POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS

Master's Degree in Electronic Engineering

Master's Thesis

Multiplier for quantum cryptography algorithms in FPGA

Politecnico di Torino

Supervisors
prof. Maurizio Martina
prof. Guido Masera

Candidato
Alberto Vaudagna
matricola: 265418

Anno accademico 2021-2022

Acknowledgements

I would like to thank Professor Masera and especially Professor Maurizio Martina who provide their valuable inputs for this thesis I was already interested in the matter of codes and hardware architecture during the class of Digitial Communication and this thesis allow me to develop on what I previously learned. I also want to thank PhD student Kristjane Koleci for the help she gave me about the algorithm necessary to develop this thesis. I would like to thank as well all my family members that supported me during these months and years , all of my friends and ex-friends that allow me to continue my studies even in the hardest moments.
I would also like to thanks all of the members of team H2politO that have been part of my life for the last couple of years and that really help me to learn a lot and to grow as a person. Finally I would like to thanks polito and all the workers and professors in general that I have met over the years and witn their passion and love have allow me to pass all the exams. Ultimately I would like to thanks all my colleague that I meet over the years in polito.

Abstract

The ever changing world require, day after day, an increasing amount of data to be transmitted and stored in servers. With this amount of data the problem of making data accessible only to the owner or to maintain this data secret among an organization has increased. Hence the field of Cryptography has gained success in modern times, especially today , where the advance of quantum computing could make the cryptography system less secure by breaking old RSA systems. It is important to develop new systems that are still secure against quantum computer attacks. To overcome such problems, the NIST (National Institute of Standard and Technology) have host a competition in order to create new technologies that are still secure against quantum computer attacks. This competition, that now is in round three, have seen around 70 different systems that after the different rounds have been reduced to the final 15. Among those, LEDACrypt, developed by the Italian researchers from Università delle Marche and from Politecnico di Milano have reach Round2, but due to a severe flow in how the private key is computed the system didn't reach Round3. Despite this failure, this system introduce a new LDPC decoders, called Q-Decoder, that have also interested in the world of communication systems. A similar system, called BIKE [2], based on the same idea of using quasi cyclic matrices to represent the private/public key have reached round 3 of the competition.
During the decoding stage, of both systems, an important operation is the multiplication of a binary vector by a circulant matrix (used to obtain the syndrome of the incoming codeword). This operation, when applied to quasi cycle matrices, is equivalent via a ingenious mathematical trick to the multiplication of two binary polynomials. Traditional this kind of operation, also knows as polynomial multiplication, is calculated when the number of bits is high using sophisticated algorithms such as the Karatsuba algorithm or the Comba algorithm. The main problem of the application of such algorithms is that they are not scale linearity with the degree of the polynomial (that depends on the number of bits of the vector). In LEDAcrypt the multiplication is performed between vectors of at least thousands of bits (up to more than twenty-thousands bits for system with higher security category). The implementation using the mentioned algorithms is not trivial in both term of hardware resource requirements and complexity and execution time. More

complicated methods have to be developed such as the FFT to transform the polynomial multiplication into a convolution with the hope of speed up the calculation. The optimization of this operation yields to an overall decrease in execution time of the decoding stage.
This thesis explore a possible implementation of such operation by noting that it is equivalent, when using cyclic matrices, to the convolution of the two binary vectors. The developed system first calculate the FFT of the two input sequences than multiplies the two result vectors together using a complex multiplier and finally obtains the result binary vectory calculating the IFFT and appliy a proper rounding methos. LEDAcrypt and Bike systems for technical reasons impose the number of bits of the incoming vector to be a prime number so it is not trivial to apply classical FFT transform to those sequences. To overcome such problem an ingenious type of padding is proposed in the thesis to extend the incoming vector to a vector that is a power of two and so traditional FFT circuit can be used.
The thesis is divided into three chapters, initially an introduction to code theory and circular matrices is provided, than an introduction of crypto systems and LEDA is presented. Finally the last chapter analyze the development system that aim to optimize one of the key operation of the Q-Decoder that is a particular type of polynomial multiplication called vector by circulant operation by computing it as cyclic convolution.

Contents

List of Tables

List of Figures

Chapter 1

Preliminaries

In this chapter, the basic mathematics tools are introduced. First a small introduction to the Coding theory will be we will presented a small introduction to Coding theory, together with some basics facts about the DFT, and an introduction to circulant matrices.

1.1 The DFT and IDFT

A very well know mathematics operator, the Fourier Transform, is often needed in embedded and DSP application to view a given signal in the frequency domain and obtain some characteristic of the original signal that is hidden in the time domain. With the Fourier transform it is possible to, given a signal x(t), analyze its frequency components. The DFT is the discrete case of the Fourier Transform where the signal is no more a time continue signal but it is made of sample took at different time intervals. The infinite sum become a sum over the number of samples of the signal. The Discrete Fourier Transform is the building block of a lot of modern technologies such as the OFDM (Orthogonal Frequency Division Multiplexing) modulation and it is extensively used in modern communication systems and digital Spectrum analyzers to be able to analyzed the frequency domain.
We suppose to have a sequence of N complex numbers $\{x_i\}_0^{N-1}$, $x_i = x_{re} + jx_{im}$ than the DFT is defined as:

$$\overline{X}_n = \sum_{k=0}^{N-1} X_k e^{-j\frac{2\pi}{N}kn} = \sum_{k=0}^{N-1} x_k(cos(\frac{2\pi}{N}kn) - jcos(\frac{2\pi}{N}kn)) \tag{1.1}$$

The domain of the Fourier transform is the Fourier domain and is the field $\mathbb{CS}$. For any transform operation there is always an inverse operator, that convert the value back to the normal domain, called the IDFT and it is defined as:

$$X_n = \frac{1}{N}\sum_{k=0}^{N-1} \overline{X}_k e^{j\frac{2\pi}{N}kn} \tag{1.2}$$

It is noticed that IDFT is nothing more than the normal DFT but with the conjugate exponent coefficients. This is due to some mathematics properties of the Fourier transform. The normalization factor that appear in the IDFT formula can be also chosen to be $\frac{1}{\sqrt{N}}$ if a unitary transform is required. In this thesis we use the convention that every vector obtained from a FFT is marked with a bar above it, the n index always indicate the sample of the vector for which the FFT is calculated and k is the summation summation index.
Another way to described the DFT is via the so called DFT matrix that represent the set of all the coefficient that are required for the transform. We define the DFT matrix of order N to be:

$$\begin{aligned} \omega &= e^{-j\frac{2\pi}{N}} \\ F &= \frac{1}{\sqrt{N}} * \begin{bmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{N-2} \\ 1 & \omega^3 & \omega^6 & \dots & \omega^{N-3} \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{N-2} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix} \\ F^{-1} &= \frac{F^*}{N} \end{aligned} \tag{1.3}$$

Than to obtain the DFT of an input vector we can just multiply it with this matrix. F^{-1} that is the IDFT matrix.
The basic DFT algorithm have a computational complexity of the order of $O(n^2)$ because to compute an entire result vector we require $(N-1)$ sums of exactly $(N-1)$ elements. There are quite a lot of tricks and methods to reduce the complexity down to something that scales as linear as possible respect the number of elements. The most known method is the Fast Fourier Transform, FFT that reduce the complexity from $O(n^2)$ to $O(n * log(n))$ and it is used when the number of samples are a power of two. The FFT will be discussed in details later on as it is a very important tool to improve the performance of one of the key operation of the LEDA Q-Decoder algorithm.

1.2 Circulant Matrices

A circulant matrix is a matrix for which each row (or column), are a shift of the previous one. A general N x N circulant matrix is in the form:

$$C = \begin{bmatrix} c_0 & c_{j-1} & c_{j-2} & \dots & c_1 \\ c_1 & c_0 & c_{j-1} & \dots & c_2 \\ c_2 & c_1 & c_0 & \dots & c_3 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_{j-1} & c_{j-2} & c_{j-3} & \dots & c_0 \end{bmatrix} \tag{1.4}$$

It is easy to see that every circulant matrix can be identified uniquely by the element of the first column (or row) identified by the vector $\underline{c} = [c_0, c_1, ..., c_{N-1}]$ and the shift amount S_m. For the purpose of this thesis S_m is always set to 1 so to describe a circulant matrix only the first row (col) is required. All the other columns (rows) are the shifted version of the first by the given amount. In literature some authors consider matrices where the circulant elements are on the columns and some consider the rows. In this thesis we consider circulant matrices in the columns.
If binary circulant matrices are considered, so all the elements are 0 or 1, than the first column (row) can also be represented by $\underline{c_p}$ that contains the position of the non null element of $\underline{c}$. If one consider sparse matrices, where the size of $\underline{c_p}$ is small respect to the size to N, than storing the element by the index of the non null entries allow to reduce the memory footprint.

Prop. 1 *The set of all N x N abinary circulant matrices form a ring under the binary operation of multiplication and addition. The zero element is the zero matrix and the identity element is the N x N identity matrix. This ring is isomorphic to* $\mathbb{F}_2$ *by this map:*

$$C \longleftrightarrow c(x) = \sum_{i=0}^{N-1} c_i x^i \tag{1.5}$$

where c_i *are the element of the first row (column) vector* $\underline{c}$*.*

This map, that is an isomorphism, is used to speed up the calculation regarding circulant matrices as a multiplication between a vector and a matrix can be view as the multiplication between two polynomials in GF(2). For LEDAcrypt it is also necessary to consider circulant matrices that are invertible. This condition is grantee, as stated by theorem Theorem 1.1.13 and 1.1.14 [3], if one consider a matrix of size p x p where p is a prime number greater than 2 such that $ord_2(p) = p - 1$ and the weight of each rows (cols) is odd.
Another useful property that circulant matrices have is the following:

Prop. 2 *Given any circulant matrix C, of order N, than it is easily to obtain its transpose by applying the following bijective map:*

$$\phi_N(x) = (N - x) \mod N \tag{1.6}$$

This map allow, in the same way, to given the transpose matrix to obtain the original matrix.

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_{l-1} \\ c_{l-1} & c_0 & c_1 & \dots & c_{l-2} \\ c_{l-2} & c_{l-1} & c_0 & \dots & c_{l-3} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_1 & c_2 & c_3 & \dots & c_0 \end{bmatrix}^T \tag{1.7}$$

Clearly the indices of the right matrix follow the above relation, for example in the below table some value of l are computed:

j	$\phi_N(j)$	**l**
0	$(N-0) \mod N$	0
1	$(N-1) \mod N$	N-1
2	$(N-2) \mod N$	N-2
$\vdots$	$\vdots$	$\vdots$
N-2	$(N-N+2) \mod N$	2
N-1	$(N-N+1) \mod N$	1

The same relation can be used to obtain from the position of the non null element of the first row the one of the first col and vice versa.

1.3 Coding Theory

The increasing number of information send across devices has since the creation of the information theory results in the development of new methods to send information reliably and to detect and correct errors generated by the channel noise. These methods are based on the idea of adding redundancy bits to the data sent over the channel so the receiver can detect and correct the errors. A good performance measure that can be applied is the code rate, $R_c = \frac{k}{n}$ where k is the number of bits of the encoded information and n is the number of bits of the original information. In fact codes allow to correct and detect errors but they always increase the bit rate of the system because additional bits are required.
In mathematical terms, a code is bijective map between two binary field with the addition corresponding the XOR operation and multiplication corresponding to the logical product between two bits:

$$C(n,k) : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^k, 0 < n < k. \tag{1.8}$$

The elements in $\mathbb{F}_2^n$ are called an information words, the elements in $\mathbb{F}_2^k$ are called a codewords, n is called the code length and k is the code dimension. The encoding procedure is the operation of mapping the information word to its appropriate codeword. The decoding procedure is the operation that given an error affected codeword obtains the error vector and the original information word.
A family of code that have an error correction capability that reach channel capacity are the so called Linear codes and are the basis of a lot of communication systems. We call the code C(n,k) a linear code if the set of it's 2^k codeword form a $\mathbb{F}_2$ subset of dimension k. The minimum code distance, d_{min}, is the minimum hamming weight of the non zero codewords. There is an important bound to the value of d_{min} of given code C(n,k) called the singleton bound:

$$d_{min} \leq n - k - 1 \tag{1.9}$$

This is an important relation because it allow to evaluate a trade off between n and k. Indeed if we want to correct a lot of errors than we would choose a code with a big d_{min} but that result in a bigger k and that is a problem because it means that the decoder need to perform more operation and so the system latency is higher.

Supposing we are considering linear code than we define the elements of the information word as the vector $\underline{v} = [v_0, v_1, ..., v_{n-1}]$, we can pick k linear independent codewords, $g_0, g_1, ..., g_{k-1}$, that form a basis of $\mathbb{F}_2^n$ than any codeword $\underline{c} = [c_0, c_1, ..., c_{n-1}]$ can be write as a linear combination of the basis elements:

$$\underline{c} = v_0 g_0 + v_1 g_1 + ... + v_{k-1} g_{k-1} \tag{1.10}$$

the above formula can be rewritten by introducing the so called generating matrix:

$$\underline{c} = \underline{v} G \tag{1.11}$$

where G is defined as a k x n binary matrix, called the code generator matrix:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} \tag{1.12}$$

The code C(n, k) is systematic if all of its codewords contain the information word vector associated to it. A good way to generate codes that are always systematic is to pick, for every information word, the corresponding codeword by appending $r = n - k$ redundancy bits $[c_k, c_{k+1}, ..., c_{n-1}]$ to the selected codeword. Than a generic codeword is in the form $\underline{c} = [v_0, v_1, ..., v_{k-1}, c_k, c_{k+1}, ..., c_n]$. In this case, the generating matrix can be written as:

$$G = [I_k | P] \tag{1.13}$$

where I_k is the identity matrix and P is a binary vector of dimension k x r. We can now define another important matrix the parity check matrix H of dimension r x n as:

$$H = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{r-1} \end{bmatrix} = \begin{bmatrix} P \\ - \\ I_r \end{bmatrix} \tag{1.14}$$

1.3.1 Error Detection

An important role of codes is the ability to detect an error in an encoded message. In a communication system normally errors are due to noise in the channel. Let's suppose we have the following system:

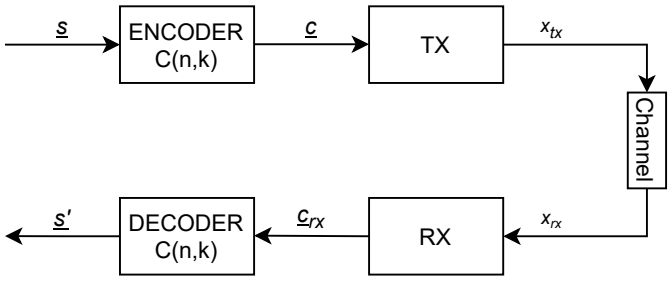


Figure 1.1: Default Communication System

The data is first send to a block that apply a given code $C(n,k)$ than sent to a transmitter that apply some kind of modulation in order to have the signal translated in frequency and ready to be sent using, for example, an antenna. At the receiver side the signal is first enter the receiver block that demodulated the signal to obtain a binary sequence. This sequence of bits is then applied to a decoder. The output of the decoder is the signal $\underline{s}'$. The code is used to correct the errors added by the channel noise and to have that $\underline{s}' = \underline{s}$.
In general we suppose that we are sending through the channel an information word $\underline{v}$ encoded via a given code C(n, k) to codeword $\underline{c_{TX}} = [c_0, c_1, c_{n-1}]$, at the output of the channel we have the codeword $\underline{y}$ that is affected by an error vector $\underline{e}$ $= [e_0, e_1, ..., e_{n-1}]$ defined as:

$$e_i = \begin{cases} 0 & y_i = c_i \\ 1 & y_i \neq c_i \end{cases} \tag{1.15}$$

we can write that for a general channel we have the following relation:

$$\underline{y} = \underline{c}_{TX} + \underline{e} \tag{1.16}$$

The idea is that if we are able to extract from $\underline{y}$ the error vector than if $\underline{e}$=0 the incoming codeword is accepted otherwise the receiver system ask for a new message until a valid codeword is received. This method is better known as the integrity check test. Knowing $\underline{y}$ and supposing that $\underline{e}$ is recovered the original non error affected codeword $\underline{c}_{TX}$ can be found by the following equation:

$$\underline{c}_{TX} = \underline{y} + \underline{e} \tag{1.17}$$

To detect the errors, we exploit the redundancy bits that the encoded message presents. We recall that the encoder has added bits and placed it after the information bits, also the mapping between the codeword and the information word is done by the generator matrix G. Given the parity check matrix H than, considering a systematic code, for any codeword $\underline{c}$ the following equation is true:

$$\underline{c}H = \underline{0} \tag{1.18}$$

where $\underline{0}$ is the zero vector.
The H matrix can be used to verify if the received codeword have errors or not by computing the so called syndrome, indicated by the vector $\underline{s}$:

$$\underline{s} = \underline{y}H \tag{1.19}$$

it is trivial to prove that if $\underline{y} = \underline{c}_{TX}$ than the syndrome is always a zero vector.
Given the syndrome, we can reformulate the method to detect errors as: Given the received vector $\underline{y}$ potentially affected by the error $\underline{e}$ and encoded by the code C(n,k), compute the syndrome $\underline{s}$:

- If $\underline{s} \neq \underline{0}$ than the received vector is rejected.
- If $\underline{s} = \underline{0}$ the message is accepted but there is still the situation that $\underline{y}$ is wrong in the case that the error vector $\underline{e}$ is s codeword in itself. This condition is impossible to avoid and is called the undetected error condition.

To minimize the number of undetected errors it is sufficient to design code for which the d=d_{min} is minimum, this condition assure that no codeword of weight 1,2, ..., $d-1$ exists and so the undetected errors are reduced.

1.3.2 Error Correction

We have to determine how can we correct a detected error. Again the redundancy added by the code C(n,k) is exploited. As said before, the operation that given $\underline{y}$ retrieve both the original information word and the error vector is the decoding operation. Mainly there are two families of decoders one that employ some knowledge of the transmission channel in term of probabilities (this happen in the decoder used for the communication systems) and decoders that do not use this information at all such as, for example the LEDAcrypt Q-Decoder. An important property of the code is the error correction capability of the code. This quantity is identified by:

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor \tag{1.20}$$

than the following statement holds:

Prop. 3 *Given a code C(n, k) with given d=d_{min} than it certainly correct all errors of weight less or equals to t.*

$$w_H(\underline{e}) \leq t \tag{1.21}$$

where w_H is the hemming weight of the error vectors.

In general the decoding procedure produce a codeword $\underline{c}_R$ that is closest as possible as one of the codeword of the code in consideration:

$$\underline{c}_R = \arg \min_{c \in C(n,k)} d_H(\underline{y}, \underline{c}) \tag{1.22}$$

1.4 LDPC Codes

Having defined some of the basic theory of codes now we can introduce a particular family of codes called Low Density Parity Check Codes (LDPC). They were first introduced by Gallager in the 60's but nowadays they are used in communication systems such as 10Gb Ethernet and in different Wi-Fi standards such as the 802.11ac, the 5 Ghz Wi-Fi Standard.
A LDPC code is a type of a more general class of codes called Linear Block codes and it is characterized by a sparse parity check matrix H in the sense that the hamming weight of the column, defined as d_v, is much smaller than r that is the size of the column. There is an association between the parity check matrix and what is called the Tanner graph, it has been proved that LDPC codes with short cycles have good error correction performances.

Def. 1 *A Tanner graph is a graph in which:*

- *Each codeword* $\underline{c}$ *is associated to a variable node*
- *Each parity check bit is associated to a check node*
- *Each* $h_{ij} = 1$ *indicate that the j-th bit of the codeword is considered in the i-th parity check equation*

Below an example of a Tanner graph is provided for a code C(5, 3) defined by the parity check matrix: $H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$ and for the information word $\underline{c} = [1, 0, 1]$:

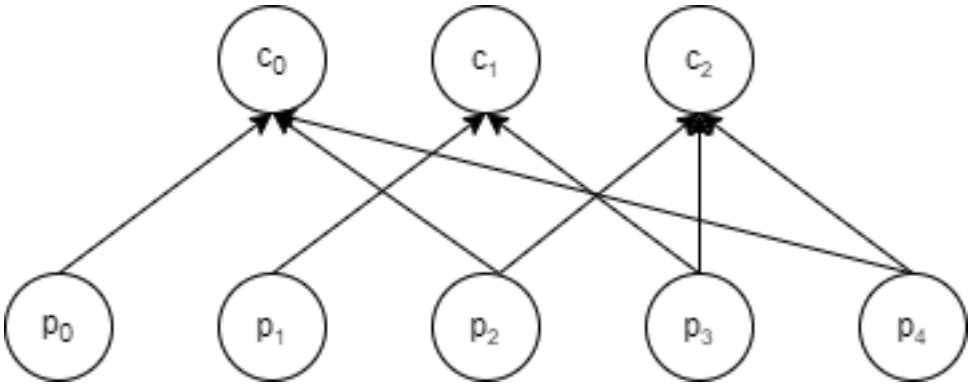


Figure 1.2: Tanner Graph of the code C(5,3)

Below the decoding algorithm for an LDPC is described, we suppose to have to decode the error affected codeword c_{RX} and that code used is described by the parity check matrix H.:

1. For each bit in b_i of $\underline{c}_{RX}$ associate the variable node V_i
2. Compute the initial syndrome $\underline{s} = \underline{c}_{RX}H^T$, if it is the null vector than the algorithm ends as it has received a codeword that have no detectable error (it can be still wrong if the error vector is it self a codeword).
3. For every node compute the UPC defined as $UPC = \underline{s}H$, each bit of the UPC is the number of failed parity check nodes whose associated syndrome have value 1 (or equivalently it is the number of connections of the syndrome bit not equals to 0).
4. Given some proper decision rule, flip some bits of the received codeword depending on the value of the UPC vector. A common choice is to flip the bits for which the UPC is maximum. Call this new codeword vector $\underline{c}^{(1)}$.
5. Recompute the syndrome $\underline{s}^{(1)} = \underline{c}^{(1)}H$, if it is 0 we end otherwise return to step 3 and iterate again until a null syndrome is achieved or the maximum number of iteration is reached.

This type of decode is called the bit flipping decoder (BF) and are employed when no channel information is known at priori. An important property of a decoder is the so called DFR, Decoder Failure Rate, that is the number of times that the decoder fail to correct the incoming information in the required number of iterations. Of course it is important to bound the maximum allowable number of iteration as otherwise the latency of the decoder would be too high for any application. Indeed in general the overall decoder speed scale with the number of iteration that are required.

1.4.1 QC-LDPC Codes

A particular class of LDPC codes are the QC-LDPC codes where the parity check matrix is a block matrix composed by r_0 x n_0 circulant matrix of size p x p. The overall matrix size is than n x k where $n = n_0 * p, k = k_0 * p$ and $r_0 = n_0 - k_0$. We consider in this thesis $r_0 = 1$ and $n_0 = \{2,3,4\}$ so for the given matrix the code rate is $C_r = (n_0 - 1)/n_0$. The H matrix is in the form:

$$H = [H_0|H_0|...|H_{n_0-1}] \tag{1.23}$$

where each H_i is a circulant matrix of size p x p. In general the encoding of such codes are not a problem as fast polynomial multipliers can be used, the big problem is the decoding procedures and in particular the calculation of the syndrome. To minimize the performances penalties a particular class of QC-LDPC are used in which the H matrix is sparse and short length cycles in the associated Tanner graph are avoided.

