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

LIST OF FIGURES

LIST OF TABLES

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].

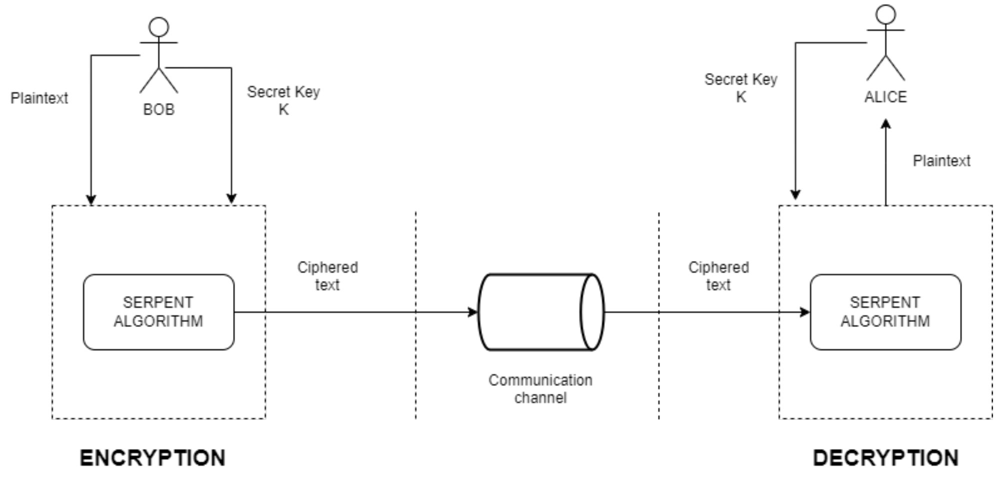


Figure 1: Block diagram of a stream encryption/decryption system.

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:

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

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

$$C = FP(\hat{B}_3 2)$$

Where :

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

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

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

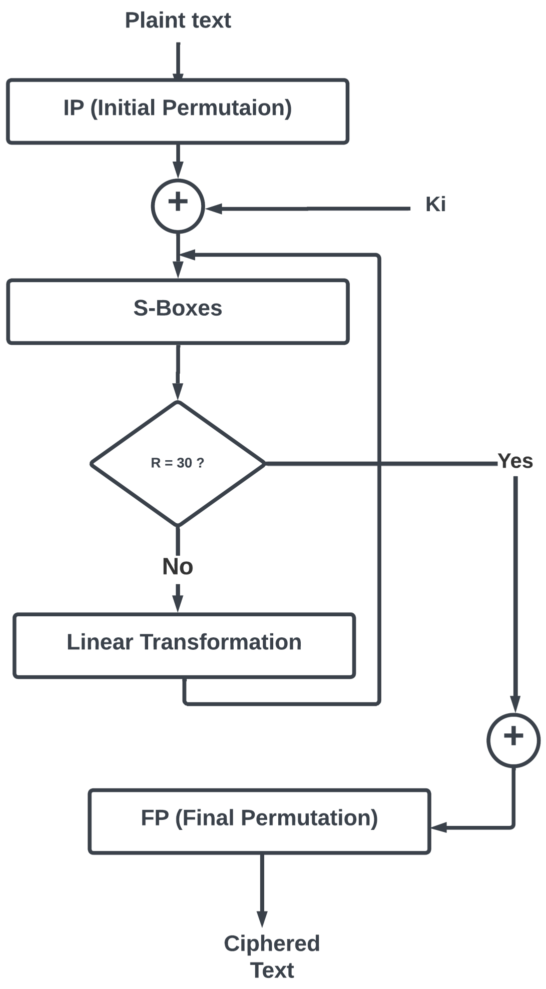


Figure 2: Serpent ciphering flowchart

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.

