Research Project on AES-128: Implementation on FPGA

Vangelis Ananiadis, Maria-Nikoletta Kalantzi (Students, UTH)
Supervising Professor: Karakonstantis Georgios

ABSTRACT

This paper explores the implementation of an AES-128 Encryption and Decryption Core. AES-128, a widely adopted symmetric key encryption standard, is essential for ensuring data security in modern computing systems. The paper provides a detailed overview of the AES algorithm, focusing on the structure, operations, and key expansion involved in both the encryption and decryption processes. Additionally, it delves into the implementation of AES-128 in hardware, specifically within an FPGA environment using Vivado. The testing and verification of the AES-128 implementation, encompassing both encryption and decryption, are conducted through simulations on the Vivado environment.

Index Terms **Cryptography, AES-128, Data Protection, Data Security, Encryption Algorithms, Cryptographic Key**

I. INTRODUCTION

What is Cryptography (Encryption)?

Cryptography is the science of securing information, ensuring that it remains confidential, authentic, and tamper-proof during transmission or storage. It is a cornerstone of modern digital communication, protecting sensitive data like financial transactions, personal messages, and secure access credentials.

At its core, encryption is a fundamental cryptographic process that transforms plaintext (readable data) into ciphertext (an unreadable format) using a mathematical algorithm and a secret key. This transformation ensures that only authorized parties with the correct decryption key can revert the ciphertext back to plaintext.

Types of Encryption

Asymmetric encryption, also known as public-key cryptography, uses two separate keys to encrypt and decrypt data. One is a public key shared among all parties for encryption. Anyone with the public key can then send an encrypted message, but only the holders of the second, private key can decrypt the message.

Asymmetric encryption is considered more expensive to produce and takes more computing power to decrypt as the public encryption key is often large, between 1,024 and 2,048 bits. As such, asymmetric encryption is often not suited for large packets of data.

Symmetric encryption, also known as a shared key or private key algorithm, uses the same key for encryption and decryption. Symmetric key ciphers are considered less expensive to produce and do not take as much computing power to encrypt and decrypt, meaning there is less of delay in decoding the data.

The drawback is that if an unauthorized person gets their hands on the key, they will be able to decrypt any messages and data sent between the parties. As such, the transfer of the shared key needs to be encrypted with a different cryptographic key, leading to a cycle of dependency.

Commented [1]: γενικα τα πειπερς εχουν λιγα (γυρω στα 6) και σχετικα index terms

Commented [2]: Cryptography, AES-128, Data Protection, Encryption Algorithms, Encryption Process, Ciphertext, Plaintext, SubBytes, Key Expansion, ShiftRows, MixColumns, AddRoundKey, Decryption Process, Cryptographic Key Recovery

Commented [3]: αλλά αφήνω αυτά εδώ αν θες να προσθέσεις κιαλλα
One total reaction
NIKH ΚΑΛΑΝΤΖΗ reacted with at 2025-01-13 05:42 am

Commented [4]: α επισης να προσθεσουμε σχετικα με fpga

Commented [5]: θεωρια? τι ειναι φαση?

Commented [6]: οοχι index terms εννοω χαχα

Common Encryption Algorithms

- **Data Encryption Standard (DES)**
- **Triple DES (3DES)**
- **Advanced Encryption Standard (AES)**
- **Twofish**
- **RSA**
- **Elliptic Curve Cryptography**

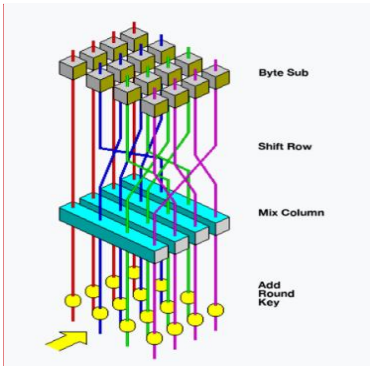
A Little About Decryption

Decryption is the process of converting ciphertext back into its original, readable form (plaintext) using a cryptographic key. It is the reverse operation of encryption and is essential for retrieving the original information from securely transmitted or stored data. In symmetric key encryption algorithms like AES-128, the same key used for encryption is applied during decryption to reverse the transformations performed on the data. The decryption process involves undoing each step of encryption, such as reversing substitutions and permutations, and applying the round keys in reverse order.