1.4.2 Application of QC-LDPC Codes

We now briefly discuss some application in were QC-LDPC codes have been adopted:

- 5G-NR: The 5G-NR standard required a low latency of 25ms and relatively high bit rate of at least 10Gbs in up-link and 20 Gbs in down-link. To reach this bit rate the systems work in the mm-wave range and use QC-LDPC codes. In this standard two base matrices are used and have a code rate of $C_{R1} = 1/3$ and $C_{R2} = 1/5$. To achieve small decoding latency, the decoders normally use a block parallel architecture and in this case the throughput is inverse proportional to the number of circulant blocks of the base matrix. In 5G the matrices are build in blocks and each block can be composed by sub matrices. The circulant part of the matrix is small size so the decoder can employ simple barrel shifter to perform the rotation needed to compute the vector by circulant operation.

- Wireless Applications: QC-LPDC have been adopted in recent standards such as the WinMAX (802.3an) and WiFi (802.11n/ac/ad). Although QC-LPDC codes are used, the decoder have been tailored by taking in consideration the statistics of channel medium. Such decoders are the: Weighted Bit Flipping (WBF), Modified Weighted Bit Flipping (MWBF) and Gradient Descending Bit flipping (GDBF).

- Cryptographic Applications: As we will see in chapter 2 QC-LDPC can be adopted in Cryptographic systems to reduce the size of the public private key. Old systems were using analytic codes such as Goppa Code [1]. QC-LDPC are used in the proposed post quantum computing Cryptographic systems such as the LEDA and BIKE system.

1.4.3 Cyclic Matrix Multiplication

An important operation that the LEDA and BIKE systems use is the the multiplication of a vector by a circulant matrix. We now discuss some basic property and possible implementation of this operation. Given a vector $\underline{v}$ and a circulant matrix C we define the vector by circulant multiplications:

$$\underline{R} = \underline{v}C = [v_0, v_1, ..., v_{N-1}] \begin{bmatrix} c_0 & c_{j-1} & c_{j-2} & \dots & c_1 \\ c_1 & c_0 & c_{j-1} & \dots & c_2 \\ c_2 & c_1 & c_0 & \dots & c_3 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ c_{j-1} & c_{j-2} & c_{j-3} & \dots & c_0 \end{bmatrix} \tag{1.24}$$

if we are considering a binary vector and matrix the operation simplifies quite a bit in particular if, as in LEDA, the matrices are sparse and contain a lot of zeros in the first row (col). In LEDA crypt the vector by circulant multiplication is done in $\mathbb{F}_2$ so the result is obtained by xor together all the product of the current row by column multiplication or is done in $\mathbb{Z}$ so normal integer addition is performed. In the case of binary operation the the generic algorithm follows:

Algorithm 1: VectorByCirculant Algorithm

Input: $\underline{\mathbf{v}}$: A binary vector with p elements, $\underline{\mathbf{C}}_p$: A integer vector of extacly D_V elements that is the index of the non null bit in the first row of the circualant matrix, **p**: The size of the vector V

Output: $\underline{\mathbf{R}}$: A binary vector that is the result of the multplication between $\underline{v}$ and the circulant matrix defined by $\underline{C}_p$

$i\_idx \leftarrow 0$;
$j\_idx \leftarrow 0$;
$\underline{R} \leftarrow 0_p$;
while $i\_idx \leq D\_V$ **do**
  $\underline{R} \leftarrow \underline{R} + shift\_circ(\underline{v}, \underline{C}_p[i\_idx])$;
  $i\_idx \leftarrow i\_idx + 1$;
end

The idea for computing the final result is to add at every iteration the result vector $\underline{R}$, initialized as the null vector, to a circulant shifted version of the input vector $\underline{v}$ by the amount specified by the index of the non null element of the first row of C. In general, as reported in [5], the vector by circulant operation can be viewed as a polynomial multiplication using the mentioned isomorphism (Prop. (1)), in this case the result is nothing more than the multiplication of $\underline{v}$ by a rotating vector depending on the position of the non null elements in $\underline{c}$. The algorithm to be implemented require at least two loops, one over all the element of index of the non null element and one for each bit in the input vector. So it has a complexity of $O(n^2)$, the performance did not scale well with the number of bits of the system.

Indeed when the number of bits is small the implementation in hardware is trivial as classical barrel shifter can be employed to perform the rotations. This approach is exactly what is used in 5G application where the code matrix is a large block matrix but the part that is circulant and requires this kind of multiplication is small, in the order of one hundred bits. Instead in the LEDAcrypt system, the size of the matrix is in ten thousands bits so barrel shifter cannot be used and custom architectures or different method to perform such multiplication are required. To have bigger keys, resulting in a increasing security, a better multiplication method is required and will be presented in chapter 3. The idea is to change this problem into the problem of calculating a convolution between two vectors and than using the FFT to reduce the overall operation complexity to $O(n * logn)$.

Chapter 2

Crypto Systems

Crypto systems are used to store securely information and relay in one way or the other in hard and complex computational problems to avoid the possibility of been exploited by malicious individuals. At an high level cryptography systems can be divided into two categories, one that contains the systems that encrypt information, and the other contains that systems that verify and validate the information. Modern cryptography system used what is call an asymmetric key schema that is the idea of breaking up the required information to decode, or to verify, the message into two pieces: a private key that is only known to the individuals that need to send/verify the secret information and a public key that is know to all of the individuals. Normally the public key are store in databases and there exists some mechanism to block particular type of keys that are found to be weak. We now discuss in more details some of the Cryptographic system that are used in today world:

RSA Systems The RSA system, developed by Rivest–Shamir–Adleman in 1977, is based on the hard problem of factoring a large number into its prime factors. Below the general method to send secure information is described:

1. Bob want to send to Alice some secret information, the message c, so he select two prime numbers t and l and compute it's product $n = t * l$
2. Bob than pick a number c, and compute d such that: $cd \equiv 1 \ mod \ (t-1)(l-1)$
3. Bob share with Alice the number c, t and l and made public the number d and n
4. Bob having the original message he convert it into an integer m and computes and send to Alice the new message: $\overline{m} = m^d \mod n$
5. Alice have received $\overline{m}$, to obtain the original message computes: $m = \overline{m}^c \mod n$. She than able to retrieve the original information from the

recovered numerical value.

Given d and n is extremely difficult to find c, t, l in a finite amount of time. This problem have also some deep links to some still open problems in number theory such as the Riemann Hypothesis for example, surely the most well known millennium prize problem.
This type of algorithms are normally slow so to be performed on big set of data so they are normally used only to secure the share of keys between two users and more modern systems such as AES is used to make the information secret.

ECC Systems Another important aspect regarding cryptographic is to validate the integrity of a given information. For example it is necessary to validate drivers of an operator system and validate software licenses. To solve these problems another family of cryptography system based on Elliptic Curves problems are used. Mathematically they are based upon problems regarding Elliptic Curves over finite fields. In particular the main idea is to perform the operations needed to validate the information using the algebra of the elliptic curves that present some interesting groups theoretical properties.
To have secure systems against attacks it is necessary to find some hard problems regarding elliptic curves. In particular a famous problem is the elliptic curve logarithm problem that can be easily stated as:

Prop. 4 (elliptic curve logarithm problem) *Given an elliptic curve, E, over a finite field $\mathbb{F}_p$ where p is a prime number, and two point P, Q (on E), find t such that $Q = tP$.*

it is still unknown if the problem is a NP or a NP-hard. One advantage of using ECC cryptographic system is that they require less space to store the keys as they are normally smaller than the equivalent RSA systems.

2.1 Quantum Computers and Algorithm Complexity

The mentioned systems are based on NP-problems to be secure against malicious attacks. NP problems are the family of computational problems for which it is easy to check the solution in polynomial time but there is no algorithm to find the solution in polynomial time. Conversely problem of type P are problems for which the solution can be found in polynomial time. A very deep conjecture, believed to be false, states that P = NP or that all problems which can be verified in polynomial times can also be solve in polynomial time (it is the first of the Millennium Prize problems states in 2000 by the e Clay Mathematics Institute). If proved this conjecture would have extremely impact in every day life as lot of different kind of systems are based on the wrongness of this conjecture.
The theory behind the study of algorithm complexity, was developed when the existence of quantum computer was only theoretical, so an interesting problem is determine if the mentioned family of problems when executed on quantum computer still maintain the same complexity properties. If a quantum computer can solve NP problems in polynomial time than the security of cartographic systems are in dangerous and a new class of even harder problems have to be developed in order to systems that are secure against all types of attacks.

Def. 2 (BQP Problems) *The bounded-error quantum polynomial(BQP) problems are the class of computational problems that a quantum computer can solve in polynomial time, with a correct answer probability greater than* 2/3*.*

There is no general proof that NP problems are inside BQF, only one NP problem, the integer factorization problems, has been proven in 1997 by P.Q. Shor, to be inside BQP. A quantum computer can factorize an integer using the Shor's algorithm. If it is possible to find problems that are completely outside the BQP class than they would still be secure against a quantum computer attacks.

Def. 3 (NP-Hard Problems) *The class of computational problems that are outside of BQF are called NP-hard problems*

Some interesting NP-hard problems regards random codes and are used in some of the proposed post quantum computer cryptography systems. These problems are:

- **Syndrome Decoding Problem** (SDP): Decoding a codeword of a given random linear code
- **Codeword Finding Problem** (CDP): Found the bounding weight codeword of a generic random linear code

these problems were found to be NP-hard in [6]. More generally the problem of computing the minimum distance of a random linear code has been found to be NP hard in [7].
Beside codes another family of NP-hard problems used for secure system is the family of problems regarding lattices such as the short lattice vectors and close lattice vectors problem that are the base of other NIST proposal for post quantum computer cryptography.

2.2 McEliece and Niederreiter Cryptosystems

Two systems were developed based on above mentioned NP-hard problems to create the secure systems: the McEliece system and the Niederreiter system.

McEliece Cryptosystem Developed by Robert McEliece in 1978 it exploits the hardness in decoding random linear codes. Originally the system employed Goppa Codes, that is a analytic code, but that results in longer keys so in modern systems implementation they had been substituted with QC-LDPC codes. Bob and Alice want to share a some secret information:

1. Bob randomly picks a secret code $C_s(n,k)$ able to correct up to t errors, and generate the generator matrix $G = G_{n,k}$. He also chose other two matrices: S a dense k x k matrix and a permutation matrix P. He subsequently compute the public key: $G' = SGP$. Bob shares the key publicly.

2. Alice want to send the 1 x k binary string $\underline{u}$ = "Hello" to Bob so she take the Bob public key G' and obtains the encrypted message $\underline{x} = \underline{m}G' + \underline{e}$ where $\underline{e}$ is a random 1 x n error vectors with hamming weight less or equals to t (as is the max number of error that the original code $C_s(n,k)$ can always correct). Alice than sends to Bob the message $\underline{x}$.

3. Bob wants to decrypt the received message $\underline{x_R}$ so first he compute the error affected codeword: $\underline{x}' = \underline{x}_R P^{-1} = \underline{u}SG + \underline{e}P^{-1}$. Bob knows that he can correct it using the private code $C_s(n,k)$ using the syndrome technique. He compute the message $\underline{u}' = \underline{u}$ S where S is the syndrome matrix of the secret code. Bob finally can find the original message, $\underline{u} = \underline{u}'S^{-1}$.

Niederreiter Cryptosystem Developed in 1986 by Harald Niederreiter is an evolution of the McEliece systems and employs syndromes and parity-check matrix instead of the generator matrix of the code. Again, as usual, Bob wants to sends to Alice the message $\underline{m}$ = "Hello".

1. Bob generates a random linear code $C_s(n,k)$, able to correct $t \geq 1$ errors, and from it computes its parity check matrix H_s. He also picks another matrix S of dimension r x r. Bob than generate the public key $H' = SH$, and shares it publicly.

2. Alice wants to send the text message $\underline{m}$="Hello", so she generates, given $\underline{m}$, $\underline{e}$ a 1 x n error binary vectors with exactly t asserted bits and she send the message $\underline{x} = H'\underline{e}^T$ that is exactly the syndrome of the original message via the public key parity check matrix. To generate $\underline{e}$ Alice can use different type of algorithm, for example she can use a constant weight encoder described in detail in [10]. The idea is to associate at every character a given probability

and build a run of bits from it by creating a so called combinatorial number system..

3. In order to to obtain the original message, Bob first computes the syndrome of the incoming message using his private parity check matrix H to obtain: $\underline{s}_p = S^{-1}\underline{x}_R$, from this using some syndrome decoding algorithm is able to obtain back the error original vector and than to correct the errors in the incoming message in order to obtain the original message. (We note that normally a decoder already correct the errors so in that case the final operation is not required)

2.3 The LEDACrypt System

The two previous mentioned systems have the advantage to be secure against quantum computer attacks but due to the fact that they are using Goppa codes they have drawback in term of key size and hardware implementation complexity. Some new type of systems have to be developed in order to reduce the key size and make them viable to be used in systems. One of those systems, developed to participate in NIST Post quantum computer cryptographic competition, is the LEDAcrypt system. It substitutes the Goppa codes with QC-LDPC codes and increases the decoding speed by introducing a tailored version of the classic BF decoder, called the Q-decoder.
In the next section we discuss the LEDApkc crypto system, based on the McEliece crypto system, even though an equivalent version, called LEDAkem, exists for the Niederreiter crypto system. More information LEDAkem can be found in the LEDAcrypt specification paper. The private key is a random QC-LDPC code. We consider a code $C_s(n,k)$ with codeword length: $n = p * n0$ and information word length of $k = p(n0-1)$, $r = n - k = p * r0$ where p is a prime number and n0 $\in \{2,3,4\}$. Both the private and the public key are matrices of size p x p. To generate both private and public keys first two matrices are generated, one is the code parity check matrix, H, of the code $C_s(n,k)$, of size p x $p * n0$ and a $p * n0$ x $p * n0$ quasi cyclic sparse binary matrix Q. H is decomposed in 1 x n0 circulant blocks each of size p x p: $H = [H_0, H_1, ..., H_{n0-1}]$ and each block is selected such that it has an odd number of asserted bits indicated as d_v. The Q matrix is a n0 x n0 matrix made by p x p circulant blocks, the hamming weight of each block form another circulant matrix in which each row/col sum to the same value m. The choice of the weight are made to guaranteed that Q is always invertible. Below an

example of the mentioned matrix is given:

$$Q = \begin{bmatrix} Q_{0,0} & Q_{0,1} & \dots & Q_{0,n0-1} \\ Q_{1,0} & Q_{1,1} & \dots & Q_{1,n0-1} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{n0-1,0} & Q_{n0-1,0} & \dots & Q_{n0-1,n0-1} \end{bmatrix} w(Q) = \begin{bmatrix} m_0 & m_1 & \dots & m_{n0-1} \\ m_{n0-1} & m_0 & \dots & m_{n0-2} \\ \vdots & \vdots & \ddots & \vdots \\ m_1 & m_{n0-1} & \dots & m_0 \end{bmatrix}$$

The prime number p is chosen accordingly with the LEDA specification (in order to guarantee that Q and H are invertible).
The private key, S_{key}, is the pair $\{H, Q\}$ and the public key, P_{key}, is generated by first compute the matrix: $L = HQ = [L_0|L_1|...|L_{n0-1}]$ and subsequently the matrix: $M = L_{n0-1}^{-1}L_0 = [M_0|M_1|...|M_{n0-2}] = [M_l|I_p]$. The public key is the matrix $P_{key} = [Z|M^T]$ where is a diagonal matrix made by $n_0 - 1$ copies of the identity p x p circulant block. To reduce the space needed to store the matrices, instead of storing the entire circulant matrix we only store the position of the ones in the row of each block of the matrix (as they completely describe the entire matrix block). The size of the private key, if the mentioned method is used, is equals to $Dim(S_{key}) = n0(d_v + m)\lceil \log_2(p) \rceil$. To generate the matrix H,Q the idea is to use a deterministic random number generator (DRNG) which seed is generated by a true number generator (TRNG) to generate the position of the non null bits of each block of H and Q. To summarize a LEDA system key generation process is described by the set of parameters:

- $\underline{m} = [m_0, m_1, ..., m_{n0-1}]$
- $m = \sum_{i=0}^{n0-1} \underline{m}_i$
- p is a prime number, p > 2 such that $ord_2(p) = 1 - p$
- $n0 = \{2,3,4\}$
- d_v

A proper selection of these parameters set the system security level and the size of the code and will be discussed in the next chapter. Now we describe how the system is able to encrypt information. Again we suppose that Bob wants to send Alice some secret information. Bob has already generated using the method described before the private key S_{key} and have shared publicly the public key P_{key}.

1. Alice wants to send to Bob a text message described by a binary vector $\underline{m} = [m_0, m_1, ..., m_{n0-1}]$, she randomly picks a vector $\underline{e} = [e_0, e_1, ..., e_{n0-1}]$ from the bucket of all the possible error vectors of weight t (that is the maximum number of errors that the code $C_s(n, k)$ can correct).

2. Alice than obtains the encrypted message $\underline{x} = \underline{e} + \underline{m}$ M where M represent the public code of the system and is defined as: $M = L_{n0-1}^{-1} L_0$

3. Bob receives the message $\underline{x}_R$, he computes given the secret matrices H and Q the matrix $L = HQ = [L_0, L_1, ..., L_{n0-1}]$ that represent a parity check matrix of the secret code. Using this matrix Bob computes the message syndrome $\underline{s} = L\underline{x}_R^T$. Bob than uses Q-Decoder to detect and correct the errors in $\underline{x}_R$ and obtain the original message. $\underline{x} = Q\_DECODER(\underline{s}, H, Q, iter_{max})$

To increase the security level of the LEDApkc system,the Kobara-Imai transformation [9] is suggested in the specification to achieve IND-CCA2 security level. Performing this transformation allows to employ, instead of the parity check matrix, a systematic generator matrix G. This transformation yield to a smaller public key size and in a speedup of the encryption procedure. This procedure is employed to hide the fact that given the public key it is possible to recovering the original message without knowing the private key by recovering the embedded information word from the encrypted message $\underline{x}_R$. This thesis we will only analyze the basic system and in particular the interest in the speedup of a key operation of "Q-decoder" that will be described later.

2.3.1 LEDAcrypt Code Parameters

As explained in the previous section the LEDA system is represented by a set of parameters that need to be designed in order to reach the required security level. In general the LEDA parameters can be generated according to algorithm 17 in [3], starting from the required security levels and the desired DFR of the code. The final system DFR needs to be evaluated using Monte Carlo simulation to be sure that it is similar to the design requirement because there is no known formula that given the system parameters evaluates the real DFR of the system.
The mentioned algorithm (???) takes as input:

- $\boldsymbol{DFR_{min}}$: The minimum required DFR.
- $\boldsymbol{\lambda c}$, $\boldsymbol{\lambda q}$: the required security level expressed as the log base two of the required computational effort needed by a classical and quantum computer to break the code
- **n0**: the number of circulant blocks of the code parity check matrix

and produce the system parameters:

- p: the dimension of each block of the circulant matrix.
- d_v, m: The hamming weight of the rows of H and Q respectably.
- t: The max number of error that the code $C_s(n, k)$ can correct.

- $iter_{max}$: The maximum number of the decoder iterations.

λc and λq set the NIST category level and represent the security level of the systems against attacks, in term of number of operation need to break an equivalent AES system.
Below an example of code parameters, for different value of n0 and category level is provided and is taken from the LEDA specification documentation:

NIST Category	**n0**	**p**	**t**	d_v	**m**	**Decodes Errors**
1	2	14939	136	11	[4,3]	14 out of 1.2e9
	3	7853	86	9	[4,3,2]	0 out of 1e9
	4	7547	69	13	[2,2,2,1]	0 out of 1e9
3	2	25693	199	13	[5,3]	2 out of 1e9
	3	16067	127	11	[4,4,2]	0 out of 1e9
	4	14341	101	15	[3,2,2,2]	0 out of 1e9
5	2	36877	267	11	[7,6]	0 out of 1e9
	3	27437	169	15	[4,4,3]	0 out of 1e9
	4	22691	134	13	[4,3,3,3]	0 out of 1e9

Table 2.1: LEDA System Parameters: Table 3.1, Pag56 [4]

As it can be seen in the table generally if the same security level need to be maintained than the value of p is inversely proportional n0 so overall the size of the code is not changing as the n0 determine the number of circulant blocks that the code has.

2.3.2 The Q-Decoder

The Q-Decoder is a modified version of the classical B.F decoder tailored for the use with the LEDA structure of public and private key. The decoder exploits the fact that the position of the ones in the expanded error vector, $\underline{e}'$ are influenced by the value of Q^T because $\underline{e}'$ is obtained from the random error vector $\underline{e}$ multiplied by Q^T. The decoder inputs are:

- The syndrome vector, $\underline{s0} = \underline{x}_R L = \underline{x}_R HQ$
- The maximum allowable iteration: $iter_{max}$
- The H and Q matrices

The output is a 1 x $n0 * p$ error vector and a flag indicating if the decoding have succeeded. The decoding algorithm start with $s^{(0)} = s$, $\tilde{e}^{(0)} = 0_{n0}$ and executes the steps:

Data: $\underline{x}$: The encoded message of size n0*p, **Q, H**: The private key pair,
IT_{max}: The maximum number of iteration of the decoder
Result: $\underline{x}_R$
$\underline{\tilde{e}} = 0$;
$l \leftarrow 0$;
$L \leftarrow H * Q$;
$\underline{s} \leftarrow \underline{x} * L^T$;
$s_{sum} \leftarrow w_h(s)$;
while $l \leq IT_{max} \&\& s_{sum} ! = 0$ **do**
  // This two multiplications are performed in the integer domain
  $\underline{\sigma} \leftarrow \underline{s}H$;
  $\underline{\rho} \leftarrow \underline{\sigma}Q$;
  $b \leftarrow \max \underline{\rho}$;
  $\underline{F} \leftarrow \{\underline{\rho} = b\}$;
  // Find the bit flipping position and flip the bits
  $e_{tmp,j} \leftarrow \underline{F}_j, j \in [0, n0 - 1]$;
  $\underline{\tilde{e}} = \underline{\tilde{e}} \oplus \underline{e}_{tmp}$;
  // Update the syndrome and its sum, binary multiplication.
  $\underline{s} \leftarrow \underline{s} + \underline{\tilde{e}}L$;
  $s_{sum} = w_h(\underline{s})$;
  $l \leftarrow l + 1$;
  if s_{sum} == *0* **then**
    break;
  end
end
$\underline{x}_R \leftarrow \underline{x} + \underline{\tilde{e}}$;

Given the basic algorithm, a speedup is possible by exploiting a LUT table to skip the calculation of the maximum value of b^l. The LUT is built as an ordered pair $\{w_j, b_j\}$ generated from the system parameter according to the LEDA specification. (An example of LUT values, for different code parameters, is provided in [8] Table 3, pag 333). Given the LUT, instead of computing $\max b^{(l)}$ directly, given $\tilde{w}_s^{(l)}$ find the biggest entry in the LUT such as $w_j \leq \tilde{w}_s^{(l)}$ and set $b^{(l)} = b_j$. This look-up make possible to avoid finding the maximum of the $\underline{\rho}^{(l)}$ vector that require to loop over all the p entries of the vector and is computational expensive. In general it is also possible to modify the above algorithm to perform the correction of $\underline{x}_R$ at every iteration. To do so the position of the bits to flip in x_R are given by the computing $\underline{fPos} = \rho^{(l)} \geq b^{(l)}$.
The algorithm require, at every iteration, at least 3 vector by circulant multiplication. Two are done considering H and Q that are sparse matrices (the number of one in each column is d_v) and one instead is done via a medium sparse matrix, L. Indeed the computation of the syndrome is one of the most critical part of the algorithm especially if we consider a system with greater security level that require either a big p or an higher number of circulant blocks n0.

