JERUSALEM COLLEGE OF TECHNOLOGY

FPGA Design and Implementation Of SEED Algorithm

Project Report

By:

Ariel Zadok

Advisor: Uri Stroh

Video is available at: https://youtu.be/PpKWyhohS5A

Abstract

This is my senior-year Electrical Engineering project at Jerusalem college of Technology.

The goal of the project was to design and implement encryption block cipher algorithm on an FPGA device.

The project was carried out on the basys3 board, a complete, ready-to-use digital circuit development platform based on the latest Artix-7 Field Programmable Gate Array (FPGA) from Xilinx. With its high-capacity FPGA (Xilinx part number XC7A35T1CPG236C), low overall cost, and collection of USB, VGA, and other ports, the Basys3 can host designs ranging from introductory combinational circuits to complex sequential circuits like embedded processors and controllers.

Raspberry Pi 3 for the user interface software.

As in many cases, one of the designer's main challenges is to carefully maintain balance between Timing considerations and Area considerations.

Therefore, I used Iterative Implementation. I manage to implement two operation modes – encryption and decryption, with a very friendly user interface software.

This implementation demonstrates a fair combination of the two factors.

The trade off and the throughput results were very satisfying and engaging.

Table of Contents

1. Introduction

1.1 A Brief History

Cryptography, or cryptology, is the practice and study of techniques for secure communication in the presence of third parties called adversaries. More generally, cryptography is about constructing and analyzing protocols that prevent third parties or the public from reading private messages, various aspects in information security such as data confidentiality, data integrity, authentication, and non-repudiation are central to modern cryptography. Modern cryptography exists at the intersection of the disciplines of mathematics, computer science, electrical engineering, communication science, and physics. Applications of cryptography include electronic commerce, chip-based payment cards, digital currencies, computer passwords, and military communications.

1.2 Modern Cryptography

Modern cryptography is heavily based on mathematical theory and computer science practice. Cryptographic algorithms are designed around computational hardness assumptions, making such algorithms hard to break in practice by any adversary. It is theoretically possible to break such a system, but it is infeasible to do so by any known practical means. These schemes are therefore termed computationally secure. Theoretical advances, e.g., improvements in integer factorization algorithms, and faster computing technology require these solutions to be continually adapted. There exist information-theoretically secure schemes that probably cannot be broken even with unlimited computing power—an example is the one-time pad—but these schemes are more difficult to use in practice than the best theoretically breakable but computationally secure mechanisms.

These techniques (encryption algorithms) used to encrypt and protect massages while transferring from sender to receiver using various functions. The sender and receiver use a special key known as cipher key which is then operated on along with the message to create an encrypted message which hides the plain text and then sent to the receiver. The receiver then using the reverse mode (decryption) of the algorithm with the same special key to decrypt the massages.

1.3 FPGA Technology

Field Programmable Gate Arrays (FPGAs) are semiconductor devise that are based around a matrix of configurable logic blocks connected via programmable interconnects. Straight from the manufacturer, they come with no hardcoded architecture, and resemble a blank "field of logical gates" (hence the name) which can be reprogrammed according to the desired application. Although one-time programmable FPGAs are available, the dominant types are those which can be reprogrammed as the design evolves.

When programmed, connections are formed between the various logic blocks, which can implement every logical function, and even implement a CPU or DSP in the FPGA fabric (since they are all essentially made from the same logic gates).

An FPGA design has an inherent parallel nature. Since it is a direct implementation of a digital circuit, the signal can propagate through the entire system like in a circuit, allowing to implement many different designs in short period of time. Here some advantages using FPGA Technology for implementing cryptographic algorithms:

- **Algorithm flexibility**
  Algorithm flexibility is the switching of cryptographic algorithms during operation of the targeted application. The majority of modern security protocols, such as SSL or IPsec, are algorithm independent and allow for multiple encryption algorithms. These encryption algorithms are negotiated on a per-session basis and a wide variety may be required. For example, IPsec allows among others DES, 3DES, CAST, IDEA, RC4, RC6 and Blowfish, as algorithms, and future extensions are also possible. Some of the advantages of protocols that do not depend on the algorithm are the ability to delete broken algorithms the choice of algorithms according to certain preferences the possibility of adding new algorithms. While this kind of flexibility is costly with traditional hardware, FPGAs can be reprogrammed on-the-fly. A good example of taking advantage of this property is where an Adaptive Cryptographic Engine is proposed. This ACE has the FPGA technology at its core. Besides the FPGA device the system also has a configuration controller and a cryptographic library. The FPGA is configured on the fly by the configuration controller.

Subsequently, adaptation to the input key occurs and the data encryption/decryption commences. The configuration controller chooses the proper configuration to be based on the requested security association.

- **Algorithm Distribution**
  It is perceivable that fielded devices are upgraded with a new encryption algorithm at some point. A reason for this could be that the product has to be compatible to new applications. From a cryptographical point of view, the distribution of a new algorithm can be necessary because a current algorithm was broken, the list of ciphers in an algorithm independent protocol (like IPSec) was expanded, or a standard expired (Data Encryption Standard) and a new one was created (Advanced Encryption Standard). Assuming a connection to a network such as the Internet exists, encryption devices equipped with FPGA can upload the new configuration code. It is important to stress that the upgrade of ASIC-implemented algorithms is practically infeasible if many devices are affected or if the systems are not easily accessible, for instance in satellites, while FPGA-implemented ones are easily upgradable.

- **Algorithm Modification**
  Even though most cryptographic algorithms are standardized there are some applications that require some modifications when implementing them. For example, we can introduce permutations at different points or specific substitution boxes if we think of adapting the well-known symmetrical algorithms that use this component. These modifications can be easily achieved with reconfigurable hardware. The UNIX password encryption constitutes an example where a standardized algorithm was slightly changed. In this case DES is used 25 times in a row and a 12-bit salt modifies the expansion mapping (the Unix password encryption has been standardized).
  It is also attractive to customize block cipher algorithms such as DES or AES with proprietary S-boxes for certain applications.

Furthermore, there are occasions when cryptographic primitives or their operation modes must be changed according to the application.

- **Resource Efficiency**
  Most security protocols used today are hybrid protocols, like IPSec, SSL, TLS . This means, that a public-key algorithm is used to transmit the session key. After the key was established a private-key algorithm is used for data encryption. This is done in order to take advantage of both the secure channel provided by the public-key algorithm for symmetrical key exchange, and the faster encryption rates provided by the private-key algorithm. Since the algorithms are not used simultaneously, the same FPGA device can be utilized for both through run-time reconfiguration. This makes better use of the resources available which is an important factor in many implementations.

- **Architecture Efficiency**
  In some cases, an architecture can be much more efficient if it is designed for a specific set of parameters. The parameters for a cryptographic algorithm can be, for example, the key, the underlying finite field, the coefficients used (the specific curve of an ECC system), and others. In general, the more specific an algorithm is implemented the more efficient it can become. As an efficient parameter-specific implementation example, I chose the symmetric cipher IDEA.
  This implementation states that with fixed keys, the main operation in the IDEA becomes a constant multiplication which is far more efficient than a general modular multiplication. In the case of IDEA, a general modular shift-and-add multiplication requires 16 partial multiplications while only eight are needed for a fixed key.

- **Cost Efficiency**
  There are two cost things, that must be considered when analyzing the cost efficiency of FPGAs: development cost and the prices of units. The costs for developing an FPGA implementation of an algorithm are much lower than for an ASIC implementation, because we are actually able to use

the given structure of the FPGA (for example, the look-up table) and to test the reconfigured chip for as many times as we wish without any further costs. This results in a shorter time-to-market period, which is nowadays an important cost factor. The unit prices are not so significant as a cost efficiency factor when comparing them to the development costs. In fact, for high-volume applications, ASIC implementations usually become the more cost-efficient choice.

- **Data transfer efficiency**
  When judging data transfer efficiency, we can safely state that general-purpose CPUs are not optimized for fast execution especially in the case of public-key algorithms. That happens mainly because they lack the instructions for modular arithmetic operations on long operands. These operations include, for example, multiplication, squaring, inversion, and addition for elliptic curve cryptosystems (ECC) and exponentiation for RSA. Although they are typically slower than ASIC implementations, the FPGA implementations have the potential of running substantially faster than software implementations.

1.4 Project Goals

In this project, I wanted to experiment with the concept of implementing an encryption block cipher algorithm on an FPGA, instead of the common software platforms.

My main goals are:

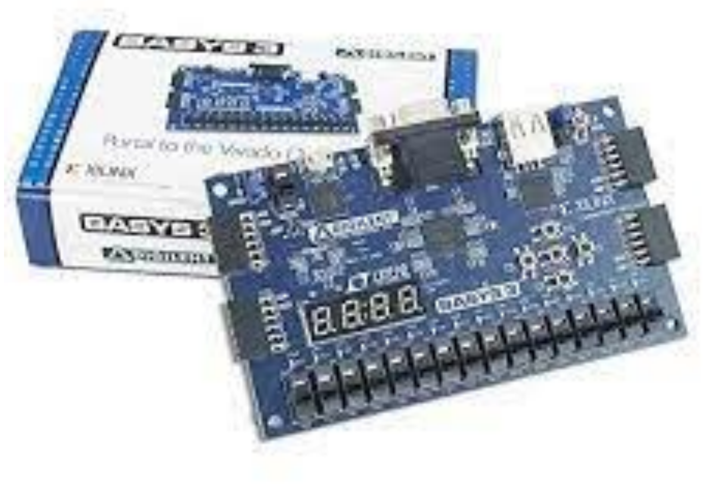
- Get familiar with different aspects of FPGA design.
- Obtain knowledge and experience in Verilog and FPGA development in general, as this field grows in popularity in recent years.
- Get familiar with the unique Basys 3 devices architecture.
- Develop a full practical system, which can be used for information security and digital signature.
- Achieve communication between an FPGA board and an external software board.
- Explore and experience some advantages of hardware design over software.
- Put my engineering skills and effort to create something practical, which will help, individual or collective to protect their data.

1.5 Platform and Tools

I used the Basys3 board, a complete, ready-to-use digital circuit development platform based on the latest Artix-7 Field Programmable Gate Array (FPGA) from Xilinx. With its high-capacity FPGA (Xilinx part number XC7A35T1CPG236C), low overall cost, and collection of USB, VGA, and other ports, the Basys 3 can host designs ranging from introductory combinational circuits to complex sequential circuits like embedded processors and controllers.

It includes enough switches, LEDs, and other I/O devices to allow many designs to be completed without the need for any additional hardware, and enough uncommitted FPGA I/O pins to allow designs to be expanded using Digilent Pmods or other custom boards and circuits.

The Artix-7 FPGA is optimized for high performance logic, and offers more capacity, higher performance, and more resources than other designs. Artix-7 35T features include:



- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1,800 Kbits of fast block RAM
- Five clock management tiles, each with a phase-locked loop (PLL)
- 90 DSP slices
- Internal clock speed exceeding 450MHz
- On-chip analog-to-digital converter (XADC)

The Basys 3 works with Xilinx's new high-performance Vivado Design Suite.

Vivado includes many new tools and design flows that facilitate and enhance the latest design methods. It runs faster, allows better use of FPGA resources, and allows to focus time evaluating design alternatives.

In my project, I used few of the Basys3 components:

- 3 Pmod ports (PmodA, PmodB, PmodC) to transmit and receive the data
  (Callout 2)
- Shared UART/JTAG USB port (Callout 13)
- FPGA programming done LED (Callout 8)
- Power Switch (Callout 15)
- Power Select Jumper (Callout 16)
- Programming mode jumper (Callout 10)

- Middle Pushbutton (Callout 7)

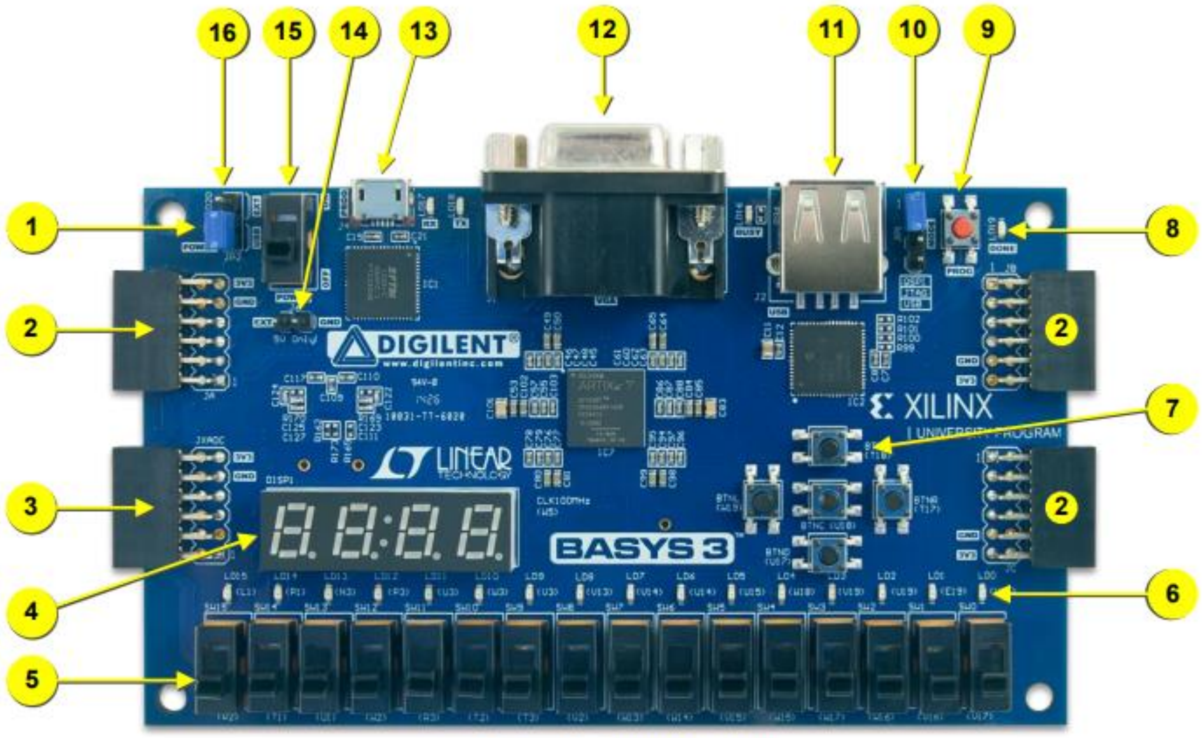


Figure 1. Basys 3 FPGA board with callouts.

