
Politecnico di Torino

# Cybersecurity for Embedded Systems (01UDNOV)

# Design of a lightweight cipher core for the SEcube™ FPGA: Grain-128AEAD

# Final Project Report

Master Degree in Computer Engineering

Referents: Prof. Paolo Prinetto, Matteo Fornero, Nicoló Maunero, Gianluca Roascio

Nicolò Bianco (280066)
Giuseppe Carrubba (269182)
Antonio Ras (270259)

# Contents

# CHAPTER 1

# Work Presentation

There are several emerging areas in which highly-constrained devices are interconnected, typically communicating wirelessly with one another, and working in concert to accomplish some task. Because the majority of current cryptographic algorithms were designed for desktop/server environments, many of these algorithms do not fit into constrained devices.
The US National Institute for Standard and Technology (NIST) has initiated a process to solicit, evaluate, and standardize lightweight cryptographic algorithms that are suitable for use in constrained environments where the performance of current cryptographic standards is not acceptable. NIST has published a call for algorithms [5] to be considered for lightweight cryptographic standards. 57 submissions have been received, and 10 were selected to continue to the final round.
The main feature of these algorithms is the **Authenticated encryption with associated data (AEAD)** scheme, which ensures both the confidentiality for the plaintexts and the integrity of the ciphertexts. AEAD algorithms are expected to maintain security as long as the nonce is unique. A typical AEAD algorithm requires four byte-string inputs and produces one byte-string output:

- One fixed-length **Key**
- One fixed-length **Nonce**
- One variable-length **Plaintext**
- One variable-length **Associated Data**

The goal of the project is to provide an implementation of one of these algorithms within the SEcube™ FPGA. The IP should be integrated with the existing **IP-core Manager**.

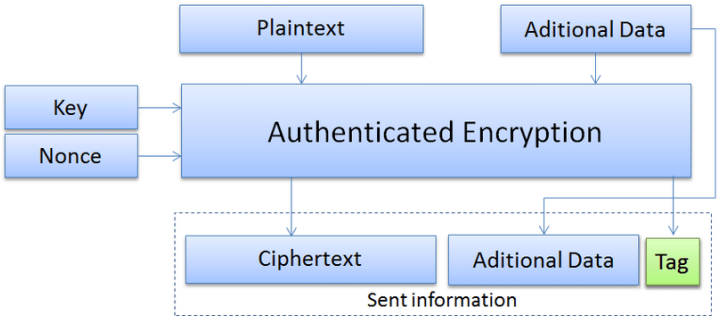


Figure 1.1: AEAD functional scheme

## 1.1 Work organization

The work has been divided in four macro parts and it has been carried out by the students depending on personal preferences and personal skills. First of all, we have analysed the algorithm and then we have discussed about possible implementations concerning the understanding of the GRAIN128-AEAD and all the properties of the FPGA embedded in the SEcube. After this part, we have divided our work to complete the remaining three parts of the project. One student handled the implementation of the Cipher Core in VHDL, performing some optimization on the core itself in order to take less space inside the FPGA. Once the Core was done, another student has developed the control part of this project also concerning about area optimization. During the development of this part, due to the complexity of this component of the system, the cooperation of all the students was required to test completely and exhaustively the component. Then, having a working version of the controller, we performed also the simulation and synthesis part. At the end another student handled the development of the API in C, necessary to test the core designed inside the FPGA. Finally, debugging was performed checking the correct behavior of the entire project uploaded inside the board, comparing the hardware results with the ”golden model” reference results from the C simulator.

# CHAPTER 2

# Grain-128AEAD Algorithm Specification

Grain-128AEAD [3] consists of two main building blocks. The first is a pre-output generator, which is constructed using a Linear Feedback Shift Register (LFSR), a Non-linear Feedback Shift Register (NFSR) and a pre-output function, while the second is an authenticator generator consisting of a shift register and an accumulator.

## 2.1 Building Blocks and Functions

The pre-output generator generates a stream of pseudo-random bits, which are used for encryption and the authentication tag. It is depicted in Fig. 2.1.

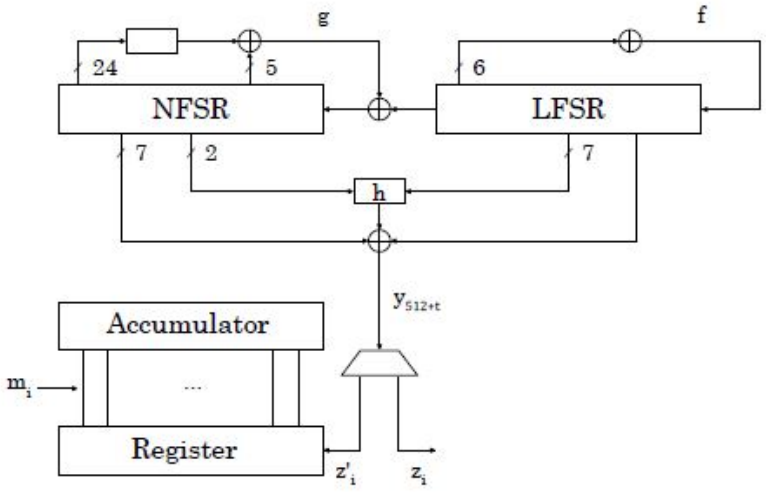


Figure 2.1: Building blocks in Grain-128AEAD

The content of the 128-bit LFSR is denoted $S_t = [s_0^t, s_1^t, ..., s_{127}^t]$ and the content of the 128-bit NFSR is similarly denoted $B_t = [b_0^t, b_1^t, ..., b_{127}^t]$. These two shift registers represent the 256-bit state of the pre-output generator.

The primitive feedback polynomial of the LFSR $f(x)$ is defined as

$$f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}$$

.

The corresponding update function of the LFSR is given by

$$s_{127}^{t+1} = s_0^t + s_7^t + s_{38}^t + s_{70}^t + s_{81}^t + s_{96}^t = \mathcal{L}(S_t)$$

The nonlinear feedback polynomial of the NFSR $g(x)$ is defined as:

$$\begin{aligned} g(x) = 1 + x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44}x^{60} + x^{61}x^{125} \\ + x^{63}x^{67} + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117} \\ + x^{46}x^{50}x^{58} + x^{103}x^{104}x^{106} + x^{33}x^{35}x^{36}x^{40} \end{aligned} \tag{2.1}$$

and the corresponding update function is given by

$$\begin{aligned} b_{127}^{t+1} = s_0^t + b_0^t + b_{26}^t + b_{56}^t + b_{91}^t + b_{96}^t + b_3^t b_{67}^t + b_{11}^t b_{13}^t \\ + b_{17}^t b_{18}^t + b_{27}^t b_{59}^t + b_{61}^t b_{65}^t + b_{68}^t b_{84}^t \\ + b_{22}^t b_{24}^t b_{25}^t + b_{70}^t b_{78}^t b_{82}^t + b_{88}^t b_{92}^t b_{93}^t b_{95}^t \\ = s_0^t + \mathcal{F}(B_t) \end{aligned} \tag{2.2}$$

Nine state variables are taken as input to a Boolean function $h(x)$. Two from the NFSR and seven from the LFSR:

$$h(s, b) = b_{12}^t s_8^t + s_{13}^t s_{20}^t + b_{95}^t s_{42}^t + s_{60}^t s_{79}^t + b_{12}^t s_8^t s_{94}^t \tag{2.3}$$

The output of the pre-output generator is then given by:

$$y_t = h(x) + s_{93}^t + \sum_{j \in \mathcal{A}} b_j^t \tag{2.4}$$

where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$.

The authenticator generator consists of a shift register, holding the most recent 64 odd bits from the pre-output, and an accumulator, both of 64 bits. We denote the content of the accumulator at instance is as $A_i$ and the shift register as $R_i$.

## 2.2 Key and Nonce initialization

Before the pre-output can be used as keystream and for authentication, the internal state of the pre-output generator and the authenticator registers are initialized with the key and nonce. The algorithm requires a Key of 128 bits, and an Initialization Vector (IV) of 96 bits.

The 128 NFSR bits are loaded with the bits of the Key $b_i^0 = k_i$ with $0 \leq i \leq 127$.
The 128 LFSR bits are loaded with the bits of the IV $s_i^0 = IV_i$ with $0 \leq i \leq 95$, the other 31 bits at 1 and the last at 0.

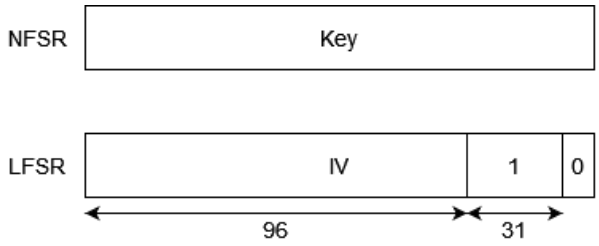


Figure 2.2: Initial state of the pre-output generator registers

Then, the cipher is clocked 256 times, feeding back the pre-output function and XORing it with the input to both the LFSR and the NFSR:

$$\begin{aligned} s_{127}^{t+1} &= \mathcal{L}(\mathcal{S}_{\sqcup}) + y_t, 0 \leq t \leq 255 \\ b_{127}^{t+1} &= s_0^t + \mathcal{F}(\mathcal{B}_{\sqcup}) + y_t, 0 \leq t \leq 255 \end{aligned} \tag{2.5}$$

At this point the pre-output generator has been initializated. The authenticator generator is now initialized by loading the register and the accumulator with the pre-output keystream as:

$$\begin{aligned} a_j^0 &= y_{256+j}, 0 \leq j \leq 63 \\ r_j^0 &= y_{320+j}, 0 \leq j \leq 63 \end{aligned} \tag{2.6}$$

Simultaneously, the key is shifted into the LFSR and the NFSR is updated:

$$\begin{aligned} s_{127}^{t+1} &= \mathcal{L}(\mathcal{S}_{\sqcup}) + k_{t-256}, 256 \leq t \leq 383 \\ b_{127}^{t+1} &= s_0^t + \mathcal{F}(\lfloor_{\sqcup}), 256 \leq t \leq 383 \end{aligned} \tag{2.7}$$

At this point the cipher is fully initialized and ready to be operative.

## 2.3 Operative Mode

For a message **m** of length L, $m = m_0, m_1, ..., m_{L-1}$, set $m_L = 1$ as padding in order to ensure that **m** and **m**||0 have different tags. After initialization, the pre-output is used to generate keystream bits $z_i$, for encryption and authentication bits $z'_i$ to update the register in the accumulator generator. The keystream is generated as:

$$z_i = y_{384+2i} \tag{2.8}$$

This means that every **even bit** from the pre-output generator is taken as a keystream bit. Viceversa:

$$z'_i = y_{384+2i+1} \tag{2.9}$$

every **odd bit** is taken as an authentication bit. The message is encrypted as:

$$c_i = m_i \oplus z_i, 0 \leq i \leq L \tag{2.10}$$

The accumulator is updated as:

$$a_j^{i+1} = a_j^i + m_i r_j^i, 0 \leq j \leq 63, 0 \leq i \leq L \tag{2.11}$$

and the shift register is updated as

$$r_{63}^{i+1} = z'_i, r_j^{i+1} = r_{j+1}^i, 0 \leq j \leq 63 \tag{2.12}$$

## 2.4 Keystream Limitation

Grain stream ciphers have been designed to allow for encrypting large chuncks of data using the same key/nonce pair. Grain-128AEAD restricts the number of keystream bits for each pair to $2^{81}$. Thus, the number of pre-output bits that can be generated under one pair is $2^{80} bit = 2^{77} bytes \approx 128 ZiB$.

## 2.5 Authenticated Encryption with Associated Data

An AEAD scheme allows for data that is authenticated, but unencrypted. Grain-128AEAD achieves this simply by explicitly forcing $y_{384+2i}$ to zero for bits that should

not be encrypted, but should still be authenticated. This means that it is possible to control the associated data on bit level, and this data can appear anywhere in the message. In more detail, we define an AEAD mask, denoted

$$d = d_0, d_1, ..., d_{L-1} \tag{2.13}$$

which specifies which bits should be encrypted. If the mask contains only ones (no associated data), the encryption is done as given above. But if the AEAD mask contains zeros, then the encryption is instead done as

$$c_i = m_i \oplus z_i \cdot d_i, 0 \leq i < L \tag{2.14}$$

and decryption in the obvious way. The use of such a mask includes great flexibility. It allows tp have unencrypted data in any positions of a packet. This provides flexibility in protocol design. The small cost to pay for this, is the fact that the cipher is producing some pre-output bits that are not used.

