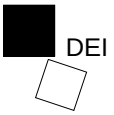
UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA



DIPARTIMENTO DI ELETTRONICA ED INFORMATICA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

TESI DI LAUREA

AN FPGA IMPLEMENTATION OF A CHAOTIC ENCRYPTION ALGORITHM

RELATORE: Ch.mo Enrico Zanoni

CORRELATORE: Meneghesso Gaudenzio

LAUREANDO: Lorenzo Cappelletti

Padova, Dicembre 2000 Anno Accademico 1999/2000

To My Parents

I would like to express my sincere thanks to Emanuel Popovici for his supervision and motivation throughout the course. I am also deeply indebted to Alfredo Sanz for his English lessons and our endless conversations. At least but not last, I blow Tiziana a gigantic kiss for her enormous support, invaluable advice and precious guidance.

Thank to Dr. Lidholm for initially stimulating my interest in the area of VHDL/FPGA. I would also like to thank Adrian, Eric, Laars, Stephen and all the other staff members of N.M.R.C. whit whom I have had contact and to the other members of U.C.C. for their friendship and hospitality upon my visits to Cork.

I am very grateful to Mum, Dad and the rest of my family for their love and encouragement throughout. Thanks Raffaele for his help despite the distance.

Cheers to Alexandra, Chiara, Filippo, Frederique, Gerald, Josune, Karen, Laura, Michele, Monika, Nicola, Paolo and all my friends and house-mates who have made Cork so enjoyable.

December 2000

Lorenzo Cappelletti
L.Cappelletti@POBoxes.com

Summary

Visual sight is an important and easy sense of communication. Recently, since computer speed, media storage and network bandwidth have seen great improvements of their performances, imaging has gained even more importance along with security, privacy and intellectual property defense.

In order for a complex imaging system to cope with these concerns, a cryptography scheme able to manage the vast amounts of data involved in image processing is required. So far, many image encryption algorithms have been restricted to the software realm, primarily due to its ease of use, ease of update, portability and flexibility. But when throughput and secret key storage security become a major issue, hardware implementations are by nature more physically secure and potentially faster.

This work aims to investigate hardware feasibility and performance of an encryption technique proposed by J. Scharinger, and based on the highly unstable non-linear dynamics of chaotic Kolmogorov flows. The algorithm is particularly attractive since only additions, subtractions and bit-shifts are required and no time-consuming operations like multiplications or exponentiation.

In this context, re-programmable devices such as FPGAs are highly attractive options since they provide hardware agility, parameterization and developing cost efficiency. The chip chosen for the implementation is a Xilinx's Virtex XCV400BG432-6, whose embedded RAM allows manipulation of small images directly in situ.

Contents

List of Figures

List of Tables

Listings

1 Introduction

After the advent of the Internet and especially nowadays, security of data and protection of privacy have become a major concern for everyone's life, although mathematicians and researchers have been trying to address this problem since the end of the World War II, when cryptology science came out from the ambit of the army to enter the Bell Laboratories. DES, RSA and PGP are only the tip of the iceberg of a vast amount of cryptographic algorithms developed by those scientists. In fact, many other cipher systems based on different mathematical properties have been designed, some of them for specific purposes like the one implemented in this thesis.

Combining together two apparently distant sciences like cryptography and chaos theory, J. Scharinger has proposed a new product cipher whose aim is to guarantee security and privacy in image and video archival applications. This encryption technique makes use, during its permutation phase, of the Kolmogorov flows which are well-known to be dynamically unstable systems. The absence of computationally heavy operations such as multiplications or divisions makes his algorithm particularly attractive for hardware implementation.

Aside from the two reports by Scharinger [10] [11], very little work seems to exist on this topic, probably because of a general scepticism by the cryptographic community about using chaotic systems for encryption purposes. Some theoretical papers have been produced by Z. Kotulski and J. Szczepañski [7] and during the Eurocrypt '91 in general. On the practical side, an attempt to design an electronic circuit which uses nonlinear dynamic approach in order to send secret messages has been described in [14, p. 335].

The objective of this thesis consists of demonstrating the feasibility of a Scharinger's cipher algorithm hardware implementation and of investigating on its features of throughput and area occupancy.

Since this is one of the first attempts to design in hardware a cryptographic system that uses chaos theory, parameterization of the circuit, rapidity in project development and flexibility during test and improvement phases are main issues. To

cope with these requirements, the couple VHDL-FPGA has been chosen, along with a divide-and-conquer approach and a software suite provided by Mentor Graphics, as rapid and efficient development tools.

1.1. Chapter Organization

The main structure of the present work is divided in four parts.

The first chapter gives a general overview of cryptography science, introducing some basic terminology, the concept of cryptosystems and two fundamental encryption techniques, permutation and substitution. The chapter continues with general remarks about chaos which are the background for Kolmogorov chaotic systems and pseudo-random sequences. Only topics that are needed for the subsequent Scharinger's algorithm description are presented.

The topic of chapter 3 is related to tools and method adopted in this thesis. A brief description of what an FPGA is constitutes the first section which in turn is followed by an explanation of concepts of entity, architecture, configuration and generic parameter proper to the VHDL language. The final discussion is for the used development suite and other chosen software tools, along with the adopted methodology.

Chapter 4 forms the main part of the corpus. Here the fundamental ideas that have been used to implement in hardware the encryption algorithm are treated. The chapter opens with a description of the chief components constituting the first realization that aimed to demonstrate the algorithm feasibility. Limits analysis and inverse Kolmogorov flow study represent the starting point for a new approach called *one-shot* and discussed in the subsequent section. The possibility offered by using more than one memory bank concludes the chapter.

The last chapter is devolved to performance evaluation. For both first and one-shot implementation, consume of physical resources and throughput capacity are reported and summarized in two tables.

2 Description of a Chaotic Cipher

Background and description of the cryptographic algorithm presented in this work are the main subjects to which this chapter is devolved. The first section is an introduction of cryptology science, its terminology and a brief definition and classification of cryptosystems, whereas the next one is related to chaos and chaotic systems, particularly Kolmogorov flows. At the end of the chapter, these two topics will be joint together to describe in detail the actual algorithm.

2.1. General Remarks About Cryptography

In this section rudimentary concepts of cryptology science are introduced. The first subsection defines technical terms common in this discipline and used throughout the rest of the work, while subsection 2.1.2 distinguishes different points of view from which a cryptosystem can be seen. The objective of this section is to find a collocation in the cryptography panorama for the algorithm implemented in hardware for this work.

2.1.1. Basic Terminology

The word *cryptography* refers to the science of keeping secrecy of messages exchanged between a sender and a receiver over an insecure channel. The objective is achieved by encoding data so that it can only be decoded by specific individuals.

The original message M being wanted to be sent is called *plaintext* since it is clearly intelligible, whereas the term used to refer to the message C being transited over an insecure channel is *ciphertext*. The process $\mathcal{E}$ of transforming a plaintext into a ciphertext is called *encryption*, while the opposite procedure $\mathcal{D}$ that turns a ciphertext into a plaintext at the receiver's side is said *decryption*. In symbols

$$\begin{aligned} \mathcal{E}(M) &= C \\ \mathcal{D}(C) &= M \end{aligned}$$

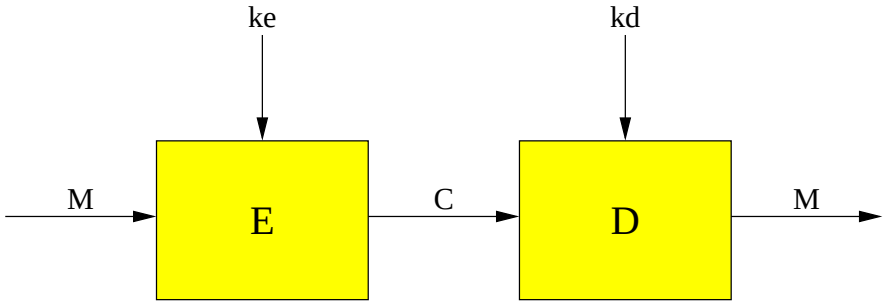


Figure 2.1: Encryption and decryption with a key.

A *cryptographic algorithm* is composed of the mathematical function used for encryption and its related inverse-function for decryption. A cryptographic algorithm is some times referred as *cipher*. The security of an algorithm can rely on the secrecy of its function, when quality, standardization and mass utilization is not a concern [12]. Where these restrictions can not be tolerated (basically in any practical situation), the problem is solved by means of a *key*, denoted with k. This key might be any one of a large number of values, which all together form the *keyspace* $\mathbb{K}$. Two different keys k_e and k_d for encryption and decryption respectively might be used. Once more in symbols:

$$\begin{aligned} \mathcal{E}_{k_e}(M) &= C \\ \mathcal{D}_{k_d}(C) &= M \end{aligned}$$

Finally, a *cryptosystem* is an algorithm plus all possible plaintexts, ciphertexts and keys.

2.1.2. Classification of Cryptosystems

As seen, definitions and symbology introduced in the previous section can be summarized by the general concept of cryptosystem, whose picture is shown in figure 2.1. In regard to the kind of distribution method established for the keys, the way a cipher treats the plaintext and the type of implementation support chosen, a cryptosystem can be seen under several points of view. A look at them will allow us to have an idea of the collocation of the present algorithm within the vast field of cryptography.

Distribution of the Secret Key

The first big classification to which cryptographic algorithms might be undertaken is the distinction between the methods with which keys are distributed. When encryption key k_e and decryption key k_d are identical, i.e. $k_e \equiv k_d$, sender and receiver

must agree on a secure channel through which transmitting the key without anybody else finding out. This is the most extensively used method and is often accompanied by the adjective *symmetric* because of the equivalence of the keys. Contrarily, the *asymmetric* method makes use of a pair of keys for each individual — one public and the other private[1].

Block and Stream Cipher

Another big classification for cryptographic algorithms consists of subdividing ciphers into two categories: stream ciphers and block ciphers. A *stream cipher* is so called because it works on a stream of data, normally one bit (but some time also one byte or $32\,\mathrm{bit}$) at a time: as soon as a new value of plaintext arrives, the correspondent cipher value is computed. On the other side, a *block cipher* operates on the plaintext one block at a time: a new block of ciphertext can not be evaluated until the previous block is finished. Moreover, a block cipher will encrypt the same plaintext with the same key always to the same ciphertext, while for a stream cryptosystem the output depends also on the history of the cipher — this leads to the problem of *synchronization* between encryption and decryption processes.

A cipher can operate in several cryptographic modes in regard to the way the plaintext, key and ciphertext interact with each other. Electronic Codebook (ECB) mode represents the more straightforward and simple solution for a block cipher. Once the key is fixed, the system will always encrypt the same block of plaintext into the same block of ciphertext, without regard to other parameters. This mode can be thought of as a double entry look-up table. While implementations can be extremely fast, this mode is also very memory demanding since a table is necessary for every couple of plain- and ciphertext and for each key $k \in \mathbb{K}$. The counterpart of ECB is the Cipher Block Chaining (CBC) mode in which new ciphertext blocks depend, by means of a sort of feedback mechanism on previous outputs.

Besides ECB and CBC, Ciphertext Feedback (CFB) represents a mode to run block ciphers as stream ciphers. This statement means that output values from a cryptosystem are serialized as in a stream cipher, but rely somehow on the previous computed values as in a block cipher. The mechanism used to realize this mode generally consists of a shift register into which new values are pushed and on which the encryption algorithm depends. Another solution consists of using a Pseudo-Random Number Generator (PRNG). It is worth noticing that in any case the mechanism has to be initialized with an *initialization vector* which concurs, besides the key, to effect the encipherment output for a given sequence of plaintext's data. This means that the ciphertext depends on previous blocks such as a stream cipher. Nevertheless, if the initialization vector depends on the key and the keys k_e

[1]For further reference see W. Diffie and M. E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, v. IT-22, n. 6, Nov. 1976, pp. 644-654.

and k_d match, there are no synchronization problems and ciphertext can be correctly deciphered.

Hardware Versus Software Implementation

The kind of algorithm classes being explained in this subsection do not have well-defined boundaries, that is, an algorithm can be moved from a class to another due to a technological improvement or a smarter implementation. In fact, any cryptographic algorithm might potentially be designed either for an hardware project and for a software program, but many times the involved operations render one of the two solutions (and sometimes both) impracticable or inconvenient.

A software implementation has, on its side, flexibility throughout different applications, portability from one platform to another, ease of use and ease of upgrade of the binary or source code. The disadvantages are in speed — especially if the algorithm belongs to the category of stream ciphers — ease of modification and manipulation by third party.

On the other side, a hardware implementation suffers mainly of deficiency in mathematical abilities, since operations as multiplications and divisions are normally difficult or cost prohibitive for realization. Nonetheless, the advantages abundantly overcome the inadequacies. The first is speed. Dedicated hardware — possibly another chip beside the main CPU — will always win a speed race against a general-purpose processor, especially if the cryptographic algorithm is a sort of stream cipher.

Besides speed, security reasons play a great role. A dedicated hardware has got a physical barrier to be surmounted before reading internal variables. Codes can be burned into the chip and tamper-proof can prevent someone from modifying a hardware encryption device [12]: chemical substances can be used to destruct the chip's logic in case a third party accesses the interior.

A final reason relies on ease installation as simple device between two existent peripherals.

2.1.3. Confusion and Diffusion

The main objective of cryptology is to achieve the perfect secrecy [12] by which no information of the plaintext can be extracted from the ciphertext. The only cryptographic algorithm able of such a performance is called *one-time pad*[2] where each character of the plain-message is ciphered with exactly one random number picked out from a truly random sequence which can be used only once for only one message.

[2]See for example D. Kahn, *The Codebreakers: The Story of Secret Writing*, New York, Macmillan Publishing Co., 1967

Since the one-time pad only has a theoretical validity and any other cipher is an approximation of it, every ciphertext unavoidably yields some information about the corresponding plaintext. In part this is also due to the redundancy to which a natural language is subjected, i.e. the fact that a plaintext contains more symbols than those necessary to provide the same amount of information. A good algorithm will tend to reduce redundancy to a minimum.

According to [12] who cites Shannon[3] the two basic techniques for conceiving redundancies, beside using a compression algorithm, are called confusion and diffusion. *Confusion* seeks to reduce the correlation between the input plaintext and the output ciphertext. The task is generally accomplished substituting every fundamental block of data for another one according to the rules dictated by the cryptographic algorithm. Despite this, repetitions or well-known sequence of blocks in the plaintext are still kept at the output. This problem is addressed by *diffusion*: a data on the input block is transposed to other coordinates on the output block. Put in another way, diffusion changes the position of data, while, during a confusion process, the data itself is modified. It is to be observed that diffusion implies a block cipher, whereas confusion can deal with streams of data, as well.

2.2. Chaos and Kolmogorov Flows

This section is opened by a simple introduction to chaos theory which is followed by a brief definition of two characteristics that distinguish some chaotic systems: ergodicy and mixing-property. A discussion about Kolmogorov flows related to the application for this work closes the section.