II. BASIC IDEA OF AES-128

An AES-128 encryptor is a hardware or software module designed to implement the Advanced Encryption Standard (AES) algorithm with a 128-bit key for secure data encryption (National Institute of Standards and Technology, 2001). AES-128 operates on fixed-size data blocks of 128 bits (16 bytes), transforming plaintext into ciphertext through a series of systematic operations, making the data unreadable without the corresponding key. The encryption process involves 10 rounds, each consisting of four key steps: SubBytes (a non-linear substitution of bytes using a pre-defined substitution box or S-Box), ShiftRows (a cyclic shift of rows to introduce diffusion), MixColumns (a mixing of data across columns to increase complexity), and AddRoundKey (a bitwise XOR operation with a round-specific key derived from the original key). The first and final rounds have slight variations, such as omitting the MixColumns step in the last round.

The AES-128 encryptor uses a process called key expansion to generate 11 round keys from the original 128-bit key, which are applied sequentially during the encryption rounds. This efficient and structured design ensures robust data security while being optimized for both hardware and software implementations. AES-128 encryptors are widely used in applications ranging from secure communication protocols to hardware encryption modules for protecting sensitive information.



Visualization of the AES round function

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES_(Rijndael)_Round_Function.png

Commented [7]: Οι εικόνες θέλουν πηγή!!

III. FURTHER ANALYSIS OF AES-128

Input

The input to the AES-128 algorithm is a plaintext block that is exactly 128 bits long. This block is often a segment of the plaintext (e.g., a portion of a file or a message). The 128-bit block is first divided into 16 bytes, which are then arranged into a 4×4 two-dimensional array called the State.

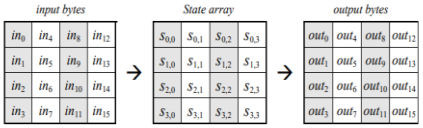
Output

The output of the AES encryption process is the ciphertext block, also 128 bits long. It represents the encrypted version of the plaintext block, transformed through a series of cryptographic operations. Like the input, the output is organized as a 4×4 array.

The State

In the AES-128 encryption algorithm, the State is a fundamental data structure that represents the intermediate result of the encryption process at every stage. It is a two-dimensional array of bytes arranged in a 4×4 grid, containing 16 bytes (128 bits) in total. At the start of the encryption process, the plaintext input is divided into 16 bytes and mapped column-wise into the State array. For example, the first byte of the input becomes the first element of the first column, the second byte becomes the first element of the second column, and so on.

Each byte in the State is manipulated through a series of transformations such as SubBytes, ShiftRows, MixColumns, and AddRoundKey during the encryption rounds. The structure of the State allows for parallelism and efficient processing, as operations are performed either at the byte level or across entire rows and columns of the array. After the final round, the State is converted back into a linear sequence to produce the 128-bit ciphertext output. This systematic use of the State array is central to the AES algorithm's ability to ensure both security and computational efficiency.



Ciphertext

Ciphertext is the result of encrypting plaintext. It is unintelligible to anyone without the decryption key. For AES-128, the ciphertext is derived by applying 10 rounds of transformations to the input block (SubBytes, ShiftRows, MixColumns, and AddRoundKey)

The 4 Steps in Each Round of Processing

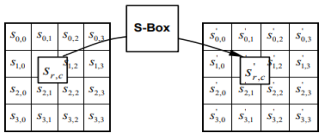
SubBytes

The SubBytes transformation is a key operation in AES encryption that substitutes every byte in the State array independently using a substitution box (S-box). The purpose of this step is to introduce non-linearity into the encryption process, making it harder for attackers to reverse-engineer the ciphertext.

The substitution is based on two steps:

1. **Calculating the Multiplicative Inverse in GF(2^8)**:**
   Each byte in the State is treated as an element of the Galois Field GF(2^8) (a finite field with 256 elements). In this field, arithmetic is performed under specific rules to ensure operations are closed. To substitute a byte, its multiplicative inverse in GF(2^8) is calculated. If the byte is 0x00, which has no inverse, it maps to itself.
2. **Applying the Affine Transformation:**
   After finding the multiplicative inverse, an affine transformation is applied to the result. This involves a bitwise operation where the byte is multiplied by a fixed matrix and XORed with a constant vector. This step adds additional diffusion to the data, enhancing security.

