OPTIMISATION OF NUMBER THEORETIC TRANSFORM/INVERSE NUMBER THEORETIC TRANSFORM ARCHITECTURE FOR CRYSTALS-KYBER KEY ENCAPSULATION MECHANISM

BY

TAM KAI YUAN

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS)

COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2026

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Dr Lee Wai Kong and who has given me this bright opportunity to engage in a digital circuit design project. It is my first step to establish a research career on digital circuit design in the cryptography field. A million thanks to you.

I am also deeply thankful to the team at SkyeChip Berhad for fostering a sense of community that has continued to inspire my optimism moving forward.

Finally, I must say thanks to my parents and my family for their support throughout the course.

COPYRIGHTS STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Technology (Honours) Computer Engineering at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ABSTRACT

Post-quantum cryptography is an emerging trend in research and industry due to the emergence of quantum computing technology. Advanced technique like Shor's algorithm, when executed on a fault-tolerant quantum computer with sufficient logical qubits, can efficiently factor large integers and solve the discrete logarithm problem, thereby breaking widely deployed classical public-key cryptographic schemes such as RSA and elliptic-curve cryptography (ECC) currently used throughout industry. Malicious attackers can store the encrypted sensitive data now and try to decrypt it later, when the quantum computing technology becomes mature; this is known as ''harvest now, decrypt later" threat. In this letter, we present an efficient hardware architecture for accelerating Number Theoretic Transform (NTT) and Inverse Number Theoretic Transform (INTT), the key operations in lattice-based post-quantum key encapsulation mechanism, ML-KEM, which is also a standard released by US's National Institute of Standards and Technology. The key contributions are area-time efficient NTT architecture, a novel shared-DSP NTT multiplier, and a highly optimized DSP-less design with 35% better area-time product (ATP) compared to state-of-the-art. The proposed method also allows flexibility to choose between a more power efficient architecture using two DSPs, a more area-time efficient one that does not use any DSPs, and a balanced design with one shared-DSP.

Areas of Study: Cryptography, hardware architecture

Keywords: Post-quantum cryptography, ML-KEM, Kyber, number theoretic transform, FPGA.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

LIST OF ABBREVIATIONS

ATP	area-time product
BFU	butterfly unit
BRAM	Block RAM
CC	clock cycle
CT	Cooley-Tukey
ECC	elliptic-curve cryptography
FPGA	field-programmable gate array
GS	Gentleman-Sande
INTT	Inverse Number Theoretic Transform
ISA	instruction set architecture
KEM	key-encapsulation mechanism
NIST	National Institute of Standards and Technology
NTT	Number Theoretic Transform
PQC	Post-Quantum Cryptography
PWM	pointwise multiplication

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

The US National Institute of Standards and Technology (NIST) had announced ML-KEM/Kyber as the new post-quantum standard for key-encapsulation mechanism (KEM) in 2022, with standard document released in 2024 [1]. Since then, many research work on developing efficient field-programmable gate array (FPGA) implementation of ML-KEM/Kyber were published [2], [3], [4]. One of the main bottlenecks in ML-KEM/Kyber is polynomial multiplication over the ring $\mathbb{Z}_q[x]/(x^{256}+1)$, with polynomial length $n = 256$. In hardware implementations, these matrix-vector operations dominate the computational cost. For such a large polynomial length, multiplication is usually performed using a more efficient algorithm, NTT, with complexity reduced to $O(n \log n)$. This motivates researchers to develop efficient polynomial multiplier NTT targeting FPGA platform for deployment in constrained devices.

Previous works targeting NTT implementation often utilize Block RAMs (BRAMs) in FPGA to host the twiddle factors [5]. However, such design approach suffers from area-time efficiency, due to the complex BRAM-to-logic paths which potentially affects the maximum clock frequency. Recent work from Nguyen et al. [4] demonstrated that BRAM-less NTT design can be more lightweight, achieving a very area-time efficient design. In this research project, we take this optimized design approach to another level by rethinking the necessity of using a DSP in Kyber NTT/INTT.

1.2 Objectives

The primary objective of this project is to design and implement a highly efficient NTT/INTT hardware architecture for the ML-KEM/Kyber Post-Quantum Cryptography (PQC) scheme on an FPGA platform with better ATP compared to current existing works.

To achieve the primary objective, several sub-objectives are pursued:

- To develop a completely BRAM-less NTT/INTT architecture using LUTRAM to improve the ATP metric without compromising the maximum clock frequency.
- To implement and evaluate three different multiplier architectures to provide flexibility to optimize for power or area-time efficiency:
  - Version 1 (Baseline): A power-efficient architecture utilizing 2 DSPs.
  - Version 2: A novel shared-DSP architecture utilizing 1 DSP.
  - Version 3: A highly ATP-optimized, DSP-less architecture using only LUT-based multipliers.

1.3 Project Scope

The scope of this project is restricted to the NTT/INTT hardware, and specifically for ML-KEM/Kyber. However, it can be expanded in future derivative projects to include pointwise multiplication (PWM) or a larger integrated system.

1.4 Contributions

The contributions of this project are summarized as below:

1. First, we present an area-time efficient NTT multiplier without using any BRAM, which served as the baseline of our implementation (Version 1). This design is already 18.4% more efficient than state-of-the-art [4].
2. A novel shared-DSP NTT multiplier was proposed by optimizing the baseline implementation. This architecture uses only one DSP to perform partial multiplication, and LUT-based multiplier to complete the remaining multiplication (Version 2). This effectively reducing one DSP consumption, achieving ATP of 708.5 and 25.7% more efficient than state-of-the-art [4].
3. We further improve the design by completely removing DSP from the architecture (Version 3), in which all multiplications are done using LUT-based multiplier. This NTT multiplier is 35% more efficient than state-of-the-art [4].

1.5 Report Organization

In this Chapter 1, we provided a brief introduction to our research project. The following sections in this report are organized as follows:

- In Chapter 2, we review the concepts involved in NTT/INTT hardware that have been explored by prior works, including general NTT approaches, modular reduction algorithms, and ML-KEM/Kyber-specific NTT architectures.
- In Chapter 3, we present an overall high-level view with explanations for our system design.
- In Chapter 4, we include the low-level implementation details and specifications for our system design, with descriptions for module functions, pin functions, and system components interaction operations.
- In Chapter 5, we briefly explain our system simulation planning and setup.
- In Chapter 6, we present the detailed simulation test result with explanations for involved performance metrics, and we evaluate our system by comparing against prior works.
- In Chapter 7, we conclude our project with recommendation on future directions.

CHAPTER 2

Literature Review

2.1 Review of NTT Approaches

2.1.1 CFNTT: A Versatile Conflict-free Memory Mapping Scheme

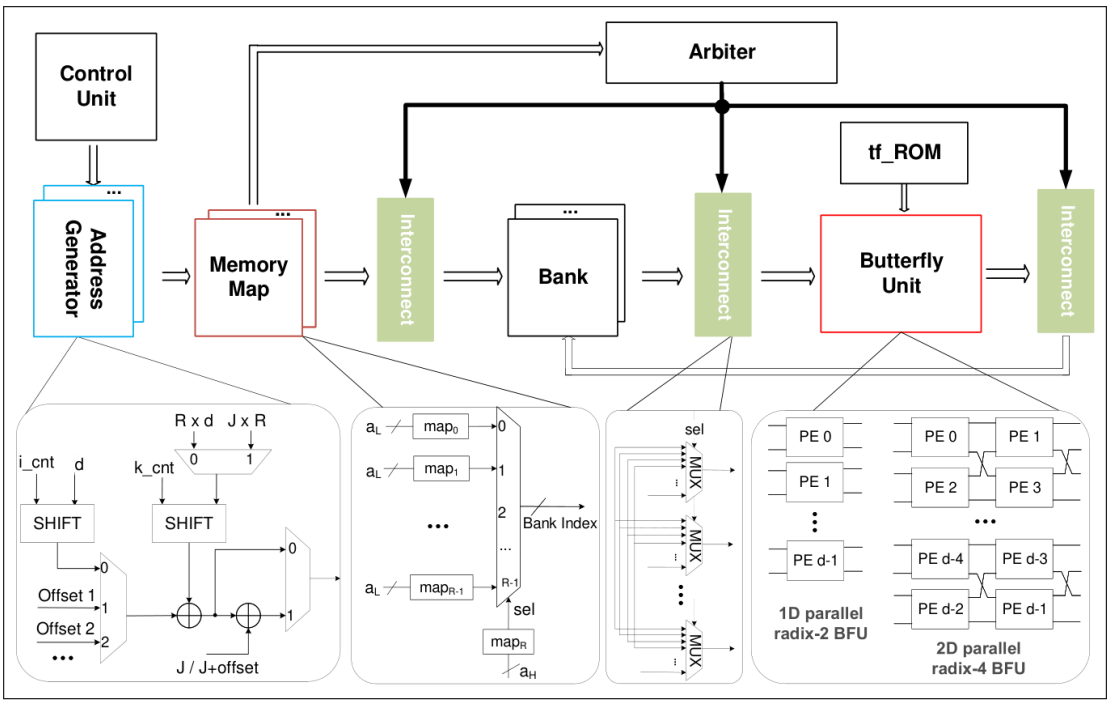


Figure 2.1 CFNTT generalized architecture [2].

CFNTT [2] presents a conflict-free memory mapping scheme for arbitrary radix in-place NTT/INTT that is highly scalable for parallelization and flexible in terms of mixing NTT/INTT radices. The advantages of this memory mapping scheme allows it to be easily applicable for a wide range of lattice-based PQC, including ML-KEM/Kyber, while still maintaining its efficiency in hardware and latency, as data do not need to be rearranged after each butterfly operation. In radix-2 NTT/INTT, the memory mapping logic is simply achieved using a bitwise XOR, which is very minimal hardware.

Algorithm 7 Conflict-free memory mapping scheme for arbitrary radix in-place NTT.

Input: a, N, r^k, d. Here a denotes the old address to be mapped. N denotes the total number of data points. r is a prime and $r^k = R$ denotes the radix of NTT. d represents the number of parallel butterfly units and it also stands for step size.

Output: BI, BA. Here BI denotes the bank index into which the old address is mapped. BA denotes the offset of the storage location within each inner bank.

1: Let B denote the number of banks, where B satisfies the condition $B = R \times d$ and $N \mid B$ in the interleaved memory system.
2: Calculating the total digit width T of old address expressed in radix r: $T = log_r N$.
3: Calculating the digit width of bank number value expressed in radix r: $M = log_r B = log_r(R \times d)$.
4: Computing the digit width of step size expressed in radix r: $C = T - M$.
5: Then the old address a can be expressed in radix r as:
6: $$a = [a_{T-1}, a_{T-2}, ..., a_1, a_0]_r = [a_{T-1}, ..., a_{T-C}, a_{M-1}, ..., a_1, a_0]_r.$$
7: Expressing the lower part $[a_{M-1}, ..., a_1, a_0]_r$ in radix B as $[b]_B$.
8: Expressing the higher part $[a_{T-1}, ..., a_{T-C}]_r$ in radix $R = r^k$ as $[b_{m-1}, b_{m-2}, ..., b_0]_R$, where m is the digit width after radix R transformation.
9: Then the old address can be expressed in mixed radix B-R as $a = [b_{m-1}, b_{m-2}, ..., b_0]_R [b]_B$.
10: Defining the step number as $SN = (b_{m-1} + b_{m-2} + ... + b_0) \bmod R$.
11: Then, the bank index for old address is calculated as: $BI = (b + SN \cdot d) \bmod B$.
12: The new bank address for old address is calculated as: $BA = a >> M = [a_{T-1}, ..., a_{T-C}]_r$.
13: **return** BI, BA.

Figure 2.2 CFNTT memory mapping algorithm [2].

The authors of CFNTT [2] released their Verilog design files online as open source, one sample system for radix-2 NTT and another for radix-4 NTT. We utilize and reconfigure their radix-2 NTT system design template for this research project for the aforementioned advantages.

2.1.2 PipeNTT: NTT With Data Reordering Units

The authors of PipeNTT [6] point out issues that come with increasing the level of parallelism, and proposes their own scalable NTT architecture based on pipelining and optimised data reordering units that avoid spatial conflicts. This has the benefits of less complicated control logic and less area consumption at the expense of introducing additional delays necessary for data reordering stages (visualised in Figure 2.3).

2.1.3 Out-of-place NTT

There are out-of-place algorithm variants of NTT that access data locations differently between input and output locations, in a way that may be more easily manageable when addressing memory with a high degree of parallelism, such as in GPU computing. For example, Stockham NTT as implemented in [7] has a continuous memory access pattern

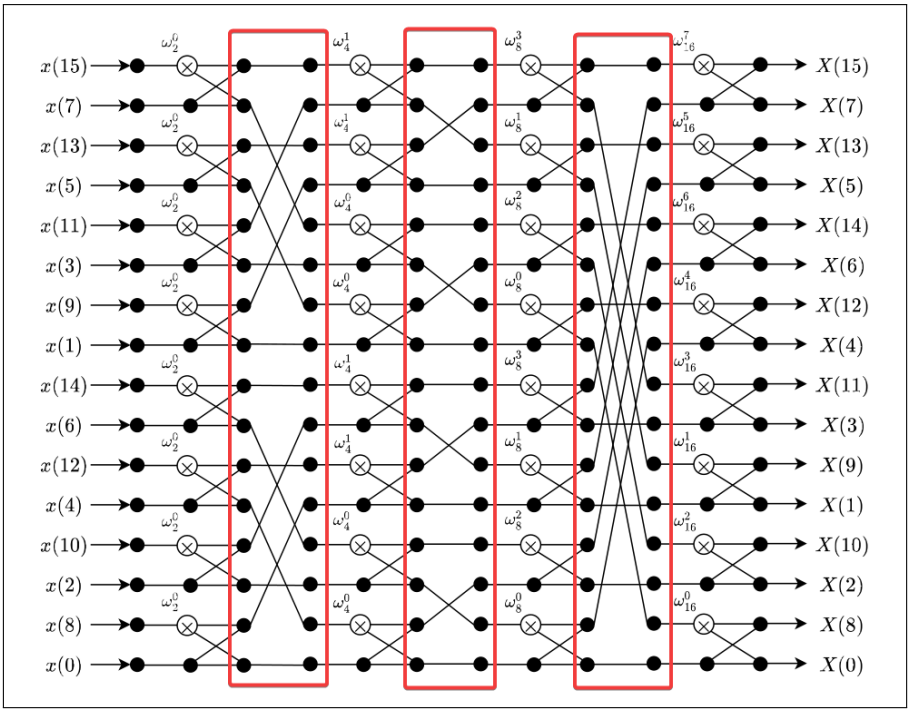


Figure 2.3 PipeNTT dataflow: Delay is introduced for the additional data-reordering stages (framed in red) to maintain constant memory access pattern.

that is inherently well-suited for parrallelisation. Across stages, its memory read pattern is constant and continuous while its memory write pattern is partially continuous. [8] implemented a constant-geometry NTT that would repeat memory access patterns across all stages. Nevertheless, increasing parallelism also means larger memory bandwidth requirements and larger hardware when implemented in FPGA, and considering the processing element bandwidth waste issue that comes with high parallelism in Kyber as discussed in our problem statement, we will not consider out-of-place NTT approaches for our project.

2.1.4 Summary of NTT Approaches Review

We select the CFNTT [2] approach to be applied in our research project for its versatility and ease of reconfiguration; the excellent efficiency of its conflict-free memory mapping scheme is reflected in the results of our system evaluation when compared to other works.

2.2 Review of Modular Reduction Algorithms

2.2.1 Classical Montgomery and Barrett Reduction

Several modular reduction algorithms help perform modular multiplications efficiently without having to rely on expensive division operations to find the modulus for a multiplication between two integers. Montgomery reduction [9] and Barrett reduction [10] are well-known universal methods for such purpose, and are still commonly applied in cryptography computations (e.g. Kyber reference code [11] adopts standard Montgomery reduction; CFNTT [2] adopts standard Barrett reduction). However, there is a common trade-off for using both methods in that for every modular multiplication we would need to perform two more for the reduction step (see steps 1, 2 for Algorithm 1, and steps 2, 4 for Algorithm 2). Barrett reduction is slightly more expensive than Montgomery reduction [12], however Montgomery reduction returns a non-exact modulus which, depending on overall scheme implementation, needs to be corrected with another round of Montgomery multiplication, reduction, or possibly hidden when multiplications involve pre-computed values, whereby the values could be pre-multiplied with the correction factors.

Algorithm 1 Montgomery reduction (as interpreted from Kyber reference C code [11])

Input: Integer a of size 32 bits (signed), constants $R = 2^{16}, q = 3329, q' = -3327 = q^{-1} \pmod R$ ▷ Satisfied conditions $R > q$, $\gcd(R, q) = 1$ [13]

Output: $aR^{-1} \pmod q$ in the range $(-q, q)$

1: $A \leftarrow aq' \pmod R$
2: $A \leftarrow (a - Aq)/R$
3: **return** A

2.2.2 KRED And Its Derivatives KRED2x, K2RED

Longa and Naehrig [14] proposed KRED and KRED2x algorithms to optimise Montgomery reduction specifically for modulo operations using Proth numbers as divisors. A Proth number is a natural number that can be expressed in $k \cdot 2^m + 1$, where k is odd and $2^m > k$. The modulo divisor prime integer, $q = 3329$ for Kyber, satisfies the condition as $q = 3329 = 13 \cdot 2^8 + 1$ and $2^8 > 13$. These reductions do not perform any additional multiplication but only involve several shift and add operations. However,

Algorithm 2 Barrett reduction (as interpreted from CFNTT reference Verilog code [2])

Input: Integer a of size $2k$ bits (unsigned), constants $q = 12289, \mu = 21843 = \left\lfloor \frac{2^{2k}}{q} \right\rfloor$, where $k = 14 = \lceil \log_2 q \rceil$

Output: $a \pmod q$ in the range $[0, q)$

1: $B_1 \leftarrow \lfloor a/2^{k-1} \rfloor$
2: $B_2 \leftarrow B_1 \cdot \mu$ ▷ Performed and stored in $2(k+1)$-bit space
3: $B_3 \leftarrow \lfloor B_2/2^{k+1} \rfloor$
4: $A_1 \leftarrow B_3 \cdot q$
5: $A_2 \leftarrow a - A_1$
6: **if** $A_2[k:0] - q < 0$ **then**
7: $\quad A \leftarrow A_2[k-1:0]$
8: **else**
9: $\quad A \leftarrow A_2[k:0] - q$
10: **end if**
11: **return** A

since the KRED and KRED2x algorithms allow the output to grow beyond $\lceil \log_2 q \rceil = 12$ bits, [5] proposed K2RED, which performs KRED twice in a row to limit the output to 12 bits. K2RED reduction also requires half as much ROM as implementing both KRED and KRED2x for storing pre-computed multiplication factors. It was implemented with 4 shift and 6 addition operations.

2.2.3 Optimized Implementation of Barrett Reduction

On the other hand, Xing et al. [15] also proposed a modified Barrett reduction algorithm to replace the additional multiplications with shift and add operations. Along with the desired remainder, it also outputs the approximate quotient from reducing the input modulo q as a byproduct, which they claim is beneficial for the Compress procedure in Kyber KEM. Guo et al. in [16] also adopted Barrett reduction for their mixed-radix implementation. However, they proposed their own optimisations for an approximate Barrett reduction algorithm using multiple sets of bit recombination procedures involving a variety of bitwise logic, and summations to consolidate the recombination sets.

2.2.4 Plantard Reduction

Plantard in [12] discussed in length on other less-known modular multiplication algorithms that are considered highly efficient but only for large multi-precision operations beyond typical word sizes, i.e., they are not suitable for optimising the cryptography schemes mentioned thus far. He proposed his modular multiplication algorithm that is purposefully made efficient for single-word (32-bit/64-bit) multiplications which is especially friendly with software implementation. Its primary advantage is that it requires less additional multiplication compared to equivalent Montgomery or Barrett counterparts. Though Plantard and Montgomery multiplications may look similar on the algorithm level in terms of number of multiplications involved with the fact that both return non-exact moduli, Plantard's algorithm does not have an explicit dependency with the original pair of operands multiplied in isolation like that of Montgomery (see a as the product of two terms in Algorithm 1; notice direct dependency on step 2), therefore it benefits from the fact that layers of precomputation can be merged and applied all at once to one of the operands right from the beginning, hence removing the need for corrective operations altogether.

Huang et al. [17], with slight modification, improved Plantard's algorithm in two ways. Firstly, they increased the input range for minimum word length of 16 bits from mere $[0, q]$ for both operands to $[-64q, 64q]$ (specifically for Kyber [18]), or up to $[-128q, 128q]$ when one operand is a precomputed constant reduced to $[0, q)$. Secondly, they shifted the output from unsigned range $[0, q]$ to signed range $\left[-\frac{q-1}{2}, \frac{q}{2}\right)$ to eliminate the need for extra addition operations to keep the output in positive range. Later, Huang et al. [18] recalculated the input range improvement from limiting a constant operand to $[0, q)$, and showed that the actual maximum input range of the other operand is at $[-137q, 230q]$ (specifically for Kyber). This enlarged input range enables lazy reduction strategies. For instance, since forward NTT for Kyber only needs 7 layers of radix-2 Cooley-Tukey (CT) BFU, each performing just an addition on lower-index BFU inputs, the coefficients that are never multiplied with another term can only grow up to $+3.5q \rightarrow 4.5q$ after 7 layers when applying the improved Plantard arithmetic, which is well within the input range after the improvement to the original Plantard arithmetic. Therefore, all additions and subtractions during forward NTT do not require reduction

steps and the output coefficients can be directly used for PWM.

Algorithm 3 Plantard modular multiplication [18]

Input: Signed integers a, b such that $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$, $q < 2^{l-\alpha-1}$, $q' = q^{-1} (\mathrm{mod}^{\pm} 2^{2l})$ ▷ $\mathrm{mod}^{\pm}$ indicates modulo in signed range

Output: $r = ab(-2^{-2l})(\mathrm{mod}^{\pm} q)$ where $r \in \left[-\frac{q-1}{2}, \frac{q}{2}\right)$

1: $A \leftarrow abq' \pmod{2^{2l}}$ ▷ bq' can be precalculated
2: $A \leftarrow \left\lfloor \frac{A}{2^l} \right\rfloor + 2^{\alpha}$
3: $A \leftarrow \left\lfloor \frac{Aq}{2^l} \right\rfloor$
4: **return** A

Plantard arithmetic has very recently been implemented for Kyber on the 64-bit RISC-V RV64IMB instruction set architecture (ISA) [3], and its implementation in combination with Montgomery arithmetic has seen at least 77% and 86% speedup in Kyber NTT and INTT, respectively, compared to NIST reference implementation. However, there is currently no NTT architecture in pure hardware implementation using Plantard arithmetic.

2.2.5 Summary of Modular Reduction Algorithms Review

We select the K2RED reduction algorithm to be applied in our research project, although we had initial interest in using the more recent Plantard reduction algorithm. This is because an implementation of Plantard algorithm required doubling the size of twiddle factor memory, and it is also not suitable for a direct implementation of general-purpose modular multiplication hardware without having to heavily optimize a really large multiplication involving two operands at least 24 bits wide. In contrast, K2RED is versatile, fast, and lightweight, and it only requires one additional multiplication with a 12-bit corrective factor which can also be pre-processed on twiddle factor constants.

2.3 Review of Existing NTT Architecture for CRYSTALS-Kyber

2.3.1 Mixed Radix-2/4 NTT

One prominent paper in Kyber KEM hardware optimisation by Guo et al. [16] presented a novel architecture based on mixed radix-2/4 NTT to adapt to the Kyber-specific

awkwardness in designing accelerator hardware without utilisation wastage as described earlier, which was suffered by [5] implemented with four radix-2 BFUs parallelised in pairs. In their mixed radix-2/4 design, they performed three stages with a radix-4 BFU alternating between even and odd terms (stored in different banks), then one last stage with two pairs of radix-2 BFUs separately on even and odd terms at the same time, thereby fully utilising the available multipliers. In combination with several other optimisations, Guo et al. [16] reported 33% lower ATP than that of [5] in Kyber KEM implementation, yet the NTT latency was also improved by 23%. The wastage of multipliers is avoided by their so-called 'lazy-last-layer' approach that temporarily changes memory bandwidth by modifying the number of banks accessed at one time between radix-4 and radix-2 cycles. It is currently the only attempt at radix-2/4 BFU configuration for Kyber in the literature, yet it is a particularly low-latency approach without over-reliance on parallelism (high memory bandwidth requirements) and produced notable ATP improvement over other prior implementations.

