
Article:
Alotaibi, A.M. and Benaissa, M. (2026) Barrett modular multiplication optimization for accelerating number theoretic transform. Sci, 8 (9). 229. ISSN: 2413-4155

https://doi.org/10.3390/sci8090229

Sci

MDPI

Article

Barrett Modular Multiplication Optimization for Accelerating Number Theoretic Transform

Ahmed M. Alotaibi and Mohammed Benaissa *

Department of Electronic and Electrical Engineering, The University of Sheffield, Sheffield S1 3JD, UK; amalotaibi1@sheffield.ac.uk
* Correspondence: m.benaissa@sheffield.ac.uk

Abstract

The practicality of post-quantum lattice-based schemes is crucial for their real-world applications. Integrating these schemes requires efficient implementations through hardware, software, and algorithmic optimisation to achieve the necessary speed and resource capability. This paper aims to improve arithmetic operations in lattice-based cryptography by accelerating the Number Theoretic Transform (NTT/INTT). It optimises the transform's main bottleneck, the twiddle-factor modular multiplication within the butterfly unit, by replacing it with constant modular multiplication derived from Barrett reduction. We introduce two constant multipliers: the Constant Barrett and a proposed Truncated-Modulus-Size Constant Barrett (TMSCB) variant, which pre-computes each twiddle constant together with its reciprocal, eliminating Barrett's data dependency and enabling area–time trade-offs. A comprehensive evaluation of the proposed constant modular multiplication is conducted against the classical Barrett multiplication, incorporating analytical complexity analysis and experimental quantitative analysis using FPGA hardware design. The proposed optimisation technique is deployed in the hardware design of the NTT/INTT for the ML-DSA and Falcon parameter sets with optimal use of the DSP cores. Performance comparisons with state-of-the-art implementations of ML-DSA NTT show 46.73% and 29.82% execution time improvements and 17% and 35.4% area resource reductions using single- and dual-butterfly units, respectively. On the Falcon, our design achieves an execution time improvement of at least 27.6%, with area savings across several butterfly configurations. These results validate the effectiveness of the Constant Barrett optimisation technique for accelerating the NTT/INTT, paving the way for more efficient implementations in other applications.

Keywords: post-quantumcryptography (PQC); lattice-based cryptography (LBC); module-lattice-based digital signature standard (ML-DSA); Fast Fourier lattice-based compact signatures over NTRU digital signature (Falcon); number theoretic transform (NTT); Barrett modular multiplication

Academic Editor: Anastasios Doulamis

Received: 6 July 2026
Revised: 18 August 2026
Accepted: 27 August 2026
Published: 1 September 2026

1. Introduction

Back in 1994, Peter Shor presented an algorithm that had the potential to solve, in polynomial time, the integer factorisation and discrete logarithm hardness problems on which current public key cryptography is based, provided that a large enough quantum computer was developed [1,2]. This emerging threat from the quantum computing era gave rise to endeavours for quantum-resilient cryptography, exemplified by the National Institute of Standards and Technology (NIST)'s standardisation competition for post-quantum cryptography.

Lattice-based cryptography has proved to be a prominent choice [3,4] among the various quantum-resilient propositions; it has security proofs based on NP-hard problems that provide resistance against today's conventional computers, as well as quantum computer attacks [5]. Additionally, it exhibits a high level of practicality derived from its simple linear algebra-based matrix/vector operations on integers [3]. In NIST's digital signature track, CRYSTALS-Dilithium is selected as the primary candidate, with Falcon also under consideration; both are lattice-based schemes [6]. The release of FIPS 204: ML-DSA, derived from the latest CRYSTALS-Dilithium specification [7,8], shows how lattice-based cryptography is rapidly becoming a core mathematical framework for a wide range of applications.

The challenge of implementing lattice-based schemes in real-world applications arises from the computational cost of the high number of polynomial multiplications required, typically performed in the frequency domain for efficiency using the Number Theoretic Transform (NTT). Therefore, exploring the hardware design space for NTT implementations in these schemes is a prerequisite to enabling the practical deployment of lattice-based cryptography. Many NTT-based polynomial multiplication hardware accelerators have been reported, including systolic array architectures [9,10] and memory-based iterative designs [11–15]. In all of these architectures, optimising the modular multiplication unit remains essential for the area–time efficiency of NTT-based polynomial multiplication.

In NTT-based polynomial multiplication, modular multiplication can be categorised into two types: multiplication by a twiddle factor, where one operand is a pre-computed constant twiddle factor, and pointwise multiplication, where both operands are variable [16]. This requires distinct optimisation strategies for each multiplication type. Since twiddle-factor operations represent the majority of multiplies, and there are relatively few pointwise multiplies, a practical approach is to investigate constant modular multiplication specifically for the twiddle-factor case.

Key modular multiplication methods such as Montgomery, Barrett, and special-form K-RED are central to NTT implementations [17–19]. Montgomery combines multiplication and reduction, while Barrett separates them. K-RED exploits moduli $q = k \cdot 2^m + 1$ for extra speed at the cost of generality. A more recent Constant Barrett approach similarly merges multiplication with reduction when one operand is fixed, making it especially attractive for NTT computations with a known ring size and modulus q [20,21].

The core research challenge is to investigate constant modular multiplication techniques for NTT acceleration that combine constant-multiplicand efficiency with the flexibility to handle arbitrary moduli, broadening fast-transform accelerators for PQC and beyond.

First, this paper presents the Truncated-Modulus-Size Constant Barrett (TMSCB) variant alongside the straightforward Constant Barrett multiplier for twiddle-factor multiplication. TMSCB bounds the intermediate arithmetic to at most $n + 1$ bits, avoiding the wider $2n$-bit datapath required by Constant Barrett, while retaining at most one final correction step. Both algorithms pair each twiddle factor with a pre-computed reciprocal to remove operation dependencies.

The proposed approach is evaluated at three levels: analytical comparison of the arithmetic requirements, a uniform standalone FPGA multiplier comparison to isolate the differences between the algorithms, and complete NTT/INTT implementations to evaluate their practical hardware impact. In the standalone comparison, q is treated as a variable to avoid modulus-specific optimisations, whereas the complete architectures use the fixed ML-DSA and Falcon parameters. The results demonstrate an improved $area \times time$ product, allowing considerable gains in speed and reduced hardware resources.

For the complete NTT/INTT evaluation, the TMSCB algorithm is integrated into an FPGA architecture for the ML-DSA parameter set, while the Constant Barrett multiplier is deployed for Falcon. ML-DSA and Falcon represent complementary implementation

conditions: ML-DSA uses wider 23-bit arithmetic, placing greater pressure on modular multiplication, whereas Falcon uses narrower 14-bit arithmetic but longer transforms of $N = 512$ or $N = 1024$, increasing the number of butterfly operations and memory accesses. The designs exploit the FPGA DSP slices efficiently by combining each slice's two's complement multiplier with its post-adder. For ML-DSA, the proposed architecture achieves speed improvements of 46.73% and 29.82% and area reductions of 17% and 35.4% in single- and dual-butterfly configurations, respectively, compared with state-of-the-art implementations. For Falcon, the architecture achieves an execution time improvement of at least 27.6% while also reducing area across all butterfly configurations. While many prior studies develop unified NTT engines that optimise both twiddle-factor and point-wise multiplications, our results demonstrate that focusing solely on the twiddle path through Barrett reduction with constant operands delivers considerable gains and can lead to more efficient transform engines across diverse parameter sets. The main contributions of this paper are:

- Introduction of two constant modular multipliers for twiddle-factor arithmetic: Constant Barrett and the proposed Truncated-Modulus-Size Constant Barrett (TMSCB).
- Comprehensive analytical and FPGA-based experimental analyses of both multipliers across varying word lengths.
- Application of the constant-operand Barrett multipliers to forward and inverse NTT/INTT, validated on the ML-DSA and Falcon parameter sets through an efficient FPGA design that makes optimal use of DSP cores.

The remainder of this paper is organised as follows: Section 2 covers the theoretical background. Section 3 presents the proposed arithmetic algorithm, followed by complexity analysis and experimental comparisons in Section 4. Section 5 presents how Constant Barrett multiplication accelerates the NTT in a practical application via a highly optimised FPGA implementation. Section 6 presents the implementation results and comparisons with prior relevant work. Finally, Section 7 summarises and concludes the paper, offering insights into future research.

2. Background

2.1. Number Theoretic Transform

The Number Theoretic Transform (NTT) is a mathematical tool that leverages the convolution theorem to enable efficient polynomial multiplication by mapping polynomials into a domain where they can be multiplied pointwise. It is a variant of the discrete Fourier transform that performs a transformation applied to an integer ring $\mathbb{Z}_q[x]/f(x)$ instead of a complex number field. Therefore, it replaces the complex arithmetic used in the discrete Fourier transform with modular arithmetic. The NTT is a critical component in modern cryptography, particularly valuable in lattice-based cryptography, and has gained widespread interest among researchers due to its performance. It allows efficient multiplication of elements in these integer rings by achieving a quasilinear complexity of $O(n \log n)$ for polynomial multiplication when implemented in a manner similar to the Fast Fourier Transform (FFT) compared to the $O(n^2)$ complexity of traditional schoolbook multiplication. However, to surpass other methods of multiplication, the NTT requires sufficiently large operands and appropriate parameter selection to overcome any conversion overhead. For example, in the ML-DSA, the ring used is $\mathbb{Z}_q[X]/(x^{256} + 1)$, involving polynomials of degree 256, which is arguably large enough to benefit from the NTT's efficiency. The coefficients of these polynomials are integers modulo $q = 8,380,417$, and the reduction polynomial is a negative-wrapped convolution $x^{256} + 1$.

Computing the negative wrapped convolution using the NTT requires computing the linear convolution first, involving the length of the transform being doubled and zero-padded. The prime modulus q must satisfy the condition $q \equiv 1 \pmod{N}$, which ensures the existence of N-th primitive roots of unity, denoted by ω. Multiplication can be achieved using Equation (1), followed by a separate polynomial reduction step. These requirements increase computational overhead.

$$\mathbf{c} = \mathrm{INTT}_{\omega^{-1}}(\mathrm{NTT}_{\omega}(\text{zero-padding}(\mathbf{a})) \circ \mathrm{NTT}_{\omega}(\text{zero-padding}(\mathbf{b}))) \tag{1}$$

However, by leveraging the properties of ψ, which represents the $2N$-th roots of unity, and ensuring that the prime modulus $q \equiv 1 \pmod{2N}$, Pöppelmann et al. introduced efficient multiplication using the form provided in Equation (2). This approach eliminates the need to zero-pad by scaling the input up with ψ the powers of the $2N$-th primitive root of unity and scaling the result down with the inverse ψ^{-1}. Consequently, it also contributes to the explicit polynomial reduction [5].