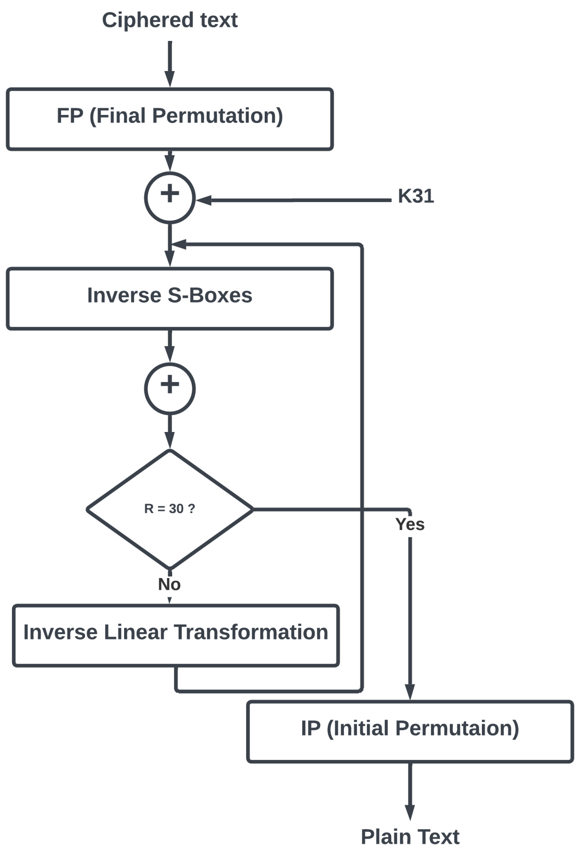


Figure 3: Serpent deciphering flowchart

3 ALGORITHM CONCEPTION :

3.1. Importance :

Conception and Imagination visions are the first essential steps in developing on FPGA. They help to set the path for designs, decide how to use resources, and make sure the system performs well. They also help in making smart choices, improving designs, and making sure the FPGA system meets its goals.
In our case, implementing a complete ciphering algorithm from scratch is a time-consuming process which invokes a serious pre-development clear conceptualization. This benefits us to have good decision takings and to follow the best paths, to not lose the track. Before diving into technical details, we imagine the overall architecture of the system.
Parallelism and Pipe-lining are two different approaches to consider while thinking about algorithm conception, these two techniques play a significant role on the performance and timings of executions. In other perspectives, data flows are also to consider to be sure that our system has the best optimized

data paths, clock domains and timing constraints.
Simplicity is another point to be sure you fell into. A big system is very complex to design in one bloc, that's why generally we devise the big bloc into sub-blocs and then interconnect them one by one. By simplifying the way we design our small blocs, all the work behind is also going to be simplified, the interconnections, tests, simulations...etc

3.2. The concept :

We are developing a very complex system, with different processings. The best approaches to implement these types of projects is to minimize the error marge by dividing the system to several small groups.
In my way, I divided it into 5 blocs, everyone of them has a specific function of the global system :

- Initial permutation : rearranges the bits in the input data to ensure that the data is properly distributed and mixed before the actual encryption process begins.
- Final permutation : rearranges the bits in the input data to ensure that the data is properly distributed and mixed before the data is delivered.
- Linear transformation : mathematical operations that applies linear equations to the data.
- S_boxes : nonlinear substitutions performing on the data
- key_scheduling : generation of the round keys used in the encryption and data

After defining my general blocs, the second part will be the definition of data paths and the data flow between them. If we take the system as an overall black-box, the inputs and outputs will be shown in the diagram below :

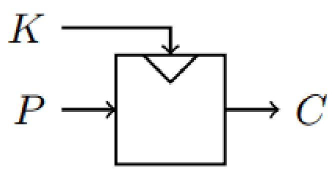


Figure 4: Black-box Ciphering bloc

K : Encryption key – P : Plain-text – C : ciphered text

Note : For those basic three inputs/output we can also add other I/Os for process and flag control

In order to regroup all the blocs, I went with the idea of implementing two state machines, The first one doing just the first and initial permutations, then giving the control to another sub-state machine that does the generation of sub-keys, s_boxe substitutions and Linear transformation. The diagram below shows us the concept to follow for building the ciphering system :