2.3.2 Radix-2 BRAM-less In-place NTT

Attempts that claim to outperform [16] include a radix-2, in-place iterative NTT approach proposed by Nguyen et al. [4] that removes the need of extra BRAM like the design in [16], which significantly contributes to reducing the ATP. However, it also introduces notable latency due to delays required after every BFU operation for coefficient reordering, resulting in a speed almost twice as slow as that of [16]. Nevertheless, it had a focus on area optimisation and also achieved $5 \sim 16\%$ better ATPs when comparing their full Kyber KEM implementations.

2.3.3 Other Mixed Radix NTT

Other mixed-radix attempts include [19] that proposed a configurable mixed-radix design supporting different schemes including Kyber. However, its configurability was based on stacking radix-2 BFUs to be packaged as higher-radix modules with no radix-specific optimisations, and their Kyber NTT core ATP is close to that of [5]. In another attempt, Suraj and Basu Roy [20] propsoed their mixed radix-8/16 Winograd NTT architecture for Kyber which required fewer modular multiplications than equivalent CT/Gentleman-Sande (GS)-based counterparts. It achieved around 4

times less NTT/INTT latency than [16]. However, their design was much larger and resulted in an ATP around 2.5 times higher than [16].

Sun and Bai [21] claimed to outperform [16] using a direct radix-4 NTT approach. It was a novel combination of sub-architectures involving K2RED reduction algorithm and ping-pong memory access scheme. However, the approach was equivalent to performing eight recursive stages of radix-2 BFU operations, which was mathematically incorrect.

2.3.4 Split Radix NTT

Another interesting, slightly more recent work by Guo et al. [22] implements split radix approach for Kyber, which is currently also the only study of split-radix NTT/INTT architecture for Kyber in the literature. It is a more (mathematically and structurally) complex implementation that combines radix-2 and radix-4 steps in an optimised way [23]. This work boasts impressive area-time optimisation with less multipliers than their earlier work [16] yet completes NTT execution in less clock cycles (CCs) than other prominent works that use more multipliers [5], [15], [24].

2.3.5 Summary of Existing NTT Architecture for CRYSTALS-Kyber Review

In contrast to every aforementioned architecture, we implement a BRAM-less, in-place iterative NTT/INTT module with two parallel radix-2 BFU without data reordering based on CFNTT [2]; the design can also be easily reconfigured to utilize 0, 1, or 2 DSP in its multiplier.

CHAPTER 3

System Model

3.1 System Design Diagram

3.1.1 System Architecture Diagram

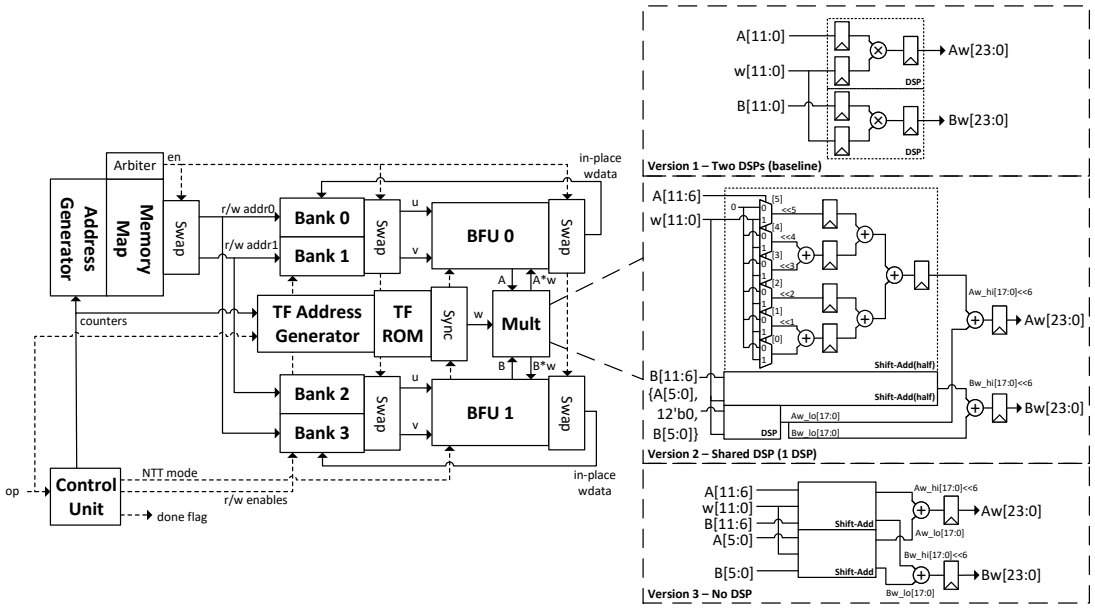


Figure 3.1 Proposed NTT/INTT architecture. The multiplier submodule can be implemented in three designs as shown in versions 1 to 3.

Our NTT/INTT module follows the same conflict-free memory mapping design principles from CFNTT [2], which only require little hardware to enable the in-place NTT/INTT design using two parallel radix-2 BFUs that can operate in either CT or GS mode, without additional delays needed to rearrange data between each butterfly operation. The overall architecture is shown in Figure 3.1. Due to the nature of NTT/INTT in ML-KEM that the 256 coefficients are split into even and odd halves, our NTT/INTT module parallelizes the radix-2 butterfly operations just by duplicating a BFU with its pair of memory banks and sharing the same address busses between the two pairs of data memory banks, one for odd coefficients and the other for even coefficients. Each simple dual-port LUTRAM memory bank stores 64 coefficients in a 64×12-bit configuration; the coefficient arrangements are static and follow the design principles of CFNTT [2].

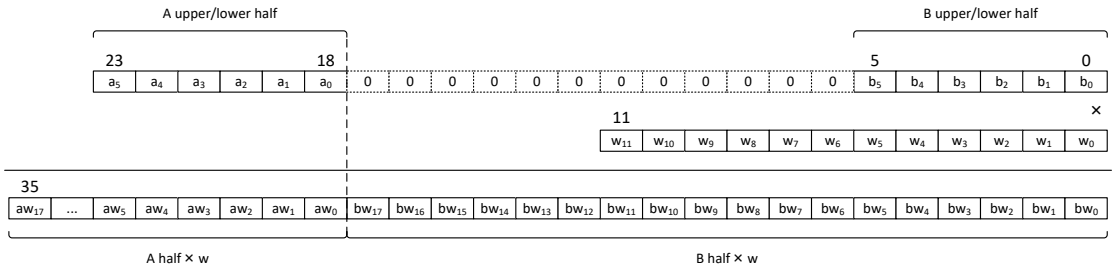


Figure 3.2 Proposed DSP-based half multiplier operation to reduce DSP48E1 utilization wastage in Artix-7 FPGAs. Two independent operand halves a, b are arranged as one 24-bit operand that is multiplied with a shared 12-bit twiddle factor constant w to produce a long output that can be directly split into two multiplication results.

The 24-bit multiplication results are reduced back within the range of $[0, q)$ using the K2RED algorithm introduced in [5], which is also presented in Algorithm 4. Referring to Algorithm 4, two KRED operations (lines 1 – 6) are successively performed on an arbitrary 24-bit unsigned input C to first obtain an intermediate 17-bit C' and then produce a signed 13-bit C''_M that falls within the range of $[-q, 2q)$. A correction step is then performed to produce an unsigned 12-bit output $C'' \in [0, q)$, in which conditions can be matched using the 13th bits of C''_M and C''_L for unsigned subtraction underflow (lines 7 – 15). The hardware implementation of the K2RED algorithm is shown in Figure 3.3. We have chosen the K2RED algorithm in this work because it can be more advantageous than conventional Montgomery [9] or Barrett [10], and even the more recent Plantard [18] algorithms; it is versatile, fast, and lightweight, yet simple as it does not require any large multiplication (except for the pre-processing of operands). The two small multiplications with constant $k = 13$ can be achieved through light shift-add operations. However, since KRED and its derivatives (including K2RED) can only bring input integers close to but not exactly the desired $[0, q)$ range [14], a little additional hardware, e.g., two 12-bit adders for addition and subtraction with multiplexers for result selection, is needed to close the final gaps in case the results are slightly lower $(-q)$ or higher $(+q)$ than the correct range.

Since the NTT/INTT operations are symmetric between the even and odd coefficient halves, they share the same twiddle factor in each butterfly stage. As such, the utilization of DSP48E1 slices in Artix-7 FPGAs that support up to 25-bit by 18-bit multiplications can be increased by making use of the larger input port to fit in two halves of 12-bit

Algorithm 4 K2RED Reduction Algorithm [5] With Output Correction

Input: A binary number $C = (c_{23}, \ldots, c_0)_2$, $k = 13 = 1101_b$, $q = 3329$

Output: $C'' = k^2C \pmod q \in [0, q)$

First KRED:

1: $C_l = (c_7, \ldots, c_0)_2$

2: $C_h = (c_{23}, \ldots, c_8)_2$

3: $C' \leftarrow k \cdot C_l - C_h$

Second KRED:

4: $C'_l = (c'_7, \ldots, c'_0)_2$

5: $C'_h = (c'_{16}, \ldots, c'_8)_2$

6: $C''_M \leftarrow k \cdot C'_l - C'_h$

Output Correction:

7: $C''_L \leftarrow C'' - q$

8: $C''_H \leftarrow C'' + q$

9: **if** $C''_M < 0$ **then**

10: **return** C''_H

11: **else if** $C''_L < 0$ **then**

12: **return** C''_M

13: **else**

14: **return** C''_L

15: **end if**

operands from different BFUs, as opposed to the direct approach of implementing two 12-bit × 12-bit multipliers in two separate DSPs (version 1 in Figure 3.1). As visualized in Figure 3.2 for a half multiplier, the upper or lower six bits of both operands A and B, a and b are arranged on the 25-bit wide DSP input with 12 zero-bits sandwiched between them, so that when sharing the multiplication with the 12-bit twiddle factor w, the 36-bit DSP output can be directly divided into two 18-bit results of $a \times w$ and $b \times w$. Then, the 18-bit results of two half multipliers are added with the upper-half result shifted left by six bits to obtain the final results of $A \times w$ and $B \times w$.

Figure 3.1 shows the internal architecture of the shared multiplier submodule in versions 2 and 3 that divides the shared multiplication operations into lower and upper halves

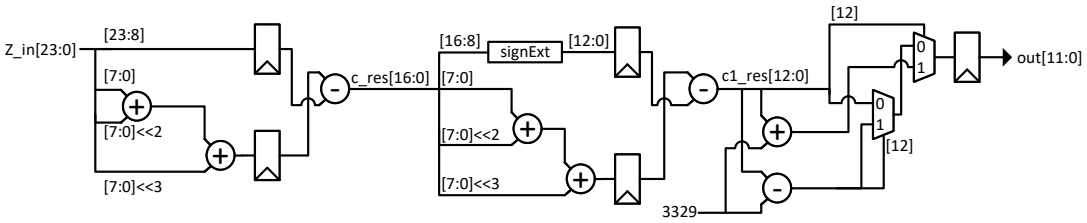


Figure 3.3 K2RED modular reduction module internal architecture based on Algorithm 4.

of the arbitrary operands A and B produced by BFU 0 and BFU 1, times the shared twiddle factor w. Depending on the desired configuration, the multiplier in each half can be either LUT-based or DSP-based designs, although we only show two possible combinations in versions 2 and 3. Our LUT-based multiplier design is based on a regular shift-add pattern. In case of the DSP-based approach, it is just a single DSP slice with registered input and output that are divided as shown in Figure 3.2.

3.1.2 Use Case Diagram and Description

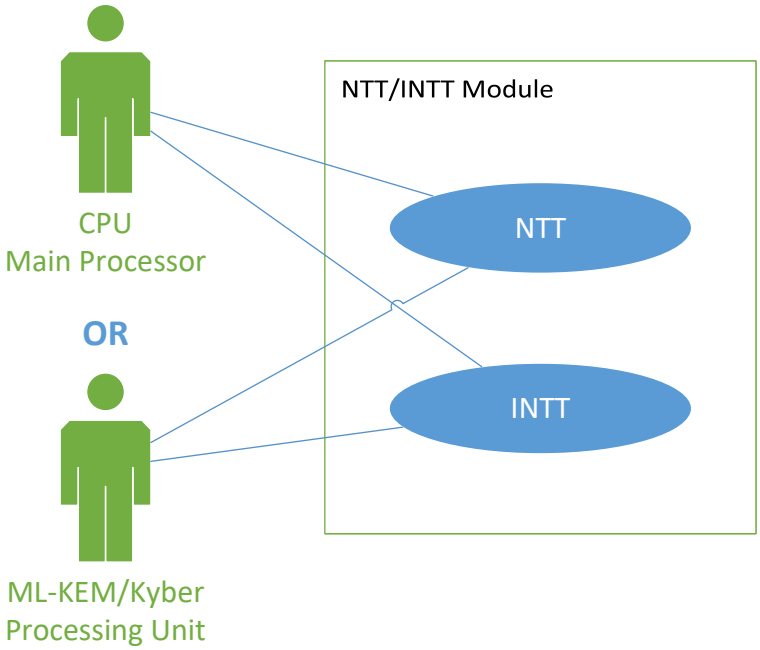


Figure 3.4 NTT/INTT module use case diagram.

Figure 3.4 displays the possible use cases of our NTT/INTT module. The module can be integrated as a coprocessor to a CPU, or as a submodule within an entire ML-KEM/Kyber processing unit, where the NTT or INTT functionality can be accessed

as needed in both cases.

3.1.3 Activity Diagram

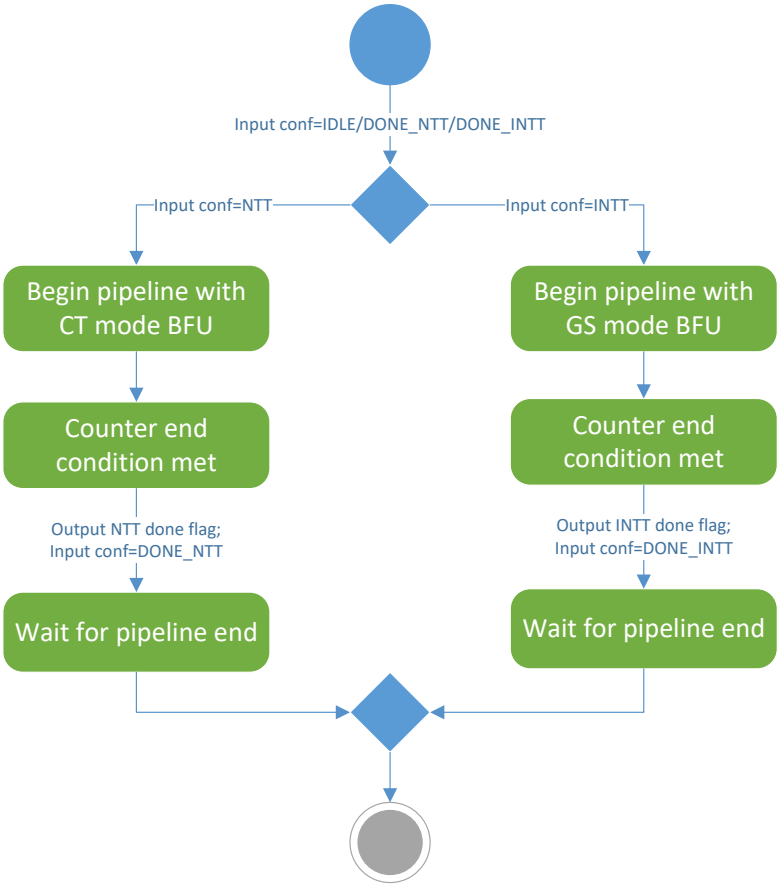


Figure 3.5 System activity diagram.

Figure 3.5 shows the overall activity diagram of our NTT/INTT module. Its Control Unit FSM input mode is sent by an external system with possible options: IDLE, NTT, INTT, DONE_NTT, and DONE_INTT. The switch from IDLE/DONE_NTT/DONE_INTT to either NTT or INTT mode begins a new pipeline with appropriate control signals to internal components according to desired NTT direction. When the FSM counters reach the end, the FSM will output a done flag to the external system which will need to immediately assert the corresponding DONE_NTT or DONE_INTT mode to the mode input so the pipeline completes correctly. Thoguh depending on the module integration, the inputs and outputs of the FSM can be easily reconfigured to better accomodate different requirements or pipelines.

CHAPTER 4

System Design

4.1 System Block Diagram

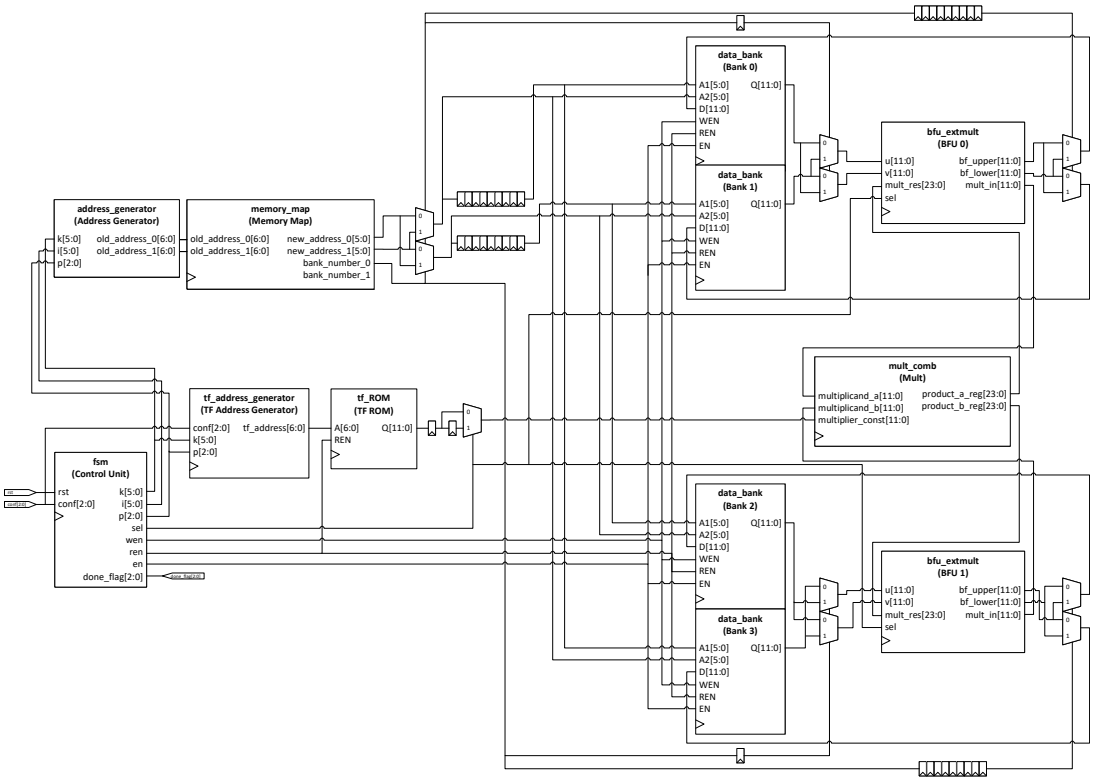


Figure 4.1 Overall system block diagram.

Figure 4.1 displays the full system block diagram that shows the direct implementation of the system architecture as shown prior in Figure 3.1; the arbiter, swap logic, and synchronization logic with simple internals are shown collapsed as glue components.

4.2 System Components Specifications

4.2.1 Control Unit: fsm.sv

The Control Unit FSM receives its operation configuration conf[2:0] as its input and operates the overall NTT/INTT module. Its FSM output done_flag[2:0] notifies the external system when its internal counters reach the end condition for NTT/INTT.

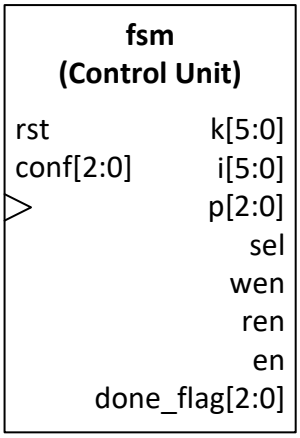


Figure 4.2 Control Unit.

Table 4.1 Control Unit Input Pin Description.

Pin Name:	Source → Destination:
rst	External Source → Control Unit
Pin Function: Synchronous reset.	
Pin Name:	Source → Destination:
conf[2:0]	External Source → Control Unit
Pin Function: Operation configuration. Possible enumerated modes: • 000: IDLE – Do nothing • 001: NTT – Perform NTT • 010: PWM (reserved) • 011: INTT – Perform INTT • 100: DONE_NTT – NTT pipeline ending • 101: DONE_INTT – INTT pipeline ending	

Table 4.2 Control Unit Output Pin Description.

Pin Name:	Source → Destination:
k[5:0]	Control Unit → Address Generator, TF Address Generator

Pin Function: NTT/INTT stage secondary counter. An update in k indicates an update of twiddle factor.	
Pin Name: i[5:0]	Source → Destination: Control Unit → Address Generator
Pin Function: NTT/INTT stage tertiary counter.	
Pin Name: p[2:0]	Source → Destination: Control Unit → Address Generator, TF Address Generator
Pin Function: NTT/INTT stage primary counter. An update in p indicates an update in NTT/INTT stage progression.	
Pin Name: sel	Source → Destination: Control Unit → Glue Components, Butterfly Units
Pin Function: Indicates forward (0) or inverse (1) NTT to system components.	
Pin Name: wen	Source → Destination: Control Unit → Data Banks
Pin Function: Write enable for coefficient data banks.	
Pin Name: ren	Source → Destination: Control Unit → Data Banks, TF ROM
Pin Function: Read enable for coefficient data banks and TF ROM.	
Pin Name: en	Source → Destination: Control Unit → Data Banks
Pin Function: Read/write prerequisite enable for coefficient data banks.	
Pin Name: done_flag[2:0]	Source → Destination: Control Unit → External Source

Pin Function:

Notifies external system when NTT/INTT counters reach the end condition. Possible flag configurations:

- 000: Idle/Processing
- 001: NTT counters complete
- 010: PWM counters complete (reserved)
- 100: INTT counters complete

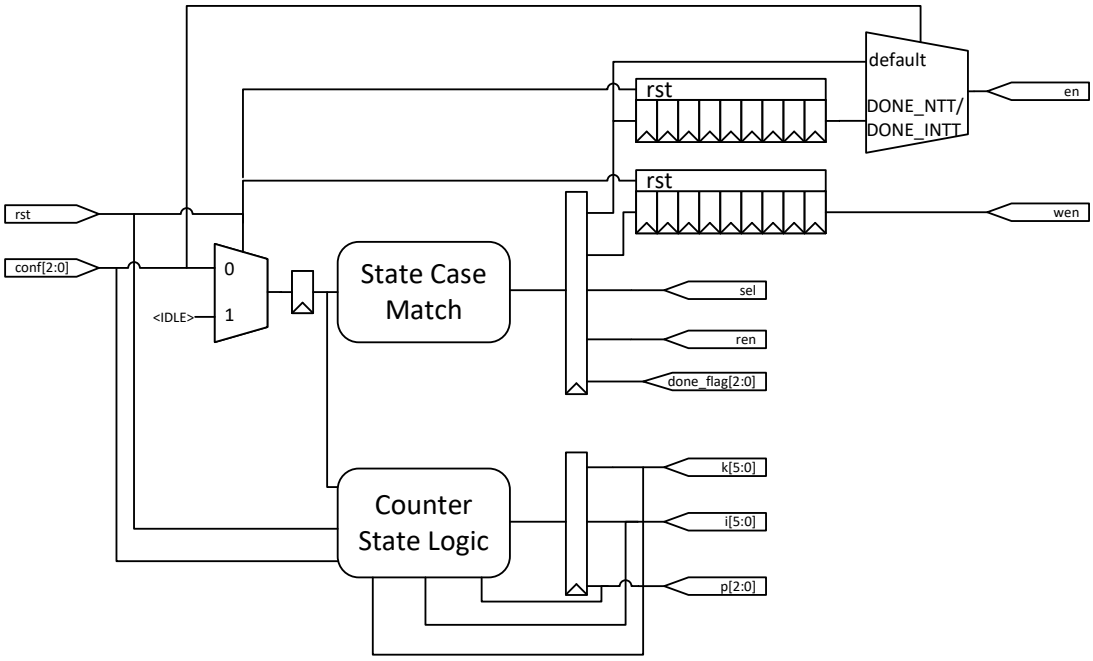


Figure 4.3 Control Unit Schematic.