2.2.1. Basic Introduction to Chaos

Uncountable definitions have been given in seeking to formally describe chaos and chaos theory as a branch of mathematics. In this ambit, only a qualitative description suitable for comprehension of the algorithm highlighted in section 2.3 is faced.

Let's start this introduction to chaos speaking about the mathematical problem of having two bodies in space. Solving the system of differential equations that arise from the application of Newton's law, the trajectories of the two bodies are completely described in terms of space and time by the well-known gravity law. This means that, given the initial conditions, all the parameters that govern the system — i.e. position, velocity and acceleration — can be determined for each of the two bodies at any time. Because of this, the system is said to be *deterministic*.

When a third body is introduced into the system, the property of the system of having a closed solution no longer holds to be true. The system of differential equa-

[3] C. E. Shannon, "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, v. 28, n. 4, 1949, pp. 656-715

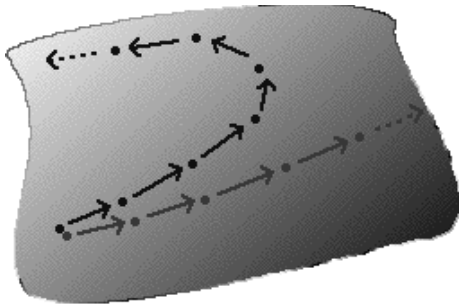


Figure 2.2: Divergence due to sensitive dependence to initial conditions.

tions still describe completely the behaviour of the three bodies, but the knowledge of the position of each element in the system at a given time t can be calculated only by iterating the differential equations written in discrete form starting from the initial condition up to time t. In other words, the output from the previous step is the input of the next iteration and a general solution can not be expressed using only one equation. The system belongs to the category of *non-linear systems*.

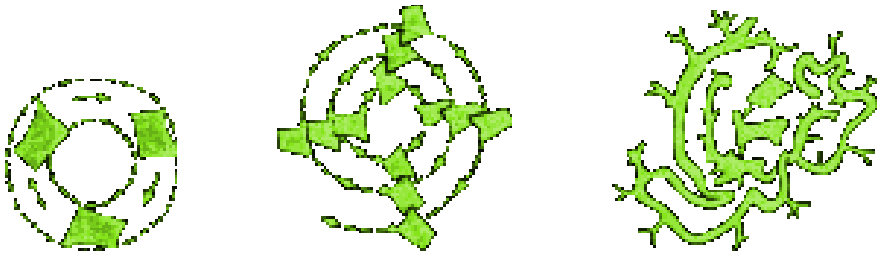
A third characteristic distinguishes the system of three bodies in space: density. A system is *dense* when, without regard to the distance between two legal points in the system, there always exists a third point between them.

These properties lead, for a reason that will not be discussed here, to another proposition by which chaotic systems are referred — *sensitive dependence on initial conditions*. To provide a general idea of such dependence without entering into the mathematical realm, let's suppose to record the trajectory followed by the system starting with the initial condition $\mathbf{t}_0$. Let's also suppose to choose another condition $\mathbf{t}_1$ very close, thank the denseness property, to $\mathbf{t}_0$. If the trajectory of $\mathbf{t}_1$ diverges sensitively from the one of $\mathbf{t}_0$ (see figure 2.2), it is impossible, chosen a third initial condition $\mathbf{t}_2$, to predict what trajectory the system will follow. In this sense, the system appears to behave *randomly*.

Summarizing the notions so far introduced, a *chaotic system* might be defined as a non-linear deterministic system so sensitive to initial conditions that it appears random. It is remarkable observing that here two antonyms as deterministic and random are used in the same sentence, since chaos theory effectively forms a bridge between two dissimilar sciences — mathematics and probability.

2.2.2. Kolmogorov Chaotic Systems

In the prior subsection a general view of what a chaotic system represents has been outlined. Here the discussion is going ahead showing that the Kolmogorov flows belong to a hierarchy of chaotic systems with specific characteristics.



(a) (b) (c)

Figure 2.3: The remarkable differences in behaviour in "phase space" between a simple system (a), a so-called ergodic system (b) and a mixed ergodic system (c) which is chaotic.

Ergodic- and Mixing-Properties

It is possible to imagine a chaotic system as a point tracing a trajectory in space. The rules governing the system often keep the point within a region $\mathbb{E}$ that is called the *phase space* of the system. In general, the function that describes the system maps values taken from $\mathbb{E}$ in itself, but not necessarily all values from the phase space are mapped so that all $\mathbb{E}$ is covered. If the set of chaotic system is restricted to systems whose output coincides with the whole phase space $\mathbb{E}$, we are working with a subset that takes the name of *ergodic systems*.

At a higher level of the hierarchy, there exists another class of chaotic systems which possesses a new property called *mixing property*. This name refers to the particular characteristic that some ergodic systems show to have. Exploiting a simple comparison to clear the idea, it can be said that a system is mixing when it spreads out into ever finer fibres until it covers the entire phase space such as a drop of ink spreads out chaotically in water (see figure 2.3(c)). This behaviour is due to the fact that trajectories diverge from each other exponentially fast.

K-Flows

Continuing to go up the hierarchy of the randomness, a smaller subset of systems showing mixing and ergodic properties can be identified. This new sort of systems are called *K-flows*. The "K" stands for Kolmogorov[4], a Russian mathematician whose work influenced several branches of modern mathematics, while the noun "flow" comes out from the fact that this kind of systems are widely spread among researches where collisions between particles dominate the dynamics. K-flows distinguish from other chaotic systems since they possess the remarkable property that

[4]Born April 25, 1903, in Tambov, Russia and died October 20, 1987, in Moscow.

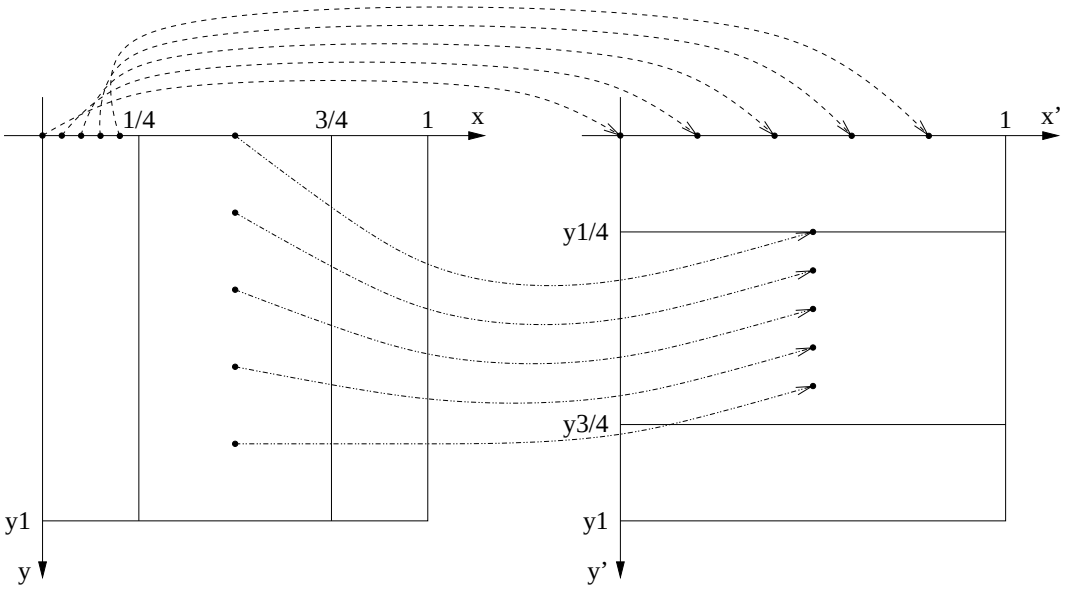


Figure 2.4: Some sample points mapped by T_π.

their next measurement can not be predicted even after an infinite number of prior measurements.

For this work the Kolmogorov flows described in [10] and [11] will be used. The continuous version T_π of this system can be expressed using the following mathematical notation. Let $\pi = (p_1, p_2, \ldots, p_k)$ be a sequence of numbers with the properties that $p_i \in \mathbb{R}$, $0 < p_i < 1$ and $\sum_i p_i = 1$, where $i = 1, \ldots, k$. Let also the unit-square $\mathbb{E} = [0, 1) \times [0, 1)$ denote the phase space of the system and F_s be defined as follow

$$F_s = \begin{cases} 0 & \text{for } s = 1 \\ p_1 + \ldots + p_{s-1} & \text{for } s = 2, \ldots, k \end{cases} \tag{2.1}$$

Then the application $T_\pi : \mathbb{E} \to \mathbb{E}$ on $(x, y) \in [F_s, F_s + p_s) \times [0, 1)$ is expressed by the following relation

$$T_\pi(x, y) = \left(\frac{1}{p_s}(x - F_s)\,,\ yp_s + F_s \right) \tag{2.2}$$

and $T_\pi(x, y) \in [0, 1) \times [F_s, F_s + p_s)$.

In other words, the phase space $\mathbb{E}$ is divided into *horizontal* strips of dimensions $1 \times p_s$ and bounded on the left and on the right by F_s and $F_s + p_s$, respectively. Every point (x, y) of each strip is mapped according to T_π into a *vertical* strip of size $p_s \times 1$ (see figure 2.4).

In order to deal with a digital circuit, the application T_π just defined has to be modified in a way that works with discrete values. This task can be accomplished thinking about the phase space as a subset of natural numbers, i.e. $\mathbb{E}_n =$

$[0, n) \times [0, n) \subseteq \mathbb{N}^2$, where n is the dimension of the square phase space. If $\delta = (n1, n2, \ldots, n_k)$ is a list of positive integers that holds the properties $0 < n_i < n$ and $\sum_i n_i = n$, where $i = 1, \ldots, k$, then the *discrete version* of the K-flow defined by eq. (2.2) will be identified by $T_{n,\delta}$.

To be suitable for our purposes, the application $T_{n,\delta} : \mathbb{E}_n \to \mathbb{E}_n$ just introduced has to possess another property. In order to avoid a generic division operation induced by the term $1/p_s$, n_i must divide n without any reminder. Thus, a new sequence of positive integers can be defined as $q_s = n/n_s$, where $s = 1, 2, \ldots, k$. If N_s is still the left border of the s-th vertical strip

$$N_s = \begin{cases} 0 & \text{for } s = 1 \\ n_1 + \ldots + n_{s-1} & \text{for } s = 2, \ldots, k \end{cases} \tag{2.3}$$

then the application on $(x, y) \in [N_s, N_s + n_s) \times [0, 1)$ that approximates the best the corresponding chaotic continuous system is given, according to [10], by the following relation

$$T_{n,\delta}(x, y) = \Big(q_s(x - N_s) + (y \bmod q_s)\,,\ (y \operatorname{div} q_s) + N_s\Big) \tag{2.4}$$

and $T_{n,\delta}(x, y) \in [0, 1) \times [N_s, N_s + n_s)$.

As we are concerned of security purpose, the dimension of the square phase space must be as big as possible. In fact, increasing n results in a greater number of different valid parameters δ from which it is possible to choose.

The operations division and modulus of eq. (2.4) are between integer numbers. Their definition is as follow

$$a = b \cdot d + r \quad \Rightarrow \quad \begin{cases} a \operatorname{div} b &= d \\ a \bmod b &= r \end{cases} \tag{2.5}$$

where a, b, d and r are natural numbers, $r < b$ and $b \neq 0$.

Finally, it can be observed that the K-flow $T_{n,\delta}$ is bijective. Defining the upper border of the s-th horizontal strip

$$M_s = \begin{cases} 0 & \text{for } s = 1 \\ n_1 + \ldots + n_{s-1} & \text{for } s = 2, \ldots, k \end{cases} \tag{2.6}$$

points $(x', y') \in [0, 1) \times [M_s, M_s + n_s)$ can be mapped back to $[M_s, M_s + n_s) \times [0, 1)$ by the inverse $T^{-1}_{n,\delta}$ given by

$$T^{-1}_{n,\delta}(x', y') = \Big((x' \operatorname{div} q_s) + M_s\,,\ qs(y' - M_s) + (x' \bmod q_s)\Big) \tag{2.7}$$

2.2.3. Pseudo-Random Sequences

The ciphering algorithm that is going to be described in section 2.3 uses random numbers in two parts of the process. Therefore, it is worth spending some words on what pseudo-random-sequence generation is without going too much into details.

A real random sequence is a sequence of numbers which shows a property of real randomness (i.e. without any correlation among the generated numbers) and which cannot be reliably reproduced [12]. The nature offers a vast variety of real random sequences. Unfortunately, these sequences cannot be exploited since computers and digital systems in general are deterministic. In fact, any finite state machine can only be in a finite number of states. This implies that no random number generator can produce a real random sequence, but only pseudo-random sequences. A *pseudo-random sequence* is a sequence where random numbers repeat after a certain number of generations, called *period*. The finite state machine has its initial state set by a key, frequently referred to as a *seed*. Such finite state machines are named Pseudo-Random Number Generator (PRNG).

In order to be suitable for cryptographic applications, a PRNG should possesses two properties. The first property is related to randomness. The PRNG should have a period long enough to pass all the statistical tests of randomness available. The second property relies on unpredictability. It should be computationally infeasible to calculate the next pseudo-random number given the entire previous produced sequence.

2.3. Algorithm Behavioural Description

This section describes, with the auxil of the concepts introduced in the prior sections, the cryptographic algorithm related to this work.

The general block diagram of the cipher under consideration is shown in figure 2.5 on the following page. The passphrase of $6,400\,\mathrm{bit}$ is unique for both encryption and decryption, so that, according to what exposed in section 2, the algorithm is symmetric. More interestingly, the algorithm works on square plaintexts (formally images) and therefore it belongs to the category of block ciphers. The only constraint to which images are subjected is that the block length has to be an integral power of 2. The cipher performs an encryption iterating the same algorithm for r rounds. According to the author [11], a number of rounds at least equal to 12 is recommended.

Permutation

The permutation component is responsible for the actualization of the concept *diffusion* outlined in section 2.1.3. Each data which composes the plainblock at the input is transposed to a new position at the output. This transformation follows the

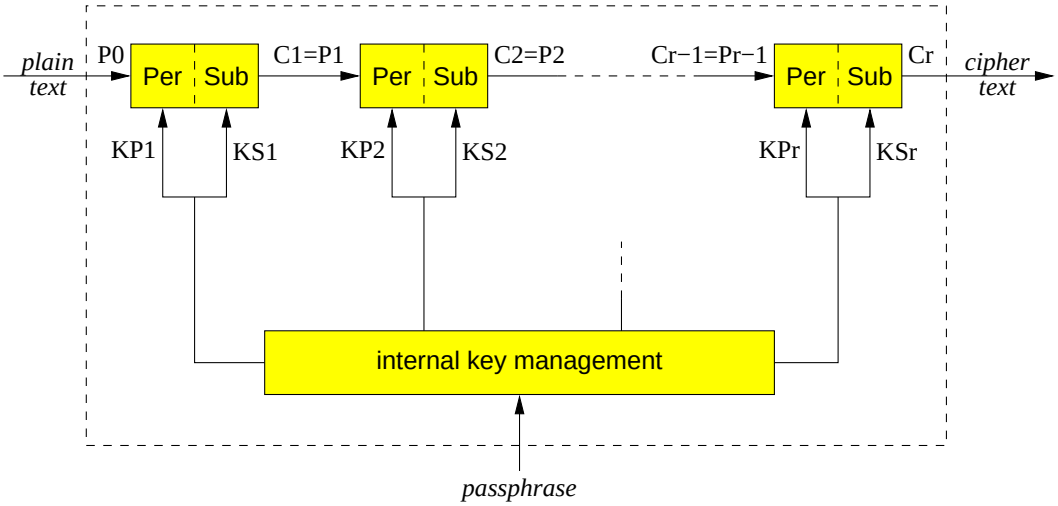


Figure 2.5: General block diagram of the cipher algorithm.

rules dictated by the chaotic Kolmogorov flow $T_{n,\delta}$ formalized by eq. (2.4) on page 11.