The result of these two steps for each byte is precomputed and stored in a lookup table called the S-box. During encryption, the S-box makes it quick to substitute bytes without recalculating these transformations every time. For example, if a byte has a value of 0x53, its replacement is found by looking at the row and column corresponding to 5 and 3 in the S-box table, which yields a value of 0xED .



**

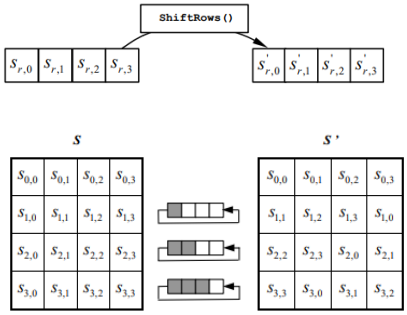
A **Galois Field**, often written as GF, is a special kind of mathematical system where you can perform arithmetic on a fixed set of numbers. What makes it unique is that this arithmetic "wraps around" in a specific way, ensuring that the results always stay within the set of numbers.
A Galois Field has a limited (finite) number of elements. For example, in **GF(2^8)**, there are exactly 2^8=256 elements, which correspond to all possible 8-bit binary numbers (from `00000000` to `11111111`, or 0 to 255 in decimal).
Addition/Subtraction: These are performed using bitwise XOR (exclusive OR). This operation ensures every addition wraps around within the field.
Multiplication: This follows more complex rules, using modular arithmetic based on a specific polynomial. The result always stays within the field's limits.

ShiftRows

The ShiftRows transformation is a key operation in the AES encryption algorithm that enhances diffusion by cyclically shifting the rows of the State array. This operation is crucial for ensuring that small changes in the input data have a broad impact on the ciphertext.

During this transformation:

1. **Row 0 remains unchanged.**
2. **Row 1 is cyclically shifted one position to the left. The leftmost byte wraps around to the end of the row.**
3. **Row 2 is cyclically shifted two positions to the left.**
4. **Row 3 is cyclically shifted three positions to the left (equivalent to one position to the right in a circular shift).**



MixColumns

This operation mixes the bytes within each column to enhance diffusion, ensuring that changes to a single byte affect the entire column.

Each column of the State is treated as a polynomial over GF(2^8) and multiplied modulo x^4+1 by a fixed polynomial:

$$a(x) = \{03\}\bullet x^3 + \{01\}\bullet x^2 + \{01\}x + \{02\},$$

where • is the multiplication operator in GF(2^8)

As matrix multiplication:

$$s'(x) = a(x) \text{ xor } s(x):$$

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

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

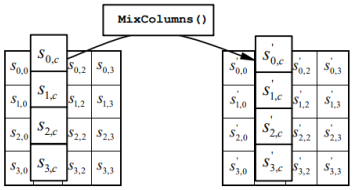
$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

Commented [8]: τέλειο!!

Commented [9]: <3

General illustration:



AddRoundKey

The AddRoundKey transformation is a simple yet critical step in AES. It involves XORing the current state matrix (S) with the round key (w) derived from the Key Expansion process. In the figure, the state matrix S consists of 4 rows and Nb columns, where Nb represents the block size in 32-bit words (4 columns for AES-128). The round key w is partitioned into subkeys (wl,wl+1,⋯,wl+Nb−1) where l is calculated as l=round×Nb. Each column of the state matrix is XORed with the corresponding subkey column, resulting in the updated state matrix S′. This operation integrates the round key into the encryption process, ensuring that the cipher remains secure.

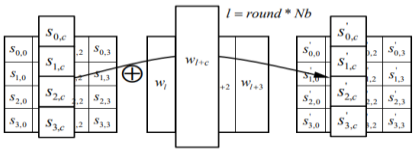
Mathematically, the transformation is expressed as:

$$s_{r,c}' = s_{r,c} \oplus k_{r,c}$$

***sr,c is the byte in row r and column c of the State.**

***kr,c is the corresponding byte of the round key.**

The AddRoundKey step introduces the cryptographic key into the State, ensuring the encryption is key-dependent. This operation is performed at the start of the encryption process (initial round) and at the end of each subsequent round.



Key Expansion

Key Expansion is the process of taking the initial 128-bit key (16 bytes) and generating a series of additional round keys needed for encryption. Each round of AES requires its own key, and AES-128 has 10 rounds plus an initial AddRoundKey step. This means a total of 11 round keys are needed.

