JERUSALEM COLLEGE OF TECHNOLOGY

XILINX ALL PROGRAMMABLE™

Xilinx Open Hardware 2021 design contest

Implementation of Hummingbird Encryption Algorithm

Elie Cudkowicz & Itamar Yunnian

Team number:

xohw21-114

Supervisor: Uri Stroh

Link to project video: https://youtu.be/mu5e4rd4wVE

Table of Contents

Table of Figures

Table of Tables

1. Introduction

1.1. Hummingbird-2 Algorithm[1]

Hummingbird-2 is an encryption algorithm with a 128 bits secret key and a 64 bits initialization vector. Like its predecessor Hummingbird-1, Hummingbird-2 has been targeted for low-end microcontrollers and for hardware implementation in lightweight devices such as RFID tags and wireless sensors and can be implemented with vey small hardware or software footprint. Therefore, it's suitable for providing security in low cost devices.

In this report, we will describe the whole implementation of the algorithm into an FPGA and we will propose a possibility to send and read all the required data (plaintext, key, initialization vector, cyphertext) to the chip using only several input and output pins.

1.2. Platform and Tools

We used the Basys3 board, a ready-to-use digital circuit, based on the Artix-7 FPGA (XC7A35T1CPG236C), and use Vivado 2020.2 to synthesis and implement the algorithm.

To implement Hummingbird-2 algorithm, we may need a total of 448 I/Os, to be able to send all the data (Plaintext = 128 bits, Key = 128 bits, IV = 64 bits) and to read the result of the encryption (Cyphertext = 128). Of course, this high number of ports is not available in Basys3 board, so we decide to use Arduino Nano based on ATmega328P chip to send the data to the board and read the result of the algorithm from it.

The algorithm code is written in Verilog Hardware Language and the Arduino code is written in C language.

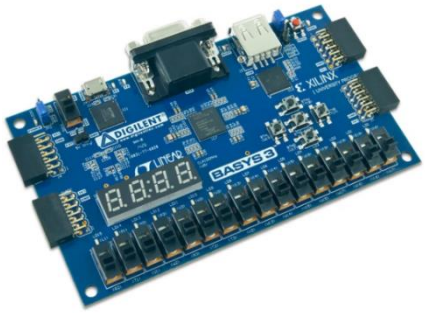


Figure 1 – Basys3 Board

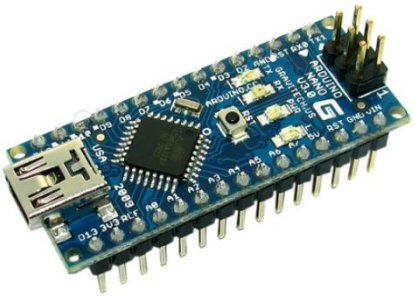


Figure 2 – Arduino Nano ATmega 328P

1.3. Project Goals

In this project, we wanted to experiment with the concept of implementing an encryption algorithm on a FPGA.

Our main goals are:

- Get familiar with Cryptography world.
- Get familiar with Verilog language, and be able to build an algorithm with it.
- Get to implement a design made all by ourselves into an FPGA.
- Get to build a network protocol between 2 different clocks domains and use it to send large data to a board with limited I/Os.

2. Hummingbird-2 Description[1]

The Hummingbird-2 does not directly fall to either traditional stream cipher or block cipher categories as it inherits properties from both.

The algorithm is built of 4 different blocks:

1. Mixing function.
2. WD16.
3. Initialization.
4. Encryption.

We will describe each of those blocks and how did we design them.

2.1. Mixing Function

The nonlinear mixing-function $f(x)$ consists of 4-bit S-Box permutation lookups on each nibble of the word (word = 16 bits) , followed by a linear mix.

The word (DATA_IN) is separated in 4 sub-words of 4 bits. Each one of those 4 bits get in one of the S-Box and get out according to the S-Box table presented in figure 4. Once the 4 outputs of the S-Boxes are concatenate back to one word $(S(x))$ a linear transformation $(L(x))$, based on left circular shift operation (<<), is made on it.

