Extending the RISC-V Instruction Set for Hardware Acceleration of the Post-Quantum Scheme LAC

Tim Fritzmann*, Georg Sigl*, Johanna Sepúlveda†
*Technical University of Munich, Munich, Germany
†AIRBUS Defence and Space GmbH, Ottobrunn, Germany
{tim.fritzmann, sigl}@tum.de, johanna.sepulveda@airbus.com

Abstract—**The increasing effort in the development of quantum computers represents a high risk for communication systems due to their capability of breaking currently used public-key cryptography. LAC is a lattice-based public-key encryption scheme resistant to traditional and quantum attacks. It is characterized by small key sizes and low arithmetic complexity. Recent publications have shown practical post-quantum solutions through co-design techniques. However, for LAC only software implementations were explored. In this work, we propose an efficient, flexible and time-protected HW/SW co-design architecture for LAC. We present two contributions. First, we develop and integrate hardware accelerators for three LAC performance bottlenecks: the generation of polynomials, polynomial multiplication and error correction. The accelerators were designed to support all post-quantum security levels from 128 to 256-bits. Second, we develop tailored instruction set extensions for LAC on RISC-V and integrate the HW accelerators directly into a RISC-V core. The results show that our architecture for LAC with constant-time error correction improves the performance by a factor of 7.66 for LAC-128, 14.42 for LAC-192, and 13.36 for LAC-256, when compared to the unprotected reference implementation running on RISC-V. The increased performance comes at a cost of an increased resource consumption (32,617 LUTs, 11,019 registers, and two DSP slices).**

I. Introduction

Public-key cryptography (PKC) builds the basis for a secure communication over insecure channels. The security of current PKC relies on hard mathematical problems (factorization and discrete logarithm problem) which can be solved in polynomial time when a large-scaled quantum computer is built. As a consequence, a quantum-resistant PKC cipher-suite based on different mathematical problems should be used. Among the different quantum-resistant (post-quantum) algorithms, lattice-based algorithms are one of the main alternatives for practical implementations of post-quantum PKC, offering a good trade-off between security and efficiency. LAC is a lattice-based post-quantum cryptosystem based on the Ring Learning with Errors (RLWE) problem [1] that has paved its way to the second round of the NIST standardization process [2]. It is one of the most performant and memory-efficient NIST candidates, characterized by small key sizes and low arithmetic complexity achieved through: i) the use of a strong error-correcting code (BCH code), which allows using polynomials with small single-byte coefficients; and ii) the use of secret and error polynomials that have ternary coefficients, which allows to replace the multiplications by additions and subtractions. Although LAC is a promising post-quantum candidate, there has been only little research about practical implementations.

Empowering electronic devices with strong security poses several challenges due to limited resources, strict performance requirements, and the vulnerability to implementation attacks. To cope with the ever-increasing embedded design complexity, co-design strategies are used. It achieves high-speed and flexible implementations by splitting the tasks of an algorithm into hardware and software elements. Co-design is further fostered through the open-source hardware initiative, allowing an ultra-fast design cycle for complex and highly customized embedded systems. RISC-V is a free and open Instruction Set Architecture (ISA), which enables a new era of processor innovation. It allows the development of open-source hardware with hardware security extensions and secure co-processors.

Our contribution: In this paper, we propose the first embedded implementation of LAC with co-design techniques. Our design is based on the extension of the RISC-V instruction set and is accelerated through tailor-made hardware accelerators. The hardware modules are deeply integrated into the execution pipeline stage of the RISC-V core to avoid additional and costly data transfer over the bus. In summary, our contributions are: i) first LAC implementation on a RISC-V platform with instruction set extension; ii) closely in the RISC-V CPU integrated hardware accelerators for timing critical steps (generation of polynomials, polynomial multiplication, and error correction); iii) extension of RISC-V instruction set for accessing the hardware accelerators; and iv) performance evaluation of LAC with constant-time error-correcting code.

II. Related Works

LAC is part of the 26 candidates selected by NIST for the second round of the post-quantum competition. In order to keep the competitiveness, the second round LAC submission selects a new parameter set, uses a fixed weight error distribution and includes a constant-time BCH code [3]. Even though the computational efficiency and low memory requirements of LAC are attractive, only few LAC implementations have been demonstrated. In [3], the LAC reference implementation and an implementation based on the x86 Advanced Vector Extension (AVX) instruction set were presented. In [4], the first embedded implementation of LAC on the popular ARM Cortex-M4 microcontroller was demonstrated. The LAC reference implementation was compiled for a RISC-V in [5]. However, so far, performance results have not been reported.

Only few works use HW/SW co-design techniques to accelerate lattice-based post-quantum cryptography [6]–[9]. The Hardware Security Module (HSM) of the Infineon smart cards was used to implement the Kyber algorithm in [6]. In [7], the IEEE-1363.1 NTRU algorithm running on an ARM Cortex-A53 was accelerated through a HW polynomial multiplication unit. The first HW/SW lattice-based scheme on a RISC-V platform was proposed in [8]. The authors present a RISC-V co-processor for NewHope which accelerates the Number Theoretic Transform (NTT)-based multiplication and the hash-based generation of polynomials. In [9], Sapphire was presented, a flexible lattice-based crypto-processor able to support some of the NIST NTT-based schemes (Frodo, NewHope, qTESLA, CRYSTALS-Kyber/Dilithium). Sapphire is configured through a RISC-V core. Despite the good results, previous works use loosely-coupled accelerators and none of the previous works have developed hardware accelerators for LAC. In contrast to other lattice-based schemes, LAC does not use an NTT-based polynomial multiplication. Instead, the multiplications can be performed through additions and subtractions. In addition, LAC uses a powerful error-correcting code. Due to these substantial differences, it is required to develop completely new hardware components to accelerate LAC. Our work aims to overcome the discussed drawbacks of the previous works.

