ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Μεταπτυχιακή Διατριβή

Design and FPGA Implementation of the SCAN Encryption Algorithm

Χριστόφορος Κάχρης

Επιβλέπων Καθηγητης: Καθηγητής Απόστολος Δόλλας

Εξεταστική επιτροπή : Καθηγητής Α. Δόλλας
Αν. Καθηγητής Κ. Καλαϊτζάκης
Αν. Καθηγητής Δ. Πνευματικάτος

Ιούνιος 2003
Χανιά

Acknowledgments

I would like to acknowledge the guidance provided by my advisor, Prof. Dollas at every stage of work on this thesis.

I also would like to thank Prof. D. Pneumatikatos and Prof. K. Kalaizakis is for reviewing my thesis and participating in my thesis defense.

Table of Contents

Chapter 1
Introduction

Encryption, the conversion of information into code, which is intelligible only for an authorized receiver, has intrigued men since ancient times. Historically, cryptographic techniques have been developed for diplomatic or military applications but today they can be found everywhere in private and public sectors where confidential information is crucial. The first system of military cryptography was the "skytale", invented by the Spartans as early as the fifth century B.C [1]. The secret message was on the parchment down the length of the skytale. The parchment is then unwound and send on its way, where it can only be read if it is wrapped around a baton of the same thickness as the first.

Cryptology, the science of code and cipher systems, start to develop not until the First World War. Until then few papers about cryptology have been published. The first notable paper about cryptography was Claude Shannon's paper "The Communication Theory of Secrecy Systems", which appears in 1949 [2]. The revolution of telecommunication, which has produced a vast amount of transmitted confidential data, has initiated the flush of the cryptology.

However, the science of cryptology differs from all the other sciences in a rather striking way: there is no public feedback about how cryptographic systems fail. Designers of cryptographic systems are at a disadvantage to most other engineers, in that information on how or if their systems fail is hard to get. The major users have traditionally been government agencies, which are very secretive about their mistakes. The National Security Agency (once called "No Such Agency") spends thousands of millions dollars to cryptanalyze new algorithms and the publition of this achievement would meaning that this algorithm will stop being used. Cryptology also presents a difficulty not found in normal academic disciplines: the need for the proper interaction of cryptography and cryptanalysis. Exposing flaws in these designs is far harder than designing them in the first place. The result is that the competitive process, which is one strong motivation in academic research, cannot take hold, since direct comparisons on how secure is one algorithm against the other is very difficult.

The last few years many encryption algorithms have been proposed, both in hardware and in software. The most widely used is the Data Encryption Standard (DES) developed at IBM [3, 4]. However, the recent cryptanalysis of the DES algorithm [7] and the A5/1 algorithm [5.6] (used in GSM cellular phones) has initiated a race for a new secure algorithm. In this thesis a real-time hardware implementation of the SCAN algorithm is presented, which is mainly targeting image encryption but can also be applied in text data or compressed video data. The SCAN encryption algorithm is a secret key block encryption algorithm, which divides the data into a series of blocks of equal length, and these blocks are sequentially processed using a key known to the sender and receiver exclusively.

The thesis is divided into 6 chapters. Chapter 2 describes several block cipher algorithms implemented in hardware that have been proposed in academic and commercial areas. Chapter 3 provides the theoretical details of the SCAN algorithm and provides some information of the algorithm, implemented in software. Chapter 4 describes the SCAN architecture that has been developed in hardware. Chapter 5 provides the performance of this architecture implemented in reconfigurable logic. Finally, chapter 6 discusses some conclusions of this work and any future work that can be done to expand the capabilities of this architecture.

Chapter 2
Relevant Research

Several algorithms have been proposed to encrypt images. Most image encryption algorithms are based on position permutations with or without confusion functions [8, 9, 10, 11] where the pixel values are scrambled to different positions on the 2D array. The other algorithms are based on chaos transformations [12,13,14] where chaotic binary sequences are generated for the rearrangement of the image pixels, on tree structures [15,16] where the pixel values are transformed by using certain functions, or on other methods [17,18,19] like the quantization based approaches. Furthermore, many image encryption algorithms have been implemented in software, due to the resulting flexibility and the algorithm complexity. This means that most of these algorithms are good for non-real-time image encryption but not for compressed video encryption. Very few widely used image encryption algorithms have been implemented in hardware, and generally in conjunction with image compression. On the other hand, many block cipher, which are used for encrypting text data, have been used to encrypt images.

In this section, we briefly discuss some image algorithms and the corresponding implementations in hardware. In sections, 2.1 and 2.3 we present three block cipher algorithms, designed specially for image encryption, that have been proposed and are based on chaos methods and tree structures, while in sections 2.4 through 2.6 we present some general block cipher algorithms implemented in hardware that can be used to encrypt images.

2.1 A Chaotic Mirror-Like Image Encryption Algorithm

Yen *et al.* [20, 21] have proposed the Chaotic Mirror-Like Image Encryption Algorithm (CMLIE). This algorithm belongs to the position permutation algorithms. Based on a binary sequence generated from a chaotic system, image pixels are rearranged according to the defined swapping operations. It possesses the features of low computational complexity, high security, and no distortion according to its author. Moreover, based on a look-up-table generated by the CMLIE algorithm is capable of being integrated with JPEG and MPEG.

The associated VLSI architecture design for the proposed image encryption and decryption algorithm is shown in figure 2-1. The basic idea of the CMLIE algorithm is to rearrange the image pixels by way of mirror-like operations according to a random sequence derived from chaotic systems. Instead of using shift registers to shift the image pixels directly, a 1D look-up table stored in memory buffers is generated and then used to permute the image pixels. Initially, there is one memory buffer storing the data that the value in position x is x for $0 \le x \le M \times N - 1$. According to the four different kinds of mirror-like operations in the algorithm, the transmitted sequence of the image pixels is generated by repeatedly using memory read and write operations accessed from the memory buffers, which store the order of the transmitted sequence. After four iterations, the order of the transmitted sequence is used to act as the addresses where the image pixels are accessed and transmitted. The key to the scheme for realizing the algorithm lies in the address generation scheme. The operating performance of this implementation is 39.68 MHz, targeting an ALTERA EPF10K50VRC240-1 FPGA. According to the authors, this implementation can be efficiently integrated with the image compression standards JPEG and MPEG.

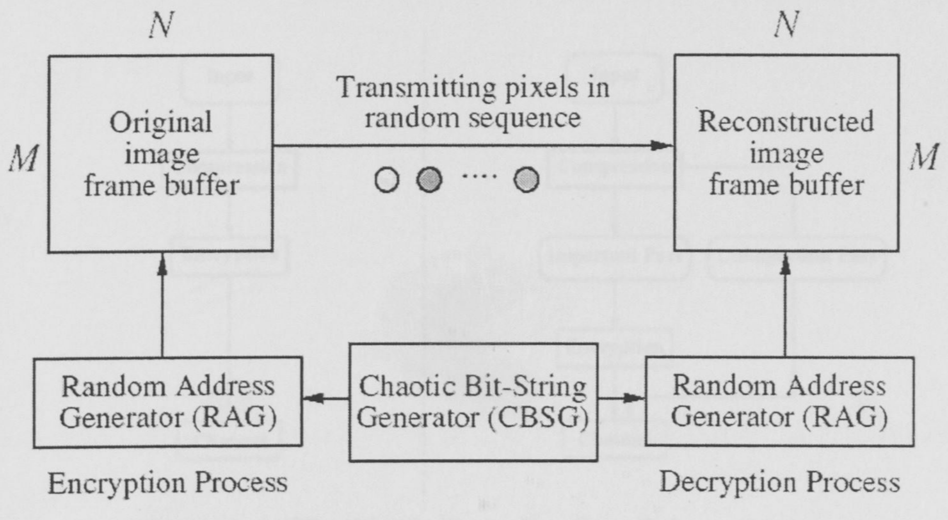

N
N
M
Original image frame buffer
Transmitting pixels in random sequence
Reconstructed image frame buffer
M
Random Address Generator (RAG)
Chaotic Bit-String Generator (CBSG)
Random Address Generator (RAG)
Encryption Process
Decryption Process

Figure 2-1. Chaotic Image Encryption

2.2 Partial Encryption of Compressed Images and Videos

Another approach to encrypt images is to combine the compression and the encryption in order to eliminate the demanding distinct processing. Cheng [15, 22] propose a novel approach called partial encryption in order to reduce encryption and decryption time in image and video communication and processing. In this approach, only part of the compressed data is encrypted, as shown in figure 2-2. The proposed algorithm can be applied in schemes of quad tree and wavelet image compression, as well as an extension for video compression. Partial encryption allows the encryption and decryption time to be significantly reduced without affecting the compression performance of the underlying compression algorithm. It is also shown that although a large portion of the compressed data is left unencrypted, it is difficult to recover the original data without decrypting the encrypted part. In the case of quad tree image compression the encrypted portion is 13%-27% of the compressed output for typical images. For wavelet compression based on zero trees, less than 2% of the compressed output is encrypted for 512x512 images. The results on video compression are similar.

The proposed algorithm has not been implemented in hardware in order to known its potential throughput.

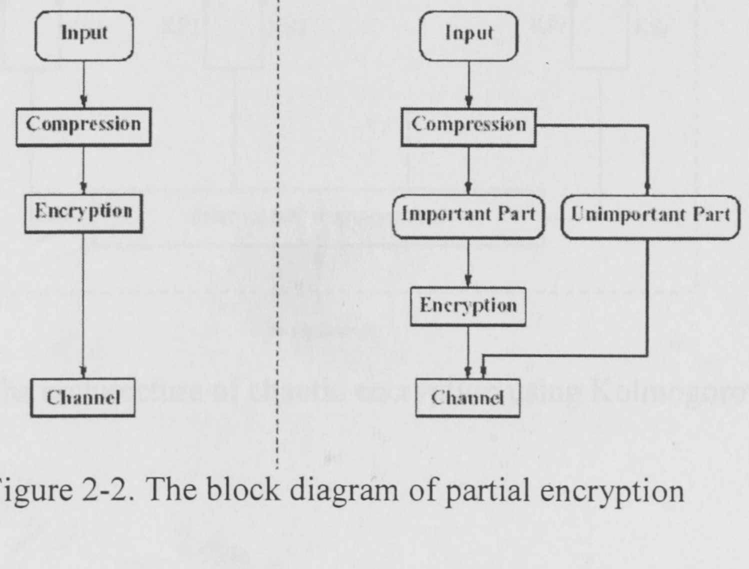


Figure 2-2. The block diagram of partial encryption

2.3 Chaotic Encryption using Kolmogorov flows

Combining two apparently distant sciences like cryptography and chaos theory, J. Scharinger [24, 25] 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. Cappelletti [26] has presented an FPGA implementation of this algorithm. The block diagram of this implementation is shown in figure 2-3. The cipher performs an encryption iterating the same algorithm for r rounds. According to the author, a number of rounds at least equal to 12 is recommended. Each iteration consists of a permutation and a substitution. The permutation component is responsible for the actualization of the diffusion concept. Each data, which composes the plain block at the input, is transposed to a new position at the output. This transformation follows the rules dictated by the chaotic Kolmogorov flow. Confusion is accomplished by the substitution component. The implementation is targeting a Xilinx Virtex FPGA and its performance is about 45 Mbits/sec using 67.7 MHz clock frequency.

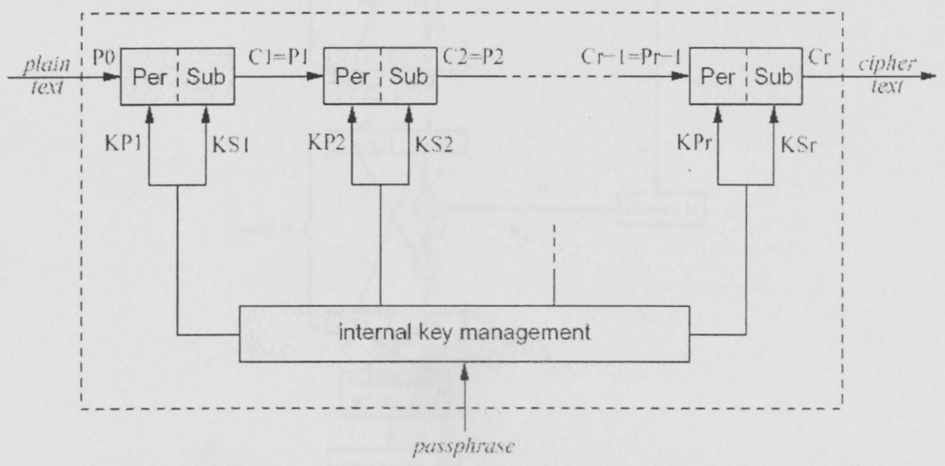


Figure 2-3. The architecture of chaotic encryption using Kolmogorov flows

2.4 The DES Algorithm

One of the wide known block cipher using private key is the DES algorithm. It was proposed by IBM during the early 1970s and it was adopted as a federal standard on November 23, 1976. In [7] it is claimed that with \$150K it is possible to crack the DES algorithm using a known plain/cipher text in only two hours. This amount is well within the budget of all countries and large and medium sized companies, any well ran criminal organization, and most terrorist group. There are many DES variants, but the most possible successor is the triple DES which is much harder to break using exhaustive search: 2^{112} attempts instead of 2^{56} attempts. The fastest implementation of the DES algorithm in hardware is reported in [29] providing a throughput of 400Mbytes/sec. In [23] there is an application of the DES algorithm in order to encrypt images, unfortunately without providing any quality results. The block diagram of the DES algorithm is shown in figure 2-4.

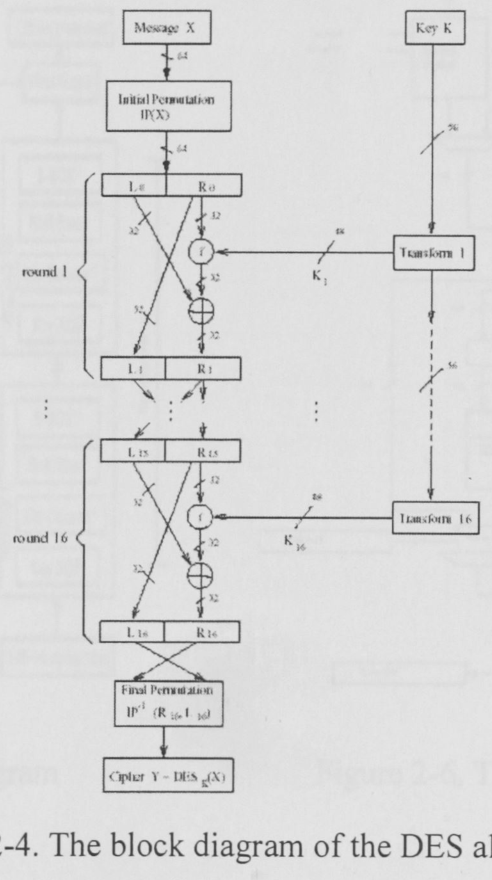
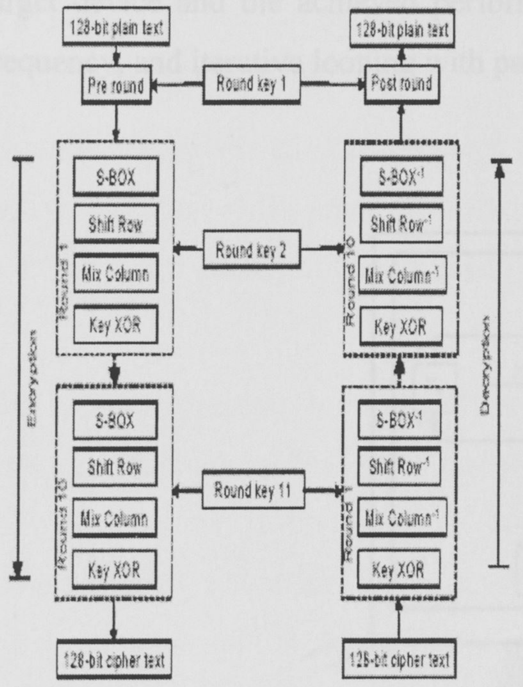


Figure 2-4. The block diagram of the DES algorithm,

2.5 Rijndael Advanced Encryption Standard

Rijndael is a private-key symmetric block encryption algorithm that supports 128, 192, and 256-bit length keys and operates on 128, 192 and 256-bit blocks. All nine combinations of key length and block size are possible. Recently, Rijndael was selected as the Advanced Encryption Standard (AES) to replace DES. Karri [27] has presented an FPGA implementation of this algorithm. The Rijndael encryption algorithm is shown in figure 2-5. The round transformation data path shown in Figure 2-6 implements the byte substitution, shift row, mix column and key xor operations. The data path consists of two 16x8 SRAMs (SRAM 0 and SRAM 1), one 256x8 ROM (SBOX), two 32-bit registers (REG_A and REG_B) and three multiplexers. The total design targets the Wildforce reconfigurable computing board and its performance is 124 Mbits/s using 13.6 MHz clock frequency.