# CHAPTER 3

# From Specification to Software Implementation

The core Grain-128AEAD algorithm described in the previous chapter was developed inside the *Lightweight Cryptography NIST call for algorithm*. For this reason the first step of the implementation was the translation from the mathematical domain to a real software implementation. In order to standardize the selection process, NIST proposed two simple API to communicate with the cipher: **encrypt_message** and **decrypt_message**.
The encryption phase produces both the ciphertext and a **Message Authentication Code** of 64 bits, that will be used by the receiver to authenticate and then decrypt.
Due to the byte-oriented nature of the NIST API, the selected programming language was **C** due to its dependability and proximity to the real hardware implementation.

## 3.1 Encryption and MAC Generation

The encryption function requires as input:

- **Key** of 128 bits
- **Initialization Vector** of 96 bits
- **Message** of variable length
- **Message length**
- **Associated Data**
- **Associated Data length**

and produces as output the **ciphertext** of the same lenght of the input message, and the MAC that will be used by the receiver to authenticate.

**Algorithm 1** Encrypt

**Require:** $key, iv, msg, msglen, ad, adlen$
**Ensure:** $ct$
1: Initialize generator with key and iv
2: Construct $m^{'} = adlen||ad||msg||0x80$
3: Let $M =$ bit lenght of $adlen||ad$
4: **while** $i < M + mlen - 1$ **do**
5: $d_i = 0, 0 \leq i \leq M - 1$
6: $d_i = 1, M \leq i \leq M + mlen - 1$
7: Encrypt using $c_i^{'} = m_i^{'} \oplus z_i d_i$
8: Authenticate using $z_i^{'}$ and generate $A_{M+mlen+1}$
9: **end while**
10: return $ct = c_i^{'}||A_{M+mlen+1}$

## 3.2 Decryption and MAC Verification

On the receiver side, the MAC has to be verified once the decryption has been performed. The receiver knows the ciphertext length as well as the associated data lenght. It requires the as input:

The encryption function requires as input:

- **Key** of 128 bits
- **Initialization Vector** of 96 bits
- **Ciphertext** of variable length
- **Ciphertext length**
- **Associated Data**
- **Associated Data length**

If the authentication goes well (i.e. the same MAC is generated by the cipher) the function returns the decrypted message, otherwise the message is discarded.

## 3.3 C simulator

The entire cipher has been described and implemented in the **grain128aead** set of files. A struct called *grain_state* is used to have the state of the cipher at a given state:

**Algorithm 2** Decrypt

**Require:** $key, iv, ct, ctlen, ad, adlen$
**Ensure:** $ct$

Initialize generator with key and iv
Construct $c^{'} = adlen||ad||ct_0, ..., ct_{ctlen-65}||0x80$
Let $M =$ bit lenght of $adlen||ad$
Let $mlen = ctlen - 64$
**while** $i < M + mlen - 1$ **do**
    $d_i = 0, 0 \leq i \leq M - 1$
    $d_i = 1, M \leq i \leq M + mlen - 1$
    Decrypt using $m_i^{'} = ct_i^{'} \oplus z_i d_i$
    Authenticate using $z_i^{'}$ and generate $A_{M+mlen+1}$
**end while**
**if** $(ct_{ctlen-64}, ..., ct_{ctlen-1}) == A_{M+mlen+1}$ **then**
    return $m = m_M^{'}, ..., m_{M+mlen-1}^{'}$
**else**
    return $-1$
**end if**

```
typedef struct {
  unsigned char lfsr[128];
  unsigned char nfsr[128];
  unsigned char auth_acc[64];
  unsigned char auth_sr[64];
} grain_state;
```

The output values of the different shift registers and blocks are computed by some dedicated functions. In detail:

```
unsigned char next_lfsr_fb();
unsigned char next_nfsr_fb();
unsigned char next_h();

//Generic shift
unsigned char shift(unsigned char fsr[128], unsigned char fb);
//Authentication SR shift
void auth_shift(unsigned char fb);
//Bit from the pre-output generator
unsigned char next_z(unsigned char keybit);
```

As said before, the API consists of only two functions: the encryption one starts by swapping the most significant bits for each byte of the input variables (i.e. 0x01 becomes 0x80). After that the *grain_init*() function is called, which initializes the LFSR and NFSR with the IV and Key, then as described in the algorithm specification the pre-output generator is let run for 256 times.

Finally, the Authentication registers are loaded with the Key bits, specifically the Accumulator with the first 64 bits, the Shift Register with the last 64.
The last thing we need to do before accumulating the encryption operations is to concatenate the lenght of the Associated Data with the Associated Data itself.

From here, we are ready to pass the bitstream to the cipher. At first the AD length concatenated with the AD is passed, shifting in the Authentication SR the pre-output bit during the odd clock cycles and accumulating if the stream bit is 1. In this way the Associated Data is used just to accumulate the TAG.

Then, the message bitstream is passed, shifting in the Authentication SR the pre-output bit during the odd clock cycles accumulating if the stream bit is 1 (as done before), but in addition during the even clock cycles, the ciphertext is generated by xor-ing the message bit with the pre-output bit. A final accumulation is done for the padding bit, which is always 1.

The last thing the cipher does is to download the MAC, that is stored in the Authentication Accumulator register, and concatenate it to the ciphertext. The decryption function does the same things of the previous one, but at the end of the decryption phase compares the received MAC with the generated one. If they are the same the decrypted message is returned, otherwise a blank message is returned.

# CHAPTER 4

# Implementation on the SEcube™ FPGA

One of the main difference between the algorithm specification and the real implementation is the memory addressing: the stream cipher should be able to encrypt/decrypt any quantity of bits, without other constraints. But in the real implementation we faced the problem of the atomicity, since the smallest amount of data that we can read or write from a memory (without using shifting techniques) is the byte. Also, it's very rare that a digital system produces a stream of bits without following a protocol. For this reason the real implementation is byte-oriented, and the padding bit that is needed at the end of the message is now a byte 0x80.

## 4.1 Cipher core

The hardware description of the cipher core resembles the C implementation. It implements a small Finite State Machine that given the wanted operation, executes it in 3 clock cycles. The choosen architecture type for the entity is behavioral, since the internal registers are declared as *std_logic_vector* and the operations are directly computed inside the states. The only process of the core implements the FSM which goes to an OFF state if the start signal is not asserted, otherwise it decodes the operation signal and moves to the desired state.

It is important to underline that the cipher when switching from the OFF state to the SELECT_OP state latches its inputs, in this way the external controller does not need to hold them for more than one clock cycle.

Regarding data, the cipher core provides one input parallel port of 16-bits that is used to load the IV and the KEY, and one output parallel port of 16-bits used to download the MAC. An address signal is required to specify which nibble of the 128 bits we are passing to the cipher, and so it is on $\log_2 \frac{128}{16} = 3$ bits. It also offers two serial ports used for the encryption/decryption phases.

For controlling the execution it receives beyond the standard clock and reset signals a 3-bit Operation signal, a 2-bit grain_round to select the cipher phase in which we

are, and produces two output signals Completed and Busy that are asserted when the cipher is busy or when it is free and ready to start a new operation.

The **operation** signal can select 7 different operations:

- **LOAD_KEY**[000]: load 16-bit from the input parallel port into the nibble of the key addressed by the data_16_addr_in signal
- **LOAD_IV**[001]: load 16-bit from the input parallel port into the nibble of the iv addressed by the data_16_addr_in signal
- **NEXT_Z**[010]: generates the pre-output bit depending on the grain_state
- **LOAD_AUTH_ACC**[011]: shift in the serial bit into the Authentication Accumulator register
- **LOAD_AUTH_SR**[100]: shift in the serial bit into the Authentication SR register
- **ACCUMULATE**[101]: load into the Authentication Accumulator register the XOR between itself and the Authentication SR.
- **READ_AUTH_ACC**[110]: puts on the output parallel port the 16-bit nibble of the Authentication Accumulator register addressed by the address signal. It is used to retrive the MAC.

The **grain_round** signal is used to specify in which phase of the flow we are. Specifically:

- **INIT**[00]: initialization of the core, the IV and the KEY are being loaded in the register
- **ADDKEY**[01]: the key is loaded into the Authentication registers
- **NORMAL**[10]: the cipher is now encrypting/decrypting

A testbench file for the raw cipher core is included. It tests the encryption of a message with the following inputs:

- Key: 0x0123456789abcdef123456789abcdef0
- IV: 0x0123456789abcdef12345678
- Message: 0x6369616f6e652121
- AD: 0x1111

And produces as output ciphertext 0x475fcba7b19681db31e4dcfeabe4d624 , which is the same generated by the C simulator.

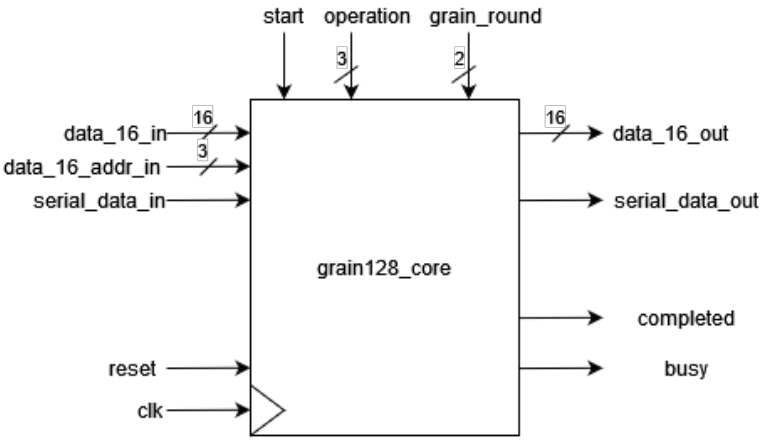


Figure 4.1: Cipher core block scheme

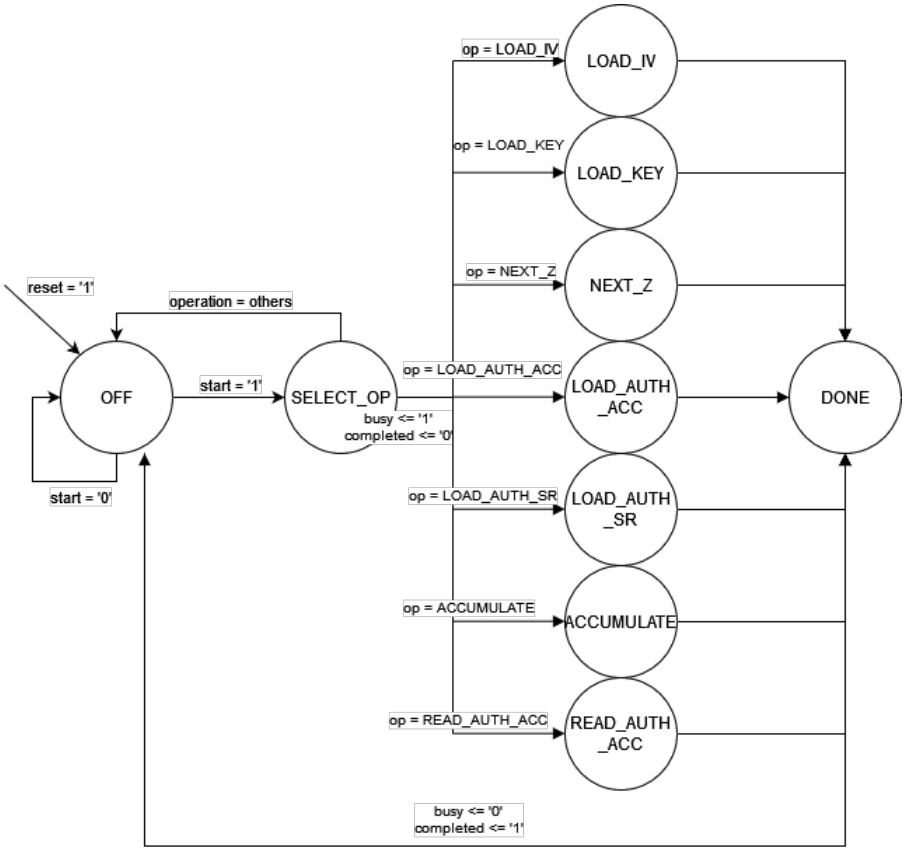


Figure 4.2: Cipher core FSM

## 4.2 Controller

For the control part of this system, a controller has been developed in order to dominate the core that implements the Grain-128AEAD algorithm. The controller also handle the flow of data to be exchanged between the core itself and the CPU, exploiting the presence of the IP Manager.

this component is based on a classic Finite State Machine (or just FSM) with encoded states. This design option has been taken due to the reduced number of possible operations and as a consequence, a reduced state flow. Implementing the controller as a micro-programmed FSM would not have been the best solution considering also the fact that the space available to program the FPGA is very limited due to its dimension.