III. LAC Post-Quantum Scheme

LAC is a public-key cryptosystem based on the RLWE problem, which was introduced in [1]. In this section, we describe the mathematical basis of LAC and its three major operations: key generation, encryption and decryption (Fig. 1).

A. Ring Learning With Errors (RLWE)

Throughout this paper, all polynomials are printed in bold while single coefficients are printed in normal font. The RLWE problem can be divided into two sub-problems. The first problem (search problem) entails to recover a secret polynomial $\boldsymbol{s}$ from the equation $\boldsymbol{b} = \boldsymbol{as} + \boldsymbol{e}$, where the public polynomial $\boldsymbol{a}$ and the result $\boldsymbol{b}$ are both known. In LAC, the coefficients of the secret polynomial $\boldsymbol{s}$ and the error polynomial $\boldsymbol{e}$ are sampled from a binomial distribution, while the coefficients of the public polynomial $\boldsymbol{a}$ are sampled from a large uniform distribution. The second problem (decision problem) entails to distinguish $(\boldsymbol{a}, \boldsymbol{b})$ from a uniform sample. As solving the RLWE problem is equivalently hard as solving the NP-hard approximate Shortest Vector Problem (SVP) in a lattice, it is well suited as the main building block for LAC.

B. Key Generation

The key generation creates the public key $pk = (seed, \boldsymbol{b})$ and the secret key $sk = \boldsymbol{s}$. The function *GenA* takes a random seed as input, expands this seed using a pseudo random number generator (SHA256 in LAC), and models the coefficients of $\boldsymbol{a}$ according to a large uniform distribution. In contrast to the coefficients of $\boldsymbol{a}$, the coefficients of $\boldsymbol{s}$ and $\boldsymbol{e}$ are sampled from a small binomial distribution χ instead of a large uniform distribution. The RLWE instance $\boldsymbol{b}$ is created by the multiplication of the secret polynomial $\boldsymbol{s}$ with the public polynomial $\boldsymbol{a}$ and by the addition of the error polynomial $\boldsymbol{e}$.

C. Encryption

The encryption transforms any plaintext μ into a ciphered message $ct = (\boldsymbol{u}, \boldsymbol{v})$ by means of the public key pk. The polynomials $\boldsymbol{a}$, $\boldsymbol{s'}$, $\boldsymbol{e'}$, $\boldsymbol{e''}$, and the RLWE instance $\boldsymbol{u}$ are created in a similar way as $\boldsymbol{a}$, $\boldsymbol{s}$, $\boldsymbol{e}$, and $\boldsymbol{b}$ within the key generation. For the encryption, the plaintext μ is first encoded with the BCH encoder into a codeword and then transformed into a polynomial. Then, the encoded plaintext is hidden in the RLWE instance $\boldsymbol{v}$ and can be sent securely over the channel.

D. Decryption

The decryption retrieves the original plaintext μ from $\boldsymbol{c}$ by means of the private secret key sk. The plaintext can be obtained by subtracting $\boldsymbol{us}$ from $\boldsymbol{v}$, which eliminates the largest noise terms. In order to remove remaining noise an error-correcting decoder (BCH decoder) is used.

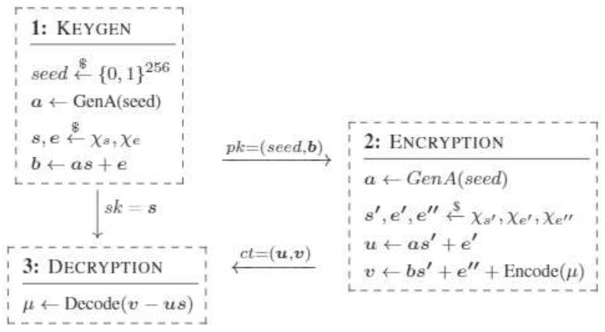
1: KEYGEN
$seed \xleftarrow{\$} \{0,1\}^{256}$
$\boldsymbol{a} \leftarrow$ GenA(seed)
$\boldsymbol{s}, \boldsymbol{e} \xleftarrow{\$} \chi_s, \chi_e$
$\boldsymbol{b} \leftarrow \boldsymbol{as} + \boldsymbol{e}$
$pk=(seed, \boldsymbol{b})$
2: ENCRYPTION
$\boldsymbol{a} \leftarrow GenA(seed)$
$\boldsymbol{s'}, \boldsymbol{e'}, \boldsymbol{e''} \xleftarrow{\$} \chi_{s'}, \chi_{e'}, \chi_{e''}$
$\boldsymbol{u} \leftarrow \boldsymbol{as'} + \boldsymbol{e'}$
$\boldsymbol{v} \leftarrow \boldsymbol{bs'} + \boldsymbol{e''} +$ Encode(μ)
$sk = \boldsymbol{s}$
$ct=(\boldsymbol{u}, \boldsymbol{v})$
3: DECRYPTION
$\mu \leftarrow$ Decode$(\boldsymbol{v} - \boldsymbol{us})$

Fig. 1. LAC algorithm

IV. Hardware Accelerators

In this section, we describe the LAC hardware accelerators developed in this work to alleviate the three performance bottlenecks: generation of polynomials, polynomial multiplication, and error correction. The polynomial generation repetitively uses a SHA256 accelerator. The SHA256 hardware accelerator used in this work was developed in our previous work [7]. Thus, it is not further discussed in this paper.