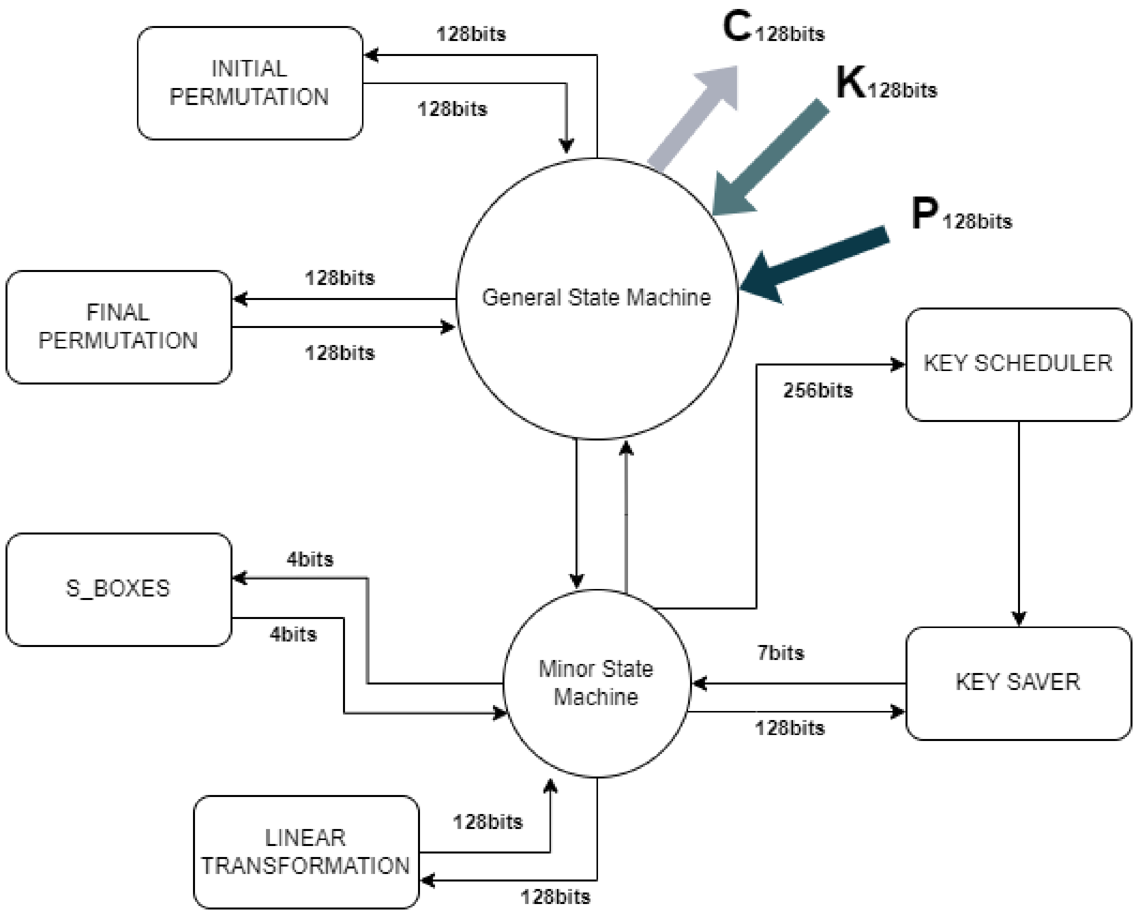


Figure 5: System Conception

K : Encryption key – P : Plain-text – C : ciphered text

4 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.

4.1. Zedboard 7000 series :

The hardware implementation will be implemented on a ZYNQ FPGA, ultimately using the digilent 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.

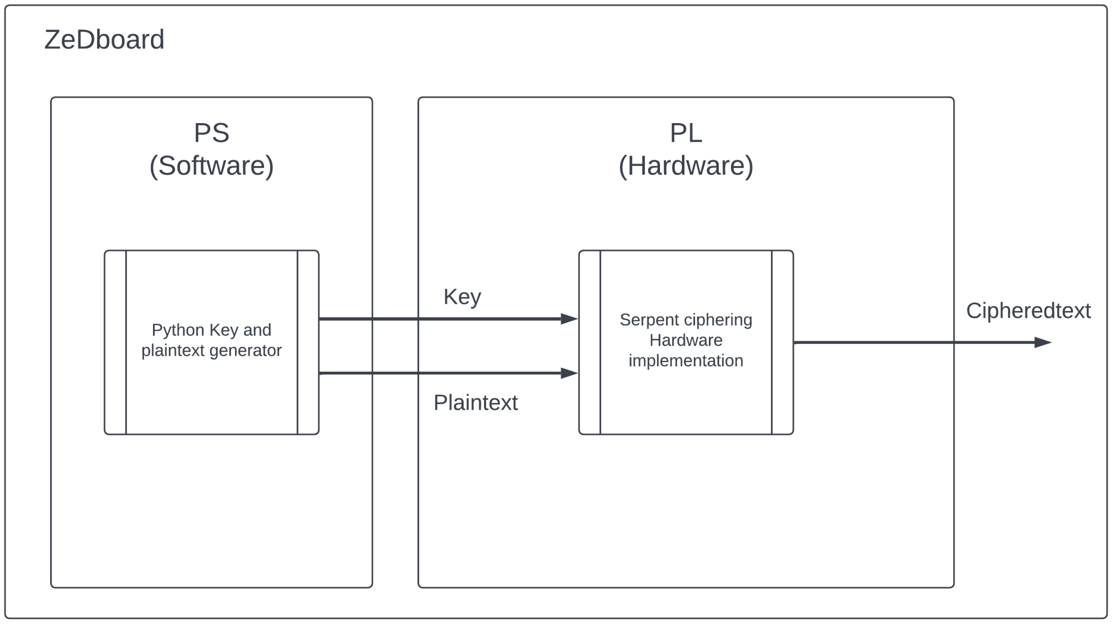

ZeDboard
PS
(Software)
Python Key and plaintext generator
PL
(Hardware)
Key
Plaintext
Serpent ciphering Hardware implementation
Cipheredtext

Figure 6: PS and PL description

4.2. Initial and Final permutations :

4.2.1 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 IN={4,68,6,125...} , The output data of the permutation algorithms will be as shown below : We

