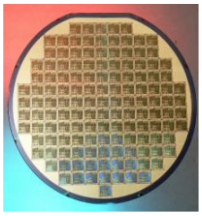
TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY



VLSI SYSTEMS RESEARCH CENTER

VLSI LABORATORY

DEPARTMENT OF ELECTRICAL ENGINEERING | DEPARTMENT OF COMPUTER SCIENCE

Cyber security message authentication

Final Report

Submitted by:

Eden Ninary Jonathan Bleicher

Supervised by:

Pavel Gushpan

Submitted on: 22/05/2024

Table of Contents

1. Introduction

The main goal of the project is to build the architecture of the cyber security message authentication, define the needs and the capabilities of the chip. Design the architecture and verify it.

At the end of the project, we will have two units in the system, a receiver and a transmitter each connected to its main system through AXI stream protocol and will know how to receive encrypted messages and authenticate them, and how to create encrypted messages and send them to the system.

1.1 Background

In this part we will examine and summarize our flow in the chip and our considerations on how to authenticate and encrypt the messages.

First, we needed to establish the basic framework of our chip and what we want to cover in our authentication techniques:

1.2 Data structure:

- Message counter – this will give the receiver knowledge that the received messages are sequential and that none are missing.
- Timestamp - this allows the receiver to know the relevance of the message in respect to time. This can prevent an attacker from intercepting the messages and withholding them for a certain amount of time and then letting them through later on.
- Secret tag – each message contains a certain tag that is known only to the transmitter and the receiver. If the tag is incorrect, the receiver can know that the message was tampered with or was not sent by the transmitter at all.
- Type - Bits labelling the type of message and how it should be interpreted by the transmitter or receiver.

Each message holds these fields (besides the timestamp, which we can leave it to the system to send a timestamp as a part of the message, since the transmitter and receiver are not aware of the actual time) to control the authentication process and to establish a correct protocol for our receiver and sender to work with.

Our message data structure(total 512 bits):

authentication tag	Msg counter	Data (w or w/o timestamp)	Type
8 bits	16 bits	488 bits	2 bit

Figure 1 - Structure of message.

1.3 System states:

1. Working state (Type bits: 00) – this is the regular state, the chip will receive and send messages to the system according to the chip specifications.

2. Update Password (Type bits: 10) - In this state the transmitter will build a message containing a new key and authentication tag as well as the type bits indicating that this is a key update message. The transmitter will encrypt this message using the old key so that the receiver will still be able to decrypt it. Upon decryption, the receiver will learn the new password and will use it to decrypt the next message.
3. Reset state (Type bits: 11) – This state is like the password update state however the transmitter encrypts the message using a hard coded key and authentication tag. The flow requires starting a session with the transmitter in this state. The receiver goes to this state by default or if it fails to authenticate a message.

2. Encryption & Authenticity

The chip applies the concepts of encryption and authenticity using the ChaCha20 algorithm which will be discussed more extensively later in this section. Authenticity is guaranteed using a secret tag known to both the sender and the receiver that must appear in each message. This tag is encrypted using the encryption algorithm to keep it safe. The use of the authentication tag promises that the message was indeed sent from the original sender and not an imposter and that the message was not tampered with on the way.

2.1 ChaCha20:

ChaCha20 is a symmetric-key stream cipher designed by Daniel J. Bernstein. It is known for its efficiency and security. Operating as a stream cipher, ChaCha20 generates a pseudorandom keystream that is XORed with plaintext to produce ciphertext. The algorithm supports a 256-bit key, which provides the security stronger than AES. One of ChaCha20's distinctive features is its speed, designed to be highly efficient in both software and hardware implementations. The following list explains why the algorithm is speedy and efficient:

- Simplicity - The algorithm’s operations are very simple and straightforward. The operations include bit rotation, addition, and bitwise xor. These operations are especially easy to implement in hardware.
- Few Rounds - ChaCha20 uses a relatively small number of rounds (typically 20), compared to some other ciphers. The fewer rounds contribute to faster encryption and decryption processes while still maintaining a high level of security.
- Low Memory Usage - ChaCha20 doesn't require large lookup tables or extensive memory usage, making it suitable for environments with limited resources. This characteristic is advantageous for applications with constrained devices or low-memory environments.
- Security - ChaCha20 is an improved version of the creator’s previous algorithm “Salsa20” which has proven to be stronger and faster than “AES”. Chacha 20 provides stronger security than “Salsa20” since it increases the amount of diffusion that the bits experience.

The disadvantages of ChaCha20 are that it is not widely adopted and that it requires key management.

We chose to use Chacha20 for all its advantages stated above and because it essentially is a collision-resistant and preimage-resistant hash function. It is a symmetric key encryption algorithm which is far simpler to implement provided proper key management. The disadvantages were not a big problem to overcome in our case since the key management is handled within the transmitter in a relatively simple manner that will be further discussed later.

2.1.1 Protocol:

ChaCha20 receives 4 inputs:

1. Key - 256 bits, this is essentially the asset that needs to be kept secret and only known to the transmitter and the receiver.
2. Nonce - 96 bits, this is a number that changes every message, it does not need to be kept secret. It is responsible for making two identical messages encrypted by the same key to have entirely different ciphertexts.
3. Message - 512 bits, this is the data itself that is later xored with the stream cipher.
4. Block count - 32 bits, when the data is handled in blocks and not altogether, this indicates which block is currently being worked on.

ChaCha20 contains 4 32-bit constants:

1. C1 = 61707865
2. C2 = 3320646e
3. C3 = 79622d32
4. C4 = 6b206574

And it creates a state matrix as such:

```
state_matrix <= {nonce, block_count, key, CONSTANT_3, CONSTANT_2, CONSTANT_1, CONSTANT_0};
```

Figure 2 - SystemVerilog code representing the state matrix.

Or in matrix form:

$$\begin{matrix} C_1 & C_2 & C_3 & C_4 \\ K_1 & K_2 & K_3 & K_4 \\ K_5 & K_6 & K_7 & K_8 \\ B_1 & N_1 & N_2 & N_3 \end{matrix}$$

Figure 3 - An abstraction of the state matrix.

Where Ci represents 32 bits of constant *i*
Ki represents the *ith* block of the key
B1represents the block count
Ni represents the *ith* block of the nonce

The blocks of the matrix are labelled 0-15 where C1 sits at block 0 and N3 sits at block 15.

The algorithm then runs 20 rounds, each round containing 4 column rounds and 4 diagonal rounds. The inner rounds are called quarter-rounds and will be defined later. The following example shows the first 4 quarter-rounds working on the columns of the matrix while the next 4 quarter-rounds work on the diagonals of the matrix.

1. quarter-round(0, 4, 8,12)
2. quarter-round(1, 5, 9,13)
3. quarter-round(2, 6,10,14)
4. quarter-round(3, 7,11,15)
5. quarter-round(0, 5,10,15)
6. quarter-round(1, 6,11,12)
7. quarter-round(2, 7, 8,13)
8. quarter-round(3, 4, 9,14)

Quarter-round is defined as follows:
Quarter-round(a,b,c,d):

1. a += b; d ^= a; d <<<= 16;
2. c += d; b ^= c; b <<<= 12;
3. a += b; d ^= a; d <<<= 8;
4. c += d; b ^= c; b <<<= 7;

Where:
"+" denotes integer addition modulo 2^32
"^" denotes a bitwise xor
"<<< n" denotes an n-bit left rotation (towards the high bits).

The operations essentially diffuse the bits very effectively, causing the state matrix at the end to appear completely random. When the finalized state matrix is xored with the original message, the encrypted message appears to be completely random but can be very easily recovered by xoring it with that same state matrix again.

Further detail on the mathematics behind the algorithm is given in Daniel J. Bernstein's original paper which can be found in the references.

2.1.2 Implementation

We implemented the ChaCha20 in SystemVerilog as 3 modules. One module implemented the quarter round operation that was explained in the section above, another module implemented the function that calls the quarter round on the rows and columns that it needs to be used on and in the correct order, and finally a top module that acts as the interface between the ChaCha block function and the rest of the chip. We built a separate testbench for each module ran extensive simulations on each of them.