A. Polynomial Multiplication

The polynomial multiplication is the basis of LAC. It is used to create RLWE instances and to remove the noise terms during the decryption. All arithmetic operations in RLWE-based schemes are performed in the ring $\mathcal{R}_n = \mathbb{Z}_q/\langle\phi_n(x)\rangle$, where $\phi_n(x)$ is usually chosen to be x^n+1 (negative wrapped convolution) or x^n-1 (positive wrapped convolution). In case of LAC, $\phi_n(x) = x^n+1$. The polynomials can be written as $\boldsymbol{a} = a_0 + a_1x + \ldots a_{n-1}$, where $n-1$ is the degree and each coefficient is an element of the finite field $\mathbb{Z}_q$. While the addition of two polynomials can be performed efficiently by a coefficient-wise addition, the polynomial multiplication is considerably more complex. A direct approach to calculate the coefficients c_i of $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b} \mod (x^n+1)$ is

$$c_i = \sum_{j=0}^{i} a_j b_{(i-j)} \; mod \; q - \sum_{j=i+1}^{n-1} a_j b_{(n+i-j)} \; mod \; q \; . \qquad (1)$$

LAC has secret and error polynomials $\boldsymbol{s}$, $\boldsymbol{s'}$, $\boldsymbol{e}$, $\boldsymbol{e'}$, and $\boldsymbol{e''}$ that are sampled from a binomial distribution with variance $\leq 1/2$, leading to ternary coefficients $\{-1,0,1\}$. For all multiplications, a ternary polynomial $\mathcal{T}$ is multiplied with a general polynomial $\mathcal{G}$ with coefficients in $\mathbb{Z}_q$. This simplifies the polynomial multiplications because the partial multiplication steps can be replaced by additions/subtractions.

Hardware Architecture Polynomial Multiplier: The hardware architecture of our ternary polynomial multiplier (*MUL TER*) is shown in Fig. 2. The architecture is based on [7], [10], which only supports the positive wrapped convolution, i.e. a polynomial reduction by $x^n - 1$. As LAC requires a negative wrapped convolution, we extended the circuit to support both convolution variants. This is done by adding the multiplexers to the circuit and enhancing the *Control Unit*. The ternary multiplier takes as input a polynomial $\boldsymbol{a} \in \mathcal{T}$, a polynomial $\boldsymbol{b} \in \mathcal{G}$ and returns the result $\boldsymbol{c} \in \mathcal{G}$. The polynomial $\boldsymbol{a}$ is forwarded to the *Control Unit* which serializes the ternary coefficients a_i, starting from a_0, at the first clock cycle, until a_{n-1}. The coefficients of the polynomial $\boldsymbol{b}$ are directly assigned to the first input of the Modular Arithmetic Units (MAU). The MAUs have three operation modes (addition,

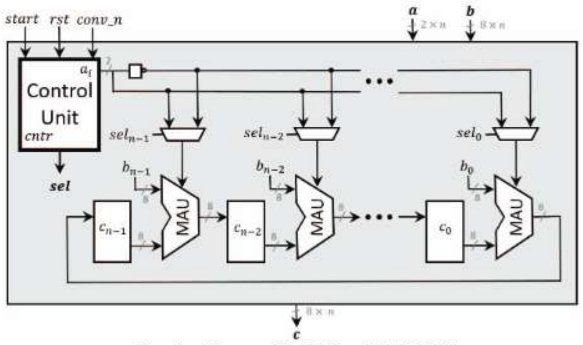

Fig. 2. Ternary Multiplier (*MUL TER*)

subtraction and forwarding), which are activated depending on the value of the coefficient a_i: i) when 1, $c_i + b_i \ mod \ q$ is calculated; ii) when -1, $c_i - b_i \ mod \ q$ is calculated; and iii) when 0, c_i is forwarded. The feedback loop from the rightmost MAU to the register c_{n-1} performs the wrap around (the polynomial modulo reduction by $\phi_n(x)$ in Eq. (1)). To select the positive or negative wrapped convolution, the multiplexers are used. They forward a_i or the negation of a_i for the positive or negative wrapped convolution, respectively. The control signal sel_i is set to zero (positive convolution) if $conv\_n = 0$ and is set to one (negative convolution) if $conv\_n = 1$ and $i > n-1-cntr$, where the value of $cntr$ is equal to the current clock cycle (from 0 to $n-1$). After n clock cycles, the *Control Unit* outputs the last coefficient a_{n-1}. The coefficients of the result are then ready to be forwarded from the registers c_0 to c_{n-1} to the output $\boldsymbol{c}$.

Software-Based Polynomial Splitting: Our multiplication architecture scales directly with the parameter n ($n = 512$ for LAC-128 and $n = 1024$ for LAC-192/LAC-256). To reuse the hardware architecture for all LAC security levels, we perform a SW-based split of the polynomials of length $n = 1024$ into smaller pieces compatible with a length-512 *MUL TER* unit. Alternatively, a larger *MUL TER* unit for high-speed applications or a smaller one for area-limited devices can be used. However, a length-512 *MUL TER* unit seems to present a good trade-off between performance and area. The results in Section VI show that the accelerated multiplication using a length-512 *MUL TER* unit is already faster than the generation of polynomials using the SHA256 accelerator, such that increasing the size of the multiplier would not lead to a large improvement of the overall LAC execution time.