It is opportune to remember that the K-flow $T_{n,\delta}$ just mentioned depends on two parameters. The first parameter is n which represents in this context the size of the image. The second parameter is the sequence $\delta = \{n_1, n_2, \ldots, n_k\}$ into which the image is partitioned. Because n is a power of 2 and because each n_s ($s = 1, \ldots, k$) must divide n without any reminder, the two major operations of division and modulo that compose the application $T_{n,\delta}$ are between a natural number in the range $[0 \ldots n)$ and a number that is a power of two, i.e. q_s. Common partitions are on the form $\{1/2, 1/2\}$, $\{1/4, 1/2, 1/4\}$ and so on, which, for the discrete case, translate to $\{2, 2\}$, $\{4, 2, 4\}$.

Substitution

Confusion is accomplished by the substitution component. Each data $\mathrm{p}(i)$ coming from the permutation block at time i is combined with the previous values to compute a new cipherdata $\mathrm{c}(i)$ that will constitute the output encrypted block C_i. Since no re-arrangement of the position each data takes within blocks are performed, but only a change in the value of data, substitution component clearly belongs to the category of stream ciphers defined in section 2.1.2.

The mathematical operation that copes with the confusion is formalized by the following equation

$$\mathrm{c}(i) = (\mathrm{p}(i) + \mathrm{prsp}(i) + \mathrm{prsc}(i)) \mod 2^{32} \tag{2.8}$$

The quantities $\mathrm{prsp}(i)$ and $\mathrm{prsp}(i)$ represent the pseudo-random sequences for plaintext and ciphertext, respectively. Their values are computed by means of the follow-

ing relations

$$\text{prsp}(i) = (\text{p}(i-24) - \text{p}(i-37) - \text{cp}(i)) \bmod 2^{32} \tag{2.9}$$

$$\text{cp}(i+1) = \begin{cases} 1 & \text{if p}(i-24) - \text{p}(i-37) - \text{cp}(i) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

$$\text{prsc}(i) = (\text{c}(i-24) - \text{c}(i-37) - \text{cc}(i)) \bmod 2^{32} \tag{2.11}$$

$$\text{cc}(i+1) = \begin{cases} 1 & \text{if c}(i-24) - \text{c}(i-37) - \text{cc}(i) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

These expressions reassemble a type of PRNG (see section 2.2.3) described in [8]. Specifically, the author of the present algorithm chose the recommended Subtract With Borrow (SWB) pseudo-random number generator whose characteristic equation is given by $x_n = x_{n-24} - x_{n-37} - c \bmod 2^{32}$.The integer numbers 24 and 37 clarify the constant present in eq.s (2.9) to (2.12).

It is important to notice that the considered SWB generator uses a seed vector $x = (x_1, \ldots, x_{37}, c)$ constituted of 37 elements of $32\,\text{bit}$ each and one element of 1 bit. This vector seed has to be initialized with a not-null vector in order for the pseudo-random generator to work properly.

Key Management

The passphrase in figure 2.5 on the page before represents the input key with which the sensitive data are protected. As indicated by the author of the algorithm, the passphrase is limited to at most $200 \cdot 32 = 6,400\,\text{bit}$ that directly feed a 250 32-bit element vector. Therefore, the remaining 50 cells are initialized with random values.

The above-mentioned vector belongs to a PRNG called “R250” [6]. This component is responsible of delivering on demand pseudo-random numbers to the prior two components. Precisely, since the author of the algorithm did not exactly specify the procedure, it has been decided that the first number serves to chose the δ-partition for the permutation and the next $2 \cdot 37$ numbers fill up the substitution’s vectors.

3 Tools and Methods

As the first aim of this work is testing the feasibility and performance of a new algorithm, great flexibility and versatility are required in the design description. The tools and methodology being used are able to aid designers to deal with architectural complexity and are capable of easy system parameterization and result comparison. Moreover, they should limit the time and resources needed for circuit verification.

There are two main basic hardware design methodology currently available: schematic and language based designs. For the purpose of this work the latter was chosen. In fact, the schematic based design, despite of better optimization implementation in terms of both area and speed, does not fulfill the requirements of simplicity and speed of designing stated above. Nevertheless, the language based counterpart suffers from the synthesis tool used as well as the code style with which designs being synthesized are written, as is the case for software developing compilers. These two factors can potentially lead to variances when comparing synthesis tool outputs.

Before going into details, it is worth noticing that the language based design choice relies primarily on two components: the hardware side, represented by the promising FPGA technology, and the programming language counterpart, represented for the current project by VHDL. The former is discussed in the first section, where the more relevant features, from this project point of view, will be illustrated; the latter is introduced in the next section that looks at basic and fundamental concepts in VHDL. Finally, the methodology which has being followed to design the present algorithm will be elucidated.

3.1. FPGA

The Field-Programmable Gate Array (FPGA) is a type of programmable device. Programmable devices are a class of general-purpose chips that can be configured for a wide variety of applications, having capability of implementing the logic of

hundreds or thousands of discrete devices. PROM, EPROM and EEPROM are the oldest members of that class, while PLD and PLA represent more recent attempts to provide the end-user with an on-site customization using programming hardware. This technology, however, is limited by the power consumption and time delay typical of these devices.

In order to address these problems and achieve a greater gate density, MPGAs showed up in the industry market. An MPGA consists of a base of pre-designed transistors with customized wiring for each design. The wiring is built during the manufacturing process, so each design requires expensive custom masks and long turnaround time.

3.1.1. Why FPGAs

FPGAs offer the benefits of both PLD and MPGA. In fact, the computing core of an FPGA consists, in general, of a highly complex re-programmable matrix of logic IC, registers, RAM and routing resources. These can be used for performing logical and arithmetical operations, for variable storage and to transfer data between different parts of the system. Furthermore, because no CPU governs the entire chip and no sequential instructions have to be processed, typically thousands of operations can be performed in parallel on an FPGA during every clock cycle. Though the clock speed of FPGAs ($20 \div 130\,\mathrm{MHz}$) is lower than of current RISC systems ($100 \div 500\,\mathrm{MHz}$) the resulting performances can be extremely satisfactory in many applications like image processing, artificial neural networks and data encryption, as reported in [3].

An implementation on a FPGA is even more attractive when considering a cryptosystem. While a software solution include ease of use, ease of upgrade, portability, and flexibility, a cryptographic algorithm and its associated keys that are implemented in hardware are, by nature, more physically secure as they cannot easily be read or modified by an outside eavesdropper.

3.1.2. The SRAM Based FPGA

The SRAM FPGA gets its name from the fact that programmable connections are made using pass-transistors, transmission gates, or multiplexers that are controlled by Static RAM (SRAM) cells. The advantage of this technology resides in the fact that it allows fast in-circuit reconfiguration. The major disadvantage, however, is the size of the chip required by the RAM technology.

The FPGA has three major configurable elements: Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), and interconnects. The CLBs provide the functional elements for constructing user's logic (see figure 3.1); it normally comprises of a Look Up Table (LUT) with 4 inputs and a flip-flop output. The IOBs

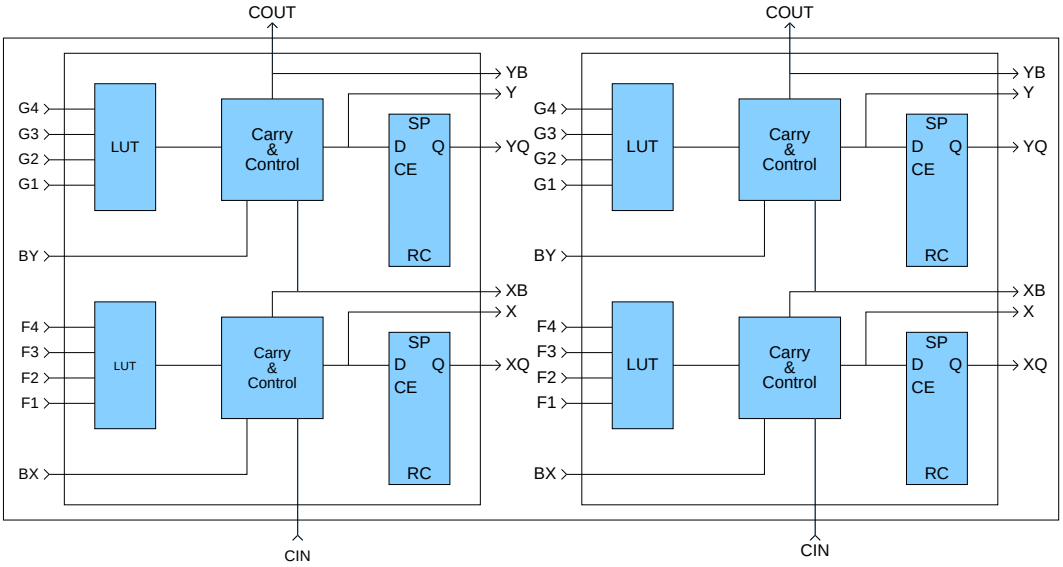


Figure 3.1: 2-slice Virtex-E CLB (Courtesy of Xilinx Inc.)

provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks. Customized configuration is established by programming internal static memory cells that determine the logic functions and internal connections implemented in the FPGA.

The XCV400E, belonging to the Virtex-E family from Xilinx, is the FPGA chosen for this implementation. Its features, summarized in table 3.1, are generous in gates and I/O pins allowing the designer long synthesis waits and troublesome pin assignments.

Besides the common structure depicted above, this FPGA family provides dig-

XCV400E	
System Gates	569,952
Logic Gates	129,600
CLB Arrays	40×60
Logic Cells	10,800
Differential I/O pairs	183
User I/Os	404
BlockRAM Bits	163,840
Distributed RAM Bits	153,600

Table 3.1: Virtex-E FPGA XCV400E features from [16]

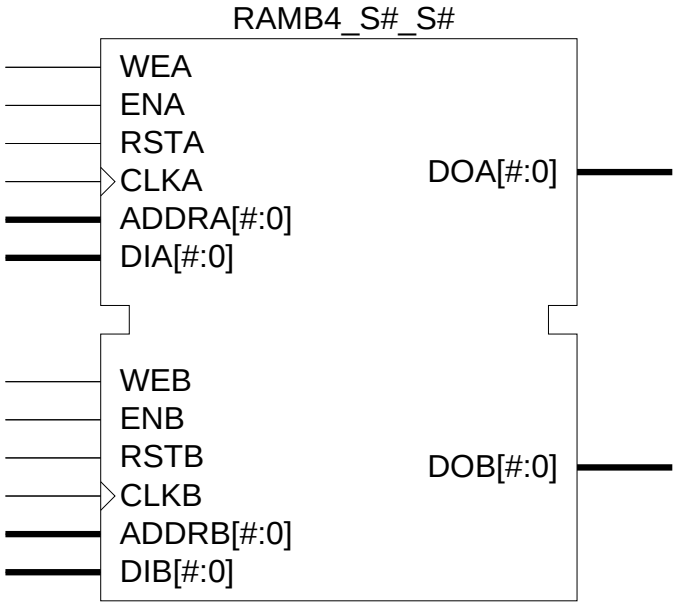


Figure 3.2: Symbol for the Block SelectRAM+ memory block (Courtesy of Xilinx Inc.)

ital Delay-Locked Loops (DLLs), dedicated electronics to deal with clock distribution problems, and tristate buffers (BUFTs), for driving on-chip busses. But the major feature, extensively used especially in the last part of this project, is the Virtex Block SelectRAM+ technology [15].

A Block SelectRAM+ is a real synchronous four CLBs high memory block which extends the full height of the chip, immediately adjacent to a CLB column location. In the chip selected there are 40 blocks for a total of 163,840 Block SelectRAM bits. Each cell, illustrated in figure 3.2, can be seen as a fully synchronous dual-ported $4096\,\mathrm{bit}$ RAM with independent control signals for each port. This structure is suitable only for shallow memory arrays, but it can simplify enormously the project. Adopting this solution, in fact, the designer is free of protocol trouble between the chip and an external memory bank, while every read/write request can be fulfilled in 2 clock cycles.

3.2. VHDL

VHSIC Hardware Description Language (VHDL) is a language for describing digital electronic systems. Its nested acronym is due to the fact that it arose out of the United State government's Very High-Speed Integrated Circuit (VHSIC) program and to the need for a standard and well-defined language to describe digital Integrated Circuits (ICs). It was subsequently embraced by the Institute of Electrical

and Electronics Engineers (IEEE), which adopted it in the form of the IEEE Standard 1076 *Standard VHDL Language Reference Manual* in 1987 and revised it in 1992. This is the latest official version and is often referred to as VHDL-93.

The main difference between VHDL and another programming language more widely used in computer science as, for example, 'C', is that VHDL enables the execution of concurrent statements. In contrast with a sequential statement, a *concurrent statement* is so called because conceptually it can be activated to perform its task together with any other concurrent statements. Concurrency is useful for modeling the way real circuits behave (you can think of any logic port such an SR flip-flop), but can potentially lead a neophyte to misunderstandings and unpredicted results.

The real power of VHDL, anyhow, resides in the possibility to look at the same system with different levels of abstraction, where different pieces of information play different roles in describing that system's model. If we start with a requirements document for our system, we can decompose it in several components that jointly fulfill the requirements proposed. Each of these components can be in turn decomposed into subcomponents, for which design we need to pay attention to a smaller amount of relevant information than being overwhelmed by masses of detail. Moreover, the design we are focusing on can be described using abstract data types or going closer to the electronic rules. The result of this process is a hierarchically composed system, built from primitive elements.

VHDL attains this goal with a rigorous formalism that comprises three fundamental concepts: entity, architecture and configuration.

3.2.1. Entities

In VHDL an entity is the equivalent of an IC package in electronics. A package is defined by its name and the number and nature of ports it uses to exchange data and interact with external circuits. It is not relevant to the function that the component performs, the resources it uses or the complexity by which it is characterized. What matters is, using the language's terminology, the number of *ports*, by which information is fed into and out the circuit, the *type*, which specifies the kind of information that can be communicated, and the *mode* which specifies whether the information flows into or out through the entity's ports.

At this point it is important to have clear in mind the function and number of signals for the entity being defined. Otherwise, when all the components are assembled together to form the whole system, the code can be subjected to several and problematic changes. Stated in a different way, the system has to be carefully split in subcomponents for which the major number of details has to be planned to avoid subsequently cumbersome fixes.