Pre-processing:

$$\bar{a} = a_i \cdot \psi_{2N}^{i} \quad \text{and} \quad \bar{b} = b_i \cdot \psi_{2N}^{i}$$

Multiplication:

$$\bar{c} = \mathrm{INTT}_{\omega^{-1}}(\mathrm{NTT}_{\omega}(\bar{a}) \circ \mathrm{NTT}_{\omega}(\bar{b})) \tag{2}$$

Post-processing:

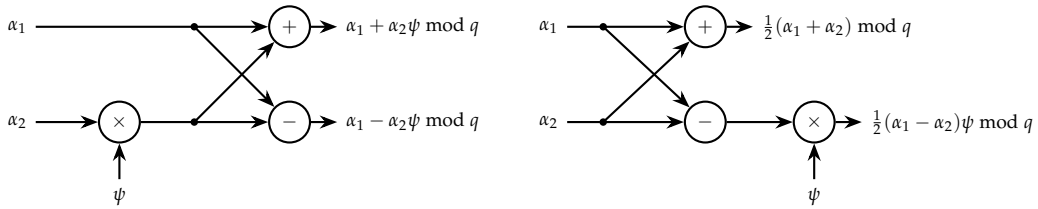
$$c = \bar{c}_i \cdot \psi_{2N}^{-i}$$

Further improvements were achieved by simplifying the computations through merging the pre-processing and post-processing steps into the NTT and INTT using the $2N$-th roots of unity ψ (as shown in Equation (3)). The merged NTT can be achieved by substituting ω^m with ψ^{2^m} as $\psi^{2^m} \equiv \omega^m \mod q$, where $m = 2^1, 2^2, \ldots, N$, thereby eliminating the need for ω [22,23].

$$\mathbf{c} = \mathrm{INTT}_{\psi^{-1}}\left(\mathrm{NTT}_{\psi}(\mathbf{a}) \circ \mathrm{NTT}_{\psi}(\mathbf{b})\right) \tag{3}$$

From a hardware perspective, the negative-wrapped formulation avoids doubling the transform length and eliminates a separate polynomial reduction stage. If the ψ-based pre-processing and post-processing factors were implemented separately, they would introduce additional coefficient reads, modular multiplications, and writes. By merging these factors into the NTT/INTT twiddle constants, as in Equation (3), no additional pass through the coefficient memory is required. The iterative CT/GS architecture therefore retains its normal stage-wise read–butterfly–write pattern, while the address-generation unit schedules the required coefficient pairs and twiddle accesses. For multiple parallel butterfly units, the main routing challenge is supplying several coefficient pairs without memory conflicts, which can be addressed through memory banking or coefficient packing, as discussed in [24].

The butterfly unit is at the heart of the NTT. It involves three arithmetic operations: modular addition, subtraction, and multiplication. Two well-known algorithms for computing the NTT are the Cooley–Tukey (CT) butterfly [25] and the Gentleman–Sande (GS) butterfly [26]. In the Cooley–Tukey butterfly structure, decimation-in-time multiplication occurs prior to the add/subtract operation, resulting in mathematical expressions such as $\alpha_1 + \alpha_2 \cdot \psi \mod q$ and $\alpha_1 - \alpha_2 \cdot \psi \mod q$, while in the decimation-in-frequency Gentleman–Sande butterfly structure, multiplication occurs post subtract operation, yielding expressions like $(\alpha_1 + \alpha_2)/2 \mod q$ and $(\alpha_1 - \alpha_2)/2 \cdot \psi \mod q$. The division by two in the Gentleman–Sande butterfly is aimed at reducing the complexity of the INTT by integrating the multiplication by N^{-1} into the butterfly operation throughout the INTT stages [27]. The CT and GS butterfly structures are illustrated in Figure 1.



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

Figure 1. Butterfly computation units in the NTT.

2.2. Modular Multiplication

Unlike modular addition and subtraction, modular multiplication is computationally intensive primarily due to the division operation. Barrett and Montgomery's algorithms are powerful tools for efficiently computing modular multiplication by avoiding division [17,18]. The Montgomery algorithm combines multiplication and modular reduction by converting numbers into the Montgomery domain, performing operations and then converting the result back into the original domain. In contrast, the Barrett algorithm first carries out multiplication and then performs modular reduction as a separate step, which is particularly efficient when using fixed moduli throughout operations. Word-level Montgomery multiplication partitions the arithmetic according to the selected word size, providing a configurable approach for different operand widths and moduli. This technique has been adopted in scalable NTT architectures such as those of Derya et al. [12] and Mert et al. [28]. Another common approach for fixed moduli often leverages the binary representation of special-form prime numbers q [29], simplifying modular reductions to bit shifts, additions, and subtractions. Notably, it becomes dramatically more efficient for specific types of moduli, such as Mersenne and Fermat moduli. Similarly, the K-RED modular reduction is a special-form modular reduction for moduli of the form $q = k \times 2^m + 1$, where k and m are integers [19]. This incorporates multiplication with a small integer k, extending the pool of special-form reductions. However, it remains restricted to that pattern, unlike Barrett and Montgomery algorithms, which impose no structural constraints on the modulus.

2.3. Related Work

Many NTT hardware accelerators have been reported, ranging from systolic array-based architectures [9,10] to the widely adopted memory-based iterative architectures [11–15]. Systolic array architectures, such as Zhao et al.'s design [9], achieve high throughput via pipelined butterfly units arranged in a two-dimensional array, completing a 256-point NTT in just 32 cycles. However, these designs incur higher area costs and limited flexibility. In contrast, iterative NTT architectures optimise resource usage by reusing single or multiple butterfly units, with polynomial coefficients passed between stages via RAM. This design manages memory access using strategies like ping-pong buffering or dedicating multiple RAMs per butterfly unit to avoid access conflicts.

Recent reported work has explored various modular reduction methods tuned for optimal butterfly efficiency. Montgomery reduction remains a popular choice, as seen in the designs in Derya et al. [12], Mert et al. [28], Di Matteo et al. [15], and Mu et al. [30]. These designs implement Montgomery reduction at the word or full-width level. Alternatively, Barrett reduction is leveraged in many designs, such as the arithmetic units of Pham et al. [11] and Dam et al. [31]. A different strategy is adopted by Li et al. [14], who utilise the K-RED reduction method. Each technique reflects a trade-off between mathematical generality and implementation-specific optimisation.

In previous constant Barrett multiplication implementations, Becker et al. [20] and Nguyen et al. [21] explored a strategy that leverages the constant multiplicand in the Barrett method and optimises it for efficient software implementation on ARMv8-A platforms using NEON instructions. Another approach, known as Plantard arithmetic, was proposed by Plantard [32] to enable efficient modular multiplication by constant unsigned integers. This technique was later implemented on the ARM Cortex-M4 by Huang et al. [33], who extended it to support signed integers. However, optimisation techniques that exploit the constant multiplicand in the Barrett method, tailored explicitly for FPGA-based NTT hardware, present promising opportunities for further adaptation and refinement.

2.4. Classical Barrett Modular Multiplication

Applying Classical Barrett Modular Multiplication to compute the modular reduction in the product of a and b modulo q, where both a and b are non-negative integers smaller than q, involves the steps outlined in Algorithm 1. The approximate quotient is calculated without expensive integer division, leveraging an offline pre-computed reciprocal R of the modulus q, as demonstrated in the pre-computation step. Division is pushed offline and stored, allowing the online stage to proceed as a streamlined sequence of simple shift operations, multiplications, and subtractions.

Algorithm 1 Classical Barrett multiplication.

Require: $a, b \in \mathbb{Z}_q, k \approx \log q$ *- where a and b are n-bit positive integers.*
Offline Pre-computation: $R = \left\lfloor \frac{2^{2k}}{q} \right\rfloor$ *- where R is a k-bit positive integer.*
Ensure: $result = a \times b \bmod q$ *- where the range of the result is* $[0, q-1]$.
1. $x = a \times b$ *- where x will be at most* $2n$ *bits long.*
2. $approx\_quotient = \left\lfloor \frac{x \times R}{2^{2k}} \right\rfloor$
3. $result = x - approx\_quotient \times q$
4. **while** $result \geq q$ **do**
   $result = result - q$ *- Number of iterations depends on the selection of k.*
5. **return** *result*

The Barrett modular multiplication algorithm exhibits variation based on the representation needed for inputs and results, whether in signed or unsigned forms. The unsigned representation covers non-negative integers, extending from 0 to $q-1$. On the other hand, the signed representation, often referred to as centred modular representation, expresses integers around zero. Each integer x in this representation is equivalent to $x \mod q$ and falls within the range $(-\frac{q-1}{2}, \frac{q-1}{2})$. Unsigned values are converted to signed by checking if they fall within the second half of the modulus range. If they do, the modulus is subtracted to obtain the signed value, while the first half remains unsigned.

The selection of the parameter k in Barrett reduction, together with integer approximation operations such as floor approximation, ceiling approximation, a power-of-two modulus, and division by a power of two, is critical for the algorithm's performance [34,35]. Substituting precise integer computations with lower-precision expressions influences both the accuracy of the quotient calculation and the cost of multiplying by R. A larger k and tight approximation ensure a closer match between the computed and actual quotients, keeping the final result within $[0, q-1]$ without extra adjustments; however, it increases multiplication expenses. Conversely, a smaller k simplifies the computation but may lead to overshooting the modulus, requiring one or more subtractions of q. A common compromise is to choose k equal to the bit-length of q, i.e., $k = n = \lceil \log_2 q \rceil$. We adopt this choice for the remainder of the paper.

As noted by Satriawan et al., Barrett modular multiplication variants hinge on four key aspects: pre-computation, integer multiplication (accurate or truncated), quotient

approximation, and corrective steps [36]. Subsequent variants often optimise one or more of these components, leading to a range of Barrett-like techniques such as Shoup multiplication [37].