$$S(x) = \{S_1(x), S_2(x), S_3(x), S_4(x)\}$$

$$L(x) = x\ xor\ (x \lll 6)\ xor\ (x \lll 10)$$

$$f(x) = L(S(x))$$

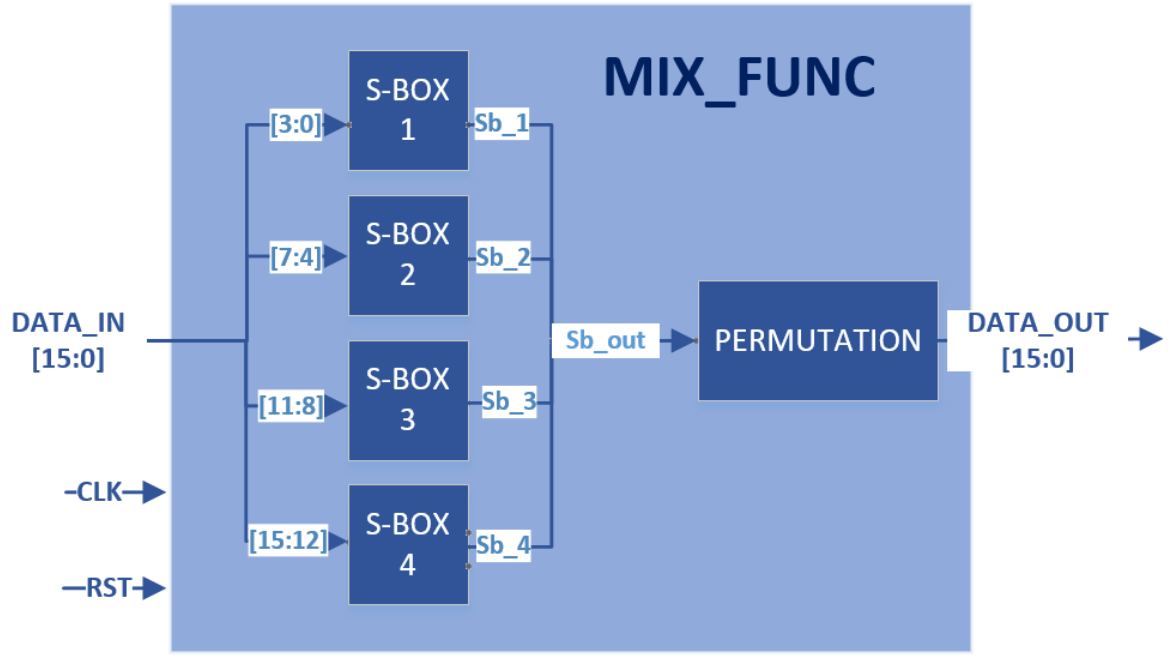
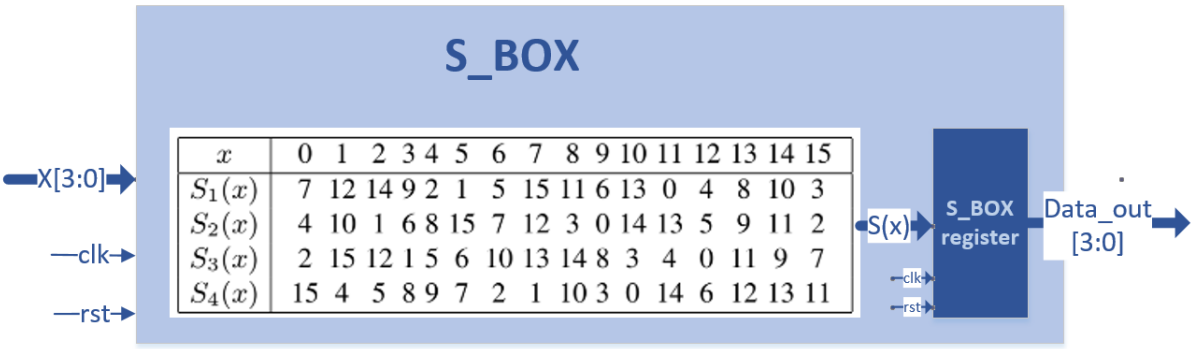


Figure 3 – Mixing Function Block Diagram



x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_1(x)$	7	12	14	9	2	1	5	15	11	6	13	0	4	8	10	3
$S_2(x)$	4	10	1	6	8	15	7	12	3	0	14	13	5	9	11	2
$S_3(x)$	2	15	12	1	5	6	10	13	14	8	3	4	0	11	9	7
$S_4(x)$	15	4	5	8	9	7	2	1	10	3	0	14	6	12	13	11

Figure 4 – S-Box Block Diagram

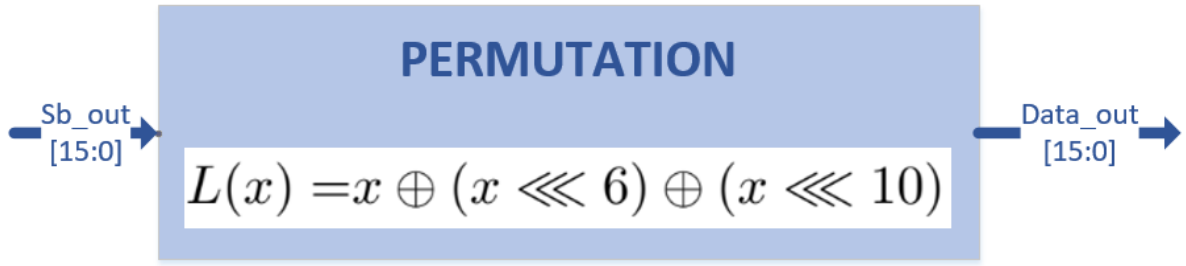


Figure 5 – Permutation Block Diagram

2.2. WD16

We further design a 16-bits keyed permutation using the mixing function as:

$$WD16(x, a, b, c, d) = f(f(f(f(x\ xor\ a)\ xor\ b)\ xor\ c)\ xor\ d)$$

The word to be encrypted (DATA_IN) is keyed by a sub-key (key_1) and then go over the mixing-function. The output of the first mixing -function is keyed by another sub-key (key_2) and go through the same function. This process is done 4 times with four different sub-keys.

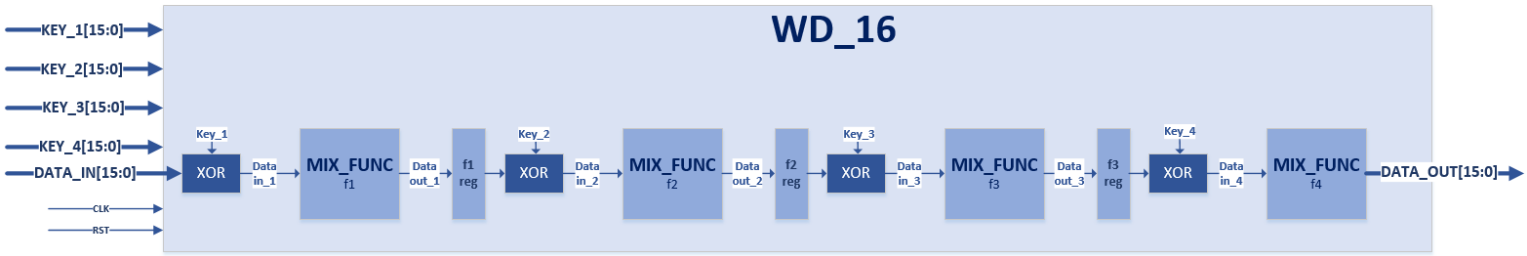


Figure 6 – WD16 Block Diagram

2.3. Initialization

The internal state of Hummingbird-2 is initialized with a four-round procedure using the 64-bits nonce IV. We first set:

$$R^0 = (IV_1, IV_2, IV_3, IV_4, IV_1, IV_2, IV_3, IV_4)$$

And then iterate for $i = 0, 1, 2, 3$ the following:

$$t_1 = WD16\left(R_1^i + i, Key_1, Key_2, Key_3, Key_4\right)$$

$$t_2 = WD16(R_2^i + t_1, Key_5, Key_6, Key_7, Key_8)$$

$$t_3 = WD16(R_3^i + t_2, Key_1, Key_2, Key_3, Key_4)$$

$$t_4 = WD16(R_4^i + t_3, Key_5, Key_6, Key_7, Key_8)$$

$$R_1^{i+1} = \left(R_1^i + t_4\right) \ll 3$$

$$R_2^{i+1} = \left(R_2^i + t_1\right) \gg 1$$

$$R_3^{i+1} = \left(R_3^i + t_2\right) \ll 8$$

$$R_4^{i+1} = \left(R_4^i + t_3\right) \ll 1$$

$$R_5^{i+1} = \left(R_5^i \; xor \; R_1^{i+1}\right)$$

$$R_6^{i+1} = \left(R_6^i \; xor \; R_2^{i+1}\right)$$

$$R_7^{i+1} = \left(R_7^i \; xor \; R_3^{i+1}\right)$$

$$R_8^{i+1} = \left(R_8^i \; xor \; R_4^{i+1}\right)$$

Note that the "+" operator in above expressions is addition modulo 65536 and that the "<</>>" operator is left/right is circular shift.

Init block get eight 16-bits values (r1…r8) and, after 4 WD16 functions, get out new values for those 8 registers. Initialization of those four registers is done by 4 rounds of INIT while the first value to get in is the 64-bits IV.

The initial state for encrypting the first plaintext word is the output of the fourth round R^4.

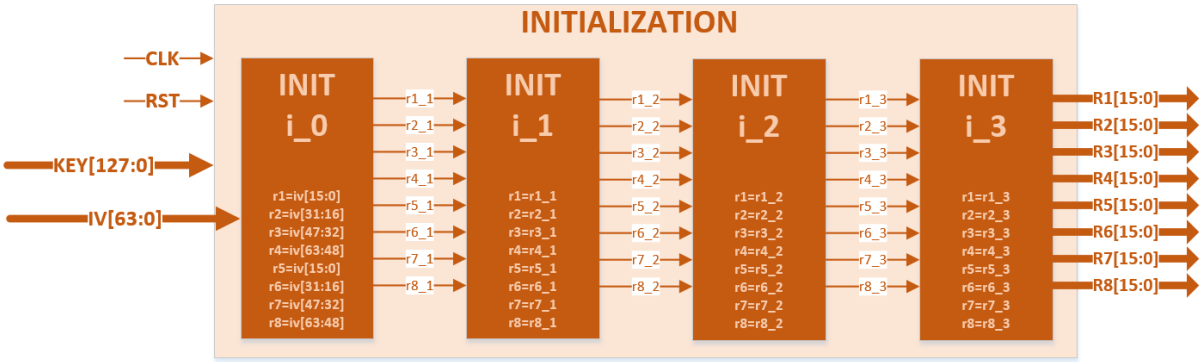


Figure 7 – Initialization Block Diagram

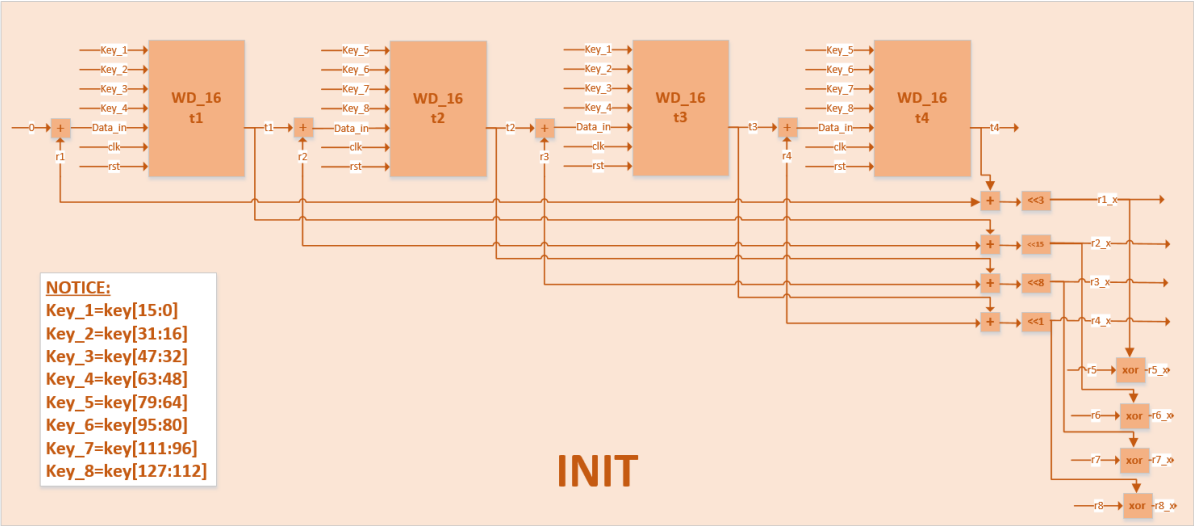


Figure 8 – Init Block Diagram

2.4. Encryption

Encryption of a single word of plaintext P_i to ciphertext word C_i requires four invocations of WD16.

$$t_1 = WD16(R_1^i + P_i, Key_1, Key_2, Key_3, Key_4)$$

$$t_2 = WD16(R_2^i + t_1, Key_5\ xor\ R_5^i, Key_6\ xor\ R_6^i, Key_7\ xor\ R_7^i, Key_8\ xor\ R_8^i)$$

$$t_3 = WD16(R_3^i + t_2, Key_1\ xor\ R_5^i, Key_2\ xor\ R_6^i, Key_3\ xor\ R_7^i, Key_4\ xor\ R_8^i)$$

$$C_i = WD16\left(R_4^i + t_3, Key_5, Key_6, Key_7, Key_8\right) + R_1^i$$

After each encrypted word, we perform the following state update:

$$R_1^{i+1} = R_1^i + t_3$$

$$R_2^{i+1} = R_2^i + t_1$$

$$R_3^{i+1} = R_3^i + t_2$$

$$R_4^{i+1} = R_4^i + R_1^i + t_3 + t_1$$

$$R_5^{i+1} = R_5^i\ xor\ (R_1^i + t_3)$$

$$R_6^{i+1} = R_6^i\ xor\ (R_2^i + t_1)$$

$$R_7^{i+1} = R_7^i\ xor\ (R_3^i + t_2$$

$$R_8^{i+1} = R_8^i\ xor\ (R_4^i + R_1^i + t_3 + t_1)$$

To encrypt a single word (16 bits), the plaintext goes through WD16 functions 4 times. At each time, the data is added to one of the internal state registers (r1…r4). After the fourth round, the word is encrypted (ciphertext) and the 8 internal registers are updated to encrypt the next word.

