Leveraging a Superscalar CVA6 to Implement NTT Instructions for Post-Quantum Cryptography

Côme Allart[1,2], Kévin Guilloux[1], Jean-Roch Coulon[1], André Sintzoff[1], Olivier Potin[2], and Jean-Baptiste Rigaud[2]

[1] Thales DIS, Meyreuil, France,
{come.allart, kevin.guilloux, jean-roch.coulon, andre.sintzoff}@thalesgroup.com
[2] Mines Saint-Étienne, CEA, Leti, Centre CMP, F-13541 Gardanne, France,
{come.allart, olivier.potin, rigaud}@emse.fr

Abstract. Post-Quantum Cryptography (PQC) algorithms were designed to be robust against attacks by both classical and quantum computers. ML-KEM and ML-DSA are two PQC algorithms ratified by the NIST. Both use the same base operations, namely Keccak and the Number Theoretic Transform (NTT). This paper focuses on the acceleration of the NTT and its counterpart NTT^{-1} functions with an implementation inside the pipeline of the superscalar embedded processor CVA6.
We implement Cooley-Tukey and Gentleman-Sande butterfly instructions to accelerate these functions, leveraging the superscalar pipeline of CVA6. Thus, we run NTT and NTT^{-1} functions in less than 2000 cycles, a performance gain of at least a factor of 5 compared to a software-optimized version for the superscalar CVA6.

Keywords: Embedded processor · RISC-V · CVA6 · Superscalar · PQC · ML-DSA · NTT · Butterfly

1 Introduction

Embedded devices need cryptography to secure their communications. However, common asymmetric cryptography algorithms [DH76,RSA78] could be broken by a quantum computer with enough qubits running Shor algorithm [Sho94]. Thus, Post-Quantum Cryptography (PQC) algorithms were designed to be robust against attacks by both classical and quantum computers. The National Institute of Standards and Technology (NIST) organized a contest to choose the best algorithms to standardize for the industry [NIS16].

ML-KEM (Module-Lattice-based Key-Encapsulation Mechanism) and ML-DSA (Module-Lattice-based Digital Signature Algorithm) were two of the winners of this contest and were ratified by the NIST [NIS24a,NIS24b]. The former

This work has received funding by the ISOLDE Chips JU project (grant nr. 101112274).

is used for key encapsulation and data encryption, and the latter is used for digital signature. Both use the Keccak hash function [NIS15] and the Number Theoretic Transform (NTT) [Sha23] to speed up polynomial multiplications. In this paper, we focus on the ML-DSA signature scheme, which consists of three algorithms: the key generation, the signature and the verification.

All these three algorithms spend most of their time inside Keccak functions [MBB+23]. Consequently, several Keccak hardware accelerators were developed to improve the execution speed of ML-DSA [DMM24,YSZ+24,LTQ+24].

The second most time-consuming operation in the three algorithms of ML-DSA is the polynomial multiplication. A naive implementation of the polynomial multiplication has a $O(n^2)$ time complexity. However, in the NTT field, polynomial multiplications are performed pointwise, so the time complexity is $O(n)$. Entry into and exit from the NTT field are performed using the NTT and NTT^{-1} functions, denoted as "NTT functions", which both are $O(n \log n)$. Finally, the full polynomial multiplication is $O(n \log n)$ and most of the time is spent in NTT and NTT^{-1}. In the end, it is faster to use the NTT than using the naive implementation. The base operations of NTT and NTT^{-1} are the Cooley-Tukey and Gentleman-Sande butterflies, respectively. The base operation of the pointwise multiplication is the Montgomery multiplication. Butterflies are described in Section 3.2 and the NTT function is explained further in Section 4.

This work studies how to leverage the pipeline of a superscalar processor to accelerate NTT functions for PQC. We focus on in-order embedded processors as we cannot afford out-of-order processors for embedded devices. We use the open-source superscalar processor CVA6 [Ope24] as the base for this work.

First, we introduce state-of-the-art methods to accelerate the execution of NTT functions. Then, we explain how the CVA6 processor works, and describe the butterfly operations. As a preliminary work, we optimize the NTT software implementation to reduce the number of memory accesses. Thus, we get a reference point to evaluate the performance gain of our hardware acceleration. We then follow a model-driven method and add a new butterfly instruction into the model of CVA6 performance [ACS+24] to explore design space and refine our design choices. In Section 6, we implement our design choices into the CVA6 pipeline. Finally, we evaluate the performance gain, the energy consumption and the cost of this solution in area and maximal clock frequency. We compare our results with related works.

2 Related Work

Four approaches are used to accelerate NTT functions in PQC algorithms.

A first solution is to optimize software, which is useful to target existing hardware. Software optimizations are proposed by [ZYHK24], targeting various sets of RISC-V options. The target CPU was the XuanTie C908, a 9-stage in-order dual-issue superscalar processor.

A second solution is to use a loosely-coupled accelerator, which executes an entire complex function without software intervention. The accelerator is usually

memory-mapped, for instance using the AXI protocol [ARM04]. Accelerators for the whole cryptography algorithms are proposed by [KSFS22] and [MSS24]. A loosely-coupled accelerator implementing ML-KEM NTT with local masking is also proposed by [RVBB24]. It has its own control unit and its own memories, and the countermeasures against side-channel attacks are based on [RPBC20]. A hybrid approach is used by [DMM24], creating both loosely- and tightly-coupled accelerators for PQC algorithms. They chose to implement the NTT in the loosely-coupled accelerator. Finally, an accelerator is proposed by [ASD25] to address issues related to the use of large numbers in NTT for homomorphic cryptography.

A third approach is to use a tightly-coupled accelerator: a processor extension executing new instructions. Core-V eXtension InterFace (CV-X-IF) [Ope21] is a protocol to connect tightly-coupled accelerators to RISC-V processors. It is used by [DDMV24] to accelerate NTT execution for ML-KEM without modifying the core pipeline. They used the RISC-V "R" format to create new instructions, packing two 13-bit polynomial coefficients in a 32-bit register. They also created the "R4" format adding a source register—a fourth operand—not to add instructions packing the coefficients.

Lastly, a fourth approach is to add instructions inside the processor pipeline. A Single-Instruction Multiple-Data (SIMD) extension for PQC is proposed by [YSZ+24] to perform operations on multiple data simultaneously. It results in a high throughput and a similarly high cost in resources. Another extension is proposed by [MBB+23] instructions to execute butterflies and accelerate the NTT functions of ML-KEM and ML-DSA. Finally, a similar extension is introduced by [NDMZ+21] for the 64-bit CVA6. Instead of writing the butterfly output coefficients into two registers, they are packed into a 64-bit register.

We compared the number of clock cycles required to execute NTT, NTT^{-1} and pointwise multiplication of ML-DSA using these solutions in Table 1.