The controller is composed of 105 states in total, that can be divided in 4 macro groups:

- **OFF** state with the initialization of all the registers and then other 22 states used for **READING** all the necessary data sequences coming from the data buffer.
- **Core Initialization** that is composed by 23 states
- **Encryption/Decryption** that includes 43 states of the FSM in order to encrypt or decrypt the message.
- **Output** phase made by 17 states in order to put the results into the data buffer.

### 4.2.1 Packet structure

Before describing each macro block in detail, first it is necessary to introduce how the packet is organized and written in the data buffer. The packet has two different structures depending on which is the state of the cipher core. The first structure on the left shows how the packet is organized when the cipher has to be initialized and it is used both for encryption and decryption. The Control Word occupies the first row (or the first 16 bits) of the packet and as a consequence it is written on the first address of the data buffer. According to the specification of the algorithm, the other parameters (128-bits for the Key and 96-bits for the IV) are written below. For the **Associated Data** and the **Message**, a maximum length of 10 and 12 words has been decided respectively. The last 4 words of the message are reserved for the authentication (MAC).

The second structure on the right, shows how the packet is organized when the cipher has been already initialized and it is necessary only to compute the encryption or decryption of the message. This operation is based on which Control word is passed to the cipher core, that it is written again in the first row of the packet. Then, in this configuration the next row includes a parameter that informs the cipher about the length of the message. The message is then written with maximum length of 12 words

for each packet. As in the previous structure, after 12 words of message there are 4 words reserved for the authentication (MAC).

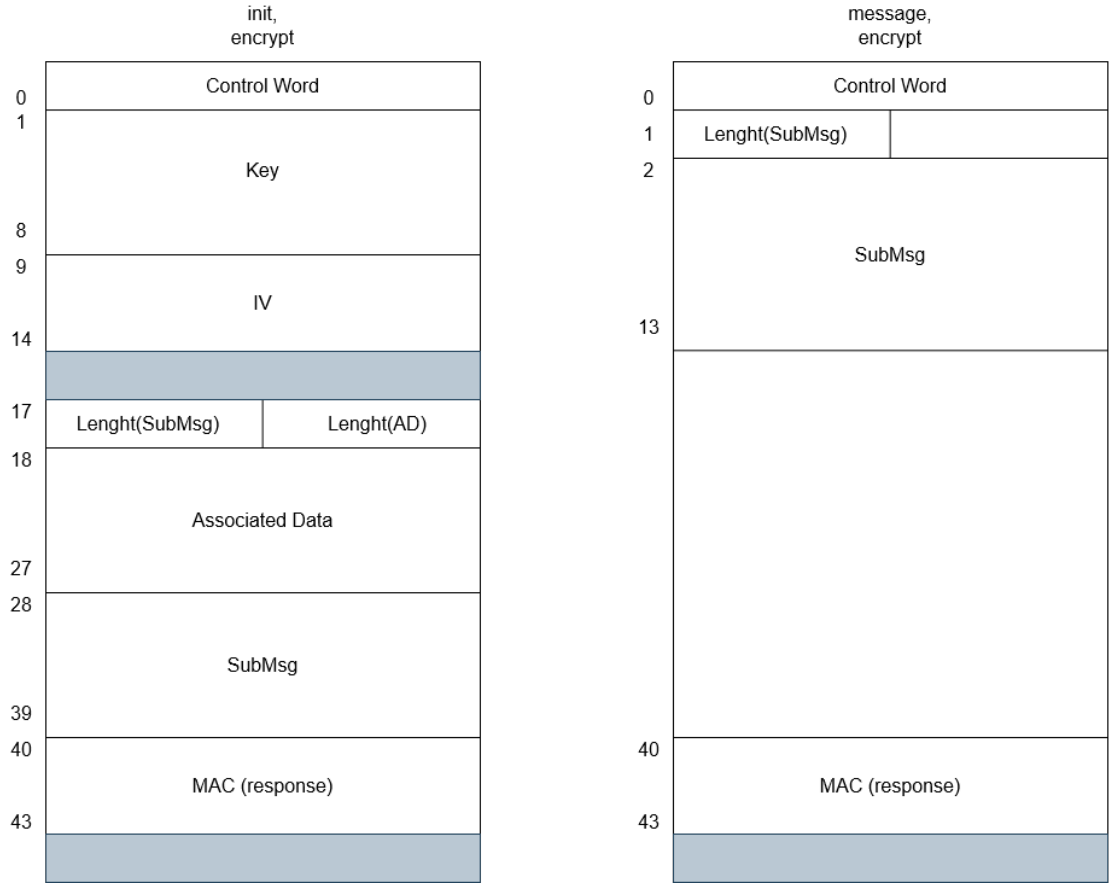


Figure 4.3: Encrypt packets structure

### 4.2.2 OFF and READING states

In these states, the system has been set up, initializing all the vectors and memories instantiated to make the cipher works in a proper manner.

The first data that has to be read on the data buffer is the Control word. Once it has been sampled, the systems moves on a state called **DECODE_OPCODE**. In this state, depending on the decoded value of the control world, the controller has too choose between 4 different operations:

- Initialization of the system and encryption of the plaintext,
- Encryption of the message with the system already been initialized,
- Initialization of the system and decryption of the ciphertext,

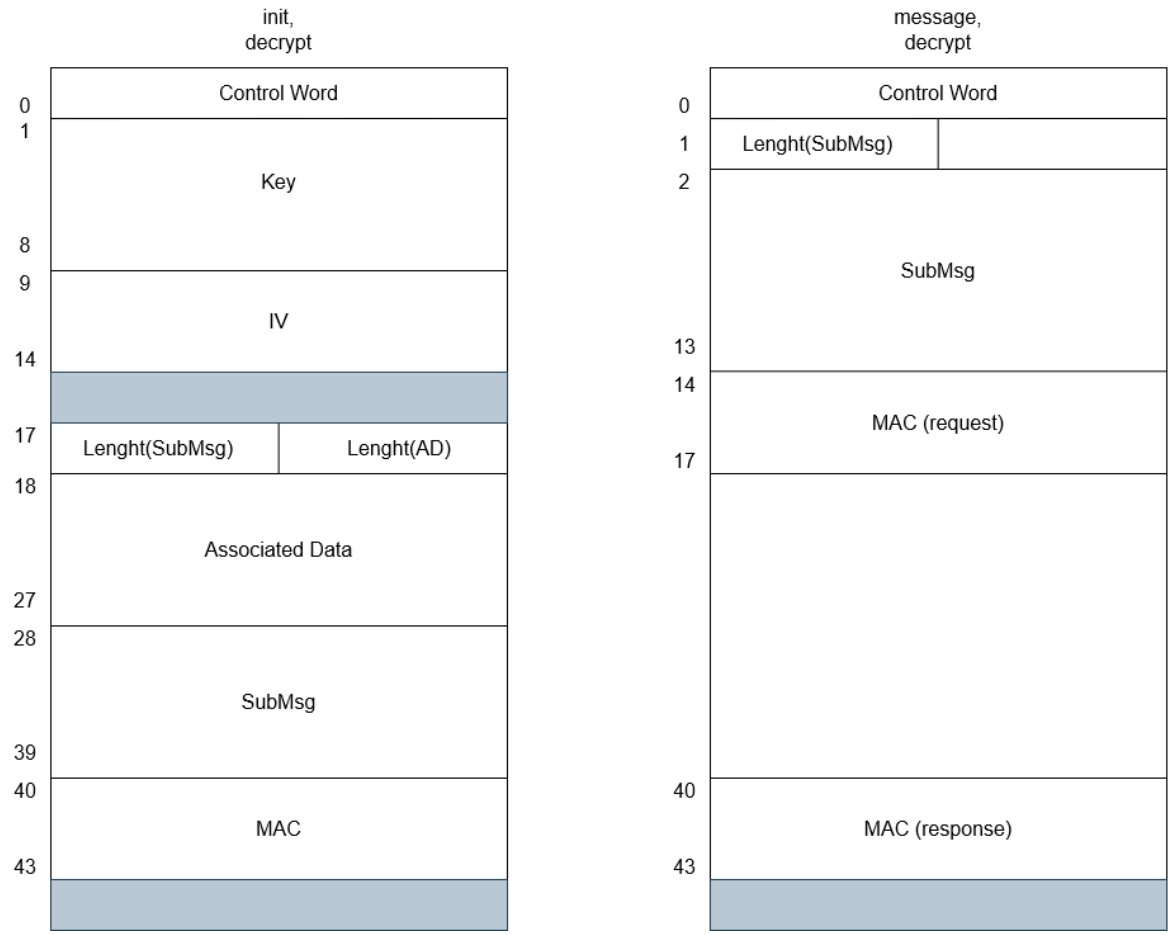


Figure 4.4: Decrypt packets structure

- Decryption of the ciphertext with the system already been initialized.

Moving on, the next states are described in order to load the parameters required by the algorithm. In case of the initialization operation, the first parameter that has to be read in the data buffer is the **KEY**. After that, we read in order the **Initialization Vector**, the two parameters that inform the system about the length of the Associated data and the message. At the end, the states needed to read exactly the **Associated Data** and the **Message** are performed.

The states are described in such a way, they follow the same pattern in order to load correctly the data sequence. The structure is made by the first state of **waiting** that waits for the buffer writing the data. A second state called **Address_parameter** is created in order to stop the system for one clock cycle for having the values updated. A third stage called **Read_parameter** is used to write the data, read previously in the data buffer, inside the signal or the memory used for that parameter.

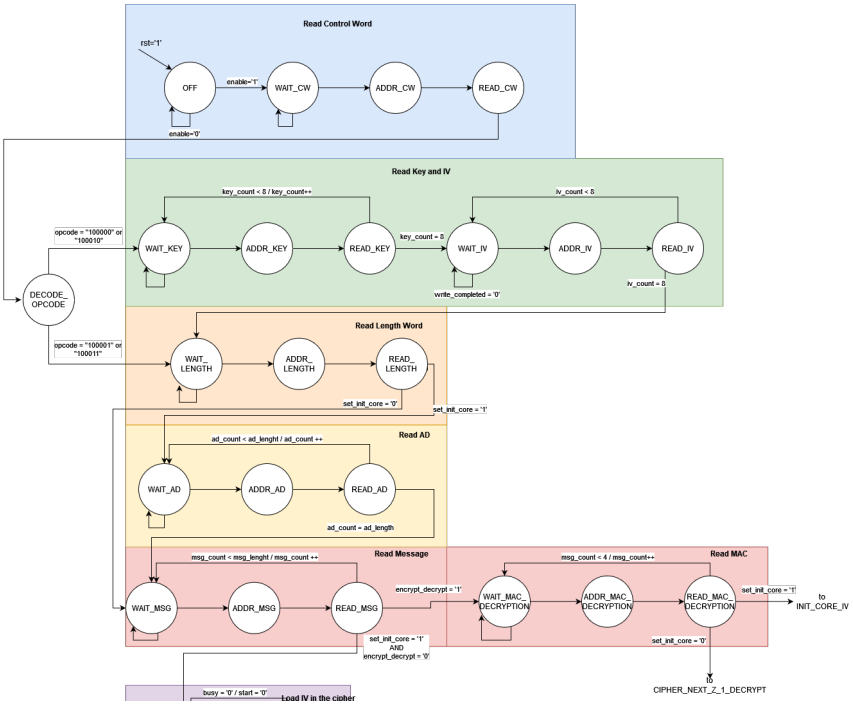


Figure 4.5: Reading states

### 4.2.3 Core initialization states

After all the data sequences have been collected, depending on the Control word, the controller starts to pass data to the Cipher Core in order to initialize it if the system requires an initialization.

This sequence of states includes all the steps needed by the cipher core to load the parameters needed such as: the Initialization Vector or just **IV** and the **KEY**. After IV and KEY have been loaded into the cipher core, the generation of the pre-output is performed making a loop of 255 clock cycle in the **preoutput states** in order to feed properly the NFSR and the LFSR instantiated inside the core cipher. Once the two Linear and Non-linear feedback shift register have been loaded, the FSM moves to the initialization of the Accumulation register and the shift register. The same pattern of sequence of states can be found for both operations. The last 8 states of this macro group, perform the generation of the authentication also called **TAG** as it is described in the specification of the algorithm.