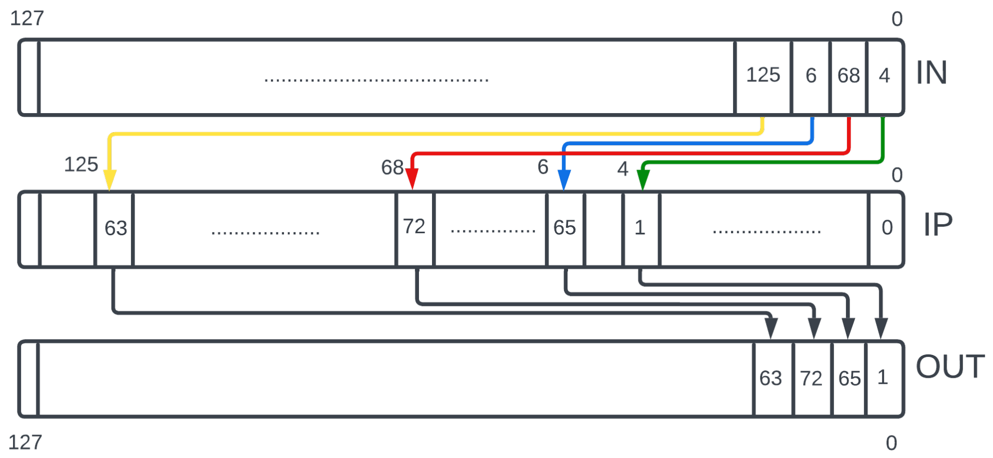


Figure 7: permutation description

will have then an output like this : OUT ={1,65,17,63...}

4.2.2 I/Os :

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

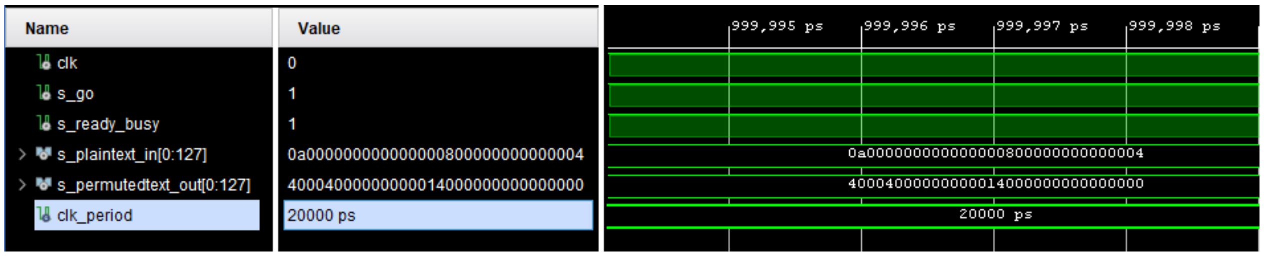
Table 1: IP & FP IOs

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

4.2.3 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 **0x0a000000000000000800000000000004** and the corresponding output should be **0x40004000000000014000000000000000** while the indexes 1,65,17,63 will be set to ones.

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



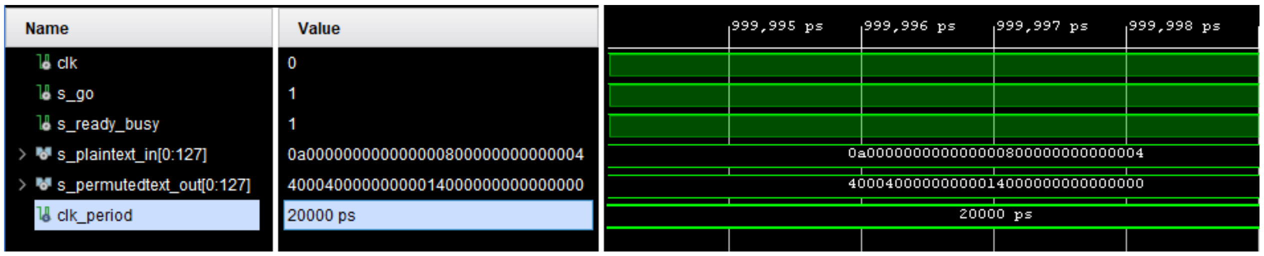
```
Name                          Value
clk                           0
s_go                          1
s_ready_busy                  1
s_plaintext_in[0:127]         0a000000000000000800000000000004
s_permutedtext_out[0:127]     40004000000000014000000000000000
clk_period                    20000 ps
```

Figure 8: 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 **0x0a000000000000000800000000000004**
The VHDL simulation below shows the testing of the corresponding hardware implementation.



Name	Value
clk	0
s_go	1
s_ready_busy	1
s_plaintext_in[0:127]	0a000000000000000800000000000004
s_permutedtext_out[0:127]	40004000000000001400000000000000
clk_period	20000 ps

Figure 9: Final permutation description

4.2.4 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.

4.3. S_boxes :

4.3.1 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.

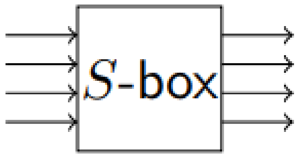


Figure 10: S_box minimal schematic

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

4.3.2 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

4.3.3 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.