Chapter 3

My System

As explained in the previous section at every iteration of the Q-Decoder it is necessary to compute at least 3 vector by circulant matrix multiplication order to compute either the syndrome or the position of the bit to flip. These operation are very important for the overall decoder speed so it would be interesting to see if some optimization can be applied to speed up the decoder. To perform such multiplication a natural choice would be to use classical polynomial multipliers based for example on the Karatsuba fo example. The main problem that such systems have is that in the case of LEDA the number of bit of the incoming vector is fixed by p so it is in the thousands. Traditional Karatsuba algorithm have a complexity in the order of $O(n^{log_2 3})$ so it not scale linear when n is bigger. We would like to find solution for which the operation complexity scales more linearly. May be a good solution is to find an optimization in order to scale as $O(n * log(n))$. In the next sections we first present an idea to optimize this multiplication by employing the FFT and than we present the developed system.

3.1 More on Circulant Matrices

The idea to improve the performance of the vector by circulant operation is based on the following lemma:

Lemma 1 *Every circulant matrix C, of dimension NxN is identified by the elements of the first column $\underline{c}$ and it is diagonalizable via the Fourier transform. We have that:*

- $C = F^* * \lambda * F$
- $\lambda = diag(F * \underline{c})$

F is the DFT matrix of order N, defined by Eq.(1.3) matrix, F^* is the conjugate matrix obtained by calculating ω_n inverse (as it is its conjugate). The main motivation of the lemma is that there is a link between the eigenvectors and eigenvalues of a circulant matrix and the the roots of unity on the imaginary plane. Using the same notation as the above lemma for ω and $\underline{c}$ we have that:

- The eigenvectors are: $u_j = (1, \omega, \omega^2, \omega^3, ..., \omega^{N-1})$, $j = 0,1, ..., N-1$
- The eigenvalues are: $\lambda_j = c_0 + c_{N-1}\omega^j + c_{N-2}\omega^{2j} + ...c_1\omega^{(N-1)}, j = 0,1, ..., N-1$

From the previous two definition we note that if re-index the second formula, using Eq.(1.6) with $l = \phi_N(j)$:

$$\lambda_l = c_0 + c_1\omega^l + c_2\omega^{2l} + ...c_{l-1}\omega^{l-1} = \sum_{k=0}^{N-1} c_k\omega^{kl} = \sum_{k=0}^{N-1} c_k e^{-j\frac{2\pi}{N}kl} \tag{3.1}$$

that is exactly the definition of the Fourier transform of the l element of the incoming vector. We can now state the second lemma that is the key to obtain a more optimized method to calculate the Vector By circulant operation:

Lemma 2 *Supposing we are calculating the multiplication between a vector $\underline{r}$ and a circulant matrix C represented by the element of its first column in the vector $\underline{c}$, than the product $\underline{V} = \underline{r}\ C$ is equivalent to:*

$$\underline{V} = IDFT(DFT(\underline{r}) \odot DFT(\underline{c})) \tag{3.2}$$

where the function DFT is the discrete Fourier transform and the IDFT is the inverse Fourier transform of order N. The $\odot$ is the element by element vector product also called the Hadamard product.

3.2 FFT

In general to optimize the DFT algorithm two strategy can be employed: one is to transform the original one dimensional Fourier transform into a two dimensional transform that is more easy to be implemented in hardware, the other is to change the original Fourier transform into a small convolution. The first method is employed in James Cooley and John Tukey FFT algorithm developed in the '60s. This method work when the number of samples N is a power of 2. In this case the computational complexity is reduced from a $= O(n^2)$ to a $O(n * log(n))$. The second strategy is employed in the work by Rader that in 1968 developed the Rader FFT algorithm [11] which allowed to compute the FFT of a sequence of p element, where p is a prime number and have computation complexity of still $O(n * log(n))$.

3.2.1 Cool-Turkey algorithm and the Butterfly Architecture

Returning on the problem of calculating the FFT $\bar{X}$ of a vector X of length N=2^l l > 0, the main idea is to split the summation into two sums one for the even and one for the odd terms:

$$\begin{aligned} \overline{X}_n &= \sum_{k=0}^{N-1} X_k e^{-j\frac{2\pi}{N}kn} = A_n + B_n \\ A_n &= \sum_{k=0}^{N/2-1} X_{2k} e^{-j\frac{2\pi}{N}2kn} \\ B_n &= \sum_{k=0}^{N/2-1} X_{2k+1} e^{-j\frac{2\pi}{N}(2k+1)n} \end{aligned} \tag{3.3}$$

we can now simplify this equation by employing symmetry properties of the complex exponent function and reduce the number of operation needed. In the next section we describe a particular type of decomposition that yield to the very well know butterfly architecture. We start by recalling the decomposition obtained in Eq.(3.3) and we note that given that N $= 2^l$ we can simplify the above relations to obtain this two new expression for A_n, B_n:

$$\begin{aligned} A_n &= \sum_{k=0}^{N/2-1} X_{2k} e^{-j\frac{2\pi}{N/2}kn} \\ B_n &= e^{-j\frac{2\pi}{N}n} \sum_{k=0}^{N/2-1} X_{2k+1} e^{-j\frac{2\pi}{N/2}kn} \end{aligned} \tag{3.4}$$

We introduce now the so called twiddle factor, defined as:

$$W_N^k = e^{-j\frac{2\pi k}{N}} \tag{3.5}$$

It is easy to verify that the following relation, useful in later calculations, hold:

$$\begin{aligned} W_N^k &= W_{N/2}^{2k} \\ W_N^k &= W_N^{k \mod N}, \forall k > N \\ W_{N/2}^k &= W_N^{2k} \end{aligned} \tag{3.6}$$

Using this factors we can rewrite Eq.(3.4) as:

$$\begin{aligned} A_n &= \sum_{k=0}^{N/2-1} X_{2k} W_{N/2}^{nk} \\ B_n &= W_N^n \sum_{k=0}^{N/2-1} X_{2k+1} W_{N/2}^{nk} \end{aligned} \tag{3.7}$$

We easily see that the twiddle factor are no more than points around a circle of radius 1 so they present some symmetry property that depends on N and the k.

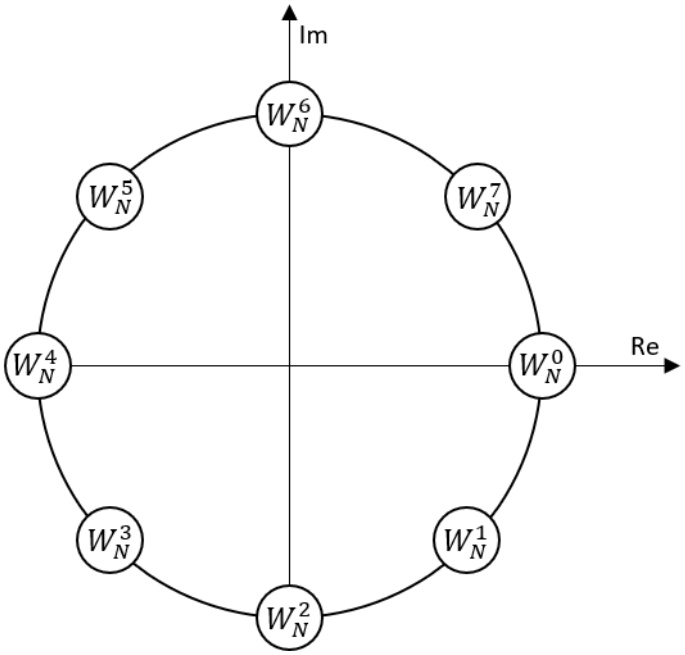


Figure 3.1: Twiddle Factor, N=8

Recalling the definition we gave for the circulation matrices in the previous section, we note that the twiddle factors are the same as the ω variable. Indeed we have this relation:

$$W_N^k = \omega^k \tag{3.8}$$

We now can finally use the identity $\omega^{n+N/2} = -\omega^n$ due to the periodicity of the complex exponent function. This relation allow us to find a recursive equation between the $\bar{X}_n, \bar{X}_{2n+1}$ and the A_n, B_n that is the basis of the Butterfly algorithm. An easy calculation show that:

$$\begin{aligned}
\overline{X}_n &= A_n + B_n \\
\overline{X}_{n+N/2} &= A_{n+N/2} + B_{n+N/2} \\
&= \sum_{k=0}^{N/2-1} X_{2k} e^{-j*\frac{2\pi}{N/2}k(n+N/2)} + e^{-j\frac{2\pi}{N}(2n+N/2)} \sum_{k=0}^{N/2-1} X_{2k+1} e^{-j\frac{2\pi}{N/2}k(2n+N/2)} \\
&= \sum_{k=0}^{N/2-1} X_{2k} e^{-j\frac{2\pi}{N/2}kn} e^{-2j\pi} + e^{-j\frac{\pi}{N}n} e^{-j\pi} \sum_{k=0}^{N/2-1} X_{2k+1} e^{-j\frac{2\pi}{N/2}kn} e^{-2j\pi k}
\end{aligned} \tag{3.9}$$

The above formula can be simplified knowing that $e^{-j2\pi} = 1$, $e^{-j\pi} = -1$ and using the expression of the twiddle factors and the relation in Eq. (3.6) we get that:

$$\overline{X}_{n+N/2} = \sum_{k=0}^{N/2-1} X_{2k} W_{N/2}^{nk} - W_{N/2}^{2n} \sum_{k=0}^{N/2-1} X_{2k+1} W_{N/2}^{nk} = A_n - B_n \tag{3.10}$$

Therefore we see that there is recursive relation between X_n, $X_{2n+N/2}$ and the twiddles factors. We also note that we can store only half of the twiddles factors as the others would always have the same expression but with a negative sign. To summarize the calculation, we found that:

$$\begin{aligned}
\overline{X}_n = A_n + B_n &= \sum_{k=0}^{N/2-1} X_{2k} W_{N/2}^{nk} + W_{N/2}^{2n} \sum_{k=0}^{N/2-1} X_{2k+1} W_{N/2}^{nk} \\
\overline{X}_{n+N/2} = A_n - B_n &= \sum_{k=0}^{N/2-1} X_{2k} W_{N/2}^{nk} - W_{N/2}^{2n} \sum_{k=0}^{N/2-1} X_{2k+1} W_{N/2}^{nk}
\end{aligned} \tag{3.11}$$

where A_n are the even elements of X and B_n are the odd terms.
For example we now calculate using the above equations the FFT of a signal with N=4 samples:

$$\begin{aligned}
W^n &= W_{N/2}^n = e^{-j\frac{2\pi}{4/2}} = e^{-j\frac{\pi}{n}} \\
\underline{W} &= [1, -j, -1, j] \\
\overline{X}_0 &= (X_0 W^0 + X_2 W^0) + W^0 (X_1 W^0 + X_3 W^0) = w_h(\underline{X}) \\
\overline{X}_1 &= (X_0 W^0 + X_2 W^1) + W^2 (X_1 W^0 + X_3 W^1) = (X_0 - jX_2) - (X_1 - X_3) \\
\overline{X}_2 &= (X_0 W^0 + X_2 W^2) + W^4 (X_1 W^0 + X_3 W^2) = (X_0 - X_2) + (X_1 + jX_3) \\
\overline{X}_3 &= (X_0 W^0 + X_2 W^3) + W^6 (X_1 W^0 + X_3 W^3) = (X_0 + jX_1) - (X_1 + jX_3)
\end{aligned}$$

We note that if the input vector is a binary vector then the result will only be a combination of twiddle factors, and from the equation of the $\overline{X}_0$ it is easy to see that it is just the hamming weight of the incoming vector (the number of non null bits in the vector).
So in general we found the expression of the single block of the FFT architecture. We fix N and analyze the k bit of the result vector, than we can rewrite the relation where s is the stage in which this circuit is used, in Eq. (3.4) as:

$$\begin{aligned} A_s &= A_{s-1} + B_{s-1} W_{N/2}^{k} \\ B_s &= A_{s-1} + B_{s-1} W_{N/2}^{k+N/2} \end{aligned} \tag{3.12}$$

We see that here we calculated the result of the first samples. The main idea of the butterfly architecture is that we can reuse these same circuits, with different twiddle factors to compute the entire samples of the signal. The result is a cascade of this single computation module. This "core" module require 2 complex multipliers and two adders. The below image represent the mentioned core module, and we see that the butterfly name come from the fact that the connection between A, B seem a "butterfly".

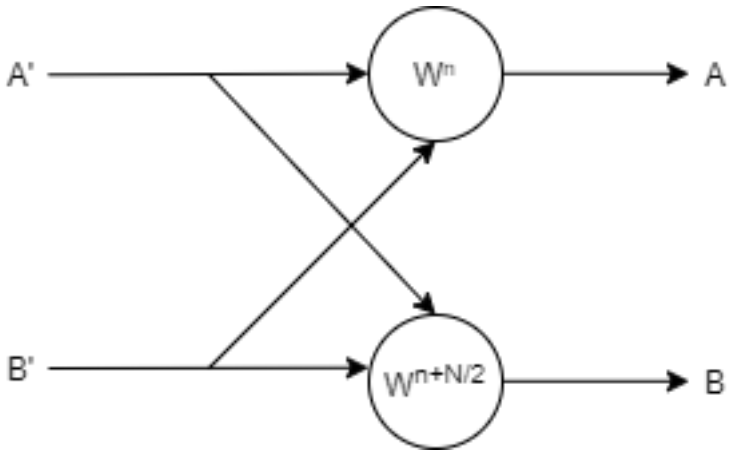


Figure 3.2: Butterfly Base Block

The idea is that this core module generate given samples 0,1 the samples 0, 3, sample 2, 3 generate samples 2 and 6 and so on. By putting together more than one of this module in cascade it is possible to compute the FFT of the incoming vector. At any stage the FFT is done on N/2 samples, so suppose we have 8 samples we would have 3 stages where the first is done with N=4, the second with N=2 and the last with N=1.

The image below explains this situation and can be generalized for bigger transform.

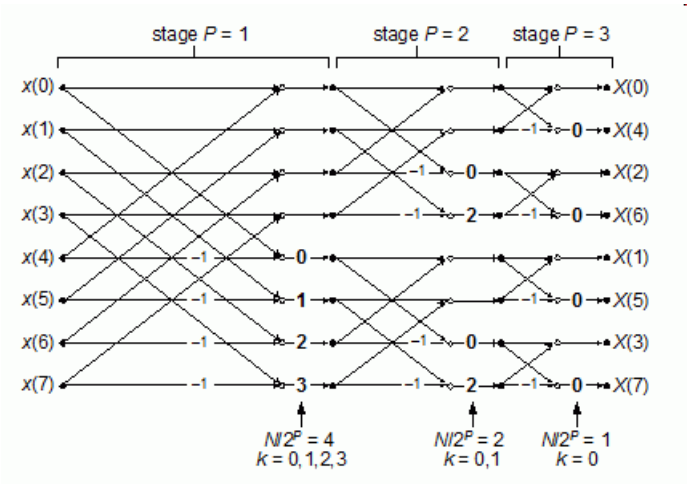


Figure 3.3: 8 sampls FFT

We see that the order of the output samples is not matching with the input order. In fact they are ordered in a bit reverse manner. The idea is that given a binary number, the bit reverse number is the same the binary number but read backward from the right to the left. The following table reports the numbers and their reverse for binary numbers with 3 bits.

$A\|_{10}$	$A\|_2$	$R\|_{10}$	$R\|_2$
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Using the same idea of splitting the initial sum into different terms, we could implement the FFT for higher radix representation. For example if we want to obtain the radix 4 FFT we would split the initial sum into 4 terms in the following

way:

$$\begin{aligned}
\overline{X}_n &= \sum_{k=0}^{N-1} X_k * e^{-j\frac{2\pi}{N}kn} = A_n + B_n + C_n + D_n \\
A_n &= \sum_{k=0}^{N/4-1} X_{4k} e^{-j\frac{2\pi}{N}4kn} \\
B_n &= \sum_{k=0}^{N/4-1} X_{4k+1} e^{-j\frac{2\pi}{N}(4k+1)n} \\
C_n &= \sum_{k=0}^{N/4-1} X_{4k+2} e^{-j\frac{2\pi}{N}(4k+2)n} \\
D_n &= \sum_{k=0}^{N/4-1} X_{4k+3} e^{-j\frac{2\pi}{N}(4k+3)n}
\end{aligned} \tag{3.13}$$

and using the same idea of the previous section we can exploit the symmetries of the twiddle factors to simplify these formulas. We obtain again another basic circuit that can be reused multiple time to obtain the finale FFT result. In particular the radix 4 butterfly is shown below:

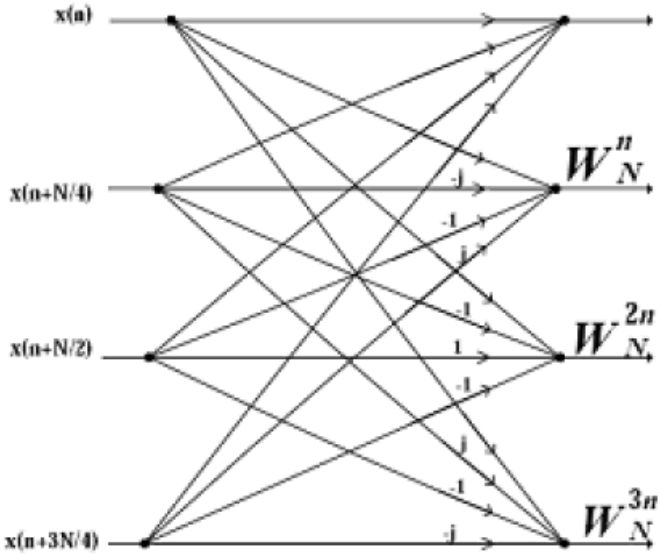


Figure 3.4: Radix 4 Butterfly unit

More in general the following relation can be found for a FFT of any arbitrary radix r.

$$\begin{aligned} n &= rl + i \\ \overline{X}_k &= \sum_{l=0}^{N/r-1} \sum_{i=0}^{r-1} x_{rl+i} W_N^{ik} W_N^{rlk} \end{aligned} \tag{3.14}$$

When the FFT is generalized to an higher radix, it is transformed into a two dimensional FFT that is more efficient to implement in hardware. On the other side increasing the system radix results in high hardware complexity as multiple samples need to be read from memory at the same time. Another solution to speed up the FFT is to introduce pipelining in order to reduce the critical path of the system. This solution can be a trade off between speed and complex hardware circuits.

3.2.2 Rader FFT Algorithm

In this section we provide a small introduction to the Rader FFT method as it introduce the need for a particular type of padding that allow to use the normal FFT, with an extended sequence, to the LEDA/BIKE system optimized polynomial multiplication.
The main idea of the Rader algorithm is the fact that when p is a prime number the sequence of length $N = p$ of indices 1,2, ..., $p - 1$ form a cyclic group, of order N, under multiplication modulo N. Due to some properties of number theory such group always has a generator g, which is a number such that all of the p-1 elements can be obtained from it. Given this generator the general DFT formula 1.1 can be rewritten as:

$$\overline{X}_{g^{-t}} = \overline{X}_0 + \sum_{k=0}^{N-2} X_{g^t} e^{-j * \frac{2\pi}{N} * g^{-(t-k)}} = \overline{X}_0 + \sum_{k=0}^{N-2} f_k g_{t-k}, t = 0,1, ..., N-2 \tag{3.15}$$

$\overline{X}_0$ is the hamming weight of the incoming vector, the sum is a cyclic convolution and can computed using the Fourier domain. In the case that N-1 is a power of two than traditional FFT algorithm can be applied otherwise it is necessary to use the Rader FFT algorithm recursively. This is a problem if the number of samples are big as it would increase the overall system latency. It is important to find a proper padding method, proposed in [12], to obtain two new sequences of length $M = 2^l$ for which classical FFT algorithm can be applied. To do so we extend the two original sequences f,g to two sequences f' and g' of length $M \geq 2N - 3$ defined as:

- f' is obtained by adding M - N null bits between the first and last element of the original sequence. $f' = [f_0, 0, 0, 0, ..., f_1, f_2, ..., f_{p-1}]$

- g' is obtained by cyclically repeating g to a size M. $g' = [g_0, g_1, ..., g_{p-1}, g_0, ..., g_{p-l}]$

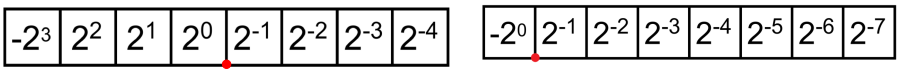
To use classical FFT algorithm we choose M to be the closest power of two respect to 2N although this choice is not ideal as it requires extra bits respect the minimum condition but allow to use classical Radix2 Cool-Turkey algorithm for example.

3.2.3 NTT Transform

The last interesting possibility is the Number Theoretical Transform(NTT) which is built upon some number theory arguments and allow to make possible to perform all the transform operations in the integer domain. We now briefly describe how it works. The main idea is to replace the twiddle factors, $W_N^k = e^{-j\frac{2\pi k}{N}}$, with the k-th root of unity thus performing the transform in the ring $\mathbb{Z}/p\mathbb{Z}$ instead of the $\mathbb{C}$ field. It is possible to show that a similar architecture to the Cooley-Turkey can be obtained in which the complex multipliers are replaced by module N circuits. This architecture is more complicate to implement in hardware but allows to avoid the use any floating/fixed point numbers thus eliminating any precision errors performed during the operation. Normally the NTT is performed in software due to it's hardware implementation complexity. Only recently due to the use of the transform in another family of post quantum computer cryptographic system, the lattice based, optimized hardware implementation have been studied and developed.

3.3 Fixed point Representation

We recall that a FP representation is a way to represent non integer numbers. Given a selected number of bit we write a FP number with the notation QX.Y where X are the total number of bits and Y represent the integer part of the value and Y are the number of bits of the non integer part of the value. For example Q16.15 indicate that we are dealing with a number with 16 bits and 15 of those bits are used for the fractional part of the number. The image below describe how a number is represented in fixed point format, the red dot is the position of the fractional point.



(a) Q8.4 number (b) Q8.7 number

Figure 3.5: Fixed point examples

For any FP the resolution is given by:

$$\epsilon = 2^{-Y} \tag{3.16}$$

that is the minimum distance between two FP numbers. Normally we would like to set the resolution and from it defining the number of fractional bits, that are given by:

$$Y = \lceil log2(\frac{1}{\epsilon}) \rceil \tag{3.17}$$

It is convenient to normalize the value with respect to a given scaling factor such as every number is in the range $[-1.0, 1.0)$. If the same scaling factor is used than it is possible to implement the required operation using standard mathematical circuits (adders, multipliers, ecc). If the scaling factor differs than it is necessary to adding additional logic to normalize the two number into the same scaling factor and than perform the required operation. Overall the main problem of using a FP number is that the precision is finite. When the scaling is applied, if the obtained number does not fit in the given resolution, it is necessary to perform a proper rounding method to obtain a number that is representable in the given format. There are different kind of rounding methods available such as:

- **Truncation**: This is the straightforward method if we have a number that is not representable in the given precision; we just consider the number with the max available number of fractional digits. For example, we suppose we are working with numbers in the form Q4.3 and we are considering the number l=0.111111, we pick the number $l' = 0.111$, we basically consider 3 digits after the dot. This method is the worst as it always introduces an error.