### 4.2.4 Encryption/Decryption states

At this point of the flow, the encryption or decryption is executed depending on the decoding of the OPCODE. Starting from the encryption states, the controller loops many times depending on the length of the message in order to generate the NEXT_Z and do the XOR operation between the NEXT_Z itself and the message. States are

described in such a way this operation is performed bit by bit and accordingly to the specification of the algorithm, every even bit of the message is taken to perform the XOR operation and if the result. At each step of these execution the accumulation register is updated when, the value of the bit of the message taken in consideration for the XOR, is true and the shift register is updated anyway. For the decryption, the states are described in a very similar way. The difference is that states loops depending on the length of the message to be decrypted. In order to write on the data buffer the message decrypted, the control that checks the authentication, is done in the next states.

### 4.2.5 Output states

In this macro group the first operation executed by the controller is the comparison for the authentication in order to obtain the message decrypted. Then, the states to write on the output the ciphertext, are described. Considering that fact that a RAM memory has been used to contain the ciphertext, some states are used to stops the controller for a clock cycle in order to perform a reading operation from the RAM and writing the decrypted or the encrypted message on the data buffer. After all, the MAC written in the reserved four words of the packet is taken to be passed on the data buffer. Considering the fact that the protocol used for writing data from the cipher to the CPU is **Polling**, we have a terminal state called **UPDATE_STATE** where the controller writes the last word equal to **0xFFFF** in order to communicate that the transaction is ended, accordingly to the behavior of the system. Finally the controller returns to the OFF state.

## 4.3 Synthesis

The integration of the cipher with the IP Manager requires to declare it in the TOP entity file, and routing the signals from the IP Manager to our cipher. Since we have just one custom IP, the process is simple and no other values should be modified.

The synthesis process is composed by different stages. from which we can get some feedbacks. We tried different synthesis strategies in order to tune the synthesizer to our need. Basically we tried to optimize the synthesis for the area at first, and then for the timing. The results are shown in the following table:

| Synthesis report | | | | |
|---|---|---|---|---|
| Strategy,Frequency | SLICEs % | LUT4s % | Worst Slack | Worst Slack (hold) |
| Area, 45MHz | 72 | 72 | 1.665 | 0.196 |
| Balanced, 60MHz | 74 | 73 | -1.791 | 0.196 |
| Balanced, 50MHz | 74 | 73 | 0.080 | 0.177 |
| Balanced, 45MHz | 74 | 73 | 2.456 | 0.256 |
| Timing, 45MHz | 80 | 79 | 0.722 | 0.306 |

We obviously have to discard the strategy that produces a negative slack, since it would create timing errors inside the FPGA. Considering the differences, the two best strategies are the Area@45MHz and the Balanced@45MHz. This tells us that there is a problem inside our design that creates such a long critical path. This forces us to decrease the operational clock frequency of 25% from the initial target.

### 4.3.1 Area Design Optimization

During the development of the hardware design blocks, we initially decided to utilize only the LUT(Look Up Table) available inside the FPGA to allocate also memory elements and signals. After several iterations of synthesis and mapping with the Lattice Diamond tool, analysing the results, we have decided to change the design and instantiate also the RAM inside the design for storing the message and the ciphertext. This decision has been taken in order to reduce the utilization of the LUT and to exploit all the resources given by the FPGA. Another important point of this decision is that, during the analysis, we have exploited the settings of the synthesis tool in order to explore more solution in the design space prioritizing Area or Timing.

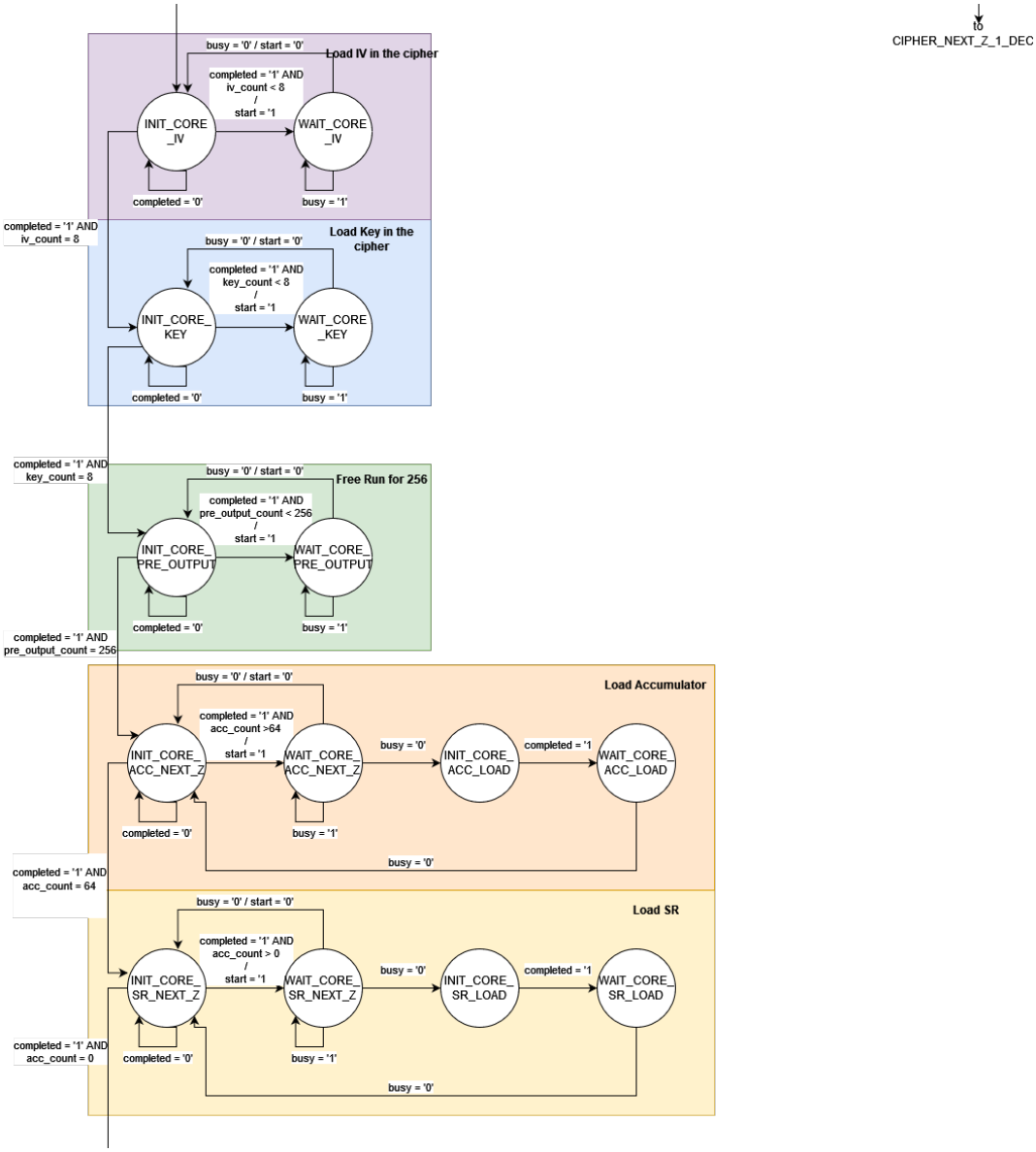


Figure 4.6: Core initialization states

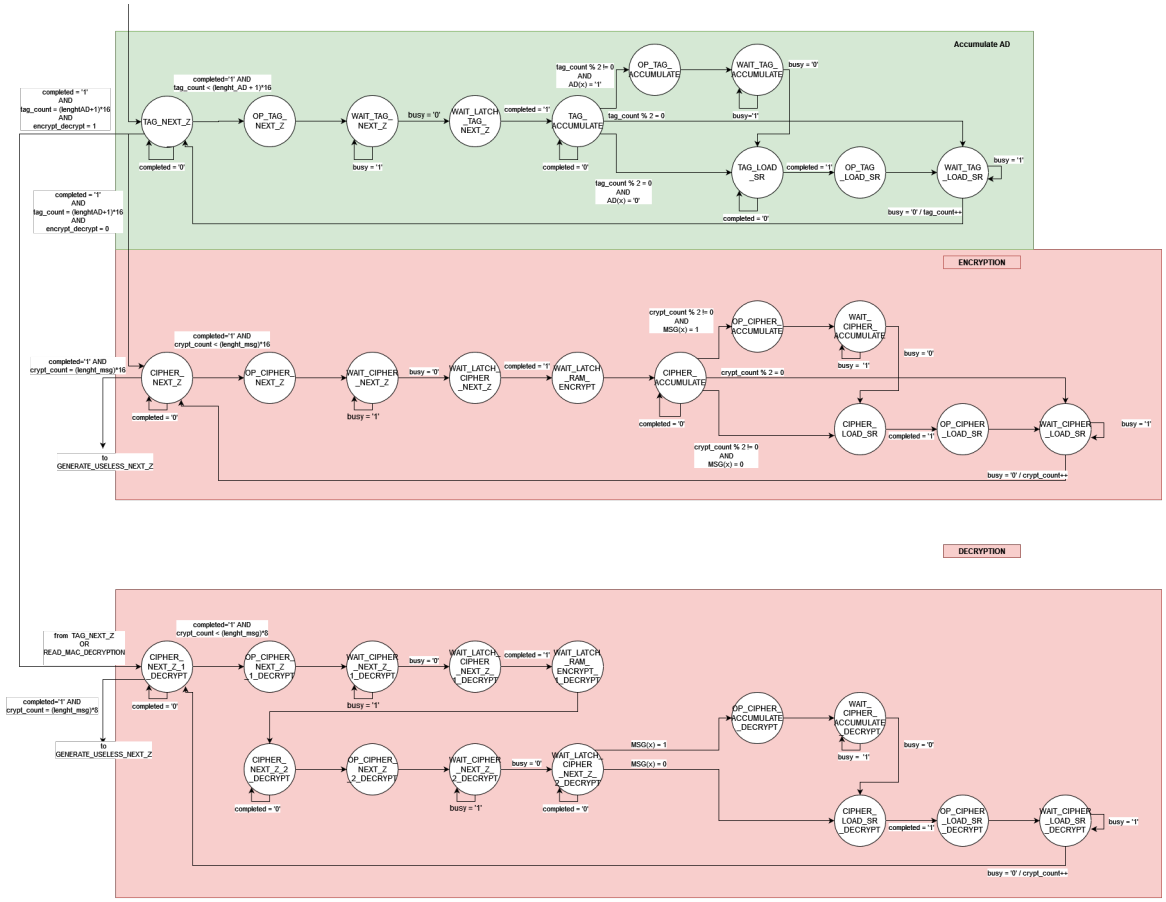


Figure 4.7: Accumulation and encryption/decryption states

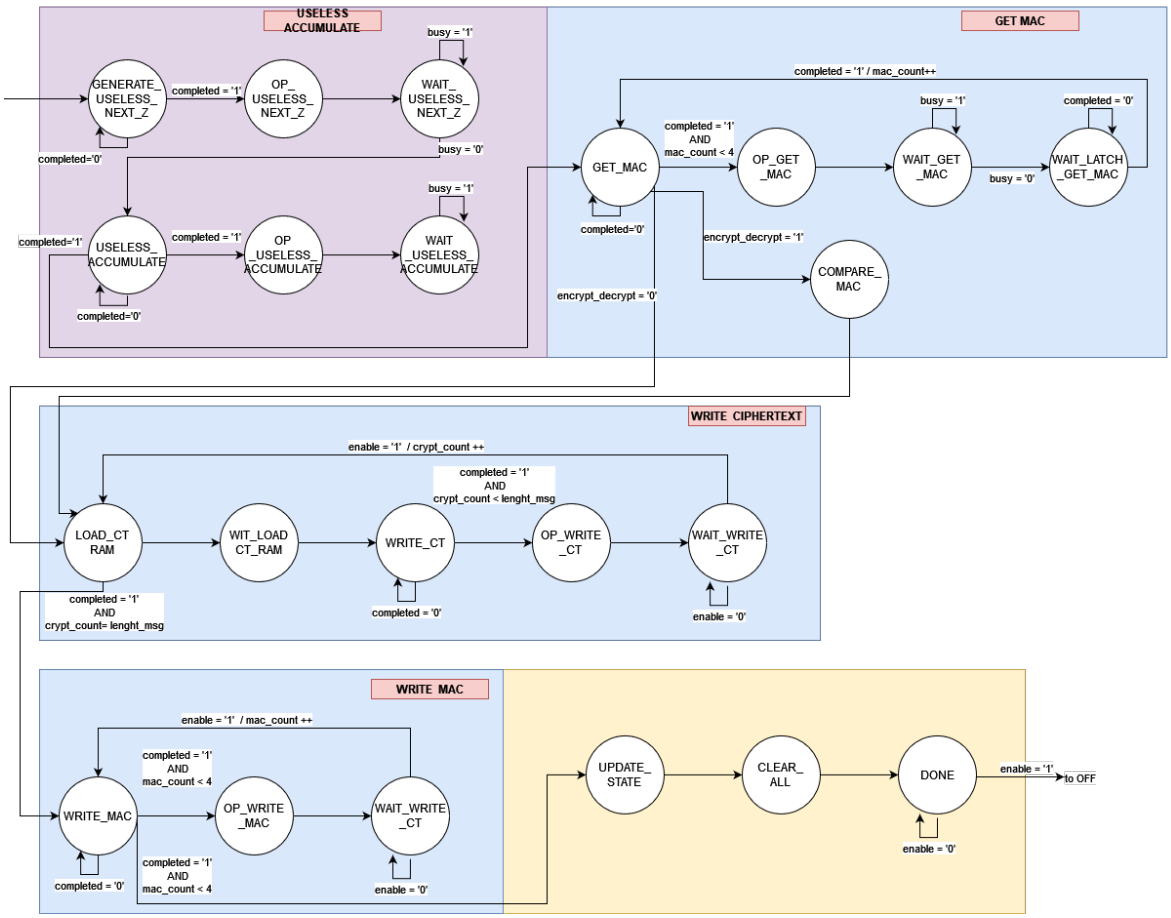


Figure 4.8: Writeback states

# CHAPTER 5

# Application Protocol Interface

This Chapter describes the communication driver developed for interfacing the user application and the Grain-128AEAD core on the FPGA. For a correct communication, the complete drivers are composed of two layers: a high-level one, composed of the specific functions for managing the creation of the packet to be sent to the IP core, and a low-level one, containing the low-level functionalities for the communication with the FPGA. These last one were written using the available functions provided by the IP manager. Before using the provided API, remember to include these two functions inside the "main.c" file, otherwise the FPGA will not be programmed well and all the low-level driver will not work correctly.

```
1 B5_FPGA_Programming();
2 FPGA_IPM_init();
```

