PQ.V.ALU.E: Post-Quantum RISC-V Custom ALU Extensions on Dilithium and Kyber

Konstantina Miteloudi[1,2], Joppe W. Bos[2], Olivier Bronchain[2], Björn Fay[2], and Joost Renes[2]

[1] Radboud University, The Netherlands
konstantina.miteloudi@ru.nl
[2] NXP Semiconductors
firstname.lastname@nxp.com

Abstract. This paper explores the challenges and potential solutions of implementing the recommended upcoming post-quantum cryptography standards (the CRYSTALS-Dilithium and CRYSTALS-Kyber algorithms) on resource constrained devices. The high computational cost of polynomial operations, fundamental to cryptography based on ideal lattices, presents significant challenges in an efficient implementation. This paper proposes a hardware/software co-design strategy using RISC-V extensions to optimize resource utilization and speed up the number-theoretic transformations (NTTs). The primary contributions include a lightweight custom arithmetic logic unit (ALU), integrated into a 4-stage pipeline 32-bit RISC-V processor. This ALU is tailored towards the NTT computations and supports modular arithmetic as well as NTT butterfly operations. Furthermore, an extension to the RISC-V instruction set is introduced, with ten new instructions accessing the custom ALU to perform the necessary operations. The new instructions reduce the cycle count of the Kyber and Dilithium NTTs by more than 80% compared to optimized assembly, while being more lightweight than other works that exist in the literature.

Keywords: CRYSTALS-Dilithium · CRYSTALS-Kyber · NTT · RISC-V · ISA extension.

1 Introduction

Research in the field of quantum computing has advanced tremendously in recent years, which has brought significant changes to the field of cryptography. The security of traditional public-key cryptography (PKC) is based on hard mathematical problems such as the factorization of large integers and the calculation of discrete logarithms, which are difficult to solve on a classic computer (i.e., in (sub-)exponential time). However, these problems would be solved in polynomial time on a large-scale quantum computer, using Shor's algorithm [22]. This has led to a growing interest in post-quantum cryptography (PQC). Post-quantum cryptography refers to algorithms that run on the same classical hardware as traditional PKC, but are deemed secure against quantum adversaries.

Since 2016, PQC algorithms are in the process of being standardized by the National Institute of Standards and Technology (NIST) [18]. In July 2022, NIST recommended two primary algorithms to be implemented for most use cases: CRYSTALS-Kyber [21] and CRYSTALS-Dilithium [16]. Dilithium is a digital signatures algorithm, for example used to verify the authenticity and integrity of a message or document and to secure digital transactions. Kyber is a Key-Encapsulation Mechanism (KEM) and it is used to securely establish keys for a variety of applications, such as secure web browsing and email encryption. Both Dilithium and Kyber are based on hard problems coming from the theory of *ideal* lattices.

Lattice-based cryptographic algorithms have gained significant attention due to their security, efficiency and versatility. Although their efficiency is comparable to their classical counterparts on many platforms, their implementation can present challenges on resource-constrained systems. Looking at the performance numbers on the pqm4 benchmarking framework [13] that compares software implementations results on Arm Cortex-M4 cores, it can, for example, be seen that more than 80% of the runtime of Dilithium signature verification is spent in Keccak (SHAKE-256). For Kyber decapsulation more than 75% of the time is due to Keccak. Although this does not include side-channel protections beside executing in a constant running time, and while these numbers may differ for RISC-V, it shows that Keccak is a dominant factor in both algorithms. Because (hardened) Keccak accelerators are already well-researched (e.g., [6,23]), we do not investigate them in this work. Instead we focus on the next bottleneck that arises, namely polynomial operations. The Number-Theoretic Transform (NTT), a specialized form of the Discrete Fourier Transform (DFT), is used by many lattice-based algorithms to reduce the time complexity of polynomial multiplication. Despite this optimization, the polynomial arithmetic remains time-consuming, making efficient hardware implementations crucial, especially in constrained settings.

Resource-constrained devices, characterized by their limited computational capabilities, energy resources, and memory, pose unique challenges for the implementation of cryptographic algorithms. Typical examples include Internet of Things (IoT) devices, such as sensors, healthcare devices, automotive processors, and other embedded systems in vehicles and industrial control systems. The importance of implementing PQC algorithms with low area in mind lies in the need to accommodate the constraints of these devices. However, these implementations are expensive due to the complexity of the operations involved. This has led to exploration of hardware/software (HW/SW) co-design strategies, which aim to combine the high-speed performance of hardware implementations with the flexibility of software designs.

In the context of HW/SW codesign, RISC-V presents a promising approach. RISC-V is an open-source instruction set architecture (ISA) that supports custom extensions, providing a flexible platform for the implementation of various algorithms, including lattice-based algorithms. The support for custom extensions allows developers to add specialized instructions tailored to specific ap-

plications. These instructions can operate on the typical processor datapath or on a co-processor connected to the main processor as a peripheral, or they can operate on a modified processor datapath that integrates custom accelerators.

Contributions. The objective of this work is to design and implement a custom hardware accelerator for the NTT and other polynomial operations of Dilithium and Kyber. The primary aim is to optimize resource utilization of this accelerator and to minimize the number of changes to the RISC-V core, while increasing the speed of NTT computations for both Dilithium and Kyber. In short, in this work we introduce PQVALUE and summarize our contributions as follows:

- A lightweight custom ALU that is tailored to the requirements of the NTT computations for both schemes. The ALU is seamlessly integrated into a 4-stage pipeline 32-bit RISC-V processor and it supports modular operations (addition, subtraction and multiplication) as well as the NTT and inverse NTT butterfly operations (Cooley-Tukey and Gentleman-Sande).
- An extension to the RISC-V ISA, by introducing ten new instructions, five for each scheme. These instructions access the custom ALU and perform the operations mentioned above. Each instruction takes one clock cycle to execute.
- We show that our ALU is the smallest to appear in the literature and only increases the resource utilization of the RI5CY core by 13–17% (depending on the operating frequency). We measured our design for specific clock frequencies (100–400 MHz), similar to those used in real-world microprocessors, and confirmed that our custom RISC-V core with the new tailored ALU operates at these frequencies without any degradation compare to the original RISC-V core.
- We implement and benchmark the various operations and confirm that PQVALUE decreases the cycle count of the Kyber and Dilithium NTTs by more than 80% compared to optimized assembly, and significantly improves the runtime of the full algorithms.