The software versions of NTT functions used as a reference by [YSZ+24], [MBB+23] and [NDMZ+21] run in tens of thousands of cycles. The software acceleration proposed by [ZYHK24] reduces this duration to 7054 cycles for the NTT function, and 3395 cycles with the vector extension. The loosely-coupled solutions bring the best performance, with an execution duration of about a thousand cycles. This is also true for pointwise multiplication, executed in 264 cycles using [MSS24], including the accumulation needed to multiply a matrix of polynomials by a vector of polynomials. SIMD acceleration executes NTT functions in less than 2000 cycles. The instructions implemented inside the pipeline have a performance of a few thousand cycles only. The tightly-coupled accelerator from [DDMV24] is not in the table because it is not compatible with ML-DSA.

The software approach is not enough to reach high performance, so we decided to implement a hardware acceleration. We chose to not use the loosely-coupled approach because we want to let the program control the NTT execution, which allows applying security patches. The tightly-coupled approach provides more control to the program. However, [DDMV24] used the fact that ML-KEM polynomial coefficients are only 13 bits long to pack them in a 32-bit destination

Table 1: Duration of NTT, NTT^{-1} and pointwise multiplication (cycles)

Acceleration	Processor	Solution	NTT	NTT^{-1}	Pt.wise mult.
Software	C908	[ZYHK24] on RV32IM	7054	7561	2026
		[ZYHK24] on RVV	3395	3540	668
Loosely-coupled	CVA6	[MSS24]	1074	1074	264$^{(1)}$
	CV32E40X	[DMM24]	1531	1531	/
SIMD	CV32E40P	32-bit RISC-V	45270	49738	/
		[YSZ$^+$24]	1750	1925	/
		Variation	-96 %	-96 %	/
Inside pipeline	RI5CY	32-bit RISC-V	17041	20372	4346
		[MBB$^+$23]	2705	3979	1274
		Variation	-84 %	-80 %	-71 %
	CVA6 single-issue	64-bit RISC-V	38043	46266	/
		[NDMZ$^+$21]	18554	21375	/
		Variation	-51 %	-54 %	/

$^{(1)}$: Multiply and accumulate, for matrix-vector multiplication.

register, which would not be possible for ML-DSA and its 23-bit polynomial coefficients.

As a consequence, we chose to implement new instructions inside the processor pipeline. This approach yields the execution control to the program and is more flexible as it is not restricted by the interface between the processor and the accelerator. It will allow producing two output values simultaneously as [MBB$^+$23] did. However, new issues appear with out-of-order execution completion as it is not possible to write the two results right into the register bank. Also, the superscalar CVA6 brings new microarchitectural opportunities to explore, compared to the single-issue RI5CY processor used by [MBB$^+$23]. Finally, we chose to not use the SIMD approach as it seems too expensive in area.

In this paper, we add support for new instructions dedicated to the acceleration of ML-DSA NTT functions inside the CVA6 pipeline. This modification is more complex than creating a loosely- or tightly-coupled accelerator because of the pipeline integration issues. However, it gives us more flexibility in our design choices. We use the model-driven design method proposed by [ACS$^+$24] to explore design space before implementation inside CVA6 pipeline. The new instructions will execute the butterfly operations. They are presented in the next section, after an introduction to the CVA6 processor.

3 Background

In this section, we describe how CVA6 works and we explain butterfly operations.

3.1 CVA6: a RISC-V Open-Source Superscalar Processor

RISC-V is an open Instruction Set Architecture (ISA) standard. It is customizable and open, which allows making processors with different sets of features.

CVA6 is an open-source processor implementing the RISC-V ISA. It was created at ETH Zurich [ZB19] and is maintained by OpenHW Foundation [Ope24]. It is highly configurable as it is possible to synthesize 32- or 64-bit processors from it, with various sets of features including ISA extensions and performance features such as a superscalar pipeline.

CVA6 pipeline has 6 stages, namely instruction fetch, branch prediction, decode, issue, execute and commit. Its specificity comes from how instructions are handled in the 3 last stages, from issue to commit. CVA6 issues instructions and reads operands in-order, but instructions are allowed to complete execution out-of-order. To allow speculation and precise exceptions, it uses a ReOrder Buffer (ROB) to store the results before they are committed in-order, non-speculatively. This ROB is included into a scoreboard which tracks dependences between instructions. This whole mechanism allows issuing short instructions (e.g.: addition) after a long instruction (e.g.: load from memory) while the latter is still running, leveraging Instruction-Level Parallelism (ILP) in the execute stage despite long instructions.

It proceeds in three steps:

1. Issue stage detects data hazards (dependences) and structural hazards (congestions in the pipeline). If there are no hazards, the instruction is sent to a functional unit (FU) in the execute stage, and is inserted into the scoreboard.
2. When the FU eventually completes execution, it sends the result to the scoreboard, which stores the result along with the related instruction.
3. Finally, when an instruction has its result in the scoreboard, it is sent in-order to the commit stage, which performs the actual architectural change. For instance, the commit stage can write the result into the register bank, or allow the store unit to perform the store request to the memory.

All this work is based on the `cv32a60x` branch from the GitHub repository, commit `b1f80bd` [Ope24]. It uses scratchpad memories via the OBI protocol.

A model of CVA6 performance was built in Python to explore design space and take decisions before starting the implementation of new performance features [ACS+24]. Its first use was to design the superscalar pipeline of CVA6. It was also used as a reference during implementation to determine if the performance goal is met, and if not, to locate differences both in the program execution and in the pipeline. It takes as an input an execution trace from CVA6—or from any RISC-V processor or Instruction Set Simulator (ISS) [RIS24]. This trace contains the list of dynamic instructions executed, including the binary encodings and addresses. This work uses the model to implement new instructions.

3.2 Butterfly

The polynomial multiplications of ML-DSA are performed in the NTT field, where they are pointwise multiplications, which is faster than the polynomial multiplication in the normal field. The NTT and NTT^{-1} functions convert the polynomial representation to and from the NTT field. This paper mainly focuses on accelerating these functions.

The NTT-accelerated polynomial multiplication between polynomials $A[X]$ and $B[X]$ is shown in Figure 1. Three base operations are used during this polynomial multiplication: the Cooley-Tukey butterfly, the Montgomery multiplication and the Gentleman-Sande butterfly. The NTT function is a sequence of Cooley-Tukey butterflies and makes the polynomial enter the NTT field. The pointwise multiplication in the NTT field uses Montgomery multiplications. The NTT^{-1} function makes the polynomial leave the NTT field with a sequence of Gentleman-Sande butterflies followed by Montgomery reductions, a special case of Montgomery multiplications.