## 5.1 High-level driver

The high-level communication with the core is implemented by the functions declared in the header file "grain128aed_fpga.h". These are based on a function called *GRAIN128AEAD_CHIPHER* that receives all the necessary data in order to manage the creation of the packets mainly depending on the length of the message/ciphertext received.
As a further detail this function uses two static functions (*GRAIN128AEAD_FPGA_init_pack* and *GRAIN128AEAD_FPGA_next_pack*) to build the two different type of packet that it has been discussed before. In this section a short remainder will be made.
The communication through packets is performed using write/read operation on Data Buffer (64x16 bits); in fact, this is the easiest way to exchange data between Firmware and IP Manager. Due to the lack of inconsistency of the data received by the functions and the data that will actually have to be processed within the FPGA, a function called *GRAIN128AEAD_FPGA_8_to_16*, in fact it convert two bytes as input in one word as output. There is also another data inconsistency regarding the writing/reading mode,

in fact ciphercore store data on memory in little endian, while API works using big endian. This brings to invert all the input data that are sent to be written in the packet and, consequentially, also all the output result data receiver from the cipher core.
A detailed description of the packets can be found in the Section 4.2.1 of this document. As it is written there, there are two types of packets:

1. **Init Packet** : It is used to send all the necessary data in order to setup the IP core. The data in charge of doing it are: *Key*, *Associated Data(AD)* and *Initialization Vector(IV)*. Furthermore it is also available a portion of memory dedicated to the data on which the user wants to perform the operation (in this case Message or Ciphertext)

2. **Next Packet** : It is used once the IP Core has been configured. It is in charge of sending the remaining data on which the user wants to perform the desired operation

For both the packets a dedicated memory location is used to handle MAC authentication.

All the API and the used funtions mentioned above are now listed:

```
GRAIN128AEAD_FPGA_RETURN_CODE GRAIN128AEAD_FPGA_encrypt (
                          uint8_t *key,
                          uint8_t *IV,
                          uint8_t *AD, uint8_t ADlen,
                          const uint8_t *msg, uint64_t msgLen,
                          uint8_t *ciphertext);
```

This is the function allow to perform the Encryption operation given all the necessary data described below:

- **key** : it indicates the key used to perform the operation;
- **IV** : it indicates the Initalized Vector useful to configure the IP core;
- **AD** : it indicates the Associated Data;
- **ADlen** : it indicates the length of the Associated Data expressed in bytes;
- **msg** : it indicates the message to be encrypted;
- **msgLen** : it indicates the length of the message expressed in bytes;
- **cithertext** : it will contain the Encryption operation result as cipertext + MAC data used for the authentication.

```
GRAIN128AEAD_FPGA_RETURN_CODE GRAIN128AEAD_FPGA_decrypt (
                          uint8_t *key,
                          uint8_t *IV,
                          uint8_t *AD, uint8_t ADlen,
                          const uint8_t *ciphertext, uint64_t cipherLen,
                          uint8_t *msg);
```

This is the function allow to perform the Decryption operation given all the necessary data described below:

- **key** : it indicates the key used to perform the operation;
- **IV** : it indicates the Initalized Vector useful to configure the IP core;
- **AD** : it indicates the Associated Data;
- **ADlen** : it indicates the length of the Associated Data expressed in bytes;
- **ciphertext** : it indicates the ciphertext to be decrypted;
- **cipherLen** : it indicates the length of the ciphertext expressed in bytes;
- **msg** : it will contain the Decryption operation result as message + MAC data used for the authentication.

```
static GRAIN128AEAD_FPGA_RETURN_CODE GRAIN128AEAD_CHIPHER(
                                uint8_t *key,
                                uint8_t *IV,
                                uint8_t *AD, uint8_t ADlen,
                                const uint8_t *dataIN,
                                uint64_t datainLen,
                                uint8_t *dataOUT, FPGA_IPM_OPCODE opcode
   );
```

This is the general function called when the user wants to perform one of the two operation describer before :

- **key** : it indicates the key used to perform the operation;
- **IV** : it indicates the Initalized Vector useful to configure the IP core;
- **AD** : it indicates the Associated Data;
- **ADlen** : it indicates the length of the Associated Data expressed in bytes;
- **dataIN** : it indicates the data on which it is performed the operation
- **datainLen** : it indicates the length of the dataIN expressed in bytes;
- **dataOUT** : it will contain the operation result with the MAC value already included;

- **opcode** : it is used to recognize which operation the function is going to be executed.

```
static void GRAIN128AEAD_FPGA_init_pack(
                    FPGA_IPM_DATA *key,
                    FPGA_IPM_DATA *iv,
                    FPGA_IPM_DATA *ad, uint8_t adLen,
                    FPGA_IPM_DATA *msg, uint8_t msgLen,
                    FPGA_IPM_DATA *res, uint8_t *i_res,
                    uint8_t readMAC, FPGA_IPM_OPCODE opcode);
```

This is the function used to create the initial packet by writing on the Data Buffer. After IP core processes trasmitted data, it will write the result on the Data Buffer. Before being written, the data input and output are inverted. This function is also in charge of reading these results and to store them :

- **key** : it indicates the key used to perform the operation;
- **IV** : it indicates the Initalized Vector useful to configure the IP core;
- **AD** : it indicates the Associated Data;
- **ADlen** : it indicates the length of the Associated Data expressed in bytes;
- **msg** : it indicates the partial data to be sent to IP core in order to start processing them
- **msgLen** : it indicates the length of the sub-message expressed in bytes;
- **res** : it will contain the operation result + eventually the MAC value if readMAC parameter is set;
- **i_res** : it is the index useful to keep track on which new result coming from IP core will be stored inside res parameter described just before;
- **readMAC** : it is a flag aims to say to the function if it will have to read and store MAC data from the Data Buffer to the res variable;
- **opcode** : it is used to indicate which operation type is going to send the packet among Encryption and Decryption.

```
static void GRAIN128AEAD_FPGA_next_pack(
                    FPGA_IPM_DATA *msg, uint8_t msgLen,
                    FPGA_IPM_DATA *res, uint8_t *i_res,
                    uint8_t readMAC, FPGA_IPM_OPCODE opcode);
```

This is the function used to create the successive packet by writing on the Data Buffer, it is assumed that the initialize packet has been already sent. After IP core processes trasmitted data, it will write the result on the Data Buffer. Before being written, the data input and output are inverted. This function is also in charge of reading these results and to store them :

- **msg** : it indicates the partial data to be sent to IP core in order to start processing them
- **msgLen** : it indicates the length of the sub-message expressed in bytes;
- **res** : it will contain the operation result + eventually the MAC value if readMAC parameter is set;
- **i_res** : it is the index useful to keep track on which new result coming from IP core will be stored inside res parameter described just before;
- **readMAC** : it is a flag aims to say to the function if it will have to read and store MAC data from the Data Buffer to the res variable;
- **opcode** : it is used to indicate which operation type is going to send the packet among Encryption and Decryption.

```
1 static FPGA_IPM_DATA GRAIN128AEAD_FPGA_8_to_16(
2                 uint8_t dataMSV, uint8_t dataLSV);
```

This is the function aims to convert two bytes as input in one word as output. it will be produced as returned value of the function.

- **dataMSV** : it is the byte to be placed on the upper part of the final result;
- **dataLSV** : it is the byte to be placed on the lower part of the final result;

## 5.2 Testing methodology

The verification phase is performed by using a golden model, written in C language, taken from the official NIST submission of the algorithm itself. Starting from it some optimization work has been done in order to fix and adjust some piece of code not totally clear from the point of view of the team project. The optimized code is available under the "c" folder of the official repository of the project here [4].
Inside the Section 7.3.4 related to the API testing will be described how to configure the golden model in order to verify the correctness of the data produce from the API side. It will be also explain how to read it in order to make a straightforward comparison.

# CHAPTER 6

# System Architecture and Behavior

This Section is meant to give an overview of the implemented design. To keep things brief, the structure will not be presented nor explained again.

## 6.1 Design overview

The complete architecture programmed inside the FPGA is finally composed by:

- Data Buffer (64x16 bits)
- IP manager
- Grain128-AEAD controller
- Grain128-AEAD core (ID = 0x01)

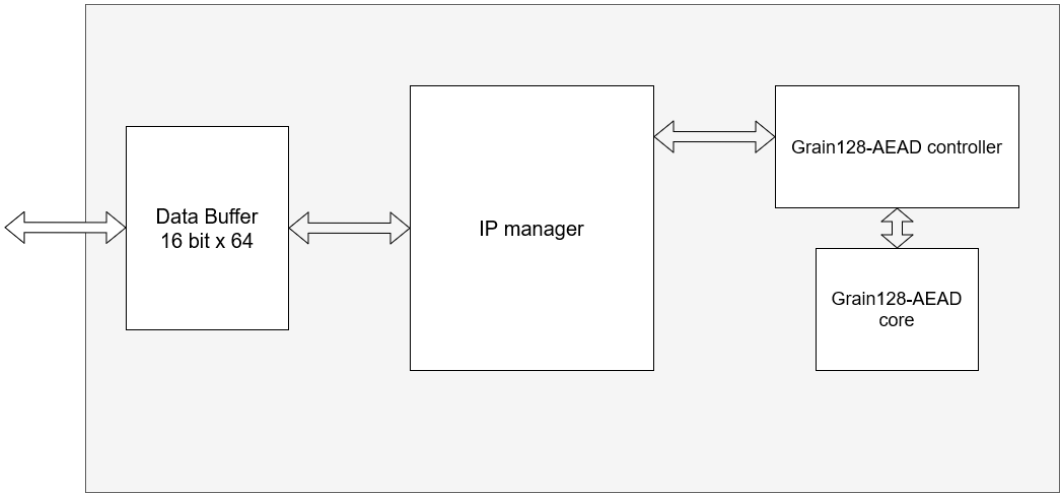


Figure 6.1: FPGA internal structure

## 6.2 CPU-FPGA communication

The communication among CPU and FPGA is done through **Polling**, a very common way of communication between a master of a computing system and a slave such as a peripheral or a coprocessor, infact the master continuously checks the status of the slave to monitor its state (which can be "something to communicate", or "nothing to communicate"). The monitoring is done classically with a continuous read of the status on a shared location, a portion of a memory or a control register. In this case the location on which monitoring activity is performed at the 64-th Data Buffer word. Below is reported how a polling transaction works [Figure 6.4].
The way on which transactions are managed are described here:

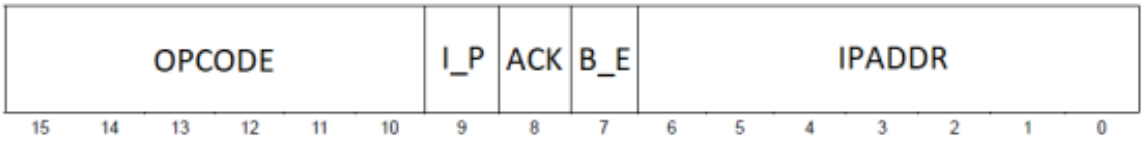


Figure 6.2: Control Word

| | | | |
|---|---|---|---|
| **Bits 15:10** | **OPCODE** | Operative Code | Hosts the possible operative code instructing the core on the task to be performed. |
| **Bit 9** | **I_P** | Communication Mode | 0 : Polling Mode<br>1 : Interrupt Mode |
| **Bit 8** | **ACK** | Acknowledgment Transaction | 0 : The transaction opened/-closed is a normal transaction<br>1 : The transaction opened/-closed acknowledges an interrupt request |
| **Bit 7** | **B_E** | Begin/End of transaction | 0 : End Transaction<br>1 : Begin Transaction |
| **Bits 6:0** | **IPADDR** | Address of the IP core | Hosts the IP core identifier. Can assume any value from 0 to 127. |

Figure 6.3: Control Word setup

All the configuration during the communication among CPU-FPGA are managed through an array 16-bits length. These bits are placed in the row 0 of the Data Buffer. For this part opcode values will be ignored, they will be explained later.
In order to open a transaction the configuration is the following:

- **I_P** : 0 (Polling Mode);
- **ACK** : 0 ;

- **B_E** : 1 (Begin Transaction);
- **IPADDR[6:0]** : 000001 (Grain128-AEAD core);

In order to close a transaction the configuration is the following:

- **I_P** : 0 (Polling Mode);
- **ACK** : 0 ;
- **B_E** : 0 (End Transaction);
- **IPADDR[6:0]** : 000001 (Grain128-AEAD core);

These configuration were simplified by using the high-driver already available inside the IP Manager files.