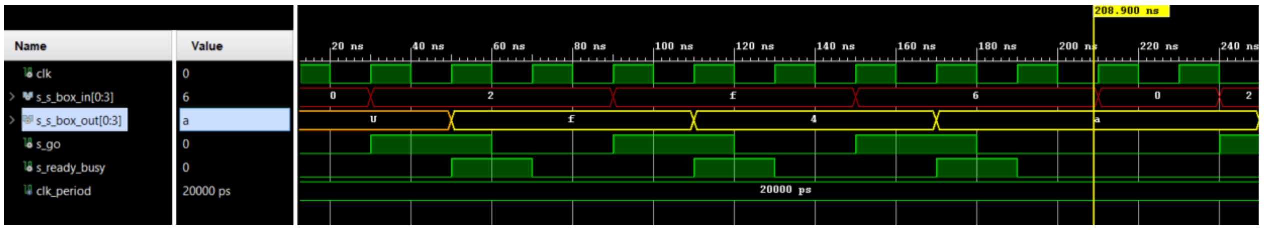


Figure 11: S_box Test-bench

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.

4.3.4 Validation truth table :

In order to corroborate the S_box IP, the truth table below will help us by showing different results and inputs :

Table 5: S_boxe workflow testing

Input Stream	S_boxe number	Expected Output
1101	1	1000
1111	2	0010
0101	3	1001
1100	4	0011
0000	5	1111
1111	6	0000
1010	7	1000
0100	0	1111
0010	1	1001
0001	2	1101

4.4. Linear transformation :

4.4.1 Definition :

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:

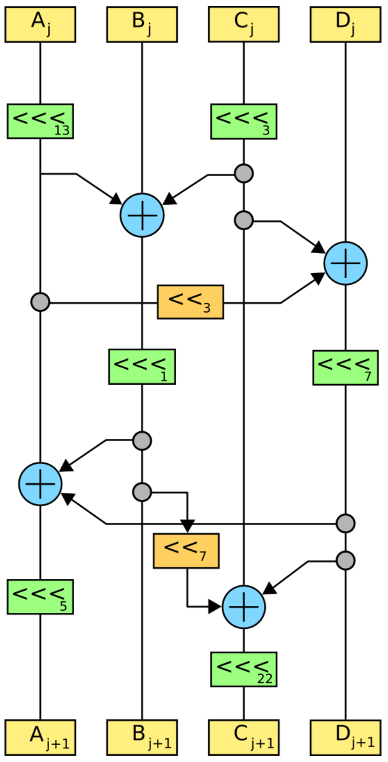


Figure 12: Linear mixing stage

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 shiftings. 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.

4.4.2 I/Os :

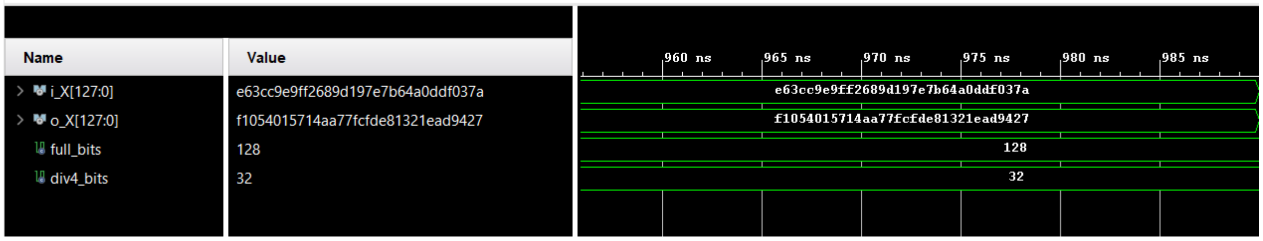
The implementation of this block on hardware can either be sequential or combinatory, I chose to implement it on combinatory without clocking it, this way I believe it will gain on speed and efficiency.

Table 6: Linear transformation I/Os

I/Os	Direction	Type	Description
Bi_input	IN	std_logic_vector(127 downto 0)	Text to apply linear transformation to
Bi_output	OUT	std_logic_vector(127 downto 0)	result text of linear transformation

4.4.3 Test-bench :

After implementation, the VHDL simulation can be seen below, to verify it we can use the python code[6] implementation of the serpent algorithm. An input text is taken, then it is computed by applying numerous mathematical formulas following the the combinatory logic algorithm shown previously.
In the image below we can see the input and output values of a specific text :



Name	Value
i_X[127:0]	e63cc9e9ff2689d197e7b64a0ddf037a
o_X[127:0]	f1054015714aa77fcfde81321ead9427
full_bits	128
div4_bits	32

Figure 13: Linear Transformation test-bench

4.4.4 Validation truth table :

To make sure the Linear transformation block is working well, the truth table below is showing us some examples of inputs and corresponding expected outputs:

Table 7: Linear transformation workflow testing

Input Stream	Expected Output
41D251D9F23B6C44238E7B32F1BCB5DB	6e1412fb366f79a01b6a023881c87647
41D251D9F23B6C44238E7B3241D251D9	7daf145b9e6b97611b6a0238f6a8a245
77EB8122FE75CC7F3503EB97C105C4DA	e666389262021d72146e1dea1ebdbd19
D92BC8ED9ABAFA3D60F1695C1C2FC155	b07ecb3921f010dc9afa8a0a02e6fa2a
4ECA43BFAB4031490669ED2BFF48F533	30ae1c9d26f2ff31fabddcd60fb3fe70

4.5. Minor state machine :

4.5.1 Definition :

The Minor state machine is the last step before the final one. The state machines are a very powerful logical tool for controlling complex processes that require sequential and conditional actions. In my case, the Machine state will help me synchronize the different blocs already defined before by following the general ciphering algorithm.