How it works:

The initial 128-bit key is divided into 4 parts called words (w[0],w[1],w[2],w[3]). Each word is 4 bytes (32 bits).

These first 4 words are used as the first round key.

The subsequent words are generated iteratively using a formula. If the index i of the word being generated is a multiple of 4, the new word (w[i]) is calculated by transforming the previous word (w[i−1]) and XORing it with the word from four steps earlier (w[i−4]). This transformation involves rotating the previous word by one byte to the left, substituting each byte using the AES S−box, and adding a constant called Rcon, which changes for each round. For other cases where imod4≠0, the word is simply generated by XORing the previous word (w[i−1]) with w[i−4].

This process continues until 44 words are generated in total. These 44 words are then grouped into sets of four to form the 11 round keys required for encryption. For example, the first round key consists of w[0] to w[3], the second consists of w[4] to w[7],

and so on. By generating unique keys for each round, Key Expansion ensures that every round of AES operates securely and without revealing patterns that attackers could exploit.

IV. DECRYPTION

Decryption, like encryption, operates on blocks of data, typically 128 bits in size. AES-128 decryption uses a symmetric key, which means the same 128-bit key is utilized for both encryption and decryption. The decryption process is designed to undo the transformations applied during encryption, specifically reversing substitution, permutation, and mixing steps, while employing the round keys in reverse order.

The AES-128 Decryption Process

The ciphertext undergoes an initial XOR operation with the last round key derived from the key schedule. This step is identical to the AddRoundKey step in encryption, as XOR is its own inverse.

For rounds 1 through 9, the following operations are applied in reverse order compared to encryption:

Inverse ShiftRows

The rows of the state matrix are shifted to the right (opposite of the left shifts in encryption). Each row is shifted by an amount corresponding to its index: the first row remains unchanged, the second row shifts by one position, the third by two, and the fourth by three.

Inverse SubBytes

Each byte in the state matrix is replaced using the inverse S-box, which reverses the substitution applied during encryption. This step restores the original byte values that were substituted during the SubBytes operation.

Inverse MixColumns

The columns of the state matrix undergo a transformation using the inverse of the MixColumns matrix. This operation reverses the diffusion effect introduced during encryption, ensuring that data spread across multiple columns is recombined correctly.

Inverse AddRoundKey

The state matrix is XORed with the corresponding round key. For decryption, round keys are applied in reverse order, starting from the last key generated during the key expansion process and working backward to the first key.

Final Round

The last decryption round is slightly different because it omits the Inverse MixColumns step. Instead, it consists of the following:

- Inverse ShiftRows
- Inverse SubBytes
- AddRoundKey using the original encryption key.

Key Expansion

The decryption process relies heavily on the key schedule generated during encryption. The AES key expansion algorithm produces a series of round keys from the original encryption key. During decryption, these round keys are applied in reverse order, starting from the final round key and progressing backward to the initial key. This reverse application ensures that the decryption process accurately retraces the transformations applied during encryption.

V. AES-128 VULNERABILITIES

AES-128 remains secure for most practical purposes when properly implemented and used in secure modes of operation. Brute-Force and Cryptanalysis attacks have been

described as being far from feasible in practice, due to their large computational complexity. However, no cryptographic system is entirely without vulnerabilities.

Side-Channel Attacks

A side-channel attack exploits indirect effects of a system or its hardware (Wright, 2021) . These effects include information on timing, power consumption, electromagnetic leaks, thermal variations, sound among others. Here we present some of these techniques implemented to recover full AES keys, with some needing as little as a single encryption round.

