ENSEIRB-MATMECA
BORDEAUX-INP

SE 3A

Lightweight Cryptography Algorithms on FPGA

Élèves :
Thibault SCHWAB
Pedro NOGUEIRA
Bernardo RECKTENVALD
Enzo CENTENARO

Encadrant :
Mathieu ESCOUTELOUP

Table des matières

I Introduction

The security of digital communications is a major challenge in many fields, from embedded systems to computer networks. This project implements a hardware architecture in SystemVerilog for encrypting and decrypting data using the SIMON algorithm, developed by the NSA. The goal is to provide an efficient and integrated solution for lightweight cryptography, suitable for embedded system constraints and secure transmissions via UART.

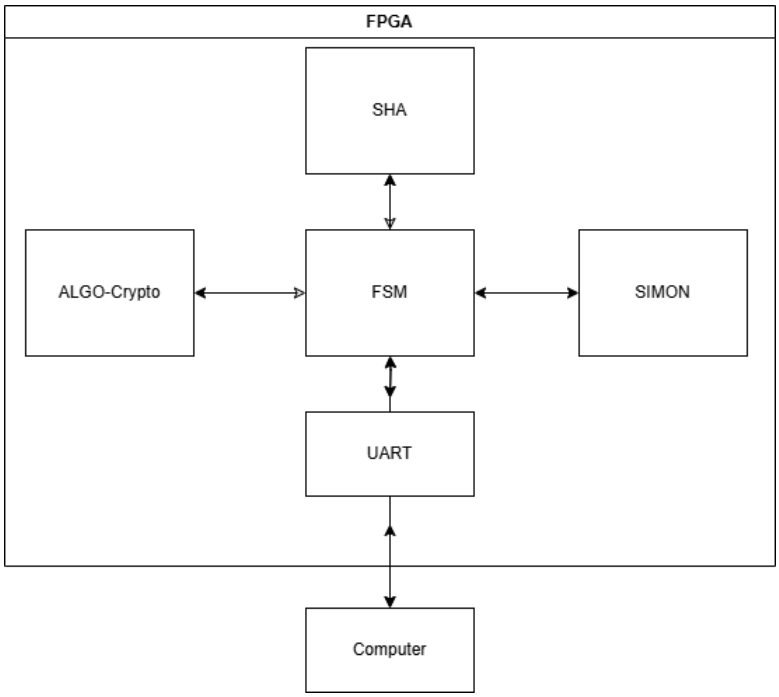

Figure 1 – TOP LEVEL

The presented *top-level* defines the overall system architecture, integrating several key components :

— **A UART interface** for serial communication, allowing encrypted/decrypted data transmission and reception.

— **A Finite State Machine (FSM) module** to manage the various encryption stages, from data reception to processing and transmission.

— **The SIMON module**, responsible for performing encryption and decryption operations based on the SIMON-32 algorithm.

— **The SHA module**, responsible for performing encryption and decryption operations based on the SHA-256 algorithm.

— **The Algo-crypto** represents any other encryption/decryption module that can be added to the architecture by being add to the FSM.

The system operates in *real-time*, processing data received via UART, encrypting/decrypting it using the SIMON algorithm, and then retransmitting it. This architecture is designed to ensure good performance while being optimized for FPGA hardware.

This project can be used in embedded applications requiring lightweight encryption, such as Internet of Things (IoT) devices or industrial control systems, where security and low resource consumption are essential.

II Context Overview

Cryptography, the art of securing communication, has been essential throughout human history. Ancient civilizations employed cryptographic methods to protect sensitive information :

— Ancient Egypt : Hieroglyphic substitution ciphers were used in inscriptions to obscure meaning from untrained readers.

— Sparta : The Spartans used a device called the *Scytale*, a wooden cylinder that allowed encrypted messages to be read only when wrapped around an identical cylinder.

— Ancient Rome : Julius Caesar popularized a shift cipher (now known as the *Caesar cipher*) that substituted each letter with another letter a fixed distance away in the alphabet.

,

As societies evolved, so did cryptographic methods. One of the most famous mechanical encryption devices was the German **Enigma** machine, used during World War II to encode military communications. The Enigma machine's complex rotor-based encryption was eventually broken by Allied cryptographers, demonstrating the perpetual arms race between encryption and decryption.