3. Proposed Optimisation for Constant Barrett Modular Multiplication

3.1. Constant Barrett Multiplication

When employing classical Barrett multiplication in scenarios where the multiplicands are constant or have pre-determined values while the multiplier remains variable, a straightforward approach is to embed the constant multiplicand b in the calculation of the offline pre-computed reciprocal as $R = \lfloor \frac{b \times 2^{2n}}{q} \rfloor$. This method offers two advantages. First, Constant Barrett multiplication allows us to perform the first two steps in parallel. It removes the dependency present in the first two steps of the classical Barrett algorithm, which requires obtaining the value of x and then using it to multiply by the reciprocal. The second advantage is providing an accurate quotient calculation, given by accurate_quotient $= \lfloor \frac{a \times R}{2^{2n}} \rfloor$, eliminating the iterative step in the classical Barrett algorithm. The relationship between these steps can be expressed mathematically, as shown in Equation (4).

$$ab - \left\lfloor \frac{a \left\lfloor \frac{b \times 2^{2n}}{q} \right\rfloor}{2^{2n}} \right\rfloor q \tag{4}$$

This mathematical formulation is then translated into algorithmic steps, as presented in Algorithm 2, which illustrates the necessary modifications to classical Barrett Algorithm 1.

Algorithm 2 Constant Barrett multiplication.

Require: $a, b \in \mathbb{Z}_q$, $n = \lceil \log_2(q) \rceil$ — *- where a and b are n-bit integers.*
Offline Pre-computation: $R = \left\lfloor \frac{b \times 2^{2n}}{q} \right\rfloor$ — *- where R is a 2n-bit positive integer.*
Ensure: $result = a \times b \bmod q$ — *- where the range of the result is* $[0, q-1]$.
1. $x = a \times b$ — *- where x will be at most 2n bits long.*
2. accurate-quotient $= \left\lfloor \frac{a \times R}{2^{2n}} \right\rfloor$
3. result $= x -$ accurate-quotient $\times q$
4. **return** result

Assuming k is rounded up to the nearest integer mean $k = n$, and that the integer multiplication step is executed at full precision, it can be observed that both Algorithms 1 and 2 exhibit a recurring theme: both require asymmetric multiplication to compute the quotient, particularly in step two, where a $2n$-bit value multiplies an n-bit value. In Algorithm 1 the variable x and in Algorithm 2 the reciprocal R variable both have a size of $2n$ bits. The doubled bit size in one of the multiplication's operands highlights the computational load. This leaves room for optimisation opportunities that can enhance computational efficiency and drive a more practical solution.

3.2. Proposed Truncated-Modulus-Size Constant Barrett Multiplication

The proposed optimisation consolidates the Constant Barrett multiplication Algorithm 2, reducing storage requirements and enhancing the efficiency of modular multiplication by a constant multiplicand and its reciprocal. This is achieved by aligning the datapath with the modulus size and relaxing the accuracy of the initial approximation, as explained below. As a result, the principal arithmetic operations are restricted to modulus-size or $n + 1$-bit operations, thus preventing computational slowdowns caused by multiplication growth. Algorithm 3 illustrates the proposed Truncated-Modulus-Size Constant Barrett multiplication.

Algorithm 3 Proposed Truncated-Modulus-Size Constant Barrett multiplication.

Require: $a, b \in \mathbb{Z}_q$, $n = \lceil \log_2(q) \rceil$ - *where a and b are n-bit positive integers.*
Offline Pre-computation: $R = \left\lfloor \frac{b \times 2^n}{q} \right\rfloor$ - *where R is an n-bit positive integer.*
Ensure: $result = a \times b \bmod q$ - *where the range of the result is* $[0, q-1]$.
1. $x = a \times b \bmod 2^{n+1}$ - *where x will be at most* $n+1$ *bits long.*
2. $approx\_quotient = \left\lfloor \frac{a \times R}{2^n} \right\rfloor$
3. $result = \left(x - \left(approx\_quotient \times q \bmod 2^{n+1}\right)\right) \bmod 2^{n+1}$
4. **if** $result \geq q$ **then**
   $result = result - q$ - *The maximum number of iterations is limited to 1.*
5. **return** *result*

The proposed TMSCB multiplication algorithm pre-computes a word-size reciprocal by multiplying b with 2^n and dividing by q. The steps of the algorithm begin by calculating $x = a \times b \mod 2^{n+1}$, with x being at most $n+1$ bits long, thus restricting the datapath to no more than $n+1$ bits. The next step computes an approximate quotient by multiplying a with the pre-computed reciprocal R and dividing by 2^n. The initial result is obtained by subtracting the product of the approximate quotient and q from x, followed by a modulo 2^{n+1} operation, maintaining the datapath throughout at most $n+1$ bits in length. Then, it checks if the result is greater than or equal to q. If so, it subtracts q, ensuring that the final output is within the range $[0, q-1]$. Finally, the algorithm returns the modular multiplication result. The mathematical formulation for the entire process can be expressed by Equation (5):

$$\begin{aligned} \text{result} &= \left[ab \bmod 2^{n+1} - \left(\left\lfloor \frac{a \left\lfloor \frac{b \times 2^n}{q} \right\rfloor}{2^n} \right\rfloor q \bmod 2^{n+1} \right) \right] \bmod 2^{n+1} \\ \text{result} &= \begin{cases} \text{result} - q & \text{if result} \geq q \\ \text{result} & \text{otherwise} \end{cases} \end{aligned} \tag{5}$$

The bounded datapath in TMSCB is deterministic rather than probabilistic. Since $R = \lfloor b2^n/q \rfloor$, the approximate quotient differs from $\lfloor ab/q \rfloor$ by at most one. Consequently, before the final correction, the intermediate result lies in the range $[0, 2q)$. Since $q \leq 2^n$, the result is always representable within $n+1$ bits, and at most one subtraction of q is required to obtain the final value in $[0, q-1]$.

4. Analysis and Evaluation

4.1. Analytical Analysis

The analytical computational complexity of the proposed Truncated-Modulus-Size Constant Barrett multiplication algorithm is compared to that of the classical (Algorithm 1) and Constant (Algorithm 2) Barrett to evaluate efficiency independently of hardware, programming languages, or implementation variations for a range of input sizes in terms of n bits. Table 1 shows a detailed breakdown of the operational count, which includes offline pre-computation and five operations: two integer multiplications, a multiplication by q, a subtraction operation, and a conditional statement involving subtraction.

In the offline stage of the classical Barrett approach, the pre-computed and stored reciprocal R is determined based on the value of the modulus q. This n-bit value remains constant and can be reused in subsequent operations as long as the value of q remains unchanged. This reusability saves both computational time and storage resources during run-time. However, in the two constant Barrett algorithms, R must be recalculated whenever the multiplicand b or the modulus q changes. This requirement introduces

additional pre-computation overhead, with the computational complexity of division being expressed as $O(n^2)$ for long division. The storage requirements for R differ between the two constant algorithms: it requires $2n$ bits for Constant Barrett Algorithm 2 and n bits for TMSCB Algorithm 3. This difference minimises memory usage in the proposed optimised algorithm. For each fixed multiplicand b and modulus q, the reciprocal R is pre-computed offline and stored in ROM; therefore, the division required to obtain R is not part of the implemented datapath.

Table 1. Operations breakdown and compare and contrast details.

Operations	1. Classical Barrett Multiplication	2. Constant Barrett Multiplication	3. Modulus-Size Constant Barrett Multiplication
Offline R	n-bit value	$2n$-bit value	n-bit value
$a \times b$	Std. n-bit multiplier; the product is $2n$-bit	Std. n-bit multiplier; the product is $2n$-bit	Trunc. n-bit multiplier; the product is (n+1)-bit
a or $x \times R$	Trunc. $2n$-bit by n-bit; the product is n-bit	Trunc. $2n$-bit by n-bit; the product is n-bit	Trunc. n-bit by n-bit; the product is n-bit
Multiply by q	Std. n-bit multiplier; the product is $2n$-bit	Std. n-bit multiplier; the product is $2n$-bit	Trunc. n-bit multiplier; the product is (n+1)-bit
1st subtraction	$2n$-bit subtraction	$2n$-bit subtraction	(n+1)-bit subtraction
2nd subtraction	n-bit subtraction	Eliminated	n-bit subtraction

In comparing the $a \times b$ step, both classical and constant Barrett algorithms utilise a standard n-bit multiplier, which produces a $2n$-bit product. This results in the time and resource complexity of $O(n^2)$ due to the need to generate all partial products. However, the proposed optimised algorithm employs truncated multiplication using $\bmod 2^{n+1}$, limiting the result to $(n+1)$ bits. By computing the product modulo 2^{n+1}, the truncated multiplication method eliminates unnecessary partial products, reducing the number of partial product operations by approximately half and resulting in fewer intermediate steps. Consequently, while the time complexity remains $O(n^2)$, a constant factor reduction of 2 is introduced to reflect the decreased number of operations.

In the second step of the algorithms, Constant Barrett multiplication Algorithm 2 and TMSCB Algorithm 3 have no dependency compared to classical Barrett. The difference is that TMSCB multiplication uses n-bit by n-bit symmetric multiplication for $a \times R$ instead of the $2n$-bit by n-bit asymmetric multiplication used in constant Barrett multiplication. This brings two advantages: firstly, it maintains a consistent multiplication size throughout the algorithm by using n-bit by n-bit symmetric multipliers, which helps benefit various implementation strategies, and secondly, it maintains the computational complexity at $O(n^2)$, while in the classical and constant Barrett algorithms, there is a growth rate factor of 2 due to the increased number of partial products generated by the $2n$-bit operand.

In the final step, the multiplication by q is truncated in the TMSCB algorithm to $(n+1)$ bits, optimising the subtraction to $n+1$-bit instead of $2n$-bit. This reduces the multiplication complexity compared to Constant Barrett multiplication but introduces an additional conditional statement with a subtraction to bring the result within the interval $[0, q-1]$. Since the approximation is either within or slightly above q but less than $2q$, a single subtraction of q corrects the result to the desired range $[0, q-1]$, managed by an if-condition rather than a while-loop, limiting the process to one iteration. Although the TMSCB algorithm adds one more comparison step compared to constant Barrett multiplication, the cost of the comparator and subtractor is lower than that of multiplication.

In general, limiting multiplication and subtraction to at most $n + 1$ bits enables the proposed algorithm to handle increasing input sizes more efficiently than the other two algorithms as the computational load grows less proportional with n. This results in improved performance and lower resource usage, especially for larger n.

4.2. Experimental Comparisons

To validate the analytical results quantitatively, the three algorithms are implemented within a uniform reference framework, shown in Figure 2, for several values of n. The design is fully combinational on a Xilinx Artix-7 FPGA (Xilinx, San Jose, CA, USA). (xc7a200tffg1156-3) and serves as a baseline for performance metrics. No assumptions are made about the binary form of q by treating it as a variable to ensure that the measured results capture general-case behaviour rather than optimisations tied to a particular modulus. Registers are inserted only at the input and output boundaries to ensure timing closure and to determine the maximum clock frequency. These registers also let us measure storage requirements, as they hold the operands a, b, R, and q, and the resulting product. In this way, all three algorithms are evaluated under identical constraints and conditions to ensure that the comparison accurately reflects their inherent differences.