As anticipated before, API know the type of operation and the packet that is going to be created through an encoded opcode (6-bit length) written in the upper part of the Control Word (first memory allocation of the Data Buffer, address 0). Below it is reported the opcode for all the possible combination among operation mode and type of created packet:

| | Encryption | Decryption |
|---|---|---|
| Init packet | 000000 | 000001 |
| Next packet | 000010 | 000011 |

## 6.3 Encryption mode

Encryption is the operation that, starting from the key, Initialization Vector, Associated Data, message and its length, it will produce a concatenated set of ciphertext + MAC, depending the number of the packets used for this operation.
For performing this operation, the CPU, in order to activate the Grain128-AEAD cipher core and send to it a generic packet, needs to open a transaction with the IP manager. The only way to communicate with it is through the Data Buffer, a 16 bit x 64 words memory.
The generation of the packets takes into account the length of the message to be encrypted, considering a maximum of 24 bytes of the message that can be sent in each packet. The first packet to be sent is init_packet with OPCODE = 000000, since it contains all the necessary data for initializing the cipher core. As soon as it is written through the data buffer, the Grain128-AEAD controller will search for it and proceeds to read the content of it. A detail of the addresses to read and write data was done in the Section 4.2. Since the choosen communication is based on **Polling**, the cipher

holds the data buffer memory until the unlock value is set to the last word of it. This means that the CPU will wait for the end of the computation by looking at that word (64-th), and doing nothing until it is not set to 0xFFFF. Once the cipher produced results, the CPU read the content of the Data Buffer at the same position when the data to be computed have been written previously ( from address 28 to 39 ), and store the results. In addition the core computes also the MAC ( 8 bytes ) related to that packet and write it starting from address 40 of the data buffer. Finally the transaction is closed. If the init packet has not been able to fit all the message to be encrypted, a set of next packet will be created as long as it will be fully consumed. In this case, the opcode to be written during the opening transaction will be OPCODE = 000001. Also in this case, during the reading phase of the data written from the cipher, the MAC data is read at the same position specified before. Each ciphertext + MAC produced by each created packet will be concatenated in order to produce the final operation result.

## 6.4 Decryption mode

Decryption is the operation that, starting from the key, Initialization Vector, Associated Data it will produce the result starting from an input done by a set of ciphertext + MAC. The final result should be equal to the same message used during encryption operation. In this case, during the creation of all the packet, a single combination of ciphertext + MAC (up to 32 bytes) will be written on the data buffer and a comparison between the MAC computed to the Grain128-AEAD core and the MAC received must be done in order to guarantee the authenticity property. In case of mismatch the cipher core will return an empty result.
The operation works as before excepted for the different OPCODE during the different packet creation (see Table above).

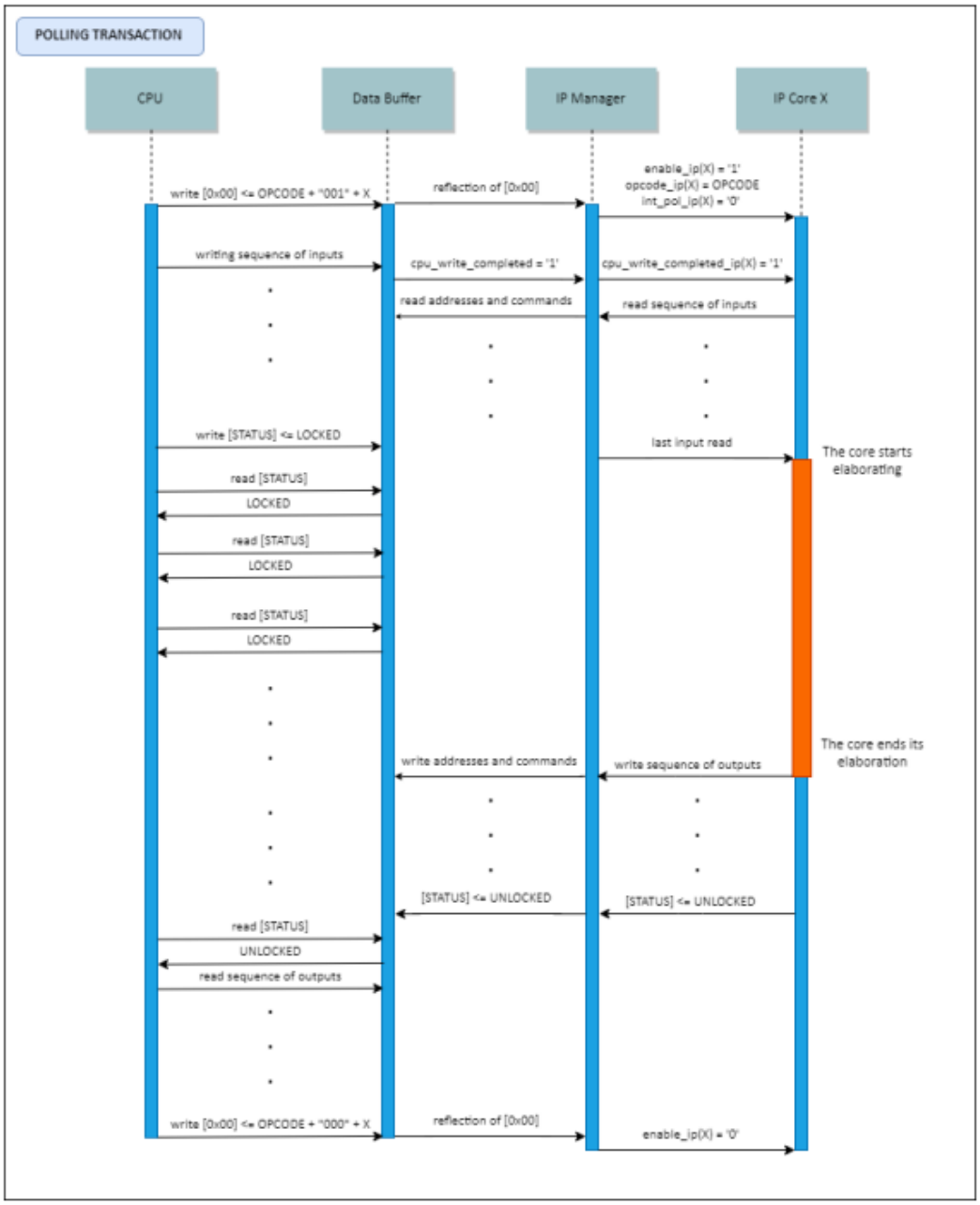


Figure 6.4: Polling transaction between the CPU and Grain128-AEAD core

# CHAPTER 7

# User Manual

The following Chapter aims to explain how to use the core and their related API, starting from their insertion in the development environment. The integration of both the IP core and API files is done starting from the basic SEcube-SDK.
All the following steps will be performed on *STM32CubeIDE* software.

## 7.1 SEcube-SDK Installation

In order to download and setup the right development environment to work on, please follow these steps:

1. Download the desired compressed archive at the URL available here [2], under *SECube SDK 1.5.1* release on the right part of the web page.
2. Unzip the compressed archive.
3. Open *STM32cubeIDE* software.
4. Go under "File » Import » General » Existing project under workspace" and click on it.
5. Click on *Browse* button and go to "SEcube-SDK-1.5.1 » SEcube USBStick Firmware » Project » STM32CubeIDE" and select the USBStick folder.
6. Check the selected project and click on "Finish" button.

## 7.2 Core Installation

Once you setup your development environment you need to import three different cores: IP Manager, Data Buffer and Grain128-AEAD core inside your working environment. To do so, please follow these steps:

1. Download the files related to the IP Manager and the data buffer from the Github repository available here [1]. In order to use the correct files in the next steps, please download the release 007.
   If you don't find this release you can also download all the required file from the official project repository on GitHub here [4]. The procedure is very straightforward since the files are already split depending on their extension. Some drivers are also available.

2. Starting from the project previously imported, following the steps that are listed in the IP manager documentation in order to well integrate both IP manager and Data Buffer. Be sure to add all the necessary files to communicate with the FPGA.

3. Since the Grain128-AEAD is not able to work at the frequency of 60 MHz, please open Fpgaipm.c file under "src/Device" folder and modify the second parameter of __HAL_RCC_MCO1_CONFIG function (line 223) by putting a prescalar of RCC_MCODIV_4 value.

```
__HAL_RCC_MCO1_CONFIG(RCC_MCO1SOURCE_PLLCLK, RCC_MCODIV_4);
```

   This value will bring the working frequency at 45 MHz.

4. Due to the lowering of the frequency some parameters inside "Fpgaipm.c" must be done in order to guarantee the correct functioning of the IP manager. The modification to be done are the following:

```
Timing.AddressSetupTime = 8;
Timing.DataSetupTime = 8;

ExtTiming.AddressSetupTime = 8;
ExtTiming.DataSetupTime = 8;
```

5. To import some files in your project you have to select "File » Import » General » File System" and then search for the files you want to include.

6. Download the files related to the Grain-128AEAD IP core from the Github repository available here [4]. You can find a folder called "API", where inside you have to look only on "inc" and "src" folders. In particular the files to be imported are called "grain128aead_fpga.c" and "grain128aead_fpga.h". Be sure to import all the files in the correct project folders.

7. Inside the repository [4] it is also available the bitstream of all the cores, described at the beginning of the section, already synthesized. It is placed in the file called "TEST_FPGA.h" under "API/inc" folder, please include it inside "Inc/Device" folder of the project. Finally, open "FPGA.c" inside "Inc" folder and set this value:

```
uint32_t g_iAlgoSize = 129857;
uint32_t g_iDataSize = 239514;
```

Now the cores are correctly inserted in the working environment. You can now create your own program. Be sure to add the API call inside the file "main.c".
After the modification in your project, if you want to program the board, please follow these simple steps:

1. Save the changes to all modified files
2. Go to "project ≫ Build Configuration ≫ Set Active" and ensure the tick is on "Release"
3. Build the project
4. Connect the SEcube to the PC
5. Flash the produced executable on the device by right-clicking on it in the Project Explorer and selecting the Release binary under "Target ≫ Program Chip". (select the label containing the string "/Release")

When the process starts, the LEDs associated to the FPGA will be set in a weak pull-up state, meaning that the FPGA is being programmed. At the end, they should turn off. As soon as they turn off, the program you wrote starts executing.

## 7.3 Testing Grain128-AEAD

One possible way to test the developed API is through UART connection. Furthermore, as it has been described in the Section 5.1, the verification of the Grain-128AEAD results will be done by using a golden model of the same algorithm written in C language.
A possible testing environment is available under the folder test of the official repository of the project available here [4]. Inside this folder you can find the following files:

1. The configuration of the usart that will be used to show the result to the end-user
2. The "main.c" file where all the FPGA configuration and API call are already present
3. The "test_grain.c" and "test_grain.h" used to handle the output data on the UART console. They need to be imported together with the "main.c" file as explained before
4. The "LWC_AEAD_KAT_128_96.txt" file containing some working example of the provided golden model for both encryption and decryption phase

These files will be used to explain how the testing phase has been perfomed.

