A Hardware Implementation for Code-based Post-quantum Asymmetric Cryptography.*

Kristjane Koleci[1], Marco Baldi[2], Maurizio Martina[1], and Guido Masera[1]

[1] Politecnico di Torino, Italy (kristjane.koleci@polito.it, maurizio.martina@polito.it and guido.masera@polito.it)
[2] Università Politecnica delle Marche, Italy (m.baldi@univpm.it)

Abstract

This paper presents a dedicated hardware implementation of the LEDAcrypt cryptosystem, which uses Quasi-Cyclic Low-Density Parity-Check codes and a decoding algorithm known as Q-decoder for the decryption function. The designed architecture is synthesized for both FPGA and ASIC technologies, featuring an intrinsic scalability over a wide range of parallelism degrees, which makes it possible to target multiple application scenarios, with different trade-offs between decryption latency and implementation complexity. The proposed system achieves a large speed-up over both software execution and a previous hardware implementation, with a the decryption latency as low as 3.16 ms for the FPGA version, and 1.2 ms when synthesized for a 65 nm CMOS technology.

1 Introduction

The development of new primitives for asymmetric cryptography able to withstand attacks based on quantum computers has become an urgent need, due to the groundbreaking advances that are being achieved in the area of quantum computing [2]. While quantum computing algorithms are able to endanger classic asymmetric primitives relying on the hardness of factorizing large integers or computing discrete logarithms, there are some mathematical trapdoors based on problems for which quantum computers do not provide any dramatic speedup [9], and are hence known as post-quantum cryptographic primitives. The interest in these primitives is also justified by the recently initiated NIST process for the standardization of post-quantum cryptosystems [20]. Such a process started in 2016 and is now at its second round of evaluation, with 26 algorithms selected out of the 69 original submissions [21].

A well-known family of post-quantum public-key cryptographic primitives are those based on error correcting codes, initiated by the seminal work of Robert McEliece [19], whose security relies on the hardness of decoding a random-looking linear block code. Such a problem, in fact, has been proven to be NP-complete in 1978 [6], and is not significantly affected by quantum computing algorithms [7]. Despite this, the original McEliece cryptosystem based on Goppa codes has not experienced a great diffusion due to the large size of its public keys, which follows from the unstructured nature of the characteristic matrices of Goppa codes. A known solution to this problem is that of replacing Goppa codes with other families of structured codes, like Quasi-Cyclic Low-Density Parity-Check (QC-LDPC) codes. The latter have been successfully used in some variants of the McEliece cryptosystem that are able to achieve compact public keys [4]. A suite of public-key cryptosystems based on QC-LDPC codes named LEDAcrypt [5] is currently under evaluation among the second round candidates of the NIST post-quantum standardization process [21].

The efficiency of post-quantum cryptographic algorithms when implemented in hardware is considered among the requirements of candidates to the NIST post-quantum cryptography standardization process [22]. This motivates research in this area, and in the design of efficient hardware solutions for the implementation of these new cryptographic primitives.

1.1 Related work

Several works have already appeared in the literature concerning the hardware implementation of post-quantum cryptographic primitives. Among them, isogeny-based cryptographic primitives have been considered in [17, 16] and lattice-based primitives have been considered in [13, 12, 8], while primitives based on the lattice-based problem variant known as ring-learning with errors (Ring-LWE) have been considered in [1].

Concerning the implementation of code-based post-quantum primitives, the classic McEliece scheme based on binary Goppa codes has been considered in [24], while variants based on Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes have been considered in [18, 15]. The LEDAcrypt post-quantum cryptographic primitives based on QC-LDPC codes have also been recently considered for hardware implementation in [14].

Concerning post-quantum digital signatures, a hardware-oriented analysis of NIST post-quantum cryptography standardization candidates is reported in [23].

1.2 Contribution

This work addresses the implementation of an hardware accelerator for the LEDAcrypt primitives. The proposed architecture is synthesizable for both ASIC and FPGA technologies. Moreover, it is scalable in terms of processing parallelism, thus achieving different trade-offs between performance and implementation complexity.

The paper is organized as follows: firstly the description of the algorithm is given in Section 2, then the overview of the architecture and additional details on one key processing unit are provided in Section 3 and 4. Finally, the synthesized results are summarized in Section 5 and the conclusions are given in Section 6.

2 Algorithm

From the complexity standpoint, a crucial algorithm for the LEDAcrypt primitives is the decoding algorithm, which is an iterative algorithm that estimates a sparse error vector $\mathbf{e}$ starting from a syndrome vector $\mathbf{s}$, by exploiting the knowledge of two secret matrices: the secret QC-LDPC code parity-check matrix $\mathbf{H}$ and the secret transformation matrix $\mathbf{Q}$. In LEDAcrypt, this is performed through an iterative algorithm derived from the classic bit-flipping decoding algorithm, and known as Q-decoder.