The length-m polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ are split into two length-$m/2$ polynomials: i) lower part ($\boldsymbol{a}^l$, $\boldsymbol{b}^l$); and ii) higher part ($\boldsymbol{a}^h$, $\boldsymbol{b}^h$). The resulting multiplication is given as in Eq. (2).

$$\boldsymbol{c} = \boldsymbol{a}\boldsymbol{b} = \boldsymbol{a}^l\boldsymbol{b}^l + (\boldsymbol{a}^l\boldsymbol{b}^h + \boldsymbol{a}^h\boldsymbol{b}^l)x^{m/2} + \boldsymbol{a}^h\boldsymbol{b}^h x^m \qquad (2)$$

Note that Karatsuba's algorithm allows to reduce the four polynomial multiplications in Eq. (2) to three. However, using Karatsuba's algorithm requires the multiplication of general polynomials, i.e. multiplications $\mathcal{G} \times \mathcal{G}$. Thus, our ternary multiplier *MUL TER* could not be used. Enhancing our multiplier to support general multiplications implies an increase of the design complexity, e.g. adders/subtractors would have to be exchanged by multipliers. As our current architecture has proven to be very efficient, the use of Karatsuba's algorithm has been left as a future work.

To use a length-512 *MUL TER* unit for the multiplication of two length-1024 polynomials, a two-level polynomial split must be performed, first a split into length-512 and then into length-256 polynomials. The reason is that the length-512 *MUL TER* unit only supports a polynomial reduction by $x^{512} \pm 1$ and not by $x^{1024} \pm 1$. Algorithm 1 shows the two-level splitting technique. First, the algorithm splits the polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ into length-512 polynomials. In Line 1-2, the shares of these polynomials are forwarded to four instances of Algorithm 2, which splits the polynomials into length-256 polynomials. After the splitting in Algorithm 2, the *MUL TER* unit is used to calculate the multiplications. The four partial results of Eq. (2) are then recombined in Line 3-7 (Algorithm 2). When the four instances of Algorithm 2 completed the recombination, Algorithm 1 starts the recombination phase in Line 3-12. While in Algorithm 2 the recombination is done as in Eq. (2), Algorithm 1 already integrates the modulo operation by $x^{1024}+1$. The polynomial reduction by $x^{1024}+1$ is simply done by wrapping coefficients larger than 1023 negatively around as done in Line 5 and Line 11. Note that an addition/subtraction of two coefficients requires a further reduction by $q = 251$, such that the coefficients remain in $\mathbb{Z}_q$.

Algorithm 1: split_mul_high($\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$)

```
   Input: $a, b \in \mathcal{R}_{1024}$
1  split_mul_low($a^l, b^l, c^{ll}$),  split_mul_low($a^h, b^h, c^{hh}$)
2  split_mul_low($a^l, b^h, c^{lh}$),  split_mul_low($a^h, b^l, c^{hl}$)
3  for $i \leftarrow 0$ to $1024 - 1$ do
4      $c_i \leftarrow c_i^{ll}$
5      $c_i \leftarrow (c_i - c_i^{hh}) \mod q$ // wrap around
6  end
7  for $i \leftarrow 0$ to $512 - 1$ do
8      $c_{i+512} \leftarrow (c_{i+512} + c_i^{lh} + c_i^{hl}) \mod q$
9  end
10 for $i \leftarrow 512$ to $1024 - 1$ do
11     $c_{i-512} \leftarrow (c_{i-512} - c_i^{lh} - c_i^{hl}) \mod q$ // wrap around
12 end
   Result: $c \in \mathcal{R}_{1024}$
```

Algorithm 2: split_mul_low($\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$)

```
   Input: $a, b \in \mathcal{R}_{1024}$ with size 512
1  mul_ter($a^l, b^l, c^{ll}$),  mul_ter($a^h, b^h, c^{hh}$)
2  mul_ter($a^l, b^h, c^{lh}$),  mul_ter($a^h, b^l, c^{hl}$)
3  for $i \leftarrow 0$ to $512 - 1$ do
4      $c_i \leftarrow c_i^{ll}$
5      $c_{i+256} \leftarrow (c_{i+256} + c_i^{lh} + c_i^{hl}) \mod q$
6      $c_{i+512} \leftarrow (c_{i+512} + c_i^{hh}) \mod q$
7  end
   Result: $c \in \mathcal{R}_{1024}$
```

B. Error Correction

The BCH decoder can be divided into three steps: i) calculation of the syndromes; ii) calculation of the error locator polynomial (usually with Berlekamp-Massey algorithm); and iii) calculation of the roots of the error locator polynomial (usually with Chien search algorithm). To mitigate the performance overhead introduced by timing side-channel countermeasures, we accelerate the Chien search algorithm [11], which is the most costly operation of the decoder (see Section VI).