4.2.2 Address Generator: address_generator.sv

The Address Generator receives the NTT/INTT counters from Control Unit to generate the respective coefficient addresses to be further processed by the Memory Map module.

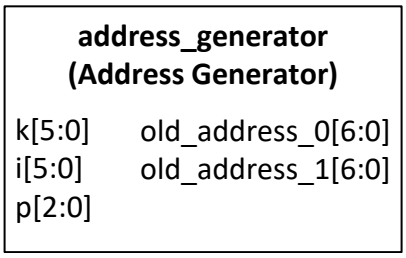


Figure 4.4 Address Generator.

Table 4.3 Address Generator Input Pin Description.

Pin Name: k[5:0]	Source → Destination: Control Unit → Address Generator
Pin Function: NTT/INTT stage secondary counter. An update in k indicates an update of twiddle factor.	
Pin Name: i[5:0]	Source → Destination: Control Unit → Address Generator
Pin Function: NTT/INTT stage tertiary counter.	
Pin Name: p[2:0]	Source → Destination: Control Unit → Address Generator
Pin Function: NTT/INTT stage primary counter. An update in p indicates an update in NTT/INTT stage progression.	

Table 4.4 Address Generator Output Pin Description.

Pin Name: old_address_0[6:0]	Source → Destination: Address Generator → Memory Map
Pin Function: Unprocessed coefficient address 0.	
Pin Name: old_address_1[6:0]	Source → Destination: Address Generator → Memory Map
Pin Function: Unprocessed coefficient address 1.	

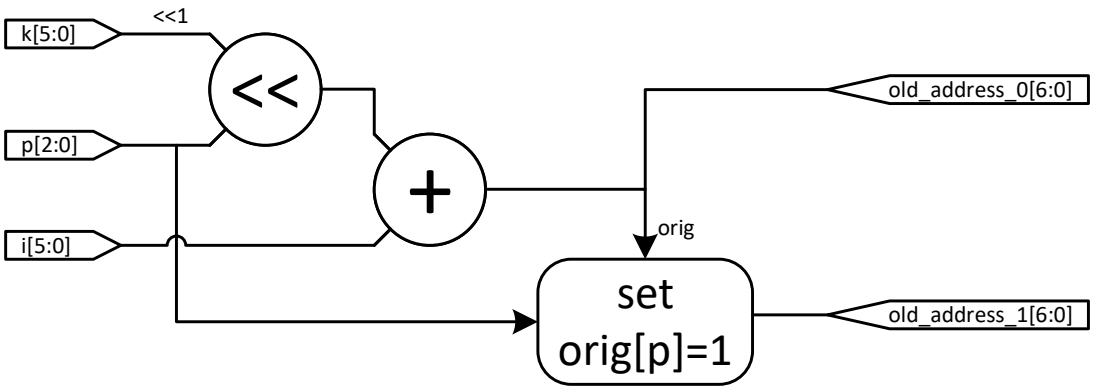


Figure 4.5 Address Generator Schematic.

4.2.3 Memory Map: memory_map.sv

The Memory Map module takes the unprocessed coefficient addresses from the Address Generator, and extracts the actual coefficient pair addresses while simultaneously determining whether the addresses should be swapped with each other.

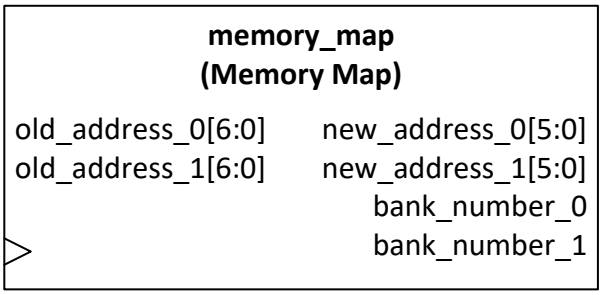


Figure 4.6 Memory Map.

Table 4.5 Memory Map Input Pin Description.

Pin Name: old_address_0[6:0]	Source → Destination: Address Generator → Memory Map
Pin Function: Unprocessed coefficient address 0.	
Pin Name: old_address_1[6:0]	Source → Destination: Address Generator → Memory Map
Pin Function: Unprocessed coefficient address 1.	

Table 4.6 Memory Map Output Pin Description.

Pin Name: new_address_0[5:0]	Source → Destination: Memory Map → System Glue Logic
Pin Function: Actual coefficient address 0.	
Pin Name: new_address_1[5:0]	Source → Destination: Memory Map → System Glue Logic
Pin Function: Actual coefficient address 1.	
Pin Name: bank_number_0	Source → Destination: Memory Map → System Glue Logic
Pin Function: Address swap control signal.	
Pin Name: bank_number_1	Source → Destination: Memory Map → None (Reserved)
Pin Function: Address swap control signal (reserved).	

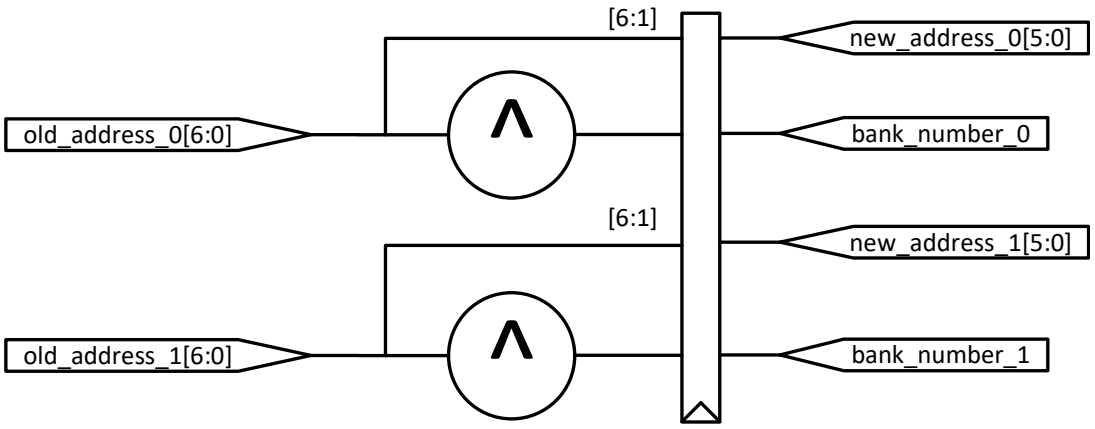


Figure 4.7 Memory Map Schematic.

4.2.4 Data Bank: data_bank_0.sv, data_bank_1.sv, data_bank_2.sv, data_bank_3.sv

The Data Bank comprises of simple dual-port LUTRAMs that store the ML-KEM coefficients in a 64 × 12-bit configuration. Each Data Bank serves an input and an

output of a BFU for in-place data processing. The module is duplicated into four files to separately store the initially loaded coefficients for simulation.

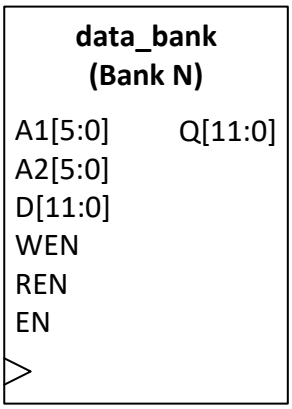


Figure 4.8 Data Bank.

Table 4.7 Data Bank Input Pin Description.

Pin Name: A1[5:0]	Source → Destination: System Glue Logic → Data Bank
Pin Function: Write address.	
Pin Name: A2[5:0]	Source → Destination: System Glue Logic → Data Bank
Pin Function: Read address.	
Pin Name: D[11:0]	Source → Destination: System Glue Logic → Data Bank
Pin Function: Write data.	
Pin Name: WEN	Source → Destination: Control Unit → Data Bank
Pin Function: Write enable.	
Pin Name: REN	Source → Destination: Control Unit → Data Bank

Pin Function: Read enable.	
Pin Name: EN	Source → Destination: Control Unit → Data Bank
Pin Function: Read/write prerequisite enable.	

Table 4.8 Data Bank Output Pin Description.

Pin Name: Q[11:0]	Source → Destination: Data Bank → System Glue Logic
Pin Function: Read output data.	

4.2.5 TF Address Generator: tf_address_generator.sv

The TF Address Generator obtains the NTT/INTT counters from Control Unit and produces the corresponding twiddle factor address according to received operation mode (NTT/INTT) from external source.

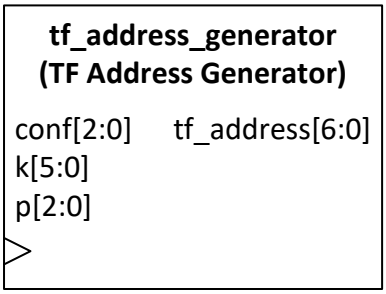


Figure 4.9 TF Address Generator.

Table 4.9 TF Address Generator Input Pin Description.

Pin Name: conf[2:0]	Source → Destination: Control Unit → TF Address Generator

Pin Function: Operation configuration. Relevant enumerated modes: • 001: NTT – Generate address in NTT order • 011: INTT – Generate address in INTT order • 100: DONE_NTT – Generate address in NTT order • 101: DONE_INTT – Generate address in INTT order	
Pin Name: k[5:0]	Source → Destination: Control Unit → TF Address Generator
Pin Function: NTT/INTT stage secondary counter. An update in k indicates an update of twiddle factor.	
Pin Name: p[2:0]	Source → Destination: Control Unit → TF Address Generator
Pin Function: NTT/INTT stage primary counter. An update in p indicates an update in NTT/INTT stage progression.	

Table 4.10 TF Address Generator Output Pin Description.

Pin Name: tf_address[6:0]	Source → Destination: TF Address Generator → TF ROM
Pin Function: Generated twiddle factor read address.	

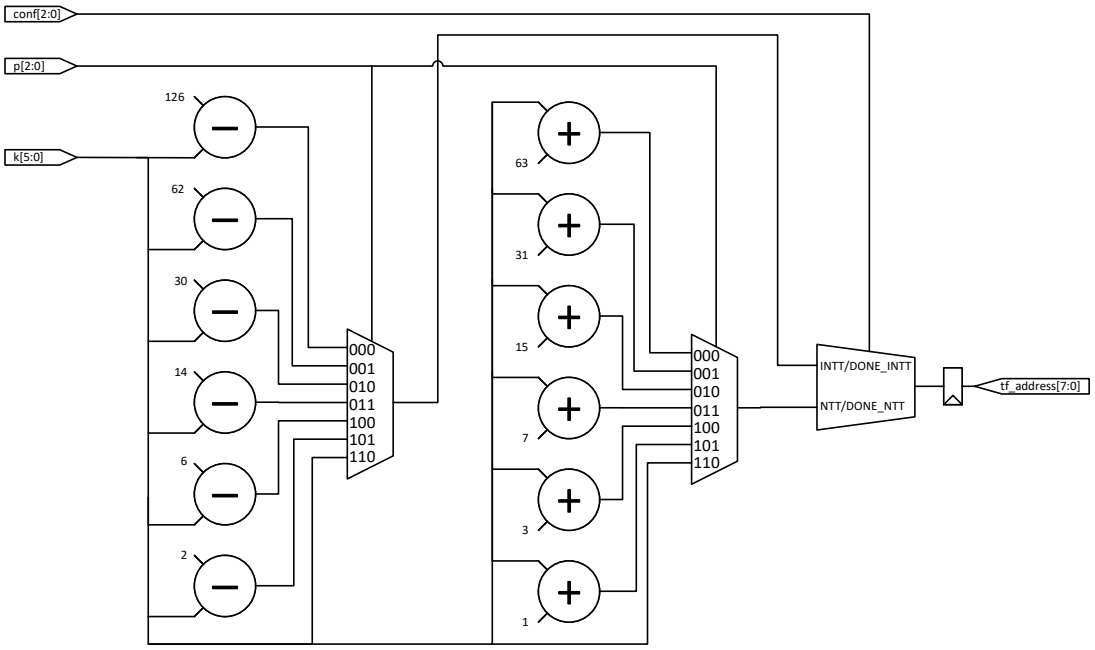


Figure 4.10 TF Address Generator Schematic.

4.2.6 TF ROM: tf_ROM.sv

The TF ROM is a LUTRAM-based ROM that stores 127 twiddle factors to be utilized for CT/GS butterfly operations.

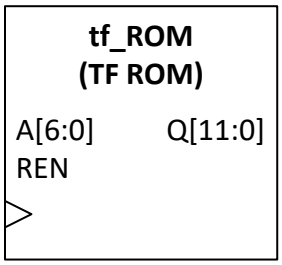


Figure 4.11 TF ROM.

Table 4.11 TF ROM Input Pin Description.

Pin Name: A[6:0]	Source → Destination: TF Address Generator → TF ROM
Pin Function: Twiddle factor read address.	
Pin Name: REN	Source → Destination: Control Unit → TF ROM

Pin Function: Twiddle factor read enable.

Table 4.12 TF ROM Output Pin Description.

Pin Name: Q[11:0]	Source → Destination: TF ROM → System Glue Logic
Pin Function: Twiddle factor read value.	

4.2.7 Multiplier: mult_comb.sv

The Multiplier shares the same twiddle factor for the multiplication with two 12-bit multiplicands received from the two different BFUs.

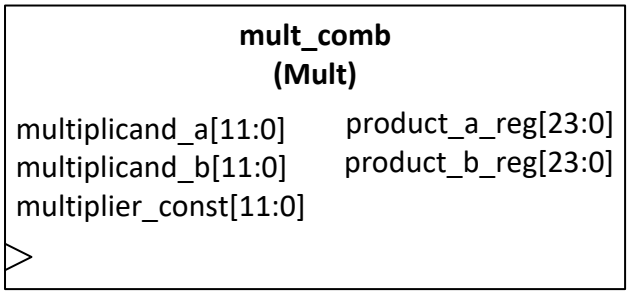


Figure 4.12 Multiplier.

Table 4.13 Multiplier Input Pin Description.

Pin Name: multiplicand_a[11:0]	Source → Destination: BFU 0 → Multiplier
Pin Function: Arbitrary 12-bit input.	
Pin Name: multiplicand_b[11:0]	Source → Destination: BFU 1 → Multiplier
Pin Function: Arbitrary 12-bit input.	

Pin Name:	Source → Destination:
multiplier_const[11:0]	System Glue Logic → Multiplier
Pin Function: Arbitrary 12-bit shared twiddle factor input.	

Table 4.14 Multiplier Output Pin Description.

Pin Name:	Source → Destination:
produce_a_reg[23:0]	Multiplier → BFU 0
Pin Function: 24-bit output of multiplicand_a × multiplier_const.	
Pin Name:	Source → Destination:
produce_b_reg[23:0]	Multiplier → BFU 1
Pin Function: 24-bit output of multiplicand_b × multiplier_const.	

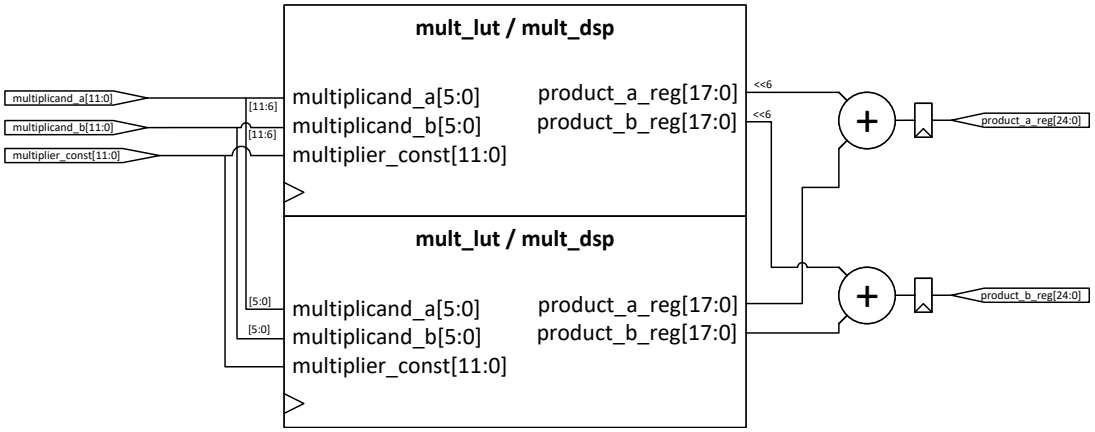


Figure 4.13 Multiplier Schematic.

4.2.8 LUT-based Half Multiplier: mult_lut.sv

The LUT-based Half Multiplier shares the same twiddle factor for the multiplication with two 12-bit multiplicands received from the two different BFUs, that are split in half to 6 bits each.

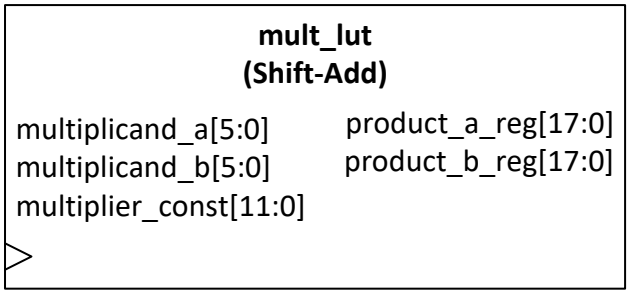


Figure 4.14 LUT-based Half Multiplier.

Table 4.15 LUT-based Half Multiplier Input Pin Description.

Pin Name: multiplicand_a[5:0]	Source → Destination: Multiplier input → LUT-based Half Multiplier
Pin Function: Arbitrary 6-bit input.	
Pin Name: multiplicand_b[5:0]	Source → Destination: Multiplier input → LUT-based Half Multiplier
Pin Function: Arbitrary 6-bit input.	
Pin Name: multiplier_const[11:0]	Source → Destination: Multiplier input → LUT-based Half Multiplier
Pin Function: Arbitrary 12-bit shared twiddle factor input.	

Table 4.16 LUT-based Half Multiplier Output Pin Description.

Pin Name: produce_a_reg[17:0]	Source → Destination: LUT-based Half Multiplier → Multiplier Internal Logic
Pin Function: 18-bit output of multiplicand_a × multiplier_const.	
Pin Name: produce_b_reg[17:0]	Source → Destination: LUT-based Half Multiplier → Multiplier Internal Logic
Pin Function: 18-bit output of multiplicand_b × multiplier_const.	

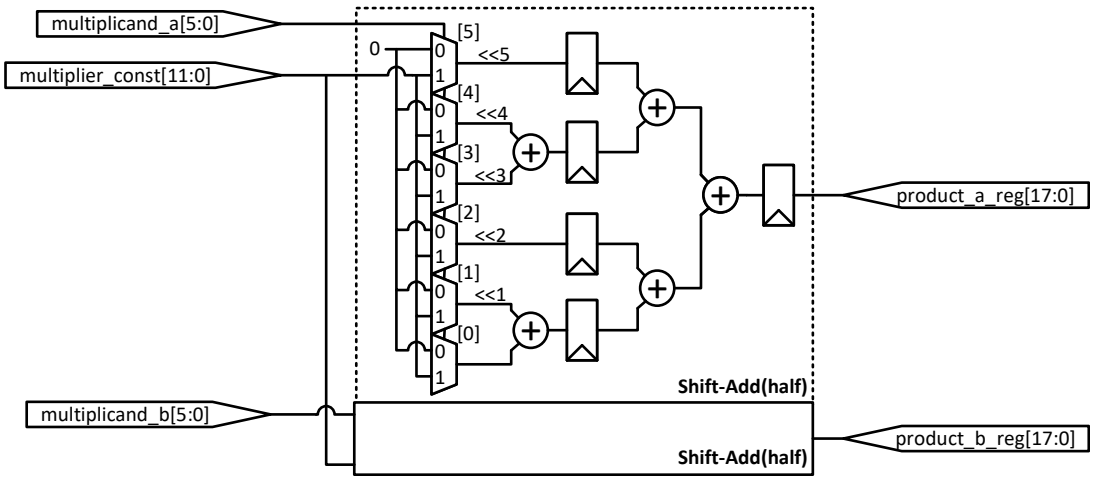


Figure 4.15 LUT-based Half Multiplier Schematic.

4.2.9 DSP-based Half Multiplier: mult_dsp.sv

The DSP-based Half Multiplier shares the same twiddle factor for the multiplication with two 12-bit multiplicands received from the two different BFUs, that are split in half to 6 bits each.

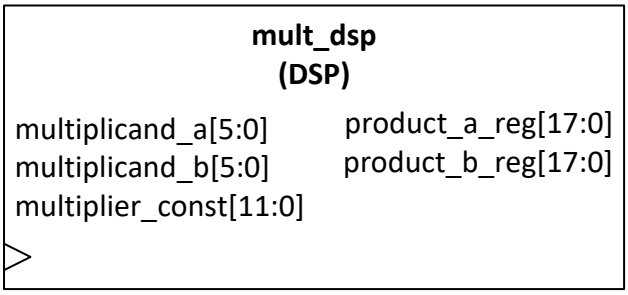


Figure 4.16 DSP-based Half Multiplier.

Table 4.17 DSP-based Half Multiplier Input Pin Description.

Pin Name: multiplicand_a[5:0]	Source → Destination: Multiplier input → DSP-based Half Multiplier
Pin Function: Arbitrary 6-bit input.	
Pin Name: multiplicand_b[5:0]	Source → Destination: Multiplier input → DSP-based Half Multiplier

Pin Function: Arbitrary 6-bit input.	
Pin Name: multiplier_const[11:0]	Source → Destination: Multiplier input → DSP-based Half Multiplier
Pin Function: Arbitrary 12-bit shared twiddle factor input.	

Table 4.18 DSP-based Half Multiplier Output Pin Description.

Pin Name: produce_a_reg[17:0]	Source → Destination: DSP-based Half Multiplier → Multiplier Internal Logic
Pin Function: 18-bit output of multiplicand_a × multiplier_const.	
Pin Name: produce_b_reg[17:0]	Source → Destination: DSP-based Half Multiplier → Multiplier Internal Logic
Pin Function: 18-bit output of multiplicand_b × multiplier_const.	

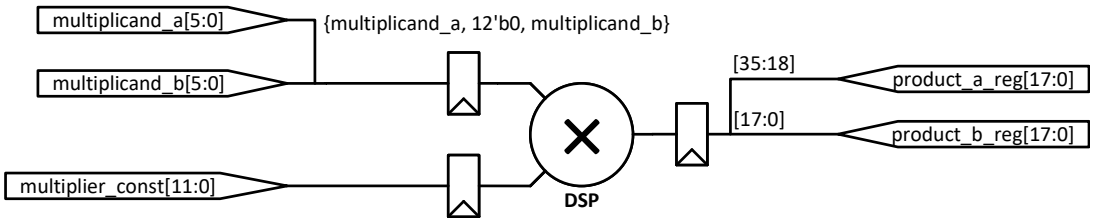


Figure 4.17 DSP-based Half Multiplier Schematic.

4.2.10 Modular Reducer: mult_red.sv

The Modular Reducer reduces any arbitrary 24-bit multiplication output of two 12-bit operands back into the 12-bit range of $[0, q)$.

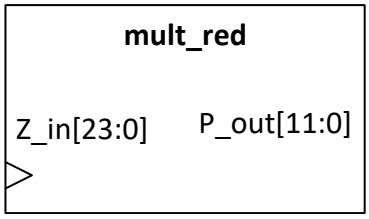


Figure 4.18 Modular Reducer.

Table 4.19 Modular Reducer Input Pin Description.

Pin Name:	Source → Destination:
Z_in[23:0]	BFU input → Modular Reducer
Pin Function: Arbitrary unsigned 24-bit input for modular reduction.	

Table 4.20 Modular Reducer Output Pin Description.

Pin Name:	Source → Destination:
P_out[11:0]	Modular Reducer → BFU Internal Logic
Pin Function: Unsigned 12-bit result output Z_in $\pmod{q}$.	

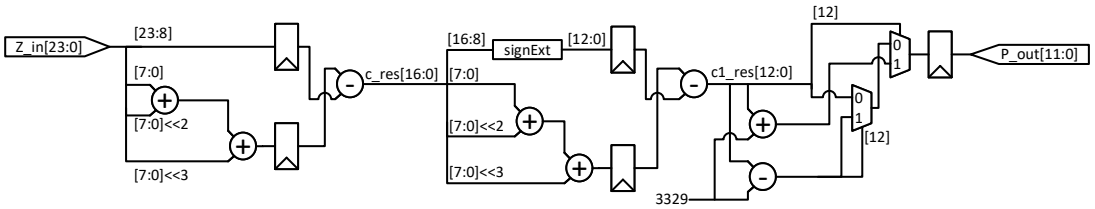


Figure 4.19 Modular Reducer Schematic.