- **Round to the nearest**: In this case depending on the value of the fractional part we either round to the lower value or to the next value by adding one. To do this we decide the threshold of 0.5, if the value is greater or equals than 0.5 we add 1 to the integer part of the number otherwise we do nothing. For example, we consider the number 2.34 the rounded result would be 2, if we consider 2.75 the result would be 3. This method is better than the truncation but it introduce a bias in the result, in fact in the case the number have a fractional part of 0.5 we always ceil the result.

- **Round to the nearest even**: We do the same as before but if the value is 0.5 we randomly select if we round or ceil the result This method is the best as it reduces the bias of the result. The main problem is that it is necessary to have a random generator that should have no correlation with the value. This is not a trivial task of for example the circuit that generate the value and the random number generator uses as the base the same clock signal.

3.4 AXI4 Interface

Most of the IPs that Xilinx provide, use a particular type of interface, derived from the ARM memory interface called AXI4 to allow the exchange of data and

configuration information.
The AXI4 interface is compose of at least:

- **tdata**:this bus is used to send/receive information and it is composed by a number of bits that are a multiple of 8. so if the data is greater than 8 bit is sign extended to the an 8 bit alignment. For example a signal of 33 bit signals is extended to 40 bit
- **tready**: it is asserted to one by the core to indicate that it is ready to receive a new data on the tdata bus if the AXI4 interface is used as in input or by the system when it is ready to accept any output data.
- **tvalid**: is asserted to 1 by the system if the AXI4 interface is used as a input and indicate that a valid data is present in the tdata bus. The signal is asserted to one by the IP when the AXI4 interface is used as an output and again indicate that a valid data is present on the bus.
- **tlast**: indicates that the current value on the tdata bus is the last data of the signal. It is used, for example, to indicate the last frame of the FFT.

Below a time diagram of a typical AXI4 interface is shown:

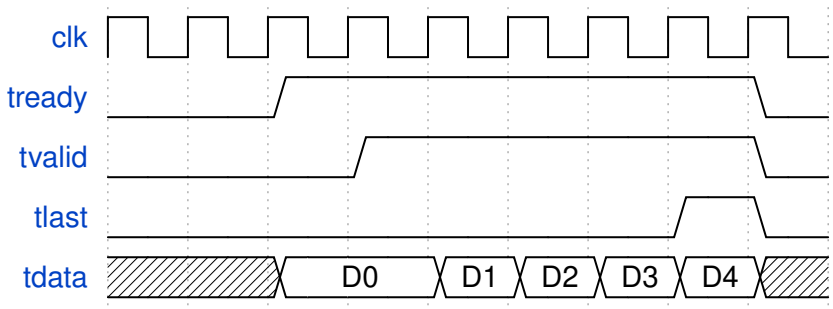


Figure 3.6: AXI4 Time Digram

Additional another signal could be present in the interface, called, tuser that are used to exchange information about the current output/input data to the system. This is used for example for the FFT IP in order to output the current FFT sample index

3.5 The big system picture

The method described in [2] requires the use of 3 FFT, two needed to compute the incoming vectors and one needed to convert back the result and a complex multiplier. Some memory might be required if the FFT output is not in natural order. The input vectors are binary vectors and the result vector is binary. The first problem to consider is which is the best architecture to use for the FFT?
During this thesis the idea was trying to build the system using already built FFT IP to verify if the system is feasible and synthetizable in a FPGA. For the thesis the Xilinx FPGAs was selected and Vivado was used as the IDE tool to both write VHDL code and to compile and synthesize it. We now describe each used IP in more details

3.5.1 FFT IP

The Xilinx FFT IP, called xfft, have the following RTL schematics:

It can be configured with different hardware architectures that can be selected when the IP is generated. It is possible to chose from:

- Pipelined, Streaming I/O Architecture
- Radix-2, Burst I/O
- Radix-2 Lite, Burst I/O.

The input width can be selected between 8 up to 33 bits.
In general the pipelined architecture yields to the best performance but it occupies more space on the FPGA. As for almost all of the Xilink IPs this module use the AXI4 interface for data and configuration. It present three major ports:

- **In Port**: It is used to load a new sample inside the core. Any sample is composed by a real part and one imaginary part. The imaginary part occupies the lower bits of the frame and the real part the higher bits. If the number of bits is not a multiple of 8 than the real and imaginary part is signed extended to next multiple of 8. A frame has size = 2*InWidth, where InWidth is the input parallelism (signed extended to the next multiple of 8 when necessary).

- **Config port**: it is used to configure the core. The core can be configured in different way depending on the condition in which it need to work. The configuration frame contains configuration information like the type of transform to perform either FFT or the IFFT and the scaling factor that the IP applies at

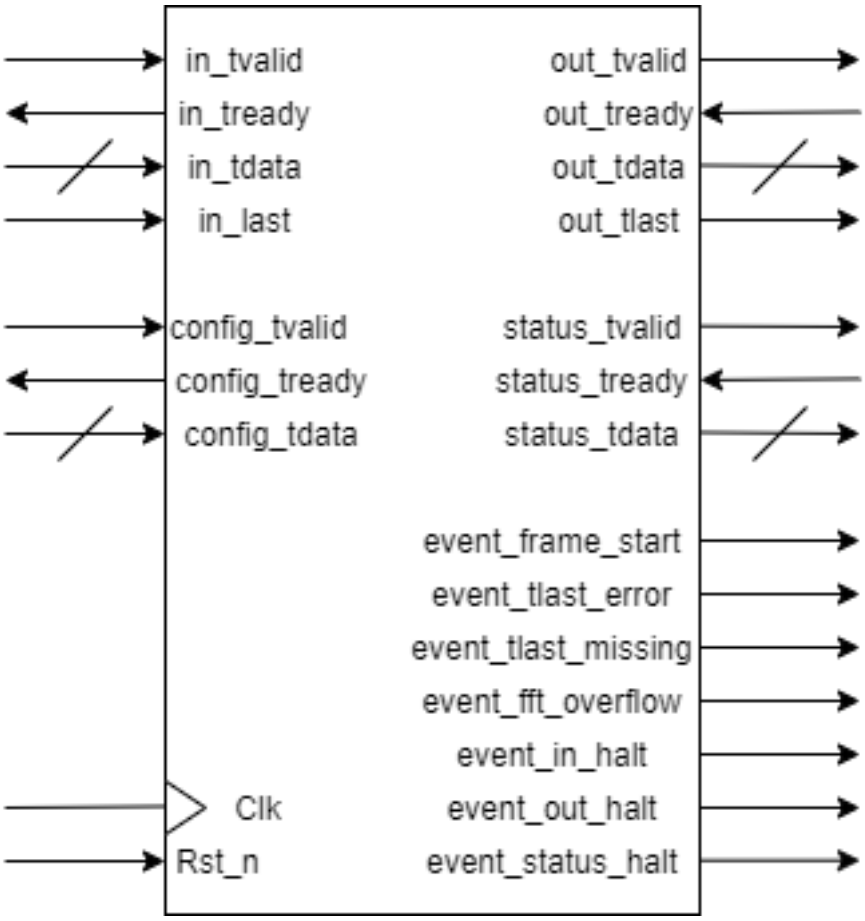


Figure 3.7: xfft RTL

every stage of the architecture. The scaling value is obtained by the following equation:

$$S = \sum_{i=0}^{log2(N-1)} 2^{b_i} \tag{3.18}$$

where b_i are the scale at any stage. The datasheet suggest, to reduce the overflows in the calculation, to have the maximum scaling in the last stage of the FFT. In the developed architecture the scaling factors are SCALING_SCH = [10 10 10 11] when N=5, where we apply a scaling of 1 in the early stages and a scale of 2 in the final stage. When the IP is used to perform the inverse transform, the IFFT, the scaling factors are all set to 0 because the scaling is already present in the transform. Using these scaling factors no additional hardware is required and it is possible to reduce the datapath parallelism.

- **Out Port**: it is used to output the FFT result. The output parallelism depend upon the IP configuration. In fact the datasheet specified that:

  - $CompWidth = InWidth + log2(N) + 1$ when the core is configured to use full precision

  - $CompWidth = InWidt$ when the core is configured to use scaling

  When the core is used with scaling, the output is a FP number in the form $QN.(N-1)$ where N is the size of the input real and imaginary part. The total output parallelism are given by:

$$FFT\_OutWidth = 2 * CompWidth \tag{3.19}$$

  because each FFT out sample is made by both a real part and imaginary part that represent, in reality, the phase of the sample. In both situation the component output is a fixed point number that maintains the same number of fractional part bits. The worst situation happens when the binary input vector, of length L, has all the elements equals to one. In this case the max output value would be exactly L in the case of the unscaled FFT or 1/L when using the scaled normalized output. This is because we recall, from the theory of the FFT, that the first sample of an FFT is always the hamming weight of the incoming vector.

- **Status port**: it is used to provide information about the current frame state. Indeed when the IP is configured to do so, it can provide the sample index and bit to determine if the calculation of the current frame resulted in an overflow. We recall, from the theory of the FFT, that normally the FFT sample order is not the natural order but it follows the so called bit reversal order. The IP did not perform any reordering by default to avoid additional memory and latency on the overall transform.

The IP finally can be configured to have multiple data channel and the FFT of multiple channels. When this mode is used the input and output data is interleaved so that the same AXI4 interface can be used for all the channels. Also the core can change the transform size at runtime. In this case the number of samples of trhe transform are set using the AXI4 configuration port.

3.5.2 Complex Multiplier IP

The Xilinx Complex multiplier IP, called cmpy, have the following RTL schematics:

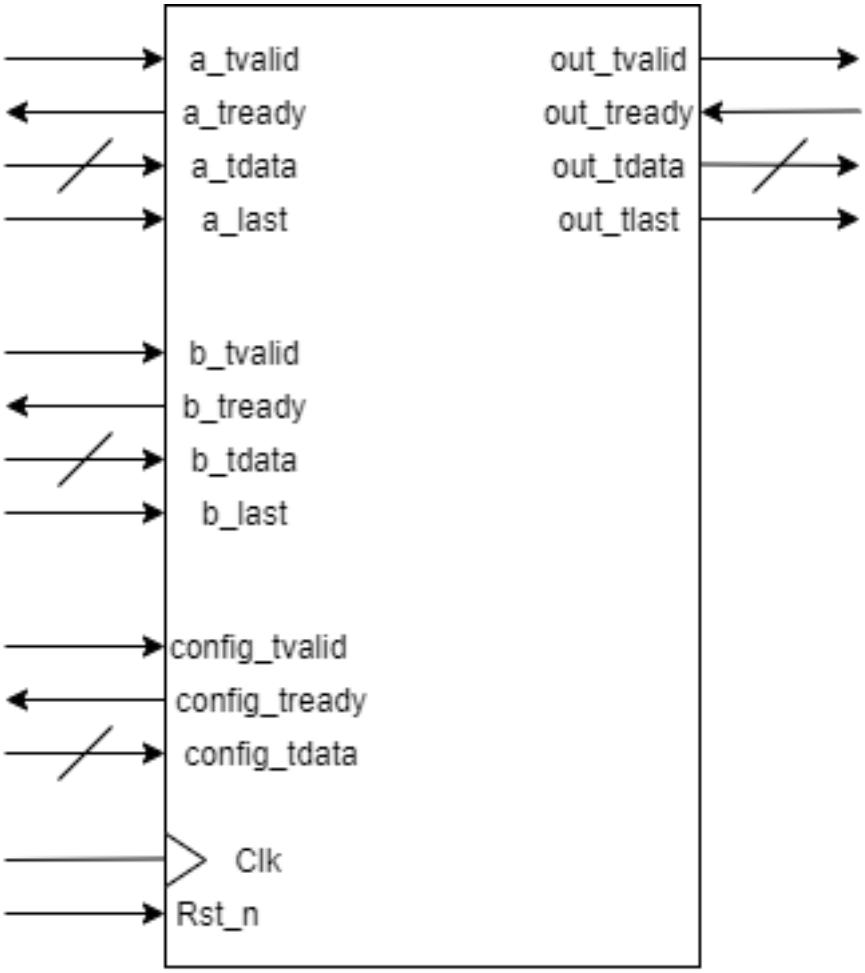


Figure 3.8: cmpy RTL

This IP as the FFT use the AXI4 interface to obtain data and configuration information. It has the following ports:

- **A/B Port**: This ports are used to provide the two samples that will be multiplied. The sample are made from a real and imaginary part where the low bits are the imaginary part while the high bits are the real part. An additional tlast signal is provided in order to indicate if the given sample is the last. In the designed system the samples are always coming correctly and are synchronized, but this IP can work either in blocking mode, where the two samples need to be present, or non blocking mode where the IP always perform the multiplication.

- **Config Port**: it used to configure the convergent rounding. config_tdata is an 8 bit interface for which only one bit is used, bit 0 and is needed to inform the IP about the convergent rounding. The IP, when configured to use the convergent rounding, reduce the output width by 1 bit but requires this configuration signal to perform the rounding. The IP performs a Round to the Nearest even and uses this bit to determine whatever to ceil or floor the result when the Value is exactly in the middle of the range. The datasheet suggest to drive this signal from FF connected to the Clock signal. It is obvious that doing so can result in correlation, that will produce biasing in the result. The proposed solution is a good drawback but doing simulation of this system it was found that applying the rounding led to not visible differences in the overall result therefor no rounding is applied by the IP core.

- **Out Port**: This port is used to output the result of $A * B$. The IP can be configured to use either 3 or 4 multipliers to compute the final result. Depending on the selected configuration, the IP calculate the output using these equations:

  – 4 Multiplier Solution:

$$\begin{aligned} out_{re} &= (A_{re} * B_{re}) - (A_{im} * B_{im}) \\ out_{im} &= (A_{re} * B_{im}) + (A_{im} * B_{re}) \end{aligned} \tag{3.20}$$

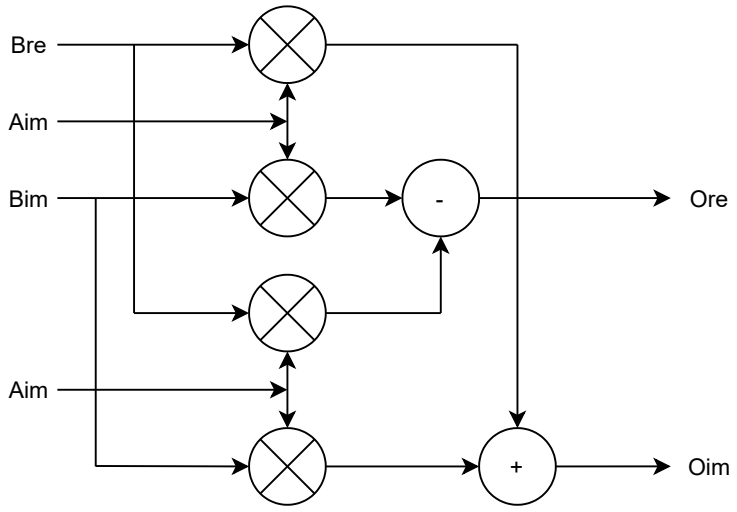


Figure 3.9: 4 Mul Complex Multiplier RTL

– 3 Multiplier Solution:

$$\begin{aligned} out_{re} &= A_{re} * (B_{re} + B_{im}) - B_{re} * (A_{re} + A_{im}) \\ out_{im} &= A_{re} * (B_{re} + B_{im}) + B_{im} * (A_{im} - A_{re}) \end{aligned} \tag{3.21}$$

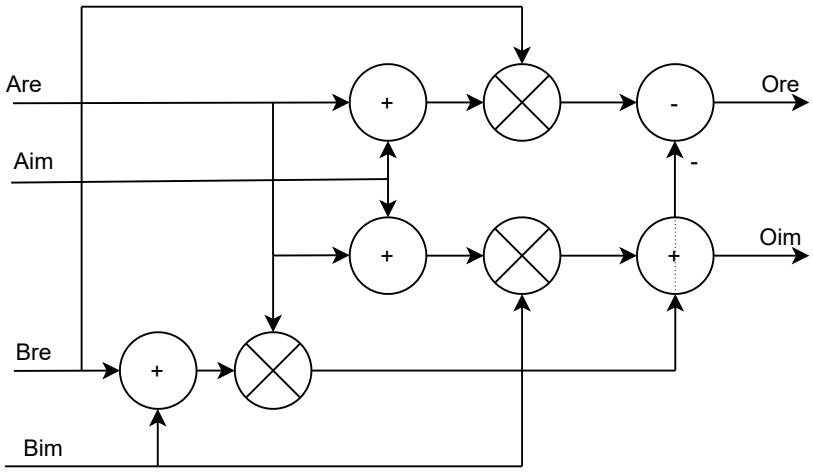


Figure 3.10: 3 Mul Complex Multiplier RTL

The multiplier IP have no specification regarding the input type so it work for both unsigned, signed, and fixed point numbers as expected because when multiplying two FP numbers no additional logic is needed (a part from rounding) if both have the same scaling factor. The output parallelism are given by the following equations and depend on the configuration:

– Full Precision: CompWidth = (2*InputWidth + 1)

– Convergent Rounding: CompWidth = 2*InputWidth

In the developed architecture, the complex multiplier output is always a FP number in the form $Q33.31$. The total output parallelism are given by:

$$XCMUL\_OutWidth = 2 * CompWidth \tag{3.22}$$

because, again, the output sample is made by both a real part and imaginary part, with the same parallelism. Optionally the IP can be configured to include the tlast signals inside the AXI4 bus. This is useful to detect when the core has completed all the required multiplications. The out_tlast is generated by doing the logical AND between the a_tlast and b_tlast.

3.5.3 Memory IP

When bit reverse ordering is selected in the IP module configuration, an external memory is required in order to reorder the sample before the final IFFT transform. The memory used in the design is a BRAM generated using the Xilix memory generator and is asynchronous in writing and synchronous with a delay of one clock cycle in reading. The width is set to 66 bits (33 for the real part and 33 for the imaginary part) and have a depth equals to the the number of samples of the system. BRAM memory has been selected because it is faster than distributed memory and allow the system to work at an higher frequency. The main draw back is that an FPGA with enough BRAM resources is required to be able to perform the synthesis.
The BRAM IP has a straight forward interface and it does not have the AXI4 bus so no additional logic is required. It has the following RTL schematics:
The BRAM has the following ports:

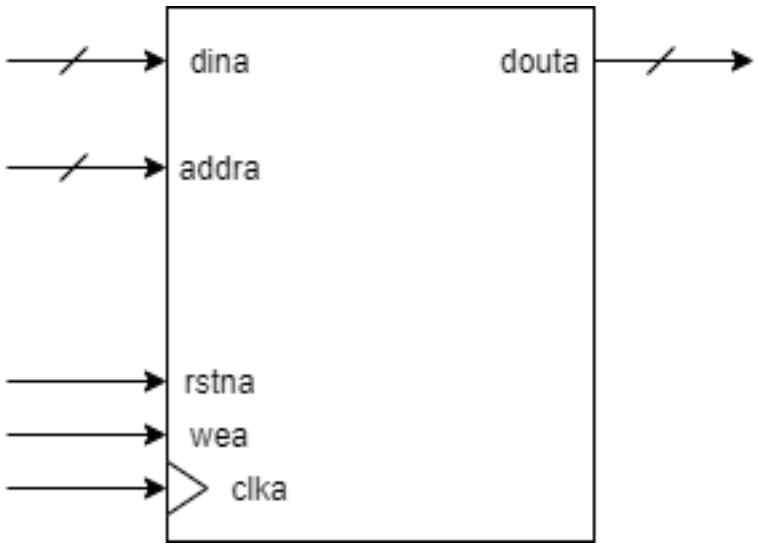


Figure 3.11: blk_mem RTL

- **dina port**: It is the the input data port
- **addra port**: It is the memory read/write address
- **rstna port**: It is the the memory content reset, if it is asserted to 1 it reset the content of the memory.
- **wea port**: It is the memory write enable, when it is asserted a new data is store in the memory cell pointed by the current value of the addra port. The memory is configured in such a way that the data that was in the location pointed to the current address is latched out of the memory.

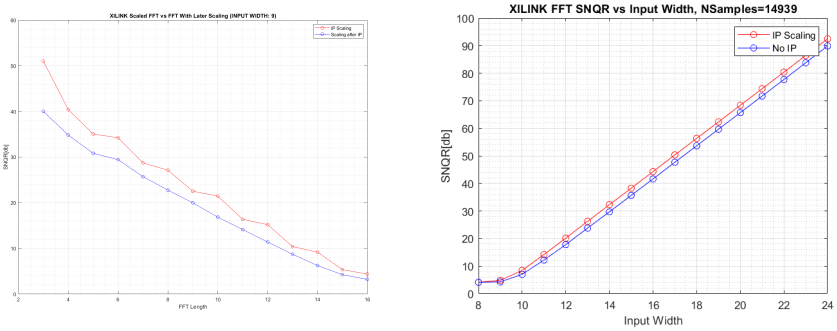
- **douta port**: It is the memory output port.

3.6 System Design And Simulation

Having described all of the building blocks of the system, now the task is to figured out which is the best parallelism to use. It is necessary to select the proper value for the input parallelism, that is the number of bits of the real and imaginary of the xfft IP core. In theory to perform the FFT of a binary stream only one bit is required but the Xilinx IP requires 8 bit as the minimum input bit number. The IP also requires the input value to be represented as 2's complement fixed point number. It turns out that the number of input bits has a big impact on the total number of errors that the developed circuit with respect to the reference implementation. This is due to the fact that the circuit works using a fixed point representation that introduces precision errors that can be modeled as a quantization noise. This noise can be expressed as a ratio respect to a referenced non quantization case, and is called SNQR (Signal to noise quantization ratio). This noise is present for all of the FFT samples and results in errors at the output of the convolution circuit. Additional errors are also due to the rounding of the different signals in the datapath. The SNQR, is defined as:

$$SNQR_{db} = 10 * \log_{10}\left(\frac{\sum_{n=0}^{N-1} R^2(n)}{\sum_{n=0}^{N-1}(R(n) - X(n))^2}\right) \tag{3.23}$$

where we consider two signals: one $\overline{X}(n)$ contains the FFT samples obtained using the fixed point Xilinx IP core and a second signal $\overline{R}(n)$ contains the FFT samples obtained using the MATLAB fft function that use IEEE754 floating point representation. Below a graph of the SNQR for the xfft IP is showed:



(a) SNQR vs Num of samples, InWidth=18

(b) SNQR vs Input Width, NSamples=14939

Figure 3.12: xfft SNQR simulations

We clearly see that the there is an influence of the input width to the overall noise level of the system, when we consider a system with less bits than the noise is more

relevant. In general, using the builtin IP scaling configuration yields to better performances. Overall the SNQR noise results in errors at the output of the circuit and so it is necessary to find the best condition to avoid as much errors as possible. The choice of the input FFT number of bits determine the overall system parallelism as it defines input number of bits of the complex multiplier input and ultimately the IFFT input width. To understand better the problem, a MATLAB software model was developed. This model use the provided Xilinx Matlab function for the FFT and implement via MATLAB the complex multiplier is implemented using the 3 multiplier architecture. The model is represented as a single MATLAB function that takes as the input:

- a_data: It is a binary vector that represent the data message.
- b_data: It is the position of the non null element on the first row of the circulant matrix
- p: represents the length of the data message.