The decoding algorithm starts from an initial syndrome $\mathbf{s}$ that is computed from the encrypted message $\mathbf{m}$ and the two secret matrices $\mathbf{H}$ and $\mathbf{Q}$ as follows:

$$\textit{Initial Syndrome} : \mathbf{s}^{(0)T} = (\mathbf{HQ})\mathbf{m}^T \tag{1}$$

Then, at every iteration $l = 1, \cdots, It_{max}$, the algorithm generates an updated syndrome $\mathbf{s}^{(l)}$

and a refined estimate of the error vector $\mathbf{e}^{(l)}$, by computing the following quantities:

$$Sigma : \boldsymbol{\sigma}^{(l)} = \mathbf{s}^{(l-1)}\mathbf{H} \tag{2}$$

$$Correlation : \boldsymbol{\rho}^{(l)} = [\rho_1^{(l)}, \rho_2^{(l)}, \ldots, \rho_n^{(l)}] = \boldsymbol{\sigma}^{(l)}\mathbf{Q} \tag{3}$$

$$Thresholds : b^{(l)} = \max_{j=1,\cdots,n} \left\{\rho_j^{(l)}\right\} \tag{4}$$

$$Positions : \mathbf{P}^l = \{v \in [1,n] | R_v^{(l)} > b^{(l)}\} \tag{5}$$

$$Errors : \mathbf{e}^{(l)} = \mathbf{e}^{(l)} = \mathbf{e}^{(l-1)} + \sum_{v \in P^l} \mathbf{q}_v \tag{6}$$

$$Syndrome : \mathbf{s}^{(l)} = \mathbf{s}^{(0)} + \mathbf{e}^{(l)}\mathbf{H}^T \tag{7}$$

where $\mathbf{q}_v$ is the v^{th} row of $\mathbf{Q}^T$. The product in 1 and 7 are performed in GF(2), while the result of equations 2 and 3 are integers. The stop condition is reached when $s = 0$ (s being the sum of the entries of $\mathbf{s}$) or $l = It_{max}$. When the decoding process terminates recovering the correct error vector, such a vector can be straightforwardly used to retrieve the cleartext message $\mathbf{m}$.

LEDAcrypt provides various parameters related to the involved matrices and vectors. Table 1 gives the main algorithm parameters for different levels of security (Category): n_0 is the number of circulant blocks forming the parity-check matrix $\mathbf{H}$, p is the size of the circulant blocks, d_v and m are the numbers of asserted bits in each column of the matrices $\mathbf{H}$ and $\mathbf{Q}$ respectively, t is the number of intentional errors used for encryption and It_{max} is the maximum number of iterations required to successfully recover the encrypted message.

The estimation of the most time consuming units in the decoder is important to achieve an efficient implementation. Therefore, we used a Matlab implementation with the relevant profiling capabilities to derive the processing time required for each main task. Table 2 provides the collected results for a code with $n_0 = 2$, $p = 27,779$ and $It_{max} = 3$. Two versions of the algorithm software implementation have been simulated: the first version is the original model [3], while the second version has been obtained by exploiting specific Matlab options to accelerate the execution. The Matlab code has been run on a laptop with 16GB of RAM and Intel Core i7-6700 HQ CPU with 2.60GHz clock frequency.

It is clear from the profiling results that the most time consuming functions in the Q-decoder are the Initial Syndrome calculation and the Syndrome update. Therefore, a parallel formulation of these tasks is desirable to map them onto a dedicated hardware architecture and to accelerate the whole algorithm. An additional advantage that is expected from the hardware implementation of the Q-decoder is the lower energy dissipation.

Table 1: LEDA parameters for different categories (C).

C	n_0	p	d_v	m	t
1	2	15,013	9	$[5, 4]$	143
	2	27,779	17	$[4, 3]$	224
	3	18,701	19	$[3, 2, 2]$	141
	4	17,027	21	$[4, 1, 1, 1]$	112
2-3	2	57,557	17	$[6, 5]$	349
	3	41,507	19	$[3, 4, 4]$	220
	4	35,027	17	$[4, 3, 3, 3]$	175
4-5	2	99,053	19	$[7, 6]$	374
	3	72,019	19	$[7, 4, 4]$	301
	4	60,509	23	$[4, 3, 3, 3]$	239

Table 2: LEDAcrypt profiling.