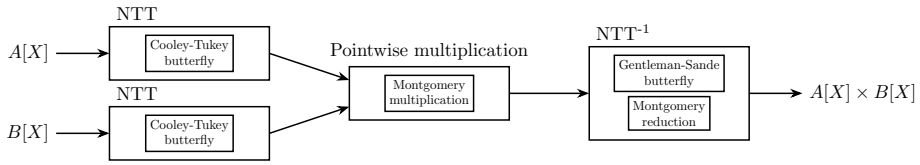
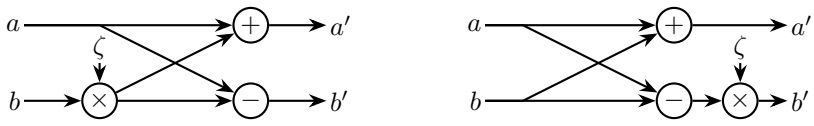


Fig. 1: NTT-accelerated polynomial multiplication

First, the Cooley-Tukey butterfly in Figure 2a performs the two assignments in Equation 1. a and b are the input coefficients. a' and b' are the output coefficients. ζ depends on the position of the butterfly in the algorithm and is independent of the input polynomial. It is a 512^{th} root of the unity. $q = 8380417$ is the modulus, a constant fixed by the ML-DSA specification regardless of the security level. All the values handled are represented on 32 bits.

The Montgomery multiplication performs a modular multiplication as shown in Equation 2. Its inputs are the a and b values, and its output is a'. The q constant is the same as for butterflies. It is used to perform the pointwise multiplication in the NTT field, and in the butterflies where one of its inputs is set to ζ. It is also used for Montgomery reductions in the NTT^{-1} function, with one of the inputs equals to 1. In all its uses, the modulus of the Montgomery multiplications of ML-DSA is always the same q constant.

$$\begin{aligned} a' &= a + \zeta.b \mod q \\ b' &= a - \zeta.b \mod q \end{aligned} \quad (1) \qquad a' = a.b \mod q \quad (2) \qquad \begin{aligned} a' &= a + b \mod q \\ b' &= \zeta.(a - b) \mod q \end{aligned} \quad (3)$$



(a) Cooley-Tukey butterfly (b) Gentleman-Sande butterfly

Fig. 2: Cooley-Tukey and Gentleman-Sande butterflies

The Gentleman-Sande butterfly in Figure 2b and Equation 3 is similar to the Cooley-Tukey butterfly, but the Montgomery multiplication by ζ is performed before outputting the b' coefficient instead of after reading the b coefficient.

We chose to speed up these three base operations to accelerate the NTT, NTT^{-1} and Montgomery multiplication of ML-DSA. First, we will focus on the NTT and its Cooley-Tukey butterflies. The issues are similar for the NTT^{-1} and its Gentleman-Sande butterflies.

In the next section, we focus on NTT software optimization. It will provide us with a relevant software reference for measuring how much our hardware solution accelerates NTT. It will also help us understand how to optimize our software code using our new instructions.

4 NTT Software Optimization

In this section, we explain how the NTT function works, and we show a way to reduce the number of memory accesses it performs to improve its performance without modifying the hardware.

To simplify the figures, we consider an NTT applied to a 16-coefficient polynomial, although ML-DSA polynomials have 256 coefficients—and 128 coefficients for ML-KEM polynomials. We denote by $A[X]$ the polynomial in the normal field and $\hat{A}[X]$ in the NTT field.

The NTT is shown in Figure 3. The dots represent the coefficients and the crosses binding the points represent the butterflies performed. The first butterfly of each layer is highlighted in red. For an n coefficient polynomial, the NTT function executes $\log_2 n$ layers of $\frac{n}{2}$ butterflies each. Consequently, the NTT for a 16-coefficient polynomial has 4 layers of 8 butterflies.

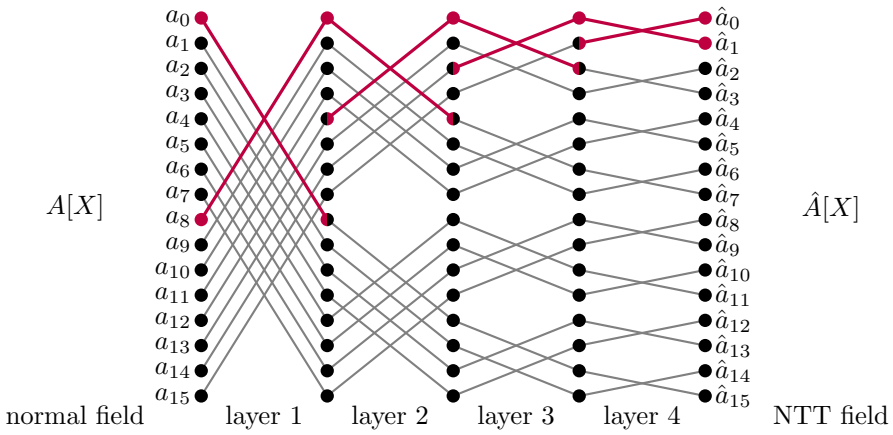


Fig. 3: 16-coefficient NTT, radix-2

A naive implementation of the NTT reads from memory the coefficients a_0 and a_8 for the first layer, performs the first butterfly, and stores the results back into memory. Then, it repeats these operations with a_1 and a_9, and again until coefficients a_7 and a_{15}, to perform the first layer of butterflies. This version of the NTT is called "radix-2" because the coefficients are processed pairwise. It reads and writes all coefficients in memory for each layer.

However, it is possible to use more CPU registers to reduce the number of memory accesses performed. From an algorithmic point of view, it is equivalent to fusing two consecutive NTT layers. This optimization could go further, fusing more layers, depending on the number of registers in the ISA. Such an optimization has been proposed for Arm Cortex-M3 and M4 [GKS20]. It has then been applied to RISC-V with the XuanTie C908 processor [ZYHK24]. In this section, we apply this software optimization with the superscalar CVA6.

Once the results from the first butterfly with the a_0 and a_8 coefficients are computed, they can be used to perform the two butterflies using them. Before so, we need to perform the butterfly with the a_4 and a_{12} coefficients in the first layer, as shown in Figure 4.

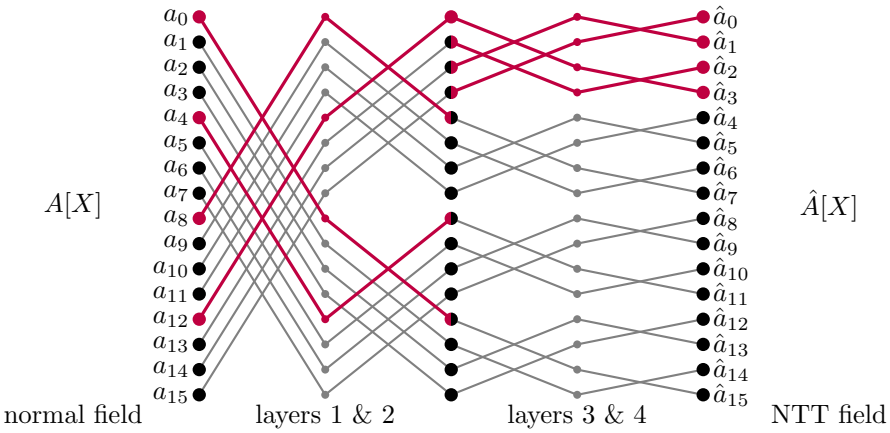


Fig. 4: 16-coefficient NTT, radix-4

This modified implementation reads and writes all coefficients in memory only once for 2 NTT layers. The intermediate values are kept in CPU registers and are not transferred to memory. We show them as smaller dots in Figure 4. This version of the NTT is named "radix-4" because the butterflies are grouped to process coefficients by 4 instead of 2. The number of butterflies to perform is the same, but fewer memory accesses are performed. Indeed, only 4 loads and 4 stores in memory are required to perform 4 butterflies, whereas 8 loads and 8 stores were required with the radix-2 implementation.