This state machine is doing the core of the calculations; generating sub-keys, s_boxe permutations, key xoring then linear transformation:

```
Algorithm 1 : Algorithme Serpent
$\hat{B}_0 \leftarrow IP(P)$ ; Initial Permutation
for $i \leftarrow 0$ to $31$ do
    if $i < 31$ then
        $\hat{B}_{i+1} := L(\hat{S}_i(\hat{B}_i \oplus \hat{K}_i))$ ;
    else
        $\hat{B}_{i+1} := \hat{S}_i(\hat{B}_i \oplus \hat{K}_i) \oplus \hat{K}_{32}$ ;
$C \leftarrow FP(\hat{B}_{32})$ ; Final Permutation
```

Figure 14: Algorithm division

In the figure above we can see that the red part is corresponding to the minor state machine.

4.5.2 I/Os :

The minor state machine like all other blocks is controlled by another top module, thus it needs to have inputs, outputs and signal controllers :

Table 8: Minor state machine I/Os :

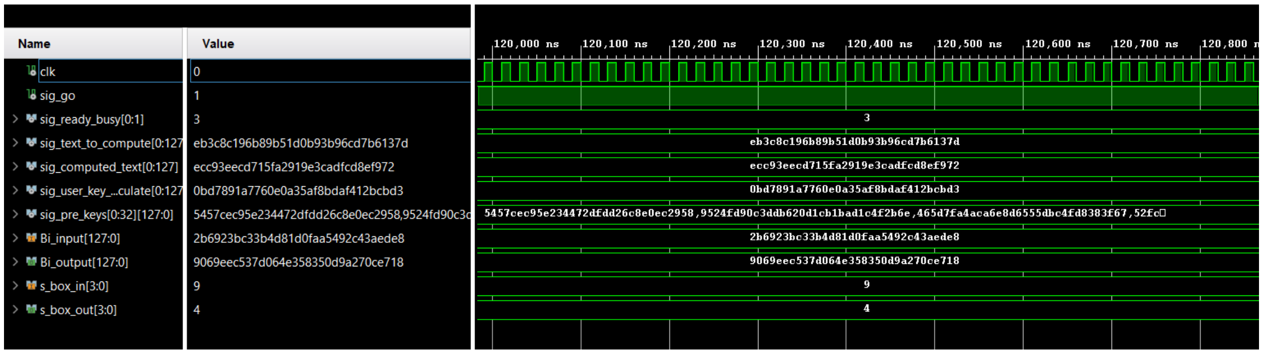
I/Os	Direction	Type	Description
clk	IN	std_logic	clock synchronous signal
go	IN	std_logic	activation synchronous signal
ready_busy	OUT	std_logic	flag asynchronous signal
text_to_compute	IN	std_logic_vector(127 downto 0)	Plain text to compute
computed_text	OUT	std_logic_vector(127 downto 0)	Resulted text
user_key_to_calculate	IN	std_logic_vector(127 downto 0)	The user key

4.5.3 Test-bench :

This block has two features that can be tested:

- Data-path : The data transmission between sub_blocks (i.e : Sending for example 128 bits from s_boxe to the linear transformation or ask a unique sub-keys from the key-scheduler...).

- Results : Checking the results we get is very important to have a good final overall result.



```
Name                            Value
clk                             0
sig_go                          1
sig_ready_busy[0:1]             3
sig_text_to_compute[0:127       eb3c8c196b89b51d0b93b96cd7b6137d
sig_computed_text[0:127]        ecc93eecd715fa2919e3cadfcd8ef972
sig_user_key_...culate[0:127    0bd7891a7760e0a35af8bdaf412bcbd3
sig_pre_keys[0:32][127:0]       5457cec95e234472dfdd26c8e0ec2958,9524fd90c3c
Bi_input[127:0]                 2b6923bc33b4d81d0faa5492c43aede8
Bi_output[127:0]                9069eec537d064e358350d9a270ce718
s_box_in[3:0]                   9
s_box_out[3:0]                  4
```

```
3
eb3c8c196b89b51d0b93b96cd7b6137d
ecc93eecd715fa2919e3cadfcd8ef972
0bd7891a7760e0a35af8bdaf412bcbd3
5457cec95e234472dfdd26c8e0ec2958,9524fd90c3ddb620d1cb1bad1c4f2b6e,465d7fa4aca6e8d6555dbc4fd8383f67,52fc
2b6923bc33b4d81d0faa5492c43aede8
9069eec537d064e358350d9a270ce718
9
4
```