2 Preliminaries

2.1 Number-Theoretics Transformations (NTTs)

The introduction of Ring- and Module-LWE has moved the arithmetic operations from general lattices to polynomial rings of the form $R = \mathbb{Z}_q[X]/(X^n + 1)$ for a prime q and integer n. The critical operations are polynomial addition and subtraction of linear complexity of n, and polynomial multiplication. The latter is a more involved operation that naïvely has quadratic complexity using schoolbook multiplication, but can be sped up using a variety of techniques such as Karatsuba, Toom-Cook, and *Number-Theoretic Transformations* (NTTs). The latter is not possible in generic polynomial rings, which is why the lattice-based constructions choose R to be of this specific form. The prime q is selected to be small, e.g., 12-bit for Kyber and 23-bit for Dilithium, while n is selected to be

a power of 2 (typically 256). Moreover, q is selected such that it has a root of unity ζ of sufficiently large order. That is, of order n or $2n$.

An NTT can come in various forms or shapes, depending on the choice of root of unity and the implementation. For example, the Kyber ring contains only n-th roots of unity while the Dilithium ring contains $2n$-th roots of unity. This leads to slightly different strategies for implementing the NTT. For Kyber, given an input polynomial f, the NTT computes (up to re-ordering)

$$\begin{aligned} R &\to \prod_{i=0}^{127} \mathbb{Z}_q[X]/(X^2 - \zeta^{2i+1}) \\ f &\mapsto \left(f \bmod (X^2 - \zeta), \ldots, f \bmod (X^2 - \zeta^{255})\right) , \end{aligned}$$

which is an isormophism via the Chinese Remainder Theorem (CRT). The inverse NTT computes the inverse isomorphism. For Dilithium, the NTT instead computes (up to re-ordering)

$$\begin{aligned} R &\to \prod_{i=0}^{255} \mathbb{Z}_q[X]/(X - \zeta^{2i+1}) \\ f &\mapsto \left(f \bmod (X - \zeta), \ldots, f \bmod (X - \zeta^{511})\right) , \end{aligned}$$

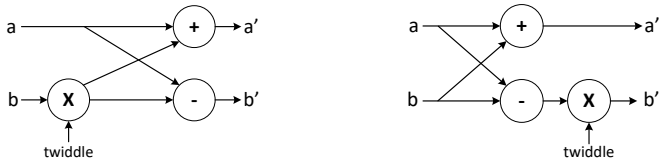
and the inverse computes the inverse isomorphism according to the CRT.

The actual implementation of the operation closely mimics the strategies of Discrete Fourier Transforms (DFTs). The NTT can be constructed as a sequence of $\log_2(n)$ layers containing exactly $n/2$ *butterflies*, where each butterfly consists of a base field multiplication, addition and subtraction. These are known as Cooley-Tukey (CT) butterflies for the forward NTT, named after the authors that first introduced the butterfly structure. It is graphically described in Figure 1a. The total complexity of the algorithm is therefore $n\log(n)$, which is significantly faster than other existing methods. The sequence of butterflies in the forward NTT introduces a re-ordering of the coefficients that deviates from the CRT isomorphism. This is not a problem as long as this is included in the inverse NTT operation (typically without performance loss). For example, this can be done by changing the butterfly structure from Cooley-Tukey to Gentleman-Sande (GS) for the inverse NTT. The GS butterfly is described in Figure 1b. Eventually, we note that Kyber NTT only uses $\log_2(n) - 1 = 7$ layers while Dilithium uses $\log_2(n) = 8$ layers because of the different isomorphism described above.

2.2 Barrett reduction

Barrett reduction [4] is an efficient method to compute modular reductions using precomputed data which only depends on the modulus used. Given a positive odd modulus q such that $2^{n-1} < q < 2^n$ and some input value $0 \leq c < q^2$ then one can compute $c' = c \bmod q$ as

$$c' = c - m \cdot q, \text{ where } m = \left\lfloor \frac{c}{q} \right\rfloor .$$



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

Fig. 1: Butterflies blockdiagrams

Algorithm 1 Barrett Reduction in Dilithium

Input: $0 \leq x < 8\,380\,417^2$,
Output: $z = x \mod 8\,380\,417$
1: $t \leftarrow (x \ll 23) + (x \ll 13) + (x \ll 3) - x$
2: $t \leftarrow t \gg 46$
3: $t \leftarrow (t \ll 23) - (t \ll 13) + t$
4: $z \leftarrow x - t$
5: **if** $z \geq 8\,380\,417$ **then**
6: $\quad z \leftarrow z - 8\,380\,417$
7: **return** z

Algorithm 2 Barrett Reduction in Kyber

Input: $0 \leq x < 3329^2$,
Output: $z = x \mod 3329$
1: $t \leftarrow 5039 \cdot x$
2: $t \leftarrow t \gg 24$
3: $t \leftarrow (t \ll 11) + (t \ll 10) + (t \ll 8) + t$
4: $z \leftarrow x - t$
5: **if** $z \geq 3329$ **then**
6: $\quad z \leftarrow z - 3329$
7: **return** z

The idea behind Barrett reduction is inspired by a technique of emulating floating point data types with fixed precision integers: namely, *approximate* $m = \lfloor c/q \rfloor$ by

$$m_1 = \left\lfloor \frac{c}{2^{2n}} \cdot \left\lfloor \frac{2^{2n}}{q} \right\rfloor \right\rfloor = \left\lfloor \frac{c \cdot \mu}{2^{2n}} \right\rfloor ,$$

where $\mu = \lfloor 2^{2n}/q \rfloor$ is a pre-computed constant. Since $m - 1 \leq m_1 \leq m$ this approximation is almost correct while only efficient integer divisions by powers of two (i.e., right shifts) are required.

Barrett modular reduction for the prime moduli used in Dilithium and Kyber is shown in Algorithm 2 and Algorithm 1, respectively. Using the special form of the modulus was also used in [3]. The Barrett reduction for Kyber is shown in Algorithm 2. One has $q = 3329$, $2^{11} < q < 2^{12}$ and $\mu = \lfloor 2^{24}/q \rfloor = 5039$. The computation of $\lfloor c \cdot \mu / 2^{24} \rfloor$ is performed in Line 1 and 2. Line 3 and 4 compute $c' = c - m_1 \cdot q$ where the multiplication by $q = 3329 = 2^{11} + 2^{10} + 2^8 + 2^0$ is performed as a sequence of shifts and additions.

The Barrett reduction for Dilithium is shown in Algorithm 1. One has $q = 8\,380\,417$, $2^{22} < q < 2^{23}$ and $\mu = \lfloor 2^{46}/q \rfloor = 8\,396\,807 = 2^{23} + 2^{13} + 2^3 - 1$. The computation of $\lfloor c \cdot \mu / 2^{46} \rfloor$ is performed in Line 1 and 2 (where the multiplication by μ makes use of the special form). Line 3 and 4 compute $c' = c - m_1 \cdot q$ where the multiplication by $q = 8\,380\,417 = 2^{23} - 2^{13} + 1$ is done as a series of shifts and additions.

