RETO AES

Design with programmable logic
TE2003B.601

Abstract

Table of contents

1. Introduction

1.1. Project overview & goal

This is the final project of the module “Design with Programmable Logic” (TE2002B.402). The goal of this project is to implement the AES-Algorithm on a FPGA-Device. It is a suitable challenge to implement all the knowledge that was gained during the module.

1.2. Organization

Team Organisation

- Divide according to the components

Project Phases

-

1.3. Requirements

Technical requirements of the device

1.4. Standards

Programming standards
In which form is the Data input delivered (Rows/Columns)

1.5. FPGA theory

FPGA Overview:
A Field-Programmable Gate Array (FPGA) is a type of integrated circuit that can be programmed after manufacturing. It consists of an array of configurable logic blocks (CLBs), interconnects, and I/O blocks. FPGAs offer flexibility and reconfigurability, allowing designers to implement custom digital circuits and systems efficiently. The exact FPGA model used in this project is MAX10 (10M50DAF484C7G).

Role in the AES Project:
In the context of the AES (Advanced Encryption Standard) project, the FPGA serves as the hardware platform for implementing the encryption and decryption algorithms. The flexibility of FPGAs enables the realization of complex cryptographic operations required by AES while allowing for optimizations tailored to specific performance and resource constraints.

Limitations for the Project:
Despite their versatility, FPGAs have certain limitations that need to be considered in the AES project:

Resource Constraints: FPGAs have finite resources in terms of logic elements, memory blocks, and I/O pins. Implementing complex algorithms like AES requires careful resource management to ensure efficient utilization without exceeding hardware limitations.
Timing Constraints: FPGAs operate based on clock cycles, and meeting timing constraints is crucial for correct functionality. The design must account for propagation delays, clock skew, and other timing considerations to ensure proper synchronization of signals and modules.
Power Consumption: While FPGAs offer flexibility, this comes at the cost of increased power consumption compared to dedicated ASIC (Application-Specific Integrated Circuit) implementations. Optimizing power usage is essential, especially for applications where energy efficiency is a concern.

Programming Complexity: Programming FPGAs requires expertise in hardware description languages (HDLs) such as Verilog or VHDL. Designers must possess the necessary skills to translate algorithmic descriptions into synthesizable hardware code, which can be challenging for complex algorithms like AES.
Security Considerations: FPGAs are susceptible to security threats such as reverse engineering, tampering, and side-channel attacks. Implementing cryptographic algorithms like AES requires attention to security best practices to safeguard sensitive data and prevent unauthorized access or manipulation.
Communication Interfaces:
In addition to implementing the AES algorithm, the FPGA project may involve serial communication interfaces for data exchange with external devices or systems. Serial communication protocols such as UART (Universal Asynchronous Receiver-Transmitter) or SPI (Serial Peripheral Interface) can be integrated into the FPGA design to enable communication with other hardware components or host systems.

1.6. AES Algorithm

Explanation of the AES algorithm

- symmetric Encryption
- Substitution and Permutation

Use of AES

2. Design

2.1. System Architecture

Overview of all components and how they are connected/related to each other

2.2. Encryption

2.2.1. Modules

AddRoundKey Module:

The AddRoundKey module is a crucial part of the AES algorithm, responsible for adding the round key to the State array using a bitwise XOR operation. This operation is performed in each round of the Cipher, including the initial round and subsequent rounds.

Technical Overview:

- Performs a bitwise XOR operation between each column of the State array and the corresponding word from the key schedule.
- Utilizes a key schedule generated through the Key Expansion routine, providing unique round keys for each round of encryption.
- Implemented as a separate module to maintain modularity and facilitate testing and debugging.

Practical Experience:

- Initially explored organizing the State array into a matrix for easier manipulation but opted for direct vector manipulation for efficiency.
- Faced challenges in understanding theoretical concepts and translating them into code, leading to a redesign using a state machine approach for sequential processing.
- Utilized templates provided by the instructor to structure the module and ensure testability and clarity in design.

SubBytes Module:

The SubBytes module performs a non-linear byte substitution on each byte of the State array using an S-box (substitution box). This transformation is an integral part of the AES algorithm, contributing to its security by introducing non-linearity and confusion.

Technical Overview:

- Operates independently on each byte of the State array using an invertible pre-existing S-box.
- Applies the affine transformation over GF(2) to substitute each byte, enhancing cryptographic strength and complexity.
- Utilizes the S-box presented in hexadecimal form to determine substitution values for each byte based on row and column indices.