2.1.2.1 Quarter Round

This module implements the basic operation of the ChaCha20 that is to be repeated on different parts of the state matrix in the right order. Figure 4 shows the SystemVerilog implementation of this basic operation which is very fast and uses relatively few logic gates. For this reason we implemented this section asynchronously.

```
//logical row 0
//a += b; d ^= a; d <<<=16;
assign a_temp0 = a_in + b_in;
assign b_temp0 = b_in;
assign c_temp0 = c_in;
assign d_temp0_t = d_in ^ a_temp0;
assign d_temp0 = {d_temp0_t[15:0], d_temp0_t[31:16]};

//logical row 1
//c += d; b ^= c; b <<<= 12;
assign a_temp1 = a_temp0;
assign c_temp1 = c_temp0 + d_temp0;
assign b_temp1_t = b_temp0 ^ c_temp1;
assign b_temp1 = {b_temp1_t[19:0], b_temp1_t[31:20]};
assign d_temp1 = d_temp0;
```

```
//logical row 2
//a += b; d ^= a; d <<< 8;
assign a_temp2 = a_temp1 + b_temp1;
assign b_temp2 = b_temp1;
assign c_temp2 = c_temp1;
assign d_temp2_t = d_temp1 ^ a_temp2;
assign d_temp2 = {d_temp2_t[23:0], d_temp2_t[31:24]};

//logical row 3
//c += d; b ^= c; b <<<= 7;
assign a_temp3 = a_temp2;
assign c_temp3 = c_temp2 + d_temp2;
assign b_temp3_t = b_temp2 ^ c_temp3;
assign b_temp3 = {b_temp3_t[25:0], b_temp3_t[31:25]};
assign d_temp3 = d_temp2;
```

Figure 4 – Quarter Round in System Verilog

```
// Initialize inputs
initial begin
    a_in = 'h11111111;
    b_in = 'h11111111;
    c_in = 'h11111111;
    d_in = 'h11111111;

    // Wait for a few clock cycles
    #10;

    // Print the initial values
    $display("Initial values:");
    $display("a_in = %h, b_in = %h, c_in = %h, d_in = %h", a_in, b_in, c_in, d_in);
    $display("a_out = %h, b_out = %h, c_out = %h, d_out = %h", a_out, b_out, c_out, d_out);

    // End simulation
    $finish;
end
```

Figure 5 – Quarter Round Testbench Stimulus

```
VSIM 22> run
# Initial values:
# a_in = 11111111, b_in = 11111111, c_in = 11111111, d_in = 11111111
# a_out = 77777777, b_out = eeeeeeee, c_out = 88888888, d_out = 44444444
# ** Note: $finish    : C:/Users/edenn/Desktop/CSMA_PROJECT_2024/shared/chacha/quarter_round_tb.sv(48)
#    Time: 10 ps  Iteration: 0  Instance: /tb_chacha_quarter_round
```

/tb_chacha_quarter_round/a_in	11111111	11111111
/tb_chacha_quarter_round/a_out	77777777	77777777
/tb_chacha_quarter_round/b_in	11111111	11111111
/tb_chacha_quarter_round/b_out	eeeeeeee	eeeeeeee
/tb_chacha_quarter_round/c_in	11111111	11111111
/tb_chacha_quarter_round/c_out	88888888	88888888
/tb_chacha_quarter_round/d_in	11111111	11111111
/tb_chacha_quarter_round/d_out	44444444	44444444

Figure 6 – Quarter Round Simulation results example 1.

```
VSIM 26> run
# Initial values:
# a_in = 00000001, b_in = 00000001, c_in = 00000001, d_in = 00000001
# a_out = 30000002, b_out = 81811899, c_out = 03030231, d_out = 03000230
# ** Note: $finish    : C:/Users/edenn/Desktop/CSMA_PROJECT_2024/shared/chacha/quarter_round_tb.sv(48)
#    Time: 10 ps  Iteration: 0  Instance: /tb_chacha_quarter_round
```

/tb_chacha_quarter_round/a_in	00000001	00000001
/tb_chacha_quarter_round/a_out	30000002	30000002
/tb_chacha_quarter_round/b_in	00000001	00000001
/tb_chacha_quarter_round/b_out	81811899	81811899
/tb_chacha_quarter_round/c_in	00000001	00000001
/tb_chacha_quarter_round/c_out	03030231	03030231
/tb_chacha_quarter_round/d_in	00000001	00000001
/tb_chacha_quarter_round/d_out	03000230	03000230

Figure 7 – Quarter Round Simulation results example 2.

In the examples above, the quarter round takes each input and converts it to something else. This can clearly be seen in the first example as “11111111” is converted to “77777777” in hexadecimal format. Both examples show how each input gets affected differently when receiving the same input. This is one of the key ideas behind the algorithm which mathematically promises effective diffusion of ones and zeros throughout the state matrix.

2.1.2.2 Block Function

The Block Function’s purpose is to receive inputs from the ChaCha Top unit that represent the state_matrix’s columns and diagonals. It returns them to the Top unit after being correctly scrambled. The block function is a synchronous unit that contains an internal state machine. The state machine has 3 states, IDLE, EVEN, and ODD. This unit instantiates 4 quarter round instances as well as a counter. The EVEN states run the state_matrix’s columns through the quarter rounds and the ODD state runs the diagonals through the quarter rounds. The states alternate between EVEN and ODD each clock cycle and after 20 cycles the algorithm is complete, the output is ready, and the state machine returns to IDLE. Although a testbench was built for this

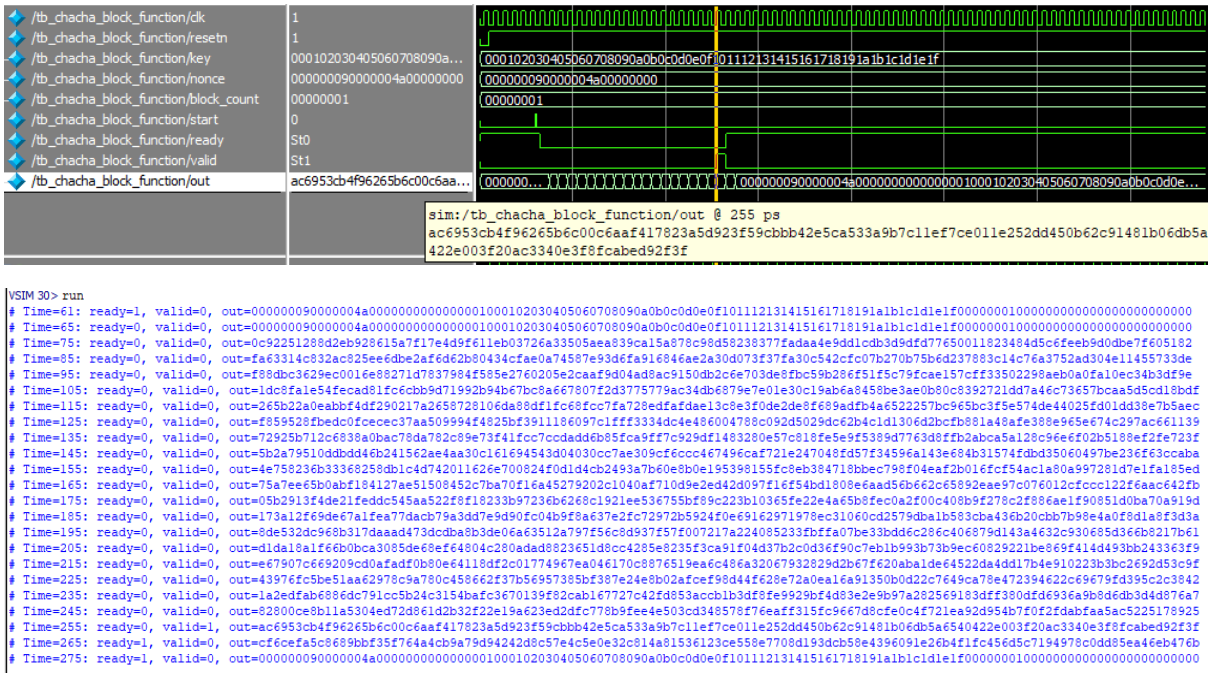


Figure 8 – Block Function Simulation.

As shown in Figure 8, the block function's purpose is to scramble the state_matrix that the ChaCha Top unit provides for it. The first blue line of the second image shows that the Block Function unit is ready to receive its information from the Top unit and it immediately inserts the given state_matrix to its output. At this stage it is clear what values the matrix is composed of and the values are patterned and easy for a human to read. In the third row from the bottom on that same image it is shown that at "Time=255" the data is ready. At this point it means that the block function completed 20 quarter rounds, alternating between column and diagonal and achieved a completely random and unintelligible matrix. The values at that point seemingly have absolutely nothing to do with the original state_matrix which is exactly the objective.