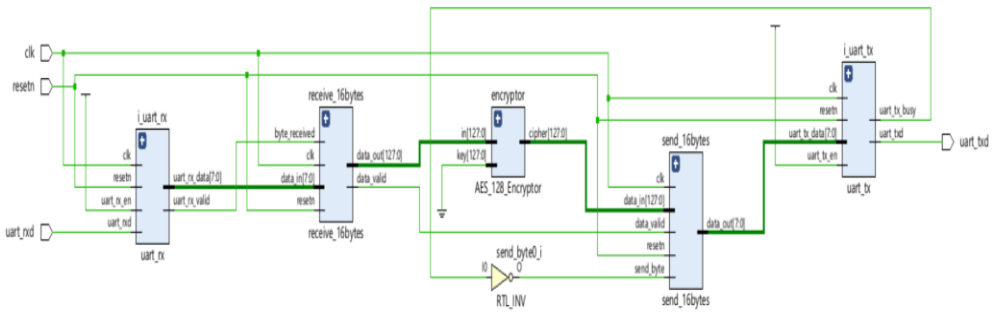
Cache Attacks target shared memory pages between non-trusting processes and have been proven effective in extracting full AES keys in five to six rounds (Giri & Menezes, 2016). **Electromagnetic Interference Attacks** capture the electromagnetic emissions on processors during encryption. A single electromagnetic trace is enough to capture the AES key on an ARM processor (Masure & Strullu, 2021). **Timing Attacks** exploit variations in the time taken by the cryptographic algorithm to process inputs. One of the weaknesses in the AES design is that it cannot be widely implemented to run in constant time, and as such timing attacks pose a security threat (Bernstein, 2005). **Power Analysis Attacks** leverage power consumption patterns of the hardware during encryption to infer sensitive data. These methods have been used to break AES encryption and in theory could require only a single round (Lo et al., 2016, Putra et al., 2020).

VI. IMPLEMENTATION OF AES-128 IN VERILOG

In order to better understand its inner workings, we implemented an AES-128 Encryption and Decryption core in Verilog. Furthermore, with the goal to verify and test our design on FPGAs we integrated universal asynchronous receiver / transmitter (UART) communication.

Encryption and Communication Pipeline

Data enters the system through the UART receiver, where it is converted to an 8-bit format. A Series to Parallel module collects one byte at a time and assembles them into 16 bytes (128-bit) blocks , which are fed into the AES-128 encryptor or decryptor. The encryptor processes the plaintext with the given key and outputs the corresponding ciphertext. Similarly, in the case where our core worked as a decryption core, it would decrypt a cipher back into plaintext. Finally, the **send_16bytes** module breaks the 128-bit block into 8-bit segments for serial transmission via the UART transmitter. Below you can see the pipeline generated by Vivado.



Implementation Details

UART Receiver (i_uart_rx): The UART receiver module is responsible for receiving serial data from another device communicating through UART and converting it into an 8-bit parallel format. This module detects valid incoming data bytes, which are output through the uart_rx_data[7:0] signal. A uart_rx_valid signal is asserted when new data is available, allowing downstream modules to process the received bytes.

Serial to Parallel Collector (receive_16bytes): This module collects 8-bit data from the UART receiver and assembles it into a 128-bit block, suitable for encryption. It continuously monitors the incoming data and, upon receiving 16 bytes, sets the data_valid signal to indicate that the 128-bit block (data_out[127:0]) is ready for processing by the encryptor. The byte_received signal is used internally to count incoming bytes and ensure accurate data assembly.

AES-128 Encryptor (encryptor): The AES-128 encryptor is the core module responsible for performing the encryption operation. It accepts two 128-bit inputs: the plaintext data (in[127:0]) and the encryption key (key[127:0]). Using the AES algorithm, it produces a 128-bit ciphertext (cipher[127:0]), which is output for further processing. The encryptor operates synchronously with the system clock and ensures high-speed encryption with precise timing to match the system's requirements.

AES-128 Decryptor: We have also developed an AES-128 Decryption module, that performs the inverse operations, converting cipher into plaintext.

Note that for both the Encryption and the Decryption Cores the key is hardcoded and handled as a parameter. Moreover, we have developed two methods for the MixColumn and InvMixColumns modules, the first which is present in the examples uses LUTS, maximising area and minimizing space, while the inverse holds true for the second, which includes dedicated hardware that compute the GF multiplications.

Parallel to Serial Distributor (send_16bytes): This module takes the 128-bit ciphertext from the encryptor and breaks it into 8-bit segments for transmission via the UART transmitter. It monitors the data_valid signal from the encryptor and ensures that each byte of the ciphertext is transmitted sequentially. The buffer_ready signal indicates when the module is ready to send the next byte, while the byte_sent signal tracks the transmission of individual bytes.

UART Transmitter (i_uart_tx): The UART transmitter module handles the serialization of 8-bit parallel data for transmission over the UART TX line. It accepts data from the send_16bytes module and ensures its accurate delivery. The uart_tx_busy signal indicates ongoing transmission, preventing data overwrites.

clk: The clock signal is the primary timing reference for the entire system. It maintains a consistent frequency and drives all synchronous modules. Its steady toggling ensures coordinated operation across the design.

resetn: This active-low reset signal initializes all modules. At the beginning of the simulation, it is pulled low, clearing all internal registers and ensuring a known starting state. When it transitions high, the system begins normal operation.

