electronics

MDPI

Article

Design and Analysis of FSM-Based AES Encryption on FPGA Versus MATLAB Environment

Sunny Arief Sudiro [1,*], Fauziah [2], Ragiel Hadi Prayitno [3], Bayu Kumoro Yakti [3], Sarifuddin Madenda [4] and Michel Paindavoine [5]

[1] Master Program of Information Tecnology, Sekolah Tinggi Manajemen Informatika dan Komputer Jakarta STI&K, Daerah Khusus Ibukota Jakarta 12140, Indonesia
[2] Computer Science, Gunadarma University, Depok 16424, Indonesia; fauziah87@staff.gunadarma.ac.id
[3] Electrical Engineering, Gunadarma University, Depok 16424, Indonesia; ragielhp@staff.gunadarma.ac.id (R.H.P.); bayuyakti@staff.gunadarma.ac.id (B.K.Y.)
[4] Doctoral Program of Information Technology, Gunadarma University, Depok 16424, Indonesia; sarif@staff.gunadarma.ac.id
[5] LEAD UMR CNRS 5022, Université Bourgogne Europe, 21000 Dijon, France; michel.paindavoine@ube.fr
* Correspondence: sasudiro@staff.jak-stik.ac.id

Abstract

The present paper compares and analyzes the design of AES-128 encryption and decryption using Finite State Machine (FSM) architecture on FPGA and MATLAB platforms. This study aims to evaluate performance disparities in terms of execution time, throughput, and hardware efficiency under identical input data and key conditions. The FSM-based AES algorithm was modeled in MATLAB for functional validation and synthesized on an Artix-7 FPGA using VHDL. The experimental results confirmed that both platforms produced identical ciphertext and plaintext outputs, verifying the correctness of the processes employed. However, the FPGA demonstrated significantly better performance in terms of execution speed. Encryption and decryption times were measured in microseconds on the FPGA, while similar operations on the MATLAB platform required hundreds of milliseconds. The FPGA implementation achieved throughput of 872.53 Mbps for encryption and 858.49 Mbps for decryption with area usage of 1263 and 1428 slices, respectively. This yields an efficiency of 0.691 and 0.601 Mbps/slice, which is considered efficient according to established benchmarks. Compared to previous MATLAB-only and FPGA pipelined implementations, the current design strikes a balance between resource usage and performance, making it ideal for lightweight cryptographic applications in embedded systems. These results provide practical insights into selecting platforms for secure, real-time data processing.

Keywords: AES-128; finite state machine (FSM); FPGA; MATLAB; encryption; decryption; throughput; efficiency; hardware implementation; embedded cryptography

Check for updates

Received: 28 November 2025
Revised: 12 January 2026
Accepted: 13 January 2026
Published: 5 February 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

In the modern era of digital communication, data security is a fundamental concern due to increasing cyberattack and unauthorized access threats. Among the various cryptographic algorithms, the Advanced Encryption Standard (AES) has emerged as the predominant algorithm due to its robustness, efficiency, and standardized structure [1]. Implementing AES with a Finite State Machine (FSM) architecture provides a modular, systematic approach to managing encryption processes, which is particularly well-suited

for hardware implementations [2]. Field-programmable gate arrays (FPGAs) offer reconfigurable logic and parallel processing capabilities, making them an ideal platform for high-throughput cryptographic systems [3,4]. Previous studies have demonstrated FPGA implementations achieving speeds of up to 3 Gbps on devices such as Spartan-6 and Artix-7, with optimizations targeting area, power, or speed [5,6]. Conversely, MATLAB is a versatile environment for modeling and simulation, enabling the validation of algorithms and verification of functionality before hardware deployment. Its high-level syntax and built-in matrix operations allow for the quick development of AES encryption for various applications, including image, video, and signal processing [7,8].

Despite extensive research on AES implementation, a detailed comparative analysis of FSM-based AES encryption in FPGA and MATLAB environments remains limited. Each platform introduces unique constraints and advantages in terms of resource utilization, timing performance, and design complexity [9]. This study aims to address this gap by presenting a structured design for FSM-controlled AES encryption implemented on FPGA hardware and MATLAB simulations. Through empirical analysis and synthesis reports, the research evaluated critical metrics such as throughput, latency, logic utilization, and scalability. While earlier studies have documented encryption times as low as 87 ms in MATLAB and throughput above 4 Gbps in optimized FPGA architectures [6,8], few have systematically compared these platforms within a unified design framework. These findings aim to provide researchers and engineers with practical insights for selecting the most suitable platform for their application-specific requirements in the design of secure embedded systems [10].

