

The Design and FPGA-Based Implementation of SERPENT ciphering

Author : AFASSI Mohamed
University of southern Brittany
Lorient, FRANCE

17 September 2023

Abstract

In this study document I will share with you the methods and approaches I used with an FPGA board using VHDL to implement the SERPENT ciphering algorithm. In today's world, keeping data safe is crucial, and we chose the Serpent cipher because it's known for being a strong and efficient way to protect sensitive information.

I will also discuss about the benefits of implementing this ciphering algorithm on Hardware and do some testing and resource bench-markings.

Various design strategies, optimization techniques, and architectural considerations to maximize the efficiency and performance of the Serpent cipher on an FPGA platform will be considered and compared.

The final working design will be implemented on a ZYNQ based architecture FPGA. for instance the ZeDboard zynq 7000 series from digilent will be the starting focus point.

Keywords : FPGA, VHDL, Serpent, ZYNQ, FPGA, ciphering

CONTENTS

1	Introduction :	3
2	Serpent algorithm :	4
2.1	Encryption :	4
2.2	Decryption :	5
3	HARDWARE Implementation :	6
3.1	Zedboard 7000 series :	6
3.2	Initial and Final permutations :	7
3.2.1	Definition :	7
3.2.2	I/Os :	8
3.2.3	Testbench:	8
3.2.4	Validation truth table :	10
3.3	S_boxes :	10
3.3.1	Definition :	10
3.3.2	I/Os :	11
3.3.3	Testbench :	11
3.4	Linear transformation :	13

1 INTRODUCTION :

In our modern era, security has become incredibly important due to the growing threat of hackers and cyberattacks, both for personal and professional reasons. It's crucial to protect sensitive information, especially in areas like the military, medical, and industrial sectors, where unauthorized access and data breaches can have serious consequences.

One of the most effective ways to keep data safe is through encryption. This method relies on the fact that our computers aren't fast enough to decipher encrypted data quickly. This has led to international competitions where experts work to develop new encryption algorithms. Back in the 1990s, the SERPENT encryption algorithm stood out as a finalist in the AES contest, demonstrating its significance in the field of digital security.

The Serpent cipher is a symmetric key block cipher that was one of the finalists in the Advanced Encryption Standard (AES) competition, which aimed to select a secure and efficient encryption algorithm to replace the aging Data Encryption Standard (DES). While the Rijndael algorithm ultimately became the AES standard, Serpent remains a respected and highly secure alternative. This cipher uses a 128-bit block size and offers key sizes of 128, 192, or 256 bits[1]. This cryptographic algorithm utilizes a 32-round substitution-permutation network that operates on a block composed of four 32-bit words. In each round, one of eight 4-bit to 4-bit S-boxes is applied simultaneously 32 times[2]. Serpent was intentionally designed to enable all operations to be carried out in parallel, employing 32-bit slices. This design choice not only maximizes parallelism but also capitalizes on the extensive crypt-analysis research conducted on the Data Encryption Standard (DES)[3].