2.1.2.3 ChaCha Top

This unit implements the interface between the Block function which scrambles a matrix and the rest of the chip which has a message to encrypt. The unit receives all of the inputs that the Chacha needs to build a state matrix (key, nonce, block number, constants) as well as the data to encrypt. After it sends the matrix to the block function and receives a scrambled state matrix, it bitwise xors the original message with the scrambled matrix. In doing so the message is fully encrypted and unintelligible and only the receiver can decrypt it using the same key and nonce. The Receiver uses this same unit for decryption, in the receiver's case the message that comes in is the ciphertext, it feeds the relevant data to the block function and receives the identical scrambled state_matrix. The ChaCha Top then xors the state matrix with the ciphertext and in doing so the plaintext is revealed.

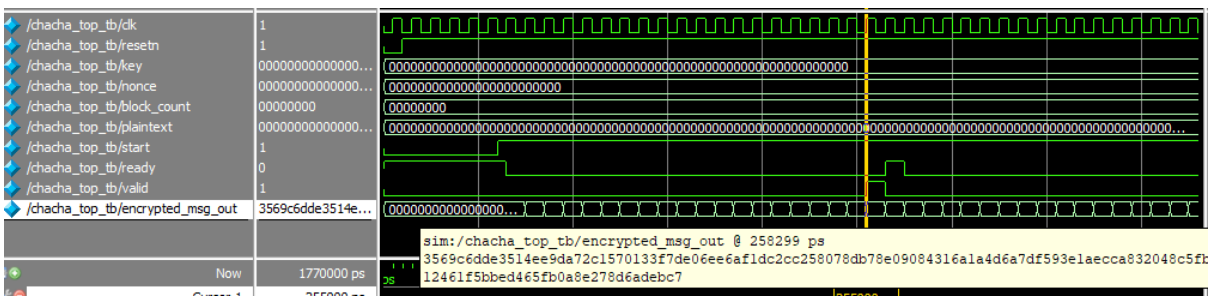


Figure 9 – ChaCha Top Simulation Results

Figure 9 above shows how a message “512’b0” in line one of the second image gets converted to a random mess of ones and zeros in the second line from the bottom of the same image when the valid bit goes up. This shows the efficacy of the module in the encryption direction. Its ability to decrypt is demonstrated later on when both Transmitter and Receiver modules are tested simultaneously.

2.2 Key generation & Authentication:

We designed a block called key_generator which is a basic implementation of a random number generator. It works by receiving an initial seed, then by xoring certain bits with other bits along with rotations we diffuse the bits and create a pseudo random key. This unit is always running so long as it receives a clock signal. This adds another layer of randomness since the calling of this unit can occur at random times so the given output is very unpredictable. This same unit also generates the secret 8-bit authentication tag in the same manner.

3. Communication:

3.1 AXI Stream:

Both the transmitter and the receiver communicate with their home systems using the AXI Stream protocol. This ensures universality since it is a standard protocol form of communication that is common in the industry. Its advantages include its simplicity for implementation along with its bandwidth capabilities.

In this project we use the most basic version of the AXI Stream protocol which contains only the minimum requirements for a master and a slave to communicate. In our case both the transmitter and the receiver contain master and slave components since they both manage 2-way communication between their home systems that they sit on. The essence of the protocol is the valid ready handshake which is demonstrated in Figure 10. The data between master and slave is only transferred when the slave is ready, and the master has valid data simultaneously:

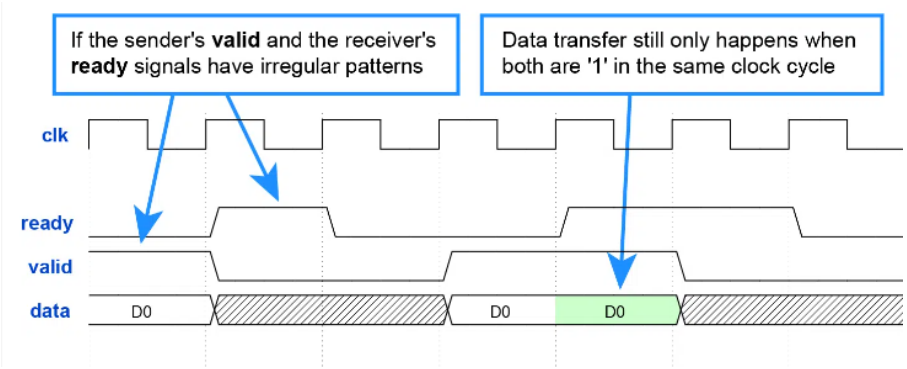


Figure 10 - Diagram taken from https://vhdlwhiz.com/how-the-axi-style-ready-valid-handshake-works/

3.1.1 AXI Stream Master and Slave Modules

The project contains two modules that make up the AXI Stream Interface. One module is the Master and the other the Slave. Both the Transmitter and the Receiver contain one module of each to communicate with the system that they are respectively on. The modules and the protocol were tested using a testbench that was designed for each module separately. Figure 11 below shows that the "valid" signal is delayed meaning that although the output was ready before then, the system did not read the data until the valid bit was high since the system was ready all along.

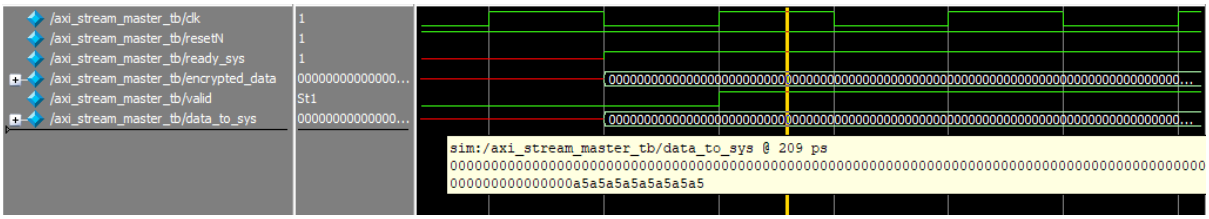


Figure 11 – AXI Stream Master under testbench

4. System Summary

The system contains two different perspectives, we have the receiving part and the sending part. Both the receiver and the transmitter are connected to a main system which is responsible for receiving and sending the encrypted messages and can communicate with each other.

The chips communicate with the system through AXI steam protocol and can receive and transmit different types of messages from and to the system.

4.1 System overview:

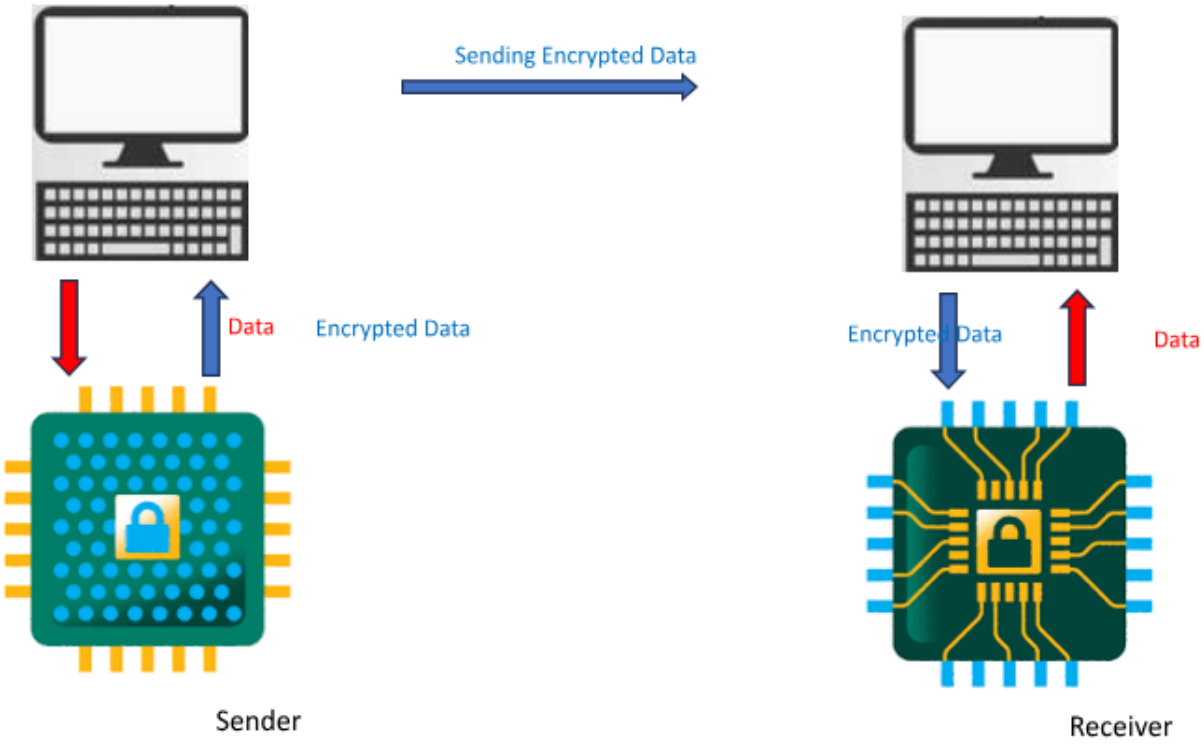


Figure 12 - Abstraction of the transmitter and receiver connecting to their home systems.