The entity concept easily allows the programmer to apply the Latin motto,

widely used in engineering science, *divide et impera*[1], not only because a complex system can be split in smaller and more tractable sub-systems, but since it also permits to face structural problems explained above during a well-distinct phase and postpone descriptive problems to a latter stage by using architectures.

It is worth observing that the last VHDL standard approved by IEEE and abbreviated to VHDL-93 permits a simpler and quicker method of component instantiation, formally known as *direct instantiation* [13, pag. 154], for which neither component declaration nor configuration are needed.

3.2.2. Architectures

An architecture, continuing with the analogy of an IC, is equivalent to the internal electronic circuit that performs the function for which the component has been designed.

At this stage all the signals the entity uses to communicate with the external world are defined and unchangeable. Inside the entity, instead, the programmer can write any type of statements allowed by the VHDL syntax, like other components, representing sub-circuits, other signals, to connect them, or processes, which contain sequential statements operating on values. The purpose is only that to apply some operations to data on input ports and generate some results to assign to output ports.

Within an architecture the programmer can concentrate on the descriptive part of the system, but the Latin *divide et impera* motto can be applied once again. VHDL, indeed, offers the opportunity to choose the desired abstraction level. Thus, for a first behavioural version, aiming to investigate a circuit feasibility or correctness, code lines containing high level data types or cumbersome mathematical formulas and few electronic details will be advisable. Then, a subsequent description can go further and optimize the implementation with clever solutions for better performance, smaller area and fulfillment of constraints.

3.2.3. Configurations

In the previous two subsections we saw that the same entity can be usefully described, unless the functional behaviours match, with different approaches that correspond to different architectures. Due to this correspondence, nothing prevents us from choosing an architecture of an entity regardless of architectures chosen for other entities. The *configuration* declaration permits of assembling the system up to our goal, architectures availability or simply taste.

To make the idea clearer, let suppose we have n entities $\{E_1, \ldots, E_n\}$. Let also suppose each of them has got two architectures, A_1^i and A_2^i, with $i \in \{1, \ldots, n\}$. If

[1] In English it can be translated to “divide and rule” or “divide and conquer”.

the entire system is only composed of all the entities $\{E_1, \ldots, E_n\}$, combining the two architectures for each of them we can virtually have 2^n different representation of the same system.

In practice the situation can be slightly different. Indeed, we can have a behavioural and abstract architecture that does not use the clock signal, common to almost all the digital systems. Such an asynchronous architecture cannot be sided by a synchronous version of another component, because they do not share a piece of information carried by the clock signal. Nevertheless, the configuration facility is highly attractive since a system that comprises of only behavioural architectures can act as reference model for correctness in successive implementations.

3.2.4. Generic Parameters

The *generic interface list* is another powerful tool provided by VHDL language. Its aim is to allow parametrized descriptions of entities. In fact, from an architecture point of view, a generic element is seen as a constant whose visibility extends into the whole architecture body corresponding to that entity declaration. The actual generic value, on the other hand, is defined in the entity that lies on a upper level in the hierarchy and is passed down to the component as a parameter.

To demonstrate the concept, we can consider a situation where we need a 2-input multiplexer for which we design an architecture. If subsequently we need a multiplexer with 4 inputs and do not use the generic facility, we have to write a new entity with two more pins and a new architecture. Instead, with a generic element in the entity definition and an opportune descriptive code, we can change only one parameter during the component instantiation and still use the same multiplexer implementation.

This approach has several advantages. The entity can result in a more dynamic and versatile component. The reduced number of lines permits handling with less architectures. It also simplifies code maintenance since, if a logical error is found, only one implementation needs being modified. At last but not least, a parametric description normally has a more regular structure that may be better synthesized by the synthesis tool.

The drawback is the increased complexity in the architecture design which requires better summarizing skills and wider perspectives. In fact, a parametric entity leads the programmer to face problems due to out-of-range values or exceptions not contemplated in the original version. In the end, especially after some practice, the advantages will certainly prevail.

3.3. Software Tools

The following section briefly describes the software tools used to write, compile, test and synthesize the project presented in chapter 2.

To develop and debug over 7,100 (lines nearly 13,000 with comments and blank lines included) of the project code several, commercial programs played an important role. Unfortunately the Graphical User Interface (GUI) provided has been found to be computationally heavy on the equipped workstations. Thus, to make the procedures semi-automatic, some scripts in Tcl and Makefile languages have been provided, having the programmer familiarize also with these marginal but anyway important tools.

3.3.1. Editing

To write and debug the different types of source code, XEmacs, version 21.1, was chosen primarily for its versatility. This powerful and extensible text editor with full GUI support includes nice features like *word completition*, for commands and references, *syntax fontification*, for keywords and comments highlighting with different colors, and *key bindings*, for quick access to the most used functions.

It is worth mentioning, for the great work done by authors, the *VHDL Mode* (version 3.30). This Major mode, besides the intrinsic features of Emacs, provides *template insertions*, that take advantage VHDL's strict syntax to speed up typing of common language structures such as entities definitions and instantiations, program control statements, signal and variables definitions. It also permits *auto-indentation* and *keywords alignment* specified in [5], for a well-structured VHDL models, so they can be efficiently used and maintained also by a third person. The last word goes to *port translation*, for cutting and pasting of entity and component declarations to components instantiations and test bench file generations.

3.3.2. Compilation and Simulation

The workstations used were equipped with the suite QuickHDL release C.2 by Mentor Graphics Corporation. The command `qvhcom`, version 8.5_4.6i, was issued over every source file along with the switch '`-93`' for VHDL-93 compatibility. The tool `qhmake` was used to produce a source listing suitable for `make` and avoid recompilation of unmodified code files.

As simulator `qhsim`, belonging to the same suite and with same version number as the compiler, was the command typed. Because of the intensive application of the *generic* feature of the VHDL language introduced in section 3.2.4, generic values were passed to the simulator through the `Makefile` written for the purpose. The GUI's wave form viewer, break-points and step-by-step facility helped the pro-

grammer to debug the code, permitting a deep insight into the VHDL source line execution.

3.3.3. Synthesis

LeonardoSpectrum Level 3 version 1999.1i by Exemplar Logic was the synthesis tool available. Since its GUI was immediately revealed to be computationally very heavy, switching to `spectrum` line command was one of the first step taken. Indeed, LeonardoSpectrum Level 3 is a versatile optimization and analysis tool with advance Tcl scripting capabilities. This programming language allowed great flexibility and trimmering on a per component basis of the synthesis process, even if the major part of configuration parameters left to be discovered.

3.3.4. Place-and-Route and Back-Annotation

Place and route process is normally a very time consuming task since it involves several software strategies and resolution steps as well as files and commands. For example, three commands were necessary with the equipped suite to accomplish the overall process: `ngdbuild`, to translate and merge the various source files of a design into a single Native Generic Database (NGD) design database, `map`, to map that database into the CLBs and IOBs of the physical device and write out this physical design to a Native Circuit Database (NCD) file, and `par`, to actually place and route a design's logic components (mapped physical logic cells) contained within an NCD file based on the layout and timing requirements specified within the Physical Constraints File (PCF).

To simulate the synthesized system version, two more commands were necessary. `ngd2vhdl` translates an NGD netlist to a VHDL netlist, where the former is produced by `ngdanno`. This command takes a mapped, placed, or routed NCD file and creates an NGD type file. It uses the Native Generic Mapping (NGM) file produced in the process to re-insert all of the optimized logic and netnames into the NGD file so that the original functional test bench can be used for simulation. Moreover, a Standard Delay Format (SDF) file is produced to allow simulations that take into account register-to-register delay time. The whole process takes the name of *back-annotation*.

3.3.5. Other Tools

Many other minor tools, but anyway not less important, were used during designing this project. The `make` command permitted a semi-automatic procedure for compiling, testing and synthesis. Revision Control System (RCS) allowed storing and retrivial of revision making the programmer to feel safe with the last working implementation. `diff` was used to compare the output files with the file obtained

with the behavioural version. Finally `od` and `gawk` afforded the conversion of the test image from and to a plain file of exadecimal data.

3.4. Adopted Methodology

This project represents for the author the first work involving a programming language for digital circuits like VHDL. Nevertheless, an operating method was followed to ensure reliable results and faster developing. Not all the steps described below were necessary or possible to accomplish at every level of the system hierarchy.

Though the VHDL source files have been tested running the commands briefly introduced in section 3.3, the code was written without using any tool dependent instruction and should be compiled and simulated under any other suite.

System Comprehension

The first step assumes a deep knowledge of functions and overall mechanism of the system or sub-system being analyses. Identified blocks and external signals are named.

Component Enucleation

At this level the concept of *entity* and the *divide et impera* approach, both introduced in section 3.2.1, are applied. The system is divided in sub-components that represent functions or macro-cells established at the previous step. As well as the general definition of the sub-blocks, the interdependencies among them should be clear, that is, nomenclature, number and type of signals through which sub-components exchange data.

Behavioural Description

For each sub-component enucleated a behavioural description of its function is produced. This step, which is translated in practice by the *architecture* statement explained in section 3.2.2, focuses its attention on algorithm flow, arithmetic formulas and, more in general, data elaboration. The purpose, indeed, is to verify hardware feasibility and functionality. Technically speaking, the behaviour of a model is described by signal assignment statements within processes: in response to changes of input signals, the corresponding process is activated, the new value read and a new value for output signals calculated.

Test Bench

The behavioural architecture just introduced can be tested using a *test bench file*. A test bench is an entity that provides stimuli for the system under test and reads its output, which can be written to a file. The output produced by a behavioural description can be considered as reference data for subsequent descriptions, since that architecture pays attention to the logical aspect of the system by definition and, thus, can be potentially free of errors.

Synchronous Description

At the previous step every sub-component is of the type asynchronous, that is, processes are executed as soon as the value of an input signal changes. Although an FPGA can be programmed to work asynchronously, digital systems are characterized by a common *clock* signal. In order to obtain better synthesis results in terms of area and speed, a fixed scheme (reported in the listing below) is strongly advised.

Listing 3.1: Standard scheme for synchronous descriptions

```
1  if  rst  = '0'  then                    -- asynchronous reset (active low)
2
3    state  :=  first_s ;
4     ...
5
6  elsif  clk'event and clk  = '1'  then  -- rising clock edge
7    case state is
8
9    when first_s   =>
10     -- first  case statements
11      ...
12     state  :=  second_s;
13
14   when second_s =>
15     -- second case statements
16      ...
17     state  :=  third_s ;
18
19    ...
20
21   when last_s =>
22     -- last case statements
23      ...
24     state  :=  first_s ;
25
26   end case;
```

```
27 end if;
```

This scheme, made essentially of *if* and *case* statements, resembles a finite state machine where the next state depends on input data and present state. Thanks to the clock signal, I/O port values can change during a clock period and their stability at rising clock edges is taken for granted.

Finally, a synchronous description focuses on the realization aspect of the project following the footprints of the behavioural description. Its correctness can be proven comparing its output to the results obtained from the previous step using the tools illustrated in section 3.3.5. In literature this synchronous description is frequently referred as *Register Transfer Language (RTL) description*.

Synchronous Description Optimization

This step, running the previous listing through the synthesis tool briefly described at the end of section 3.3.3, aims to optimize the source program for the synthesis process. Indeed, although the trick explained above, modules might not be synthesized.

VHDL does not know anything about the FPGA being targeted, thus it does not make any assumption of features of the chip being programmed and allows lots of operations on its basic data types. Nevertheless, the hardware is not likely to implement any operation such as multiplication, division or modulo. If the code for synchronous description uses such or even more complex functions, it cannot be synthesized and other strategies are required.

Synthesis and Timing Version

If the previous stage is successful, the synchronous architecture of the entire system can be processed through the commands of the place-and-route suite mentioned in section 3.3.4. The result is one long file, containing all the primitive blocks belonging to the technology adopted, described in VHDL and interconnected to perform the same function of the original system. Along with this file, another listing in SDF format is produced. This complex of data automatically generated can simulate a system behaviour closer to reality, allowing area and timing measurements and critical path identification.

Multiple Configurations

As explained in section 3.2.3, component architectures of a system can be intertwined to investigate system performances. At this stage all the sub-components have been tested and new configurations for a variety of solutions can be written,

tested and synthesized. Because the test bench interface does not change, the same test source file can be used.

4 Architectural Implementation Analysis

This chapter presents the actual design depicted in chapter 2 using the methodology and tools introduced in chapter 3. The description starts with the first attempt of design, where the algorithm feasibility was the primary purpose. In four subsections, the main components permutation, substitution, Pseudo-Random Number Generator (PRNG) and memory is analyzed showing briefly their function and presenting the major problems faced. The second section introduces limitations of the first implementation and presents a new solution, called *one-shot*. New subcomponents designed to deploy this approach are outlined in the next section. The chapter will be concluded with a description of the major bottleneck of this algorithm, along with the delineation of some possible solutions and a brief comparison to other related works.

4.1. Algorithm Feasibility

Following the method illustrated in section 3.4, the algorithm, whose the block diagram is shown in figure 2.5 on page 13, was spontaneously divided in three chief components: permutation, substitution and PRNG, for each a behavioural description and a test bench was written.

To prove the correctness of the permutation block, the same image reported in [11] fed the circuit input and the resulting output was compared with the encrypted image in the same technical report. The obtained data was used as sample for the following synchronous and timing architectures. Comparison was attained using program `diff` outlined in subsection 3.3.5.

For substitution components, instead, no data file with which to compare the behavioural architecture output was available. Nevertheless, a small change in the substitution code allowed to perform an anti-substitution and the algorithm's correctness was shown by the matching of these data to the original ones. Subsequently, the same comparison method adopted for the permutation block was used.

For the PRNG the situation was slight different. A simple research on the Internet revealed the existence of the Scientific Library written in 'C' by GNU Free Software Foundation. In this library a source file, implementing the same pseudo-random number generator used in this project, reports the theoretical value of the PRNG after 10,000 iterations. Once that number was generated, PRNG's code has been considered free of logical bugs.

Proven that the three components performed the operations for which they had been designed, the system for one-round encryption was assembled and the behavioural architecture output used, once again, as data sample for the subsequent descriptions. It is worth highlighting that at this stage the encryption procedure works on a step-by-step basis, that is, a substitution is not started until a permutation has been finished. Therefore at most three memory areas for image data storage are necessary: an input area for images being encrypted, a temporary area for permutated images and an output area for encrypted data. These memory banks can be reduced to two if the output image overwrites the input area. Extra memory is necessary for the substitution process and the PRNG component in order to work.

At least but not last, all the I/O memory operations are performed using a common interface summarized at the end of the present section.

4.1.1. Permutation

The permutation procedure illustrated in section 2.3 consists, in the final analysis, of just a change in coordinates of the input values. If either the input and the output data were stored using flip-flops, a zero-delay time permutation would be possible, since the synthesis would result to as an intricate as banal route network. This method reassembles the ECB of section 2.1.2. Unfortunately, considering the amount of data the algorithm has to deal with and the target chip, this approach is not achievable and a read/write solution must be provided.