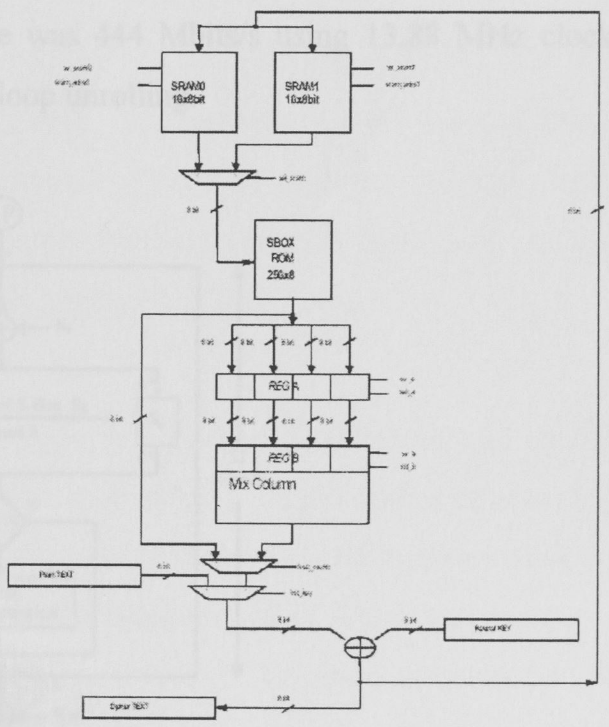


Figure 2-5. The block diagram

Figure 2-6. The architecture

2.6 The Serpent Block Cipher Algorithm

An other candidate of the AES was the Serpent Block Algorithm. The Serpent algorithm is a 32-round Substitution-Permutation (SP) network operating on four 32-bit words. The algorithm encrypts and decrypts 128-bit input data via a key of 128, 192, or 256 bits in length. The Serpent algorithm consists of three main components:

- Initial Permutation IP
- Thirty-two rounds consisting of a Round Function that performs Key Masking, S-Box Substitution, and data mixing via a Linear Transformation
- Final Permutation FP

A block diagram for the Serpent algorithm is shown in figure 2-7. Elbirt [28] has presented an FPGA implementation of this algorithm using various architecture variants, such as iterative looping and loop unrolling. The Xilinx Virtex FPGA was selected as the target device and the achieved performance was 444 Mbits/s using 13.88 MHz clock frequency, and iterative looping with partial loop unrolling.

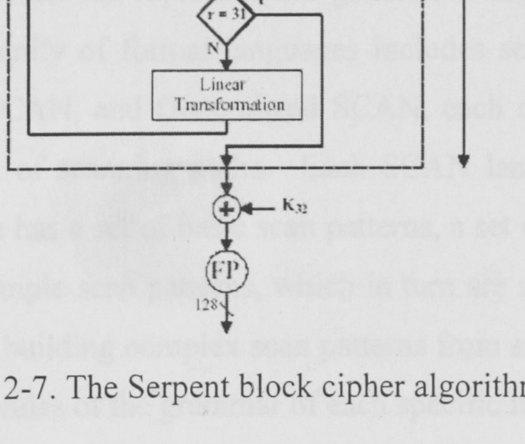


Figure 2-7. The Serpent block cipher algorithm

Chapter 3
The SCAN Algorithm

In reality, SCAN [8 - 11] is a class of formal languages, which can be applied to compression, encryption, data hiding, or combinations thereof. This section describes the SCAN language in detail and provides some experimental results on the quality of the methodology obtained from the implementation of the algorithm in software [10]. The language and the experimental results are not part of this thesis, but their included to provide the specifications of the architecture and a base that can be used to compare with the results of the implementation in hardware.

3.1 SCAN Methodology

A scanning of a two dimensional array $P_{m \times n} = \{p(i, j) : 1 \leq i \leq m,\ 1 \leq j \leq n\}$ is a bijective function from $P_{m \times n}$ to the set $\{1, 2, . . \ mn\text{-}1, mn\}$. In other words, a scanning of a two dimensional array is an order in which each element of the array is accessed exactly once, or a permutation of the array elements. The terms scanning, scanning path, Scan pattern, and Scan word are used interchangeably in this paper.

The SCAN represents a family of formal languages based on two-dimensional spatial accessing methodologies, which can represent and generate a large number of scanning paths easily. The SCAN family of formal languages includes several versions such as Simple SCAN, Extended SCAN, and Generalized SCAN, each of which can represent and generate a specific set of scanning paths. Each SCAN language is defined by a grammar and each language has a set of basic scan patterns, a set of transformations, and a set of rules to compose simple scan patterns, which in turn are used to obtain complex scan patterns. The rules for building complex scan patterns from simple scan patterns are specified by the production rules of the grammar of each specific language.

3.2 SCAN-based encryption

The basic idea of the proposed encryption method is to rearrange the pixels of the image and change the pixel values. The rearrangement is done by a set of scanning patterns (encryption keys) generated by an encryption-specific SCAN language, which is formally defined by the grammar $G = (\Gamma, \Sigma, A, \Pi)$. Grammar G comprises of non-terminal symbols $\Gamma = \{A, S, P, U, V, T\}$, of terminal symbols $\Sigma = \{c, d$, o, s, $r, a, e, m, y, w, b, z, x, B, Z, X$, *(,)*, *space*, 0, 1, 2, 3, 4, 5, 6, 7}, its start symbol is A, and its production rules Π are given by:

$A \rightarrow S \mid P$
$S \rightarrow UT$
$P \rightarrow VT(A\ A\ A\ A)$
$U \rightarrow c \mid d \mid o \mid s \mid r \mid a \mid e \mid m \mid y \mid w \mid b \mid z \mid x$
$V \rightarrow B \mid Z \mid X$
$\mathrm{T} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$

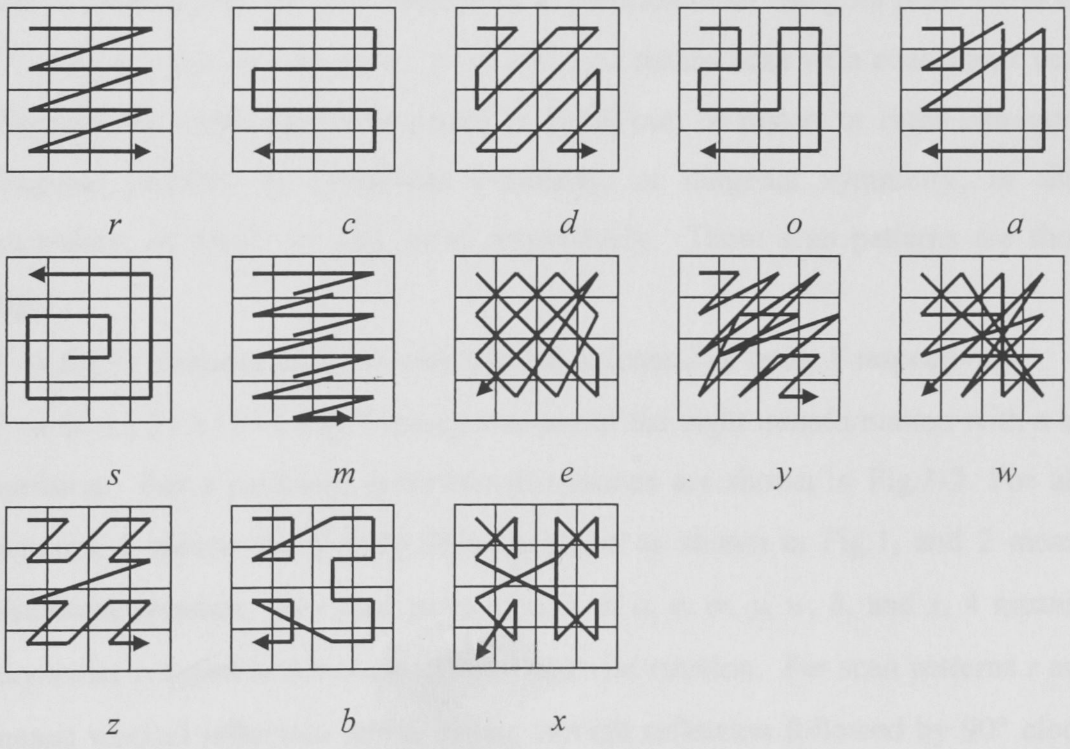


Fig. 3-1. Basic scan patterns

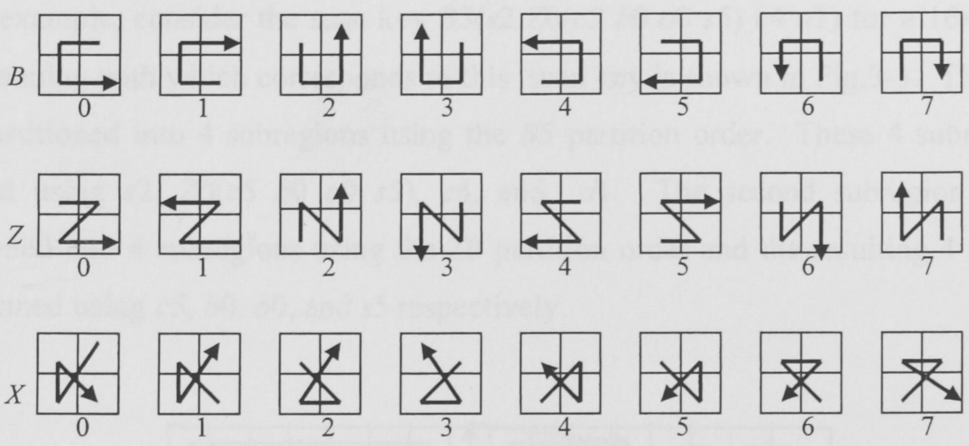


Fig. 3-2. Partition patterns and transformations

The semantics of this encryption-specific SCAN language are described as follows:

(a) $A \rightarrow S \mid P$ means process the region by scan S or partition P.

(b) $S \rightarrow UT$ means scan the region with scan pattern U and transformation T.

(c) $P \rightarrow VT(A\ A\ A\ A)$ means partition the region with partition V and transformation T, and process each of the four subregions in partition order using As from left to right.

(d) $U \rightarrow c \mid d \mid o \mid s \mid r \mid a \mid e \mid m \mid y \mid w \mid b \mid z \mid x$ means scan with continuous raster, or diagonal, or continuous orthogonal, or spiral out, or raster, or right orthogonal, or diagonal parallel, or horizontal symmetry, or diagonal symmetry, or diagonal secondary, or block, or zeta, or xi respectively. These scan patterns are shown in Fig.1.

(e) $V \rightarrow B \mid Z \mid X$ means partition with letter B or letter Z or letter X respectively.

(f) $T \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$ means use one of the eight transformation with a scan or partition. For a partition, these transformations are shown in Fig.3-2. For all scan patterns, 0 means the identity transformation as shown in Fig.1, and 2 means 90° clockwise rotation. For scan patterns c, o, s, a, e, m, y, w, b, and x, 4 means 180° clockwise rotation and 6 means 270° clockwise rotation. For scan patterns r and z, 4 means vertical reflection and 6 means vertical reflection followed by 90° clockwise rotation. For scan pattern d, 4 means 90° clockwise rotation followed by horizontal refection and 6 means 180° clockwise rotation followed by vertical refection. For all scan patterns, 1, 3, 5, and 7 are reverses of scanning paths specified by 0, 2, 4, and 6 respectively.

As an example, consider the scan key *B*5(*s*2 *Z*0(*c*5 *b*0 *o*0 *s*5) *c*4 *d*1) for a 16×16 image. The scanning path which corresponds to this scan key is shown in Fig.3-3. The image is first partitioned into 4 subregions using the *B*5 partition order. These 4 subregions are scanned using *s*2, *Z*0(*c*5 *b*0 *o*0 *s*5), *c*4, and *d*1. The second subregion is further partitioned into 4 subregions using the *Z*0 partition order and the resulting 4 subregions are scanned using *c*5, *b*0, *o*0, and *s*5 respectively.

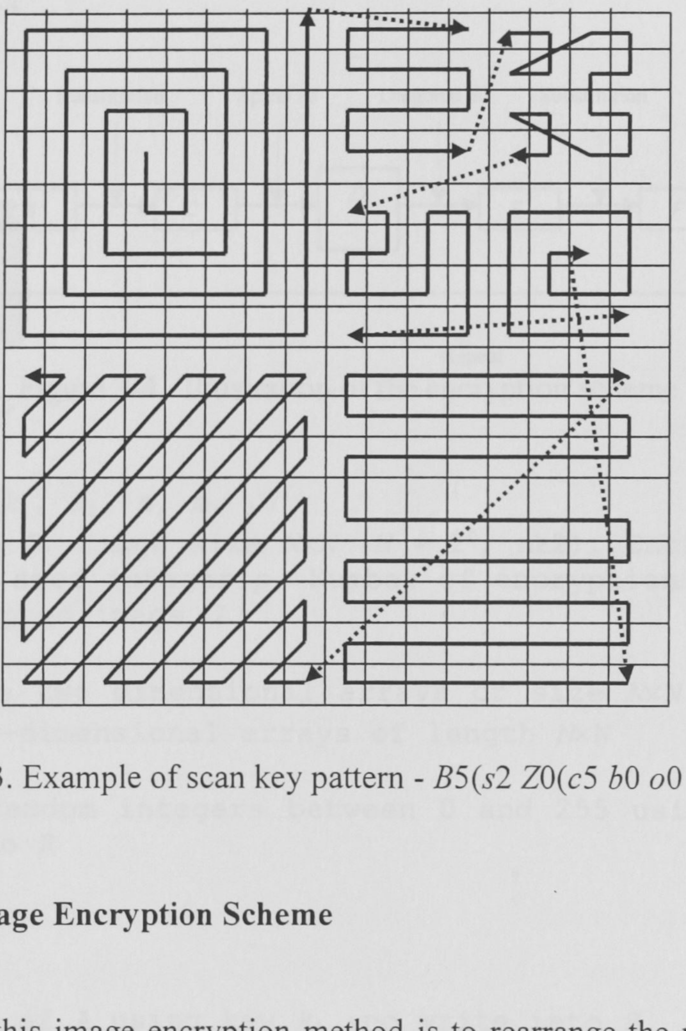


Figure 3-3. Example of scan key pattern - *B*5(*s*2 *Z*0(*c*5 *b*0 *o*0 *s*5) *c*4 *d*1)

3.3 The SCAN Image Encryption Scheme

The basic idea of this image encryption method is to rearrange the pixels of the image and change the pixel values. The pixel rearrangement is done by scan keys. The pixel values are changed by a simple substitution mechanism, which adds confusion and diffusion properties to the encryption method. The permutation and substitution operations are applied in intertwined and iterative manner. First, the encryption algorithm is described in detail. Next, the confusion and diffusion properties of the algorithm are presented in detail, and experimental results are shown to demonstrate these properties of the encryption method, as well as pixel rearrangement. Finally, various extensions of the encryption method and the size of the encryption key space are discussed.

3.4 The SCAN Image Encryption Algorithm

The encryption is done by the *Encrypt*( ) function which is described below, and is illustrated in Fig.3-4.