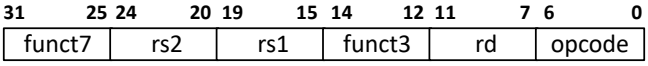


Fig. 2: RISC-V R-type instruction format

2.3 RISC-V

RISC-V defines a flexible and extensible Instruction Set Architecture (ISA). The ISA defines small instruction sets that must be implemented by all the RISC-V cores along with additional extensions that can optionally be implemented depending on the target application. This makes this ISA portable to a large variety of applications ranging from embedded system to high-performance computers. The ISA does not define how instructions must be implemented, or their latency. Compared to Arm architectures, cryptographic implementations on RISC-V generaly take advantage of its large register file, but suffer from the limited instruction sets when the appropriate extensions are not implemented [23].

For the purpose of this work, we focus on the 32-bit RISC-V architecture where both the datapath and the instructions operate on 32-bit words. These architectures have 32 registers from which 27 can be freely used. We have a closer look at the R-type instructions that are highlighted in Figure 2. The instruction is uniquely defined by the three fields `opcode`, `funct3` and `funct7`, that combine to a total of 17 control bits. Each instruction has a specific combination of control bits and the CPU will use them to execute the corresponding operation. The R-type instructions have 2 source and 1 destination registers that are the registers that the instruction must operate on. These are encoded as `rs1`, `rs2` and `rd` respectively in Figure 2. Each of these registers are encoded on 5 bits as 32 different registers could be accessed.

2.4 Related Work

Research into PQC hardware accelerators has increased in recent years, with a growing number of published works addressing its various aspects. There have been hardware implementations of the full scheme of Dilithium [20,15,26,5,25,12] and Kyber [7,24,19]. They use various optimizations in order to have small and fast designs while also exploring how these designs can evolve into co-processors capable of running the full schemes. Some studies have also focused on designing co-processors that can accelerate more than one scheme. Aikata, Mert, Imran, Pagiliarini, and Roy [1] create a unified architecture for both Dilithium and Kyber by designing common building blocks for polynomial multiplication and SHAKE. Moreover, Banerjee, Ukyab and Chandrakasan [3] create custom RISC-V instructions for polynomial arithmetic and sampling to support a co-processor for Frodo, NewHope, qTESLA, Kyber and Dilithium. All publications mentioned above connect their hardware accelerators as peripherals to the main processor. However, these peripherals, while beneficial, can be slowed down by the bus of the processor and the transfer of data between the different components. An alternative approach is to integrate the hardware accelerator directly into the

datapath of the processor. For example Karl, Schupp, Fritzmann and Sigl [14] accelerate the NTT transformation and pointwise multiplications as well as the functions SHAKE128 and SHAKE256 for Dilithium and Falcon. The NTT accelerator is connected as peripheral and it needs its own memory in order to reduce the transaction cost, but this increases also the resources, while the Keccak round function for SHAKE is incorporated into RISC-V datapath.

There are two publications where a custom ALU with ISA extensions is implemented to accelerate Dilithium and Kyber that can be directly compared with our own. Fritzmann, Sigl and Sepúlveda [8] present an enhanced RISC-V architecture that integrates tightly coupled accelerators directly into the processor's pipeline datapath to speed-up NewHope, Kyber, and Saber while extending the RISC-V ISA by twenty-nine new instructions. Nannipieri, Di Matteo, Zulberti, Albicocchi, Saponara and Fanucci [17] introduce an extension to the RISC-V ISA to facilitate the NTT operations of the Dilithium and Kyber cryptographic schemes.

3 Extensions for Polynomial Arithmetic

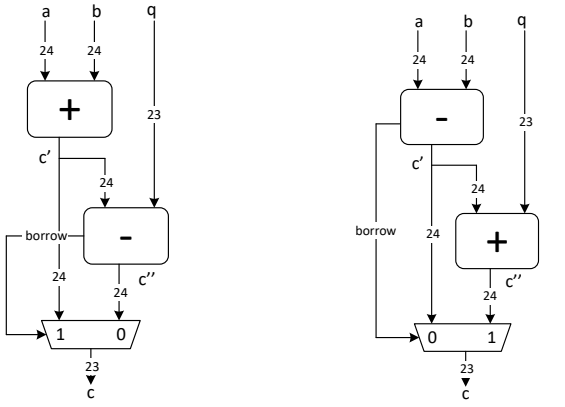
In this section, we describe our dedicated ALU for polynomial arithmetic including adders, subtractors, multipliers and butterflies. The main challenge to support polynomial arithmetic for both Dilithium and Kyber is the difference in prime size. For Dilithium we have the 23-bit prime $q = 8\,380\,417$ while for Kyber, we have the 12-bit prime $q = 3\,329$. As our goal is to have a small hardware design, we re-use the same base adders and multipliers for both schemes.

Representation: When implementing arithmetic modulo q, the question that arises is how to represent the values. Optimized (embedded) software typically use the signed interval $[-(q-1)/2, (q-1)/2]$, as it may have advantages for the number of reductions that take place in the butterflies [13,11,10]. Instead, we choose the canonical interval $[0, q-1]$ as it leads to slightly simpler hardware. Of course, the choice does not have a large impact as transforming between representations is trivial and cheap.

3.1 Single-Cycle Modular Arithmetic

First, we describe the modular addition and the modular subtraction units. Both have a similar structure as detailed in Figure 3. We start by describing the addition for Dilithium, then its subtraction and finally we describe how this can be re-used for Kyber.

The modular adder is based on an addition and a subsequent subtraction approach as shown in Figure 3a. Concretely, both inputs `a` and `b` are first added together with a 24-bit unsigned integer adder to obtain $\texttt{c'} = \texttt{a} + \texttt{b}$. Next, we perform a 24-bit unsigned subtraction such that $\texttt{c''} = \texttt{c'} - q$. The final result `c` is then selected conditionally: $\texttt{c} = \texttt{c'}$ if $\texttt{c'} < q$ (i.e., a borrow occurred in



(a) $\mathtt{c} = \mathtt{a} + \mathtt{b} \mod \mathtt{q}$ (b) $\mathtt{c} = \mathtt{a} - \mathtt{b} \mod \mathtt{q}$

Fig. 3: Modular addition and subtraction circuits.

