MASTER COURSE IN ELECTRONIC ENGINEERING

THESIS

A new algorithm for structured matrix-vector product for Post quantum cryptography

Supervisor: Guido Masera

Author: Paolo Mazzetti

Academic year 2020 - 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last few years the huge steps forward made in technology have put ourselves in the condition of facing and trying to solve new problems and challenges that were not even thinkable just a few years ago.
One of this fields is cryptography: this sector of science takes care of finding ways to establish a secure communication, so that a message can safely be transmitted from the sender, who is in charge to encrypt it, to the receiver, whose duty is to decrypt it, such that third parties will not be able to recover it.
The sender and the receiver will know more information rather than just the message, but the cryptography algorithm has to be safe enough so that attackers cannot decrypt the message even with a brute-force approach. As a general idea, the sender encrypts the message using a key (it is a secret or a public key based on the algorithm) and the receiver decrypts the message using either the same (secret) key or another secret key.
Cryptography is nowadays used, for example, to make safe the login process to access our bank account, or to send a message using any instant messaging platform (many of them nowadays use with the end-to-end cryptography, and with it no one can read the messages except from the sender and the receiver, even the platform itself is not able to do it), but it wasn't thought for just these reasons.

One of the first example is the Caesar cipher: every letter of the message we want to send is substituted by another letter that is placed a fixed number of positions forward or backward in the alphabet. If the receiver known the number of shifts used by the sender, it is rapidly able to retrieve the message and read it. This algorithm is very easy to be implemented: however the encrypted message we send can be easily decrypted by an attacker in two ways: in fact, in every language certain characters are more frequent than others, so using this information one can easily determine the number of shift performed by the sender; moreover the attacker can use a brute-force approach: he can try every number of shifts until it gets back a readable and meaningful message.
A system like the one presented above is clearly not enough to safely send a message: many algorithms have been developed in the last decades, so that any attacker cannot try to decrypt a message with a brute-force approach because with actual computers he would not be able to decrypt the message in a reasonable amount of time.

With the advent of computers the realization of much more complex ciphers has been possible. On the other hand, modern computers have also allowed significant progresses in the cryptanalysis, the counter part of the cryptography. Although this effect has to be taken into account, cryptographic algorithms must be computational efficient in both encryption and decryption, while for the cryptanalysis we require much more resources that make this process usually not worth.
Symmetric-key cryptography is the first modern cryptography class of algorithms: both the sender and the receiver know the secret key, which is used to both encrypt and decrypt the message. However the key has to be complex enough so that an attacker would not be able to break the algorithm with a brute-force approach, and since the key has to be known by both the people that are communicating there is at least a moment in which the key has to shared over a channel and so it is vulnerable by any attacker.

The problems described above have been overcome with the rise of the asymmetric-key cryptography: for each actor in the communication a pair of keys is produced; a public key is used to encrypt the message, while a secret key, known only by the receiver, is used to decrypt the message. The main advantage of this class of algorithms is that no key has to be send from the sender to the receiver, so the attacker can break the algorithm only retrieving the secret key from the public key: this process in general is very computationally expensive so these algorithm, like the RSA algorithm, are widely used today.
However there are always new challenges to face: in the next few years quantum computers will be available. These computers are not just normal computers a little bit more powerful: the way they perform the operations will break in a reasonable period of time most of cryptographic algorithm used today. In order to solve this issue the post-quantum cryptography has been developed: in the last few years new algorithms have been proposed and improved in order to be resistant to quantum computers when they will be available.

Among the algorithms that have been defined there are the code based ones, which are based on problems well studied in the last few decades, such as the error-correcting codes (ECC). Several classes of ECC have been defined throughout the years, like the Low Density Parity Check (LDPC) codes: their main advantage is that they can be stored and managed limiting as much as possible the memory required without compromising the required security level and at the moment no algorithms are known to decode them in a polynomial time on a quantum computer. There are several post-quantum cryptography algorithms based on the ECC and one of them LEDAcrypt: it is a valuable candidate for the post-quantum cryptography since it provides good security levels without requiring too many resources in the encoding and decoding processes, since it exploits the Quasi Cyclic LDPC codes, a subclass of LDPC codes which can lead to important resources savings and simple encoding procedures.

LEDAcrypt performs several operations involving vectors and matrices and hence efficient implementations of them are required: in particular one important operation (common to many algorithms, not only LEDAcrypt) is the cyclic multiplication, in which a vector and a circulant matrix have to be multiplied. Several possible implementations have been presented in the past, since the matrix we have to deal with is also sparse and binary: these characteristics can be exploited in several ways in order to realize an efficient implementation.
The present work focuses on the cyclic multiplication and suggests a different solution with respect to the ones already present in literature. In fact, the implementation is based on the convolution theorem: the operation involves the input vector and the first row of the matrix and the final product can be obtained without directly computing the multiplication. Specifically the convolution theorem is based on the Fourier transform: this work is instead based on the Number Theoretic Transform, a specific version of the Fourier transform that works in modular arithmetic: this gives us the advantage of using integer units and avoiding any kind of precision error. This work concentrates on the architecture implementation but also presents a comparison with the existing alternative approaches in order to analyze benefits and drawbacks of this solution and evaluate whether this multiplication method is faster or less expensive in terms of area and/or power.

Chapter 2

Modern cryptography

The first cryptography algorithms were all based on the symmetric-key approach: the key is known by both the sender and the receiver and it is used to both encrypt and decrypt the message.
Starting from the 70s, public-key (or asymmetric-key) cryptography algorithms took place: they solve one of the major problems of the previous algorithms, that is the secure management of the keys.
In the last few years new challenges have to be considered and post-quantum cryptography is a major field of study for the near future.

2.1 Symmetric-key cryptography algorithms

With a symmetric-key algorithm [1] the same key is used to encrypt the plaintext and then decrypt the ciphertext.
If Bob wants to send a message to Alice, he uses the key to encrypt the message, then Alice uses the same key to decrypt what she receives. The category of symmetric-key cryptography algorithms also includes algorithms in which two keys are used (one to encrypt the message and the other one to decrypt the message), but one key can be obtained from the other one with an easy transformation (this is not true for the asymmetric-key cryptography algorithms).
The major drawback of this system is the fact that the key has to be known by both the sender and the receiver: so the two parties have to find a way to secretly share the key before the communication can safely start.
Moreover, each couple of communicating people needs a pair of key, so the number of keys required rises as the square of the people present in the network.
Symmetric-key algorithms can be divided in two groups:

- Stream ciphers;
- Block ciphers.

The stream cipher encrypt every element of the plaintext one by one, while with a block cipher a subset of the plaintext is considered and encrypted in one step.

2.2 Asymmetric-key cryptography algorithms

Asymmetric-key algorithms [2] differs from the previous ones by the fact that two keys are used, and one of them is public: a key is used to encrypt the plaintext and the other one is used to decrypt the ciphertext. The two keys are related, but every algorithm is designed in such a way that any attacker is not able to easily derive one key from the other one.
Each member of the network is identified by a pair of keys, and one of them is public: with one of this algorithms, if Bob wants to safely send a message to Alice, he can encrypt the message with Alice's public key, the Alice will decrypt the message with her private key.
Asymmetric-key algorithms can be used also to produce a digital signature: in this case, if Bob wants only to sign a message he uses his private key to do it, then Alice can use Bob's public key to decrypt the sign, so knowing that the message has been sent from Bob and has not been modified.
The most famous asymmetric-key algorithms are based on the hardness of finding the solution on one of the two following problems:

- Discrete logarithm;
- Integer factorization.

2.2.1 The integer factorization problem

This problems consists of finding the factors of an integer number: if the get prime numbers, we can also talk of prime factorization. Several cryptography algorithms can rely on the fact that for a very large integer number, there are no algorithms able to find its factors in a polynomial time on a classical computer, but only in a sub-exponential time.

The RSA algorithm

The RSA algorithm has been presented in 1977 by Ron Rivest, Adi Shamir and Leonard Adleman.
Starting from two close prime numbers p and q we can compute their product $n = pq$. The product n is part of the public key, while the two prime numbers are not and if n is large enough (typically 2048 bit) it is really hard to find its factors. We then compute $\lambda(n)$ as the least common multiple between $p-1$ and $q-1$ (where λ represent the Carmichael function): from this value we pick an integer number e which is coprime with $\lambda(n)$ and in the end we compute d as the inverse of e under the modulo $\lambda(n)$.
The integer e is part of the public key, while d belongs to the private one: since no fast algorithms are able d from e, or at least they are not known yet, the algorithm has to be considered secure enough.
This algorithm can be used to safely send a message: we encrypt the message m with the public key of the receiver ($c = m^e(modn)$) and we then decrypt it with the secret key (we get back m as $c^d(modn)$).

With the RSA algorithm we can also sign a message so that when we receive a message we are able to understand who is the sender: in this case we encrypt the message with the sender's secret key d and we decrypt it with his public key e.

2.2.2 The discrete logarithm problem

Another problem on which modern cryptography algorithms are based on is the discrete logarithm problem. First of all we can define a cyclic group: a group is defined as cyclic if we can find a generator of the group, so that if we repeatedly apply the same operation to it we can obtain all the elements of the group itself. For example the multiplicative group modulo 5, $\mathbb{Z}_5^*$, is cyclic and the number 2 is a generator of the group: in fact, continuously multiplying 2 by itself and performing the operation modulo n we can obtain all the elements of the group. Considering a multiplicative cyclic group, the discrete logarithm can be defined: considering the same example, for the group $\mathbb{Z}_5^*$ and the generator 2, the discrete logarithm of 3 is 3 because $2^3 = 8 = 3\, mod\, 5$.
Discrete logarithms cannot be efficiently computed in general on a classical computer, so this problem is the base on which several cryptographic systems rely on.
The Diffie–Hellman key exchange algorithm is based on the discrete logarithm problem: it has similar performances with respect to the RSA algorithm in terms of security and speed, it is just based on a different on a different hard problem.

2.3 Post-quantum cryptography algorithms

The advent of quantum computers requires new algorithm for the cryptography: several systems used today rely on the hardness on integer factorization of numbers with hundreds of digits, while other systems are based on the discrete logarithm problem: both this problems can be solved using the Shor's algorithm on a quantum computer in a polynomial time. Hence, cryptography algorithms for which there is no algorithm that could break them in a polynomial time on a quantum computer have to be proposed.
In general post-quantum cryptography algorithms are lattice based or code based, but there are also other possibilities [3].

Shor's algorithm

Shor's algorithm is able to find the prime factors of an integer N. It can be demonstrated that, on a quantum computer, it can find these factors in a polynomial time, that is approximately $logN$. It is much faster with respect to any algorithm running on a classical computer, which require at least a sub-exponential time. The quantum part of the algorithm exploits the quantum Fourier transform, a fast procedure on a quantum computer. Thanks to the Shor's algorithm, all the cryptography algorithms presented so far can be broken in a reasonable time, so completely different algorithms have to be introduced in order to resist to the attacks from a quantum computer.

2.3.1 Lattice based algorithms

Lattice based algorithms [4] are good candidates for the post-quantum cryptography because they are resistant to attacks from both classical and quantum computers and they also present very good performances; however, they seem to be the least conservative among the different solutions.
A lattice $\mathcal{L}$ in $\mathbb{R}^n$ is a subset of $\mathbb{R}^n$: given a basis vector $\{b_1, ..., b_n\}$ for $\mathbb{R}^n$, every linear combination of them is an element of the lattice L; the coefficients of the basis are integers values, so we have:

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z} \right\} \tag{2.1}$$

Of course for a lattice several bases exist: in any case, the number of elements of each basis is always n in the space $\mathbb{R}^n$, and all the elements of the basis are orthonormal with respect to the other ones.
There are several optimization problems that involve lattices and lattice-based cryptography has been developed based on the fact that these problems are not *P* problems on any classical or quantum computer. The *P* versus *NP* problem is still an open question, and it asks if a problem for which a given solution can be checked quickly (so in a polynomial time) is also a problem that can be solved quickly: *NP* problems are non-deterministic polynomial time problems, so for them no solution in a polynomial time is known. Many lattice-based problems are known to be NP-hard (which means that they are at least hard to solve as NP problems), and NP-hard problems cannot be solved in a polynomial time (even if this is not proved, it is suspected it is so).
Several lattice-based problems can be used for cryptographic purposes and among them there are:

- The Shortest Vector Problem (SVP): given a generic basis for a lattice $\mathcal{L}$, the goal is to find the shortest non-zero vector $v \in \mathcal{L}$. The problem is NP-hard, in fact there are no polynomial time algorithms available to find the exact solution. A polynomial time algorithm exists only to find an approximated version of v: in fact we can find a vector whose dimension is no more than a certain value we decide. This is a problem on which cryptographic algorithms can rely on since so far, even with quantum computers, no fast algorithms are known to find the exact solution;

- Closest vector problem (CVP): given a generic basis for a lattice $\mathcal{L}$ and a vector v (that may belong to $\mathcal{L}$ or not), the goal is to find in $\mathcal{L}$ the closest vector to v. It can be seen as a generalization of the previous problem and it is as hard as the SVP is.

The NTRUEncrypt public key cryptosystem

This algorithm is based on the Shortest Vector Problem, so it can resist to an attack by a quantum computer. It works with a polynomial ring and its security performances are granted by the fact that, at the moment, there are no algorithms, even on quantum computers, capable of such fast polynomial factorization.

2.3.2 Code based algorithms

Code based algorithms are also good candidates for post-quantum cryptography: they are based on problems well studied in the last few decades, such as the error-correcting codes (ECC). In fact, they were first introduced by Richard Hamming in the 1940s and since then many versions and many uses have been proposed. The general idea of the ECC is introduce redundancy in order to be able to retrieve a message sent on a noisy communication channel without the need of asking for a retransmission. In Code based algorithms instead, a certain numbers of intentional errors is introduced, then thanks to the ECC used we are able to correct them and decode the correct message.
Code based algorithms are good candidates for post-quantum cryptography since as for the lattice problems, until now there are no known algorithms able to break an ECC in a polynomial time, even on a quantum computer. In fact, decoding a general ECC is known to be an NP-hard problem and algorithms such as the Shor's one cannot be used.
Even though the preliminary security requirements can be met, the major issue regarding code based algorithms is the keys size: in fact codes have to be large enough to provide a sufficient security level but it is also required an efficient method to store compact keys. In order to do so there are several error correcting codes which are able to provide an adequate security level limiting as much as possible the resources required to store data. Among these codes linear codes (codes for which any linear combination of codewords is also a codeword) are widely used in post-quantum cryptography algorithms since in general they provide a good security level with an efficient decoding procedure.

LDPC codes

Low Density Parity Check codes, or LDPC codes, are linear error correcting codes developed by Robert G. Gallager in 1960 but well studied only in the last few decades. Their main advantage is that they can be used to transmit over a noisy channel and the amount of information to be sent can reach almost the Shannon limit.
LDPC codes are generated thanks to a low density parity check matrix and they can be graphically represented with Tanner graphs. LDPC can be regular or irregular: for the first ones the regular structure of the matrix allows to store less elements and hence reducing the memory resources required.

Alongside the LDPC codes there are also the MDPC ones, i.e. Medium Density Parity Check codes. Basically they are LDPC with an higher density: hence more memory to store the keys is needed, but the similar security performances can be reached.

