# RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform

Emre Karabulut
Department of Electrical and Computer Engineering
North Carolina State University
NC, USA
ekarabu@ncsu.edu

Aydin Aysu
Department of Electrical and Computer Engineering
North Carolina State University
NC, USA
aaysu@ncsu.edu

***Abstract*—Lattice-based cryptography has been growing in demand due to their quantum attack resiliency. Polynomial multiplication is a major computational bottleneck of lattice cryptosystems. To address the challenge, lattice-based cryptosystems use the Number Theoretic Transform (NTT). Although NTT reduces complexity, it is still a well-known computational bottleneck. At the same time, NTT arithmetic needs vary for different algorithms, motivating flexible solutions.**

**Although there are prior hardware and software NTT designs, they do not simultaneously offer flexibility and efficiency. This work provides an efficient and flexible NTT solution through domain-specific architectural support on RISC-V. Rather than using instruction-set extensions with compiler modifications or loosely coupling a RISC-V core with an NTT co-processor, our proposal uses application-specific dynamic instruction scheduling, memory dependence prediction, and datapath optimizations. This allows achieving a direct translation of C code to optimized NTT executions. We demonstrate the flexibility of our approach by implementing the NTT used in several lattice-based cryptography protocols:** `NewHope`**,** `qTESLA`**,** `CRYSTALS-Kyber`**,** `CRYSTALS-Dilithium`**, and** `Falcon`**. The results on the FPGA technology show that the proposed design is respectively 6×, 40×, and 3× more efficient than the baseline solution, Berkeley Out-of-Order Machine, and a prior HW/SW co-design, while providing the needed flexibility.**

***Index Terms*—Lattice-Based Cryptography, RISC-V, NTT**

## I. Introduction

Lattice-based cryptography is gaining traction as a successor of public-key cryptosystems for *quantum-secure* key-exchange and digital signature protocols [1]–[5]. In a lattice cryptosystem, the Schoolbook polynomial multiplication accounts for up to 98.8% percent of the overall computation time [6]. Number Theoretic Transform (NTT) reduces this $\mathcal{O}(n^2)$ complexity of the polynomial multiplication to $\mathcal{O}(n \cdot \log n)$. Therefore, NTT is a fundamental component of lattice cryptography implementations. At the same time, NTT is still a well-known bottleneck of many lattice-based cryptosystems [7].

NTT implementations include hardware [8]–[18], software [19]–[24], and HW/SW co-design methods [25], [26]. Although hardware solutions provide significant performance over software or HW/SW co-design techniques, the implementations violate the flexibility needed for evolving lattice cryptosystems such as the ones being reviewed for the ongoing US post-quantum encryption standardization [27]. Indeed, flexibility is necessary to evaluate multiple candidates submitted to the NIST standard—while the `CRYSTALS-Dilithium` [4] algorithm operates with polynomials of degree 255 with 23-bit coefficients, `NewHope` [1] uses polynomials of degree 1023 with 14-bit coefficients. Even the hardware specially designed to cater to multiple algorithms [13] can fail in flexibility: modulo prime number $q$ update in NIST Round-2 of `CRYSTALS-Kyber` [28] requires taping out a new chip as it cannot be implemented with the earlier design [13].

Software NTT implementations for embedded systems include architecture-specific assembly optimizations (e.g. for ARM [19], [20], [22], [24]) or software libraries like NFLlib [21]. While such solutions have more flexibility than hardware-based solutions, they have a lower performance.

HW/SW co-design techniques can combine the advantages of pure hardware-based and software-based implementations. Surprisingly, only a few HW/SW co-design implementations have been demonstrated to accelerate the NTT. Those designs, loosely couple a RISC-V core for I/O operations with a fully-fledged lattice crypto accelerator [13], [25]. The only exception is a recent white paper proposing custom instruction extensions for RISC-V [26]. The custom instruction, in this case, is still tailored for the specific use-cases and has to be adapted to support multiple algorithms or to accommodate changes in the arithmetic constructions (e.g. the updates in CRYSTALS-Kyber [28]). Furthermore, even for the specific use-cases, the custom instruction has to be hard-coded into compilers and requires manual assembly coding.

Given the prior work on NTT and the drawback of previous HW, SW, and HW/SW co-designed techniques, there is a need for efficient, flexible, and easy-to-develop solutions.

This paper proposes a novel, RISC-V based solution for efficient and flexible NTT implementation. Rather than using compiler optimizations or assembly coding with custom instruction extensions, our proposal uses architectural support/customization for the NTT. Specifically, the proposed architecture applies memory dependence prediction to optimize memory load-store operations, dynamically schedules instructions with out-of-order execution to address data dependence limitations, and combines the target algorithm's instructions with ALU cascading (a.k.a. collapsing) at run-time to improve hardware efficiency. While the basic principles of these techniques [29]–[31] and the idea of algorithm-specific

architectural improvements [32] are known, their adaptation to lattice cryptography is novel.