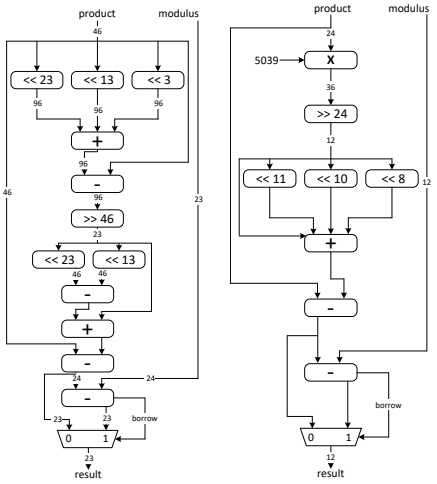
the subtraction), and $\mathtt{c} = \mathtt{c''}$ otherwise. After the selection, only the 23 least significant bits are needed, as it guarantees that $\mathtt{c} \in [0, q-1]$.

The modular subtraction is described in Figure 3b and works in a similar fashion. Namely, first compute $\mathtt{c'} = \mathtt{a} - \mathtt{b}$ using a 24-bit unsigned subtractor. The output signal $\mathtt{c}$ is then set to $\mathtt{c} = \mathtt{c'} + q$ if $\mathtt{c'} < q$ (i.e., a borrow occurred in the subtraction) and $\mathtt{c} = \mathtt{c'}$ otherwise. The selection between both cases is performed based on the borrow bit and a multiplexer. Again, only the 23 least significant bits are needed afterwards.

Exactly the same circuit can be used for all $q < 2^{23}$, and hence in particular for both Dilithium and Kyber. The only difference is in the position of the borrow bit. Interestingly, as additions and subtractions are based on similar designs, the same circuit could be shared also between modular addition and subtraction. However, as detailed in Section 3.2, one addition and one subtraction needs to be performed in parallel to have butterflies in a single cycle. Hence, we use independent circuits for modular addition and subtraction.

As detailed in Section 2.2, modular multiplication $\mathtt{c} = \mathtt{a} \cdot \mathtt{b} \mod q$ is performed in two steps. The first step is an integer multiplication where both inputs $\mathtt{a}$ and $\mathtt{b}$ are multiplied. In order to reduce the cost of the integer multiplier, we share it between the modular multiplications used in both Dilithium and Kyber. This means adding a 23×23-bit multiplier with 46-bit output. As Kyber requires only a 12×12-bit multiplication with 24-bit output, this is far from an optimal multiplier size. We discuss possible alternatives at the end of this section.

The second step for modular multiplication is to perform the Barrett reduction. The Barrett reduction circuits tailored for both Dilithium and Kyber are shown in Figure 4. These circuits are the direct implementation of Algorithm 1 and Algorithm 2. While several reduction methods exist, we chose Barrett reduction. It avoids expensive division operations and the special form of the modulus in Dilithium and Kyber allows the multiplication to be performed as a sequence



(a) Dilithium (b) Kyber

Fig. 4: Dedicated Barrett reductions hardware from [3].

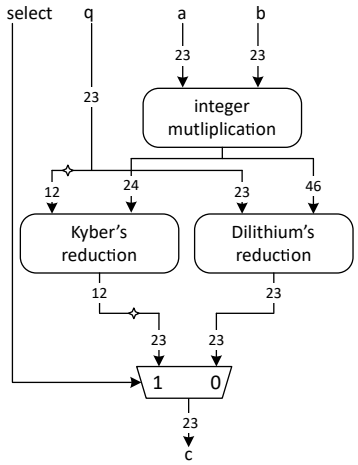


Fig. 5: Modular multiplication unit. Computes $\texttt{c} = \texttt{a} \cdot \texttt{b} \bmod q$ for Dilithium and Kyber primes q.

of shifts and additions, reducing the cost in time and area. For Dilithium, the Barrett reduction has a logical depth of 7 additions/subtractions. For Kyber, a total for 13 additions/subtractions is needed, of which 8 are for the constant multiplication by 5039. Indeed, it can be decomposed into a sum of powers of two as follows: $5039 = 2^{12} + 2^9 + 2^8 + 2^7 + 2^5 + 2^3 + 2^2 + 2^1 + 2^0$. We rely on the synthesizer's optimization strategies to efficiently implement this, allowing it to choose between structures like adder trees or utilizing carry chains. In case of an adder tree is used, the total logical depth of 8 adders is needed for the reduction (with 4 for the constant multiplication). We note that other options to perform the modular reduction are also possible (see [15]), which target FPGA implementations.

Finally, as the 23×23 base multiplier is shared for both Dilithium and Kyber, both Barrett reductions are connected to it. The result of the modular multiplication is then chosen via a select signal and a multiplexer as illustrated in Figure 5. As mentioned above, this base multiplier is not optimal for Kyber, leading to the underutilization of one DSP block when Kyber is executed. While a potential solution could be a vector instruction that executes two multiplications for the Kyber prime in parallel, thereby doubling the throughput and fully utilizing the DSP block, we chose the simpler shared multiplier design without the additional logic for parallel multiplications. This decision prioritizes area savings and design simplicity, especially given the differing performance requirements between Kyber and Dilithium. Another interesting approach would be to re-use the 32×32-bit multiplier from the original ALU if available, leading

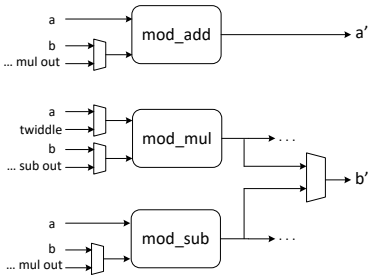


Fig. 6: Custom ALU unit.

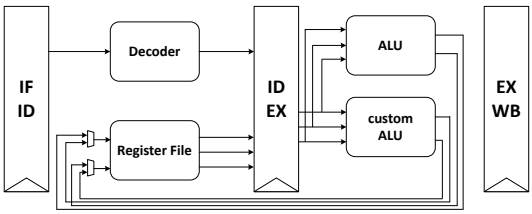


Fig. 7: Modified datapath.

to an even smaller design. As the performance would depend strongly on the availability and latency of the instructions in the RISC-V core, it is not clear what the benefit would be. Therefore, we take the general approach and include our own multiplier.

3.2 Single-Cycle Butterflies

From the hardware for modular arithmetic, one can efficiently implement both Cooley-Tukey and Gentleman-Sande butterflies by connecting them together as illustrated in Figure 1. The simplest option is to use a sequence of assembly instructions for the addition, subtraction, and multiplication (see Section 3.4 and Figure 8c) with a latency of 3 cycles. In this section, we also describe the option to directly connect these operations within the custom ALU in order to perform a butterfly in a single cycle.

From Figure 1, we observe that the Cooley-Tukey butterfly first performs the multiplication followed by two independent addition and subtraction operations. Conversely, the Gentleman-Sande butterfly performs the addition and subtraction before the multiplication with the twiddle factor. As a result, in order to use a single unit for all three operations, multiplexers are needed. The resulting connections between the different operations are detailed in Figure 6. For the Cooley-Tukey butterfly, the multiplier is connected to `b` and `twiddle`, and both the subtraction and addition are connected to `a` and the output of the multiplication. For Gentleman-Sande, the output of the subtraction is directly connected to the multiplier that takes the twiddle factor as its second input.