2.3.3 LEDAcrypt

Among the code based algorithms that have been defined throughout these years there is LEDAcrypt [5]: it is a post-quantum asymmetric-key cryptographic algorithm based on Low Density Parity Check (LDPC) codes. In particular quasi-cyclic LDPC codes are used, so that the size of the keys can be shrunk without compromising security: in fact only the first row of the sparse matrix has to be stored.
LEDAcrypt algorithm is composed of a secret key and a public key:

- For the secret key we construct a group of sparse binary circulant matrices $H = [H_0|H_1|\dots|H_{n_0-1}]$ (with n_0 in general between 2 and 4), so that we need to store only the positions of the ones of the first row, greatly limiting the size needed to store them. From this secret matrix H we generate another group of sparse matrices Q and the secret key is $SK = \{H, Q\}$;
- Multiplying H by Q we can obtain the matrix $L = [L_0|L_1|\dots|L_{n_0-1}]$: L_{n_0-1} has to be invertible and we use it to compute $M = L_{n_0-1}L$ which corresponds to the public key. M is a dense circulant matrix, so the whole first row has to be stored.

Once we have the keys we can safely use this algorithm: in the message we want to send we insert a certain number of intentional errors, than thanks to the public key of the receiver we produce the vector we send him; the receiver instead relies on his secret key to successfully decrypt the message.

The algorithm is designed with an hard-decision decoder: this means that for each bit it has to be decided whether it is a 0 or a 1. In Ledacrypt an optimized bit-flipping decoder is used: it performs several cyclic multiplications between a binary sparse circulant matrix and a vector (that be either sparse or dense, either binary or integer) and the process is iterated until the message is correctly decoded or a maximum number of iterations has been reached. In fact, due to the intrinsic LDPC codes characteristics, a decryption failure rate (DFR) has to be accepted, but it can be made very low and almost negligible.
The decoding procedure exploits the correlation among the intentional errors introduced in order to enhance its performances; moreover the threshold on the basis of which we decide whether we need to flip a bit or not is variable and it is computed at each iteration.

The algorithm has reached the round 2 submission for the Post-Quantum Cryptography NIST call since it provides good security levels limiting as much as possible the key size.

Chapter 3

Cyclic multiplication

The vector by circulant matrix operation is an important computational part of the LEDAcrypt algorithm: it consists of the multiplication between a vector and a sparse binary circulant matrix, that is a matrix in which every row is a shifted version of the previous one. Since it is used several times throughout the algorithm, an optimization of this computation can lead to an overall improvement of the performances of the whole algorithm.

3.1 Vector by circulant operation

In general we want to perform the multiplication exploiting the advantage of the sparse binary circulant matrix: we don't want to realize a straightforward implementation of the multiplication since it would require too much resources and it will be too slow. Moreover, in order to reduce as much as we can the memory occupied by the matrix, we store in memory the positions of the 1's of the first row only.
Several solutions have been proposed in the past: let's see one of them with a simple example. We want to multiply a vector $x = [x_0\ x_1\ x_2\ x_3\ x_4]$ by a circulant matrix Y whose first row is equal to $y = [0\ 1\ 0\ 0\ 1]$. The output vector will be:

$$\begin{aligned} z_0 &= x_1 + x_4 \\ z_1 &= x_2 + x_0 \\ z_2 &= x_3 + x_1 \\ z_3 &= x_4 + x_2 \\ z_4 &= x_0 + x_3 \end{aligned} \tag{3.1}$$

As we can see, the output vector is made of a combination of shifted versions of the input vector. So we can realize an architecture in which we sum shifted versions of the input vector: for each shifted version the offset is given by the position.
A first implementation can consider the update of one component of the output vector at a time, but an improved version can manage the update of several components at a time, as shown in [6] and in figure 3.1. For each 1's position we generate the proper shifted version of the input vector and we sum it to the

output vector (the choice among the adders and the EXOR gates depends on whether the output vector is respectively integer or binary). The position value determines also how much we need to shift each row of the input vector memory: in order to handle all the possible cases a collapse unit is present, (it is a collection of multiplexers).

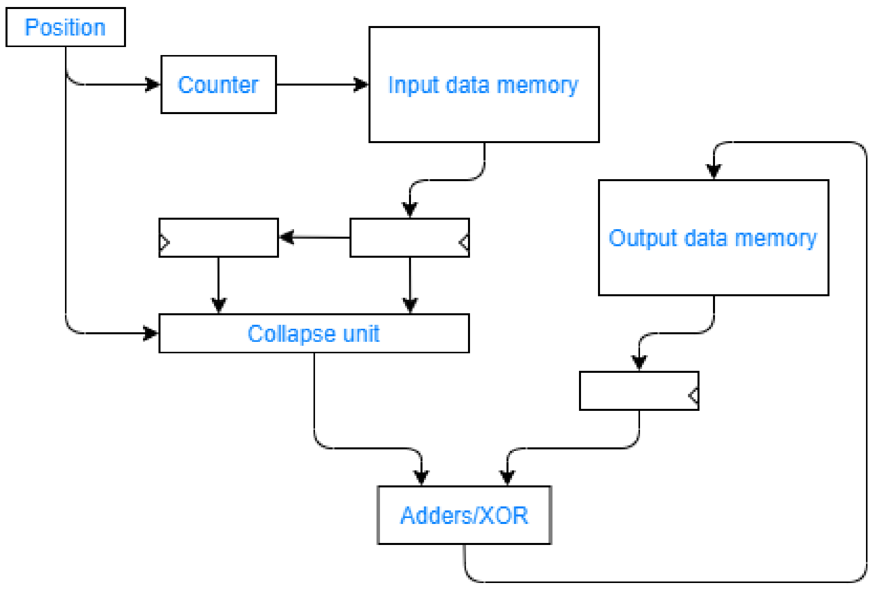


Figure 3.1: Vector by circulant architecture

3.1.1 An attempt to optimize the solution

The bottleneck of the previous solution is the collapse unit, in fact this component influences a lot both the area and the maximum speed the circuit can achieve: the most important thing is that its contribution grows exponentially if we increase the memory parallelism, so we would like to have a small parallelism but this impacts on the number of clock cycles required to complete the operation.
Hence, one thing we can try to do is exploiting another level of parallelism: the idea is that we can organize the memory in an interleaved way so our optimal solution will depends both on the memory parallelism and on the number of blocks we decide to use. In this way we can keep low the memory parallelism reducing the total number of clock cycle required: an example with a split factor equal to 2 is presented in figure 3.2, where the split factor indicates in how many blocks we divide the data memory.
This solution reduces the dimension of the collapse units at the cost of minor additional components and an additional level of multiplexers. Unfortunately, the drawback of this solution is that the cost of these multiplexers and the additional support hardware required exceeds the one of the collapse unit: in fact the collapse unit can be organized as a logarithmic shifter, while this is not possible for these new multiplexers. The two solutions are comparable with a split factor equal to 2, while the cost of the new solution increases too much if we increase the split factor, so it has to be discarded.

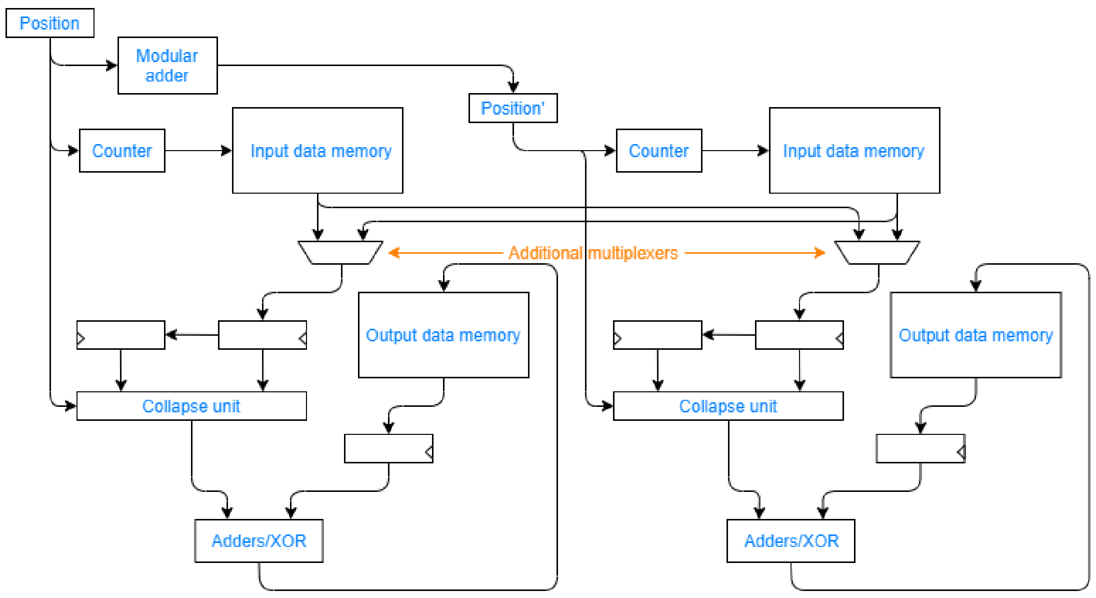


Figure 3.2: Vector by circulant split architecture

3.2 Sparse vector by circulant

In the LEDAcrypt algorithm in some cases we also have to multiply a sparse binary matrix by a sparse vector: the output vector in this case can be produced with a completely different architecture, in which we consider the positions of the 1's in both the vector and in the first row of the matrix.

3.3 A different approach

Since minor improvements of these solutions have been proposed so far, a completely new algorithm is presented and implemented in the next chapters in the attempt to find a better solution.
In the next chapters we will analyze a solution capable of handling both dense and sparse vectors. The main advantage of the new approach used in the continuation of the work is that the solution with minor changes will be capable of handling also non-binary circulant matrices, expanding the applications of the algorithm.

Chapter 4

The cyclic convolution theorem and its use

Given two vectors x and y of N components, the resulting convolution vector w, of *2N-1* components, is $w = x \star y$:

$$w_i = \sum_{k=max\{1,i-N+1\}}^{min\{i,N\}} x_k \cdot y_{k-i+1} \tag{4.1}$$

The cyclic convolution $z = x \circledast y$ is the result of the sum of the first N components with the last *N-1* components. It is a vector of N components, defined as:

$$z_i = \sum_{k=0}^{N-1} x_k \cdot y_{(i-k)\, mod\, N} \tag{4.2}$$

We can explicitly write the computation of each component of this vector, considering an example of vectors with 4 components:

$$\begin{aligned} z_0 &= x_0 \cdot y_0 + x_1 \cdot y_3 + x_2 \cdot y_2 + x_3 \cdot y_1 \\ z_1 &= x_0 \cdot y_1 + x_1 \cdot y_0 + x_2 \cdot y_3 + x_3 \cdot y_2 \\ z_2 &= x_0 \cdot y_2 + x_1 \cdot y_1 + x_2 \cdot y_0 + x_3 \cdot y_3 \\ z_3 &= x_0 \cdot y_3 + x_1 \cdot y_2 + x_2 \cdot y_1 + x_3 \cdot y_0 \end{aligned} \tag{4.3}$$

From this example, we can derive that each component of the vector x is multiplied by a shifted version of the vector y. We can rearrange (4.3) in a matrix form according to [7], writing the matrix $C(y)$:

$$C(y) = \begin{bmatrix} y_0 & y_{N-1} & y_{N-2} & \cdots & y_1 \\ y_1 & y_0 & y_{N-1} & \cdots & y_2 \\ y_2 & y_1 & y_0 & \cdots & y_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{N-1} & y_{N-2} & y_{N-3} & \cdots & y_0 \end{bmatrix} \tag{4.4}$$

As we can see, this is a circulant matrix: in fact, each column is a shifted version of the first column. Hence we can write the relationship:

$$z = x \circledast y = x \times C(y) \tag{4.5}$$

Given this relationship, the cyclic convolution theorem states that:

$$z = DFT^{-1}(DFT(x) \odot DFT(y)) \tag{4.6}$$

where $\odot$ stands for element-wise product between the 2 vectors and DFT represent the Discrete Fourier Transform. Hence, we can compute a vector by circulant product, as in equation (4.5), computing a couple of DFT and an inverse DFT. If we use the DFT the time required to compute the cyclic convolution is $O(N^2)$ time, but we can use the FFT so that we require only a $O(N \log N)$ time. However, most of the FFT algorithms are tailored for a number of points that is a power of 2. If $N \neq 2^m$ there are several possibilities in order to still compute the product in a $O(N \log N)$ time like the prime-factor FFT algorithm, which re-writes the FFT of $N = N_1 N_2$ points as a rearrangement of two FFT of sizes N_1 and N_2, but this is valid only if N_1 and N_2 are relatively prime numbers. If instead N is already prime, two algorithms have been proposed in the previous years: the Bluestein's algorithm and the Rader's algorithm, where both the algorithms rewrites the DFT as a cyclic convolution.
The Bluestein's algorithm starts from the expression of the DFT:

$$X_k = \sum_{i=0}^{N-1} x_i e^{-\frac{2\pi j}{N} ik} \tag{4.7}$$

Each ik product in the exponent can be replaced exploiting the identity:

$$ik = -\frac{(k-i)^2}{2} + \frac{i^2}{2} + \frac{k^2}{2} \tag{4.8}$$

Rewriting the whole expression, we get:

$$X_k = e^{-\frac{\pi j}{N} k^2} \sum_{i=0}^{N-1} (x_i e^{-\frac{\pi j}{N} i^2}) e^{-\frac{\pi j}{N} (k-i)^2} \tag{4.9}$$

Since this summation is a convolution between $x_i e^{-\frac{\pi j}{N} i^2}$ and $e^{\frac{\pi j}{N} i^2}$ we can rearrange this computation with a pair of FFT (plus the pre-computed FFT of $e^{-\frac{\pi j}{N} k^2}$) exploiting the convolution theorem. The two vectors are zero-padded in order to have a size equal to $m = 2^{\lceil \log_2 (2N-1) \rceil}$: since m is a power of 2, we can now use one of the known FFT algorithms that require a $O(N \log N)$ time.

The alternative to these solution is proceeding with a zero-padding: in this case we zero pad the vectors of size N up the following power of 2, so that the size becomes $m = 2^{\lceil \log_2 N \rceil}$ and then we can proceed with one of the known fast algorithms. However, with the zero padding procedure two problems arise: if N is slightly larger than a power of 2, many zeros have to be introduced, possibly complicating too much the computations: nevertheless this is a problem that has to be handled also by Bluestein's or Rader's algorithms; moreover zero padding is not always possible, because based on the computation we have to do it can produce a non correct result: this is our case.

Let's consider an example in which we have $x = [0\ 0\ 0\ 0\ 1]$ and $y = [0\ 1\ 1\ 0\ 1]$, where y is the vector that produces the circulant matrix, the matrix multiplication $x \times C(y)$ is:

$$x \times C(y) = [0\,0\,0\,0\,1] \times \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{4.10}$$

The result of this computation is $z = [1\ 1\ 0\ 1\ 0]$ and it can be achieved either directly computing the product or exploiting the convolution theorem. The zero pad on both the vectors x and y should give a resulting vector z' in which the first 5 elements are equal to the ones of z, while the last three elements can be discarded. However, proceeding in this way, we get $z' = [1\ 0\ 0\ 0\ 0\ 1\ 1\ 0]$: considering only the first five elements of this new vector, the result is clearly not correct. This is because zero pad on the x vector is possible, but if we zero pad the y vector the circulant matrix will be:

$$C'(y') = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{4.11}$$

Instead, since we want that the first five elements of z' are equal to z, we need that the 5×5 top-left sub matrix in $C'(y')$ is equal to $C(y)$, as shown in red in the matrix below:

$$C'(y') = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{4.12}$$

In this matrix, the vector y' is also encircled with a dashed blue line: as we can see, it is not a zero padded version of y, but the additional elements are decided so that in the top-left corner of $C'(y')$ there is always $C(y)$. It is always possible to extent the vector y in order to maintain $C(y)$ in the top-left corner of $C'(y')$: however, for our application, the rule is always valid if the vector is extended to a number of elements equal to $m = 2^{\lceil \log_2 (2N-1) \rceil}$, as for the Bluestein's algorithm (so with a vector of 5 elements we have to produce a vector of 16 elements).