The key contributions of this paper are:

- A tracker design that recognizes and analyzes NTT algorithms' character during run-time execution. This unit identifies the control flow of NTT and caters to different NTT algorithms and configurations for flexibility.
- An application-specific out-of-order execution support that contains: (1) a predictive, NTT-specific memory load-store functionality reducing the cost of memory operations, and (2) an efficient, dynamic scheduler of assembly instructions resolving data dependency and maximizing memory and ALU unit's utilization.
- An ALU cascading scheme that enables efficient solutions using RISC-V native ISA without any additional instructions to perform the NTT and underlying field arithmetic.

The results validate the efficiency and flexibility of our approach. We implement the NTT of the latest NIST Round-2 post-quantum standard candidates: `NewHope` [1], `qTESLA` [2], `CRYSTALS-Kyber` [3] (and its Round-1 version), `CRYSTALS-Dilithium` [4], and `Falcon` [5]. On our proposed architecture, implementing actually becomes an automatic compilation of the reference software from the C language. The FPGA results show that our architecture enhancements improve the efficiency (in area-delay product) by 6×, 40×, and 3×, compared to the baseline design [33] `Berkeley Out-of-Order Machine (BOOM)` [34], and prior HW/SW co-design [25]. Furthermore, the proposed architecture ensures constant-time NTT operations for FPGA to prevent timing side-channels.

## II. Background and Prior Work

This section introduces the background on NTT with the underlying field arithmetic, and it discusses the target architecture and software stack.

### A. The NTT

Efficient lattice-based cryptosystems perform the multiplication of two polynomials within the polynomials rings of the form $\mathbb{Z}_q[x]/\phi(x)$. The polynomials in this ring have coefficients modulo prime number $q$ and $\phi(x)$ is a reduction polynomial of $(x^n+1)$, which allows efficient polynomial division. While the baseline schoolbook polynomial multiplication has the $\mathcal{O}(n^2)$ complexity, the Number Theoretic Transform converts polynomials in $\mathbb{Z}_q[x]/\phi(x)$ to a different domain (with a conversion cost of $\mathcal{O}(n \cdot \log n)$) where the multiplication becomes a coefficient-wise polynomial multiplication. Thus, NTT reduces the computational complexity of the multiplication.

The NTT is a version of Fast Fourier Transform (FFT) operating over the ring $\mathbb{Z}_q/\phi(x)$. While FFT uses the twiddle factor $\omega$ $n$-th root of unity of form $e^{(2\pi j/n)}$, NTT has $\omega \in \mathbb{Z}_q$ satisfying $\omega^n \equiv 1 \pmod q$ and $\forall i < n$, $\omega^i \neq 1 \pmod q$, where $q \equiv 1 \pmod n$.

Algorithm 1 presents an NTT algorithm [5]. The NTT takes a polynomial $A(x) \in \mathbb{Z}_q$, whose coefficients are $a_0, a_1, \ldots, a_{n-1}$, as an input and transforms the polynomial to $\bar{A}(x)$ with $\bar{A}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ in $\mathbb{Z}_q$. The NTT has $\mathcal{O} \log n$ steps and each step performs $n/2$ butterfly computations consisting of modular multiplication, addition, subtraction, and related load and store operations. Algorithm 1 shows the Cooley-Tukey butterfly [5] whereas some NTT algorithms use Gentleman-Sande [1], Fig. 4 shows the differences in their core operations. Iterative NTT algorithm applies modular reduction after each arithmetic operation in NTT [15], while some NTT algorithms load a new twiddle factor in the second loop rather than in the innermost loop [28]. Our architecture supports all such variations.

**Algorithm 1** In-Place NTT Algorithm Based on Cooley-Tukey Butterfly [5]

**Input:** $A(x) \in \mathbb{Z}_q[x]/(x^n+1)$
**Input:** primitive $n$-th root of unity $\omega \in \mathbb{Z}_q$, $n = 2^l$
**Output:** $\bar{A}(x) = \mathbf{NTT}(A) \in \mathbb{Z}_q[x]/(x^n+1)$

1: **for** $i$ from 1 by 1 to $\ell$ **do** ▷ *The third branch*
2: $m = 2^{l-i}$
3: **for** $j$ from 0 by 1 to $2^{i-1}-1$ **do** ▷ *The second branch*
4: $i_w \leftarrow (l+j)$
5: $W \leftarrow \omega^{i_w}$ ▷ *load-operation*
6: **for** $k$ from 0 by 1 to $m-1$ **do** ▷ *The first branch*
7: $U \leftarrow A[2 \cdot j \cdot m + k]$
8: $V \leftarrow A[2 \cdot j \cdot m + k + m]$
9: $P \leftarrow V \cdot W \pmod q$ ▷ *modular multiplication*
10: $O \leftarrow U - P$ ▷ *butterfly*
11: $E \leftarrow U + P$
12: $A[i_e] \leftarrow E$ ▷ *store-operation*
13: $A[i_o] \leftarrow O$
14: **end for**
15: **end for**
16: **end for**
17: return $\bar{A}(x) \in \mathbb{Z}_q[x]$

