &MPYS; mov  $\#q$ , &OP2;
5: sub &RESHI, tmp; mov  $a$ ,  $b$ ;
6: add tmp,  $a$ ; sub tmp,  $b$ ;

```

Algorithm 12 32-bit CT Algorithm on MSP430**Input:** two 32-bit integers $a(= a_1||a_0), b(= b_1||b_0)$, twiddle factor $\zeta = (\zeta_1||\zeta_0)$ **Output:** $a = a + b\zeta, b = a - b\zeta$

```

1: mov  $\zeta_0$ , &MPYS32L; mov  $\zeta_1$ , &MPY32SH;
2: mov  $b_0$ , &OP2L; mov  $b_1$ , &OP2H;
3: mov &RES2, tmp0; mov &RES3, tmp1;
4: mov &RES0, &MPYS32L; mov &RES1, &MPY32H;
5: mov  $\#q_0^{-1}$ , &OP2L; mov  $\#q_1^{-1}$ , &OP2H;
6: mov &RES0, &MPYS32L; mov &RES1, &MPYS32H;
7: mov  $\#q_0$ , &OP2L; mov  $\#q_1$ , &OP2H;
8: sub &RES2, tmp0; subc &RES3, tmp1;
9: mov  $a_0$ ,  $b_0$ ; mov  $a_1$ ,  $b_1$ ;
10: add tmp0,  $a_0$ ; addc tmp1,  $a_1$ ;
11: sub tmp0,  $b_0$ ; subc tmp1,  $b_1$ ;

```

applies the CT algorithm to the NTT^{-1} to reduce the number of modular multiplications. However, as reported in [HZZ⁺24], applying the CT algorithm to NTT^{-1} increases memory access overhead owing to additional twiddle factors and requires approximately 0.5 KB of extra memory to store them. This approach is not suitable for MSP430, where memory resources are extremely limited. Therefore, in this work, we choose to use the CT algorithm for the NTT and the GS algorithm for the NTT^{-1} .

Kyber Algorithm 11 outlines the instruction sequence for the CT algorithm as implemented on MSP430. Since MSP430 lacks a native multiply instruction, multiple **mov** instructions are required to utilize the hardware multiplier. We apply the following strategies to maximize the efficiency of hardware multiplier usage. First, accessing the hardware multiplier incurs a cost of three cycles, equivalent to a memory access. Therefore, minimizing accesses to the multiplier is essential for reducing the overall computation costs. When a multiplication result is not reused later and is only used as input for the next multiplication, we avoid placing the value in a general-purpose register and instead move it directly into the multiplier operand registers {MPYS/MACS}, saving 1 cycle per operation. Second, at the beginning of the CT algorithm, the two polynomial coefficients a and b must be loaded from memory. Since a must be preserved for further computations, it is loaded into a general-purpose register. However, b is only used to compute $b \cdot \zeta$ and is not needed afterward, so it can be loaded directly from memory into the hardware multiplier. Both memory access and multiplier access take three cycles each, meaning that loading b into a general-purpose register and then moving it to the multiplier would take a total of six cycles. By directly loading b into the multiplier, the total cost is reduced to five cycles, saving 1 cycle. The GS algorithm has a similar structure to the CT algorithm and is therefore omitted.

Dilithium In the case of Dilithium, a 32-bit butterfly unit is required. As described in Section 4, a single coefficient is placed in two registers, and the operation can be simply extended by doubling the number of instructions. Algorithm 12 presents the proposed 32-bit CT Algorithm implemented in Assembly. The butterfly first performs the modular multiplication using Algorithm 6, and then employs two **add** and two **sub** instructions, with careful attention to carry and borrow handling. The considerations discussed for the Kyber butterfly unit are applied in the same manner.

As described in Section 4.2, the difference from [SKS25] lies in the Montgomery multiplication. However, as mentioned earlier, accessing RES2 immediately after the multiplication incurs a 5-cycle wait penalty (line 3 of Algorithm 12). Therefore, when layer merging allows multiple butterfly operations to be executed in sequence, we eliminate this penalty by overlapping it with independent operations on other coefficients (lines 9–11). For Kyber, line 2 of Algorithm 11 incurs a 3-cycle wait penalty. Similarly, this penalty can be overlapped by executing the operations corresponding to lines 5–6 of another butterfly in between.

5.3 Merging Strategy

Kyber When computing the NTT transformation, each layer requires memory access to all polynomial coefficients to perform butterfly operations, resulting in a significant number of memory accesses. To mitigate this, the layer merging strategy has been widely adopted. In n -layer merging, 2^n coefficients are loaded, and the butterfly operations for n consecutive layers are computed before storing the results. The efficiency of a layer merging strategy depends on the number and structure of the registers in the architecture. A greater number of registers allow more coefficients to be allocated in registers, enabling the merging of more layers. Since Kyber uses an incomplete NTT structure, both NTT and NTT^{-1} consist of seven layers. State-of-the-art implementations for the CortexM4 [AHKS22, HZZ⁺22] and RISC-V [HZZ⁺24, ZYHK25] architectures adopt the 4+3 merging structure. This is feasible because RISC-V provides 32 general-purpose registers and Cortex-M4F can utilize FPU registers. In contrast, Cortex-M3 lacks an FPU and has only 13 general-purpose registers, leading to the 3+3+1 structure used in [HZZ⁺24]. The MSP430 features twelve 16-bit general-purpose registers. Since Kyber’s polynomial coefficients can be placed in a single register, coefficient storage is not problematic. However, the MSP430 experiences higher register pressure compared to other architectures. Unlike the FPU registers of the Cortex-M4F, the hardware multiplier registers of the MSP430 have the same access cost as memory and cannot be used as temporary storage. Therefore, determining an optimal merging structure and carefully scheduling multiplication instructions are essential for achieving better performance. We propose a layer merging strategy suitable for the MSP430 by carefully allocating its limited registers.