Actually, we can mix different radixes. We denote a group of fused NTT layers by its number of layers: "1" for a single radix-2 layer, "2" for a pair of layers performing radix-4 operation, etc. For instance, Figure 3 shows a "1+1+1+1" version of the 16-coefficient NTT and Figure 4 shows a "2+2" version. Following the same logic, we can introduce a "1+3" version, shown in Figure 5.

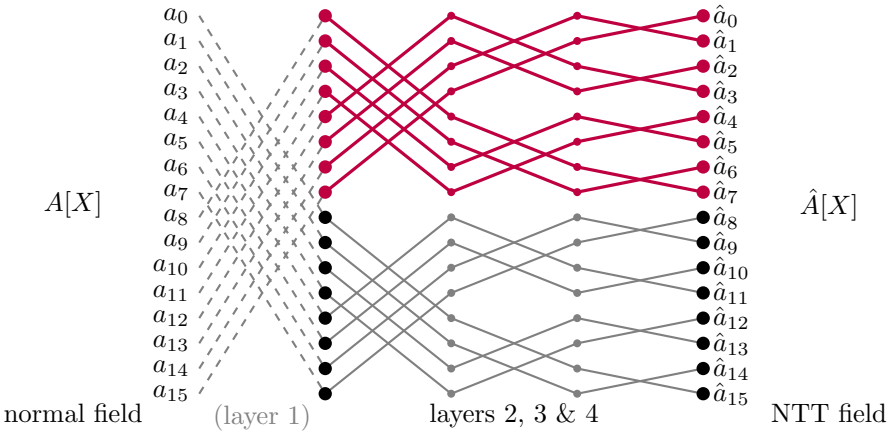


Fig. 5: 16-coefficient NTT, radix-8 for the 3 last layers

The group of layers 2 to 4 are performing radix-8 operation. It performs 12 butterflies with only 8 loads and 8 stores, instead of 24 loads and 24 stores with the radix-2 implementation. Similarly, we can fuse all the 4 layers of the 16-coefficient NTT to build the version denoted as "4", performing radix-16 operation. This version only performs 16 loads and 16 stores to perform the whole NTT, instead of 64 loads and 64 stores for the initial radix-2 version.

Thus, the more we fuse layers, the fewer memory accesses we perform. However, this optimization is limited by the number of registers in the ISA. These registers need to store the polynomial coefficients. Also, they must store the ζ values and some loop counters in addition to the coefficient values.

In ML-DSA, polynomials have 256 coefficients, so the NTT function performs 1024 butterflies in 8 layers. We implemented five versions of layer fusion:

- "1+1+1+1+1+1+1+1", shortened as "8×1", performs radix-2 operation;
- "2+2+2+2", using radix-4 fusion;
- "3+3+1+1", using radix-8 for the 6 first layers;
- "2+3+3", "3+2+3" and "3+3+2", mixing radix-4 and radix-8 fusions;
- "4+4", using radix-16 operation.

They all execute the same number of butterflies, but they perform different numbers of memory accesses. We implemented this in C, letting the GCC 13.1.0 compiler allocate registers. We measured execution time using superscalar

CVA6. The fastest fusion should be the same for other 32-register RISC-V processors as the optimization is an instruction number reduction and not a pipeline-dependent instruction scheduling. The NTT can use unsigned or signed numbers. We executed both to see which would run faster.

Results are shown in Table 2, where the execution duration is given according to the layer fusion for columns and according to the signedness for rows. In each row, the version without layer fusion is the slowest, then the implementations with 4 fused layer groups are faster, and finally that 3 fused layer groups are even faster. However, the "4+4" implementation with 2 layer groups is not the fastest, even though it should perform fewer memory accesses.

Table 2: NTT execution duration with software optimization (cycles)

Fusion	8×1	2+2+2+2	3+3+1+1	**2+3+3**	3+2+3	3+3+2	4+4
Signed	29584	21639	22059	20152	19924	21487	33183
Unsigned	22298	11304	11584	**10335**	10806	10731	13099

The "4+4" version is not the fastest in each row because many memory accesses to the program stack are performed by the compiler-emitted code for this fusion, in addition to the accesses to polynomial coefficients. Indeed, the code handles polynomial coefficients, ζ values, loop counters and temporary values. Fusing layers increases the radix, thus it increases the amount of coefficients used simultaneously. Fusing too many layers puts a too high register pressure, so the register allocator spills registers to the stack, significantly slowing down the execution. This result is in line with [ZYHK24], which showed that the greatest layer fusion which does not spill registers with RISC-V is 3 layers deep.

Thus, the fastest implementation we obtained with ML-DSA NTT layer fusion is the unsigned "**2+3+3**" fusion, running in **10335 cycles**. This will be our reference to measure the performance gain our solution brings. In the remainder of this paper, we study the acceleration of this function by implementing a new butterfly instruction in CVA6 pipeline. We use a model-driven design method, so we first model our changes to explore design space.

5 Modeling Butterfly Instructions Execution

Before implementing the butterfly instruction, we had to decide how to do it in the CVA6 pipeline. We used the model developed to design the superscalar feature of CVA6 [ACS+24] to explore design space and then implement the butterfly instruction in CVA6 pipeline. This Python model only contains the control path logic of CVA6, so it can be easily modified to explore the performance impact of microarchitectural changes. It has an accuracy over 99% on several benchmarks including CoreMark [EEM09]. It takes an execution trace from the CVA6 as the input to model performance, not behavior.

5.1 Modeling a New Instruction

This model-driven method was previously used to explore microarchitectural changes which do not change the instruction set. We adapted this method to add new instructions that are not yet implemented in CVA6. Unfortunately, the instruction that we want to model is not known by CVA6, so we cannot use it to produce an execution trace for our model yet.

Instead, we chose an instruction which exists in the ISA but is not used in the NTT algorithm so that it can be easily identified and replaced by the modeled new instruction in the execution trace loaded by the model. We took the `xor` instruction for the NTT.

Our goal is to execute the NTT function in less than 2000 cycles to make a significant difference with the 2705 cycles obtained by a state-of-the-art solution implementing butterfly instructions [MBB+23]. We first focus on the NTT, but the pipeline improvements investigated are relevant for the NTT^{-1} and the Montgomery multiplication as well.

Butterflies read three values (their two coefficients a and b, and the ζ value) and write two (the a' and b' coefficients). In the NTT and NTT^{-1} functions, a coefficient used as the input of a butterfly is not used anymore for future operations, so we can store the output coefficients of our instruction in the same registers as the input coefficients. Our butterfly instruction needs only 3 operands: the two registers of the coefficients, which are simultaneously source and destination registers, denoted as `a` and `b`; and the source register of the ζ value, denoted as `z`. So this new butterfly instruction is denoted as "`btf a, b, z`".