Tasks	initial model	
$\mathbf{s}^{(0)}$	1.380 s	78.5 %
$\boldsymbol{\sigma}$	0.225 s	14.5 %
$\boldsymbol{\rho}$	0.103 s	5.8 %
$\mathbf{s}^{(l)}$	0.016 s	0.91 %
$\mathbf{m}$	0.003 s	0.2 %
Total	1.7570 s	100 %
Tasks	optimized model	
$\mathbf{s}^{(0)}$	0.023 s	50 %
$\boldsymbol{\sigma}$	0.003 s	6.5 %
$\boldsymbol{\rho}$	0.003 s	6.5 %
$\mathbf{s}^{(l)}$	0.014 s	30.5 %
$\mathbf{m}$	0.003 s	6.5 %
Total	0.046 s	100 %

3 Architecture

The complete architecture is divided into three main blocks (Figure 1): the Memory, the Control Unit (CU) and the Data Path (DP). The Memory unit contains several memory components that store the data structures used by the decoding process. The CU is implemented as a Finite State Machine (FSM) that drives the execution of the algorithm. Finally, the DP includes the

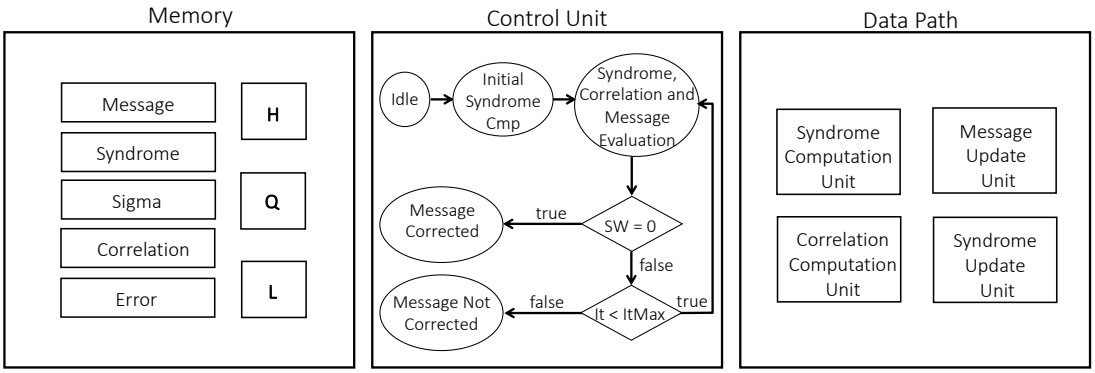


Figure 1: Control Unit, Data Path and Memory

resources necessary to process all the algorithm variables and it is structured into four main units: (i) the Syndrome Computation Unit (SCU), which includes the evaluation of the Initial Syndrome $\mathbf{s}^{(0)}$ as in (1), (ii) the Correlation Computation Unit (CCU), which evaluates the Correlation $\boldsymbol{\rho}$, namely (3), (iii) the Message Update Unit (MUU), which derives the errors $\mathbf{e}$ in the message and its correction (6), and (iv) the Syndrome Update Unit (SUU), which iteratively updates the syndrome ($\mathbf{s}^{(l)}$) based on the obtained errors (7).

The Control Unit FSM basically follows the sequence of processing tasks given in Section 2. The iterative nature of the algorithm and the data dependencies within a single iteration impose a sequential execution of the key steps. However, several operations inside the syndrome and correlation computations allow for a parallel execution. Moreover, the similarities among these operations suggest the reuse of some hardware resources in both SCU and CCU.

3.1 Memory Organization

The vectors and matrices sizes are derived from Table 1. There are three kinds of vectors handled by the algorithm: vectors containing positions, binary vectors and integer vectors. Vectors containing positions in the range $[0, p]$ require $n_p = \lceil \log_2(p) \rceil$ bits. Binary vectors are stored in a matrix format, as N_b bit words, where N_b is the decided degree of processing parallelism. Integer vectors are stored in a matrix format, as n_c and n_s bit words for *Correlation* and *Sigma* values, respectively.

Based on the dynamic range of each data, the expected size for the required memories can be evaluated as in Table 3. The reported size values are related to the case of two circulant block ($n_0 = 2$). Therefore, every array (except the syndrome $\mathbf{s}$ and the matrix $\mathbf{Q}$) is represented as a two-block component. As an example, the message $\mathbf{m}$ is expressed as $\mathbf{m} = [\mathbf{m_0}\ \mathbf{m_1}]$ and $\mathbf{L} = \mathbf{HQ} = [\mathbf{L}_0\ \mathbf{L}_1]^T$. Similarly, the matrix $\mathbf{Q}$ is divided into four components ($\mathbf{Q}_{00}$ to $\mathbf{Q}_{11}$).