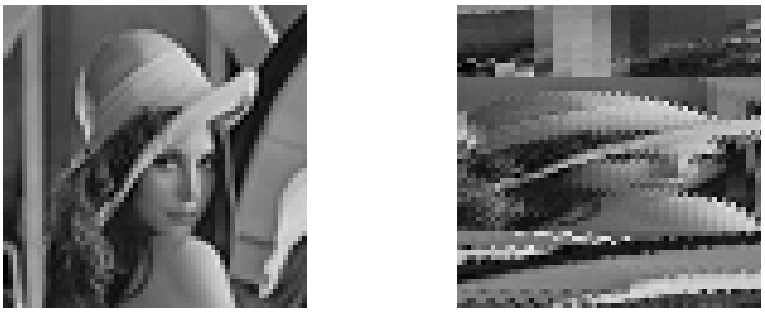
The procedure's core, whose listing is reported below, is constituted of three loop cycles that scan the whole image. The outer loop is used for the current partition part(), chosen according with key, which is fetched at the start signal, and upper and lower bounded by cNs and nNs−1, respectively. The second loop is for the x coordinate of input image, while the last and most internal loop is for the y coordinate, ranged between 0 and n−1.

Listing 4.1: First behavioural description of the permutation procedure

```
1  -- variables initialization
2  read_mem(key_loc, key);
3  part := choose_part(key);
4  nNs := 0;
5
6  partition : for Ns_index in part'low to part'high loop
```

```
7     next partition when part(Ns_index) = 0;
8     cNs := nNs;
9     nNs := nNs + n/part(Ns_index);
10    qs  := part(Ns_index);
11
12    x_axes : for x in cNs to nNs−1 loop
13
14      y_axes : for y in 0 to n−1 loop
15
16        read_mem(img_in + y∗n + x, datum);
17        permutation(qs, cNs, x, y, x1, y1);
18        write_mem(img_out + y1∗n + x1, datum);
19
20      end loop; −− y_axes
21    end loop; −− x_axes
22  end loop; −− partition
```

The actual calculation for a data permutation is reported on the following page. Because qs in practice is always a power of 2, operations *division* and *modulo* can be undertaken to the synthesis tool providing only a *case* statement that restricts qs declared as natural to its actual domain, that is $\{x \mid x = 2^n, n \in \mathbb{N}\}$.



(a) Original (b) Permutated

Figure 4.1: Sample square image before and after permutation.

Listing 4.2: Core of the permutation procedure

```
1  -- purpose: performs the actual permutation algorithm
2  procedure permutation (
3    constant qs, cNs : in    natural;
4    constant x, y    : in    natural;      -- old coordinates
5    variable x1, y1  : inout natural) is   -- new coordinates
6  begin  -- permutation
7    x1 := qs * (x - cNs) + (y mod qs);
8    y1 := div(y, qs) + cNs;
9  end procedure permutation;
```

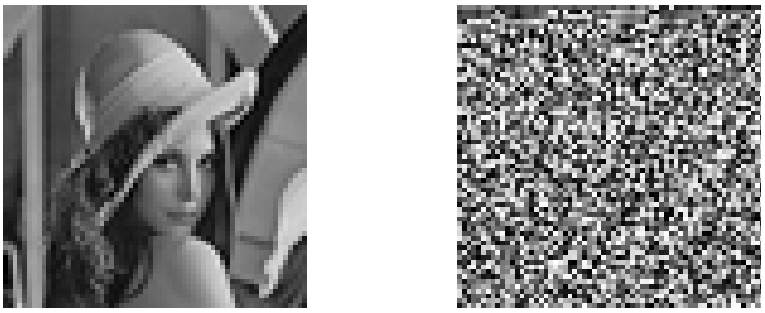
An example of output from this stage of encryption can be seen in figure 4.1, where a $\delta = \{4, 2, 4\}$ is evident from the permutated image.

4.1.2. Substitution

Likewise permutation makes a change only in input data coordinates, substitution procedure changes only input data values. The following listing represents the most significant part from the behavioural architecture.

Listing 4.3: Core of the substitution procedure

```
1  cp := 0;
2  cc := 0;
3
4  -- actual substitution
5  for i in 0 to img_prsn_size - 1 loop
6
7    read_data(i, p(index(i, 0)));
8
```



(a) Original (b) Substitued

Figure 4.2: Sample square image before and after substitution.

```
9   prsp := p(index(i , s_lag)) − p(index(i , r_lag)) − cp;   −− mod 2**be;
10  prsc := c(index(i , s_lag)) − c(index(i , r_lag)) − cc;   −− mod 2**be;
11
12  c(index(i , 0)) := p(index(i , 0)) + prsp + prsc;   −− mod 2**be;
13
14  cp  := Icond( p(index(i , s_lag)), p(index(i , r_lag)), cp );
15  cc  := Icond( c(index(i , s_lag)), c(index(i , r_lag)), cc );
16
17  write_data(i , c(index(i , 0)));
18
19 end loop; −− i
```

Since the algorithm comprises two Pseudo-Random Sequences (PRSs), p() and c() for plain and cipher data respectively, two $37 \cdot 2^{32}\,\mathrm{bit}$ circular arrays need to be stored. Moreover, the component requires to dialogue with the PRNG, because the two abovementioned structures are initialized with pseudo-random numbers. Therefore, the computationally less expensive code (only 32-bit wide sums are required) is balanced by a more complex management.

It is worth observing that the number 32, as width of elements for circular arrays (see following subsection), is here reported only because that is the number of bits which assures, according to [8], better performance. Nonetheless, the component was designed using a generic value be, as well as for the two Fibonacci's lags r and s which take the name r_lag and s_lag respectively.

An example of output from this component can be seen in figure 4.2. From the first lines of the right image it is evident that the two vectors were initialized with poor random numbers.

4.1.3. PRNG

The Pseudo-Random Number Generator component is not critical in this project as long as it provides a long enough period and sufficient statistical properties.

According to [10] [11], the PRNG called "R250" was chosen. This PRNG consists in a 250-element vector of pseudo-random numbers maintained as a circular array. Every new number is computed through a very fast XOR operation, supplying a reliable and small circuit. Finally the vector has to be initialized with a $250 \cdot 2^{32}\,\mathrm{bit}$ passphrase which, for our purpose, is barely formed by natural numbers between 0 and 249.

As well as the substitution component outlined in the previous subsection, the PRNG was designed with a generic width for the elements in the vector. Nevertheless, only the 32-bit version has been tested.

4.1.4. Other Components

It is worth spending some words for component *memory*, as it is used over the all algorithm, not only for image storage, but also for temporary data processing, pseudo-random number sequences and data vectors in general. A brief description of component *multiplexer* will conclude this subsection.

Memory

Since the first experiment with the permutation's behavioural architecture, the role played by this component immediately appeared to be crucial. To try avoiding more problems in subsequent developments, a general approach that goes under the name of *two-way handshake protocol* [13, subsection 17.2.2] was implemented. The protocol name comes from the fact that two signals are required to get two devices to communicate: cs (short for "chip select") tells the memory a read/write operation is pending, while data_ready serves as back signal for status *data available* and *free bus*.

Listing 4.4: Memory entity definition for 2-way handshake protocol

```
1  entity memory is
2
3    generic(addr_bits : positive;
4            data_bits : positive;
5            delay     : time);
6
7    port(addr_in  : in  std_logic_vector(addr_bits-1 downto 0);
8         data_in  : in  std_logic_vector(data_bits-1 downto 0);
9         data_out : out std_logic_vector(data_bits-1 downto 0);
10        chip_sel : in  std_logic;
```

```
11          mem_write : in  std_logic;
12          data_ready : out std_logic;
13          clk       : in  std_logic;
14          do_init   : in  std_logic);
15
16    subtype word is std_logic_vector(data_bits-1 downto 0);
17
18    constant nwords : integer := 2 ** addr_bits;
19
20    type ram_type is array(0 to nwords-1) of word;
21
22  end;
```

This solution has at least a couple of advantages. First, it can deal with implementations where the response time is not fixed. This is the case of the behavioural architecture in this project. Second, it offers an easy way to move toward a simplified *one-way handshake* where only one signal (namely cs) accomplishes the communication protocol. For instance, the Block SelectRAM+ technology, depicted in section 3.1.2 and utilized in the final part of this project, uses such handshaking scheme.

To hide the two implementations' details from programmer, a VHDL package, defining functions like read_mem and write_mem, was written. The increased flexibility can be observed in test bench files, since the same test architecture is indistinctly used for both behavioural and synchronous descriptions.

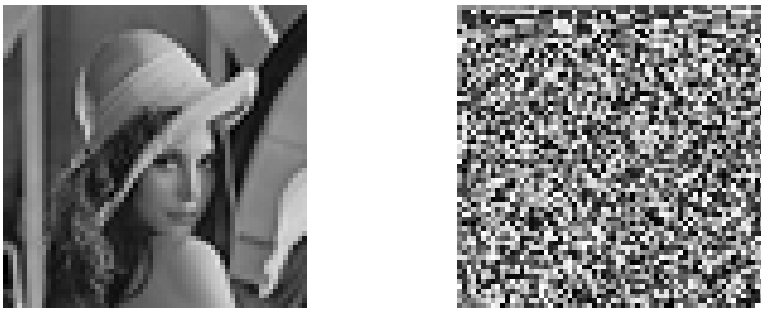
Multiplexer

The memory entity abovementioned is a common resource shared by almost every component in the system. To avoid bus conflicts between concurrent accessing devices a behavioural asynchronous architecture of a multiplexer was designed. The entity, taking advantage of the *generic* facility described in section 3.2.4, provides up to four inputs for as many as components, which are automatically selected by a '1'-value of the multiplexed cs signal. An opportune statement asserts during simulations that two devices will never access the bus simultaneously.

As final remark, the same entity can also be utilized in conjunction with the tristate buffer (BUFT) technology supported by some FPGAs.

4.1.5. Entire System

The whole system is built up instantiating components for permutation, substitution and PRNG entities outlined in the previous subsections.



(a) Original (b) Encrypted

Figure 4.3: Sample square image before and after one-round encryption.

This architecture, instead, takes care of the coordination aspect among the different parts the system is composed of. After receiving a start signal, a controller provides a key for permutation component, which is started immediately after. Initialization of the two substitution's vectors with pseudo-random numbers provided by the PRNG is the next step, which is followed by a request of substitution on the permutated data. A done signal raised to '1' will notify that one-round encryption is completed. Data to and from the memory located in the test bench architecture are multiplexed using the component described in subsection 4.1.4.

The final result can be evaluated from the two images reported in figure 4.3.

4.2. Some Improvements

While so far the hardware feasibility of the algorithm considered in this work has been proved, the present section overlooks a limit of the first implementation just discussed and offers a new approach called *one-shot* that will be described in detail in section 4.3.

4.2.1. Limits and New Ideas

The algorithm presented in chapter 2 belongs to the class of algorithms called *block cipher*, since it works on blocks of data (see section 2.1.2). The reason why the algorithm cannot be seen as a *stream cipher* (which operates on the plaintext a bit at a time) is due to the permutation performed by chaotic Kolmogorov flows on the input data. If the entire system is looked at as a r-round product cipher, the data at the output of the first round is already written in an apparently random order and the following round has to wait for the first one to be finished.

Even if a subset of the image is chosen, the situation does not change. Considering the partition $\delta = \{n_1, \ldots, n_k\}$ of the Kolmogorov flow $T_{n,\delta}$, where $n = \sum_{i=1}^{k} n_i$, it is known from section 2.2.2 that the function $T_{n,\delta}$ scrambles data from its domain $[N_s, N_s + n_s) \times [0, n)$ to the codomain $[0, n) \times [N_s, N_s + n_s)$, where N_s is the left border of the vertical strip which contains the points that are going to be transformed. As can be seen, even working on a narrower range of data like an output strip, the subsequent round still has to wait for the previous one, since the input and output strips are horizontal and vertical, respectively.

In light of these considerations, a *full loop unrolling* [3] architecture, where the r rounds are chained together to form an asynchronous cryptosystem, is meaningless. Hence, it seems no optimization can be attained working at round level, that is to look at the system as a set of r blocks, one for each round.

4.2.2. Using the Inverse Kolmogorov Flow

To overcome the problem of the block ciphering and achieve some improvements, a one-round black box is opened.

In the first solution presented in subsection 4.1.5, the permutated data is written onto the memory to be read again by the following substitution process. If it is possible to feed the substitution component with data just elaborated by the permutation block, not only area would be saved, but also some clock cycles. In fact, by connecting the two major blocks together, at least one write and one read operation is eliminated, leading to potentially less area occupancy due to multiplexer and other logic for memory bus arbitration and virtually zero transfer time between the two components.

However, the idea just explained cannot be immediately put into practice. Indeed, the Kolmogorov flow $T_{n,\delta}$, as illustrated in [10] and summarized in section 2.2.2, shuffles data transforming an ordered sequence of coordinates $(x, y) \in [0, n) \times [0, n)$, into a pseudo-random sequence $(x', y') \in [0, n) \times [0, n)$, while substitution re-reads the same sequence of coordinates (x', y') in a sequence fashion again.

The difficulty can be dodged exploiting the inverse Kolmogorov flow $T_{n,\delta}^{-1}$. Ranging coordinates (x', y') between 0 and $n - 1$, an anti-permutation component will produce an ordered stream of data picking up the values in a pseudo-random basis from the plaintext image. Now, permutated data can be directly used as input for the substitution block without violating the algorithm functionality. It is worth noticing that the problem due to coordinate scrambling is not vanished. Simply the input values, in the place of the output data, are now read in an apparently random fashion.

4.2.3. Packing atoms

Throughout the last subsection, the general term *data* was used to refer to a unspecified amount of bits manipulated by either permutation and substitution components. In the matter of fact, this situation is slightly different.

Reconsidering what is stated in section 2.2.2, a larger value for the side n of the square image gives a more secure algorithm. With $n = 64$, for instance, about 2^{50} possible keys are available. This value is close to the one obtained with DES [10]. Because the amount of memory for data storage is limited, increasing n means reducing the number of bits undertaken to any single permutation operation. The lower bound is obviously represented by the minimum physical quantity of information, i.e. one bit. To avoid ambiguity, let define the term *atom* be the unity of data processed by the permutation. If variable atom is the representative in VHDL of atoms, then

$$0 \leq \mathsf{atom} < 2^{\mathsf{atom\_bits}}$$

where atom and atom_bits are natural numbers.

On the other hand the period of cycle for the Add With Carry and Subtract With Borrow PRNGs used in the substitution component depends proportionally on the base b within which pseudo-random numbers are generated. Thus, the greater the base, the longer the period. For this algorithm the author chose $b = 2^{32}$ according to the table at the end of [8], which implies 32-bit width for data. To avoid ambiguity again, let us define the term *prsn* (pseudo-random sequence number) as the unity of data processed by the substitution. If prsn is the corresponding variable, then again

$$0 \leq \mathsf{prsn} < 2^{\mathsf{prsn\_bits}}$$

where prsn and prsn_bits are natural numbers, but where only $\mathsf{prsn\_bits} = 32$ is meaningful.