The sending system hands over the data to the transmitter chip, which wraps the data and encrypts it. It then returns the encrypted data to the system for it to be sent to the receiving system.

The receiver system gives the encrypted data to the receiver chip, the chip then opens the message and validates the authenticity of the message. After the receiver validates the correctness of the data with the special key, it sends the decrypted data back to the receiving system.

The complete end to end process gives the systems the ability to communicate privately through public channels with a very high probability of correctness of the messages.

4.2 System encrypt process:

The system uses the Chacha20 algorithm or encryption and decryption. The transmitter first uses a hard coded key (that is hard coded into the receiver as well) to send the next randomly generated key. The systems will then use the next key until they decide to update the keys again.

After the transmitter finishes framing the message with all the relevant information (as well as the authentication tag), it uses its secret key to encrypt the message and then returns it to the system to send to the receiver.

This message is securely encrypted due to the properties of the Chacha20 algorithm that we covered earlier.

4.3 System authentication process:

The authentication process we chose is an 8-bit authentication tag that is a part of the message. The first message is sent with a hard coded authentication tag which both the transmitter and receiver agree upon. The next authentication tag is randomly generated at the transmitter chip and is sent to the receiver via the first message encrypted by the hard coded encryption key.

The second part of the authentication process includes the message counter which is another 16 bits. This counter needs to be correctly incremented with each message that the transmitter sends. Together with the authentication tag, the strength of the authentication process comes from the inability of the attacker to generate a message that will have the authentication tag bits (which are unknown to the attacker) in the correct place, as well as a counter (whose contents are not known to the attacker either) incremented to the right value and in the correct place in the message. As stated earlier this is due to the Chacha20 algorithm being preimage resistant, which leaves the attacker with the ability as good as chance to succeed.

5. Modules

5.1 Transmitter

5.1.1 Transmitter Overview

The transmitter consists of 5 sub-modules:

1. AXI stream slave
2. AXI stream master
3. Transmitter Manager
4. ChaCha20 unit
5. Key Generator unit

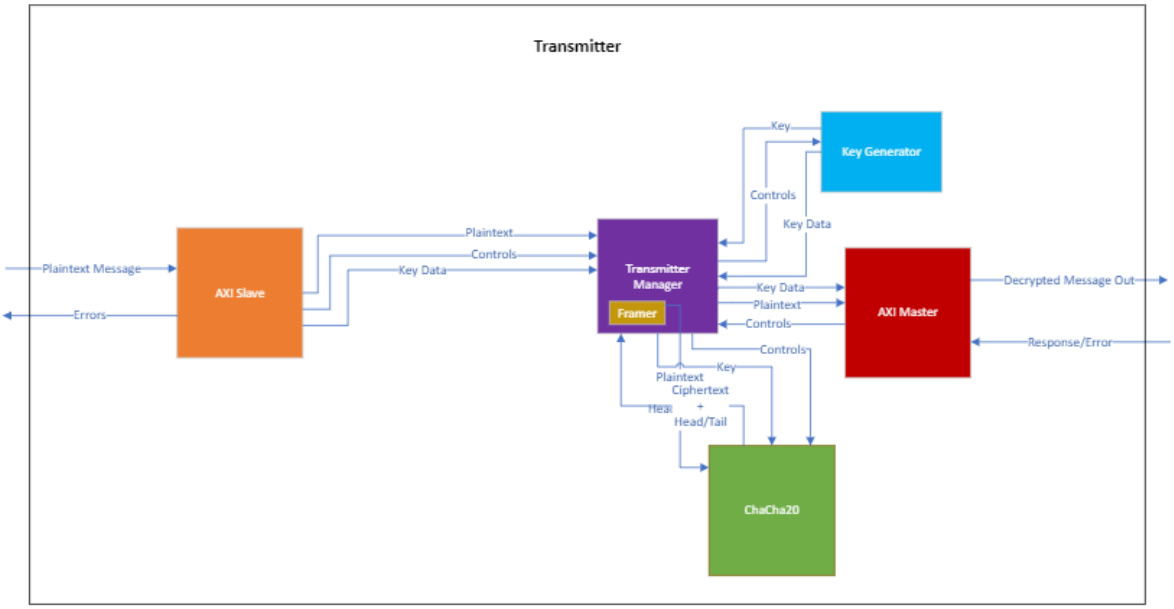


Figure 13 - Transmitter Modules

5.1.2 Transmitter Workflow

The workflow of the transmitter in our Cyber Security Message Authentication project involves a series of steps, from receiving the message to encrypting it and finally returning the encrypted message. The process begins with the transmitter receiving a message from the main system, formatted in accordance with the AXI stream protocol. The raw message undergoes initial processing, where it is framed to conform to the communication protocol. Next, the transmitter initiates the encryption phase using the selected ChaCha20 algorithm. During encryption, the ChaCha20 algorithm employs a key, nonce, and counter to transform the plaintext message into ciphertext. The key, nonce, and counter contribute to the creation of a unique stream cipher, ensuring that even the same plaintext produces different ciphertexts. Once the encryption process is complete, the transmitter embeds additional information into the message data structure, such as timestamps, message counters, and secret tags, to facilitate authentication at the receiver's end. This data structure represents the encrypted message, and it is transmitted back to the main system or AXI stream protocol. In summary, the transmitter's workflow encompasses receiving, framing, encrypting, embedding authentication elements, and transmitting the encrypted message, contributing to the overall cyber security infrastructure of the project.

5.1.3 Transmitter Manager

The transmitter manager controls the current state machine according to our stage in the encryption process, generally it goes from receiving the un-encrypted data and start building the authentication and encryption process step by step to ensure correctness of our end goal of encrypting and authenticating the data.

5.1.3.1 Transmitter manager state machine

1. **Reset State** - This state serves as the initial setup phase. It transitions to the IDLE state upon system initialization, establishing the foundation for subsequent

operations. In this state, the manager sets various parameters and initializes counters, preparing the system for regular operation.

2. **IDLE State** - The IDLE state is the default state where the manager is ready to receive messages. It signals its readiness to the slave module, prompting the system to send the next message. The message and nonce counters are activated, and the system awaits a valid message from the slave.
3. **PREP_MSG State** - During the PREP_MSG state, the manager processes the incoming message in preparation for encryption. It distinguishes between a regular message and a key update based on the state bits. For a regular message, it constructs the framed message, incorporating the authentication tag, message counter, and plaintext data. If it's a key update, the manager includes the new key and authentication tag in the framed message. The manager then transitions to the SEND_TO_CHACHA state.
4. **SEND_TO_CHACHA State -** In this state, the manager signals the ChaCha module to begin the encryption process. It sets the start signal for ChaCha and waits for the encrypted message. The manager remains in this state until the ChaCha module indicates readiness. Once the encryption is complete, the manager progresses to the SEND_TO_MASTER state, signalling the master module that the encrypted message is ready.
5. **SEND_TO_MASTER State** - In the SEND_TO_MASTER state, the manager forwards the encrypted message to the master module. It ensures that the master is ready to receive the data. If the master is prepared, the manager sends the encrypted message and increments the nonce counter. The system then returns to the IDLE state, ready to process the next message. If the master is not ready, the manager remains in the SEND_TO_MASTER state until the master indicates its readiness to receive the data.

   These states collectively orchestrate the transmitter manager's workflow, handling message preparation, encryption, and transmission, ensuring the secure and authenticated communication of messages within the cyber security framework.