UART RX Signals:

received_data: This signal shows the data received from the UART receiver module.

Each byte is captured and validated through the corresponding control signals. Notice that as the UART reception completes, the received_data line goes high, indicating successful data capture.

uart_rx_data[7:0]: These bits represent the 8-bit chunks of data being received. The values align with the input stream, showcasing the received data integrity.

uart_rx_valid: This signal goes high when valid data is available on uart_rx_data[7:0]. Its transitions indicate the exact timing when data reception is completed and ready for further processing.

UART TX Signals:

uart_tx_busy: This signal indicates the transmitter's status. When high, the UART transmitter is actively sending data. Its transitions demonstrate when the system is ready to handle new data.

data_to_send[7:0]: These bits represent the 8-bit chunks of data being prepared for transmission. Each byte matches the encrypted data ready to be sent out.

uart_tx_data[7:0]: The actual transmitted data appears here. When this signal toggles, it corresponds to the transmission of processed and encrypted data.

receive_16bytes Module Signals:

data_in[7:0]: This input receives incoming bytes, byte by byte. Its values directly correspond to the uart_rx_data[7:0] signal.

data_out[127:0]: Once all 16 bytes are received, they are combined into a single 128-bit block for processing by the encryption engine. The transition of this signal from all zeros to a valid data block indicates successful reception of 16 bytes.

data_valid: This signal goes high once the full 128-bit block is assembled, marking it as ready for encryption. Its rising edge aligns with the completion of the 16-byte reception process.

byte_received: This signal tracks individual byte receptions. Each time it goes high, a new byte has been captured and added to the assembly buffer.

Note that the UART Verilog Implementation is from GitHub user ben-marshall (uart repository).

send_16bytes Module Signals:

data_in[127:0]: This input receives the 128-bit encrypted block from the AES-128 engine. Its transition from zero to a valid encrypted block marks the start of the transmission process.

data_out[7:0]: This signal breaks down the 128-bit block into individual bytes for transmission via UART. Each byte's transition represents its readiness to be sent.

buffer_ready: This signal goes high when the module is ready to send the next byte. Its synchronization with send_byte ensures efficient data transmission.

send_byte: This signal pulses high to trigger the transmission of each byte, one at a time, through data_out[7:0].

data_valid: This signal validates the data being transmitted. It ensures that the outgoing bytes are correctly synchronized with the transmission process.

AES-128 Encryption Engine Signals:

in[127:0]: This signal receives the 128-bit input data for encryption. Its transition from zero to valid input data aligns with the completion of the receive_16bytes module.

key[127:0]: This static signal holds the encryption key. Its constant value ensures consistent encryption results.

cipher[127:0]: The output of the AES encryption engine. Its transition from zero to the encrypted result confirms successful encryption. This signal provides the data block that will be sent through the send_16bytes module.

Simulation Analysis

The waveform results from the simulation confirm the correct functionality of the AES-128 encryption and decryption core system and its supporting modules. The signals behave as expected, validating the interaction between the UART modules, data processing, and the encryption engine.

The testbench emulates a UART driver communicating at 9600 baud rate. The driver sends 1 bit every 10416ns, effectively sending a single data byte every 104160ns (10416ns/bit for a total of 10 bits, 1 start bit + 8 data bits + 1 stop bit). After sending 16 bytes, the testbench starts listening, and displaying the bytes stored inside the uart_tx 8-bit memory, every 104160ns.

For the examples shown below the values we used were taken from the official NIST Publication for AES-128:

Plaintext	**3243f6a8885a308d313198a2e0370734**
Key	**2b7e151628aed2a6abf7158809cf4f3c**
Expected Cipher	**3925841d02dc09fbdc118597196a0b32**