For what is stated above, it should be clear that permutation and substitution can be tied together only if atoms just permutated are padded to the width of prsn. However, this practical procedure, besides changing the original algorithm version, wastes a considerably amount of time and resources. The solution presented in this work, instead, consists of using an *atom packer* to fill up a prsn_bits-bit wide vector with atoms.

4.3. One-Shot Approach

To translate the ideas described in the previous section into VHDL code, a rewriting of the entities for permutation and substitution processes is inevitable. In the first implementation, both components operate independently from each other. In this second attempt, more cooperation is required. This is due to the fact that one-round

encryption is accomplished with only one pass for each data in the place of two for the first implementation. From this it derives the name *one-shot*.

4.3.1. Permutation One-Shot

The new entity of the function permutation, whose a block diagram is shown in figure 4.4 on the following page, is provided with four input signals: qs and cMs, which refer to the numbers q_s and M_s of the inverse Kolmogorov flow $T_{n,\delta}^{-1}$ defined in section 2.2.2, and the signals x1 and y1, which refer to the coordinates of atoms at the output. The range of the last two signals is from 0 to $\mathsf{n}-1$, where n represents the number of atoms on one side of the square image.

Given these input values, the process actual_permutation can work out the coordinates x and y for the atoms of the plainimage. The subsequent process read_atom can easily read the new atom from the memory with the relative address of the first image location

$$\mathsf{atom\_indx} = \mathsf{y} \cdot \mathsf{n} + \mathsf{x}$$

and pass the value to the pack_data which will in turn notify that a new prsn is available when a sufficient number of atoms has been packed.

In the former description, an important detail has been deliberately omitted. What will happen if the memory from which read_atom reads atoms does not contain one atom per location? The image can still be seen as a vector of atoms, but atom_indx can no longer be used as a memory address. Nevertheless, the component has been designed to also handle this situation.

Variable atom_indx, as shown in figure 4.5 on page 40, contains all the necessary information to extract the correct address addr and the position atom_pos of the atom included in the datum fetched from the memory. Constants side_bits and data_bits are component's generics that represent the width of coordinates and permutation data output, respectively.

4.3.2. Boundary

This component provides one-shot permutation for values qs and cMs and passes over signals x and y (see figure 4.6 on page 40).

The purpose of this entity is not strictly necessary, since the knowledge of x, y and δ is sufficient to evaluate variables q_s and M_s of the algorithm. However, taking advantage of the fact that x and y increase sequentially from 0 to $n-1$, the logic necessary to calculate qs and cMs is shrunk to a simple check of the form

Listing 4.5: Simple check statement used in boundary component

```
1 if x = 0 and y = nMs then
2
```

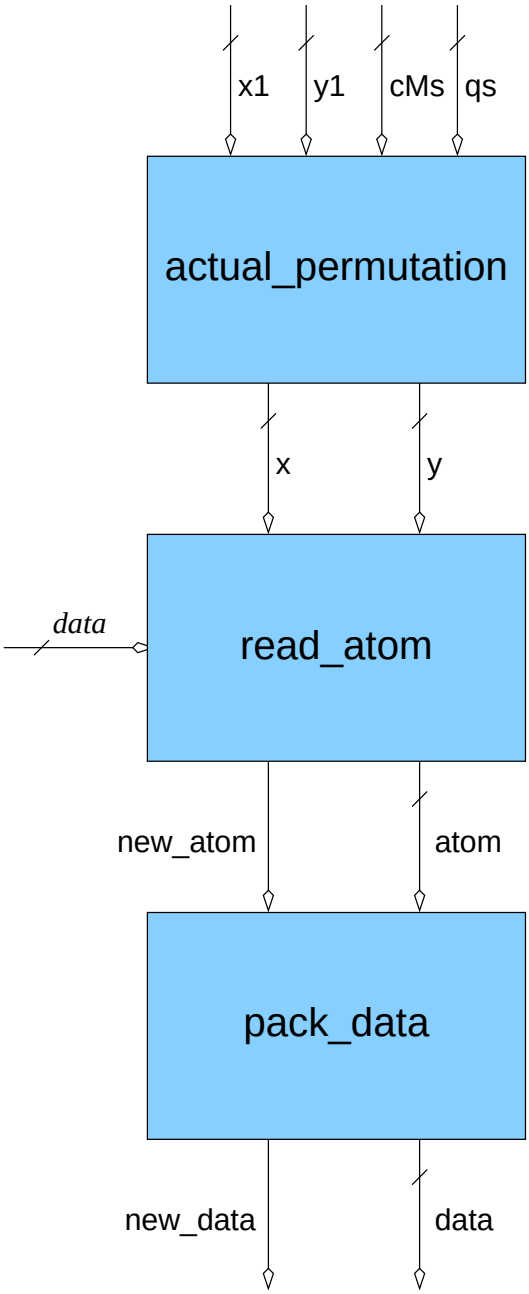


Figure 4.4: Block diagram of the component permutation architecture one-shot.

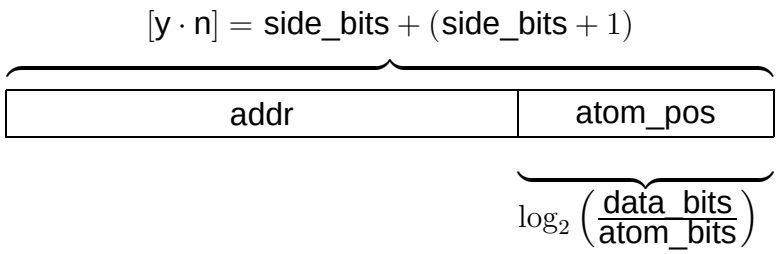


Figure 4.5: Splitting of atom_indx for one-shot permutation general case.

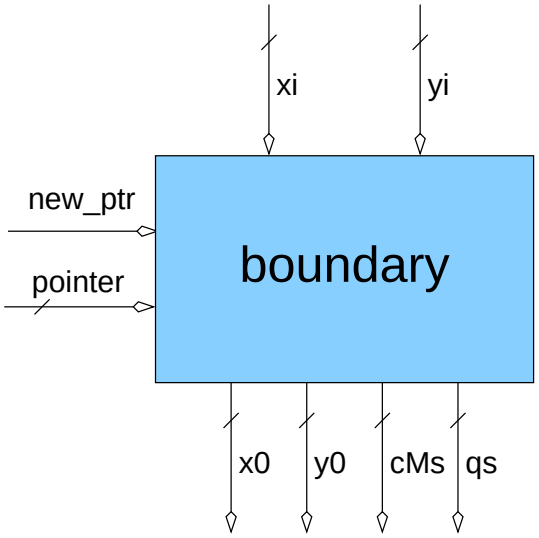


Figure 4.6: Boundary component.

```
3     qs  <= ...
4     cMs <= ...
5     nMs := ...
6
7   end if;
```

where nMs is the next M_s, that is the upper bound of the current δ-partition.

4.3.3. Substitution One-Shot

A careful study of the substitution procedure revealed that the component could be decomposed in three sub-entities shown in figure 4.7 on the next page: two PRSs and one adder. Briefly introduced here, this component will be resumed also in subsection 4.4.1 where the meaning of the dashed and dotted lines is explained.

A Pseudo-Random Sequence is simply the class of PRNG studied in [8] and outlined in section 4.1.2 (see also variable prsp and prsc in listing 4.3 on page 31). Its purpose consists in providing new pseudo-random numbers by summing two opportune elements, taken from an r-long vector of prsns, and a value supplied by the above entity. At cost of a 32-bit input signal prsn0 and two control flags wr_prsn0 and wr_done, component prs has also been provided with writing capabilities instead of using a different entity to update the vector with the new generated value. In doing so, a 2-input multiplexer for memory arbitration has been saved. Finally, two instances of prs are necessary in order to maintain as many as pseudo-random number vectors, one for plain data (PRSP) and one for cipher data (PRSC).

The adder's function simply consists of summing up two pseudo-random numbers, generated by PRSP and PRSC, with the new value coming from the permutation one-shot component. The result represents the 32-bit ciphered value at the output of the substitution one-shot.

4.3.4. PRNG

The Pseudo-Random Number Generator has not been altered from the first implementation since its utility is only for the initialization phase.

4.3.5. System One-Shot

The component system one-shot consists of a controller with a double function: it generates the output coordinates x1 and y1 in a sequential manner and saves the ciphered value at the output of the substitution one-shot back to the memory. It should be clear at this point that more than one coordinate increment is necessary in order to generate one cipher output. To accomplish this task, the current component needs to have some kind of feedback from permutation and substitution components.

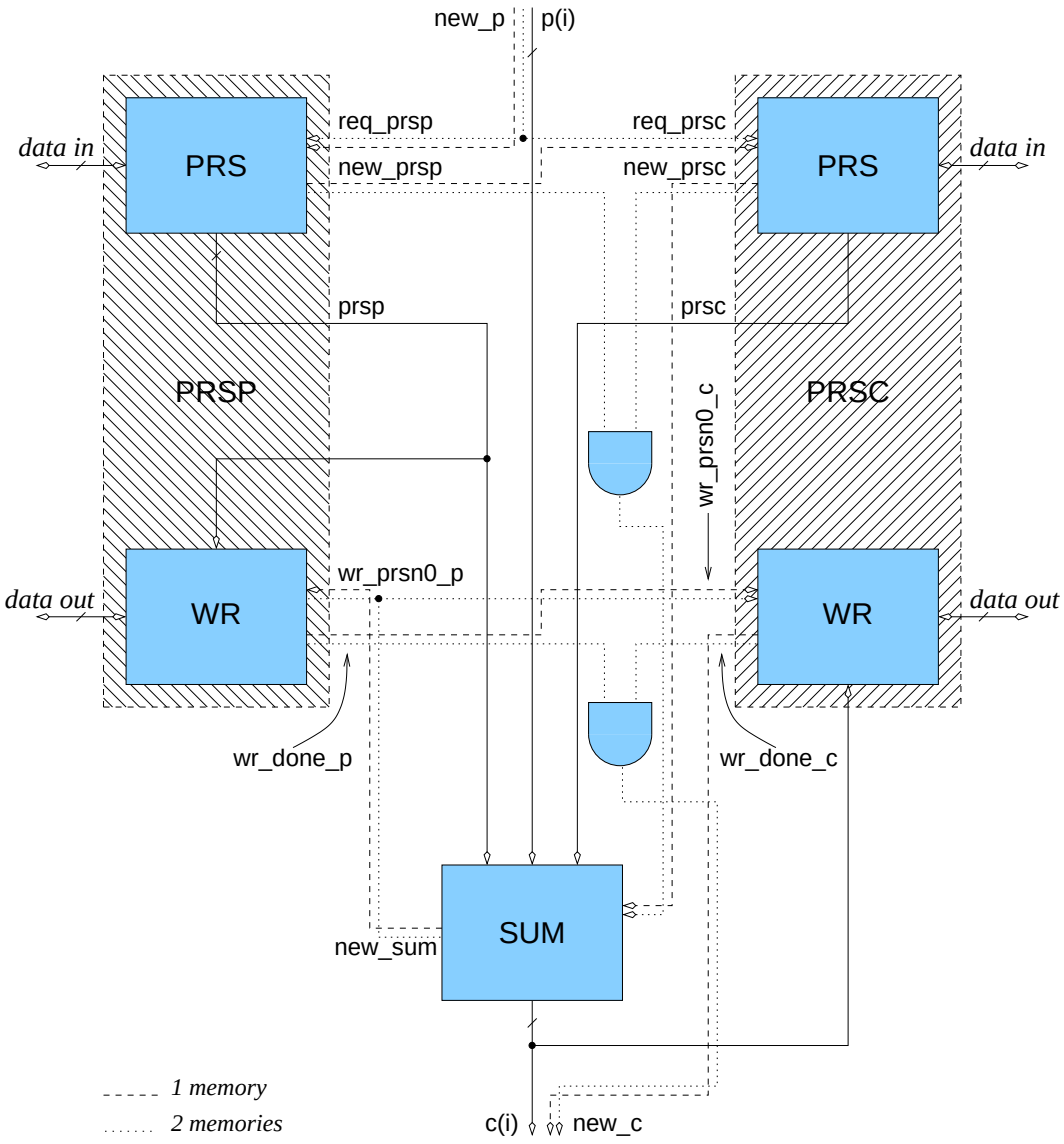


Figure 4.7: Block diagram of the component substitution architecture one-shot.

	P1S	S1P	S1C	RNG	CTR
MII	R				
MIO					W
MSP		R/W			W
MSC			R/W		W
MRV				R/W	

Table 4.1: Summary of read and write access to memory per system's component.

4.3.6. Entire System

Connecting together all components so far described, the whole system can be represented with the block diagram drawn in figure 4.8 on the next page. UBD, P1S, S1S stand for boundary, permutation one-shot and substitution one-shot, respectively; the two blocks CTR constitute, in realty, just one entity, which corresponds to the system one-shot and the signals new_pdata and new_cdata represent the abovementioned feedback. In the picture are also shown several memory areas which the system needs to operate. MII, MIO, MSP, MSC and MRV are short for Memory Image Input, Memory Image Output, Memory Substitution Plain, Memory Substitution Cipher and Memory pseudo-Random Vector, the last belonging to PRNG.

4.4. Using More Than One Memory

Version one-shot of the cryptosystem was presented in the prior section, at the end of which it was also said that blocks MII, MIO, MSP, MSC and MRV serve as memory areas to perform one round of the encryption process. As can be seen form figure 4.8 on the following page, these memory areas are connected to different parts of the system, each of which can access its own bank of memory only when the other components are not using the shared memory bus. The situation is summarized in table 4.1.

What will happen if each MII, MIO, MSP, MSC and MRV, in place of representing an area of the same memory, constitutes a memory instance by itself, separated from the others? A double advantage will be achieved. First of all, each component will access its own memory bank regardless other component I/O requests; second the whole system will benefit from a reduction in area because multiplexers for memory bus arbitration are no longer needed. In realty the reduction is not so drastic, mainly because multiplexers are just a routing problem and then because some memory banks necessarily need to be accessed by two components. In fact, reading table 4.1 by rows, it is immediately clear that MSP and MSC need to be initialized by PRNG.