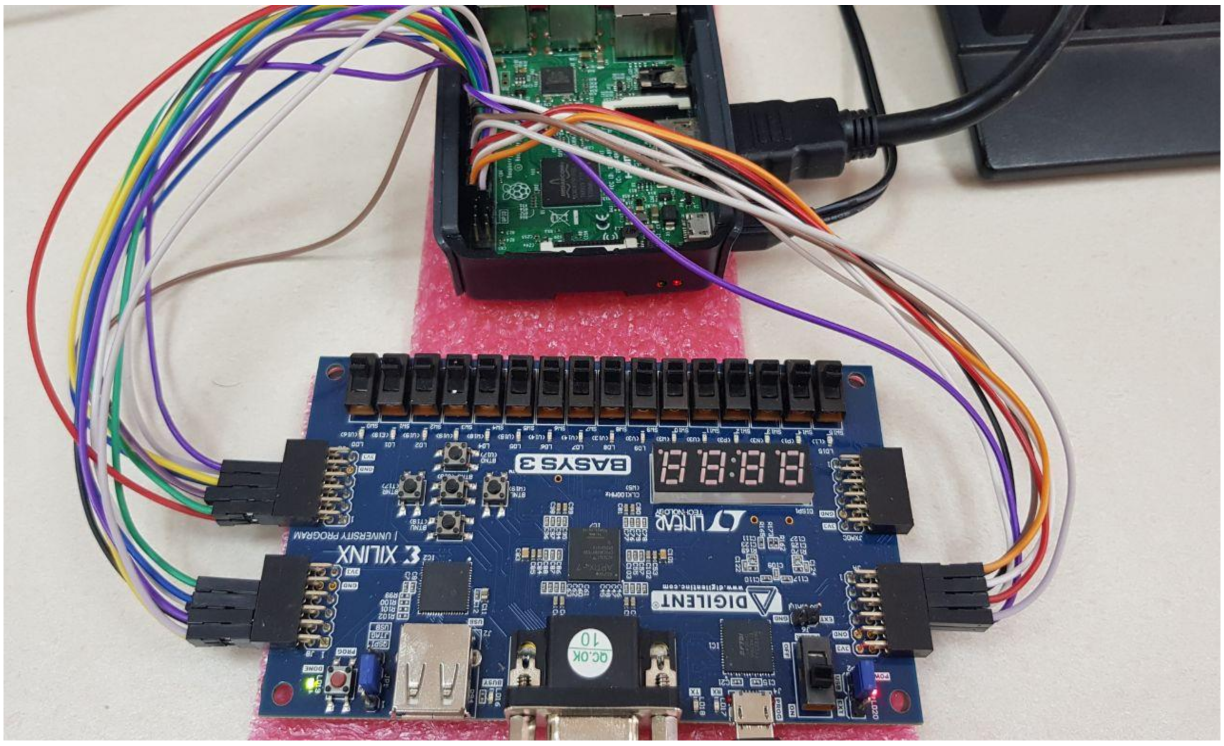
Callout	Component Description	Callout	Component Description
1	Power good LED	9	FPGA configuration reset button
2	Pmod port(s)	10	Programming mode jumper
3	Analog signal Pmod port (XADC)	11	USB host connector
4	Four digit 7-segment display	12	VGA connector
5	Slide switches (16)	13	Shared UART/ JTAG USB port
6	LEDs (16)	14	External power connector
7	Pushbuttons (5)	15	Power Switch
8	FPGA programming done LED	16	Power Select Jumper

Table 1. Basys 3 Callouts and component descriptions.

I also used Raspberry Pi 3 for dealing with user input and output operations.

RPi3

Here you can see the basys3 connected throw 23 jumper wires to the Raspberry Pi3:



Software

The main working environments I used was:

- Verilog
- Vivado 2018.3
- Vivado 2016.2

Also:

- Precision RTL 2018.1
- Modelsim 10.6d
- Python3 for the RPI3

Hardware

- Basys3 board
- Artix-7 FPGA (XC7A35T1CPG236C)
- Raspberry Pi 3 (and a monitor)
- 23 jumper wires

2. Project Overview

2.1 General

Since the goal of the project was to implement an encryption block cipher algorithm on an FPGA, I used the raspberry pi 3 and a monitor to deal with user input and output operations.

The user inputs a Secret special key and a massage (plaintext), and of course the mode operation he wants to use – Encryption or Decryption.

The raspberry pi is converting the key and the plaintext to binary, padding to each one of them to 128-bits and wait for the user to start the requested operation - simply by clicking on the middle pushbutton of the Basys3.

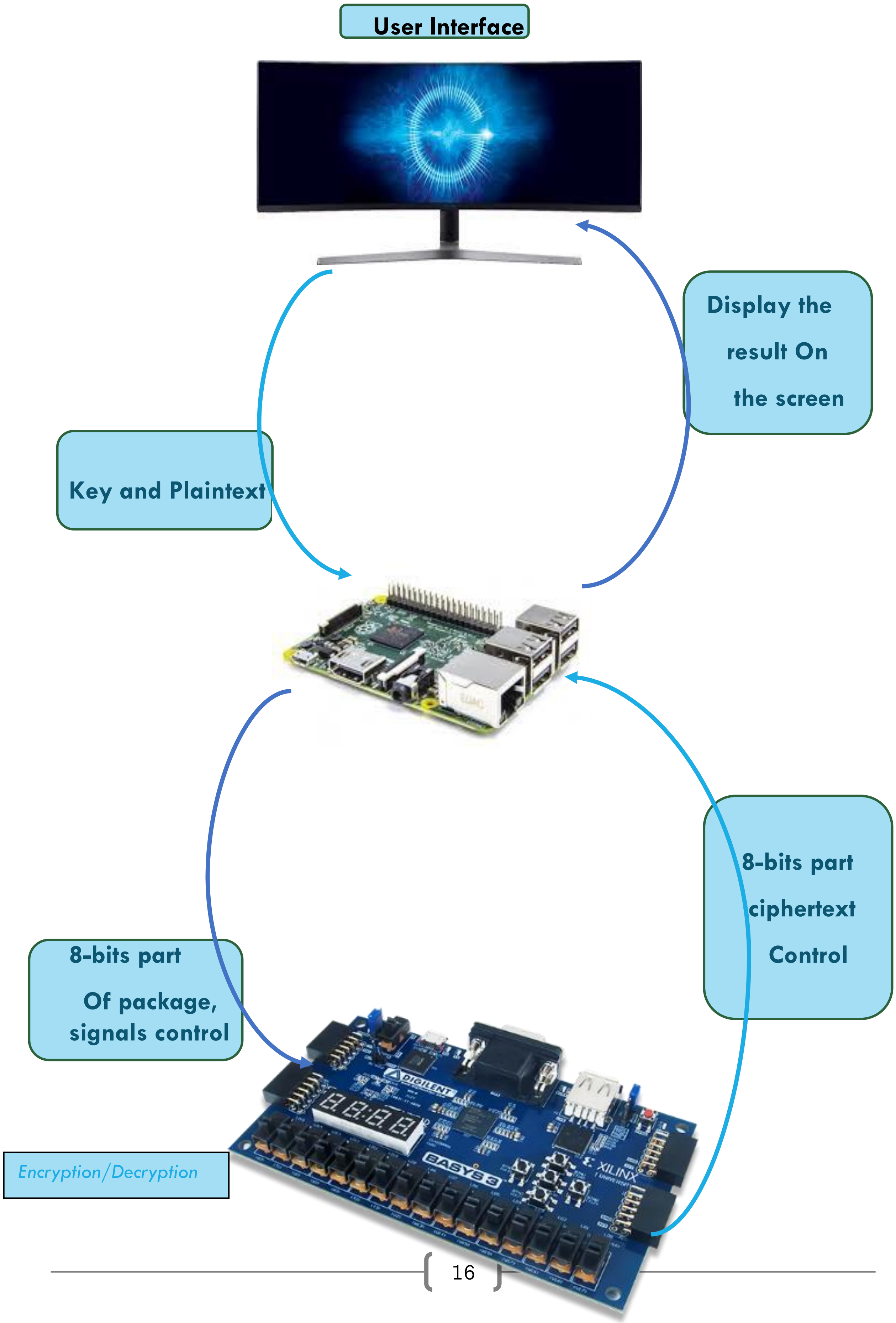
As a result, the raspberry pi start transmitting 8 bit (byte) throw his pins to the Basys3's Pmods with enable, load and start signals. When the basys3 receive voltage in the load signal he knows he need to sample the 8-bits in his Pmod and store them. When load signal stays low (and when an internal counter that count the length of a package reach his maximum), the basys3 knows he can start encrypting/decrypting.

As soon as the encryption/decryption operation finished, the basys3 sends a done signal to the raspberry pi to inform him to get ready to receive the ciphertext.

Subsequently, the basys3 sends a load signal towards the raspberry pi, as a result, the raspberry pi knows he needs to sample his pins and store the data.

Finally, the raspberry pi converting the ciphertext from binary and displays them on the screen for the user.

2.2 Simple - Data Flow Diagram


User Interface
Display the result On the screen
Key and Plaintext
8-bits part ciphertext Control
8-bits part Of package, signals control
Encryption/Decryption

2.3 SEED Algorithm

Today’s requirements for improved time performance and secure communications impose system designers the need to look for efficient hardware implementations of cryptographic algorithms. Additionally, more and more sensitive data like bank accounts, medical records, personal emails and secret keys are stored digitally and must be kept secure. For all these reasons the new encryption algorithms have to operate efficiently in a variety of current and future applications, performing different encryption tasks. Various cryptographic algorithms such as Triple-DES, AES (Advanced Encryption Standard), KASUMI and many others have been developed to accomplish this.

Another algorithm aiming at this is the SEED block cipher. SEED is a cipher developed by the Korean Information Security Agency. It is used broadly throughout South Korean industry and recently has been adopted by the International Organization for Standardization (ISO/IEC 18033-3 standard) for usage in a wide range of applications, both software and hardware.

In digital signal processing, the design of fast and computationally efficient algorithms has been a major focus of research activity. The objective is the design of algorithms and their respective implementation in a manner that the required computations are performed very fast.

This Project was executed via Iterative Implementation, an efficient implementation, in terms of hardware resources / time performance of the SEED block cipher is presented. In order to reduce the required hardware resources and to improve the cipher performance, feedback logic and inner-round pipeline techniques are used. For the Iterative Implementation, positive edge-special triggered registers are used. Recently, many designs have been proposed for the hardware implementation of the SEED block cipher. From those, the implementations in ASIC technologies were used. In my design, a very compact implementation is presented. Each functional module is implemented only once and is used sequentially in order to minimize the area, which however, results in small throughput.

My implementation is intended for applications with high performance requirements so, therefore I executed via Iterative Implementation.

Down below you can see a simple diagram of the Algorithm:

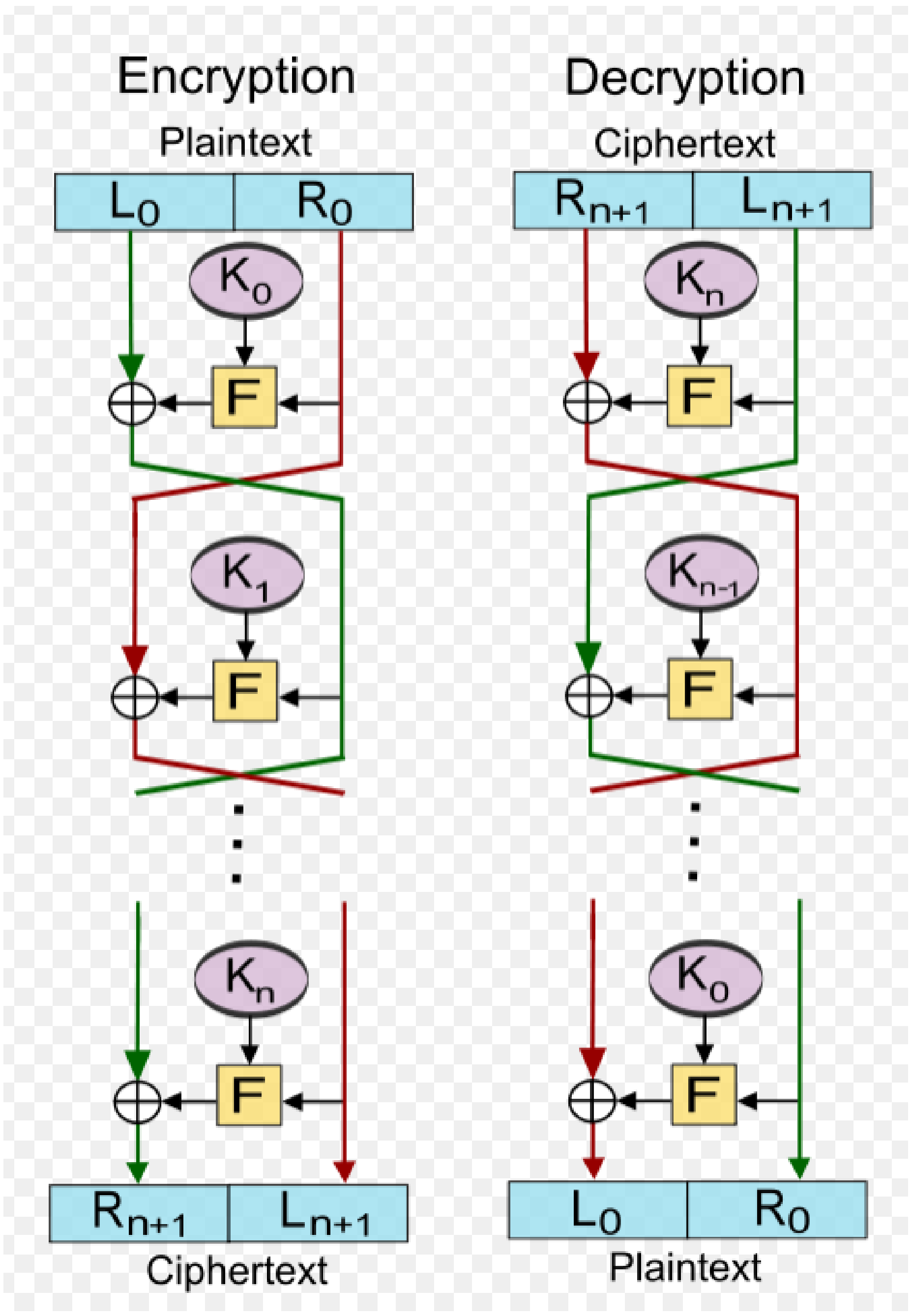


Figure.2 – Fiestel Network

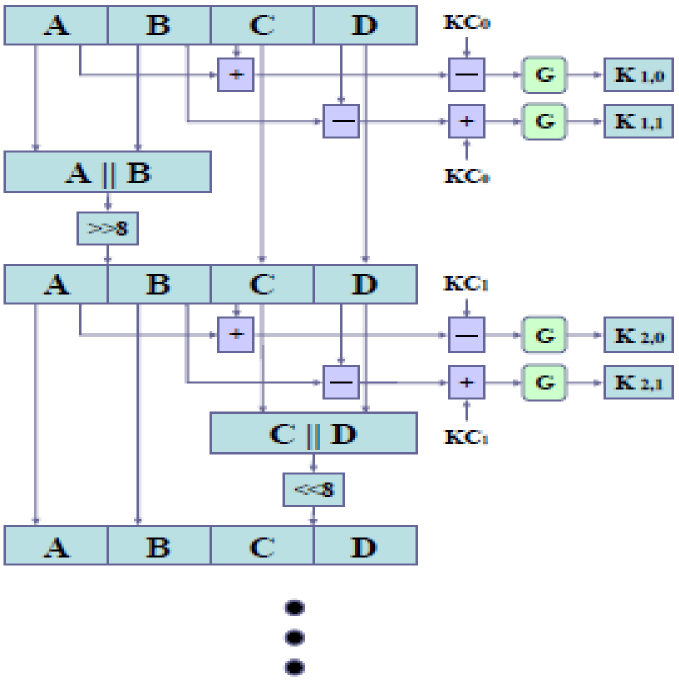


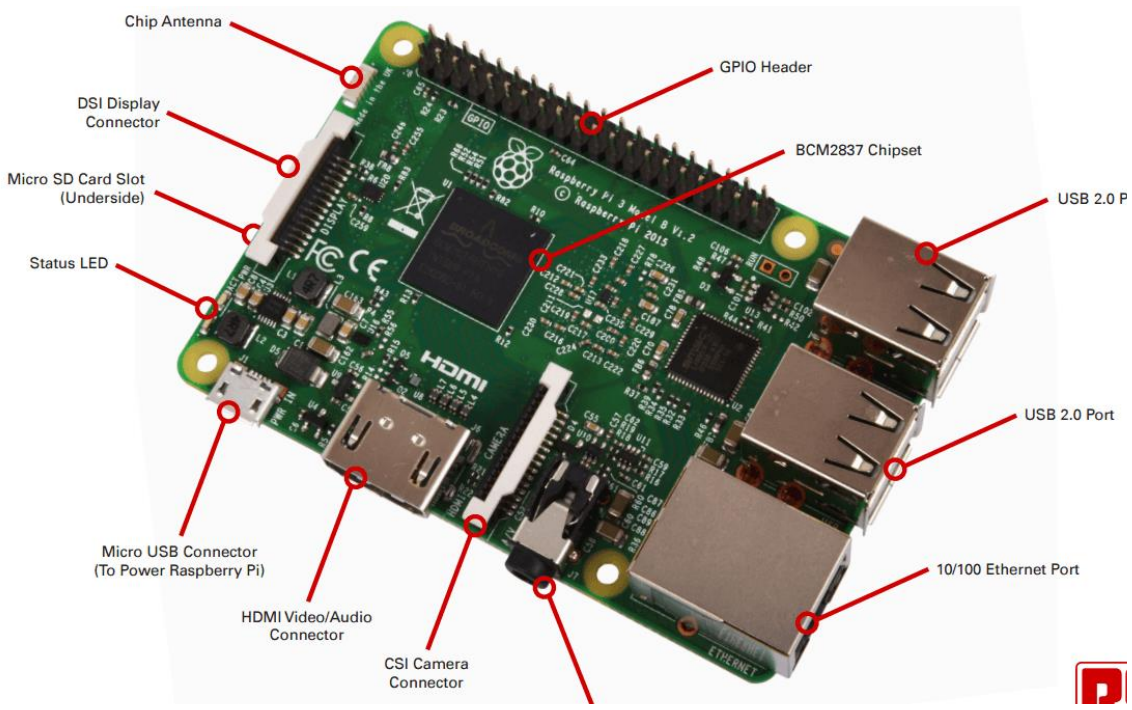
Figure.1 – Key Schedule

2.4 Raspberry Pi 3

The Raspberry Pi 3's four built-in USB ports provide enough connectivity for a mouse, keyboard, or anything else that you feel the RPi needs, but if you want to add even more you can still use a USB hub.