The model detects `xor a, b, z` instructions in its input trace and transforms them into new `btf a, b, z` instructions. The `xor a, b, z` instruction has `b` and `z` as two source registers and `a` as the destination register. To transform this instruction into a `btf` instruction, the `b` register is added to the list of destination registers, and the `a` register is added to the source registers. This new instruction with additional source and destination registers allows the model to detect data hazards (dependences). This change in the instruction set is not microarchitecture-specific.

5.2 Modeling the Microarchitecture

To explore the microarchitectural design space, we focused on structural hazards (potential congestions in the pipeline). Our circuit proposal is shown in Figure 6.

The butterfly instruction in the issue stage of the pipeline has three operands to read, in blue in Figure 6, and two operands to write, in red. Our solution leverages the superscalar processor which allows reading up to 4 operands per cycle and writing 2, initially to process two instructions simultaneously. We modeled this as follows:

- When a `btf` instruction arrives in the first place, the model blocks the next instruction unless it is a load instruction at ①. Otherwise, if the `btf` instruction arrives in the second place, it is blocked by the model so that it later comes in the first place.

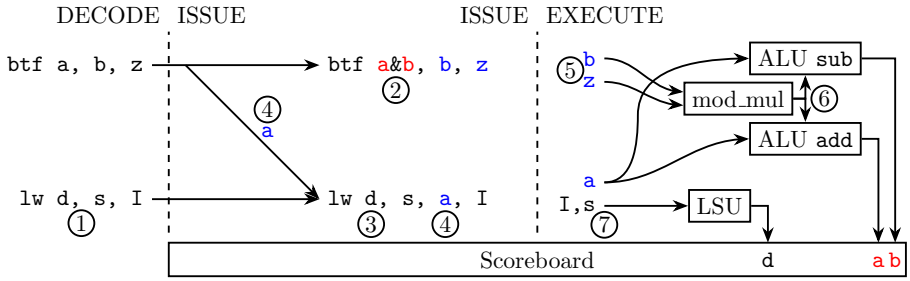


Fig. 6: Modeling of our circuit proposal

- To model the allocation of two entries in the scoreboard to store the two output values a and b at ②, we added a new attribute to the class modeling instructions, indicating the number of entries the instruction occupies. This attribute is used by the model to process the scoreboard occupancy and determine if it is full.
- As a consequence, the `btf` instruction is processed twice by the commit stage before being retired from the scoreboard: once for a and once for b.
- A third scoreboard entry can be allocated for the destination register `d` of the `lw` instruction at ③, to execute a `btf` and a `lw` in parallel.
- As the `lw` instruction only has one source register, the second register bank read port is available at ④ to read the a source operand for the `btf` instruction, as if the `lw` instruction read the 2 registers `s` and a.

In the execute stage, we add a FU, denoted as "mod_mul", performing a modular multiplication between the two inputs b and z in ⑤. The result from this modular multiplication is then forwarded to both the ALUs simultaneously at ⑥, to produce the a and b outputs, transmitted to the scoreboard via the two result busses. In parallel, the `s` register and the `I` immediate value are transmitted to the LSU at ⑦, which then returns its results for the `d` register to the scoreboard.

To model this, we programmed the model to block the `btf` instructions in the issue stage when one or both ALUs are unavailable. Also, all FUs are blocked by the `btf` instruction except the load unit to guarantee that only a `lw` instruction can be issued simultaneously.

5.3 Design Space Exploration

We reached this proposed design iteratively.

First, we modeled a version without the loads in parallel with butterflies. We modified the C code of the NTT to use `xor` instructions instead of the butterfly operations, and we compiled it for each NTT layer fusion as defined in Section 4. We executed all the resulting programs in CVA6 to get the execution traces to provide to the model. The best predicted performance was **2719 cycles**. It was

reached with the "4+4" layer fusion because the hardware butterfly does not use registers for temporary values as the whole operation is performed in the execute stage hardware.

We then optimized by-hand the compiler-emitted assembly code for the "4+4" NTT function, with the feedback from the model. We reached a predicted performance of **2440 cycles**

To go further, we parallelized `btf` instructions with memory accesses. Both loads or stores allocate a scoreboard entry. However, stores have 2 source registers, which would sum up to 5 source registers with the 3 from the butterfly. Also, there are more loads than stores in the NTT function because ζ constants are loaded in addition to the polynomial coefficients. So we modeled simultaneous execution of `btf` and `lw` instructions and optimized our assembly implementation accordingly. We obtained a new predicted performance of **2017 cycles**.

But the scoreboard of the model was now full, preventing from issuing new instructions. This was because the `btf` instruction occupies 2 scoreboard entries and the load 1, so we are now filling up to 3 entries each cycle. We added a third commit port (triple commit) into the model to remove up to 3 entries from the scoreboard each cycle and balance the pipeline. As a result, the number of cycles fell to **1747 cycles**, and the model now never triggers a stall because of a full scoreboard. This design space exploration is summarized in Table 3.

Table 3: Predicted NTT execution duration during our design space exploration

Target	Acceleration	Software	Duration (cycles)
Real CVA6	$\emptyset$	C code (2+3+3)	**10335**
Model	`btf`	C code (4+4)	**2719**
		Assembly (4+4)	**2440**
	parallel `btf`/`lw`		**2017**
	parallel `btf`/`lw` + triple commit		**1747**

By exploring design space with few modifications in the CVA6 performance model, we obtained a solution executing `lw` and `btf` instructions in parallel. It should execute NTT in **1747 cycles** instead of the current 10335 cycles. In the next section, we implement this solution in CVA6 pipeline.

6 Implementation of the Butterfly Instruction in CVA6

We implement the solution from the previous design space exploration inside CVA6 pipeline. This solution was designed for the Cooley-Tukey butterfly to accelerate the NTT. We extend it to implement the two other operations: Montgomery multiplication and Gentleman-Sande butterfly. It will allow accelerating not only the NTT, but also the pointwise multiplication and the NTT^{-1}.

6.1 Instruction Encoding and Decoding

We created three instructions, described in Table 4. The `btf.ct` and `btf.gs` instructions perform Cooley-Tukey and Gentleman-Sande butterflies, respectively. They use a custom variation of the RISC-V R-type encoding named R^{btf}-type, where the destination register `rd` is also a source register denoted as `rs2'` and the first source register `rs1` is also a destination register denoted as `rd'`.

The `btf.mm` instruction performs the Montgomery multiplication. It uses the R-type encoding. All `op`, `funct3` and `funct7` fields are the same as [MBB+23].

We modified the CVA6 decoder to decode these new instructions as R-type instructions. Then, the issue stage transforms them as R^{btf}-type instructions.

6.2 Support of R^{btf}-type Instructions into Issue and Commit Stages

The issue stage processes new R^{btf}-type instructions as described in Table 5.