Reference [8] explores the application of AES-128 encryption to grayscale images using MATLAB. The authors provide a structured overview of AES operations, including SubBytes, ShiftRows, MixColumns, AddRoundKey, and Key Expansion. They then demonstrated the integration of these operations within a modular MATLAB program. The study included experiments on three grayscale images with dimensions of 120 × 160, 256 × 256, and 336 × 412 pixels, respectively. Observed encryption and decryption times were 50.125 and 57.266 s for the smallest image, 115.703 and 154.032 s for the medium-sized image, and 214.047 and 293.125 s for the largest image, respectively. The findings indicated that processing time increased substantially in proportion to image size, thereby underscoring the performance limitations of using MATLAB for large-scale data processing. Despite its instructional value, the paper has several critical limitations. First, the present volume does not include finite state machine (FSM) design details or diagrams despite the title suggesting such an approach. Implementation is limited to MATLAB, and there is no comparison with hardware platforms, such as FPGAs, which would be essential for evaluating performance in real-time systems [8].

Paper [6] provides a thorough implementation of the 128-bit Advanced Encryption Standard (AES) using MATLAB, focusing on simulating encryption and decryption processes. The authors provide a thorough description of the structure of the AES algorithm and the implementation of its core byte-oriented operations, including SubBytes, ShiftRows, MixColumns, and AddRoundKey. These operations are outlined using MATLAB functions. The experimental section includes measurements of key setup time, single-round encryption time, and full encryption and decryption times. The highest recorded encryption and decryption times were 87.57 and 88.007 milliseconds, respectively. These results were obtained using a modern PC with 4 GB of RAM and an Intel i5 processor. These results were considerably faster than those of certain embedded implementations (e.g., 2.06 s in an EDK-based system), demonstrating MATLAB's efficiency for prototyping. Despite the aforementioned merits, the study is subject to several limitations. First, it lacks broader performance metrics, such as memory usage, throughput, and scalability for large datasets.

Additionally, the design omits formal modeling, such as state diagrams or finite state machine (FSM) representations, which would have strengthened the clarity and portability of the AES logic structure.

Efficient hardware implementation is a central theme in leading electrical engineering and computer science journals [11]. For example, the authors' proposed method enhances a conventional, non-pipelined Advanced Encryption Standard (AES) algorithm by integrating a PN sequence generator that dynamically creates S-box values and the initial key. This approach centers on a linear feedback shift register (LFSR) with specific feedback taps and a secret seed value. It aims to enhance security by making these fundamental components unpredictable to an attacker. The security of this modification was evaluated using the Strict Avalanche Criterion (SAC). However, a critical evaluation reveals certain shortcomings in the research. The selection of the PN generator's parameters is justified solely as "proof of concept" without deeper analysis of their cryptographic impact. Furthermore, the claim that the design is "invulnerable to attacks" is an overstatement based solely on avalanche effect results. Furthermore, while the design improves throughput, it comes at the cost of significantly increased area usage. This trade-off is acknowledged but not thoroughly examined for its practical implications. For the hardware implementation, the authors' primary design was implemented on a Spartan6 XC6SLX150 FPGA device, with plaintext and a 128-bit key as inputs to the AES algorithm. The synthesis results for this component showed a resource utilization of 5566 slices. The design achieved a maximum operating frequency (F_(max)) of 237.45 MHz, resulting in a calculated throughput of 3.03 Gbps based on a 10-cycle latency. The efficiency, measured as Mbps/slice, was 0.54 for this device implementation [11].

Several other studies have focused on various aspects using FPGAs. One study [12] examined the use of pipelining techniques to reduce area and delay time in parts of the AES process that require a large number of resources, such as the Sub-Bytes section. This approach has been shown to increase throughput to 79.7 Gbps and FPGA efficiency to 13.3 Mbps/slice. Research [13] focuses on anticipating unauthorized access to data by external parties by implementing the AES algorithm in hardware using an FPGA, which is not overly complex, flexible, or efficient. The present study focuses on utilizing Slice Registers (SRs), Look-Up Tables (LUTs), Input/Output (I/Os), and Global Buffers (BUFGs) to enhance data security at the hardware level. Research [14] focuses on using a real-time hardware platform to encrypt multimedia data, especially video, using the AES algorithm. The proposed system uses a CMOS camera as an input device and processes the data directly using an AES encryption processor developed with Xilinx System Generator. This processor is integrated as a dedicated peripheral with a Microblaze 32-bit soft RISC processor. A comparison of previous AES research can be seen in Table 1.

For the target application domain, an FSM-based AES design is preferable because it offers an optimal balance of performance, resource efficiency, and design simplicity—critical requirements for secure embedded systems. An FSM-controlled architecture significantly reduces hardware area and power consumption by reusing a single round data path across multiple encryption rounds, making it well suited for low- to mid-range FPGA devices. This reduction is achieved by comparison with fully pipelined or unrolled designs. The finite state machine's deterministic sequencing provides precise control over each AES operation, resulting in predictable latency and straightforward integration with embedded processors, memory interfaces, and control logic. Additionally, the modular, state-oriented structure simplifies verification and validation against high-level reference models, such as MATLAB implementations. This enhances the reliability and portability of the design. While FSM-based implementations generally have lower peak throughput than highly parallel architectures, it remains sufficient for many real-time and embedded security

applications where cost, scalability, and power efficiency are prioritized over extreme data rates.