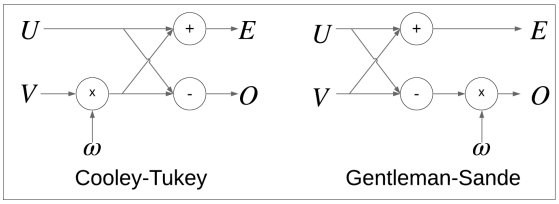


Fig. 1. Butterfly Configurations

### B. Modular Arithmetic

To perform fast modular reductions, efficient lattice cryptosystems avoid modular divisions. Instead, they use two techniques: Montgomery [15] and Barrett reduction [13]. It is not necessary to support both reduction algorithms because they are functionally equivalent. We choose Montgomery due to its feasibility to ALU cascading—while Barret [14] performs two subsequent subtractions, which cannot be collapsed with the single subtractor of our target core, Montogomery applies different types of subsequent operations (e.g. multiplication, addition, shift) as shown in Algorithm 2. This reduction technique eliminates the lower bits of the input product by

**Algorithm 2** Modular Multiplication with Montgomery Reduction

**Input:** $C = A \cdot B$ (a $2K$-bit positive integer)
**Input:** $q$ modulus prime number
**Input:** $q_{inv} = -(q^{-1}) \pmod R$ where $R = 2^K$
**Output:** $\bar{C} = C \cdot R^{-1} \pmod q$

1: $T \leftarrow C \cdot q_{inv}$
2: $X \leftarrow T \pmod R$
3: $Y \leftarrow X \cdot q$
4: $U \leftarrow (Y + C) >> K$
5: $V \leftarrow U - q$
6: **if** $(V < 0)$ **then** $\bar{C} = U$ **else** $\bar{C} = V$
7: return $\bar{C} \in \mathbb{Z}_q[x]$

adding multiple $q$ primes and shifting the result rather than subtracting multiple $q$ primes.

### C. The Target RISC-V Architecture

As our core, we chose an area-optimized, in-order RISC-V micro-processor `PicoRV32` [33] architecture that supports 32-bit integer base and multiplication-division instructions (`RV32IM`). The main reason of the choice is that RISC-V ISA is open source and suitable for embedded development, promoting energy-efficiency with reasonably high performance. Although we embedded our architectural support inside, the RISC-V core still fully supports `RV32IM`. Our architecture extension serves simply as an observer with no effect in the core unless an NTT algorithm is detected.

### D. Software Stack

Our hardware implementation detects and analyzes NTT algorithm instructions among other instructions. To aid in NTT detection, we define specific address locations—five ranges for coefficient input, twiddle factor, prime, and inverse prime numbers—in the memory with volatile variable declarations in C language. A software developer needs to simply assign the NTT algorithm inputs to the predefined memory locations. Our solution requires no changes in the compiler infrastructure as the compiler cannot eliminate the volatile declarations to optimize code. We define the memory declarations with 32-bit values to standardize and simplify hardware structure even though the input is typically smaller than 32 bits.

## III. The Inefficiencies in the Baseline Design

This section motivates our design choices by demonstrating the inefficiencies in a baseline design and by identifying ways to improve them.

Fig. 2 shows the resulting code and the cycle count of the innermost NTT loop of `CRYSTALS-Kyber` when the reference C code [3] is compiled with `-O3` (performance-optimized) flag. This loop executes $n/2 \cdot \log n$ times for a single NTT, e.g., 11264 times for the NTT of `qTesla`. We categorize the NTT's innermost loop operations into four tasks: memory load operations, butterfly operations, modular (Montgomery) reduction operations, and memory store operations. Our goal is to optimize the innermost loop's execution of the NTT algorithm.

| Dumped Assembly Code | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | `lhu a5,0(s2)` | `slli a5,a5,0x10` | `srai a5,a5,0x10` | `mul a5,a5,a0` | `lui t1,0xffffff` | `mul a2,a5,t1` | `slli a2,a2,0x10` | `srai a2,a2,0x10` | `lui a7,0x1` | `mul a2,a2,a7` | `srli a5,a5,0x10` | `add a5,a5,a2` | `lhu a3,0(a4)` | `sub a3,a3,a5` | `slli a3,a3,0x10` | `srai a3,a3,0x10` | `sh a3,0(s2)` | `lhu a3,-2(a4)` | `add a5,a5,a3` | `slli a5,a5,0x10` | `srai a5,a5,0x10` | `sh a5,-2(a4)` |
| Cycle Count | 4 | 8 | 15 | 21 | 25 | 31 | 36 | 43 | 47 | 51 | 55 | 62 | 66 | 70 | 77 | 84 | 92 | 96 | 100 | 107 | 114 | 122 |
| Load | | | | | | | | | | | | | | | | | | | | | | |
| Mont. Red. | | | | | | | | | | | | | | | | | | | | | | |
| Butterfly | | | | | | | | | | | | | | | | | | | | | | |
| Store | | | | | | | | | | | | | | | | | | | | | | |

