IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2020, No. 2, pp. 49–72. DOI:10.13154/tches.v2020.i2.49-72

Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT

Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei and Leibo Liu*

Institute of Microelectronics, Tsinghua University, Beijing, 100084, China.
zhangn16@mails.tsinghua.edu.cn;{bohanyang,chenchen1984,yinsy,wsj,liulb}@tsinghua.edu.cn
* Corresponding Author.

Abstract. NewHope-NIST is a promising ring learning with errors (RLWE)-based post-quantum cryptography (PQC) for key encapsulation mechanisms. The performance on the field-programmable gate array (FPGA) affects the applicability of NewHope-NIST. In RLWE-based PQC algorithms, the number theoretic transform (NTT) is one of the most time-consuming operations. In this paper, low-complexity NTT and inverse NTT (INTT) are used to implement highly efficient NewHope-NIST on FPGA. First, both the pre-processing of NTT and the post-processing of INTT are merged into the fast Fourier transform (FFT) algorithm, which reduces N and $2N$ modular multiplications for N-point NTT and INTT, respectively. Second, a compact butterfly unit and an efficient modular reduction on the modulus 12289 are proposed for the low-complexity NTT/INTT architecture, which achieves an improvement of approximately 3× in the area time product (ATP) compared with the results of the state-of-the-art designs. Finally, a highly efficient architecture with doubled bandwidth and timing hiding for NewHope-NIST is presented. The implementation results on an FPGA show that our design is at least 2.5× faster and has 4.9× smaller ATP compared with the results of the state-of-the-art designs of NewHope-NIST on similar platforms.

Keywords: NewHope, FPGA, post-quantum cryptography, ring learning with errors, number theoretic transform

Introduction

Conventional public key cryptography algorithms, such as RSA and ECC, can be broken by implementing the Shor algorithm [Sho94] on a quantum computer. With the development of quantum computers, there is an urgent need to replace RSA and ECC with cryptographic algorithms resisting quantum computing attacks, which are known as post-quantum cryptography (PQC). The National Institute of Standards and Technology (NIST) started the PQC Standardization Process in 2016 [NIS16]. A total of 69 adequate candidate algorithms were submitted during the first round of competition, and 26 submissions survived to the second round. Lattice-based cryptography is a promising family of PQC schemes because of its high speed, moderately larger key size [ACZ18] and long-standing open problem for classical computation[BBD09]. The lattice-based candidates accounted for 38% and 46% of the two rounds of NIST PQC process, respectively.

NewHope[AAB+19], called NewHope-NIST in this paper to distinguish from NewHope-Simple [ADPS16b] and NewHope-USENIX [ADPS16a], is a lattice-based candidate for key encapsulation mechanism (KEM) in the second round of the NIST PQC standardization process. The security of NewHope-NIST is based on the hardness of the ring learning

Received: 2019-10-15 Accepted: 2019-12-15 Published: 2020-03-02

with errors (RLWE) problem [AASA+19]. NewHope-NIST is based on NewHope-Simple, a variant of NewHope-USENIX. NewHope-USENIX is an RLWE-based key exchange scheme proposed by Alkim et al. It was evaluated in a version of the SSL/TLS protocol for the key establishment by Google in its Chrome browser and has demonstrated its practicality [Bra16]. NewHope-Simple improved NewHope-USENIX by avoiding error reconciliation at the cost of a slightly larger communication load between the server and client. NewHope-NIST uses the same encryption-based approach as NewHope-Simple and uses a modulus switching technique to reduce the bandwidth requirement. The main mathematical objects in NewHope-NIST are polynomials over $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle x^N + 1\rangle$, where q is 12289, and N is 1024 or 512.

The performance on different platforms affects the applicability of the PQC algorithms. The field-programmable gate array (FPGA) is one of the most important platforms for implementing cryptographic algorithms because of the excellent balance between flexibility and performance. Although performance evaluation did not play a major role in the early portion of the evaluation process of the NIST PQC process, this aspect was considered for the second-round candidates [Moo19]. This paper presents a highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT without pre-processing and low-complexity INTT without post-processing.

Related Work There have been studies on the hardware architecture of NewHope. [OG19] provided the first FPGA implementation of NewHope-Simple on a low-cost FPGA. [KLC+17] proposed a high-performance implementation with four butterfly units (BFUs) and a variant of Montgomery reduction for NewHope-USENIX. [JGCS19] designed a cryptoprocessor supporting the three NewHope schemes on an FPGA. [FSM+19] implemented NewHope-NIST using a RISC-V-based SoC with an NTT accelerator and a SHA accelerator. [BPC19, BUC19a] designed and fabricated cryptographic processors consisting of RLWE-based PQC modules in the TSMC 40nm process. Optimizing the NTT implementation has the top priority considering that NTT is the most time-consuming component in most RLWE-based cryptography. In addition to improving the architecture of NTT, reducing computational complexity, which is expressed as the number of modular multiplications, is one of the most critical optimization methods. [PG12, APS13] used Negative Wrapped Convolution (NWC) [Win96] to reduce the complexity of polynomial multiplication. [RVM+14] further decreased the complexity of forward NTT by avoiding the pre-processing of NTT with the twiddle factors computed on-the-fly in an RLWE cryptoprocessor. This idea was adopted by some subsequent works [Win96, DB16, RVVV17, FS19]. Unfortunately, this optimization cannot be applied to inverse NTT (INTT). [FS19] avoided the extra cycles for the post-processing of INTT at the cost of additional hardware multipliers, but this method does not reduce the number of multiplications. [DB16] optimized the number of modular multiplications of post-processing by pre-computing $N^{-1}\gamma_{2N}^{-i}$, but the cost is that the memory for pre-computed twiddle factors of INTT was increased from N to $3N/2$. [POG15] reduced the complexity of INTT by avoiding the scaling of γ_{2N}^{-i} after INTT in a software implementation on an 8-bit processor. This method does not increase the memory for pre-computed twiddle factors as [DB16]. In this method, the scaling of N^{-1} was hidden with a pre-computed NTT, but not eliminated. Modular multiplication is one of the most sophisticated arithmetic operations in NTT. Lazy modular reduction allows the results not to be fully reduced. This method can avoid some reduction steps for software implementation on a processor with a data bus width larger than the bit size of the processed data [SD18]. However, this method requires additional resources for custom hardware. The commonly used Barrett reduction and Montgomery reduction require additional multiplications and are more suitable for the non-specific modulus. For the specific moduli 7681 and 12289, [LSSR+15] and [RV17] used a shift-addition-multiplication-subtraction-subtraction (SAMS2) technique and reduced

the number of multiplications for modular reduction. However, potential timing attacks might be enabled by data-dependent operations. [PPM17] successfully gained information on the secret polynomial from modular reduction because the execution time of modular reduction is related to the bit size of a dividend used in the reduction algorithm.

Contribution This paper proposes a highly efficient architecture of NewHope-NIST using the low-complexity NTT/INTT, which is generally applicable to scenarios where the NWC is used. In particular, the main contributions are as follows:

1. A low-complexity INTT is proposed by merging the post-processing ($N^{-1}\gamma_{2N}^{-i}$) of INTT into the Gentleman-Sande decimation-in-frequency (DIF) FFT algorithm [GS66]. This method reduces the number of modular multiplications of the INTT from $(N/2)\log_2 N + 2N$ to $(N/2)\log_2 N$ compared with naive implementation, accounting for 30.8% and 28.6% modular multiplications for 512-point INTT and 1024-point INTT, respectively.

2. A compact BFU supporting both DIT and DIF, along with efficient and constant time modular reductions without additional multiplications utilizing the characteristic of $2^{14} \equiv 2^{12} - 1 \pmod{12289}$, is proposed to design a highly efficient architecture supporting both the low-complexity NTT and INTT. This architecture achieves the best performance and an improvement of approximately 3× in the area time product (ATP) compared with the results of state-of-the-art designs.

3. Architectural optimization, i.e., bandwidth doubling to match the memories and processing units, as well as timing hiding between operations, is used to design a highly efficient and constant time architecture for NewHope-NIST, which achieves at least 2.5× faster and 4.9× smaller ATPs than other NewHope-NIST implementations on similar devices.

1 Background

1.1 NewHope-NIST

NewHope-NIST uses a public key encryption (PKE) scheme to construct a KEM. The key generation, encryption and decryption of the PKE scheme of NewHope-NIST are shown in Algorithm 1, 2 and 3, respectively [AAB+19]. SHAKE is a family of strong hash functions [NIS15]. The subfunction Sample performs binomial sampling from the output of SHAKE256, and GenA performs reject sampling from the output of SHAKE128. The polynomial multiplication in NewHope-NIST is explicitly accelerated with NTT. Because the results of the subfunction Sample are noise polynomials, they can be considered to be already bit-reversed to eliminate the reorder operation before NTT. The subfunctions Compress and Decompress mainly perform modulus switching to reduce the bandwidth

Algorithm 1 NewHope-CPA-PKE Key Generation

function NewHope-CPA-PKE.Gen()
1: $seed \leftarrow \{0, ..., 255\}^{32}$
2: $(noiseseed, publicseed) \leftarrow SHAKE256(64, seed)$
3: $\hat{\boldsymbol{a}} \leftarrow GenA(publicseed)$
4: $\boldsymbol{s} \leftarrow PolyBitRev(Sample(noiseseed, 0))$
5: $\hat{\boldsymbol{s}} \leftarrow NTT(\boldsymbol{s})$
6: $\boldsymbol{e} \leftarrow PolyBitRev(Sample(noiseseed, 1))$
7: $\hat{\boldsymbol{e}} \leftarrow NTT(\boldsymbol{e})$
8: $\hat{\boldsymbol{b}} \leftarrow \hat{\boldsymbol{a}} \circ \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}$
9: **return** $(pk = EncodePK(\hat{\boldsymbol{b}}, publicseed), sk = EncodePolynomial(\hat{\boldsymbol{s}}))$

Algorithm 2 NewHope-CPA-PKE Encryption

function NewHope-CPA-PKE.Encrypt($pk, \mu, coin$)