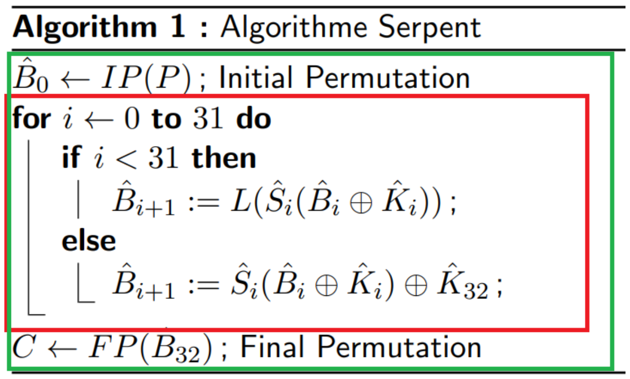
Figure 15: Minor state machine simulation

We can see in the simulation that the data-paths are working just fine, because all the signals are interconnected and the data is flowing just right. But we cannot yet verify the integrity of the data because, my algorithm implementation is way different in logic than the python one. Thus we don't have the same coding rounds.

4.6. General state machine :

4.6.1 Definition :

After finishing all the blocks, we need a big support to interconnect all of them. In our case a major state machine will do the job to finish the overall algorithm. If we refer another time to this figure :



Algorithm 1 : Algorithme Serpent

```
$\hat{B}_0 \leftarrow IP(P)$ ; Initial Permutation
for $i \leftarrow 0$ to $31$ do
    if $i < 31$ then
        $\hat{B}_{i+1} := L(\hat{S}_i(\hat{B}_i \oplus \hat{K}_i))$ ;
    else
        $\hat{B}_{i+1} := \hat{S}_i(\hat{B}_i \oplus \hat{K}_i) \oplus \hat{K}_{32}$ ;
$C \leftarrow FP(B_{32})$ ; Final Permutation
```

Figure 16: Algorithm division

The General state machine work is the green part, it is going to do the initial permutation, call the minor state machine then compute the final permutation step.

4.6.2 I/Os :

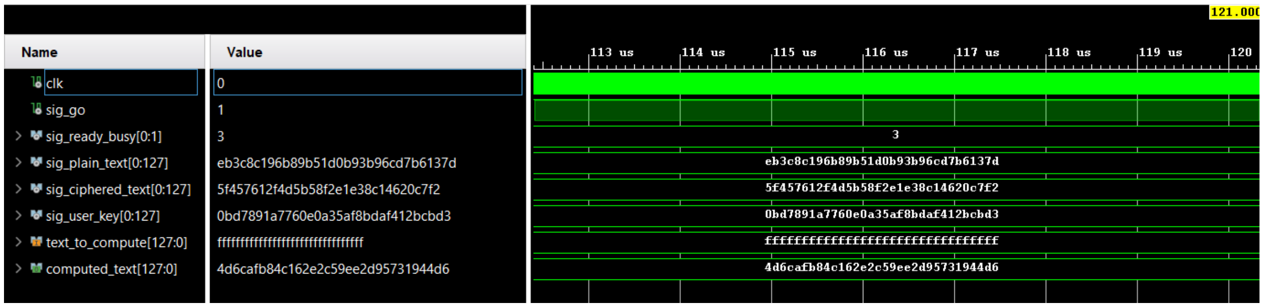
Referring to figure 4 page 11, we can already see the input outputs of the overall black-box, which are Encryption key, plaint text and ciphered text, to those I added some other control signals to have a more accurate authority over the bloc.

Table 9: General state machine 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
plain_text	IN	std_logic_vector(127 downto 0)	Plain text P
ciphered_text	OUT	std_logic_vector(127 downto 0)	Ciphered text C
user_key	IN	std_logic_vector(127 downto 0)	User key k

4.6.3 Test-bench :

Just like The minor state machine, we have to testing chunks (data-path and result), those are going to be seen in the simulation below :



Name	Value
clk	0
sig_go	1
sig_ready_busy[0:1]	3
sig_plain_text[0:127]	eb3c8c196b89b51d0b93b96cd7b6137d
sig_ciphered_text[0:127]	5f457612f4d5b58f2e1e38c14620c7f2
sig_user_key[0:127]	0bd7891a7760e0a35af8bdaf412bcbd3
text_to_compute[127:0]	ffffffffffffffffffffffffffffffff
computed_text[127:0]	4d6cafb84c162e2c59ee2d95731944d6

121.000
113 us
114 us
115 us
116 us
117 us
118 us
119 us
120
3
eb3c8c196b89b51d0b93b96cd7b6137d
5f457612f4d5b58f2e1e38c14620c7f2
0bd7891a7760e0a35af8bdaf412bcbd3
ffffffffffffffffffffffffffffffff
4d6cafb84c162e2c59ee2d95731944d6

Figure 17: Major state machine simulation

Here we observe that the computed_text signal that make the bridges between the Minor and Major state machines is not null, thus we can say that the connection between those two state machines is established without problems.
The connection between the Major state machine, initial and final permutation can be verified by looking to the final result; we see that it is not undefined, thus all the data-paths are 100% working.