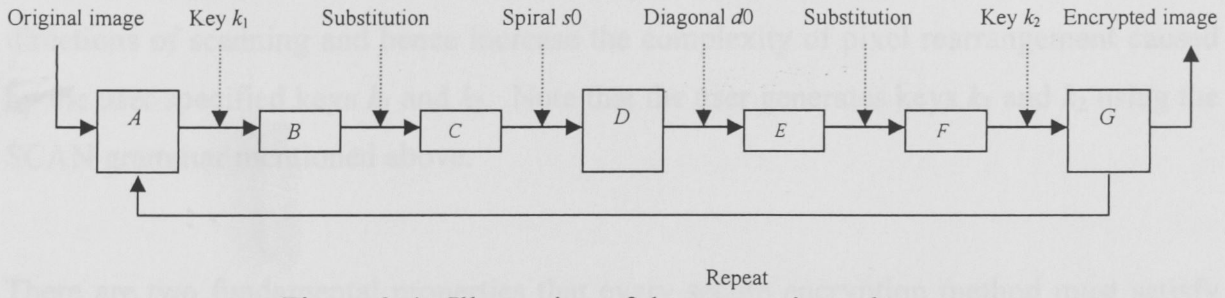


Figure 3-4. Illustration of the encryption scheme

```
Encrypt(I, N, k1, k2, p, m, J)
Inputs:  Image I, Image size N×N (N = 2^n, n≥2), Encryption keys k1
and k2, Random seed integer p, Number of encryption iterations m
Output:  Encrypted image J
{
Let A, D, G be two dimensional arrays of size N×N and let B, C,
E, F, R be one-dimensional arrays of length N×N

Generate N×N random integers between 0 and 255 using random seed
p and assign to R

Copy I into A
Repeat m times
  {
   Read pixels of A using key k1 and write into B
   C[1]=B[1]
   C[j]=(B[j]+((C[j-1]+1)R[j])mod256)mod256, for 2≤j≤N×N
   Read pixels of C and write into D using spiral key s0
   Read pixels of D using diagonal key d0 and write into E
   F[1]=E[1]
   F[j]=(E[j]+((F[j-1]+1)R[j])mod256)mod256 for 2≤j≤N×N
   Read pixels of F and write into G using key k2
  }
Copy G into J and return J
}
```

The encryption key actually consists of four components, namely, the two scan keys k_1 and k_2, the random seed integer p, and the number of encryption iterations m. These four encryption key components are known to both the sender and the receiver before the communication of encrypted image. The random numbers needed by *Encrypt*() can be obtained by any method such as a linear congruential generator with seed p. The encryption algorithm uses four scan keys to increase the complexity of pixel rearrangement. The keys k_1 and k_2 are specified by the user as part encryption key. The other two keys spiral $s0$ and diagonal $d0$ (shown in Fig. 3-1) are fixed as part of encryption algorithm. These two keys $s0$ and $d0$ are chosen because they have opposite directions of scanning and hence increase the complexity of pixel rearrangement caused by the user specified keys k_1 and k_2. Note that the user generates keys k_1 and k_2 using the SCAN grammar mentioned above.

There are two fundamental properties that every secure encryption method must satisfy [3, 4]. The first is the confusion property, which requires that ciphertexts (encrypted data) have random appearance (uniformly distributed pixel values). The second is the diffusion property with respect to plaintexts (original data) and keys, which requires that similar plain texts produce completely different ciphertexts when encrypted with the same key, and similar keys produce completely different ciphertexts when encrypting the same plaintext. The proposed encryption method satisfies both the confusion and diffusion properties, as explained below.

The confusion and diffusion properties are achieved by transforming the sequence B into sequence C using $C[j]=(B[j]+((C[j-1]+1)R[j])\bmod 256)\bmod 256$, and similarly, sequence E into sequence F. The sequences C and F get uniformly distributed pixel values because uniform random sequence R is used to multiply the pixel values in the transformation. Since pixels in F are placed in G, G also gets uniformly distributed pixel values and gets the confusion property. The sequence C gets the diffusion property because a single change in value $B[j]$ changes $C[j]$, which changes $C[j+1]$, which changes $C[j+2]$ and these changes propagate up to the end of the sequence $C[N\times N]$. It can be shown that a single pixel change in A causes all pixels in G to be changed in one iteration as follows: Suppose a single pixel is changed in A. Then, the corresponding pixel at some location j in B changes. Subsequently, all pixels between j and $N\times N$ in C change. Then, the corresponding pixels in D including the top left pixel change (because spiral scan ends at top left corner). Then, the corresponding pixels in E including the first pixel change (because diagonal scan begins at top left corner). Then, all pixels in F change and then all pixels in G change. A single change in encryption scan key also changes all pixels in G in one iteration, because a change in scan key causes at least one pixel at some location in B to be changed, which causes all pixels in G to be changed as shown above.

The decryption is done by reversing the operations of encryption. Note that the decryption requires the encryption key which consists of k_1, k_2, p and m. Decryption is done as follows: Read pixels of G using key k_2 and write into F. Then, transform F into E by $E[1]=F[1]$, $E[j]=(F[j]-((F[j-1]+1)R[j])\bmod 256)\bmod 256$ for $2\leq j\leq N\times N$. Then, read pixels of E and write into D using diagonal scan $d0$. Then, read pixels of D using the spiral scan $s0$ and write into C. Then, transform C into B by $B[1]=C[1]$, $B[j]=(C[j]-((C[j-1]+1)R[j])\bmod 256)\bmod 256$ for $2\leq j\leq N\times N$. Then, read pixels of B and write into A using key k_1. Repeat this process m times to get the decrypted image. Note that the random array R is obtained with random seed p.

3.5 Experimental Results

Several experiments were conducted to test various properties of the SCAN image encryption method, which include pixel rearrangement, confusion, and diffusion. In all the following experiments, a fixed sequence of random numbers was generated by the C language library random number generator with seed 100, and used in the *Encrypt*() function. Note also that all images are of size 256×256.

(a) The *Encrypt*() algorithm was first used to encrypt the Lena image with the encryption scan keys $B2(x0\ y5\ s6\ r3)$, $c5$ and five encryption iterations. The original and encrypted images are shown in Fig.3-5.

(b) In order to determine how well *Encrypt*() rearranges the pixels, the confusion and diffusion parts (i.e transforming *B* to *C* and *E* to *F*) of *Encrypt*() were eliminated and only the rearrangement parts of *Encrypt*() were used with keys *B*2(*x*0 *y*5 *s*6 *r*3), *c*5. The encrypted images of Lena after one, two, and five iterations are shown in Fig. 3-6. The pixels are rearranged in completely random looking manner in just a few iterations.

(c) In order to measure the dispersion at a point and at a block, two measures are defined as follows.

$PSpread_{k,n}(p) = (\Sigma dist(E(p), E(q)))/|N(p)|,\ q \varepsilon N(p)$

$BSpread_{k,n}(B) = (\Sigma PSpread_{k,n}(p))/|B|,\ p \varepsilon B$

where *p* is any point in the original image, *N*(*p*) is the set of neighboring points of *p* in the original image, *k* is a set two encryption scan keys, *n* is the number of encryption iterations, *E*() function is the permutation induced by scan key *k* and iteration *n*, *dist*() is the Euclidean distance function, *B* is any region in the original image, and | | is the magnitude in number of pixels.

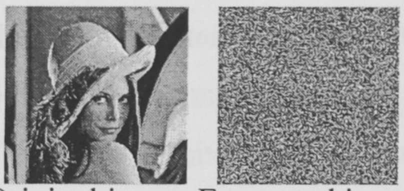

Original image Encrypted image

Fig. 3-5. Encryption of Lena image using *Encrypt*()

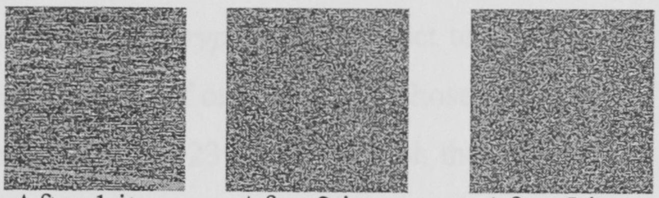

After 1 iter. After 2 iter. After 5 iter.

Fig. 3-6. Pixel rearrangement property of *Encrypt*()

(d) In order to determine whether large spreads occur for blocks at any locations and with any scan keys, 10×10 blocks were chosen at 100 random locations and at each location two random scan keys were generated, and the spread was computed for these blocks and keys after 10 iterations.

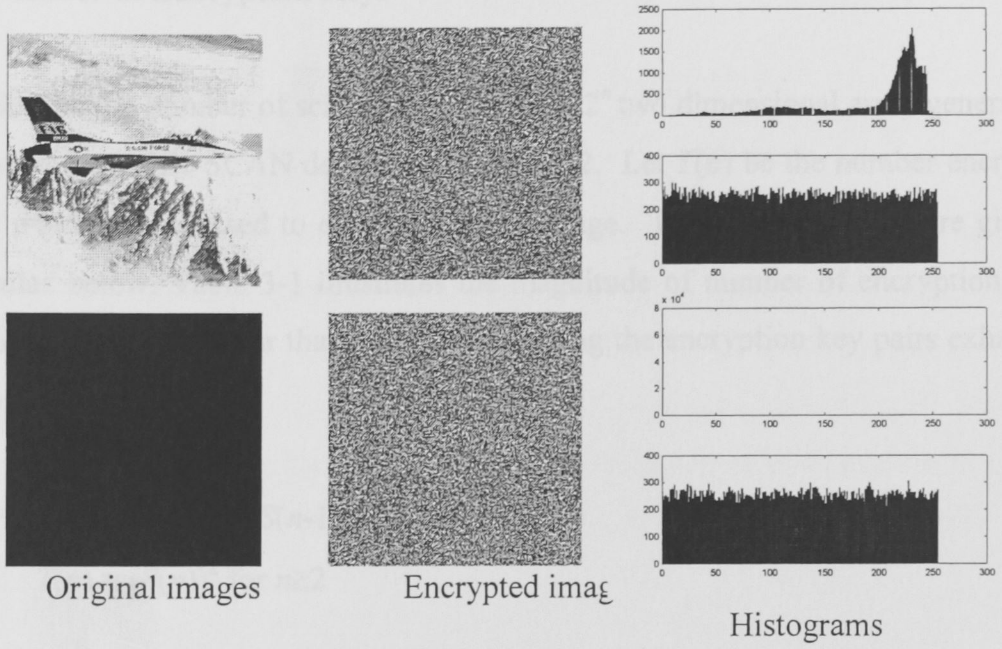


Fig. 3-7. Confusion property of *Encrypt*()

(e) In order to determine the confusion property of *Encrypt*(), the Jet image and a pure black image were encrypted using *Encrypt*() with scan keys *B*2(*x*0 *y*5 *s*6 *r*3), *c*5 and number of iterations 10. The original and encrypted images and their histograms are shown in Fig. 3-7. The encrypted images have uniform histograms regardless of the original images, thus proving the confusion property of *Encrypt*().

(f) In order to determine the diffusion property of *Encrypt*() with respect to images, the jet image was modified by incrementing the value of one randomly chosen pixel by 1. The value of pixel (100, 23) was incremented from 239 to 240. Both the original Jet and modified Jet were encrypted using *Encrypt*() with keys *B*2(*x*0 *y*5 *s*6 *r*3), *c*5 and applying 10 iterations. Fig. 3-8 shows the pixelwise difference of the two encrypted images, which demonstrates that the two encrypted images have no similarities, even though the original images differ by only one pixel, thus showing the diffusion property of *Encrypt*() with respect to images.

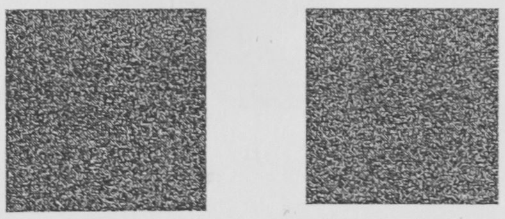


Fig. 3-8. Diffusion property of *Encrypt*() with a difference of one pixel from the original

3.6 Number of Encryption Keys

Let $S(n)$ be the number of scan patterns of a $2^n \times 2^n$ two dimensional array generated by the encryption specific SCAN defined in section 3.2. Let $T(n)$ be the number encryption key pairs which can be used to encrypt a $2^n \times 2^n$ image. Then $S(n)$ and $T(n)$ are given by the formulas below. Table 3-1 illustrates the magnitude of number of encryption key pairs. From the table, it is clear that attack by searching the encryption key pairs exhaustively is impossible.

$$S(2) = 104$$

$$S(n) = 104 + 24(S(n\text{-}1))^4 \text{ for } n \geq 3$$

$$T(n) = (S(n))^2 \text{ for } n \geq 2$$

Table 3-1. Number of encryption keys.

Image size	Number of encryption keys is greater than
64×64	10^{1200}
128×128	10^{4800}
256×256	10^{19000}
512×512	10^{76000}
1024×1024	10^{304000}

The above formulas can be derived as follows. For a $2^n \times 2^n$ $n \geq 2$ image, there are 13 basic scan patterns, shown in Fig. 3-1, each with 8 transformations, resulting in 104 basic scan-transformation patterns. When $n \geq 3$, there are additionally 24 ways, shown in Fig. 3-2 to partition the image into subregions of size $2^{n-1} \times 2^{n-1}$, each having $S(n\text{-}1)$ scan patterns recursively. This results in $S(2) = 104$, $S(n) = 104 + 24(S(n\text{-}1))^4$ $n \geq 3$. A scan key pair has two scan keys, each of which can be any of $S(n)$ scan patterns, resulting in $T(n) = (S(n))^2$.

Chapter 4
The SCAN Architecture

In this section, we present the architecture that has been developed in order to achieve the real-time implementation of the SCAN algorithm in hardware. The general architecture can be applied to several technologies such as ASIC or FPGA. The specific values for several components such as Block RAMs and FIFOs have been selected depending on the resources for the specific FPGA that the design had to be downloaded. In section 4.1, the general architecture is presented, while in the other sections each components of this architecture is analyzed.

4.1 The Architecture Overview

The block diagram of the architecture is shown in figure 4-1. The total design consists of three Address Generators, one 12-bit Counter, two 2048x12 bits FIFOs, two 4096x8 bits block RAMs and two Substitution Units. The Address Generator Unit reads the key from a register file and creates the corresponding address based on the SCAN algorithm. According to figure 3-4 in the case of the encryption the RAM 1 is read sequential and the data are written in RAM 2 based on the address produced by the Address Generator 1 using the key1 and stored in FIFO 1. Then the data are read from RAM 2 again sequential using the Address Counter and the data are written in RAM 1 in the address produced by the Address Generator 3 using the key s0. Meanwhile the Address Generator 2 starts calculating and storing the addresses from key2 in the FIFO 2 and Address Generator 1 storing the addresses from key 1 in the FIFO 1. The data are read once again sequential from RAM 1 and are written in RAM 2 in the address produced from AG3 using the key d0. Finally the data are read from RAM 2 and using the addresses from FIFO 2 are written in RAM 1. This process can be repeated as many times we want. According to the algorithm, five iterations of this process produce a highly encrypted image. The whole process has been summarized in the table 5-1.