Finally, we note that to perform a single-cycle butterfly, three read and two write ports on the register files are required. A typical register file has two read ports and one write port, resulting in two clock cycles for reading the inputs and two clock cycles for writing the outputs of the butterfly operations. However, PULPino's register file has three read and two write ports, which allows us to read and write in a single clock cycle.

3.3 Integration and Additional Instructions

Figure 7 depicts the modified datapath with our ALU at the execution stage. There, the *custom ALU* is described in Figure 6 and contains a modular adder, a subtractor and a multiplier. For simplicity, the datapath is abstract and many of its components are excluded from the diagram. We edited the decoder unit, the decode stage and the execute stage. The decoder is responsible for interpreting the instruction fetched from memory and generating control signals for the subsequent pipeline stages. To accommodate the custom ALU, we extend the decoder to recognize additional custom opcodes representing the new operations supported by the custom ALU. This includes adding logic to the decoder to generate new control signals that will be used to steer data to and from the custom ALU. In the decode stage, the instruction is further analyzed, and the operands are fetched from the register file. We modified this stage to handle the additional control signals generated by the decoder for the custom ALU. This ensures that, when an instruction targeting the custom ALU is encountered, the appropriate operands are fetched and routed to the custom ALU. We integrated the custom ALU into the execution stage. The custom ALU is connected in parallel with the existing ALU, and both share the same input interfaces. A multiplexer is used to select between the output of the custom ALU and the regular ALU based on the control signals generated by the decoder.

Table 1 lists the new instructions which are of R-type format and they have the same opcode. Between the two schemes, the instructions are distinguished by the funct7 and vary by a single bit. The Least Significant Bit (LSB) of funct7 is utilized to control the modulus to be used and the effective width of the signals. The operations themselves are distinguished by the funct3 field. Modular addition, subtraction, and multiplication operations are analogous to their integer counterparts performed by the classic ALU. The operands for these operations are read from two source registers, and the result is written back to a single destination register. For the butterfly instructions, three read and two write ports are needed, which are already available in the PULPino core we consider. There, the coefficients `a` and `b` are read from the first and second source registers, while the `twiddle` is read from the destination register. The new coefficients `a'` and `b'` are then computed and written back to the first and second source registers. Furthermore, we managed to maintain a low number of sources by creating only the bare minimum control signals for our multiplexers. In addition, by ensuring that the same operands that go to the regular ALU also go to our custom ALU, we were able to avoid increasing the pipeline registers. This approach allowed us to maintain the efficiency of our design while expanding its capabilities.

3.4 PQVALUE: NTTs

We have described hardware blocks to support various modular arithmetic operations with specific moduli, as well as their integration into a RISC-V microcontroller. In this section, we elaborate on how software can use the added

Table 1: Custom arithmetic instructions for RISC-V ISA

opcode	funct3	funct7	operation name
1110111	001	0000000	pq.mod_add_dil
1110111	010	0000000	pq.mod_sub_dil
1110111	011	0000000	pq.mod_mul_dil
1110111	100	0000000	pq.ct_btrfly_dil
1110111	101	0000000	pq.gs_btrfly_dil
1110111	001	0000001	pq.mod_add_kyb
1110111	010	0000001	pq.mod_sub_kyb
1110111	011	0000001	pq.mod_mul_kyb
1110111	100	0000001	pq.ct_btrfly_kyb
1110111	101	0000001	pq.gs_btrfly_kyb

instructions to improve the performance of polynomial arithmetic. The instructions detailed in Section 3.3 can be used to build the butterflies as detailed in Figure 8c and Figure 8e. We consider the case where no single-cycle butterfly is available (2 read and 1 write ports) and where it is available (3 read and 2 write ports), respectively. We refer to the first design as PQVALUE[1], and the latter as PQVALUE[2]. For reference, we also include optimized RISC-V software butterflies in Figure 8a for comparison.

As can be seen from Section 2.1, NTTs and inverse NTTs can be implemented in place with 8 separate layers, each containing 128 independent butterflies. A straightforward implementation could process the layers of butterflies one by one, in which case for each butterfly the input coefficients are read, the butterfly is computed, and the results are stored back in memory. Hence, the polynomial is read and stored at every layer: this leads to signification number of memory access that can become a bottleneck. Hence, a common technique in software implementations is to merge multiple layers in order to reduce these overheads [11,10]. More specifically, multiple coefficients are loaded into the register file such that multiple layers can be (partially) computed without need for loading/storing for every butterfly. To merge ℓ layers one would load and store 2^ℓ (in-place) inputs/outputs and apply all $\ell \cdot 2^{\ell-1}$ butterflies before loading the next batch of coefficients. As this is repeated $256/2^\ell = 2^{8-\ell}$ times, the total cost of the ℓ merged layers is $2^{8-\ell} \cdot (2 \cdot 2^\ell \, \texttt{IO} + \ell \cdot 2^{\ell-1} \, \texttt{BFLY})$, where `IO` is the cost of a read/write to memory and `BFLY` is the cost of a single butterfly. For example, with single-cycle read/writes and using a single-cycle butterfly the cost of 4 merged NTT layers is (at least) $2^4 \cdot (32 + 4 \cdot 8) = 1024$ cycles. Since our hardware designs affect `BFLY` but not `IO`, it is worthwhile merging layers even with our extensions as it reduces the overheads of memory accesses. Finally, the twiddle factors need to be loaded once at a cost of 256 `IO`. Note that an actual implementation will have some additional overhead for stack and control flow operations: we give more precise results in Section 4.