Table 1. Side by side comparison of previous research on AES.

Ref.	Device Family	Resource Usage (Slices)	Fmax (MHz)	Latency (Cycles)	Throughput	Efficiency (Mbps/Slice)
[5]	Spartan-6/Virtex-5/6/Spartan-3 (comparisons shown)	e.g., Virtex-5 AES-CTR: 5974 slices	622.4	Not reported	Not reported	Not reported
[11]	Spartan-6 XC6SLX150	5566 slices	237.45	10	3.03 Gbps	0.54
[12]	Virtex-5 XC5VLX110T	5974 slices	622.4	N/R (fully pipelined design)	79.7 Gbps	13.3
[14]	Spartan-3A DSP XC3SD3400A	13,564 slices	100.329	11	4.28 Gbps	0.32
[15]	Zynq-7000 XC7Z020 (Zedboard)	2794 slices (AES-128)	219	19 (then II = 1)	28 Gbps	10

In light of these findings, our study proposes a comprehensive design and comparative analysis of FSM-based AES encryption on MATLAB and FPGA platforms. Leveraging MATLAB for rapid simulation and functional validation and FPGA for high-throughput, resource-constrained deployment aims to bridge the gap between software prototyping and hardware realization. The evaluation process focuses on critical performance metrics, incorporating factors such as execution time, throughput, and resource utilization. Furthermore, this study addresses the limitations of prior research by incorporating FSM modeling and detailed synthesis results. This dual-platform investigation provides valuable insights for researchers and engineers seeking optimized, tailored cryptographic solutions that balance algorithmic flexibility and real-time hardware efficiency.

2. Materials and Methods

The Advanced Encryption Standard (AES) is a symmetric block cipher algorithm that secures digital data using substitution, permutation, and key-mixing operations. The research is conducted in several steps:

1. An in-depth literature review of various encryption and decryption models with a focus on algorithmic structures and implementation methods to identify their strengths and weaknesses.
2. A modified AES algorithm employing a finite state machine (FSM) approach was then designed using MATLAB. The algorithm was developed as a software prototype and tested using multimedia data (text, audio, images, and videos) in bitstream form.
3. The modified, FSM-based algorithm was then converted into a circuit design and implemented in an FPGA using VHDL.
4. The testing process was conducted through behavioral simulation with numerical data under the assumption that the multimedia data had been prepared for encryption and decryption.
5. The final stage yielded a functional decryption IP core ready for FPGA integration.

2.1. Implementation Environment and Design Tools

The AES architecture proposed in this study is resource-efficient and was developed and validated using a standard hardware and software co-design workflow. The hardware description logic was written in VHDL, and the design was synthesized using the Xilinx ISE 14.7 design suite (Xilinx, Inc., San Jose, CA, USA). Comprehensive behavioral simulations

were conducted using the integrated ISim Simulator to guarantee logical correctness and analyze signal timing. The clock frequency constraint has a default setting of 100 MHz, which is equivalent to 10,000 periods of a wave. The final synthesized IP core was implemented and tested on a Digilent Nexys board (Digilent co NI, 11500 N Mopac Expy, Austin, TX, USA) with a Xilinx Artix-7 100T FPGA Part Number: XC7A100T-1CSG324C). The design properties of the FPGA are shown in Figure 1. This board served as the target hardware platform for all performance evaluations.

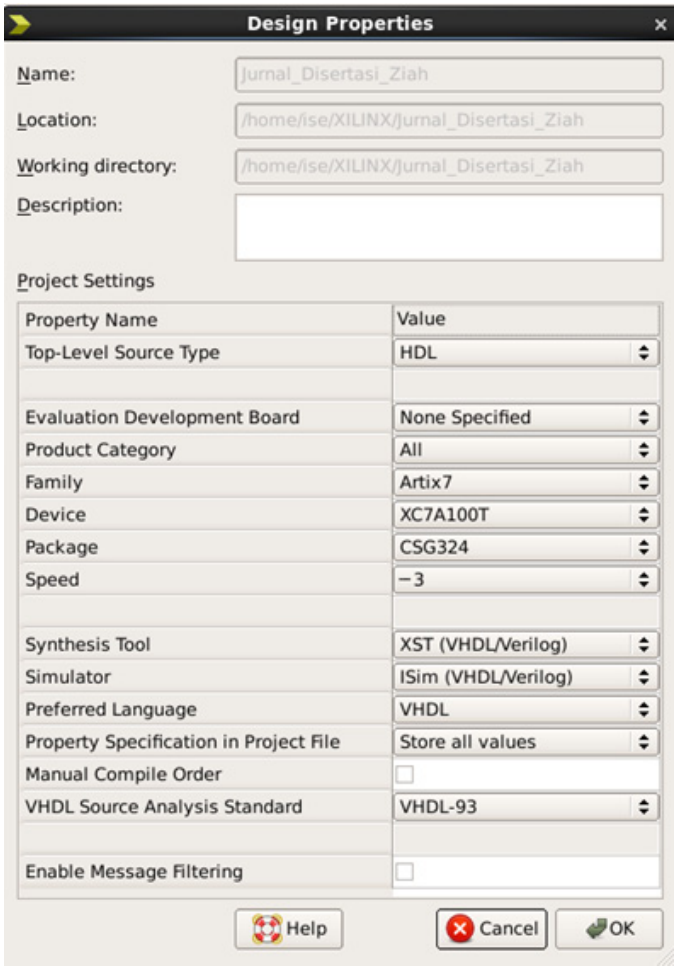


Figure 1. Design Properties used in this Research.