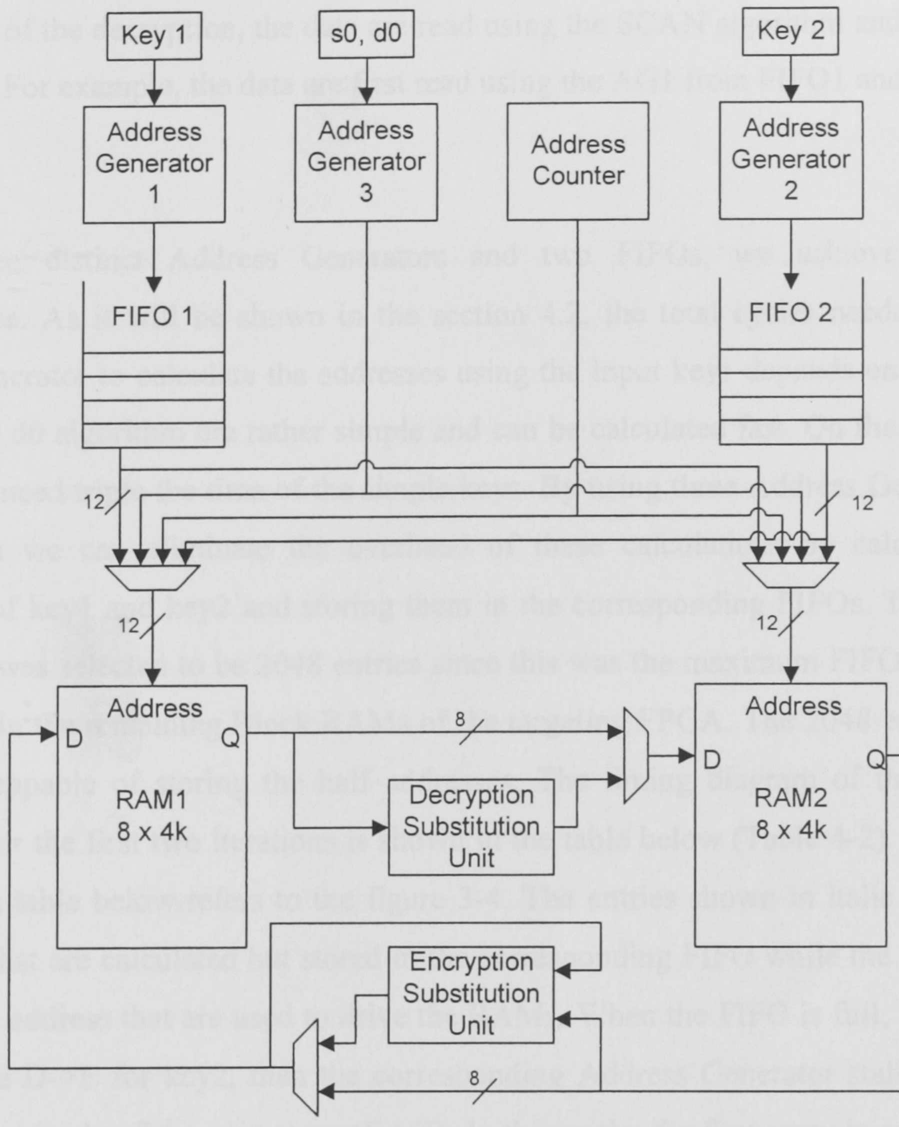


Figure 4-1. The block diagram of the architecture

	RAM 1 Address	RAM 2 Address
Encryption	Address Counter (read)	FIFO 1 (write)
	Address Generator 3 (write)	Address Counter (read)
	Address Counter (read)	Address Generator 3 (write)
	FIFO 2 (write)	Address Counter (read)
Decryption	FIFO 1 (read)	Address Counter (write)
	Address Counter (write)	Address Generator 3 (read)
	Address Generator 3 (read)	Address Counter (write)
	Address Counter (write)	FIFO 2 (read)

Table 4-1. RAM Addresses

In the case of the decryption, the data are read using the SCAN algorithm and are written sequential. For example, the data are first read using the AG1 from FIFO1 and are written sequential.

Using three distinct Address Generators and two FIFOs, we achieve maximum performance. As it will be shown in the section 4.2, the total cycles needed from the address generator to calculate the addresses using the input keys depends on these keys. The s0 and d0 algorithm are rather simple and can be calculated fast. On the other hand, some keys need triple the time of the simple keys. By using three Address Generator and two FIFOs we can eliminate the overhead of these calculations by calculating the addresses of key1 and key2 and storing them in the corresponding FIFOs. The depth of the FIFOs was selected to be 2048 entries since this was the maximum FIFOs that could be created in the remaining Block RAMs of the targeting FPGA. The 2048 entries of the FIFO are capable of storing the half addresses. The timing diagram of these parallel functions for the first two iterations is shown in the table below (Table 4-2). The second row of this table below refers to the figure 3-4. The entries shown in italic refer to the addresses that are calculated but stored in the corresponding FIFO while the bold entries refer to the address that are used to drive the RAMs. When the FIFO is full, for example in the stage D->E for key2, then the corresponding Address Generator stalls and waits until the first cycle of the next stage (F->G). In this cycle, the first entry is read from the FIFO to drive the RAM, while the Address Generator starts again writing to the FIFO. The Address Generators stalls using a gate clock which results to an efficient and low-power function.

	First Iteration				**Second Iteration**			
	A->B	C->D	D->E	F->G	A->B	C->D	D->E	F->G
AG1	**Key1**	-	*Key1*	*Key1*	**Key1**	-	*Key1*	*Key1*
AG2	-	*Key2*	*Key2*	**Key2**	-	*Key2*	*Key2*	**Key2**
AG3	-	**S0**	**D0**	-	-	**S0**	**D0**	-

Table 4-2. Timing table

The block RAMs are 4KBytes, thus they can store one 64x64 pixels image. As it was presented in section 3.6, the encryption security depends on the image size. By using 64x64 pixels images, a good compromise between security and performance has been achieved. Using this block size the number of different keys is 10^{1200}. Using a larger block size, for example 128x128 pixels, then the image would require 16KBytes of internal RAM to be stored, which exceeds the available resources of the FPGA targeting device.

Furthermore, two substitutions units have been added to implement the confusion and diffusion property that was mention in section 3.4. The Encryption Substitution Unit (ESU) has a feedback signal since the current output depends on the previous output of this unit. On the other hand, the Decryption Substitution Unit (DSU) depends only on the inputs. The ESU has been added between the RAM2 and RAM1, since the substitution is performed after the use of key 1 (Figure 3-4). In contrast, the DSU has been added between the RAM1 and RAM2 since the first decryption using the key 2 has been performed as soon as the data have been read from the RAM1 using the AG2.

Finally, we must note that some registers that have been used for the Address Generators and the Address Counter have been omitted from the schematic for simplicity. The use of these registers is to delay the write addresses while the data are passing through the Substitution Units.

4.2 The Address Generator Unit

The Address Generator Unit (AGU) reads the key from a register file and calculates the corresponding address for the 64x64 block RAM. The AGU consists of several units, one unit for the partition pattern and one for each scan pattern. The scan patterns (Figure 3-1) can be divided in two groups. In the first group belong all the scan patterns that can be generated by simple iterative loops. In this group belongs the r, c, d, o, a, s, m, e, y and w scan patterns. In the second group belongs all the other scan patterns (z, b and x), which need recursive loop to calculate the addresses. In the current design, the c scan pattern has been implemented from the first group, since all the other patterns have the same format, and the z, b and x scan patterns from the second group. The partition patterns use the same algorithm as the recursive scan patterns (Z, B, X) as it is shown in figure 3-2.

Each pattern has been encoded by 8 bits (Table 4-3). The first bit shows if it is a valid pattern or not. The next four bits represents one of the 16 different scan and partition patterns, while the last three bits represent the transformation number as it has been presented in section 3.2. The exact representation is shown in the table 4-4.

Key:

Valid	Scan-Partition pattern				Digit		
7	6	5	4	3	2	1	0

Table 4-3. The format of the pattern registers.

Pattern	**Code (6..3)**
B	0000
Z	0001
X	0010
r	0011
c	0100
d	0101
o	0110
a	0111
s	1000
m	1001
e	1010
y	1011
w	1100
z	1101
b	1110
x	1111

Transformation Number	**Digit (2..0)**
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table 4-4. The pattern encoding

A simple register file for the key B0 (B5 (c0 c1 c2 c3) c4 c5 c6) is shown in Table 4-5. The corresponding image partition using these keys is shown in the figure 4-2.

Index	Pattern	Code
0	0	0 0000 000
1	B0	1 0001 000
2	B5	1 0001 101
3	C0	1 0100 000
4	C1	1 0100 001
5	C2	1 0100 010
6	C3	1 0100 011
7	C4	1 0100 100
8	C5	1 0100 101
9	C6	1 0100 110
10	0	0 0000 000

Table 4-5. An example of a register file

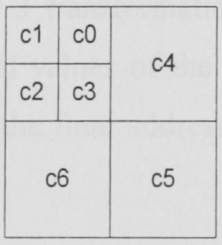


Figure 4-2. The image partition

The Address Generator reads the register file that contains the key and then drive this register to the appropriate pattern as it is shown in the figure below.

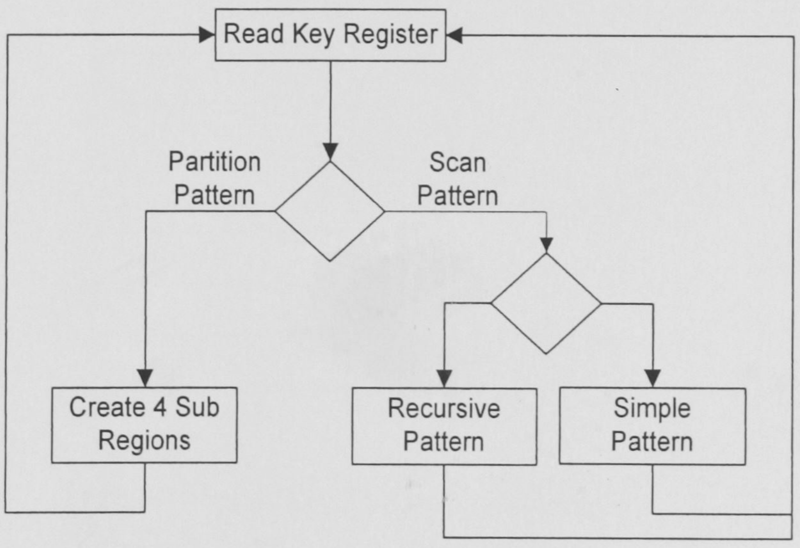


Figure 4-3. The Address Generator FSM

In section 4.2.1 is described the c scan pattern, in section 4.2.2 is described the combination of the z, b and x scan patterns and finally in section 4.2.3 the partition patterns (Figure 3-2) are described.

4.2.1 The Simple Scan Patterns

The Simple Scan Patterns are algorithm that calculates the Scan address using some iterative loops. The software implementation of the C Scan algorithm consists of two nested loops. The boundaries of these loops depend on the transformation number. For example, the outer limits for the 0-3 transformations is 0 to n, while for the 4 to 8 transformations is n to 0. The initial values of the address are forwarded to the rotate procedure, which further calculates the final addresses depending on the transformation number.

The hardware implementation of this algorithm consists of a FSM and a computation module, the rotate module. The FSM controls the number of iterations while the rotate module computes the final address depending on the transformation number. The FSM and the rotate module are shown in the figure 4-4 and 4-5.

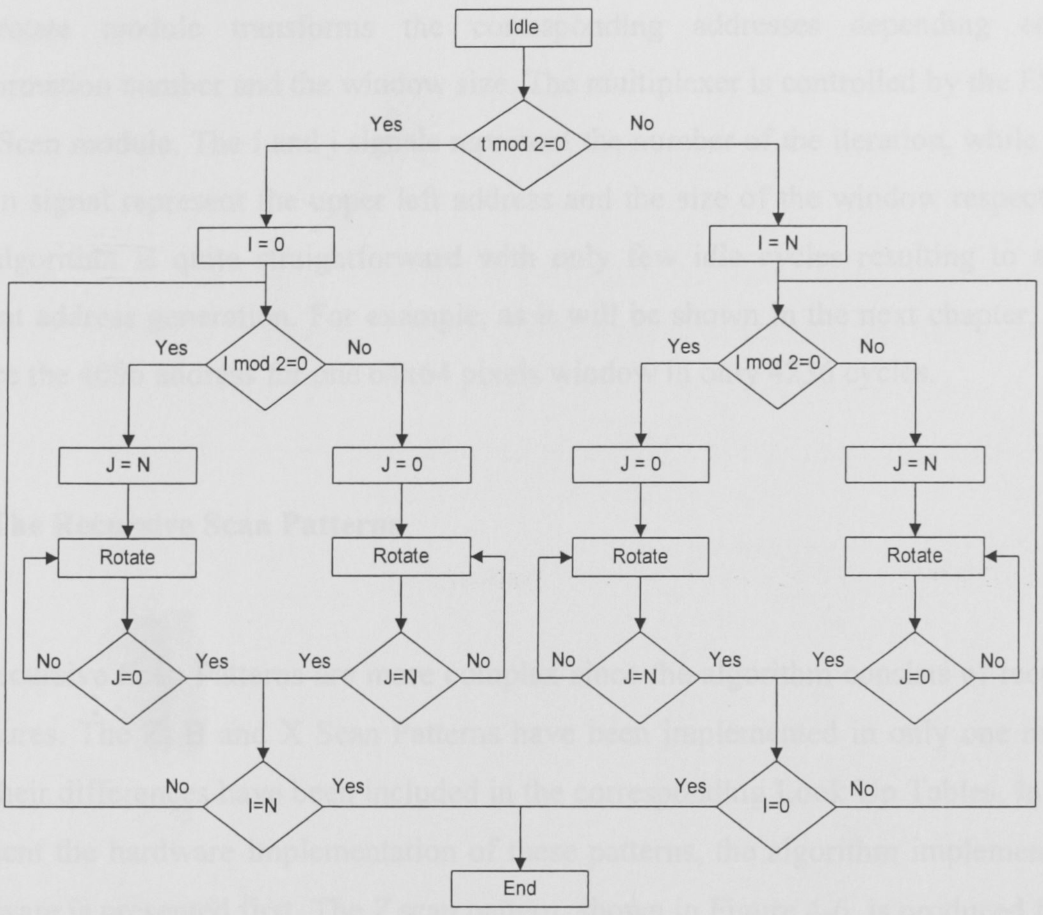


Figure 4-4. The C Scan Pattern FSM

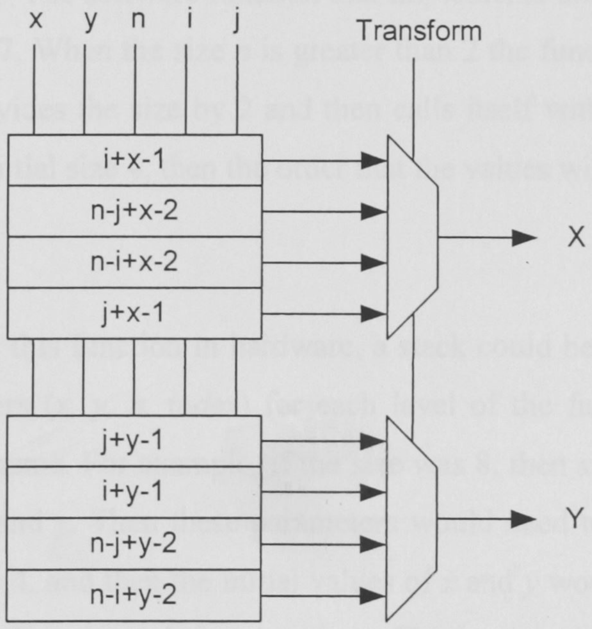


Figure 4-5. The Rotate Module

The rotate module transforms the corresponding addresses depending on the transformation number and the window size. The multiplexer is controlled by the FSM of the C Scan module. The i and j signals represent the number of the iteration, while the x, y and n signal represent the upper left address and the size of the window respectively. This algorithm is quite straightforward with only few idle cycles resulting to a very efficient address generation. For example, as it will be shown in the next chapter, it can produce the 4096 address for one 64x64 pixels window in only 4236 cycles.

4.2.2 The Recursive Scan Patterns

The Recursive Scan Patterns are more complex since the algorithm consists of recursive procedures. The Z, B and X Scan Patterns have been implemented in only one module since their differences have been included in the corresponding Look Up Tables. In order to present the hardware implementation of these patterns, the algorithm implementation in software is presented first. The *Z* scan pattern, shown in Figure 4-6, is produced from a function that calls itself recursively when the size is greater than 2, and if the size is 2 it calculates the address. The software function that implements this pattern is quite trivial, as shown in Figure 4-7. When the size *n* is greater than 2 the function first produces new values for *x* and *y*, divides the size by 2 and then calls itself with the new values. If we call the pattern with initial size 4, then the order that the values will be produced is shown in Table 4-6.