Using this new circulant matrix and the zero padded vector $x' = [0\ 0\ 0\ 0\ 1\ 0\ 0\ 0]$ as a result we get $z' = [1\ 1\ 0\ 1\ 0\ 1\ 1\ 0]$: simply discarding the last 3 numbers, we get the same resulting vector z as before.

If instead we have to compute the $C(y) \times x$ product the procedure is almost the same: instead of handling the 1's positions of the first column, we will manage the 1's positions of the first row, but the rest of the algorithm and the architecture will be the same.

Hence, so far we have considered two possibilities if the number of points of the *DFT* is a prime number:

- The use of one special algorithm among Bluestein and Rader;
- The padding approach described above.

At this moment a consideration on the data we have has to be made: since we always handle sparse circulant matrices, and sometimes we also have sparse vectors, the padding approach has to be preferred: in fact, when we have a little number of non-zero values we can optimize the algorithm reducing the amount of resources required since many partial results are zero and we can avoid their computation. Instead, with one of the special algorithms shown before, this advantage cannot be fully exploited: since we have to perform a convolution, we may have the possibility to compute two sparse *DFT*, but at least the *IDFT* will be not sparse, so we require a full allocation of the resources.
Moreover, we have to consider that in general the input values of the *FFT* are complex numbers, stored in a floating point representation: instead in the case of all the considered cryptographic algorithms, all the numbers to be handled are integers (and sometimes they are binary: this additional advantage is considered later on in the work). Hence, instead of using the *FFT* we will use the Number Theoretic Transform (*NTT*).

4.1 The Number Theoretic Transform

The Number Theoretic Transform is a specialized version of the Fourier transform: instead of using complex numbers, it works with integers in modular arithmetic and the main advantage is that we avoid any kind of precision error.
In modular arithmetic, only integer values are allowed and numbers "wrap" on themselves when they reach a certain value (the modulus, P). For example, if the modulus is $N = 8$, only integer values from 0 to 7 are allowed and every number greater than 7 is re-expressed with a number between 0 and 7: for example 8 becomes 0, 9 becomes 1 and so on. As a general rule, every number greater than the modulus is divided by the modulus, then the reminder is kept. Negative integers are included (down to $-P + 1$), or they can also be re-expressed by their positive counter part: so $-1 mod 8$ becomes 7, $-2 mod 8$ becomes 6 and so on.

The transformation of a vector x of $N = 4$ elements with DFT and the NTT is presented: the DFT can be expressed with the DFT matrix as shown below:

$$\mathcal{F}(x) = \begin{bmatrix} w^{0\cdot0} & w^{0\cdot1} & w^{0\cdot2} & w^{0\cdot3} \\ w^{1\cdot0} & w^{1\cdot1} & w^{1\cdot2} & w^{1\cdot3} \\ w^{2\cdot0} & w^{2\cdot1} & w^{2\cdot2} & w^{2\cdot3} \\ w^{3\cdot0} & w^{3\cdot1} & w^{3\cdot2} & w^{3\cdot3} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{4.13}$$

where $w = e^{\frac{-2\pi i}{N}}$. With the NTT instead, w, called α from now, is a natural number so that $\alpha^N = 1 \, mod \, P$ and all the results are expressed in $mod \, P$, but the computation is the same:

$$\mathcal{N}(x) = \begin{bmatrix} \alpha^{0\cdot0} & \alpha^{0\cdot1} & \alpha^{0\cdot2} & \alpha^{0\cdot3} \\ \alpha^{1\cdot0} & \alpha^{1\cdot1} & \alpha^{1\cdot2} & \alpha^{1\cdot3} \\ \alpha^{2\cdot0} & \alpha^{2\cdot1} & \alpha^{2\cdot2} & \alpha^{2\cdot3} \\ \alpha^{3\cdot0} & \alpha^{3\cdot1} & \alpha^{3\cdot2} & \alpha^{3\cdot3} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} (mod \, P) \tag{4.14}$$

We can also express the NTT with a summation, equivalent to the one of the DFT:

$$X(k) = \sum_{i=0}^{N-1} x(i)\alpha^{ik} \, (mod \, P) \tag{4.15}$$

where N is the number of points on which we perform the transformation, P is the modulus and α is the radix.

The NTT is defined if the modulus is prime and in the form $P = kN + 1$, for some natural number k: P can assume infinite values, but usually the minimum value of P is used, in order to limit the numbers' size in the NTT and so, for an hardware implementation, the number of bits needed. However, since we are performing a convolution, another condition on the minimum value for P has to be set: if the two vectors involved have size N and each element of both the vectors can assume a maximum value equal to M, we need to have $P > M^2 \cdot N$, otherwise we could have an overflow in the computation of the convolution that yields to a non correct result; hence the minimum P value is given by these two constraints.
Once we have P, with N, P and k we can compute the radix α:

- First of all, since P is prime, at least one primitive root of unity r can be defined: it is a value such that, if we compute $r^x \, (mod P)$ with x going from 1 to $P-1$, we get all the numbers between 1 and $P-1$ (the order in which these numbers are is not important); if r is not a primitive root of unity when this value is raised to all the possible x values some output values will be equal to each other and no all the numbers between 1 and $P-1$ will be present;

- Once r has been computed, the radix for the NTT can be derived as $\alpha = r^k \, (mod \, P)$.

The computation of r and then α has to be done just once, since depends on the number of points of the NTT N and on the modulus P, which also can has to be computed only one time since it depends only on the number of points N. However, with an high number of points it could require too much time span r between 2 and $P-1$ and for each of these values compute r^x $(mod\, P)$ for any value of x between 1 and $P-1$. But another possibility exists: since P is prime, we know that for every integer a the relationship $a^{P-1} = 1\, mod\, P$ is valid; but $a = r$ is a primitive root if and only if its order is equal to $P-1$, so if the smallest x value for which we get $r^x = 1$ $(mod\, P)$ is equal to $P-1$. If the order of a is lower than $P-1$, it has to be a number that divides $P-1$: hence a fast way to check if any integer a is the primitive root r we are looking for is to factorize $P-1$ and check the condition:

$$a^{\frac{P-1}{f}} \neq 1\, (mod\, P) \tag{4.16}$$

If this condition is satisfied for any factor f of $P-1$, the integer a is the primitive root we were looking for. So we span r between 2 and $P-1$ and we perform the above modular exponentiation only for the factors of $P-1$.
A final consideration has to be made on the inverse NTT $(INTT)$, whose general expression is:

$$x(i) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)\alpha^{-ik}\, (mod\, P) \tag{4.17}$$

In modular arithmetic a fractional value is not allowed, but an inverse value can be defined: the inverse of a number a is the number b such that, considering a modulus m, the condition $a \cdot b = 1\, (mod\, m)$ is met. The inverse of a exists if and only if a and m are relatively prime: in our case the inverse value if N always exists since N and the modulus P are relatively prime by definition (N is a power of 2, P is a prime number greater than N). Moreover, since $\alpha^{-ik} = (\frac{1}{\alpha})^{ik}$ we compute also the inverse value of α, whose existence is again always guaranteed since the modulus P is prime.

Hence up to now we have decided to use the NTT for the convolution process and to use it padding the vectors involved so that any algorithm requiring a number of points equal to a power of 2 can be used.

An improvement on the choice of *P*

So far we have seen two conditions on the P value, but additional considerations can be made.
The NTT has been presented with a prime modulus for simplicity reasons, but P doesn't have to be necessarily prime: it can be any number such that we can define an *nth* root of unity for it. If we are able to find this number and we are also able to define the inverse values of α and N, we can pick a value of P much lower than $kN+1$. If P is much lower, we are able to reduce the number of bits required in the architecture.

Although a primitive root of unity can always be defined for a prime number, this is not true in general; in 1801 Gauss stated that we can define a primitive root of P only if one of the following conditions is met:

- $P = 1, 2, 4$;
- $P = a^k$;
- $P = 2 \cdot a^k$;

Where a is an odd prime number and k can be any positive integer number. Thanks to this in some cases, especially for high numbers, we can find a valid P value much lower than before.

For example, if we have $N = 512$ and we use binary vectors, with this new rule the minimum P value is 521, while with the previous one we would have obtained 7681, which is a much higher value. Even if all these P values are possible, we will exclude the even values because an odd P is needed for efficient modular multiplications, as explained in the next chapter.

Regarding the convolution constraint on the P value ($P > M^2N$), it has been presented assuming that each vector involved in the convolution has N components and each of them can assume a maximum value equal to M. Since we will handle vectors that can be also sparse, in theory a lower constraint on P can be set: for example we can take the mean value of the vector, but the problem in this case is that we have to know it in advance in order to pre-compute all the parameters we need for the NTT (and this is not possible) and the result can be wrong in some cases since a lower P value can lead to an overflow condition. Hence we will keep this constraint even if it could lead to an expensive solution.

4.1.1 The choice of the algorithm

We always perform a vector by circulant operation relying on the convolution theorem, but in the choice of the algorithm we have to consider several things:

- The circulant matrix is always binary (so the vector that represents it is always binary), while the vector can be either binary or integer;
- The result of the product can be either binary or integer;
- We may have to handle sparse vectors (only one vector or both vectors can be sparse).

Based on these considerations it is hard to design a general algorithm suitable for every condition: hence, several specific algorithms will be presented, in order to choose the one that optimizes speed, area and power for the specific needs.
Unfortunately, many considerations can be made to produce the best NTT algorithm for each specific case, but not a lot can be done on the $INTT$ algorithm: in fact, the input vector of the $INTT$ is in general a dense integer vector, independently on the input vectors the convolution has.

This requires a general *INTT* algorithm (and then architecture), without taking advantage of any characteristic of the input vectors.

General considerations

Since the *NTT* is just a particular case of *DFT*, any fast algorithm known to efficiently compute a fast Fourier transform can be used. In this section several algorithms will be presented and discussed, starting from the classic ones till to specific algorithms designed for the input data we can have (sparse and/or binary).
Moreover, several algorithms require to take the input in a bit-reversal order or produce the output in this way: we will briefly discuss which is the best solution with the input data we treat, since the bit-reversal operation could be avoided but if we use it we can have several advantages in terms of hardware resources and time requested for the computation.

Cooley - Tukey algorithm

The first algorithm we consider is the famous Cooley - Tukey algorithm, whose principle is shown in the figure 4.1 for a case of 8 points:

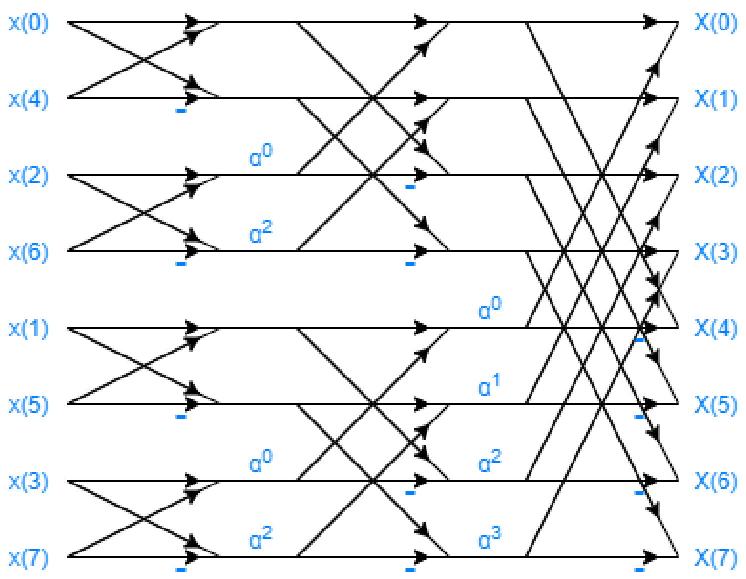


Figure 4.1: Cooley - Tukey NTT

The algorithm is the same of the *DFT*, but in this case we work with modulo arithmetic instead of using complex numbers: this leads to an important advantage, since we use integer units instead of floating point units and there are no precision errors.
As we can see from the figure 4.1, the inputs we need are not taken in sequence but in a bit-reversal order, while the output is produced in order.

In the figure 4.1 a radix-2 algorithm is presented: since we have to deal with large vectors an higher radix could be used. With a radix-4 algorithm we can halve the number of stages, but two problem arise: it adds a lot of complexity (the "butterfly" is much more complex since it requires more additions, subtractions and multiplications) and it is valid only if the number of points of the NTT is a power of 4 we will use only the radix-2 algorithm.

Sparse Cooley - Tukey algorithm

This algorithm is a specific version derived the Cooley - Tukey one and is suitable for a sparse vector. When we handle a sparse vector, the sparsity is present for a limited number of stages while the last stages and the output vector will be dense. So the algorithm is divided in two parts: considering an NTT of 8 or more points, the first part is shown in figure 4.2, while the last stages are done as in the classic Cooley - Tukey algorithm shown in figure 4.1.

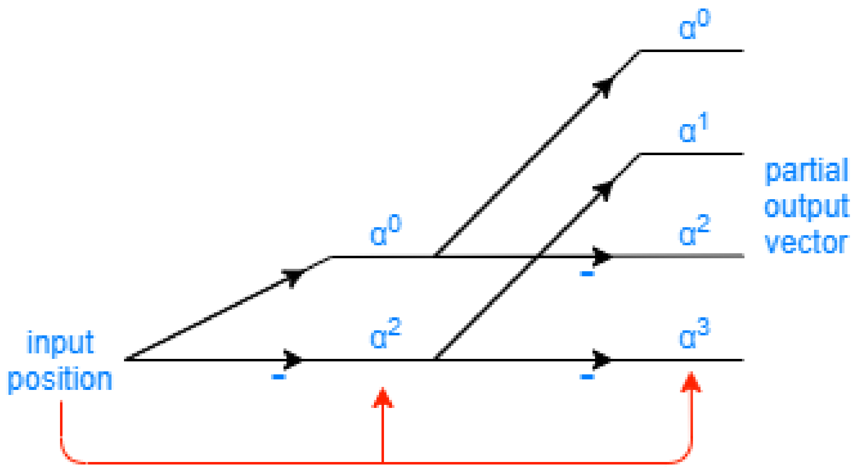


Figure 4.2: Sparse Cooley - Tukey NTT

Considering a sparse vector for which we have only stored the positions of the element different from zero, this sparse algorithm takes as input all of these positions and for each of them computes the first stages of the NTT leading to several advantages:

- We can save all the addition operations;
- We can simplify the subtraction operations;
- We can save some of the multiplications.

The sparse algorithm is used until the data start to begin too much dense: for the last stages of the NTT all the operations are performed: in fact we want to skip the highest number of operations we can, but if we consider a too much big block we will repeat some of the operations, especially the multiplications, wasting the advantages given by this approach. Hence in general the number of stages for the sparse part (2 in figure 4.2) is based on how much the data are sparse (so how many elements different from zero we have in the input vector): for the architecture this will be a parameter so it will be suitable for a vector of any density.

For the sparse algorithm, the reason why each subtraction can be simplified is because for each modular subtraction a final conditional addition is required in order to re-write the result in the correct range $[0, P-1]$: in this case we can avoid this operation since the result of the subtractions is always in the correct range (only subtractions $P-x$ are performed at this stage, while the trivial subtractions $x-0$ are skipped).
Speaking about the multiplications instead, for each position value the allocation of the multipliers (highlighted with the red arrows in red in figure 4.2) is done only based on whether they are needed or not in order to skip all the useless computations. To do so we just need to look at each bit value of the position of the '1': for each bit we have a stage of the NTT for which we allocate the multipliers if the bit value is '1', otherwise we will skip the computation saving clock cycles and not wasting dynamic power.

Reversed Cooley - Tukey algorithm

From [8] we can derive another fast algorithm, whose working principle is shown in figure 4.3.