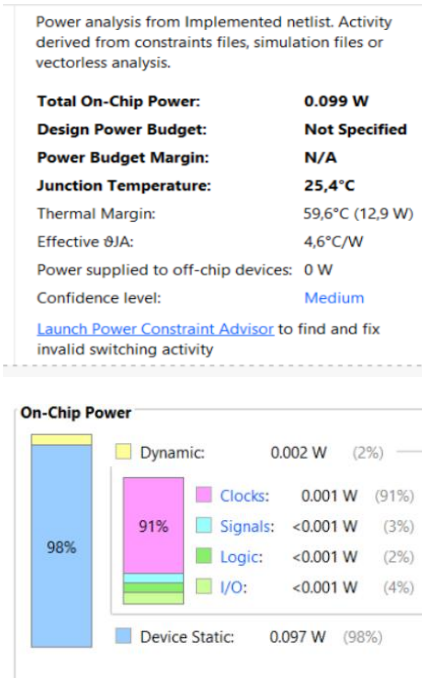
By examining the results and waveforms from the simulation we verify the correct operation of the AES-128 Encryption Core, implemented on an FPGA. Similar results were provided from simulating the Decryption Core.

```
VCD info: dumpfile AES_128_FPGA_tb.vcd opened for output.
[byte  0]     Sent Byte: 32 (104170000ns)
[byte  1]     Sent Byte: 43 (208330000ns)
[byte  2]     Sent Byte: f6 (312490000ns)
[byte  3]     Sent Byte: a8 (416650000ns)
[byte  4]     Sent Byte: 88 (520810000ns)
[byte  5]     Sent Byte: 5a (624970000ns)
[byte  6]     Sent Byte: 30 (729130000ns)
[byte  7]     Sent Byte: 8d (833290000ns)
[byte  8]     Sent Byte: 31 (937450000ns)
[byte  9]     Sent Byte: 31 (1041610000ns)
[byte 10]     Sent Byte: 98 (1145770000ns)
[byte 11]     Sent Byte: a2 (1249930000ns)
[byte 12]     Sent Byte: e0 (1354090000ns)
[byte 13]     Sent Byte: 37 (1458250000ns)
[byte 14]     Sent Byte: 07 (1562410000ns)
[byte 15]     Sent Byte: 34 (1666570000ns)
[byte  0] Received Byte: 39 (1770730000ns)
[byte  1] Received Byte: 25 (1874890000ns)
[byte  2] Received Byte: 84 (1979050000ns)
[byte  3] Received Byte: 1d (2083210000ns)
[byte  4] Received Byte: 02 (2187370000ns)
[byte  5] Received Byte: dc (2291530000ns)
[byte  6] Received Byte: 09 (2395690000ns)
[byte  7] Received Byte: fb (2499850000ns)
[byte  8] Received Byte: dc (2604010000ns)
[byte  9] Received Byte: 11 (2708170000ns)
[byte 10] Received Byte: 85 (2812330000ns)
[byte 11] Received Byte: 97 (2916490000ns)
[byte 12] Received Byte: 19 (3020650000ns)
[byte 13] Received Byte: 6a (3124810000ns)
[byte 14] Received Byte: 0b (3228970000ns)
byte_sent asserted
[byte 15] Received Byte: 32 (3333130000ns)
../../sim_1/new/AES_128_FPGA_tb.v:87: $finish called at 3334130000 (1ps)
```

Power-Timing-Area Analysis

We then tested the core in Vivado, in order to analyze its power, timing and area requirements

Power Analysis



Key Power Metrics

Total On-Chip Power: The total on-chip power consumption is reported as 0.099 W, combining both static and dynamic components. This value represents the overall power demand of the module under the specified conditions, reflecting the design's energy efficiency.

Dynamic Power: Dynamic power, which arises from switching activities in the circuit, accounts for 0.002 W (approximately 2% of the total power). This indicates low dynamic power consumption, a desirable trait for designs requiring frequent activity or high throughput.

Static Power: The static or leakage power is 0.097 W (approximately 98% of the total power). This significant contribution suggests that power gating techniques might not be employed in this design. Optimization techniques could be explored to minimize leakage, depending on the application and constraints.

Breakdown by Component

Clocks: The clocking network, which is crucial for synchronizing operations, consumes 0.001 W (91% of the dynamic power). This high percentage indicates the dominant role of the clock in the overall dynamic power profile, typical for designs with large clock domains or high frequencies.

Signals: Signal routing consumes less than 0.001 W (3% of the dynamic power). This reflects the efficient management of interconnects within the module, minimizing unnecessary switching activities.

Logic: The logic components consume less than 0.001 W (2% of the dynamic power). This low consumption reflects efficient synthesis and optimization of the logic paths during the design implementation phase.

I/O: Input/Output pads contribute less than 0.001 W (4% of the dynamic power). This highlights the limited role of external interfaces in the module's power profile, likely due to the encryption focus of the design.