Hardware Architecture Galois Field Multiplier: The LAC cryptosystem uses a BCH(n=511,k=367,t=16) for LAC-128/LAC-256, and a BCH$(511, 439, 8)$ for LAC-192. Both BCH codes require arithmetic operations in the Galois Field GF(2^m) with $m = 9$. Let α be a primitive element in $GF(2^m)$, then the finite field has the elements $F_q^* = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\}$. The next element $\alpha^{2^m-1} = 1$,

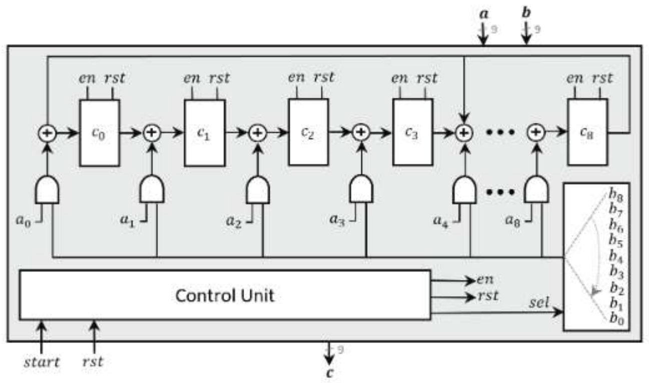

Fig. 3. Galois Field Multiplier (*MUL GF*)

followed by $\alpha^{2^m} = \alpha, \ldots$ As a result, the primitive element α is the generator for the closed finite field. The multiplicative group $F_q^* = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\}$ is also called power representation. To obtain the vector representation of the finite field, x of the primitive polynomial $p(x) = 1 + x^4 + x^9$ is substituted with α, resulting into $p(\alpha) = 1 + \alpha^4 + \alpha^9$. This term is set to zero, leading to the equation $\alpha^9 = 1 + \alpha^4$ (addition and subtraction are equal in the binary case). Using this knowledge, the vector representation can be constructed. For example, $\alpha^9 = 1 + \alpha^4 \mathrel{\widehat{=}} (100010000)$, $\alpha^{10} = \alpha^9\alpha = (1 + \alpha^4)\alpha = \alpha + \alpha^5 \mathrel{\widehat{=}} (010001000)$, and $\alpha^{11} = \alpha^2 + \alpha^6 \mathrel{\widehat{=}} (001000100)$.

The vector representation is specially suitable for finite field additions as the XOR operation can be applied to add two field elements in the vector representation. The GF-multiplication is a little more complicated. Our GF-multiplication module (*MUL GF*), which is presented in Fig. 3, is based on the architecture in [12]. It has a shift-and-add structure with interleaved reduction by the primitive polynomial. The multiplier takes as input two field elements $\boldsymbol{a}$ and $\boldsymbol{b}$ in the vector representation and outputs the result $\boldsymbol{c} = \boldsymbol{ab}$. The bits a_i of the input $\boldsymbol{a}$ are directly assigned to the first input of the AND-gates. The *Control Unit* selects and forwards sequentially the bits b_i of the input $\boldsymbol{b}$ to the second input of the AND-gates, beginning in the first clock cycle with the last element b_8. Moreover, it ensures that the registers are empty at the beginning using the rst signal and it triggers the calculation process by setting the enable signal en to one after receiving the $start$ signal. The heart of the architecture is the shift register $\boldsymbol{c}$, which contains the output bits after $m = 9$ clock cycles. It has a feedback loop that depends on the primitive polynomial. In our case, the result of register c_8 is added to the inputs of c_0 and c_4. The AND-gates and XOR-gates are used to perform the multiplications and additions, respectively. After 9 clock cycles, the *Control Unit* stops the rotation of the shift registers by setting the en signal to zero. Now, the result of the GF-multiplication is ready and can be forwarded from the registers c_0 to c_8 to the output $\boldsymbol{c}$.

Hardware Architecture Chien Search: The last step of the decoding process, which is also the most time-consuming step, is to find the roots of the error locator polynomial $\Lambda(x) = \lambda_0 + \lambda_1 x + \cdots + \lambda_t x^t$, where t is the maximum amount of correctable errors and λ_i a finite field element. The root finding problem can be solved using the Chien search algorithm [11]. It successively substitutes x of the error locator polynomial $\Lambda(x)$ with $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$, where n is the code-length:

$$\Lambda(\alpha^i) = \lambda_0 + \lambda_1\alpha^i + \cdots + \lambda_t\alpha^{it} \ . \tag{3}$$

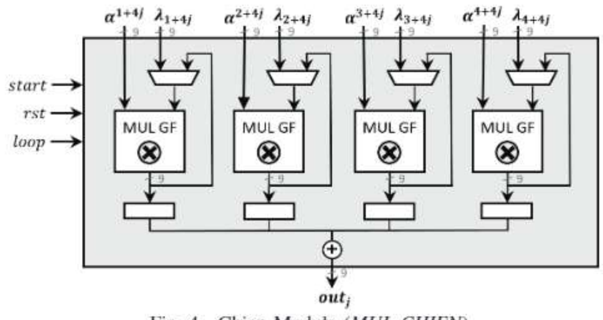
α^{1+4j} λ_{1+4j}
α^{2+4j} λ_{2+4j}
α^{3+4j} λ_{3+4j}
α^{4+4j} λ_{4+4j}
start
rst
loop
MUL GF
9
out_j

Fig. 4. Chien Module (*MUL CHIEN*)

If α^l turns out to be a root, there is an error at the location number $n - l$. As the codeword in LAC is systematic and the message length is only 256 bits, not all powers of alpha have to be checked, i.e. only $\Lambda(\alpha^{112})$ to $\Lambda(\alpha^{368})$ for LAC-128/LAC-256, and $\Lambda(\alpha^{184})$ to $\Lambda(\alpha^{440})$ for LAC-192.