Table 3: Required capacity for main memory components, evaluated for codes with $n_0 = 2$ and $p = 27,779$

Variable	type	Size Evaluation	Maximum Size	Minimum Size
$\mathbf{s}$,$\mathbf{m_0}$ and $\mathbf{m_1}$	binary	p	12.2 kB	3.4 kB
$\boldsymbol{\sigma_0}$ and $\boldsymbol{\sigma_1}$	integer	$p * n_s$	61 kB	14kB
$\boldsymbol{\rho_0}$ and $\boldsymbol{\rho_1}$	integer	$p * n_c$	97.6 kB	27.2
$\mathbf{L_0}$, $\mathbf{L_1}$	position	$d_v * m * n_p$	524 B	223 B
$\mathbf{H_0}$, $\mathbf{H_1}$	position	$d_v * n_p$	41 B	32 B
$\mathbf{Q_{00}}$, $\mathbf{Q_{11}}$	position	$m_0 * n_p$	15 B	8 B
$\mathbf{Q_{01}}$, $\mathbf{Q_{10}}$	position	$m_1 * n_p$	13 B	6B

3.2 Initial Syndrome and Correlation Computation Units

Using the matrix representation, for the case $n_0 = 2$, the SCU basically computes the product

$$\begin{bmatrix}\mathbf{m}_0 & \mathbf{m}_1\end{bmatrix} \begin{bmatrix}\mathbf{L}_0 \\ \mathbf{L}_1\end{bmatrix} = \begin{bmatrix}\mathbf{m}_0\mathbf{L}_0 \\ \oplus \\ \mathbf{m}_1\mathbf{L}_1\end{bmatrix} = \begin{bmatrix}\mathbf{s}_0 \\ \oplus \\ \mathbf{s}_1\end{bmatrix} \tag{8}$$

where $\oplus$ indicates the bit wise xor operation, while the CCU calculates

$$\boldsymbol{\sigma} = \mathbf{s} \begin{bmatrix}\mathbf{H}_0 & \mathbf{H}_1\end{bmatrix} = \begin{bmatrix}\boldsymbol{\sigma_0} & \boldsymbol{\sigma_1}\end{bmatrix} \tag{9}$$

$$\boldsymbol{\rho} = \begin{bmatrix}\boldsymbol{\sigma_0} & \boldsymbol{\sigma_1}\end{bmatrix} \begin{bmatrix}\mathbf{Q}_{00} & \mathbf{Q}_{01} \\ \mathbf{Q}_{10} & \mathbf{Q}_{11}\end{bmatrix} = \begin{bmatrix}\boldsymbol{\sigma_0}\mathbf{Q}_{00} & \boldsymbol{\sigma_0}\mathbf{Q}_{01} \\ + & + \\ \boldsymbol{\sigma_1}\mathbf{Q}_{10} & \boldsymbol{\sigma_1}\mathbf{Q}_{11}\end{bmatrix} = \begin{bmatrix}\boldsymbol{\rho}_0 & \boldsymbol{\rho}_1\end{bmatrix} \tag{10}$$

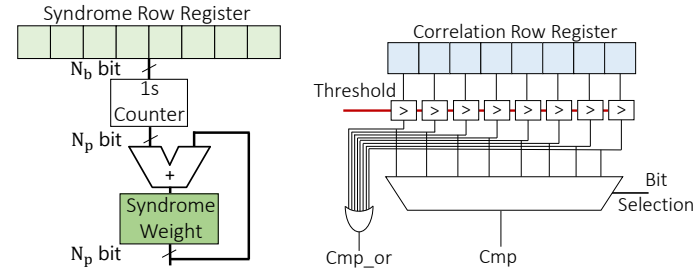
In both units, the key requirement is the calculation of the product between a vector and a sparse cyclic matrix, that is a VectorByCirculant operation. In the SCU a binary vector $\mathbf{m}$ is multiplied, while in the CCU the product is needed twice, involving a binary vector in (9) and an integer vector in (10). However, a unified architecture can be conceived to cover both types of product, as detailed in Section 4.

3.3 Message Update Unit

The update of the message requires to find the positions of the errors intentionally inserted at the encoding side. This estimation is based on Syndrome and Correlation. The circuit in Figure 2(a) shows the evaluation of the syndrome weight. Initially, the syndrome weight is set to 0, then the Syndrome memory is read row by row and the number of ones per row is accumulated in the Syndrome Weight register.

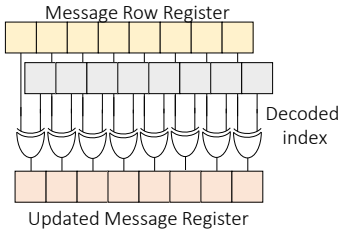
The circuit for the evaluation of the error is in Figure 2(b). A row is read from the Correlation memory and its elements are compared with the threshold b: if at least one value in the row is higher than b (`Cmp_or` $= 1$), the error position is saved into the Error memory, in terms of row address and location within the row (index). The process ends when the last row of the Correlation memory is checked. The threshold is derived from a Look-Up-Table that returns a value of b given the input range of Syndrome Weighs (SWs) [14].

Finally, Figure 2(c) shows the update of the message vector. The circuit receives a message word (N_b elements) from the Message memory and the corresponding error positions from the Error memory. A set of N_b xor gates applies the correction and the updated message is stored back into the Message memory.