Practical Experience:

- Initially faced challenges in defining the module's functionality and integrating it into the AES algorithm.
- Revised the design approach by incorporating a state machine for sequential processing and ensuring clarity in the byte substitution process.
- Utilized pre-existing S-box implementations and reference materials to streamline the development process.

ShiftRows Module:

The ShiftRows() module is a transformation of bytes. The last three rows are cyclically shifted over different numbers of bytes. These rows (2, 3 and 4) are shifted a certain number of positions to the left, with each row having a different shift amount. Row 2 experiences a 1-byte left shift, row 3 a 2-byte left shift and row 4 a 3-byte left shift. This process disrupts the original data´s structure, encrypting the plain text.

Technical Overview:

- ShiftRows aims to disrupt the relationship between bytes in the data.
- The inputs used are *TxtIn* a logic vector of 128 bytes. Also, *Clk*, *Rst*, and *Start*, being the signal from de Master FMS, to start the transformation.
- The outputs used *TxtOut* a logic vector of 128 bytes. Also, *Finish* the signal to the Master FMS, that the transformation is finished.

Practical Experience:

- The main challenge we faced was the parallel transmission of data from the input vector to the output vector.
- Also, we received guidance on the machine with states from Rick Swenson.

MixColumns Module:

Galois Fields are algebraic structures used in number theory and cryptography. They are represented as GF(2^m) is a positive integer. In these fields, addition and multiplication operations are performed using modular arithmetic with an irreducible polynomial of degree m over the finite field of two elements, GF(2).

The MixColumns module is an operation in the AES (Advanced Encryption Standard) cipher applied to each column of a 4x4 state matrix during the encryption process. It consists of multiplying each column by a fixed matrix, followed by modular addition with respect to an irreducible polynomial in GF(2^3).

The MixColumns matrix is as follows:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Technical Overview:

- It requires Galois Finite Fields to convert the input data in the desired output.
- The inputs used are, TxtIn, a logic vector of 128 bytes, that is transformed on a 4x4 matrix. Furthermore we use *Clk*, *Rst*, and *Start*, being the input signals received from de Master FMS, to start the transformation process.
- After multiplying the Galois Finite Field Matrix with the 4x4 matrix, the result is transformed back into a logic vector 128 bytes called TxtOut.
- The outputs used are *TxtOut, a logic* vector of 128 bytes and *finish,* the signal to communicate the Master FMS, that the transformation is finished.

Practical Experience:

- The main challenge we faced was coding the Galois Multiplication in VHDL. The first implementation we made used mainly processes and signals. Nevertheless, the signals were not correctly communicating the data which ended up in receiving erroneous results when simulating. For example, in the first attempt the value of the output was the next sequence “XXXXXXXXXXXXX”, and after analyzing the data flow we realized that the signals were receiving the input. For the last attempts the output vector did not match the result, since it had different values that didn’t match the correct result of the test case.

2.2.2. Philosophy

Since the state machine dictates the overall flow of the AES, maintaining synchronization with it is essential. To accomplish this, the logic for each module within the encryption process is designed with consistent naming conventions. Seamless interaction between the modules and the state machine, granting an error free. Each module within the encryption process has its own dedicated state machine (FM). The Master FMS receives the completion signal and start signal from the finished module. Based on this information, the Master FMS triggers the activation of the next module in the encryption sequence by issuing a start flag to its corresponding FM. This approach promotes a well-structured and modular design, making the encryption process easier to understand, maintain, and potentially modify.

2.2.3. Testbench

The testbench is a crucial part of the program creation, because at this part we verify the functionality of the code and if there's any problem. We can use it for debugging and find errors or even realize that the program is just compiling, but not functional. There is a testbench case for each module. We standardize at least three problem cases to verify the program functionality.

2.2.4. Challenges

AddRoundKey Module: The main challenge was understanding and implementing the bitwise XOR operation between the State array columns and corresponding key schedule words. Initially, there was confusion about the module's objectives and expected functionality. They experimented with organizing the vector into a matrix for manipulation but later switched to direct vector manipulation for efficiency. They also faced difficulties in translating theoretical concepts into code, leading to a redesign using a state machine approach for sequential processing.

SubBytes Module: The initial challenge was defining the module's functionality and integrating it into the AES algorithm. They struggled with clarity in byte substitution and incorporating a state machine for sequential processing. However, they managed to overcome these challenges by utilizing pre-existing S-box implementations and reference materials, streamlining the development process.

ShiftRows Module: The primary challenge here was implementing parallel data transmission efficiently from the input vector to the output vector. They also encountered difficulties in incorporating clock cycles for synchronization and proper data processing. However, with guidance on state machine implementation from Rick, they were able to adjust their design approach and improve efficiency and clarity in their code.