Merging four layers requires 16 coefficients, but since the MSP430 has only 12 registers, at most three layers can be merged. Of the 12 registers, two are allocated for the polynomial pointer and loop counter, and one is used as a temporary register for Montgomery multiplication (as detailed in Section 4.2, only one temporary register is required owing to the direct use of the hardware multiplier). As a result, five registers remain in the case of 2-layer merging, and only one register remains in the case of 3-layer merging. The remaining registers are used to allocate constant values such as q , q^{-1} , and ζ . Constants that cannot be assigned to registers owing to limited availability are handled as immediate values. This condition benefits 2-layer merging, but 3-layer merging lacks sufficient registers to store twiddle factors, making it more advantageous to apply 3-layer merging to the lower layers. Therefore, we implement a 2+2+3 merging structure, applying 2-layer merging to the upper layers and 3-layer merging to the lower layers. Although the 1+3+3 structure results in the same number of memory accesses as 2+2+3, it incurs more frequent use of immediate values owing to two 3-layer merges, leading to increased runtime.

Table 7: Number of instructions required per point-wise multiplication in Kyber. Note that **gpr** means general-purpose register, and **hmr** means the hardware multiplier register.

Operation (cycles)	Schoolbook using the MAC	Karatsuba
Add (1)	-	3
Sub (1)	-	2
Mul: gpr to hmr (3)	10	8
Mul: hmr to gpr (3)	6	8
Montgomery reduction (19)	3	4
Total cycles	105	129

Dilithium In Dilithium, each coefficient should be placed in two registers, unlike Kyber, where a single coefficient can be placed in a single register. As a result, the register pressure in Dilithium is higher than in Kyber. Merging three layers requires eight coefficients, which implies that at least sixteen registers are needed. Therefore, we implement the NTT in the same manner as [SKS25] and merge at most two layers. Constants such as q , q^{-1} , and ζ should also be placed in two registers, so immediate values are employed. Unlike Kyber, Dilithium computes a complete NTT with eight layers, and thus we implement a 2+2+2+2 merging structure. For the small NTTs of cs_1 and cs_2 , we use the same approach as in Kyber.

5.4 Point-wise Multiplication Utilizing MAC

Kyber Polynomial multiplication between two polynomials is computed as follows: $NTT^{-1}(NTT(f) \circ NTT(g))$. After applying NTT conversion to f and g , $\hat{f}$ and $\hat{g}$ in the NTT domain consist of 128 polynomials of degree 1 such as $(\hat{f}_{2i} + \hat{f}_{2i+1}X)$ and $(\hat{g}_{2i} + \hat{g}_{2i+1}X)$ where $i = 0, \dots, 127$. Then, point-wise multiplication proceeds for $i = 0, \dots, 127$ as follows:

$$(\hat{f}_{2i} + \hat{f}_{2i+1}X) \cdot (\hat{g}_{2i} + \hat{g}_{2i+1}X) \bmod (X^2 - \zeta^{2i+1}). \quad (1)$$

Using the standard schoolbook method, this requires five Montgomery multiplications and two additions:

$$\hat{h}_{2i} = \hat{f}_{2i}\hat{g}_{2i} \bmod q + ((\hat{f}_{2i+1}\hat{g}_{2i+1} \bmod q) \cdot \zeta) \bmod q \quad (2)$$

$$\hat{h}_{2i+1}X = \hat{f}_{2i+1}\hat{g}_{2i} \bmod q + \hat{f}_{2i}\hat{g}_{2i+1} \bmod q \quad (3)$$

[ABCG20] proposed a lazy reduction technique that reduces the number of Montgomery reductions from five to three by deferring modular reduction. In [SKS25], the authors apply this technique to Dilithium's FNT by leveraging the **MAC** instruction on MSP430, which accumulates intermediate results into **RESLO** and **RESHI**. We adopt the same approach and apply it to Kyber.

As a result, the number of Montgomery reductions is reduced by two, and all **add** operations can be eliminated. Additionally, since accumulation is performed within the multiplier, fewer **mov** instructions to general-purpose registers are required. To optimize multiplication on resource-constrained devices, we also consider the well-known Karatsuba-based multiplication:

$$\hat{h}_{2i} = ((\hat{f}_{2i+1} + \hat{f}_{2i}) \cdot (\hat{g}_{2i+1} + \hat{g}_{2i})) \bmod q - \hat{f}_{2i+1}\hat{g}_{2i+1} \bmod q - \hat{f}_{2i}\hat{g}_{2i} \bmod q \quad (4)$$

$$\hat{h}_{2i+1}X = (\hat{f}_{2i+1}\hat{g}_{2i+1} \cdot \zeta) \bmod q + \hat{f}_{2i}\hat{g}_{2i} \quad (5)$$

Since the reduced $\hat{f}_{2i}\hat{g}_{2i}$ and $\hat{f}_{2i+1}\hat{g}_{2i+1}$ are reused, this method reduces one reduction and one multiplication compared to the original schoolbook method.