1: $(\hat{\boldsymbol{b}}, publicseed) \leftarrow DecodePk(pk)$
2: $\hat{\boldsymbol{a}} \leftarrow GenA(publicseed)$
3: $\boldsymbol{s'} \leftarrow PolyBitRev(Sample(noiseseed, 0))$
4: $\boldsymbol{e'} \leftarrow PolyBitRev(Sample(noiseseed, 1))$
5: $\boldsymbol{e''} \leftarrow Sample(coin, 2)$
6: $\hat{\boldsymbol{t}} \leftarrow NTT(\boldsymbol{s'})$
7: $\hat{\boldsymbol{u}} \leftarrow \hat{\boldsymbol{a}} \circ \hat{\boldsymbol{t}} + NTT(e')$
8: $\boldsymbol{v} \leftarrow Encode(\mu)$
9: $\boldsymbol{v'} \leftarrow INTT(\hat{\boldsymbol{b}} \circ \hat{\boldsymbol{t}}) + \boldsymbol{e''} + \boldsymbol{v}$
10: $h \leftarrow Compress(\boldsymbol{v'})$
11: **return** $c = EncodeC(\hat{\boldsymbol{u}}, h)$

Algorithm 3 NewHope-CPA-PKE Decryption

Function NewHope-CPA-PKE.Decrypt(c, sk)

1: $(\hat{\boldsymbol{u}}, h) \leftarrow DecodeC(c)$
2: $\hat{\boldsymbol{s}} \leftarrow DecodePolynomial(sk)$
3: $\boldsymbol{v'} \leftarrow Decompress(h)$
4: $\mu \leftarrow Decode(\boldsymbol{v'} - INTT(\hat{\boldsymbol{u}} \circ \hat{\boldsymbol{s}}))$
5: **return** μ

requirement. Encode and Decode perform message encoding and message decoding, respectively, which convert the message to a polynomial in $\mathbb{R}_q$, and vice versa. The subfunctions EncodePK, DecodePK, EncodePolynomial, DecodePolynomial, EncodeC, and DecodeC perform the conversion between polynomial and byte arrays only. The main time-consuming operations in the above algorithms are NTT and INTT. Specifically, key generation involves two NTTs; encryption involves two NTTs and one INTT; decryption involves one INTT.

1.2 NTT and INTT

The classic NTT is defined over a finite field $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. It is derived from replacing the complex arithmetic in the discrete Fourier transform (DFT) with modular arithmetic. The classic NTT is defined as $\hat{a}_i = INTT_N(\hat{\boldsymbol{a}})_i = \sum_{j=0}^{N-1} a_j \omega_N^{ij} \bmod q$, where $i = 0, 1, ..., N-1$, ω_N denotes a primitive N-th root of unity in $\mathbb{Z}_q$. The classic INTT can be obtained by replacing ω_N of the classic NTT by ω_N^{-1} and multiplying by the final scalar N^{-1} after the summation. The classic INTT is defined as $a_i = INTT_N(\hat{\boldsymbol{a}})_i = N^{-1} \sum_{j=0}^{N-1} \hat{a}_j \omega_N^{-ij} \bmod q$, where $i = 0, 1, ..., N-1$. The NTT and INTT can be evaluated with FFT over the finite field $\mathbb{Z}_q$.

According to the convolution theory [Win96], polynomial multiplication can be performed as

$$INTT_{2N}(NTT_{2N}(zeropadding(\boldsymbol{a})) \odot NTT_{2N}(zeropadding(\boldsymbol{b}))), \tag{1}$$

where the symbol $\odot$ denotes point-wise multiplication and the function $zeropadding(\boldsymbol{a})$ expands the length of $\boldsymbol{a}$ from N to $2N$ with zeros at the end. Thus, multiplication over the ring $\mathbb{Z}_q[x]/\langle f(x)\rangle$ can be calculated with three $2N$-point NTT/INTT, followed by a reduction with the modular polynomial $f(x)$.

When $f(x)$ is set to be a special form $x^N - 1$, multiplication over $\mathbb{Z}_q[x]/\langle f(x)\rangle$ can be efficiently performed using positive wrapped convolution, which requires three N-point NTT/INTT without doubling the size of NTT/INTT as Equation 2. And the reduction of $f(x)$ is performed for free.

$$\boldsymbol{c} = INTT_N(NTT_N(\boldsymbol{a}) \odot NTT_N(\boldsymbol{b})). \tag{2}$$

When $f(x)$ is changed from $x^N - 1$ to another special form $x^N + 1$, the multiplication over $\mathbb{Z}_q[x]/\langle x^N + 1\rangle$ can be performed with NWC, which still requires only three N-point NTT/INTT without explicit reduction as the multiplication over $\mathbb{Z}_q[x]/\langle x^N - 1\rangle$. The NWC method asks the prime q to satisfy $q \equiv 1 \pmod{2N}$, then ω_N and its square root γ_{2N} exist. However, this method introduces a "twist" that appends pre-processing before NTT and post-processing after INTT as detailed below. Let $\bar{a}_i = a_i\gamma_{2N}^i, \bar{b}_i = b_i\gamma_{2N}^i, \bar{c}_i = c_i\gamma_{2N}^i$. To compute $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$ over $\mathbb{Z}_q[x]/\langle x^N + 1\rangle$, NWC is performed as

$$\bar{\boldsymbol{c}} = INTT_N(NTT_N(\bar{\boldsymbol{a}}) \odot NTT_N(\bar{\boldsymbol{b}})). \tag{3}$$

In this way, classic N-point NTT is performed on the scaled vector $\bar{\boldsymbol{a}}$ and $\bar{\boldsymbol{b}}$; after the classic N-point INTT, the scaled vector $\bar{\boldsymbol{c}}$ is obtained; the final result $\boldsymbol{c}$ can be recovered by computing $c_i = \bar{c}_i\gamma_{2N}^{-i}$. Thus, the NWC method avoids doubling the size of NTT/INTT and the explicit reduction, but it requires the coefficients to be scaled with γ_{2N}^i before NTT, and the results scaled with γ_{2N}^{-i} after INTT.

In this article, the **pre-processing** denotes the coefficient-wise multiplications of a_i and γ_{2N}^i before NTT. Note that INTT is generally implemented by FFT and final scaling by N^{-1}. As a result, we use **post-processing** to denote the final scaling by N^{-1} in the classic INTT and the coefficient-wise multiplication by γ_{2N}^{-i} after the classic INTT.

Properties of the Twiddle Factor. If N is a power of two, the twiddle factors of NTT has three important properties, similar as that of the twiddle factors in FFT:

$$\begin{aligned} \text{symmetry property:} \quad & \omega_N^{k+N/2} = -\omega_N^k \\ \text{periodicity property:} \quad & \omega_N^{k+N} = \omega_N^k \\ \text{scale property:} \quad & \omega_{N/m}^{k/m} = \omega_N^k, \end{aligned} \tag{4}$$

where m is also a power of two and smaller than N. Because $\gamma_{2N}^2 \equiv \omega_N$, it is easy to see that γ_{2N} has similar properties as ω_N.

2 Low-Complexity NTT and INTT

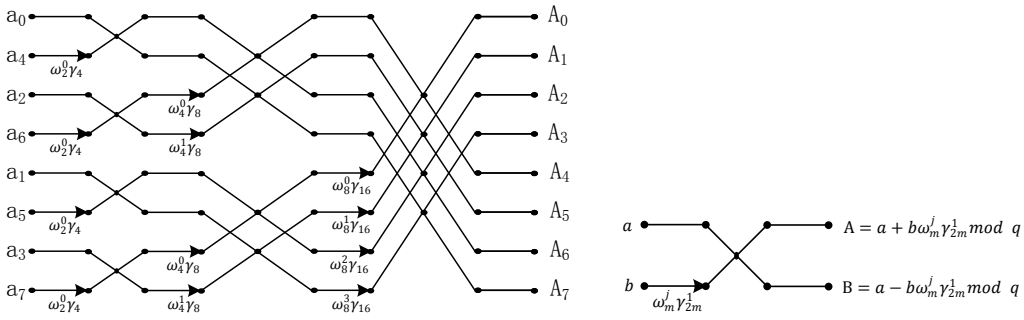
2.1 Low-Complexity NTT

When straightforwardly calculating the forward NTT as [KLC+17, JGCS19, FSM+19, FS19], the main FFT requires $(N/2)\log_2 N$ modular multiplications and the pre-processing requires N modular multiplications. For the point N being 512 and 1024, the number of modular multiplications of pre-processing accounts for 22.2% and 20% of that of the main FFT algorithm, respectively. [RVM+14] proposed a low-complexity NTT with twiddle factors computed on-the-fly. This method merged the pre-processing into the Cooley-Turkey DIT FFT algorithm by changing the initialization of the twiddle factors in the algorithm. Based on this idea, the low-complexity NTT with twiddle factors pre-computed and stored in memories, as well as the derivation process, is presented in this subsection for the completeness of this article. The method merges the pre-processing into the DIT FFT by merely changing the value of the pre-computed twiddle factors.

The derivation of the low-complexity NTT is inspired by the strategy of the Cooley-Turkey FFT [CT65]. We follow the divide-and-conquer method of FFT that divides in time domain. First, the pre-processing and the main NTT are written together as a summation of N items:

$$\hat{a}_i = \sum_{j=0}^{N-1} a_j\gamma_{2N}^j\omega_N^{ij} \bmod q, \tag{5}$$

where $i = 0, 1, ..., N-1$.