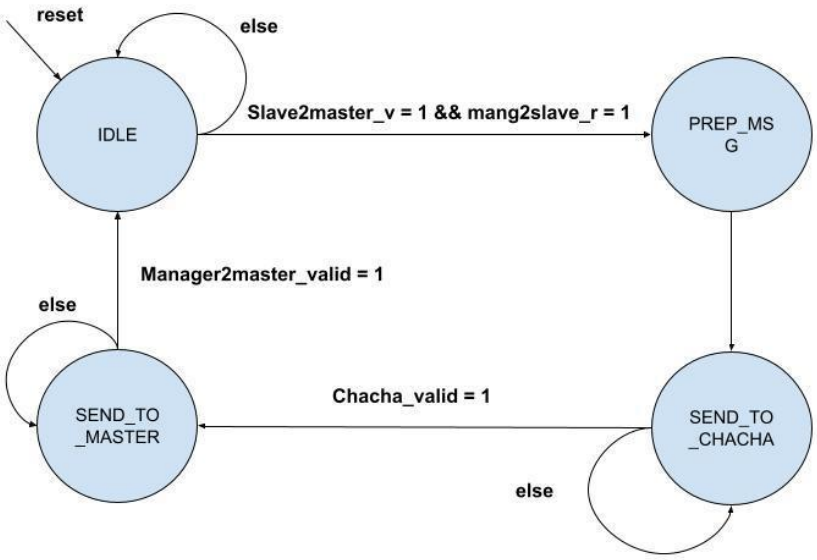


Figure 14 - State Machine Diagram

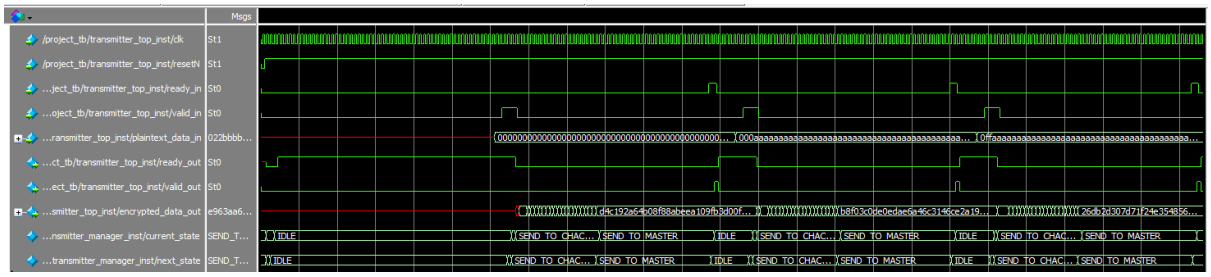


Figure 15 – Manager State Transitions in Simulation

5.1.4 Key Generator

This module is a variation of a xorshift pseudo-random number generator. The unit receives a constant seed and with each clock cycle xors, rotates, and shifts bits around. The manager accesses this unit at random which means that the result that is always being calculated is almost guaranteed to be different every time, and even when it begins to repeat itself, it wont appear to be repeating in the same order as the previous time due to the random access. This unit provides the manager with the secret key for the chacha as well as the secret authentication tag. We created a testbench for this unit as well and ensured that after about 60 clock cycles, no part of the original number is recognizable anymore.

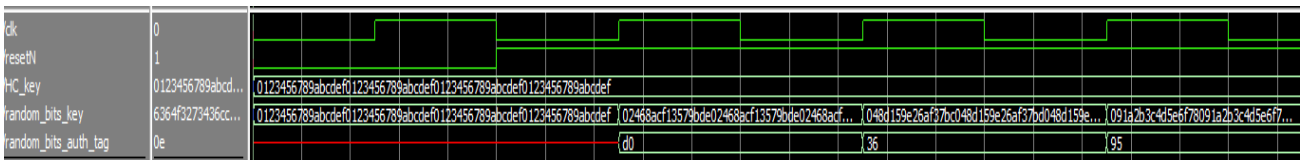


Figure 16 - Here the diffusion of ones is shown after just 3 cycles. We don't use the value here for at least 60 more cycles of diffusion.

5.1.5 Transmitter Simulation

We built a testbench that wraps the Transmitter and feeds it plaintext messages using the AXI stream protocol. As shown in figure 17, the plaintext comes into “data_in” in very clearly patterned forms and goes out through “data_out” completely scrambled and unintelligible. The “valid” and “ready” signals are also shown here complying with the AXI Stream protocol.

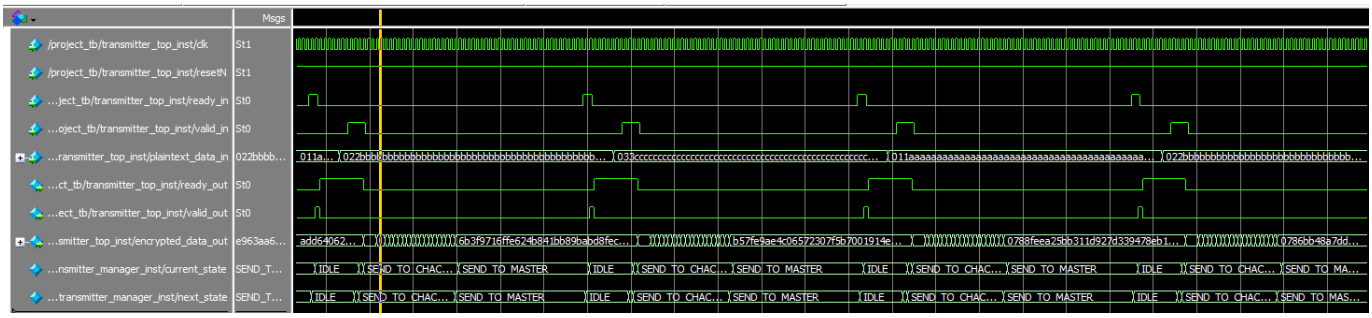


Figure 17 - Transmitter Top Module workflow.

5.2 Receiver

5.2.1 Receiver overview

The receiver consists of 4 sub-modules:

1. AXI stream slave
2. AXI stream master
3. Receiver Manager
4. ChaCha20 unit

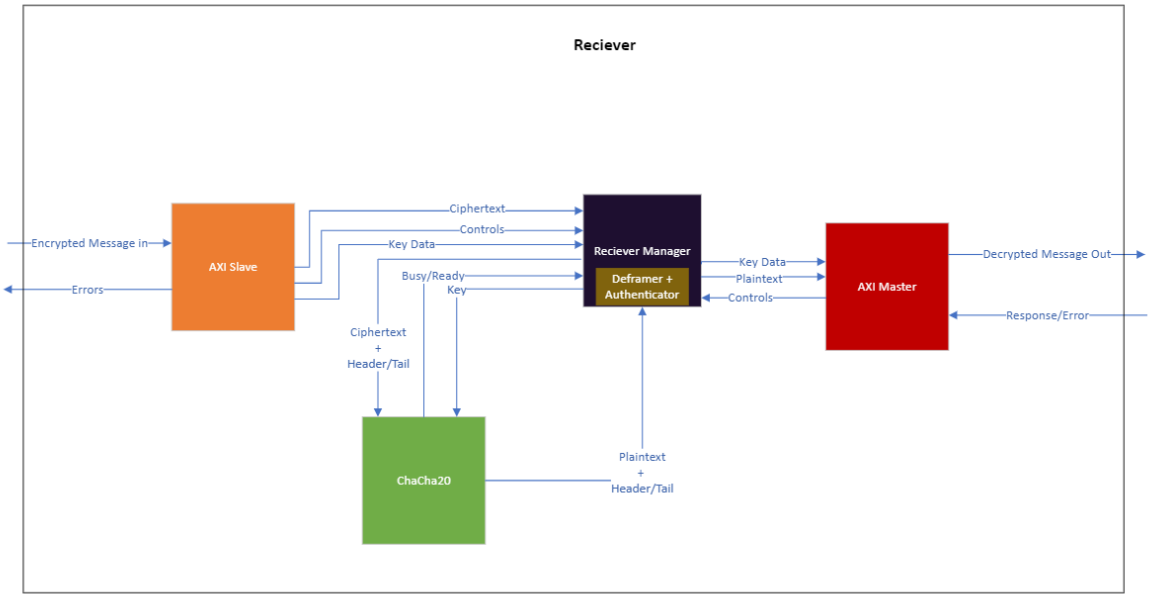


Figure 18 – Receiver Modules

5.2.2 Receiver Workflow

The receiver works very similarly to the transmitter. Since we are using a symmetric key encryption, the receiver follows the same steps as the transmitter for the most part, except for the fact that several of the steps are inverted. The receiver receives an encrypted message, funs the Chacha on it in the exact same way with the current key and nonce that it has stored and due to the symmetric properties of the Chacha, the decrypted message appears in the format shown in Figure 1. More details are discussed in the state machine section.

5.2.3 Receiver Manager

The receiver manager controls the current state machine according to our stage in the decryption process. It is very similar to the transmitter with minor differences.