In order to implement this function in hardware, a stack could be used. The stack would preserve the parameters (*x, y, n, index*) for each level of the function, resulting into a rather complicated scheme. For example, if the size was 8, then *sx,* and *sy* would need to be calculated from *x* and *y*. Then these parameters would need to be passed to the next level where the size is 4, and then the initial values of *x* and *y* would have to be retrieved in order to calculate the new values *sx* and *sy*. This approach would need very few storage elements, but it would spend many cycles to recalculate many values.

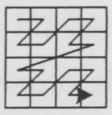


Fig. 4-6. The Z Scan Pattern

In contrast, the hardware scheme that was developed in this work, functions in the reverse order. The parameters of each level are first calculated until the end, and then processing continues to the next level. This way the order that the values are produced is as shown in table 4-7, which results into a simple, fast and area-efficient implementation scheme.

Level1	X_1	Y_1
Level2	X_{11}	Y_{11}
Level2	X_{12}	Y_{12}
Level2	X_{13}	Y_{13}
Level2	X_{14}	Y_{14}
Level1	X_2	Y_2
Level2	X_{21}	Y_{21}
…	…	…

Table 4-6. Software scheme

Level1	X_1	Y_1
Level1	X_2	Y_2
Level1	X_3	Y_3
Level1	X_4	Y_4
Level2	X_{11}	Y_{11}
Level2	X_{12}	Y_{12}
Level2	X_{13}	Y_{13}
…	…	…

Table 4-7. Hardware Scheme

```
void z_sq_arr(int x, int y, int n, char transform)
{
   int I, J, i, sx, sy, order[4];
   if (n > 2)
      for (i = 0; i < 4; i++)
      {
         origin(order[i], x, y, n, sx, sy);
         z_sq_arr(sx, sy, n/2, transform);
      }
   else
      for (i = 0; i < 4; i++)
      {
         origin(order[i], x, y, n, sx, sy);
         I = sx - 1;
         J = sy - 1;
      }
}
```

Fig. 4-7

In the reconfigurable logic scheme five FIFOs, a Control Unit and the Transformation Unit were used. The FIFOs have 16-bit width, since *x* and *y* are 8 bits each, and 4 words depth. The maximum width of an image is 64 (2^6) pixels, which means that five FIFOs of four deep, 16-bit wide (each) had to be used. The FIFO 1, 2, 3, 4 and 5 is used for the 4, 8, 16, 32, 64 sizes of the windows respectively. The FIFO Select Unit is a multiplexer that defines the input and the output of the FIFOs. The Transformation Unit calculates *I* and *J* using the *x, y* and *n* values.

The code in Figure 4-7 is implemented in the following manner: First we load the initial values of *x, y* and *n*. In case that *n* is 2 we calculate the four different addresses without using the FIFOs. In case that *n* is greater than 2 we perform the following steps:

- calculate the 4 new values (*X1, Y1, X2, Y2, …*),
- read the first value from FIFO1,
- divide the size by 2,
- increment the FIFO Select Unit to index to the next FIFO (FIFO2) and

Then, we calculate the new values and if the new size is 2 then we start reading the contents of the FIFO until the FIFO is empty, otherwise we store the four new values to the current FIFO. When the *FIFO empty* signal is asserted we multiply the size by 2 and decrement the *FIFO Select Unit* in order to read again from FIFO1 the next value (*X2, Y2*). This process is continued until the last FIFO (FIFO1) is empty.

The architecture of the iteration Scan Patterns is shown in the figure 4-8 and the detailed finite state machine is shown in figure 4-9. The *Write FIFO* and *Create Address* states are repeated for 4 cycles, since the number of iteration for the algorithm is four. The first time the size is greater than 4 the *Read FIFO* and the *Increment FIFO* states are ignored. The Transform module mainly consists of a Look Up Table (LUT) that can be used to provide the order that an 2x2 block must be scanned depending on the scan pattern.

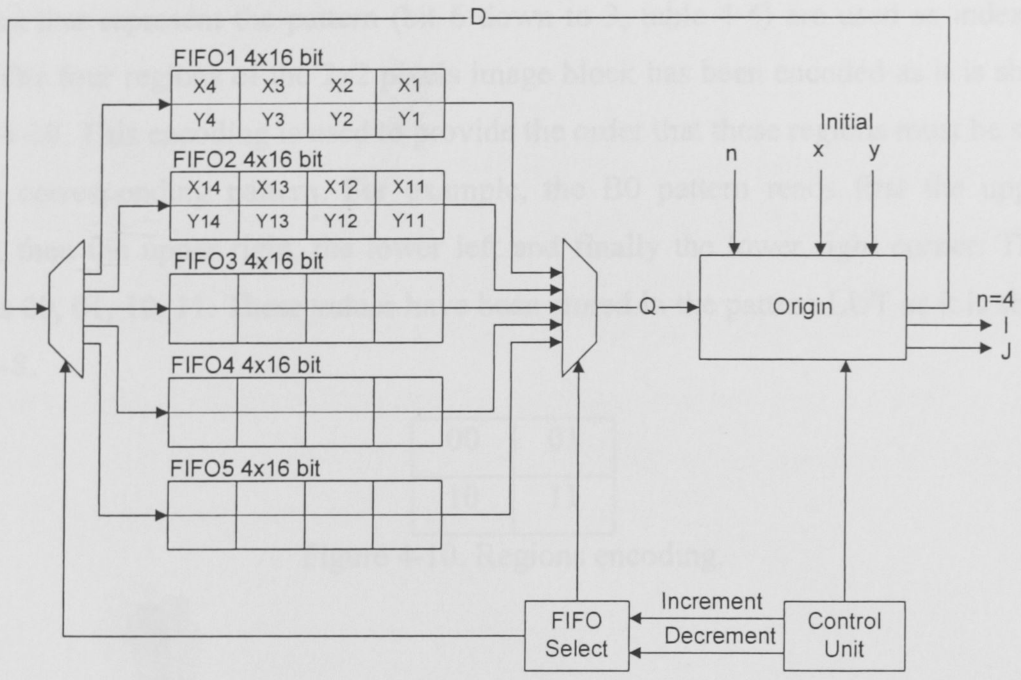


Figure 4-8. The architecture for the Z, B, X algorithms

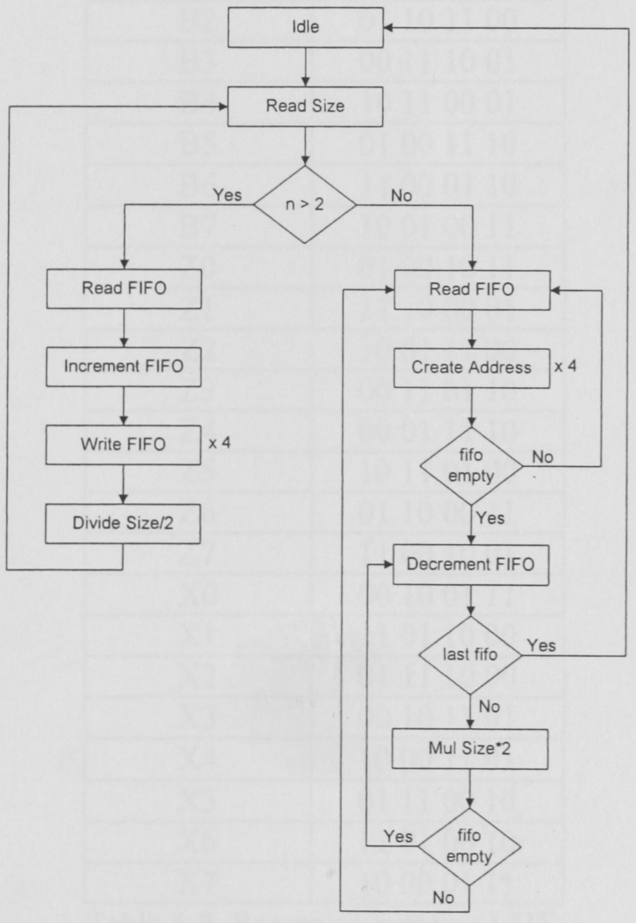


Figure 4-9. The FSM for the Z, X, B algorithms

The bits that represent the pattern (bit 6 down to 3, table 4-4) are used as index to the LUT. The four regions of the 2x2 pixels image block has been encoded as it is shown in figure 4-10. This encoding is used to provide the order that these regions must be scanned by the corresponding pattern. For example, the B0 pattern reads first the upper left corner, then the upper right, the lower left and finally the lower right corner. Thus the order is 00, 01, 10, 11. These values have been stored in the pattern LUT as it is shown in table 4-8.

00	01
10	11

Figure 4-10. Regions encoding.

Pattern	Encoding
B0	00 01 10 11
B1	11 10 01 00
B2	01 10 11 00
B3	00 11 10 01
B4	10 11 00 01
B5	01 00 11 10
B6	11 00 01 10
B7	10 01 00 11
Z0	01 00 10 11
Z1	11 10 00 01
Z2	10 01 11 00
Z3	00 11 01 10
Z4	00 01 11 10
Z5	10 11 01 00
Z6	01 10 00 11
Z7	11 00 10 01
X0	00 10 01 11
X1	11 01 10 00
X2	01 11 10 00
X3	00 10 11 01
X4	10 00 11 01
X5	01 11 00 10
X6	11 01 00 10
X7	10 00 01 11

Table 4-8. Recursive patterns LUT

4.2.3 The Partition Patterns

The Partition Patterns splits the initial image into four sub-images and provides the initial address to the Scan Pattern Units (SPU). The similarity of the Partition Patterns with the *z*, *b* and *x* Scan Patterns is obvious, since the same algorithm is used, as shown in the figure 3-2. The PPU calculates if it is necessary the four initial addresses, and provide this values to SPU in order to calculate the correspond address as it is shown if figure 4-11. Following the Scan Units there is a RAM decoder, which decodes the 2-D location indexes (I, J) into a valid address for the internal RAM.

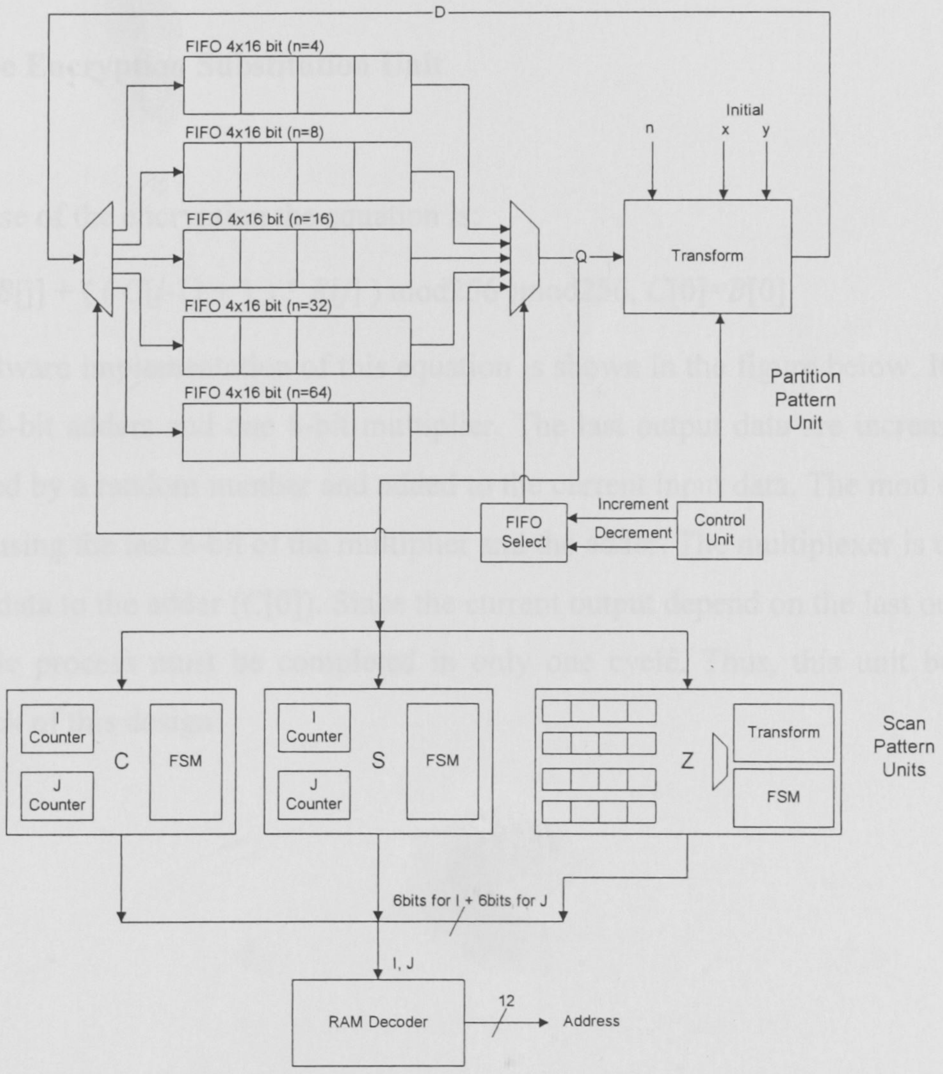


Figure 4-11. The Address Generator

This RAM decoder can be easily changed in order to drive different types of internal (or even external) RAMs, in case we want to encrypt larger blocks of images. In the simple case of 4KX8 bits RAM this is implemented by just concatenating the I and J bits.

4.3 The Substitution Unit

As it was presented in section 3.4 the diffusion and the confusion property is based on a formula that alters the pixel's value using a random number. The Substitution Units are used to change these pixels.

4.3.1 The Encryption Substitution Unit

In the case of the encryption the equation is:

$C[j] = (B[j] + ((C[j-1] + 1) * R[j]) \bmod 256) \bmod 256, C[0]=B[0].$

The hardware implementation of this equation is shown in the figure below. It is consists by two 8-bit adders and one 8-bit multiplier. The last output data are increased by one, multiplied by a random number and added to the current input data. The mod operation is utilized using the last 8-bit of the multiplier and the adder. The multiplexer is used to load the first data to the adder ($C[0]$). Since the current output depend on the last output value, the whole process must be completed in only one cycle. Thus, this unit becomes the bottleneck of this design.

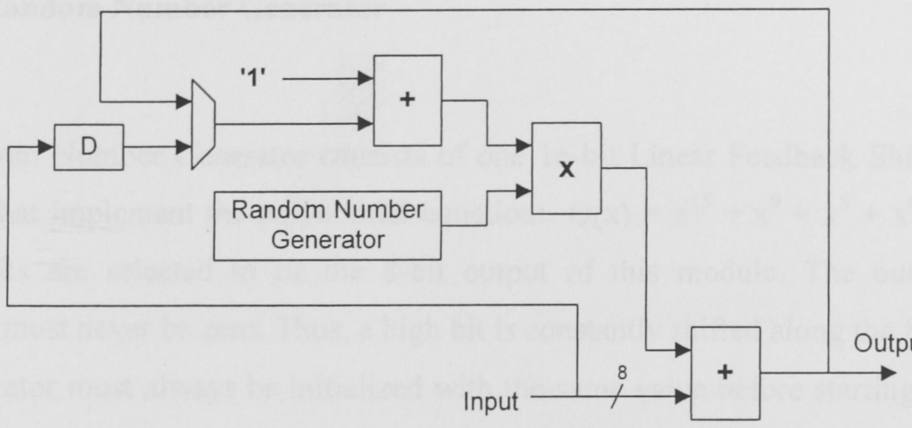


Figure 4-12. The Encryption Substitution Unit.

4.3.2 The Decryption Substitution Unit

In the case of the decryption the equation is:

$E[j] = (F[j] - ((F[j-1] + 1) * R[j]) \bmod 256) \bmod 256$, $E[0] = F[0]$.

This module consists of one 8-bit adder, one 8 bit multiplier and one 8-bit subtractor (Figure 4-13). The input data are delayed by one cycle, increased by one, multiplied by a random number and added to the current input data.