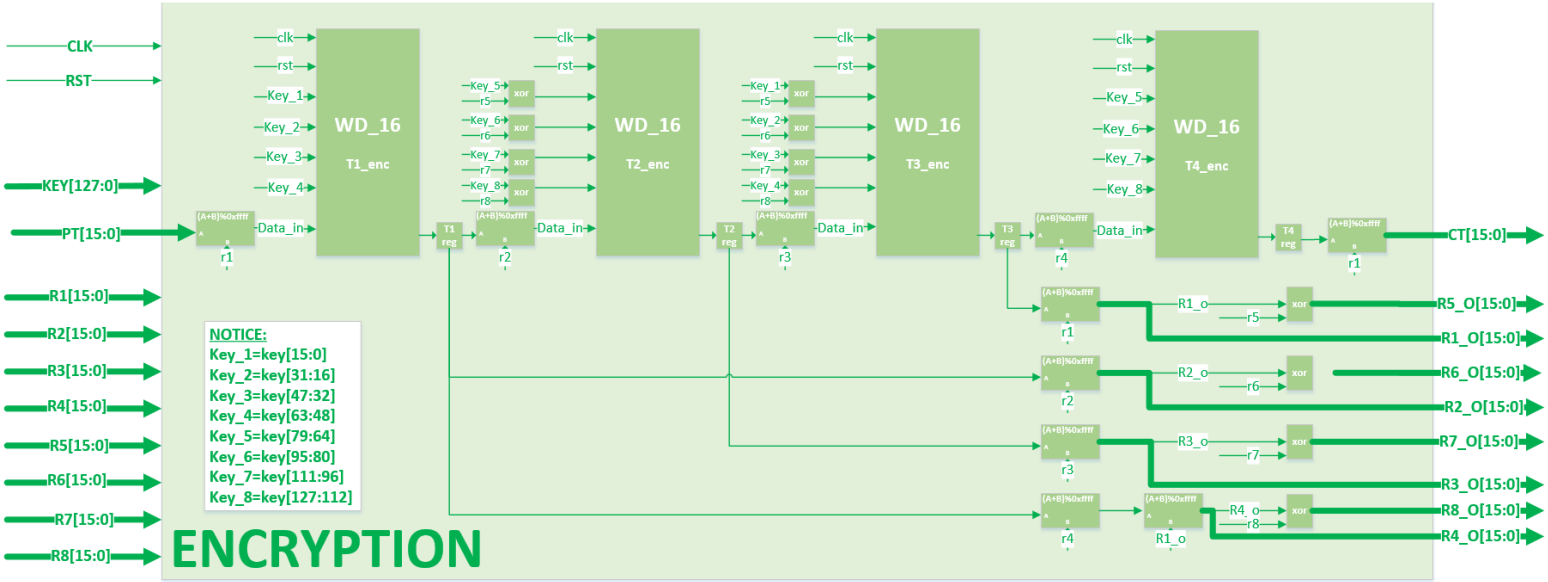


Figure 9 – Encryption Block Diagram

3. Top Level

3.1. Hummingbird Top Level

To encrypt a whole plaintext (128-bits), we first need to initialize the 8 internal registers according to the Initialize Vector (64-bits). Once, internal states are updated, we divide the plaintext into several words of 16-bits (8 words if the plaintext is 128-bits) and each of them is encrypted according to the current internal states and the Key (128-bits). After each encryption, we get a 16-bits part of the ciphertext and can move on to the next word with the new internal states. At the end of the eight round we will have the 128-bits ciphertext.