Special circuits for multiplying a field element with the constants α^i exist [12]. However, for each constant a different circuit would be required. To be flexible while keeping the area overhead low, we use the general multiplier *MUL GF*. To speed up the arithmetic operations of Eq. (3), four field multiplications and additions are calculated in parallel (Fig. 4). Thus, the equation is split into two parts for $t = 8$ and into four parts for $t = 16$:

$$\begin{aligned}\Lambda(\alpha^i) = \lambda_0 + \sum_{j=0}^{t/4-1} \lambda_{1+4j}\alpha^{i(1+4j)} + \lambda_{2+4j}\alpha^{i(2+4j)} \\ + \lambda_{3+4j}\alpha^{i(3+4j)} + \lambda_{4+4j}\alpha^{i(4+4j)} = \lambda_0 + \sum_{j=0}^{t/4-1} out_j \ .\end{aligned} \tag{4}$$

To avoid an update of the input values in each i-th check ($\Lambda(\alpha^i)$), a feedback loop from the output to the input is set, i.e. the values λ_{1+4j} to λ_{4+4j} are only loaded to the second input of the GF multipliers in the first round. In all other rounds, the result is fed back ($loop$-signal enabled), thus increasing the performance. The first input can remain constant with α^{1+4j} to α^{4+4j}.

V. Integration into RISC-V

The system used in this work is the microcontroller platform PULPino [13], which integrates a four-stage 32-bit processor known as RISCY. This processor fully supports the RISC-V base integer instruction set (I), the compressed instruction set (C), and the multiplication instruction set (M). The main components of our post-quantum enhanced RISCY core are shown in Fig. 5: a prefetch buffer, an instruction decoder, a general purpose register bank (GPR), an arithmetic logic unit (ALU), a multiplication unit (MULT), our post-quantum ALU (PQ-ALU), and a load and store unit (LSU). The *PQ-ALU* includes our four hardware accelerators. The *MOD q* module was not discussed so far. It is used for a constant-time modulo reduction and is based on the Barrett algorithm.

To access and control the custom accelerators, the RISC-V instruction set was extended by four post-quantum instructions: $pq.mul\_ter$, $pq.mul\_chien$, $pq.sha256$, and $pq.modq$. The RISC-V base ISA has four core instruction formats (R/I/S/U), whereas the R-type (shown in Fig. 6) is the most suitable format for the post-quantum instructions as no immediate value is required. The $opcode$ is set to $0x77$ for all post-quantum instructions and is used to activate the *PQ-ALU*. Depending on the $func3$ field, the instruction decoder sets the corresponding sel signal to activate the accelerator. The values

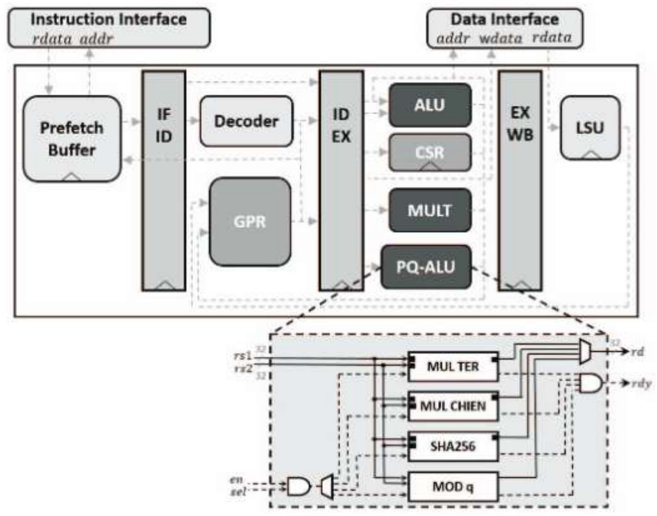

Fig. 5. RISC-V Core (RISCY) with PQ-ALU

31 25	24 20	19 15	14 12	11 7	6 0
$func7$	$rs2$	$rs1$	$func3$	rd	$opcode$

Fig. 6. R-type instruction format

of the fields $rs1$, $rs2$, and rd select the corresponding register of the *GPR*. As the input and output values of the units *MUL TER*, *MUL CHIEN*, and *SHA256* are too large to fit into two input registers $rs1$/$rs2$ and one output register rd, these units use input and output buffers.

The *MUL TER* unit has three operation modes: read input coefficients, calculate multiplication, and write output coefficients. During the read input operation, five general coefficients (8-bits each) and five ternary coefficients (2-bits each) are repetitively packed into the source registers $rs1$ and $rs2$ until all 512 coefficients are available at the accelerator. During the write output operation, the output register rd is repetitively packed with four coefficients (8-bits each). Remaining bits of the input registers, which were not used for the coefficients, are used to control the accelerator (configure write input/output, read/write address, and $conv\_n$ and $start$ signals).

The *MUL CHIEN* unit has three operation modes: read four field elements for the left two multipliers, read four elements for the right two multipliers, and calculate/return the result. One field element requires 9-bits. Therefore, four elements are packed into the source registers. The remaining bits are used to control the operation modes and the $loop$ signal.

At the *SHA256* unit, $rs1$ is used for the input values (8-bits each) and $rs2$ for the read/write address as well as the configuration signals (generate hash and reset internal state).