Table 8: Cycle counts for NTT, NTT^{-1} , point-wise multiplication, and Keccak-f1600. For a fair comparison, the implementation from [HZZ⁺24] uses the performance results reported in [ZYHK25]. [†] : simulator.

Work	Platform (Board)	16-bit (NTT/NTT ⁻¹ /PWM)	16-bit($q = 257$) NTT/NTT ⁻¹ /PWM	32-bit NTT/NTT ⁻¹ /PWM
Asm[AHKS22]	Cortex-M4 (STM32F4)	5 992/6 282/1 613	5 524/5 563/1 225	8 093/8 415/1 955
Asm[HZZ ⁺ 24]	Cortex-M3 (ATSAM3X8E)	8 026/8 779/4 311	–	–
Asm[HKS25]	Cortex-M3 (NUCLEO-F207ZG)	–	–	21 876/26 524/–
Asm[HZZ ⁺ 24]	RISC-V	13 218/11 427/2 891	–	–
Asm[ZYHK25]	(K230)	5 714/6 005/1 673	–	7 054/7 561/2 026
Asm[LWW23]	Cortex-M0 (–)	20 972/20 303/12 101	–	–
Asm[KS25]	AVR (ATmega1280p)	58 364/67 792/–	–	–
Asm[SKS25]	MSP430 (MSP430F6779 [†])	–	30 399/57 804/18 869	90 736/114 026/21 076
Asm[LAKS17]	MSP430	178 100/–/–	–	–
C (Ref)	(MSP430F67791)	102 505/175 840/51 939	–	158 038/206 692/32 546
Asm (This work)		43 732/50 378/16 765	22 545/45 982/13 429	82 635/105 409/20 826

Work	Platform (Board)	Keccak-f1600
Asm[HAZ ⁺ 24]	Cortex-M4 (STM32F4)	9 218
Asm[HAZ ⁺ 24]	Cortex-M3 (ATSAM3X8E)	9 789
C[ZYHK25]	RISC-V	15 579
Asm[ZYHK25]	(K230)	7 808
C (Ref)	MSP430	86 599
Asm (This work)	(MSP430F67791)	55 316

This requires four multiplications, three additions, two subtractions, and four Montgomery reductions. However, MAC cannot be applied due to the need to preserve intermediate values for reuse. As summarized in Table 7, the schoolbook approach using MAC achieves the lowest cycle count, outperforming the Karatsuba implementation by 24 cycles due to one fewer Montgomery reduction and reduced register access.

Dilithium Since Dilithium uses a complete NTT, the point-wise multiplication of the two polynomials after the NTT becomes a multiplication between monomials. This can be simply implemented using 32-bit Montgomery multiplication in Algorithm 6. For the small NTTs of cs_1 and cs_2 , we use the same approach as in Kyber.

6 Performance Analysis

6.1 Benchmarking Setup

We benchmark our MSP430 implementations on MSP430F67791 board with 32 KB of RAM and 512 KB of flash memory. We use IAR Embedded Workbench for MSP430 (version 8.10.3) as the IDE and the IAR C/C++ Compiler for MSP430 (version 8.10.3) with the optimization level set to `high(speed)`, equivalent to `-O3`. We measure cycle counts using the CYCLECOUNTER register, stack usage using the linker option (Enable stack usage tracking), and code size from the .map file. All proposed implementations were written in assembly, while the high-level functions were implemented in C using the reference C code, specifically the Kyber implementation from [KS25] and the Dilithium implementation from [HKS25].

6.2 Performance Analysis of Keccak-f1600 and NTT-based Operations

Table 8 presents a performance comparison of the NTT, NTT^{-1} , point-wise multiplication, and Keccak-1600 implementations across various platforms. Since prior research on the MSP430 is limited, we include comparisons with other architectures for reference. This comparison is not intended to be fair or competitive, but is provided for reference purposes only. For 16-bit NTT, NTT^{-1} , and point-wise multiplication, we redesigned and optimized state-of-the-art implementation methods to align with the constraints of the MSP430 architecture and consequently achieved performance improvements of 134%, 249%, and 210%, respectively, compared to the reference C implementation. The NTT transform in [LAKS17] is slower than the C implementation in [KSSW22] because it generates twiddle factors on-the-fly during each iteration. The MSP430 requires up to 12 cycles to perform a single 16-bit multiplication, whereas the Cortex-M4 supports a powerful multiplication instruction that completes in a single cycle, resulting in a significant performance gap compared to our implementation. RISC-V, with its 3-cycle multiplication instruction, achieves better performance, making our implementation approximately 3-4 times slower than that of [HAZ⁺24]. Furthermore, [ZYHK25] achieved even faster performance by leveraging dual-issue optimization.