4.2.11 Butterfly Unit (BFU): bfu_extmult.sv

The BFU performs CT or GS butterfly operation on received operands depending on received operation mode. It delegates the multiplication of twiddle factors to the external Multiplier module.

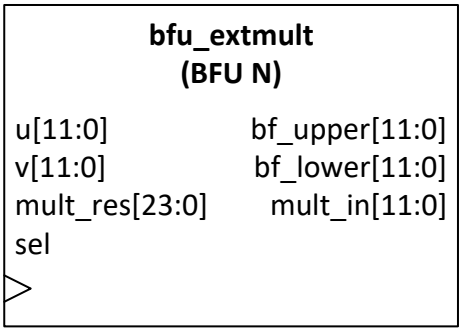


Figure 4.20 BFU.

Table 4.21 BFU Input Pin Description.

Pin Name: u[11:0]	Source → Destination: System Glue Logic → BFU
Pin Function: Arbitrary unsigned 12-bit input for butterfly operation upper result.	
Pin Name: v[11:0]	Source → Destination: System Glue Logic → BFU
Pin Function: Arbitrary unsigned 12-bit input for butterfly operation lower result.	
Pin Name: mult_res[23:0]	Source → Destination: Multiplier → BFU
Pin Function: Unsigned 24-bit input of delegated twiddle factor multiplication result.	
Pin Name: sel	Source → Destination: Control Unit → Butterfly Units
Pin Function: Indicates forward (0) or inverse (1) NTT for CT or GS butterfly mode, respectively.	

Table 4.22 BFU Output Pin Description.

Pin Name: bf_upper[11:0]	Source → Destination: BFU → System Glue Logic
Pin Function: Upper result of butterfly operation.	
Pin Name: bf_lower[11:0]	Source → Destination: BFU → System Glue Logic
Pin Function: Lower result of butterfly operation.	
Pin Name: mult_in[11:0]	Source → Destination: BFU → Multiplier

Pin Function: Delegated 12-bit operand for multiplication with twiddle factor at external Multiplier input.

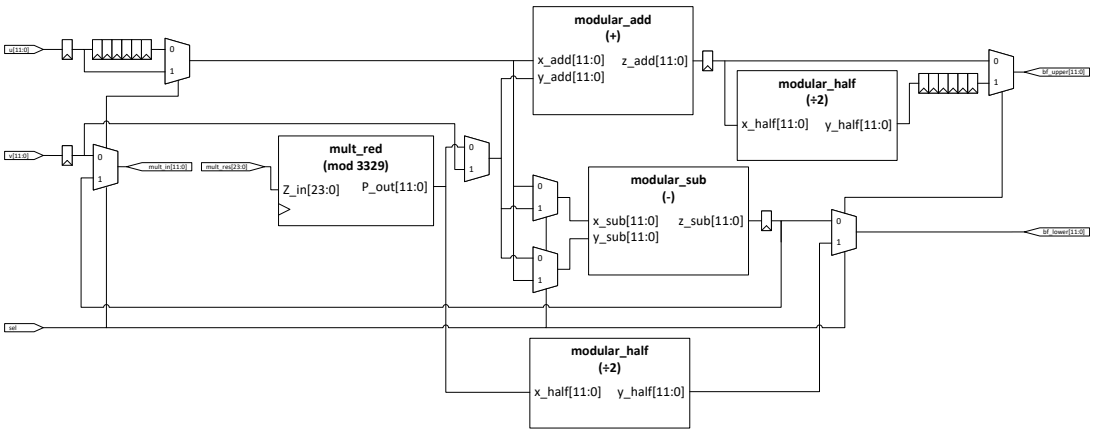


Figure 4.21 BFU Schematic.

4.2.12 Modular Adder: modular_add.sv

The Modular Adder adds two arbitrary 12-bit operands and reduces the result back into the 12-bit range of $[0, q)$.

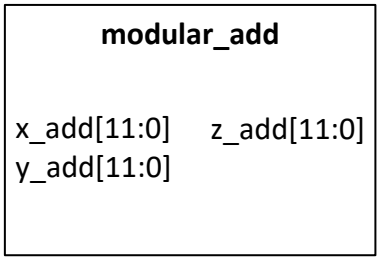


Figure 4.22 Modular Adder.

Table 4.23 Modular Adder Input Pin Description.

Pin Name: x_add[11:0]	Source → Destination: BFU Internal Logic → Modular Adder
Pin Function: Arbitrary unsigned 12-bit input.	

Pin Name:	Source → Destination:
y_add[11:0]	BFU Internal Logic → Modular Adder
Pin Function: Arbitrary unsigned 12-bit input.	

Table 4.24 Modular Adder Output Pin Description.

Pin Name:	Source → Destination:
z_sub[11:0]	Modular Adder → BFU Internal Logic
Pin Function: Unsigned 12-bit result output x_half + y_half $\pmod{q}$.	

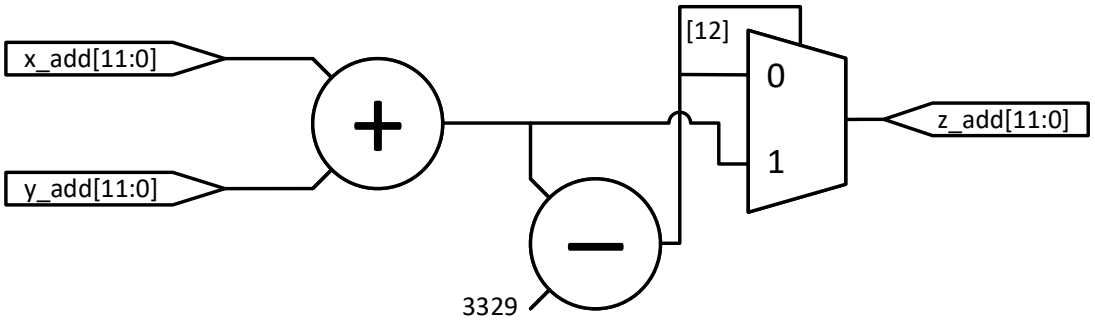


Figure 4.23 Modular Adder Schematic.

4.2.13 Modular Subtractor: modular_sub.sv

The Modular Subtractor subtracts one arbitrary 12-bit operand from another and reduces the result back into the 12-bit range of $[0, q)$.

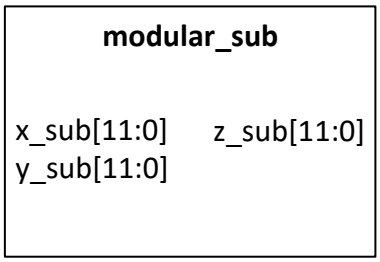


Figure 4.24 Modular Subtractor.

Table 4.25 Modular Subtractor Input Pin Description.

Pin Name:	Source → Destination:
x_sub[11:0]	BFU Internal Logic → Modular Subtractor
Pin Function: Arbitrary unsigned 12-bit input.	
Pin Name:	Source → Destination:
y_sub[11:0]	BFU Internal Logic → Modular Subtractor
Pin Function: Arbitrary unsigned 12-bit input.	

Table 4.26 Modular Subtractor Output Pin Description.

Pin Name:	Source → Destination:
z_sub[11:0]	Modular Subtractor → BFU Internal Logic
Pin Function: Unsigned 12-bit result output x_half − y_half $\pmod{q}$.	

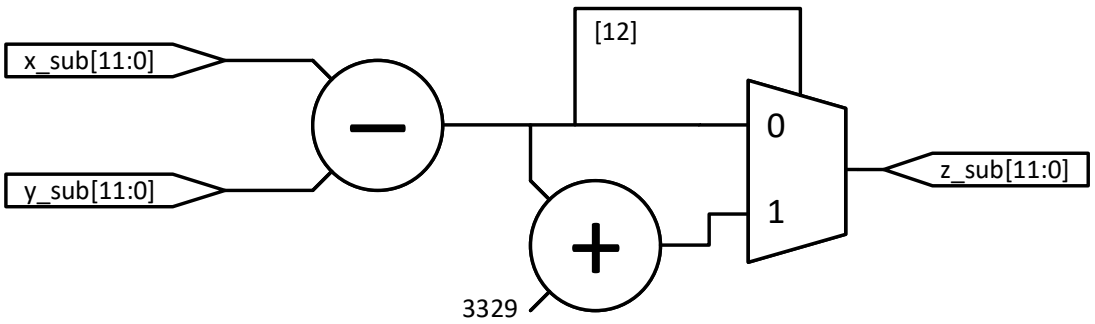


Figure 4.25 Modular Subtractor Schematic.

4.2.14 Modular Halver: modular_half.sv

The Modular Halver divides any arbitrary 12-bit input and reduces the result back into the 12-bit range of $[0, q)$.

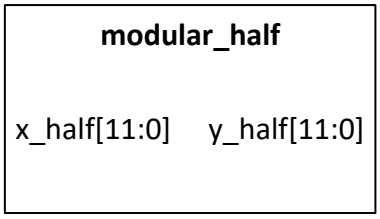


Figure 4.26 Modular Halver.

Table 4.27 Modular Halver Input Pin Description.

Pin Name: x_half[11:0]	Source → Destination: BFU Internal Logic → Modular Halver
Pin Function: Arbitrary unsigned 12-bit input.	

Table 4.28 Modular Halver Output Pin Description.

Pin Name: y_half[11:0]	Source → Destination: Modular Halver → BFU Internal Logic
Pin Function: Halved unsigned 12-bit result output x_half $\div 2 \pmod{q}$.	

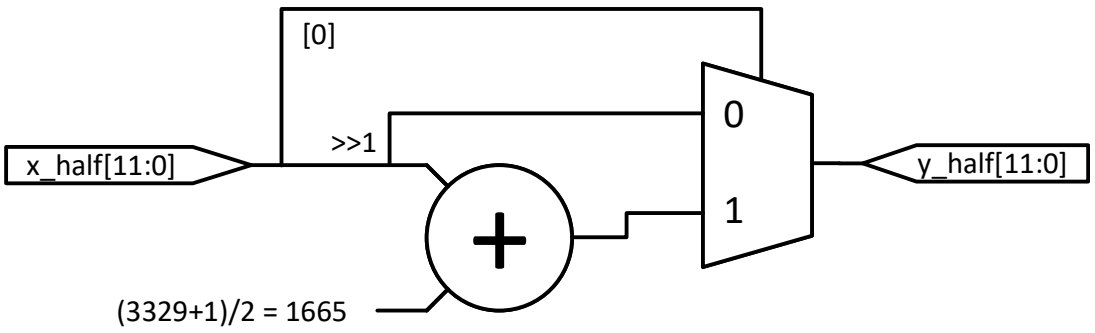


Figure 4.27 Modular Halver Schematic.

4.3 System Components Interaction Operations

The NTT/INTT operation in this CFNTT-based architecture [2] is as follows, referring to Figures 3.1 and 4.1. The control unit is a finite state machine that accepts operation mode (e.g. NTT, DONE_NTT) as input and produces NTT/INTT status flags as output to allow

an external system to control the NTT/INTT process. During NTT or INTT mode, the control unit runs multiple co-dependent increment- or decrement-by-one counters that are interpreted by the coefficient and twiddle factor address generators to produce the corresponding memory bank addresses. At the same time, it also produces an internal NTT mode signal to indicate forward or inverse NTT mode to the BFUs. The memory map submodule extracts and passes through the read/write addresses of the butterfly operand pairs u (for upper output) and v (for lower output) from the address generator to the memory banks, and it also determines whether the operand pair addresses should be swapped with each other. In this work, we follow a radix-2 configuration, so it is determined by a bitwise XOR on the addresses produced by the address generator. The swap control signal is forwarded by the arbiter to ensure that each swap decision is maintained throughout the pipeline from reading the BFU inputs to writing the BFU outputs in-place to the same memory locations. Finally, when the done flag is raised based on the counters and the external system puts the control unit into DONE_NTT or DONE_INTT mode, the control unit switches off the bank read/write enable controls while maintaining the internal NTT mode signaling to allow the pipeline to complete correctly.

CHAPTER 5

Simulation

5.1 Simulation Software Setup

We use Vivado 2025.2 to perform behavioral simulation and post-implementation timing simulation on Artix-7 model xc7a35tfgg484-3; synthesis and implementation strategies are left at default settings. A timing constraint is added for synthesis and implementation at 4.4ns period for all three versions of system design.

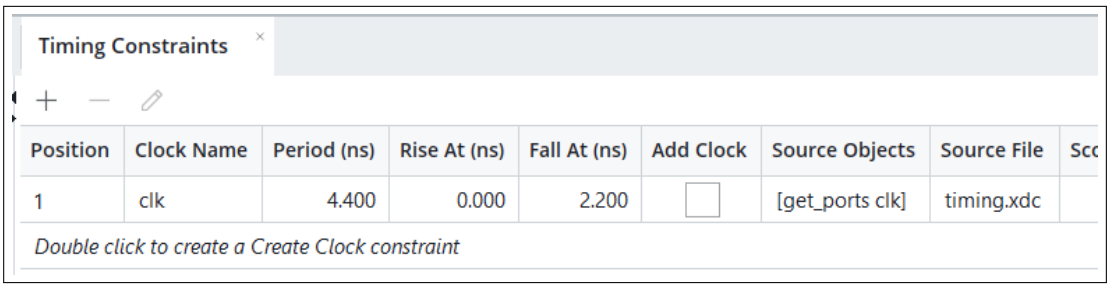


Figure 5.1 Timing constraint settings used across all three system design versions.

5.2 System Operation

The simulation testbenches use the same clock speed as the timing constraint at 4.4ns period. The simulation flow is the same for both behavioral and post-implementation timing simulations:

1. Data banks 0 to 3 are initialized with sample coefficients $0, 1, 2, \ldots, 255$ at coefficient locations $0, 1, 2, \ldots, 255$.
2. System performs NTT on the initial coefficients.
3. System performs INTT on the result of NTT.

After running the above steps, we expect the coefficients to remain the same as initial values. A validation is also to be performed after NTT, just before INTT, to ensure that all coefficients in NTT domain are correct as well. The validations are done automatically in behavioral simulation testbench script, but for post-implementation timing simulation, with no available direct access to memory contents, validation is done by visually

comparing the waveform of written data at memory banks with that of the successful behavioral simulation run.

5.3 Concluding Remark

The simulation flow is simple but effectively tests the system as any logical error that occurs throughout the recursive NTT/INTT 7-stage process that intensively mutates data will be reflected at the results during validation.

CHAPTER 6

System Evaluation and Discussion

6.1 System Testing and Performance Metrics

Referring to Table 6.1, the equivalent number of slices (ENS) is calculated through the formula $\text{LUT}/4 + \text{FF}/8 + \text{BRAM} \times 200 + \text{DSP} \times 100$, which is also found in other prior work [4], [25]. For the calculation of ATP, we provide two metrics (ATP-T and ATP-CC) to accommodate different reporting styles in the literature. ATP-CC considers clock cycles as the time variable, whereas ATP-T use the actual time in calculation, whereby the frequency also comes into consideration.

$$\text{ATP-T} = \text{ENS} \times \text{NTT Time } (\mu\text{s}) \tag{1}$$

$$\text{ATP-CC} = \text{ENS} \times \text{NTT Clock Cycles} \tag{2}$$

6.2 Testing Setup and Result

The behavioral simulation performs NTT and then INTT on an arbitrary set of test coefficients to verify the correctness of NTT/INTT using test vectors and direct memory access, and the activity is reproduced in a post-implementation timing simulation to reverify logic correctness in waveform compared to that of behavioral simulation, and estimate power consumption as shown in Table 6.2.

A load/store protocol is not implemented in the NTT/INTT module, since it should depend on the arbitrary requirements of a surrounding system; however, it may hypothetically be done by simply reusing the control unit counters and CFNTT [2] memory mapping algorithm to access the memory data in an organized manner, while pipelining the operation with the NTT/INTT process. Therefore, for an experimental measurement, we only initialize the memory banks with coefficient values as simulation begins and then count the NTT/INTT clock cycles from system idle until the last data has been written back to memory.

6.2.1 Result Screenshots for System Version 1: 2 DSP

Name	Used
top_poly_mul	233
compact_bf_0 (compact_bf__1)	93
Leaf Cells (55)	55
mult_pe0 (modular_mul__1)	38
compact_bf_1 (compact_bf)	91
Leaf Cells (57)	57
mult_pe0 (modular_mul)	34
Leaf Cells (49)	49

Figure 6.1 Slice Logic Distribution for Version 1: 2 DSP.

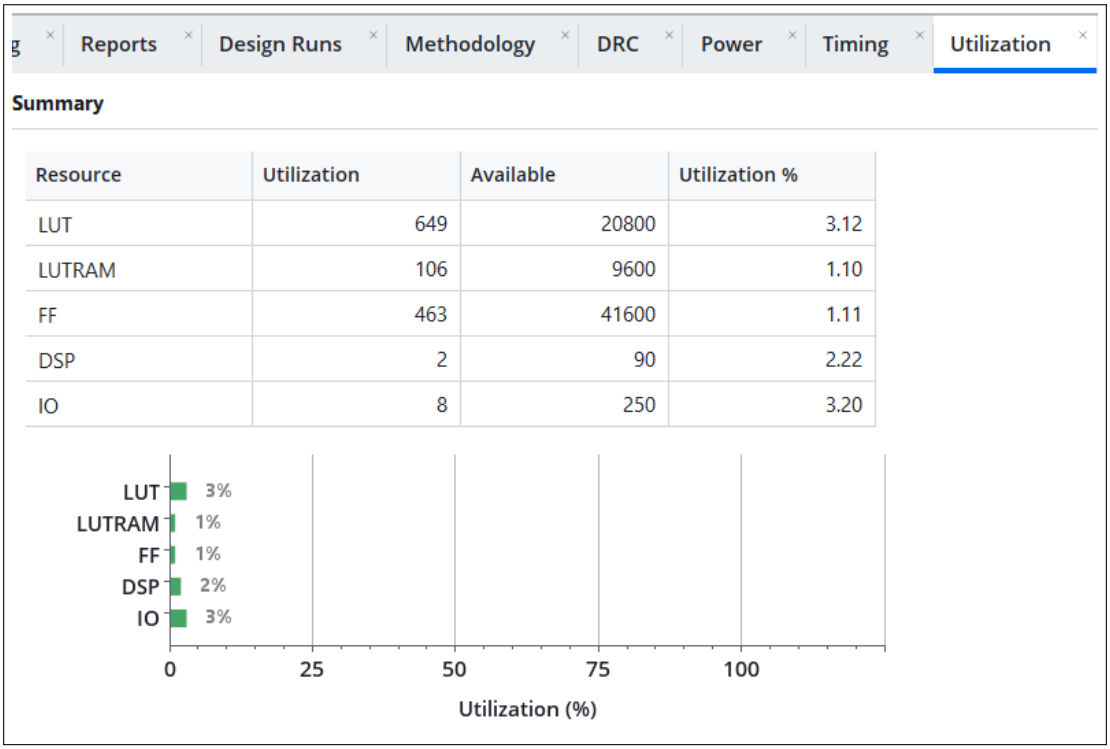


Figure 6.2 Utilization Summary Report for Version 1: 2 DSP.

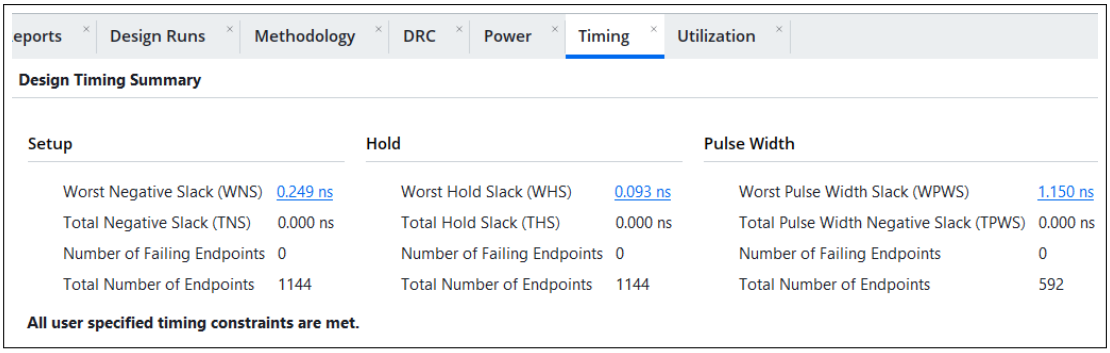


Figure 6.3 Design Timing Summary for Version 1: 2 DSP.

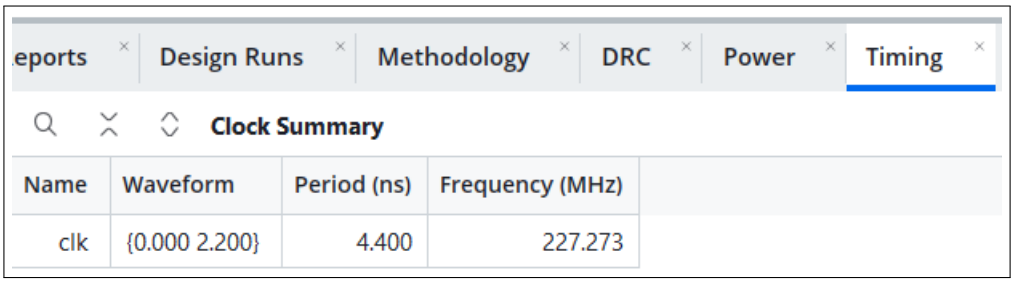


Figure 6.4 Clock Summary for Version 1: 2 DSP.

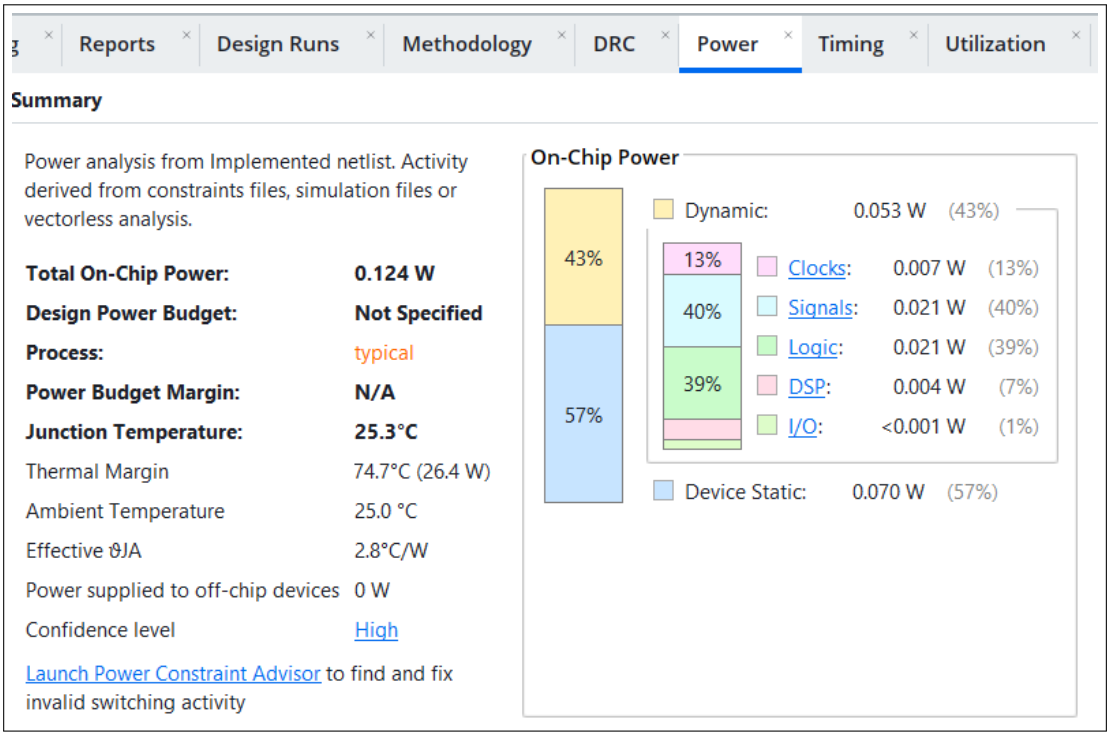


Figure 6.5 Power Summary for Version 1: 2 DSP.

```
Completed NTT [Time: 4475000]

Pipeline idle:
bank 0:'{2429,624,2725,2707,922,2319,306,818,304,2619,1479,102,1603,1732,2276,2187,2986,1482,1057,1225,1376,2934,2
bank 1:'{425,1865,2483,1488,1971,1075,1380,1155,2874,2169,2864,1679,681,517,2024,2637,2373,1157,1124,400,2664,1960
bank 2:'{795,1356,2197,2194,803,606,2718,531,155,2159,2014,3200,316,931,1094,2158,198,1290,1019,2206,614,1974,2217
bank 3:'{2845,31,2668,517,231,613,3143,1586,1442,1712,2634,1923,558,1999,2159,1973,247,449,2220,2233,2880,2679,323

-----------------RESULTS-----------------------
Bank 0 coefficients in NTT domain matched testvectors [PASS]
Bank 1 coefficients in NTT domain matched testvectors [PASS]
Bank 2 coefficients in NTT domain matched testvectors [PASS]
Bank 3 coefficients in NTT domain matched testvectors [PASS]
-------------------------------------------------

Completed INTT [Time: 9095000]

Pipeline idle:
bank 0:'{0,6,10,12,18,20,24,30,34,36,40,46,48,54,58,60,66,68,72,78,80,86,90,92,96,102,106,108,114,116,120,126,130,
bank 1:'{2,4,8,14,16,22,26,28,32,38,42,44,50,52,56,62,64,70,74,76,82,84,88,94,98,100,104,110,112,118,122,124,128,1
bank 2:'{3,5,9,15,17,23,27,29,33,39,43,45,51,53,57,63,65,71,75,77,83,85,89,95,99,101,105,111,113,119,123,125,129,1
bank 3:'{1,7,11,13,19,21,25,31,35,37,41,47,49,55,59,61,67,69,73,79,81,87,91,93,97,103,107,109,115,117,121,127,131,

-----------------RESULTS-----------------------
Bank 0 returned to original state [PASS]
Bank 1 returned to original state [PASS]
Bank 2 returned to original state [PASS]
Bank 3 returned to original state [PASS]
-------------------------------------------------
$stop called at time : 9175 ns : File "C:/Users/happy/Desktop/paper/design/1dsp/source/tb_top_behav.sv" Line 89
```

Figure 6.6 Successful Behavioral Simulation Console Output for Version 1: 2 DSP.