In conjunction with the hardware development, MATLAB was used as a high-level modeling and simulation tool. It was employed to validate the functional behavior of the AES algorithm, verify intermediate transformation stages, and provide baseline performance metrics in a controlled software environment. The design methodology for all transformations employed sequential logic techniques governed by a finite state machine (FSM) because the output of each stage depended on previous inputs and required storage elements.

2.2. Architectural Design: An FSM-Based AES

The primary architectural innovation of this work is the redesign of the dataflow of the AES algorithm, which is managed by a finite state machine. This approach was selected to overcome the inherent inefficiencies of conventional sequential logic in field-programmable gate array (FPGA) implementations. In such cases, hardware components are often redundantly re-declared for each computational round. Our FSM-based architecture enables

the central innovation of this research: the use of reusable components. Key functional units, particularly the resource-intensive XOR gate arrays required for the AddRoundKey and MixColumns transformations, were modeled as shared resources. These components, including specialized two-, three-, and four-input XOR blocks, were instantiated once and subsequently invoked programmatically by various states as needed throughout the cryptographic process. This paradigm significantly reduced the overall hardware footprint.

Concurrently, MATLAB was used on the software side. The research focused on comparing the processing speeds of MATLAB and FPGA from input to output. MATLAB and FPGA were given a similar method (AES with FSM) and similar input data, enabling analysis of the results and the speed of each platform.

The architecture comprised four primary transformations, each managed as a distinct process within the FSM: KeySchedule, AddRoundKey, SubBytes, and ShiftRows. To optimize performance further, the SubShift entity was designed to execute the SubBytes and ShiftRows transformations in parallel. This combined the nonlinear substitution and byte-wise permutation steps into a single, efficient clock cycle operation.

2.3. MATLAB Algorithms

The Advanced Encryption Standard (AES) is one of the most widely used symmetric encryption algorithms. It is known for its security, speed, and efficiency in hardware and software implementations. A Finite State Machine (FSM) approach is often used to manage the complexity of AES operations in hardware environments, such as Field Programmable Gate Arrays (FPGAs).

This algorithm uses a finite state machine (FSM) to control the flow of operations across multiple rounds of the AES encryption process. Encryption begins with an initial key addition (AddRoundKey), followed by a series of transformation steps: SubBytes, ShiftRows, MixColumns, and AddRoundKey. These steps are repeated for ten rounds. The FSM architecture ensures structured, modular execution, making it highly suitable for hardware-based cryptographic systems that require deterministic control and optimized performance.

This implementation incorporates a key scheduling function that generates round keys dynamically. It also handles the transition logic between states, including a special case for the final round where the MixColumns step is omitted, as specified in the AES. The timing methodology in MATLAB 2016b uses the tic/toc function. This research used a 10th generation i7 processor with 16 GB of RAM, and MATLAB performed byte-oriented operations. This research's decryption process utilized AES MATLAB algorithms, which are described in Appendix A.

AES encryption transforms plaintext into ciphertext. The decryption process restores the original message. A Finite State Machine (FSM) approach offers a structured and modular design, making it an efficient implementation of AES decryption, particularly in hardware platforms such as FPGAs.

This algorithm uses an FSM to manage the sequence of inverse operations across ten rounds to implement AES decryption. The process begins by applying the last round key and performing **Inverse ShiftRows** and **Inverse SubBytes**. Then, a loop executes **AddRoundKey**, **Inverse MixColumns**, and other inverse transformations in the reverse order of encryption. The final step of the procedure utilizes the original key for a final key addition to retrieve the plaintext.

The FSM approach organizes each decryption phase into distinct states, ensuring clarity, modularity, and synchronization. This makes the approach ideal for hardware synthesis or simulation environments, where precise control over each step is essential.

2.4. FSM Logic, Dataflow, and Memory Structure for FPGA

Each of the encryption and decryption processes is governed by a dedicated seven-state finite state machine (FSM), which controls the data flow and state transitions, as depicted in Figures 2 and 3, respectively.