In the modern era, with the widespread use of the Internet, strong cryptographic algorithms are crucial to securing online communications, financial transactions, and sensitive data. Weak or outdated encryption can lead to severe security breaches, emphasizing the need for robust cryptographic techniques.

III Lightweight Cryptography

While modern encryption algorithms such as RSA, AES, and ECC provide strong security, they are often computationally expensive, making them unsuitable for embedded systems and Internet of Things (IoT) devices. These devices often have :

— Limited processing power and memory.

— Strict energy constraints (e.g., battery-powered sensors).

— Real-time operation requirements.

Lightweight cryptography is designed to address these constraints while maintaining adequate security levels. Unlike traditional cryptographic methods, lightweight algorithms optimize performance for low-power devices. Although the security level is not as high as that for a conventional cryptography algorithm, it is mostly often enough. IoT applications such as controlling and measuring the water level of a reservoir don't need the same security as a secure transfer bank do. Some notable examples include :

— SIMON and SPECK : Block ciphers developed by the NSA, designed for hardware and software efficiency.
— SHA : A widely used cryptographic hash function designed for secure data integrity.
— PRESENT : A lightweight block cipher designed for RFID and sensor networks.
— ASCON : A lightweight authenticated encryption algorithm used in secure communications for embedded systems.

Using RSA or AES in small embedded devices would be inefficient due to high computational costs and power consumption. Instead, lightweight cryptography ensures security while enabling practical implementations in IoT devices, medical implants, and smart cards.

IV UART Blocks Overview

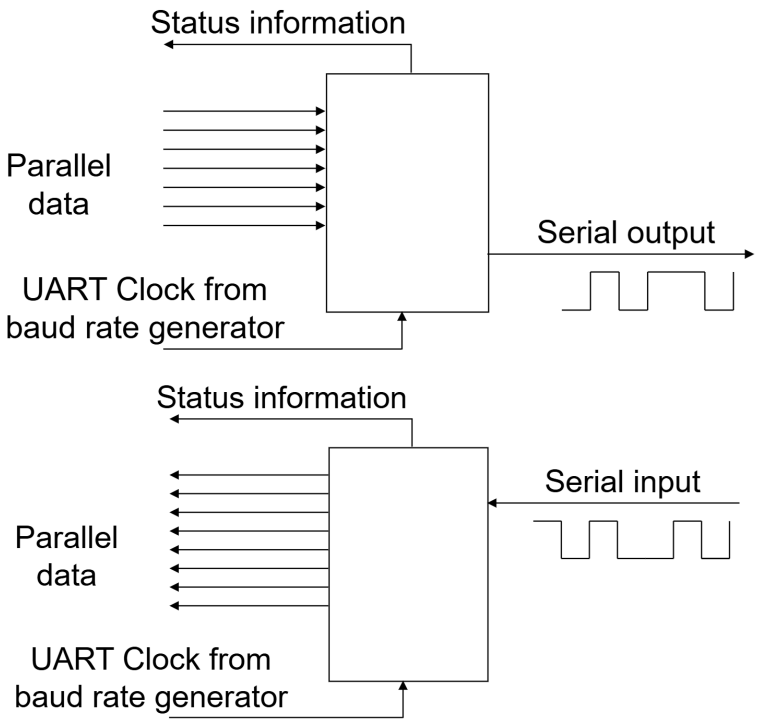

FIGURE 2 – UART DESIGN

The architecture includes two main modules :
— **Transmitter (tx)** : Sends serial data on the UART line.
— **Receiver (rx)** : Receives serial data and makes it available to the system.

IV.1 UART Transmitter Functionality

The `tx` module is responsible for serial data transmission. Its operation is as follows :
— It is clocked by the `clk` signal.
— It waits for a validation signal `i_DV` from the FSM to start transmission.
— Once activated, it transmits the byte `i_Byte` bit by bit on `o_Serial_Data`.