The script when p is not a power of two applies the required padding to be able to use the classical FFT algorithm. The output of the function is a binary vector that is the result of the multiplication of the data vector, a_data, by the circulant matrix defined by b_data. For the simulations of the system, two functions where created one called FFT_VecyByCircIP employs the Xilinx IP to perform the FFT operation and one called FFT_VecyByCircMAT that uses the builtin MATLAB fft function. The code of FFT_VecyByCircIP is provided in the appendix (A.1). First the system was simulated with a number of samples that is a power of two to verify that the system worked as expected in the best situation as no padding is necessary. The simulation script perform the multiplication operation and compares the result with the reference vector by circulant multiplication. After both results are computed, the scripts compute the number of differences between the two binary vectors (that is the difference in their hamming weight). For every length the system is simulated 10 times with random input data and the average of the errors are calculated. The simulation results are shown in the below plot:

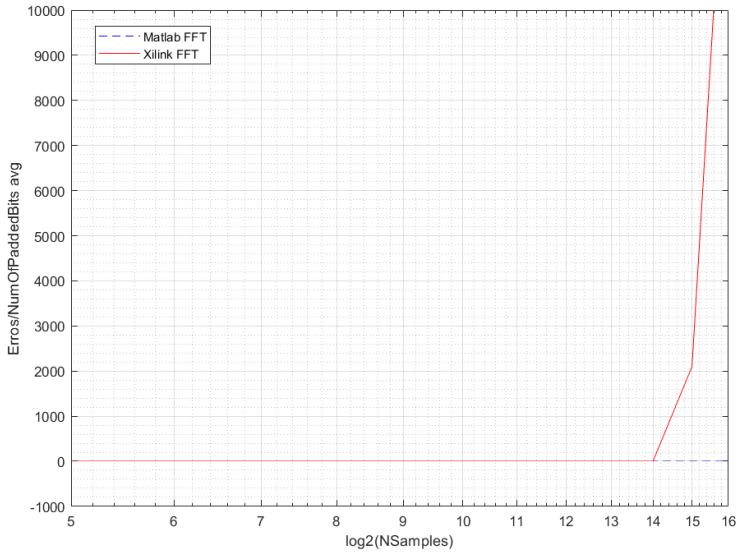


Figure 3.13: Power of two length Simulation

No errors are present until a frame length of 2^{14}. This is due to the fact, as already explained above, 16bit are not enough for the input parallelism of the FFT module due to quantization noise. The main problem is that the maximum number of input bits for the IFFT IP is 34, the complex multiplier at the output has 2N+1 bits and the two input FFT have N bit for input and output. This situation set a limit at maximum number of bit at the input of the FFT IP that is:

$$\#Bits_{fft}^{in,MAX} \leq 16 \tag{3.24}$$

This situation becomes even more problematic if we consider that for the LEDA/Bike system the input vector is not a power of two. In this case it is necessary either to use more complex FFT algorithm such as the Rader Algorithm, described in 3.2.2, or padding can be applied to reach the required length that is a power of two. When the padding is used the initial prime sequence is extended to at most the nearest power of two respect to 2p. This means that, when we use padding, the maximum p is 8191. Looking at the LEDA system parameters table, 2.1, we see that there exists some solution for which p is less than the maximum value but have an higher number n0 is required, that yields to a bigger key length and overall decoding latency. To better show the problems that occurs when the samples number, p, is greater than 8192 a simulation is carried out in which for all the primes in the range 7000 to 8500 the system is simulated using the same method as before. The following graph is obtained the red curve is system that uses FFT_VecByCircIP function while the blue curve is the system that uses FFT_VecByCircMAT function:

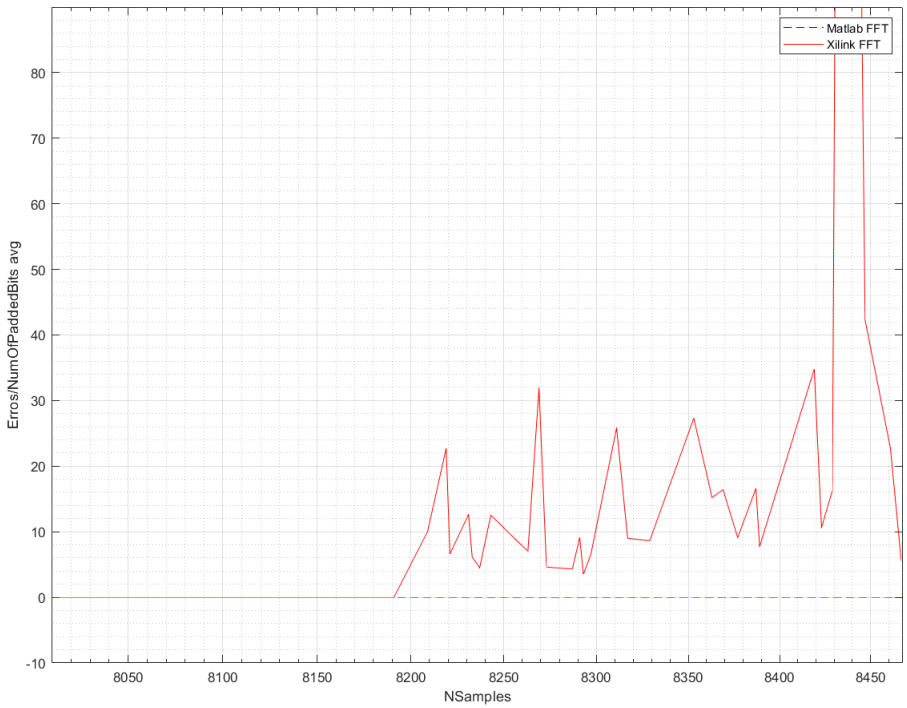
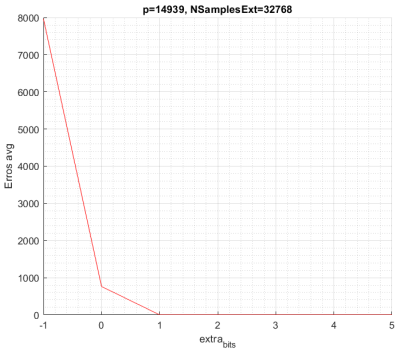
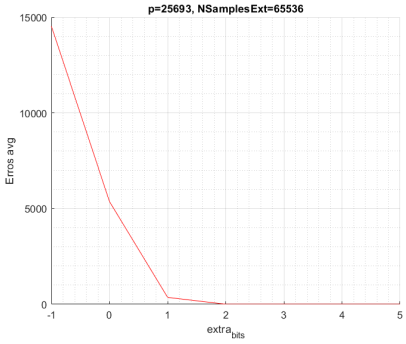


Figure 3.14: Prime Padding Simulation

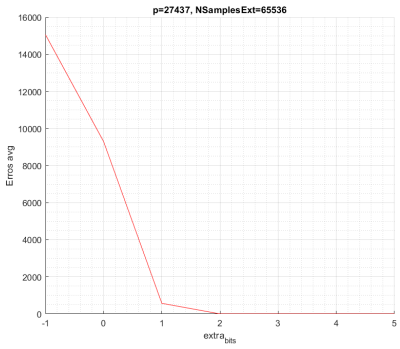
The blue curve is always flat as when the MATLAB FFT function is used no quantization errors are present.
To overcome the problem a clever trick can be used, the input FFT width is extended by some number of bits and than the additional bits are truncated from the result of the complex multiplier. This solution requires no change in the datapath parallelism as the size of the BRAM still remains at 33bit. Simulations shows that this solution always works, when a proper number of additional bits is selected, and allows to perform bigger size FFTs and realize LEDA/BIKE systems with a reduced key size. To understand which is the best value of extended bits to use, a new simulation is performed where the system is simulated at a fixed p and the the values of extra bits are changed. The simulation is performed for the different categories of LEDA systems:



(a) Category=1, n0=2, p=14939



(b) Category=3, n0=2, p=25693



(c) Category=5, n0=3, p=27437

Figure 3.15: Best Num of extended bits

Setting the extra number of bits to at least 2 allows to never have error up until a transform of size 2^{16}. The provided Xilinx IP does not support transform sizes greater than 2^{16} so no simulation was performed for higher transforms. Now that the proper value of the extended bits is selected, it is possible to simulate again the entire system to verify that it has no more errors for transforms bigger than 8191 samples.

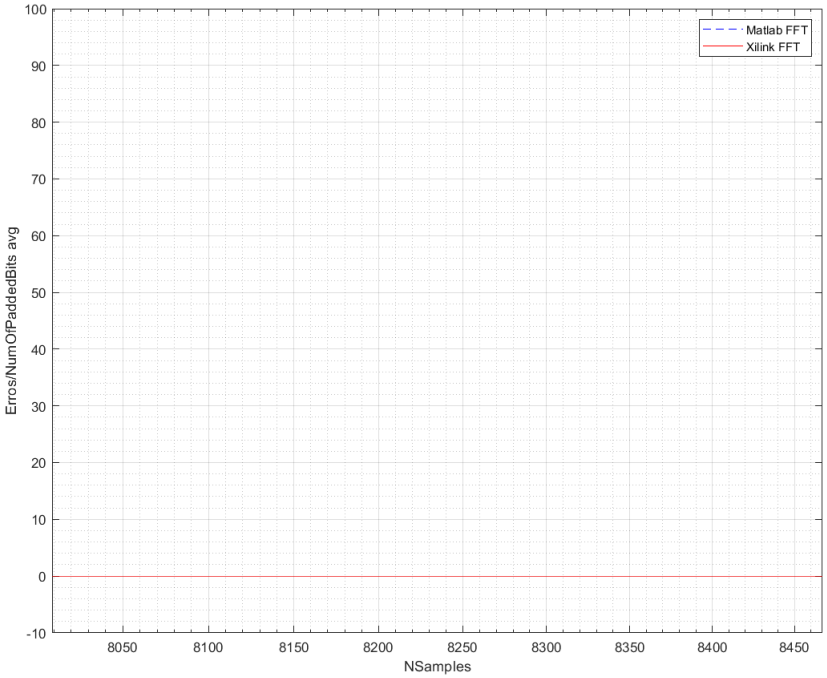
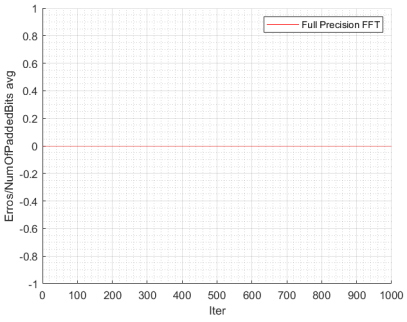
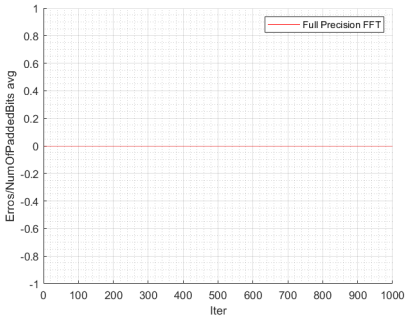


Figure 3.16: System with Input Extended bits Simulation

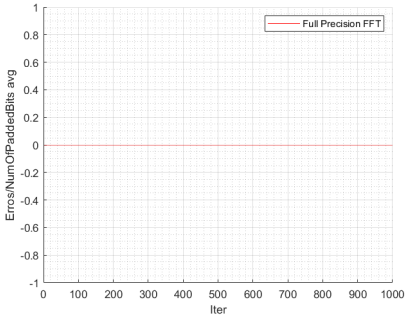
We can see that now the system is correct when the transform length is greater than 8192 in contrast to what happen for the previous simulation 3.14.
All the previous simulations were carried out in the worst case scenario, as the number of asserted bits in b_data weren't bounded to d_v as it is in LEDAcrypt specification. Now it is possible to simulate a more realistic system. We perform the simulation for the same values as in 3.15 but now we consider proper sequences for b_data. We still let a_data be generated randomly and we increase the number of iterations. Below the simulation results are shown:



(a) Category=1, n0=2, p=14939



(b) Category=3, n0=2, p=25693



(c) Category=5, n0=3, p=27437

Figure 3.17: Best Num of extended bits

It is possible to see that the proposed system still works under the proper LEDA specification condition so it is now possible to design the RTL of the circuit. To summarize we found the following parameters for the datapath:

- Number of samples for the FFT: $NSamples_{ext} = NextPow2(2p - 3)$
- FFT Sample width: $FFT^{IN}_{width} \geq 18$
- Rounding mode set to convergent for the three xfft IPs, and to truncation for the complex multiplier IP.

3.7 System RTL Description

The entire design is divided into mainly three blocks, one is the main multiplier that contains the FFT IPs and is the "real" RTL of the design, second if padding is required two additional modules are inserted before the main multiplier RTL block called a_padder and b_padder. The circuit has as main inputs two memory interfaces for the message and position memory and an output memory interface that contains the output multiplication result. A serial to parallel shift register (SIPO) is present in order to sequence the binary output of the main RTL into the Nb bits of each row of the output memory. A simple FSM, not described, is present to menage the output memory saving and generate the start signals for the APadder,BPadder and main RTL circuit.

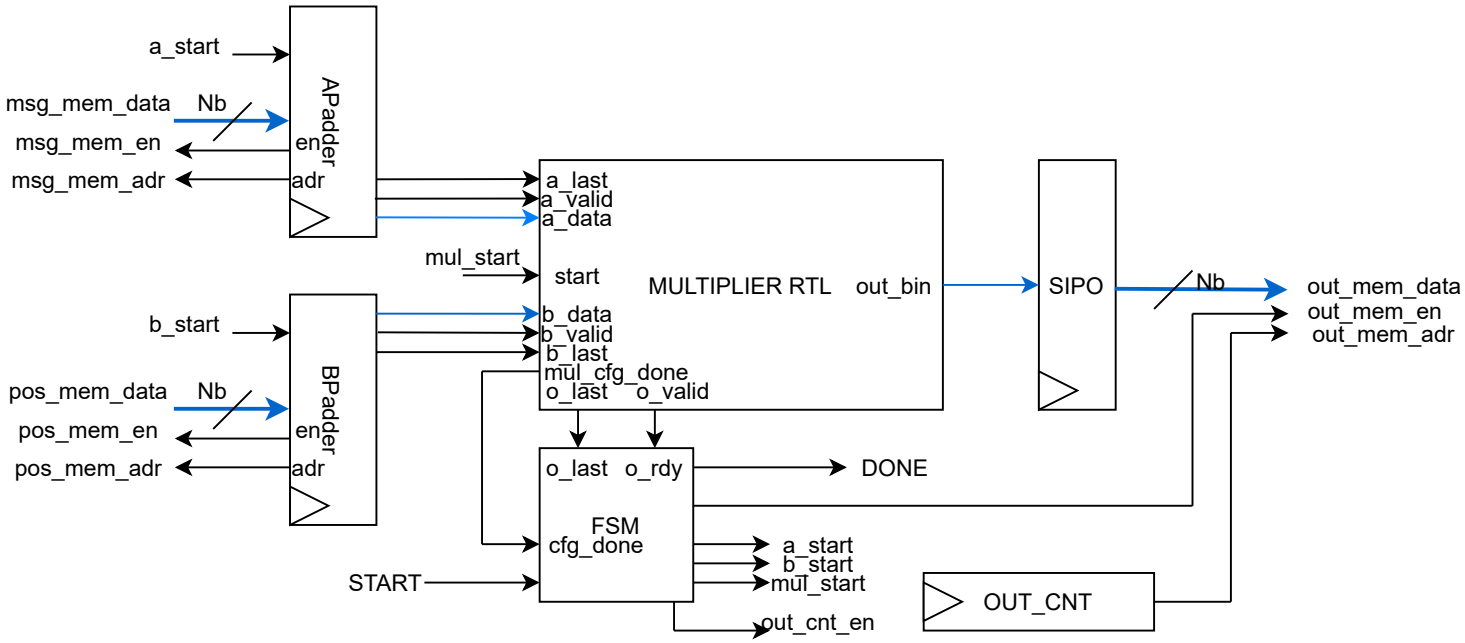


Figure 3.18: Vbc Complete RTL

3.7.1 Main Multiplier RTL

This module is the main component of the overall system and it performs the multiplication of two power of two sequences. The input data is a binary sequence and each bit represent a sample of the FFT. The output is a binary signal that is the result of multiplication of the a_data vector with the b_data vector. The RTL of the main multiplier is showed below:

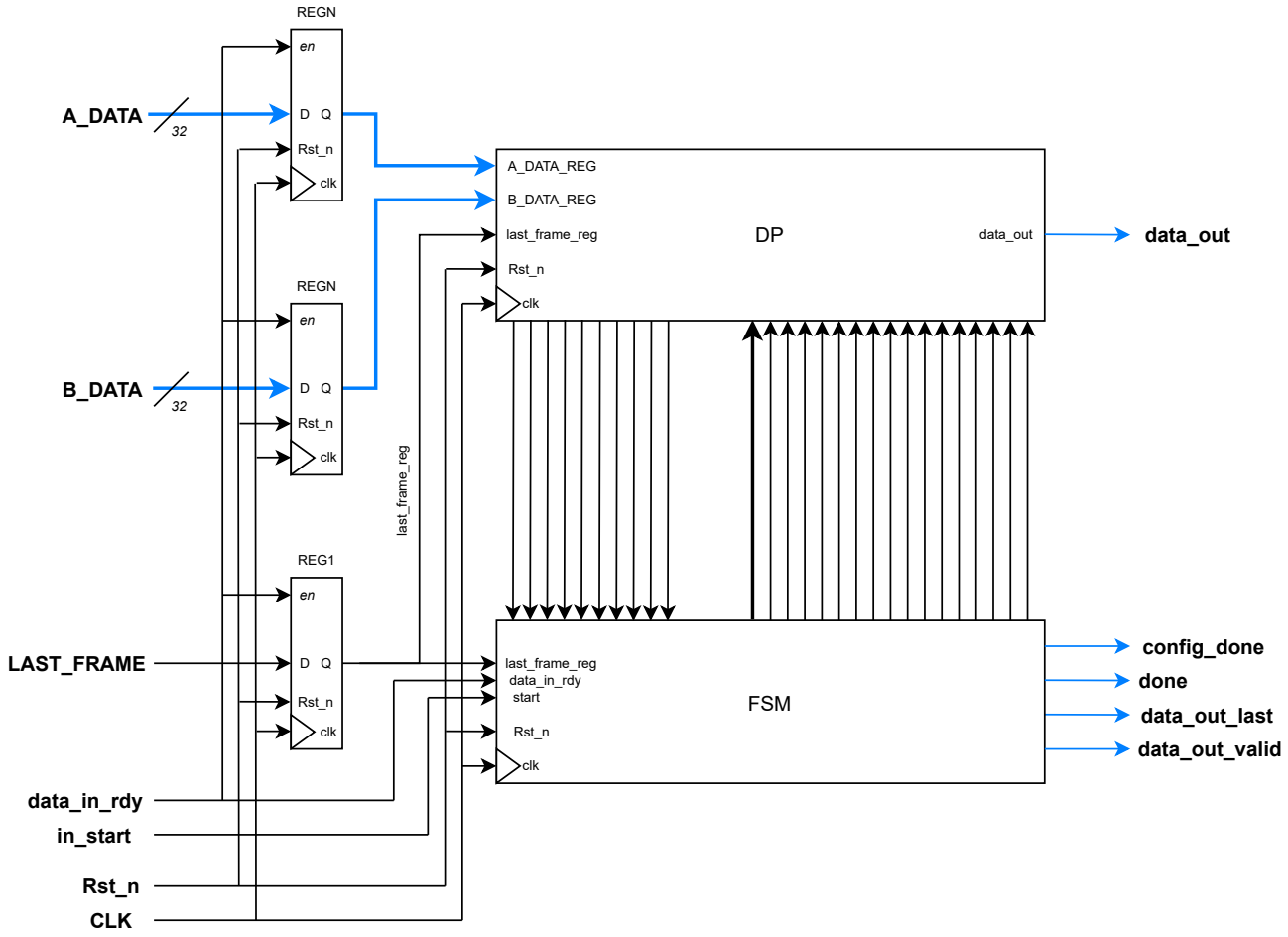


Figure 3.19: Main Multiplier overall RTL

It is divided into two main entities, one is the control logic, FSM, and is shared across different implementation system architecture with different number os FFT samples and an entity, the datapath which instead is different when different number of samples are used. A new Xilinx IP has to be generated when the FFT sample size or input number of bits is changed. Now we describe in more details bot the FSM module and the DP:

Control FSM The control FSM controls the datapath, it is organized in 15 states and implemented as a Moore State Machine so the output only depend on the current state variables. When the system is power-up the signal *Rst_n* is asserted low to indicate that system have to be reset. When *Rst_n* is low the state machine is started and the first operation that it performs is to wait for two clock signals in order to complete the resets of the different IPs. After that, the FSM enter its idle state in which it waits for a start signal, when start is asserted high

than the FSM generates the signals that configure the different IPs. After that the FSM asserts high the signal $config\_done$ the system is ready to received a data sample. The FSM remains in the same state, DoFFTState, until the last frame of data is present indicated by the signal $last\_frame$ asserted high. The system wait for the FFT to be completed and at the same time it routes the FFT out data into the complex multiplier IP. When the a new data is present at the output of the complex multiplier IP the FSM enables the datapath BRAM in order to save the sample into the memory. When the last multiplication result is present the FSM enable the BRAM for read and output the data into the IFFT IP. When all IFFT is loaded into the IP and a valid output is present ($ip\_ifft\_out\_tvalid$ is asserted to '1') the FSM routes the data into the BRAM (by asserting the signal fsm_sram_data_seld to one) and save the results. The FSM enters its final stage in which it routes the BRAM data into the OutConverter module. When the last BRAM address is reached, the FSM asserts high the signal $data\_out\_last$ and return to to the idle state. The FSM follows the state diagram:

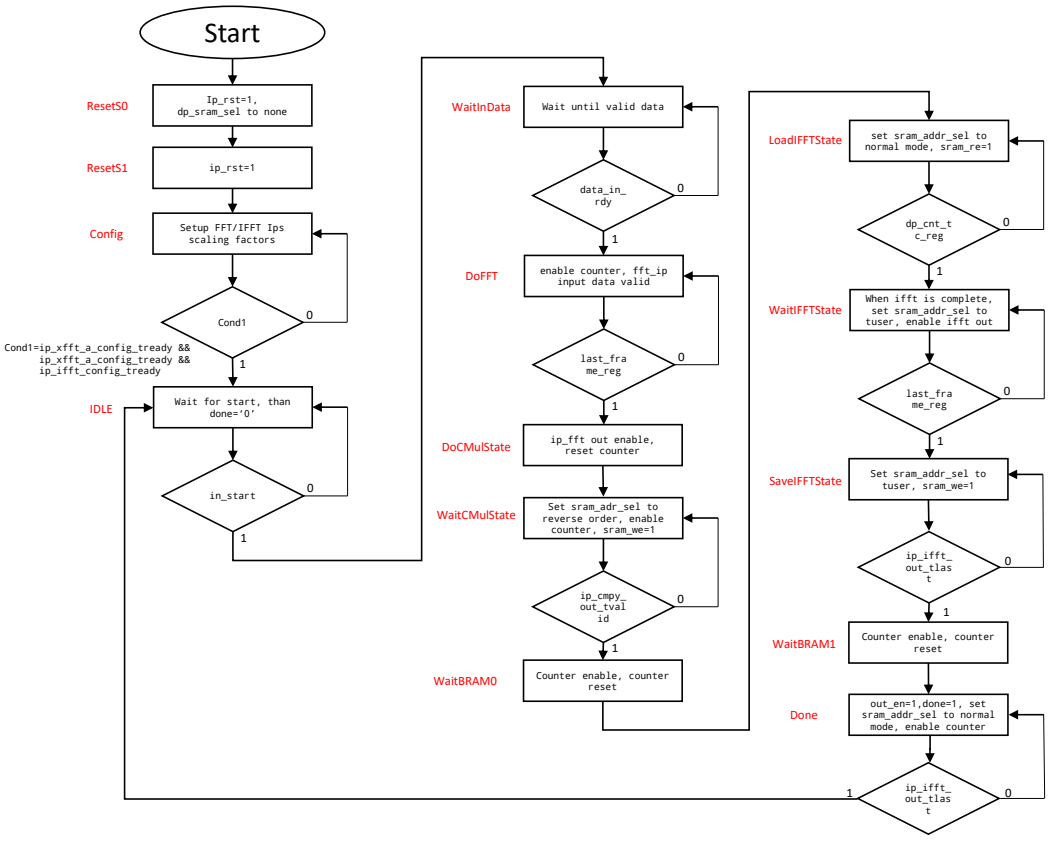


Figure 3.20: FSM State Transition

The FSM input signals that are coming from the Datapath IPs are indicated with the prefix *ip_*, the signal that are coming from the other parts of datapath are indicated with the prefix *dp_*. The FSM logic require, as expected, the main clock as an input and a global reset signal *Rst_n*.