In my project the components I used are:

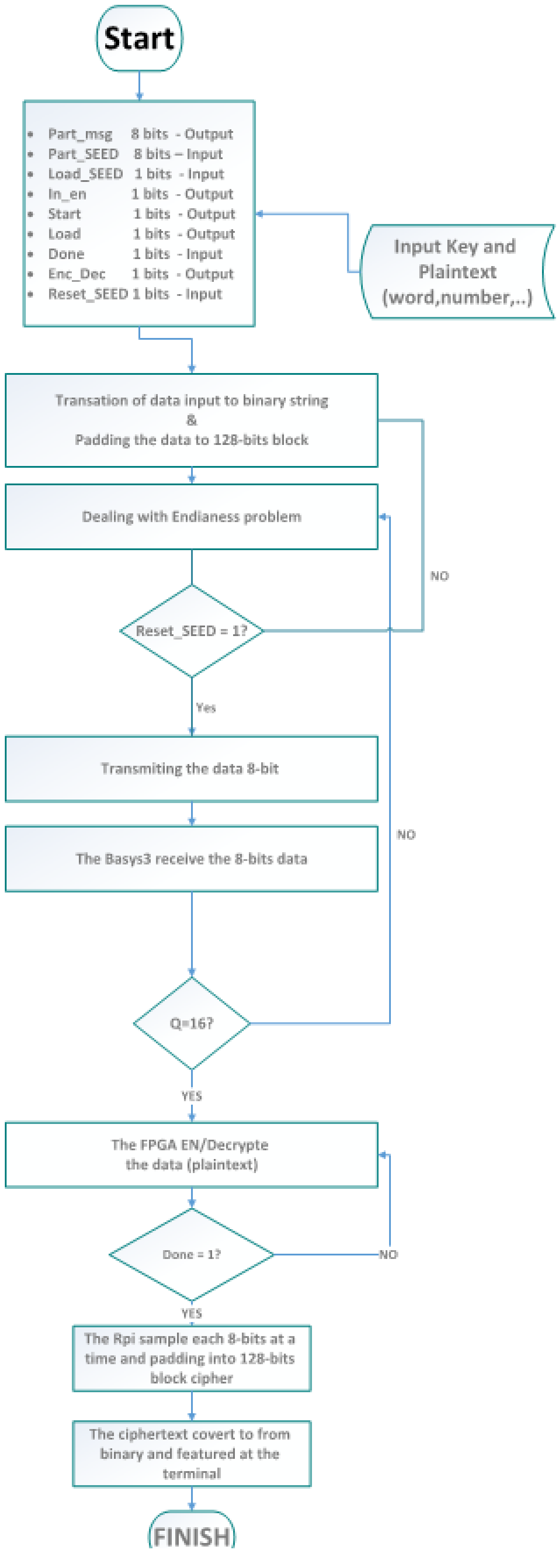
- Three USB ports (keyboard, mouse and disk-on-key).
- Full Size HDMI Video/Audio Connector.
- Micro USB Power Connector.
- 23 pins Extended GPIO.



It receives the key and the plaintext from the user, stores and pads them.

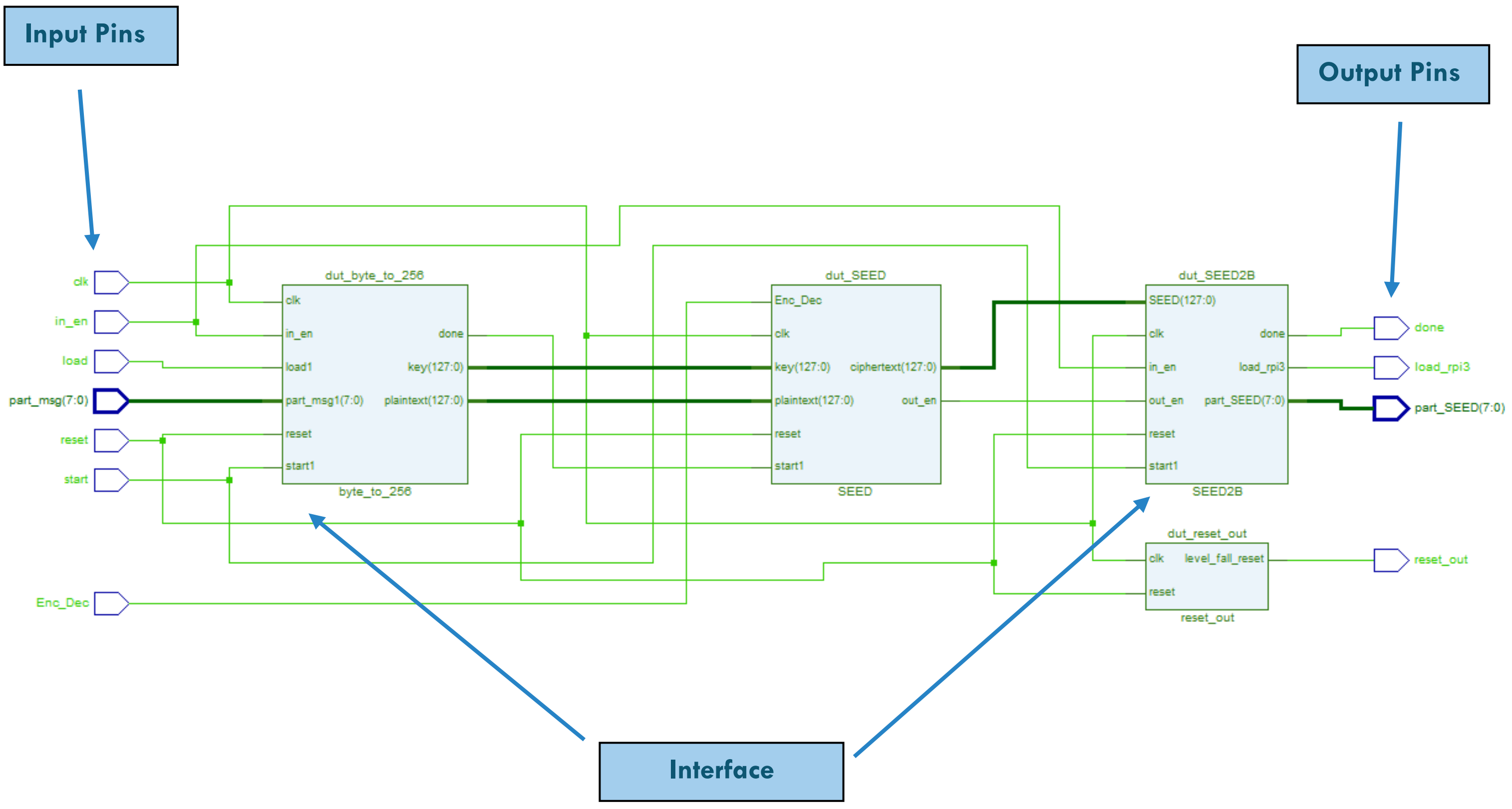
When the user gives the signal, he transmits the plaintext and the key, also it receives the ciphertext and display the result on the screen.

Down below is the Data Flow Diagram of the Raspberry Pi 3:


Start
Part_msg 8 bits - Output
Part_SEED 8 bits – Input
Load_SEED 1 bits - Input
In_en 1 bits - Output
Start 1 bits - Output
Load 1 bits - Output
Done 1 bits - Input
Enc_Dec 1 bits - Output
Reset_SEED 1 bits - Input
Input Key and Plaintext (word,number,..)
Transation of data input to binary string & Padding the data to 128-bits block
Dealing with Endianess problem
Reset_SEED = 1?
NO
Yes
Transmiting the data 8-bit
The Basys3 receive the 8-bits data
NO
Q=16?
YES
The FPGA EN/Decrypte the data (plaintext)
Done = 1?
NO
YES
The Rpi sample each 8-bits at a time and padding into 128-bits block cipher
The ciphertext covert to from binary and featured at the terminal
FINISH

3. Building Blocks

3.1 Top Level



This is the Top-Level module of the implementation.

It is doing all the encryption/decryption operations and of course the receiving and transmitting operations from/and the raspberry pi.

It has four blocks:

- Byte_to_256
- SEED
- SEED2B
- Reset_out

We will talk about each one of them separately.

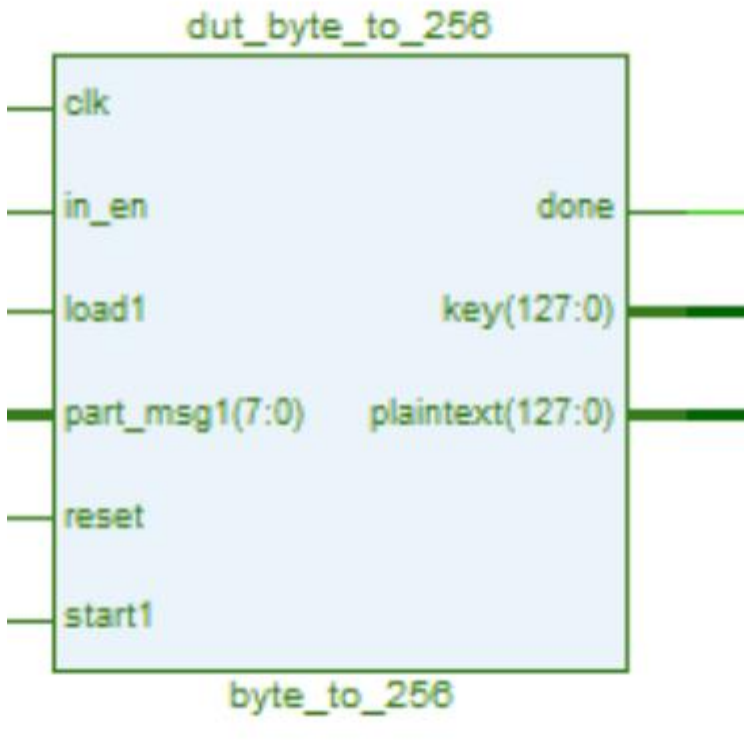
Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- In_en - High when there are more blocks to send for encryption
- Load - when high - new 8 bits had been loaded to pins
- Part_msg - 8 bits as part of a block
- Start - when high it means that now was the first package
- Enc_Dec – determines whether encryption or decryption is performed.

Output Pins:

- Part_SEED – 8 bits of the 256 bits SEEDed message
- Load_rpi3 – when high – raspberry pi will sample the bits from his pins.
- Done – When high inform the raspberry pi – encryption/decryption operation is over.
- Reset_out – When pressed send a stable signal and ignore bouncing.

3.1.1 byte_to_256



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- In_en - High when there are more blocks to send for encryption
- Load1- when high - new 8 bits had been loaded to pins
- Part_msg1 - 8 bits as part of a block
- Start1 - when high it means that now was the first package

Output Pins:

- Plaintext – 128 bits user's plaintext
- Key – 128 bits user's secret key
- Done – high when all 32 bytes of block arrived

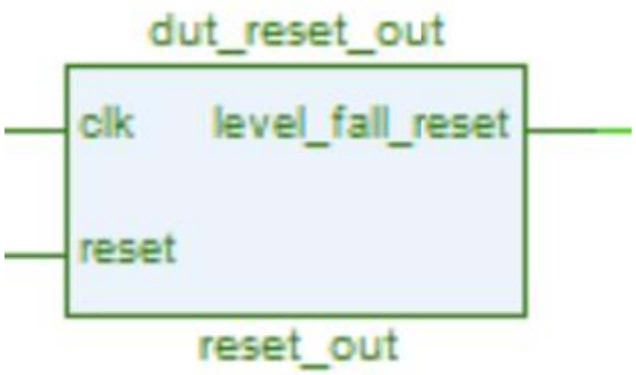
This module in charge of all the input system of the implementation.

It receives 32 times 8 bits from input pins and concatenate it to 256-bit block,

then sends him to another block for the encryption/decryption operation.

When it finishes receiving, transmit a pulse (done) that confirm that the block is ready.

3.1.2 Reset_Out



Input Pins:

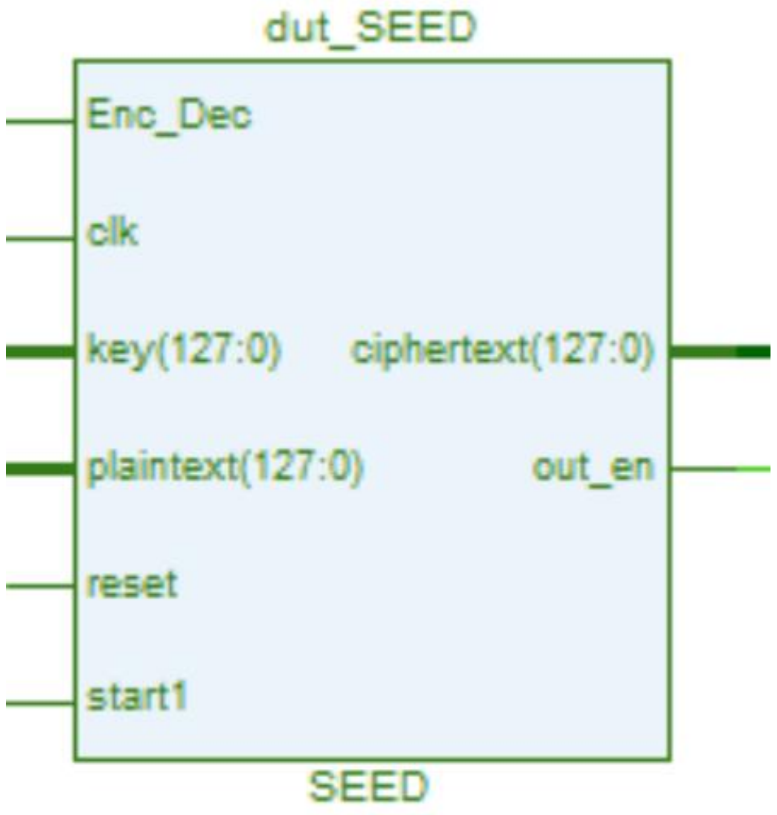
- Clk – internal 100MHz clock
- Reset - active high synchronous reset

Output Pins:

- Level_fall_reset – a stable pulse of RESET

I made tests and found out that if I want that a signal will pass through the jumper wires and the Raspberry Pi 3 will notice them, they should be stable for relatively long time. This module sends this kind of signal, when RESET button is pressed, sends a stable signal and ignore bouncing.