(a) Dataflow (b) Butterfly

Figure 1: Dataflow of 8-point Low-Complexity NTT

By splitting the summation into two groups according to parity of the index of a, Equation 5 can be rewritten as follows:

$$\hat{a}_i = \sum_{j=0}^{N/2-1} a_{2j}\omega_N^{2ij}\gamma_{2N}^{2j} + \sum_{j=0}^{N/2-1} a_{2j+1}\omega_N^{i(2j+1)}\gamma_{2N}^{2j+1} \bmod q, \quad i = 0, 1, ..., N-1. \tag{6}$$

With the scale property of twiddle factors, Equation 6 can be expressed as

$$\hat{a}_i = \sum_{j=0}^{N/2-1} a_{2j}\omega_{N/2}^{ij}\gamma_N^j + \omega_N^i\gamma_{2N} \sum_{j=0}^{N/2-1} a_{2j+1}\omega_{N/2}^{ij}\gamma_N^j \bmod q, \quad i = 0, 1, ..., N-1. \tag{7}$$

Then Equations 7 are grouped into two parts according to the size of index i. Applying the symmetry property and periodicity property of twiddle factors, we obtain:

$$\begin{aligned} \hat{a}_i &= \sum_{j=0}^{N/2-1} a_{2j}\omega_{N/2}^{ij}\gamma_N^j + \omega_N^i\gamma_{2N} \sum_{j=0}^{N/2-1} a_{2j+1}\omega_{N/2}^{ij}\gamma_N^j \bmod q \\ \hat{a}_{i+N/2} &= \sum_{j=0}^{N/2-1} a_{2j}\omega_{N/2}^{ij}\gamma_N^j - \omega_N^i\gamma_{2N} \sum_{j=0}^{N/2-1} a_{2j+1}\omega_{N/2}^{ij}\gamma_N^j \bmod q \\ & i = 0, 1, ..., N/2-1. \end{aligned} \tag{8}$$

Let $\hat{a}_i^{(0)} = \sum_{j=0}^{N/2-1} a_{2j}\omega_{N/2}^{ij}\gamma_N^j \bmod q, \hat{a}_i^{(1)} = \sum_{j=0}^{N/2-1} a_{2j+1}\omega_{N/2}^{ij}\gamma_N^j \bmod q$. Equations 8 are expressed as follows:

$$\begin{aligned} \hat{a}_i &= \hat{a}_i^{(0)} + \omega_N^i\gamma_{2N}\hat{a}_i^{(1)} \bmod q \\ \hat{a}_{i+N/2} &= \hat{a}_i^{(0)} - \omega_N^i\gamma_{2N}\hat{a}_i^{(1)} \bmod q, \quad i = 0, 1, ..., N/2-1. \end{aligned} \tag{9}$$

It is easy to see that $\hat{a}_i^{(0)}$ and $\hat{a}_i^{(1)}$ are essentially the same as in Equation 5, except that they are scaled down to $N/2$ points from N points. In other words, $\hat{a}_i^{(0)}$ and $\hat{a}_i^{(1)}$ are $N/2$-point NTTs of a_{2j} and a_{2j+1}, respectively. In this way, N-point NTT can be resolved with two $N/2$-point NTTs. The same decimation process can be applied recursively to the computation of $\hat{a}_i^{(0)}$ and $\hat{a}_i^{(1)}$ until 2-point NTT. Taking 8-point NTT as an example, the dataflow is depicted in Figure1.

Because $\gamma_{2m}^2 \equiv \omega_m \pmod q, m = 2^1, 2^2, ..., N$, considering the scale property of twiddle factors, we have

$$\begin{aligned} \omega_m^j\gamma_{2m} &\equiv \gamma_{2m}^{2j+1} \\ &\equiv \gamma_{2N}^{(2j+1)N/m} \pmod q, \qquad m = 2^1, 2^2, ..., N, \quad j = 0, 1, ..., m/2-1. \end{aligned} \tag{10}$$

Algorithm 4 Low-Complexity NTT Algorithm without Pre-processing

Let the vectors $\boldsymbol{a}$ and $\boldsymbol{A}$ denote $(a_0, a_1, ...a_{N-1})$ and $(A_0, A_1, ...A_{N-1})$, respectively, where $a_i \in \mathbb{Z}_q, A_i \in \mathbb{Z}_q, i = 1, 2, ...N-1$. Let ω_N be a primitive N-th root of unity in $\mathbb{Z}_q$ and let $\gamma_{2N} = \sqrt{\omega_N}$.
Input: $\boldsymbol{a}, N, q; \gamma_{2N}^i, i = 0, 1, ..., N-1$.
Output: $\boldsymbol{A} = NTT(\boldsymbol{a})$
$\boldsymbol{A} \leftarrow scramble(\boldsymbol{a})$
for $s = 1$ to $\log_2 N$ **do**
  $m \leftarrow 2^s$
  for $j = 0$ to $m/2 - 1$ **do**
    $\omega = \gamma_{2N}^{(2j+1)N/m}$
    for $k = 0$ to $N/m - 1$ **do**
      $u = A_{km+j}$
      $t = \omega \cdot A_{km+j+m/2} \bmod q$
      $A_{km+j} = (u + t) \bmod q$
      $A_{km+j+m/2} = (u - t) \bmod q$
    end for
  end for
end for
return $\boldsymbol{A}$

There are N different values in Equation 10. As a result, the $N/2$ powers of ω_N need not be stored. Only the N powers of γ_{2N} need to be pre-computed and stored. The details of the low-complexity DIT NTT are shown in Algorithm 4.

2.2 Low-Complexity INTT

When straightforwardly calculating the INTT, the main FFT requires $(N/2)\log_2 N$ modular multiplication and the post-processing requires $2N$ modular multiplications. The post-processing requires a considerable number of modular multiplications for N no more than 1024. The ratio of the number of modular multiplications between the post-processing and the main FFT can be up to 44.4% and 40% for N being 512 and 1024, respectively. [POG15] reduced the complexity of INTT by merging the scaling of γ_{2N}^{-i} into the Gentleman-Sande DIF FFT algorithm with the twiddle factors pre-computed and stored in memories. Based on this method, we further merge the scaling of N^{-1} into the DIF FFT. Thus, all the modular multiplications in post-processing are eliminated. This is achieved by changing the value of the pre-computed twiddle factors of INTT and slightly modifying the butterfly unit of the DIF FFT. This method does not require to increase the number of modular multiplications of the main FFT algorithm or storage space of pre-computed twiddle factors.

Similar to the derivation of the low-complexity NTT, the derivation of the low-complexity INTT is inspired by the strategy of another kind of FFT, i.e., the Gentleman-Sande FFT [GS66]. We follow the divide-and-conquer method of FFT that divides in the frequency domain. At the beginning, the post-processing and the INTT are written together as follows:

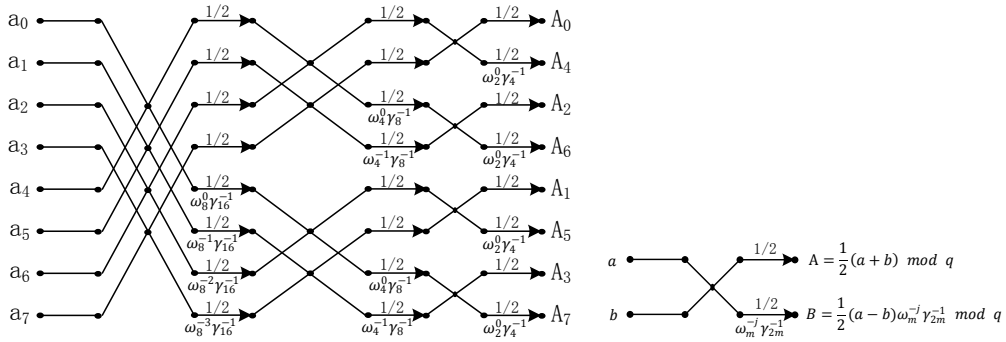
$$a_i = N^{-1}\gamma_{2N}^{-i} \sum_{j=0}^{N-1} \hat{a}_j \omega_N^{-ij} \bmod q, \tag{11}$$

where $i = 0, 1, ..., N-1$.

By splitting the items in the summation into two parts according to the size of the index of $\hat{a}$, Equation 11 can be rewritten as follows:

$$a_i = N^{-1}\gamma_{2N}^{-i} \left(\sum_{j=0}^{N/2-1} \hat{a}_j \omega_N^{-ij} + \sum_{j=N/2}^{N-1} \hat{a}_j \omega_N^{-ij} \right) \bmod q, \quad i = 0, 1, ..., N-1. \tag{12}$$

Because of the symmetry property and periodicity property of twiddle factors, the index



(a) Dataflow (b) Butterfly

Figure 2: Dataflow of 8-point Low-Complexity INTT

of the second summation in Equation 12 can be changed from $[N/2, N-1]$ to $[0, N/2-1]$, as follows:

$$a_i = N^{-1}\gamma_{2N}^{-i}\left[\sum_{j=0}^{N/2-1}\hat{a}_j\omega_N^{-ij} + \sum_{j=0}^{N/2-1}\hat{a}_{(j+N/2)}\omega_N^{-i(j+N/2)}\right] \bmod q, \quad i = 0, 1, ..., N-1. \tag{13}$$

In the following, Equations 13 are grouped into two parts according to the parity of i:

$$\begin{aligned} a_{2i} &= N^{-1}\gamma_{2N}^{-2i}\left[\sum_{j=0}^{N/2-1}\hat{a}_j\omega_N^{-2ij} + (-1)^{2i}\sum_{j=0}^{N/2-1}\hat{a}_{(j+N/2)}\omega_N^{-2ij}\right] \bmod q, \\ a_{2i+1} &= N^{-1}\gamma_{2N}^{-(2i+1)}\left[\sum_{j=0}^{N/2-1}\hat{a}_j\omega_N^{-(2i+1)j} + (-1)^{(2i+1)}\sum_{j=0}^{N/2-1}\hat{a}_{(j+N/2)}\omega_N^{-(2i+1)j}\right] \bmod q, \\ & i = 0, 1, ..., N/2-1. \end{aligned} \tag{14}$$

With the scale property of twiddle factors, Equation 14 can be simplified as

$$\begin{aligned} a_{2i} &= (\frac{N}{2})^{-1}\gamma_N^{-i}\sum_{j=0}^{N/2-1}\left[\frac{\hat{a}_j + \hat{a}_{(j+N/2)}}{2}\right]\omega_{N/2}^{-ij} \bmod q, \\ a_{2i+1} &= (\frac{N}{2})^{-1}\gamma_N^{-i}\sum_{j=0}^{N/2-1}\left\{\left[\frac{\hat{a}_j - \hat{a}_{(j+N/2)}}{2}\right]\omega_N^{-j}\gamma_{2N}^{-1}\right\}\omega_{N/2}^{-ij} \bmod q, \\ & i = 0, 1, ..., N/2-1. \end{aligned} \tag{15}$$

Let $\hat{b}_j^{(0)} = \frac{\hat{a}_j + \hat{a}_{(j+N/2)}}{2} \bmod q, \hat{b}_j^{(1)} = \left[\frac{\hat{a}_j - \hat{a}_{(j+N/2)}}{2}\right]\omega_N^{-j}\gamma_{2N}^{-1} \bmod q$, we have