(a) Syndrome Weight evaluation logic.

(b) Error Position evaluation.



(c) Message Update.

Figure 2: Key circuits in the Message Update Unit.

3.4 Syndrome Update Unit

The syndrome update equations (6) and (7) are modified into a different form that simplifies the hardware implementation:

$$\mathbf{e}_{syn}^{(l)} = \mathbf{e}^{(l)}\mathbf{L}$$
$$\mathbf{s}^{(l)} = \mathbf{s}^{(l-1)} \oplus \mathbf{e}_{syn}^{(l)}$$

where $\mathbf{L} = \mathbf{HQ}$ and $\mathbf{e}_{syn}$ is the error location vector for the syndrome. The latter equation can be implemented by means of the same circuit given in Figure 2(c), while a sparse vector by circulant product is needed for the calculation of $\mathbf{e}_{syn}^{(l)}$.

This operation is implemented as described in Algorithm 1.

Algorithm 1 SparseVectorByCirculant

Input: $\mathbf{e}^l$,$\mathbf{L}$;
Output: $\mathbf{e}_{Syn}^l$;
$\mathbf{r} \Leftarrow \mathbf{0}$;
$indexSyn = 0$;
while $indexPos < d_v$ **do**
  while $indexErr < MaxPos$ **do**
    $\mathbf{e}_{Syn}^l(indexSyn) = mod(\mathbf{e}^l(indexErr) - \mathbf{L}(indexPos), p)$;
    $indexSyn = indexSyn + 1$;
  end while
end while

Algorithm 2 VectorByCirculant

Input: $\mathbf{v}(1,n)$,$\mathbf{Pos}(1,d)$;
Output: $\mathbf{r}(1,n)$;
$\mathbf{r} \Leftarrow \mathbf{0}$;
$indexPos = 1$;
while $indexPos \leq d$ **do**
  $k = \mathbf{Pos}(indexPos)$;
  $\mathbf{v}_{shifted} = [\mathbf{v}(k:n), \mathbf{v}(1:k-1)]$;
  $\mathbf{r} \Leftarrow \mathbf{r} + \mathbf{v}_{shifted}$
end while

4 Vector By Circulant product architecture

The Vector By Circulant unit, included in both the SCU and CCU, evaluates the product $\mathbf{r} = \mathbf{Av}$ of a cyclic and sparse binary matrix $\mathbf{A}$ by a vector (integer or binary), $\mathbf{v}$. In a cyclic matrix, all the rows are cyclic shifts of the first one. An example of the Vector By Circulant product is given in (11) and (12) for the case of size $p = 15$, with $d_v = 2$ (a_2 and a_5 asserted values in the first row of $\mathbf{A}$). The real values of $\mathbf{p}$ are given in Table 1. The strategy described is the same employed in the QcBits Algorithm[10].

$$\mathbf{A} = \begin{bmatrix} a_0 & a_1 & \dots & a_{14} \\ a_{14} & a_0 & \dots & a_{13} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \dots & a_0 \end{bmatrix} \tag{11}$$

$$\begin{array}{lllllllllll} r_0 & = & a_0v_0 & + & a_1v_1 & \cdots & + & a_{14}v_{14} & = & v_2 & + & v_5 \\ r_1 & = & a_{14}v_0 & + & a_0v_1 & \cdots & + & a_{13}v_{14} & = & v_3 & + & v_6 \\ r_2 & = & a_{13}v_0 & + & a_{14}v_1 & \cdots & + & a_{12}v_{14} & = & v_4 & + & v_7 \\ \vdots & & & & & & & & = & \vdots & & \\ r_{14} & = & a_1v_0 & + & a_2v_1 & \cdots & + & a_0v_{14} & = & v_1 & + & v_4 \end{array} \tag{12}$$

The straightforward implementation of the product, i.e. the direct mapping of these equations into an hardware architecture is not efficient, because of the size of $\mathbf{v}$ and $\mathbf{A}$.

However, it can be seen from (12) that the elements of r are obtained by combining shifted versions of the $\mathbf{v}$ vector. For example, in the given example for $p = 15$, $\mathbf{r}$ can be calculated by taking two circular shifts of $\mathbf{v}$, respectively starting at positions 2 and 5 (the asserted values in the first row), and adding them element by element. In general, $\mathbf{r}$ can be calculated by adding (modulo-2 addition for the binary case) d_v shifted versions of $\mathbf{v}$ (indicated as $\mathbf{v}_{shifted}$). Their initial positions are determined by the elements in the first row of $\mathbf{A}$, stored in the vector **Pos**. The product calculation is detailed in Algorithm 2.