First, as for any other R-type instruction, operands `a`, `b` and `z` of the `btf.ct` and `btf.gs` instructions have the role of `rd`, `rs1` and `rs2`, respectively. The source registers `b` and `z` are read from the register bank, and a scoreboard entry is allocated to receive the result to write into the destination register `a`.

Second, we highlighted in bold in Table 5 the additions specific to R^{btf}-type instructions. The `btf` instruction allocates a second scoreboard entry to store `rd'` (operand `b`) so that operands `a` and `b` are the destination registers `rd` and `rd'`. Also, it reads `rs2'` (operand `a`) from the register bank using an available read port. When a `lw` follows the `btf.ct` or `btf.gs` instruction, the issue stage reads from the register bank the base address and allocates a scoreboard entry to receive the result read from memory. The `lw` instruction also takes an immediate

Table 4: Encoding of the three proposed instructions

	7	5	5	3	5	7
R^{btf}-type	funct7	rs2	rs1/*rd'*	funct3	rd/*rs2'*	op
btf.ct a, b, z	0000000	z	b	100	a	1110111
btf.gs a, b, z	0000000	z	b	101	a	1110111

	7	5	5	3	5	7
R-type	funct7	rs2	rs1	funct3	rd	op
btf.mm dest, a, b	0000000	b	a	011	dest	1110111

Table 5: Operands when issuing a `btf.ct/gs` and a `lw` simultaneously

immediate	from register bank		scoreboard
(unused)	btf.z = rs2	btf.b = rs1	btf.a = rd **btf.b = rd'**
lw.offset = imm	**btf.a = rs2'**	lw.base = rs1	lw.dest = rd

value, but it only reads one source register, which allows reading the `a` operand of the `btf.ct` or `btf.gs` instruction without adding a read port to the register bank. Therefore, a triple entry allocation has been added to the scoreboard to store the results for `btf.a`, `btf.b` and `lw.dest`, but no read ports were added to the register bank. The `btf.mm` instructions are standard R-type instructions, so they are not processed specifically in the issue stage.

In the commit stage, we implemented the triple commit, which is simply an extension of the dual commit feature of the superscalar CVA6.

6.3 Implementation in the Execute Stage

The execute stage performs the different butterfly operations and the Montgomery multiplication with the values provided by the issue stage. We implemented in the execute stage of CVA6 the solution from Figure 6 in Section 5. We extended it to support `btf.mm` and `btf.gs` as well as `btf.ct` instructions. The resulting circuit is shown in Figure 7. The parts added to the superscalar CVA6 are highlighted in green and red. The added or extended multiplexers are annotated to indicate which instructions use their new inputs.

The first operation of the `btf.ct` instruction—which performs the $(a', b') = a \pm \zeta.b \mod q$ (cf. Equation 1)—is the modular multiplication. In Figure 7, the `b` operand goes through the ① multiplexer to reach the MUL multiplier, where it is multiplied by the `z` operand. We added a 64-bit combinatorial output to the preexisting multiplier, despite its normally pipelined architecture to return a 32-bit result after 2 cycles. This new output is connected to a new modulo q Montgomery reduction FU, which is a combinatorial hardware transcription of the modular reduction from the reference implementation of ML-DSA [NIS24b]. The reduction constant q is a hard-coded sparse constant.

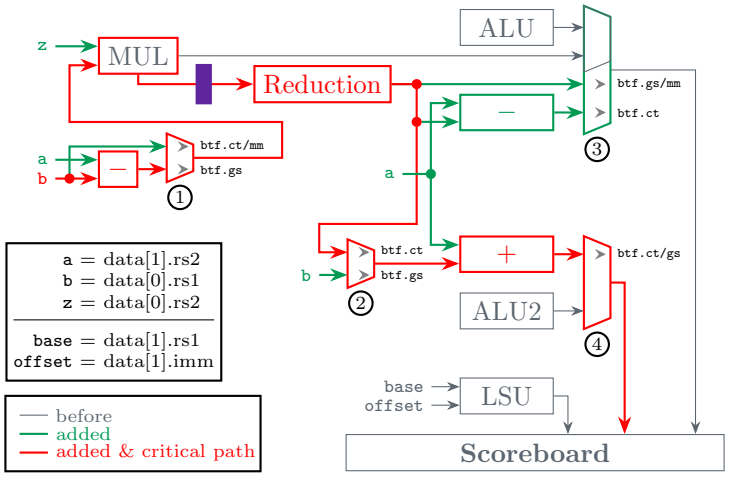


Fig. 7: Modified execute stage to perform butterfly instructions

The result of the Montgomery reduction is transmitted to the adder—via multiplexer ②—and to the subtractor to perform the addition and the subtraction with the `a` operand. These two results are transmitted to the scoreboard via multiplexers ④ and ③, respectively.

In case of a `btf.mm` instruction, the result from the Montgomery reduction is transmitted directly to the scoreboard via multiplexer ③.

Finally, in case of a `btf.gs` instruction, the `z` operand is multiplied with the new value for the `b` operand, at the end of the execution, instead of its initial value. Thus, we added a subtractor upstream from the multiplier so that the subtraction is performed before the multiplication. Its result is only used for the `btf.gs` instruction thanks to the ① multiplexer. After the multiplication and reduction, the result is transmitted to the scoreboard via multiplexer ③ after the reduction, bypassing the subtractor. This sequence of operations allows calculating the new value of the `b` operand. For operand `a`, we reuse the adder already added for the `btf.ct` instruction, with the initial value of `b` as its input thanks to multiplexer ②.

In parallel of all these operations, the LSU can perform loads from memory.

During the implementation in CVA6 pipeline, we noticed that it was stalling a lot because of a full scoreboard despite the 3 commit ports, whereas the scoreboard of the model was not full, so it did not stall as much. It meant that the detection of the scoreboard being full was not the same between the model and the RTL implementation. Indeed, in the CVA6 RTL, each scoreboard entry is left empty during one clock cycle between two uses. The reason is that the issue stage stalls when the scoreboard is currently full, ignoring that instructions being committed will free their scoreboard entries the next cycle. As a consequence, we made the scoreboard-full detector consider scoreboard entries with instructions being committed as available.

Compared with the version previously modeled, we added the register depicted in purple between the multiplication and reduction FUs. It cuts the critical path in red in Figure 7 and avoids a 26% reduction of maximal frequency. This added register introduces a new cycle of latency for our three instructions, except for the additive output of the `btf.gs` instruction.

We implemented three new instructions in the superscalar CVA6 pipeline to accelerate polynomial multiplication functions of ML-DSA. The next section evaluates our hardware implementation.

7 Hardware Implementation Results

First, we measure the performance gain our solution brings to the NTT, NTT^{-1} and pointwise multiplication. Then, we evaluate the cost of our solution in area and maximum frequency. Finally, we evaluate the power consumption variation of our solution with the NTT, compared to the execution of the same function without hardware acceleration. We compare our results with the state of the art.