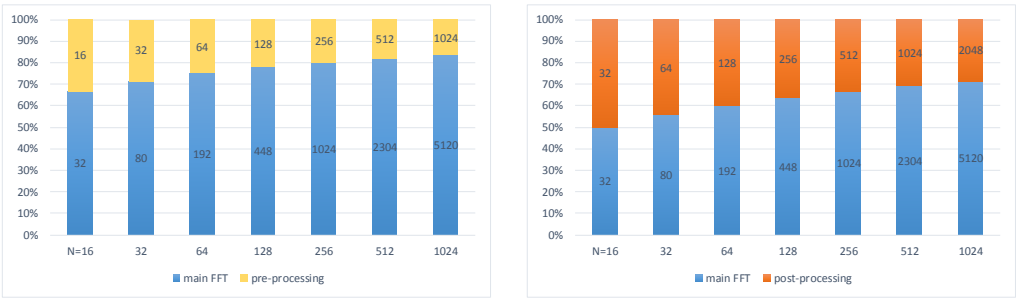
$$\begin{aligned} a_{2i} &= (\frac{N}{2})^{-1}\gamma_N^{-i}\sum_{j=0}^{N/2-1}\hat{b}_j^{(0)}\omega_{N/2}^{-ij} \bmod q, \\ a_{2i+1} &= (\frac{N}{2})^{-1}\gamma_N^{-i}\sum_{j=0}^{N/2-1}\hat{b}_j^{(1)}\omega_{N/2}^{-ij} \bmod q, \qquad i = 0, 1, ..., N/2-1. \end{aligned} \tag{16}$$

Algorithm 5 Low-Complexity INTT Algorithm without Post-processing

Let the vectors $\boldsymbol{a}$ and $\boldsymbol{A}$ denote $(a_0, a_1, ...a_{N-1})$ and $(A_0, A_1, ...A_{N-1})$, respectively, where $a_i \in \mathbb{Z}_q, A_i \in \mathbb{Z}_q, i = 1, 2, ...N-1$. Let ω_N be a primitive N-th root of unity in $\mathbb{Z}_q$ and $\gamma_{2N} = \sqrt{\omega_N}$.

Input: $\boldsymbol{a}, N, q; \gamma_{2N}^{-i}$, where $0, 1, ..., N-1$.
Output: $\boldsymbol{A} = INTT(\boldsymbol{a})$

for $s = \log_2 N$ to 1 **do**
  $m \leftarrow 2^s$
  for $j = 0$ to $m/2 - 1$ **do**
    $\omega = \gamma_{2N}^{-(2j+1)N/m}$
    for $k = 0$ to $N/m - 1$ **do**
      $u = A_{km+j}$
      $t = A_{km+j+m/2} \bmod q$
      $A_{km+j} = \frac{u+t}{2} \bmod q$
      $A_{km+j+m/2} = \frac{u-t}{2} \cdot \omega \bmod q$
    end for
  end for
end for
$\boldsymbol{A} \leftarrow scramble(\boldsymbol{a})$
return $\boldsymbol{A}$



(a) NTT (b) INTT

Figure 3: Number of modular multiplications of NTT and INTT.

Recalling the definition of N-point INTT as Equation 11, we can easily see that Equation 16 is similar to Equation 11, except that it is scaled down to $N/2$ points from N points. In other words, a_{2i} and a_{2i+1} correspond to $N/2$-point INTT of $\hat{b}_j^{(0)}$ and $\hat{b}_j^{(1)}$, respectively.

In this way, N-point INTT can be resolved with two $N/2$-point INTTs. The same decimation process can be applied recursively to the computation of a_{2i} and a_{2i+1} until 2-point INTT if N is a power of two. Taking 8-point INTT as an example, the dataflow is depicted in Figure 2.

Because $\gamma_{2m}^{-2} \equiv \omega_m^{-1} \pmod q, m = 2^1, 2^2, ..., N$, considering the scale property of twiddle factors, we have

$$\begin{aligned} \omega_m^{-j}\gamma_{2m}^{-1} &\equiv \gamma_{2m}^{-(2j+1)} \\ &\equiv \gamma_{2N}^{-(2j+1)N/m} \pmod q, \qquad m = 2^1, 2^2, ..., N, \quad j = 0, 1, ..., m/2-1. \end{aligned} \tag{17}$$

There are N different values in Equation 17. As a result, we do not need to pre-compute and store the $N/2$ powers of ω_N^{-1}. Only the N powers of γ_{2N}^{-1} are pre-computed and stored. The details of the proposed DIF INTT are shown in Algorithm 5.

The low-complexity NTT does not require pre-processing with N modular multiplications. As a result, an N-point low-complexity NTT requires only $(N/2)\log_2 N$ modular multiplications instead of $(N/2)\log_2 N + N$ modular multiplications for NTT implemented by pre-processing followed by FFT. Similarly, our proposed low-complexity INTT does not require post-processing with $2N$ modular multiplications. As a result, an

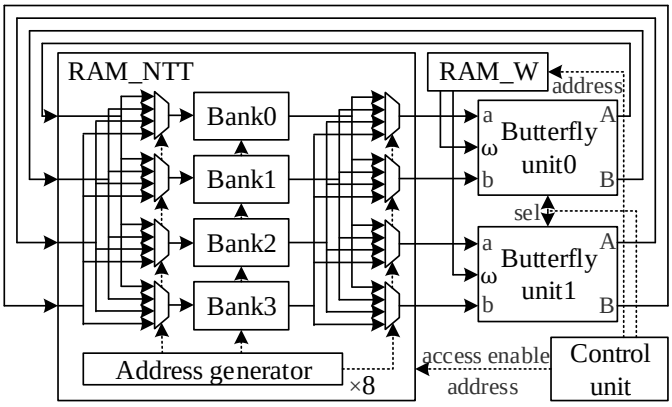


Figure 4: The architecture of NTT/INTT.

N-point low-complexity INTT requires only $(N/2)\log_2 N$ modular multiplications instead of $(N/2)\log_2 N + 2N$ modular multiplications for INTT implemented by FFT followed by post-processing. This means that our proposed low-complexity NTT and INTT have the same number of modular multiplications as the FFT algorithm. The decreased number of multiplications depends on the point of NTT/INTT, i.e., N. The smaller the value of N, the higher the proportion of the decrement, as depicted in Figure 3. The decreased modular multiplication by the low-complexity INTT can be up to 50% for 16-point INTT. For $N = 512$ in NewHope, the low-complexity NTT and INTT can reduce 18.2% and 30.8% of modular multiplications, respectively. For $N = 1024$ in NewHope, the low-complexity NTT and INTT can reduce 16.7% and 28.6% of modular multiplications, respectively. [DB16] reduced the number of modular multiplications of post-processing from $2N$ to N by combining the multiplications of N^{-1} and γ^{-i}, but this approach increased the memory for pre-computed twiddle factors of INTT from N to $3N/2$, whereas our method reduces more modular multiplication and does not increase memory for pre-computed twiddle factors. The method of [FS19] hides the clock cycles consumed by the post-processing of INTT, but it does not eliminate the computations, and the cost is two additional modular multipliers. [POG15] eliminated the scaling of γ_{2N}^{-i} of the post-processing, but it does not eliminate the scaling of N^{-1}, whereas our method eliminates both of them.

The low-complexity NTT/INTT can be used in polynomial multiplication over the ring $\mathbb{Z}_q[x]/\langle x^N + 1\rangle$, where q is a prime and satisfies that $q \equiv 1 \pmod{2N}$ and N is a power of two. These conditions are the same as those required by NWC. The low-complexity NTT/INTT does not require any additional conditions. As a result, this method is generally applicable to NWC, which is widely used in RLWE-based cryptographic algorithms.

3 Architecture of the Low-Complexity NTT/INTT

3.1 The Overall Architecture of NTT/INTT

The overall architecture for the low-complexity NTT/INTT is designed and presented in Figure 4. Because two different algorithms 4 and 5 are used for DIT NTT and DIF INTT, the architecture is designed to support both DIT NTT and DIF INTT. The proposed architecture consists of a coefficient memory *RAM_NTT*, a twiddle factor memory *RAM_W*, two BFUs, and a control unit. Two BFUs are used in the hardware architecture to speed up the throughput. The BFUs perform in pipeline mode, and each BFU can read and write two data points every clock cycle when the pipeline is fulfilled. Their architecture supports two types of butterflies, as shown in Figure 1(b) and

2(b). Their details are presented in Section 3.2. *RAM_NTT* is designed as a multi-bank memory to meet the bandwidth requirements of the two BFUs. The memory contains four banks, eight MUXs, and eight address generators. The address generators give out bank addresses to control the MUXs and give out new addresses and enable signals for the banks. The banks are implemented with dual-port block RAMs in FPGA devices, so they can provide eight ports for data access. The storage capacity of each bank of *RAM_NTT* is $(N/4)\lceil \log_2 q \rceil$. *RAM_W* is used to store precomputed twiddle factors for DIT NTT and DIF INTT. Its storage capacity is $2N\lceil \log_2 q \rceil$. The control unit generates addresses and enable signals for *RAM_NTT* and *RAM_W*.

An address conflict-free multi-bank memory access scheme for DIT NTT with two parallel radix-2 BFUs is presented in this paragraph. The partition approach of the coefficients into the four banks is derived from [WHEW14, LSW01]. Accordingly, the address generators are designed following Equation 18:

$$\begin{aligned} BankAddr &= \sum_{i=0}^{\lceil \frac{1}{2}\log_2 N \rceil - 1} addr[2i+1:2i] \bmod 4 \\ NewAddr &= addr >> 2, \end{aligned} \tag{18}$$

where *addr* denotes the raw address generated by the control unit; *BankAddr* denotes the bank address for selecting the banks; and *NewAddr* denotes the new address for the banks. However, the multi-bank scheme does not work well when $\log_2 N$ is odd, because accessing conflict exists in the last s-loop in Algorithms 4 for NTT with two BFUs. Taking $N = 8$ for example, the raw addresses {0,1,2,3,4,5,6,7} are mapped in the bank addresses {0,1,2,3,1,2,3,0}, according to the multi-bank scheme in [WHEW14, LSW01]. In the last s-loop in Algorithm 4, the four data points in raw addresses {0,4,1,5} are accessed simultaneously by the two BFUs, but the raw addresses {1,4} are both mapped in bank 1, which means accessing conflict. To resolve this problem, the last s-loop of Algorithm 4 is modified. It is interesting that the concurrently accessed four data points in the penultimate s-loop can be accessed by two butterflies in the last s-loop just with data crossed. Based on this observation, the execution order of the last s-loop is rearranged as follows:

$$\begin{aligned} &\textbf{for } j = 0 \textbf{ to } N/4 - 1 \\ &\quad A_j \leftarrow A_j + \gamma_{2N}^{2j+1} A_{j+N/2} \\ &\quad A_{j+N/2} \leftarrow A_j - \gamma_{2N}^{2j+1} A_{j+N/2} \\ &\quad A_{j+N/4} \leftarrow A_{j+N/4} + \gamma_{2N}^{2j+N/2+1} A_{j+3N/4} \\ &\quad A_{j+3N/4} \leftarrow A_{j+N/4} - \gamma_{2N}^{2j+N/2+1} A_{j+3N/4} \end{aligned} \tag{19}$$

Thus, the four data points being accessed in parallel by two BFUs are always in four different banks. For DIF INTT, it can be seen that the dataflow topological structures of DIT NTT and DIF INTT are mirror-symmetric. As a result, the address conflict-free multi-bank memory access scheme for DIT NTT is also applicable to DIF INTT, as long as the first s-loop of Algorithm 5 is rearranged according to the execution order of the second s-loop, just like the schedule of the last s-loop of Algorithm 4.