Datapath The main multiplier datapath RTL is showed below:

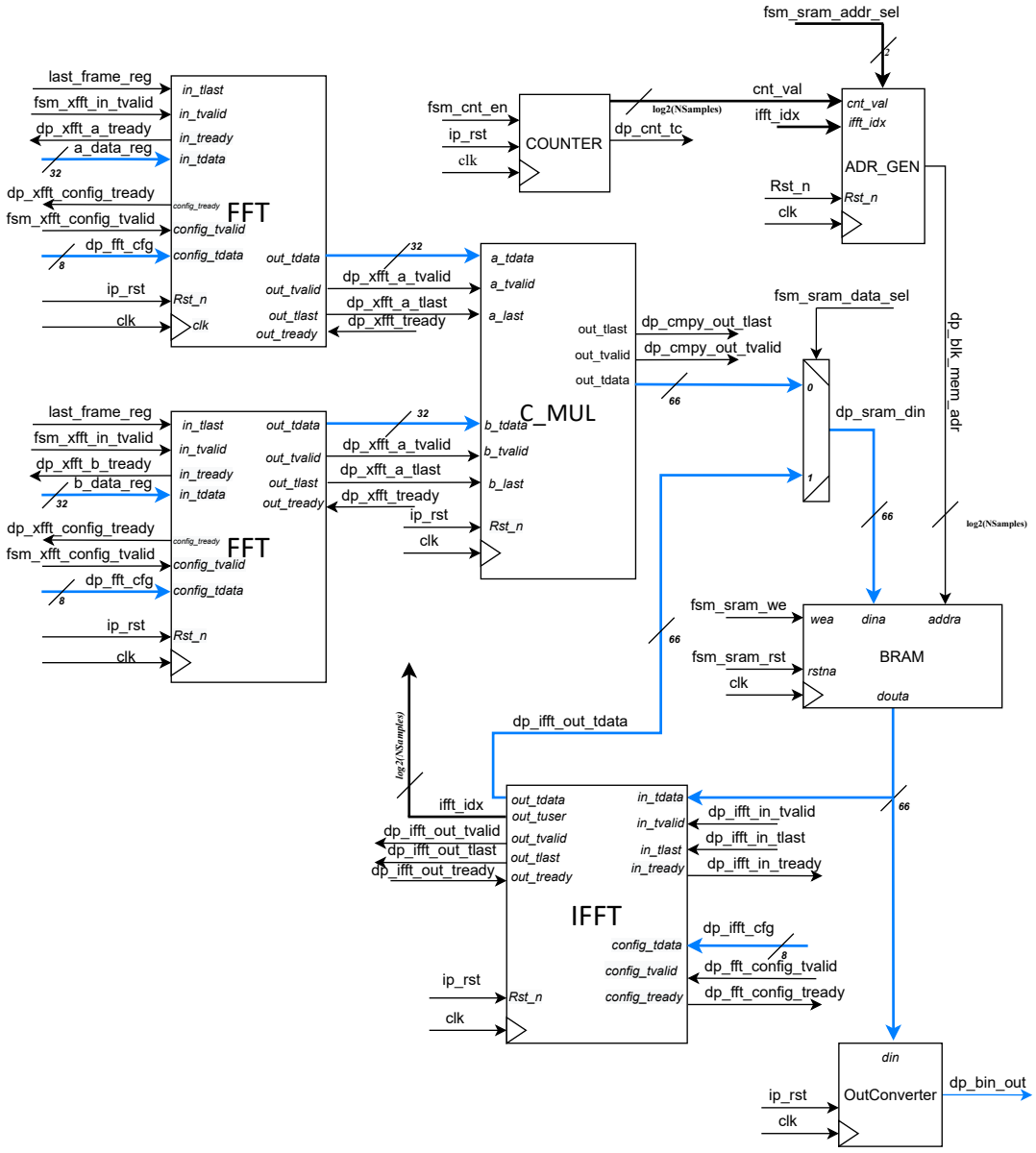


Figure 3.21: Main MUL DP RTL

It is composed by this components:

- Two FFT IP configured with the proper scaling factor as explained in the previous sections.
- One Complex Multiplier IP configured to not use any rounding.
- A FFT IP configured with no scaling and to perfom the inverse FFT (IFFT).
- One BRAM: Used to to store and reorder the temporary FFT samples.
- A MUX used to select the proper BRAM input, when the signal $fsm\_sram\_data\_sel$ is asserted low and the complex multiplier output is selected as the BRAM input data, when the signal is asserted high the IFFT output data is selected as the BRAM input data.
- A module, called OutConverter, needed to obtain the binary output result. The component convert the IFFT Q33.30 fixed point format back to a binary 0 or 1. First F.P number is denormalized by multiply it by the $2^{SAMPLES\_LOG2}$ that is equivalent to a shift the input data left by $SAMPLES\_LOG2$ bits, than the integer and the fractional part are extracted from the result. If the fractional part is greater than 0.5 than the 1 is added to the integer part. The result is obtained by calculating $BinOut = IntPart \mod 2$ that is equivalent to just look at the last bit of the IntPart of the shifted result. Basically the idea is to round the shifted result and simple round to the nearest method is employed. Below an RTL of the OutConverter block is shown:

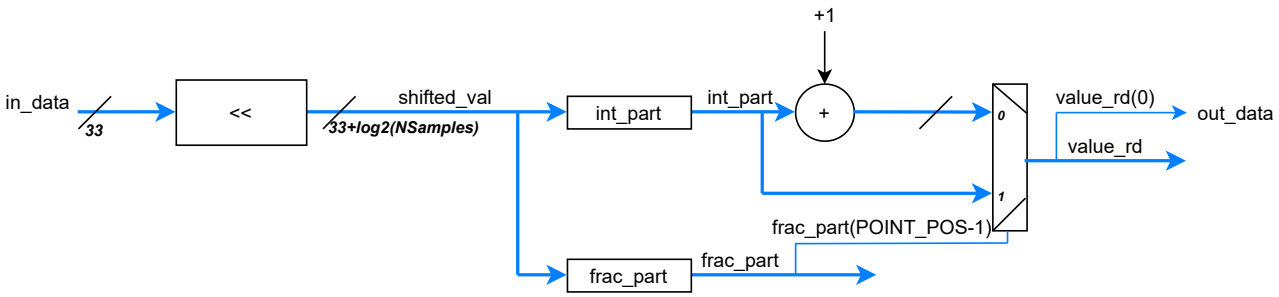


Figure 3.22: OutConverter RTL

The int_part and frac_part module extract the integer and fractional bits from the shifted value and are implemented by splitting the original vector bits. The POINT_POS represents the position of the fractional point of the shifted number. Depending of the value of the bit after the fractional point position, as already explained before, either the original integer part is returned or the value is rounded by adding one to the int_part.

The module is parameterized depending on the number of samples that the current architecture use, in order to allow to use the same module across different implementations. The VHDL code of the OutConverter is reported in the appendix B (B.1)

- A module, called ADR_GEN used to generate the BRAM read/write address. The component provide the correct address to the BRAM depending on the current state machine state. It is controlled by the signal fsm_sram_addr_sel and route to the output either the value coming from the datapath counter or the reverse bit order indices obtain from the IFFT tuser port. The output are selected depending on this condition:
  - If $fsm\_sram\_addr\_sel = SRAM\_ADR\_SEL\_NONE$, $out = 0$
  - If $fsm\_sram\_addr\_sel = SRAM\_ADR\_SEL\_CNT\_VAL$, $out = cnt\_val$
  - If $fsm\_sram\_addr\_sel = SRAM\_ADR\_SEL\_CNT\_VAL\_REV$, $out = cnt\_val\_rev$
  - If $fsm\_sram\_addr\_sel = SRAM\_ADR\_SEL\_CNT\_TUSER$, $out = ifft\_idx$
  - If Rst_n=0, $out = 0$
- A counter mod N is needed to count the current sample number and determine the BRAM index. The counter provides two outputs: one is the current counted value and the other is a terminal count value that is one when the counter value reach the maximum countable number which is $2^N - 1$. The module is parameterized depending on the number of samples that the current architecture use, in order to allow to use the same module across different implementations. The VHDL code of the module is reported at the end of the chapter B.2 but it could be replaced by the Xilink provided counter IP if needed.

In general the datapath signals following a name convention: all the signals that originate in the control unit are prefixed by $fsm\_$ while the signals that originate from the datapath are prefixed by $dp\_$.
The binary input samples present at the port A and B are first sampled by a register than are converted to the required xfft input value, if the input is "1" the value "0111111111111111" is selected otherwise the null value is selected. The final FFT frame is obtained by combining this temp vector with a null vector, of the same size that is the imaginary part of the FFT input frame value. The two values now enter the two different FFT IPs. When the last input sample is present at the input of the module, the signal last_frame is asserted high and the IP begin to execute the transform. The output of the two IPs are connected

directly to the Complex multiplier IP that is enabled by the control unit. When the signal *dp_cmpy_out_tvalid* is high a valid output is present at the output of the complex multiplier. The output value is either sent to the BRAM directly, if the circuit is implemented for a transform size less than 8191, or it is truncated by removing the extra two bits (the lower 2 bits of both real and imaginary) and it is stored into the BRAM. When the complex multiplier output the last samples, $dp\_cmpy\_out\_last =' 1'$, all the samples have been loaded into the BRAM and had been reordered (we recall that the FFT output samples are in bit reverse order) and are ready to be sent to the IFFT IP. Subsequently the samples are read from the BRAM and loaded into the IFFT IP. When the IFFT output data is valid, $dp\_ifft\_out\_tvalid =' 1'$, a valid IFFT sample is available at the IP output port. The output samples are routed back to the BRAM via a MUX selected by the signal $fsm\_sram\_data\_sel$. After all the IFFT samples are stored in BRAM, they are read sequentially and the real part of each of them (that is the higher bits) are sent to the OutConverter that convert the IFFT Q32.30 F.P. number into a binary number that is the main output of the datapath. When the last output sample is present the signal *out_last* is asserted high and the state machine enter the idle state waiting for a new start. An additional output, represented as a unsigned number, is connected to the non binary bounded value of the out converter.

3.7.2 Data Padders RTL

When the multiplication have to be performed with sequences that are not a power of two, that is what happens in LEDA/BIKE systems, padding is required to use classical FFT circuits. Indeed when the number of samples, that are the number of bits of the incoming vector, is not a power of two, as described in section 3.2.2 a valid solution is to extend the initial sequence to a sequence that is a power of two by performing a particular type of padding. We recall that given the sequence of prime length p, a new sequence of size L = NextPow2(2p - 3) is generated by applying the padding technique described in 3.2.2. So it is necessary to introduce two additional modules to the design that perform the mentioned padding. These are called APadder and BPadder and are described in the next two paragraphs.

APadder The APadder circuit has to insert a run of L - p zeros between the first and the second element of the data vector. To do so the circuit first reads the data vector from a memory organized as matrix where each row contains exactly Nb bits of the original data vector. The memory have $\lceil \frac{p}{Nb} \rceil$ rows where the left bits, in the last row, are filled with zeros. This memory arrangement was chosen to match the LEDApkc system implementation detailed in [13]. Nb define the overall system parallelism. This choice makes possible to just replace the normal vector by circulant multiplier with the developed circuit without changing the other part of the overall decoder. The system works with three counters, one counts the number

of emitted bits and determine the output value, one counter keeps track of the current row emitted bit, when Nb bits are emitted the row counter is reset to zero and the memory address is incremented. The circuit ends when the bit counter reaches the total number of extended bits L. The proposed circuit works following the algorithm report below:

Algorithm 2: APadder Algorithm

Data: **MEM**: A Memory containing $R = \lceil \frac{p}{Nb} \rceil$ rows of the original binary message, **Nb**: The number of bits per row in the memory MEM, **p**: a prime number that is the size of message in bits
Result: OUT: A binary vector of size P_VAL_EXT
$P\_VAL \leftarrow p - 1$;
$P\_VAL\_EXT \leftarrow NextPow2(2 * P\_VAL - 3) - 1$;
$PAD\_BIT\_COUNT \leftarrow P\_VAL\_EXT - P\_VAL$;
$LAST\_PAD\_BITS\_POS \leftarrow PAD\_BIT\_COUNT$;
// Initialize the counters;
$b\_idx \leftarrow 0$;
$m\_idx \leftarrow 0$;
$r\_idx \leftarrow 0$;
$curr\_row \leftarrow 0_{Nb}$;
while $b\_idx < P\_VAL\_EXT$ **do**
  $curr\_row \leftarrow MEM(m\_idx)$;
  if $b\_idx == 0$ **then**
    $OUT(b\_idx) \leftarrow curr\_row(0)$;
    $r\_idx \leftarrow r\_idx + 1$;
  else if $b\_idx \leq LAST\_PAD\_BITS\_POS$ **then**
    // Emit Pad Bits ;
    $OUT(b\_idx) \leftarrow 0$;
  else
    // Emit left memory bits ;
    $OUT(b\_idx) \leftarrow curr\_row(r\_idx)$;
    if $r\_idx == Nb - 1$ **then**
      $r\_idx \leftarrow 0$;
      $m\_idx \leftarrow m\_idx + 1$;
    else
      $r\_idx \leftarrow r\_idx + 1$;
    end
  $b\_idx \leftarrow b\_idx + 1$;
end

To implemented the algorithm the following RTL has been developed (for simplicity all the clocks and reset signal are omitted but all modules have both a clock and a Rstn_n signal in common):

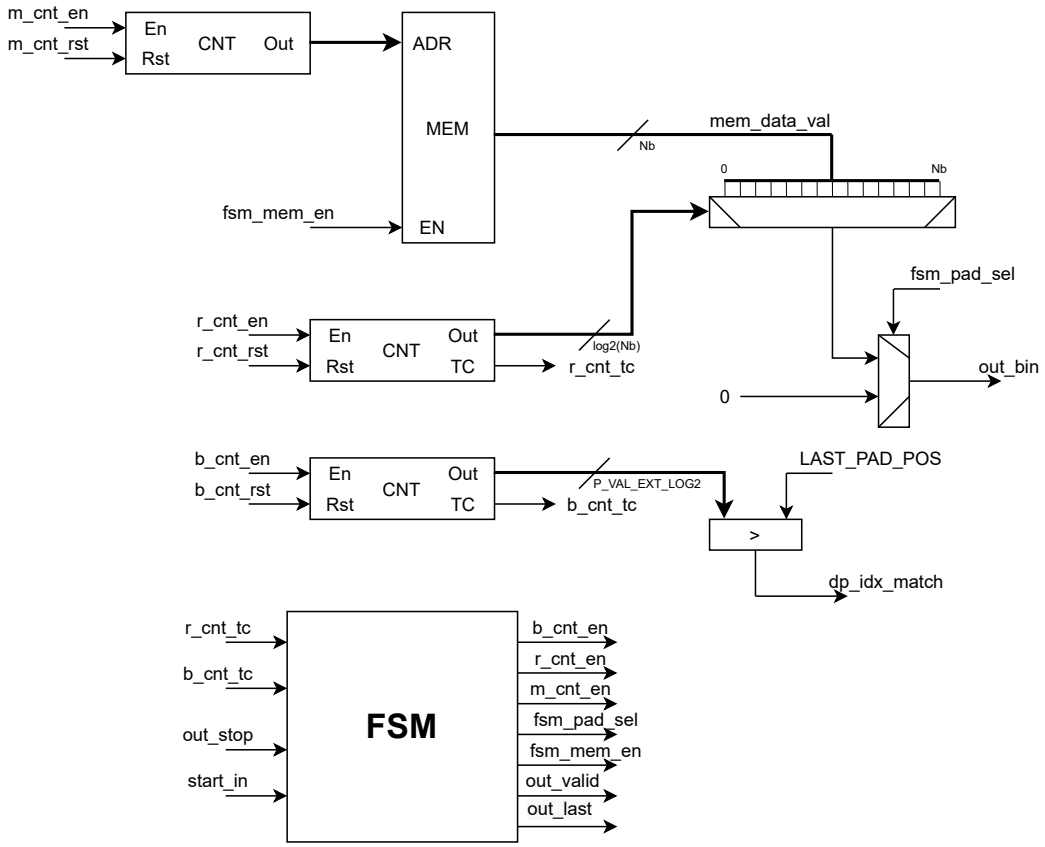


Figure 3.23: RTL of APadder

Three counters are used: one counts the total number of bits b_cnt_val, one counts the memory row bit index r_vnt_val and the last, m_cnt_val, determines data memory read address. A MUX is used to select which of the current Nb row's bits have to be selected and is controlled by the r_cnt_val; another MUX is used at the output of the circuit to select either a 0, used for padding, or the selected bit of the current memory row. At last a comparator is used to determine the state of the circuit.
To control the circuit a simple state machine is used. It is made by 7 states: ResetState, IdleState, BeginFirstRowState, EndFirstRowState, EmitPadBits, EmitRowBitsState, DoneState. The FSM first starts in the reset state, than enters the IdleState and wait for the signal start_in to be asserted high, than enters the BeginFirstRowState where the first bit of the first row the memory is emitted and the

r_cnt_val is incremented by asserting the signal r_cnt_en to one. Than the FSM enters the PadEmitState in which the output MUX is selected to emit a 0 and the FSM wait for the signal dp_match_idx to be asserted high. Than the FSM enters the EndFirstRowState in which the remaining Nb-2 bits of the first row are emitted. When the signal r_cnt_tc='1', all the bits of the first row are emitted, the memory read address is incremented and the FSM enters the EmitRowState in which all the remaining rows of the message memory are handled. Every time r_cnt_tc='1' the memory read address is incremented by asserting the signal mem_cnt_en to one. When the signal b_cnt_tc='1' all the bit have been processed and the FSM enters the done state in which the signal done is asserted to one and all the counters are cleared. Finally the FSM return to the idle state and waits for a new start signal. In every state if the signal out_stop is asserted high the FSM is stopped in that state until the signal go low. This external signal, that is optional, could be used for synchronization with other modules. Below the FSM flowcharts, for both the state update the outputs, are reported:

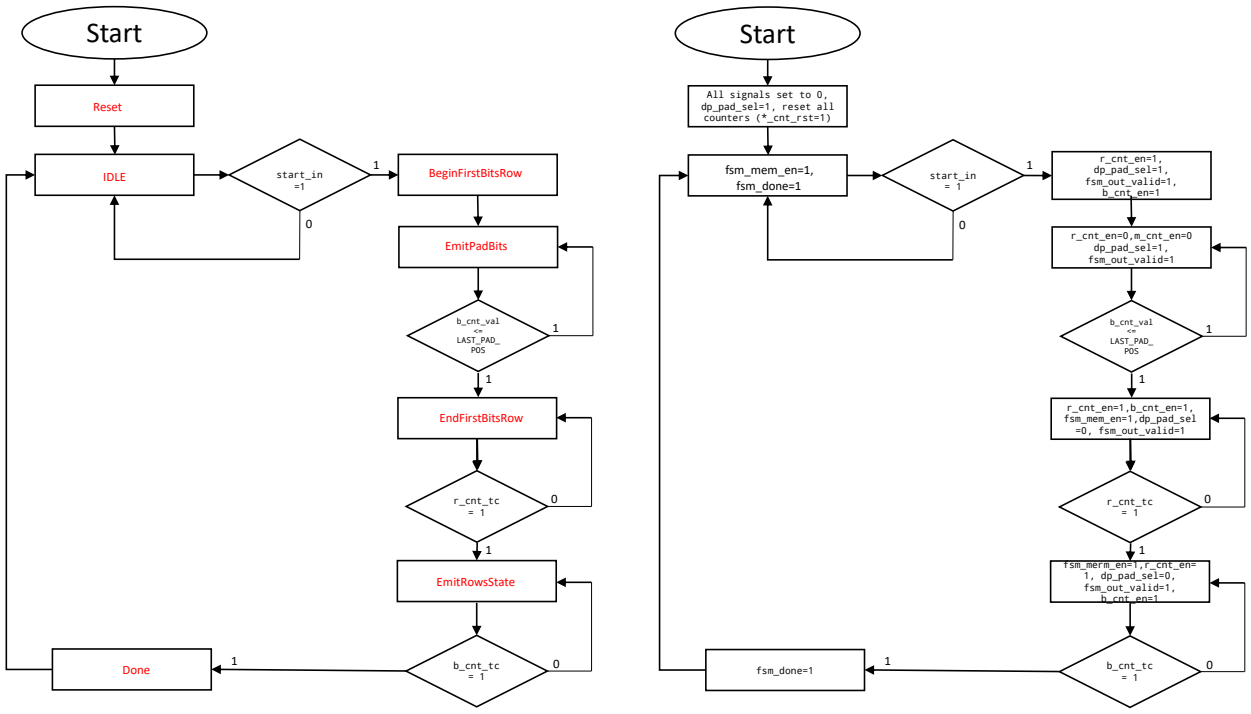


Figure 3.24: APadder - FSM flochart

BPadder The goal of this module is to take the input positions of the non null bits of the first row of the circulant matrix and generate a sequence of bits. The padding is applied by cyclically repeating the obtained sequence to reached the required length L. In the case that to reach L a non integer multiple of p is required, the sequence is just truncated at L. Below we explain with an example the required padding, we consider a system with $p = 11$, $L = NextPow2(2p-3) = 32$:

$$\begin{aligned} \underline{b}_p &= [0, 1, 2, 6] \\ \underline{b} &= [1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0] \\ \underline{b}_{ext} &= [b_0, b_1, ..., b_{10}, b_0, b_1, ..., b_{10}, b_0, b_1, ..., b_9] \end{aligned} \tag{3.25}$$

In general it is useful for description on how the circuit work to think of the original message as a frame of data. The padding basically repeat this frame multiple time and than adds the remain bit that would not fit a frame to reach the desired length L. To understand better how this padding can be implemented it is useful to introduce the following quantities:

$$\begin{aligned} FRAME\_SIZE &= p \\ LEFT\_COUNT &= mod(FRAME\_SIZE, L) \\ PAD\_SIZE &= L - LEFT\_COUNT \\ FRAME\_COUNT &= \frac{PAD\_SIZE}{FRAME\_SIZE} \end{aligned} \tag{3.26}$$

The proposed circuit works following the algorithm report below:

Algorithm 3: BPad Algorithm

Data: **MEM**: A Memory containing D_V entries that are the index of the bit equals to one in the first column of the circulant matrix, **p**: a prime number that is the size of the first row of the matrix