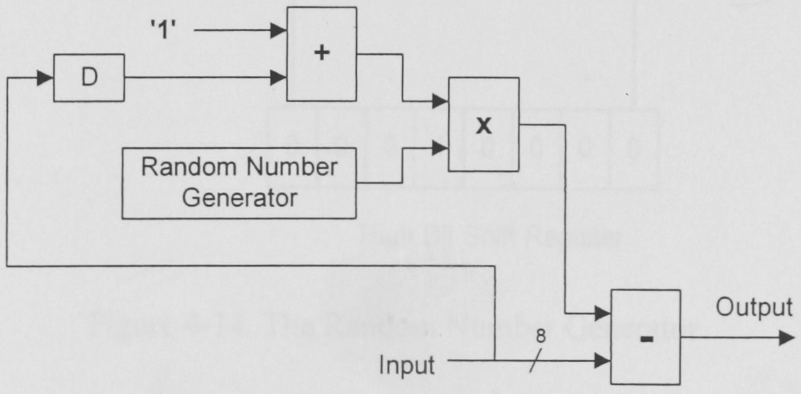


Figure 4-13. The Decryption Substitution Unit

4.4 The Random Number Generator

The Random Number Generator consists of one 16-bit Linear Feedback Shift Registers (LFSRs) that implement the polynomial equation: $Q(x) = x^{15} + x^{9} + x^{5} + x^{4} + 1$. Eight random bits are selected to be the 8-bit output of this module. The output of this Generator must never be zero. Thus, a high bit is constantly shifted along the 8-bit output. The Generator must always be initialized with the same value before starting encrypting or decrypting data. The architecture of the Generator is shown in figure 4-14.

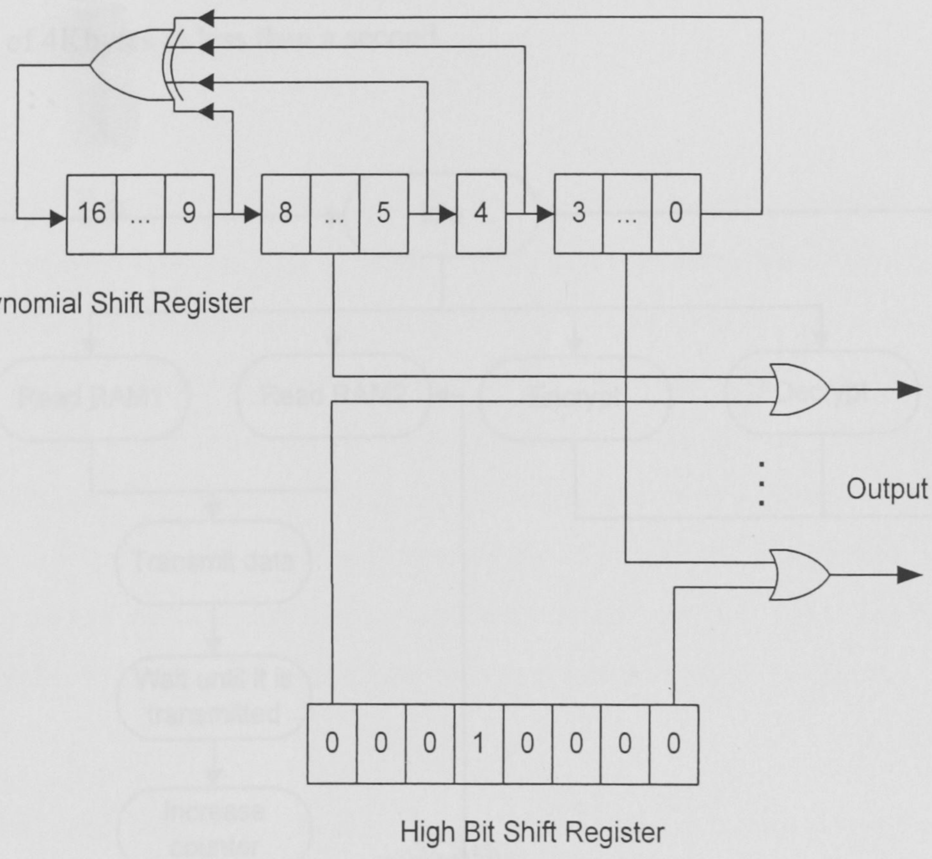


Figure 4-14. The Random Number Generator

4.5 The UART Unit

A UART Unit has been added to the design to debug, verify and use the SCAN encryption application. The UART Unit has been provided by CMOSexod [37] in Verilog. The core consists of a baud-rate generator, a receiver unit and a transmitter unit. The UART Unit has been used to read the contents of the Virtex Block RAMs. A 12-bit counter that increases every time a new byte has been transmitted to the host PC has been used. Using the UART interface, we can initiate an encryption, a decryption and a read back of RAM1 or RAM2. The main FSM of the design is shown in the figure below. The UART is working at 38400bps using a 60MHz clock frequency, thus it can transmit a block RAM of 4Kbytes in less than a second.

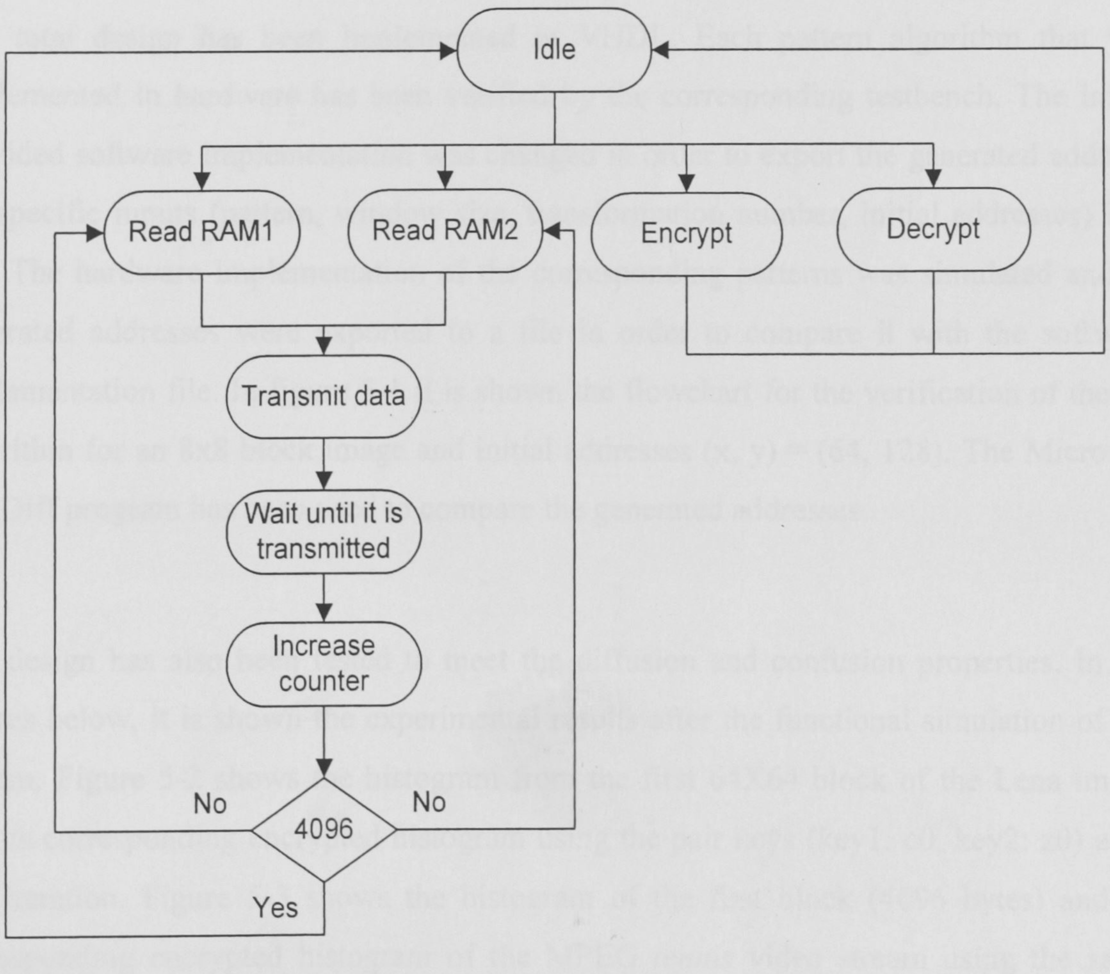


Figure 4-15. The main FSM

Chapter 5
Simulation, Verification and Performance

In this chapter, we present the simulation of the design and compare its properties (confusion and diffusion) with the software approach. Furthermore, we present the performance evaluation of the implementation in order to meet the real-time specifications. Finally, we present the target board that it has been evaluated and the total system integration.

5.1 Simulation and Verification

The total design has been implemented in VHDL. Each pattern algorithm that was implemented in hardware has been verified by the corresponding testbench. The initial provided software implementation was changed in order to export the generated address, for specific inputs (pattern, window size, transformation number, initial addresses) in a file. The hardware implementation of the corresponding patterns was simulated and its generated addresses were exported to a file in order to compare it with the software implementation file. In figure 5-1 it is shown the flowchart for the verification of the Z3 algorithm for an 8x8 block image and initial addresses (x, y) = (64, 128). The Microsoft WinDiff program has been used to compare the generated addresses.

The design has also been tested to meet the diffusion and confusion properties. In the figures below, it is shown the experimental results after the functional simulation of the system. Figure 5-2 shows the histogram from the first 64X64 block of the Lena image and its corresponding encrypted histogram using the pair keys (key1: c0, key2: z0) after one iteration. Figure 5-3 shows the histogram of the first block (4096 bytes) and its corresponding encrypted histogram of the MPEG *tennis* video stream using the same keys. The histogram of the video stream is more uniformly distributed as it is compressed by the MPEG algorithm. Finally, Figure 5-4 shows the histogram of the encrypted data if the plain data is zero. As it is shown, the histogram of the images after the encryption is uniformly spread to the x-axis. The figures below, compared to these of figure 3-7, show that the confusion and diffusion properties have been preserved in the hardware implementation.

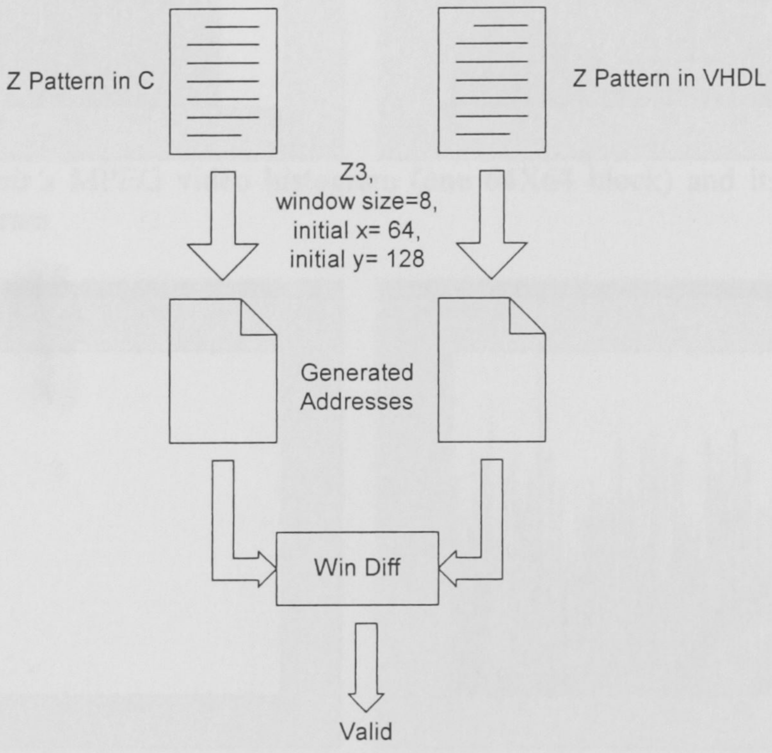


Figure 5-1. The Flowchart of the verification

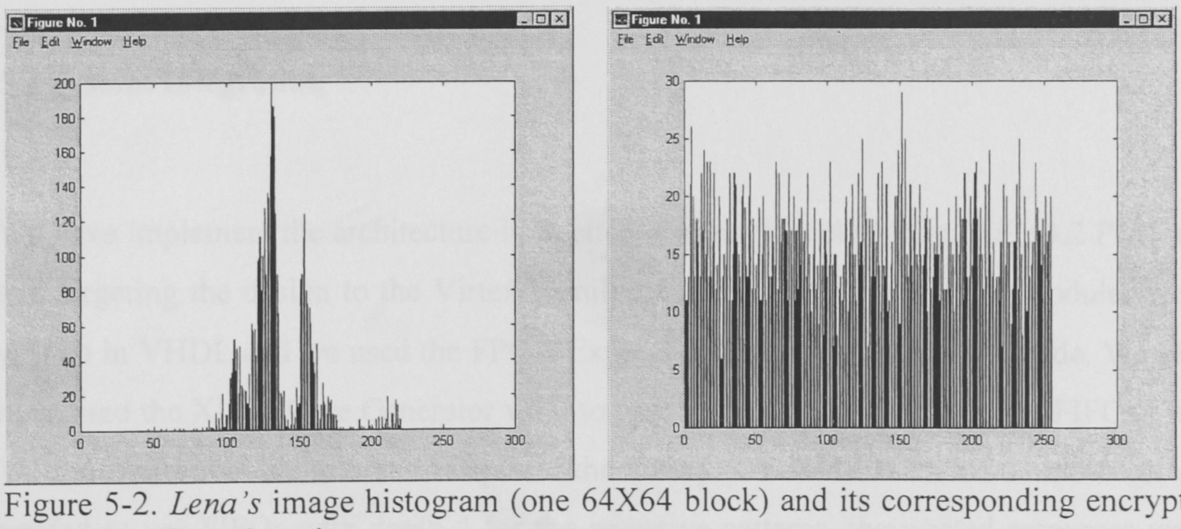


Figure 5-2. *Lena's* image histogram (one 64X64 block) and its corresponding encrypted histogram after 1 iteration.