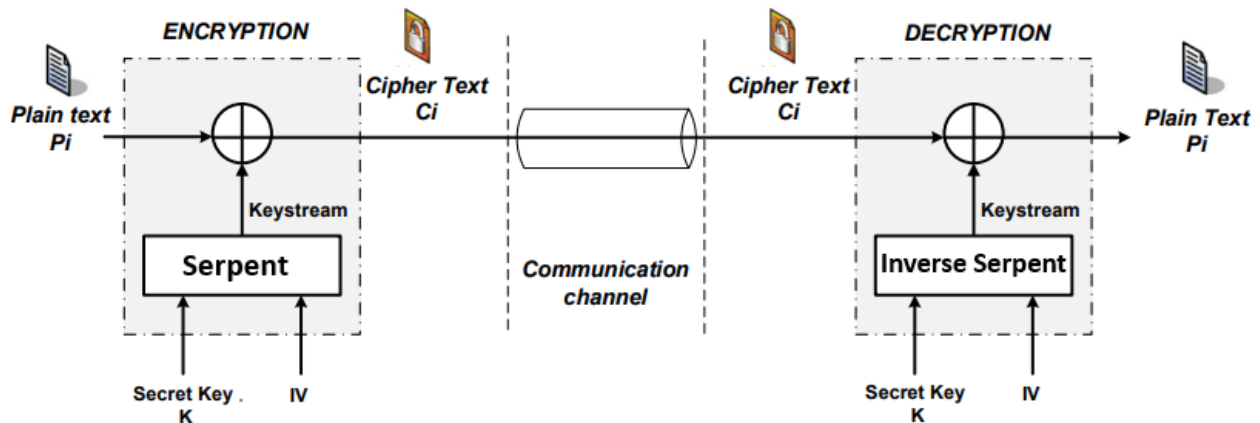


Figure 1: Block diagram of a stream encryption/decryption system.

2 SERPENT ALGORITHM :

2.1. Encryption :

Serpent ciphering algorithm work using the little endian system, which stores least significant bits at the smallest address or the first address in other terms. The algorithm has 32 rounds each round has specific order and set of functions, The whole data path from the plain-text P to the cipher text C can be formally described by the following pseudo-code algorithm :

Algorithm 1: Algorithm for Encryption

Data: Input data P
Result: Encrypted data C
 $\hat{B}_0 \leftarrow InitialPermutation(P);$
for $i \leftarrow 0$ **to** 31 **do**
 if $i < 31$ **then**
 $\hat{B}_{i+1} \leftarrow L(\hat{S}_i(\hat{B}_i \oplus \hat{K}_i));$
 else
 $\hat{B}_{i+1} \leftarrow \hat{S}_i(\hat{B}_i \oplus \hat{K}_i) \oplus \hat{K}_{32};$
 $C \leftarrow FinalPermutation(FP(\hat{B}_{32}));$
return C

This pseudo-code can be translated to the following sequences of equations:

$$\hat{B}_0 = IP(P)$$

$$\hat{B}_{i+1} = R_i(\hat{B}_i)$$

$$C = FP(\hat{B}_{32})$$

Where :

$$R_i(X) = L(\hat{S}_i(X \oplus \hat{K}_i)) \quad i = 0.....30$$

$$R_i(X) = \hat{S}_i(X \oplus \hat{K}_i) \oplus \hat{K}_{32} \quad i = 31$$

These equations are described using the flowchart at the next page:

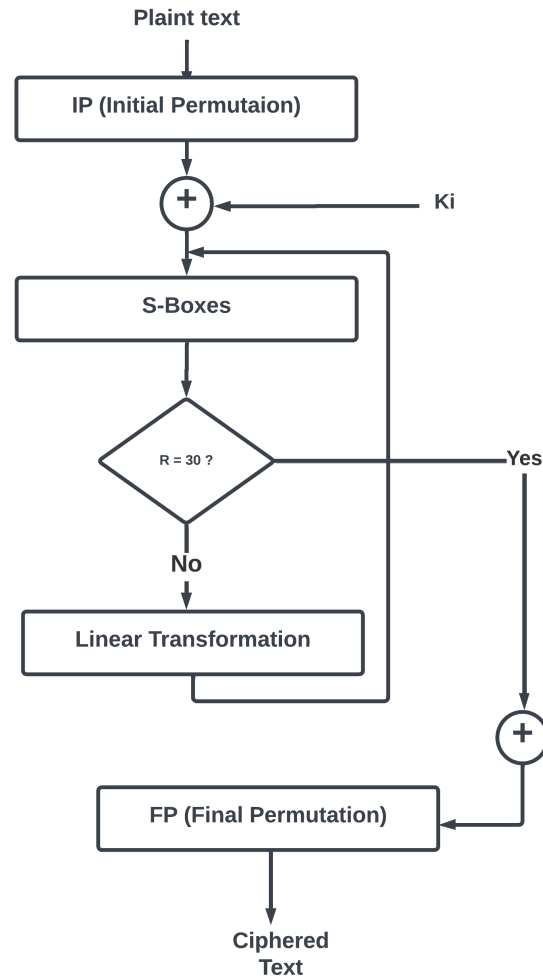


Figure 2: Serpent ciphering flowchart

2.2. Decryption :

Deciphering, or decryption, represents the opposite process of encryption. In algorithms like Serpent, the decryption process is the application of inverse operations in reverse order to transform back cipher-text into its original plain-text. The same encryption key initially used for ciphering is employed for deciphering, although the sub-keys are applied in reverse order. The inverse operations include reversing the S-box substitutions, undoing the permutation, and employing the sub-keys in the reverse order. After all the reverse rounds, we get the recovery of the original plain-text from the cipher-text.

Deciphering flowchart is described below.

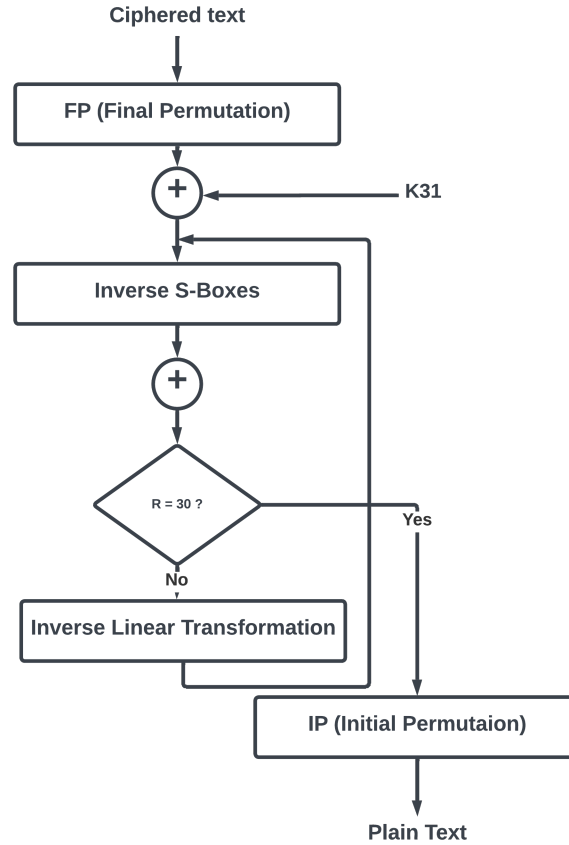


Figure 3: Serpent deciphering flowchart

3 HARDWARE IMPLEMENTATION :

In our hardware implementation, the decryption part will not be used, only the encryption process will be considered using a 128bits user-key, thus we eliminate the 196 and 256bits possibilities.

3.1. Zedboard 7000 series :

The hardware implementation will be implemented on a ZYNQ FPGA, ultimately using the diligent Zedboard 7000 Series Card for this purpose.

Several reasons are taken into consideration to chose this board, the most relevant one being that in the testing phase, we will use python scripts in the PS area to generate random keys and plain-texts and then communicate them with our implementation in the PL Hardware area.

The testing phase will be discussed afterwards in depth and we will also talk about the hardware security side of the implementation.

The synoptic below describes the PL&PS concepts of my implementation.