3.1.3 SEED



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- Start1 - when high it means you can start encryption/decryption operation
- Enc_Dec – determines whether encryption or decryption is performed.
- Plaintext – 128 bits user's plaintext
- Key – 128 bits user's secret key

Output Pins:

- Ciphertext – 128 bits result of the SEED algorithm
- Out_en – high when the encryption/decryption operation is over

In this module all the encryption and decryption executed.

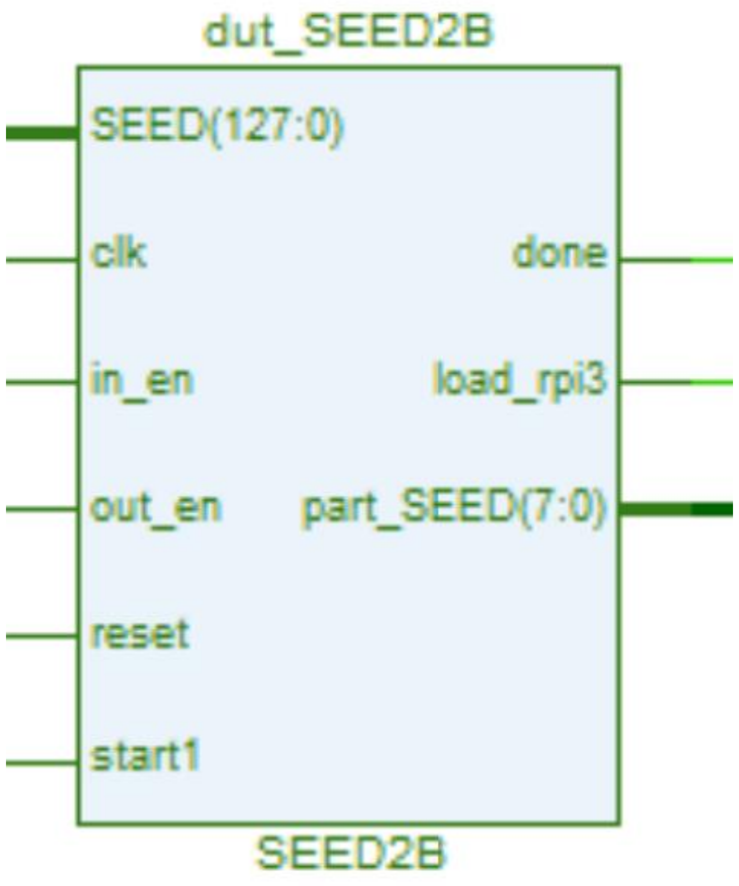
His execution requires 16 loops of this single round.

The execution time of each round is two clk_en clock cycle. The output of each round is used as input (through module B) of the next round.

The output of the left branch is used as input in the next left branch (through module A).

Down below you can find a full explanation about SEED's internal functions.

3.1.4 SEED2B



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- Start1 - when high it means that now was the first block
- In_en – High when there are blocks to send for encryption
- SEED – 128 bits result of the SEED algorithm

Output Pins:

- part_SEED – 8 bits of the 128 bits result of the SEED algorithm
- load_rpi3 – when high inform Raspberry Pi 3 that the encryption/decryption operation is over

- done – high when needs to inform Raspberry Pi 3 to read the pins of 8 bits of the 128 bits result of the SEED algorithm.

At this module when the SEED is ready - 16 registers are loaded with the 16 bytes of the 128 bits result of the SEED algorithm.

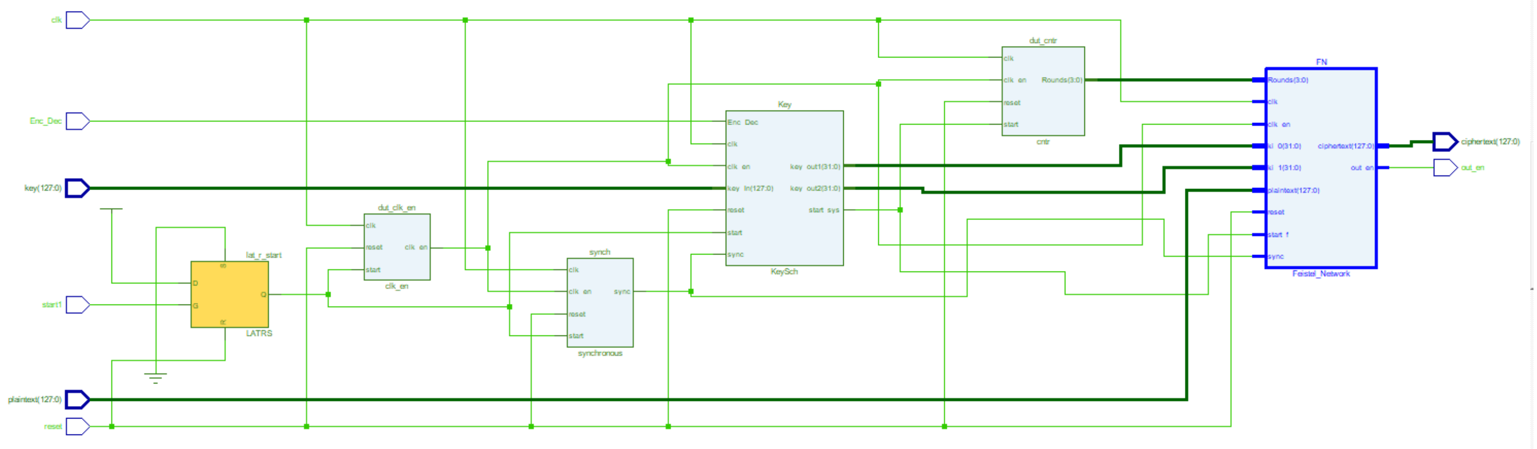
Each clock enables the information to continue to next register till all the bits had been transmitted.

When load_rpi3 is high the first register will be transmitted from the Pmods to the Raspberry Pi 3.

According to my experiments, the optimal pulse for the Raspberry Pi 3 load signal is 20ms,

less than this it won't notice the pulse.

3.2 SEED



This module has five blocks:

- Key – the Key schedule
- FN – the Feistel network
- Synch – synchronous signal

- Clk_en – clock enable signal
- Cntr – counter for the Rounds

3.2.1 Key Schedule

This module is the "Key" of the Algorithm, it receives the 128-bit cipher key from the user, generates 32 Sub-Keys and transfers them to the Feistel Network.

The 128-bit cipher key is divided into four 32-bit blocks (Key0, Key1, Key2, Key3).

These blocks are processed by six registers: four registers transfer when sync is low, two registers transfer when sync is high, 4 Multiplexer, 16 rounds of addition/subtraction with round coefficients, 8-bit left/right rotation, and the G-function in order to generate subkeys for all 16-rounds.

Two 32- bit positive edge-special triggered registers are used before the G-function. The round keys are repeatedly generated on-the- fly by the structure down below. Only one structure block is designed for one process of round key generation and is sequentially used. The first 64-bit round key is generated by the concatenation of two 32-bit keys, calculation of two expressions (consisting of an addition and a subtraction each one) and passing the results through the two G functions. Also, four 32-bit registers are used for the temporary storing of the key values.

Each 32-bit key is concatenated with the neighboring key and rotated by 8-bits to the left or to the right in order to generate the new values of the Key0, Key1, Key2 and Key3.

This process is repeated 16 times in total in order to generate the 16 round keys.

The two 32-bit subkeys of the first round, $K_{1,0}$ and $K_{1,1}$ are generated as following:

$K_{1,0}$ = G (A + C – KC_0), $K_{1,1}$ = G(B – D + KC_0).

The two 32-bit subkeys of the 2nd round, $k_{2,0}$ and $k_{2,1}$ are generated from the input key with 8-bit right rotation of the first 64-bits(A||B) as follows:

A||B <= (A||B)>>8.

$K_{2,0}$ = G (A + C – KC_1), $K_{2,1}$ = G(B + KC_1 – D).

The two subkeys of the 3rd round, $K_{3,0}$ and $K_{3,1}$ are generated from the 8-bit left rotation of the last 64-bit(C||D) as follows:

C||D <= (C||D) <<8.

$K_{3,0}$ = G (A + C – KC_2), $K_{3,1}$ = G(B – D + KC_2).

The rest of the subkeys are generated iteratively. A pseudo code for the key schedule is as follows:

Input: (Key0, Key1, Key2, Key3)

```
for (i=1; i<=16; i++)
{
$K_{i,0}$ <= G (A + C – $KC_{i-1}$);
$K_{i,1}$ <= G (B – D + $KC_{i-1}$);
if (i%2 == 1)
A||B <= (A||B) >>8
else
C||D <= (C||D) <<8
}
```

Output: (Keyi0, Keyi1), i=1 to 16

where, each constant KC_i is generated from a part of the golden ratio $\frac{\sqrt{5}-1}{2}$ number.

Here is the Table KCi Constants:

Constants in hexadecimal form (KCi)			
i	Value	i	Value
0	0x9e3779b9	8	0x3779b99e
1	0x3c6ef373	9	0x6ef3733c
2	0x78dde6e6	10	0xdde6e678
3	0xf1bbcdcc	11	0xbbcdccf1
4	0xe3779b99	12	0x779b99e3
5	0xc6ef3733	13	0xef3733c6
6	0x8dde6e67	14	0xde6e678d
7	0x1bbcdccf	15	0xbcdccf1b

Decryption mode:

The decryption mode is the reverse operation of the encryption mode. It can be implemented by using the encryption algorithm with reverse order of the round subkeys.

Alternatively, the decryption mode can be implemented if the adder operations are replaced by subtractions throughout the round function F.

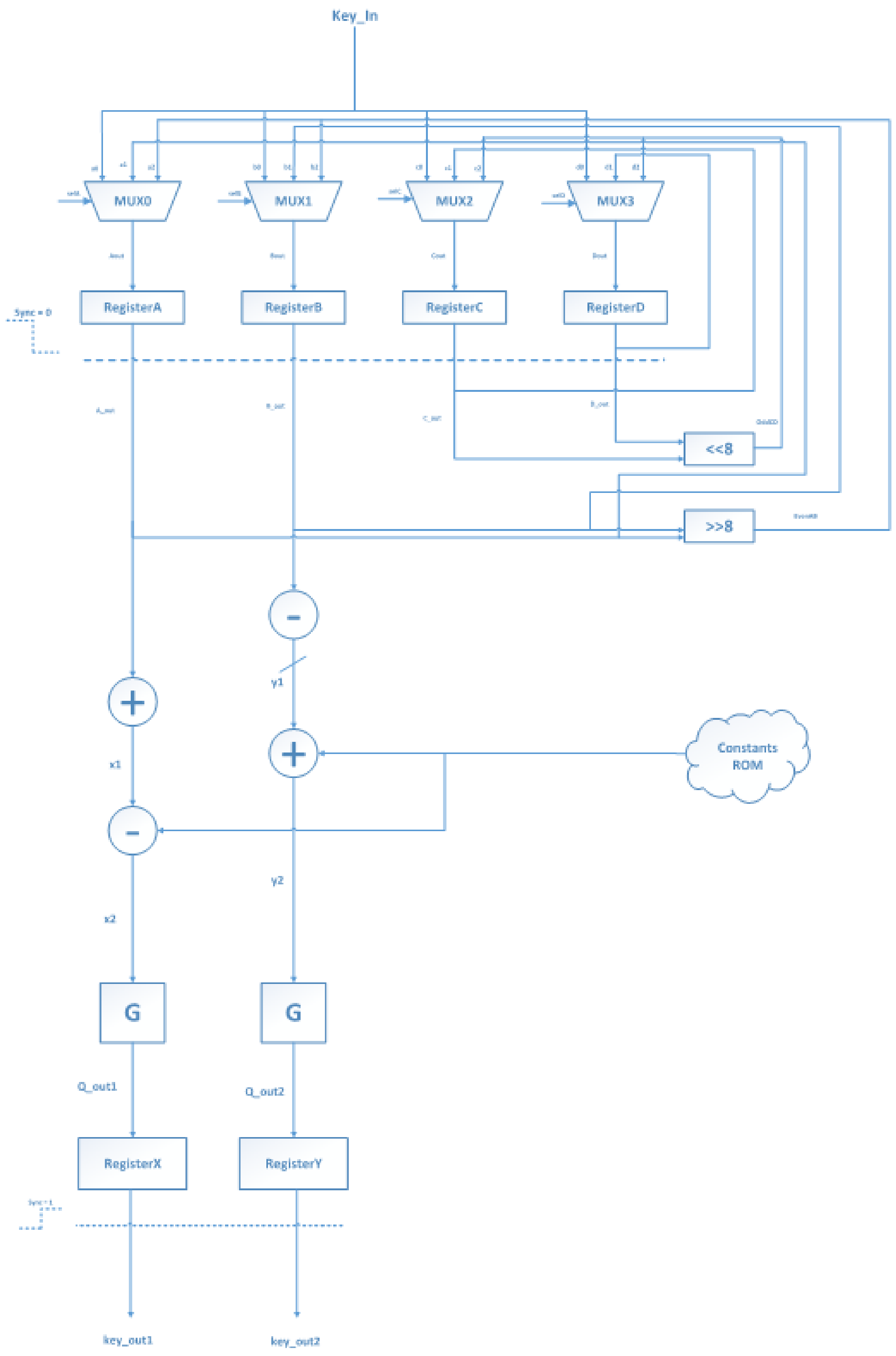
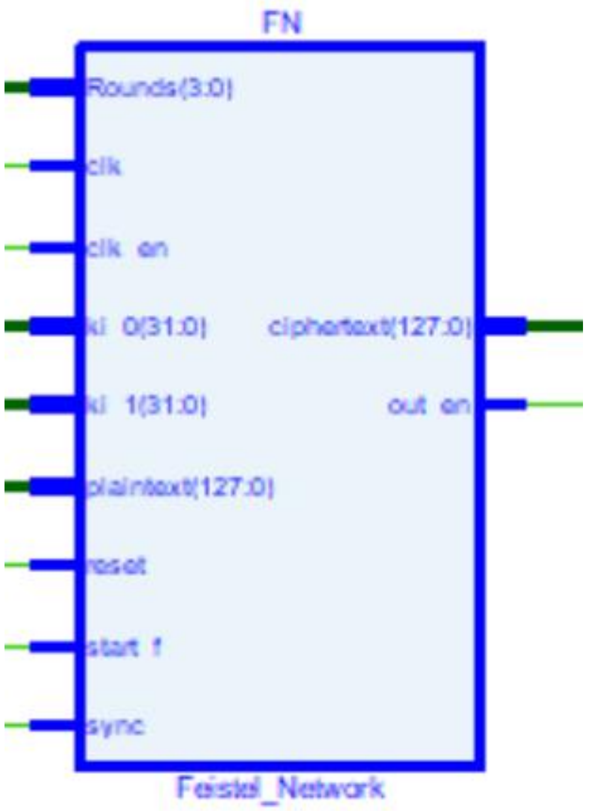


Fig. Proposed architecture of SEED Key Scheduling

3.2.2 Feistel Network



<u>Input Pins:</u>

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- sync - synchronous signal for the registers
- start_f – when high it means that Key is finish creating all his SKs and ready to go
- clk_en – clock enable signal for setup violations
- Rounds – count the rounds of the Algorithm
- Plaintext – 128 bits plaintext
- Ki_0 – 32 bits Subkey1
- Ki_1 – 32 bits Subkey2

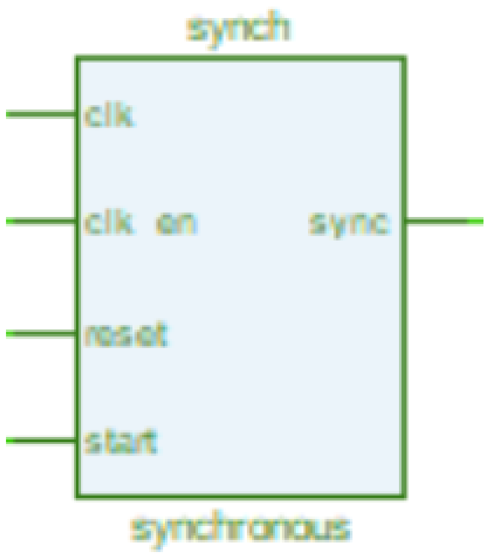
Output Pins:

- Ciphertext – 128 bits ciphertext
- Out_en – high when encryption/decryption operation is over

This module is the core of the algorithm, his inputs are the 128-bit plaintext and the two 32-bit subKeys. The 128-bit plaintext block of the Feistel Network is divided into two 64-bit blocks (L, R), and the right 64-bit block is an input to the round function F, with a 64-bit subkey generated by the Key schedule. L is the most significant 64-bit of 128-bit input, and R is the least significant 64-bit.