To efficiently implement Algorithm 2, we proceed in a partially parallel way and update N_b elements of $\mathbf{r}$ at a time. Let us assume that the $\mathbf{A}$ matrix is stored in the sparse format, i.e. all non-zero elements in the first row are available in a linear array (**Pos**). Moreover, we assume that $\mathbf{v}$ is stored in the memory M_v, organized as p/N_b words of N_b elements. At every read from the memory, we obtain N_b elements of $\mathbf{v}$ in parallel. Said x the initial position of a shifted version of $\mathbf{v}$ (that is an element of the **Pos** vector), we find the first element in $\mathbf{v}_{shifted}$ by calculating the address $i = \lfloor x/N_b \rfloor$ and the index $j = x \mod N_b$. If N_b is a power of two, i is simply equal to the $\log_2 N_b$ most significant bits of x and j is equal to the remaining least significant bits of x. Therefore, the desired shifted version of $\mathbf{v}$ is obtained by selecting in the read word all elements with index $\geq j$; the vector is then completed by means of additional reads from M_v, at addresses $> i$. Overall, up to $\lceil p/N_b \rceil$ reads are needed to obtain the complete $\mathbf{v}_{shifted}$. For example, with $\mathbf{p} = 15$, $N_b = 4$ and $x = 3$, the first vector element, v_3, is read at the first cycle, together with elements v_0 to v_2, which are not used at this time. In the two following cycles, elements v_4 to v_7, and v_8 to v_{11} are taken from M_v. Finally elements v_{12}, v_{13} and v_{14} are read at the fourth cycle. The architecture of the complete architecture for the product Vector by Circulant is shown in Figure 3.

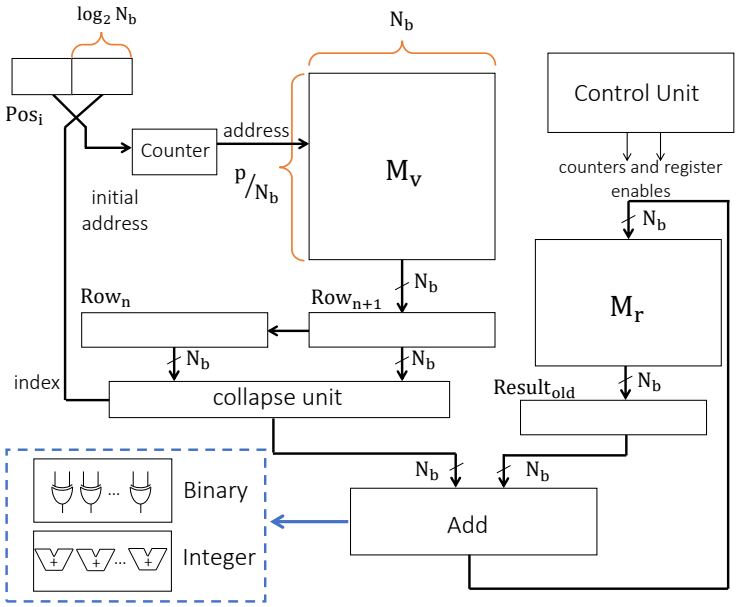


Figure 3: The Data-Path of the VectorByCirculant Unit

In order to process the data read from the M_v memory, we need two N_b-element registers (Row_n and Row_{n+1}) that store two consecutive rows of M_v. At every cycle, a new row is loaded

into the $\text{Row}_n + 1$ register and the previous row is moved to the Row_n register, at the same time. Moreover, a collapse unit merges the elements in the two registers in a sequence of N_b ordered elements starting from the position pointed by index. The circuit, at the first cycle, skips elements with index lower than the initial shift position and reuse them at the last cycle, to complete the tail of the sequence.

The collapse unit provides N_b consecutive elements to an accumulator that calculates the final product as the sum of several shifted versions of $\mathbf{v}$. The temporary accumulated values are stored in the result memory, M_r, which is set to zero at the beginning. Then, at every iteration, the Add unit receives a new portion of a shifted vector from the collapse unit and combines it with the old accumulated values read from M_r. The Add unit actually contains plain xor gates in the case of a binary vector and a set of complete adders for an integer vector.

The complete result is available in the M_r memory after the loading of the last $\mathbf{v}_{shifted}$.

This solution is scalable and provides a speed-up by a factor n_b over the sequential implementation.

5 Simulations and Synthesis results

The proposed architecture has been described in a parameterized form, for multiple degrees of parallelism, ranging from $N_b = 8$ up to $N_b = 64$. The synthesis has been completed using Synopsys Design Compiler and a CMOS 65nm technology library. Functional and post-synthesis simulations have been done to exactly estimate the required number of cycles and to derive the power dissipation.

The results provided by the Modelsim simulations are given in Table 4 for two codes with $n_0 = 2$ and $p = 27,779$ and $p = 15013$.