For 32-bit NTT, NTT^{-1} , and point-wise multiplication, we achieved performance improvements of 91%, 96%, and 56%, respectively, compared to the reference C implementation, and further improvements of 10%, 8%, and 1%, respectively, over [SKS25]. Although the MSP430 provides a 32-bit multiplier, unlike a 32-bit architecture, implementing a 32-bit NTT requires twice as many registers and instructions. Consequently, it is more than twice as slow as the 16-bit NTT. In Dilithium with $q = 257$, the 16-bit NTT is 94% faster than our Montgomery multiplication, demonstrating the advantages of the Fermat prime number-based multiplication and the 3-layer light butterfly. For NTT^{-1} , we observe a 10% speedup, since the final reduction and the extension of the sign bit to the upper 16 bits are performed at the end.

For Keccak, the architectural limitations of the MSP430 prevent the use of well known techniques such as lazy rotations and bit-interleaving. The RISC-V RV32IM architecture [ZYHK25], with its ample register resources, supports one-round in-place processing of the entire state and applies dual-issue optimization, achieving approximately $2\times$ speedup over the reference C implementation. By contrast, the MSP430 suffers from high register pressure and structural constraints, which limits its ability to adopt these techniques. To address this, we optimized our implementation by applying data rearrangement and a zig-zag processing strategy tailored for MSP430. As a result, by improving the memory access efficiency, we achieve a performance improvement of 57% over the reference C implementation.

6.3 Performance of the CRYSTALS-Kyber

Table 9 presents comparisons of Kyber’s performance across various platforms. As mentioned earlier, MSP430 has significantly less available RAM compared to other platforms, making efficient memory usage a critical focus of our optimizations. Our goal was to implement Kyber’s core operations efficiently without increasing stack usage, and we successfully achieved performance improvements across all security levels without requiring additional stack space. Compared to the reference implementation in C, our implementation achieves performance improvements of 46.1% (and 51.3%, 51.0% respectively), 45.6% (and 60.0%, 57.8% respectively), and 46.2% (and 62.3%, 59.0% respectively) for the Keygen, Encaps, and Decaps operations of Kyber at security levels 1 (and 3, and 5, respectively). Finally, as shown in Table 9, cycle counts differ by at least a factor of two across 8-, 16-, and 32-bit architectures. Given the enhanced architectural features, including doubled register width, a larger number of registers, support for efficient multiplication instructions, and

Table 9: Cycle counts (cc), and stack usage (B) for KeyGen, Encaps, and Decaps of all security levels of Kyber. 1,000 cc is denoted by k. *: **A** generated on-the-fly. § : Asymmetric multiplication and Better accumulation applied.

Work	Platform Board	variant		Kyber512		Kyber768		Kyber1024	
				cc	stack	cc	stack	cc	stack
Asm [AHKS22]	32-bit Cortex-M4 (STM32F4)	speed*§	K	443 k	4 272	718 k	5 312	1 138 k	6 336
			E	536 k	5 376	870 k	6 416	1 324 k	7 432
		stack*	D	487 k	5 384	796 k	6 432	1 227 k	7 448
			K	444 k	2 220	724 k	2 736	1 149 k	3 256
			E	540 k	2 308	879 k	2 808	1 341 k	3 328
			D	492 k	2 324	807 k	2 824	1 246 k	3 352
Asm [HZZ ⁺ 24]	32-bit Cortex-M3 (ATSAM3X8E)	speed*§	K	518 k	3 268	842 k	4 044	1 333 k	4 812
			E	626 k	3 860	1 017 k	4 636	1 548 k	5 404
		stack*	D	587 k	3 860	958 k	4 636	1 471 k	5 404
			K	519 k	2 212	844 k	3 084	1 342 k	3 596
			E	628 k	2 300	1 025 k	2 772	1 563 k	3 284
			D	590 k	2 308	967 k	2 788	1 486 k	3 300
Asm [HZZ ⁺ 24]		speed*§	K	—	—	1 052 k	—	—	—
			E	—	—	1 261 k	—	—	—
			D	—	—	1 179 k	—	—	—
C(Ref) [ZYHK25]	32-bit RISC-V (K230)	—	K	—	—	1 222 k	—	—	—
			E	—	—	1 602 k	—	—	—
			D	—	—	1 691 k	—	—	—
Asm [ZYHK25]		speed§	K	—	—	496 k	—	—	—
			E	—	—	606 k	—	—	—
			D	—	—	578 k	—	—	—
Asm [LWW23]	32-bit Cortex-M0 (—)	stack*	K	942 k	3 136	1 492 k	3 648	2 292 k	4 160
			E	1 489 k	2 720	2 259 k	3 232	3 246 k	3 744
			D	1 594 k	2 720	2 359 k	3 232	3 347 k	3 744
Asm [KS25]	8-bit AVR (ATmega1280p)	stack*	K	5 290 k	2 220	8 608 k	2 736	13 640 k	3 256
			E	7 385 k	2 308	11 169 k	2 808	16 946 k	3 328
			D	6 763 k	2 324	10 250 k	2 824	15 773 k	3 352