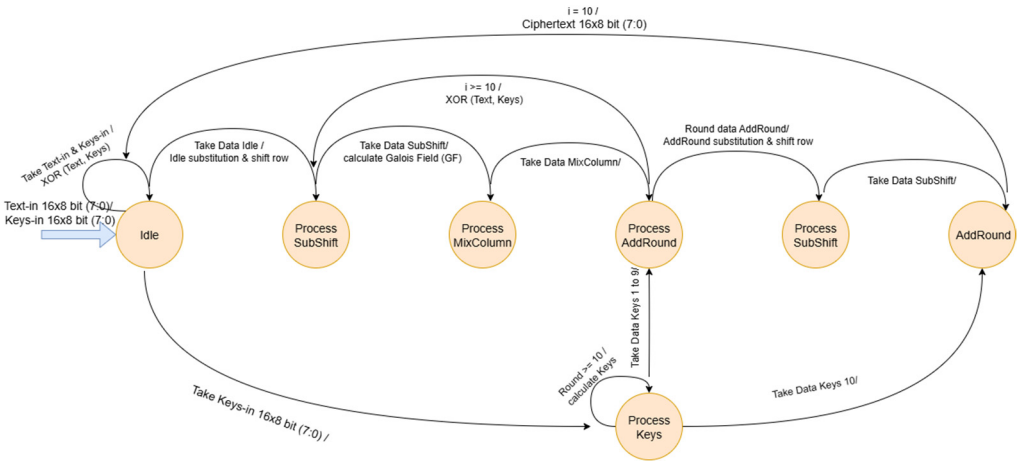


Figure 2. FSM-Based Encryption Integration.

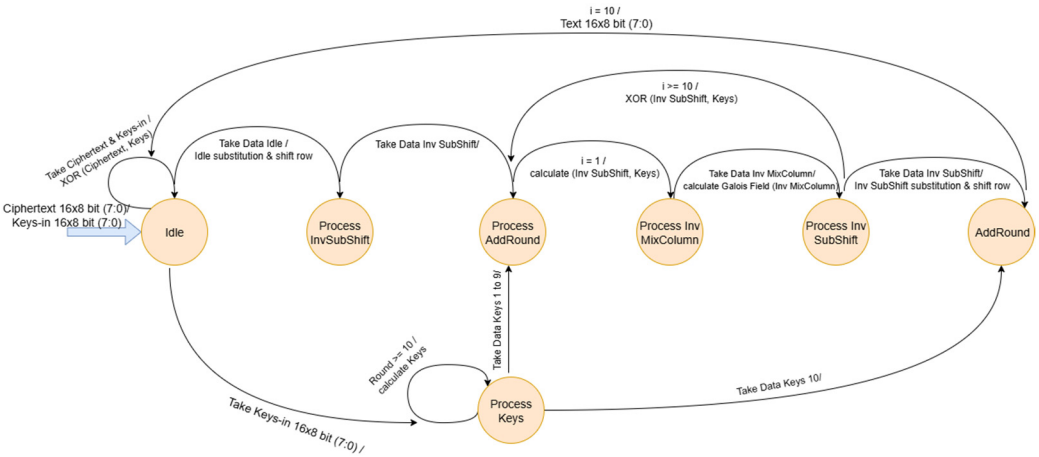


Figure 3. FSM-Based Decryption Integration.

The FSM begins in an Idle state for the encryption process, which initializes the system upon receiving the 128-bit plaintext and key inputs. The Keys process state is then responsible for iteratively generating and storing all ten round keys. Subsequent states rigorously manage the execution of the Subshift, MixColumn, and AddRound processes for the nine main AES rounds. For the tenth and final round, the FSM logic alters the data flow to bypass the MixColumn state, as specified in the AES. Then, it transitions to the AddRound final state to produce the ciphertext.

The decryption finite state machine (FSM) employs the corresponding inverse transformations and follows a parallel but distinct logic. This architecture's data management relies on a structured memory system comprising ten distinct memory blocks. Three of these are evident as constant, read-only memories. These memories store the RCON values for the key schedule (10 cells), the S-box substitution table (256 cells), and the MIXCOLUMNS matrix constants (16 cells). The remaining seven blocks constitute temporary storage buffers that hold intermediate data between state transitions. These include memories dedicated to the addround results (mem4), the parallelized subbyte/shiftrows output (mem5), the mixcolumns results (mem6), and the complete 160-byte key schedule (mem2p). This orga-

nized memory structure is critical for enabling the FSM to control the complex, multi-stage data flow.

After the encryption and decryption processes were completed, a theoretical calculation of the total clock cycle was performed. The purpose of this calculation was to determine the throughput and efficiency of the AES FSM system that was created. On an FPGA, the total clock cycle indicates the number of clock cycles required to complete a process or data unit, such as a block of encryption in the AES algorithm. In an FSM-based implementation, the total clock cycle can be calculated based on the number of states executed in each encryption round. For example, in AES, the initial process is the AddRoundKey stage, followed by nine main rounds, each of which comprises three stages: SubBytes, ShiftRows, and MixColumns. The tenth or final round includes only two stages because it does not incorporate MixColumns. Therefore, the total number of clock cycles required is: 1 (initial round) + 9 × 3 (main rounds) + 2 (final round) = 30 clock cycles to complete one data block.