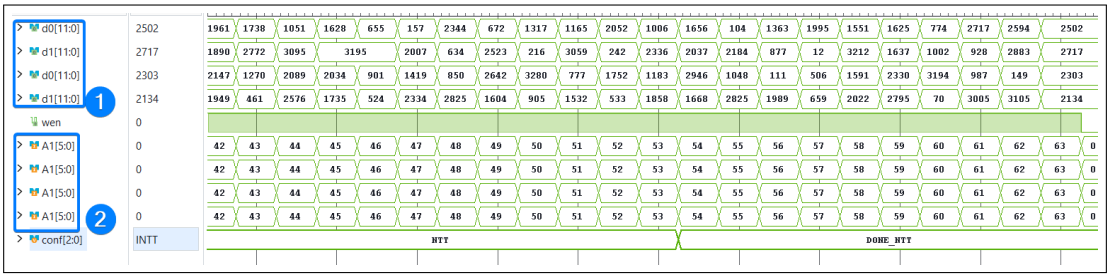


Figure 6.7 Behavioral Simulation Waveform (NTT) for Version 1: 2 DSP.

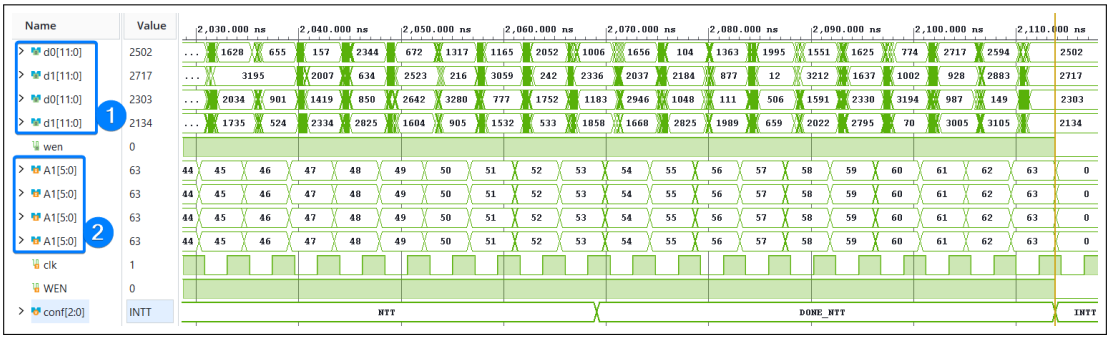


Figure 6.8 Post-Implementation Timing Simulation Waveform (NTT) for Version 1: 2 DSP.

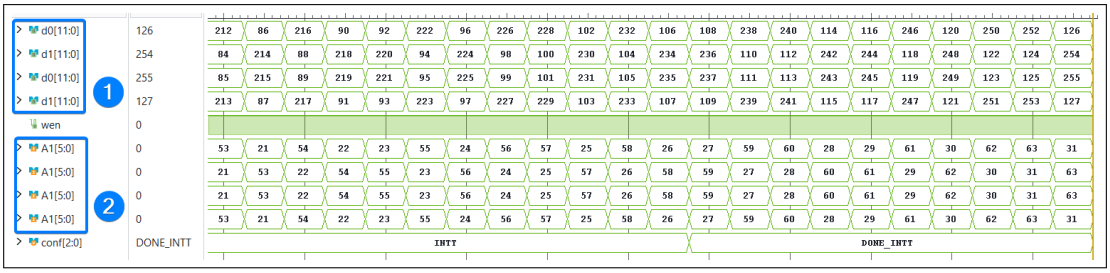


Figure 6.9 Behavioral Simulation Waveform (INTT) for Version 1: 2 DSP.

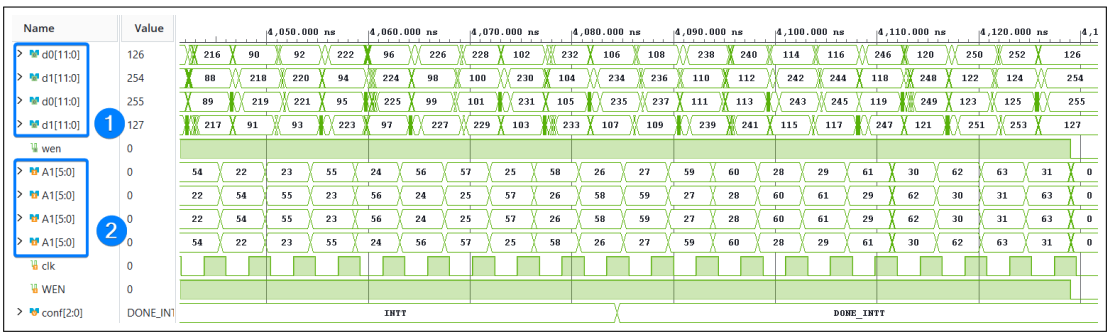


Figure 6.10 Post-Implementation Timing Simulation Waveform (INTT) for Version 1: 2 DSP.

Figures 6.7 to 6.10 show the waveform comparisons between behavioral simulation and post-implementation behavioral simulation. In the screenshots, signals marked '1' show the data to be written into the data banks, with the write enable signal visible below. Signals marked '2' show the write addresses for each data bank.

As the console output of behavioral simulation (Figure 6.6) confirms the logical correctness of the system, we visually compare the post-implementation timing simulation waveforms with the behavioral simulation waveforms to revalidate the system correctness, since we cannot directly access the data bank memory contents in the testbench script.

6.2.2 Result Screenshots for System Version 2: 1 DSP

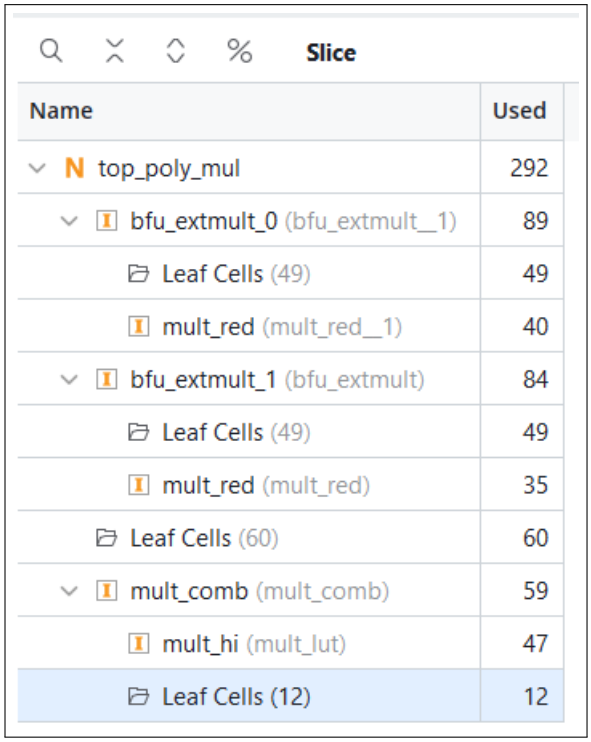


Figure 6.11 Slice Logic Distribution for Version 2: 1 DSP.

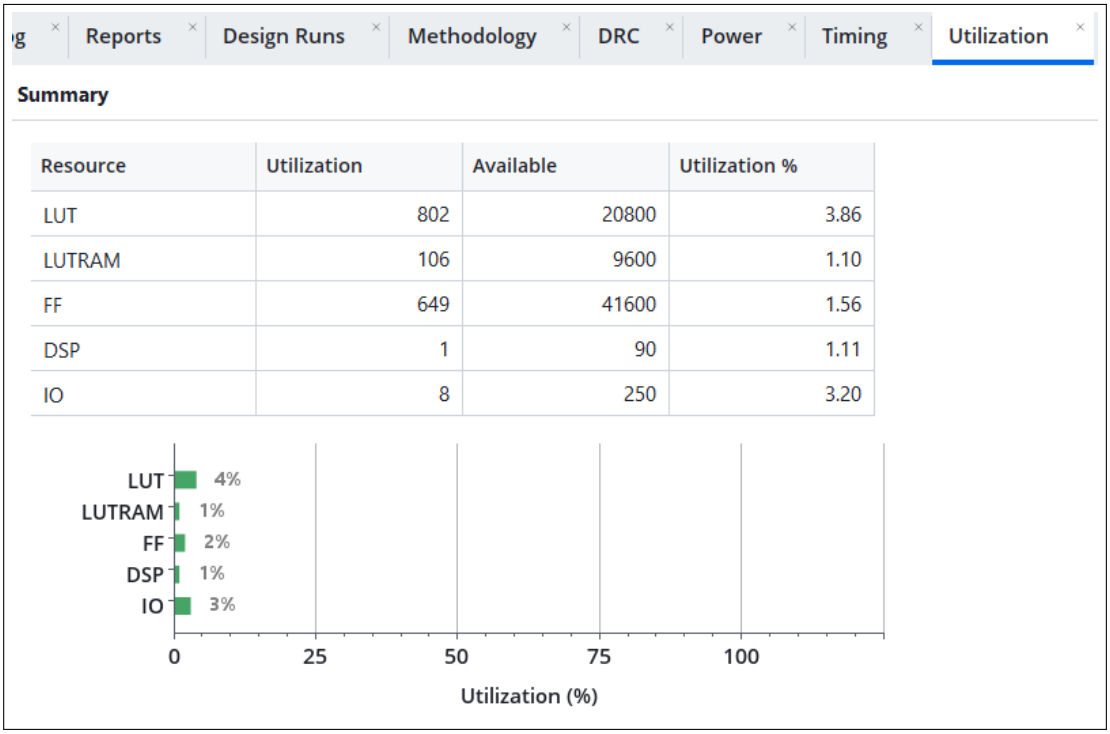


Figure 6.12 Utilization Summary Report for Version 2: 1 DSP.

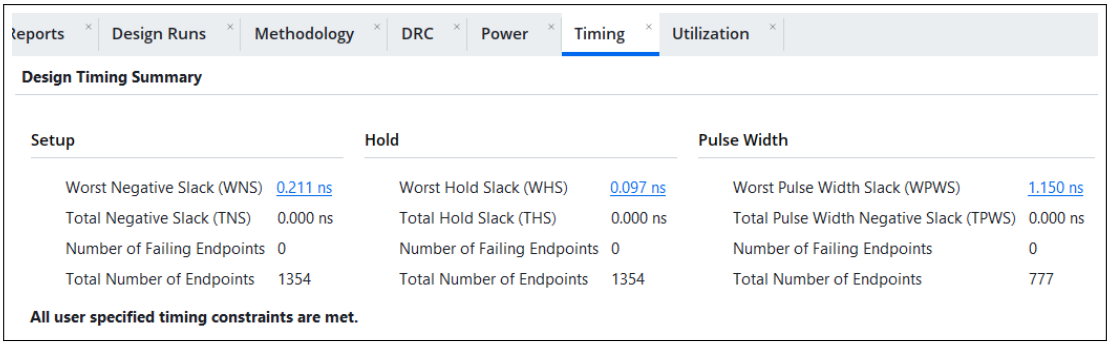


Figure 6.13 Design Timing Summary for Version 2: 1 DSP.

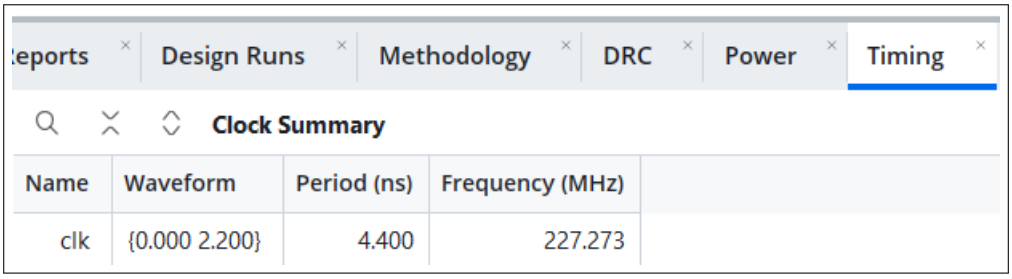


Figure 6.14 Clock Summary for Version 2: 1 DSP.

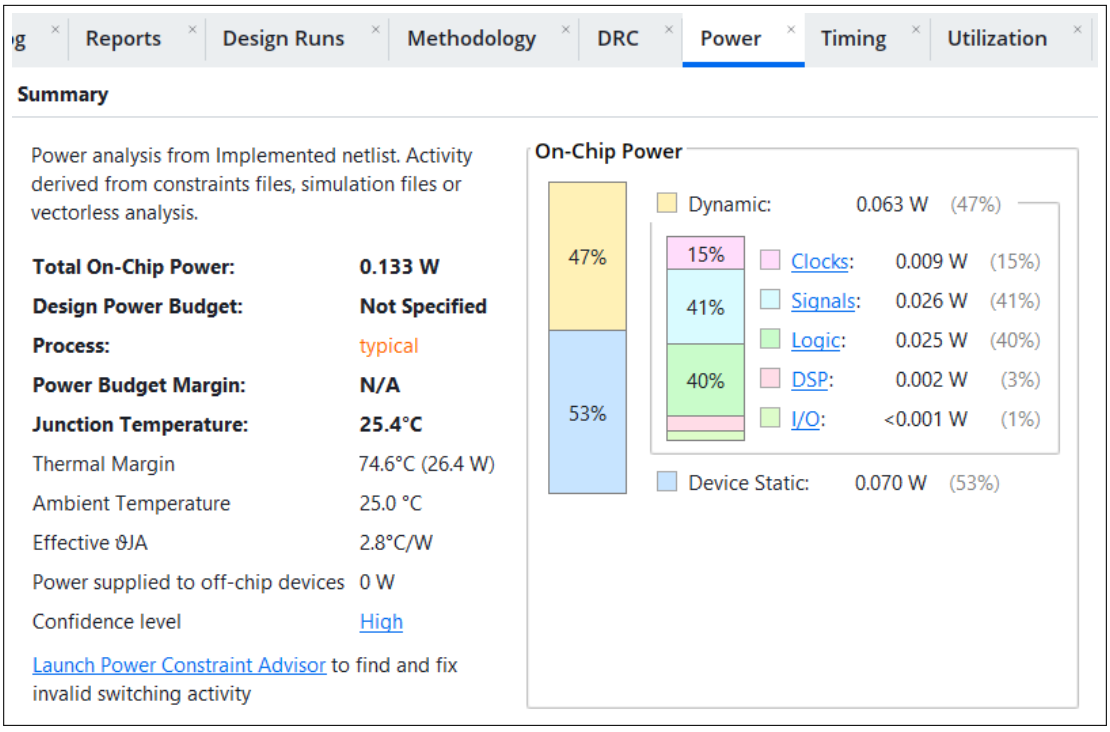


Figure 6.15 Power Summary for Version 2: 1 DSP.

```
Completed NTT [Time: 4475000]

Pipeline idle:
bank 0:'{2429,624,2725,2707,922,2319,306,818,304,2619,1479,102,1603,1732,2276,2187,2986,1482,1057,1225,1376,2934
bank 1:'{425,1865,2483,1488,1971,1075,1380,1155,2874,2169,2864,1679,681,517,2024,2637,2373,1157,1124,400,2664,19
bank 2:'{795,1356,2197,2194,803,606,2718,531,155,2159,2014,3200,316,931,1094,2158,198,1290,1019,2206,614,1974,22
bank 3:'{2845,31,2668,517,231,613,3143,1586,1442,1712,2634,1923,558,1999,2159,1973,247,449,2220,2233,2880,2679,3

-----------------RESULTS-----------------------
Bank 0 coefficients in NTT domain matched testvectors [PASS]
Bank 1 coefficients in NTT domain matched testvectors [PASS]
Bank 2 coefficients in NTT domain matched testvectors [PASS]
Bank 3 coefficients in NTT domain matched testvectors [PASS]
-------------------------------------------------

Completed INTT [Time: 9105000]

Pipeline idle:
bank 0:'{0,6,10,12,18,20,24,30,34,36,40,46,48,54,58,60,66,68,72,78,80,86,90,92,96,102,106,108,114,116,120,126,13
bank 1:'{2,4,8,14,16,22,26,28,32,38,42,44,50,52,56,62,64,70,74,76,82,84,88,94,98,100,104,110,112,118,122,124,128
bank 2:'{3,5,9,15,17,23,27,29,33,39,43,45,51,53,57,63,65,71,75,77,83,85,89,95,99,101,105,111,113,119,123,125,129
bank 3:'{1,7,11,13,19,21,25,31,35,37,41,47,49,55,59,61,67,69,73,79,81,87,91,93,97,103,107,109,115,117,121,127,13

-----------------RESULTS-----------------------
Bank 0 returned to original state [PASS]
Bank 1 returned to original state [PASS]
Bank 2 returned to original state [PASS]
Bank 3 returned to original state [PASS]
-------------------------------------------------
$stop called at time : 9195 ns : File "C:/Users/happy/Desktop/paper/design/1dsp/source/tb_top_behav.sv" Line 89
```

Figure 6.16 Successful Behavioral Simulation Console Output for Version 2: 1 DSP.

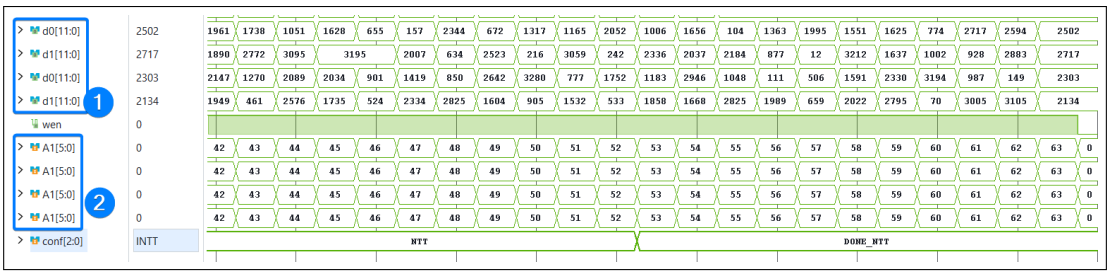


Figure 6.17 Behavioral Simulation Waveform (NTT) for Version 2: 1 DSP.

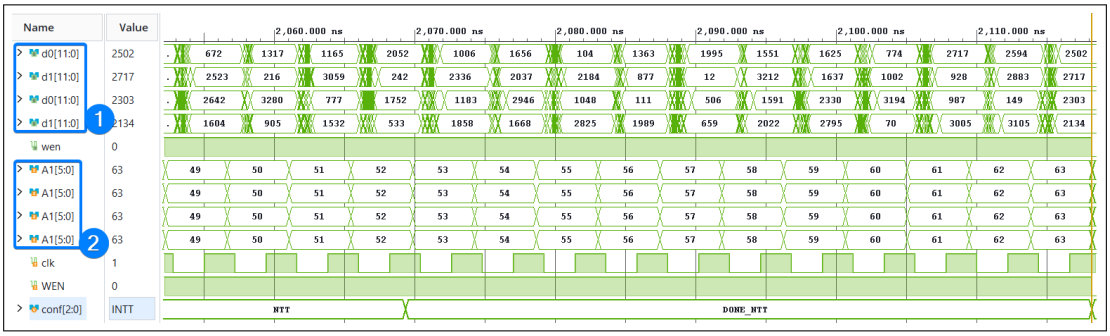


Figure 6.18 Post-Implementation Timing Simulation Waveform (NTT) for Version 2: 1 DSP.

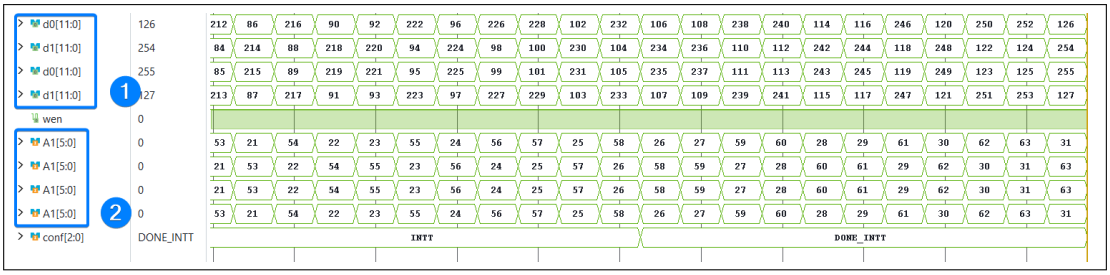


Figure 6.19 Behavioral Simulation Waveform (INTT) for Version 2: 1 DSP.

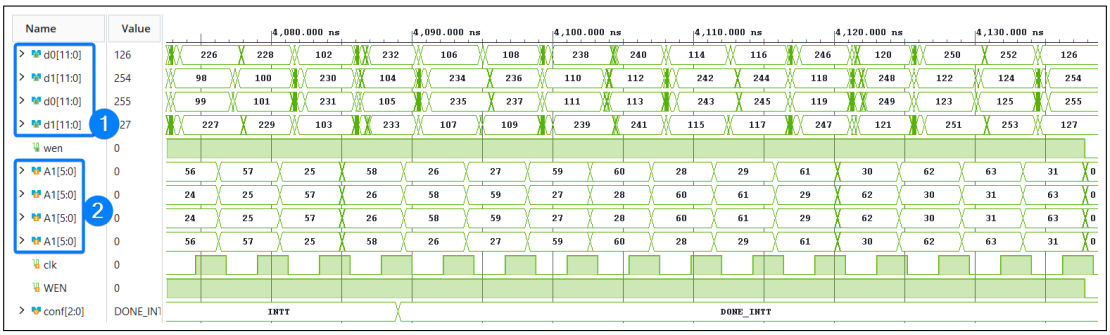


Figure 6.20 Post-Implementation Timing Simulation Waveform (INTT) for Version 2: 1 DSP.

Figures 6.17 to 6.20 show the waveform comparisons between behavioral simulation and post-implementation behavioral simulation. In the screenshots, signals marked '1' show the data to be written into the data banks, with the write enable signal visible below. Signals marked '2' show the write addresses for each data bank.

As the console output of behavioral simulation (Figure 6.16) confirms the logical correctness of the system, we visually compare the post-implementation timing simulation waveforms with the behavioral simulation waveforms to revalidate the system correctness, since we cannot directly access the data bank memory contents in the testbench script.

6.2.3 Result Screenshots for System Version 3: 0 DSP

Slice

Name	Used
top_poly_mul	326
mult_comb (mult_comb)	112
mult_hi (mult_lut__1)	54
mult_lo (mult_lut)	48
Leaf Cells (10)	10
bfu_extmult_0 (bfu_extmult__1)	95
Leaf Cells (56)	56
mult_red (mult_red__1)	39
bfu_extmult_1 (bfu_extmult)	85
Leaf Cells (48)	48
mult_red (mult_red)	37
Leaf Cells (34)	34

Figure 6.21 Slice Logic Distribution for Version 3: 0 DSP.