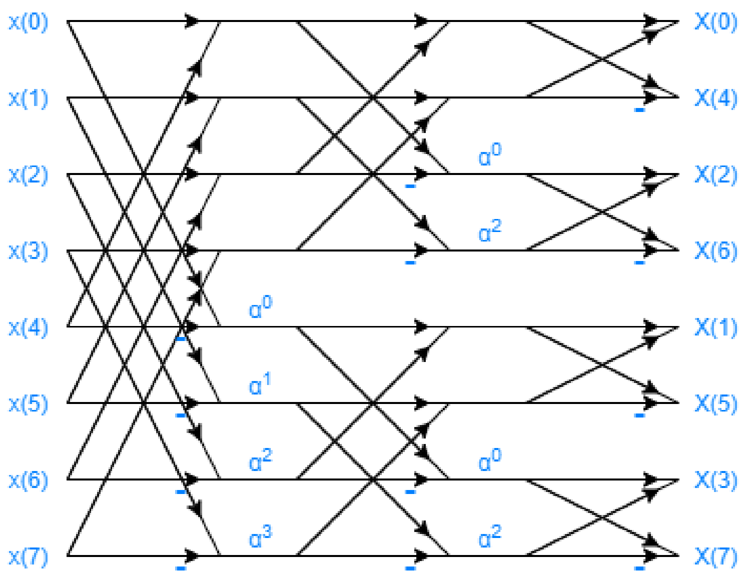


Figure 4.3: Fast NTT

Essentially this is the decimation in time approach, in which the input is taken in order and the output is produced in bit-reversal order; it is equivalent to the Cooley - Tukey one in terms of complexity and number of stages.

Binary versions of the previous algorithms

In some cases we have to handle binary data: the main problem is that all the operations has to be done *modulo P*, so we cannot simply keep the LSB of every operation because this will lead to wrong results. As an example, we can consider a simple case with $P = 5$: if the final result we get is 6, performing the modulo operation the correct result is 1 while if we keep the LSB we get a 0, while if the final result is 11 the LSB corresponds to the value we need.
Nevertheless there are two small advantages in handling binary data:

- If we have binary data as input, for the first layer we don't need adders and multipliers but only XOR and AND gates; however this is not valid for all the following layers: in fact if we multiply 1 by α the result is α and from the following computation we need integers adders and multipliers;
- If also the result has to be binary after the last modulo operation we need an extra *modulo 2* operation, but this one can be simplified: since each output value has to be equal to 0 or 1, we just need to keep the LSB of the final result (so we can avoid the computation of all the other bits, saving again clock cycles and dynamic power).

A direct sparse algorithm and its binary version

If the data are very sparse we can directly compute each output value without exploiting one of the previous fast algorithm: this algorithm can have good performances, but we always have to compute one output at a time unless we do an expensive memory replication or we accept the presence of a collapse unit as the one described in chapter 3, that is exactly what we want to avoid in our architecture. Moreover we need to design a solution valid for LEDAcrypt: but this proposed algorithm is too much dependent on the parameter of LEDAcrypt d_v; instead we want an architecture that maintains good performances even if we change a little this parameter. Moreover, we also want to propose a generic cyclic multiplication architecture, that can be valid not only for LEDAcrypt but for other algorithms, in which the parameter d_v can be much different since it can influence the security level reached by the algorithm.
A binary version of this algorithm can significantly reduce the number of multiplications required: however multipliers are still needed at least for the NTT of the integer vector, the element-wise multiplication and the $INTT$, so this would be a great advantage only if the goal is to compute the NTT of a sparse binary vector only, not if we need to compute a convolution.
For all these reasons we will avoid to use this solution.

A brief comparison between the algorithms

In general we have algorithms in which the input is taken in a bit-reversal order or the output is produced in this way. The bit-reversal can be an heavy computation because it can be done in two ways:

- We can simply reverse the index, so reading it from the LSB to the MSB: this procedure has no cost, but in general each memory is designed in such a way that reading or writing consecutive locations is much faster than accessing always random locations;
- We can rearrange the memory so that we can access consecutive locations: this procedure makes the computation faster, but the pre-processing stage can require too much time and also additional hardware resources, limiting the area for the other operators we need.

Anyway the bit-reversal procedure is not always needed: if we process ordered inputs we get bit-reversal outputs, and then we can use an algorithm that takes bit-reversal input to produce an ordered output.
Unfortunately for our choice we have also to consider the input data we have to deal with: we always have sparse matrices and sometimes also sparse vectors. The *NTT* can be efficiently designed for sparse input data only if data are considered as in a bit-reversal order, hence the bit-reversal procedure is required in order to exploit all the advantages the sparse data give us.
In the next chapter we will detail an architecture capable of doing the bit-reversal procedure in parallel with other operations, in order to not require additional time for the completion of the convolution process.

Based on all the considerations above we are now able to decide which algorithms we need for the vector by circulant architecture and for the sparse vector by circulant one. In fact we will use two different architectures in order to fully exploit the advantages of the two solutions. For the vector by circulant architecture we will use:

- The sparse Cooley - Tukey algorithm;
- The Cooley - Tukey algorithm;
- The other fast algorithm.

Instead for the sparse vector by circulant architecture we will use twice the sparse Cooley - Tukey algorithm avoiding the use of the other fast algorithm.
These two or three algorithms are combined together with the additional bit-reversal procedure in two possible structures, based on whether both the vectors are sparse or not; moreover their slightly simplified versions are used in case of binary data. The detailed structures are presented in the next chapter.

Chapter 5

Multiplier architecture

In this chapter two architectures are presented, one considering the vector by circulant operation and another one for the sparse vector by circulant operation. But before showing them, we need add some details on how the operations are made.

5.1 Modular operations

First of all we have to consider that every addition, subtraction and multiplication has to be done in modular arithmetic. Additional hardware is required with respect to normal inter units, and in the following each operation is described along with its cost.

5.1.1 Modular additions

For the additions, we know we are doing a sum between two numbers that are *mod* P: if the two addends are close to P the result could be higher than P, but it will never be higher than $2(P-1)$. Hence, we just need a conditional subtraction to write the result in the correct form, as shown in algorithm 1 and in figure 5.1.

```
Data: $a, b$: addends
Result: $c$: sum
begin
    $x = a + b$;
    if $x > P$ then
        $c = x - P$;
    else
        $c = x$;
    end
end
```

Algorithm 1: Modular addition

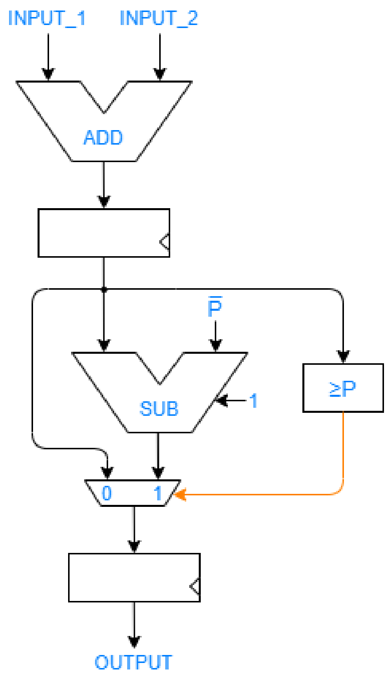


Figure 5.1: Modular adder

For the subtraction the considerations are very similar: the result will be either correct or negative, but no more than $-P+1$. Hence also in this case we just need to perform a conditional operation: if the result is negative we just need to add P, as shown in algorithm 2 and in figure 5.2.

```
Data: $a, b$: minuend and subtrahend
Result: $c$: subtraction
begin
    $x = a - b$;
    if $x < 0$ then
        $c = x + P$;
    else
        $c = x$;
    end
end
```

Algorithm 2: Modular subtraction

For both the cases, the conditional adder or subtractor has a fixed input, so the design by contraction technique can be used to limit as much as possible the required hardware.
In some cases we need to perform subtractions for which we know in advance that the result will be correct, since we always subtract a number between 0 and $P-1$ from P. In this case we would like to skip the conditional addition, so we take the result just after the subtractor, ensuring a reduction in the dynamic power consumption.

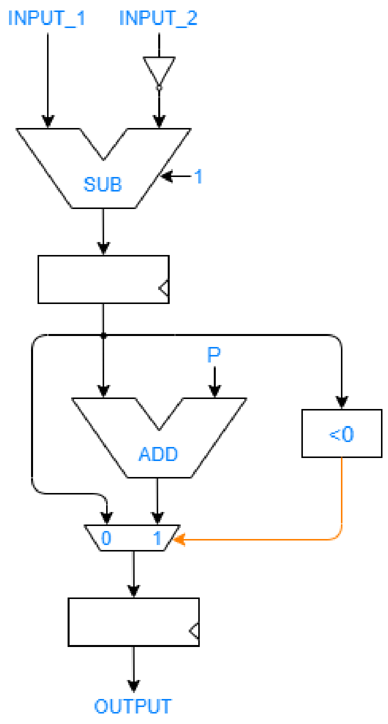


Figure 5.2: Modular subtractor

Since all of these three cases can share some hardware components we design a unique component capable of doing all of these three operations. The circuit is shown in figure 5.3 and it has two control signals:

- Normal_sub: if it is equal to 1 we perform a normal subtraction and in one clock cycle we have the result;
- ADD_sub_n: if the previous signal is equal to zero we make a modular addition or a modular subtraction based on this signal value and we have the result in two clock cycles.

5.1.2 Modular multiplications

For the modular multiplication we have a much greater problem, but let's consider an example to better explain what we need to do. Supposing we want to multiply 7 and 15 modulo 17: in a straightforward implementation we directly perform the multiplication and we obtain 105; now we need to divide this number by 17 and keep the reminder of the division, equal to 3: this is our result. As we can see, for every multiplication we need to perform a division, and this requires a lot of extra hardware that will hugely affect speed, power and area of the architecture.
However there is a way to perform fast modular multiplications: the Montgomery multiplication. As explained in [9], this method avoids the need of performing complex divisions: in fact we will perform divisions and multiplication by R, a constant that has to be greater than P and co-prime with it. Since we don't

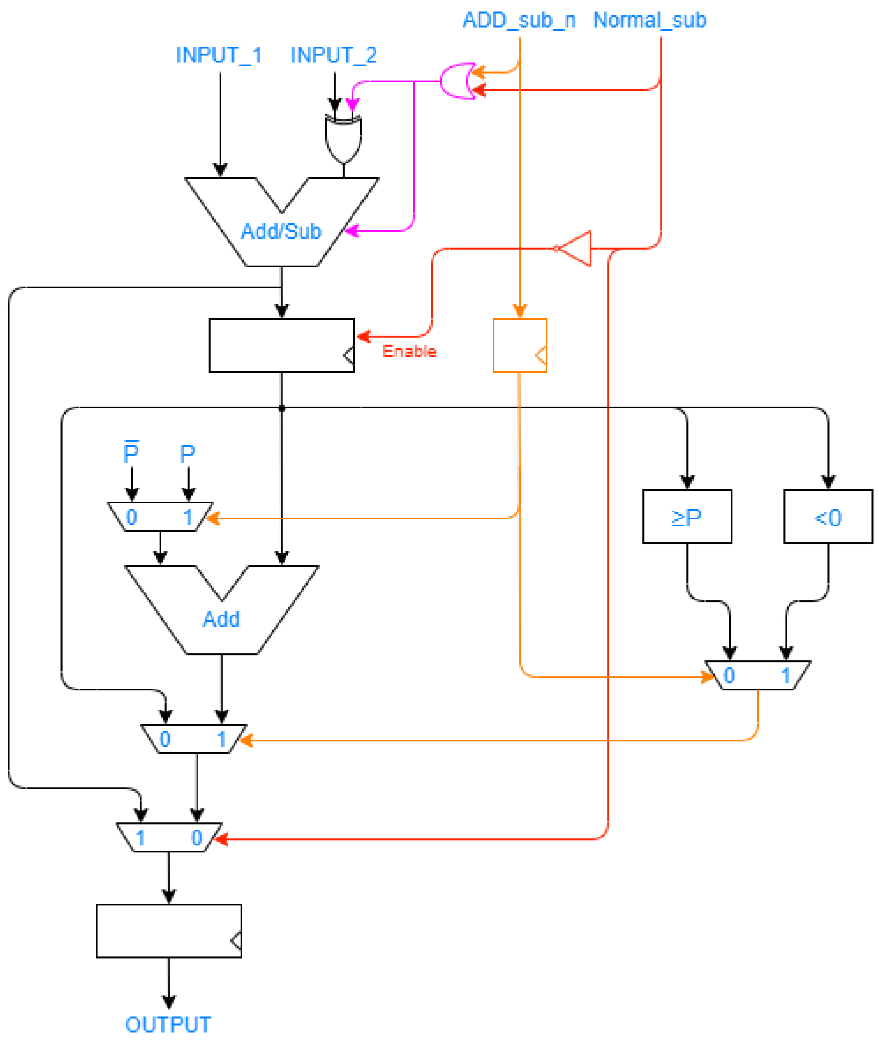


Figure 5.3: Modular adder/subtractor

have other constraints we choose R as the minimum power of 2 satisfying these conditions: now every division is just bit shifting and every multiplication is just bit masking, so both operations are trivial and have no cost.
However, we need a pre-processing computation, since the algorithm requires that the number are expressed in the Montgomery form; after the multiplication we can go back to the normal form with a post-processing computation.
The additional pre-processing and post-processing computations makes the Montgomery method slower for a single multiplication, but if we have a chain of multiplications these computations become negligible: that's way it is often used in cryptosystems like *RSA*, since there we need to perform several modular exponentiations. In our case we alternate additions and multiplications, but since the addition of two numbers in Montgomery form gives a number which is still in the Montgomery form we can simply convert each input of the *NTT* into the Montgomery form and then we perform the out conversion after the *INTT*.

In algorithm 3, as well as in [10] the pseudo code of the Montgomery multiplication is shown.

```
Data: $a, b$: factors
Result: $c$: product
begin
    $x = a \cdot b$;
    $tmp = mod\,(x, R)$;
    $s = mod\,(tmp \cdot k, R)$;
    $t = x + s \cdot P$;
    $u = t/R$;
    if $u < P$ then
        $c = u$;
    else
        $c = u - P$;
    end
end
```

Algorithm 3: Montgomery algorithm

In figure 5.4 the HW scheme for the Montgomery multiplication is shown: we simply do the multiplication between the two factors (already written in the Montgomery form) and then we perform the reduction function: this consists in several simplified operations in order to put the result (in Montgomery form) in the correct range $[0, P-1]$. This block also asserts a signal when the output of the multiplier is ready: this signal is useful since when we synthesize this block we may need to add or remove some registers reducing or increasing the number of clock cycles required to produce a result, and with this signal the control unit will not be affected at all.

Moreover this hardware unit is capable of other three different tasks:

- In some cases we don't need all the multipliers to be active at the same time, hence when needed we simply skip the operation with a delay line equal to the one of the multiplier: n this way we can save dynamic power, but most importantly we can simplify the write-back operation when only a reduced number of multipliers have to be used;

- For the Montgomery conversion this block can be used: one input is the number to be converted, that has to be multiplied by $R^2\ mod\ P$ (a value that can be stored in memory);

- For the Montgomery out conversion we just need to use the reduction function: as shown in figure, we simply skip the integer multiplication with the multiplexer.

Hence there are three control signal, and the one asserted defines the behaviour of this unit.