Fig. 2. Cycle Count and Operation Breakdown of the Baseline NTT Innermost Loop Execution for `CRYSTALS-Kyber`

It takes 122 clock cycles for the baseline code to execute the innermost loop a single time. There are four major inefficiencies in the code. First, the data dependency, read-after-write (RAW), causes the processor to wait for the current instruction to finish (i.e., to be fetched, decoded, executed (or memory accessed) and written back before proceeding to the next one. This increases latency due to inefficient usage of the pipeline stages, e.g., between cycles 51 and 66. Second, there are independent instructions placed in sequence, e.g., between cycles 25 to 35 performing load and multiplication. Out-of-order execution can address these two inefficiencies. Third, there are redundant load-store address calculations and load-store executions. For instance, the load at cycle 96 uses the same address with the last store in cycle 122. Memory dependence prediction can reduce these inefficiencies. Fourth, due to the ring structure, the NTT algorithm often requires a logical shift following an arithmetic operation, causing RAW stalls, e.g., between cycles 31 and 43. The shifts can be cascaded to the prior instruction to improve performance.

Our proposed architecture enhancements reduce the cycle count of this loop from 122 to 6 (see Fig. 5).

## IV. The Proposed RISC-V Architecture Extensions

The proposed architectural extension has three key features to address the inefficiencies and to provide flexibility:

1) The architecture supports different NTT algorithms and different configurations. To that purpose, we designed a *tracker* that recognizes and analyzes the NTT algorithm's character. This indicates the control flow of operations and aids our implementation on scheduling the target NTT algorithm's instructions.
2) Our implementation shares but re-purposes ALU resources of the RISC-V core with ALU cascading to increase efficiency. We designed a *controller* to supervise this process. The controller also prevents undesirable memory load-store operations during this process and warrants to preserve instruction execution order by controlling program counter (PC) register of the RISC-V core.
3) Our implementation includes a *butterfly optimizer* to remove redundant instructions, manage memory dependencies, and out-of-order execute NTT butterfly and Montgomery reduction operations.

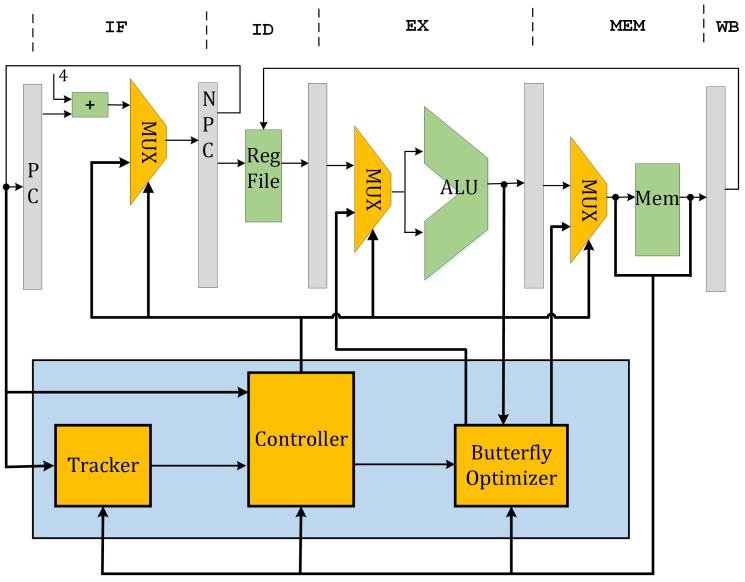


Fig. 3. The Block Diagram of the RISC-V Pipeline and Proposed Extensions

Fig. 3 shows the proposed architecture extensions with the three building blocks integrated into the RISC-V pipeline. The next three subsections elaborate on these contributions.

### A. The Tracker Design for the NTT Recognition and Analysis

Fig. 4 depicts the generic structure of the NTT, which has three branches (in assembly) and where the innermost branch loads $U$, $V$, and $\omega$, executes the butterfly and the modular reduction, and finally stores the resulting $O$ and $E$. When the RISC-V core starts executing the instruction group between PC1 and PC2, the tracker analyzes the target NTT algorithm by observing arithmetic instructions and memory load-store operations' orders on the predefined memory locations. This indeed enables distinguishing the characteristics of different NTT algorithms such as Cooley-Tukey vs. Gentleman-Sande, iterative vs. in-place, and differences in twiddle factor access patterns. The algorithm analysis occurs at run-time; hence, the proposed optimizations do not need to be hard-coded for each distinct NTT specification.

The tracker takes the program counter (PC) register and the memory ports as inputs. The tracker is a finite state machine (FSM) whose states change with the PC jump actions and load-store operations during the first three iterations of the innermost branch—nega-cyclic NTT (e.g., in `CRYSTALS-Kyber`) requires three iterations to be distinguished. The tracker generates two PC values, one memory address, and the detected NTT configuration as the output. PC1 and PC2 (Fig. 4) shows the boundaries of the innermost loop, and hence, the optimization target.