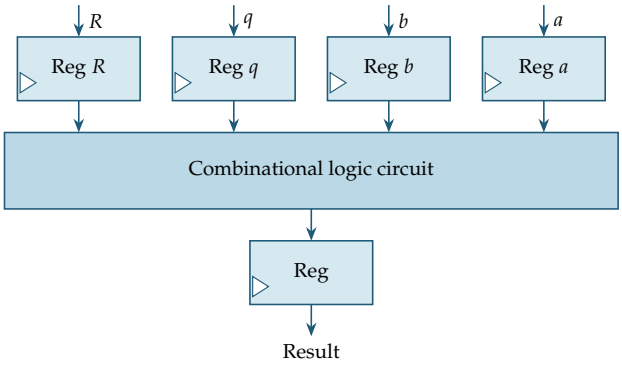


Figure 2. Design framework for algorithm efficiency evaluation. Arrows indicate the direction of signal flow, while the triangle on each register denotes its clock input.

Table 2 shows the comparative implementation results in terms of the maximum operating frequency in MHz, the area occupied in LUTs, storage requirements in terms of the number of registers used, and the area–time product (A*T), which assesses how well each algorithm balances computational speed with resource consumption. Figure 3 visualises the data from the table, illustrating speed and area trends across increasing bit sizes, which helps forecast performance for larger bit sizes.

The TMSCB multiplication consistently uses fewer LUTs than the Constant and classical Barrett approaches. This reduction is attributed to the decreased operand size for multiplication with reciprocals R and modulus q, which leads to fewer operations overall. Furthermore, there is a consistent 16.67% improvement in the number of registers used in the TMSCB multiplication algorithm compared to the Constant Barrett approach. This improvement is driven by reduced storage requirements achieved through algorithmic optimisation. In terms of speed, the Truncated-Modulus-Size Barrett and constant Barrett multiplication perform better than the classical approach due to eliminating a dependency and the ability to perform the first two steps in parallel. When the number of bits n is small, the Constant Barrett multiplication outperforms the TMSCB multiplication. However, as n grows, the situation reverses. This is because, for smaller n, the bottleneck occurs at the conditional statement in step number four due to the additional conditional statement required by TMSCB Algorithm 3, resulting in extra logic in the critical path. However, as the value of n increases, the bottleneck shifts to the multiplication by R, where

the multiplication complexity becomes more significant than the conditional statement. The lower area–time product metric (A*T) shows the advantage of the TMSCB algorithm. As the input size n grows, the efficiency of the TMSCB algorithm becomes increasingly evident, with considerable gains in computational speed and reduced hardware resources. The $n = 64$ case in Table 2 evaluates standalone modular multiplication architectures within a uniform combinational framework, rather than a complete large-modulus NTT/INTT accelerator for use in a homomorphic-encryption implementation. The results provide evidence that the advantage of TMSCB increases with operand width. However, system-level validation of large-modulus NTT architectures using FHE parameters remains an area for future investigation.

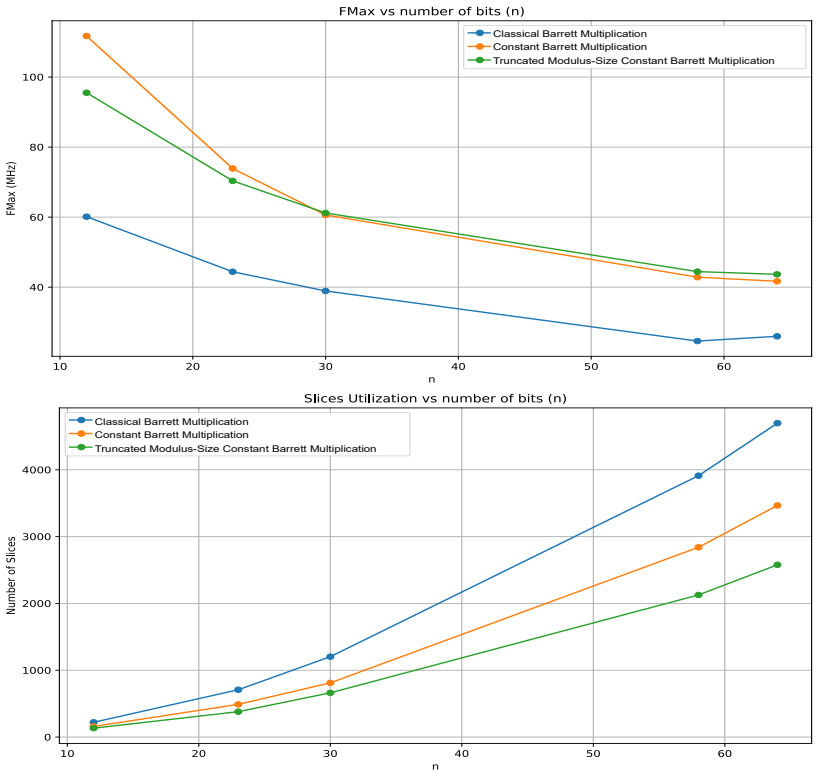


Figure 3. Visual representation of maximum clock speed (MHz) and resource utilisation across algorithms.

4.3. Framework for Constant Barrett in NTT

When adapting Barrett reduction for NTT implementations, several unique challenges and opportunities for optimisation arise in modular multiplication. The selection of the prime modulus q in the NTT is vital for two primary reasons. First, q must be suitable for the algorithmic requirement of the NTT, which is precisely the existence of the $2N$-th roots of unity. Second, q must enable efficient modular multiplication. In lattice-based PQC, the modulus q is fixed by the scheme, so designers cannot choose arbitrary primes. Therefore, adapting classical Barrett reduction to this fixed q usually involves taking advantage of bit-level constant multiplication in hardware to maximise implementation efficiency.

The efficiency of bit-level constant multiplication in hardware implementation depends on two key factors: the Hamming weight and the pattern of adjacent ones in the binary representation of the constant. A constant with a low Hamming weight requires fewer shifts and additions. Similarly, constants with adjacent ones can be optimised using two's complement arithmetic, which allows for subtraction combined with shifts, further reducing the number of operations.

Table 2. Algorithm performance and area utilisation across different bit sizes on Artix 7 Platform.

n	FMax	Slices	LUTs	Regs	A × T [1]	Algorithm
12	95.53	133	396	60	1.392	Truncated-Modulus-Size Constant Barrett
	111.72	159	505	72	1.423	Constant Barrett
	60.13	221	756	61	3.675	Classical Barrett
23	70.36	380	1275	115	5.401	Truncated-Modulus-Size Constant Barrett
	73.90	489	1639	138	6.617	Constant Barrett
	44.41	708	2559	116	15.941	Classical Barrett
30	61.19	662	2255	150	10.818	Truncated-Modulus-Size Constant Barrett
	60.62	811	2807	180	13.379	Constant Barrett
	38.92	1203	4390	151	30.912	Classical Barrett
58	44.44	2127	7817	290	47.858	Truncated-Modulus-Size Constant Barrett
	42.85	2840	10,534	348	66.271	Constant Barrett
	24.61	3912	14,344	291	158.984	Classical Barrett
64	43.67	2578	9581	320	59.031	Truncated-Modulus-Size Constant Barrett
	41.68	3466	12,867	384	83.156	Constant Barrett
	25.97	4698	17,735	321	180.929	Classical Barrett

[1] Area–time product metric, where A denotes area (measured in a number of FPGA slices) and T denotes time (the inverse of the maximum operating frequency, $1/F_{\max}$).

In the following section, we integrate our constant multiplication algorithms into an iterative, memory-based architecture where each twiddle factor and its reciprocal are pre-computed and stored at the same memory address for faster access. In this design, we exploit the efficiency of bit-level constant multiplication by q and leverage a constant multiplicand. This architecture allows us to benchmark it against other techniques like Montgomery and K-RED.

5. Constant Barrett-Based NTT Accelerator

We evaluated Constant Barrett variations in two NTT designs for PQC applications, ML-DSA (q = 8,380,417, $n = 23$) and Falcon (q = 12,289, $n = 14$), on a Xilinx UltraScale+ FPGA, leveraging its DSP48E2 slices, each equipped with a built-in signed 27 × 18-bit multiplier. The primary hardware constraint is balancing datapath width against DSP capacity: for ML-DSA, our TMSCB variant (Algorithm 3) truncates all intermediates to n bits so that every multiplication fits in a single DSP slice, whereas for Falcon, the Constant Barrett implementation (Algorithm 2) fits the $2n$-bit multiply into a single DSP48E2 slice. We select algorithms based on how well their operand widths map to DSP resources, thereby achieving the best area–performance efficiency. Moreover, the fully pipelined design executes in a fixed number of cycles, ensuring constant-time behaviour and eliminating timing variations.

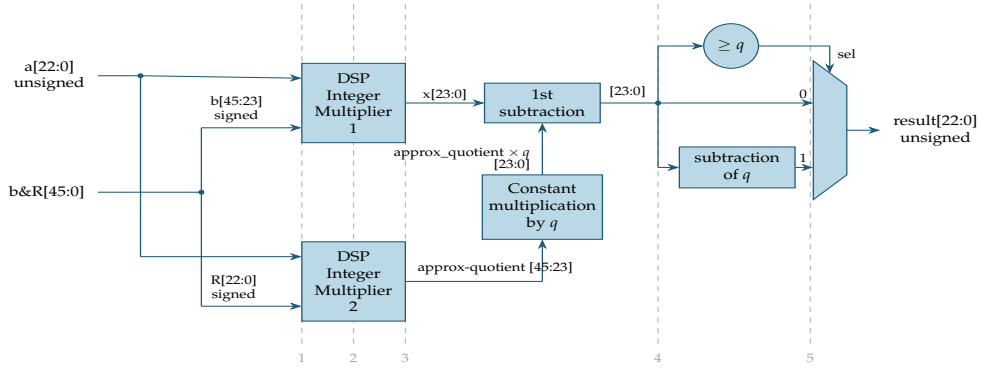
5.1. Constant Modular Multiplication Modules

5.1.1. Truncated-Modulus-Size Constant Barrett (TMSCB) Module

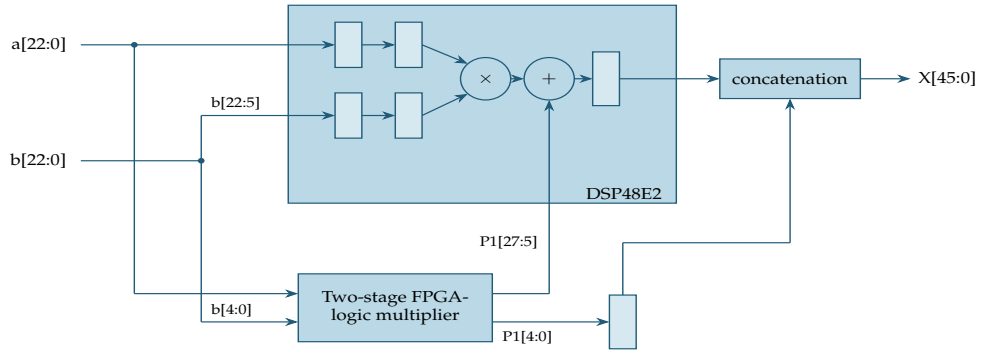
The hardware design of the TMSCB module illustrates how Algorithm 3 is mapped onto the DSP48E2's signed two's complement multiplier for the ML-DSA parameter set, combining the n-bit twiddle factor and its reciprocal into a single concatenated input bus. ML-DSA operations involve non-negative integers; thus, all operands and results lie in $[0, q-1]$. To fully exploit the DSP48E2's built-in signed multiplier, we recast each pre-computed twiddle constant as a signed value in $\left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$, ensuring no bit is wasted and maximising hardware utilisation.