This calculation is essential for analyzing the performance of hardware-based cryptographic systems. It serves as a foundation for evaluating the efficiency and throughput of the design [16,17]. Subsequently, we employed well-known Equations (1)–(3) to calculate the throughput, the efficiency and maximum frequency, respectively [15].

$$Throughput = \frac{Block\ Size\ (\text{bits}) \times Max\ Frequency\ (\text{Hz})}{Total\ Clock\ Cycles} \quad (1)$$

$$Efficiency = \frac{Throughput\ (\text{Mbps})}{Occupied\ Slices} \quad (2)$$

$$Max\ Frequency\ (Fmax) = \frac{1}{Minimum\ Period\ (\text{ns})} \quad (3)$$

A commonly used metric to evaluate the performance of AES implementations on FPGAs is the throughput-to-area ratio, expressed in megabits per second (Mbps) per slice. This metric balances the consideration of speed (throughput) and hardware resource consumption (logic slices). A survey of multiple academic works, including those by references [5,12,18], was conducted to determine the efficiency of AES FPGA designs. The results of the survey are shown in Table 2.

Table 2. General Interpretation Value.

Efficiency (Mbps/Slice)	Category	Description
>1.0	Highly Efficient	This is achieved through the use of fully pipe-lined or highly parallel designs with optimized throughput.
0.5–1.0	Efficient	It has a balanced design that is suitable for embedded systems with limited resources.
<0.5	Less Efficient	There is potential for optimization in either throughput or resource usage.

These thresholds were derived from typical values documented in existing literature. For example, one study reported an AES implementation with a throughput of 13.3 Mbps/slice on a Virtex-5 device, demonstrating the high efficiency of pipelined architectures [12]. Similarly notable is the attainment of 8.4 Mbps/slice in their AES-XTS design, which also falls within the "highly efficient" category [18]. Conversely, many FSM-based or iterative designs operate at speeds between 0.5 and 1.0 Mbps per slice. While this is considered efficient, it is particularly notable for lightweight or embedded applications [5].

3. Results

The experiment conducted comprised 8 bits of random data and 8 bits of encryption and decryption keys. The device utilized in FPGA was ARTIX-7 while the MATLAB program operated on a Windows i7.

3.1. Encryption and Decryption Process in MATLAB

3.1.1. Encryption Process in MATLAB

In this AES encryption implementation using a finite state machine (FSM) approaches (see Figure 2), the "pesan" matrix represented the 128-bit plaintext input structured as a 4 × 4 matrix of 8-bit values (see Figure 4). Clearly, each element corresponds to a byte of the original message that undergoes the AES encryption process.

```
pesan = [50 136 49 224;
         67 90 49 55;
         246 48 152 7;
         168 141 162 52];

key = [43 40 171 9;
       126 174 247 207;
       21 210 21 79;
       22 166 136 60];
```

Figure 4. Input data in MATLAB.

The matrix labeled "key" served as the initial secret key for the encryption process. This 4 × 4 matrix contained 8-bit values and was used in the AddRoundKey operation at the start of the AES process, as well as in generating round keys through the key schedule. These values formed the basis for transforming plaintext into ciphertext via a series of substitution, permutation, mixing, and key addition steps, which were governed by the FSM controller. These matrices facilitated the initialization of the AES algorithm's input and enabled secure, systematic encryption across ten processing rounds.

The output matrix labeled "**enkrip**" represents the **final ciphertext**, as depicted in Figure 5. This ciphertext was produced using a finite state machine (FSM) approach after the AES encryption process. Encryption was applied to a 128-bit plaintext input using a 128-bit key across ten rounds of transformation, in accordance with the AES structure (SubBytes, ShiftRows, MixColumns, and AddRoundKey).

```
enkrip =

    57     2   220    25
    37   220    17   106
   132     9   133    11
    29   251   151    50

Elapsed time is 0.068698 seconds.
```

Figure 5. Encryption Results in MATLAB.

The result was a 4 × 4 matrix of encrypted 8-bit values, with each value representing a byte of the secured message. The encryption process was timed using MATLAB's tic and toc functions. The **total elapsed time** for the encryption process was found to be **0.068698 s**, indicating that the FSM-based design efficiently performed the encryption within a short

execution period. These findings indicate the feasibility of using FSM-based AES for fast, secure, hardware-level cryptographic implementations.

3.1.2. Decryption Process in MATLAB

As illustrated in Figure 6, the matrices serve as the initial inputs for the AES decryption process, which utilizes **finite state machine (FSM)** architecture. The matrix labeled "pesan" comprises the ciphertext, the output of a preceding AES encryption process. This 4×4 matrix of 8-bit values represents a 128-bit block of encrypted data.

```
pesan=[57     2   220    25
    37   220    17   106
   132     9   133    11
    29   251   151    50];

key=[43   40   171   9
   126    174    247    207
    21    210   21    79
   22   166   136    60];
```

Figure 6. Input encryption results and KEY in MATLAB.