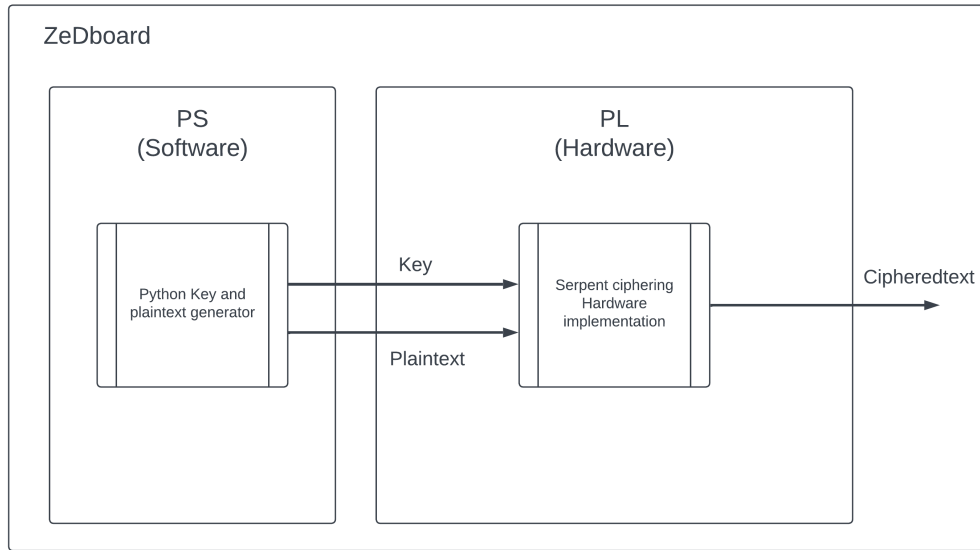


Figure 4: PS and PL description

3.2. Initial and Final permutations :

3.2.1 Definition :

Our ciphering algorithm has two modes of function; Standard mode where it permutes input values with some other values following an array of Numbers, or bit-slice mode where the final and initial permutation to convert data into the bit-slice representation marked X.

Taking in consideration the array below :

0, 32, 64, 96, 1, 33, 65, 97, 2, 34, 66, 98, 3, 35, 67, 99,
 4, 36, 68, 100, 5, 37, 69, 101, 6, 38, 70, 102, 7, 39, 71, 103,
 8, 40, 72, 104, 9, 41, 73, 105, 10, 42, 74, 106, 11, 43, 75, 107,
 12, 44, 76, 108, 13, 45, 77, 109, 14, 46, 78, 110, 15, 47, 79, 111,
 16, 48, 80, 112, 17, 49, 81, 113, 18, 50, 82, 114, 19, 51, 83, 115,
 20, 52, 84, 116, 21, 53, 85, 117, 22, 54, 86, 118, 23, 55, 87, 119,
 24, 56, 88, 120, 25, 57, 89, 121, 26, 58, 90, 122, 27, 59, 91, 123,
 28, 60, 92, 124, 29, 61, 93, 125, 30, 62, 94, 126, 31, 63, 95, 127

Each line has 16 numbers, every number doesn't repeat another time. Considering this, if an input is

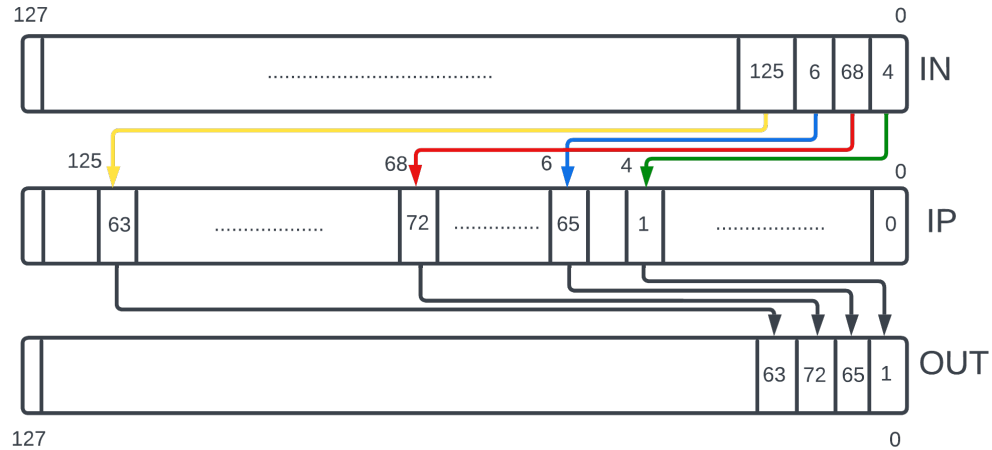


Figure 5: permutation description

IN={4,68,6,125...} , The output data of the permutation algorithms will be as shown below : We will have then an output like this : OUT = {1,65,17,63...}