As the register file on RISC-V has 32 general purpose registers, we can merge at most 4 layers. Although we have 16 inputs/outputs, some registers are reserved for other values such as twiddle factors. The Dilithium forward NTT can be constructed by merging 4 layers twice and loading the twiddle factors once throughout the layers. The Kyber forward NTT has an early-abort strategy that achieves a cheaper NTT at the cost of a more expensive base multiplication. It can be constructed by merging 4 layers, then merging 3 layers, loading the twiddle factors once throughout the layers. The inverse NTT functions in the same way using Gentleman-Sande butterflies, and with the addition of a final `poly_mul` to remove the constant factor n. Note that this can also help to remove any factors of the Montgomery domain that might remain. For example, the values in the matrix $\mathbf{A}$ are sampled into the Montgomery domain for Kyber, which remains as we only use Barrett reduction in our implementation. The additional inversion/multiplication makes the inverse NTT more expensive than the forward NTT, but as it is used less frequently than the forward NTT, the overall impact on Kyber or Dilithium is minimal. In that case, the final division by 256 can include the Montgomery constant as well. Finally, the base multiplication is a direct application of `mod_mul` on each of the coefficients.

```
.macro montgomery al, ah, qi, q
    mul \al, \a, \qi
    mulh \al, \al, \q
    sub \al, \ah, \al
.endm

.macro ct_butterfly a, b, qi, q, zeta,
      tmp
    mul \tmp, \zeta, \b
    mulh \b, \zeta, \b
    montgomery \tmp, \b, \qi, \q
    sub \b, \a, \tmp
    add \a, \a, \tmp
.endm
```

(a) Cooley-Tukey, RV32

```
.macro montgomery al, ah, qi, q
    mul \al, \a, \qi
    mulh \al, \al, \q
    sub \al, \ah, \al
.endm

.macro gs_butterfly a, b, qi, q, zeta,
      tmp
    sub \tmp, \a, \b
    add \a, \a, \right
    mul \b, \zeta, \tmp
    mulh \tmp, \zeta, \tmp
    montgomery \b, \tmp, \qi, \q
.endm
```

(b) Gentleman-Sande, RV32

```
.macro ct_butterfly a, b, z, tmp
    pq.mod_mul \tmp, \z, \b
    pq.mod_sub \b, \a, \tmp
    pq.mod_add \a, \a, \tmp
.endm
```

(c) Cooley-Tukey, PQVALUE[1]

```
.macro gs_butterfly a, b, zeta, tmp
    pq.mod_sub \tmp, \a, \b
    pq.mod_add \a, \a, \b
    pq.mod_mul \b, \zeta, \tmp
.endm
```

(d) Gentleman-Sande, PQVALUE[1]

```
.macro ct_butterfly a, b, zeta
    pq.ct_btrfly \a, \b, \zeta
.endm
```

(e) Cooley-Tukey, PQVALUE[2]

```
.macro gs_butterfly a, b, zeta, tmp
    pq.gs_btrfly \a, \b, \zeta
.endm
```

(f) Gentleman-Sande, PQVALUE[2]

Fig. 8: Butterfly with custom assembly and two read, one write ports.

4 Results

In this section we present area and performance results of our hardware design. We conduct synthesis for PQVALUE, and compare the unmodified RI5CY core with the modified core that integrates our ALU. We target two different platforms: Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) in order to assess their performance and resource utilization. We also measure the cycle counts of our HW/SW co-design implementations and compare to existing literature.

4.1 Setup

For integration and benchmarking, we use the open-source PULPino platform with its RI5CY core that has a 4-stage pipeline implementing the RISC-V ISA [9]. It has support for the integer (I), compressed (C) and multiplication (M) instruction set extensions. Notably, two 32-bit words are multiplied in 6 cycles where 1 cycle is spent in the `mul` instruction, and 5 cycles are spent in `mulh` (to compute the top half). The memory accesses are all single-cycle. PULPino provides a comprehensive framework and a set of scripts that streamline the process of running automated tests and simulations. This infrastructure simplifies the task of compiling and running simulations, collecting and analyzing results, and comparing these results against expected outputs. To verify the correct functionality of our modified RI5CY core and measure the cycle counts, we utilize the PULPino framework in conjunction with ModelSim HDL Simulator.

For the synthesis on ASICs we use Cadence Genus Synthesis Solution and the TSMC 28nm standard cell library. We set timing constraints at frequencies of 100 MHz, 200 MHz, 300 MHz, and 400 MHz. We choose these values to mimic closely the typical operating ranges of modern microprocessors. We have confirmed that up to 600 MHz, our ALU is not in the critical path of the processor. This approach facilitates potential integration of our design in existing computational systems, while also accounting for the characteristics and limitations of real-world microprocessor architectures. For the synthesis on FPGAs we used Vivado 2019.2 and we target the ZedBoard Zynq-7000 development board. The timing constraint for the FPGAs is already set by PULPino team at 20 Mhz. From the whole PULPino system, we excluded the communication peripherals and the memories and we measured the overhead in the RI5CY core only.

4.2 Hardware Cost of PQVALUE

We report the utilization on ASICs for both of our designs, comparing three parameters: cells count, total cell area, and gate equivalents. The first two metrics are reported by the tool. The gate equivalents are calculated by dividing the total cell area with the size of the two-input NAND gate of the specific library. For the TSMC 28nm library, the size of a two-input NAND gate is $0.378\mu\text{m}^2$.

We provide a comparison between the two designs: the unmodified RI5CY core and the RI5CY core modified with a custom ALU (referred to as RI5CY +

Table 2: Resource utilization of PQVALUE in ASICs. The Total Cell Area is measured in μm^2. All numbers are rounded to the nearest integer.

Freq.	Metric	RI5CY	RI5CY + PQVALUE	Difference	
				Abs.	Rel.
100 MHz	Total Cell Area	15 848	18 315	2 467	16%
	Cell Count	19 044	22 204	3 160	17%
	Gate Equivalent	41 926	48 452	6 526	16%
200 MHz	Total Cell Area	15 844	18 256	2 412	15%
	Cell Count	19 026	21 951	2 925	15%
	Gate Equivalent	41 915	48 297	6 382	15%
300 MHz	Total Cell Area	16 087	18 496	2 409	15%
	Cell Count	19 756	22 723	2 967	15%
	Gate Equivalent	42 559	48 932	6 373	15%
400 MHz	Total Cell Area	16 532	18 834	2 302	14%
	Cell Count	20 912	23 670	2 758	13%
	Gate Equivalent	43 737	49 825	6 088	14%

PQVALUE). The Total Cell Area represents the physical area occupied by the design on the silicon. The Cell Count represents the number of logic cells used in the design. Table 2 shows that as the operating frequency increases, the Total Cell Area, Cells Count, and Gate Equivalents for both designs increase. This is expected as meeting higher frequency requirements often necessitates the use of more resources. However, we notice that at 200 MHz, all the metrics have values slightly lower than the ones in 100 MHz. This is probably due to the variability involved in the synthesis process. At 200 MHz, it's possible that the synthesis tool found a combination of logic cells that resulted in lower area utilization and cell count. At 300Mhz and 400Mhz, the resource utilization significantly increases. Notably, the percentage difference between the two designs decreases as the timing constraint increases. This suggests that the unmodified RI5CY core scales its resources more aggressively to meet higher frequency requirements compared to the modified RI5CY + PQVALUE design. The relative increase of our design compared to the unmodified RI5CY core ranges from 13% to 17% for all metrics across the different frequencies that we measured with.