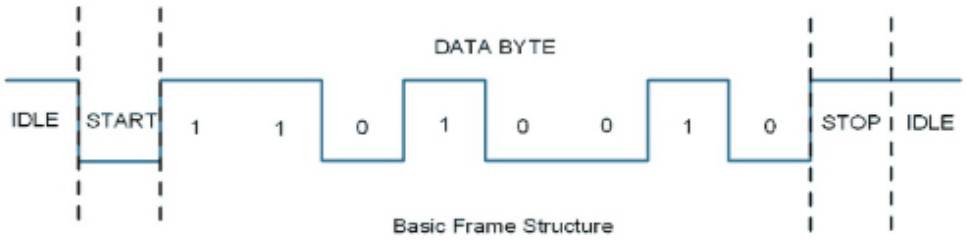

FIGURE 3 – UART TRAME

— It indicates its status via `o_Sig_Active`, which remains high while transmission is in progress.
— At the end of transmission, it generates a `o_Sig_Done` signal to notify the FSM that transmission is complete.

Integration with the FSM

In the architecture, the transmitter is controlled by the FSM :

— `i_DV` is connected to `fsm_control_out`, a signal indicating that a byte is ready for transmission.
— `i_Byte` is fed by `fsm_data_out`, containing the byte to be transmitted.
— `o_Sig_Active` is used to hold the FSM until transmission completes.
— `o_Sig_Done` is used to advance the FSM state and send the next byte.

V FSM Implementation Choices

The Finite State Machine (FSM) manages data flow and controls the encryption/decryption process. It transitions between five main states :

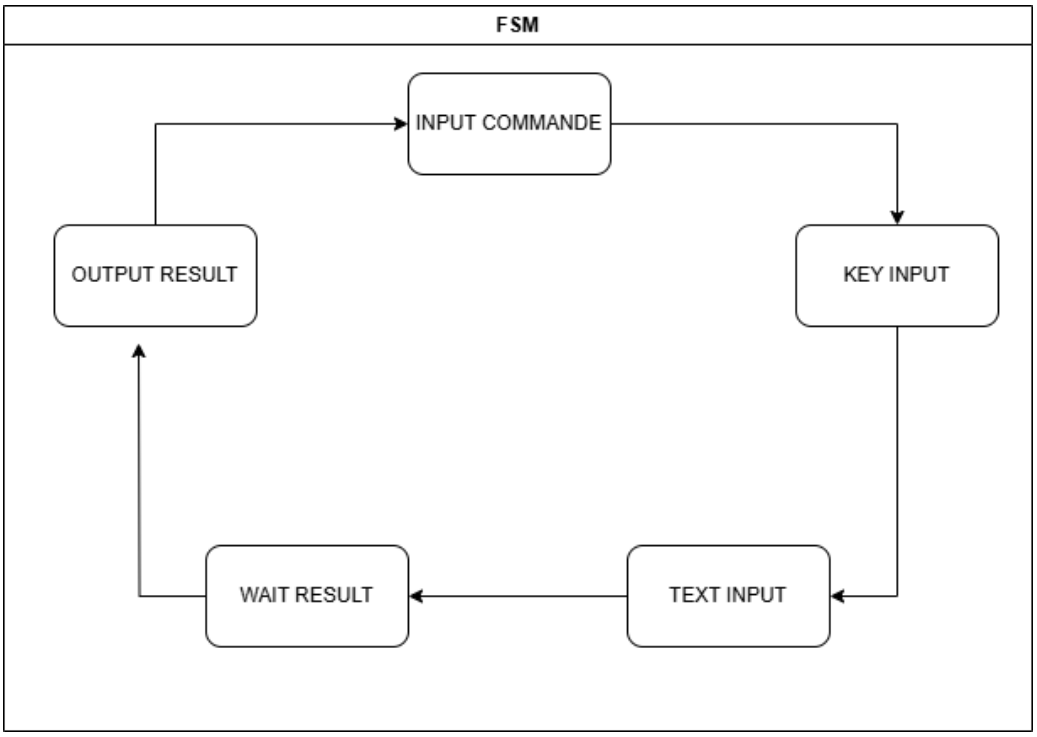

FIGURE 4 – FSM

- **INPUT_COMMAND** : Receiving commands that define the algorithm and operation (encryption or decryption).
- **KEY_INPUT** : Loading the 8 keys required for SIMON encryption.
- **TEXT_INPUT** : Loading the 4 plaintext or ciphertext blocks.
- **WAIT_RESULT** : Waiting for the encryption/decryption result.
- **OUTPUT_RESULT** : Transmitting the result via UART.