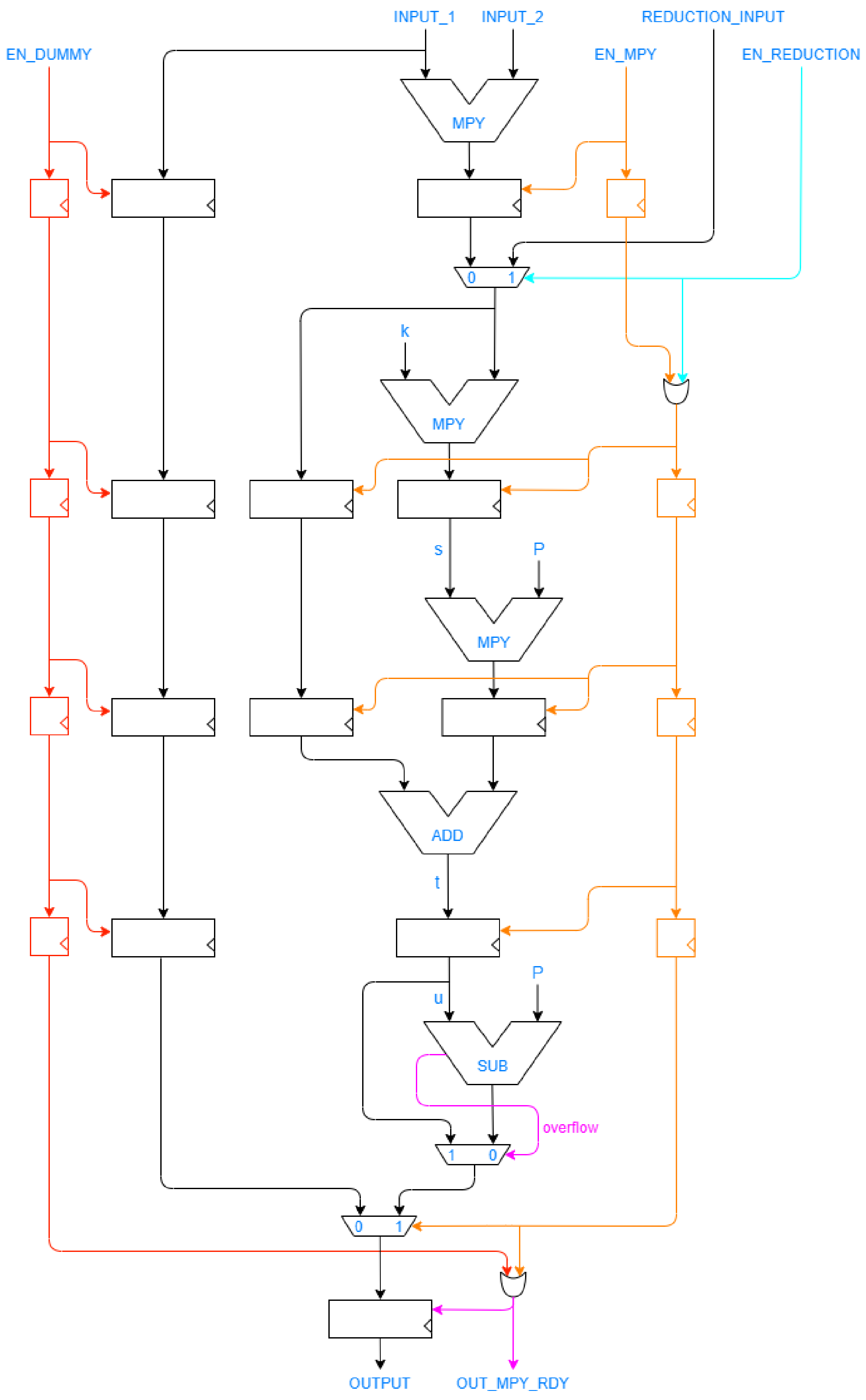


Figure 5.4: Modular multiplier

5.2 The top-level architecture

We are now able to show the architectures of the two cases, based on whether only the matrix or also the vector are sparse. For each case we have two sub-cases: in fact the matrix is always binary, while the vector can be either binary or integer; moreover the output vector can be either binary or integer.

5.2.1 Vector by circulant architecture

The general architecture is shown in figure 5.5.

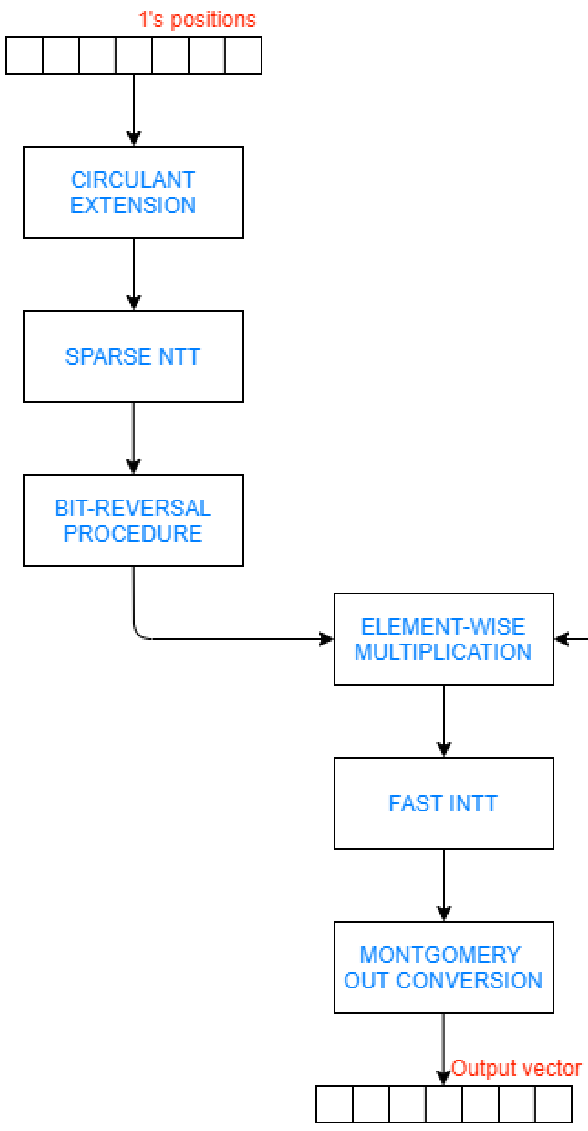


Figure 5.5: Vector by circulant architecture

The circulant extension block is responsible of producing the additional positions as previously described in the work.
Once its task is complete we can compute the sparse NTT with the suited algorithm, and then we apply the bit-reversal procedure on the output vector.

Meanwhile we perform the Montgomery conversion for the input vector and then the fast normal NTT; once we have done the two NTT we are ready for the final steps which are element-wise multiplication, $INTT$ and Montgomery out conversion.

5.2.2 Sparse vector by circulant architecture

The general architecture is shown in figure 5.6.

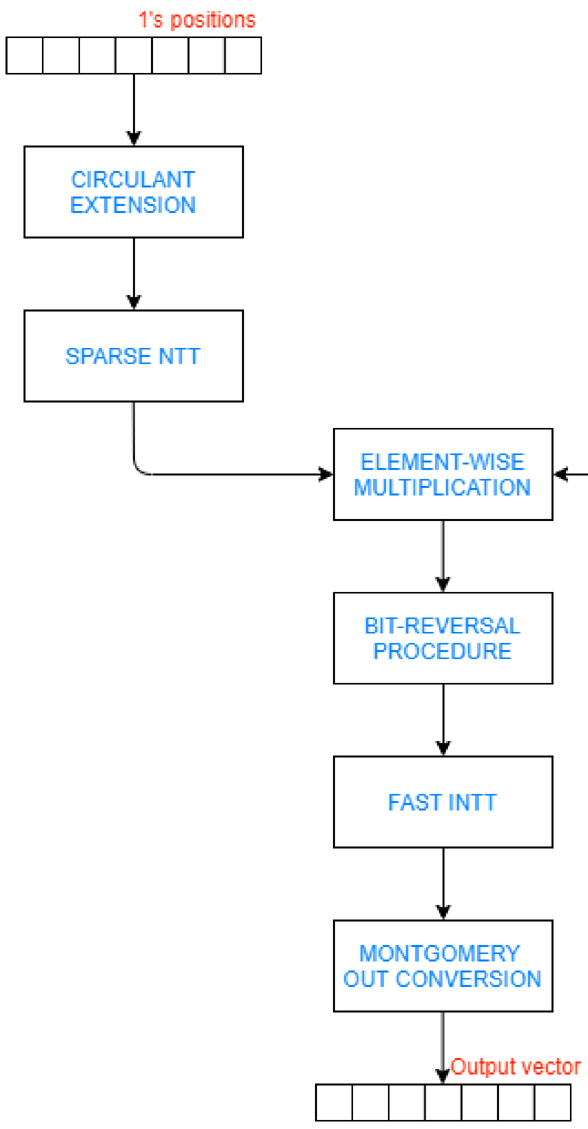


Figure 5.6: Sparse vector by circulant architecture

There are very small changes with respect to the previous architecture:

- We need to use twice the sparse NTT algorithm: the structure is slightly modified in order to handle an integer vector if needed;
- The bit-reversal procedure is performed after the element-wise multiplication, but also in this case it has to be done only once.

5.3 Detailed architecture

From now on we will consider only the binary vector by circulant case whose result is again a binary vector. Similar considerations can be done for all the other architectures and will be briefly described where needed.

5.3.1 Resources sharing

As we have seen, each modular adder, subtractor and especially multiplier requires a lot of hardware: since all these blocks are used in both the NTT and in the $INTT$, and the multipliers are also used for the element-wise multiplications, we will reuse these blocks as much as possible. In this way we slightly complicate the control unit and we need some additional multiplexers, but we can save a lot of area achieving the same speed.

5.3.2 Resources allocation

First of all we have to define how many hardware resources we can use: we assume that only the multipliers deeply impact on the area. Data memories have an impact but they cannot be reduced, while the number of multipliers is a design choice. Once we have chosen the number of multipliers we define all the rest of the architecture, like the number of adders/subtractors and we also set the number of blocks of the data and constant memories.
In order to choose the right number of multipliers we have to consider several things:

- If we double the number of multipliers we roughly halve the number of clock cycles required to produce the result;
- The area required for a multiplier is deeply affected by the number of bits of the inputs, and these value depends on the number of points of the NTT: hence we can have more multipliers if the size of the vectors involved in the product are small;
- Having too many multipliers leads to another side effect: since we need to read several constants from the memory and they are not consecutive values, we need memory replication or interleaved memories with several multiplexers.

First of all we have tried to map a Montgomery multiplier on a Cyclone IV GX family FPGA in order to have an idea of the logic required for it: with $N = 2^{15}$ (the minimum value for the LEDAcrypt algorithm) we need the 3% of its combinational functions. We consider multipliers as the limiting factors, but since we have also adders/subtractors, multiplexers and several data and constant memories, we will design our architecture with 8 multipliers in order to have a good trade-off between speed and area.
With 8 multiplexers we will use 8 adders/subtractors and the data memory parallelism will be equal to 8. In general we don't have a lower parallelism, but if so we will simply work with interleaved data memories.

5.3.3 The handshake protocol

Since the architecture is complex, every block is designed as a stand alone unit and only once it is verified and validated it is put together with the other ones. Every block requires a certain number of clock cycles to produce its result, but this number can be changed if needed (e.g. a critical path is too long and registers have to be inserted in order to increase the achievable clock frequency). Hence, in order to easily put together all the blocks, the handshake protocol will be widely used in the architecture: every unit asserted a DATA_RDY signal when it has finished, so that the higher level control unit can rely on this signal and can be designed independently on how many clock cycles each unit requires. Moreover this approach makes it possible the opportunity of a later optimization of every block without affecting all the other ones.

5.3.4 Control unit organization

The control unit is split in several parts and organized in a hierarchical way, in order to simplify its validation. There are three levels of control units:

- The lowest level involves the basic units as the constant memory management, the circulant extension and the bit reversal unit: each control unit receives a start signal from an upper level control unit and produced the required outputs;
- The second level involves all the computations as the ones of the $NTTs$: these control units manage units as the constant memory management one in order to produce the proper result;
- The highest level is the master control unit: it is responsible of the whole convolution operation activating the blocks and the control units in the correct sequence.

With this approach each control unit will be much more simple and easy to be tested, however some additional hardware resources are required: in fact some units will be used by more than one control unit (e.g. the one described in 5.3.5, that is used by both the NTT, each of them with its own control unit, or all the adders and multipliers), hence additional multiplexers in front of these units are needed and the master control unit gives access to the proper control unit when needed.

5.3.5 Constant memory management

Almost every multiplication involves the radix of the NTT, α, raised to a certain power (from 0 to $N/2 - 1$): since all these values are known in advance we will store all of them in one or more ROM memories, so every time we need them we simply read them instead of always recomputing them. This will lead to the presence of an extra memory, but we will save a lot of dynamic power and number of clock cycles to complete the algorithm.

However, once we have chosen the number of multipliers several considerations have to be made in order to properly design the constant data memories:

- In general we don't need consecutive values, so if we have a memory with 2 read ports we will require 4 clock cycles to read the constants needed for all the multipliers;
- Multipliers don't need to work every clock cycle but we can alternate their task with the one of the adders and the subtractors, complicating the control unit: in this way in the worst case (as in the last stage of the Cooley-Tukey algorithm) they work every 3 clock cycles, so we need to design a structure able to produce in no more than 3 clock cycles the constant values for each multiplier. In the second-to-last stage they work every 3 clock cycles as well, but they use twice the same constant values: so we need to feed them with new inputs every 6 clock cycles, but these time they are all even values. However the initial architecture will be designed such that the multipliers can work every clock cycle (an optimization for the whole architecture will be described in the next chapter): in this case constants have to be provided in the fastest possible way and so this block will have two operating modes: if we assert the burst mode signal we don't wait for an acknowledge but we produce the constants values as soon as they are ready.

Based on these considerations we will design the memory structure shown in figure 5.7. Constants are organized in several interleaved memories and we are always able to produce the required constants in the requested time. As we can see, several multiplexer are needed even with only 8 multipliers: with an higher number we would have required much more of them or a certain memory replication leading to a much higher area occupation. Each constant is read and stored in a register file (whose size matches the number of multipliers), then once all the constants are ready the buffer is enabled and the multipliers can start their computation.

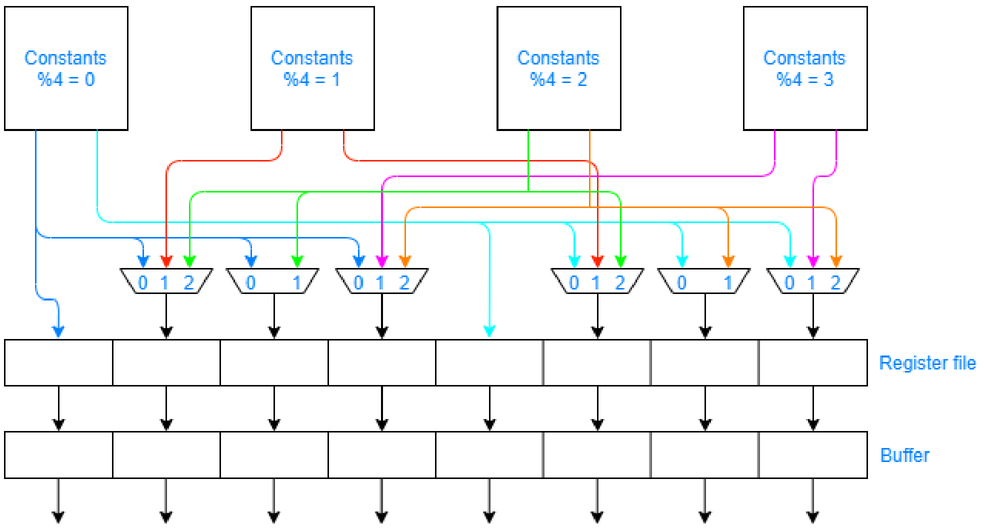


Figure 5.7: Constant data management

This block also produces three output signals:

- CONSTANTS_COMPUTATION_BUSY: it is asserted when this block receives a START signal and starts its task in order to provide the constants to the multipliers;
- CONSTANTS_RDY: it is asserted when the 8 constants (or less) for the multipliers are all in the buffer structure. Every *NTT* algorithm checks this signal and proceeds only if all the constants are ready. Each *NTT* algorithm asserts also the CONSTANT_ACK signal according to the handshake protocol when the constants are no more needed, so that this block can provide the next 8 constants (for all the stages in which more than 8 constants are needed). We may lose a clock cycle in some cases, but the handshake procedure allows us to design the constant generation architecture and the *NTT* one as completely independent blocks. So if at a later time we need to change one of them affecting the latency we will not need to update the control unit.
- CONSTANTS_STEP_ENDED: it is asserted when all the constants required by one step have been produced. Only after the reception of a CONSTANT_ACK after this signal this block can return to the IDLE state, ready to provide constants for a new *NTT* stage.