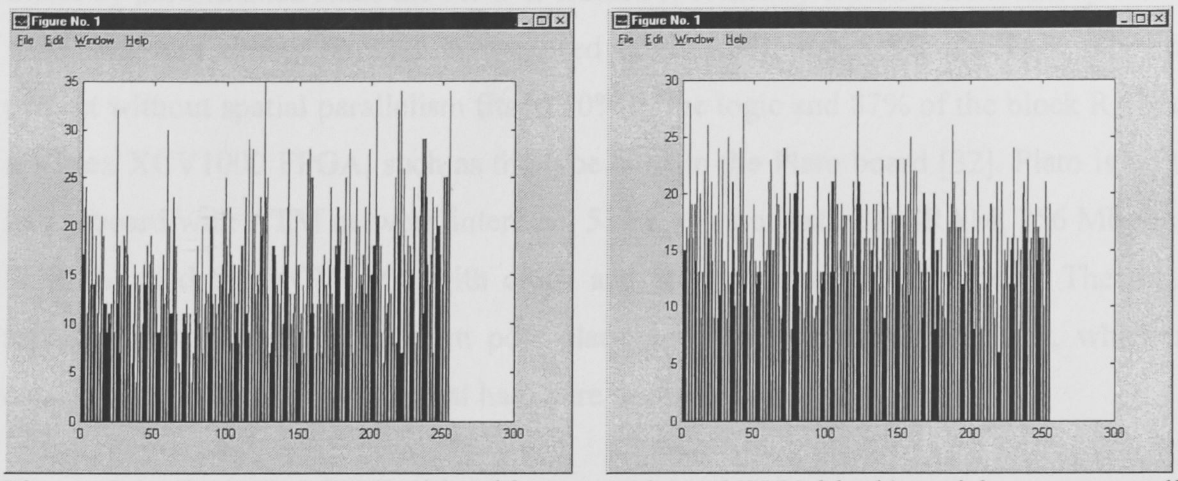


Figure 5-3. *Tennis's* MPEG video histogram (one 64X64 block) and its corresponding encrypted histogram

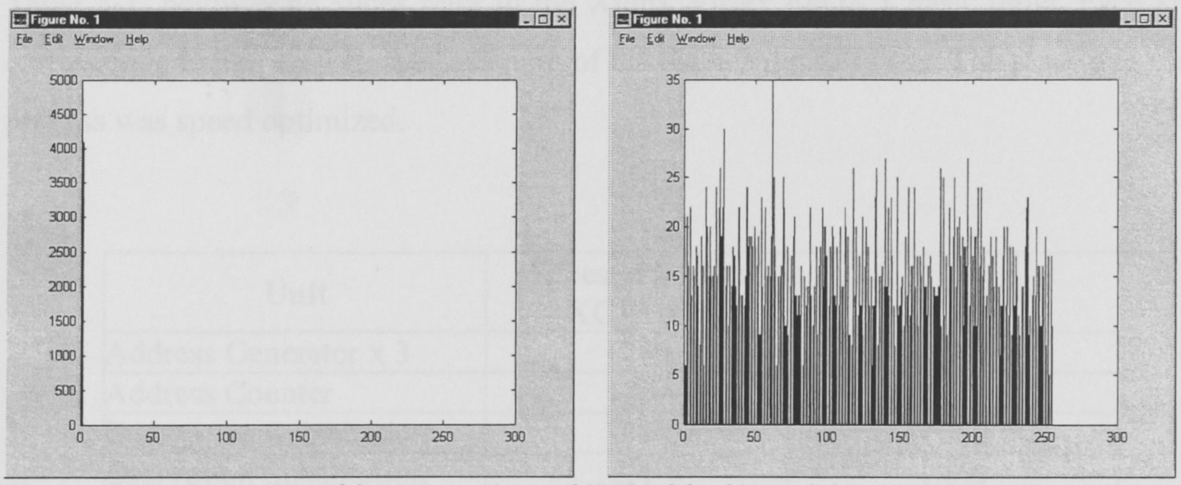


Figure 5-4. Zeros data histogram (one 64X64 block) and its corresponding encrypted histogram

5.2 System Integration

We have implement the architecture in Section 4 using the new Xilinx ISE v4.2 Platform and targeting the design to the Virtex Family of Xilinx Corporation. The modules were written in VHDL and we used the FPGA Express v.3.6.1 to synthesize the code. We also have used the Xilinx Core Generator v.4.2 to produce the RAMs [31] and the FIFOs [30]. We must note that the minimum depth of the library-provided FIFOs is 16. Although we needed to use FIFOs with depth 4 for the recursive patterns, the wasted resources were offset by performance since the FIFO is created specifically for the Virtex family. For the functional and timing simulation, we used the ModelSim 5.5 SE simulator. The total project without spatial parallelism fits in 10% of the logic and 87% of the block RAMs of a Virtex XCV1000 FPGA, such as the type used in the Plato board [32]. Plato is a PCI-based board with ATM network interface, 512 Kbytes of external SRAM, 256 Mbytes of SDRAM, and the XCV-1000 with clock and programming ROM circuits. The results reported in this section are from post place and route timing simulations, which are conservative with respect to actual hardware performance.

The complete baseline design runs at 60 MHz and the area distribution is described in Table 5-1, below. The slices refer to the Xilinx Virtex FPGA's basic units. Each slice corresponds to half a CLB, the basic unit of the older Xilinx FPGAs. The place and route process was speed optimized.

Unit	Slices of Virtex XCV1000	Percentage
Address Generator x 3	1270	58.5%
Address Counter	13	1.1%
Decryption Substitution	28	2.1%
Encryption Substitution	24	1.8%
Random Number Generator	21	1.6%
UART	57	
Control Unit	189	14.8%
Total Design	**1602**	**100%**

Table 5-1. Logic Area distribution

As it is shown in the table, the three Address Generators have been automatically combined by the Place and Route process in order to minimize the number of slices.

The Address Generator includes the Pattern Partition and the Scan Partitions. The exact Slices for each pattern are shown in the table below. The ZBX Scan patterns and the Partition Patterns have the same slices, since the implement the same algorithm.

Unit	**Slices of Virtex XCV1000**	**Percentage**
C Scan Pattern	145	19.3%
ZBX Scan Patterns	302	40.3%
Partition Patterns	303	40.4%
Total Address Generator	**750**	**100%**

Table 5-2. Address Generator Area distribution

The Virtex 1000 FPGA family has 32 block RAMs, 512 bytes each. Table 5-3 shows the distribution of the memory elements of the design. The remaining block RAMs cannot be used to increase the trace FIFOs, since the Xilinx Core Generator creates FIFOs with power of two entries.

Unit	**Block RAMs of Virtex XCV1000**	**Percentage**
Block RAM1	8	25%
Block RAM2	8	25%
Trace FIFO1	6	18.7%
Trace FIFO2	6	18.7%
Total	**28**	**87.4%**

Table 5-3. Memory distribution

The ISE Xilinx platform provides a Power Estimation tool called XPower. Using this tool, we can estimate the power consumption of the design. Unfortunately, this tool can estimate the power consumption only for small modules, usually below 500 slices. In the table below, we present the power estimation of the SCAN patterns and the other modules used in the current design. We must note that these estimations are using 60MHz clock frequency, 2.5V Vcc and 25^{o}C Ambient Temperature.

Unit	Power Consumption
ZBX Patterns	127mW
C Pattern	84mW
Decryption Substitution Unit	68mW
Encryption Substitution Unit	67mW
Random Number Generator	66mW

Table 5-4. Power Consumption

5.3 Performance Evaluation

After the pixel values are placed into internal RAMs, the number of cycles just to read and write an image 1024X1024X8 bits using 5 iterations is 26,214,400 cycles as shown in table 5-2. The five functions refer to the 4 keys that are used (key1, s0, d0, key2) and the function of storing the elements to the internal RAMs for each block. We assume that using a dual port block RAMs we can overlap the export of one block RAM with the input of the next block RAM that is going to be encrypted.

64	Pixels
x 64	Pixels
4,096	
X 5	Functions (4 keys + 1 I/O of Memory)
20,480	
x 5	Iterations
102,400	
x 256	Number of 64X64 Blocks for the 1024x1024 image
26,214,400	

Table 5-2. Total number of operations

The actual number of cycles to encrypt the images is mainly dependent on the encryption key (key1 and key2). If the key is simple, such as *c0*, the number of cycles is very close to the ideal limit, shown above. Thus, every cycle a new address is generated based on the key pattern. In case there are many recursive scan patterns, the total number of cycles increases. Table 5-3, below, contains the total number cycles to encrypt 64X64X8 bit images (i.e. our grain of operation), and the corresponding throughput using different keys (the key column refers to both key1 and key2). These results are without any advanced design methods, such as parallelism of the RAM, etc., which will be described subsequently.

The throughput of the baseline design without the use of the trace FIFOs, for the following keys, using s0 and d0 as intermediate keys, and five iterations is:

Key	Cycles for 64X64 blocks	Sustained Throughput (Mbytes/sec)
1. c0	4,236	2.64
2. B0(B0(c0 c0 c0 c0) c0 c0 c0)	4,508	2.56
3. B0(B0(c0 c0 c0 c0) B0(c0 c0 c0 c0) c0 c0)	4,612	2.53
4. B0(z0 z0 c0 c0)	9,686	1.64
5. z0	14,925	1.20

Table 5-3. Number of cycles for several keys

The recursive algorithms are the most complicated and uses all of the available resources. All of the FIFOs are being used and the increased number of cycles is due to the several stages of the FSM (Figure 4-7) for every four new addresses.

Using the trace FIFOs, we can reduce the number of cycles for the most demanding keys, such as key4 or key 5 in table 5-3, increasing the throughput. The table below shows the total cycles for each iteration for several keys and the corresponding throughput using the trace FIFOs.

Key	Key1	Key2	Cycles (1st)	Cycles (2nd-5th)	Throughput (Mbytes/sec)
1.	C0	C0	16807	16669	2.68
2.	B0(b0(c000) c0 c0 c0)	Z0	20459	20049	2.24
3.	C0	B0(z0 z0 c0 c0)	16807	16669	2.68
4.	C0	Z0	20187	20049	2.24
5.	Z0	B0(z0 z0 c0 c0)	25623	16669	2.43
6.	Z0	Z0	30892	23428	1.82

Table 5-4. Number of cycles for several keys

Of course, it is obvious that if we use the remaining Block RAMs (Table 5-3) for additional image data instead of the trace FIFOs, then we can increase linearly the throughput to 2.40 Mbytes/sec in the worst case (key1 and key2:z0). The aim of the trace FIFOs is to drive multiple Block RAMs as it is will be shown below, in order to overcome the memory overhead that it is added.

If a compression scheme (such as SCAN compression [11]) is used before the encryption of the video, for the worst-case keys (user key1 =z0, user key2 = z0), using five iterations, the results are shown in Table 5-5, below:

Threshold	Claire 352x288			Heart 256x256		
Value	Comp%	Error	Throughput (fps)	Comp%	Error	Throughput (fps)
0	11.05	0.00	12.31	37.33	0.00	17.31
1	52.26	0.63	22.94	45.47	0.02	19.90
2	73.09	1.04	40.69	50.24	0.46	21.81
5	89.60	1.70	105.28	77.32	0.18	47.84
10	93.77	2.35	175.75	90.17	3.20	110.38

Table 5-5. Throughput in fps for compressed video

The threshold value refers to the quality of the compression. When the threshold value is zero then a lossless compression is performed, resulting to a poor compression.

In addition, this throughput is capable of encrypting both MPEG-4 and H.263 video streams. In [33] there is a detailed statistical analysis for various video traces encoded in MPEG-4 and H.263. From the table below (Table 5-6) it is shown that the minimum throughput of 1.82Mbytes/sec (14.56 Mbits/sec) is sufficient for encrypting the most demanding video stream.

	Maximum Bit Rate	
	Jurassic Park	Soccer
MPEG 4		
Low quality	1.6Mbit/s	-
Medium quality	1.7Mbit/s	-
High quality	3.3Mbit/s	3.6Mbit/s
H.263		
16 kbit/s target bit rate	0.092Mbit/s	0.092Mbit/s

64 kbit/s target bit rate	0.36Mbit/s	0.39Mbit/s
256 kbit/s target bit rate	1.4Mbit/s	1.5Mbit/s
VBR	3.4Mbit/s	4.5Mbit/s

Table 5-6. Maximum Bit Rate for compressed video

The throughput of the encryption depends on the number of iterations for each data stream. In the case of uncompressed image, 5 iterations are necessary to provide enough security [11]. On the contrary, in the case of the compressed video (i.e. MPEG) it has been proven [34] that there is a uniform distribution for every byte value and a uniform distribution of the possible digrams (pair of two adjacent numbers). This attribute can be exploited by reducing the number of iterations for the encryption of compressed video in order to achieve higher throughput without any compromise in the security of the encryption. The table below shows the throughput in terms of the number of iterations for the worst-case key (z0 s0 d0 z0).

Number of iterations	**Throughput (Mbytes/sec)**
5	1.82
3	1.97
1	5.53

Table 5-7. Throughput for various numbers of iterations

Spatial parallelism can be easily employed in this architecture because separate 64X64 byte blocks can be processed independently. Using an FPGA family with larger internal RAM such as the Xilinx Virtex II family XC2V8000, which provides 186 Kbytes of internal RAM, we can create 22 pairs, resulting in 58.96 Mbytes/sec in the best case (c*0*) or 40.04 Mbytes/sec in the worst case (z*0*). This throughput is enough for encrypting 40fps of uncompressed 1024X1024X8-bit grayscale images or 13fps of uncompressed 1024X1024X8-bit color images, using a the worst key. We must note that only the block RAMs have to be added, since the same FIFOs can be used to drive these RAMs. The main overhead may be the additional fan out of these FIFOs.

In addition, since the logic part of the design is only 10% of the available resources, we can add more Address Generators with different keys in order to improve the security of the encryption.

Another approach would be to use external SRAM module to encrypt images. Using these modules, we can encrypt larger image blocks, thus increasing the encryption security. For example, the Compaq Pamette PCI board [36] has four 128Kbytes modules that have 8nsec read access time. Using one pair we can encrypt 256x256 pixels images, and if we combine the two pairs we can encrypt 512x512 pixels images, increasing the number of possible keys to 10^{76000}.

5.4 Hardware Verification

As it was mentioned in section 4.5 a UART Unit has been added to the system, to interface with a PC using the serial port. Between the host PC and the Plato board, a MAX232 converter has been used to convert the RS-232 signals to 5V signals for the Plato board. The Xilinx Multilinx Unit has been used to download the bitstream of the design using the Hardware Debugger. The schematic of the system is shown in the figure below.

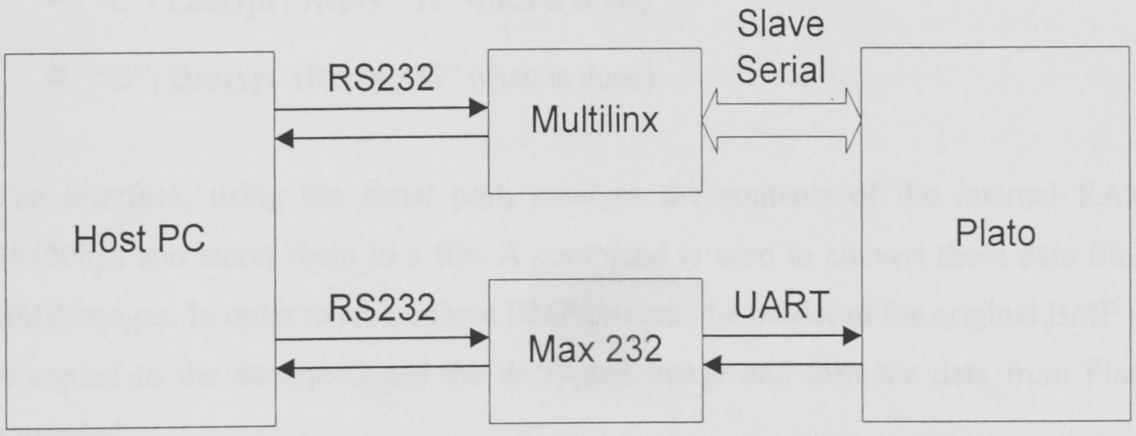


Figure 5-5. The Hardware block diagram

We must note that the serial port has been used only for verification and demonstration purpose and not for real-time image encryption, since the serial port can be a main overhead to the I/O operation. For example, the time to send a block RAM to the host PC at 38400bps is 0.85sec, while the time to encrypt this block using five iterations is only 2msec in the worst case. If a high speed interface is used, such as the USB or the PCI interface that the Plato board support, then a real-time encryption of images can be achieved.

The Xilinx Core Generator program, that creates the Block RAMs, can be configured to load the original image during the configuration process. Thus after the download of the bitstream to the Virtex, the Block RAM contains the original image.

In order to interface with the Plato board using the serial port a graphical user interface has been developed in Visual Basic. This interface uses the serial port of a host PC to send the commands to the design as it was shown in section 4-5. The commands are sent as a plain ASCII character and are shown below:

- "A": Read RAM1 block
- "B": Read RAM2 block
- "C": Encrypt (Reply: "H" when is done)
- "D": Decrypt (Reply: "G" when is done)

The interface, using the serial port, receives the contents of the internal RAMs at 38400bps and stores them in a file. A command is used to convert these data files into BMP images. In order to create these BMP images, the header of the original BMP image is copied to the encrypted and the decrypted image and then the data from Plato are appended.

A snapshot of the graphical interface is shown in the figure below. Some sample images before the encryption, after the encryption and after the decryption is shown in figure 5-7.