Down below you can find a full explanation about Feistel Network internal functions.

3.2.3 Synch



Input Pins:

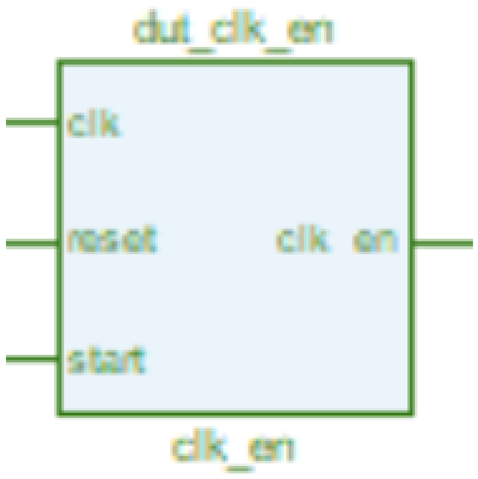
- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- start – when high it means that Key is finish creating all his SKs and ready to go
- clk_en – clock enable signal for setup violations

Output Pins:

- sync - synchronous signal for the registers

This module oversees the synchronous of the system, it sends signal (sync) that change every clock enable. Each time sync is changing different registers are executing. This helps control the data and synchronize the whole system.

3.2.4 clk_en



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- start – when high it means that Key is finish creating all his SKs and ready to go

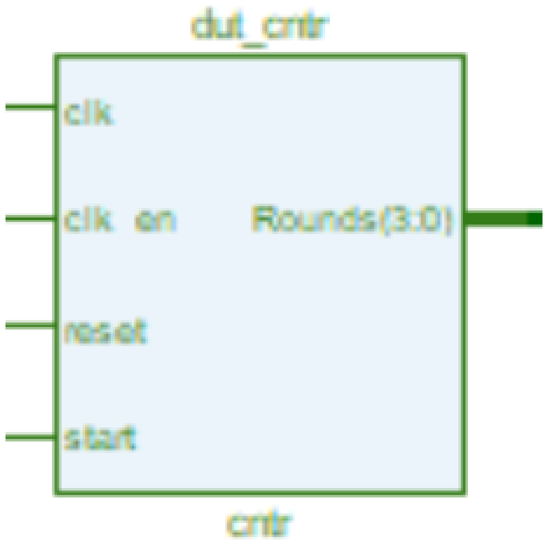
Output Pins:

- clk_en – clock enable signal for setup violations

This module generates a clock enable signal every 2-clock cycle.

The whole system executed when clock enable is high, his purpose is to fix the setup violation of the critical paths.

3.2.5 cntr



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- start – when high it means that Key is finish creating all his SKs and ready to go
- clk_en – clock enable signal for setup violations

Output Pins:

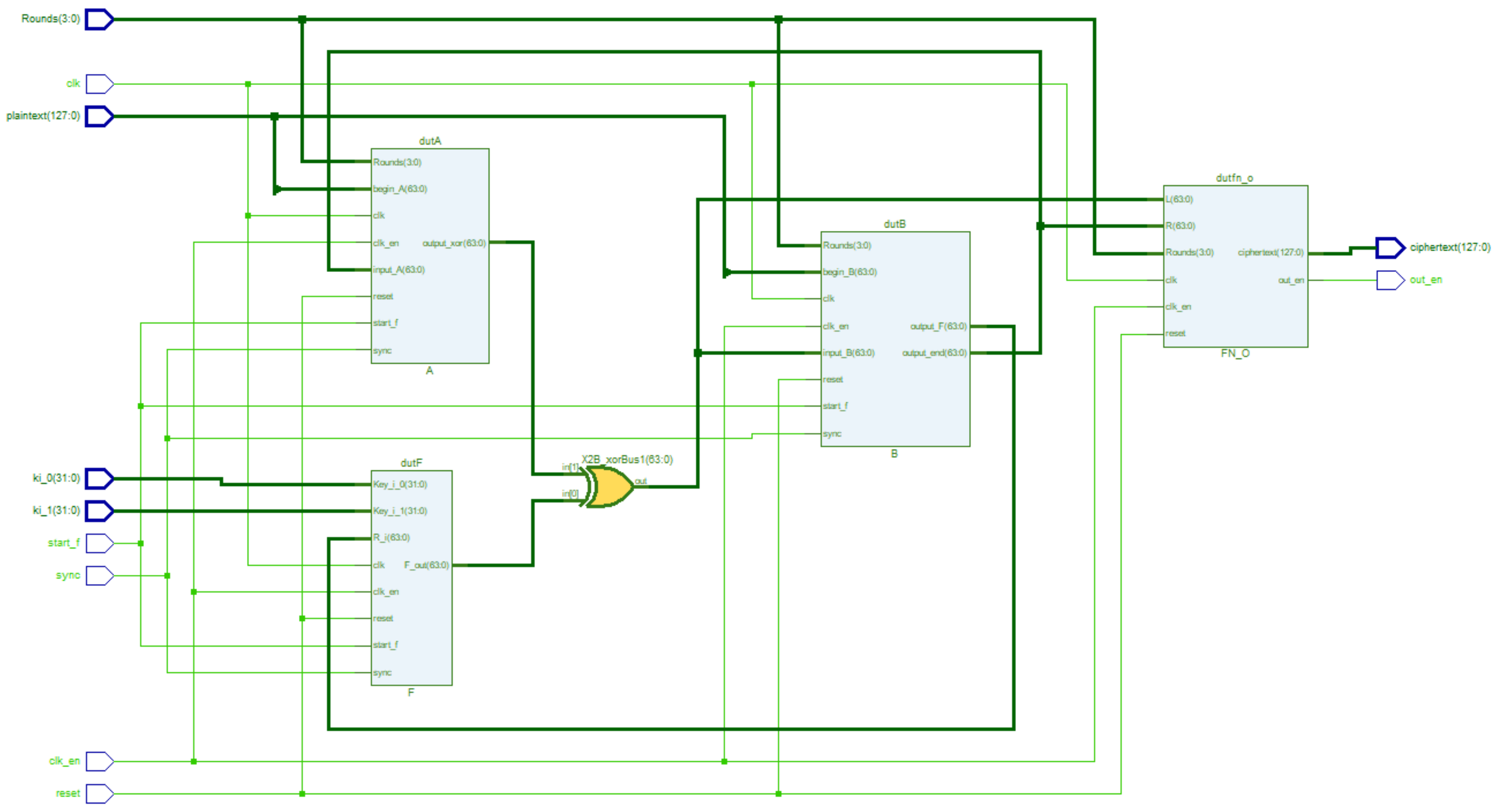
- Rounds - count the rounds of the Algorithm

This module counts every positive edge of clock enable, and generate the round's number to the whole system.

Every two clock enable cycle (four internal clock cycle) counter increasing in one.

The Round's signal equal the counter value divided by two.

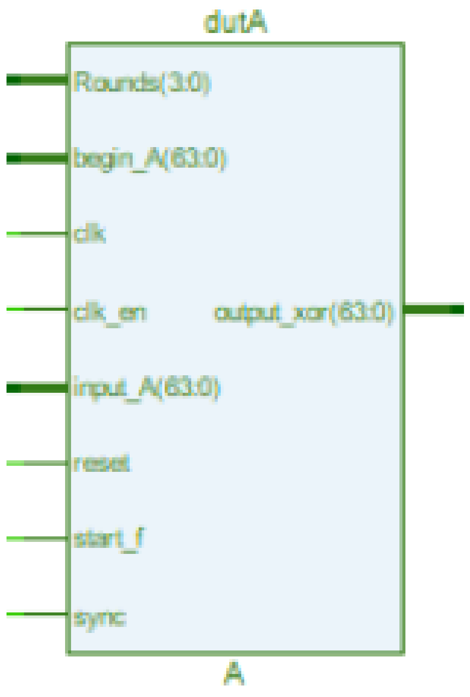
3.3 Fiestel Network



This module has four blocks:

- Module A – oversees the plaintext/ciphertext MS 64-bit
- Module B – oversees the plaintext/ciphertext LS 64-bit
- Function F – the main function of the algorithm
- FN_O – State Machine for the Feistel Network's output signals

3.3.1 Module A



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- Start_f – when high it means that Key is finish creating all his SKs and ready to go
- clk_en – clock enable signal for setup violations
- Rounds – count the rounds of the Algorithm
- Begin_A – plaintext/ciphertext MS 64-bit
- Input_A – last round LS 64-bit

Output Pins:

- output_xor - send the 64-bit input to XOR logic

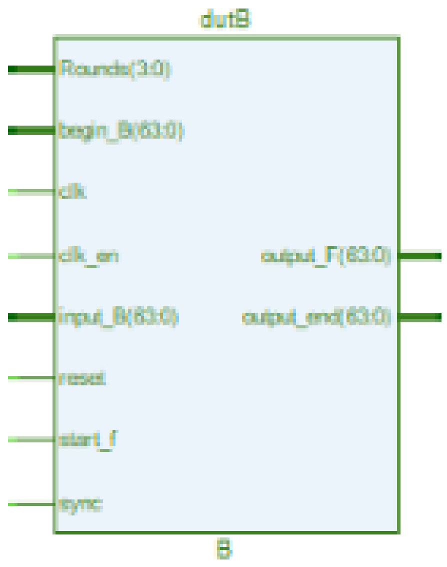
This module is the "left register" of the Feistel Network.

In the first round his job is to store the plaintext/ciphertext MS 64-bit till function F finished (sync is high) and then send them to the xor logic.

In other rounds instead of storing the plaintext/ciphertext MS 64-bit, it stores the last round LS 64-bit.

The input register is necessary in order to store the input data during the cipher operation.

3.3.2 Module B



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- Start_f – when high it means that Key is finish creating all his SKs and ready to go
- clk_en – clock enable signal for setup violations
- Rounds – count the rounds of the Algorithm
- Begin_B – plaintext/ciphertext LS 64-bit
- Input_B – 64-bit last round XOR's result

Output Pins:

- output_F - 64-bit sent to F function
- output_end - 64-bit sent to be hold and store till F function will finish

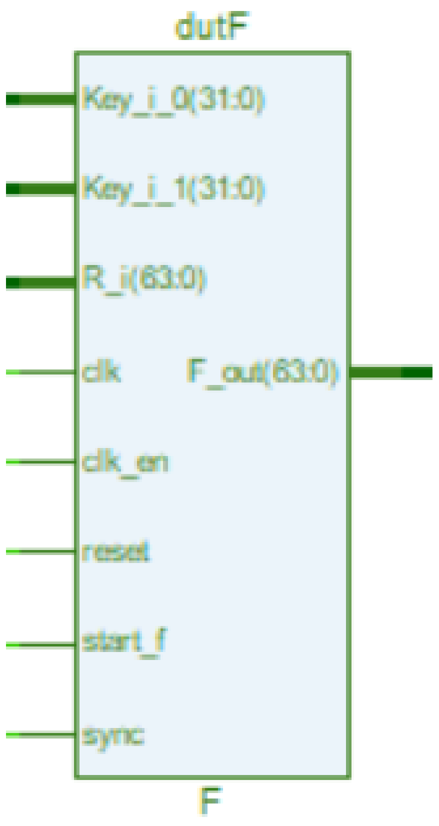
This module has too parts, Combinatorial logic and Sequential logic.

The combinatorial logic is for the first round, the module takes the right 64-bit plaintext and sends them to F function.

The Sequential logic is holding and storing the bits till function F will finish executing his current round (sync is high), then sends the bits to module A.

The input register is necessary in order to store the input data during the cipher operation.

3.3.3 F – function



Input Pins:

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- Start_f – when high it means that Key is finish creating all his SKs and ready to go
- clk_en – clock enable signal for setup violations
- sync - synchronous signal for the registers
- R_i – plaintext/ciphertext LS 64-bit
- Key_i_0 – 32-bit Sub-Key0
- Key_i_1 – 32-bit Sub-Key0

Output Pins:

- F_out - 64-bit transfers to the XOR logic

This module – F function is the main function of the alogorithm which is at the heart of Feistel Network.

A 64-bit input block of the round function F is divided into two 32-bit blocks (R0, R1) and wrapped in 4 phases. Firstly, a mixing phase of two 32-bit subkey blocks (Ki0, Ki1) occur and then 3 layers of function G, with additions for mixing two 32-it blocks take place.

Outputs C', D' of function F with two 32-bit input blocks C, D and two 32-bit subkeys $K_{i,0}$, $K_{i,1}$ are as follows:

$$\begin{aligned} C' &= G[G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} + (C \oplus K_{i,0})] + G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\}] \\ &\quad + G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} + (C \oplus K_{i,0})] \\ D' &= G[G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} + (C \oplus K_{i,0})] + G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\}] \end{aligned}$$

This module has four registers, two in the beginning to hold and store the subkeys and the 64-bit input and executing when sync is low. Another two registers in the end holding and storing the 64-bit result for the synchronization of the system, executing when sync is high.