Result: OUT: A binary vector of size P_VAL_EXT

$P\_VAL \leftarrow p - 1$;
$P\_VAL\_EXT \leftarrow NextPow2(2 * P\_VAL - 3) - 1$;
$b\_idx \leftarrow 0$;
$f\_idx \leftarrow 0$;
$m\_idx \leftarrow 0$;
while $TRUE$ **do**
  $mem\_check\_idx \leftarrow MEM(m\_idx)$;
  $check\_idx \leftarrow b\_idx - f\_idx * P\_VAL$;
  if $mem\_check\_idx == check\_idx$ **then**
    $m\_idx \leftarrow m\_idx + 1$;
    $OUT(b\_idx) \leftarrow 1$;
  else
    $OUT(b\_idx) \leftarrow 0$;
  end
  if $b\_idx == (f\_idx + 1) * P\_VAL$ **then**
    $m\_idx \leftarrow 0$;
    $f\_idx \leftarrow f\_idx + 1$;
  end
  if $b\_idx == P\_VAL\_EXT$ **then**
    break;
  else
    $b\_idx \leftarrow b\_idx + 1$;
  end
end

To implemented the algorithm the following RTL has been developed:

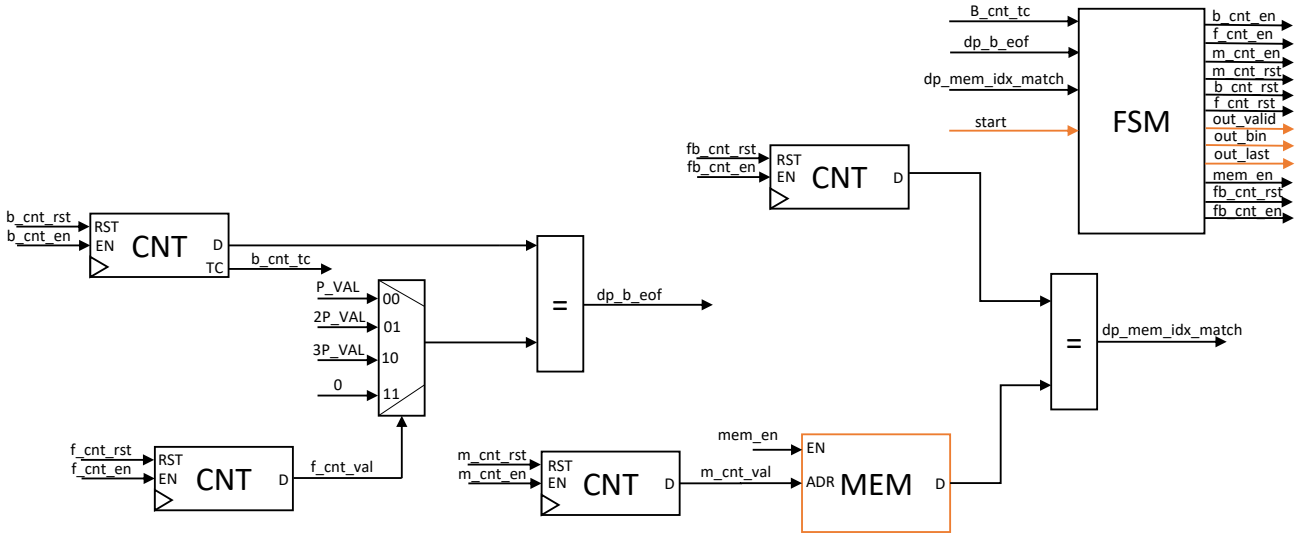


Figure 3.25: BPadder RTL

it is composed mainly by four counters: one counts the current frame number, one counts the current frame bit index, one count the current memory index and the last counts the total emitted bits and two comparators that are used to detect if the total emitted bits are equal to the total number of bits and to the last frame index. To calculate the last frame index a multiplexers is used to select either the value P_VAL, 2*P_VAL and 3*P_VAL. The MUX is controlled by the frame index and the output is selected according to:

f_cnt_val	$dp\_eof\_frame\_idx$
00	P_VAL
01	$2 * P\_VAL$
10	$3 * P\_VAL$
11	0

when $dp\_mem\_idx\_match = 1$ the value read from memory equals the current frame bit index, fb_cnt_val the memory address is incremented and the output of the circuit is set to one otherwise the output is set to zero.
To control the different counters a simple state machine is developed that is composed only by 4 states: ResetState, IdleState, EmitBitState, DoneState. The machine first enters its reset state than the IdleState. When the signal start is asserted high the state machine enter the EmitBitState in which the output signal is determined. When the total bit counter reaches the value P_EXT_VAL the machine

enter its done state in which the signal done is asserted high. Finally the machine return back to the idle state and wait for a new start. Below the flowchart for both the state evolution and the output is reported:

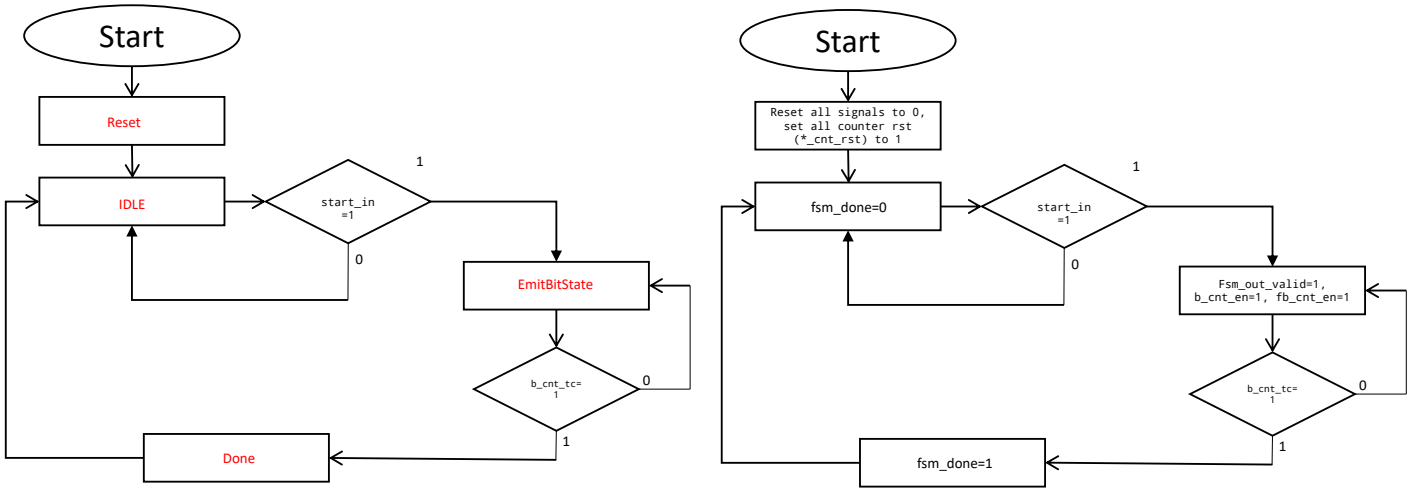


Figure 3.26: BPadder - FSM flochart

3.8 System Simulation and Performances

3.8.1 RTL Simulation

The design was carried out in a modular way so three design were developed, one that can handle LEDA systems with p less than 8192, one for systems that can handle p up to 16386 and finally one that can handle systems with p less than 32768. These systems are configured at compile time by setting the LEDA system parameters in the entity generic fields. The system applies the appropriate padding and than perform the convolution via the Fourier transform. As already mentioned before the maximum possible p value is 32749 due to the fact that Xilinx does not provide FFT IP that can accept sequence of higher dimension (we recall that for the necessary padding, the length is double to a sequence of size at least 2p-3). To validate the correctness of the system all the values of p of the table 2.1 expect p=36877 have been simulated using Questa Sim simulation scripts that Vivado can generate.
To simulate the circuit the message data and the position of the non null entries of the first line of the circulant matrices are stored into two memory. The content of these memories are loaded, when the simulation starts, from some configuration text files. The binary circuit result is logged into another file. A MATLAB script is used to generate from the LEDA system parameters (p, n0, d_v, m) the L matrix and a random message and save them into the appropriate files with the appropriate Nb parallelism. The message data is stored into a memory with a width equals to Nb bits and a height equals to $\lceil \frac{p}{Nb} \rceil$. The position memory have an height of d_v and a width equals to $\lceil \log_2(p) \rceil$. The below table summarizes the execution time for each of the LEDA system of table 2.1 with different Nb parallelism:

		Nb				
p	**n0**	**16**	**32**	**64**	**128**	**256**
7547	4	302.131 us	302.600 us	302.605 us	302.811 us	302.845 us
7853	3	303.049 us	303.157 us	303.157 us	303.305 us	303.310 us
14341	4	656.683 us	656.735 us	656.815 us	656.820 us	656.913 us
14939	2	659.347 us	659.410 us	659.455 us	659.455 us	659.470 us
16067	3	664.459 us	664.489 us	664.500 us	664.500 us	664.540 us
22691	4	1569.6 us	1569.7 us	1569.9 us	1570.1 us	1570.2 us
25693	2	1586.1 us	1586.3 us	1586.5 us	1586.7 us	1586.9 us
27437	3	1596.3 us	1596.4 us	1596.7 us	1598.9 us	1599.0 us
36877[1]	-	-	-	-	-	-

Table 3.1: Simulation time for 3 FTT IP Multiplier systems

As we can see the Nb value does not make a significant impact to the overall performance, so it can be fixed arbitrarily depending on the overall decoder parallelism. Fixing Nb=16, for example, it is possible to calculate the maximum working frequency of the system and than the required simulation cycles. The next table show the results:

p	**n0**	T_{sim}	T_{clk}	**# Cycles**
7547	4	302.131 us	4.1 ns	73691
7853	3	303.049 us	4.1 ns	73915
14341	4	656.683 us	4.5 ns	145930
14939	2	659.347 us	4.5 ns	146522
16067	3	664.459 us	4.5 ns	147658

Table 3.2: Architecture "1": Normal DP: cycles, Tclk

One of the consuming operation is the fact that we need to access twice the DP memory in order to reorder the FFT output sample back into the natural order. To try to improve the performance of the circuit it is possible to configure the FFT IP with natural output order to avoid the requirement of an external BRAM into the main datapath. To verify if this solution is valid the same simulation as also carried for this system that require minor change into the main datapath FSM to remove the BRAM write and read states. We show the results of this simulation, again Nb is fixed to 16 and we report the max working frequency and the number of required cycles in the table below:

p	**n0**	T_{sim}	T_{clk}	**# Cycles**
7547	4	286.933 us	3.9 ns	73573
7853	3	288.118 us	3.9 ns	73877
14341	4	598.360 us	4.1 ns	145942
14939	2	600.787 us	4.1 ns	146534
16067	3	605.444 us	4.1 ns	147670

Table 3.3: Architecture "2": Natural Order FFT, # cycles, Tclk

! As it can be seen in the above two table the LEDA systems with p greater than 2$\hat{1}$6 are missing as the are too big to be synthesized in the chosen FPGA. The

[1]Configuration is not available using Xilinx IPs due to padded transform size been too big

main problem of these systems are that they require to much BRAM blocks that the chosen FPGA cannot provide. To overcome this problem a final variation of the circuit is possible. In this architecture only two FFT IPs are used, one for the b_data input and one is shared between the a_data and the output of the complex multiplier. To make the system work the FFT is required to have 33bit as the input width so the a_data input is extended to 33bit by applying zeros. This additional bits are removed after the module before the complex multiplier by performing easy truncation. Probably more simulation should be carried in order to detect what would be the best rounding method to reduce the possible number of errors. To select the correct FFT input depending on stage of the operation a mux is used. This architecture requires a change version of the FSM entity to be able to provide the new control signals.
This implementation can be used to reduce the resource requirement and allow the circuit to be implemented into a smaller FPGA for example. Below we show the simulation required number of cycles and max working frequency with Nb=16 for all the LEDA systems analyzed in this thesis:

p	**n0**	T_{sim}	T_{clk}	**# Cycles**
7547	4	301.737 us	4.1 ns	79595
7853	3	303.049 us	4.1 ns	73915
14341	4	802.433 us	4.55 ns	176359
14939	2	806.994 us	4.55 ns	177361
16067	3	811.937 us	4.55 ns	178448
22691	4	1569.6 us	5.5 ns	284364
25693	2	1586.1 ms	5.5 ns	288182
27437	3	1596.3 ms	5.5 ns	290237

Table 3.4: Architecture "3": 2 FFT IP DP, # cycles, Tclk

We clearly see that architecture "2" performances slightly better than architecture "1" but it has an easier datapath as almost all external components outside the main IPs are removed. The only module left is the OutConverter that is still required to perform the final rounding. For the two IPs solution the draw back is that the overall resource requirements are reduced but the DP is more complicated as additional muxes are required to route the correct input and output of the FFT IP that is shared for the FFT and IFFT operation. This additional complexity results is a worst overall performances both in term of simulation speed and working frequency.

3.8.2 RTL synthesization

After having verified that the design works the next phase is to try to synthesize it in order to evaluate the circuit performances. First it is necessary to select an FPGA to target. It is necessary that the selected FPGA have enough DSP blocks to implement the FFT/CMUL IPs and enough BRAM resources in order to accommodate the BRAM present in the design datapath. To understand the required hardware resources needed for the two IPs it is possible to obtain the number of needed DSP from the IP configuration wizard. This requirements, of course, change when the number of samples increased. As for simulation step, we are interested in the cases that represent the LEDA systems of table: 2.1. The above table summarize the IPs requirement for these three systems:

Number Samples	**IP**	**DSP48 Slices**	**BRAM**
$p = 8192$	xfft(FFT) xfft(IFFT) cmul bram	36 72 4 (none)	47 78 (none) 1 (18k), 29 (36k)
$8192 < p \leq 16384$	xfft(FFT) xfft(IFFT) cmul bram	42 84 8 (none)	99 152 (none) 0 (18k), 59 (36k)
$16384 < p \leq 32768$	xfft(FFT) xfft(IFFT) cmul bram	42 84 8 (none)	188 296 (none) 0 (18k), 118 (36k)

Table 3.5: IPs Resource Requirements

As it can be seen the resources required from the systems differs depending on the number os samples. In the case of systems with p >= 8192 the same complex multiplier is used and and the mentioned trick of extend the FFT input number of bits to make the system works is applied. For the smaller system an lower number of extended bits are required and so the complex multiplier requires less DSP slices. We can see that when we consider big LEDA systems the number of BRAM explodes and so it's not so easy to implement such system in smaller FPGA. The selected FPGA is based on the Artix 7 (xc7a200tsbg484-1) family of FPGA that offer quite good performances and enough hardware resources to implement the high number of samples system. After the FPGA is selected is now possible to synthesize the design to obtain the real system resource utilization. The synthesize has been performed using Vivado 2018.3, Below the table of the resource utilization

is reported for the different systems:

Num Samples	Resource	Utilization	Available	%
$p \leq 8192$	LUT	17287	133800	12.92
	LUTRAM	4993	46200	10.81
	FF	29058	267600	10.86
	BRAM	114	365	31.23
	DSP	148	740	20.00
	IO	78	285	27.37
$8192 < p \leq 16384$	LUT	19887	133800	14.86
	LUTRAM	5573	46200	12.06
	FF	33351	267600	12.46
	BRAM	234	365	64.11
	DSP	176	740	27.02
	IO	81	285	28.42

Table 3.6: Architecture 1: 3 IP FFT main_multiplier

as it can been seen the number of DSP blocks required increase with the FFT number of samples. Also the number of flips flop increase with the number of samples as the bigger FFT IPs have more pipeline stages. Below we report the synthesis result for the other two suggested system that use either the natural order FFT IPs or the 2 IP architecture:

Num Samples	Resource	Utilization	Available	%
$p \leq 8192$	LUT	17381	133800	12.99
	LUTRAM	4976	46200	10.77
	FF	29226	267600	10.92
	BRAM	120.50	365	33.01
	DSP	148	740	20.00
	IO	78	285	27.37
$8192 < p \leq 16384$	LUT	19804	133800	11.90
	LUTRAM	5571	46200	8.97
	FF	33265	267600	10.68
	BRAM	234	365	64.01
	DSP	176	740	15.00
	IO	77	285	25.96

Table 3.7: Architecture 2: 3 IP FFT normal order main_multiplier

Num Samples	Resource	Utilization	Available	%
$p \leq 8192$	LUT	13415	133800	10.03
	LUTRAM	3896	46200	8.43
	FF	22381	267600	8.36
	BRAM	94	365	25.75
	DSP	116	740	15.68
	IO	74	285	25.96
$8192 < p \leq 16384$	LUT	149898	133800	11.13
	LUTRAM	4178	46200	9.04
	FF	24651	267600	9.21
	BRAM	184.5	365	50.55
	DSP	134	740	18.11
	IO	77	285	27.02
$16384 < p \leq 32678$	LUT	16804	133800	12.56
	LUTRAM	4580	46200	9.91
	FF	26515	267600	9.91
	BRAM	359.50	365	98.49
	DSP	134	740	18.11
	IO	80	285	28.07

Table 3.8: Architecture 3: 2 IP FFT main_multiplier

We can see that the former architecture require a little bit less resources respect to the normal architecture with external BRAM. The latter requires less resources and is better suited for an implementation into small FPGAs.

Chapter 4

Results

Having analyzed all the performances of the different developed architectures we now compare the proposed solution with other state of the art polynomial multipliers that can be used for the LEDA/BIKE systems. The next table compare the developed architectures:

ARCH	p	LUT(%)	FF(%)	BRAM(%)	$T_{sim}(\mu s)$	# Cycles
1	7853	12.92	10.86	31.23	303.05	73915
	14939	14.86	12.46	64.1	659.35	146522
2	7853	11.90	10.68	33.01	288.12	73573
	14939	12.99	10.92	64.0	600.79	146534
3	7853	10.03	8.36	25.75	303.05	73915
	14939	11.13	9.21	50.55	806.99	177361
	22691	12.56	9.91	98.49	1569.6	284364

The next graphs shows in a better way the results of the previous table:

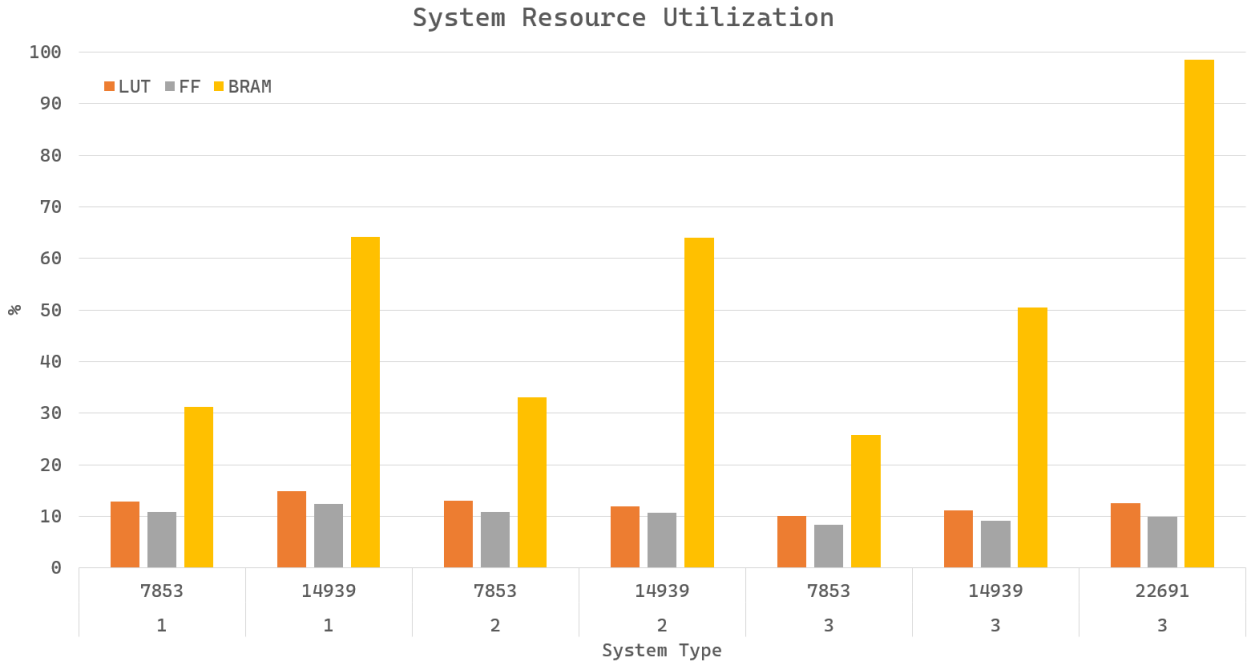


Figure 4.1: FFT Based Multiplier Hardware resources utilization

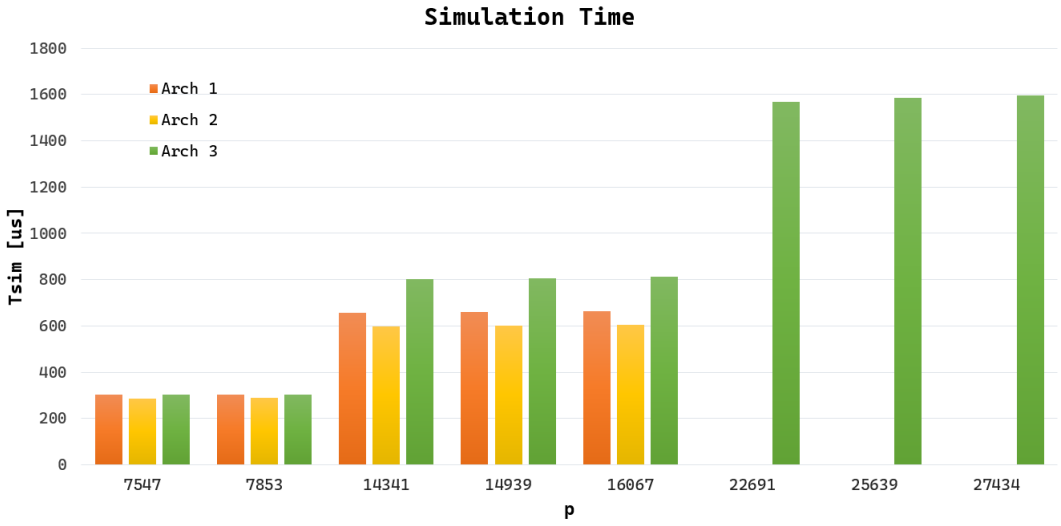


Figure 4.2: Architecture "1","2","3" T_{sim}

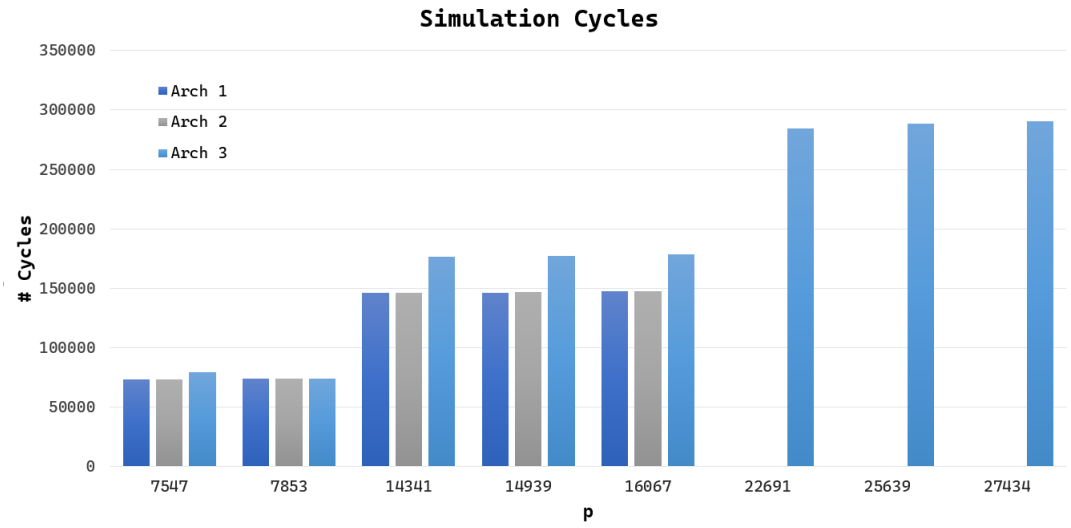


Figure 4.3: Architecture "1","2","3" # Cycles

Finally the table below reports the maximum working frequency of for different value of p and architecture types:

p	Architecture	fMax
$p \leq 8192$	1	243 MHz
	2	256 MHz
	3	243 MHz
$8192 \leq p \leq 16384$	1	243 MHz
	2	256 MHz
	3	243 MHz
$16384 \leq p \leq 32768$	1	-
	2	-
	3	181 MHz