4.6.4 Validation truth table :

To simplify the truth table, we are going to use one user key and different input palintexts.
User-key : 9cebe380585ead1e46ed7d975e5c4079

Table 10: General state machine truth table

Plain text P	Ciphered text C
52c95b9e9ca2e086f7fcc40c5096111a	fa90602c5a14679d28969642a4e92cee
5ed4e4f218da99669fe99601d4304304	a016d463a1b2b183e598545f2e94fccf
6f7927c66c805fb856e7abf7db37f4e1	c4d011d92a8ba00426e98c314e7a2809
1aa5a86bff2ebf86d890daeace596ab1	135fd989e54fdc05b9c2a918c1955a55
9ffb8ac13145b4b72e38f208c69941b6	46c873fda362a493b54f03d455f25dcb

5 FPGA ANALYSIS :

The overall analysis is going to be related to post synthesis, because my implementation at the current version does not go beyond this step due to some configuration that cannot be synthesised.

5.1. Timings :

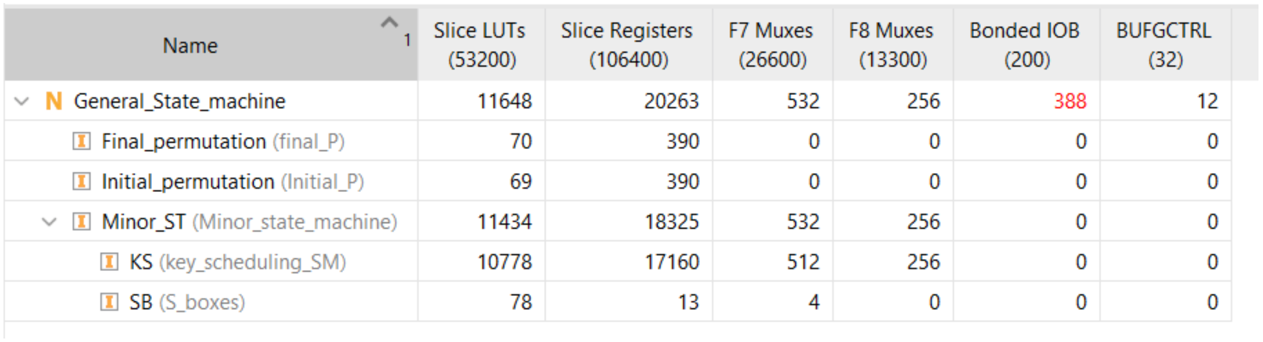
In this implementation, the FPGA is set to work on any possible clock frequency but it is not yet tested. On my simulation, I use 50 MHz frequency as it is very stable and easy to observe on simulation.
The implementation is tested at a threshold of 100Ghz and everything works very fine.

5.2. Throughput :

Taking the most stable timings we can use with my implementation, which is 50MHz. Every plain-text we enter will have a latency of 120µs to have it's corresponding ciphered text. From this we can calculate the throughput efficiency of my implementation :
Eff = 1S / 120µS = 8333 Ciphering operations in one second

5.3. Resource utilization :

In term of resource utilization on the FPGA the table below is showing the number of different component saturation :



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Bonded IOB (200)	BUFGCTRL (32)
General_State_machine	11648	20263	532	256	388	12
Final_permutation (final_P)	70	390	0	0	0	0
Initial_permutation (Initial_P)	69	390	0	0	0	0
Minor_ST (Minor_state_machine)	11434	18325	532	256	0	0
KS (key_scheduling_SM)	10778	17160	512	256	0	0
SB (S_boxes)	78	13	4	0	0	0

Figure 18: Resource utilization

6 CONCLUSION :

To conclude, My serpent implementation runs at a good speed and it does not use a lot of FPGA resource. It proves that FPGA based algorithm are very fast in term of quickly processing data, which is very important when you need a strong encryption pioneered with a fast data movement.

Even thought the implementation is not 100% optimized, the performance difference between it and the python implementation is majestic. In hardware level data flows and movements are quite impressively fast. Which surpasses other implementation using Python or even C++.

I used smart techniques to make the design efficient, which is a good starting point for future improvements and fine-tuning. However, it's important to remember that thorough testing, verification, and strict security measures are essential when using my FPGA-based Serpent cipher in real systems. Security threats are always evolving, so it's crucial to keep cryptographic algorithms secure by adding security layers.

To sum up, a lot of improvement are going to be applied to this serpent ciphering to make it even robust and fast. The final point will never be reachable, improvements will always be possible as technology continues to advance. The overall experience while developing this algorithm is very satisfying and enriching, this project has been a fulfilling journey of exploration and learning. Building the FPGA implementation of the Serpent cipher presented various challenges, and it required a significant amount of research and dedication. Throughout the process, I gained a deeper understanding of FPGA design, encryption algorithms, and their real-world applications.

REFERENCES

[1] Serpent algorithm definition. [Online]. Available: 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)

[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