VI. Experimental Results

A. Timing Side-Channel within the BCH-Code

In 2018, it was shown in [14] that the use of an unprotected error-correcting code could be exploited by timing side-channel attacks in order to reveal the secret key. As a countermeasure, a constant-time version of the error-correcting code used in LAC was proposed in [15]. Also the $2nd$-round submission of LAC claims to have a constant-time BCH code implementation, which can be activated using a compile flag. However, our performance measurements of the BCH implementation in Table I have shown that the introduced countermeasures do not lead to a constant-time implementation. The implementation of the NIST $2nd$-round LAC submission decoder has a non-negligible difference in clock cycles for 0 and 16 number of errors (16 is the maximum amount of errors for a BCH$(511, 367, 16)$ code). As this time difference can be exploited by an attacker, the implementation of [15] is chosen as baseline for this work. This implementation also slightly differs in a few clock cycles, depending on the input. However, [15] has conducted a leakage test which proves its timing side-channel resistance.

TABLE I
Cycle count BCH$(511, 367, 16)$ on RISC-V for the implementation of the 2nd-round submission and Walters [15]

Scheme	Fails	Syndr.	Error Loc.	Chien	Decode
LAC Subm.	0	$61,994$	158	$107,431$	$171,522$
LAC Subm.	16	$59,616$	$10,172$	$107,690$	$179,798$
Walters *et al.*	0	$89,335$	$33,810$	$380,546$	$514,169$
Walters *et al.*	16	$89,335$	$33,867$	$380,748$	$514,428$

B. Implementation Results

The RISC-V system with extended RISCY core was synthesized and implemented using the programmable logic of the Xilinx Zynq UltraScale+ ZCU102 platform. The code for the RISC-V platform was compiled using the official RISC-V compiler from Berkeley (version 8.2.0) and was loaded via a SPI interface to the instruction memory. Lattice-based PKE-schemes can be constructed with two different security versions: a version secure against Chosen-Plaintext Attacks (CPA) and the stronger version secure against Chosen-Ciphertext Attacks (CCA). The implementation in [8] only provides results for the CPA-secure version, which calls during the encapsulation the encryption (of Sec. III) and during the decapsulation the decryption, whereas the CCA-secure version has another re-encryption step during the decapsulation.

Table II summarizes the performance results of five different implementations: i) the LAC reference implementation on ARM Cortex-M4 [4]; ii) the LAC reference implementation on RISC-V; iii) the LAC reference implementation with constant-time BCH code on RISC-V; iv) the optimized LAC implementation on RISC-V with instruction set extension; and v) the optimized NewHope co-design [8]. In comparison to the LAC reference implementation on ARM Cortex-M4, the reference implementation on RISC-V is about $20\,\%$ slower. A further performance decrease can be measured when the time-protected BCH code is used. However, timing side-channels can be used to break the system. Therefore, the constant-time implementation of [15] is used as starting point for the optimized RISC-V implementation. Note that all instruction set extensions have a constant runtime and that our hardware accelerators support all LAC security levels. That is, the accelerators can be configured during runtime to meet the different security levels without any additional hardware resources. To the best of our knowledge, so far LAC was only implemented in software. Therefore, we compare our LAC solution with the loosely coupled RISC-V implementation of NewHope [8]. The work in [8] reports performance results for the CPA-secure NewHope version (NIST security level V).

Table II contains the measurements of the four most expensive operations, which were accelerated in this work. The functions *GenA* and *Sample poly* are used to generate the public, secret and error polynomials. The two functions repetitively call the SHA256 (LAC) and Keccak/SHAKE function

TABLE II
CYCLE COUNT FOR THE KEY ENCAPSULATION AND PERFORMANCE BOTTLENECKS. BRACKETS INDICATE THE NIST SECURITY LEVEL.

Scheme	Device	Security Class	Key-Generation	Encapsulation	Decapsulation	GenA	Sample poly	Multiplication	BCH Dec.
LAC-128 ref. [4]	ARM Cortex-M4	CCA (I)	2,266,368	3,979,851	6,303,717	–	–	–	–
LAC-192 ref. [4]	ARM Cortex-M4	CCA (III)	7,532,180	9,986,506	17,452,435	–	–	–	–
LAC-256 ref. [4]	ARM Cortex-M4	CCA (V)	7,665,769	13,533,851	21,125,257	–	–	–	–
LAC-128 ref.	RISC-V	CCA (I)	2,980,721	4,969,233	7,544,632	159,097	190,173	2,381,843	161,514
LAC-192 ref.	RISC-V	CCA (III)	10,162,116	13,388,940	22,984,529	287,609	165,092	9,482,261	78,584
LAC-256 ref.	RISC-V	CCA (V)	10,516,000	18,165,942	27,879,782	287,736	344,541	9,482,263	171,622
LAC-128 const. BCH	RISC-V	CCA (I)	2,981,055	4,969,238	7,897,403	159,192	190,256	2,381,843	514,280
LAC-192 const. BCH	RISC-V	CCA (III)	10,162,502	13,388,952	23,126,138	287,736	165,185	9,482,261	220,181
LAC-256 const. BCH	RISC-V	CCA (V)	10,515,588	18,165,040	28,220,945	287,609	344,436	9,482,263	513,687
LAC-128 opt.	RISC-V	CCA (I)	542,814	640,237	839,132	154,746	159,134	6,390	160,295
LAC-192 opt.	RISC-V	CCA (III)	816,635	1,086,148	1,324,014	282,264	156,320	151,354	52,142
LAC-256 opt.	RISC-V	CCA (V)	1,086,252	1,388,366	1,759,756	282,264	291,007	151,355	160,296
NewHope opt. [8]	RISC-V	CPA (V)	357,052	589,285	167,647	42,050	75,682	$> 73,827$	–