Based on the algorithms of the low-complexity DIT NTT and low-complexity DIF INTT, the proposed architecture can complete NTT or INTT in approximately $(N/4)\log_2 N$ clock cycles. Pre-processing of NTT and post-processing of INTT are all merged into the main process of the low-complexity DIT NTT and DIF INTT, so they do not consume any clock cycles. The scramble function requires $N/4$ cycles to reorder the data before or after the transform, benefiting from the multi-bank memory scheme, whereas normal single-bank memory requires N cycles for the scramble function. If four BFUs are used,

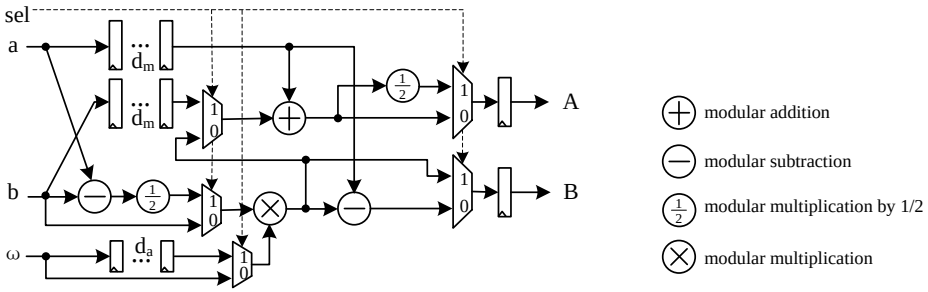


Figure 5: The architecture of butterfly unit.

instead of two BFUs, the required clock cycles can be further reduced, but the memory banks will be doubled and the routing will be complicated.

3.2 Compact Butterfly Unit

Because the DIT and DIF decimation methods are used for NTT and INTT, respectively, two different butterfly structures are required, as shown in Figure 1(b) and 2(b). If two different BFUs are responsible for NTT and INTT, respectively, doubled hardware resources are required compared with the case of a single type of butterfly structure.

In this section, a compact butterfly architecture that supports both DIT NTT and DIF INTT is proposed. The proposed butterfly architecture is depicted in Figure 5. Our BFUs also support modular multiplication, modular addition, and modular subtraction for polynomial operations with little additional control logic, which is not shown in Figure 5 for brevity. The signal *sel* controls all the MUXs and determines the function of the BFU. When *sel* equals zero, the BFU performs the DIT NTT butterfly; that is, $A = a + b\omega \bmod q, B = a - b\omega \bmod q$. When *sel* equals one, the BFU performs the DIF INTT butterfly; that is, $A = (a + b)/2 \bmod q, B = (a - b)\omega/2 \bmod q$.

The modular multiplication by 1/2, i.e., $x/2 \bmod q$, does not need any multiplications. For a given odd prime q, $1/2 \bmod q$ equals $(q + 1)/2$. When x is even, $x/2 \bmod q$ equals $(x \gg 1)$. When x is odd, $x/2$ can be express as

$$\frac{x}{2} \equiv (2\lfloor\frac{x}{2}\rfloor + 1)\frac{q+1}{2} \equiv \lfloor\frac{x}{2}\rfloor(q+1) + \frac{q+1}{2} \equiv \lfloor\frac{x}{2}\rfloor + \frac{q+1}{2} \pmod q, \tag{20}$$

where $\lfloor\frac{x}{2}\rfloor$ equals $(x \gg 1)$ and $(q + 1)/2$ is a constant. Thus, the result of $x/2 \bmod q$ is selected from $(x \gg 1)$ or $(x \gg 1) + (q + 1)/2$, which is achieved by a shifter, an adder, and a MUX.

There is only one modular multiplier in our butterfly architecture. Due to the different execution orders of modular multiplication in DIT and DIF butterflies, some registers are required to balance the pipeline latency for the correctness of timing. Assume that the modular multiplier has d_m pipeline stages; d_m registers are added to the input data a and b to balance the pipeline latency. If the modular adder and subtractor have d_a pipeline stages, d_a registers should be added to the path of ω to balance the pipeline latency. There are $(2d_m + d_a + 2)\lceil\log_2 q\rceil$ registers in our BFU architecture, excluding the registers in the modular multiplier, adder and subtractor.

The occupied hardware resources by our compact BFU supporting both DIT and DIF are much less than that of two independent DIT BFU and DIF BFU, which means a doubled number of modular multipliers and modular adders. [BPC19, BUC19a] implemented a unified butterfly architecture that supports DIT and DIF. It requires only one modular multiplier, but it requires two modular adders and two modular subtractors. Its architecture requires one more modular adder compared with our compact BFU.

3.3 Modular Reduction

In this section, an efficient and constant time modular reduction method for the modulus $q = 12289$ of NewHope-NIST and its hardware architecture are proposed. For the convenience of description, $x[msb : lsb]$ is used to represent a $(msb - lsb + 1)$-bit data according to the grammar specification of the Verilog hardware description language. The bits of $x[msb : lsb]$ is taken from the lsb_{th} bit to the msb_{th} bit of the data x. Let z be the multiplication result of a and b, i.e. $z = a \cdot b$, where $0 \leq a < q, 0 \leq b < q$. Thus the largest value of z is 28'h9000000, which has a bit length of 28. To obtain $z \bmod q$ efficiently, the feature $2^{14} \equiv 2^{12} - 1 \pmod{12289}$ is used recursively. Thus, z is expressed as follows:

$$\begin{aligned}
z &\equiv 2^{14} z[27:14] + z[13:0] \\
&\equiv 2^{12} z[27:14] - z[27:14] + z[13:0] \\
&\equiv 2^{14} z[27:16] + 2^{12} z[15:14] - z[27:14] + z[13:0] \\
&... \\
&\equiv 2^{12}(z[27:26] + z[25:24] + z[23:22] + z[21:20] + z[19:18] + z[17:16] + z[15:14]) \\
&\quad - (z[27:26] + z[27:24] + z[27:22] + z[27:20] + z[27:18] + z[27:16] + z[27:14]) \\
&\quad + z[13:0] \\
&\equiv 2^{12} c - d + z[13:0] \pmod{q},
\end{aligned} \tag{21}$$

where $c = z[27:26] + z[25:24] + z[23:22] + z[21:20] + z[19:18] + z[17:16] + z[15:14]$, and $d = z[27:26] + z[27:24] + z[27:22] + z[27:20] + z[27:18] + z[27:16] + z[27:14]$. Because the largest value of z is 28'h9000000, the largest value of $z[27:26] + z[25:24]$ is four in the case of $z[27:24]$ =4'h7. It is not difficult to see that the value of c is less than or equal to 19, which is in the case $z[27:14]$ =14'h1FFF, so c can be expressed with 5 bits. The largest value of d can be obtained from the summation of chopped high bits of 28'h9000000; i.e., $\max(d) = 2 + 9 + (9 \ll 2) + (9 \ll 4) + (9 \ll 6) + (9 \ll 8) + (9 \ll 10)$ =14'h2FFF.

Then, $2^{12}c$ is also simplified with the special property of modulus q as

$$\begin{aligned}
2^{12} c &\equiv 2^{14} c[4:2] + 2^{12} c[1:0] \\
&\equiv 2^{12} c[4:2] + 2^{12} c[1:0] - c[4:2] \\
&\equiv 2^{14} c[4] + 2^{12}(c[3:2] + c[1:0]) - c[4:2] \\
&\equiv 2^{12}(c[4] + c[3:2] + c[1:0]) - (c[4] + c[4:2]) \\
&\equiv 2^{12} e - (c[4] + c[4:2]) \pmod{q},
\end{aligned} \tag{22}$$

where $e = c[4] + c[3:2] + c[1:0]$. Because c is not greater than 19, it is easy to see that the maximum value of e is 6, which is 3 bits in length. Then, Equation 22 can be further simplified as

$$\begin{aligned}
2^{12} c &\equiv 2^{14} e[2] + 2^{12} e[1:0] - (c[4] + c[4:2]) \\
&\equiv 2^{12}(e[2] + e[1:0]) - (e[2] + c[4] + c[4:2]) \\
&\equiv f \pmod{q},
\end{aligned} \tag{23}$$

where $f = 2^{12}(e[2] + e[1:0]) - (e[2] + c[4] + c[4:2])$. Because e is not greater than 6, the maximum value of $e[2] + e[1:0]$ is 3. Thus, f must be less than q. It is easy to see that $2^{12}(e[2] + e[1:0]) \geq e[2] + c[4] + c[4:2]$. As a result, $2^{12}c \bmod q = f$.