Table 4.1: Architecture "1","2","3": fMax

Polynomials multipliers have been studied widely in literature: Karatsuba based multiples, tailored for LEDApkc, have also been studied in [17], NTT based multipliers have been studied in [18] but they are used for multiply polynomial of smaller degree. The graphs below shows the hardware resources of the different systems for different value of p. We synthesized and simulate the three architectures for the p=9643,17627,22853 that correspond to system P1,P5 and P6 in [17]. The graphs below compares different kind of polynomials multipliers:

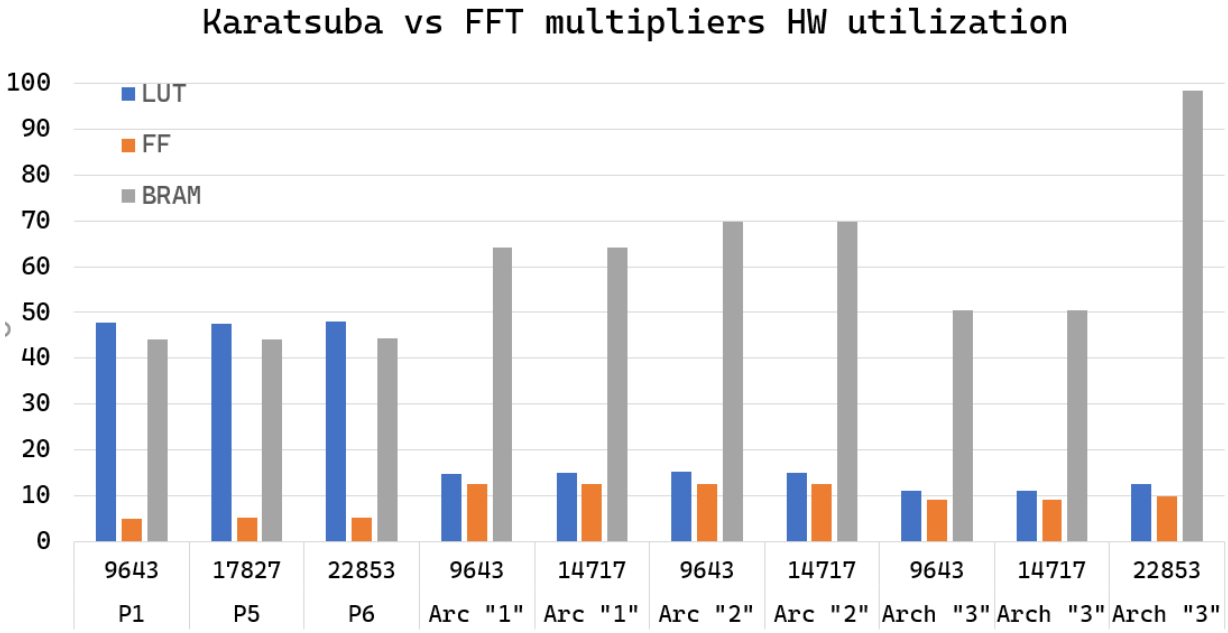


Figure 4.4: Hardware resources comparison between different type of multipliers

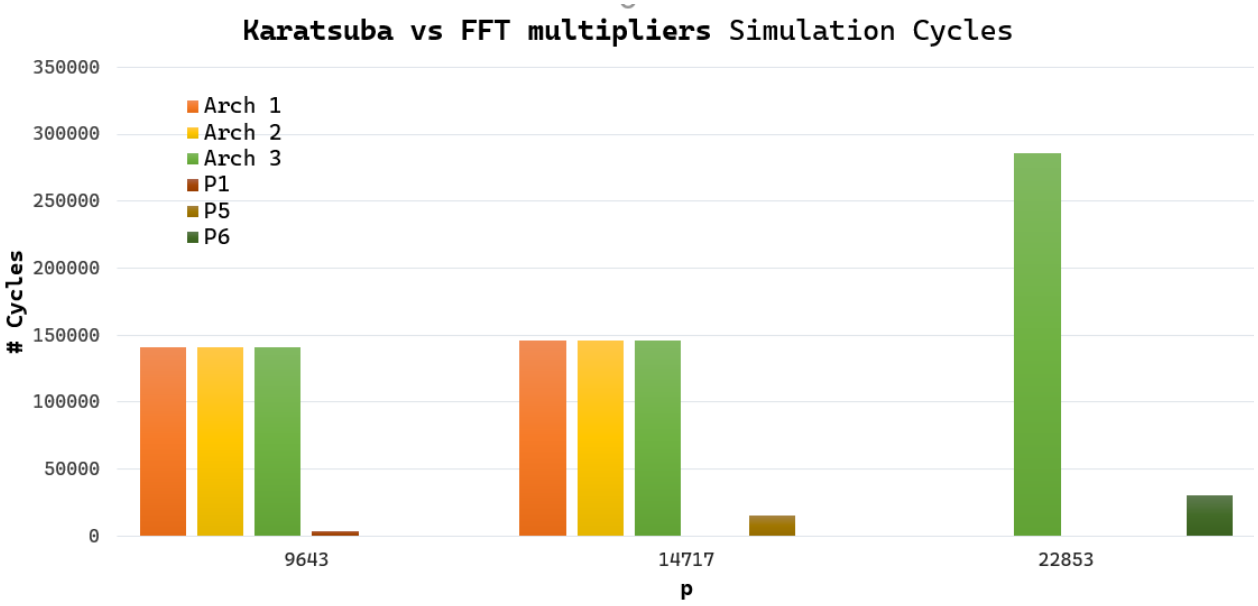


Figure 4.5: Simulation Cycles comparison between different type of multipliers

We see that in term of hardware resources the proposed system shows advantages or disadvantages depending on the value of p. This is due mainly to the use of the Xilinx FFT IP that imposes big BRAM utilization when the number of samples is high; indeed for systems with $p \geq 8192$ the BRAM utilization increases drastically. We see that our implementation uses less LUT than the Karatsuba implementation and achieves an higher fMax. In term of simulation cycles we see that our solution is slower than the Karatsuba method. Nonetheless the suggested optimization of using a convolution and hence the Fourier transform instead of normal binary multiplication shows some improvements in hardware resources. We hoped for an improvement also in execution time and number of cycles but due to the use FFT IP from Xilinx, the overhead of highly configurable IP cores reduces the overall performance gains. Indeed the IPs have an high degree on configurability but on the other hand are not usable immediately and optimized for LEDA/BIKE systems as, for example, they need additional padding to perform the operation correctly. This padding increases the hardware resource requirements as it extend the sequence to another one that is at least double the size of the original one. This make the system latency increasing as more additional time is required to perform the three Fourier transforms needed in the system. To overcome such problems probably the best solution would be to implement custom FFT components that should use other algorithm than the normal Coley-Turkey architectures to allow to perform the transform on sequences that have prime length. On the other hand the suggested implementation allows to compute the multiplication of two binary

polynomials of any size and it is not bound to sequences that are a power of two but still uses classical FFT circuits.
As analyzed in the previous section additional optimizations of the design are possible by using only two FFT IP instead of three or using natural order configured FFT IPs to avoid the use of an external BRAM memory. Using the Xilinx FFT IP imposes requirements that makes the system parallelism higher it should be. For example the minimum input width for the IP is 8 bit but in reality as we have a binary sequence where each sample is made by just a single bit we know that the FFT result is just a combination of the different twiddle factors. If we stick on using the Xilinx IP a final idea would be to change the FFT architecture to an higher radix one but this was not considered in this thesis because the Xilinx IP does not provide higher radix pipelined FFT transform. Non pipelined FFT architectures are slower than the normal radix-2 pipelined implementation so they were not considered.
Additional optimizations such as using the NTT instead of the FFT, for example, are possible if custom FFT modules are developed. For example a software optimization of the polynomial multiplication used in the BIKE system has been proposed in [14]. This implementation tries to speed up the calculation by employing mathematical tricks regarding symmetries in polynomial rings. Such symmetries can arise, for example, by exploiting the Frobenius map detailed in [15]. In general such advanced FFT algorithms are trying to find new symmetries between the different twiddle factors. In fact it is easy to see that if we are calculating an FFT of a real sequence only half of the FFT samples are required as the other are their complex conjugate. When, to avoid floating point numbers and precision errors, the FFT is performed in the integer domain, by applying the NTT, new symmetries arise that can be exploited to optimize the implementation. Another solution, maybe easier to implement, is to use another type of FFT, called truncated FFT [16] in order to avoid the use of padding. Until now such advanced FTT optimizations have only been implemented in software by exploiting vectorized instruction set such as AVX2 for Intel based CPUs and NEON for ARM CPU.
A natural continuation of this thesis work is to study the possibility of using this advanced FFTs techniques in hardware to solve the problem of padding. First of all is required to understand if it is possible to apply these techniques to the LEDA/BIKE system[1] and, by performing more simulation, understanding if the padding is still required or not. After that, it would be necessary to understand how it is possible to map such complex algorithms to hardware and see what is the real speedup from the proposed software implementations or if the hardware implementation is too difficult and hence kills the supposing gains.

[1]This advanced optimizations techniques, based on number theoretical arguments, highly depends on the chosen value of p and is not guarantee that they always work for all the value of p. LEDA for example impose some restriction to the possible value of p

Appendix A

MATLAB Scripts

Listing A.1: FFT Vector by Circulant script

```
function [Out, OutPadded, OutInt] = ...
FFT_VecByCirc_IP (a_data, b_data_pos, p, varargin)
InputWidth = 16;

if nargin > 3
extra_bits = varargin{1};
else
extra_bits = 2;
end

round_mode = 'fix';

%% Calculate the number of samples that we have to reach
% Obtain the nearse pow2 number of samples.
NSamplesLog2 = nextpow2(p);

NSamples = 2^(NSamplesLog2);

% We need to have the element of L sorted.
% b_data_pos_sorted = sort(b_data_pos);

% we need to adjust the value ad we need the transponse of it if we
% wanna use the same input as the normal case. This is due to the fact
% that out matrix is circulant on the rows while this method and the
% literature have circulant matrices on column. So we need to
% transponse the matrix and this is the trick that allow us to
% transponse the index. We add one because matlab counting the
```

```
% vector elements starting from 1....
b_data_pos_tran = mod(p-(b_data_pos), p);
b_data = VectorFromPos(b_data_pos_tran+1, p);

% We have to pad the input vector with 0 in order to reach the required
% input size that match the numberb_data_pos_sorted of samples we have.
hasPad = (NSamples - p) > 0;
if hasPad > 0
% Obtain the extended sequence length that is the next power two
% near to 2p.
NSamples = 2^(nextpow2(2*p-3));

%
% Add the padding bits to reach the desired size that is a multiple
% of two. We require a number of element, M, such that M >= 2p - 3.
% In our case as we want to perform a FFT of a sequence that is a
% power of two, we pick M = NextPow2(2p-3). To reach this size we
% add padding bits, in the following way:
%   - for a_data we insert NSamples - p null bits between the first
%     element and the second element of the original vector.
%   - for b_data  we cyclical repeat b_data to reach the desired
%     size. To do so we first calculate how many time we can fit
%     b_data vector in the NSamples, given this we calcualate the
%     remaining bits that are the first n-th bit of b_data.
% The result that the cyclic convolution produce have size
% NextPow2(2p) and we are only consider the first p bits of it.
% The example below explain the situation:
%   - a_data_ext = [a_data(1), 0, 0, ..., a_data(2:p)]
%   - b_data_ext = [b_data, b_data, ..., b_data(1:leftBits)]
%

% Extend the a_data vector
a_input_ext = [a_data(1) zeros(1, NSamples - p) a_data(2:end)];

% extend the b_data vector
numDataFrame = floor(NSamples / p);
leftBits = NSamples - numDataFrame*p;
% Replicate the input signal numDataFrame times to add pad bits
b_input_rep = repmat(b_data, 1, numDataFrame);
b_input_ext = [b_input_rep b_data(1:leftBits)];

% Some asserts to make sure we have not proper padding
```

```
assert (leftBits > 0 && leftBits < p);
assert(length(b_input_ext) == NSamples && ...
length(a_input_ext) == NSamples);

% Set the number of pad bits

else
a_input_ext = a_data;
b_input_ext = b_data;

end

% If we add padding we need to recalculate the log2 samples based on
% the new sample number.
NSamplesLog2 = log2(NSamples);

% Generate null imaginary part for both operator.
im_null_data = zeros(1, NSamples);

fft_config.C_NFFT_MAX = NSamplesLog2;
fft_config.C_ARCH = 3;
fft_config.C_HAS_NFFT = 0;
fft_config.C_USE_FLT_PT = 0;
% Must be 32 if C_USE_FLT_PT = 1
fft_config.C_INPUT_WIDTH = InputWidth+extra_bits;
% Must be 24 or 25 if C_USE_FLT_PT = 1
fft_config.C_TWIDDLE_WIDTH = InputWidth+extra_bits;
% Set to 0 if C_USE_FLT_PT = 1
fft_config.C_HAS_BFP = 0;
% Set to 0 if C_USE_FLT_PT = 1
fft_config.C_HAS_SCALING = 1;
% Convergent rouding.
fft_config.C_HAS_ROUNDING = 1;

%% FFT
a_fft = MyXFFTScaled(fft_config, round_mode, ...
a_input_ext, im_null_data, 1);
b_fft = MyXFFTScaled(fft_config, round_mode, ...
b_input_ext, im_null_data, 1);

%% Do the complex multiplication
cNumOfBits = 2*InpuWidth+1;
```

```
CMulOut = MyCMul(a_fft, b_fft, cNumOfBits, round_mode);

CMulOutShift = CMulOut; % fftshift(CMulOut);

assert(cNumOfBits < 35);
ifft_config=fft_config;
ifft_config.C_INPUT_WIDTH = cNumOfBits;
ifft_config.C_TWIDDLE_WIDTH = cNumOfBits;

%% Do the IFFT
IFFT_Out = MyXFFTScaled(ifft_config, round_mode, ...
real(CMulOutShift), imag(CMulOutShift), 0);

%% Now convert the IFFT output into a sequence of bits
OutIntPadded = round(real(IFFT_Out*NSamples));
OutPadded = mod(OutIntPadded, 2);

% Now we can take the first 1:p bits that in bot case, are the element
% of the result vector.
Out = OutPadded(1:p);
OutInt = OutIntPadded(1:p);
end
```

Listing A.2: Xilinx MATLAB IP adaption code

```
% This function is implemented using the xilink FFT IP.
function [Out, OutRe, OutIm, ovfl, quant] = MyXFFTScaled(fft_config, roun
NSamples = length(bit_in_re);

assert(length(bit_in_re) == length(bit_in_im));

fft_quant = quantizer([fft_config.C_INPUT_WIDTH fft_config.C_INPUT_WIDTH-
if isFFT
quant_input_data_re = bit_in_re;
quant_input_data_re(quant_input_data_re == 1) = realmax(fft_quant);
quant_input_data_im = bit_in_im;
else
quant_input_data_re = bit_in_re;
quant_input_data_im = bit_in_im;
end

input_data = zeros(1, NSamples);
for i=1:length(input_data)
```

```
input_data(i)     = quant_input_data_re(i) + 1i.*quant_input_data_im(i);
end

assert(fft_config.C_HAS_SCALING==1);

% when we perform an IFFT, we do not want to scale at any stage,
% because we arleady have a 1/N intrinsic in the operation.
scale_factor = 2;
if isFFT==0
scale_factor = 0;
end

nfft = fft_config.C_NFFT_MAX;
if fft_config.C_ARCH == 1 || fft_config.C_ARCH == 3
scaling_sch = ones(1,floor(nfft/2))*scale_factor;
if mod(nfft,2) == 1
scaling_sch = [scaling_sch scale_factor/2];
end
else
scaling_sch = zeros(1, nfft);
end

% Calculaate the FFT
[fft_out, blkexp_fft, overflow_fft] = xfft_v9_1_bitacc_mex(fft_config, 0
scaling_sch, isFFT);
ovfl = overflow_fft;
quant = fft_quant;

Out = fft_out;
OutRe = real(Out);
OutIm = imag(Out);
end
```

Appendix B

VHDL Code

Listing B.1: OutConverter VHDL

```
entity OutConverter is
generic (
  constant S_LOG2: integer := 5
);
port
(
  Clk: in std_logic; Rst_n: in std_logic;
  in_data: in std_logic_vector(32 downto 0);
  out_data: out std_logic
);
end OutConverter;

architecture Behavioral of OutConverter is

-- We have format Q33.32 as the input from IFFT.
  constant IN_I_PART_WIDTH: integer := 2;
  constant IN_F_PART_WIDTH: integer := 30;
  constant IN_WIDTH: integer := IN_I_PART_WIDTH+IN_F_PART_WIDTH;

  constant EXT_WIDTH: integer := IN_WIDTH+SAMPLES_LOG2;
  constant EXT_FRAC_WIDTH: integer := IN_F_PART_WIDTH-S_LOG2;
  constant EXT_INT_WIDTH: integer := EXT_WIDTH-IN_F_PART_WIDTH+S_LOG2;

  constant PADDING_WIDTH: integer := EXT_WIDTH-IN_WIDTH;
  constant NULL_BITS: signed(PADDING_WIDTH-1 downto 0) := (others => '0'

  signal signed_val:  signed(EXT_WIDTH downto 0);
```

```
  signal shifted_val: std_logic_vector(EXT_WIDTH downto 0);

  signal data_out_int:  std_logic_vector(EXT_I_WIDTH-1 downto 0);
  signal data_out_frac: std_logic_vector(EXT_F_WIDTH-1 downto 0);
  signal value_rd:      std_logic_vector(EXT_I_WIDTH-1 downto 0);

begin

signed_val  <= NULL_BITS & signed(in_data);
shifted_val <= std_logic_vector(shift_left(signed_val, S_LOG2));

data_out_frac <= shifted_val(EXT_FRAC_WIDTH-1 downto 0);
data_out_int  <= shifted_val(EXT_WIDTH-1 downto EXT_WIDTH-EXT_I_WIDTH);

-- Now perform the rouding, any value with frac part >= 0.5 is rounded
-- to 1 always.
value_rd <= std_logic_vector(unsigned(data_out_int) + 1)
when data_out_frac(EXT_F_WIDTH-1)='1'
else data_out_int;

out_data <= value_rd(0);
end Behavioral;
```

Listing B.2: Counter VHDL Code

```
entity Counter is
generic (
  constant N: integer := 8
);
Port (
  Rst_n : in std_logic;
  Clk : in std_logic;
  En : in std_logic;
  CntRst: in std_logic; -- sync reset.
  CntValue : out unsigned(N-1 downto 0);
  TC : out std_logic
);
end Counter;

architecture Behavioral of Counter is
  constant TCMask: unsigned(N-1 downto 0) := (others => '1');
  signal Value: unsigned(N-1 downto 0) := (others => '0');
begin
```

```
UpdateProc: process(Clk, Rst_n, En)
begin
  if Rst_n='0' then
    Value <= (others => '0');
  elsif rising_edge(Clk) then
        if CntRst='1' then
          Value <= (others => '0');
        end if;
        if En='1' then
          if Value < TCMask then
                Value <= Value + 1;
          else
                Value <= (others => '0');
          end if;
        end if;
   end if;
end process;
        TC <= '1' when Value=TCMask else '0';
        CntValue <= Value;
end Behavioral;
```

Bibliography

[1] V. Goppa. A new class of linear error-correcting codes. Problemy Peredachi Informatsii, 6 no. 3:24–30, 1970. 9

[2] N. Aragon, Paulo S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, Jean-Christophe Deneuville, P. Gaborit, S. Ghosh, "BIKE: Bit Flipping Key Encapsulation"

[3] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, P. Santini, "LEDAcrypt: Low-dEnsity parity-check coDe-bAsed cryptographic systems Specification revision 3.0"

[4] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, P. Santini, "LEDAcrypt: Low-dEnsity parity-check coDe-bAsed cryptographic systems Specification revision 2.0"

[5] M. Rossi, M. Hamburg, M. Hutter, M.E. Marson, "A Side-Channel Assisted Cryptanalytic Attack Against QcBits", CHES 2017, LNCS 10529, pp. 3–23, 2017. DOI: 10.1007/978-3-319-66787-4 1

[6] R. McEliece, H. van Tilborg, "On the inherent intractability of certain coding problems", IEEE Transactions on Information Theory (Volume: 24, Issue: 3, May 1978), pp 384-386

[7] A. Vardy, "The intractability of computing the minimum distance of a code" IEEE Transactions on Information Theory (Volume: 43, Issue: 6, Nov 1997) pp. 1757-1766

[8] J. Hu, M. Baldi, P. Santini. N. Zeng, S, Ling, H. Wang, "Lightweight Key Encapsulation Using LDPC Codes on FPGAs", IEEE Transactions on Computers (Volume: 69, Issue: 3, March 1 2020), pp. 327-341

[9] K. Kobara and H. Imai, "Semantically secure McEliece public-key cryptosystems — conversions for McEliece PKC -", vol. 1992, Feb. 2001, pp. 19-35, DOI: 10.1007/3540-44586-2_2

[10] A. Barenghi and G. Pelosi, "Constant Weight Strings in Constant Time: a Building Block for Code-based Post-quantum Cryptosystems," in Proceedings of the 17th ACM International Conference on Computing Frontiers, ACM: May 2020, pp. 1-6, iSBN: 978-1-4503-7956-4/20/05

[11] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime," Proc. IEEE 56, 1107–1108 (1968)

[12] https://www.skybluetrades.net/blog/2013/12/2013-12-31-data-analysis-fft-9.html

[13] K. Koleci, P. Santini, M. Baldi, F. Chiaraluce, M. Martina, G. Masera: "Efficient Hardware Implementation of the LEDAcrypt Decoder", IEEE Access (Volume 9, Pag 66223-66240) DOI: 10.1109/ACCESS.2021.3076245

[14] J. van der Hoevena, R. Larrieub: Optimizing BIKE for the Intel Haswell and ARM Cortex-M4, 9 July 2021

[15] J. van der Hoevena, R. Larrieub: http://www.texmacs.org/joris/ffft/ffft.html, October 21, 2020

[16] J. van der Hovena: The Truncated Fourier Transform and Applications, 12 November, 2012

[17] D. Zoni, A. Galimberti, W. Fornaciari Flexible and Scalable FPGA-Oriented Design of Multipliers for Large Binary Polynomials, 21 April 2020, IEEE Access (Volume 8 Pag. 75809 - 75821), DOI: 10.1109/ACCESS.2020.2989423

[18] C. P. Rentería-Mejía and J. Velasco-Medina, Hardware Design of an NTT-Based Polynomial Multiplier, 2014 IX Southern Conference on Programmable Logic (SPL)