Table 4: Cycle count for the most relevant functions for $n_0 = 2$, $p = 27779$ and $p = 15013$ codes.

N_b	$\mathbf{s}^0$	$\boldsymbol{\sigma}$	$\boldsymbol{\rho}$	$\mathbf{m}$	$\mathbf{s}^l$	**Total**
$p = 27,779$ code						
8	$2.5M$ 58%	$1M$ 25%	$438k$ 10%	$66k$ 1.5%	$204k$ 4.7%	$4.3M$
16	$1.2M$ 54%	$531k$ 24%	$219k$ 19%	$36k$ 1.6%	$204k$ 9.2%	$2.2M$
32	$0.6M$ 50%	$267k$ 22%	$111k$ 9.2%	$26k$ 2.2%	$204k$ 17%	$1.2M$
64	$0.3M$ 42%	$135k$ 18%	$54k$ 7.5%	$26k$ 3.6%	$204k$ 28%	$719k$
$p = 15013$ code						
8	$913k$ 55%	$304k$ 18%	$304k$ 18%	$36k$ 2%	$107k$ 6.4%	$1.6M$
16	$457k$ 51%	$152k$ 17%	$152k$ 17%	$21k$ 2%	$107k$ 12%	$889k$
32	$229k$ 45%	$76k$ 15%	$76k$ 15%	$16k$ 3%	$107k$ 21%	$507k$
64	$115k$ 36%	$38k$ 12%	$38k$ 12%	$18k$ 6%	$107k$ 34%	$316k$

The results of the synthesis with Design Compiler for the case $n_0 = 2$ and $p = 27,779$ are summarized in Table 5. The critical path delay is fairly constant across the considered degrees of parallelism and it is associated to the *collapse unit*.

As for the area occupation, it can be seen that, as expected, it increases regularly with the degree of parallelism.

Table 5: ASIC synthesis. Timing, area and power figures for the whole decoder and the main units. For each level of parallelism, the percentage increments are also given (in red). The decoder supports the code with $n_0 = 2$ and $p = 27,779$.

N_b	**8**	**16**		**32**		**64**	
Critical Path (ns)	3.72	3.72		3.72		3.86	
$\mathbf{s}^0$ (μm^2)	1,820	2,550	(34%)	4,146	(58%)	13,973	(266%)
$\boldsymbol{\sigma}$ (μm^2)	3292	5,408	(56%)	9,792	(75%)	25,636	(200%)
$\boldsymbol{\rho}$ (μm^2)	4,665	9,317	(100%)	17,643	(90%)	67,010	(279%)
$\mathbf{m}$ (μm^2)	3,854	5,729	(30%)	9,905	(76%)	18,172	(86%)
$\mathbf{s}^l$ (μm^2)	2,050	2,376	(15%)	3,007	(26%)	4,266	(41%)
Total Area (μm^2)	15,618	25,380	(62%)	44,493	(75%)	129,047	(190%)
Static Power (mW)	1.11	1.73		2.98		5.52	

The FPGA synthesis has been carried out with Xilinx Vivado, targeting the Artix-7 xc7a50tcpg236-3 device and setting the clock frequency at 100 MHz. The occupied resources are detailed in Table 6.

Table 6: FPGA synthesis. Occupied resources for the Artix-7 xc7a50tcpg236-3 device, in two applications: $n_0 = 2$, $p = 27,779$ code and $n_0 = 2$, $p = 15,013$ code. The total numer of IO and BUFG are 5 and 1 respectively independently of the parallelism.

Resource	Available	**Utilization with** $p = 27,779$				**Utilization with** $p = 15,013$			
N_b		8	16	32	64	8	16	32	64
LUT	32600	1554	2462	4330	8557	1327	2428	4125	8790
FF	65200	1007	1572	2675	4890	877	1592	2665	5488
BRAM	75	38	38	38	38	18.5	22.5	22.5	38

Although the hardware implementation of QC-LDPC code decoders for wireless communications has been deeply investigated [11], only a few dedicated architectures for code-based post-quantum cryptography are available in the open literature. The FPGA results reported in Table 6 can be compared against the LEDAcrypt implementation proposed in [14], which supports the $n_0 = 2$, $p = 15,013$ code with a bit parallelism of 32 bits: the reported resource usage for a Xilinx Virtex-6 device is 650 FFs and 2222 LUTs, the achievable clock frequency is 140 MHz and the required number of cycles is $2.62 \cdot 10^6$. Therefore, the architecture described in [14] has a decryption latency of 2,620,000 cycles at 140 MHz, corresponding to 18.7 ms. On the other side, the solution described in this paper achieves a decryption latency, for the same code and degree of parallelism, equal to 507,000 cycles at 100 MHz, equal to 5.05 ms.