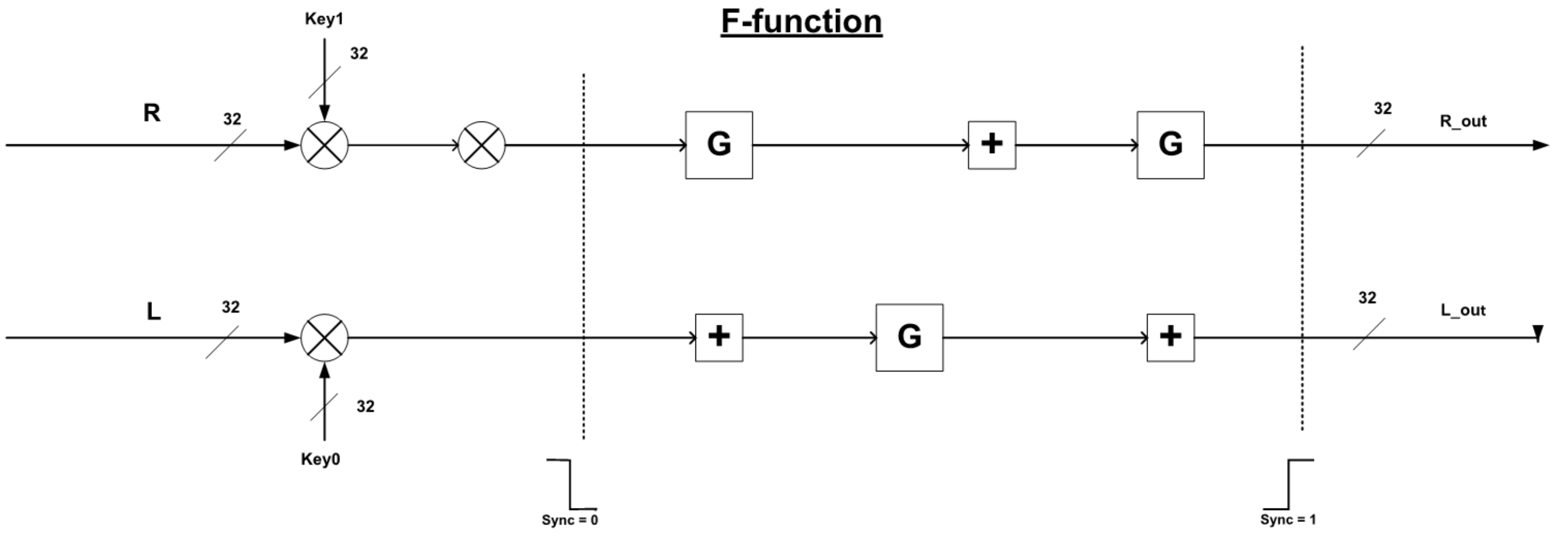
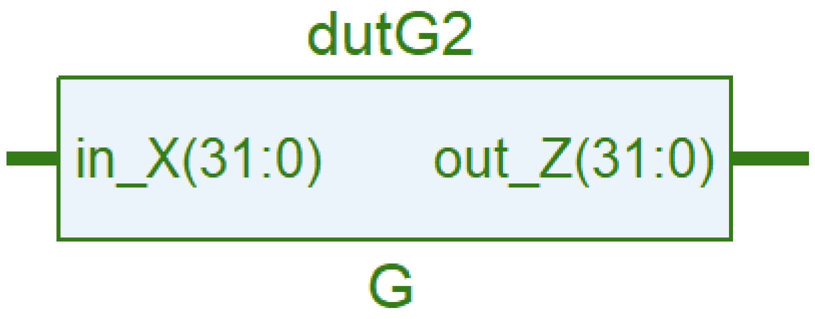


Fig. Proposed architecture of SEED F - function

3.3.4 G – function



The function G has two layers: a layer of two 8 ☐ 8 S-boxes and a layer of block permutation of sixteen 8-bit sub-blocks. The second layer is a set of permutations in each S-box.

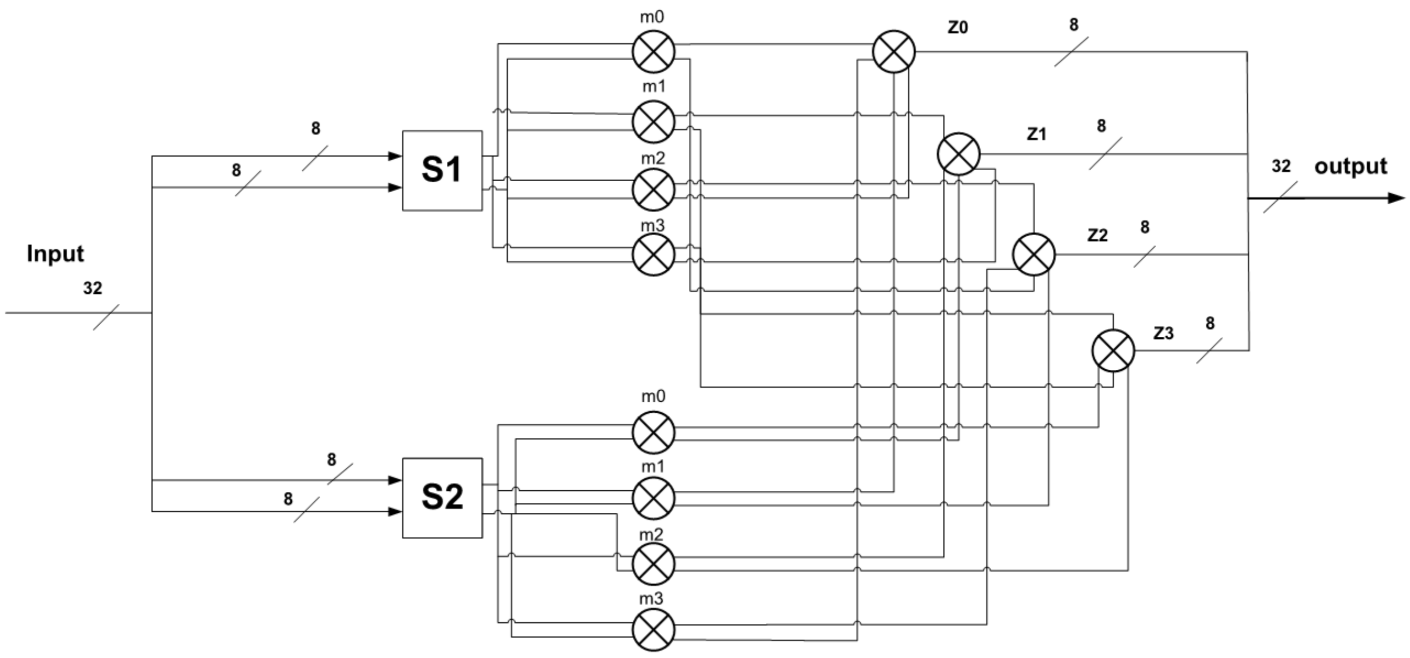
The outputs Z_0, Z_1, Z_2, Z_3 of the function G with four 8-bit inputs X_0, X_1, X_2, X_3 are as follows:

$$
\begin{aligned}
Z_0 &= (S_1(X_0)\,\&\,m_0) \oplus (S_2(X_1)\,\&\,m_1) \oplus (S_1(X_2)\,\&\,m_2) \oplus (S_2(X_3)\,\&\,m_3) \\
Z_1 &= (S_1(X_0)\,\&\,m_1) \oplus (S_2(X_1)\,\&\,m_2) \oplus (S_1(X_2)\,\&\,m_3) \oplus (S_2(X_3)\,\&\,m_0) \\
Z_2 &= (S_1(X_0)\,\&\,m_2) \oplus (S_2(X_1)\,\&\,m_3) \oplus (S_1(X_2)\,\&\,m_0) \oplus (S_2(X_3)\,\&\,m_1) \\
Z_3 &= (S_1(X_0)\,\&\,m_3) \oplus (S_2(X_1)\,\&\,m_0) \oplus (S_1(X_2)\,\&\,m_1) \oplus (S_2(X_3)\,\&\,m_2)
\end{aligned}
$$

where, m_0 = 0xfc, m_1 = 0xf3, m_2 = 0xcf and m_3 = 0x3f.

The structure of function G with S-boxes is shown down below:

G – function with S boxes



To increase the efficiency of G function, I define the extended S-boxes (SS-boxes, 4Kbyte)

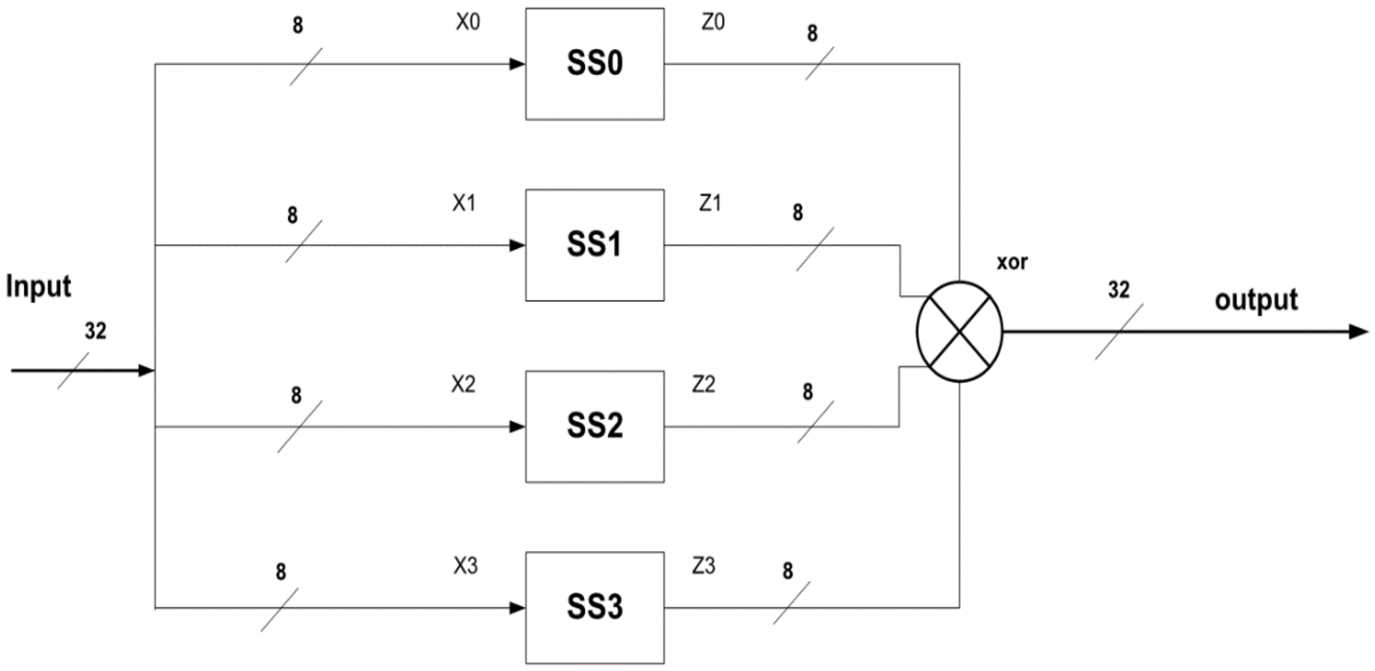
$$SS_3(X) = S_2(X) \& m_2 \,||\, S_2(X) \& m_1 \,||\, S_2(X) \& m_0 \,||\, S_2(X) \& m_3$$
$$SS_2(X) = S_1(X) \& m_1 \,||\, S_1(X) \& m_0 \,||\, S_1(X) \& m_3 \,||\, S_1(X) \& m_2$$
$$SS_1(X) = S_2(X) \& m_0 \,||\, S_2(X) \& m_3 \,||\, S_2(X) \& m_2 \,||\, S_2(X) \& m_1$$
$$SS_0(X) = S_1(X) \& m_3 \,||\, S_1(X) \& m_2 \,||\, S_1(X) \& m_1 \,||\, S_1(X) \& m_0$$

and implement the function G like below equation. (Z is 32bit output of the function G such that Z=Z3||Z2||Z1||Z0)

$$Z = SS_3(X_3) \oplus SS_2(X_2) \oplus SS_1(X_1) \oplus SS_0(X_0)$$

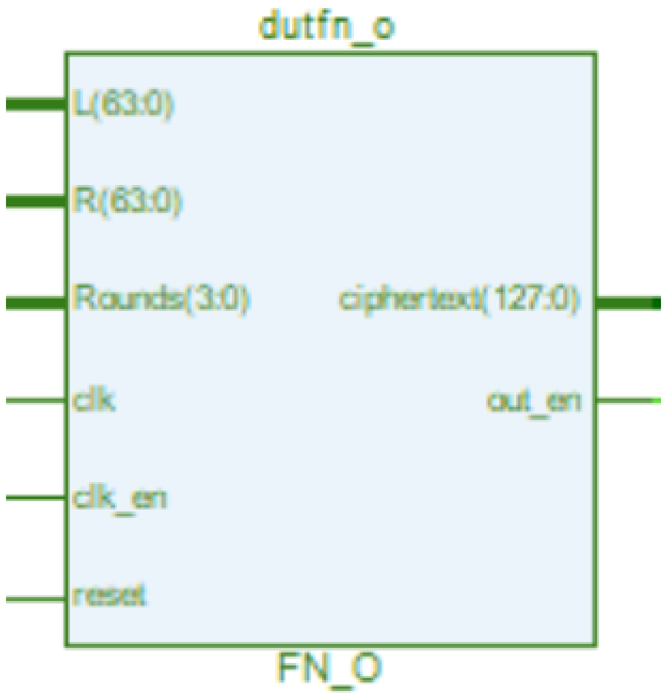
You can find pictures of the tables of the S-boxes and the SS-boxes in the project's archive.

The structure of function G with SS-boxes is shown down below:



Input
32
8
X0
SS0
Z0
8
8
X1
SS1
Z1
8
8
X2
SS2
Z2
8
8
X3
SS3
Z3
8
xor
32
output

3.3.5 FN_O



dutfn_o
L(63:0)
R(63:0)
Rounds(3:0)
clk
clk_en
reset
ciphertext(127:0)
out_en
FN_O

<u>Input Pins:</u>

- Clk – internal 100MHz clock
- Reset - active high synchronous reset
- clk_en – clock enable signal for setup violations
- Rounds - synchronous signal for the registers
- R – LS 64-bit from last round
- L – MS 64-bit from last round

<u>Output Pins:</u>

- ciphertext – 128-bit ciphertext
- out_en – high when encryption over

This module receives the 128-bit ciphertext from every round, and transfer

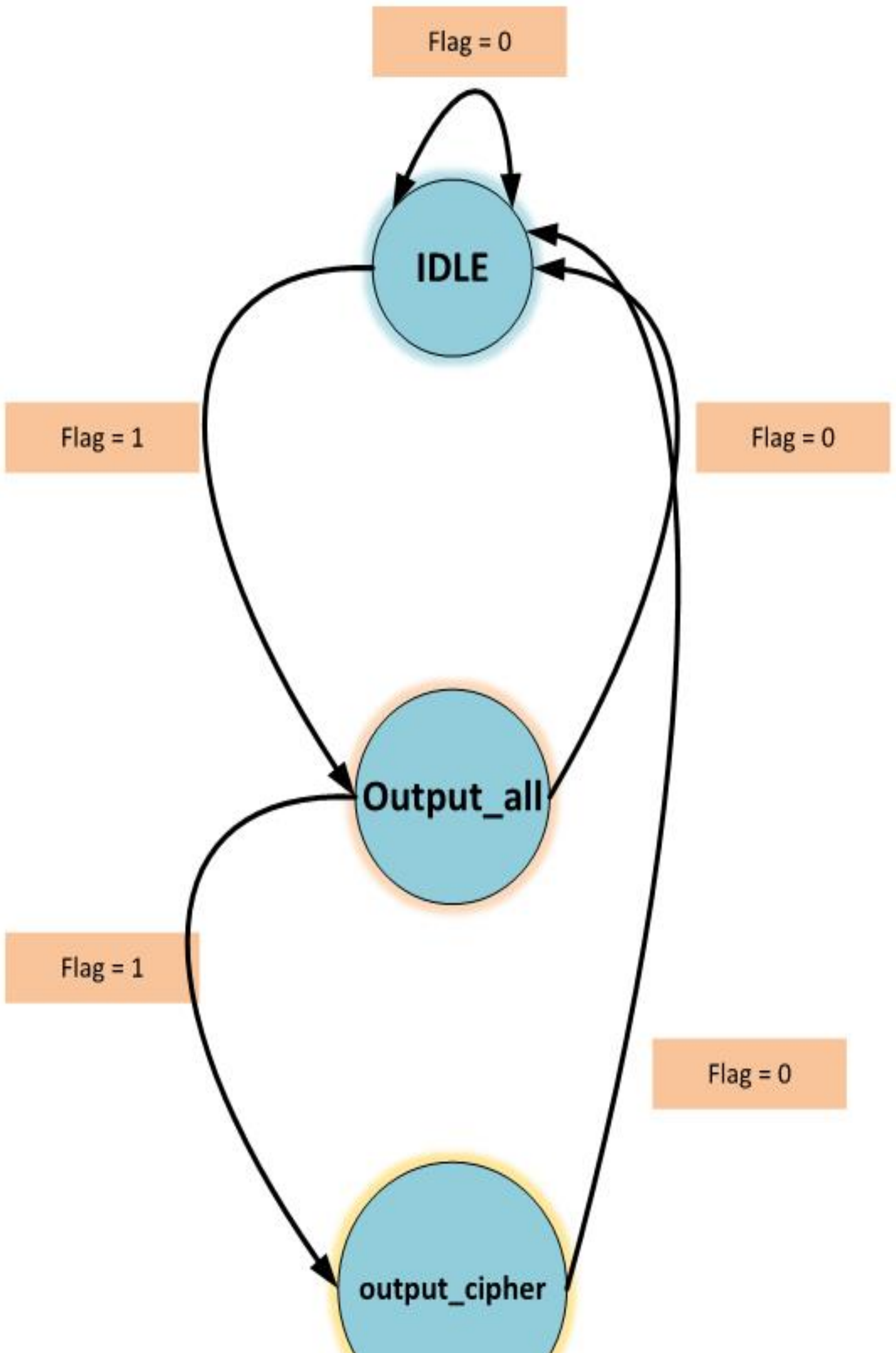
the 128-bit ciphertext exactly after 16 rounds.