Figure 4a depicts the TMSCB's top-level datapath, where the unsigned operand a and the concatenated signed constants b and R enter a five-stage pipeline. Integer multiplication

is the primary challenge of hardware resources. This challenge arises because we multiply 23-bit unsigned values by 23-bit signed values, which exceeds a single DSP48E2 block's built-in multiplier. Therefore, two options were available to overcome this limitation. The first option is to cascade additional DSP blocks, which may not be efficient as the DSP core is most efficient when fully utilised. The second option is to employ FPGA logic to handle the multiplication of the remaining five bits. FPGA logic is more efficient when the operand size is small, making it the preferred solution.



(**a**) Top-level architecture of the proposed TMSCB multiplication module.



(**b**) DSP integer multiplier implementation.

Figure 4. DSP-based architectures for twiddle-factor modular multiplication, optimised for the ML-DSA parameter set.

The detailed design of the integer multiplication, comprising three pipeline stages, is illustrated in Figure 4b. The integer multiplication module is divided into two segments. The first segment involves partial product multiplication, performed parallelly in two clock cycles using the DSP and the logic multiplier. The DSP core handles the multiplication of the larger portion of the operands, precisely a signed 18-bit number by an unsigned 23-bit number. At the same time, the FPGA logic multiplier processes the remaining 5-bit partial multiplication. The second segment handles partial product addition and concatenation. The partial product results are combined using the DSP48E2's post-adder feature, which facilitates the direct addition of partial products within the DSP slice, eliminating the need for additional logic. Finally, the most significant and least significant bits are concatenated to generate the final result.

Following integer multiplication, the multiply-by-q step exploits q's fixed value via bit-level constant multiplication approach based on its binary representation $2^{23} - 2^{13} + 1$. This requires two shifts, one addition, and one subtraction. To handle signed values, sign extension is performed prior to the shift operations to preserve the sign and mitigate the possibility of it being displaced. This process is demonstrated in a Verilog code snippet;

refer to Listing 1, line 13. To balance latency, an intermediate pipeline stage is added between the subtraction and conditional subtraction operations; see Listing 1, line 17.

Listing 1. Verilog code of the proposed Modulus-Size Constant Barrett multiplication.

```
1  input   wire        clk,
2  input   wire [22:0] A,
3  input   wire [45:0] B,
4  output  reg  [22:0] result;
5
6  wire [45:0] A_R;
7  wire [45:0] x;
8
9  integer_multiplication inst_1(clk, A, B[45:23], x);
10 integer_multiplication inst_2(clk, A, B[22:0], A_R);
11
12 reg  [23:0] result1_1;
13 wire [23:0] qoution_q =
14     (((({23{A_R[45]}}, A_R[45:23]}) << 23) -
15       (({{23{A_R[45]}}, A_R[45:23]}) << 13) +
16       ({{23{A_R[45]}}, A_R[45:23]}));
17
18 always @(posedge clk) begin
19     result1_1 <= x[23:0] - qoution_q;
20     result <= (result1_1 >= q) ?
21               (result1_1 - q) : result1_1;
22 end
```

5.1.2. Constant Barrett Multiplication Module

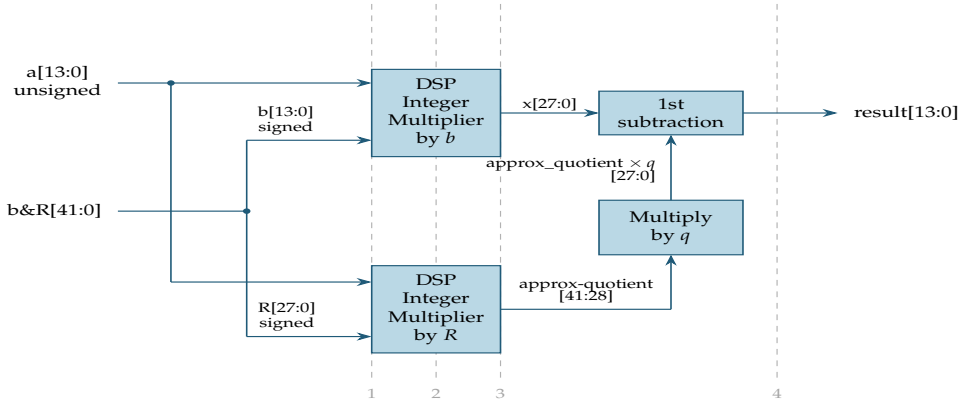
The hardware design of the Constant Barrett module illustrates how Algorithm 2 is mapped onto the DSP48E2's signed two's complement multiplier for the Falcon parameter set. Like the TMSCB design, it uses a fully pipelined DSP datapath but shortens the pipeline to four stages by omitting the final correction step. The conditional subtraction can be removed because the Constant Barrett algorithm already guarantees that the output lies in $[0, q-1]$, as shown in Figure 5a, eliminating the need for any comparator or multiplexer. Regarding the signed integer multiplication, the DSP48E2 directly handles the 14×14-bit multiplication of a and b in its built-in multiplier. For the term $a \times R$, we employ a simple trick using AND logic to make the DSP handle the 28-bit R so that it fits the DSP's 27-bit input port, as illustrated in Figure 5b. The resulting partial product is then merged through the post-adder and concatenation, just as in the TMSCB design.

While both the Constant Barrett and TMSCB modules use two DSP-based multiplies followed by subtraction, they make different trade-offs. The Constant Barrett design runs in four stages and performs full-width multiplies by b, R, and q, omitting any correction logic. In contrast, the TMSCB variant truncates all intermediates to n bits so each multiply fits in one DSP slice. This requires a fifth pipeline stage for conditional subtraction to guarantee that the result is in $[0, q-1]$. Consequently, TMSCB achieves tighter integer multiplication at the cost of one extra pipeline stage, whereas Constant Barrett employs a width of ($2n$-bit) operands to minimise latency.

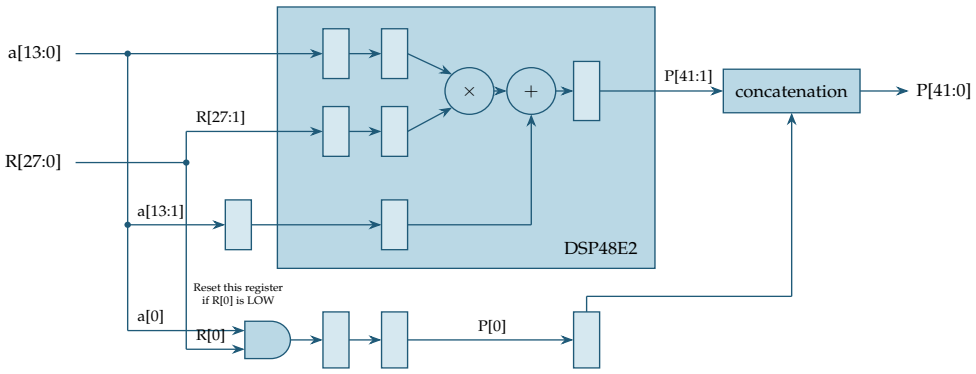
5.2. Integration in NTT Design

The Constant Modular Multiplication Modules perform the twiddle-factor multiplications within the butterfly unit in the NTT. The twiddle factors are pre-computed and stored in a ROM memory similar to several existing speed-optimised NTT implementations [11–14]. For a forward NTT of size N, the number of twiddle-factor multiplications is $\frac{N}{2}\log_2 N$. Consequently, the ML-DSA parameter set with polynomial degree $N = 256$ stores 255 distinct twiddle factors, and a single forward NTT performs 1024 twiddle-factor multiplications. Falcon-512 ($N = 512$) requires 511 twiddle factors and 2304 multiplications, whereas Falcon-1024 ($N = 1024$) needs 1023 factors and 5120 multiplications. Unlike other designs in the literature that store only the twiddle factors, the proposed design pre-computes and stores two values in ROM: the twiddle factor and its reciprocal. Both

numbers are generated offline, concatenated, and written to the same ROM address in two's complement format. Since the twiddle factor and reciprocal are concatenated and read from the same ROM address, the additional reciprocal increases the ROM word width but does not require an additional memory-access cycle.



(**a**) Top-level architecture of the Constant Barrett multiplication module.



(**b**) DSP integer multiplication by R.

Figure 5. DSP-based architectures for twiddle-factor modular multiplication, optimised for the Falcon parameter set.

The architecture of the NTT design follows a memory-based approach, chosen because the NTT algorithm consists of multiple layers of nested loops. This architecture enables the expansion of parallel butterfly units by unrolling these loops and parallelising each operation. Throughput can be adjusted by modifying the number of butterfly units, providing a tool for a more extensive evaluation of our constant multiplier. The butterfly unit is a unified structure that can be configured using a multiplexer to selectively perform either the Cooley–Tukey (CT) butterfly configuration for the NTT operation or the Gentleman–Sande (GS) butterfly configuration for the INTT operation. For the ML-DSA parameter set, the butterfly unit has a latency of seven clock cycles and can be broken down into three components, as shown in Figure 6. The first component is a modular multiplier based on the highly optimised TMSCB multiplication algorithm from the previous section. The second component consists of five clock cycle shift registers, which introduce controlled delays in the data path to match the latency of the modular multiplication module, thus achieving a fully pipelined design. The final component includes modular addition and subtraction functions and a division-by-two operation. This component operates in two clock cycles, and combining division by two with addition and subtraction helps balance the critical path, ultimately leading to a higher clock frequency. Similarly, the Falcon implementation

employs the same architecture but compresses the twiddle-factor multiplier into one fewer clock cycle, reducing the overall butterfly latency by one cycle.