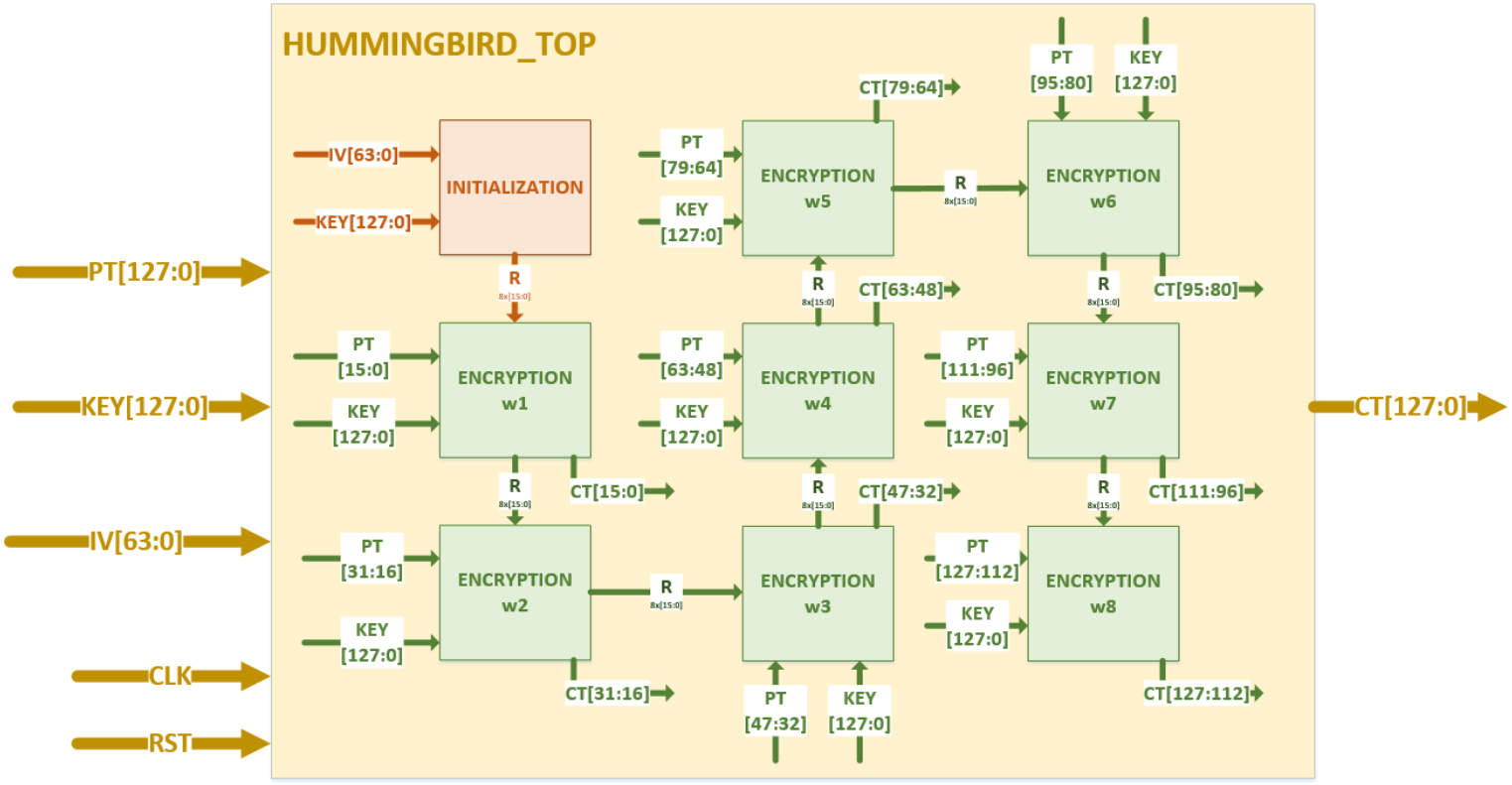


Figure 10 – Hummingbird Top Block Diagram

3.2. Hummingbird Top Wrapper

In the Basys3 board, where we implement our design, only a few number of ports are available such that we can't input all the required data (plaintext, key, IV) to the algorithm and we can't read the result (ciphertext) from it.

In order to solve that problem, we design a Top-Level Wrapper. This Wrapper will have a single bit data input and a single bit data output. An Arduino Nano will send all the data serially, and it will be saved in 3 registers: PT (128-bits) for the plaintext, KEY (128-bits) for the key and IV (64-bits) for the initialize vector. In the same way, at the end of the encryption, the ciphertext will be saved also in a register CT (128-bits) and will be send back to the Arduino Nano bit by bit.

Note – since the Arduino and the Basys3 board are in 2 different clocks domains, we had to build a handshake protocol between them to pass the data correctly. We will describe this protocol in the next section.

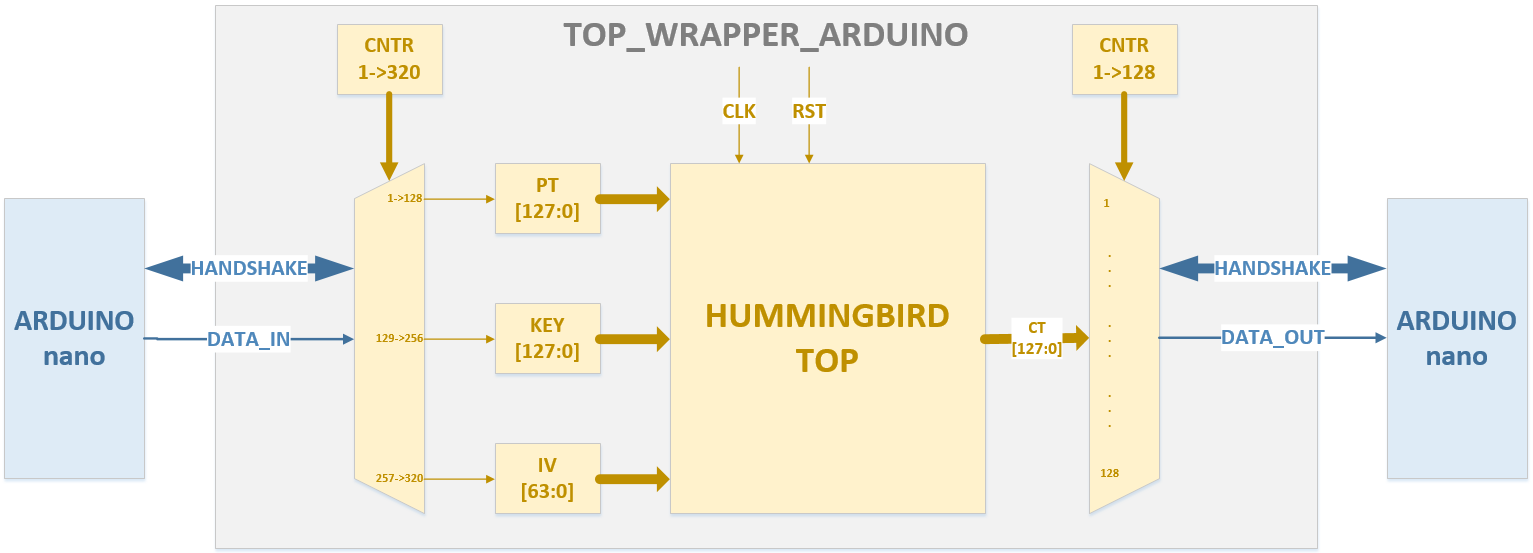


Figure 11 – Top Wrapper Arduino Block Diagram

3.3. Handshake Protocol Between Arduino and Basys3 Board

The Basys3 Board with the Artix-7 chip in it, works with a single clock of 100MHz. In the other hand, the Arduino Nano we used works with a clock of 16Mhz only.

To send data from the Arduino to the Basys3 and from the Basys3 back to the Arduino, we need to build a handshake protocol between them.

When the Arduino has data to send to the Basys3, it needs to send a request first to check that the Basys3 is ready to get the data. If the Basys3 is ready (it's not dealing with precedent data), it can give acknowledge to the Arduino and start to read and store the data. Once Basys3 will finish to deal with the current data, it will low his acknowledge signal, so the Arduino will know that it can request to send the next data.

We noticed that data sent from Arduino to Basys3 can be unstable. This noise can cause trouble in interpreting the request and acknowledge signals. To solve this problem, we add a counter to each of those signals (request and acknowledge coming from Arduino) that count the number of cycles when the signal is high/low. Only if the signal is stable for 10 clock cycles, we will consider it as high/low.