For each stage of the *NTT* we need to read these constant values with a certain offset: hence this unit receives only an input value ranging from 1 (for the stage in which all the constants are required) to $\log_2 \frac{N}{2}$ (for the stage in which we need only α^0 and $\alpha^{\frac{N}{2}}$) and produces all the requested constants.
In order to read the proper constants we need a dedicated address generation: for the ROM with the constants with exponent $\%4 = 0$ the component is in figure 5.8, in which OFFSET is the decoded version of the input value and STEP is one fourth of this value. For the other 3 memories instead a simple counter can be used, since when we need the values stored in them we need to take all of them.

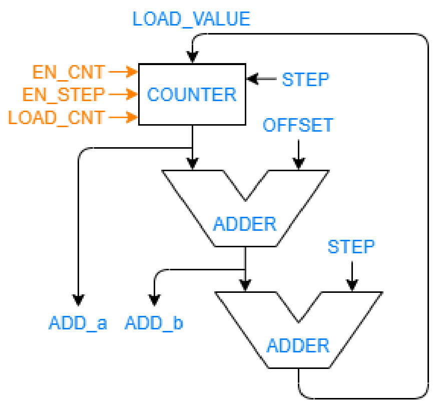


Figure 5.8: Address generation first ROM

5.3.6 Circulant extension

The first block we need is the circulant extension one: we have a memory in which the 1's positions are stored and we have to produce the additional values; moreover we use this block to feed the Sparse NTT component with the 1's positions when they are needed. The architecture is shown in figure 5.9.

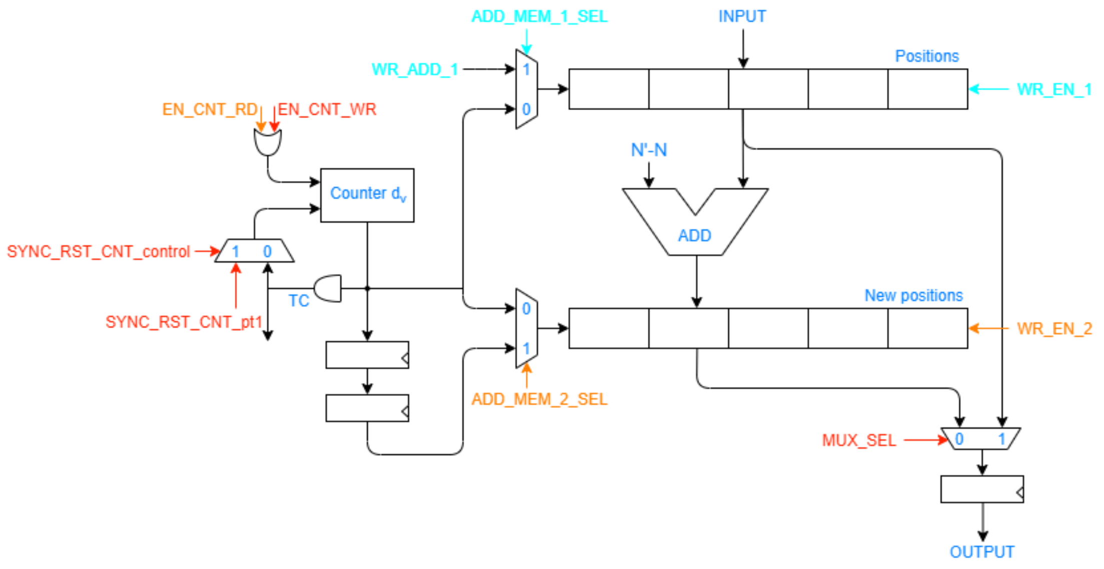


Figure 5.9: Circulant extension

The starting positions are stored in the Positions memory and we just use an adder to produce the additional 1's positions, stored in the New Positions memory. The counter is used to produce both the read and the write address: we span over the first memory to produce all the values needed. As we can see, there are several control signals:

- The ones highlighted in light blue are used from the external to write the positions values into the proper memory;
- The ones highlighted in orange are managed by a control unit responsible of computing and storing the new values in the new memory;
- The ones highlighted in red are managed by a control unit responsible of giving the output values to the sparse NTT algorithm one at a time.

Since this block uses a very small number of control signals and no future optimizations or modifications are necessary, this block doesn't rely on a fully handshake protocol.

5.3.7 Sparse NTT architecture

We read one position value at a time from the Circulant extension block and we compute the NTT with the sparse algorithm described in 4.1.1 for a limited number of stages (e.g. with a complete NTT of 2^{15} points we compute 9 stages in this way). The idea is that the sparsity of the input vector can be exploited for a limited number of stages: in fact even if the input vector is sparse, the result of the NTT will be dense. Once we have computed the partial result for a position, we store this intermediate output value in the final memory of the complete NTT and we proceed with the following position; after the computation of all the positions we proceed with the last stages of the NTT with a dense algorithm. The algorithm we use is the Cooley-Tukey one: the starting position is considered as a bit reversal value, so the output vector of this NTT will be ordered.
The architecture is the one sketched in figure 5.10, in which for simplicity reasons several multiplexers in front of the shared resources are not shown.

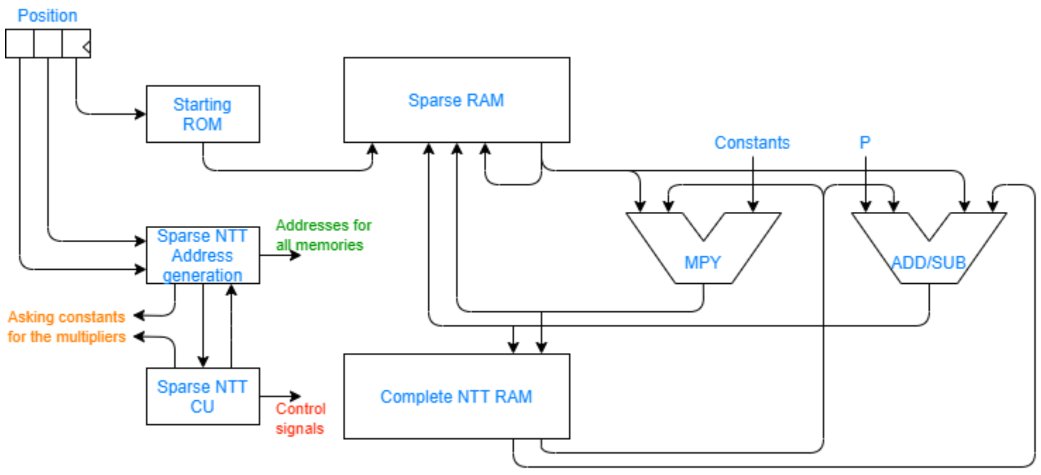


Figure 5.10: Sparse NTT architecture

There are three different steps:

- In order to skip some clock cycles at the start and to simplify the control unit, since the data memory parallelism (as well as the number of multipliers and adders) is equal to 8 the first three stages of the NTT are pre-computed with MATLAB and stored in a ROM memory. Hence the 3 LSB of the position register are used to directly address this memory and read the proper memory location;

- After that, according to the sparse NTT algorithm we compute the first stages of the NTT: therefore for every bit of the position register involved in the sparse algorithm (the bits in the middle of the register) we execute multiplications followed by simplified subtractions if the position bit is equal to 1 or the simple extension procedure (which is nothing but trivial additions by 0, performed without involving the adder units in order not to waste dynamic power) if the position bit is equal to 0;

- The partial result for each position value is stored in the Complete NTT RAM memory (the address in which we store this output value is given by the remaining MSB of the position value), then we proceed with the dense algorithm in order to complete the computation of the NTT.

The computation for the sparse output is handled by the Sparse NTT CU, which is also in charge of asking the right constant values for the multiplications. The dense stage are instead performed thanks to another control unit, better described in 5.3.10 along with the complete dense algorithm.

5.3.8 Bit-reversal architecture

Since the first *NTT* produces an ordered output and the second one a bit-reversed output, we have to bit-reverse one of the two vectors before going forward. The bit-reverse procedure is performed on the output of the sparse *NTT* algorithm in order to get a bit-reversed vector and when we apply the *INTT* we will obtain an ordered output vector: in this way we perform the bit-reversal operation only once throughout the algorithm.
The bit-reversal operation is just a memory reordering procedure: since we don't require neither multipliers nor adders the operation can be performed while we proceed with the *NTT* of the dense vector. Since this operation is performed in parallel with the second NTT this procedure is not part of the total time required to complete the cyclic multiplication, assuming that the bit-reversal procedure is faster with respect to the whole NTT algorithm (hence this is the unique requirement). Therefore we limit as much as possible the area required allocating the minimum quantity of resources for the algorithm.
The architecture is shown in figure 5.11.

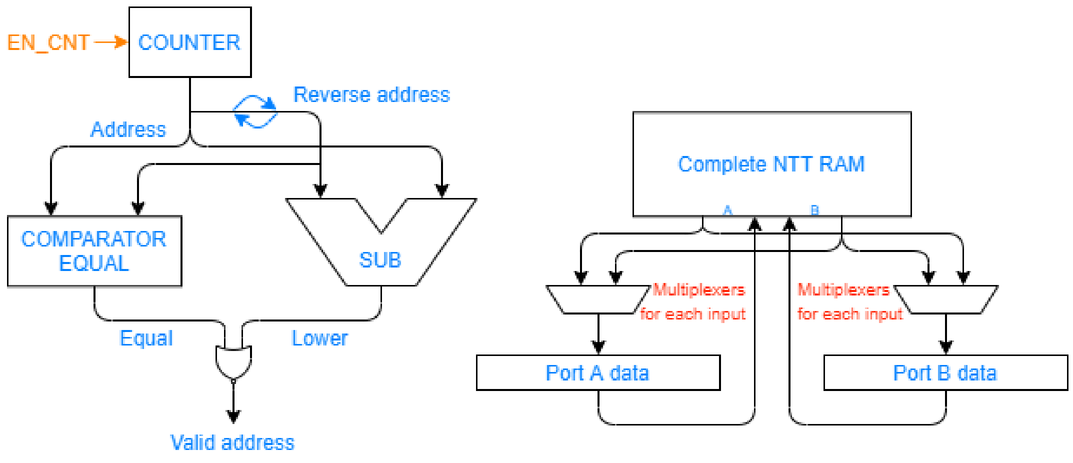


Figure 5.11: Bit reversal architecture

This block works as follows:

- We span over the whole memory and we compare every address with the reversed one: if we need to swap the memory locations (this is not always the case, since in roughly half of the cases the index and the reversed index are equal or the reverse index is lower than the index, meaning we have already performed the swap operation) we assert the VALID_ADDRESS signal otherwise we go to the next address;
- For each valid address we read the memory content from the two ports and contextually we perform the switch operation relying on multiplexers driven by the LSB of the two addresses, then we store back the data into the NTT RAM memory and we proceed with the algorithm going to next address.

For the sparse vector by circulant architecture the bit-reversal procedure is done after the element-wise multiplication and so it is part of the critical path of the whole structure. This fact should suggest a fast implementation of this block: for example an algorithm like the one in [11] can be considered and implemented.

5.3.9 Montgomery conversion

The input vector has to be converted into the Montgomery form before proceeding with the NTT computation. If we have a binary vector the computation is easy, since we just need multiplexers in order to choose between 0 and the Montgomery value of 1 (which can be pre-computed). If instead we have an integer vector, the computation require the use of the multipliers: every input has to be multiplied by a pre-computed constant, $R^2 \, mod \, P$, in order to have the correct Montgomery value.
Since we are considering a binary vector we will use the multiplexers as in figure 5.12: for every portion of the vector (8 value per time since the data parallelism is equal to 8) we choose the proper value and we store it in the second NTT RAM memory. Since this vector has to be zero padded we also store zeros in the remaining part of the memory. This whole operation doesn't require neither the multipliers nor the adders, so it can be performed in parallel with the first NTT.

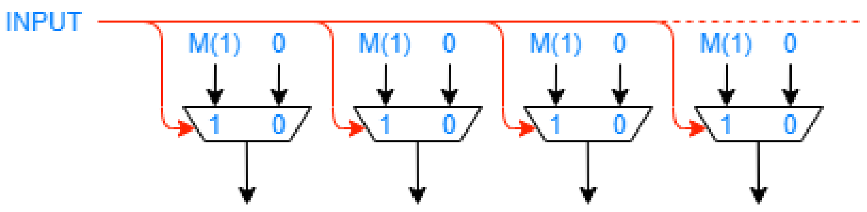


Figure 5.12: Montgomery conversion for a binary vector

5.3.10 Fast NTT architecture

This NTT algorithm is used for a dense vector: since the input is dense every operation has to be done and the goal is to maximize the throughput of the unit. As every NTT algorithm, it involves only the multipliers, the adders/subtractors and a RAM memory. In this case we have an ordered input and we produce a bit reversed output.

Multiplications optimization

First of all we need to consider how a memory works: when we proceed with the algorithm we want to always read new data when we store the results on the same port and independently on the operating mode of each port of a RAM memory (write first, read first or no change) we cannot achieve this goal. In fact when we write into a port the output data is not what we want, hence we need to find different solutions.
For the multiplications we need to read the data from one port only (the other input comes from the Constant memory management unit): therefore in order to maximize the throughput we read the data from the port A and we write the result on the port B, whose address is just the address A delayed by the same latency of the multiplier. In this way we can perform all the multiplication with a throughput tending to 1.

However another problem arises with the multiplications: every time we need to compute them we need to ask the proper constants to the Constant memory management unit which gives them in a few clock cycles. This delay we have to wait is not acceptable since new constant values have to be required many times throughout the algorithm: in order to reduce this overhead the constants are required as soon as the previous ones are not needed anymore and we limit as much as possible the number of times we need to ask the constants, in fact we use the same constants all the times they are needed before asking new ones. As shown in figure 5.13, the sequence of multiplications for the last stages is arranged in order to ask the constants only 2 times instead of 4. At the cost of a more complex address generation unit we can maximize the throughput when we perform the multiplications.

For the stages in which new constant values have to be provided at every clock cycle the Constant memory management unit can work in burst mode, hence the multipliers will work at their maximum throughput.
There are also stages in which a number of constants lower than the number of the multipliers have to be provided: in this case for the unused multipliers we activate the dummy operating mode (using the red path shown in figure 5.4) and in this way we can avoid to introduce additional multiplexers.