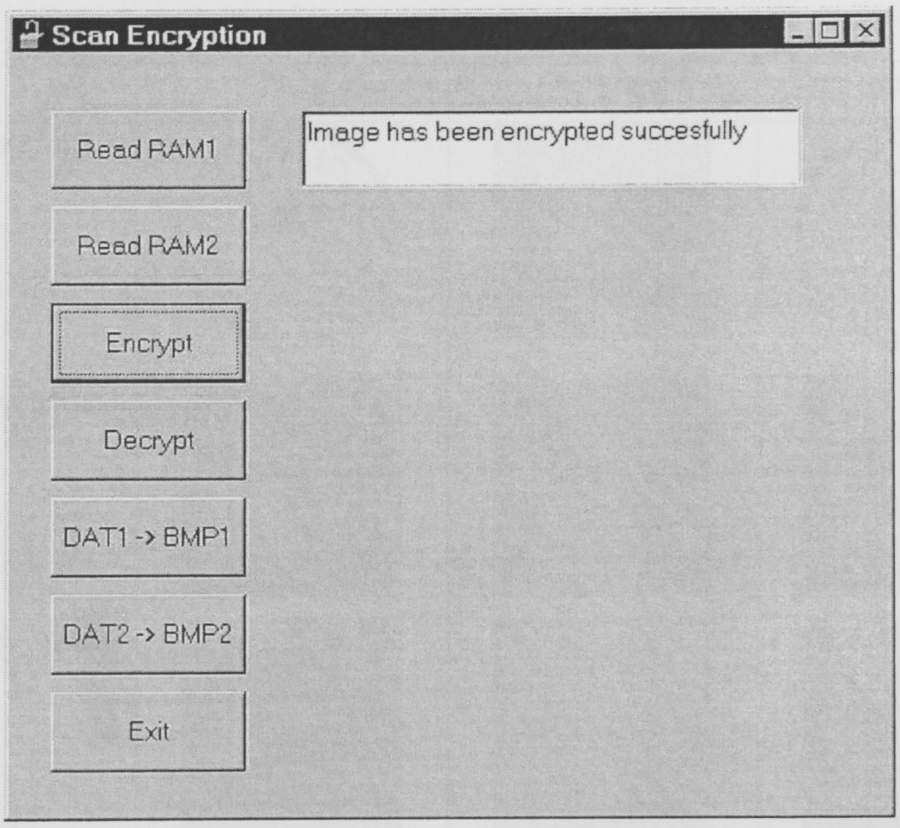


Image has been encrypted succesfully

Figure 5-6. Snapshot from the user interface

The photo of the integrated system is shown in figure 5-8. On the breadboard there is the MAX 232 converter, while the Multilinx unit is shown below the Plato board. The Plato uses a 60MHz clock oscillator module, verifying the frequency that was reported by the Xilinx tools, thus the 1.82Mbytes/sec can be achieved. We must note that the corresponding throughput of the software implementation using a Pentium 4, at 1.5GHz and 256 Mbytes RAM is about 54.6Kbytes/sec, since it takes 0.075sec to encrypt a 64x64-grayscale image using 5 iterations. Thus, the hardware implementation is 33x faster than the software approach using only one block RAM. If we use an other FPGA such as the new Virtex-2, we can achieve 40.04Mbyte/sec as it was mentioned in the previous section, which results to 726x faster than the software approach. The performance comparison in shown in table 5-8. For the last row, we used the Synopsys Design Compiler, using the "umc 0.13u (worst case)" libraries and this speed refers to the critical path of the design, the substitution unit.

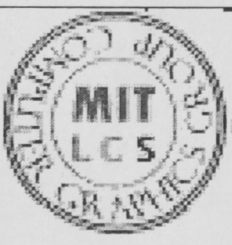
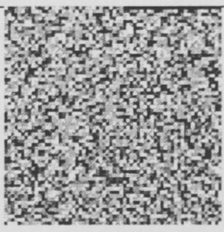
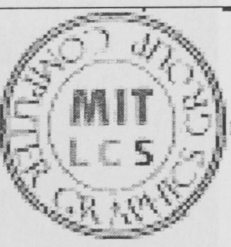
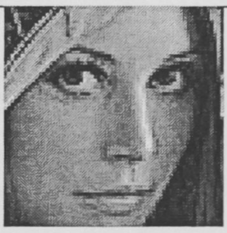
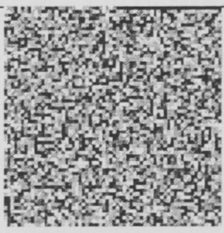
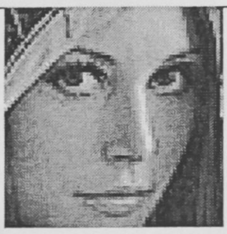
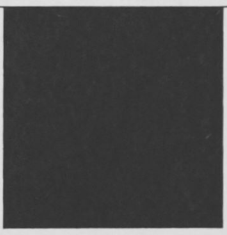
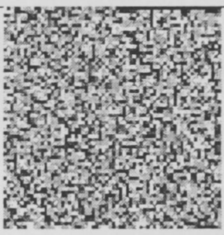
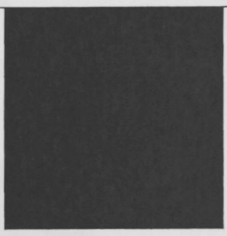
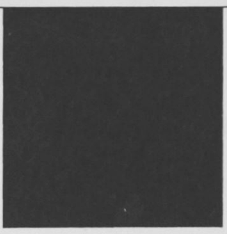
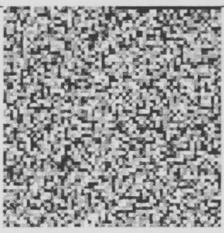
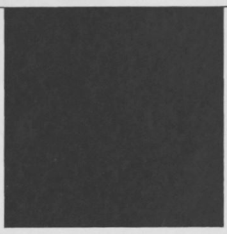
	Plain	Encrypted	Decrypted
MIT 2 iterations			
Lena 5 iterations			
Black 2 iterations			
Black 5 iterations			

Figure 5-7 Samples

Chip	**Throughput**	**Frequency**
Pentium 4	54Kbytes/sec	1.5GHz
FPGA Virtex 1000	1.82 Mbytes/sec (1 block RAM)	60 MHz
FPGA Virtex2 8000	40.04 Mbytes/sec (22 block RAMs)	60 MHz
ASIC 0.13u	12.96 Mbytes/sec (1 block RAM)	400 MHz

Table 5-8. Performance comparison

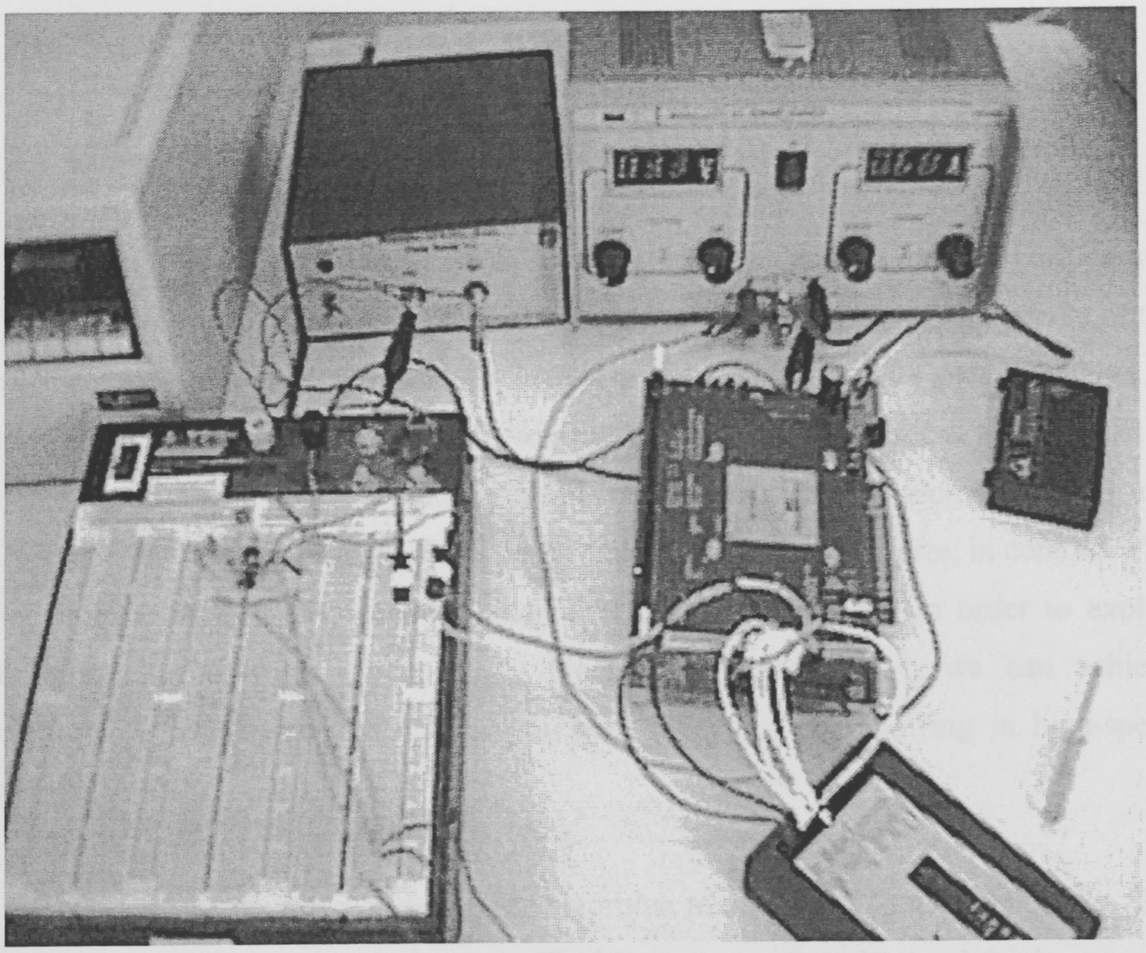


Figure 5-8. The Plato board with the MAX232 converter

Chapter 5
Conclusions and Future Work

In this dissertation, it is presented the SCAN image and video encryption algorithm and an architecture for its efficient implementation in reconfigurable logic. The results of the implementation show that the same architecture can be used to encrypt still images or frames of video, if they are preceded by some compression module. It can also be used as a naïve algorithm in order to encrypt compressed video streams such as MPEG and H.263. The fact that this algorithm is a block cipher with large block size, in contrast with other block ciphers, shows that the internal RAM is a key factor in order to exploit parallelism in hardware. Using many blocks of internal RAM, we can achieve performance that can be used to encrypt large blocks of data flowing in high-speed networks.

In future work, the mapping of the SCAN algorithm to the IRAM (Intelligent RAM) [35] architecture can be explored, because IRAM merges processing and high memory bandwidth into a single chip. In addition, the current design can be improved by developing a hardware architecture for the SCAN compression scheme [11] and integrating to the current design. Furthermore, the SCAN algorithm can be used for information hiding [11]. For example using the SCAN patterns a small image can be encrypted into another image. First, the pixels that are insensitive to the eye must be removed and then these pixels can be used in order to hide the desired information. Finally, the SCAN architecture can be used to encrypt data transmitted over a network. For example, the design can be used to encrypt packets at the Network Interface Card (NIC), before the packets are transmitted to an Ethernet network.

References

[1] D. Kahn, *The Code breakers: The Story of Secret Writing*, New York, Macmillian Publishing Co., 1967

[2] C.E. Shannon, *Communication Theory of Secrecy Systems*, Bell System Technical Journal, v. 28, n. 4, 1949, pp. 657-715

[3] B. Schneier, *Applied Cryptography*, Springer Verlag, New York, 1993.

[4] B. Beckett, *Introduction to Cryptology*, Blackwell Scientific Publications, 1988.

[5] J. Keller, *A hardware-based attack on the A5/1 stream cipher*, In *Proc. APC 2001*, München, Okt. 2001, pp. 155-158.

[6] A. Biryukov, *Real Time Cryptanalysis of A5/1 on a PC*, Fast Software Encryption Workshop 2000, April 2000, New York City.

[7] Electronic Frontier Foundation (EFF), *Cracking DES: Secrets of Encryption Research*, Wiretap Politics and Chip Design, O'Reilly & Associates, Sebastopol 1998.

[8] N. Bourbakis, C. Alexopoulos, Picture data encryption using SCAN patterns, *Pattern Recognition Journal, vol. 25, no6, 1992, pp.576-581*

[9] S. Maniccam and N. Bourbakis, On compression and encryption for digital video, *Int. ISCA Conf. on Parallel and Distributed Systems, pp. 652-657,* Aug. 2000, NV, USA

[10] C. Alexopoulos, SCAN: An efficient data processing-accessing formal methodology, *PhD thesis, Dept. of Computer Engineering and Informatics, University of Patras, 1989.*

[11] S.Maniccam, A Lossless compression, encryption and information hiding methodology for images and video, *PhD thesis, Dept. ECE, Binghamton University, 2000*

[12] J. Scharinger, Fast Encryption of image data using chaotic Kolmogorov Flows, *Journal of Electronics Imaging vol. 17, no2 1998, pp318-325*

[13] L. Chang, Large encrypting of binary images with higher Security, *Pattern Recognition Letters, vol. 19, no 5, 1998, pp461-168*

[14] J. Cheng and J-J-Guo, A new chaotic image encryption algorithm, *Proc. 1998 National Symposium on Telecommunications*, pp.358-362, Dec. 18-19, 1998

[15] X. Li, Image compression and encryption using tree structures, *Pattern Recognition Letters, vol. 18, no11, 1997, pp 1253-1259*

[16] H. Chang, J. Liu, A linear quadtree compression scheme for image encryption, *Signal Processing: Image communication, vol. 10, no 4, 1997, pp279-290*

[17] T. Chuang, J. Lin, New approach to image Encryption, *Journal of Electronic Imaging, April 1998, pp350-6*

[18] T. Chuang, J. Lin, A new multiresolution approach to still image encryption, *Pattern Recognition and Image Analysis, vol. 9, no3, 1999, pp431-439*

[19] T-S Chen et. al. Virtual image encryption based upon vector quantization, IEEE Trans. On Image Processing, 7, 10, 1485-88, 1998

[20] J. C. Yen, J.I Guo, *A New Chaotic Mirror-Like Image Encryption Algorithm and Its VLSI Architecture,* Pattern Recognition and Image Analysis, Vol. 10, No. 2, 2000, pp. 236–247.

[21] J. I. Guo, J. C. Yen, J. Y Lin, The FPGA Realization of a New Image Encryption/Decryption Design,

[22] H. Cheng, X. Li, Partial *Encryption of Compressed Images and Videos*, Presently at Department of Computer Science, University ofWaterloo

[23] Y. K .Lee, L. H. Chen, An Adaptive Image Steganographic Model Based on Minimum-Error LSB Replacement

[24] 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/`.

[25] 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/`.

[26] L. Cappelletti, *An FPGA Implementation of a Chaotic Encryption Algorithm*, Master Thesis, Padova, Decemner 2000

[27] R. Karri, Y. Kim, *Field Programmable Gate Array Implementation of Advanced Encryption Standard Rijndael*,

[28] A. J. Elbirt, C. Paar, *An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher*,

[29] J. Kaps and C. Paar, *Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine*, in 5th Annual Workshop on Selected Areas in Cryptography (SAC '98) (S. Tavares and H. Meijer, eds.), vol. LNCS 1556, (Queen's University, Kingston, Ontario, Canada), Springer-Verlag, August 1998.

[30] Synchronous FIFO v3.0, Xilinx, Core Generator, Product Specification, October 4, 2001.

[31] Single-Port Block Memory for Virtex v4.0, Xilinx, Core Generator, Product Specification, October 4, 2001.

[32] A. Dollas, D. Pnevmatikatos, Architecture and Applications of PLATO, a Reconfigurable Active Network Platform. *IEEE Symposium on Field Programmable Custom Computing Machines,* IEEE Computer Society Press, 2001.

[33] F. Fitzek, M. Reisslein. MPEG-4 and H.263 Video Traces for Network Performance Evaluation. *IEEE Network, Vol. 15, No. 6, pages 40-54, November/December 2001*

[34] L. Qiao, K. Nahrstedt. Comparison of MPEG Encryption Algorithms. *Special Issue on Data Security in Image Comm. and Network, 22(3), 1998*

[35] D. Patterson, T. Anderson, N. Cardwll, A case for Intelligent RAM: IRAM, *IEEE Micro, April 1997*

[36] M. Shand, Compaq Computer Corporation. System Research Center, *PCI Pamette Schematics*, September 2, 1999.

[37] Jeung Joon Lee, *Micro Synthesizable Universal Asynchronous Receiver Transmitter*, CMOSexod, April 2001