The protocol to send data from the Basys3 to the Arduino is similar that the protocol we described in the above lines.

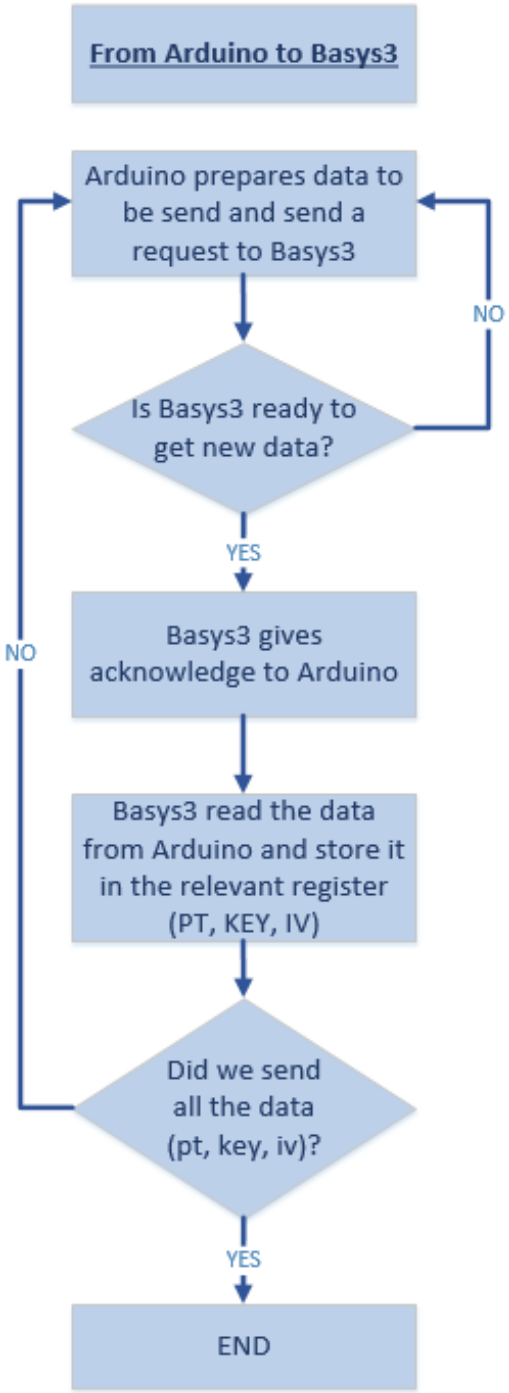


Figure 13 – Handshake Flow from Arduino to Basys3

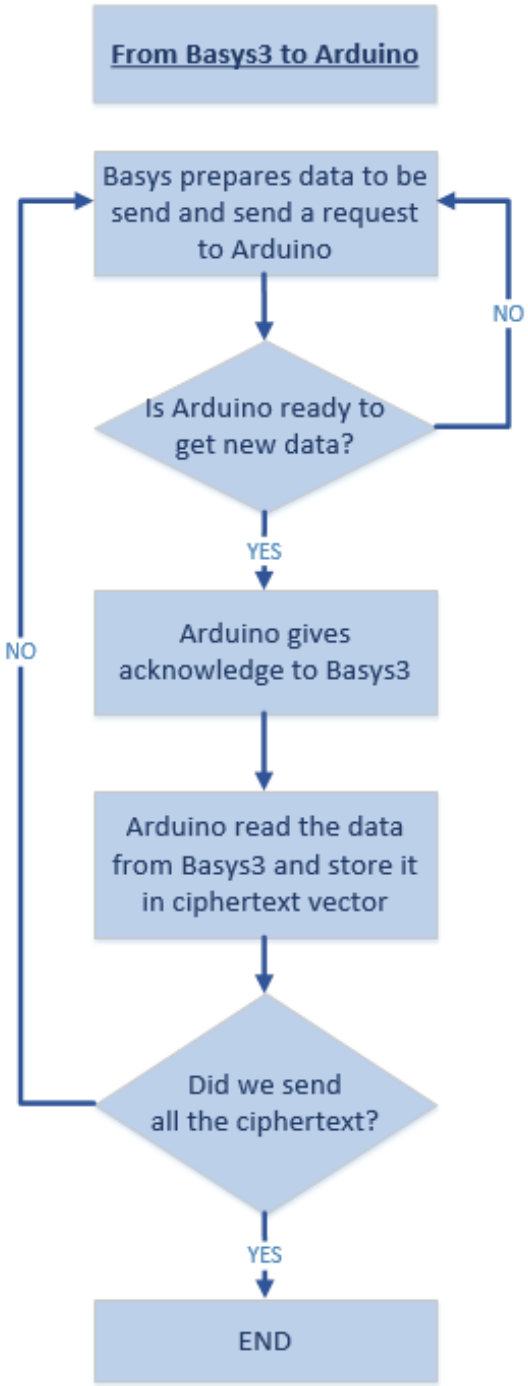


Figure 12 – Handshake Flow from Basys3 to Arduino

4. Implementation

4.1. Implementation on Basys3 Board

We implement our algorithm on Basys3 Board, a ready-to-use digital circuit development platform based on Artix-7 FPGA (Xilinx part number: XC7A35T1CPG236C), with the help of Vivado tool.

A lot of components and features are available on the board, we used the follows:

- Pmod ports – we used Pmod JA to connect between the Basys3 Board and the Arduino.
- 16 LEDs – We used the LEDs on the board to be able to see the data received from the Arduino (plaintext, key, iv) and to see the data send to the Arduino (ciphertext).
- 16 Switch-buttons – We used the Switch-buttons to choose which data to display on LEDs.
- Push-Button – We uses one push-button to reset the design.