Table 6: Performance results (cycles)

Base CPU	Solution	NTT	NTT^{-1}	Pt.wise mult.
Superscalar CVA6	RISC-V 32 bits	**10335**	**11333**	**2188**
	Our solution	**1913**	**1892**	**807**
	Variation	**-81 %**	**-83 %**	**-63 %**
RI5CY	32-bit RISC-V	17041	20372	4346
	[MBB+23]	2705	3979	1274
	Variation	-84 %	-80 %	-71 %
Single-issue CVA6	64-bit RISC-V	38043	46266	/
	[NDMZ+21]	18554	21375	/
	Variation	-51 %	-54 %	/
CV32E40P	32-bit RISC-V	45270	49738	/
	[YSZ+24]	1750	1925	/
	Variation	-96 %	-96 %	/

7.1 Performance Improvements for Polynomial Operations

We used the NTT, NTT^{-1} and pointwise multiplication functions to evaluate the performance gain of solution. The results are reported in clock cycles in Table 6.

Our contributions use the superscalar version of CVA6. Our reference for the NTT is the software version we optimized in Section 4. We followed the same method to optimize the NTT^{-1}. The pointwise multiplication is a simple for loop which performs a Montgomery multiplication for each coefficient. We then provide the duration of each of these three functions, optimized by-hand in assembly for our solution. The subsequent rows recall the results from the bibliography for comparison.

Our solution runs the NTT in **1913 cycles**, the NTT^{-1} in **1892 cycles** and the pointwise multiplication in **807 cycles**. Compared to [MBB+23], all our results in cycle number are lower. The difference seems to come from the use of a superscalar processor. Indeed, the relative gains are the same order of magnitude for the NTT functions. Also, we executed NTT without hardware acceleration in the single-issue configuration of CVA6, and we got **17314 cycles**, which is close to their starting point of 17041 cycles.

The NTT is longer than the NTT^{-1} for our solution. This is because the two results of the `btf.ct` used in NTT are returned after 2 cycles, whereas the `btf.gs` used in NTT^{-1} returns its first result after 1 cycle. As a result, the CVA6 scoreboard is filled by `btf.ct` instructions during the NTT, but not by `btf.gs` instructions during the NTT^{-1}. It makes the processor stall and reduces its performance.

We require fewer cycles than [NDMZ+21], while their work is also based on CVA6. The difference exists both before and after implementation.

First, before their implementation, their software implementation seems to lack NTT layer fusion, which significantly increases the number of memory accesses. Their work is based on a single-issue version of CVA6. Also, the compiler might have to add instructions to work with 32-bit words on a 64-bit platform.

Second, their butterfly instructions return both coefficients on a single 64-bit value, so additional instructions are needed to unpack the coefficients into distinct registers. Also, the twiddle factors are set with additional instructions, instead of being provided as an input operand of the butterfly instructions.

Finally, the performance we obtained for NTT and NTT^{-1} is the same order of magnitude as a solution using SIMD instructions [YSZ+24].

7.2 Area and Maximum Frequency

We performed gate-level syntheses of the reference superscalar CVA6 with and without our solution using Synopsys Design Compiler to determine its cost in maximal frequency and in area. The results are shown in Table 7.

We compared our solution to the reference superscalar CVA6, at the maximal frequency of each implementation. The cost in area is **4.7 kG** and the cost in maximal frequency is **2.8%**. It includes the logic added to decode, issue and execute the new instructions, as well as the combinatorial output added to the multiplier and the additional third commit port. The critical path is in the issue stage, where we added logic to move operands and allocate more entries in the scoreboard, as shown in Table 5.

We then compared these results with those from [MBB+23] at 400 MHz. Their solution is not on the critical path. They get a cost in area of 6.8 kG, which is a little more than ours, but they provide more features with butterflies for ML-KEM whereas we only implemented butterflies for ML-DSA.

7.3 Power Consumption of our Solution

Finally, we performed a power analysis of CVA6 pipeline using Synopsys PrimeTime on the NTT, at the maximal frequency. The results are shown in Table 8.

Our solution increases mean power consumption by **21%**. However, the total energy consumption is reduced by **78%** because our solution reduces execution

Table 7: Area and maximum frequency results

Implementation	Area (cost)	Cost in max. freq.
Superscalar CVA6	109.4 kG (ref.)	(ref.)
Our solution	114.1 kG (+4.7 kG)	-2.8%
RI5CY	43.7 kG (ref.)	(ref.)
[MBB+23]	49.8 kG (+6.8 kG)	Not on critical path

Table 8: Power and energy consumption for NTT at maximum frequency

Implementation	Mean power cons. var.	Energy cons. var.
Superscalar CVA6	(ref.)	(ref.)
Our solution	+21%	-78%

time more than it increases power consumption. These results include the reduced cycle number and the increased clock period, according to Equation 4.

$$E = P \times \Delta t = P \times \text{clock cycle number} \times \text{clock cycle duration} \quad (4)$$

Instructions for NTT were implemented in a 64-bit single-issue CVA6 by [NDMZ+21]. They performed power measurements on an FPGA, using the whole algorithms as benchmarks. For the ML-DSA key generation, they reduced the power consumption by 6.3% and the duration by 15.0%, resulting in a reduction in energy consumption of 20.3%. Their power consumption decreased whereas ours increased. They indicate that the decrease in power consumption could be related to a reduction in memory accesses. However, the memory is not included in our consumption measurements to focus on CVA6 pipeline. It could explain why we do not observe the same power consumption variation. Also, their energy consumption reduction is lower than ours. It could be because they executed the whole key generation and not only the NTT functions, so the relative performance gain delivered by the NTT acceleration is much lower, which implies a lower reduction in energy consumption.

8 Discussion

We stuck with the method from [ACS+24] where everything is tested into the model before starting implementation. However, it would probably have been simpler to circumvent this rule for the decoding of the new instructions. In this work, we chose an existing instruction from the instruction set which is not used by our target program, so that CVA6 can execute it to emit the execution trace that we feed to the model. We then modified the model to transform this real instruction into a new fictitious one. Instead, we could have first modified the decoder inside the CVA6 processor and used a real new instruction from start, so the model could simply read this new instruction from its input trace.

This work focused on the ML-DSA butterflies, but it would be possible to do the same with the ML-KEM butterflies. To support both, we could add a second Montgomery reduction module into the execute stage, with the modulus constant from ML-KEM instead of the one from ML-DSA. Then, by decoding new instructions for ML-KEM, we could support butterfly instructions for both ML-KEM and ML-DSA.

The NTT function is slowed down in our implementation because the scoreboard is filled with `btf.ct` instructions, which return 2 results after 2 cycles. We could have increased the size of the scoreboard to host more of these `btf.ct` instructions. However, it would have increased the area, so we have chosen not to do so and we accept the performance reduction.

Alternatively, we could remove the register we added in the execute stage to cut the critical path. If a high clock frequency is not required, removing this register would result in an execution of the NTT in 1752 cycles, which is closer to the 1747 cycles predicted by the model. However, the maximal clock frequency would be reduced by 26%.