### 7.3.1 Requirements

In order to well perform the test you need the following things:

1. A working PC
2. The SECube board with its ST-Link programmer
3. The Putty software to provide the UART communication between the PC and the SECube board
4. A C-language compiler to compile the golden model implementation

### 7.3.2 Hardware Setup

In order to setup the board, please follow the steps provided on the Section 9 in the documentation of the SEcube-SDK available here [2].

### 7.3.3 Software Setup

The software setup is mainly divided into two parts:

1. The compilation of the golden model. In order to do this it is requested to compile the file contained under the c folder of the project repository available here [4] by simply writing these lines of code on the console:

```
gcc -o test.exe testbench.c grain128aead.c
./test.exe
```

2. The configuration of the UART communication through Putty. Once you opened the software, please click on the "Serial" label under the "Connection" section and set the following configuration:

   (a) COMx as serial line name, depending to which serial port is connected your SECube to your computer (e.g. COM5 )

   (b) Baudrate : 115200

   (c) Data bits : 8

   (d) Stop bits : 1

   (e) Parity : None

   (f) Flow control : None

   After setting all these parameters, please click the "Open" button to start communication, after this a console should appear.

### 7.3.4 Grain128-AEAD : Encryption

In order to test the Encryption phase the files contained inside "test" folder of the official repository of this project are used. Some configuration need to be done in order to launch the testing phase.
For the point of view of the golden model the steps to be followed are these:

1. Open the "c" folder of the official repository of the project
2. Open "grain128aead.h" and "testbench.c" file using whatever text editor you want
3. Modify the following parameter inside "grain128aead.h" file setting the configuration you would like to test (NB The Initialization Vector and Key parameter have a constant size in bytes and they cannot be modified), in particular:

```
#define MSG_PACKETS 1
#define MSG_SIZE 10
#define AD_SIZE 4
```

   (a) MSG_PACKETS : it contains the number of packet that the API needs to create in order to compute the final result for both encryption and decryption phase. This parameter is used for formatting the final golden model result.

   (b) MSG_SIZE : it is the directive containing the number of bytes of the message/ciphertext you want to enrypt/decrypt.

   (c) AD_SIZE : it is the directive containing the number of bytes of the Associated Data used during both the operations.

4. Open "testbench.c" file and write the data you want to use to accomplish your desired operation. The variable you have to modify are the follow:

```
const char *key_string = "0123456789abcdef123456789abcdef0"
const char *iv_string = "0123456789abcdef12345678"
const char *msg_string = "aaaabbbbccccddddeeee"
const char *ad_string = "0011abcd"
```

5. Once you have done that, you are ready to compile the source code and to have a look of the result. Please, follow the Section 7.3.3 to see how to compile the files
6. After both compilation and execution phase, the result should appear as follow

This ends the part related to the golden model. On the other hand it is necessary to configure some files, contained in the firmware, in order to perform the same operation as before, by using the same data. The steps to be done are the following:

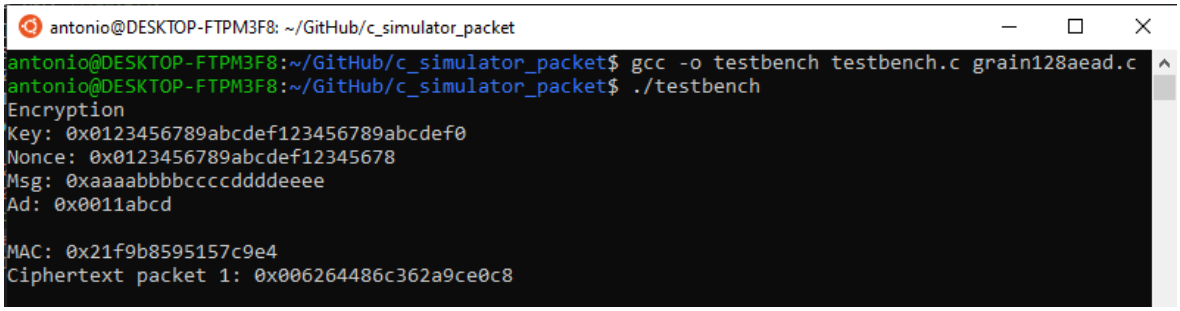


Figure 7.1: Golden model result

1. Open "main.c" file, inside it two function have been implemented:

```
int test_grain128aead_encryption(uint8_t adLen, uint64_t msgLen);
int test_grain128aead_decryption(uint8_t adLen, uint64_t ctxLen);
```

   the functions are used to test the operation specified into the name of the function itself. They receive two parameters:

   (a) adLen : Length of the Associated Data, represented in number of bytes

   (b) msgLen/ctxLen : Length of the Message/Ciphertext, represented in number of bytes

2. Set these two parameters depending the data you want to test. Both the value must be equal to the ones specified before inside the "grain128aead.h" file of the golden model

```
test_grain128aead( 4 ,  // AD length
                   10 ); // MSG length
```

3. Go inside the *test_grain128aead_encryption* function and set **key**, **iv**, **ad** and **msg** data. These data must be equal to the ones set previously inside the golden model. An example will be provided at the end of the explanation.

```
const char *key_string = "0123456789abcdef123456789abcdef0"
const char *iv_string = "0123456789abcdef12345678"
const char *msg_string = "aaaabbbbccccddddeeee"
const char *ad_string = "0011abcd"
```

4. Once you have set all the data you are ready to build the project, to download the firmware on the SECube board and to perform the real test.

After that the previous steps are completed, it is the time to run the firmware previously downloaded and to open the UART communication between the PC and the SECube board (see Section 7.3.3 to see how to do it). The steps to perform are the following:

1. Open Putty software and configuring it as it has been described in Section 7.3.3
2. Once the console appears, please push the "q" key on your keyboard
3. After pushing the "t" key, some configuration messages will be shown as in the picture below. This part could takes a long time due to the configuration of the FPGA [Figure 7.2]

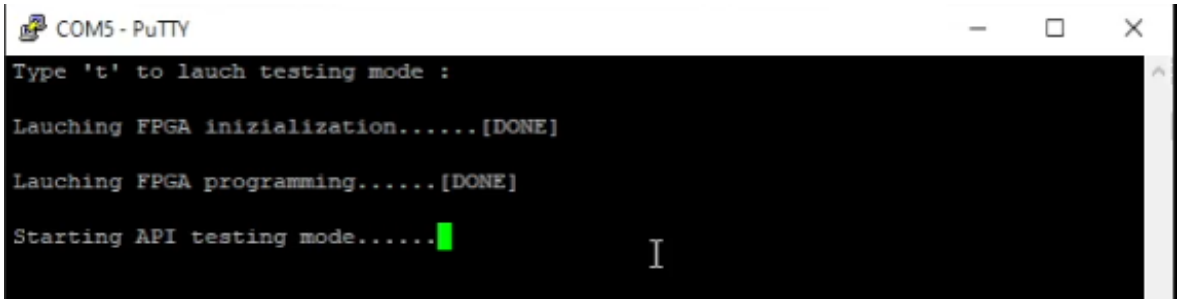


Figure 7.2: FPGA initialization on Putty

4. The Encryption API is ready to start. In order to launch it, you need to push the "e" button.
5. After printing the data specified inside firmware, the final result will appear [Figure 7.3], ready to be compared with the one produced from the golden model [Figure 7.5]. A clear overview of the results is shown in Figure 7.4 .

The comparison shows an equal ciphertext for both API and golden model but a different MAC, it is due to a bug inside the IP core. All the following results will have the same problem, so the comparison will be done only on ciphertext data.

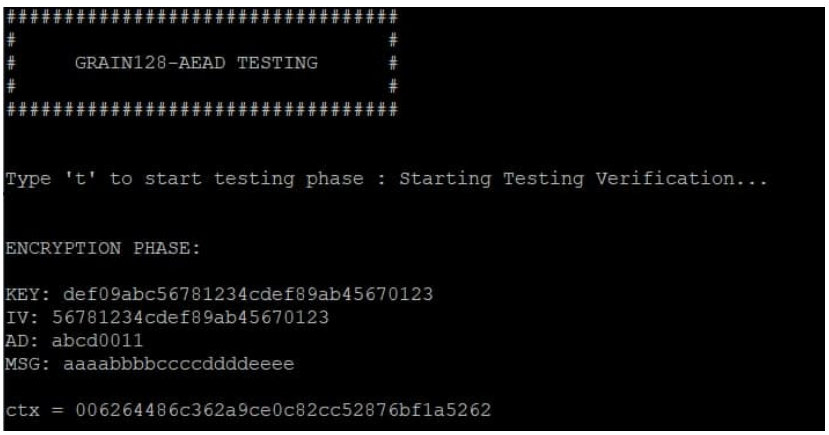


Figure 7.3: Encryption API result

| Source | Ciphertext | MAC |
|---|---|---|
| **Golden Model** | *0x006264486c362a9ce0c8* | *0x21f9b8595157c9e4* |
| **IP core** | 0x006264486c362a9ce0c8 | 0x2cc52876bf1a5262 |

Figure 7.4: One packet Encryption results comparison

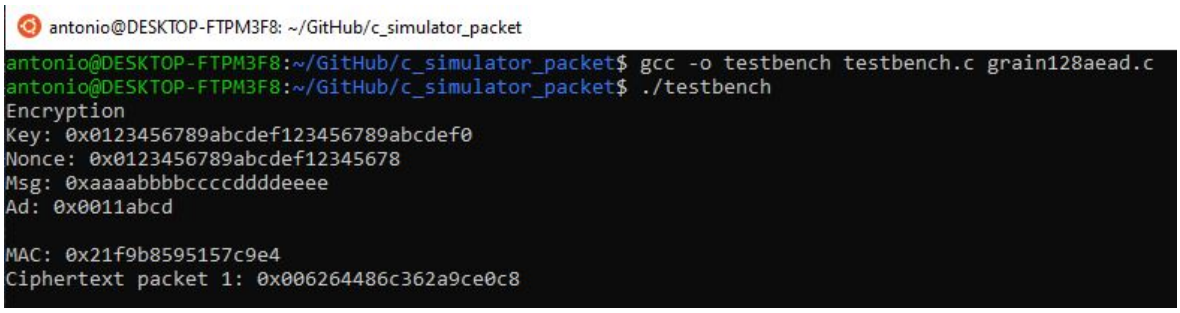


Figure 7.5: One packet Encryption Golden model result

An example based on two packet is also shown. As you can see in Figure 7.6 the golden ciphertext is split in different part depending the number of packet used during the computation ( two in this case ). It also contains one single MAC since it has not been developed using multi-packet approach.
For the point of view of the ciphertext produced by IP core [Figure 7.7], it is composed of a set of ciphertext + MAC depending on the number of packet used during the operation. Also in this case MAC data are different as specified before. It is also possible to see that only the ciphertext, coming from the computation of the first packet, is correct. A clear comparison of these results is shown in Figure 7.8 .