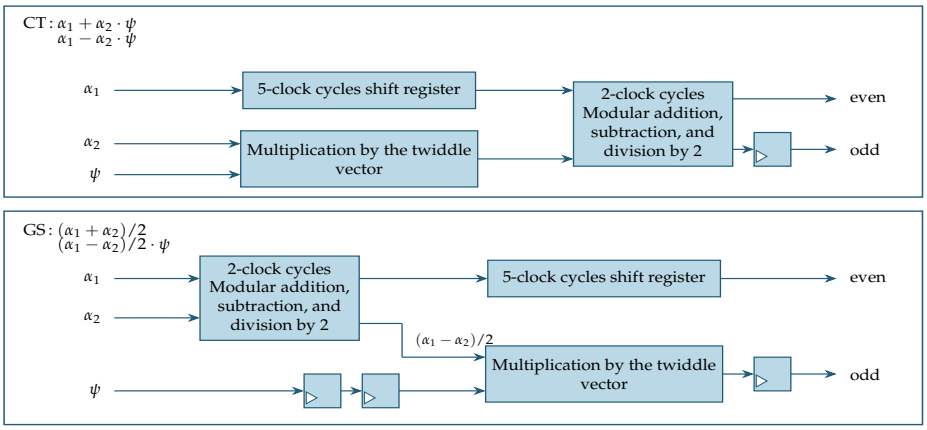


Figure 6. Top-level design for the proposed butterfly unit.

The accelerator designs for NTT/INTT primarily focus on optimising modular multiplication with the twiddle factor. While our work adopts the architectural principles of Derya et al. and Mert et al. [12,28], it extends these architectures in two ways. First, it folds the pre-processing and post-processing stages directly into the butterfly, giving the forward and inverse NTTs an identical latency of 1175 clock cycles with a single butterfly unit for the ML-DSA parameter set. Second, it adds a dedicated ROM that stores the constants for modular multiplication and optimises DSP usage for signed integer multiplication. Figure 7 illustrates the overall NTT hardware architecture.

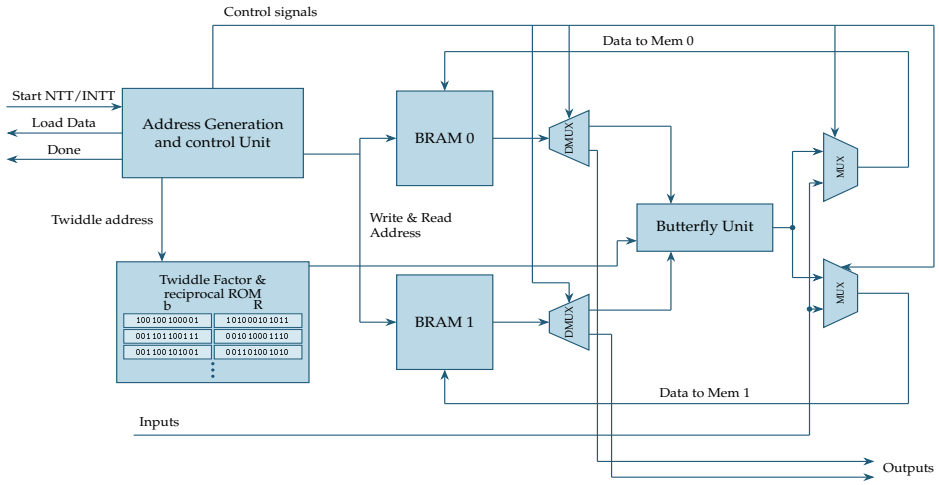


Figure 7. Overall architecture of the NTT hardware design.

6. Results and Comparison

We validated our methodology by implementing an NTT hardware design in Verilog HDL on a Xilinx FPGA Kintex UltraScale + (xcku035-fbva900-3-e) FPGA. The resource usage and maximum frequency were captured from the Xilinx Vivado 2023.1 design tool. The hardware metric used to measure the circuit area includes the number of LUTs, Regs, BRAMs, and DSPs, while the time per operation in microseconds is used to determine the design's speed through $Fmax^{-1} \times No.\,of\ clock\ cycles$. Only results obtained after the place and route were completed are reported.

In addition, for the purpose of comparison with state-of-the-art designs, the implementation is extended to support multiple parallel butterfly units featuring two, four, and eight

butterflies. For a fair comparison with other designs, the equivalent number of slices (ENS) metric is also evaluated. ENS is a metric commonly used to estimate FPGA resource usage by converting the utilisation of various resource types, such as DSPs and BRAMs, into an equivalent number of slices. This provides a more unified view of how different resources contribute to the overall utilisation of the design [38] and offers insights for ASIC design estimation. A DSP is approximated to be 102 slices, and a BRAM is approximated to be 116 slices. Finally, we use UltraScale slice parameters with eight LUTs and 16 REGs per slice as the benchmark for a unified ENS metric, and therefore, the final equation is as follows:

$$\text{Total ENS} = \max\left(\left\lceil \frac{\text{LUTs}}{8} \right\rceil, \left\lceil \frac{\text{REGs}}{16} \right\rceil\right) + (\text{DSPs} \times 102) + (\text{BRAMs} \times 116) \quad (6)$$

A key aspect of our design approach was ensuring that twiddle-factor multiplication no longer defined the critical path, thereby enhancing overall performance across multiple parameter sets. Table 3 combines results for both ML-DSA and Falcon (512, 1024), providing a detailed breakdown of resource utilisation, maximum clock frequency, and number of clock cycles, with particular emphasis on the butterfly units. The table comprises four primary modules: twiddle factor and reciprocal ROM, multiple memory blocks for coefficient storage, an address generator, and a butterfly unit.

Under the ML-DSA parameters and using Algorithm 3, the butterfly unit is the most resource-intensive part of the NTT/INTT module, consuming approximately 70% of the LUTs, representing 735 out of 1049, and 62% of the registers, representing 547 out of 890, along with two DSPs. It integrates modular addition, subtraction, division by two, and twiddle-factor multiplication, the latter being the most demanding. Each modular multiplier uses 191 LUTs, 202 registers, and two DSPs, while DSP-focused optimisations allow the single-butterfly design to reach 533 MHz, accounting for most of the speed-up gains. Meanwhile, the Falcon-512 and Falcon-1024 implementations use the Constant Barrett multiplication in Algorithm 2, showing similar performance trends. For Falcon-512, the design runs at 422 MHz, with the butterfly unit consuming 42.7% of the LUTs, representing 249 out of 583, and 50.3% of the registers, representing 247 out of 491, along with two DSPs. Falcon-1024 operates at 414 MHz and follows a resource distribution pattern similar to Falcon-512. Across both parameter sets, increasing the butterfly count consistently reduces the number of clock cycles, thereby lowering execution time and achieving a balanced trade-off between area and speed for each butterfly configuration.

Consequently, twiddle-factor multiplication no longer dictates the critical path, making data access and storage in the coefficient memory the next performance bottleneck. Nonetheless, our results confirm that improving the twiddle-factor multiplication phase is key to enhancing efficiency in NTT/INTT architectures.

Having established the baseline performance, the following part compares the proposed architecture against prior designs for each parameter set. Table 4 compares our hardware module to state-of-the-art ML-DSA, Falcon-512, and Falcon-1024 designs. We use ENS as the primary area metric and execution time as the speed metric. Overall, our design demonstrates that it is highly competitive across various configurations, offering a solid balance between speed and resource utilisation. Specifically, it achieves better results for single- and dual-butterfly-unit architectures. Below, we compare the most relevant designs, starting with the ML-DSA parameter set.

Li, Derya, and Mert each implement scalable, iterative NTT engines similar to our multi-BU architecture, but they differ in their choice of modular reduction. Li et al. [14] employ K^2-RED, whereas Derya and Mert adopt a word-level Montgomery technique. Our design improves on Li et al. by leveraging pre-stored reciprocals and Barrett multiplication, a more general-purpose approach. At the same time, we couple these algorithmic gains with efficient DSP utilisation in a pipelined datapath. This advantage is evident in the

butterfly unit comparison shown in Table 5. Li et al.'s K^2-RED butterfly implements integer multiplication using the Karatsuba–Ofman algorithm, followed by K^2-RED modular reduction using only shifts and additions. The design occupies 383 ENSs and operates at a clock frequency of 255 MHz. In contrast, our TMSCB butterfly achieves better results by requiring only 296 ENSs while supporting a higher clock frequency of 533 MHz. For the complete ML-DSA NTT design, our four-BU ML-DSA configuration achieves a lower execution time of 0.74 µs versus 1.14 µs and a reduced area with an ENS of 1761 compared to 2605. More recently, Tu and Xie [39] employed a mixed-radix architecture with Montgomery multiplication. For ML-DSA, they reported 1.42 µs with an ENS of 1446, compared with 1.20 µs and an ENS of 951 for our two-BU design.

Table 3. Detailed implementation results of hardware module for NTT.

Hardware Module	LUTs	REGs	BRAMs	DSPs	Fmax (MHz)	CCs
ML-DSA parameters $N = 256,\ q = 8{,}380{,}417,\ 23$ bits						
▷ **NTT/INTT: Single Butterfly Unit**	1049	890	2	2	533	1175
↪ Twiddle-factor ROM	0	0	1	0		
Coefficients memorised	23	0	1	0		
Address generator and control logic	276	135	0	0		
▷ **Butterfly unit**	735	547	0	2		
Modular multiplication	191	202	0	2		
Modular add/sub/div2	383	140	0	0		
Shift registers and other logic	204	115	0	0		
NTT/INTT: Two Butterfly Units	1553	1339	3	4	552	663
NTT/INTT: Four Butterfly Units	2916	2332	5	8	547	407
NTT/INTT: Eight Butterfly Units	5572	4153	9	16	481	279
Falcon-512 parameters $N = 512,\ q = 12{,}289,\ 14$ bits						
▷ **NTT/INTT: Single Butterfly Unit**	583	491	3	2	422	2452
↪ Twiddle-factor ROM	0	0	1	0		
Coefficients memorised	14	0	2	0		
Address generator and control logic	320	173	0	0		
▷ **Butterfly unit**	249	247	0	2		
Modular multiplication	44	49	0	2		
Modular add/sub/div2	52	86	0	0		
Shift registers and other logic	153	112	0	0		
NTT/INTT: Two Butterfly Units	751	982	3	4	478	1300
NTT/INTT: Four Butterfly Units	1174	1636	5	8	498	724
NTT/INTT: Eight Butterfly Units	2073	2970	9	16	486	436
Falcon-1024 parameters $N = 1024,\ q = 12{,}289,\ 14$ bits						
NTT/INTT: Single Butterfly Unit	595	674	5	2	414	5268
NTT/INTT: Two Butterfly Units	821	999	5	4	435	2708
NTT/INTT: Four Butterfly Units	1206	1651	5	8	468	1428
NTT/INTT: Eight Butterfly Units	2093	2989	9	16	458	788

▷ denotes a top-level hardware module, and ↪ denotes a constituent sub-block reported within the module immediately above.