State transitions are controlled by signals such as `control_in`, `result_ready`, and `uart_tx_done`. A set of LEDs allows visualization of the system's current state.

VI SIMON Module Implementation Choices

The `simon` module follows a modular approach consisting of :

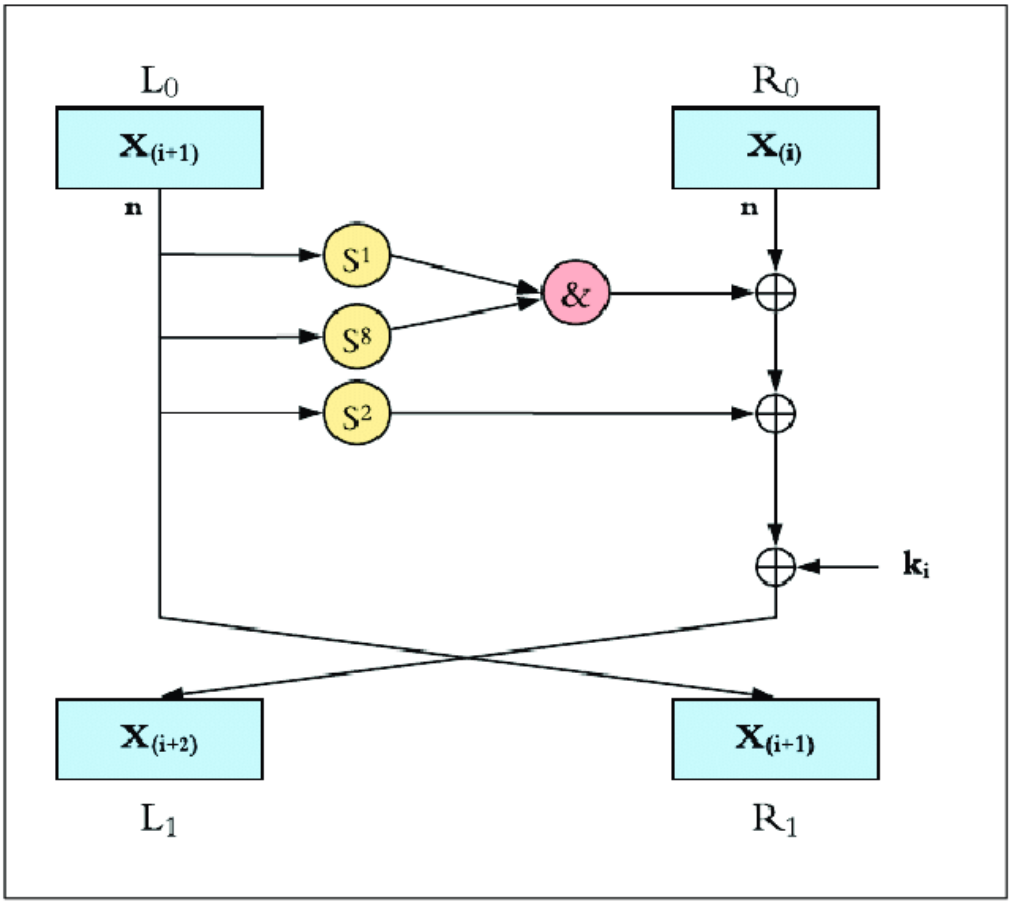

FIGURE 5 – Simon round algorithm

The next round value is calculated by shift, and and xor following the schematic above.