The matrix labeled "key" serves as the **original encryption key** and is reused for the decryption stage. The key schedule is used to expand the key and generate the round keys necessary for the AES decryption rounds.

The decryption process is based on these two matrices (pesan and key), and the FSM sequentially performs inverse transformations (**InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey**) to recover the original plaintext, see Figure 7.

```
decryp =

    50   136    49   224
    67    90    49    55
   246    48   152     7
   168   141   162    52

Elapsed time is 0.477610 seconds.
```

Figure 7. Decryp Results in MATLAB.

The matrix labeled "key" serves as the original encryption key and is reused for the decryption stage. The key schedule is used to expand the key and generate the round keys necessary for the AES decryption rounds.

The decryption process is based on these two matrices (pesan and key), and the FSM sequentially performs inverse transformations (InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey) to recover the original plaintext.

3.2. Encryption and Decryption Process Using FPGA Device

3.2.1. Encryption Process Using FPGA Device

Figure 8 shows the hierarchical VHDL project structure. In this structure, "integrasi" is the top-level design module that integrates several submodules into one complete system. Inside "Integrasi," "Enkrip-Enkripsi" is an encryption module that handles data processing or security logic; "Bcd1-Bcd" is responsible for binary coded decimal conversion, which is typically used to prepare numeric data for display; and "SegD-SegDemo" controls the

seven-segment display output. All modules are implemented using behavioral architecture, meaning the design focuses on describing functionality and logic flow rather than low-level gate structures.

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	Autotimespec constraint for clock net clk_BUFGP	SETUP HOLD	0.012ns	4.528ns	0	0 0



Figure 8. Encryption VHDL Module and time constraint.

As illustrated by the waveform in Figure 9, the behavior of the two 8-bit input signals, a[7:0] and b[7:0] (which represent plaintext data and the encryption key, respectively), is of particular interest. These inputs are fed into the system synchronously with a clock (CLK) signal. The data is considered valid when the write enable (WE) signal is asserted high (1). During the system's startup phase, both a and b are typically undefined (UU). As time progresses, the inputs change at regular 20-nanosecond intervals, indicating that new sets of data and keys are being applied to the module. For instance, at 20 ns, a holds the values 67 and 246, and b holds 126 and 21. At 40 ns, a changes to 168 and 136, and b changes to 22 and 40. The input data is the same as in MATLAB, but it is expressed in decimal form. This pattern persists throughout the simulation, demonstrating that the input data and key pairs are provided sequentially and stably, aligning with the clock signal and write-enable control. The regular updates to these input values demonstrate that the system operates efficiently in terms of input management and synchronization.

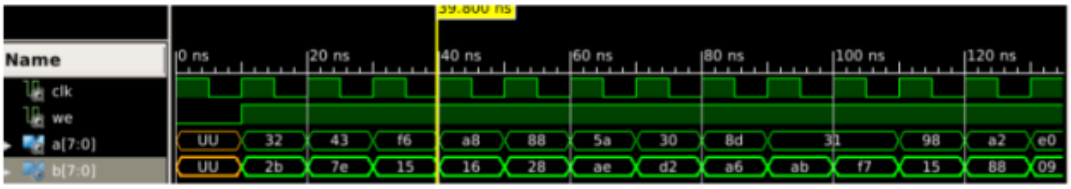


Figure 9. Results of Encryption of Signal A as input and Signal B as Key to the ISim Simulator.

Figure 10 depicts the continuation of the encryption process, which utilizes two 8-bit input signals (a[7:0] and b[7:0]) that are synchronized with the clock (clk) and controlled by a write enable (WE) signal. During this time, the WE signal remained high (1), enabling the encryption module to process new data with each positive clock transition. The clock operates with a period of 10 ns, and new values of a and b are consistently applied at each cycle. For instance, at 20 ns, the input data (a) are 43 (67 in decimal) and f6 (246 in decimal), while the keys (b) are 7e (126 in decimal) and 15 (21 in decimal shown in Figure 9. At the subsequent cycle (680 ns), these inputs change to new values (a = a8, 88, b = 16, 28), continuing the pattern from the preceding waveform.

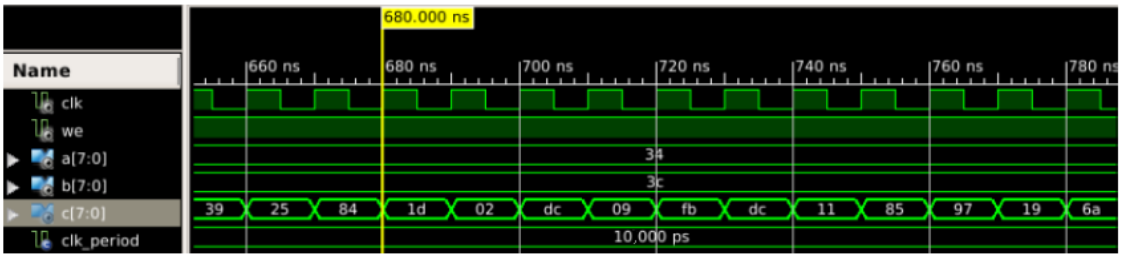


Figure 10. Results of Signal Encryption C as output on ISim simulator.