5.2.3.1 Receiver State Machine

1. RESET State - This initial state sets the system to a reset state, preparing it for subsequent operations. In this state, default values for keys and authentication tags are established. The system transitions from this state to IDLE to await incoming messages.
2. **IDLE State** - The system rests in an idle state, ready to receive and process messages. It signals its readiness to the external system and initializes necessary counters. It remains in this state until a valid, encrypted message is detected.
3. **START_DECRYPT** - Upon detecting a valid message, the system enters the START_DECRYPT state. It signals the ChaCha20 decryption module to begin processing the received ciphertext. The system moves to the next state once the decryption module is ready to proceed.
4. **FINISH_DECRYPT** - In this state, the system completes the decryption process. It deactivates the decryption module and checks if the decrypted message is valid. If the decryption is successful, the system proceeds to the AUTHENTICATE state; otherwise, it stays in the FINISH_DECRYPT state.
5. **AUTHENTICATE** - This state involves verifying the authenticity of the decrypted message. The system checks if the received authentication tag matches the expected tag. If there is a mismatch, the system may reset the encryption key to a hardcoded value (HC_KEY) and retry decryption. If the mismatch persists, the system signals an authentication failure and moves to the SEND_TO_MASTER state.
6. **SEND_TO_MASTER** - If the message passes authentication, the system sends the decrypted plaintext message to the master module through the AXI stream. The master module is informed of the message's validity and processes it accordingly. The system then returns to the IDLE state, ready to receive the next message.

Each state in the receiver manager's state machine contributes to the overall process of securely receiving, decrypting, authenticating, and forwarding messages within the cyber security message authentication system.

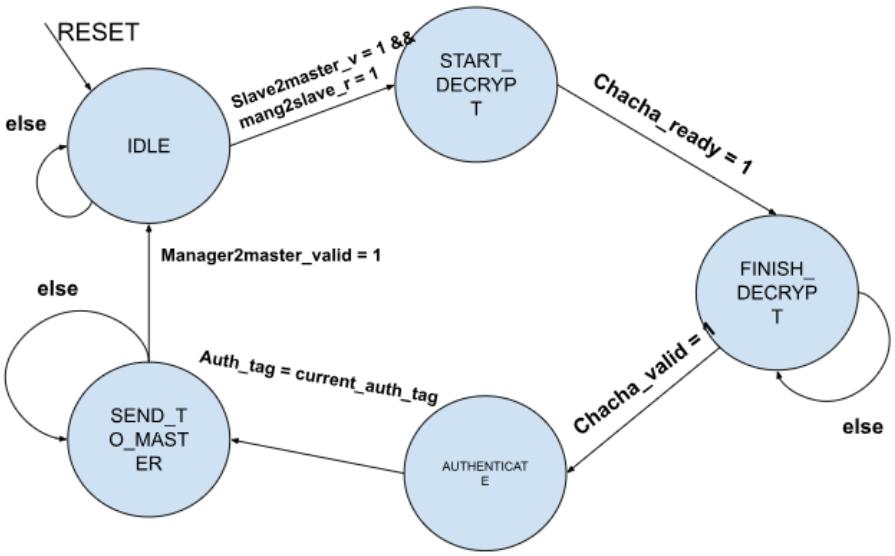


Figure 19 – State Machine Diagram

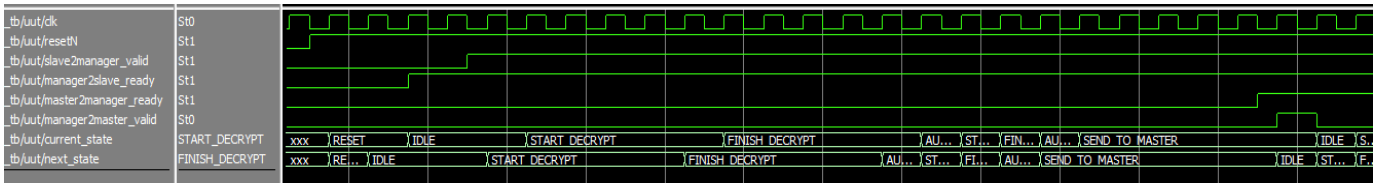


Figure 20 – Receiver Sate Transmissions in Simulation

5.2.4 Receiver Simulations

Similarly to the transmitter, we bult a testbench for the receiver as well. In this case, the testbench feeds the receiver an encrypted message through “data_in” and the receiver spits out a decrypted message in the “data_out”. The decrypted message can consist of the data sent, or a message notifying the system that the message was not received or was inauthentic, all of course using the AXI Stream handshake protocol. Shown in Figure 21, the data comes in as seemingly random numbers and letters, yet the outputs are clearly patterned messages. The states are shifting as explained previously and the AXI Stream handshake can be seen in accordance with he states and the data.

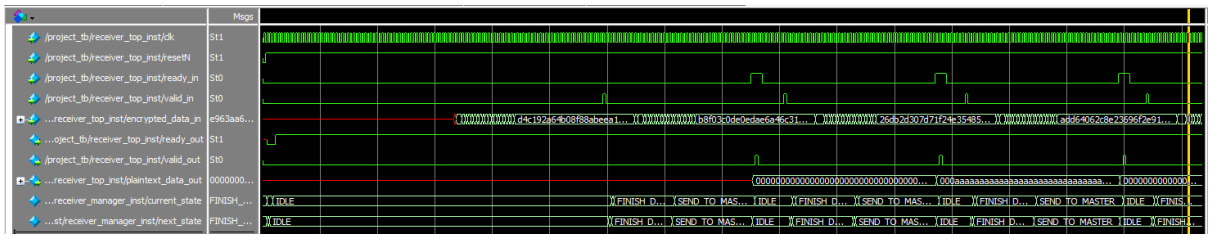


Figure 21 – Receiver Under Testbench

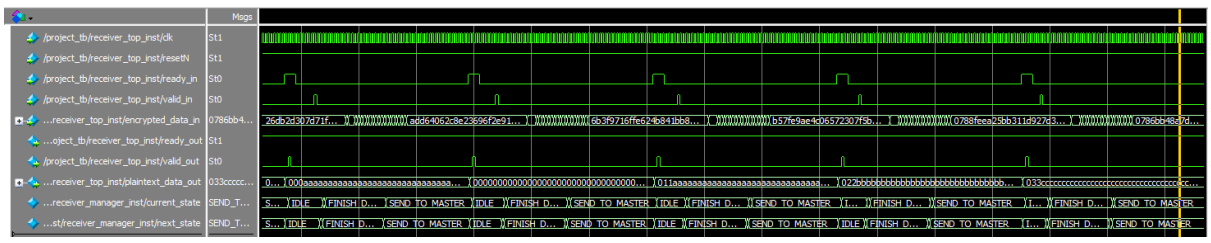


Figure 22 – Continuation of Figure 14

5.3 Project Testbench

We bult a Testbench that simulates the entire system containing both the transmitter and the receiver. The test feeds the transmitter raw data to encrypt, and the transmitter connects directly to the receiver and feeds it the encrypted message. The receiver decrypts the data and returns the raw data that was first given to the Transmitter. This testbench helped prove that both designs work seamlessly together and that all the protocols indeed work. The Test stimuli included raw data, password changes, message repeats, and even some intentionally malicious messages which the receiver was able to successfully detect. A small example of the test stimuli is given in Figure 23 below.

```
initial begin //transmitter
    resetN = 0;
    sys2trans_ready = 0;
    sys2trans_valid = 0;

    #10;
    resetN = 1;

    //first message 11 -1
    sys2trans_plaintext = 488'b11;
    #20;
    sys2trans_valid = 1;
    #40;
    sys2trans_valid = 0;
    #500;
    sys2trans_ready = 1;
    #20;
    sys2trans_ready = 0;

    #50;

    //next message 00 -2
    sys2trans_plaintext = 488'b0001;
    #20;
    sys2trans_valid = 1;
    #40;
    sys2trans_valid = 0;
    #500;
    sys2trans_ready = 1;
    #20;
    sys2trans_ready = 0;
```

```
initial begin //receiver
  sys2rec_ready = 0;
  //1
  #1100;
  sys2rec_ready = 1;
  #40;
  sys2rec_ready = 0;

  //2
  #600;
  sys2rec_ready = 1;
  #40;
  sys2rec_ready = 0;

  //3
  #600;
  sys2rec_ready = 1;
  #40;
  sys2rec_ready = 0;

  //4
  #600;
  sys2rec_ready = 1;
  #40;
  sys2rec_ready = 0;

  //5
  #600;
  sys2rec_ready = 1;
  #40;
  sys2rec_ready = 0;
```

Figure 23 – Project Testbench Stimulus

Figure 24 below shows the patterned input entering the transmitter, the ciphertext transferring between the transmitter and the receiver, and finally the patterned message showing up at the output of the receiver.

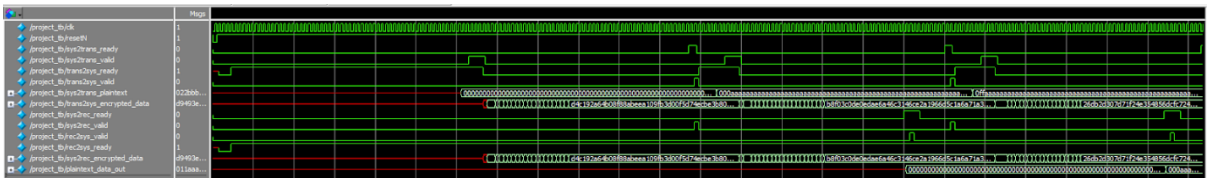


Figure 24 – Example of Project Testbench feeding transmitter and receiving output from Receiver.