Table 4. The comparison of our hardware module with prior work.

Hardware Module	BC [1]	LUT	REG	BRAM	DSP	fmax (MHz)	Number of (CCs)	Time (µs)	ENS [2]	Platform
ML-DSA parameters $N = 256$, $q = 8{,}380{,}417$, 23 bits										
Li et al. [14]		1623	1325	1.5	3	255	1025	4.13	684	Kintex UltraScale
Derya et al. [12]	1	2119	1058	3	8	117	1318	11.26	1429	Artix-7
Mert et al. [28]		888	-	5	7	125	1096	8.77	1475	Virtex-7
Our work		1049	890	2	2	533.333	1175	2.20	568	Kintex UltraScale+
Land et al. [13]		444	421	1	17	311	536	1.71	1906	Artix-7
Matteo et al. [15]	2	1178	655	3.5	9	275	1074	3.91	1472	Zynq UltraScale+
Tu and Xie [3] [39]		1564	1772	5.5	6	191	271	1.42	1446	Virtex-7
Our work		1553	1339	3	4	551.572	663	1.20	951	Kintex UltraScale+
Pham et al. [11]		2637	1071	1	8	385	268	0.70	1262	Kintex UltraScale
Li et al. [14]	4	5478	4955	6	12	250	284	1.14	2605	Kintex UltraScale
Dam et al. [31]		3254	2869	3	4	171	530	3.10	1163	Virtex-7
Our work		2916	2332	5	8	546.747	407	0.74	1761	Kintex UltraScale+
Taghavi et al. [40]	128	34,595	18,565	0	256	400	7	0.017	30,437	Virtex UltraScale+
Zhao et al. [9]	32	25,674	3137	6	64	540	32	0.06	10,434	Artix-7
Derya et al. [12]	8	11,000	5422	12	64	117	198	1.68	9295	Artix-7
Mert et al. [28]		5071	-	12	56	125	200	1.60	7738	Virtex-7
Li et al. [14]		11,006	8791	12	24	220	156	0.71	3840	Kintex UltraScale
Our work		5572	4153	9	16	481.232	279	0.58	3373	Kintex UltraScale+
Falcon-512 parameters $N = 512$, $q = 12{,}289$, 14 bits										
Derya et al. [12]	1	2119	1058	3	8	117	1318	22.09	1429	Artix-7
Mu et al. [30]		489	245	3	3	278	2311	8.31	716	Virtex 7
Mert et al. [28]		537	-	5.5	3	125	2340	18.72	1012	Virtex-7
Our work		583	491	3	2	422	2452	5.81	625	Kintex UltraScale+
Mu et al. [30]	2	1071	557	4	6	248	1159	4.67	1210	Virtex 7
Zhang et al. [27]		741	330	5	2	245	1289	5.26	877	Artix-7
Tu and Xie [3] [39]		956	794	5.5	3	269	629	2.34	1064	Virtex-7
Our work		751	982	3	4	478	1300	2.72	850	Kintex UltraScale+
Dam et al. [31]	4	3254	2869	3	4	171	594	3.47	1163	Virtex-7
Our work		1174	1636	5	8	498	724	1.45	1543	Kintex UltraScale+
Mert et al. [28]		2514	-	12	24	125	324	2.59	4155	Virtex-7
Our work	8	2073	2970	9	16	486	436	0.90	2936	Kintex UltraScale+
Falcon-1024 parameters $N = 1024$, $q = 12{,}289$, 14 bits										
Mu et al. [30]	1	515	306	3	3	278	5127	18.44	719	Virtex 7
Mert et al. [28]		575	-	11	3	125	5160	41.28	1654	Virtex-7
Our work		595	674	5	2	414	5268	12.72	859	Kintex UltraScale+
Mu et al. [30]	2	1205	572	4	6	256	2567	10.03	1227	Virtex 7
Zhang et al. [27]		847	375	6	2	244	2569	10.53	1006	Artix-7
Tu and Xie [3] [39]		621	534	5	3	261	1391	5.33	964	Virtex-7
Our work		821	999	5	4	435	2708	6.23	1091	Kintex UltraScale+
Mu et al. [30]	4	1467	930	4.5	13	189	1294	6.85	2032	Virtex 7
Dam et al. [31]		3254	2869	3	4	171	1348	7.88	1163	Virtex-7
Our work		1206	1651	5	8	468	1428	3.06	1547	Kintex UltraScale+
Li et al. [14]	16	22,648	15,030	24	48	200	343	1.72	10,511	Kintex UltraScale
Mu et al. [30]	8	3472	1789	7.5	25	179	654	3.65	3854	Virtex 7
Mert et al. [28]		2584	-	16	24	125	680	5.44	4627	Virtex-7
Hu et al. [41]	8	2801	2680	6	8	250	670	2.68	1863	Artix-7
Our work		2093	2989	9	16	458	788	1.71	2938	Kintex UltraScale+

[1] Butterfly count: Number of butterfly units used in the design. [2] A single DSP and BRAM blocks equivalent to 102 and 116 slices, respectively. [3] EMINEM uses one mixed-radix butterfly with four GS processing elements and three Montgomery multipliers, roughly comparable to two radix-2 butterfly units.

Compared with other scalable designs, Derya et al. [12] and Mert et al. [28] share architectural similarities with our work but incur higher arithmetic cost due to their use of configurable Montgomery reduction. Their single-BU NTT cores achieve execution times of 11.26 µs with ENS 1429 and 8.77 µs with ENS 1475, respectively, for ML-DSA. In contrast, our Barrett-based design achieves only 2.20 µs with ENS 568, demonstrating better efficiency in both area and time. Derya et al.'s word-level Montgomery butterfly unit offloads reduction work into DSP arrays (see Table 5), consuming 705 LUTs, 488 REGs, and eight DSPs, trading generality for performance; by contrast, our use of Barrett reduction offers a reduced butterfly area and faster clock frequency.

Next, we compare fixed-configuration NTT designs, each tailored to a specific parameter set, butterfly count, and memory access pattern, against our scalable architecture. Pham et al.'s optimised classical Barrett butterfly hardwires a constant 23-bit multiplier across two cascaded DSPs [11]. It minimises logic to only 351 LUTs and 209 REGs alongside two DSPs, as shown in Table 5. It uses an Optimised Barrett reduction with bit-level constant multiplication by q and R, which limits the clock frequency to 385 MHz due to the dependency presented on the classical Barrett structure. Their NTT core achieves 0.70 µs latency and an ENS of 1262 but is tightly integrated with the ML-DSA parameter set and lacks support for configurable butterfly unit counts. By contrast, our four-BU design for ML-DSA reaches a similar execution time of 0.74 µs and an ENS of 1761 at 551 MHz due to pipelined TMSCB multiplication, which removes this dependency. Our butterfly unit occupies 735 LUTs and 547 REGs but achieves a higher clock frequency. Land et al. [13] target logic-area efficiency in a two-BU ML-DSA design, offloading reduction into 17 DSPs to achieve a time of 1.71 µs and an ENS of 1906 by prioritising low LUT and REG usage. However, their heavy reliance on DSPs resulted in higher overall ENS. In contrast, our design achieves an improved time of 1.20 µs and an ENS of 951, outperforming Land et al. using two butterfly units.

Compared to highly parallel architectures, which achieve very low latency at the cost of substantial hardware resources, our approach maintains a high clock frequency while keeping resource usage at a more reasonable level. For instance, our largest memory-based ML-DSA design uses eight butterfly units and achieves 0.58 µs with an ENS of 3373, compared with Zhao et al.'s systolic array design using 32 butterfly units, which achieves 0.06 µs with a considerably higher ENS of 10434. Similarly, Taghavi et al. [40] employ a highly parallel architecture with 128 butterfly units, achieving 0.017 µs latency but with a substantially higher ENS of 30437. These results confirm that our architecture delivers speed and area advantages for the ML-DSA parameter set.

Table 5. Comparison of butterfly units in NTT architectures within ML-DSA.

Butterfly Unit Module	LUTs	REGs	DSPs	Fmax (MHz)	ENS
K2-RED (Li et al. [14])	615	604	3	255	383
Word-level Montgomery (Derya et al. [12])	705	488	8	117	905
Optimised Barrett (Pham et al. [11])	351	209	2	385	248
TMSCB (this work)	735	547	2	533	296

The performance of the proposed architecture is next evaluated for the Falcon-512 and Falcon-1024 parameter sets. For Falcon-512, our one–eight-BU configurations achieve execution times from 5.81 µs down to 0.90 µs with an ENS spanning 625 to 2936. For Falcon-1024, our implementations similarly reduce execution time from 12.72 µs with one BU to 1.71 µs with eight BUs, with ENS values between 859 and 2938. More recently, Hu et al. [41] reported an eight-butterfly Falcon-1024 design achieving 2.68 µs with an ENS of 1863, compared with 1.71 µs and an ENS of 2938 for our design.

Derya et al. [12], Mert et al. [28], and Li et al. [14] each examine both ML-DSA and Falcon, providing a direct opportunity to compare the portability of similar architectures across parameter sets. While our design already outperforms prior work for ML-DSA at degree 256, this advantage becomes even more pronounced as the polynomial degree increases for Falcon-512 and Falcon-1024. Derya et al.'s execution time is 22.09 µs with an ENS of 1429, compared to our 6.01 µs with an ENS of 625. At Falcon-1024, Mert et al.'s execution time is 41.28 µs with an ENS of 1654, while our design achieves a faster execution time of 12.72 µs with a lower ENS of 859. Li et al. [14] reach 1.72 µs for Falcon-1024 using 16 butterfly units and an ENS of 10511; in contrast, our design achieves 1.71 µs with only eight butterflies and a 3.6× area reduction, demonstrating a substantial area saving at comparable latency. These results confirm that the proposed architecture delivers practical speed and area improvements for real-world lattice-based cryptographic implementations.