<u>Explanation about the states:</u>

Idle – nothing will happen all the outputs are zero.

Output_all – both outputs will transfer, clk_en and 128-bit ciphertext.

Output_cipher – transfer only the 128-bit ciphertext without clk_en pulse.


Flag = 0
IDLE
Flag = 1
Flag = 0
Output_all
Flag = 1
Flag = 0
output_cipher

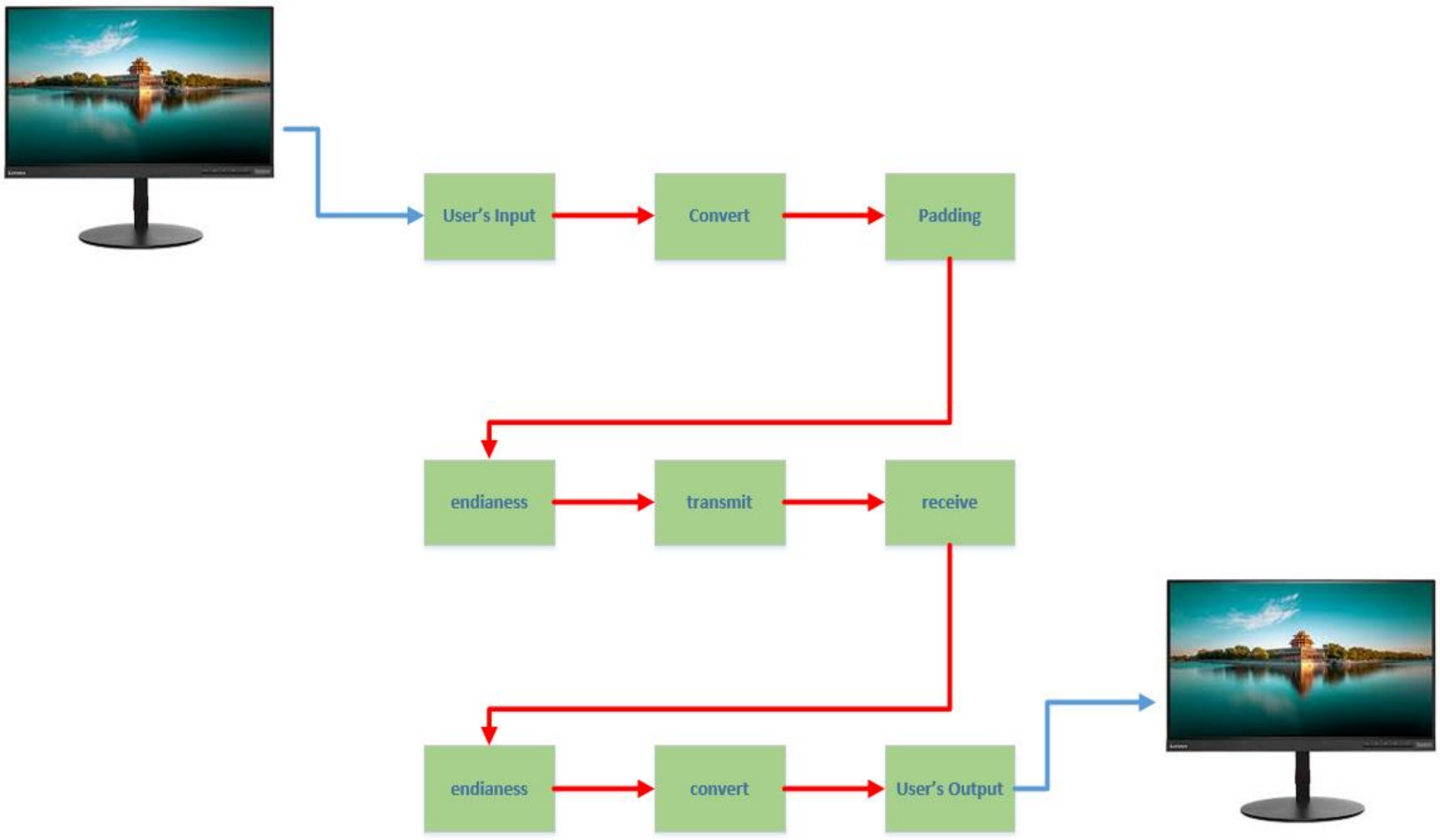
3.4 Raspberry Pi 3

Input Pins:

- load_SEED - indicate to read part_SEED
- part_msg – 8 bits part of message
- done – indicates that the encryption/decryption process is done
- Reset_SEED – synchronize reset

Output Pins:

- Enc_Dec – determines whether encryption or decryption is performed.
- part_SEED – part of SEED Algorithm result
- in_en - synchronous signal for the registers
- start – indicates that the first block was sent
- load – load message



4. Summary and conclusions

4.1 Design Reuse

The software program is in little - endian. It can be executed on any 32-bit or 64-bit processor with a Python compiler. The GPIO designed for Raspberry Pi, but except of this function, the functions were written in a way that they can be reused easily on another processor.

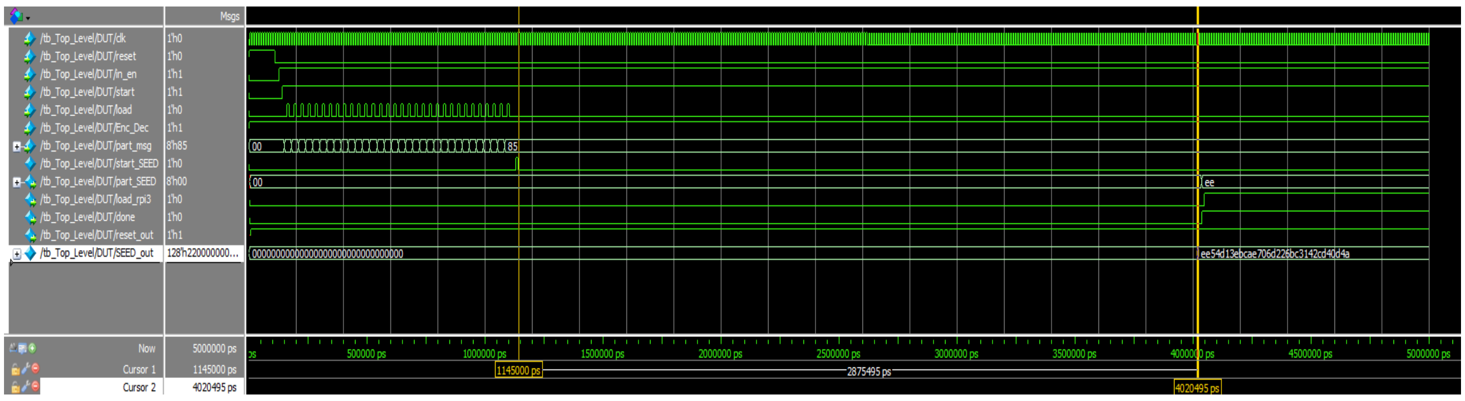
The hardware program is in little – endian. It doesn't depend on any special feature of the Basys 3 Artix-7 FPGA architecture, so it can be implemented in any FPGA device, given enough area and sufficient amount of I/O's.

<u>Note</u>: The raspberry pi 3 GPIO detailed below are BOARD numbers (**not** BCM)

The connections between the Hardware (Basys 3) and Software (RPi 3) are specified on the table below:

Basys 3	Raspberry Pi 3 GPIO	Explanation
pmod JB1 (input)	5 (output)	In enable signal – there are more blocks to send
pmod JB2(input)	7 (output)	Start signal – notify Basys 3 that the first block was sent
pmod JB3 (input)	11 (output)	Load block to Basys 3 signal – pulse when sending 8 bits (part of block)
pmod JB4 (input)	13 (output)	Determines whether encryption or decryption is performed
pmod JB7 (input)	36 (input)	Load to RPi 3 – pulse when sending 8 bits of SEED result
pmod JB8 (input)	40 (input)	High when Basys 3 finished encryption process
pmod JB9 (input)	38 (input)	Big & stable pulse when the reset button was pressed
pmods [JA1, JA2, JA3, JA4, JA7, JA8, JA9, JA10] (output)	[26, 24, 22, 18, 16, 12,10, 8] (input), respectively	Part (8 bits) of SEED result.
pmods [JC1, JC2, JC3, JC4, JC7, JC8, JC9, JC10] (input)	[37, 35, 33, 31, 29, 23, 21, 19] (output), respectively	Part (8 bits) of message to encrypt/decrypt (SEED)

4.2 Timing and Utilization



```
                                   Msgs
/tb_Top_Level/DUT/clk              1'h0
/tb_Top_Level/DUT/reset            1'h0
/tb_Top_Level/DUT/in_en            1'h1
/tb_Top_Level/DUT/start            1'h1
/tb_Top_Level/DUT/load             1'h0
/tb_Top_Level/DUT/Enc_Dec          1'h1
/tb_Top_Level/DUT/part_msg         8'h85
/tb_Top_Level/DUT/start_SEED       1'h0
/tb_Top_Level/DUT/part_SEED        8'h00
/tb_Top_Level/DUT/load_rpi3        1'h0
/tb_Top_Level/DUT/done             1'h0
/tb_Top_Level/DUT/reset_out        1'h1
/tb_Top_Level/DUT/SEED_out         128'h220000000...

Now        5000000 ps
Cursor 1   1145000 ps
Cursor 2   4020495 ps
```

Figure 1. Verilog simulation (Mentor Graphics' ModelSim)

Here you can see the gate level simulation. With frequency of 100MHz:

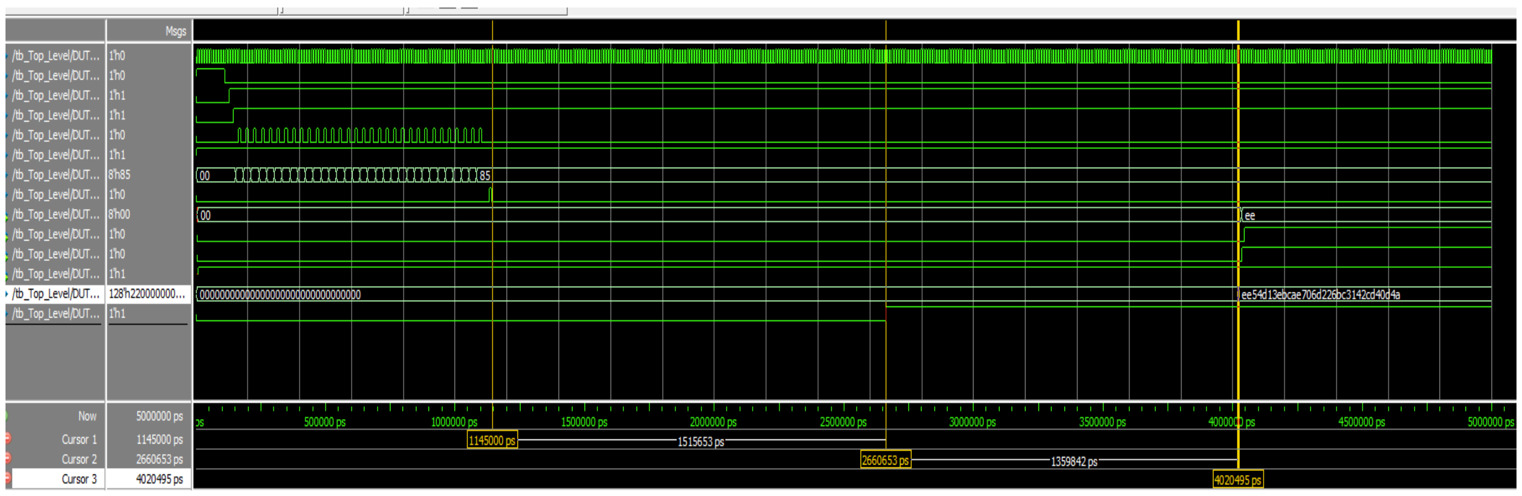
The all algorithm for an empty message in utopian world takes only 287.5 clock – cycles.

Load = 1 clock cycles.

Key schedule = 1505ns = 150.5 clock cycles.

Feistel Network = 1359ns = 136 clock cycles.

SEED algorithm = 2875ns = 287.5 clock cycles.

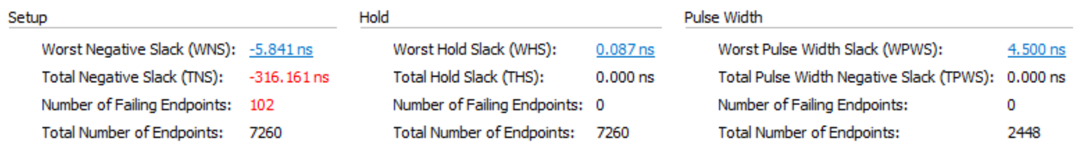


```
                     Msgs
/tb_Top_Level/DUT... 1'h0
/tb_Top_Level/DUT... 1'h0
/tb_Top_Level/DUT... 1'h1
/tb_Top_Level/DUT... 1'h1
/tb_Top_Level/DUT... 1'h0
/tb_Top_Level/DUT... 1'h1
/tb_Top_Level/DUT... 8'h85
/tb_Top_Level/DUT... 1'h0
/tb_Top_Level/DUT... 8'h00
/tb_Top_Level/DUT... 1'h0
/tb_Top_Level/DUT... 1'h0
/tb_Top_Level/DUT... 1'h1
/tb_Top_Level/DUT... 128'h220000000...
/tb_Top_Level/DUT... 1'h1

Now        5000000 ps
Cursor 1   1145000 ps
Cursor 2   2660653 ps
Cursor 3   4020495 ps
```

Figure 2. Verilog simulation (Mentor Graphics' ModelSim)

In figure 2 it's shown that the whole operation took 2875ns, since I use 10ns clock, so the SEED algorithm took 287 clock cycles as we expected.