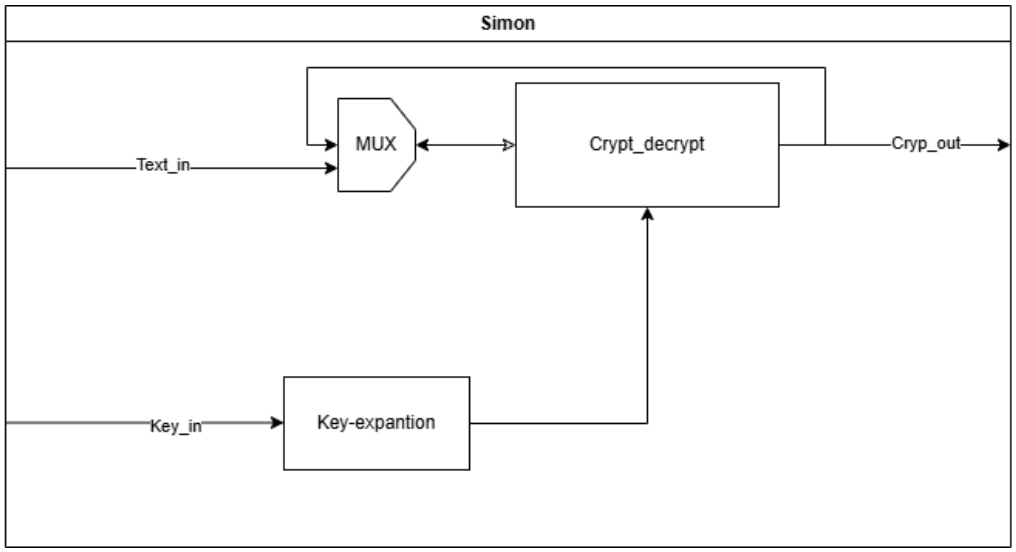

FIGURE 6 – Simon implementation

— **Key Expansion** : Generating round keys from the initial key.

— **Encryption/Decryption** : Applying SIMON transformations to data blocks.

— **Data Multiplexing** : Managing input/output based on the selected operation.

The implementation is made to calculate the next round for each clock to limit the amount of resource used by the SIMON instead of the full logic solution that would have cost too much in resources. A pipeline would have been possible if the baud rate of the uart wasn't limiting the frequency of data input.

VI.1 Input and Output Parameters

The module takes the following inputs :

— `clk` : Main system clock.

— `rst` : Asynchronous reset.

— `cryp_decryp` : Selects between encryption (1) and decryption (0).

— `k_in[3:0]` : Array containing the 4 initial 16-bit keys.

— `text_in[1:0]` : Input text blocks (16 bits each).

— `wait_data` : Indicates data readiness.

The module outputs :

— `crypt_out[1:0]` : Encrypted or decrypted blocks.

— `done` : Signal indicating processing completion.

The encryption and decryption use the same logical gate to optimize the cost of the resource. The MUX chooses between encryption and decryption by switching between the inputs.

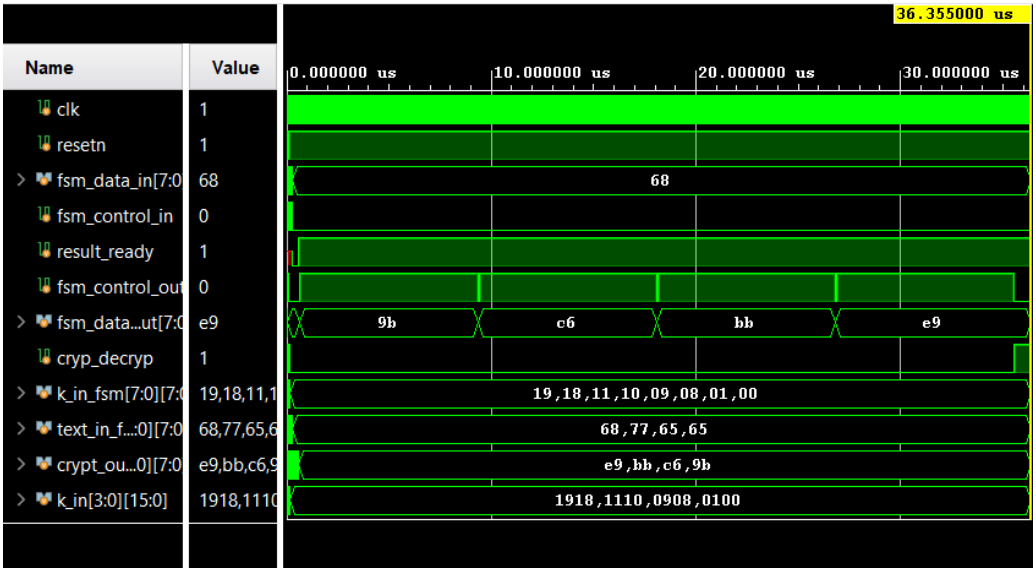

FIGURE 7 – Simon simulation

VII SHA Module implementation choices

SHA (Secure Hash Algorithm) is a family of cryptographic hash functions that operates by repeatedly applying a series of logical and bitwise operations to transform an input message into a fixed-size hash value, for SHA-256 which is what was implemented, the output digest is 256 bits. The algorithm processes the input in blocks, using padding and length extension to ensure a consistent input size of multiples of 512-bit. It employs bitwise rotations, modular additions, and compression functions to create a unique, deterministic output that cannot be decrypted. It implementation comes as follows :