Reports | Design Runs | Methodology | DRC | Power | Timing | Utilization

Summary

Resource	Utilization	Available	Utilization %
LUT	940	20800	4.52
LUTRAM	106	9600	1.10
FF	790	41600	1.90
IO	8	250	3.20

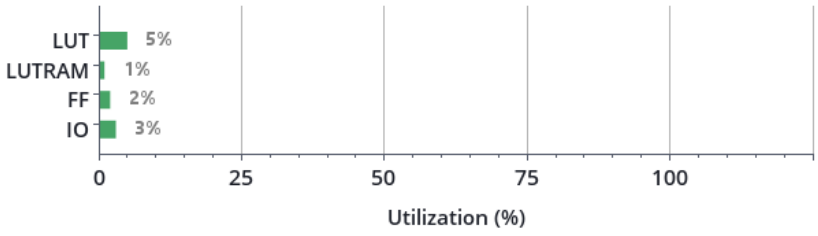


Figure 6.22 Utilization Summary Report for Version 3: 0 DSP.

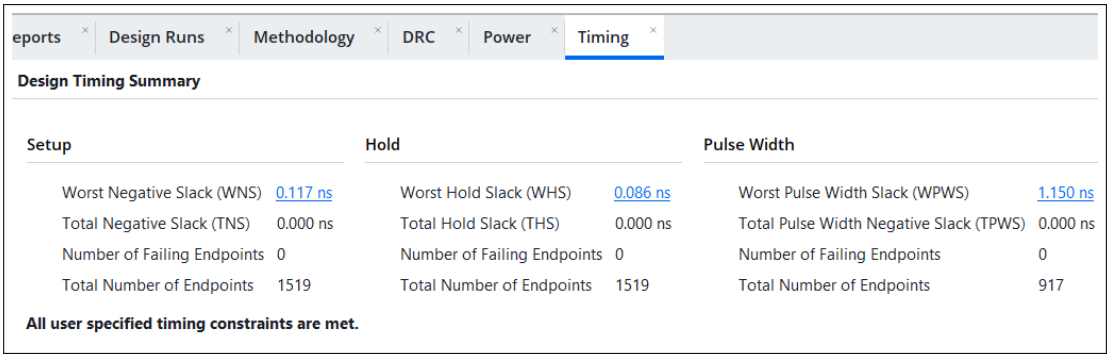


Figure 6.23 Design Timing Summary for Version 3: 0 DSP.

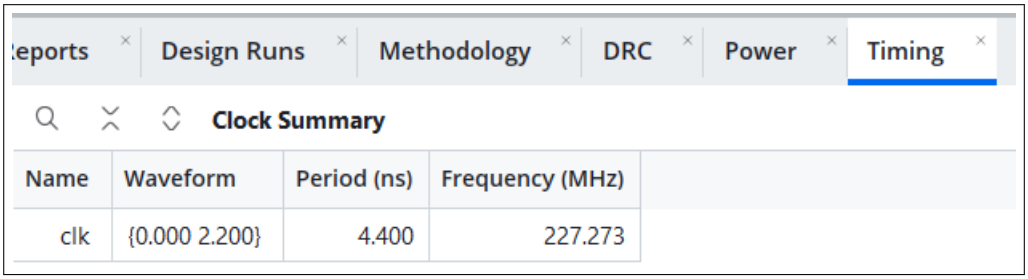


Figure 6.24 Clock Summary for Version 3: 0 DSP.

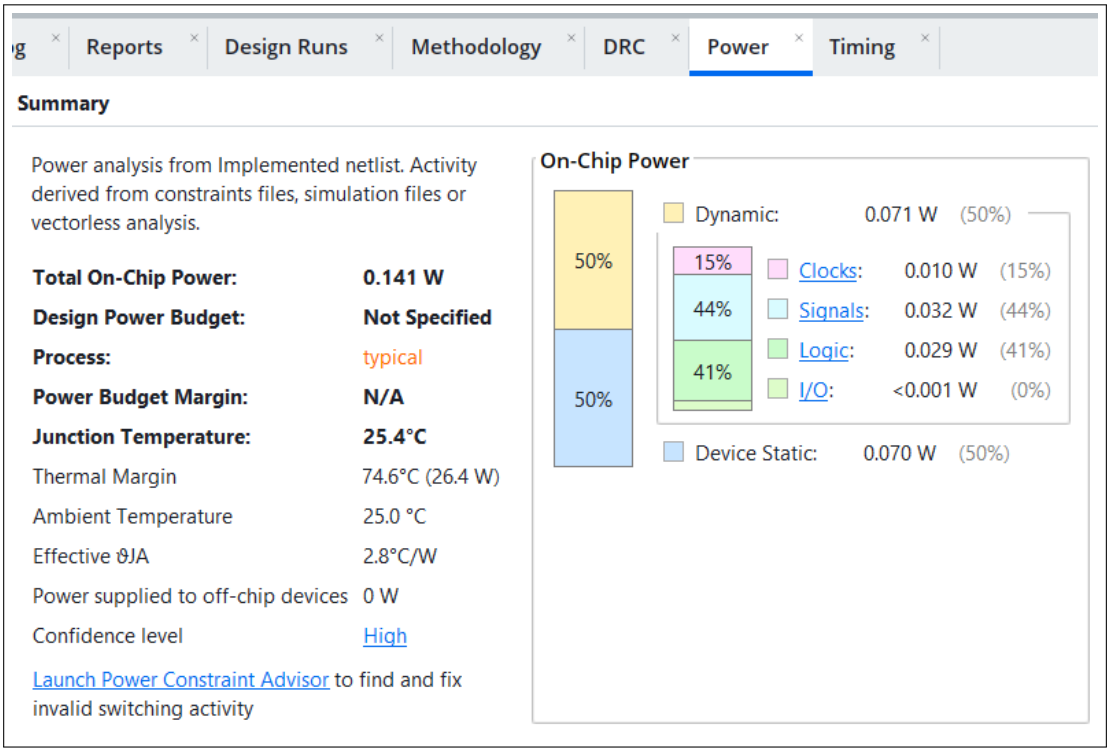


Figure 6.25 Power Summary for Version 3: 0 DSP.

```
Completed NTT [Time: 4475000]

Pipeline idle:
bank 0:'{2429,624,2725,2707,922,2319,306,818,304,2619,1479,102,1603,1732,2276,2187,2986,1482,1057,1225,1376,2934,2
bank 1:'{425,1865,2483,1488,1971,1075,1380,1155,2874,2169,2864,1679,681,517,2024,2637,2373,1157,1124,400,2664,1960
bank 2:'{795,1356,2197,2194,803,606,2718,531,155,2159,2014,3200,316,931,1094,2158,198,1290,1019,2206,614,1974,2217
bank 3:'{2845,31,2668,517,231,613,3143,1586,1442,1712,2634,1923,558,1999,2159,1973,247,449,2220,2233,2880,2679,323

----------------RESULTS-----------------------
Bank 0 coefficients in NTT domain matched testvectors [PASS]
Bank 1 coefficients in NTT domain matched testvectors [PASS]
Bank 2 coefficients in NTT domain matched testvectors [PASS]
Bank 3 coefficients in NTT domain matched testvectors [PASS]
------------------------------------------------

Completed INTT [Time: 9105000]

Pipeline idle:
bank 0:'{0,6,10,12,18,20,24,30,34,36,40,46,48,54,58,60,66,68,72,78,80,86,90,92,96,102,106,108,114,116,120,126,130,
bank 1:'{2,4,8,14,16,22,26,28,32,38,42,44,50,52,56,62,64,70,74,76,82,84,88,94,98,100,104,110,112,118,122,124,128,1
bank 2:'{3,5,9,15,17,23,27,29,33,39,43,45,51,53,57,63,65,71,75,77,83,85,89,95,99,101,105,111,113,119,123,125,129,1
bank 3:'{1,7,11,13,19,21,25,31,35,37,41,47,49,55,59,61,67,69,73,79,81,87,91,93,97,103,107,109,115,117,121,127,131,

----------------RESULTS-----------------------
Bank 0 returned to original state [PASS]
Bank 1 returned to original state [PASS]
Bank 2 returned to original state [PASS]
Bank 3 returned to original state [PASS]
------------------------------------------------
$stop called at time : 9195 ns : File "C:/Users/happy/Desktop/paper/design/0dsp/source/tb_top_behav.sv" Line 89
```

Figure 6.26 Successful Behavioral Simulation Console Output for Version 3: 0 DSP.

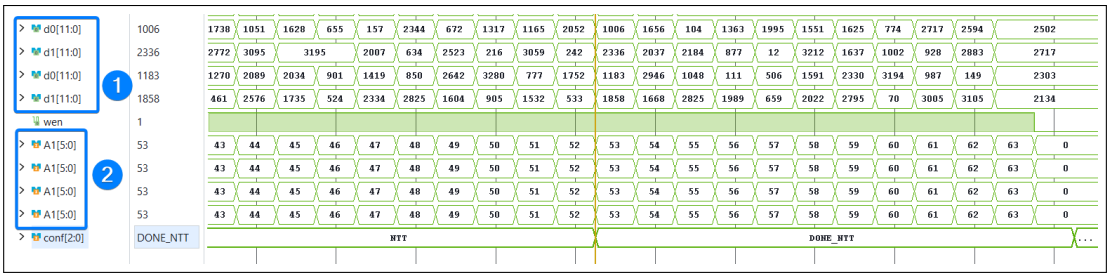


Figure 6.27 Behavioral Simulation Waveform (NTT) for Version 3: 0 DSP.

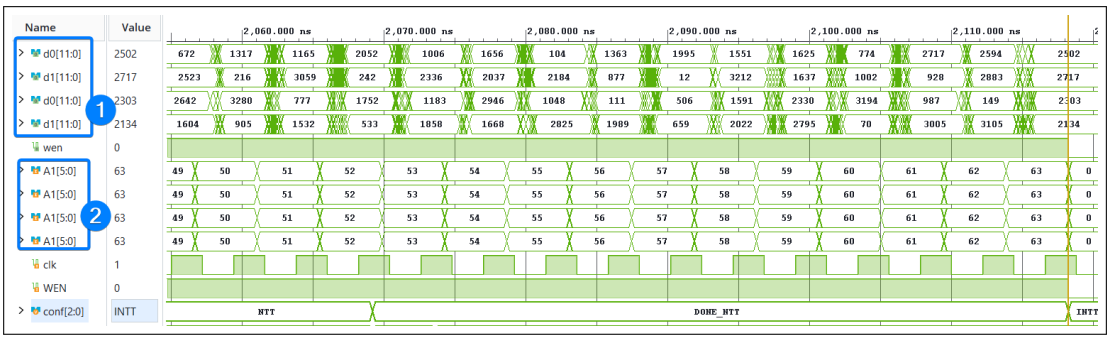


Figure 6.28 Post-Implementation Timing Simulation Waveform (NTT) for Version 3: 0 DSP.

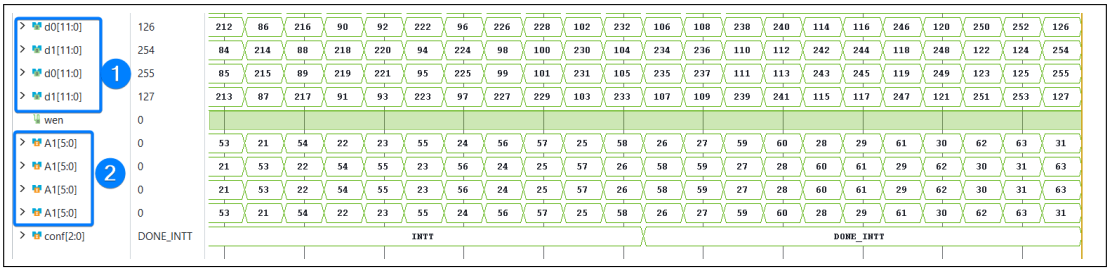


Figure 6.29 Behavioral Simulation Waveform (INTT) for Version 3: 0 DSP.

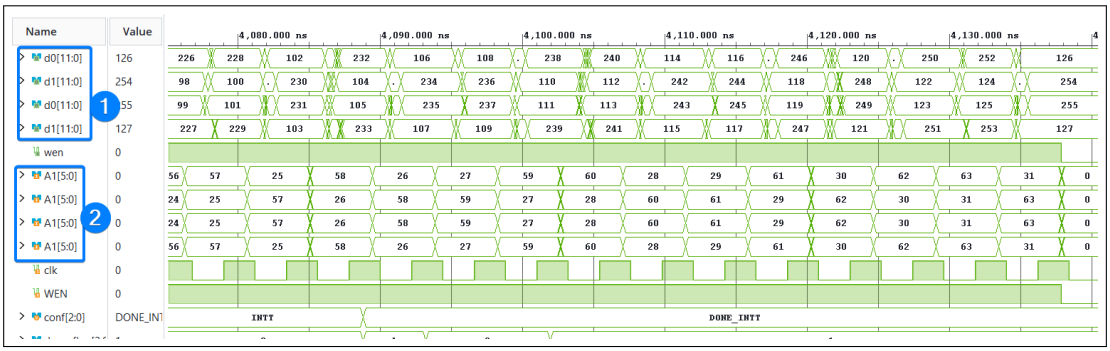


Figure 6.30 Post-Implementation Timing Simulation Waveform (INTT) for Version 3: 0 DSP.

Figures 6.27 to 6.30 show the waveform comparisons between behavioral simulation and post-implementation behavioral simulation. In the screenshots, signals marked '1' show the data to be written into the data banks, with the write enable signal visible below. Signals marked '2' show the write addresses for each data bank.

As the console output of behavioral simulation (Figure 6.26) confirms the logical correctness of the system, we visually compare the post-implementation timing simulation waveforms with the behavioral simulation waveforms to revalidate the system correctness, since we cannot directly access the data bank memory contents in the testbench script.

6.2.4 Result Tabulation

Table 6.1 Comparison With Other Works

Year	Work	NTT Cycles	INTT Cycles	Slice	LUT	FF	BRAM	DSP	ENS	Frequency (MHz)	Time (NTT; μs)	ATP-T	ATP-CC
2021	[5]	324	324	312	801	717	2	4	1089.875	222	1.459	1590.628	353.120
2023	[16]	225	225	1054	1740	643	1*	4	1115.375	200	1.125	1254.797	250.959
2023	[22]	313	313	259	784	441	1*	2	651.125	200	1.565	1019.011	203.802
2025	[4]	448	448	319	1005	599	0	2	526.125	227	1.974	1038.344	235.704
2025	[26]	3584	3584	N/A	557	59	0	0	146.625	100	35.840	5255.040	525.504
2025	[25]; $K = 2^2$	2016	2016	N/A	676	633	0	0	248.125	307	6.567	1629.381	500.220
2025	[25]; $K = 2^3$	1009	1009	N/A	1137	1176	0	0	431.25	306	3.297	1421.998	435.131
2025	[25]; $K = 2^4$	504	504	N/A	2081	2237	0	0	799.875	304	1.658	1326.109	403.137
2026	This Work; 2 DSP	458	458	233	649	463	0	2	420.125	227	2.018	847.653	192.417
2026	This Work; 1 DSP	459	459	292	802	649	0	1	381.625	227	2.022	771.656	175.166
2026	This Work; 0 DSP	459	459	326	940	790	0	0	333.75	227	2.022	674.851	153.191

* We assume effective utilization of 1 BRAM for in-place NTT/INTT of only 256 coefficients, to compensate for the higher BRAM usage indicated for their entire ML-KEM architecture.

Table 6.2 Simulated Power Consumption

Version	DSP	Static (mW)	Dynamic (mW)	Total (mW)
1	2	70	53	123
2	1	70	63	133
3	0	70	71	141

6.3 Objectives Evaluation

The results in Table 6.1 show that our NTT/INTT architecture achieves significant improvements in both ATP-T and ATP-CC over other notable works. Looking at ATP-T, our Version 2 (1 DSP) is 24.3% lower than the lowest ATP results to date [22]. It is also 25.7% lower than that of the more recent state-of-the-art [4]. On top of that, our Version 3 with 2 LUT-based half multipliers achieves an even lower ATP-T of 33.8% and 35% compared to [22] and [4], respectively. This is because the ENS of our design is much lower than the others, despite requiring the most clock cycles. Even then, the actual process time used by our architecture remains within the range of other works with closer ENS.

6.4 Concluding Remark

Referring to Table 6.2, our proposed implementation utilizing two DSPs (version 1) is the most power efficient among all proposed architectures. This is mainly because the DSP units are built-in to the FPGA hardware with a more optimized power efficiency. On the other hand, although our implementation without any DSP (version 3) can achieve

the best ATP, it suffers from a 14.6% increase in dynamic power consumption compared to version 1. The shared-DSP multiplier (version 2) strikes a balance between these two options, in which the ATP is still reasonably low and the dynamic power consumption is just 8.1% higher than version 1. This demonstrates the scalability of the proposed NTT architecture, where the hardware area can be parametrically optimized to meet specific power and performance constraints.

CHAPTER 7

Conclusion and Recommendation

7.1 Conclusion

The proposed NTT/INTT module achieves a very low ATP, which is up to 35% better ATP compared to state-of-the-art [4]. This indicates that it is especially suitable for deployment in a constrained environment.

7.2 Recommendation

In the future, it is possible to develop the architecture of entire ML-KEM/Kyber scheme based on this NTT/INTT module, targeting different security levels.

REFERENCES

[1] National Institute of Standards and Technology (US), ''Module-lattice-based key-encapsulation mechanism standard,'' National Institute of Standards and Technology (U.S.), Washington, D.C., NIST FIPS 203, Aug. 13, 2024, NIST FIPS 203. doi: 10.6028/nist.fips.203. Accessed: Apr. 18, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf

[2] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, ''CFNTT: Scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme,'' *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 94–126, Nov. 19, 2021, ISSN: 2569-2925. doi: 10.46586/tches.v2022.i1.94-126. Accessed: Apr. 18, 2025. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9291

[3] X. Ji et al., ''ECO-CRYSTALS: Efficient cryptography CRYSTALS on standard RISC-V ISA,'' *IEEE Transactions on Computers*, vol. 74, no. 2, pp. 401–413, Feb. 2025, ISSN: 1557-9956. doi: 10.1109/tc.2024.3483631. Accessed: Apr. 26, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10723802/

[4] T.-H. Nguyen, D.-T. Dam, P.-P. Duong, B. Kieu-Do-Nguyen, C.-K. Pham, and T.-T. Hoang, ''Efficient hardware implementation of the lightweight CRYSTALS-Kyber,'' *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 72, no. 2, pp. 610–622, Feb. 2025, ISSN: 1558-0806. doi: 10.1109/tcsi.2024.3443238. Accessed: Apr. 22, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10642971/

[5] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, ''High-speed NTT-based polynomial multiplication accelerator for post-quantum cryptography,'' in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, ISSN: 2576-2265, Jun. 2021, pp. 94–101. doi: 10.1109/arith51176.2021.00028. Accessed: Apr. 20, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9603378/

[6] Z. Ye, R. C. C. Cheung, and K. Huang, ''PipeNTT: A pipelined number theoretic transform architecture,'' *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 10, pp. 4068–4072, Oct. 2022, ISSN: 1549-7747, 1558-3791. doi: 10.1109/tcsii.2022.3184703. Accessed: Apr. 22, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9802731/

[7] X. Feng, S. Li, and S. Xu, ''RLWE-oriented high-speed polynomial multiplier utilizing multi-lane Stockham NTT algorithm,'' *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 3, pp. 556–559, Mar. 2020, ISSN: 1558-3791. doi: 10.1109/tcsii.2019.2917621. Accessed: Apr. 29, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8717615/

[8] D. D. Chen et al., *High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems*, Cryptology ePrint Archive, Paper 2014/646,

2014. doi: 10.1109/tcsi.2014.2350431. [Online]. Available: https://eprint.iacr.org/2014/646

[9] P. L. Montgomery, ''Modular multiplication without trial division,'' *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985, ISSN: 0025-5718, 1088-6842. doi: 10.1090/s0025-5718-1985-0777282-x. Accessed: Apr. 23, 2025. [Online]. Available: https://www.ams.org/mcom/1985-44-170/S0025-5718-1985-0777282-X/

[10] P. Barrett, ''Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor,'' in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed., Berlin, Heidelberg: Springer, 1987, pp. 311–323, ISBN: 978-3-540-47721-1. doi: 10.1007/3-540-47721-7_24.

[11] R. Avanzi et al. ''CRYSTALS-Kyber algorithm specifications and supporting documentation,'' Accessed: Apr. 20, 2025. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms

[12] T. Plantard, ''Efficient word size modular arithmetic,'' *IEEE Transactions on Emerging Topics in Computing*, vol. 19, pp. 1506–1518, 3 Apr. 26, 2021. Accessed: Apr. 28, 2025. [Online]. Available: https://thomas-plantard.github.io/pdf/Plantard21.pdf

[13] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography* (Discrete Mathematics and Its Applications). CRC Press, 2018, ISBN: 978-0-429-88132-9. [Online]. Available: https://cacr.uwaterloo.ca/hac/

[14] P. Longa and M. Naehrig, *Speeding up the number theoretic transform for faster ideal lattice-based cryptography*, Cryptology ePrint Archive, Paper 2016/504, 2016. [Online]. Available: https://eprint.iacr.org/2016/504

[15] Y. Xing and S. Li, ''A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA,'' *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 328–356, Feb. 23, 2021, ISSN: 2569-2925. doi: 10.46586/tches.v2021.i2.328-356. Accessed: Apr. 26, 2025. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8797

[16] W. Guo and S. Li, ''Highly-efficient hardware architecture for CRYSTALS-Kyber with a novel conflict-free memory access pattern,'' *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 11, pp. 4505–4515, Nov. 2023, ISSN: 1549-8328, 1558-0806. doi: 10.1109/tcsi.2023.3306347. Accessed: Apr. 18, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10231113/

[17] J. Huang et al., ''Improved Plantard arithmetic for lattice-based cryptography,'' *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 4, 2022. [Online]. Available: https://eprint.iacr.org/2022/956

[18] J. Huang et al., ''Yet another improvement of plantard arithmetic for faster Kyber on low-end 32-bit IoT devices,'' *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 3800–3813, 2024, ISSN: 1556-6021. doi: 10.1109/tifs.2024.3371369. Accessed: Apr. 27, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10453274/

[19] P. Duong-Ngoc and H. Lee, ''Configurable mixed-radix number theoretic transform architecture for lattice-based cryptography,'' *IEEE Access*, vol. 10, pp. 12 732–12 741, 2022, ISSN: 2169-3536. doi: 10.1109/access.2022.3145988. Accessed: Apr. 30, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9690849/

[20] S. Mandal and D. Basu Roy, ''Winograd for NTT: A case study on higher-radix and low-latency implementation of NTT for post quantum cryptography on FPGA,'' *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 12, pp. 6396–6409, Dec. 2024, ISSN: 1558-0806. doi: 10.1109/tcsi.2024.3470335. Accessed: Apr. 30, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10711850/

[21] J. Sun and X. Bai, ''A high-speed hardware architecture of an NTT accelerator for CRYSTALS-Kyber,'' *Integrated Circuits and Systems*, vol. 1, no. 2, pp. 92–102, May 2024, ISSN: 2995-1976. doi: 10.23919/ics.2024.3419562. Accessed: Apr. 22, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10572273/

[22] W. Guo and S. Li, ''Split-radix based compact hardware architecture for CRYSTALS-Kyber,'' *IEEE Transactions on Computers*, vol. 73, no. 1, pp. 97–108, Jan. 2023, ISSN: 1557-9956. doi: 10.1109/tc.2023.3320040. Accessed: May 4, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10264919

[23] E. Chu and A. George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Boca Raton: CRC Press, Nov. 11, 1999, 336 pp., ISBN: 978-0-367-80233-2. doi: 10.1201/9780367802332.

[24] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, ''Instruction-set accelerated implementation of CRYSTALS-Kyber,'' *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4648–4659, Nov. 2021, ISSN: 1558-0806. doi: 10.1109/tcsi.2021.3106639. Accessed: Apr. 20, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9525069/

[25] K. Javeed and D. Gregg, ''Efficient number theoretic transform architecture for crystals-kyber,'' *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 72, no. 1, pp. 263–267, 2025.