6. Synthesis and Layout

This part of the project consists of two stages that turn the design into something that can be produced in hardware. Since our Transmitter and Receiver are relatively similar, we focused only on synthesizing the transmitter.

6.1 Synthesis

This stage converts our hierarchical design in system Verilog which consisted of several files and was written in a way that was easy for us to read and understand, to a gate level design. The output of this process is a Verilog file that contains identical logic to the previous stage, however it is written as instantiations and connections of standard library cells.

```
    module
chacha_top_IN_WIDTH512_KEY_WIDTH256_NONCE_WIDTH96_BLOCK_COUNT_WIDTH32_WIDTH32_OUT_WIDTH512 (
        clk, resetn, key, nonce, block_count, data_in, start, ready, valid,
        data_out );
  input [255:0] key;
  input [95:0] nonce;
  input [31:0] block_count;
  input [511:0] data_in;
  output [511:0] data_out;
  input clk, resetn, start;
  output ready, valid;

  wire   [511:0] key_stream;

  xr02d1 U1 ( .A1(key_stream[9]), .A2(data_in[9]), .Z(data_out[9]) );
  xr02d1 U2 ( .A1(key_stream[99]), .A2(data_in[99]), .Z(data_out[99]) );
  xr02d1 U3 ( .A1(key_stream[98]), .A2(data_in[98]), .Z(data_out[98]) );
  xr02d1 U4 ( .A1(key_stream[97]), .A2(data_in[97]), .Z(data_out[97]) );
  xr02d1 U5 ( .A1(key_stream[96]), .A2(data_in[96]), .Z(data_out[96]) );
  xr02d1 U6 ( .A1(key_stream[95]), .A2(data_in[95]), .Z(data_out[95]) );
  xr02d1 U7 ( .A1(key_stream[94]), .A2(data_in[94]), .Z(data_out[94]) );
  xr02d1 U8 ( .A1(key_stream[93]), .A2(data_in[93]), .Z(data_out[93]) );
  xr02d1 U9 ( .A1(key_stream[92]), .A2(data_in[92]), .Z(data_out[92]) );
  xr02d1 U10 ( .A1(key_stream[91]), .A2(data_in[91]), .Z(data_out[91]) );
  xr02d1 U11 ( .A1(key_stream[90]), .A2(data_in[90]), .Z(data_out[90]) );
  xr02d1 U12 ( .A1(key_stream[8]), .A2(data_in[8]), .Z(data_out[8]) );
  xr02d1 U13 ( .A1(key_stream[89]), .A2(data_in[89]), .Z(data_out[89]) );
  xr02d1 U14 ( .A1(key_stream[88]), .A2(data_in[88]), .Z(data_out[88]) );
  xr02d1 U15 ( .A1(key_stream[87]), .A2(data_in[87]), .Z(data_out[87]) );
```

Figure 25 – Example of the synthesized chacha_top module. It is implemented at the gate level with standard library cells "xor02d1" among others.

In the project we used Synopsis's Design Vision as the logic synthesis tool and a Tower Semiconductor standard cell library.

6.2 Layout

The Layout stage of the project consists of converting the Verilog file that resulted from the synthesis stage to a physical layout of each the standard cells that make up the chip. This stage is done using Cadence's Innovus tool.

6.2.1 sizing

The first stage of the layout included determining the initial size of the chip. Our transmitter has many inputs and outputs and therefore the chip would have had to be very large to accommodate for just the inouts had we not excluded them from the layout. We chose to ignore them and to do this stage only for the internal logic of the chip which we were able to fit on a 1mm X 1mm chip.

6.2.2 Power Grid

The next stage consisted of laying a power grid which provides VDD and VSS to each cell in the design. This step is important because it is responsible for bringing a reliable voltage to each cell. Without doing this step in grid format, some cells may receive weekend / noisy supply signals which can cause the entire chip to malfunction. The grid was implemented on the M2 metal layer. In addition to this, Using the Special Route feature, we laid out another grid in the M1 layer which bridges between the M2 layer and the cells themselves.

6.2.3 Standard Cells

In this stage we filled the area of the chip with the library standard cells. The spaces that are left are filled in with filler cells to maintain the continuity of the N-well and the implant layers (This is a standard Fab constraint). The tool does this automatically in a way that best utilizes area. One thing that it does for example is that it sets all the cells in rows and flips over every other row so that the cells in different rows can share VSS and VDD. This trick saves area as well as metal since it minimizes the number of wires needed as well.

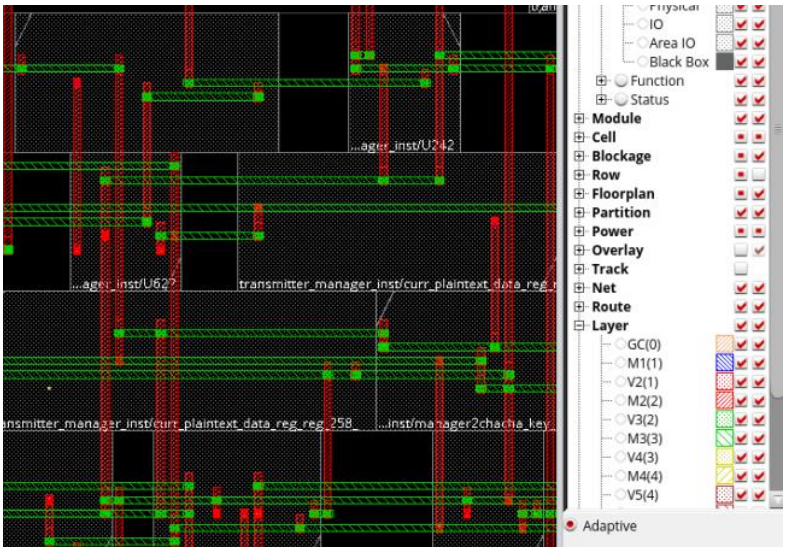


Figure 26 – Cells placed before filler.

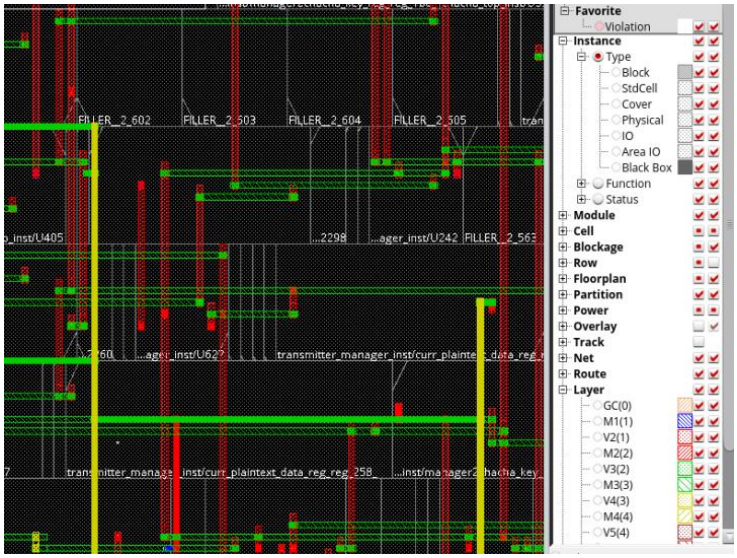


Figure 27 – Placement after fillers were added.