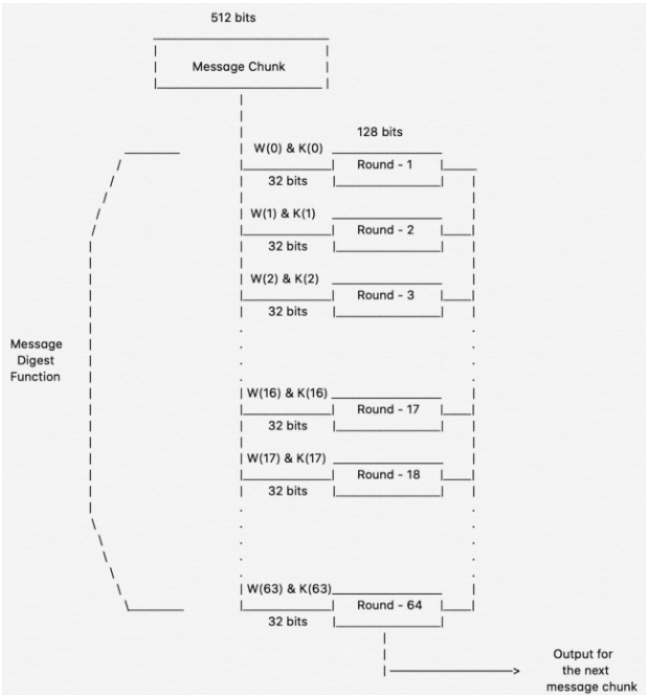

FIGURE 8 – Sha module block diagram.

Where W is the scheduled word (32 bits) for each round out of the 512-bit block and K the round key (also 32 bits). The digest of each 512-bit block is stored in 8 buffers of 32 bits each, which corresponds to an output of 256 bits.

Therefore, the implementation was as follows :

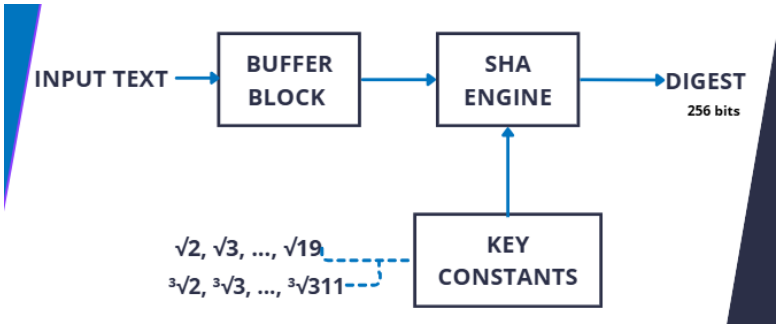

FIGURE 9 – Sha module architecture implementation.

The implementation includes the main core as well as wrappers that provides interfaces for simple integration.

The actual core consists of the following files :

— sha256_core.v - The core itself with wide interfaces.
— sha256_w_mem.v - W message block memory and block expansion logic.
— sha256_k_constants.v - K constants ROM memory.

The top level entity is called sha256_core. This entity has wide interfaces (512 bit block input, 256 bit digest). In order to make it usable, as explained earlier we've used the UART to input serially the 512 bit blocks so an UART interface was also needed.

The provided top level wrapper, sha256.v provides a simple 32-bit memory like interface. The core (sha256_core) will sample all data inputs when given the init or next signal. the wrapper contains additional data registers. This allows you to load a new block while the core is processing the previous block.

VIII SHA outputs

During simulations, we successfully implemented a fully functional module. The image below shows the appending phase for the input message, where 0's are added until the block has a size of 512 bits in which the last byte corresponds to the message size. Therefore, we have multiple zeros before the final size.