6 Conclusions and Future Work

This paper presents a hardware implementation of the LEDAcrypt post-quantum cryptographic primitives based on QC-LDPC codes. A key advantage of the present design is the scalability of the obtained decoder, which allows for different trade-offs between computational time and

implementation cost. The decryption latency ranges between 3.16 ms up to 16 ms, for an internal degree of parallelism between 8 and 64.

As a future work, the effect of an extended parallelism in the critical processing tasks could be explored more in detail. Given the large impact of processing parallelism on the total amount of occupied resources or Silicon area, several architecture-level choices have to be explored in order to balance the reduction of the decoding time and the increase of the implementation cost. A second line of research for future works can aim at improving the implementation efficiency by means of joint algorithm and architecture optimizations. As an example, possible algorithm simplifications can be investigated to evaluate both the effects on the cryptographic primitives and the provided advantages in terms of their hardware implementation.

References

[1] Rashmi Agrawal, Lake Bu, Alan Ehret, and Michel A. Kinsy. Open-source fpga implementation of post-quantum cryptographic hardware primitives. In *Field Programmable Logic and Applications (FPL), 2019 International Conference on*, Sep. 2019.

[2] Frank Arute, Kunal Arya, Ryan Babbush, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 2019.

[3] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini. Ledacrypt. https://github.com/LEDAcrypt/LEDAcrypt/tree/master last viewed February 2019, 2019.

[4] M. Baldi, M. Bodrato, and F. Chiaraluce. A new analysis of the McEliece cryptosystem based on QC-LDPC codes. In *Proceedings of the 6th international conference on Security and Cryptography for Networks (SCN 2008)*, pages 246–262, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt: QC-LDPC code-based cryptosystems with bounded decryption failure rate. In Marco Baldi, Edoardo Persichetti, and Paolo Santini, editors, *Code-Based Cryptography*, pages 11–43, Cham, 2019. Springer International Publishing.

[6] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *Information Theory, IEEE Transactions on*, 24(3):384 – 386, 5 1978.

[7] Daniel J. Bernstein. Grover vs. McEliece. In *Proceedings Post-Quantum Cryptography: Third International Workshop (PQCrypto 2010)*, pages 73–80, Darmstadt, Germany, 5 2010. Springer Berlin Heidelberg.

[8] K. Braun, T. Fritzmann, G. Maringer, T. Schamberger, and J. Seplveda. Secure and compact full ntru hardware implementation. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 89–94, Oct 2018.

[9] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. *National Institute of Standards and Technology Internal Report*, 8105, 2016.

[10] Tung Chou. Qcbits: Constant-time small-key code-basedcryptography. https://www.win.tue.nl/~tchou/papers/qcbits.pdf, 2016.

[11] C. Condo, M. Martina, and G. Masera. A network-on-chip-based turbo/ldpc decoder architecture. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1525–1530, 2012.

[12] F. Farahmand, M. U. Sharif, K. Briggs, and K. Gaj. A high-speed constant-time hardware implementation of ntruencrypt sves. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 190–197, 12 2018.

[13] J. Howe, C. Moore, M. O'Neill, F. Regazzoni, T. Gneysu, and K. Beeden. Lattice-based encryption over standard lattices in hardware. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[14] J. Hu, M. Baldi, P. Santini, N. Zeng, S. Ling, and H. Wang. Lightweight key encapsulation using ldpc codes on fpgas. *IEEE Transactions on Computers*, pages 1–1, 2019.

[15] J. Hu and R. C. C. Cheung. Area-time efficient computation of niederreiter encryption on qc-mdpc codes for embedded hardware. *IEEE Transactions on Computers*, 66(8):1313–1325, Aug 2017.

[16] B. Koziel, R. Azarderakhsh, and M. M. Kermani. A high-performance and scalable hardware architecture for isogeny-based cryptography. *IEEE Transactions on Computers*, 67(11):1594–1609, Nov 2018.

[17] B. Koziel, R. Azarderakhsh, M. Mozaffari Kermani, and D. Jao. Post-quantum cryptography on fpga based on isogenies on elliptic curves. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(1):86–99, Jan 2017.

[18] Ingo Von Maurich, Tobias Oder, and Tim Güneysu. Implementing qc-mdpc mceliece encryption. *ACM Trans. Embed. Comput. Syst.*, 14(3), April 2015.

[19] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, January 1978.

[20] National Institute of Standards and Technology. Post-quantum cryptography project.

[21] National Institute of Standards and Technology. Post-quantum cryptography project - round 2 submissions.

[22] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process.

[23] Deepraj Soni. A hardware evaluation study of nist post-quantum cryptographic signature schemes. In *Second PQC Standardization Conference*, Santa Barbara, CA, August 2019.

[24] Wen Wang, Jakub Szefer, and Ruben Niederhagen. Fpga-based niederreiter cryptosystem using binary goppa codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 77–98, Cham, 2018. Springer International Publishing.