C(Ref) [KSSW22]	16-bit MSP430	—	K	3 244 k	6 146	5 480 k	10 176	8 620 k	15 296
			E	4 288 k	8 804	7 204 k	13 344	10 760 k	18 976
			D	4 170 k	9 576	7 070 k	14 436	10 559 k	20 548
Asm (This work)	(MSP430F67791)	stack*	K	2 221 k	2 740	3 622 k	3 190	5 710 k	3 704
			E	2 946 k	2 816	4 503 k	3 266	6 820 k	3 780
			D	2 852 k	2 824	4 357 k	3 274	6 640 k	3 794

Table 10: code size (B) for Kyber and Dilithium

Work	Kyber512	Kyber768	Kyber1024	Dilithium2	Dilithium3	Dilithium5
C (Ref)	21 636	23 662	24 100	35 172	37 460	39 334
ASM (This work)	153 840	153 992	155 002	182 524	182 714	183 728

the inclusion of inline barrel shifters, it is reasonable to expect a performance improvement of more than twofold. As Kyber is a KEM algorithm, it's performance is compared with ECDH, which is currently used as a key exchange algorithm in network environments. [LSH⁺15] proposed an optimized ECDH P191 implementation on the MSP430. The Full ECDH was reported at 7 726 k cycles, whereas the Full Kyber512 C reference shows slower performance at 11 850 k cycles. However, our Kyber implementation substantially narrows this gap, achieving 8 019 k cycles. As a result, our optimized Kyber512 implementation not only ensures stronger security but also achieves comparable performance to ECDH P191. Therefore, migrating from ECDH to Kyber on the current MSP430 MCU would enhance security without negatively impacting application performance.

Table 10 presents the flash memory usage of Kyber. Although it increases from approximately 23 KB to 154 KB, it remains well within the 512 KB capacity of the target

Table 11: Cycle counts (cc), and stack usage (B) for KeyGen, Sign, and Verify of all security levels of Dilithium. 1,000 cc is denoted by k. For Dilithium, because the signing procedure includes aborts, benchmarks are typically reported as averages over 10,000 runs. Since execution on the MSP430 is time-consuming, our MSP430 implementation benchmarks use seeds that yield the average number of aborts per security level—4, 5, and 4, respectively. *: **A**, **y** generated on-the-fly. † : simulator.

Work	Platform Board	variant		Dilithium2		Dilithium3		Dilithium5	
				cc	stack	cc	stack	cc	stack
Asm [AHKS22]	32-bit Cortex-M4 (STM32F4)	stack*	K	1 596 k	8 508	2 827 k	9 540	4 829 k	11 696
			S	4 093 k	49 k	6 623 k	69 k	8 803 k	116 k
			V	1 572 k	36 k	2 692 k	58 k	4 707 k	93 k
C(Ref) [ZYHK25]	32-bit RISC-V (K230)	—	K	—	—	4 123 k	—	—	—
			S	—	—	13 671 k	—	—	—
			V	—	—	4 145 k	—	—	—
Asm [ZYHK25]	(K230)	speed	K	—	—	1 934 k	—	—	—
			S	—	—	5 069 k	—	—	—
			V	—	—	1 889 k	—	—	—
Asm [HKS25]	8-bit AVR (ATmega1280p)	stack*	K	44 181 k	9 282	78 786 k	11 330	135 525 k	13 378
			S	58 600 k	12 059	181 787 k	14 107	457 611 k	16 155
			V	44 081 k	12 751	76 267 k	13 775	134 076 k	16 079
Asm [SKS25]	16-bit MSP430 (MSP430F6779†)	stack*	K	12 776 k	13 236	23 593 k	17 397	38 985 k	21 005
			S	83 892 k	18 172	168 330 k	22 564	234 416 k	26 834
			V	12 749 k	16 622	22 870 k	19 158	38 580 k	23 458
C(Ref) [HKS25]	16-bit MSP430 (MSP430F6779†)	—*	K	13 505 k	13 236	24 638 k	17 397	40 631 k	21 005
			S	105 419 k	17 148	203 764 k	21 540	271 094 k	25 810
			V	13 869 k	16 622	24 480 k	19 158	41 012 k	23 458
Asm (This work)	(MSP430F6779†)	stack*	K	9 142 k	13 236	17 053 k	17 397	27 399 k	21 005
			S	63 892 k	18 172	129 404 k	22 564	167 436 k	26 834
			V	9 251 k	16 622	16 758 k	19 158	27 345 k	23 458

architecture and is therefore acceptable. We applied loop unrolling to improve performance, and the loop can be rolled back to reduce the code size.

6.4 Performance of the CRYSTALS-Dilithium

Table 11 presents comparisons of Dilithium’s performance across various platforms. Compared to the reference implementation in C, our implementation achieves performance improvements of 47.7% (and 44.5%, 48.3% respectively), 65.0% (and 57.5%, 61.9% respectively), and 49.9% (and 46.1%, 50.0% respectively) for the Keygen, Sign, and Verify operations of Dilithium at security levels 2 (and 3, and 5, respectively). Compared to [SKS25], our implementation achieves performance improvements of 39.8% (and 38.4%, 42.3% respectively), 31.3% (and 30.1%, 40.0% respectively), and 37.8% (and 36.5%, 41.1% respectively) for the Keygen, Sign, and Verify operations of Dilithium at security levels 2 (and 3, and 5, respectively). These primary performance improvements stem from our optimized Keccak implementation.