3.2.2 I/Os :

The twos IPs are the same in term of inputs and outputs besides the permutation array :

Table 1: IP & FP IOs

I/Os	Direction	Type	FP	IP
clk	IN	std_logic	clock signal	clock signal
go	IN	std_logic	activation signal	activation signal
ready_busy	OUT	std_logic_vector(0 to 1)	flag signal	flag signal
plaintext_in	IN	std_logic_vector(0 to 127)	text to final permute	text to initial permute
permutedtext_out	OUT	std_logic_vector(0 to 127)	result	result

3.2.3 Testbench:

For every input byte we will see it's corresponding value at the same address as the value on the output. If we follow the same logic and put ones in the indexes 4, 68, 6, 125 the input of the 128bits will be **0x0a000000000000008000000000000004** and the corresponding output should be **0x40004000000000001400000000000000** while the indexes 1,65,17,63 will be set to ones.

The VHDL simulation below shows the testing of the corresponding hardware implementation :

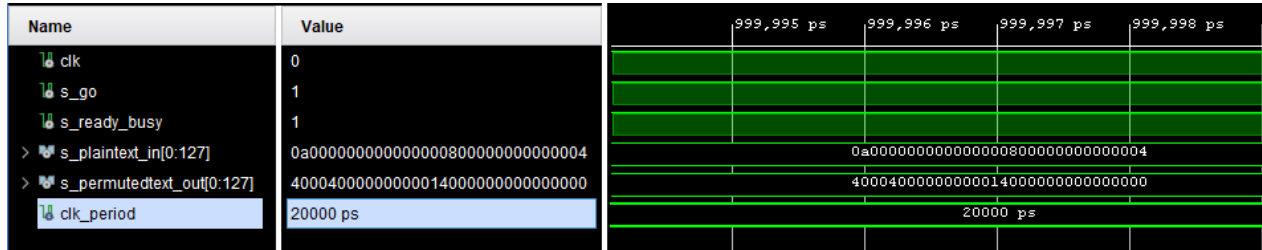


Figure 6: Initial permutation description

The final permutation is the same as the initial permutation but the coding array is the inverse of the first one, thus the final permutation is applying the backwards algorithm of the first one. In the next page, the coding array of the final permutation is represented.

0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60,
64, 68, 72, 76, 80, 84, 88, 92, 96, 100, 104, 108, 112, 116, 120, 124,
1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61,
65, 69, 73, 77, 81, 85, 89, 93, 97, 101, 105, 109, 113, 117, 121, 125,
2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62,
66, 70, 74, 78, 82, 86, 90, 94, 98, 102, 106, 110, 114, 118, 122, 126,
3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63,
67, 71, 75, 79, 83, 87, 91, 95, 99, 103, 107, 111, 115, 119, 123, 127,

For the purpose of testing the final permutation if we take as input the output of the last test, IN={1,65,17,63...} or **0x40004000000000001400000000000000**, and feed it to the last permutation algorithm we should logically get the input of the first permutation which is OUT={4,68,6,125...} or **0x0a000000000000000080000000000004**

The VHDL simulation below shows the testing of the corresponding hardware implementation.