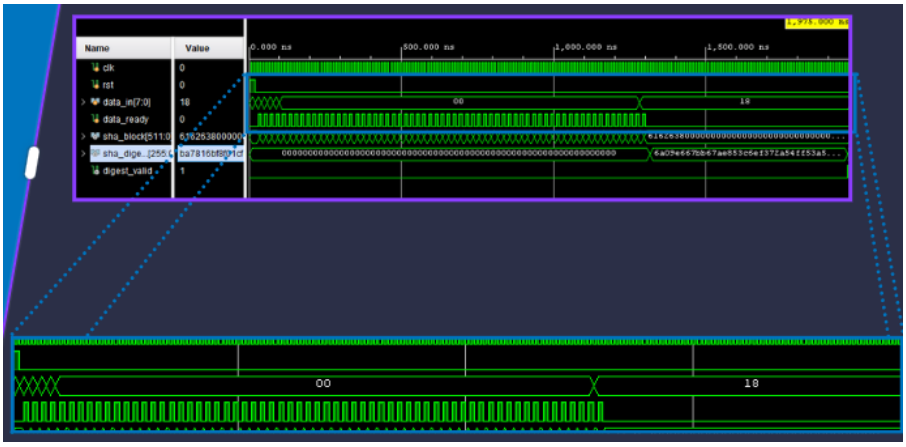

FIGURE 10 – Appending phase.

Once the input data is ready, the SHA engine executes 64 rounds of processing, after which the digest is generated. This completion is indicated by the *digest_valid* signal, which is asserted at the end of the frame, as shown in the image below.

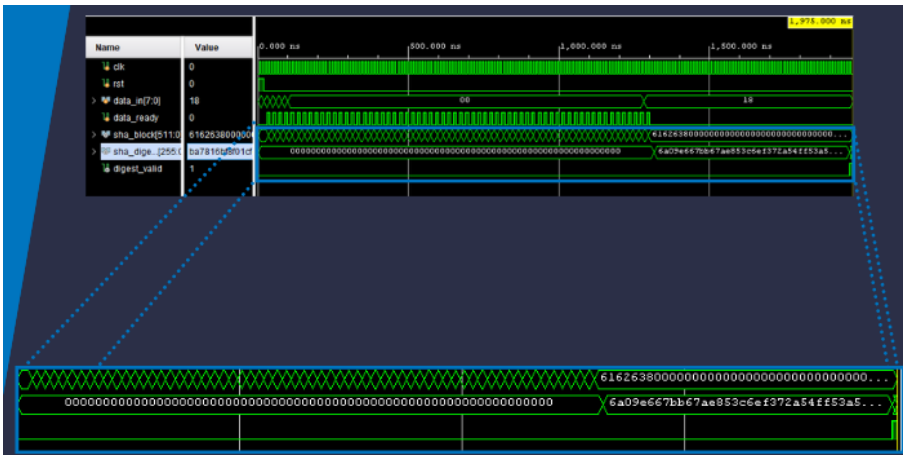

FIGURE 11 – Obtaining digest.

IX Conclusion and Future Work

This project successfully demonstrated the implementation of lightweight cryptographic algorithms, SIMON and SHA, on an FPGA. By integrating a UART interface and a Finite State Machine (FSM), the design ensures efficient data flow management and real-time encryption and decryption. The modular architecture allows for flexibility in adding new cryptographic algorithms while maintaining optimized resource usage.

The SIMON implementation prioritized resource efficiency by computing one encryption round per clock cycle instead of using a full combinational approach, reducing hardware costs. The SHA module, structured with dedicated memory blocks and pipelining, effectively handled message hashing while ensuring compatibility with embedded system constraints. The FSM orchestrated the encryption and decryption processes, efficiently managing key input, text input, and result transmission.

Through UART communication, the system enables secure data exchange, making it suitable for lightweight embedded applications such as IoT and industrial control systems. Future improvements could include integrating the SHA module with the FSM and the integration of additional cryptographic algorithms, optimization of latency through pipelining, and testing on more complex embedded platforms to assess scalability and real-world applicability.

This project highlights the feasibility of deploying lightweight cryptographic solutions in hardware, emphasizing the importance of balancing security, performance, and resource constraints in embedded system design.

We extend our sincere gratitude to Mathieu Escouteloup for his guidance and support throughout this project. His insights and expertise were invaluable in helping us navigate the challenges we faced throughout the project.