Our implementation achieves significant performance improvements, but signature generation still remains costly. When operating at up to 25 MHz, KeyGen and Verify require 0.4–1.1 seconds of execution time, whereas Sign requires approximately 2.6–6.7 seconds. Unlike Kyber, the use of 32-bit coefficients and large matrix dimensions results in very slow performance on the MSP430. In environments where execution time is critical, using a KEM algorithm such as Kyber within KEM-TLS [SSW20] can be considered as an alternative. Table 10 presents the flash memory usage of Dilithium. Although it increases from approximately 37 KB to 183 KB, it remains well within the 512 KB capacity of the target architecture and is therefore acceptable. We applied loop unrolling to improve performance, and the loop can be rolled back to reduce the code size.

7 Discussion of Side-Channel Analysis Perspective

Unlike the Cortex-M3, which includes variable-latency instructions, the MSP430 executes all instructions in constant time. Recently, a timing vulnerability in the Kyber reference code was reported in KyberSlash [BBB⁺25]. In accordance with these findings, we updated the affected components. In addition, our implementation has no secret-dependent branch or jump instructions. Therefore, our Kyber and Dilithium implementations achieve constant-time execution but do not include countermeasures against side-channel attacks beyond timing attacks. Countermeasures such as masking are left for future work. Nevertheless, our polynomial multiplication implementation can be combined with the masking countermeasure against power analysis. This can be achieved simply by repeating the NTT-based polynomial multiplication as many times as the number of shares required for masking [CGTZ23]. Coron *et al.* [CGL⁺24] proposed a masked implementation of Dilithium and designed core gadgets. While these gadgets are efficient mainly from the perspective of theoretical complexity, the internal implementation uses modular reduction. When applying this methodology to the MSP430 in the future, it can be further accelerated and realized in constant time by replacing it with our modular arithmetic.

8 Conclusion

This work presents the implementations of Keccak, Kyber and Dilithium on a 16-bit MSP430 platform, focusing on optimizing NTT-based polynomial multiplication and the Keccak-f1600 permutation, which dominates computational workload of Kyber and Dilithium. For NTT, we redesign the state-of-the-art implementation methodologies to align with the MSP430's 16-bit register structure and instruction set. The optimized 16-bit and 32-bit NTTs achieve performance improvements of 134–249% and 56–96% over the reference C implementation, respectively. For Keccak, we find that existing optimization techniques are not well suited to MSP430 due to its limited register capacity. To overcome this, we propose two new methods: twisting, which reduces memory access overhead, and zig-zaging, which improves data reuse and efficiency. Our implementation, the first reported Keccak implementation on MSP430, achieves a 57% speedup over the reference code, setting a new performance record for the platform. The twisting technique used for Keccak provides efficient access to the MSP430's internal state. However, this technique is not limited to the MSP430 exclusively. It is also effective on any architecture where the indexed mode is more costly than the indirect autoincrement mode. Furthermore, our zig-zag technique can be generalized to all architectures that use a two-operand instruction set, including 8-bit AVR and 16-bit RL78 [Ren22]. Our NTT implementations operate on 16-bit words and can therefore serve as reference implementations for extension to other 16-bit platforms. As a result, our Kyber implementation achieves a 45.6–62.3% improvement in cycle count over the C reference, while our Dilithium achieves a 44.5–65.0% improvement. In addition, our Keccak implementation is also applicable to other PQC algorithms, including SLH-DSA, and others, on the 16-bit MSP430 platform.

Acknowledgements

We are deeply grateful to the editor and the anonymous reviewers for their helpful comments during the review process. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. RS-2025-25411243, 50%), the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the MSIT (No. RS-2024-00441762, Global Advanced Cybersecurity Human Resources Development, 50%), and the Integrated Quantum Network (IQN) Research Hub (EP/Z533208/1).