MixColumns Module:

2.3. Decryption

2.3.1. Modules

The modules of this component are the same as the encryption ones, changing slightly in the logic since it is a symmetric encryption algorithm. There are four modules in total, them being:

- AddRoundKey: This module is used twice within the whole decryption process, first it's used as an input for the next modules, and it's also the last step of the process, if it is also the last round of the process, this module becomes the key for the specific block.

- Sub-Bytes: Replace each byte with another byte depending on a precalculated look-up table called "Rijndael Inverse S-Box". The content of the byte to replace is used as an index for the table so that the value contained in there turns into the replacement. This S-Box must be the exact opposite of the one for encryption.
  Allow:
  - The design must ensure that the inverse operation doesn't introduce vulnerabilities or weaken the security of the encryption scheme.
  - Can enhance resistance to certain types of attacks.

  Practical Experience
  - In real-world applications, efficiency is crucial. The design should be optimized for performance to ensure that decryption processes are fast and scalable, especially when dealing with large volumes of data.
  - Test vectors provided in AES specifications are used to verify the accuracy of the decryption process, including the correct functioning of the InvSubBytes module in reversing byte substitutions.

- ShiftRows: This is a rotational operation that must be done as a process for scrambling the bytes, it is done by doing exactly what the name suggests: shifting the rows of our matrix.

- MixColumns: The inverse MixColumns operation in AES is the step that undoes the transformation performed by MixColumns during encryption. It relies on a specific inverse matrix and involves multiplying each column of the internal state of the data block by a fixed matrix in the Galois finite field

(GF(2^8)) and performing certain bit manipulations to restore the original state before encryption. This is essential for the correct decryption process in AES.

The InvMixColumns matrix is as follows:

$$InverseMixColumn = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Practical Experience:

- In the VHDL code implementation, we receive a 128-byte vector; Example: “XXXXXXXXX…” in the hexadecimal. In the program, we arrange it from least significant to most significant by rows, although Galois multiplications and reductions are executed by column. The Galois reduction is performed by calling a function that analyzes whether the result exceeds 8 bytes when multiplied. After performing these operations, we rearrange the output by rows from the least significant byte to the most significant. The difficulties we encountered were that the vector, when undergoing Galois multiplication, did not execute properly and displayed different values, appearing in locations where they did not belong.

2.3.2. Philosophy

The decryption process is one out of three total processes within the AES encryption algorithm, therefore there is a state machine that indicates to us when the decryption, and each one of the modules, are going to work, signaling the state of the process and enabling the next one.

There must be a synchronization with the state machine since it is the guideline of the AES encryption algorithm, so the logic for the decryption to work was done by being consistent in the names within the planning of the modules, this so that the logic would match with the state machine and the decryption.

Because there is a general state machine, there are no state machines within the decryption process since we settled all the limitations and conditions within each module.

2.3.3. Testbench

A testbench is used to verify the correctness of our programs, in the decryption case a testbench was made for each of our modules and one for the whole process, this was done so that we could first check each module and debug it if needed and when all modules where assembled our final test bench would help us verify that the logic of the modules is working correctly together, since each module was done individually.

2.3.4. Challenges

One of the major challenges was settling on a standard for not only our documentation but for the programming, therefore making it easier for us to debug if needed, this required being consistent with names and types of data used, if not done correctly each module will work on its own, but when assembled as a whole, the logic won’t work and the testbench won’t either.

When the modules were done, the finite state machine wasn't ready, so we had to wait to make changes per module instead of having to modify the program of the modules together, this part helped us debug once the state machine was done, by debugging the modules individually so the logic wouldn't change but they would match with the state machine, making both processes work.

Another major challenge was communicating with the other areas of the AES process since there would always be needed input for the decryption to work, this input was the output of another area of the algorithm, and we would deliver the output, therefore communication with the "keys" area and the state machine module where important to synchronize and settle a standard so the process would be understood in any part of the algorithm.

2.4. Key Schedule

2.4.1. Modules

This component requires a different architecture and planning as it generates the keys for the other components to work based in an original given key, all this at the beginning of the program, so that the other components have already available the keys to either encrypt or decrypt the text, this main module is subdivided into the following 3 modules:

RotWord module:

This module takes the last column of the previous key, if its the first step it takes the original key, then processes it and rotates the values from the column taking the first one and moving it to the bottom, the output will be the moved column. This is repeated for every generated key.