Timing Analysis

The timing analysis of the AES-128 encryption module provides valuable insights into the module's performance across different clock frequencies, with the target FPGA's (Nexys A7 100T) operating frequency limit set at 100 MHz. For a 100MHZ clock, the worst negative slack

(WNS) was measured at -28.78 ns, and the total negative slack (TNS) was -3626.91 ns, indicating that the design fails to meet timing requirements at the maximum frequency. This significant timing violation suggests that certain critical paths within the design are too slow to operate reliably at this high frequency. However, when the clock frequency was reduced to 25.77 MHz (38.8 ns clock period), the timing improved significantly, with a WNS of 0.02 ns and a TNS of 0 ns, indicating that all paths meet the timing constraints at this frequency. Similarly, at 25 MHz (40 ns clock period), the WNS was measured at 1.22 ns, with no total negative slack, further confirming that the design operates comfortably at these lower frequencies. At 25.87 MHz (38.78 ns clock period), the analysis also reported 0.00 WNS, which is the theoretical limit for a clock on the design.

The analysis confirms the module's reliability at moderate frequencies while highlighting opportunities for optimization at higher speeds, such as pipelining the design, or reducing logic depth.

25MHz

Setup		Hold	
Worst Negative Slack (WNS):	1,220 ns	Worst Hold Slack (WHS):	
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	
Number of Failing Endpoints:	0	Number of Failing Endpoints:	
Total Number of Endpoints:	1020	Total Number of Endpoints:	

All user specified timing constraints are met.

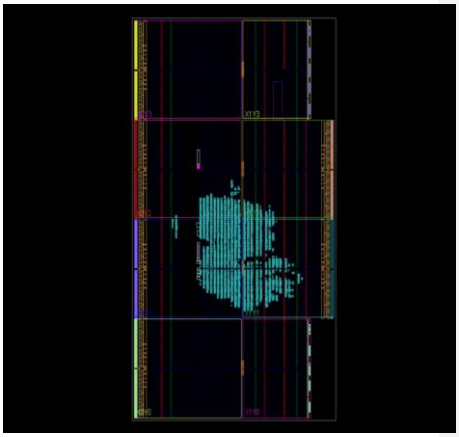
100MHz

Setup		Hold	
Worst Negative Slack (WNS):	-28,780 ns	Worst Hold Slack (WHS):	
Total Negative Slack (TNS):	-3626,912 ns	Total Hold Slack (THS):	
Number of Failing Endpoints:	191	Number of Failing Endpoints:	
Total Number of Endpoints:	1020	Total Number of Endpoints:	

Timing constraints are not met.

Area Analysis

The area analysis provides a detailed understanding of how the AES-128 core utilizes the resources available on the FPGA. By examining the implementation results in Vivado, we can observe the layout of the design and how efficiently it is mapped onto the FPGA's available area. The area utilization screenshot highlights the distribution of logic cells, look-up tables (LUTs), and other resources required to implement the design.



Area on the Nexys A7 100T FPGA by Vivado

References

Bernstein, D. (2005). ***Cache-timing attacks on AES*****.**

Giri, R., & Menezes, B. (2016). Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks. ***IEEE European Symposium on Security and Privacy (EuroS&P)*****, 261-275.**

Lo, O., Buchanan, W., & Carson, D. (2016). Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA). ***Journal of Cyber Security Technology*****,** ***1*****(2), 88-107. https://doi.org/10.1080/23742917.2016.1231523**

Masure, L., & Strullu, R. (2021). Side Channel Analysis against the ANSSI's protected AES implementation on ARM. ***Cryptology ePrint Archive, Paper 2021/592*****.**

National Institute of Standards and Technology. (2001). ***ADVANCED ENCRYPTION STANDARD (AES)*****.**

Putra, S. D., Sumari, A. D. W., Asrowardi, I., Subyantoro, E., & Zagi, L. M. (2020). First-Round and Last-Round Power Analysis Attack Against AES Devices. ***2020 International Conference on Information Technology Systems and Innovation (ICITSI), Bandung, Indonesia, 2020, pp. 410-415*****. 10.1109/ICITSI50517.2020.9264976**

Wright, G. (2021). ***What is a side-channel attack?*** **TechTarget. https://www.techtarget.com/searchsecurity/definition/side-channel-attack**