References

- [ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for $\{R, M\}$ lwe schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 336–357, 2020. 410, 411, 426, 429
- [ABD⁺20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede 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/>. 410
- [ABD⁺20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede 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/>. 410
- [ACC⁺21] 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. *Cryptology ePrint Archive*, 2021. 426
- [AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-m4. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, pages 853–871. Springer, 2022. 410, 411, 424, 425, 426, 428, 430, 432, 433
- [ARG16] Ahmed I Abdul-Rahman and Corey A Graves. Internet of things application using tethered msp430 to thingspeak cloud. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 352–357. IEEE, 2016. 410
- [Atm96] Atmel Corporation (now Microchip Technology). Atmel avr microcontrollers. Company Documentation, 1996. Atmel (acquired by Microchip) developed the AVR family of microcontrollers. 410
- [Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology—CRYPTO’86: Proceedings*, pages 311–323. Springer, 1986. 421
- [BBB⁺25] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. Kyberslash: Exploiting secret-dependent division timings in kyber implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):209–234, Mar. 2025. 434
- [BCP⁺22] Seungyeon Bae, Yousung Chang, Hyeongjin Park, Minseo Kim, and Youngjoo Shin. A performance evaluation of ipsec with post-quantum cryptography. In *International Conference on Information Security and Cryptology*, pages 249–266. Springer, 2022. 410
- [BDH⁺12] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny VanKeer. Keccak implementation overview. 2012. <https://nvlabs.github.io/keccak/>

- [//keccak.team/files/Keccak-implementation-3.2.pdf](https://keccak.team/files/Keccak-implementation-3.2.pdf). 414, 415, 417, 418
- [BDPVA22] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. extended keccak code package (xkcp). <https://github.com/XKCP/XKCP>, 2022. GitHub repository, accessed 2025-10-15. 411
- [BHK⁺21] 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. *Cryptology ePrint Archive*, 2021. 410, 411, 421, 422, 426
- [BK22] Hanno Becker and Matthias J Kannwischer. Hybrid scalar/vector implementations of keccak and sphincs+ on aarch64. In *International Conference on Cryptology in India*, pages 272–293. Springer, 2022. 415, 417
- [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: 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9–11, 2019, Proceedings 11*, pages 209–228. Springer, 2019. 410, 411, 426
- [CC17] Carlos Hernandez Chulde and Jose Polo Cantero. Internet of things implementation with nodes based on low-power microcontroller msp430. In *2017 International Conference on Information Systems and Computer Science (INCISCOS)*, pages 33–40. IEEE, 2017. 410
- [CGL⁺24] Jean-Sébastien Coron, François Gérard, Tancrede Lepoint, Matthias Trannoy, and Rina Zeitoun. Improved high-order masked generation of masking vector and rejection sampling in dilithium. *Cryptology ePrint Archive*, 2024. 434
- [CGMM15] Mickaël Cazorla, Sylvain Gourgeon, Kevin Marquet, and Marine Minier. Survey and benchmark of lightweight block ciphers for msp430 16-bit microcontroller. *Security and Communication Networks*, 8(18):3564–3579, 2015. 410
- [CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4), 2023. 434
- [CPS19] Eric Crockett, Christian Paquin, and Douglas Stebila. Prototyping post-quantum and hybrid key exchange and authentication in tls and ssh. *Cryptology ePrint Archive*, 2019. 410
- [Cro00] Crossbow Technology, Inc. Crossbow technology - embedded sensor solutions. Company Website, 2000. Crossbow Technology developed MicaZ wireless sensor nodes. 410
- [CYS24] JunHyeok Choi, SeungYong Yoon, and Seog Chung Seo. Optimized falcon verify on cortex-m4 for post-quantum secure uav communications. *ICT Express*, 2024. 410
- [Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015. 413
- [FG21] Christian Franck and Johann Großschädl. Optimized implementation of sha-512 for 16-bit msp430 microcontrollers. In *International Conference on Information Technology and Communications Security*, pages 86–99. Springer, 2021. 416

- [GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *International Workshop on Post-Quantum Cryptography*, pages 67–82. Springer, 2013. 426
- [GW22] Ruben Gonzalez and Thom Wiggers. Kemtls vs. post-quantum tls: Performance on embedded systems. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 99–117, Cham, 2022. Springer Nature Switzerland. 410
- [HAZ⁺24] Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting keccak and dilithium implementations on armv7-m. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):1–24, 2024. 415, 417, 430, 431
- [HBD⁺20] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS⁺. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. <https://sphincs.org/>. 410
- [HKS25] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying polynomials without powerful multiplication instructions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):160–202, 2025. 410, 411, 421, 422, 426, 430, 433
- [HZZ⁺22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022. 410, 411, 421, 422, 428
- [HZZ⁺24] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Yet another improvement of plantard arithmetic for faster kyber on low-end 32-bit iot devices. *IEEE Transactions on Information Forensics and Security*, 2024. 410, 411, 421, 422, 427, 428, 430, 432
- [Ins] Texas Instruments. *MSP430*. Accessed: 2025-04-04, <https://www.ti.com/homepage.html>. 414, 426