Based on the above analysis, the hardware architecture of modular multiplication is proposed as depicted in Figure 6. The modular multiplication is implemented with a 4-stage pipeline architecture. At the first pipeline stage, $z = a \cdot b$ is calculated, which is carried out with a single DSP unit in the target FPGA. At the second pipeline stage, d and $2^{12}c + z[13:0] \bmod q$ are calculated in parallel. At the third pipeline stage, a modular

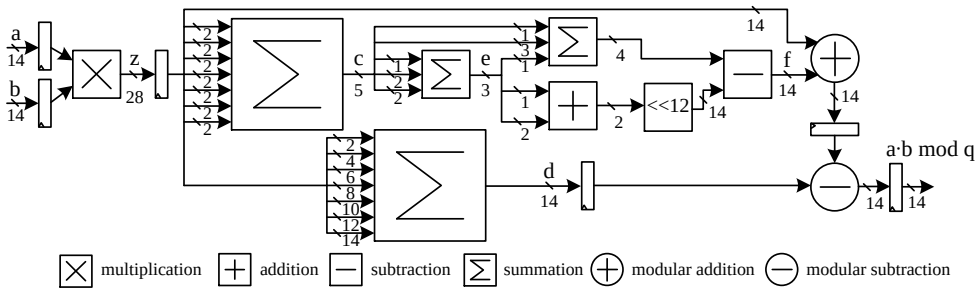


Figure 6: The architecture of modular multiplication.

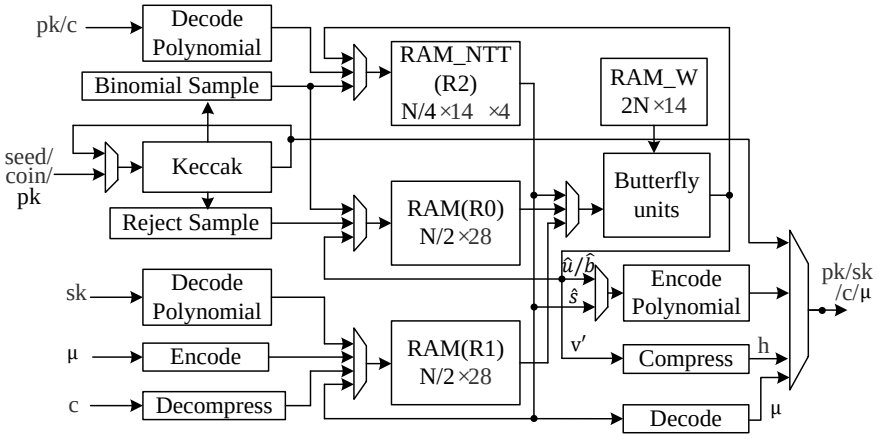


Figure 7: The architecture of NewHope-NIST.

subtraction is executed to obtain the result. At the last pipeline stage, the results are output. Among the timing paths, the path from z to d, i.e., the summation to get d, is the critical path in timing. Although the summation for c has 7 inputs as the summation for d, the widths of its inputs are all two, which is much smaller than that of the summation for d. As a result, the delay of the summation for c is smaller than that for the summation for d.

Barrett reduction and Montgomery reduction are the most commonly used algorithms to obtain modular reduction after multiplication. Both of these algorithms require two additional multiplications, which are expensive in time and hardware resources. For the specific modulus $q = 7681$, [LSSR+15] used a SAMS2 technique for modular reduction. [RV17] adapted the SAMS2 technique to the modulus $q = 12289$. This technique requires a multiplication and at most six additional 15-bit subtractions for the modulus $q = 12289$. The number of additional subtractions depends on the result of the first reduction in SAMS2, which could be problematic for a side-channel attack if the reduction is not dealt with carefully in constant time. Our proposed modular reduction does not require additional multiplications as Barrett and Montgomery algorithms. The specific operations of our reduction are not data-dependent as in [LSSR+15, RV17]. As a result, our reduction on the modulus $q = 12289$ is efficient and time-constant.

4 Architecture of NewHope-NIST

The hardware architecture is designed to support key generation, encryption, and decryption of NewHope-NIST, instead of a hardware corresponding to a function. Figure 7 shows the architecture of the hardware design. The blocks Binomial Sample and Reject Sample perform sampling in the functions Sample and GenA, respectively. The blocks Compress and Decompress perform ciphertext compression and decompression, respectively. The blocks Decode Polynomial and Encode Polynomial transform the data format between a byte array and coefficients of a polynomial. The blocks Encode and Decode perform message encoding and decoding, respectively. The block Keccak performs the functions of SHAKE256 and SHAKE128. We modified the open-source code [Ope] as [KLC+17] did. It takes 24 clock cycles to execute 24 rounds in the function KECCAK-f.

Doubled bandwidth matching carries through the architecture to reduce clock cycles. The blocks RAM_NTT(named R2), RAM_W and BFUs follow the architectures in Section 3. There are two additional RAMs, named R0 and R1, in the architecture to store data of the intermediate polynomial. As the two BFUs can deal with two point-wise operations, R0 and R1 are arranged with two coefficients in an address to match the doubled bandwidth with the BFUs. Most other blocks, such as Binomial Sample, Reject Sample, Compress, Decompress, Decode Polynomial, Encode Polynomial and Encode, are also designed to be able to process two data points every cycle, to match the doubled bandwidth of the memories. Message decoding requires four coefficients to generate one bit of plain text when $N = 1024$. Decode is designed to take in four data at most at one cycle, and the pending data for Decode are arranged in the multi-bank memory R2. As a result, this method can output one bit every cycle for both $N = 512$ and $N = 1024$.

Timing hiding is achieved by simultaneously performing operations without resource conflict and data dependency to further decrease clock cycles. Resource conflict means that two operations use the same processing unit. Data dependency means that an operation uses data from the results of another operation. The temporal and spatial detail of our implementation with the above architecture is described with pseudo-code. The pseudo-code of key generation, encryption and decryption are shown in Algorithm 6, 7, and 8, respectively. The operations in the same state of the pseudo-code are performed simultaneously. In our architecture, a specific RAM may be read and write by operations in the same line, such as $R2$ at State 4 in Algorithm 6, $R2$ at State 3 and State 5 in Algorithm 7. The reason is that the operations sequentially access the RAM; thus, the operation that writes the RAM can be executed as soon as the data in the corresponding address are read out by another operation. As a result, although data dependencies exist, the operations can be performed simultaneously at the operation level. The polynomials are stored in a bit-reversed order after the INTT at State 7 of Algorithm 7. As a result, the polynomial is accessed in a bit-reversed order at State 8, which requires the access address to be bit-reversed and does not require additional clock cycles.

The architecture of NewHope-NIST is designed to perform in constant time. The required cycles of the functions in NewHope-NIST are discussed below without considering a small number of stalls caused by pipeline setup. NTT and INTT always access coefficients

Algorithm 6 Pseudo-code for implementation of NewHope-CPA-PKE Key Generation

Input: $seed$.
Output: pk, sk.

1: $(noiseseed, publicseed) \leftarrow SHAKE256(64, seed)$; output $publicseed$
2: $R_2 \leftarrow Sample(noiseseed, 0)$
3: $R_2 \leftarrow NTT(R_2)$; $R_0 \leftarrow GenA(publicseed)$
4: output $EncodePolynomial(R_2)$; $R_0 \leftarrow R_0 \circ R_2$; $R_2 \leftarrow Sample(noiseseed, 1)$
5: $R_2 \leftarrow NTT(R_2)$
6: output $EncodePolynomial(R_0 + R_2)$

Algorithm 7 Pseudo-code for implementation of NewHope-CPA-PKE Encryption

Input: $pk, \mu, coin$.
Output: $EncodePolynomial(\hat{u}), h$.

1: $R_2 \leftarrow Sample(coin, 0)$
2: $R_2 \leftarrow NTT(R_2); R_0 \leftarrow GenA(pk[0:31])$
3: $R_0 \leftarrow R_0 \circ R_2; R_1 \leftarrow R_2; R_2 \leftarrow Sample(coin, 1)$
4: $R_2 \leftarrow NTT(R_2)$
5: output $EncodePolynomial(R_0 + R_2)$; $R_2 \leftarrow DecodePolynomial(pk[32 : 7n/4 + 31])$
6: $R_2 \leftarrow R_2 \circ R_1; R_0 \leftarrow Sample(coin, 2)$
7: $R_2 \leftarrow INTT(R_2)$
8: $R_0 \leftarrow PolyBitRev(R_2) + R_0; R_1 \leftarrow Encode(\mu)$
9: output $Compress(R_0 + R_1)$

Algorithm 8 Pseudo-code for implementation of NewHope-CPA-PKE Decryption

Input: c, sk.
Output: μ.

1: $R_2 \leftarrow DecodePolynomial(c[0 : 7n/4 - 1]); R_1 \leftarrow DecodePolynomial(sk)$
2: $R_2 \leftarrow R_2 \circ R_1$
3: $R_2 \leftarrow INTT(R_2); R_1 \leftarrow Decompress(c[7n/4 : 17n/8 - 1])$
4: $R_2 \leftarrow R_1 - R_2$
5: output $Decode(R_2)$

from the multi-bank memory R2 and require $(N/4)\log_2 N$ cycles. The function Encode requires $N/2$ cycles, while Decode requires 256 cycles, benefiting from taking four data points from R2. All the point-wise multiplication, addition and subtraction operations of the polynomial require $N/2$ cycles. The function Sample requires $N/64 \times (26 + 64/2)$ cycles, where 26 cycles are for SHAKE256 and 64/2 cycles are for accessing memory. Both the functions Decode Polynomial and Encode Polynomial require $max(14N/bandwidth, N/2)$ cycles, where the bandwidth is 16 in our design. The required cycles for $GenA$ are variable because of rejection sampling. $GenA$ is related only to the public key, so it does not require protection against timing attacks. Despite this, we implement $GenA$ in constant time for the entire design to be time-constant in form. In our design, the operation $GenA$ is scheduled to execute simultaneously with NTT as State 3 in Algorithm 6 and State 2 in Algorithm 7. In case $GenA$ takes a longer time than NTT, the algorithm is simply restarted. Our design requires one cycle for $SHAKE128Absorb$, 25 cycles for $SHAKE128Sequeeze$, and 86 cycles for the rejection sample. When the total number of $SHAKE128Sequeeze$ required in $GenA$ is less than 22 for $N = 1024$ or 10 for $N = 512$, $GenA$ requires fewer cycles than NTT. The probability that $GenA$ takes longer time is approximately 2.3E-37, 8.6E-13 for $N = 1024$ and 512, respectively, which is so low that performance loss is negligible.