Technical Overview:

- This component of the Key Schedule receives a 32 bit standard logic vector, which is provided by the original key. In order to perform its objective, what it does is to assign the corresponding values to an output vector.
- To manage the rotation, a specific structure is considered to manage the rows of the last column of the previous key. This is to consider that 1 byte of the vector represents a different row, starting the vector as the column from top to bottom.

SubBytes module:

The SubBytes module performs a non-linear byte substitution on each byte of the received rotatedWord of 4 bytes or 32 bits, provided by the previous module using an already defined S-box (substitution box). This is divided in “high” and “low” where high are the most significant bits of the hexadecimal number and the low the least significant bits, we compare what we have in the data and change the original value into the one that is given by the S-Box.

		Y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
X	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S-box utilized in the development of the SubBytes Module

The implementation of this module is vital, although on the encryption and decryption modules there is an implementation of a SubBytes module, the functionality of the one used in the Key Schedule is particular,

and also calling an already defined function on other modules would represent an inefficiency in the implementation of the AES algorithm.
It is important to mention that this submodule is contributing to the security of the encryption and decryption by introducing non-linearity and confusion.
Technical Overview:

- As mentioned, this module receives a 32 bit rotatedWord, then it makes use of a temporary vector of the same length, that separates the vector of 32 bits into bytes, to conduct the operation optimally, and then after de substitution, it assigns this temporary vector to the output vector.
- Operates independently on each byte of the temporary vector using an invertible pre-existing S-box, presented in hexadecimal form to determine substitution values for each byte based on row and column indices.

Practical Experience:

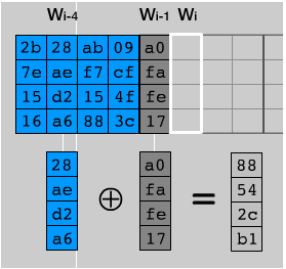
- Initially explored in the definition of the S-box comparison system, there was a try using the “when case” comparison, but eventually a constant array was chosen, and the comparison is made using the “to_integer” function.
- Faced challenges with the implementation of the clock, reset, enable, and finish signals, but eventually the implementation of those signals was implemented optimally.

XOR Module

After the RotWord and SubBytes steps, an extra series of steps is applied in order to generate each of the ten keys. After it is done, it is stored in a different RAM module, in order to be exported to the encryption and decryption modules This is what our module does.
The XOR Module consists of three steps that are necessary for each of the columns.

- **First XOR operation with Rcon Table:** Similar to SubBytes, our first operation algorithm in the XOR module takes a column from a pre-established, constant table named Rcon. It is involved in a XOR operation with the first column in our table and **the column received from the subBytes module**. This gives us the first column from the new key, in dark gray. This column is then used in the second XOR operation.

- **Second XOR Operation, repeated three times**: After the first column is done, we need to fill the 3 next columns in the 128-bit block. This is done by performing a second operation with the new column (dark gray) and the **column three steps behind it (Wi-4).** This operation gives us the next column (in light gray). This process is repeated a total of three times, which gives us four complete columns, a 128-bit block. This is the first of our ten keys.



- Repeat every step for each column in the key schedule: After this step, the same algorithm is run through the next nine keys, which gives us our final result, and the rest of the 128-bit blocks can be stored in the RAM memory.

2.4.2. Philosophy

The design process was based on the meticulous key generation of the AES (Advanced Encryption Standard) model, carefully considering the inherent functionality of the algorithm. In order to efficiently address this complex task, the team decided to break down the component into more manageable units. This strategy allowed for effective distribution of work, with small teams working on specific parts of the process, all coordinated by project administrators. Such an approach not only facilitated collaboration among team members but also promoted coherence in the design and implementation of the key generation system.

Prior to the commencement of the implementation phase, a crucial stage of planning and design was undertaken. In this regard, a potential solution was established for the entire team, including the design of a state machine and a specific component for the model. This preemptive measure was adopted with the purpose of avoiding redundancies and minimizing the possibility of conflicts in collaborative work. Furthermore, it ensured that each team member was equipped with the same information, significantly facilitating the debugging and validation process of the system.

The emphasis on planning and coherence in the methodological approach was fundamental to the success of the AES algorithm implementation project. By adopting a structured and well-coordinated strategy, the team was able to overcome the challenges inherent in the complexity of the algorithm and achieve a key generation system that met the required standards of security and efficiency. This experience underscores the importance of organization and collaboration in high-complexity software development projects.

2.4.3. Testbench

In our testbenches, we adopted a rigorous testing methodology that focused on evaluating component by component. This involved conducting comprehensive tests for each element before proceeding to implement the final set. Additionally, upon reaching the final stage of development, we conducted a comprehensive test bench to verify the overall functioning of the code.