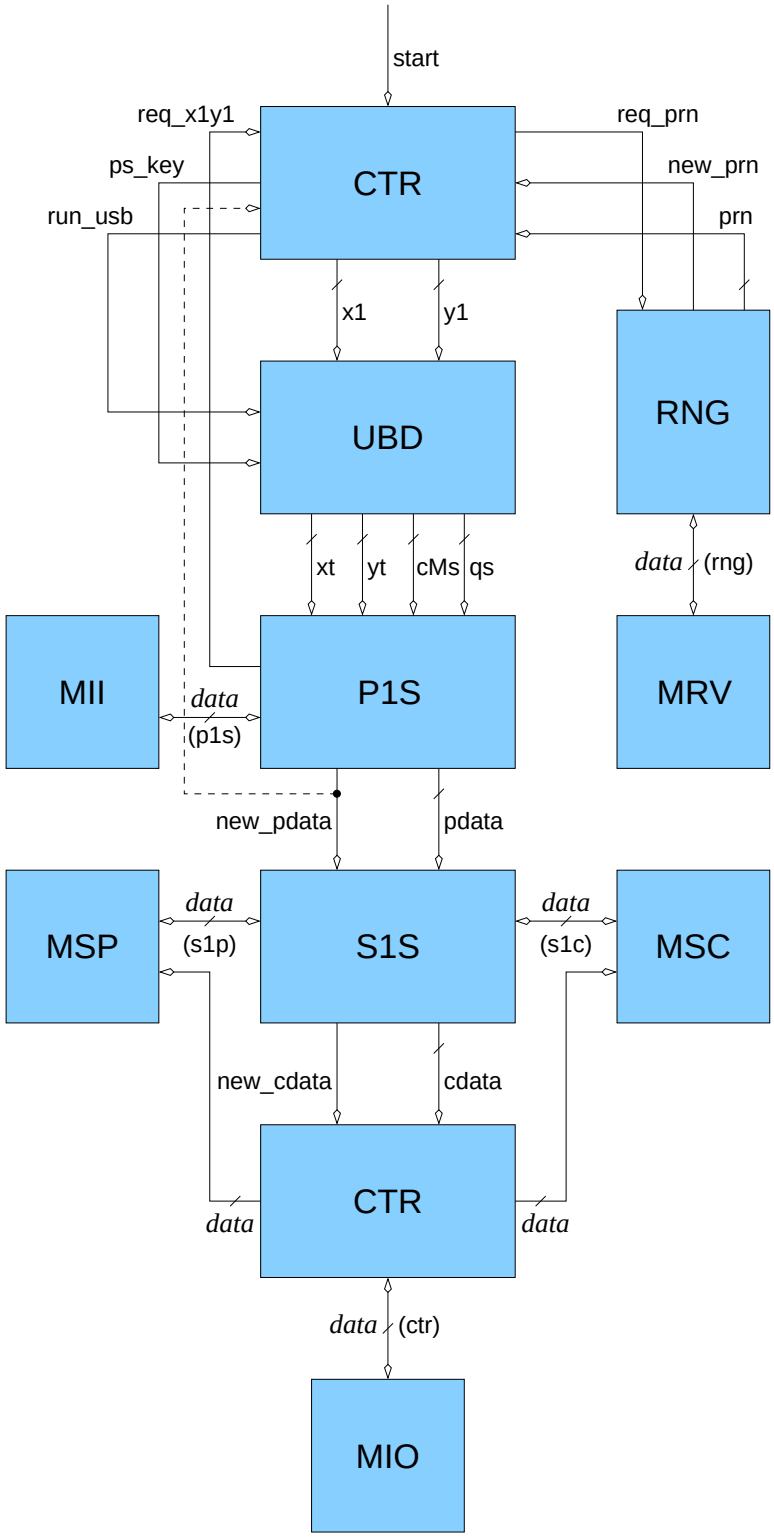


Figure 4.8: Block diagram of the whole one-shot architecture of the system.

4.4.1. Substitution one-shot multi-memory

In this work not all memories have been split up due to lack of time. Nevertheless, the above-mentioned idea was successfully tested on version one-shot of the substitution component. Recalling the block diagram from figure 4.7, substitution one-shot is composed by two memories, one for the plain vector and the other for the cipher vector. If the two memories are looked at as a single bank separated from the main memory, substitution one-shot does not share any I/O access with other components during its working (writing operations by PRNG are required only during initialization phase), whereas the two PRSs have to wait for each other. This is what has been done with the configuration of the system that carries the name osh2mem, and was, moreover, the first attempt in using the Block SelectRAM+ technology.

If also the two PRSs can operate independently, MSP and MSC constitute two component instances and can be accessed in parallel. For this case, the configuration name is osh3mem.

Finally, an ulterior configuration called opr3syn was tested. Simply changing the order with which components of substitution one-shot talk to each other, the required values for the prsp and prsc calculations can be read from the two vectors before the corresponding permutated input data is actually available. In other words, new values for prsp and prsc are worked out as soon as substituted data at the adder's output is ready.

What said in the prior subsection, the access to the memory introduces a final important observation which claims, before going into detail, a brief introduction.

Let's suppose that read and write operations are equivalent in terms of time required for execution and that that time is worth one unity. Let's also suppose the square image being encrypted has a side containing n atoms and m prsn-long data, both defined in subsection 4.2.3.

Considering the first implementation discussed in section 4.1, two distinct passes were necessary to deploy an encryption. The former is the permutation that works manipulating atoms: $n \times n$ unities of time are required to read all of the image and $n \times n$ more unities to save the permutated one. The latter is the substitution which works processing prsn data: $m \times m + m \times m$ unities of time are necessary to read and write all image's data, for each of which $2 + 2$ read and $1 + 1$ write operations are required for the two PRS's vectors maintenance. Summing up everything:

$$\begin{aligned} [n \times n + n \times n] + [(m \times m + m \times m) + (2+2)m + (1+1)m] &= \\ 2(n \times n) + 2(m+3)m &= \\ 2(n \times n + m \times m) & \end{aligned}$$

since surely $m \gg 3$.

With the second implementation presented in section 4.2, some improvements have been reached. Because permutation does not save atoms to memory, but passes

them directly to substitution, the same reasoning led above gives the following result:

$$[n \times n] + [m \times m + (2+2)m + (1+1)m] \quad = \\ n \times n + m \times m$$

showing the time has been halved.

Performance can further be improved. With opr3syn configuration a real $n \times n + m \times m$ unities of time can be achieved, since the update of the two PRSs' vectors occurs during the permutation fetching phase. If two memories were used for input and output images, the time would decrease to $n \times n$, since permutation and substitution are simultaneous and $n \ll m$.

In order to reduce this limit, more work on permutation optimization is required. This does not seem to be a trivial task due to the high confusion introduced by the chaotic Kolmogorov flows in coordination calculation.

5 Performance Evaluation

This chapter presents the results obtained from the synthesis of the two major implementations described in chapter 4. The first section will introduce the meaning of the parameters used to measure those implementations. The subsequent section describes and comments the collected measurements. The chapter will conclude with a brief comparison to implementations of other algorithms.

5.1. Data Interpretation

Most of results presented in this chapter are obtained from running commands and tools of subsection 3.3.3 and 3.3.4, where synthesis and place-and-route procedures were outlined. From the reports of those programs, in fact, is possible to extract the following informations:

Slices, flip-flops and LUTs These three parameters are related to the area and resources occupied by the synthesized system. Especially meaningful is the number of slices which represent the number of Configurable Logic Block (CLB) utilized (see section 3.1.2). To note that no established metric exists to measure the hardware resource costs. Area measurement in term of CLBs, indeed, does not yield a true measure of actual FPGA utilization, since hardware resources within CLB slices may not be fully utilized by the place-and-route software so as to relieve routing congestion. For a summary of the available resources see table 3.1 on page 17.

$\Delta_{\mathbf{max}}$ The maximum delay is calculated taking into account every possible path in the circuit between two points. For each paths, the tool computes the time necessary for a signal to transit along the path and get stable.

$f_{\mathbf{max}}$ The maximum frequency is the reciprocal of the maximum period the software calculates for system under test. It can be used to set the oscillation frequency

of the clock generator on board of the FPGA and is taken into account by the post place-and-route simulation takes into account the parameter $f_{\max}$ just mentioned.

Knowing n_d, the number of bits elaborated by the component, and t, the time taken to elaborate those bits, the parameter *throughput* TP, defined as

$$TP = \frac{n_d}{t} \tag{5.1}$$

can provide a coefficient to describe performances of the system under test.

Unfortunately throughput TP depends on the frequency the system runs. If, for some reason, the code is compiled and ported on another FPGA and the frequency changed, the value calculated for TP is no longer valid. To overcome this inconvenient, the coefficient TP_{cc} has been defined in this work, where the subscript stand for *clock cycles*

$$TP_{cc} = \frac{n_d}{n_{cc}} \tag{5.2}$$

The parameter n_{cc} can be obtained from the following simple formula:

$$n_{cc} = t \cdot f \tag{5.3}$$

where the quantity t is that used in eq. (5.1) and f is the clock frequency at which the FPGA runs.

It is also worth spending some words, before concluding this section, about technical details of the image used as input sample by the several versions of the encryption algorithm. The image comes in the Portable Graymap (PGM+) format, which divides the image in two parts: an header of 13 bytes — which contains image specifications, as sub-format, image size and levels of gray — and the body of the image. The sample picture consists in a 64×64 256-level image. This guarantees 4,096 bytes of data, 2^{15} bits and 1,024 32-bit long words.

5.2. Data Analysis

In the present section the results obtained from the simulations will be presented by means of tables and commented in two separated subsections.

5.2.1. First Implementation

The purpose of the first implementation widely discussed in section 4.1 consists in proving the feasibility of the algorithm presented in chapter 2.

In order to have a clearer view of the component and subcomponent interdependencies, figure 5.1 on the next page shows the tree diagram of the logical structure

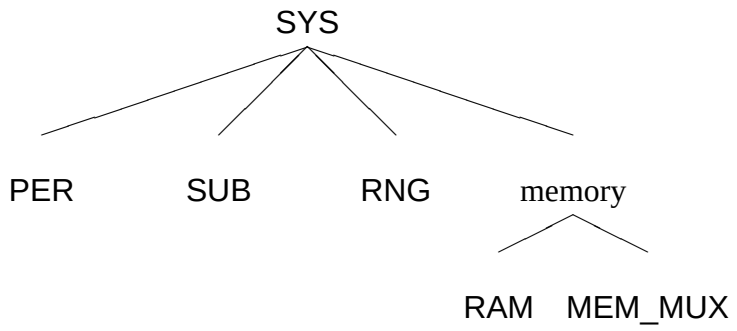


Figure 5.1: Tree diagram of the first implementation.

of the whole system. Using different font, the picture discerns real components, which have an actual entity, from logical components. SYS, for instance, stands for *system* and comprehends the entire system outlined in subsection 4.1.5. Similarly, PER, SUB and RNG are shorts for *permutation*, *substitution* and *PRNG* components, depicted in sections 4.1.1, 4.1.2 and 4.1.3, respectively. RAM, representing the memory from and to which the data are written, is considered external to the system, whereas the multiplexer MEM_MUX is taken into account in SYS's evaluation. These last two forms the logical component "memory" of the figure.

With the aid of the formulæ from the prior section, table 5.1 on the following page can be built. Values from column "Slices" to column "$f_{\max}$" have been obtained from the log files of the place-and-rout process which follows a synthesis that was set for a speed optimization. As it can be seen from the notes at the bottom of the table, the number of slices does not reflect the real area utilization since some LUTs are used as routes, through which signals pass without being modified.

The notes at the bottom of the table also show that atom and memory cells were both 8-bit wide. This means that one atom was processed per reading cycle and that 4096 cycles were necessary to complete the task. For the case in which the atom was 2-bit long, the number of atoms to process would be quadrupled and so the time t, while the number of bits n_d would be constant. This decreases the component's performance by a factor of 4.

On the other hand, SUB's data are 32-bit long and four reading cycles are necessary to fill up a prsn which justifies the very low rate in the last column. If cells' width was 32 bits, only one cycle would be required and the performance of this component would do increase by a factor of four.

That was the main reason that yield to a new version depicted in section 4.4 and analyzed in the remaining part of the chapter.

	Slices	FFs	LUTs	Δ_{max} (ns)	f_{max} (MHz)	f (MHz)	t (µs)	n_d[†] (bit)	TP (Mbit/s)	n_{cc}	TP_{cc} (Mbit/clk)
PER	114	93	137[▷]	16.7	59.8	60.0	411	$8 \cdot 4096$[⋆]	79.7	24659	1.329
SUB	240	231	272	13.4	74.7	74.1	1,078	$8 \cdot 4096$[‡]	30.4	79873	0.410
RNG	130	152	178[◁]	11.8	101.7	not relevant					
SYS	586	560	700[◇]	16.4	63.3	63.2	1,737	$8 \cdot 4096$[*]	18.9	109838	0.298

[†] memory cells are 8-bit wide
[⋆] atom is 8-bit wide
[*] prsn is 32-bit wide
[◁] 2 used as a route-thru
[▷] 4 used as a route-thru
[◇] 12 used as a route-thru

Table 5.1: Summary of the synthesis and place-and-route processes for the first implementation.

5.2.2. One-Shot Implementation

In this section the procedure followed for the first implementation will be repeated for the version one-shot of the system.

The tree diagram of the present implementation is shown below. As it can be noticed, version one-shot is more complex and spans across several components and instantiations. “Controller” is the logical component whose function is performed

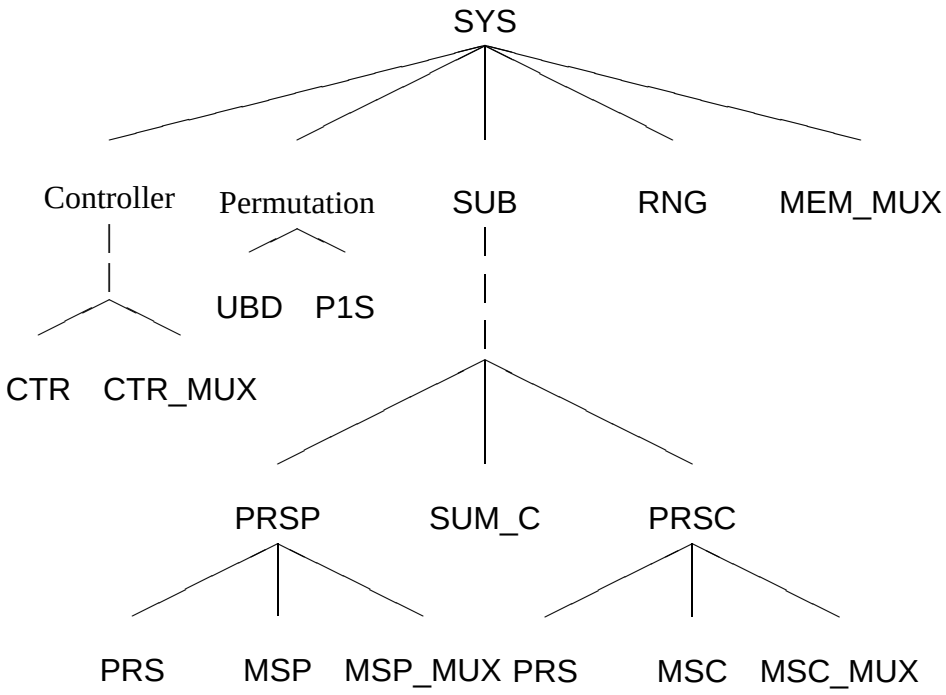


Figure 5.2: Tree diagram of the implementation one-shot.

by *substitution one-shot* described in subsection 4.3.5; CTR_MUX is the multiplexer that allows the initialization of the memory areas MSP and MSC. “Permutation” and SUB are described in section 4.3.1 and 4.3.1, respectively. MSP_MUX and MSC_MUX multiplexers are non used if the two memory banks MSP and MSC actually correspond to two different memory instantiations, as explained for versions osh3mem and opr3syn in section 4.4.1. RNG, in the end, is exactly the same architectures used in the prior section.