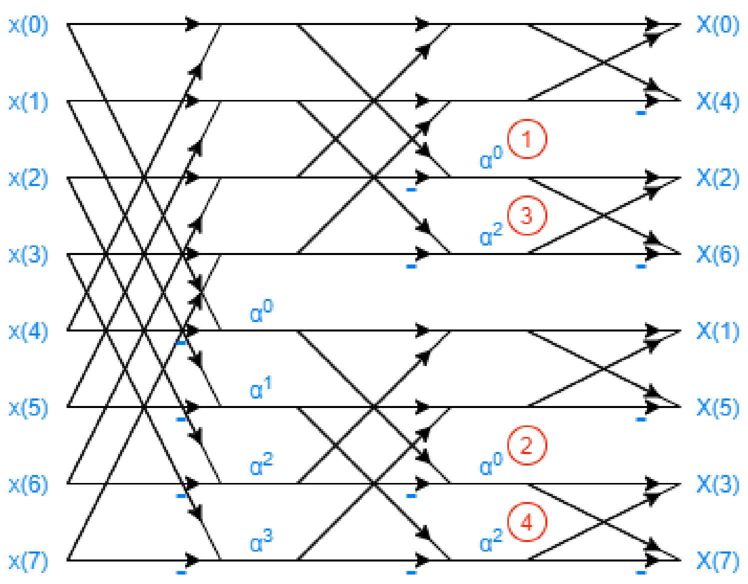


Figure 5.13: Multiplications sequence

Optimizations for the first layer

The input vector is dense, but we know that at least the second half is full of zeros due to the zero padding procedure, so we can avoid to compute all the modular additions and subtractions in the first stage. Since the input is a binary vector we can also skip the multiplications of the first layer: in fact the Montgomery value of 1 is still the identity element of the multiplication, and so instead of wasting dynamic power we can simply read the constants from the ROM memories and store them in the second NTT RAM memory for all the multiplications of the first stage.

5.3.11 Element-wise multiplication

This is a core operation of the convolution procedure that has to be computed after the two *NTTs* have been performed: in our architecture this operation doesn't require additional resources, since we will simply use the multipliers already allocated. We read the input from the outputs of the two *NTT* RAM memories and we store them in one of the two memories. A simple counter of the ones already allocated can be used in order to provide the correct addresses for both the memories and the multiplexers in front of the memories and the multipliers are properly managed by the master control unit.

5.3.12 INTT architecture

In this case we have a dense input vector, so again all the operations have to be computed: we will just skip some modular additions and all the modular subtractions of the last layer, since the final output vector of the convolution procedure is less than the half of the vector that the *INTT* algorithm can produce.

The algorithm we use is the same described in 5.3.10: we just proceed in the opposite direction, since in this case the input is in the bit reversed form and we produce and ordered output. We also need other constant values: we have a unit equal to the one described in 5.3.5 but with different constants that are sent to the multipliers thanks to the master control unit, which also in this case is in charge or properly manage the multiplexers in front of the shared resources.

5.3.13 Montgomery out conversion

For this part of the algorithm we use the reduction function embedded in the multipliers (using the light blue path shown in figure 5.4), so again no extra hardware is required. We span on all the memory and we out convert all the values in it.
Since in our case the output vector is binary, we just need the LSB of the out converted number; if instead the output is an integer vector, we have to keep the whole out converted number.

5.4 Architecture improvements

Once the architecture has been designed and completed, some improvements have been considered in order to speed up the computation or to reduce the required resources.

5.4.1 Insertion of plain modular adders

One of the problems we have with the designed architecture is related to the time required by the adder/subtractor units in order to produce their results: in fact for every operation involving them we need to use both the memory ports to read the input values, wait for the component latency and then store back the results in the same locations.
We cannot use the approach used for the multipliers and reach a throughput of 1 (as explained in 5.3.10), since when we store back the results the memory output cannot be used for the next addition or subtraction independently on the operating mode of each memory port.
Hence with a straightforward implementation we can reach a throughput of only 2/6: but adding these units and changing the latency of the adders/subtractors from 2 to 3 clock cycles we can reach a throughput of 8/10, which is 2.4 times higher with respect to the previous one. This result has been achieved with a very limited addition of hardware resources since modular adders don't require much hardware (they are optimized with respect to the adders/subtractors since they have to perform one operation only), but the reduction in the total latency required by the convolution operation is huge since additions and subtractions are performed throughout the whole algorithm.

5.4.2 Control unit merging

The management of the general NTT algorithm is a block which involves a control unit with many states and a complex address generation module. In particular the address generation module requires a not negligible amount of resources since it involves several counters:

- Stage counter: it is in charge of taking note of the stage of the NTT we are computing;
- Constant counter: it takes note on how many times we need to use the same constant values before asking new ones; the terminal counter is obtained with a comparator since its value is a function of the stage counter output:
- Address counter: it provides the address for both the ports of the memory and for both the operations we need to compute (multipliers and additions/subtractions); it is a slightly modified version of the one used by the Constant management unit.

However the first two $NTTs$ are basically the same algorithm but we just proceed in two opposite directions, as well as for the $INTT$ algorithm in which we perform the same operations involving only different constant values.
In order to minimize the required resources we use a unique address generation module with an important degree of freedom: in fact the stage counter can be loaded with the value we need making this block suitable for the sparse NTT algorithm also, in which only a few dense stages have to be computed. Moreover we can change its counting direction, which gives us the opportunity of using this unit for both the $NTTs$: in fact while the first one takes a bit-reversed input, the second one considered the input as an ordered vector.

5.4.3 Reduction function optimization

In the architecture we have used the reduction function block embedded in the modular multiplier: this gives us the possibility of not using additional resources but increases significantly the latency of the whole algorithm, since in order to produce the final result we need a new read cycle on the whole memory.
In order to speedup the computation the reduction function will be a stand-alone component and it is fed with the output data of the multipliers once they have computed the multiplication by N^{-1}.
With minor modifications in the control unit we have strongly optimized the performances: moreover since we just need the LSB of the out converted number the design by contraction can be fully exploited since this new reduction function block is intended to produce a binary output only. This means that the addition of hardware resources is minimum and almost counterbalanced by the removal of the reduction function option in the multiplier architecture.

5.5 Architecture flexibility and scalability

The proposed architecture is scalable in various terms:

- We can change the parameter d_v: this change will impact only on the number of clock cycles required to produce the result, since there will be more or less 1's positions for the sparse NTT algorithm;

- We can change the number of sparse stages of the sparse NTT: the optimum value depends on both d_v and N and this modification affects only the dimension of the RAM memory for the sparse NTT while the number of clock cycles required remains almost unchanged if we choose the optimum value, but neither minor nor major changes have to be done for the architecture. This degree of freedom offers the maximum flexibility for every input vector we need to handle, making this solution suitable for input vector of any sparsity.

We can also change the number of multipliers we need, contextually changing the number of adders and the memory parallelism, in order to strongly reduce the number of clock cycles or the area depending on what we need. However some considerations have to be made:

- The constant memory management unit has to be reconfigured since it has been designed and optimized considering the need to produce up to 8 constant values per clock cycle. The process requires some considerations on the available resources in order to optimize this unit: in fact there are three possibilities which are memory replication (leading to the need of additional ROM memories), a growth in terms of complexity (since we need multiplexers with several additional inputs) or a combination between the previous two solutions: the best option depends on the specific case, based on which additional hardware resources we can use;

- The starting ROM memory with the first three pre-computed stages of the sparse NTT is totally scalable: however the memory dimension scales quadratically with the number of multipliers, hence even if it is feasible a consideration can be made here, keeping the ROM memory as it is adding only a few multiplexers (this will require a few additional states for the sparse NTT algorithm, but since this is an addition and not a modification of the algorithm the process is straightforward and rapid).

Nevertheless these modifications affect only minor components and thanks to the handshake protocol widely used in the architecture with an easy adaptation the proposed design is totally scalable.

Chapter 6

Architecture Synthesis and Simulation

In order to efficiently simulate and validate the architecture, a complete fully parametric MATLAB model of the architecture has been realized: in this way we can compare the results of the designed architecture with the ones of this reference model.
The architecture instead has been designed with a top-down approach: each block has been divided into smaller and simpler ones and then, once all of them has been tested and validated, they have been put together.
Moreover the architecture has been designed as parametric as possible: this has given us the chance to validate the architecture for the smallest and simplest cases, then we can change the parameters value still granting a correct result.
The testing procedure relies also on a Python code: in fact MATLAB and VHDL output files are not directly comparable but they are manipulated with Python in order to do this needed operation in the easiest and most powerful language we can use.
The simulation and synthesis results will be shown for several N and d_v values used by Ledacrypt: since the design algorithm extend the data to a power of 2, in general only one parameter between two consecutive power of 2 is shown: all the other ones will lead to the same simulations results and to only minor differences for the synthesis part (this is because every synthesis reaches a local optimum solution only), as it can be seen by the last two cases of every following table, which are the extreme possible values between 2^{15} and 2^{16}.

6.1 Simulation results

The number of clock cycles required to complete the cyclic multiplication is obtained performing Modelsim simulations: the results are shown in table 6.1 and they represent the most probable case, that is the one in which every bit of each 1's position has the same probability of being equal to 0 or 1 (the value of every bit affects the total number of clock cycles differently, since in the sparse algorithm if it is equal to 1 multiplications have to be computed, otherwise they are not needed).

In this first analysis we consider an implementation in which all the portions of the convolution algorithm are executed one after the other: however, many improvements can be done.

N	d_v	Clock cycles
15013	9	199k
27779	17	431k
35027	17	909k
65029	23	921k

Table 6.1: Simulation results

As we can see, the results are fine but not great. In contrast, the vector by circulant operation described in chapter 3 requires at most $d_v \cdot N$ clock cycles, so in the first case of the table roughly $135k$.
However the several improvements proposed in section 6.5 could lead to a strong reduction in terms of clock cycles required to compute the computation without the need of adding too many resources, and the final results will tend to the ones of the vector by circulant architecture.
Moreover, the number of clock cycles required by the proposed architecture is not linearly dependent with d_v, and increasing this value will lead to a small increment in the total number of clock cycles, making this new architecture suitable for both sparse and dense input vectors, as it can be seen in table 6.2 in which 4 possible d_v values along a geometric progression have been considered. In fact, the number of sparse stages for the sparse part can be reduced (contextually increasing the number of dense stages) in order to almost cancel the d_v growth. However, the number still grows since we are reducing the power of the sparse algorithm of skipping many operations.

N	d_v	Clock cycles
15013	9	199k
15013	25	208k
15013	69	220k
15013	192	229k
15013	533	237k

Table 6.2: Simulation results with different d_v values

6.2 Synthesis results for FPGA applications

The architecture has been synthesized on one of the FPGA Artix-7 200 family, since the NIST has recommended to use Artix-7 FPGAs. We have considered a clock period of 10 ns which is always met and synthesized the architecture for several N and d_v values. The results are shown in table 6.3 in terms of resource utilization and in table 6.4 in terms of percentage of resources used.

N	d_v	LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
15013	9	5396	43	3774	52	40	43	1
27779	17	6259	43	4053	100	40	45	1
35027	17	7062	43	4358	199	40	46	1
65029	23	7029	43	4351	199	40	46	1

Table 6.3: FPGA resource utilization

N	d_v	LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
15013	9	4.01	0.09	1.40	14.25	5.41	15.01	3.13
27779	17	4.65	0.09	1.51	27.40	5.41	15.79	3.13
35027	17	5.25	0.09	1.62	54.52	5.41	16.14	3.13
65029	23	5.22	0.09	1.62	54.52	5.41	16.14	3.13

Table 6.4: FPGA resource utilization (%)

As it can been seen from table 6.4, the resources used are almost in every case only a small percentage of the total ones: hence this architecture is suitable for low area applications. Nonetheless, the BRAM utilization is not negligible, especially for high values of N: since the memory usage cannot be reduced (it is a direct consequence of the N value), an attempt to optimize the solution can be increasing the parallelism of the architecture: with an higher utilization of LUT and DSP (but still below critical values) we can significantly reduce the number of clock cycles required by the whole convolutional operation. However, increasing the number of multipliers will lead to an important growth in the power consumption and this is an important aspect that has to be considered in the attempt to optimize the architecture.

6.3 Synthesis results for ASIC applications

In this case the architecture is synthesized on the 45 nm technology node using Synopsys. The RAM memories as well as the ROM ones are not considered in this synthesis analysis (aside from the position memories of the circulant extensions architecture, which are small and can be synthesized with flip-flops) and the results are shown in table 6.5.

N	d_v	t_{cp}	Total area	Static power
15013	9	3.31 ns	77599 μm^2	1.48 mW
27779	17	3.51 ns	81567 μm^2	1.62 mW
35027	17	3.71 ns	93582 μm^2	1.83 mW
65029	23	3.71 ns	93582 μm^2	1.83 mW

Table 6.5: ASIC resource utilization

The critical path, as expected, is due to the multiplier: for this reason this block has not been designed with a fixed latency but it asserts an output ready signal when the product is computed. Increasing the latency of the multiplier will ensure a reduction in the critical path with very minor changes to the other existing components.
Most of the area and the static power are due to the multipliers: for this reason every attempt to reduce the number of bit needed for the operations is useful. The multipliers area and static power change with the number of points used for the NTT (and so with the number of bit used by the operators) and this behaviour is shown in table 6.6. The non linear behaviour is due to the values assumed by k and P since they can strongly influence the components used for the multiplications by these constant values. Since the $NTTs$ are computed considering number of points equal to a power of 2, this opportunity has to be considered when N has to be chosen: increasing N without exceed a power of 2 could improve security behaviour without impacting on the total resources used by the cyclic multiplier.

N	bit	Total area	Static power
15013	16	3042 μm^2	65 μW
27779	17	3803 μm^2	79 μW
35027	18	4724 μm^2	98 μW
65029	18	4724 μm^2	98 μW
99053	19	4946 μm^2	102 μW

Table 6.6: Multiplier resources

6.4 Performance comparison

In this section the proposed design is compared with several other architectures which perform the same kind of product, in the two cases of FPGA and ASIC application.

6.4.1 FPGA applications

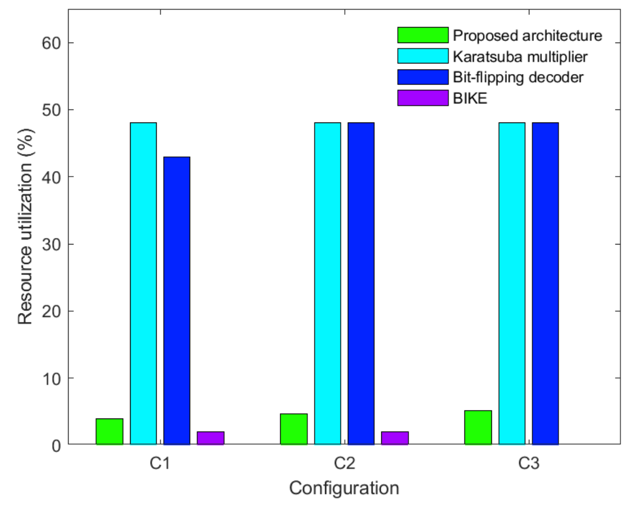
The results obtained by the proposed design are now compared to several existing architectures, which have also used Artix-7 200 FPGAs to obtain the resource utilization values:

- In [12] a Karatsuba multiplier has been presented and the architecture has been submitted for the round 1 of the Post-Quantum Cryptography at NIST as part of the Ledacrypt algorithm;
- In [13] an efficient bit-flipping decoder has been presented and submitted for the round 2 of the Post-Quantum Cryptography at NIST (also in this case it is part of the Ledacrypt algorithm): even though in our case we have the cyclic multiplier only, we consider the comparison meaningful;
- In [14] a multiplier has been proposed as part of the BIKE algorithm: also this solution has been presented for the round 2 of the Post-Quantum Cryptography at NIST.