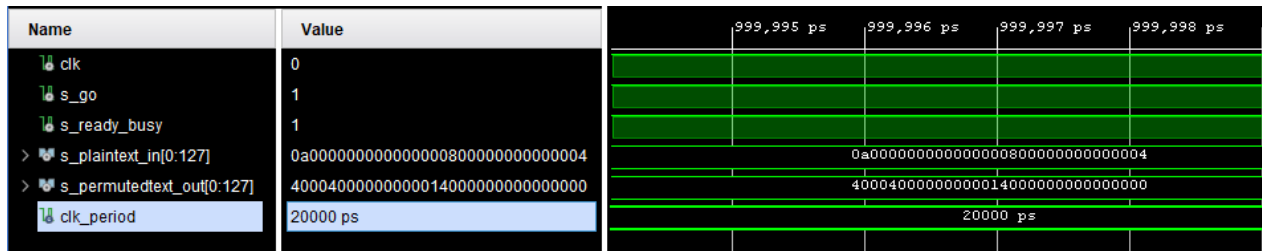


Figure 7: Final permutation description

3.2.4 Validation truth table :

To validate all the functions of both the IP and FP workflow, the two tables below are a guide of different study cases :

Table 2: IP workflow testing

Input	Expected Output
319034efafae596520775cf71a064656	40e9545cc26a4772078e6b14afe38fbe
7c257e98e8eff33158a7c355656167aa	4fdae90165f04e6f6fdc89f7925e9216
c1b721defa57a65469431c2e54ef0d4e	cf65614a979c1dff40c237498d2cbfb0
f163f04cedf6e4fa0c9347fc06f5b9ea	ccc8671c7dd705ebded916237f76fa50

Table 3: FP workflow testing

Input	Expected Output
ccc8671c7dd705ebded916237f76fa50	f163f04cedf6e4fa0c9347fc06f5b9ea
cf65614a979c1dff40c237498d2cbfb0	c1b721defa57a65469431c2e54ef0d4e
4fdae90165f04e6f6fdc89f7925e9216	7c257e98e8eff33158a7c355656167aa
40e9545cc26a4772078e6b14afe38fbe	319034efafae596520775cf71a064656

If you look closely to the two tables, the inputs of IP are the outputs of FP and vice versa, Because these two IPs are complimentary.

3.3. S_boxes :

3.3.1 Definition :

S_boxes is the third algorithm that is applied to the plain-text after initial permutation and key Xoring, Serpent's S_boxes are 4 bits to 4 bits permutations process which involves the use of 8 small S_boxes 4 times, to complete the 32 rounds general algorithm.

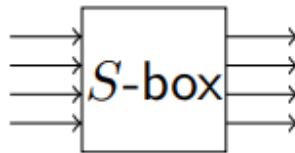


Figure 8: S_box minimal schematic

The predefined ciphering S_boxes are as follow:

- S0 : 3, 8, 15, 1, 10, 6, 5, 11, 14, 13, 4, 2, 7, 0, 9, 12

- S1 : 15, 12, 2, 7, 9, 0, 5, 10, 1, 11, 14, 8, 6, 13, 3, 4
- S2 : 8, 6, 7, 9, 3, 12, 10, 15, 13, 1, 14, 4, 0, 11, 5, 2
- S3 : 0, 15, 11, 8, 12, 9, 6, 3, 13, 1, 2, 4, 10, 7, 5, 14
- S4 : 1, 15, 8, 3, 12, 0, 11, 6, 2, 5, 4, 10, 9, 14, 7, 13
- S5 : 15, 5, 2, 11, 4, 10, 9, 12, 0, 3, 14, 8, 13, 6, 7, 1
- S6 : 7, 2, 12, 5, 8, 4, 6, 11, 14, 9, 1, 15, 13, 3, 10, 0
- S7 : 1, 13, 15, 0, 14, 8, 2, 11, 7, 4, 12, 10, 9, 3, 5, 6

3.3.2 I/Os :

The s_box I/Os are very simple, 4 bits in, 4 bits out. The ins/outs of this block are as follow :