5 Results and Comparison

The low-complexity NTT/INTT and NewHope-NIST are designed with Verilog HDL and verified on a 28 nm Xilinx Artix-7 FPGA (XC7Z020CLG484-3), which is recommended by NIST and widely adopted in the state-of-the-art evaluations. The hardware resources and highest frequency are obtained from Vivado 2019.1.1 with the default strategy for synthesis and implementation. In this section, the implementations results of both NTT/INTT and NewHope-NIST are discussed and compared with the state-of-the-art.

5.1 Implementation Results of NTT/INTT

Table 1 lists the key results for the realization of the low-complexity NTT/INTT of 1024 points and 512 points. Other state-of-the-art NTT designs with the same modular

Table 1: Implementation Results of NTT/INTT on FPGA and Comparison.

Design	size	NTT/INTT Cycles	Freq. (MHz)	Time (μs)	LUT/ ATP	FF/ ATP	DSP/ ATP	BRAM/ ATP	Device
This Work	1024	2569/2569	244	10.5	847/8.9	375/3.9	2/21.1	6/63.2	XC7Z020
[FS19]		10240/10240	-	-	980/41.1	395/16.6	26/1091	2/83.9	XC7Z020
[KLC+17]		1592/1592[a]	150	10.6	2832/30.1	1381/14.7	8/84.9	10/106.1	XC7Z020
[JGCS19]		6206/6206	251	24.7	343/8.5	493/12.2	3/74.2	6/148.4	XC7Z020
[FSM+19]		24609/24609	-	-	886/89.4	618/62.3	26/2622	1/100.9	XC7Z020
[BUC19b]		6155/6155	-	-	7690/194	16/0.4	11/277.5	13/327.9	XC7A200T
This Work	512	1289	245	5.3	741/3.9	330/1.7	2/10.5	5/26.3	XC7Z020
[FS19]		4608/4608	-	-	980/18.4	395/7.4	26/489	2/37.6	XC7Z020
[RVM+14]		3443/4775	278	14.8	994[b]/14.7	944[b]/14	1/14.8	3/44.3	V6LX75T
[BUC19b]		2826/2826	-	-	7690/88.7	16/0.2	11/126.9	13/150	XC7A200T

[a] : The cycles for order reverse are not included.
[b] : The hardware resources occupied by the TRNG and Gaussian sampler have been removed.

$q = 12289$ for lattice-based PQC algorithms are also listed in the table for comparison. As the FPGA has various types of hardware resources, the ATPs are measured by multiplying time by the number of LUTs, FFs, DSPs, and BRAMs, respectively, for a comprehensive comparison. When computing ATPs, the works that do not show frequency are supposed to run at the same frequency as our design; and the cycle value is set as the average of NTT and INTT. It can be seen from Table 1 that our NTT is the fastest and has the smallest ATP compared with the state-of-art NTT designs with the same point numbers except the ATP measured by FF compared with [BUC19b], which has much larger ATPs measured with LUT, DSP, and BRAM. Note that the clock cycles in Table 1 do not include the cycles for the scramble function. If these cycles are considered, our design is more advantageous, because our design requires only $N/4$ cycles for the scramble function, while most other designs generally require N cycles.

[FS19] optimized the last round of INTT and avoided the cycles required by post-processing. However, this approach does not reduce the computational complexity that is represented by the number of modular multiplications. That technique requires $6N$ Montgomery modular multiplications for the optimized last round of INTT, whereas our low-complexity INTT requires only $N/2$ cycles for the last round and avoids the post-processing of INTT. The price of avoiding post-processing is two additional multipliers, which are used only in the last round of INTT. As a result, [FS19] requires up to 26 DSPs, whereas our design requires only 2 DSPs. If the design in [FS19] runs at the same frequency as ours, its ATPs measured by LUT, FF, DSP, and BRAM are 4.0×, 3.7×, 45.5× and 1.2× that of our design for $N = 1024$, respectively. The corresponding speedups of ATPs for $N = 512$ are 4.2×, 3.8×, 41.0× and 1.3×. [FSM+19] designed a hardware NTT accelerator for a PQC processor. Compared with our design, it consumes more registers and DSPs and almost the same amount of LUTs and less BRAMs, while it requires almost 10× clock cycles.

[KLC+17] proposed an NTT-based high-performance hardware architecture for NewHope-USENIX. It uses four BFUs to reduce clock cycles of NTT and uses a variant of Montgomery reduction to optimize reduction. Its number of BFUs is two times that of ours, whereas the occupied DSPs are four times that of ours. This is because Montgomery reduction requires additional multipliers, but our proposed reduction does not. Its clock cycles are lower than those of our design, benefiting from doubled BFUs, but the occupied hardware resources are all greater than our design. Considering the two factors, its ATPs measured by LUT, FF, DSP, and BRAM are 3.4×, 3.7×, 4.0× and 1.7× that of ours, respectively.

[JGCS19] proposed a fast and configurable NTT module, but it did not eliminate

either the pre-processing of NTT or the post-processing of INTT. It requires many more clock cycles than our design. The ATP measured by LUT is similar. However, its ATPs measured by FF, DSP, and BRAM are 3.1×, 3.5× and 2.3× that of our design, respectively.

[RVM+14] proposed a method to eliminate the pre-processing of NTT for the first time, but the method cannot be applied to eliminate the post-processing of INTT. It uses only one multiplier to compute butterfly and twiddle factors on the fly. As a result, it uses fewer DSPs and BRAMs but requires many more clock cycles. Its ATPs measured by LUT, FF, DSP and BRAM are 3.8×, 8.0×, 1.4× and 1.7× that of our design, respectively.

[BUC19a] utilized DIT and DIF to eliminate the scramble function of NTT and designed a unified butterfly architecture for the DIT and DIF. The results on FPGA are exhibited on an extended version [BUC19b]. The results show that it requires more than 2.1× cycles than our design. Much more LUTs, DSPs and BRAMs are also required. The possible reasons are the support of several moduli and single-port memory architecture that asks for double memory for NTT. It requires very few registers because of one cycle ALU, the corresponding result of which is that the frequency is as low as 72MHz on the TSMC 40nm process.

5.2 Implementation Results of NewHope-NIST

The key generation, encryption, and decryption of NewHope-NIST are implemented together on an FPGA for the parameters $N = 1024$ and $N = 512$, respectively. The key results of our implementation are presented in Table 2. Our NewHope-NIST design is the fastest and has the smallest ATP compared with the state-of-art NewHope-NIST designs. The encapsulation/decapsulation of NewHope-CPA-KEM in NewHope-NIST has only two/one more SHAKE256 than the encryption/decryption of NewHope-NIST, while each SHAKE256 requires only 26 cycles in our architecture. As a result, they are directly compared. [JGCS19] implemented NewHope-NIST on the same XC7Z020 FPGA device. It consumes 2.5×/2.9× time than our design for the key generation plus decryption/encryption. When implemented on a higher-end device XCZU4EG (16 nm vs 28 nm) at 2× frequency, it still consumes 1.2×/1.3× time. Because [JGCS19] supports three algorithms, it is reasonable that it occupies more hardware resources. [BSNK19] evaluated NewHope-NIST with high-level synthesis (HLS) on a Virtex-7 FPGA. It requires 163×/451× time, 160×/194× LUTs and 70×/77× registers for encapsulation/decapsulation, compared with encryption/decryption of our design. The main reasons for the huge difference are probably our low-complexity NTT/INTT, highly efficient architecture and the relatively inefficient HLS of [BSNK19]. [FSM+19] implemented NewHope-NIST using a RISC-V-based SoC with an NTT accelerator and a SHA accelerator. It requires 34 $\sim$ 47× cycles while consuming 1.7× LUTs, 1.2× Registers, 13× DSPs and 0.13× BRAMs for only the NTT and SHA accelerators. [BUC19b] presented a configurable processor for lattice-based PQC. Its NewHope-NIST consumes 4.8×/8.5×/11.7× and 4.5×/8.1×/11.6× cycles than our design for the key generation/decryption/encryption for the parameter N to be 1024 and 512, respectively. The occupied LUTs, DSPs, and BRAMs are more than our design, while the Registers are less. The reasons and consequences are discussed in Section 5.1. In summary, our design is at least 2.5× faster, and the ATPs are at least 4.9× smaller than other NewHope-NIST designs on similar devices.

The calculation of key generation/encryption/decryption of NewHope-NIST is similar to the calculation of Alice0/Bob/Alice1 of NewHope-USENIX and NewHope-Simple. Thereby, the related works are listed for comparison. NewHope-USENIX is designed for high performance in [KLC+17]. Alice0/Bob/Alice1 of [KLC+17] consumes 1.3×/0.9×/1.3× time compared with the key generation/encryption/decryption of our design. The performance of Alice1 is even slightly better than the decryption of our design. However, [KLC+17] consumes approximately 1.8 $\sim$ 4× hardware resources. Its ATPs measured by LUT, FF, DSP and BRAM are 3.6×/3.9×/2.4×, 3.0×/3.0×/2.0×, 5.2×/5.0×/3.5× and 2.3×/2.2×/1.5×

Table 2: Implementation Results of NewHope-NIST on FPGA and Comparison.