- [KAK⁺20] Hyeokdong Kwon, SangWoo An, YoungBeom Kim, Hyunji Kim, Seung Ju Choi, Kyoungbae Jang, Jaehoon Park, Hyunjun Kim, Seog Chung Seo, and Hwajeong Seo. Designing a cham block cipher on low-end microcontrollers for internet of things. *Electronics*, 9(9), 2020. 410
- [KKA⁺20] YoungBeom Kim, Hyeokdong Kwon, SangWoo An, Hwajeong Seo, and Seog Chung Seo. Efficient implementation of arx-based block ciphers on 8-bit avr microcontrollers. *Mathematics*, 8(10), 2020. 410
- [KPR⁺] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. <https://github.com/mupq/pqm4>. 426
- [KS25] YoungBeom Kim and Seog Chung Seo. An optimized instantiation of post-quantum mqtt protocol on 8-bit avr sensor nodes. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 248–266, 2025. 410, 411, 421, 426, 430, 432

- [KSS22] Youngbeom Kim, Jingyo Song, and Seog Chung Seo. Accelerating falcon on armv8. *IEEE Access*, 10:44446–44460, 2022. 410, 411
- [KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 19–30, 2022. 431, 432
- [KSY⁺22] Youngbeom Kim, Jingyo Song, Taek-Young Youn, Seog Chung Seo, et al. Crystals-dilithium on armv8. *Security and Communication Networks*, 2022, 2022. 410, 411
- [KU24] Aykut Karakaya and Ahmet Ulu. A survey on post-quantum based approaches for edge computing security. *Wiley Interdisciplinary Reviews: Computational Statistics*, 16(1):e1644, 2024. 410
- [LAKS17] Zhe Liu, Reza Azarderakhsh, Howon Kim, and Hwajeong Seo. Efficient software implementation of ring-lwe encryption on iot processors. *IEEE Transactions on Computers*, 69(10):1424–1433, 2017. 411, 430, 431
- [LSH⁺15] Zhe Liu, Hwajeong Seo, Zhi Hu, Xinyi Hunag, and Johann Großschädl. Efficient implementation of ecdh key exchange for msp430-based wireless sensor networks. In *proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 145–153, 2015. 410, 432
- [LWW23] Lu Li, Mingqiang Wang, and Weijia Wang. Implementing lattice-based pqc on resource-constrained processors: A case study for kyber/saber’s polynomial multiplication on arm cortex-m0/m0+. In *International Conference on Cryptology in India*, pages 153–176. Springer, 2023. 430, 432
- [LZ22] Zhichuang Liang and Yunlei Zhao. Number theoretic transform and its applications in lattice-based cryptosystems: A survey. *arXiv preprint arXiv:2211.13546*, 2022. 413
- [MDDS24] Lukas Malina, Patrik Dobias, Petr Dzurenda, and Gautam Srivastava. Quantum-resistant and secure mqtt communication. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–8, 2024. 410
- [Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985. 421
- [Nat15] National Institute of Standards and Technology (NIST). FIPS 202: Sha-3 standard: Permutation-based hash and extendable-output functions. Technical Report 202, NIST, 2015. 413
- [Nat24a] National Institute of Standards and Technology (NIST). FIPS 203: Module-lattice-based key-encapsulation mechanism standard. Technical Report 203, NIST, 2024. 410, 412, 413
- [Nat24b] National Institute of Standards and Technology (NIST). FIPS 204: Module-lattice-based digital signature standard. Technical Report 204, NIST, 2024. 410, 412, 413, 425
- [NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. <https://csrc.nist.gov/Projects/post-quantum-cryptography>. 410, 435, 437, 439

- [Ope24] OpenSSL. *OpenSSL Documentation*, 2024. <https://docs.openssl.org/master/>. 426
- [PFH⁺20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. <https://falcon-sign.info/>. 410
- [PST20] Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in tls. In *Post-Quantum Cryptography: 11th International Conference, PQCrypto 2020, Paris, France, April 15–17, 2020, Proceedings 11*, pages 72–91. Springer, 2020. 410
- [Ren22] Renesas. *RL78 Family User’s Manual: Software Rev.2.30 Apr 2022 Single-Chip Microcontrollers*, 2022. https://www.renesas.com/en/document/mah/rl78-family-users-manual-software-rev230?srsltid=AfmB0or4QQtpdm0v3CmBoG9yZpmLA_zhrdRXD9zbhn8tqgCKSpVHQx0. 434
- [Sei18] Gregor Seiler. Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018. <https://eprint.iacr.org/2018/039>. 411, 421, 422
- [Sen06] Sentilla Corporation (formerly Crossbow Technology). Sentilla - wireless sensor networks. Company Website, 2006. Sentilla acquired Crossbow’s sensor network division, including SkyMote. 410
- [Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999. 410
- [SKS25] Donghyun Shin, Youngbeom Kim, and Seog Chung Seo. Optimizing crystals-dilithium implementation in 16-bit msp430 environment utilizing hardware multiplier. *ICT Express*, 11(1):59–65, 2025. 423, 424, 425, 428, 429, 430, 431, 433
- [SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS ’20*, page 1461–1480, New York, NY, USA, 2020. Association for Computing Machinery. 410, 433
- [STM87] STMicroelectronics. Stm32 and embedded systems. Company Website, 1987. STMicroelectronics produces STM32 microcontrollers and embedded systems. 410
- [Tex] Texas Instruments. *MSP430 Family Instruction Set Summary*. Texas Instruments. https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf. 414
- [Tex30] Texas Instruments. Texas instruments - embedded processors and microcontrollers. Company Website, 1930. Texas Instruments develops a range of microcontrollers, including the MSP430 series. 410
- [Tex18] Texas Instruments. *MSP430x5xx and MSP430x6xx Family User’s Guide*, 2018. <https://www.ti.com/lit/ug/slau208q/slau208q.pdf>. 414, 417, 423
- [Uni04] University of California, Berkeley. Uc berkeley - tinyos and telos mote. Research Lab, 2004. Telos motes were developed as part of UC Berkeley’s TinyOS project. 410

- [XKCCP] XKCP. Xkcp, 2023, <https://github.com/XKCP/XKCP>. 415
- [ZYHK25] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized software implementation of keccak, kyber, and dilithium on rv {32, 64} im {B}{V}. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):632–655, 2025. 410, 411, 415, 417, 428, 430, 431, 432, 433