Not for all sub-components a test bench have been provided, especially because many of them are to small or simple to justify the design of a new architecture that

proves their functionality. It was preferred to assemble them in bigger entities which worked as the components they were derived from, and to write a small wrapper which used existent test benches.

In doing so, it was not possible to measure directly resource demands and performances for all entities. Nevertheless, the log files of the synthesis process was used to grasp an approximation of the values of slices, flip-flops, maximum delay and maximum frequency. These components are marked "only synthesis" on table 5.2 on the following page. On the other hand, for the wrapped version of components "Per" and S1S, performances have been checked using the same value of clock frequency as that used in the first implementation. This allows a rough but useful comparison between the two implementations.

Permutation

The new architecture of "Per" results slightly faster than its counterpart presented in table 5.2, because it is less complex and therefore better synthesisable — the maximum frequency has been increased by $12\,\mathrm{MHz}$. The Tcl script revealed also that 5 clock cycles per atom and 22 clock cycles to build a new packed data were necessary in the average.

Substitution

New substitution has been tested in its version provided with only one memory instance for MSP and MSC (serial), since this is the version that closer reassembles the behaviour of the SUB on table 5.2. Despite of the higher values of TP and TP_{cc}, the performance does not surprise because the generic parameters prsn_bits and data_bits were set for 32 bits. This means that only 1 reading cycle, in place of 4 of the first version, was necessary. To elaborate one prsn — i.e. read it, calculate the new value and write it back to the memory — an average of 31 clock cycles per shot was required.

System

The three versions of the one-shot system widely described in section 4.4.1 have been fully tested. It is worth noticing that values of time t presented in table 5.2 do not include the time required for initialization of the substitution's vectors. As it was expected, no substantial differences in terms of area exist between the three solutions, while great improvement, which pushes TP from $18\,\mathrm{Mbit/s}$ of the first version up to the present $67\,\mathrm{Mbit/s}$, was achieved.

The first speed up (see row osh2) is due to the fact that permutation and substitution communicate directly to each other without accessing the memory. In this case 25 clock cycles are necessary for reading, permutating and packing each data,

		Slices	FFs	LUTs	Δ_{max} (ns)	f_{max} (MHz)	f (MHz)	t (µs)	n_d (bit)	TP (Mbit/s)	n_{cc}	TP_{cc} (Mbit/clk)
UBD		25	34		15.25	63.9	only synthesis					
P1S		87	93		12.93	72.0						
Per		includes wrapper					60.0	376	$8 \cdot 4096$	87.3	22530	1.454
CTR		69	99		9.00	103.4	only synthesis					
PRS		76	152		7.48	127.1						
SUM_C		33			7.34	129.3						
S1S	ser	181	361		11.55	83.8	74.1	484	$32 \cdot 1024$	67.7	35842	0.914
	par						only synthesis					
	pre				12.75	81.6						
RNG		130	152	178*	11.82	101.7	not relevant					
SYS	osh2	688	736	842▷	14.9	70.8	63.2	972	$8 \cdot 4096$ Per $32 \cdot 1024$ SUB	33.7	61440	0.533
	osh3	677	734	830◇	14.7	67.7		729		45.0	46080	0.711
	opr3			829◇	15.1	66.5		486		67.4	30720	1.067

* 2 used as a route-thru ▷ 18 used as a route-thru ◇ 17 used as a route-thru

Table 5.2: Summary of the synthesis and place-and-route processes for the version one-shot.

while 31 cycles are required for accessing the memory 6 times and thus performing a substitution. This last value is reduced to 16 clock cycles with version osh3, where the two PRS vectors are accessed simultaneously. Just one cycle — i.e. the time to compute the sum — is necessary in the third architecture, since substitution's accesses are paralleled with permutation's.

6 Conclusions

The core operations of an encryption algorithm that makes use of the Kolmogorov flows was presented after an introduction on the significance of cryptology and chaos theory. The importance of developing tools such as VHDL and FPGA was examined and a design methodology established. A first architecture whose aim was to demonstrate chaotic cipher suitability for hardware implementation was discussed, limits investigated and a new approach called one-shot realized. Each solution was analyzed in order to report their area occupancy and throughput. The best result of 677 slices and $1.067\,\mathrm{Mbit/clk}$ has been obtained with the least system architecture called opr3.

As predicted in section 4.4.1, the real bottleneck is now represented by the "Per" component: the ratio permutation over substitution is currently 25/16. More work on this project should decrease this ratio making the former component more effective.

A hint could come from the observation that, under some conditions, a number of subsequent packed data at the output of permutation element are built picking out atoms from the same set of memory cells but with different position within each memory element. The present architecture do not detect this specific case and repeatedly reads the same values from the memory. One of these cases is represented by the simulation analyzed in this subsection.

In [4] several old and more recent algorithms have been implemented in VHDL and tested on a FPGA, a Xilinx Virtex XCV1000BG560-4. Comparing the results presented in section 5.2.2 to the performance evaluation tables found in that report, it appears clear that this algorithm places among the first implementation attempts made by the authors of the cited work. It rather reveals that it is far way form the $4.86\,\mathrm{Gbit/s}$ of Serpent architecture PP-32 or the $2.4\,\mathrm{Gbit/s}$ of RC6 architecture SP-10-2.

In the other hand, those architectures hardly surprising are much more greedy in term of area consumption: from 2,600 to nearly 11,000 slices against the 677 slices of the fastest version of the present work.

Acronyms

AWC *Add With Carry* is a class of pseudo-random number generators.

BUFT A *tristate buffer* is a driver with an additional control pin that let the output floating when disabled.

CBC *Cipher Block Chaining* uses the output of one encipherment step to modify the input of the next, so that each ciphertext block is dependent on all the previous plaintext blocks.

CFB CFB or *Ciphertext Feedback* is an operating mode for a block cipher. CFB is intended to provide some of the characteristics of a stream cipher from a block cipher. CFB is a way of using a block cipher to form a random number generator. The resulting pseudorandom confusion sequence can be combined with data as in the usual stream cipher. CFB assumes a shift register of the block cipher block size. An initial value first fills the register, and then is ciphered. Part of the result, often just a single byte, is used to cipher data, and the resulting ciphertext is also shifted into the register. The new register value is ciphered, producing another confusion value for use in stream ciphering. One disadvantage of this, of course, is the need for a full block-wide ciphering operation, typically for each data byte ciphered. The advantage is the ability to cipher individual characters, instead of requiring accumulation into a block before processing.

CLB A *Configurable Logic Block* is a physically co-located grouping of LUT's, flip flops, and carry logic.

CPU A *CPU* is the part of a computer that interprets and executes instructions.

DES *Data Encryption Standard* is a product cipher that operates on 64-bit blocks of data, using a 56-bit key.

DLL The *Delay-Locked Loops* can be used to eliminate skew between the clock input pad and the internal clock input pins throughout the device. Moreover, the DLL provides advance control of multiple clock domains and may operates as a clock mirror.

ECB *Electronic Codebook* is the straightforward use of a block cipher algorithm to encipher one block: a block of plaintext always encrypts into the same block of ciphertext.

EEPROM An *Electrically Erasable PROM* is a non-volatile storage device using a technique similar to the floating gates in EPROMs but with the capability to discharge the floating gate electrically. Usually bytes or words can be erased and reprogrammed individually during system operation.

EPROM An *Erasable-Programmable Read-Only Memory* is a PROM that can be erased by exposure to ultraviolet light and then reprogrammed.

FPGA A *Field-Programmable Gate Array* is a type of logic chip that can be programmed. An FPGA is similar to a PLD, but whereas PLDs are generally limited to hundreds of gates, FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs. Once the design is set, hardwired chips are produced for faster performance.

GUI An interface for issuing commands to a computer utilizing a pointing device, as a mouse, that manipulates and activates graphical images on a monitor.

HDL *Hardware Description Language* is a kind of language used for the conceptual design of integrated circuits. Examples are VHDL and Verilog.

IC An *Integrated Circuit* is a tiny slice or chip of material on which is etched or imprinted a complex of electronic components and their interconnections.

IEEE The *Institute of Electrical and Electronics Engineers*, founded in 1884, is an organization composed of engineers, scientists, and students. The IEEE is best know for developing standards for the computer and electronics industry.

IOB a physically co-located group of buffers, latches, flip flops, and input-/output pads used for sending signals off of the FPGA and receiving signals onto the FPGA.

LUT A *Look Up Table* is a block of logic in a CLB that uses SRAM technology to implement asychronous digital logic.

MPGA The *Metal Programmed Gate Array* family provides a low-risk conversion path from programmable gate arrays to production quantity devices. By significantly reducing the production costs of a product without technical or time-to-market risks, MPGAs prolong the life cycle of a finished design.

NCD An FPGA file that holds mapping, placement, and routing data about a design implemenation. A valid NCD can hold mapping information, or mapping/routing information, or mapping/placement/routing information.

NGD An FPGA file that holds logical netlist information about a design.

NGM An FPGA file that holds information about optimized logic and netlists. An NGM file is used by the *backannotation* process to reconstruct the original netlist from the optimized NCD netlist for timing simulation. This enables you to use the functional simulation testbench in timing simulation.

PCF A file that contains timing and location constraints of the logic in the physical domain.

PGM+ *Portable Graymap* format for gray scale images.

PGP *Pretty Good Privacy* is an encryption program based on RSA public-key cryptography. PGP allows users to exchange files and messages, with both privacy and authentication, over all kinds of networks. Because PGP is based on public-key cryptography, no secure channels are needed to exchange keys between users. PGP can also provide digital signatures for files or messages.

PLA A *Programmable Logic Array* is a PLD that offers flexible features for more complex designs.

PLD A *Programmable Logic Device* is an integrated circuit that can be programmed in a laboratory to perform complex functions. A PLD consists of arrays of AND and OR gates. A system designer implements a logic design with a device programmer that blows fuses on the PLD to control gate operation.

PRNG A *Pseudo-Random Number Generator* refers to any computer random number generator which is not explicitly labeled as “physically random”, “really random”, or other such description.

PROM A *Programmable Read-Only Memory* is a memory that can be programmed only once.

PRS A *Pseudo-Random Sequence* is a sequence of pseudo-random numbers generated by a PRNG.

RAM Pronounced *ramm*, acronym for *Random Access Memory*, a type of computer memory that can be accessed randomly; that is, any byte of memory can be accessed without touching the preceding bytes.

RCS A version control system that automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, and form letters.

RISC A *Reduced Instruction Set Computer* is a type of microprocessor that recognizes a relatively limited number of instructions.

RSA An public-key encryption technology developed by RSA Data Security, Inc. The acronym stands for Rivest, Shamir and Adelman, the inventors of the technique. The RSA algorithm is based on the fact that there is no efficient way to factor very large numbers.

RTL *Register Transfer Language* is a kind of Hardware Description Language (HDL) used in describing the registers of a computer or digital electronic system, and the way in which data is transferred between them.

SDF A *Standard Delay Format* file holds timing information for circuit simulation.

SRAM A *Static RAM* is a device in which each bit of storage is a bistable flip-flop, commonly consisting of cross-coupled inverters. It is called "static" because it will retain a value as long as power is supplied, unlike dynamic random access memory which must be regularly refreshed.

SWB *Subtract With Borrow* is a class of pseudo-random number generators.

VHDL *VHSIC Hardware Description Language* is a large high-level VLSI design language with Ada-like syntax. It arose out of the United State government's Very High-Speed Integrated Circuit (VHSIC) program. The DoD standard for hardware description, now standardised as IEEE 1076.

VHSIC *Very High-Speed Integrated Circuit* is a very high-speed computer chip which uses LSI and very large scale integration VLSI technology.

VLSI *Very Large-Scale Integration* is the process of placing thousands (or hundreds of thousands) of electronic components on a single chip.

LSI *Large-Scale Integration* is the process of placing hundreds of electronic components on a single chip.

Bibliography

[1] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1996.

[2] D. Dineen. Design space evaluation of the IDEA encryption algorithm. Master's thesis, National Microelectronics Reserch Center, University College, Cork, Ireland, May 1998.

[3] A. J. Elbirt and C. Paar. An FPGA implementation and performance evaluation of the serpent block chiper. Preprint to be presented at FPGA2000, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, USA, 2000. Document `elbirtpaarserpentfpga2000.pdf` available at `http://ece.wpi.edu/Research/crypt/publications/documents/`.

[4] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA implementation and performance evaluation of the AES block chiper candidate algorithm finalists. Technical report, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, USA, April 2000. Document `aelbirtetalAES3.pdf` available at `http://ece.wpi.edu/Research/crypt/publications/documents/`.

[5] *VHDL Modelling Guidelines*, September 1994. `ftp://ftp.estec.esa.nl/pub/vhdl/doc/ModelGuide.pdf`.

[6] S. Kirkpatrick and E. P. Stoll. A very fast shift-register sequence random number generator. *Journal of Computational Physics*, 40:517–526, 1981.

[7] Z. Kotulski and J. Szczepañski. Discrete chaotic cryptography (DCC). Technical report, Institute of Fundamental Technological Reaserch, Polish Academy of Sciences, 1997. `http://www.wspc.com/journals/ijbc/09/0906/kotu.html`.

[8] G. Marsaglia and A. Zaman. A new class of random number generators. *The Annals of Applied Probability*, 1(3):462–480, August 1991.

[9] M. Riaz and H. M. Heys. The FPGA implementation of the RC6 and CAST-256 encryption algorithm. In *Electrical and Computer Engineering*, Edmonton, Alberta, May 1999. IEEE Canadian Conference on Electrical and Computer Engineering CCECE '99. `http://www.engr.mun.ca/~howard/PAPERS/fpga.ps`.

[10] J. Scharinger. Fast encryption of image using chaotic kolmogorov flows. Technical report, Johannes Kepler University, Department of System Theory, Linz, Austria, April 1998. Document `Scharinger98b.htm` available at `http://www.cast.uni-linz.ac.at/Department/Publications/Pubs1998/`.

[11] J. Scharinger. Secure and fast encryption using chaotic kolmogorov flows. Technical report, Johannes Kepler University, Department of System Theory, June 1998. Document `Scharinger98f.htm` available at `http://www.cast.uni-linz.ac.at/Department/Publications/Pubs1998/`.

[12] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[13] S. Sjoholm and L. Lindh. *VHDL for Designers*. Prentice Hall, 1997.

[14] S. H. Strogatz. *Nonlinear Dynamics and Chaos*. Addison-Wesley, 1994.

[15] Xilinx. *Using the Virtex Block SelectRAM+ Feature*, March 2000. `http://support.xilinx.com/xapp/xapp130.pdf`.

[16] Xilinx. *Virtex-E 1.8 V Field Programmable Gate Arrays*, May 2000. `http://support.xilinx.com/partinfo/ds022.pdf`.

[17] Dictionary.com. `http://www.dictionary.com/`.

[18] Webopedia. `http://webopedia.internet.com/`.

[19] Xilinx's glossary page. `http://www.xilinx.com/support/techsup/journals/implmnt/glossary.htm`.