Scheme	Cycles (k)	Freq. (MHz)	Time (μs)	LUT	FF	DSP	BRAM	Device
parameters: N=1024, q=12289								
NewHope-NIST (this work)	Key/Enc/Dec: 8.0/12.5/4.8	200	40/62.5/24	6781	4127	2	8	XC7Z020
NewHope-NIST [JGCS19]	Key+Dec/Enc: 30.6/34.0	190	160/178	13244	8272	24	18	XC7Z020
NewHope-NIST [JGCS19]	Key+Dec/Enc: 30.6/34.0	406	75/83	13961	8149	25	18	XCZU4EG
NewHope-NIST [BUC19b]	Key/Enc/Dec: 38.0/106.7/56.1	-	-	14975	2539	11	14	XC7A200T
NewHope-NIST [BSNK19]	Encaps: 680.1 Decaps: 722	66.7 66.7	10196 10825	135689 164937	26257 28999	- -	- -	Virtex-7
NewHope-NIST [FSM+19][a]	Key/Encaps/Decaps: 357.1/589.3/167.6	-	-	11321	4843	26	1	-
NewHope-USENIX [KLC+17]	Alice0/Alice1: 6.9/2.8 Bob: 10.3	133 131	51.9/21.1 78.6	18756 20826	9412 9975	8 8	14 14	XC7Z020
NewHope-Simple [OG19]	Alice0/Alice1: 115.8/55.3 Bob: 179.3	125 117	918/443 1532.4	5142 4498	4452 4635	2 2	4 4	XC7A35T
parameters: N=512, q=12289								
NewHope-NIST (this work)	Key/Enc/Dec: 4.2/6.6/2.5	200	21/33/12.5	6780	4026	2	7	XC7Z020
NewHope-NIST [BUC19b]	Key/Enc/Dec: 18.7/53.5/29.1	-	-	14975	2539	11	14	XC7A200T
RLWE [RVM+14]	Enc/Dec: 13.3/5.8	278	47.9/21	1536	953	1	3	V6LX75T
RLWE [RV17][b]	Enc: 1.2 Dec: 0.64	231 232	5.2 2.8	31880 7272	31540 8641	28 24	226[c] 20[c]	EP4SGX
parameters: N=640, q=32768								
FrodoKEM [HOKG18]	Key: 3277 Encaps: 3318 Decaps: 3359	167 167 162	19622 19867 20733	6621 6745 7220	3511 3528 3549	1 1 1	6 11 16	XC7A35T
parameters: N=976, q=65536								
FrodoKEM [HOKG18]	Key: 7621 Encaps: 7683 Decaps: 7746	167 167 162	45632 46006 47811	7155 7209 7773	3528 3537 3559	1 1 1	8 16 24	XC7A35T

[a] : The hardware resources occupied by RISC core and Peripherals/Memory have been removed.
[b] : The data are from the cryptoprocessors *R2M-LBE/LBD* with radix-2 butterfly as our design.
[c] : This BRAM is 9K.

that of our design (Alice0/Bob/Alice1 vs key generation/encryption/decryption), respectively. NewHope-Simple is optimized for area in [OG19]. It occupies $0.5 \sim 1.1\times$ hardware resources, but requires $18 \sim 25\times$ time. The ATPs of [OG19] are approximately $9 \sim 28\times$ that of our design. Note that [KLC+17] and [OG19] realized the schemes on two independent hardware components, whereas the three functions are all realized on a single hardware component in our design. Therefore, the hardware resource advantage of our design is actually larger.

Some other implementations of lattice-based PQC algorithms are also listed in Table 2, although they cannot be directly compared with our design because of significant differences in algorithms and parameters. [RVM+14] and [RV17] implemented an RLWE encryption scheme introduced in [LPR10]. [RVM+14] requires more time but consumes much fewer hardware resources than our design. One important reason is that the Keccak module for SHA3 occupies 4333 LUTs and 2262 Registers, which account for 63.9% and 56.2% of

the total hardware resources in our design, whereas there is no Keccak in the design of [RVM+14]. [RV17] requires less time but consumes much more hardware resources than our design because it uses the multiple-path delay commutator (MDC) architecture for NTT, which achieves high throughput at the cost of a large number of hardware resources. Compared with the implementation of FrodoKEM [HOKG18], our design achieves more than 300× performance advantage while consuming a similar amount of hardware resources.

For RLWE-based PQC, several masking schemes have been proposed against differential power analysis (DPA). [RRdC+16, RSRVV15] proposed a masking method to split the secret polynomial into two random shares. [RdCR+16] inspired additive homomorphic masking by introducing encryption on a secret random message in the decryption algorithm. [BUC19a] discussed the application of the additive homomorphic masking in NewHope. [OSPG18] also used randomized sharing in their masked RLWE implementation. In the computing perspective, most of the masking schemes are on the NTT level or higher. The main overhead of these maskings is due to the additional NTTs introduced by the maskings. The low-complexity NTT/INTT can be used in these masking schemes if the ability to resist DPA requires strengthening. Note that the LUTs and Registers consumed by the NTT/INTT only occupy 12.5% and 9.1% of the entire NewHope-NIST. As a result, the cost of the maskings can be relatively small if additional NTT/INTT hardware is equipped for masking of NewHope-NIST.

6 Conclusion

This paper presents a highly efficient architecture of NewHope-NIST using low-complexity NTT/INTT. The implementation results show that the low-complexity NTT/INTT and the architecture of NewHope-NIST have a clear advantage in speed and ATP. Furthermore, the low-complexity NTT/INTT can benefit other NTT-inside algorithms, such as Crystals-Kyber[ABD+19], Crystals-Dilithium[DKL+19], qTesla[BAA+19]) and Falcon[FHK+19]. It will be interesting and essential to take further measures to resist side-channel analysis on our implementation. Constant execution time is realized, and the cost of countermeasure against DPA is discussed for our implementation. However, specific countermeasures against side-channel analysis, such as power analysis and electromagnetic analysis, and the protective effects are not covered in this work.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China (Grant No. 61672317) and in part by the National Key R&D Program of China (Grant No. 2018YFB2202101). We thank Min Zhu from Wuxi Micro Innovation Integrated Circuit Design Co.Ltd for his help in engineering. We also thank the editors and reviewers for their valuable comments.

References

[AAB+19] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. Newhope: algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2019), 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[AASA+19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the first round of the nist post-quantum cryptography standardization process. NISTIR 8240, January 2019. https://doi.org/10.6028/NIST.IR.8240.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2019), 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[ACZ18] Dorian Amiet, Andreas Curiger, and Paul Zbinden. Fpga-based accelerator for post-quantum signature scheme sphincs-256. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):18–39, Feb, 2018.

[ADPS16a] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange—a new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, August 2016. USENIX Association.

[ADPS16b] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Newhope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. https://eprint.iacr.org/2016/1157.

[APS13] A. Aysu, C. Patterson, and P. Schaumont. Low-cost and area-efficient fpga implementations of lattice-based cryptography. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 81–86, June 2013.

[BAA+19] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Krämer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. qtesla: algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2019), 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-Quantum Cryptography*. Springer-Verlag Berlin Heidelberg, 2009.

[BPC19] U. Banerjee, A. Pathak, and A. P. Chandrakasan. 2.3 an energy-efficient configurable lattice cryptography processor for the quantum-secure internet of things. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 46–48, Feb 2019.

[Bra16] M. Braithwaite. Experimenting with post-quantum cryptography. Google Security Blog, Jul 2016. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[BSNK19] Kanad Basu, Deepraj Soni, Mohammed Nabeel, and Ramesh Karri. Nist post-quantum cryptography- a hardware evaluation study. Cryptology ePrint Archive, Report 2019/047, 2019. https://eprint.iacr.org/2019/047.

[BUC19a] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 4(3):17–61, Aug, 2019.

[BUC19b] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols (extended version). Cryptology ePrint Archive, Report 2019/1140, 2019. https://eprint.iacr.org/2019/1140.

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

[DB16] C. Du and G. Bai. Towards efficient polynomial multiplication for lattice-based cryptography. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1178–1181, May 2016.

[DKL+19] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2019), 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[FHK+19] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (2019), 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[FS19] T. Fritzmann and J. Sepúlveda. Efficient and flexible low-power ntt for lattice-based cryptography. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 141–150, May 2019.

[FSM+19] T. Fritzmann, U. Sharif, D. Müller-Gritschneder, C. Reinbrecht, U. Schlichtmann, and J. Sepulveda. Towards reliable and secure post-quantum co-processors based on risc-v. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1148–1153, March 2019.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. ACM.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Guneysu. Standard lattice-based key encapsulation on embedded devices. *cryptographic hardware and embedded systems*, 2018:372–393, 2018.

[JGCS19] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. Spqcop: Side-channel protected post-quantum cryptoprocessor. Cryptology ePrint Archive, Report 2019/765, 2019. https://eprint.iacr.org/2019/765.

[KLC+17] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High performance post-quantum key exchange on fpgas. Cryptology ePrint Archive, Report 2017/690, 2017. https://eprint.iacr.org/2017/690.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LSSR+15] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 663–682, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[LSW01] Hsin Fu Lo, Ming Der Shieh, and Chien Ming Wu. Design of an efficient fft processor for dab system. In *IEEE International Symposium on Circuits and Systems*, IEEE International Symposium on Circuits and Systems, pages 654–657 vol. 4, 2001.

[Moo19] Dustin Moody. Round 2 of the nist pqc "competition" - what was nist thinking? PQCrypto 2019 in Chongqing, China, May 2019. https://csrc.nist.gov/Presentations/2019/Round-2-of-the-NIST-PQC-Competition-What-was-NIST.

[NIS15] NIST. Fips pub 202 – sha-3 standard: Permutation-based hash and extendable-output functions, 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[NIS16] NIST. Announcing request for nominations for public-key post-quantum cryptographic algorithms. 81 Federal Register 92787, December 2016. https://federalregister.gov/a/2016-30615.

[OG19] Tobias Oder and Tim Güneysu. Implementing the newhope-simple key exchange on low-cost fpgas. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 128–142, Cham, 2019. Springer International Publishing.

[Ope] OpenCores. Sha3(keccak). https://opencores.org/projects/sha3. [Online; accessed 09-November-2012, updated 11-October-2018].

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 1:142–174, Feb, 2018.

[PG12] Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, pages 139–158, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In Kristin Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 346–365, Cham, 2015. Springer International Publishing.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 513–533, Cham, 2017. Springer International Publishing.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-lwe masking. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography*, pages 233–244, Cham, 2016. Springer International Publishing.

[RRdC+16] Oscar Reparaz, Sujoy Sinha Roy, Ruan de Clercq, Frederik Vercauteren, and Ingrid Verbauwhede. Masking ring-lwe. *Journal of Cryptographic Engineering*, 6(2):139–153, Jun 2016.

[RSRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 683–702, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[RV17] C. P. Rentería-Mejía and J. Velasco-Medina. High-throughput ring-lwe cryptoprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2332–2345, Aug 2017.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 371–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[RVVV17] S. S. Roy, F. Vercauteren, J. Vliegen, and I. Verbauwhede. Hardware assisted fully homomorphic function evaluation and encrypted search. *IEEE Transactions on Computers*, 66(9):1562–1572, Sep. 2017.

[SD18] S. Streit and F. De Santis. Post-quantum key exchange on armv8-a: A new hope for neon made simple. *IEEE Transactions on Computers*, 67(11):1651–1662, Nov 2018.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994.

[WHEW14] W. Wang, X. Huang, N. Emmart, and C. Weems. Vlsi design of a large-number multiplier for fully homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):1879–1887, Sept 2014.

[Win96] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Springer-Verlag, Berlin, Heidelberg, 1996.