[26] M. O. Alshomrany et al., ''Lightweight fpga-based number-theoretic transform for ml-kem in post-quantum cryptography,'' *IEEE Access*, vol. 13, pp. 203 620–203 632, 2025.

Appendix A Source Code

Source Code A.1 Definition header (define.sv).

```
`define N_NUM 256
`define PRIME 3329
`define DATA_WIDTH 12
`define KYBER

`define N_BIT $clog2(`N_NUM)
`define BIT_PRIME $clog2(`PRIME)
`define K_NUM `N_NUM/2
`define K_BIT $clog2(`K_NUM)
`define P_BIT $clog2(`K_BIT)
`define BANK_NUM 4
`define BANK_SIZE `N_NUM/`BANK_NUM
`define BANK_ADDR_BIT $clog2(`BANK_SIZE)

`ifndef TYPEDEFS
`define TYPEDEFS
package define;
    typedef enum logic [2:0] {
        IDLE = 3'b000,
        NTT = 3'b001,
        PWM = 3'b010,
        INTT = 3'b011,
        DONE_NTT = 3'b100,
        DONE_INTT = 3'b101
    } states_e;
endpackage
`endif
```

Source Code A.2 Top level module (top_poly_mul.sv).

```
`timescale 1ns / 1ps
`include "define.sv"

(*DONT_TOUCH = "true"*)
module top_poly_mul
  import define::*;
#(
    parameter int ROM_ADDR_W = `K_BIT,
    int ADDR_W = `BANK_ADDR_BIT,
```

```
    int W = `DATA_WIDTH
) (
    input clk,
    rst,
    input states_e conf,
    output logic [2:0] done_flag
);

  //fsm port signal
  logic [ADDR_W-1:0] k;
  logic [ADDR_W-1:0] i;
  logic [`P_BIT-1:0] p;
  logic sel;
  logic wen, ren, en;

  //address_generator port signal
  logic [ADDR_W:0] old_address_0, old_address_1;

  //memory map port signal --- new_address
  logic [ADDR_W-1:0] b0, b1;
  //memory map port signal --- bank_number
  logic a0, a1;

  //arbiter port signal
  logic sel_a_0;

  //write address for bank
  logic [ADDR_W-1:0] new_address_0_reg, new_address_1_reg;

  //read address for bank
  logic [ADDR_W-1:0] new_address_0, new_address_1;

  //data from bank
  logic [W-1:0] q0, q1, q2, q3;

  //data for bfu
  logic [W-1:0] u0, v0, u1, v1;
  //data from bfu
  logic [W-1:0] bf_0_upper, bf_0_lower, bf_1_upper, bf_1_lower;

  //data for bank
```

```
logic [W-1:0] d0, d1, d2, d3;

//twiddle factor from ROM
logic [W-1:0] w;

//tf address for ROM
logic [ROM_ADDR_W-1:0] tf_address;


(*DONT_TOUCH = "true"*)
fsm fsm (
    .clk(clk),
    .rst(rst),
    .conf(conf),
    .sel(sel),
    .k(k),
    .i(i),
    .p(p),
    .wen(wen),
    .ren(ren),
    .en(en),
    .done_flag(done_flag)
);

(*DONT_TOUCH = "true"*)
address_generator address_generator_0 (
    .k(k),
    .i(i),
    .p(p),
    .old_address_0(old_address_0),
    .old_address_1(old_address_1)
);

(*DONT_TOUCH = "true"*)
memory_map memory_map_0 (
    .clk(clk),
    .old_address_0(old_address_0),
    .old_address_1(old_address_1),
    .new_address_0(b0),
    .new_address_1(b1),
    .bank_number_0(a0),
```

```
    .bank_number_1(a1)
);

(*DONT_TOUCH = "true"*)
arbiter arbiter_0 (
    .a0(a0),
    .sel_a_0(sel_a_0)
);

(*DONT_TOUCH = "true"*)
network_bank_in network_bank_in_0 (
    .b0(b0),
    .b1(b1),
    .sel_a_0(sel_a_0),
    .new_address_0(new_address_0),
    .new_address_1(new_address_1)
);

shift_9_norst #(
    .W(ADDR_W)
) dff_n0 (
    .clk (clk),
    .din (new_address_0),
    .dout(new_address_0_reg)
);
shift_9_norst #(
    .W(ADDR_W)
) dff_n1 (
    .clk (clk),
    .din (new_address_1),
    .dout(new_address_1_reg)
);

(*DONT_TOUCH = "true"*)
data_bank_0 bank_0 (
    .clk(clk),
    .A1 (new_address_0_reg),  //write address
    .A2 (new_address_0),
    .D  (d0),
    .WEN(wen),
    .REN(ren),
```

```
    .EN (en),
    .Q  (q0)
);

(*DONT_TOUCH = "true"*)
data_bank_1 bank_1 (
    .clk(clk),
    .A1 (new_address_1_reg),
    .A2 (new_address_1),
    .D  (d1),
    .WEN(wen),
    .REN(ren),
    .EN (en),
    .Q  (q1)
);

(*DONT_TOUCH = "true"*)
data_bank_2 bank_2 (
    .clk(clk),
    .A1 (new_address_1_reg),  //write address
    .A2 (new_address_1),
    .D  (d2),
    .WEN(wen),
    .REN(ren),
    .EN (en),
    .Q  (q2)
);

(*DONT_TOUCH = "true"*)
data_bank_3 bank_3 (
    .clk(clk),
    .A1 (new_address_0_reg),
    .A2 (new_address_0),
    .D  (d3),
    .WEN(wen),
    .REN(ren),
    .EN (en),
    .Q  (q3)
);

(*DONT_TOUCH = "true"*)
```

```
network_bf_in network_bf_in_0 (
    .clk(clk),
    .rst(rst),
    .sel_a_0(sel_a_0),
    .q0(q0),
    .q1(q1),
    .u0(u0),
    .v0(v0)
);

(*DONT_TOUCH = "true"*)
network_bf_in_odd network_bf_in_1 (
    .clk(clk),
    .rst(rst),
    .sel_a_0(sel_a_0),
    .q0(q2),
    .q1(q3),
    .u0(u1),
    .v0(v1)
);

logic [2*W-1:0] mult_res_0, mult_res_1;
logic [W-1:0] mult_in_0, mult_in_1, w_q1, w_q2;

always_ff @(posedge clk) begin
  w_q1 <= w;
  w_q2 <= w_q1;
end

(*DONT_TOUCH = "true"*)
mult_comb mult_comb (
    .clk             (clk),
    .multiplicand_a  (mult_in_0),
    .multiplicand_b  (mult_in_1),
    .multiplier_const(sel ? w_q2 : w_q1),
    .product_a_reg   (mult_res_0),
    .product_b_reg   (mult_res_1)
);

(*DONT_TOUCH = "true"*)
bfu_extmult bfu_extmult_0 (
```

```
    .clk(clk),
    .u(u0),
    .v(v0),
    .sel(sel),
    .mult_res(mult_res_0),
    .mult_in(mult_in_0),
    .bf_upper(bf_0_upper),
    .bf_lower(bf_0_lower)
);

(*DONT_TOUCH = "true"*)
bfu_extmult bfu_extmult_1 (
    .clk(clk),
    .u(u1),
    .v(v1),
    .sel(sel),
    .mult_res(mult_res_1),
    .mult_in(mult_in_1),
    .bf_upper(bf_1_upper),
    .bf_lower(bf_1_lower)
);


(*DONT_TOUCH = "true"*)
network_bf_out network_bf_out_0 (
    .clk(clk),
    .bf_0_upper(bf_0_upper),
    .bf_0_lower(bf_0_lower),
    .sel_a_0(sel_a_0),
    .d0(d0),
    .d1(d1)
);

(*DONT_TOUCH = "true"*)
network_bf_out_odd network_bf_out_1 (
    .clk(clk),
    .bf_0_upper(bf_1_upper),
    .bf_0_lower(bf_1_lower),
    .sel_a_0(sel_a_0),
    .d0(d2),
    .d1(d3)
```

```
  );

  (*DONT_TOUCH = "true"*)
  tf_address_generator tf_address_generator_0 (
      .clk(clk),
      .conf(conf),
      .p(p),
      .k(k),
      .tf_address(tf_address)
  );


  (*DONT_TOUCH = "true"*)
  tf_ROM tf_rom_0 (
      .clk(clk),
      .A  (tf_address),
      .REN(ren),
      .Q  (w)
  );

endmodule
```

Source Code A.3 Address generator (address_generator.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module address_generator (
    input logic [`BANK_ADDR_BIT-1:0] k,
    input logic [`BANK_ADDR_BIT-1:0] i,
    input logic [`P_BIT-1:0] p,
    output logic [`BANK_ADDR_BIT:0] old_address_0,
    old_address_1
);

  logic [`BANK_ADDR_BIT:0] old_address_1_reg;

  assign old_address_1 = old_address_1_reg;

  assign old_address_0 = (({k, 1'b0}) << p) + i;  // 2k * 2^p + i

  always_comb begin
    old_address_1_reg = old_address_0;
```

```
    old_address_1_reg[p] = 1'b1;
  end

endmodule
```

Source Code A.4 Arbiter (arbiter.sv).

```
`timescale 1ns / 1ps
module arbiter (
    input  logic a0,
    output logic sel_a_0
);

  always_comb begin
    if (a0 == 0) sel_a_0 = 0;
    else sel_a_0 = 1;
  end

endmodule
```

Source Code A.5 BFU (bfu_extmult.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module bfu_extmult #(parameter int W = `DATA_WIDTH)(
    input clk,
    input sel,
    input [W-1:0] u,v,
    input [2*W-1:0] mult_res,
    output [W-1:0] bf_upper,bf_lower,
    output [W-1:0] mult_in
);

logic [W-1:0] u_q1,u_q5;
logic [W-1:0] mux_out1,mux_out2,mux_out3;
logic [W-1:0] mux_out5  ;
logic [W-1:0] v_q1      ;
logic [W-1:0] mult_out,mult_out_half,add_out,sub_out;
logic [W-1:0] add_out_q1,add_out_q1_half,sub_out_q1,add_out_q5;
logic [W-1:0] sub_op1,sub_op2;

always_ff @(posedge clk) begin
    u_q1       <= u;
```

```
    v_q1       <= v;
    sub_out_q1 <= sub_out;
    add_out_q1 <= add_out;
end
//mux about signal u
shift_6_norst #(.W(`DATA_WIDTH)) shf_u (.clk(clk), .din(u_q1),
↪  .dout(u_q5));
assign mux_out1 = sel == 0 ? u_q5 : u_q1;

//mux about signal v
assign mux_out3 = sel == 0 ? v_q1 : sub_out_q1;
assign mult_in  = mux_out3;
(*DONT_TOUCH="true"*)
mult_red mult_red (.clk(clk), .Z_in(mult_res), .P_out(mult_out));
assign mux_out2 = sel == 0 ? mult_out : v_q1;
assign sub_op1  = sel == 0 ? mux_out1 : mux_out2;
assign sub_op2  = sel == 0 ? mux_out2 : mux_out1;
(*DONT_TOUCH="true"*)
modular_sub #(.W(`DATA_WIDTH)) sub (.x_sub(sub_op1),
↪  .y_sub(sub_op2), .z_sub(sub_out));

(*DONT_TOUCH="true"*)
modular_add #(.W(`DATA_WIDTH)) add (
    .x_add(mux_out1),
    .y_add(mux_out2),
    .z_add(add_out )
);

(*DONT_TOUCH="true"*)
modular_half #(.W(`DATA_WIDTH)) half_lower (.x_half(add_out_q1),
↪  .y_half(add_out_q1_half));
shift_6_norst #(.W(`DATA_WIDTH)) shf_add (.clk(clk),
↪  .din(add_out_q1_half), .dout(add_out_q5));
(*DONT_TOUCH="true"*)
modular_half #(.W(`DATA_WIDTH)) half_upper (.x_half(mult_out),
↪  .y_half(mult_out_half));
assign bf_upper = sel == 0 ? add_out_q1 : add_out_q5;
assign bf_lower = sel == 0 ? sub_out_q1 : mult_out_half;
endmodule
```

Source Code A.6 Data bank (data_bank.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module data_bank #(
    parameter int ADDR_W = `BANK_ADDR_BIT,
    int W = `DATA_WIDTH,
    int SIZE = `BANK_SIZE
) (
    input logic clk,
    input logic [ADDR_W-1:0] A1,  //write address
    input logic [ADDR_W-1:0] A2,  //read address
    input logic [W-1:0] D,
    input logic WEN,
    input logic REN,
    input logic EN,
    output logic [W-1:0] Q
);
  (*ram_style = "distributed"*) logic [W-1:0] bank[SIZE];

  always_ff @(posedge clk) begin
    if (EN) begin
      if (WEN) bank[A1] <= D;
      else bank[A1] <= bank[A1];
    end
  end

  always_ff @(posedge clk) begin
    if (EN) begin
      if (REN) Q <= bank[A2];
      else Q <= Q;
    end
  end
endmodule
```

Source Code A.7 Control unit finite state machine (fsm.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module fsm
  import define::*;
(
    input logic clk,
```

```
    rst,
    input states_e conf,
    output logic sel,
    output logic [`BANK_ADDR_BIT-1:0] k,
    output logic [`BANK_ADDR_BIT-1:0] i,
    output logic [`P_BIT-1:0] p,
    output logic wen,
    output logic ren,
    output logic en,
    output logic [2:0] done_flag
);


  localparam logic [`BANK_ADDR_BIT-1:0] KEnd = (`K_NUM / 2) - 3;

  logic wen_reg, ren_reg, en_reg;
  logic sel_reg;
  logic en_reg_q, en_reg_q_tmp;
  states_e conf_state;
  logic [`BANK_ADDR_BIT-1:0] k_reg, i_reg;
  logic [`P_BIT-1:0] p_reg;
  logic [2:0] done_reg;
  logic [2:0] end_stage, begin_stage;

  assign i = i_reg;
  assign k = k_reg;
  assign p = p_reg;

  assign en = (conf == DONE_NTT || conf == DONE_INTT) ? en_reg_q :
  ↪  en_reg_q_tmp;

  shift_10 #(
      .W(1)
  ) shif_wen (
      .clk (clk),
      .rst (rst),
      .din (wen_reg),
      .dout(wen)
  );
  shift_9 #(
      .W(1)
```

```
) shif_en (
    .clk (clk),
    .rst (rst),
    .din (en_reg_q_tmp),
    .dout(en_reg_q)
);

always_ff @(posedge clk) begin
  done_flag <= done_reg;
  en_reg_q_tmp <= en_reg;
  ren <= ren_reg;
  sel <= sel_reg;
end

always_ff @(posedge clk) begin
  if (rst) conf_state <= IDLE;
  else conf_state <= conf;
end

always_comb begin
  sel_reg  = 0;
  en_reg   = 0;
  wen_reg  = 0;
  ren_reg  = 0;
  done_reg = 3'b0;
  case (conf_state)
    IDLE: begin
      sel_reg  = 0;
      en_reg   = 0;
      wen_reg  = 0;
      ren_reg  = 0;
      done_reg = 3'b0;
    end
    NTT: begin
      sel_reg = 0;
      en_reg  = 1;
      wen_reg = 1;
      ren_reg = 1;
      if ((p_reg == 0) && (k_reg == KEnd) && (i_reg == 0)) begin
        done_reg = 3'b001;
      end else begin
```

```
        done_reg = 3'b0;
      end
    end
    PWM: begin
      sel_reg = 0;
      en_reg  = 1;
      wen_reg = 1;
      ren_reg = 1;
      if ((p_reg == 0) && (k_reg == KEnd) && (i_reg == 0))
      ↪  done_reg = 3'b010;
      else done_reg = 3'b0;
    end
    INTT: begin
      sel_reg = 1;
      en_reg  = 1;
      wen_reg = 1;
      ren_reg = 1;
      if ((p_reg == `BANK_ADDR_BIT) && (k_reg == 0) && (i_reg ==
      ↪  `BANK_SIZE - 3))
        done_reg = 3'b100;
      else done_reg = 3'b0;
    end
    //NTT emptying pipeline phase
    DONE_NTT: begin
      sel_reg = 0;  //NTT = 0
      en_reg  = 0;
      wen_reg = 0;
      ren_reg = 0;
    end
    DONE_INTT: begin
      sel_reg  = 1;  //INTT = 1
      en_reg   = 0;
      wen_reg  = 0;
      ren_reg  = 0;
      done_reg = 3'b001;
    end
    default: ;
  endcase
end

assign end_stage   = (conf == NTT) ? 0 : `BANK_ADDR_BIT;
```

```
  assign begin_stage = (conf == NTT) ? `BANK_ADDR_BIT : 0;

  always_ff @(posedge clk) begin
    if (!rst && (conf_state == NTT || conf_state == PWM ||
    ↪  conf_state == INTT)) begin
      if (i_reg == (1 << p_reg) - 1) begin  // if i = 2^p - 1
        i_reg <= 0;  // RST i
        if (k_reg == (`BANK_SIZE >> p_reg) - 1) begin  // if k =
        ↪  (128 / 2^p) - 1
          k_reg <= 0;  // RST k
          if (p_reg == end_stage) p_reg <= begin_stage;  // RST p if
          ↪  p at end stage
          else begin
            if (conf_state == INTT) p_reg <= p_reg + 1;  // INC p
            ↪  (INTT)
            else p_reg <= p_reg - 1;  // DEC p (NTT)
          end
        end else k_reg <= k_reg + 1;  // INC k
      end else i_reg <= i_reg + 1;  // INC i
    end else begin  // RST ALL
      p_reg <= begin_stage;
      i_reg <= 0;
      k_reg <= 0;
    end
  end

endmodule
```

Source Code A.8 Memory map (memory_map.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module memory_map (
    input logic clk,
    input logic [`BANK_ADDR_BIT:0] old_address_0,
    input logic [`BANK_ADDR_BIT:0] old_address_1,
    output logic [`BANK_ADDR_BIT-1:0] new_address_0,
    output logic [`BANK_ADDR_BIT-1:0] new_address_1,
    output logic bank_number_0,
    output logic bank_number_1
);
```

```
  logic [`BANK_ADDR_BIT-1:0] new_address_0_tmp, new_address_1_tmp;
  logic bank_number_0_tmp, bank_number_1_tmp;

  assign bank_number_0_tmp = ^old_address_0;
  assign new_address_0_tmp = old_address_0[`BANK_ADDR_BIT:1];

  assign bank_number_1_tmp = ^old_address_1;
  assign new_address_1_tmp = old_address_1[`BANK_ADDR_BIT:1];

  always_ff @(posedge clk) begin
    bank_number_0 <= bank_number_0_tmp;
    bank_number_1 <= bank_number_1_tmp;
    new_address_0 <= new_address_0_tmp;
    new_address_1 <= new_address_1_tmp;
  end

endmodule
```

Source Code A.9 Modular adder (modular_add.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module modular_add #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] x_add,
    input logic [W-1:0] y_add,
    output logic [W-1:0] z_add
    );

    localparam logic [W-1:0] M = `PRIME;
    logic [W:0] s, s_corr;

    assign s = x_add + y_add;
    assign s_corr = s - M;
    assign z_add = s_corr[W]? s[W-1:0] : s_corr[W-1:0];

endmodule
```

Source Code A.10 Modular halver (modular_half.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module modular_half #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] x_half,
```

```
    output logic [W-1:0] y_half
    );

    localparam bit [W-1:0] M = `PRIME;
    localparam bit [W-1:0] MHalf = (M+1)/2;

    logic [W-1:0] x_sh;
    logic [W-1:0] s;

    assign x_sh = x_half >> 1;
    assign s = x_sh + MHalf;
    assign y_half = x_half[0] == 1? s : x_sh;
endmodule
```

Source Code A.11 Modular subtractor (modular_sub.sv).

```
`timescale 1ns / 1ps
`include "define.sv"

module modular_sub #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] x_sub,
    input logic [W-1:0] y_sub,
    output logic [W-1:0] z_sub
    );

    localparam logic [W-1:0] M = `PRIME;

    logic [W:0] d;
    logic [W-1:0] corr;

    assign d = x_sub - y_sub;
    assign corr = d[W-1:0] + M;
    assign z_sub = d[W]? corr : d[W-1:0];

endmodule
```

Source Code A.12 Multiplier (mult_comb.sv).

```
`timescale 1ns/1ns
module mult_comb (
    input  logic        clk             , // System clock
    input  logic [11:0] multiplicand_a  , // 12-bit Input A
    input  logic [11:0] multiplicand_b  , // 12-bit Input B
```

```
    input  logic [11:0] multiplier_const, // 12-bit Input CONST
    output logic [23:0] product_a_reg   , // 24-bit Output result
    output logic [23:0] product_b_reg     // 24-bit Output result
);

    logic [17:0] product_a_lo, product_a_hi;
    logic [17:0] product_b_lo, product_b_hi;
    logic [23:0] product_a   ;
    logic [23:0] product_b   ;

    (*DONT_TOUCH = "true"*)
    mult_lut mult_hi (
        .clk             (clk                ),
        .multiplicand_a  (multiplicand_a[11:6]),
        .multiplicand_b  (multiplicand_b[11:6]),
        .multiplier_const(multiplier_const    ),
        .product_a_reg   (product_a_hi        ),
        .product_b_reg   (product_b_hi        )
    );

    (*DONT_TOUCH = "true"*)
    mult_lut mult_lo (
        .clk             (clk                ),
        .multiplicand_a  (multiplicand_a[5:0]),
        .multiplicand_b  (multiplicand_b[5:0]),
        .multiplier_const(multiplier_const   ),
        .product_a_reg   (product_a_lo       ),
        .product_b_reg   (product_b_lo       )
    );

    always_comb begin
        product_a = {product_a_hi,6'b0} + {6'b0, product_a_lo};
        product_b = {product_b_hi,6'b0} + {6'b0, product_b_lo};
    end

    always_ff @(posedge clk) begin
        product_a_reg <= product_a;
        product_b_reg <= product_b;
    end
```

```
endmodule
```

Source Code A.13 LUT-based half multiplier (mult_lut.sv).

```
`timescale 1ns/1ns
module mult_lut (
    input  logic        clk             , // System clock
    input  logic [ 5:0] multiplicand_a  , // 6-bit Input A
    input  logic [ 5:0] multiplicand_b  , // 6-bit Input B
    input  logic [11:0] multiplier_const, // 12-bit Input CONST
    output logic [17:0] product_a_reg   , // 18-bit Output result
    output logic [17:0] product_b_reg     // 18-bit Output result
);

    logic [11:0] const_shift0_a;
    logic [11:0] const_shift1_a;
    logic [11:0] const_shift2_a, const_shift2_a_reg;
    logic [11:0] const_shift3_a;
    logic [11:0] const_shift4_a;
    logic [11:0] const_shift5_a, const_shift5_a_reg;
    logic [11:0] const_shift0_b;
    logic [11:0] const_shift1_b;
    logic [11:0] const_shift2_b, const_shift2_b_reg;
    logic [11:0] const_shift3_b;
    logic [11:0] const_shift4_b;
    logic [11:0] const_shift5_b, const_shift5_b_reg;
    logic [13:0] sum_a_ll      ;
    logic [13:0] sum_a_ll_reg  ;
    logic [14:0] sum_a_lh      ;
    logic [13:0] sum_a_hl      ;
    logic [13:0] sum_a_hl_reg  ;
    logic [14:0] sum_a_hh      ;
    logic [13:0] sum_b_ll      ;
    logic [13:0] sum_b_ll_reg  ;
    logic [14:0] sum_b_lh      ;
    logic [13:0] sum_b_hl      ;
    logic [13:0] sum_b_hl_reg  ;
    logic [14:0] sum_b_hh      ;
    logic [17:0] product_a     ;
    logic [17:0] product_b     ;

    always_comb begin
```

```
    const_shift0_a = multiplicand_a[0]? multiplier_const: '0;
    const_shift1_a = multiplicand_a[1]? multiplier_const: '0;
    const_shift2_a = multiplicand_a[2]? multiplier_const: '0;
    const_shift3_a = multiplicand_a[3]? multiplier_const: '0;
    const_shift4_a = multiplicand_a[4]? multiplier_const: '0;
    const_shift5_a = multiplicand_a[5]? multiplier_const: '0;
    const_shift0_b = multiplicand_b[0]? multiplier_const: '0;
    const_shift1_b = multiplicand_b[1]? multiplier_const: '0;
    const_shift2_b = multiplicand_b[2]? multiplier_const: '0;
    const_shift3_b = multiplicand_b[3]? multiplier_const: '0;
    const_shift4_b = multiplicand_b[4]? multiplier_const: '0;
    const_shift5_b = multiplicand_b[5]? multiplier_const: '0;
end

always_ff @(posedge clk) begin
    const_shift2_a_reg <= const_shift2_a;
    const_shift5_a_reg <= const_shift5_a;

    const_shift2_b_reg <= const_shift2_b;
    const_shift5_b_reg <= const_shift5_b;
end

always_comb begin
    sum_a_ll = {const_shift1_a, 1'b0} + {1'b0, const_shift0_a};
    sum_a_lh = {const_shift2_a_reg, 2'b0} + {1'b0,
    ↪  sum_a_ll_reg};

    sum_a_hl = {const_shift4_a, 1'b0} + {1'b0, const_shift3_a};
    sum_a_hh = {const_shift5_a_reg, 2'b0} + {1'b0,
    ↪  sum_a_hl_reg};

    product_a = {sum_a_hh, 3'b0} + {3'b0, sum_a_lh};
end

always_comb begin
    sum_b_ll = {const_shift1_b, 1'b0} + {1'b0, const_shift0_b};
    sum_b_lh = {const_shift2_b_reg, 2'b0} + {1'b0,
    ↪  sum_b_ll_reg};

    sum_b_hl = {const_shift4_b, 1'b0} + {1'b0, const_shift3_b};
```

```
        sum_b_hh = {const_shift5_b_reg, 2'b0} + {1'b0,
        ↪  sum_b_hl_reg};

        product_b = {sum_b_hh, 3'b0} + {3'b0, sum_b_lh};
    end

    always_ff @(posedge clk) begin
        product_a_reg <= product_a;
        product_b_reg <= product_b;
        sum_a_ll_reg  <= sum_a_ll;
        sum_a_hl_reg  <= sum_a_hl;
        sum_b_ll_reg  <= sum_b_ll;
        sum_b_hl_reg  <= sum_b_hl;
    end

endmodule
```

Source Code A.14 DSP-based half multiplier (mult_dsp.sv).