4.2. Constraints and connection between Basys3 and Arduino

Table 1 – Constraints of Top Wrapper Arduino on Basys3

Signals (from top_wrapper_arduino)	**Connected to Basys3 pin**	**Description**
CLK	W5	100MHz clock
RST	U18	Active high Synchronous reset
REQ_ARD	J1	Request from Arduino to Basys3 to send data (pt, key, iv)
ACK_BAS	L2	Acknowledge from Basys3 to Arduino to receive data
DATA_IN	J2	1-bit data send from Arduino to Basys3
REQ_BAS	G2	Request from Basys3 to arduino to send data (ct)
ACK_ARD	H1	Acknowledge from Arduino to Basys3 to receive data
CT_OUT	K2	1-bit data send from Basys3 to Arduino
CT_VALID	H2	Signals sent from Arduino to Basys3, reports that the ciphertext is valid and can be send back from Basys3 to arduino.
DATA_TO_LED [16]	U16,E19,U19,V19, W18,U15,U14,V14, V13,V3,W3,U3, P3,N3,P1,L1	Possibility to display all the data (pt, key, iv, ct) on basys3 LED according to Switch-Buttons LED_OUT
LED_OUT [8]	R2,T1,U1,W2, R3,T2,T3,V2	8 first left Switch-Buttons to display the data on LED. *See Debug section to check which Switch-Button can display which data.*

4.3. Design Area

As we described in the introduction, Hummingbird-2 is targeted for low-end microcontrollers and for hardware implementation in lightweight devices. To meet this request, Hummingbird-2 needs to be able to fit on small chip like Artix -7 and to be implemented with very small hardware or software footprint

As we can see from Vivado Implementation results, Hummingbird-2 utilizes less than a half of the free resource of the chip.

Resource	Utilization	Available	Utilization %
LUT	9192	20800	44.19
FF	10687	41600	25.69
IO	33	106	31.13

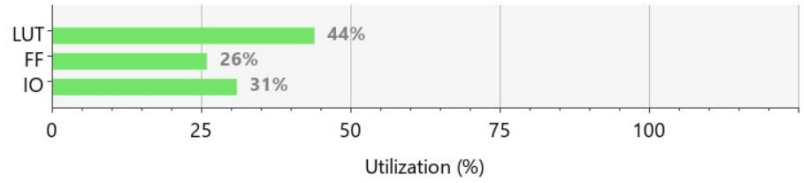


Figure 14 – Vivado Result: Utilization of Resources on the chip

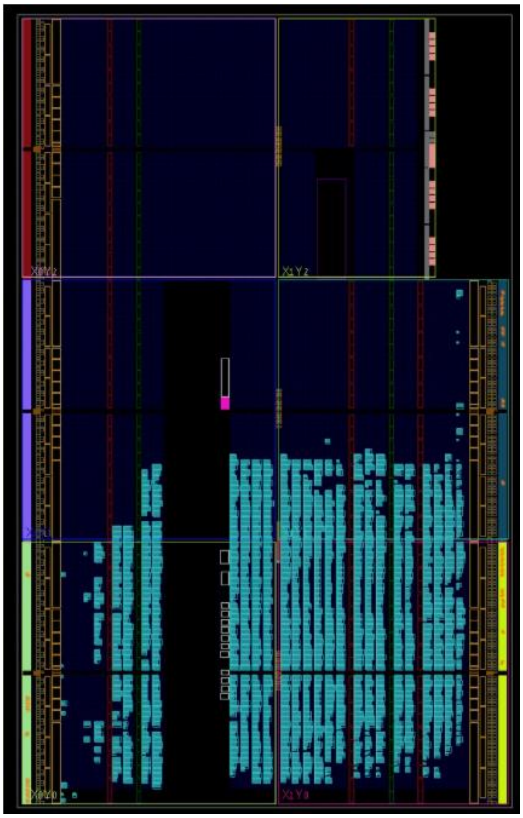


Figure 15 – Vivado Result: Chip Overview after implementation

4.4. Debug Feature

We used the 16 LEDs on the Basys3 Board and the Switch-Buttons to help us debugging our design after implementation.

Since the data used by the algorithm is sent by Arduino Nano bit by bit and need to be store in registers on the chip, we wanted to insert a possibility to check that the data was both send and store correctly. In addition, we also wanted to be able to check that the result of the encryption (the ciphertext) is correct and can be send back to the Arduino.

Using the 8 left Switch-buttons on the board, as described in the table below, we were able to display on the LEDs of the board all the required data.

Table 2 – Switch-Buttons to display data for debug

Led_out value (Switch-Button)	**Data displayed**
10000000	Pt[127:112]
11000000	Pt[111:96]
11100000	Pt[95:80]
11110000	Pt[79:64]
11111000	Pt[63:48]
11111100	Pt[47:32]
11111110	Pt[31:16]
11111111	Pt[15:0]
01111111	Key[127:112]
00111111	Key[111:96]
00011111	Key[95:80]
00001111	Key[79:64]
00000111	Key[63:48]
00000011	Key[47:32]
00000001	Key[31:16]
00000000	Key[15:0]
10100000	Iv[63:48]
10110000	Iv[47:32]
10111000	Iv[31:16]
10111100	Iv[15:0]
10010000	ct[127:112]
10011000	ct[111:96]
10011100	ct[95:80]
10011110	ct[79:64]
10011111	ct[63:48]
10001000	ct[47:32]
10001100	ct[31:16]
10001110	ct[15:0]

5. Test Vectors

In order to check the correctness of our design we used 2 test vectors that has been proved on Hummingbird-2 algorithm.

Table 3 – Test Vector

Test Vector 1	Plaintext	0x 00000000000000000000000000000000
	Key	0x 00000000000000000000000000000000
	IV/Nonce	0x 0000000000000000
	Ciphertext	0x EFC4A887054F91A946578144256ECF3A
Test Vector 2	Plaintext	0x 1100332255447766998BBAADDCCFFEE
	Key	0x 23016745AB89EFCDDCFE98BA54761032
	IV/Nonce	0x 34127856BC9AF0DE
	Ciphertext	0x D15BADF81423F420B1BAC2542945383D

The test vectors have been taken from "The Hummingbird-2 Lightweight Authenticated Encryption algorithm" essay attached and the end of this report.[1]

Note: In the above essay, the test vectors are reported in another order, but since the data is processed in little-endian fashion, we report them in the same way we used them during simulation and implementation.