The methodology of our tests was characterized by its systematic and detailed approach. For each component, we started by declaring a specific input value and executed the corresponding test. Based on the output obtained, we generated a new test case in which the expected output should match the original input data. This process was iteratively repeated 2 to 3 times in each test, allowing us to validate the consistency and reliability of the component in different scenarios.

This meticulous approach in developing test cases provided us with a deep understanding of the behavior of each individual component, as well as its interaction with the system as a whole. Furthermore, by repeating the process several times, we were able to identify potential errors or inconsistencies early in the development process, which significantly contributed to the quality and stability of the final code

2.4.4. Challenges

We want to talk specifically about the challenges during the implementation, how we manipulate some variables or some signals, the way in which we progressed through each part of the module, some misunderstandings when designing the state machine and the design of the module in general, rather than talking about general challenges in the overall organization of the whole project, or the way tasks were distributed.

Firstly we want to mentionate what we faced module by module, and when we say module it’s very important to emphasize that "Key" is the primary module, and it's divided into 3 submodules (by the moment), but we’ll refer to those submodules as modules, prefixed by their respective names, mostly for the sake of practicality, although we already know that these are the parts of the ‘Key Schedule’ and not an equivalent. In the Rot Word module, we didn’t have any problem, it was the easiest of them, because it just moved some values. Of all the 16 values (128 bits) that our key had, only 4 (32 bits) of them were used. And that was its only role, so we had no difficulty in that process. In the module.

In the SubByte module we had a small problem when we did a testbed and tried to implement the SBox in the programme, but after a while the people in charge of that finally managed to do it.

Finally, as mentioned above, we are still discussing whether it is better to create separate projects (files) for the XORs and RAM module. Because some of us have never worked with a RAM before, but some of our team has, it is more bearable to be able to move forward on that aspect.

2.5. Top-Level Finite-State-Machine

2.5.1. Modules

The top-level Finite-State Machine (FSM) in this AES encryption project consists of nine states that coordinate the execution of the AES encryption algorithm. Each state represents a specific stage of the AES encryption process.

State sequence

ST0 Key Schedule: This module is responsible for expanding the original encryption key into a set of round keys.

ST1 AddRoundKey: Adds the current round key to the state array.

ST2 SubBytes Loop: Performs byte substitution according to a predefined lookup table.

ST3 ShiftRows Loop: Shifts the rows of data in the state array.

ST4 MixColumns Loop: Performs the column mixing operation on the state matrix.

ST5 AddRoundKey Loop: Controls the number of rounds performed during the AES encryption process.

ST6 SubBytes Final: Performs byte substitution in the final round.

ST7 ShiftRows Final: Shifts the rows of data in the state matrix in the final round.

ST8 AddRoundKey Final: Adds the last round key to the state array.

ST9 Finish: The plaintext is already ciphered, indicating that the encryption process has finished.

2.5.2. Philosophy

The design prioritizes modularity and reusability, allowing the same modules to be used across multiple states of the FSM. This approach enhances maintainability and facilitates future modifications or extensions to the encryption algorithm.

The hierarchical structure promotes modular design and facilitates integration with the FSM. Despite the repetition of modules across different states, clear naming conventions and comments enhance code readability and comprehension.

2.5.3. Testbench

The testbench generates input stimuli corresponding to each FSM state and verifies the outputs against expected results. By simulating the behavior of the system under different conditions, it validates the correctness and robustness of the implementation.

2.5.4. Challenges

During the development of the state machine, several challenges emerged. The first one was to correctly differentiate the inputs and outputs, as well as to use the appropriate symbology to represent each signal and transition clearly and understandably. This was crucial to ensure proper communication between the different components of the state machine and to ensure its proper operation.

Additionally, determining the transition conditions between states in the loop of the AES encryption process proved to be a significant challenge. Establishing the precise conditions governing the flow of the state machine at each stage of the encryption algorithm was essential to ensure that the encryption process was carried out correctly and efficiently.

Despite these initial challenges, with a meticulous approach and thorough testing, these obstacles were overcome, and a functional and effective state machine for AES encryption was developed.

3. Implementation & Result

Description of the last step, where all components finally come together

3.1. Integration of all Components

Description how all components are put together

3.2. Testing

Test the complete system (how is it done?)

3.3. Result

Results of the test
how good/accurate/efficient does it work

4. Conclusion

4.1. Summary

Summary of what was done

4.2. Review

Review how the project enrolled

4.3. Learnings

what are the biggest learnings of the project

- Work in a Top-Down structure

4.4. Acknowledgments

4.5.

5. References

[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]