Table 3 provides a comparison between the two designs, when targeting the same FGPA with the same operating frequency. Slice Look-Up Tables (LUTs) are a fundamental resource in FPGAs used to implement combinational logic functions. The modified RI5CY + PQVALUE shows a percentage increase of 5.29% in LUTs, a percentage increase of 0.72% in Registers, and a percentage increase of 33.3% in Digital Signal Processors (DSPs). This increase was expected as our ALU is pure combinational logic and we did the minimum modification in the registers of RI5CY core. The 2 extra DSPs are from our base 23×23 integer multiplier.

Table 3: Resource utilization of PQVALUE in the Zynq-7000 FPGA.

Resources	RI5CY	RI5CY + PQVALUE	% Difference
Slice LUTs	6 797	7 157	5.3%
Slice Registers	2 212	2 228	0.7%
DSPs	6	8	33.3%

4.3 PQVALUE in Pulpino: NTTs

In this section, we discuss the concrete improvements that PQVALUE has on the NTTs on Pulpino. We focus on Dilithium for simplicity, but very similar numbers can be achieved for Kyber. Recall that Pulpino requires 1 cycle for memory accesses, 1 cycle for `mul`, `add`, and `sub`, and 5 cycles to execute `mulh`. Therefore the full Cooley-Tukey and Gentleman-Sande butterflies as shown in Figure 8a can be performed in 15 clock cycles. Looking at the estimates from Section 3.4 for 4 merged layers with `IO` $= 1$ and `BFLY` $= 15$, we expect the forward NTT to require about 16 640 cycles. In this case 15 360 cycles are spent on butterflies, and 1 280 on reads/writes. We confirm this by implementing and measuring the cycle count on the Pulpino platform, resulting in 17 041 cycles in total. The additional 401 cycles are expected as they come from stack and control flow operations.

By reducing the butterfly operations to 3 cycles in the case of PQVALUE^1, we reduce the of the butterfly operations from 15 360 to 3 072 cycles. As the cost of reads/writes and stack/control flow operations remains at 1 280 and 401 respectively, the expected cost is 4 753 cycles. The total impact of butterflies is reduced from 90% for a full software implementation, to 65% in PQVALUE^1. We confirm this precise cycle count by implementing the code from Figure 8c and benchmarking. For the single-cycle butterfly, the butterflies require 1 024 cycles and the forward NTT in total 2 705 cycles. The butterflies only contribute to 38% of the runtime, and are now dominated by the overhead for reads and writes. Again, we confirm by implementing the code from Figure 8e and measuring the number of cycles.

Similarly, we implement the operations for the inverse NTT, polynomial multiplication and addition. We provide an overview of the result in Table 4. Overall, we see that PQVALUE^1 leads to a performance improvement of 72% (70%) for the (inverse) NTT, while PQVALUE^2 leads to a reduction by 84% (80%) for the (inverse) NTT. Both improve the polynomial multiplication by 71%, and have no impact on polynomial addition or subtraction.

4.4 PQVALUE in Pulpino: Dilithium

In order to evaluate the impact of the polynomial arithmetic on Dilithium, we integrate them into the reference implementation and compare its run time for

Table 4: Cycle counts of polynomial operations in Dilithium for various instructions. PQVALUE[2] is with single-cycle butterfly, PQVALUE[1] is without single-cycle butterfly.

	RV32	PQVALUE[1]	PQVALUE[2]
`poly_NTT`	17 041	4 753 *(-72%)*	2 705 *(-84%)*
`poly_iNTT`	20 372	6 027 *(-70%)*	3 979 *(-80%)*
`poly_mul`	4 346	1 274 *(-71%)*	1 274 *(-71%)*
`poly_add`	1 274	1 274	1 274

various configurations in Table 5. The reported number are the minimal run-time for each of the algorithms. This choice has an influence for the Sign results, because of its non-deterministic execution time. The reported numbers include a single execution of the rejection loop. Average run-time can be obtained by multiplying by the average number of aborts reported in [2]. For all the algorithms the public matrix is computed on-the-fly in order to save stack.

The two left-most columns compare Dilithium-3 with and without PQVALUE[2] entirely in software. Concretely, we consider the RISC-V optimized Keccak-1600 software implementation proposed in [23] and assembly optimized polynomial addition, subtraction, Montgomery and base multiplication. In this context, we observe that PQVALUE improves the performances by a small factor. Indeed, KeyGen is sped up by a factor 1.09, Sign by 1.25 and Verify by a factor 1.13. This is because Keccak is the bottleneck as usual in pure software implementations [13]. In the two right-most columns of Table 5, we consider a configuration where a Keccak co-processor with 24 cycles per permutation is available. In that context, the bottleneck operations are polynomial operations and therefore the impact of PQVALUE is larger. Concretely, Keygen is sped up by a factor 1.72, Sign by 2.32 and Verify 1.91. We note that in this case, the rest of the operations such as packing, rejection sampling, norm checks, and hint manipulations take a large portion of the overall execution time as they are not assembly optimized. The impact of NTT extensions will be even larger on a fully optimized implementation.

4.5 Comparison

As mentioned in Section 2.4, we compare our custom ALU to two other works that target RISC-V extensions [8,17]. Fritzmann, Sigl and Sepúlveda [8] include an arithmetic unit PQR-ALU for vectorized modular arithmetic and NTT operations, a vectorized modular multiply accumulate unit, a Keccak accelerator for pseudo-random bit generation, and a binomial sampling unit for the generation of distributed samples. The arithmetic units for modular arithmetic and NTT operations are added to the decode stage of RISC-V leading to a *decrease* in the overall processor clock frequency (see [8, Section 5.3]). This is a high price to pay, as all functionality (possibly more critical than PQC) that runs on the

Table 5: Minimum cycle count for Dilithum-3 for various available instructions. Available Keccak co-processor is marked with ^{k}. PQVALUE^2 has a single cycle butterfly.

	RV32		PQVALUE^2		RV32^k		PQVALUE^{2,k}	
	kCycles	%	kCycles	%	kCycles	%	kCycles	%
KeyGen	4 316	100.0	3 970	100.0	825	100.0	479	100.0
Poly	518	12.0	172	4.3	518	62.8	172	36.0
Keccak	3 514	81.4	3 514	88.5	23	2.8	23	4.9
Others	283	6.6	283	7.1	283	34.3	283	59.1
Sign	5 253	100.0	4 218	100.0	1 817	100.0	782	100.0
Poly	1 469	28.0	434	10.3	1 469	80.9	434	55.6
Keccak	3 459	65.8	3 459	82.0	22	1.3	22	2.9
Others	324	6.2	324	7.7	324	17.9	324	41.5
Verify	4 178	100.0	3 712	100.0	976	100.0	511	100.0
Poly	697	16.7	232	6.3	697	71.4	232	45.5
Keccak	3 223	77.1	3 223	86.8	21	2.2	21	4.2
Others	257	6.2	257	6.9	257	26.4	257	50.3

Table 6: Size and efficiency comparison of post-quantum ALUs.