The tracker determines the ring size by observing the memory load addresses at the first iteration because one of them corresponds to the $(\frac{n}{2}\text{-}1)th$ address. Thus, our hardware implementation can configure itself for the ring size, $n$, at run-time. The tracker transmits the detected NTT configuration

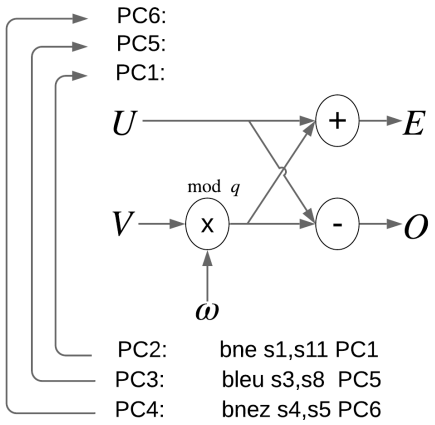


Fig. 4. The Innermost For-Loop and Branch Instructions

to the controller in 4-bit encoded format: the first bit of the frame shows whether the butterfly operation is Cooley–Tukey or Gentleman-Sande, while the second bit of the frame shows whether the $n$-th root of unity $\omega$ is updated in the second loop or in the innermost loop. The third bit represents if the NTT algorithm is nega-cyclic, while the fourth one indicates whether the $q$ primer number is bigger than 32 bits or not.

### B. The Controller Design for RISC-V Resource Sharing

Our architecture extension shares the resources of the RISC-V core but re-purposes them for optimized NTT execution. The objective of the controller is therefore to enable our memory dependency reduction, out-of-order execution, and ALU cascading optimizations, which are all tuned for the NTT. The controller achieves this functionality by switching the authorization owner from the RISC-V core to our butterfly optimizer through the select signals of MUXes (Fig. 3). After the target instructions (between PC1 and PC2) execute, the controller sets the NPC to PC2 before returning the control back to the RISC-V core. While the RISC-V core decides if the PC2 branch is taken or not taken, the controller predicatively loads the value at the next address of $\omega$.

The details of the controller's operation are as follows. The controller keeps the next PC (NPC) assignment at the current PC value to prevent the RISC-V core to proceed to the next instruction. The RISC-V core bypasses all instructions between the two PC values because the butterfly optimizer handles the instructions that belong to the NTT butterfly and modular reduction operations. When the butterfly optimizer completes, the controller releases the control to RISC-V by assigning PC2 value to the NPC. Thus, after releasing the control, the RISC-V core gives a decision on whether the corresponding branch is taken or not taken. If the branch is taken, the PC value jumps from PC2 to PC1, which means that the controller can get the authorization back. The controller repeats the sequence until the NTT algorithm execution is done. Our architecture does not have its own handler routine for interrupt or multi-threaded processing, it obeys the existing RISC-V core's interrupt or multi-threaded routines. Although

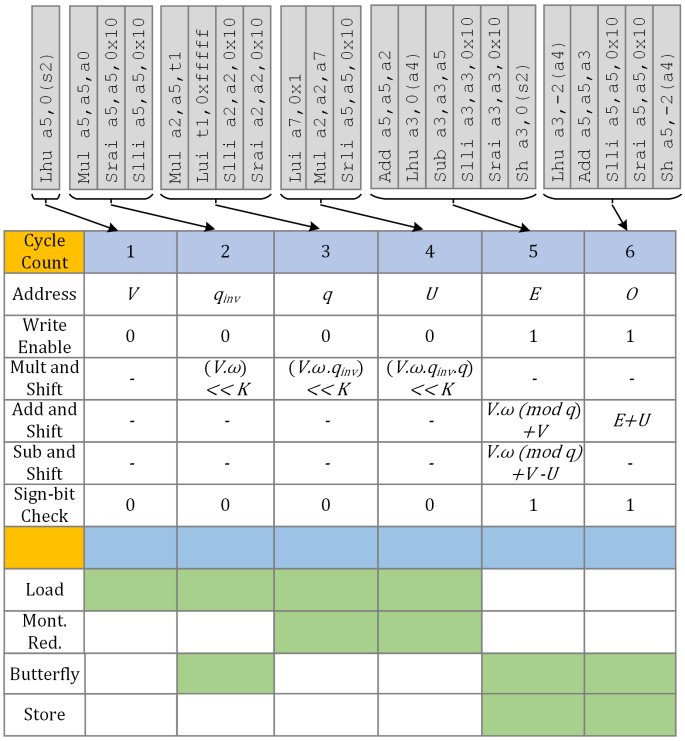


Fig. 5. Scheduling of Optimized NTT Operations

our implementation does not formally guarantee that there will never be any mispredictions, it has built-in capabilities to detect errors and raise a trap signal for a software trap handler.

The controller also loads twiddle factor, $\omega$, from memory and sends to the butterfly optimizer while the RISC-V core executes the branch instruction and memory is available.

### C. The Butterfly Optimizer Design

The butterfly optimizer performs the optimization techniques including out-of-order executions, memory dependence prediction, and ALU cascading [29]–[31]. Fig. 5 shows the resulting, optimized schedule of the NTT innermost loop, which reduces from the baseline cycle count of 122 to 6. The figure shows the case for `CRYSTALS-Kyber`, which has the Cooley-Tukey butterfly configuration with Montgomery reduction—the NTT of other algorithms can result in different schedules.