Table 4: S_boxe I/Os

I/Os	Direction	Type	Description
clk	IN	std_logic	clock synchronous signal
go	IN	std_logic	activation synchronous signal
ready_busy	OUT	std_logic	flag asynchronous signal
s_box_in	IN	std_logic_vector(3 downto 0)	4 bits to compute
s_box_out	OUT	std_logic_vector(3 downto 0)	4 bits result
sboxe_num	IN	integer	sboxes number selection

3.3.3 Testbench :

On each round, every byte is applied to one S_box starting from 0 to 7 and restarting from 0 afterwards. If we take for example as input **IN = 0b001011110110 = 0x2F6**, then we'll run 3 rounds because we have 12 bits or 3 bytes. The output we'll wait for here is

OUT = 0b111100101010 = 0xF4A

To break down this result we need to look to our three bytes and compare them each with it's related S_boxes:

- 0x2 is the first byte, it will correspond to the s_box S0, the index 2 has 0xF as value.
- 0xF is the second byte, we will compare it to S1, at the last index we have value 0x4.
- 0x6 is the third byte, it's output should be sourced from the S2 s_box, thus we will get 0xA, which is the value at the index 6.

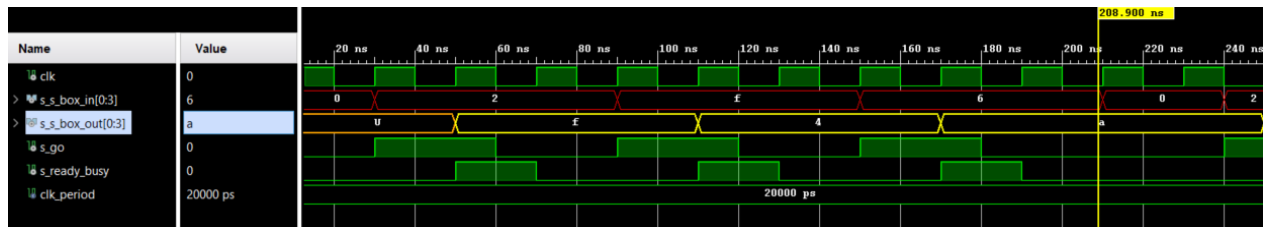


Figure 9: S_box Test-bench

As we can see on the S_box test-bench simulation, our hypothesis input and output values are right, nevertheless we need to wait one clock cycle to receive our output, but I think we can do better by using unclocked processes to speed-up the input processing.

3.4. Linear transformation :

Besides the last round ($i=32$). After S_boxes application we do a linear transformation[4] of our data as it provides a high level of diffusion and confusion in the data[5]. These two properties are essential for the security of a block cipher.

- Diffusion: ensures that a small change in the input data (plain-text or cipher-text) results in a significantly different output. This property helps spread the influence of individual bits across the entire cipher-text, making it difficult for an attacker to discern any patterns or relationships between the input and output.
- Confusion: ensures that the relationship between the key and the cipher-text is complex and difficult to analyze. It makes it challenging for an attacker to deduce information about the key or the plain-text from the cipher-text alone.

The Linear transformation follows the combinatory logic algorithm shown below:

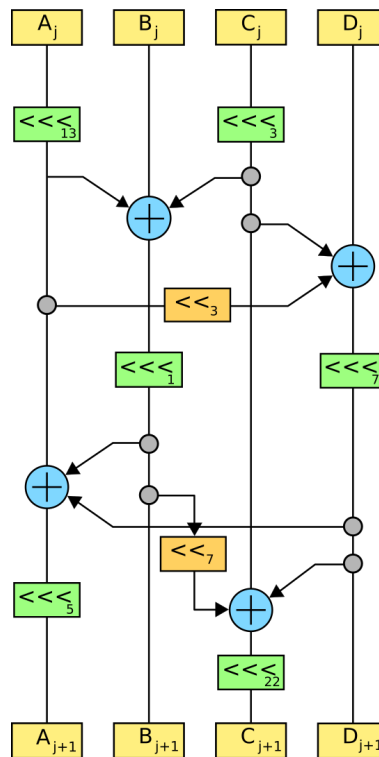


Figure 10: Linear mixing stage

This block takes as input the 128 bits plain-text, trim it to 4 length identical 32bits words. then performs a bunch of operations on it, such as Xorings, rotations and shifting. Every operation has a different amount of shifting or rotating, in order to have the maximum randomness in term of

diffusion and confusion security properties.

Note: "<<<" symbol means rotation and "<<" means shifting.

The implementation of this block on hardware can either be sequential or combinatory, I chose to implement it on combinatory without clocking it, this way I believe it will gain on speed and efficiency.

After implementation, the VHDL simulation can be seen below, to verify it we can use the python code[6] implementation of the serpent algorithm and enter in the same input on the linear transformation block as I did on the test-bench.

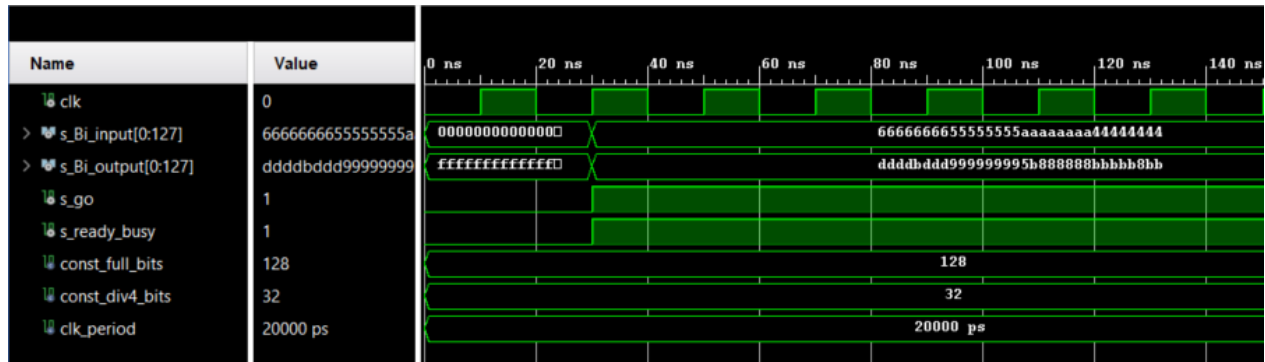


Figure 11: Linear Transformation test-bench

REFERENCES

- [1] Serpent algorithm definition. [Online]. Available: [https://en.wikipedia.org/wiki/Serpent_\(cipher\)](https://en.wikipedia.org/wiki/Serpent_(cipher))
- [2] E. B. Ross Anderson and L. Knudsen. Serpent : a flexible block cipher with maximum assurance. [Online]. Available: <https://www.cl.cam.ac.uk/~rja14/Papers/ventura.pdf>
- [3] W. E. H. Y. M. M. R. L. Fethi Dridi, Safwan El Assad. The design and fpga-based implementation of a stream cipher based on a secure chaotic generator. [Online]. Available: <https://hal.science/hal-03106699/document>
- [4] Linear transformation algorithm. [Online]. Available: [https://en.wikipedia.org/wiki/Serpent_\(cipher\)#Linear_transformation_\(LT\)](https://en.wikipedia.org/wiki/Serpent_(cipher)#Linear_transformation_(LT))
- [5] MKS075. Diffusion and confusion. [Online]. Available: <https://www.geeksforgeeks.org/difference-between-confusion-and-diffusion/>
- [6] F. Stajano. Python implementation of the ciphering serpent algorithm. [Online]. Available: <https://www.cl.cam.ac.uk/~fms27/serpent/serpent-0.py.html>