	Resources				**Kyber perf.**		
	LUT	**Reg.**	**DSP**	**BRAM**	**Core**	**NTT**	**NTT^{-1}**
PQR-ALU [8]	2 908	170	9	0	RI5CY	1 935	1 930
PQ ALU [17]	555	0	15	1	CVA6	18 448	18 448
PQVALUE^2	459	0	2	0	RI5CY	2 577	3 851

RISC-V core is impacted. In comparison, we have added our ALU to the execution stage, while we managed to maintain the original clock frequency ensuring optimal performance of our design.

Nannipieri, Di Matteo, Zulberti, Albicocchi, Saponara and Fanucci [17] introduce an extension to the RISC-V ISA to facilitate the NTT operations of the Dilithium and Kyber cryptographic schemes. They implement two distinct custom ALUs, each tailored to a specific scheme. This approach, however, leads to a significant increase in resource overhead compared to our design, which employs a unified ALU for both schemes. The resource utilization and performance of PQR-ALU and PQ ALU are detailed in Table 6.

As shown in Table 6, our proposed PQVALUE^2 demonstrates a significant advantage in terms of resource utilization compared to the other two designs. The PQVALUE^2 uses fewer LUTs, DSPs, and Block RAMs (BRAMs) than the other two designs. In terms of performance for Kyber, PQVALUE^2 performs the

NTT operation in 2 577 cycles and the inverse NTT operation in 3 851 cycles. This is slower than the PQR-ALU [8] which performs both operations in under 2 000 cycles, meaning that we present a different area/performance trade-off that puts the focus on small designs. We believe this is worthwhile as the efficiency is sufficiently high to move the bottlenecks elsewhere (see Table 5 for Dilithium), and we have the additional benefit that we are able to maintain operating frequency of the RI5CY core. We require significantly fewer cycles than the PQ ALU [17], which requires 18 448 cycles for both operations. However, direct comparison is difficult as they benchmark on the CVA6 core with external DDR4 memory. For [17] a similar conclusion can be made for Dilithium, while the efficiency of the NTT is not reported for Dilithium by Fritzmann et al. [8].

References

1. Aikata, A., Mert, A.C., Imran, M., Pagliarini, S., Roy, S.S.: Kali: A Crystal for Post-Quantum Security Using Kyber and Dilithium. IEEE Transactions on Circuits and Systems I: Regular Papers pp. 1–12 (2022)
2. Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (Version 3.1) (2021), https://pq-crystals.org/dilithium/
3. Banerjee, U., Ukyab, T.S., Chandrakasan, A.P.: Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. IACR Transactions on Cryptographic Hardware and Embedded Systems (4), 17–61 (2019)
4. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO'86. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (Aug 1987). https://doi.org/10.1007/3-540-47721-7_24
5. Beckwith, L., Nguyen, D.T., Gaj, K.: High-performance hardware implementation of crystals-dilithium. In: 2021 International Conference on Field-Programmable Technology (ICFPT). pp. 1–10 (2021)
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Building power analysis resistant implementations of Keccak. In: Second SHA-3 candidate conference. vol. 142 (2010)
7. Bisheh-Niasar, M., Azarderakhsh, R., Mozaffari-Kermani, M.: A Monolithic Hardware Implementation of Kyber: Comparing Apples to Apples in PQC Candidates. In: Progress in Cryptology – LATINCRYPT 2021. pp. 108–126. Springer International Publishing (2021)
8. Fritzmann, T., Sigl, G., Sepúlveda, J.: RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 239–280 (2020)
9. Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10), 2700–2713 (2017)
10. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, A.: Compact dilithium implementations on cortex-M3 and cortex-M4. IACR TCHES **2021**(1), 1–24 (2021). https://doi.org/10.46586/tches.v2021.i1.1-24, https://tches.iacr.org/index.php/TCHES/article/view/8725

11. Güneysu, T., Oder, T., Pöppelmann, T., Schwabe, P.: Software speed records for lattice-based signatures. In: Gaborit, P. (ed.) Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013. pp. 67–82. Springer, Heidelberg (Jun 2013). https://doi.org/10.1007/978-3-642-38616-9_5
12. Gupta, N., Jati, A., Chattopadhyay, A., Jha, G.: Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium. IEEE Transactions on Circuits and Systems I: Regular Papers pp. 1–10 (2023)
13. Kannwischer, M.J., Petri, R., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
14. Karl, P., Schupp, J., Fritzmann, T., Sigl, G.: Post-Quantum Signatures on RISC-V with Hardware Acceleration. ACM Trans. Embed. Comput. Syst. (2023)
15. Land, G., Sasdrich, P., Güneysu, T.: A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware. In: Smart Card Research and Advanced Applications. pp. 210–230. Springer International Publishing (2022)
16. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
17. Nannipieri, P., Di Matteo, S., Zulberti, L., Albicocchi, F., Saponara, S., Fanucci, L.: A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. IEEE Access **9**, 150798–150808 (2021)
18. National Institute of Standards and Technology: Post-Quantum Cryptography Standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization
19. Ni, Z., Khalid, A., e Shahwar Kundi, D., O'Neill, M., Liu, W.: Efficient Pipelining Exploration for a High-performance CRYSTALS-Kyber Accelerator. Cryptology ePrint Archive, Paper 2022/1093 (2022)
20. Ricci, S., Malina, L., Jedlicka, P., Smékal, D., Hajny, J., Cibik, P., Dzurenda, P., Dobias, P.: Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs. ARES 21, Association for Computing Machinery, New York, NY, USA (2021)
21. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D., Ding, J.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
22. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134. IEEE Comput. Soc. Press (1994)
23. Stoffelen, K.: Efficient cryptography on the RISC-V architecture. In: Schwabe, P., Thériault, N. (eds.) LATINCRYPT 2019. LNCS, vol. 11774, pp. 323–340. Springer, Heidelberg (Oct 2019). https://doi.org/10.1007/978-3-030-30530-7_16
24. Xing, Y., Li, S.: A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(2), 328–356 (2021)
25. Zhao, C., Zhang, N., Wang, H., Yang, B., Zhu, W., Li, Z., Zhu, M., Yin, S., Wei, S., Liu, L.: A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 270–295 (2021)
26. Zhou, Z., He, D., Liu, Z., Luo, M., Choo, K.K.R.: A Software/Hardware Co-Design of Crystals-Dilithium Signature Scheme. ACM Trans. Reconfigurable Technol. Syst. **14**(2) (2021)