The key decision in optimizing the NTT innermost loop is to keep the memory *always* busy. At each execution, there are four memory loads (for $U$,$V$, $q$ and $q_{inv}$) and two stores (for $E$ and $O$), which take 6 cycles to complete (in a pipelined manner). Another important aspect is to combine ALU instructions and to interleave them with the execution of memory load-store instructions. Fig. 5 shows that NTT ALU function can be combined and completed in 5 cycles.

The details of the schedule in Fig. 5 are as follows. The butterfly optimizer sends the higher coefficient address to the memory at the first cycle. The butterfly optimizer sends $q_{inv}$ address to the memory at the second cycle and overlaps the load operation with the multiplication and shift operations for $\omega$, $V$. During the next three cycles, it performs Montgomery reduction, which consecutively performs the logical shift operations after multiplication, addition, and subtraction. We exploited this opportunity to perform ALU cascading—we augmented the RISC-V ALU by cascading the logical shift unit after the multiplier and adder/subtractor data path, to combine multiple instruction into a single one. The butterfly optimizer automatically routes such detected group of instructions into the cascaded ALU. This optimization approach results in reducing the 72 cycles of the reference ALU operations of Montgomery to 4 cycles. The new longer data path is not the critical path of the RISC-V core; hence, it does not cause to reduce the operating clock frequency.

The butterfly overlaps the last step of reduction and load operation for the lower coefficient address. The unit executes the Cooley-Tukey butterfly operation with load-store operations in parallel. The butterfly optimizer controls the operation order and ALU data path with an FSM that is initialized by the controller unit.

The butterfly optimizer also avoids redundant address generations for loading $U$,$V$ and storing $E$ and $O$ (see Fig. 5). The address generations for these load operation are the same with store operations. Thus, the butterfly reuses the address location instead of re-calculating them.

TABLE I
COMPARISON OF OUR PROPOSAL WITH RISC-V ISA BASED SOFTWARE IMPLEMENTATIONS

| Design | *(n, K)* | LUTs/DSPs | # of CC | Lat. Impr. | Eff. |
|---|---|---|---|---|---|
| PicoRV32-RV32IM | Kyber | 1053 / 4 | 161681 | – | 0.40 |
| | Dilithium | | 122600 | – | 0.53 |
| | NewHope[a,d] | | 189515 | – | 0.34 |
| | NewHope[b,d] | | 442623 | – | 0.15 |
| | Falcon[a] | | 481096 | – | 0.14 |
| | Falcon[b] | | 1063310 | – | 0.06 |
| | qTESLA[b] | | 274549 | – | 0.24 |
| | qTESLA[c] | | 575117 | – | 0.11 |
| **This Work** + PicoRV32-RV32IM | Kyber | **417 / 0** + 1053 / 4 | 43756 | ×3.7 | 1.21 |
| | Dilithium | | 43756 | ×2.8 | 1.21 |
| | NewHope[a,d] | | 81064 | ×2.4 | 0.65 |
| | NewHope[b,d] | | 180187 | ×2.5 | 0.29 |
| | Falcon[a] | | 81114 | ×6 | 0.65 |
| | Falcon[b] | | 180237 | ×6 | 0.29 |
| | qTESLA[b] | | 225225 | ×1.5 | 0.23 |
| | qTESLA[c] | | 491215 | ×1.5 | 0.11 |
| BOOM-RV64GC | Kyber | 191K / 36 | 41725 | ×9 | 0.01 |
| | Dilithium | | 17419 | ×7 | 0.03 |
| | NewHope[a,d] | | 21456 | ×8.9 | 0.02 |
| | NewHope[b,d] | | 49786 | ×9 | 0.01 |
| | Falcon[a] | | 68456 | ×7 | 0.01 |
| | Falcon[b] | | 97563 | ×10.9 | 0.01 |
| | qTESLA[b] | | 27337 | ×10 | 0.02 |
| | qTESLA[c] | | 58082 | ×9.9 | 0.01 |

[a]: $n$=512, [b]: $n$=1024, [c]: $n$=2048,
[d]: We slightly adjusted the reference NewHope code by replacing the modular division operation (%) with the Montgomery subroutine; this favors the baseline and the `BOOM`, not our proposal.

## V. IMPLEMENTATION RESULTS

We used the Verilog HDL and RTL coding to implement the proposed hardware. We synthesized, placed, and routed the code on the Xilinx VIRTEX-7 FPGA (xc7vx690tffg1761-2) with the 2018.3 version of the Xilinx Vivado tool.