7. Conclusions

By introducing and thoroughly evaluating the Truncated-Modulus-Size Constant Barrett (TMSCB) multiplier alongside a Constant Barrett variant, this work demonstrates that tailoring Barrett reduction for a fixed twiddle-factor operand speeds up NTT/INTT computations. An FPGA implementation targeting ML-DSA and Falcon parameter sets confirms that optimisation efforts focused on constant-operand modular multiplication are an effective way to accelerate NTT computations. In particular, aligning internal datapaths to the modulus word length yields area–time advantages. The standalone multiplier results for larger operand widths, including the 64-bit case, indicate that the relative arithmetic advantage of TMSCB increases as the modulus width grows. However, the present work does not evaluate a complete large-modulus NTT/INTT architecture or a full homomorphic-encryption implementation. Therefore, extending and validating the proposed approach using representative FHE parameter sets remains an important direction for future work. Future investigations should also consider different NTT architectures and other constant modular multiplication methods, including Montgomery- and K-RED-based approaches.

Author Contributions: Conceptualization, A.M.A. and M.B.; methodology, A.M.A.; software, A.M.A.; validation, A.M.A. and M.B.; formal analysis, A.M.A.; investigation, A.M.A.; resources, M.B.; data curation, A.M.A.; writing—original draft preparation, A.M.A.; writing—review and editing, A.M.A. and M.B.; visualization, A.M.A.; supervision, M.B.; project administration, M.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data supporting the reported results are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Shor, P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.* **1999**, *41*, 303–332. [CrossRef]
2. Chen, L.; Jordan, S.; Liu, Y.K.; Moody, D.; Peralta, R.; Perlner, R.; Smith-Tone, D. *Report on Post-Quantum Cryptography*; NIST Interagency/Internal Report 8105; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2016. [CrossRef]
3. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM* **2009**, *56*, 34:1–34:40. [CrossRef]
4. Nejatollahi, H.; Dutt, N.; Ray, S.; Regazzoni, F.; Banerjee, I.; Cammarota, R. Post-quantum lattice-based cryptography implementations: A survey. *ACM Comput. Surv.* **2019**, *51*, 129:1–129:41. [CrossRef]

5. Pöppelmann, T.; Güneysu, T. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *Proceedings of the Progress in Cryptology—LATINCRYPT 2012*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7533, pp. 139–158. [CrossRef]
6. Alagic, G.; Apon, D.; Cooper, D.; Dang, Q.; Dang, T.; Kelsey, J.; Lichtinger, J.; Miller, C.; Moody, D.; Peralta, R.; et al. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*; NIST Interagency/Internal Report 8413; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2022. [CrossRef]
7. *FIPS 204*; Module-Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication 204. National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024. [CrossRef]
8. Bai, S.; Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schwabe, P.; Seiler, G.; Stehlé, D. *CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation, Version 3.1*; Submission to the NIST Post-Quantum Cryptography Standardization Process, Round 3; Specification document; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2021. Available online: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf (accessed on 26 August 2026).
9. Zhao, Y.; Xie, R.; Xin, G.; Han, J. A high-performance domain-specific processor with matrix extension of RISC-V for module-LWE applications. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, *69*, 2871–2884. [CrossRef]
10. Rentería-Mejía, C.P.; Velasco-Medina, J. Lattice-based cryptoprocessor for CCA-secure identity-based encryption. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 2331–2344. [CrossRef]
11. Pham, T.X.; Duong-Ngoc, P.; Lee, H. An efficient unified polynomial arithmetic unit for CRYSTALS-Dilithium. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2023**, *70*, 4854–4864. [CrossRef]
12. Derya, K.; Mert, A.C.; Öztürk, E.; Savaş, E. CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication. *Microprocess. Microsyst.* **2022**, *89*, 104451. [CrossRef]
13. Land, G.; Sasdrich, P.; Güneysu, T. A hard crystal—Implementing Dilithium on reconfigurable hardware. In *Proceedings of the Smart Card Research and Advanced Applications*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2022; Volume 13173, pp. 210–230. [CrossRef]
14. Li, B.; Yan, Y.; Wei, Y.; Han, H. Scalable and parallel optimization of the number theoretic transform based on FPGA. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2024**, *32*, 291–304. [CrossRef]
15. Di Matteo, S.; Sarno, I.; Saponara, S. CRYPHTOR: A memory-unified NTT-based hardware accelerator for post-quantum CRYSTALS algorithms. *IEEE Access* **2024**, *12*, 25501–25511. [CrossRef]
16. Mazonka, O.; Thari Moopan, M.N.; Maniatakos, M. Exploring generalization of Shoup modular multiplier. In *Proceedings of the Great Lakes Symposium on VLSI 2024*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 222–227. [CrossRef]
17. Montgomery, P.L. Modular multiplication without trial division. *Math. Comput.* **1985**, *44*, 519–521. [CrossRef]
18. Barrett, P. Implementing the Rivest–Shamir–Adleman public-key encryption algorithm on a standard digital signal processor. In *Proceedings of the Advances in Cryptology—EUROCRYPT '86*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1986; Volume 263, pp. 311–323. [CrossRef] [PubMed]
19. Longa, P.; Naehrig, M. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Proceedings of the Cryptology and Network Security*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2016; Volume 10052, pp. 124–139. [CrossRef]
20. Becker, H.; Hwang, V.; Kannwischer, M.J.; Yang, B.Y.; Yang, S.Y. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2022**, *2022*, 221–244. [CrossRef]
21. Nguyen, D.T.; Gaj, K. Fast Falcon signature generation and verification using ARMv8 NEON instructions. In *Proceedings of the Progress in Cryptology—AFRICACRYPT 2023*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2023; Volume 14064, pp. 417–441. [CrossRef]
22. Pöppelmann, T.; Oder, T.; Güneysu, T. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *Proceedings of the Progress in Cryptology—LATINCRYPT 2015*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2015; Volume 9230, pp. 346–365. [CrossRef]
23. Roy, S.S.; Vercauteren, F.; Mentens, N.; Chen, D.D.; Verbauwhede, I. Compact ring-LWE cryptoprocessor. In *Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2014*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8731, pp. 371–391. [CrossRef]
24. Alhassani, A.; Benaissa, M. ML-KEM (CRYSTALS-Kyber) on FPGA using the residue number system. *Cryptography* **2026**, *10*, 47. [CrossRef]
25. Cooley, J.W.; Tukey, J.W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **1965**, *19*, 297–301. [CrossRef]
26. Gentleman, W.M.; Sande, G. Fast Fourier transforms: For fun and profit. In *Proceedings of the November 7–10, 1966, Fall Joint Computer Conference*; Association for Computing Machinery: New York, NY, USA, 1966; pp. 563–578. [CrossRef]

27. Zhang, N.; Yang, B.; Chen, C.; Yin, S.; Wei, S.; Liu, L. Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 49–72. [CrossRef]
28. Mert, A.C.; Karabulut, E.; Öztürk, E.; Savaş, E.; Aysu, A. An extensive study of flexible design methods for the number theoretic transform. *IEEE Trans. Comput.* **2022**, *71*, 2829–2843. [CrossRef]
29. Banerjee, U.; Ukyab, T.S.; Chandrakasan, A.P. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2019**, *2019*, 17–61. [CrossRef]
30. Mu, J.; Ren, Y.; Wang, W.; Hu, Y.; Chen, S.; Chang, C.H.; Fan, J.; Ye, J.; Cao, Y.; Li, H.; et al. Scalable and conflict-free NTT hardware accelerator design: Methodology, proof, and implementation. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2023**, *42*, 1504–1517. [CrossRef]
31. Dam, D.T.; Nguyen, T.H.; Tran, T.H.; Le, D.H.; Hoang, T.T.; Pham, C.K. High-efficiency multi-standard polynomial multiplication accelerator on RISC-V SoC for post-quantum cryptography. *IEEE Access* **2024**, *12*, 195015–195031. [CrossRef]
32. Plantard, T. Efficient word-size modular arithmetic. *IEEE Trans. Emerg. Top. Comput.* **2021**, *9*, 1506–1518. [CrossRef]
33. Huang, J.; Zhang, J.; Zhao, H.; Liu, Z.; Cheung, R.C.C.; Koç, Ç.K.; Chen, D. Improved Plantard arithmetic for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2022**, *2022*, 614–636. [CrossRef]
34. Langhammer, M.; Pasca, B. Efficient FPGA modular multiplication implementation. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 217–223. [CrossRef]
35. Hwang, V.; Kim, Y.; Seo, S.C. Multiplying polynomials without powerful multiplication instructions. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2025**, *2025*, 160–202. [CrossRef]
36. Satriawan, A.; Mareta, R.; Lee, H. Integer modular multiplication with Barrett reduction and its variants for homomorphic encryption applications: A comprehensive review and an empirical study. *IEEE Access* **2024**, *12*, 147283–147300. [CrossRef]
37. Shoup, V. NTL: A Library for Doing Number Theory. Software Library and Documentation. 2001. Available online: https://libntl.org/ (accessed on 5 August 2026).
38. Liu, W.; Fan, S.; Khalid, A.; Rafferty, C.; O'Neill, M. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 2459–2463. [CrossRef]
39. Tu, Y.; Xie, J. EMINEM: Efficient FPGA Implementation of Mixed-RadIx NTT Hardware AccElerators for NIST Post-QuantuM Cryptography Falcon, Dilithium, and HAWK. *ACM Trans. Reconfigurable Technol. Syst.* **2025**, *18*, 1–25. [CrossRef]
40. Taghavi, B.; Azarderakhsh, R.; Mozaffari Kermani, M. ParallelNTT: Maximizing Performance of Forward and Inverse NTT on FPGA for ML-DSA and ML-KEM. In *Proceedings of the Proceedings of the Great Lakes Symposium on VLSI 2025*; Association for Computing Machinery: New York, NY, USA, 2025; pp. 372–378. [CrossRef]
41. Hu, H.; Chu, Y.; Du, G.; Wang, X.; Song, Y.; Zhang, D.; Li, Z. A Falcon Signature Verification Accelerator Using Area-Efficient NTT Architecture With Simplified Barrett Modular Multiplier. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2026**, *34*, 860–871. [CrossRef]