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-5.841 ns	Worst Hold Slack (WHS):	0.087 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	-316.161 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	102	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7260	Total Number of Endpoints:	7260	Total Number of Endpoints:	2448

Here you can see the timing report by Vivado, as you can see no Hold violation.

All the critical paths with Setup violations are multicycle paths – two clock for each path by the signal clock enable, so in my project there is no Setup violations.

The minimum slack is 4.159ns. (20ns – 15.841ns) = 4.159ns

I used Vivado 2018.3 to implement this project on an Artix-7 FPGA device with enough I/O's (XC7A35TCPG236-1)

<u>Utilization Graph:</u>

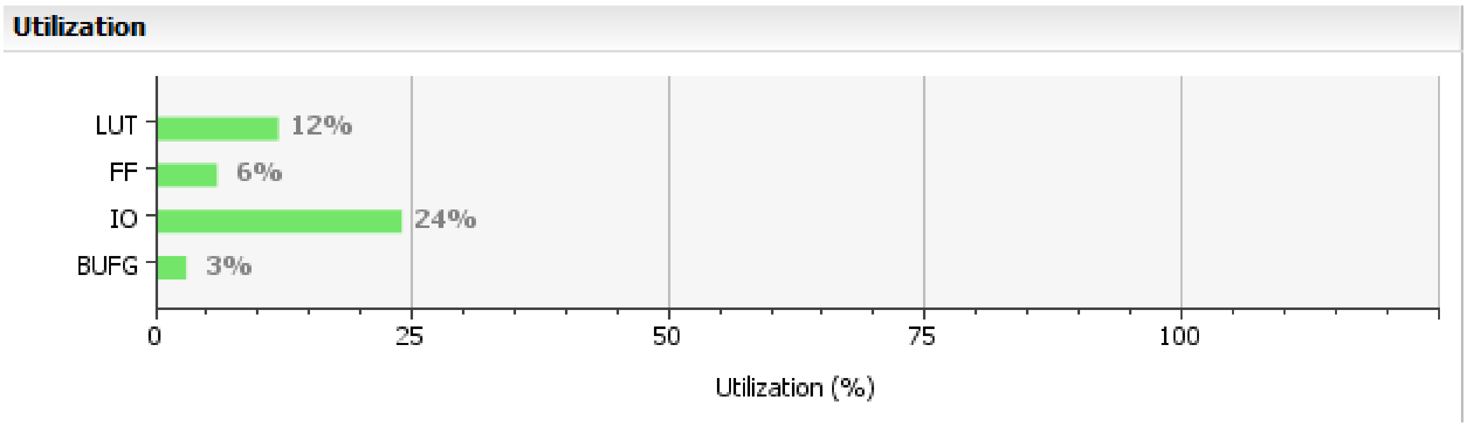
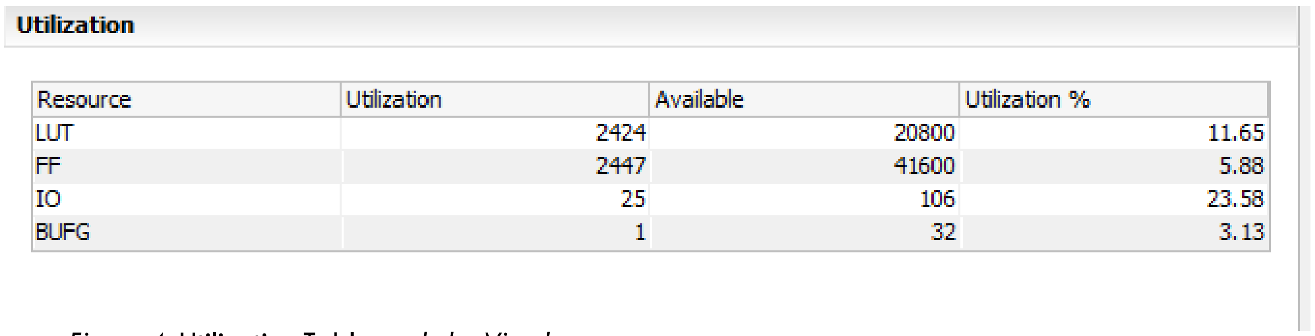


Figure 3. Utilization Graph made by Vivado

Utilization Table:



Utilization

Resource	Utilization	Available	Utilization %
LUT	2424	20800	11.65
FF	2447	41600	5.88
IO	25	106	23.58
BUFG	1	32	3.13

Figure 4. Utilization Table *made by Vivado*

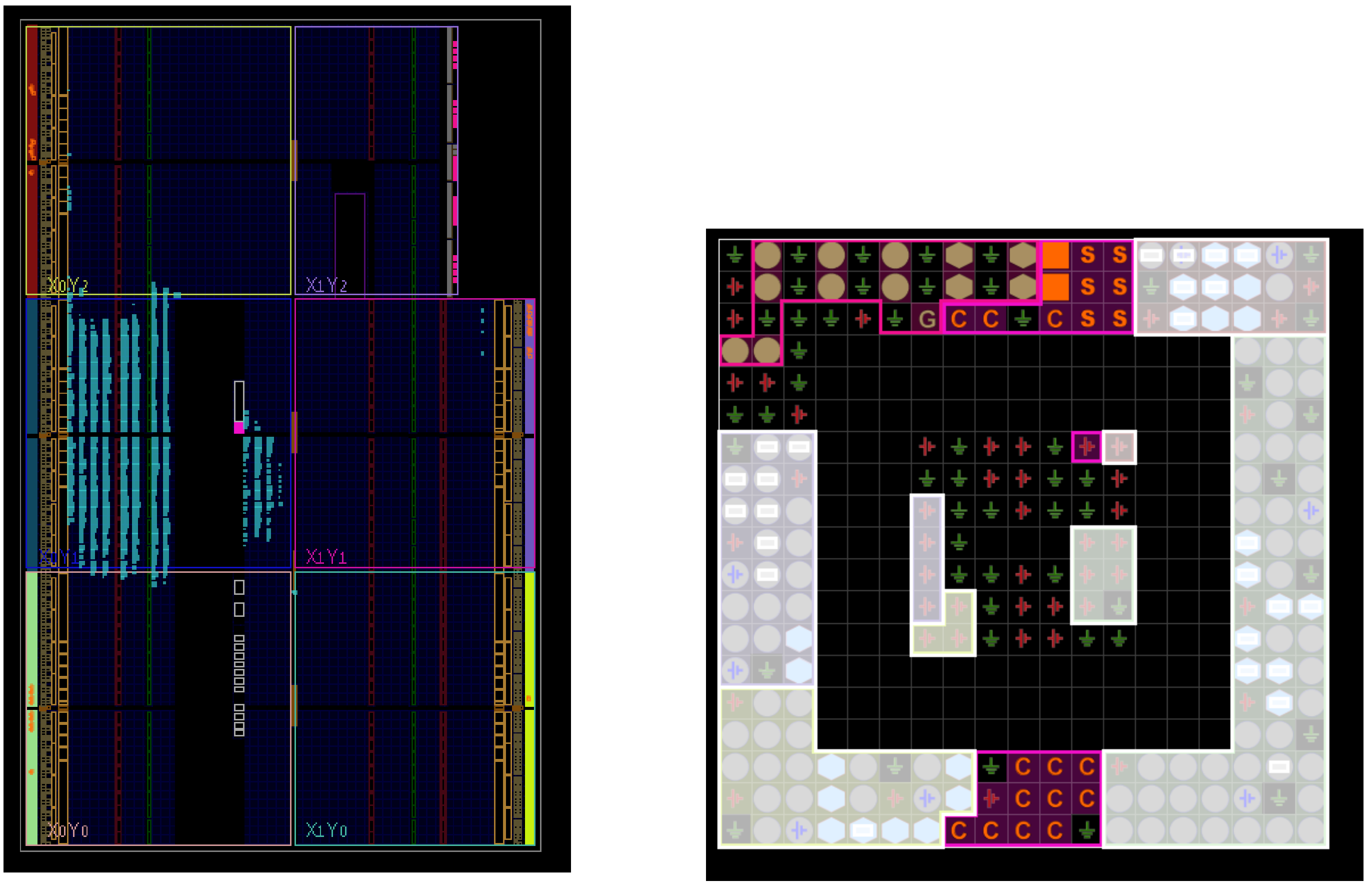


Figure 5. P&R and I/O planning made by Vivado

In figure 3&4 we can see that this implementation uses a small percentage and small amount of FFs and LUTs.

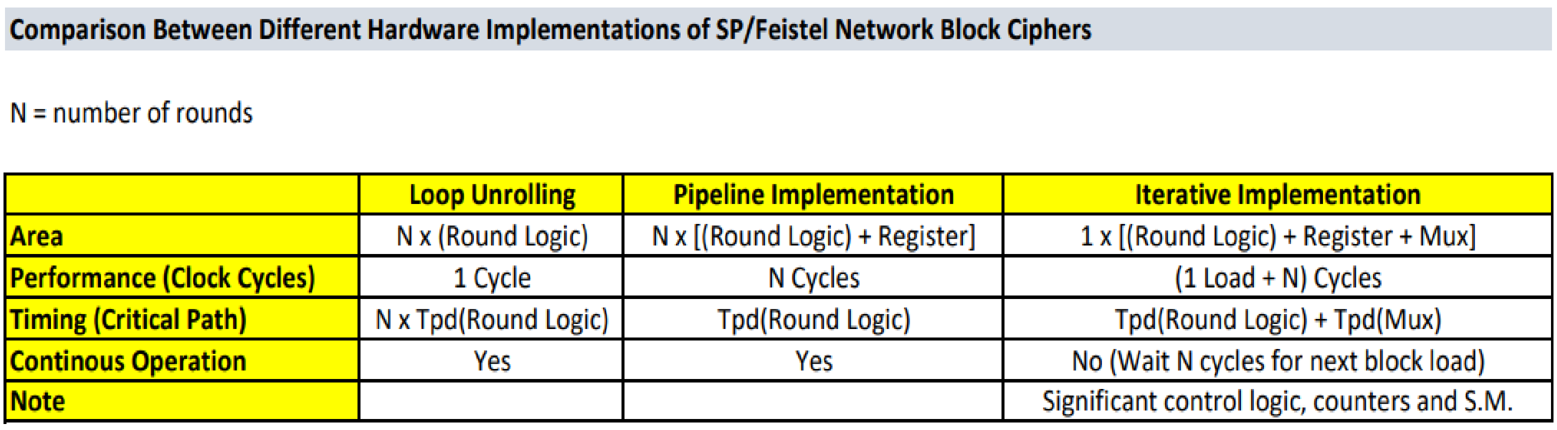
In figure 5 we can see the way Vivado arranged the optimization design and the I/O that my chosen artix-7 has the I/O that I use.
According to the timing report provided by Vivado, the minimum slack between two flip-flops is 4.159 ns. I used internal Basys 3 clock (100 MHz). It means that my maximum frequency for the design is 170 MHz.
As shown and explained, my project uses low area with large frequency clock.

4.3 Trade Off and Throughput

Trade Off:

The main challenge while implementing the design was building accurate and robust modules, which implement the algorithm without wasting area, and without wasting any clock cycles within the main algorithm module SEED or while transferring signals between Basys 3 and RPi3.

Therefore, I choose Iterative Implementation.



Comparison Between Different Hardware Implementations of SP/Feistel Network Block Ciphers

N = number of rounds

	Loop Unrolling	Pipeline Implementation	Iterative Implementation
Area	N x (Round Logic)	N x [(Round Logic) + Register]	1 x [(Round Logic) + Register + Mux]
Performance (Clock Cycles)	1 Cycle	N Cycles	(1 Load + N) Cycles
Timing (Critical Path)	N x Tpd(Round Logic)	Tpd(Round Logic)	Tpd(Round Logic) + Tpd(Mux)
Continous Operation	Yes	Yes	No (Wait N cycles for next block load)
Note			Significant control logic, counters and S.M.

Here are some advantages using Iterative Implementation:

- Using a little amount of the FPGA area.
- Can work at high frequencies.
- Small change of critical Path.

Throughput:

In this project I used the 100MHz internal clock.

The throughput when you need to create the subkeys is:

128/ ((287.5) * 10ns) = 44.5Mbps

But when you don't need to create the subkeys – means you are using the same key

Or when implement a design with only encryption mode so your key can execute parallel with you Feistel Network you can get throughput equal to

128/ ((138) * 10ns) = 92.75Mbps

4.4 Summary

Every aspect of this project required a very steep learning curve on my part. As one of my goals was to enter the world of FPGA development without any prior knowledge or experience, some extensive self-education was in order. Luckily, in our day and age, the internet provides more than enough resources on every possible topic, and Xilinx itself has an abundance of user manuals, workshops, and tutorials (in a sometimes-overwhelming quantity), to guide me.

Before diving into any relevant work, I had to first learn VERILOG, understand the work relations between the Basys 3 and Raspberry Pi 3, the workflow and structure of Vivado, and how to use the tools they provide. I learned about creating my own IPs, and gradually became more comfortable with the tools at my disposal.

On the theoretical side, I learned about Cryptography, and found a vast new world with a large community, and countless academic papers on every imaginable topic – from various algorithms, to implementation methods (which are in a constant state of development and improvement) and more.

Due to time constraints (in the real-world, not on the FPGA), the steep learning curve and a lot of technical difficulties on the way, I weren't able to venture very far into this interesting world, but I hope to continue the development under more relaxed circumstances in the future.

Overall, I have succeeded to implement the algorithm in a simple elegant way and create a relatively-low-area design that can fit into many FPGA devices and enable use of encryption operation even in limited resources environments. That was done while "keeping an eye" on the design timing and achieving a fair clock speed.

5. References

[1] Basys3 FPGA Board Reference Manual. Available:

https://reference.digilentinc.com/_media/basys3:basys3_rm.pdf

[2] Verilog Operations. Available:

http://verilog.renerta.com/source/vrg00031.htm

[3] Raspberry Pi 3 Pinout. Available:

https://pinout.xyz/pinout/i2c

[4] Military Technical Academy. Cryptographic Applications using FPGA Technology. Available:

https://pdfs.semanticscholar.org/b6d1/f5582c38dac0fe94df734ec58a81bb8ef3ea.pdf

[5] School of Electronics Engineering, VIT University, Vellore, Tamil Nadu 632014, India. Encrypting and decryption using FPGA. Available:

https://iopscience.iop.org/article/10.1088/1757-899X/263/5/052030/pdf

[6] raspberry Pi Technical Specification. Available:

https://www.terraelectronica.ru/pdf/show?pdf_file=%252Fds%252Fpdf%252FT%252FTechicRP3.pdf

[7] SEED 128 Algorithm Specification. Available:

https://webcache.googleusercontent.com/search?q=cache:0_P0trY9fDMJ:https://reinliebe.tistory.com/attachment/cfile9.uf%40133B0E134BE61D4A70E53F.pdf+&cd=4&hl=iw&ct=clnk&gl=il

[8] Network Working Group, Request for Comments: 4269. Available:
https://tools.ietf.org/html/rfc4269

[9] Computer Science, Hellenic Open University, Greece. *Paris Kitsos and Athanassios N. Skodras.* **AN FPGA IMPLEMENTATION AND PERFORMANCE EVALUATION OF The SEED BLOCK CIPHER**
https://www.researchgate.net/publication/261380649_An_FPGA_implementation_and_performance_evaluation_of_the_seed_block_cipher