5.1. Test vector 1

We build a Testbench and send the Test Vector 1 through our algorithm of Hummingbird-2. After running a simulation with ModelSim SE-64 we got the following result:

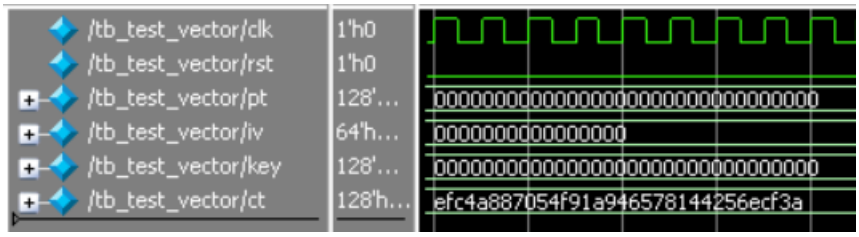


Figure 16 – Simulation result of Test Vector 1

5.2. Test vector 2

We build a Testbench and send the Test Vector 2 through our algorithm of Hummingbird-2. After running a simulation with ModelSim SE-64 we got the following result:

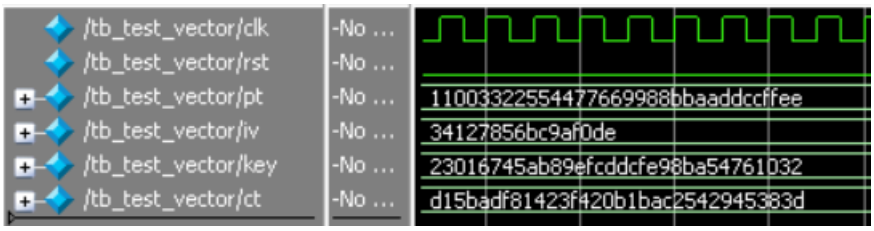


Figure 17 – Simulation Result of Test Vector 2

6. Conclusion

In this project we learned a lot about cryptography, FPGA implementation but especially we earned experience that will help us in our future life as Electronics Engineer. Some of the most important lessons we learned were:

- FPGA implementation results such as timing and area.
- Cryptography algorithm required accuracy during all stages of the design. Because of the effect avalanche of each algorithm, even a small mistake can cause a lot of damage and can be very hard to debug.
- Connecting 2 different boards with 2 clock domains required a handshake protocol that we build.
- We had to treat with noises that occurs between the Arduino and the Basys3. To solve this problem, we used Pull-Down Resistors for each signal that come from the Arduino to the Basys3. In addition, we implement, in our design, counters that check that the signals are stable before using them.

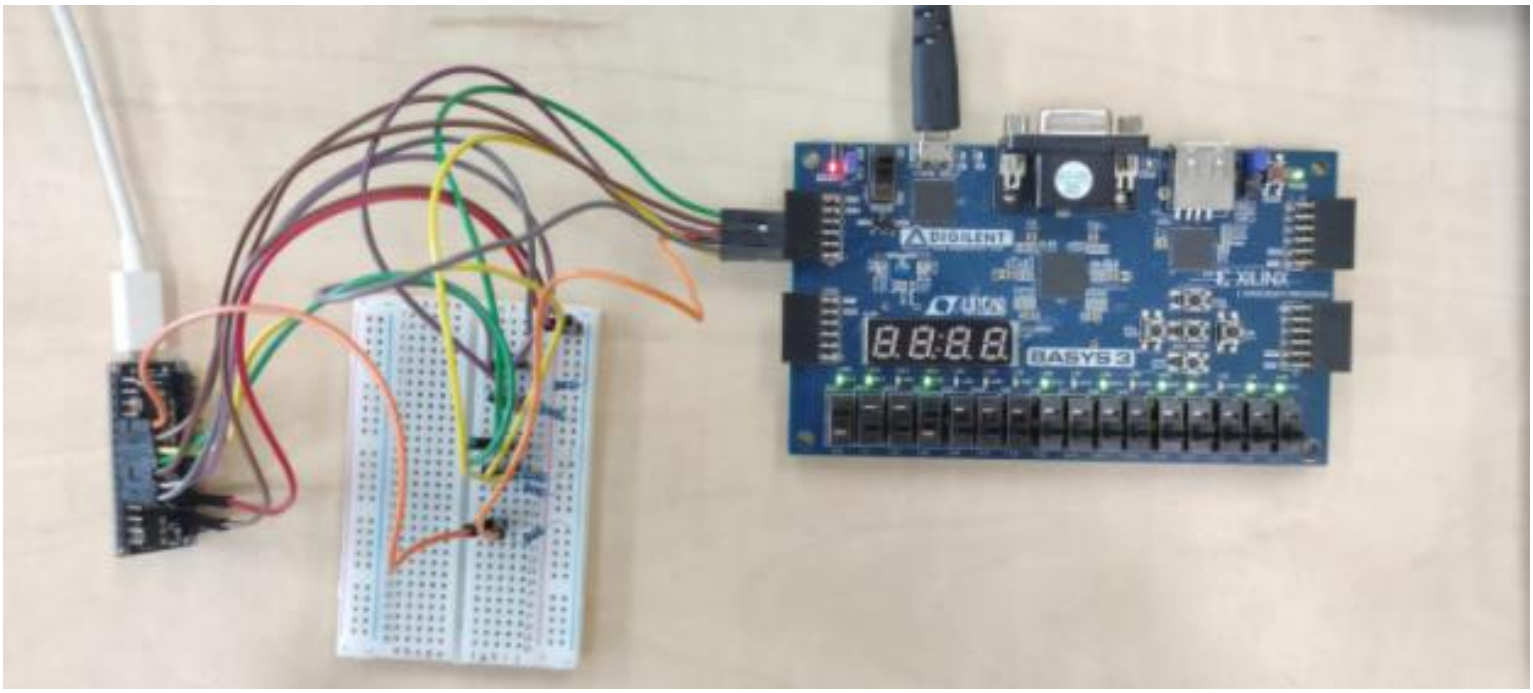


Figure 18 – Final Project: Arduino + Basys3 board

7. References

[1] Hummingbird-2 Algorithm

https://eprint.iacr.org/2011/126.pdf

[2] Project directory on GitHub:

https://github.com/eliecudkowicz/Implementation-of-Hummingbird-Encryption-Algorithm/