TABLE II
NTT IMPLEMENTATION RESULTS AND COMPARISON TO PRIOR WORK

| Method | Work | # of Supported PQ Algorithms | Platform | $n$ | LUTs / REG / DSP / BRAM | Estimated LUTs Count[a] [35] | Cycle Count | Max Efficiency |
|---|---|---|---|---|---|---|---|---|
| HW/SW | [25] | 1 | Zynq-7000 | 1024 | 886 / 618 / 26 / 1 | 4006 | 24609 | 0.101 |
| HW[b] | [17] | – | VIRTEX-7 | 1024<br>2048 | 21K / 16K / 10 / 12<br>25K / 20K / 11 / 192 | 22K<br>26K | 7597<br>15852 | 0.059 |
| HW[b] | [16] | 2 | VIRTEX-7 | 1024 | 4737 / 3243 / 8 / 2 | 5697 | 16569 | 0.106 |
| HW | [8] | – | SPARTAN-6 | 256<br>512<br>1024 | 250 / – / 3 / 2<br>240 / – / 3 / 2<br>250 / – / 3 / 2 | 610<br>600<br>610 | 3840<br>11264<br>27648 | 0.593 |
| HW | [15] | 1 | VIRTEX-7 | 1024 | 34K / 16K / 476 / 228 | 91K | 80 | 1.371 |
| HW | [18] | 2 | VIRTEX-6 | 256<br>512 | 1349 / 860 / 1 / 2<br>1536 / 953 / 1 / 3 | 1469<br>1656 | 1691<br>3443 | 1.753 |
| HW | [14] | 2 | Zynq-7000 | 256<br>512<br>1024 | 980 / 395 / 26 / 2 | 4100 | 2056<br>4616<br>10248 | 0.528 |
| **HW/SW** | **This Work** | **6** | **VIRTEX-7** | **256<br>512<br>1024<br>2048** | **417 / 462 / 0 / 0** | **417** | **43756<br>81064<br>180237<br>491215** | **0.2958** |

[a]:To normalize the area estimation, we converted 1 Xilinx DSP48 to 120 LUTs following earlier work [35]. [b]:HLS design.

We also implemented the `Berkeley Out-of-Order Machine (BOOM)` RISC-V core for comparison. `BOOM` has a high-performance architecture, which supports complex out-of-order execution, branch-prediction and speculative execution and uses 64-bit base integer, integer multiplication and division, atomic, and single- and double-precision floating-point instructions [34]. We compiled the reference C codes submitted to the NIST post-quantum standardization using NTT—`NewHope` [1], `qTESLA` [2], `CRYSTALS-Kyber` [3], `CRYSTALS-Dilithium` [4], and `Falcon` [5]—with `riscv-gnu tool-chains` referenced in the corresponding git repository for `picoRV32` with our extensions [33] and for `BOOM` [34].

We first compare our results with different RISC-V cores: the baseline `PicoRV32-32IM`, the extended `PicoRV32` with our proposed techniques, and the `BOOM-RV64GC`. Table I shows that, as expected, `BOOM` achieves the fastest solutions and reduces the baseline latency by up to 10.9×, while our proposed solution achieves a relatively lower speedup of up to 6×. But our proposed extension occupies far fewer resources compared to `BOOM`. Therefore, our solution is up to 40× more efficient (in area×delay$^{-1}$ metric) compared to baseline and `BOOM`. This validates the efficiency of domain-specific architectures.

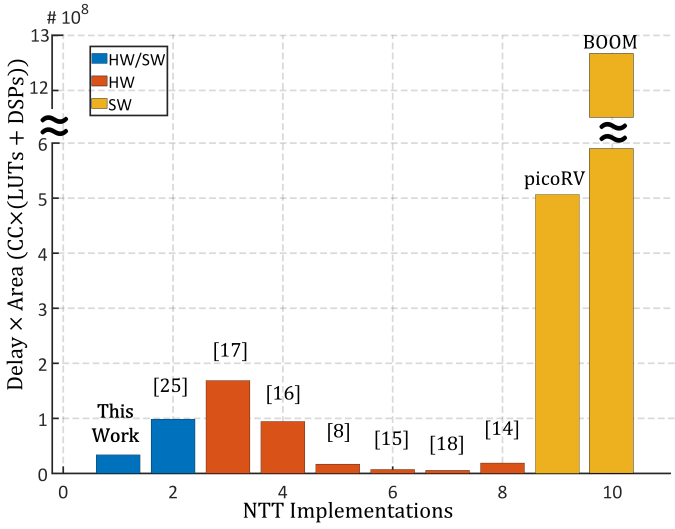


Fig. 6. Implementation Methods and Efficiency Comparison

Next, we compare our solution with prior hardware, software, and HW/SW co-design methods. Our solution arguably falls under the category of HW/SW co-design. Note that these implementations target different technologies, consider different paramteters ($n$ and $q$), and use different EDA tools; hence, the results present a first-order comparison. Fig. 6 shows the summary of our comparison and Table II provides the details—note that the table does not include ASIC or software microcontroller implementations. The figure reflects different implementation methods and compares their efficiency based on the inverse of area-delay product. The proposed design is superior to the existing HW/SW co-design work (by 3×) and even outperforms some of the pure hardware solutions (by up to 5×). The table shows that, unlike earlier pure hardware designs, our solution can flexibly support all 5 NIST proposals using NTT including the recent updates of `CRYSTALS-Kyber`.