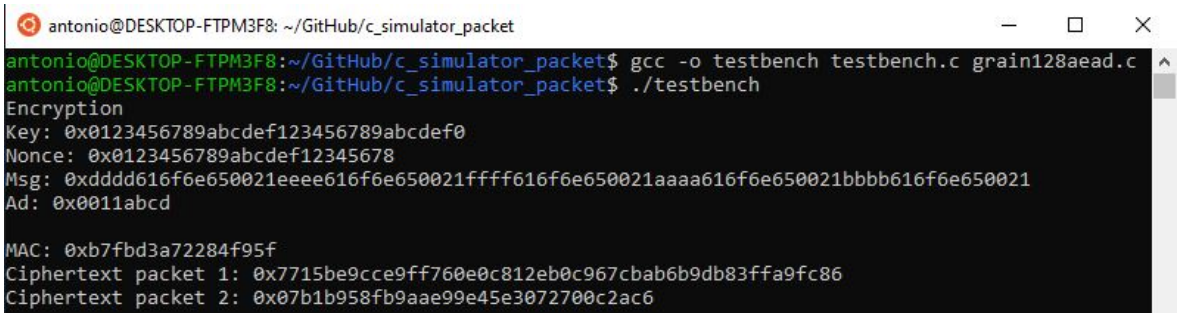


Figure 7.6: Two packet Encryption Golden model result

```
################################
#                              #
#     GRAIN128-AEAD TESTING    #
#                              #
################################

Type 't' to start testing phase : Starting Testing Verification...

ENCRYPTION PHASE:

KEY: def09abc56781234cdef89ab45670123
IV: 56781234cdef89ab45670123
AD: abcd0011
MSG: dddd616f6e650021eeee616f6e650021ffff616f6e650021aaaa616f6e650021bbbb616f6e650021

ctx = 7715be9cce9ff760e0c812eb0c967cbab6b9db83ffa9fc86e120e31300bf7d131cc410573b90ab27b3d36347dd14261665bef612e57ad7ce
```

Figure 7.7: Two packet Encryption API result

| | Golden Model | IP core |
|---|---|---|
| **Ciphertext 1** | *0x7715be9cce9ff760e0c812eb0c967cbab6b9db83ffa9fc86* | *0x7715be9cce9ff760e0c812eb0c967cbab6b9db83ffa9fc86* |
| **MAC 1** | / | *0xe120e31300bf7d13* |
| **Ciphertext 2** | *0x07b1b958fb9aae99e45e3072700c2ac6* | *0x1cc410573b90ab27b3d36347dd142616* |
| **MAC 2** | / | *0x65bef612e57ad7ce* |
| **Total MAC** | *0xb7fbd3a72284f95f* | |

Figure 7.8: Two packet Encryption results comparison

### 7.3.5 Grain128-AEAD : Decryption

The decryption operation is tested immediately after the encryption phase. The steps to lauch this testing are described as follow:

1. Open "main.c" and uncomments this block of operations inside *test_grain128aead* function.

```
//uint8_t msg_decrypted[(msgLen + mac_count*8)];

//GRAIN128AEAD_FPGA_decrypt (key_string, iv_string, ad_string,
    adLen, res_vhdl, resLen, msg_decrypted);

//print_uart("Decryption\r\n");
//print_uart("msg_decrypted = ");
//for (size_t i = 0; i < (msgLen)*2 ; i++) {
//  print_uart_hex(msg_decrypted[i]);
//}
```

2. Open "testbench.c" inside golden model and uncomments this block of operations.

```
//decrypt_message(key, nonce, ct, ad, msg_decrypted);

//Decrypt
```

```
/*printf("\nMessage Decrypted: 0x");
for(size_t count = 0; count < (MSG_SIZE); count++)
    printf("%02x", msg_decrypted[count]);
printf("\n"); */
```

3. Run golden model as it was explained in Section 7.3.3 and see the result. It should also appear if the decrypted message has not been authenticated using the ciphertext + MAC generated in the same golden model during encryption operation.

4. Build the project, download the firmware on board and open Putty following the configuration described in Section 7.3.3. Follow the instructions on the terminal and, at the end, it should also appear the initial message set to perform encryption operation.

Two example of decryption are shown below. In order to simplify understanding of this operations, the ciphertext (combined with MACs) passed to the API decryption function will be the same produced during the Encryption operation in the previous subsection.
In the Figure 7.9 is reported the decryption example by using only one packet. As you can immediately see, the decrypted message computed by the API is exactly equal to the one passed during encryption.

The second example is based on the creation of two packets. The result [Figure 7.10] of the operation is not totally consistent with the message initially passed during the decryption operation. However, as it is shown in Figure 7.11, some parts of it are the same.

Finally, they are also reported the golden model results of the two example described so far. Both Figure 7.12 and Figure 7.13 show a complete and successful simulations of both Encryption and Decryption operation.

```
##################################
#                                #
#     GRAIN128-AEAD TESTING      #
#                                #
##################################

Type 't' to start testing phase : Starting Testing Verification...

ENCRYPTION PHASE:

KEY: def09abc56781234cdef89ab45670123
IV: 56781234cdef89ab45670123
AD: abcd0011
MSG: aaaabbbbccccddddeeee

ctx = 006264486c362a9ce0c82cc52876bf1a5262

DECRYPTION PHASE:

KEY: def09abc56781234cdef89ab45670123
IV: 56781234cdef89ab45670123
AD: abcd0011
CT+MAC: 006264486c362a9ce0c82cc52876bf1a5262

msg_decrypted = aaaabbbbccccddddeeee
```

Figure 7.9: One packet decryption result

```
##############################
#                            #
#     GRAIN128-AEAD TESTING      #
#                            #
##############################

Type 't' to start testing phase : Starting Testing Verification...

ENCRYPTION PHASE:

KEY: def09abc56781234cdef89ab45670123
IV: 56781234cdef89ab45670123
AD: abcd0011
MSG: dddd616f6e650021eeee616f6e650021ffff616f6e650021aaaa616f6e650021bbbb616f6e650021cccc616f6e650021

ctx = 7715be9cce9ff760e0c812eb0c967cbab6b9db83ffa9fc86e120e31300bf7d131cc410573b90ab27b3d36347dd1426167444174847080143c6a8c4ef7
8660a45e

DECRYPTION PHASE:

KEY: def09abc56781234cdef89ab45670123
IV: 56781234cdef89ab45670123
AD: abcd0011
CT+MAC: 7715be9cce9ff760e0c812eb0c967cbab6b9db83ffa9fc86e120e31300bf7d131cc410573b90ab27b3d36347dd1426167444174847080143c6a8c4e
f78660a45e

msg_decrypted = dddd616f6e658a52b0ba84e470180021ffff616f6e650021aaaa616f6e650f58183fa90fd0360021cccc616f6e650021
```

Figure 7.10: Two packet decryption result

***Golden model decrypted MESSAGE:***
dddd616f6e650021eeee616f6e650021ffff616f6e650021aaaa616f6e650021bbbb616f6e650021

***API decrypted MESSAGE:***
dddd616f6e658a52b0ba84e4701800021ffff616f6e650f58183fa90fd0360021cccc616f6e650021

Figure 7.11: Two packet decryption result comparison

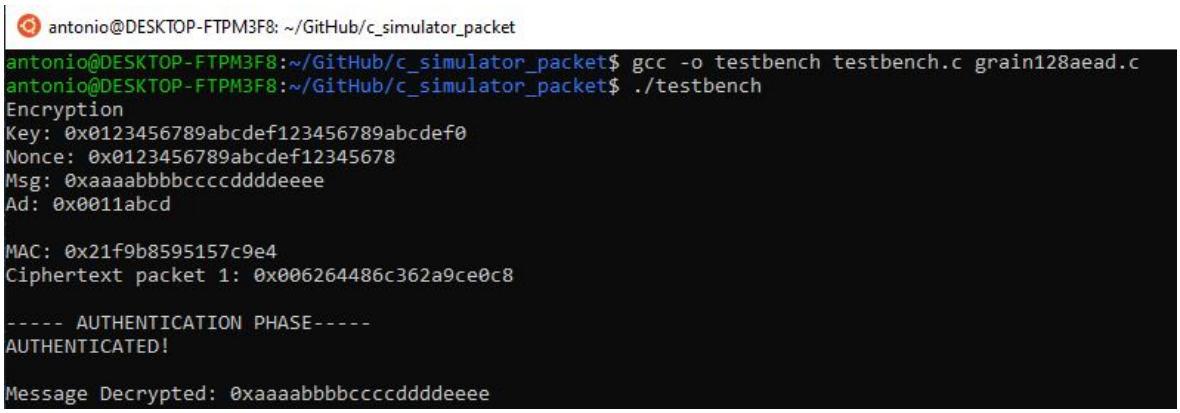


Figure 7.12: One packet decryption golden model result

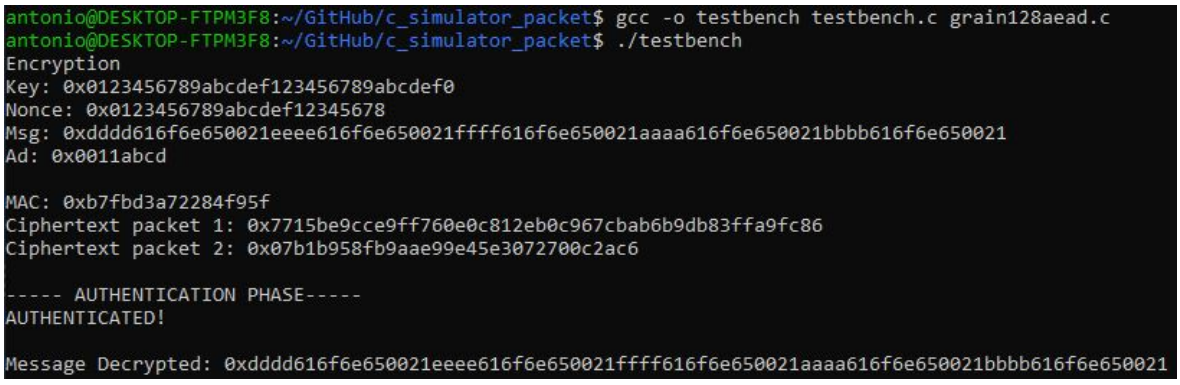


Figure 7.13: Two packet decryption golden model result

# CHAPTER 8

# BUGs

During the development of this project, some BUGs have been discovered. For some of them, a solution has been found and so they have been solved. For others, no solution has been found. Here we will describe the list of bugs that have not been solved during our work:

1. **MAC Encryption and Decryption problem:**
   While doing API-level testing for our FPGA, we encountered inconsistencies for MAC encryption and decrpytion. going into details, we noticed that initially the MAC was encrypted incorrectly with respect to the results returned by the golden model. Testing different vector inputs we realized that this error is committed not from the API side but within our description in VHDL of our controller. Going deep into the analysis we found that the same encryption error is also made in the MAC decryption phase. This implies that the authentication feature is still guaranteed, because the same mistake made in encryption is made in the decryption phase. As described before, the result of the encryption for the MAC does not match with the result given by the golden model. After thorough analysis, we were unable to find a solution for this problem.

2. **Data Buffer strange behavior:**
   While performing several tests in the API-level implementation we encountered a strange behavior of the data buffer. During tests phase we were able to print all the information exchanged between the CPU and the CORE. Key, Nonce, Message and associated data, during our tests, are printed before being written in the data buffer. This allows us to check the correctness of data loaded by the API. During the analysis we were also able to print the contents of each address in the data buffer and, sometimes, some bytes changed their value randomly. This happens sometimes in the input data, and sometimes in the writings on the data buffer during the various operating phases of our project. This behavior happens without any logic and we were unable to find a solution for this problem.

3. **Encryption inconsistences:**
   Due to the problem n.2 described above, messages longer than 10 words are encrypted wrongly. The first 10 words are encrypted correctly, then something wrong happen in the data buffer and for this reason, encryption and decryption have different results with respect to the one expected. We run several tests in the VHDL implementation and we do not encounter any problems.

4. **Golden Model:**
   A problem was discovered inside the Golden model when the encryption operation is performed for multiple packets. Our real implementation works managing single or multiple packets depending on the length of the message to be encrypted or decrypted. The golden model does not have this abstractionism regarding the packet management. During our tests for encryption and decryption for multiple packets we encountered a wrong results given by the golden model. This problem has not been solved despite our attempts. After discovering this bug, we checked the correctness of encryption by running the decryption and comparing the original message with the one obtained after the operation.

In conclusion, about the data buffer, our hypothesis are based on the fact that those changes could compromise the synchronization between CPU and FPGA. Concerning the MAC, we do not know exactly what the problem is, but we can affirm and guarantee that the VHDL part of this project works perfectly. Regarding the encryption and decryption of the message, it works in every case and in every condition no matter the input pattern.

# CHAPTER 9

# Conclusion

Coming to the conclusions of this project and our work, we can certainly affirm that it was very challenging to design the system, but at the same time we have really appreciated it for several reasons. One is related to the use of an FPGA that has to communicate with the CPU and vice-versa. This aspect was almost new to us from a practical point of view. Another reason was that, for those who really like hardware design, we have enjoyed facing the development of such a complex hardware components like the core and its controller. This also took us to experiment our knowledge on the hardware specification and optimization by using the synthesis tool provided by Lattice. Moreover, we were able to better understand to optimize our design components in order to make the fit inside the very small FPGA embedded in the SEcube board with still respecting some timing constraints.

# Bibliography


[1] Blu5 Labs Ltd. *IP-core Manager for FPGA-based design*. URL: https://github.com/SEcube-Project/IP-core-Manager-for-FPGA-based-design.

[2] Blu5 Labs Ltd. *SEcube-SDK*. URL: https://github.com/SEcube-Project/SEcube-SDK.

[3] Willi Meier Jonathan Sönnerup Martin Hell Thomas Johansson and Hirotaka Yoshida. *Grain-128AEAD - A lightweight AEAD stream cipher*. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/grain-128aead-spec-round2.pdf.

[4] A. Ras N. Bianco G. Carrubba. *SECube-Grain-128AEAD*. URL: https://github.com/nicosilverx/SECube-Grain-128AEAD.

[5] NIST. *Lightweight Cryptography*. URL: https://csrc.nist.gov/projects/lightweight-cryptography.