9 Conclusion

Polynomial multiplications represent the second most time-consuming part of ML-DSA, after Keccak. They use NTT and NTT^{-1} extensively. The basic blocks of NTT and NTT^{-1} are the Cooley-Tukey and Gentleman-Sande butterflies, which both use a Montgomery multiplication. The butterflies are applied to the coefficients of a polynomial by layers.

In this paper, as a preliminary work, we explored various NTT layer fusions on RISC-V to find the better software implementation of the NTT and NTT^{-1} functions. They were executed in respectively **10328** and **11333** cycles.

We then followed the model-driven design method from [ACS+24]. We extended it to implement new instructions into the superscalar CVA6. These instructions are Cooley-Tukey and Gentleman-Sande butterflies, and the Montgomery multiplication. The butterflies are special as they read all their three operands and write two of them.

This modeling phase allowed us to explore the design space, focusing on the NTT function and its Cooley-Tukey butterflies. Especially, it indicated that our initial design proposal would execute the NTT function in 2440 cycles, so we searched for a better solution, and so before we started modifying the CVA6 pipeline. The design space exploration led to a solution executing memory accesses and butterflies in parallel. The model predicted that this solution would execute the NTT in 1747 cycles.

So we implemented this solution to execute Cooley-Tukey butterfly instructions into the pipeline of a superscalar CVA6. We extended it to also support Gentleman-Sande butterfly instructions as well as Montgomery multiplication instructions. The implementation consisted in modifying: the issue stage to read the three operands for butterfly instructions and to allocate two scoreboard entries for their two results; the execute stage to perform new calculation while reusing the multiplier and the preexisting result busses; and the commit stage to perform a third commit per cycle. Still, we allow the execution of a load instruction in parallel with any of our new instructions. During our changes in CVA6 pipeline, we used the model as a reference to check if our actual implementation matches the performance predicted by the model. This comparison with the model resulted in an update of the "scoreboard full" condition detection.

With these modifications and the addition of a register in the execute stage to cut the critical path, we reached a performance of **1913 cycles** for the NTT, and **1892 cycles** for the NTT^{-1}, with an area cost of **4.7 kG**. The NTT and NTT^{-1} are sped up by at least a **factor of 5** compared to our best software implementation. Our solution reduces the energy consumption of the NTT by **78%** compared to the same execution without our new instructions, thanks to the runtime reduction. The modified version of CVA6 is still completely functional and can boot Linux.

The performance gains and area costs of our solution are close to similar state-of-the-art implementations. In particular, our final performance is better because our solution is based on a superscalar processor which allows the execution of two instructions simultaneously.

References

[ACS+24] Côme Allart, Jean-Roch Coulon, André Sintzoff, Olivier Potin, and Jean-Baptiste Rigaud. Using a performance model to implement a superscalar cva6. In *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, CF '24 Companion, page 43–46. Association for Computing Machinery, 2024.

[ARM04] ARM. AMBA AXI Protocol Specification v1.0. https://developer.arm.com/documentation/ihi0022/b, 2004.

[ASD25] George Alexakis, Dimitrios Schoinianakis, and Giorgos Dimitrakopoulos. High-performance pipelined ntt accelerators with homogeneous digit-serial modulo arithmetic. 2025.

[DDMV24] Alessandra Dolmeta, Stefano Di Matteo, and Emanuele Valea. Exploring Accelerator Integration with Core-V eXtention InterFace (CV-X-IF) for Kyber. RISC-V Summit Europe, June 2024. Poster.

[DH76] W. Diffie and ME. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[DMM24] Alessandra Dolmeta, Maurizio Martina, and Guido Masera. Athos: A hybrid accelerator for pqc crystals-algorithms exploiting new cv-x-if interface. *IEEE Access*, 12:182340–182352, 2024.

[EEM09] EEMBC. Coremark. https://www.eembc.org/coremark/, 2009.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact dilithium implementations on cortex-m3 and cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, Dec. 2020.

[KSFS22] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on risc-v with hardware acceleration. Cryptology ePrint Archive, Paper 2022/538, 2022. https://eprint.iacr.org/2022/538.

[LTQ+24] Lu Li, Qi Tian, Guofeng Qin, Shuaiyu Chen, and Weijia Wang. Compact Instruction Set Extensions for Dilithium. *ACM Transactions on Embedded Computing Systems*, 23(2):1–21, March 2024.

[MBB+23] Konstantina Miteloudi, Joppe Bos, Olivier Bronchain, Björn Fay, and Joost Renes. PQ.v.ALU.e: Post-quantum RISC-v custom ALU extensions on dilithium and kyber. Cryptology ePrint Archive, Paper 2023/1505, 2023.

[MSS24] Stefano Di Matteo, Ivan Sarno, and Sergio Saponara. CRYPHTOR: A Memory-Unified NTT-Based Hardware Accelerator for Post-Quantum CRYSTALS Algorithms. *IEEE Access*, 12:25501–25511, 2024.

[NDMZ+21] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A risc-v post quantum cryptography instruction set extension for number theoretic transform to speed-up crystals algorithms. *IEEE Access*, 9:150798–150808, 2021.

[NIS15] NIST. FIPS 203 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015.

[NIS16] NIST. NIST IR 8105; report on post-quantum cryptography. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2016.

[NIS24a] NIST. FIPS 203 Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2024.

[NIS24b] NIST. FIPS 204 Module-Lattice-Based Digital Signature Standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2024.

[Ope21] OpenHW Group. OpenHW Group Specification: Core-V eXtension interface (CV-X-IF) - Development. https://docs.openhwgroup.org/projects/openhw-group-core-v-xif, 2021.

[Ope24] OpenHW Group. Cva6 risc-v cpu. https://github.com/openhwgroup/cva6, 2024.

[RIS24] RISC-V International. Spike risc-v isa simulator. https://github.com/riscv-software-src/riscv-isa-sim, 2024.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable SCA countermeasures against single trace attacks for the NTT - a performance evaluation study over kyber and dilithium on the ARM cortex-m4. Cryptology ePrint Archive, Paper 2020/1038, 2020.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[RVBB24] Rafael Carrera Rodriguez, Emanuele Valea, Florent Bruguier, and Pascal Benoit. Hardware implementation and security analysis of local-masked NTT for CRYSTALS-kyber. Cryptology ePrint Archive, Paper 2024/1194, 2024.

[Sha23] Sahil Sharma. The NTT and residues of a polynomial modulo factors of $x^{2^d} + 1$. Cryptology ePrint Archive, Paper 2023/1812, 2023.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. pages 124–134, 1994.

[YSZ+24] Zewen Ye, Ruibing Song, Hao Zhang, Donglong Chen, Ray Chak-Chung Cheung, and Kejie Huang. A highly-efficient lattice-based post-quantum cryptography processor for iot applications. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):130–153, Mar. 2024.

[ZB19] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[ZYHK24] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):632–655, December 2024.