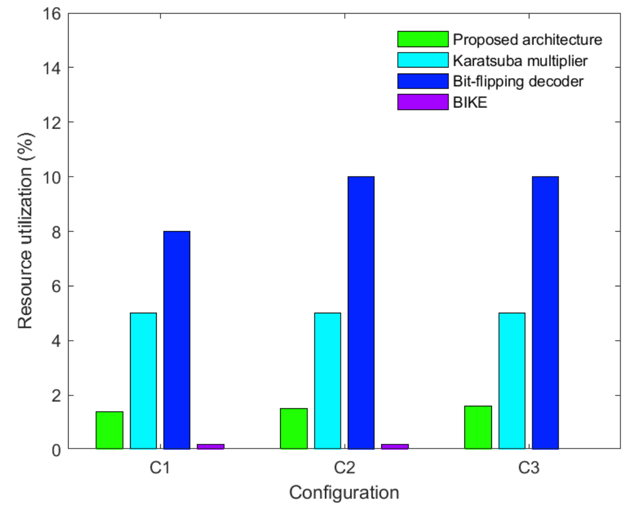
The comparison is shown for the only common metrics for the 4 architectures and the results are shown in figures 6.1a, 6.1b and 6.1c. For each graph the several configurations considered are presented in table 6.7: the N values are not exactly the same for all the cases, but the closest values among all the architectures have been taken in order to have the most meaningful comparison. For the C3 configuration case, the BIKE algorithm is not defined. The scale for the y axis is different for each graph in order to show the maximum possible information.

Configuration	N
C1	$9643 \div 10883$
C2	$24533 \div 25603$
C3	$35027 \div 37619$

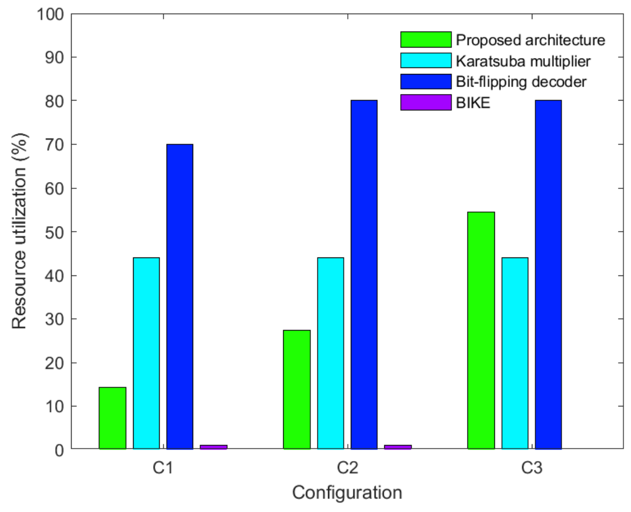
Table 6.7: Configurations for FPGA comparison



(a) LUT utilization comparison



(b) FF utilization comparison



(c) BRAM utilization comparison

Figure 6.1: Resources utilization comparison

As it can be seen from the FPGA data, the proposed design uses much less resources with respect to the first two alternatives, meaning that this solution is suitable for low area applications.
However, the BRAM usage double every time we exceed a power of 2: as it can be seen the proposed architecture is a great choice for $N < 2^{15}$, while other options have to be considered for higher values of N. A solution to the BRAM utilization issue can be increasing the parallelism of the unit, contextually halving the memory length, because sometimes the BRAM usage raises a lot when more blocks have to be considered together. Although this consideration can be valid, in this case doubling the parallelism leaves the BRAM usage unchanged, meaning that other solutions have to be found in order to reduce the memory utilization. A possible solution has been described in 4.1 in the choice of the P value for the NTT: in case of sparse input data if sometimes we can accept a wrong cyclic multiplication output we can reduce the P value and hence the memory usage, since P directly influence this parameter (in fact for each memory location we need $log_2 P$ bit). However, the accuracy reduction has to be estimated: it is a function of the input vector sparsity and its value has to be considered in the whole post-quantum algorithm since it can lead to wrong decoded messages.

Considering the data shown in the previous figures, the BIKE algorithm seems the best solution: it is an algorithm that uses a very limited amount of resources since it exploits the rotation technique to perform the multiplication, so its performances deeply depend on the input vector sparsity and they can worsen significantly for more dense input data.
Moreover, a comparison on the latency of the 4 algorithms has to be considered: it is shown in table 6.8 and shows both the latency and the frequency chosen for each synthesis; for the BIKE algorithm, since it uses different frequencies, these data are shown in table 6.9.

Configuration	Proposed architecture	Karatsuba multiplier [12]	Bit-flipping decoder [13]	BIKE [14]
	$f = 100\,MHz$	$f = 143\,MHz$	$f = 100\,MHz$	f: table 6.9
C1	$1.9\,ms$	$0.005\,ms$	$0.071\,ms$	$0.093\,ms$
C2	$4.3\,ms$	$0.028\,ms$	$0.120\,ms$	$0.535\,ms$
C3	$9.1\,ms$	$0.051\,ms$	$0.116\,ms$	N/A

Table 6.8: Latency comparison

Configuration	Frequency
C1	$f = 277\,MHz$
C2	$f = 163\,MHz$

Table 6.9: BIKE frequency data

As we can see, the BIKE algorithm has not good performances; the comparison with the Bit-flipping decoder is not that meaningful since its latency is the one of the whole decoder: however it has been considered to show that it presents better performances with respect to the BIKE algorithm even tough the cyclic multiplication is only a part of the whole decoder operation.

Nevertheless, a more meaningful comparison can be the LUT $\times$ latency product since it is a figure which represents the trade-off between area and speed. In this case the comparison is shown in figure 6.2 and shows that the architecture proposed in this work is not a good solution; however this value can be significantly reduced with the improvements shown in 6.5 and the synthesis has been performed with a clock frequency of 100 MHz: an higher frequency value can still be used to achieve a better result. Moreover the latency performances are almost maintained for the cases of vector with higher density and this is not true for the other algorithms.

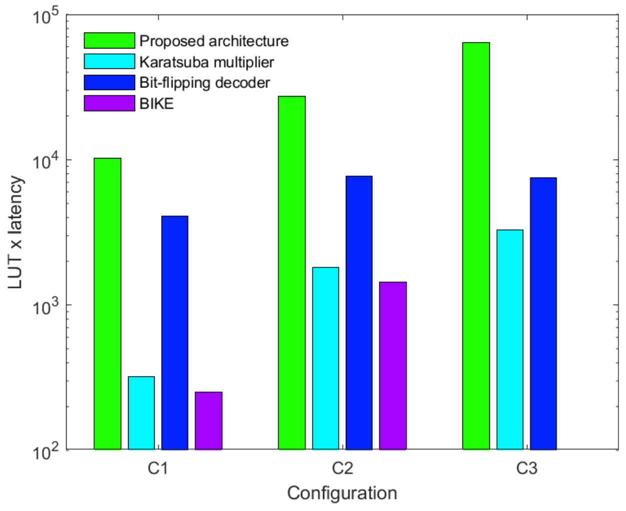


Figure 6.2: LUT $\times$ latency comparison

6.4.2 ASIC applications

In this case, the results obtained by the proposed design are compared with an architecture with the vector by circulant architecture described in [6]. The comparison is not that meaningful, since in the reference case the whole decoder has been synthesized and several parallelism values can be chosen (all the power of 2 between 8 and 64). Only the case with $N = 27779$ is available and the comparison is shown in table 6.10 for the smallest and the largest possible parallelism.

Architecture	t_{cp}	Total area	Static power
Proposed design	$3.51\,ns$	$81567\,\mu m^2$	$1.62\,mW$
Vect. by circ., $N_b = 8$	$3.72\,ns$	$15618\,\mu m^2$	$1.11\,mW$
Vect. by circ., $N_b = 64$	$3.86\,ns$	$129047\,\mu m^2$	$5.52\,mW$

Table 6.10: ASIC resource utilization comparison

As it can be seen, the critical path in the architecture proposed in this work is lower (and it can even be reduced at almost no cost increasing the latency of the multipliers), but especially the area is much higher in almost every case. The static power values of the proposed design are also not negligible, since in some cases they are even higher with respect to the ones of the vector by circulant architecture.
However, the proposed architecture has better performances in case of more dense input data and another consideration has to be made for the specific Ledacrypt and BIKE cases: in fact in this algorithm a sequence of cyclic multiplications have to be done and with the use of the convolution theorem we can pass from the "time domain" to the "frequency domain" only at the beginning and at the end of the multiplications sequence (the quotes are present since we are not handling time/frequency signals and since the output of the NTT has not the same physical meaning the DFT has; in this case the time domain corresponds to the bit domain).
Since for a sequence of multiplications we can save several $NTT/INTT$ operations and perform the $INTT$ after the last multiplication only, we can reduce the latency of a significant amount, making this algorithm a good candidate every time a sequence of cyclic multiplications have to be computed.

6.5 Potential improvements

The architecture can be optimized in order to significantly reduce the number of clock cycles required to compute the result. Most of the proposed solutions can lead to an important improvement almost without the need of additional hardware resources.

6.5.1 A complete control unit revision

A consideration has to be made on the number of cycles required to produce the correct output: as we have seen so far, the HW resources are not used the 100% of the time. In fact we alternate the use of multipliers and adders/subtractors: a possible improvement is to optimize the use of the resources when we perform the 2 *NTT*, so when the first *NTT* uses the multipliers the second one uses the adders/subtractors and vice versa. This implementation require a more complicated control unit, since we need to handle 2 *NTT* at the same time, but we can save the number of clock cycles required to compute a whole *NTT*.
The control unit instead doesn't require a too complicated modification for the sparse vector by circulant architecture: since both the direct *NTT* consider the input vectors as in bit-reversal, they need the same sequence of constants for the multipliers.
However just a revision of the control unit will lead to a reduction of approximately one third of the total number of clock cycles.

6.5.2 Element-wise multiplication and INTT

The element-wise multiplications require many clock cycles to be completed and during this period of time the adders/subtractors cannot be used by the *INTT* algorithm since the input vector is still not ready. However we don't need all the output of the element-wise to proceed with the *INTT*: as soon as we have enough of them we can split the resources between the element-wise multiplication and the *INTT* and so overlap a few stages of the *INTT* with the ones of the element-wise operation.

6.5.3 A different hardware resources allocation

Considering how much area an adder/subtractor requires with respect to a multiplier, there is the possibility to significantly increase the number of adders/subtractors without impacting a lot on the total area. In this case we can reduce a lot the number of clock cycles required to compute the convolutional product, with the most of them required by the multipliers. In order to do so, we need to do adapt the data memory parallelism to the number of adders/subtractors and we need to insert additional multiplexers in front of the multipliers since now they are less than the memory parallelism.

6.5.4 Input and output vectors in bit-reversal order

So far we have considered that both the input and the output vectors have to be ordered: however if they are expressed in the bit-reversal order we can have some simplifications. For the sparse vector by circulant we will have significant improvements, since now the bit reversal architecture is no more needed and it was part of the total latency required by the convolutional product. For the vector by circulant architecture instead we will just have a control unit simplification: since we will have to read the constants values in the same order we can more easily perform the two NTT in parallel.

6.5.5 Constant memory management revision

One of the main problem increasing the N value is the number of BRAM blocks we need: however this cannot be reduced since depends on both P and N and we always need two RAM memories for the two NTT.
Increasing the N value also leads to an important growth in the dimension of the ROM memories in the Constant memory management block. In the first implementation we have decided to store all the possible constant values in order to speed up he computation, but we could have also decided to compute the constants on the fly: we can store a limited number of constants (e.g. all the constants with an exponent a power of 2) and computed all the needed ones with dedicated multipliers every time they are needed. This solution increases the latency of this block but since it uses a fully handshake protocol the rest of the architecture is unaffected.
In order to further reduce the resource utilization we can also exploit the multipliers used to compute the NTT: in this case we need an additional input for the multiplexers in front of them and also a minor change in the control units which compute the NTT, since they have to drive the proper inputs to the multipliers when we need to compute the constants values.

Chapter 7

Conclusions

The multiplication between a vector and a sparse binary circulant matrix is an important operation in many post-quantum cryptography algorithms.
With this work we have presented a cyclic multiplication architecture relying on the convolutional theorem and the Number Theoretic Transform. The architecture has several advantages with the others mentioned throughout the work: only minimum modifications are needed to handle every possible combination of input and output data type (binary or integer), with area and number of clock cycles that remain almost unaffected.
Moreover one of the main advantage of this architecture is that the total number of clock cycles and the area are almost uninfluenced by the density of the input vectors: in fact changing the density of the vector would lead to a slightly different architecture realization in order to maintain the same performances.
The architecture has many degrees of freedom in order synthesise the best solution for each specific case.

7.1 Future work

A future work can be a fully parametric implementation of the architecture: since most of the architecture's blocks rely on the handshake protocol only minor and local modifications have to be implemented. Moreover, in order to optimize the architecture performances, the improvements described in section 6.5 can be considered and implemented. Most of them will lead to a significant reduction in terms of clock cycles required in order to complete the multiplication almost without the need of additional resources.

Bibliography

[1] Quantum Backdoor. *Symmetric-key algorithm in Cryptography*. Ed. by Medium. 2020. URL: `https://medium.com/@.Qubit/symmetric-key-algorithm-in-cryptography-3d839bba8613`.

[2] Nassos Michas. *An Introduction to Public Key Cryptography*. Ed. by Better Programming. 2020. URL: `https://betterprogramming.pub/an-introduction-to-public-key-cryptography-3ea0cf7bf4ba`.

[3] Kanad Basu et al. *NIST Post-Quantum Cryptography - A Hardware Evaluation Study*. Cryptology ePrint Archive, Report 2019/047. 2019. URL: `https://ia.cr/2019/047`.

[4] Daniele Micciancio and Oded Regev. "Lattice-based cryptography". In: *Post-quantum cryptography*. 2009. DOI: `10.1007/978-3-540-88702-7_5`.

[5] Marco Baldi et al. *Low-density parity-check code-based cryptographic systems*. 2020, p. 101. URL: `https://www.ledacrypt.org/documents/LEDAcrypt_spec_2_5.pdf`.

[6] Kristjane Koleci et al. "A Hardware Implementation for Code-based Post-quantum Asymmetric Cryptography". In: *ITASEC*. 2020.

[7] Geppino Pucci. *Convolutions and the Discrete Fourier Transform*. URL: `http://www.dei.unipd.it/~geppo/DA2/DOCS/FFT.pdf`.

[8] Kazushi Kawamura, Masao Yanagisawa, and Nozomu Togawa. "A Loop Structure Optimization Targeting High-level Synthesis of Fast Number Theoretic Transform". In: *19th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA* (2018), pp. 106–111. DOI: `10.1109/ISQED.2018.8357273`.

[9] Peter L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521. DOI: `10.1090/S0025-5718-1985-0777282-X`.

[10] Nayuki. *Montgomery reduction algorithm*. 2020. URL: `https://www.nayuki.io/page/montgomery-reduction-algorithm`.

[11] Jechang Jeong and William J. Williams. "A fast recursive bit-reversal algorithm". In: *International Conference on Acoustics, Speech, and Signal Processing* 3 (1990), pp. 1511–1514. DOI: `10.1109/ICASSP.1990.115695`.

[12] Davide Zoni, Andrea Galiberti, and William Fornaciari. "Flexible and Scalable FPGA-Oriented Design of Multipliers for Large Binary Polynomials". In: *IEEE Access* 8 (2020), pp. 75809–75821. DOI: 10.1109/ACCESS.2020.2989423.

[13] Davide Zoni, Andrea Galiberti, and William Fornaciari. "Efficient and Scalable FPGA-Oriented Design of QC-LDPC Bit-Flipping Decoders for Post-Quantum Cryptography". In: *IEEE Access* 8 (2020), pp. 163419–163433. DOI: 10.1109/ACCESS.2020.3020262.

[14] Jan Richter-Brockmann, Johannes Mono, and Tim Guneysu. "Folding BIKE: Scalable Hardware Implementation for Recongurable Devices". In: *IEEE Transactions on Computers* (2021). DOI: 10.1109/TC.2021.3078294.