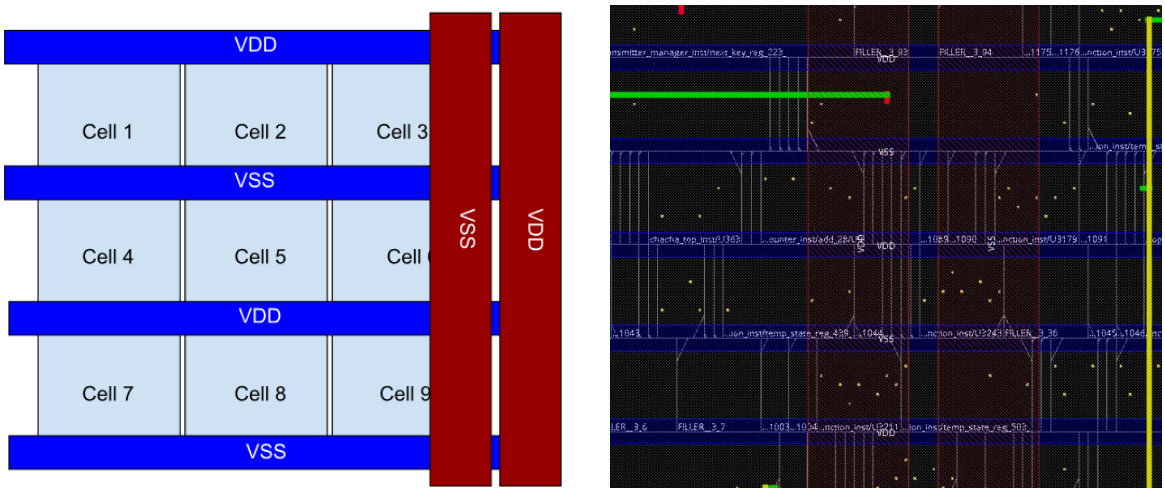


Figure 28 – Illustration of rows sharing VDD and VSS on left and the real layout on the right.

6.2.4 Clock Tree Synthesis

Cock Tree Synthesis is the stage where the clock grid is planned. This is also done automatically with the tool. The goal of CTS is to provide the clock signal to each synchronous unit in the design. Unlike the power supplies, this is a bit trickier since it involves a changing signal, and it is very time sensitive (whereas the VDD and VSS are just constants). This involves calculating how long it takes each clock signal to get to each unit and placing buffers where needed so that the clock signal will be uniform in all the units. The buffers are also used as repeaters that amplify the signal where it gets weaker. This is known as clock integrity, and it is crucial for the design to work properly. The goal is to get a model that is similar to a well-balanced tree just like the result we got in figure 29.

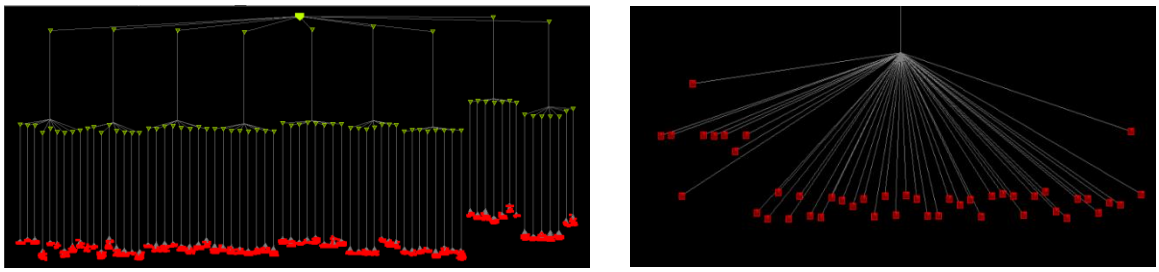


Figure 29 – The resulting Clock Tree. Buffers are marked with yellow triangles and the red leaves are the synchronous units of the design. (zoom in of leaves on the right).

6.2.5 Routing

The final stage of the Layout that we completed was the routing stage. Routing is the process of making all the necessary connections between pins of the cells in the design. This involves laying out the metal layers to make the connections. Here as well, the process was optimized automatically by the tool. This process is very complicated when broken down since it has to abide by many factory constraints and account for parasitic capacitance and resistance and signal integrity and many more factors that would otherwise make this too complicated to complete manually.

6.3 Final Result

Figure 30 contains an image of the chip after completion of the layout process. This process was successful since we were able to achieve it in a relatively small die (1mm X 1mm) and the density is spread out evenly. The image shows only the metal layers since this is a standard cell design and the active layers are hidden.

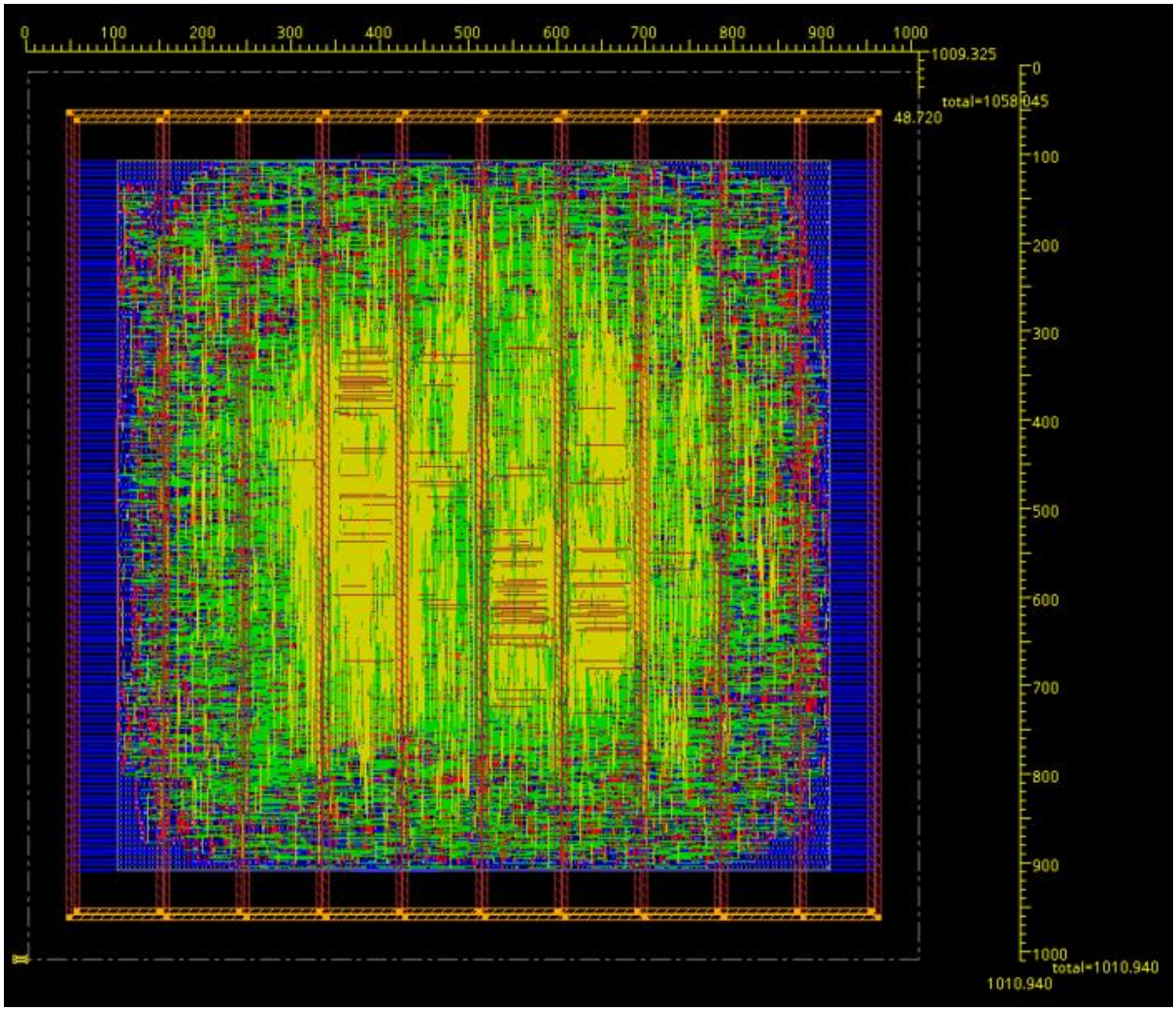


Figure 30 – Final result after the synthesis and layout process.

7. Recommendations

7.1 Principles and Tools in Computer Security 046820

While working on the project, we took the "Principles and Tools in Computer Security" course taught by Ittay Eyal, which proved to be incredibly beneficial. The course was well-taught, and provided a thorough understanding of key concepts and protocols in cyber security that directly complimented the project work. For future students undertaking similar projects, we highly recommend enrolling in this course either before or in parallel to the project. It offers many insights and knowledge on the concepts of security and how to build a truly secure system, or more importantly how not to build one.

8. References

1. Progress in Cryptology – INDOCRYPT 2020, Karthikeyan Bhargavan, Elisabeth Oswald Manoj Prabhakaran.
2. A 3.65 Gb/s Area-Efficiency ChaCha20 Cryptocore, Ronaldo Serrano, Marco Sarmiento, Ckristian Duran, Trong-Thuc Hoang, and Cong-Kha Pham, The University of Electro-Communications (UEC), Tokyo, Japan
3. AN IMPLEMENTATION OF CHACHA20 STREAM, CYPHER IN ALL-PROGRAMMABLE SoCs, IGOR SEMENOV.
4. ChaCha, a variant of Salsa20, Daniel J. Bernstein, The University of Illinois at Chicago, 2008