TABLE III
RESOURCE UTILIZATION

	LUTs	Registers	BRAMs	DSPs
Peripherals/Memory	8,769	7,369	32	0
RISC-V core total	53,819	13,928	0	10
– Ternary Multiplier	31,465	9,305	0	0
– GF-Multipliers	86	158	0	0
– SHA256	1,031	1,556	0	0
– Modulo (Barrett)	35	0	0	2
NTT accelerator [8]	886	618	1	26
Keccak accelerator [8]	10,435	4,225	0	0

(NewHope). The SHA256 hardware module has a lower performance compared to the Keccak implementation. However, Table III shows that the required resources are considerably lower, mainly due to the size of the internal state 256-bit (SHA256) vs. 1600-bit (Keccak). Changing the SHA256 accelerator with a Keccak accelerator to further increase the performance of LAC has been left for a future work. A particularly high speed improvement was achieved for the polynomial multiplication. It has a comparable performance as in [8], which requires two forward and one inverse NTT operation ($24,609$ cycles each) as well as further cycles for the coefficient-wise multiplications. While our ternary multiplier requires a high amount of LUTs, the NTT accelerator requires a lot of DSPs and one BRAM. In comparison to the reference implementation on RISC-V with constant-time BCH code, the total BCH decoding time was improved by a factor of 3.21 and 4.22 for the security categories 128/256 and 192, respectively.

Compared to the work in [8], our LAC implementation requires around 3.12 million of additional cycles to complete the whole protocol. The overhead is mainly due to the slower SHA256, the additional error-correcting code, and the re-encryption step. In terms of area, our LAC implementation requires additional 21,296 LUTs and 6,176 registers when compared to [8] (decoder overhead is negligible). However, our accelerators use 24 DSP slices less and do not require any BRAM block. Note that LAC has considerably lower key and ciphertext sizes, e.g., for the highest security level (V) LAC/NewHope has $\|pk\| = 1054/1824$, $\|sk\| = 1024/1792$, and $\|ct\| = 1424/2176$ bytes.

VII. CONCLUSION

The generation of polynomials, which repetitively calls the SHA256 function, the polynomial multiplication of ternary and general polynomials, and the Chien algorithm within the error-correcting decoder are the performance bottlenecks of LAC. In this work, we developed for all bottlenecks flexible and constant-time hardware accelerators. Using the polynomial splitting method and a general structure of GF multipliers for the Chien search allows to support all security categories without changing the accelerators. The combination of SW instructions and the use of tailored instruction set extensions for accessing the HW accelerators increases the performance while keeping the design highly flexible. The results show that LAC can be very competitive and is specially suitable for applications where low key and ciphertext sizes are required.

Acknowledgments. This work was partly funded by the German Ministry of Education, Research and Technology in the context of the project Aquorypt (grant number 16KIS1017K).

REFERENCES

[1] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2010, pp. 1–23.

[2] National Institute of Standards and Technology, "Announcing request for nominations for public-key post-quantum cryptographic algorithms," 2016, https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms.

[3] X. Lu, Y. Liu, D. Jia, H. Xue, J. He, and Z. Zhang, "2nd Round - Supporting documentation: LAC," 2019, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions.

[4] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking nist pqc on arm cortex-m4," Cryptology ePrint Archive, Report 2019/844, 2019.

[5] B. Marshall, "SCARV: PQ-RISCV," 2019, https://github.com/scarv/pq-riscv.

[6] M. R. Albrecht, C. Hanser, A. Hoeller, T. Pöppelmann, F. Virdia, and A. Wallner, "Implementing rlwe-based schemes using an rsa co-processor," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 169–208, 2019.

[7] T. Fritzmann, T. Schamberger, C. Frisch, K. Braun, G. Maringer, and J. Sepúlveda, "Efficient hardware/software co-design for ntru," in *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*. Springer, 2018, pp. 257–280.

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

[9] U. Banerjee, T. Ukyab, and A. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 4, pp. 17–61, Aug. 2019.

[10] B. Liu and H. Wu, "Efficient architecture and implementation for NTRUEncrypt system," in *Circuits and Systems (MWSCAS), 2015 IEEE 58th International Midwest Symposium on*. IEEE, 2015, pp. 1–4.

[11] R. Chien, "Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes," *IEEE Transactions on information theory*, vol. 10, no. 4, pp. 357–363, 1964.

[12] S. Lin and D. J. Costello, *Error control coding*. Prentice Hall Englewood Cliffs, 2004, vol. 2.

[13] A. Traber, S. Stucki, F. Zaruba, M. Gautschi, A. Pullini, and L. Benini, "Pulpino: A risc-v based single-core system," Geneva, 2015, openRISC.

[14] J.-P. D'Anvers, M. Tiepelt, F. Vercauteren, and I. Verbauwhede, "Timing attacks on error correcting codes in post-quantum secure schemes." *IACR Cryptology ePrint Archive*, vol. 2019, p. 292, 2019.

[15] M. Walters and S. S. Roy, "Constant-time bch error-correcting code." *IACR Cryptology ePrint Archive*, vol. 2019, p. 155, 2019.