```
`timescale 1ns/1ns
module mult_dsp (
    input  logic        clk             , // System clock
    input  logic [ 5:0] multiplicand_a  , // 6-bit Input A
    input  logic [ 5:0] multiplicand_b  , // 6-bit Input B
    input  logic [11:0] multiplier_const, // 12-bit Input CONST
    output logic [17:0] product_a_reg   , // 18-bit Output result
    output logic [17:0] product_b_reg     // 18-bit Output result
);
    logic [11:0] multiplier_const_reg;
    logic [23:0] multiplicand_comb, multiplicand_comb_reg;
    logic [35:0] mult_result_comb    ;

    always_comb begin
        multiplicand_comb = {multiplicand_a, 12'b0, multiplicand_b};
        mult_result_comb  = multiplicand_comb_reg *
        ↪  multiplier_const_reg;
    end

    always_ff @(posedge clk) begin
        multiplicand_comb_reg <= multiplicand_comb;
        multiplier_const_reg  <= multiplier_const;
        product_a_reg         <= mult_result_comb[35:18];
```

```
        product_b_reg         <= mult_result_comb[17:0];
    end

endmodule
```

Source Code A.15 Modular reducer (mult_red.sv).

```
`timescale 1ns / 1ps
`include "define.sv"

module mult_red #(parameter int W = `DATA_WIDTH) (
    input  logic           clk  ,
    input  logic [2*W-1:0] Z_in ,
    output logic [  W-1:0] P_out
);

    logic [  7:0] c_low_1, c1_low_1;
    logic [  9:0] c_low_2, c1_low_2;
    logic [ 10:0] c_low_3, c_low_2_sum, c1_low_3, c1_low_2_sum;
    logic [ 11:0] c_low_3_sum, c_low_3_sum_q1, c1_low_3_sum,
    ↪  c1_low_3_sum_q1;
    logic [ 15:0] c_high, c_high_q1;
    logic [ 16:0] c_res;
    logic [W-1:0] P_d;
    logic [  W:0] c1_res, c1_high, c1_res_ext, c1_res_low,
    ↪  c1_high_q1, c1_res_high;


    always_ff @(posedge clk) begin
        c_low_3_sum_q1  <= c_low_3_sum;
        c_high_q1       <= c_high;
        c1_low_3_sum_q1 <= c1_low_3_sum;
        c1_high_q1      <= c1_high;
        P_out           <= P_d;
    end

    assign c_high      = Z_in[2*W-1:8];
    assign c_low_1     = Z_in[7:0];
    assign c_low_2     = {Z_in[7:0], 2'b0};
    assign c_low_3     = {Z_in[7:0], 3'b0};
    assign c_low_2_sum = c_low_2+{2'b0,c_low_1};
    assign c_low_3_sum = {1'b0,c_low_3}+{1'b0,c_low_2_sum};
```

```
    assign c_res        = {5'b0, c_low_3_sum_q1} - c_high_q1;
    assign c1_high      = 13'(signed'(c_res[16:8]));
    assign c1_low_1     = c_res[7:0];
    assign c1_low_2     = {c_res[7:0], 2'b0};
    assign c1_low_3     = {c_res[7:0], 3'b0};
    assign c1_low_2_sum = c1_low_2+{2'b0,c1_low_1};
    assign c1_low_3_sum = {1'b0,c1_low_3}+{1'b0,c1_low_2_sum};

    assign c1_res_ext  = {1'b0, c1_low_3_sum_q1} - c1_high_q1[W:0];
    assign c1_res      = c1_res_ext[W:0];
    assign c1_res_low  = c1_res - 3329;
    assign c1_res_high = c1_res + 3329;

    assign P_d = c1_res[W] == 1? c1_res_high[W-1:0] :
        c1_res_low[W] == 1? c1_res[W-1:0] : c1_res_low[W-1:0];
endmodule
```

Source Code A.16 Data bank address input muxes (network_bank_in.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module network_bank_in #(parameter int ADDR_W = `BANK_ADDR_BIT)(
    input logic [ADDR_W-1:0] b0,b1,
    input logic sel_a_0,
    output logic [ADDR_W-1:0] new_address_0,new_address_1);

    always_comb
    begin
      case(sel_a_0)
        1'b0:begin
          new_address_0 = b0;
          new_address_1 = b1;
        end
        1'b1:begin
          new_address_0 = b1;
          new_address_1 = b0;
        end
        default:;
      endcase
    end
```

```
endmodule
```

Source Code A.17 BFU input muxes for even coefficients (network_bf_in.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module network_bf_in #(parameter int W = `DATA_WIDTH)(
    input logic clk,rst,
    input logic sel_a_0,
    input logic [W-1:0] q0,q1,
    output logic [W-1:0] u0,v0
    );

   logic sel_a_0_tmp;
   always@(posedge clk)
   begin
     if(rst)
       sel_a_0_tmp <= 0;
     else
       sel_a_0_tmp <= sel_a_0;
   end

   assign u0 = sel_a_0_tmp == 0 ? q0 : q1;
   assign v0 = sel_a_0_tmp == 1 ? q0 : q1;

endmodule
```

Source Code A.18 BFU input muxes for odd coefficients (network_bf_in_odd.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module network_bf_in_odd #(parameter int W = `DATA_WIDTH)(
    input logic clk,rst,
    input logic sel_a_0,
    input logic [W-1:0] q0,q1,
    output logic [W-1:0] u0,v0
    );

   logic sel_a_0_tmp;
   always@(posedge clk)
   begin
     if(rst)
       sel_a_0_tmp <= 0;
```

```
     else
       sel_a_0_tmp <= sel_a_0;
   end

   assign u0 = sel_a_0_tmp == 1 ? q0 : q1;
   assign v0 = sel_a_0_tmp == 0 ? q0 : q1;

endmodule
```

Source Code A.19 BFU output muxes for even coefficients (network_bf_out.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module network_bf_out #(parameter int W = `DATA_WIDTH)(
    input logic clk,
    input logic [W-1:0] bf_0_upper,bf_0_lower,
    input logic sel_a_0,
    output logic [W-1:0] d0,d1
    );

    logic sel_a_0_reg;


    shift_9_norst #(.W(1)) shift0
    ↪  (.clk(clk),.din(sel_a_0),.dout(sel_a_0_reg));

    always_comb begin
      case(sel_a_0_reg)
      1'b1: begin
        d0 = bf_0_lower;
        d1 = bf_0_upper;
      end
      1'b0: begin
        d0 = bf_0_upper;
        d1 = bf_0_lower;
      end
      default:;
      endcase
    end
endmodule
```

Source Code A.20 BFU output muxes for odd coefficients (network_bf_out_odd.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module network_bf_out_odd #(parameter int W = `DATA_WIDTH)(
    input logic clk,
    input logic [W-1:0] bf_0_upper,bf_0_lower,
    input logic sel_a_0,
    output logic [W-1:0] d0,d1
    );

    logic sel_a_0_reg;



    shift_9_norst #(.W(1)) shift0
    ↪  (.clk(clk),.din(sel_a_0),.dout(sel_a_0_reg));

    always_comb begin
      case(sel_a_0_reg)
      1'b0: begin
        d0 = bf_0_lower;
        d1 = bf_0_upper;
      end
      1'b1: begin
        d0 = bf_0_upper;
        d1 = bf_0_lower;
      end
      default:;
      endcase
    end
endmodule
```

Source Code A.21 Barrel shifter (no reset) 6× (shift_6_norst.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module shift_6_norst #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] din,
    input logic clk,
    output logic [W-1:0] dout
);

logic [W-1:0] t0,t1,t2,t3,t4,t5;
```

```
always_ff @(posedge clk) begin
    t0 <= din;
    t1 <= t0; t2 <= t1; t3 <= t2; t4 <= t3;
    t5 <= t4;
end

assign dout = t5;
endmodule
```

Source Code A.22 Barrel shifter (no reset) 9× (shift_9_norst.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module shift_9_norst #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] din,
    input logic clk,
    output logic [W-1:0] dout
);

logic [W-1:0] t0,t1,t2,t3,t4,t5,t6,t7,t8;

always_ff @(posedge clk) begin
    t0 <= din;
    t1 <= t0; t2 <= t1; t3 <= t2; t4 <= t3;
    t5 <= t4; t6 <= t5; t7 <= t6; t8 <= t7;
end

assign dout = t8;
endmodule
```

Source Code A.23 Barrel shifter (with reset) 9× (shift_9.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
(*DONT_TOUCH="true"*)
module shift_9 #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] din,
    input logic rst,clk,
    output logic [W-1:0] dout
);

logic [W-1:0] t0,t1,t2,t3,t4,t5,t6,t7,t8;
```

```
always_ff @(posedge clk)
    begin
        if(rst) begin
            t0 <= 0; t1 <= 0; t2 <= 0; t3 <= 0; t4 <= 0;
            t5 <= 0; t6 <= 0; t7 <= 0; t8 <= 0; end
        else begin
            t0 <= din;
            t1 <= t0; t2 <= t1; t3 <= t2; t4 <= t3;
            t5 <= t4; t6 <= t5; t7 <= t6; t8 <= t7;
        end
    end

assign dout = t8;
endmodule
```

Source Code A.24 Barrel shifter (with reset) 10× (shift_10.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
(*DONT_TOUCH="true"*)
module shift_10 #(parameter int W = `DATA_WIDTH)(
    input logic [W-1:0] din,
    input logic rst,clk,
    output logic [W-1:0] dout
);

logic [W-1:0] t0,t1,t2,t3,t4,t5,t6,t7,t8,t9;

always_ff @(posedge clk)
    begin
        if(rst) begin
            t0 <= 0; t1 <= 0; t2 <= 0; t3 <= 0; t4 <= 0;
            t5 <= 0; t6 <= 0; t7 <= 0; t8 <= 0; end
        else begin
            t0 <= din;
            t1 <= t0; t2 <= t1; t3 <= t2; t4 <= t3;
            t5 <= t4; t6 <= t5; t7 <= t6; t8 <= t7;
            t9 <= t8;
        end
    end
```

```
assign dout = t9;
endmodule
```

Source Code A.25 Twiddle factor ROM address generator (tf_address_generator.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module tf_address_generator
  import define::*;
(
    input clk,
    states_e conf,
    input [`BANK_ADDR_BIT-1:0] k,
    input [`P_BIT-1:0] p,
    output logic [`K_BIT-1:0] tf_address
);

  logic [`K_BIT-1:0] tf_address_reg_0, tf_address_reg_1;
  logic [`K_BIT-1:0] tf_address_tmp;
  //sel = 0 NTT mode; sel = 1 INTT mode
  assign tf_address_tmp = (conf == NTT || conf == DONE_NTT) ?
  ↪  tf_address_reg_0 : tf_address_reg_1;

  always_ff @(posedge clk) begin
    tf_address <= tf_address_tmp;
  end

  always_comb begin
    case (p)
      6: begin
        tf_address_reg_0 = k;
        tf_address_reg_1 = k;
      end
      5: begin
        tf_address_reg_0 = k + 1;
        tf_address_reg_1 = 2 - k;
      end
      4: begin
        tf_address_reg_0 = k + 3;
        tf_address_reg_1 = 6 - k;
      end
      3: begin
```

```
        tf_address_reg_0 = k + 7;
        tf_address_reg_1 = 14 - k;
      end
      2: begin
        tf_address_reg_0 = k + 15;
        tf_address_reg_1 = 30 - k;
      end
      1: begin
        tf_address_reg_0 = k + 31;
        tf_address_reg_1 = 62 - k;
      end
      0: begin
        tf_address_reg_0 = k + 63;
        tf_address_reg_1 = 126 - k;
      end
      default: begin
        tf_address_reg_0 = 0;
        tf_address_reg_1 = 0;
      end
    endcase
  end
endmodule
```

Source Code A.26 Twiddle factor ROM (tf_ROM.sv).

```
`timescale 1ns / 1ps
`include "define.sv"
module tf_ROM #(
    parameter int ROM_ADDR_W = `K_BIT,
    int W = `DATA_WIDTH
) (
    input logic clk,
    input logic [ROM_ADDR_W-1:0] A,
    input logic REN,
    output logic [W-1:0] Q
);

  (*rom_style = "distributed" *) logic [W-1:0] data;

  always @(posedge clk) begin
    if (REN)
      case (A)
```

```
        `include "kyber_tf.txt"
        default: data <= 'x;
      endcase
  end
  assign Q = data;
endmodule
```

Source Code A.27 Top level post-implementation timing testbench (tb_top.sv).

```
`timescale 1ns / 1ps
`include "define.sv"

module tb_top
    import define::*;
();
reg                     clk, rst;
states_e                conf, conf_reg;
logic                   conf_en                 ;
wire  [            2:0] done_flag               ;
logic [`DATA_WIDTH-1:0] banks_init[4][`BANK_SIZE];
logic [`DATA_WIDTH-1:0] banks_view[4][`BANK_SIZE];
logic [3:0] bankview0 [4][64];
logic [3:0] bankview1 [4][64];
logic [3:0] bankview2 [4][64];
logic [3:0] bankview3 [4][64];

top_poly_mul poly_mul (
    .clk      (clk      ),
    .rst      (rst      ),
    .conf     (conf     ),
    .done_flag(done_flag)
);

initial begin
    clk = 1'b0;
    forever #2.2 clk = ~clk;
end

initial begin

    $display($sformatf("\nOriginal vector:"));
    wait (done_flag == 1);
```

```
    $display($sformatf("\nCompleted NTT [Time: %0t]", $time));
    wait (tb_top.poly_mul.fsm.wen == 0);
    $display("\nPipeline idle:");
    wait (conf == INTT);
    @(negedge clk) wait (done_flag == 1);
    $display($sformatf("\nCompleted INTT [Time: %0t]", $time));
    wait (tb_top.poly_mul.fsm.wen == 0);
    #10;

    $display("\n-----------------RESULTS-----------------------");
    $display("-----------------------------------------------");
    $stop;
end

initial begin
    rst = 1;
    @(negedge clk) rst = 0;
end

initial begin
    conf = IDLE;
    @(negedge clk);
    conf = NTT;
    @(negedge clk);
    wait (done_flag > 0);
    conf = DONE_NTT;
    wait (!tb_top.poly_mul.fsm.wen);
    conf = INTT;
    @(negedge clk);
    wait (done_flag > 0);
    conf = DONE_INTT;

end

endmodule
```

Source Code A.28 Top level behavioral testbench (tb_top_behav.sv).

```
`timescale 1ns / 1ps
`include "define.sv"

module tb_top_behav
```

```
  import define::*;
();
  reg clk, rst;
  states_e conf, conf_reg;
  logic                   conf_en;
  wire  [            2:0] done_flag;
  logic [`DATA_WIDTH-1:0] banks_init[4][`BANK_SIZE];
  logic [`DATA_WIDTH-1:0] banks_ntt_tv[4][`BANK_SIZE];

  top_poly_mul poly_mul (
      .clk      (clk),
      .rst      (rst),
      .conf     (conf),
      .done_flag(done_flag)
  );

  initial begin
    clk = 1'b0;
    forever #5 clk = ~clk;
  end

  initial begin
    $readmemb({"bank0_ntt_64.tv"}, banks_ntt_tv[0]);
    $readmemb({"bank1_ntt_64.tv"}, banks_ntt_tv[1]);
    $readmemb({"bank2_ntt_64.tv"}, banks_ntt_tv[2]);
    $readmemb({"bank3_ntt_64.tv"}, banks_ntt_tv[3]);

    banks_init[0] = tb_top_behav.poly_mul.bank_0.bank;
    banks_init[1] = tb_top_behav.poly_mul.bank_1.bank;
    banks_init[2] = tb_top_behav.poly_mul.bank_2.bank;
    banks_init[3] = tb_top_behav.poly_mul.bank_3.bank;

    $display($sformatf("\nOriginal vector:"));
    $display($sformatf("bank 0:%p",
    ↪  tb_top_behav.poly_mul.bank_0.bank));
    $display($sformatf("bank 1:%p",
    ↪  tb_top_behav.poly_mul.bank_1.bank));
    $display($sformatf("bank 2:%p",
    ↪  tb_top_behav.poly_mul.bank_2.bank));
    $display($sformatf("bank 3:%p",
    ↪  tb_top_behav.poly_mul.bank_3.bank));
```

```
wait (done_flag == 1);
$display($sformatf("\nCompleted NTT [Time: %0t]", $time));
wait (tb_top_behav.poly_mul.fsm.wen == 0);
$display("\nPipeline idle:");
$display($sformatf("bank 0:%p",
↪  tb_top_behav.poly_mul.bank_0.bank));
$display($sformatf("bank 1:%p",
↪  tb_top_behav.poly_mul.bank_1.bank));
$display($sformatf("bank 2:%p",
↪  tb_top_behav.poly_mul.bank_2.bank));
$display($sformatf("bank 3:%p",
↪  tb_top_behav.poly_mul.bank_3.bank));
$display("\n-----------------RESULTS-----------------------");
if (banks_ntt_tv[0] === tb_top_behav.poly_mul.bank_0.bank)
  $display("Bank 0 coefficients in NTT domain matched
  ↪  testvectors [PASS]");
else $display("Bank 0 NTT incorrect; did not match testvectors
↪  [FAIL]");
if (banks_ntt_tv[1] === tb_top_behav.poly_mul.bank_1.bank)
  $display("Bank 1 coefficients in NTT domain matched
  ↪  testvectors [PASS]");
else $display("Bank 1 NTT incorrect; did not match testvectors
↪  [FAIL]");
if (banks_ntt_tv[2] === tb_top_behav.poly_mul.bank_2.bank)
  $display("Bank 2 coefficients in NTT domain matched
  ↪  testvectors [PASS]");
else $display("Bank 2 NTT incorrect; did not match testvectors
↪  [FAIL]");
if (banks_ntt_tv[3] === tb_top_behav.poly_mul.bank_3.bank)
  $display("Bank 3 coefficients in NTT domain matched
  ↪  testvectors [PASS]");
else $display("Bank 3 NTT incorrect; did not match testvectors
↪  [FAIL]");
$display("-------------------------------------------------");
wait (conf == INTT);
@(negedge clk) wait (done_flag == 1);
$display($sformatf("\nCompleted INTT [Time: %0t]", $time));
wait (tb_top_behav.poly_mul.fsm.wen == 0);
$display("\nPipeline idle:");
```

```
  $display($sformatf("bank 0:%p",
  ↪  tb_top_behav.poly_mul.bank_0.bank));
  $display($sformatf("bank 1:%p",
  ↪  tb_top_behav.poly_mul.bank_1.bank));
  $display($sformatf("bank 2:%p",
  ↪  tb_top_behav.poly_mul.bank_2.bank));
  $display($sformatf("bank 3:%p",
  ↪  tb_top_behav.poly_mul.bank_3.bank));

  $display("\n-----------------RESULTS-----------------------");
  if (banks_init[0] === tb_top_behav.poly_mul.bank_0.bank)
    $display("Bank 0 returned to original state [PASS]");
  else $display("Bank 0 did not return to original state [FAIL]");
  if (banks_init[1] === tb_top_behav.poly_mul.bank_1.bank)
    $display("Bank 1 returned to original state [PASS]");
  else $display("Bank 1 did not return to original state [FAIL]");
  if (banks_init[2] === tb_top_behav.poly_mul.bank_2.bank)
    $display("Bank 2 returned to original state [PASS]");
  else $display("Bank 2 did not return to original state [FAIL]");
  if (banks_init[3] === tb_top_behav.poly_mul.bank_3.bank)
    $display("Bank 3 returned to original state [PASS]");
  else $display("Bank 3 did not return to original state [FAIL]");
  $display("-----------------------------------------------");
  $stop;
end

initial begin
  rst = 1;
  @(negedge clk) rst = 0;
end

always_ff @(posedge clk) begin
  if (conf_en) conf <= conf_reg;
  else conf <= conf;
end

always_comb begin
  conf_reg = IDLE;
  conf_en  = 0;
  if (rst) begin
    conf_reg = NTT;
```

```
      conf_en  = 1;
    end

    if (!tb_top_behav.poly_mul.fsm.wen)  // wait for ntt to stop
    ↪  writing
      case (conf)
        DONE_NTT: begin
          conf_reg = INTT;
          conf_en  = 1;
        end
        default: ;
      endcase

    if (done_flag > 0) begin
      case (conf)
        NTT: begin
          conf_reg = DONE_NTT;
          conf_en  = 1;
        end
        INTT: begin
          conf_reg = DONE_INTT;
          conf_en  = 1;
        end
        default: ;
      endcase
    end
  end

endmodule
```

Poster

Optimisation of Number Theoretic Transform/ Inverse Number Theoretic Transform Architecture for CRYSTALS-Kyber Key Encapsulation Mechanism

Kyber logo | pq-crystals

INTRODUCTION

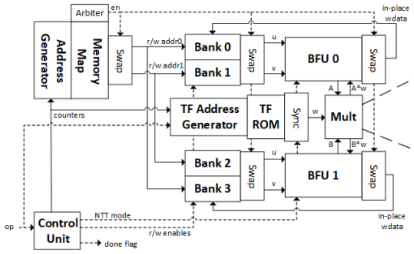
CRYSTALS-Kyber (or ML-KEM) is the primary post-quantum key-encapsulation mechanism (KEM) standardised for general quantum-resistant encryption.

We optimise its primary bottleneck computation hardware (NTT/INTT) on an FPGA platform to be **area-time efficient**..

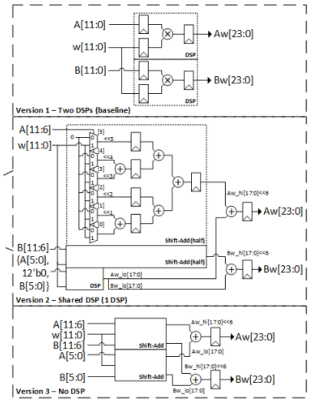
Artix-7 FPGA

METHOD

- CFNTT-based architecture (conflict-free memory mapping)
- K2RED algorithm for modular reduction
- BRAM-less and DSP-less for smallest area



NTT/INTT architecture with configurable multiplier



DISCUSSION

Three possible configurations, with trade-offs:

- Version 1 (2 DSP): Best **power** efficiency
- Version 2 (1 DSP): **Balanced** between area and power
- Version 3 (0 DSP): Smallest **area**

RESULTS

All three design versions hold the **lowest** area-time products (ATPs) among other previous works.
(up to 35% improvement compared to state-of-the-art)

This indicates that it is especially suitable for deployment in a constrained environment.

By Student: Tam Kai Yuan
Bachelor of Information Technology (Honours) Computer Engineering
Supervisor: Ts Dr Lee Wai Kong

FYP2