## VI. CONCLUSIONS

This paper proposes a domain-specific architecture support and optimization to accelerate the NTT on the FPGA technology. Through dynamic instruction scheduling and optimized ALU, the proposed RISC-V core can achieve up to 6× speedup over the baseline solution, is more efficient than a prior HW/SW co-design by 3×, and even reaches the efficiency of pure hardware accelerators. The proposed design can also identify the NTT execution and automatically map a C code into optimized executions without any change of compiler or manual assembly coding.

## VII. ACKNOWLEDGEMENTS

We thank reviewers for their valuable feedback. This research is supported in part by the by the National Science Foundation under Grant No. 1850373. We acknowledge Xilinx for their FPGA donation.

## REFERENCES


[1] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope," in *25th USENIX*, 2016, pp. 327–343.

[2] E. Alkim, P. S. Barreto, N. Bindel, P. Longa, and J. E. Ricardini, "The lattice-based digital signature scheme qtesla." *IACR Cryptology ePrint Archive*, vol. 2019, p. 85, 2019.

[3] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber: a cca-secure module-lattice-based kem," in *2018 IEEE EuroS&P*. IEEE, 2018, pp. 353–367.

[4] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.

[5] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru."

[6] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, June 2014, pp. 2796–2799.

[7] H. Nejatollahi, N. Dutt, F. Ray, Sandip an d Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–41, 2019.

[8] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," in *2013 IEEE International Symposium on HOST*. IEEE, 2013, pp. 81–86.

[9] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 68–85.

[10] S.S. Roy et al., "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," Cryptology ePrint Archive, Report 2019/160, 2019.

[11] E. Ozturk, Y. Doroz, E. Savas, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, Jan 2017.

[12] S. Song, W. Tang, T. Chen, and Z. Zhang, "Leia: A 2.05mm2140mw lattice encryption instruction accelerator in 40nm cmos," in *2018 IEEE Custom Integrated Circuits Conference (CICC)*, April 2018, pp. 1–4.

[13] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Transactions on CHES*, pp. 17–61, 2019.

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

[15] A. C. Mert, E. Ozturk, and E. Savas, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," Cryptology ePrint Archive, Report 2019/109, 2019.

[16] E. Ozcan and A. Aysu, "High-level-synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, 2019.

[17] K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast number theoretic transform," in *2018 19th ISQED*, March 2018, pp. 106–111.

[18] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *CHES*, 2014, pp. 371–391.

[19] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of kyber on cortex-m4," in *International Conference on Cryptology in Africa*. Springer, 2019, pp. 209–228.

[20] P. S. Matthias J. Kannwischer, J. Rijneveld and K. Stoffelen, "Pqm4: Post-quantum crypto library for the arm cortex-m4," https://https://github.com/mupq/pqm4.

[21] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "Nfllib: Ntt-based fast lattice library," in *Topics in Cryptology - CT-RSA 2016*, San Francisco, CA, USA, pp. 341–356.

[22] E. Alkim, Y. A. Bilgin, M. Cenk, and F. Gérard, "Cortex-m4 optimizations for {R,M}lwe schemes," Cryptology ePrint Archive, Report 2020/012, 2020, https://eprint.iacr.org/2020/012.

[23] G. Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography." *IACR Cryptology ePrint Archive*, vol. 2018, p. 39, 2018.

[24] E. Alkim, P. Jakubeit, and P. Schwabe, "Newhope on arm cortex-m," in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2016, pp. 332–349.

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

[26] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "Isa extensions for finite field arithmetic - accelerating kyber and newhope on risc-v," Cryptology ePrint Archive, Report 2020/049, 2020, https://eprint.iacr.org/2020/049.

[27] "Nist-post-quantum cryptography," https://csrc.nist.gov/projects/post-quantum-cryptography, accessed: 2020-03-30.

[28] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. Schanck, G. Seiler, and D. Stehle, "Crystals-kyber–algorithm specifications and supporting documentation," *NIST Technical Report*, 2019.

[29] J. E. Smith, "Dynamic instruction scheduling and the astronautics zs-1," *Computer*, vol. 22, no. 7, pp. 21–35, 1989.

[30] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The performance potential of data dependence speculation collapsing," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. USA: IEEE Computer Society, 1996, p. 238–247.

[31] H. Sasaki, M. Kondo, and H. Nakamura, "Dynamic instruction cascading on gals microprocessors," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, V. Paliouras, J. Vounckx, and D. Verkest, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 30–39.

[32] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 373–386.

[33] C. Wolf, "Picorv32 - a size-optimized risc-v cpu," https://github.com/cliffordwolf/picorv32.

[34] J. Zhao, "Boom: Berkeley out-of-order machine," https://github.com/riscv-boom/riscv-boom.

[35] K. H. Chethan and N. Kapre, "Hoplite-dsp: Harnessing the xilinx dsp48 multiplexers to efficiently support nocs on fpgas," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–10.