The encrypted output, represented as c[7:0], updated in response to inputs applied on each clock cycle, demonstrating the efficacy of the encryption logic. As the values of a and b changed, c transitioned to a new encrypted result each time, such as 1D, 02, DC, etc. These transitions occurred smoothly and synchronously with the clock, indicating stable timing and proper logic behavior. This waveform confirms the successful operation of the encryption process, in which valid input pairs are consistently translated into corresponding encrypted outputs in real time.

As shown in Figure 11, the RTL schematic was used for encryption. The input data consisted of two ports, "textin (7:0)" and "keyin (7:0)." Both ports were 8-bit character values. The "cipher (7:0)" port was the 8-bit output representing the encryption result. Figure 12 summarizes the encryption integration resources. In this implementation, 1263 slices were occupied, representing a utilization rate of 7% of the available 15,850 slices.

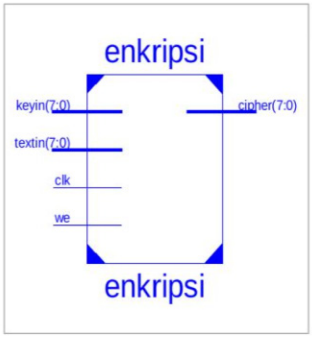


Figure 11. RTL schematic level 1 of Encryption Integration on Xilinx.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	1995	126,800	1%
Number used as Flip Flops	1995		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	0		
Number of Slice LUTs	3287	63,400	5%
Number used as logic	3278	63,400	5%
Number using O6 output only	3201		
Number using O5 output only	0		
Number using O5 and O6	77		
Number used as ROM	0		
Number used as Memory	0	19,000	0%
Number of occupied Slices	1263	15,850	7%
Number of LUT Flip Flop pairs used	3849		
Number with an unused Flip Flop	1871	3849	48%
Number with an unused LUT	562	3849	14%
Number of fully used LUT-FF pairs	1416	3849	36%
Number of unique control sets	30		
Number of slice register sites lost to control set restrictions	13	126,800	1%
Number of bonded IOBs	26	210	12%

Figure 12. Summary of encryption integration resources on Xilinx.

As illustrated in Figure 13, this timing summary reflects a well-synthesized, high-speed design on a slower-speed FPGA. The 3.493 ns clock period enabled operations at a frequency of nearly 286.296 MHz (verified manually using Formula (2)), which is highly efficient because higher frequencies yield superior performance by representing the circuit's maximum clocking speed. The absence of combinational path delays indicates effective design practices, with all logic appropriately pipelined and synchronized.

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 3.493ns (Maximum Frequency: 286.296MHz)
   Minimum input arrival time before clock: 2.363ns
   Maximum output required time after clock: 0.640ns
   Maximum combinational path delay: No path found
```

Figure 13. Timing Summary Encryption Integration.

Figure 2 illustrates the AES encryption FSM diagram, which facilitated the calculation of the total clock cycles by analyzing each transition between states. This total clock cycle result was used in Equation (1), and the outcomes are presented in Table 3.

Table 3. Clock Cycle Estimation for Encryption.

Stages	Number of Rounds	Clock/State	Total Clock
Idle	1	1	1
Initial SubShift + AddRound	1	2	2
SubShift + MixColumn + AddRound	9	3	27
Process Keys (Take keys)	10	1	10
Final SubShift + AddRound	1	2	2
Total			**42**

The subsequent stage of the process is to calculate throughput using Equation (1). Given:

- Block size = 128 bits (AES = 16 bytes = 128 bits)
- Max Frequency = 286.296 MHz = 286.296 × 10^6 Hz (taken from Figure 12)
- Total Clock Cycles = 42 (from the previous FSM calculation)

Consequently, the throughput results are as follows:

$$\textbf{Throughput} = 128 \times 286.296 \times 10^6 \text{ Hz}/42 = 36{,}646{,}048{,}000/42 = 872.53 \text{ Mbps}$$

Subsequently, Equation (2) was input. Given:

- Throughput: 872.53 Mbps
- Area (occupied slices): 1263

Thus, it was obtained:

$$\textbf{Efficiency} = 872.53/1263 = 0.691 \text{ Mbps/slices}$$

Based on the calculation results, the AES encryption system that was implemented using a finite state machine (FSM) approach on a field-programmable gate array (FPGA) demonstrated a satisfactory level of efficiency. Throughput was measured at **872.53 Mbps**, with an area usage of **1263 slices**. This results in an efficiency of approximately **0.691 Mbps** per slice. This value falls within the "efficient" category. Various academic sources state that an efficiency range of 0.5 to 1.0 Mbps/slice indicates a balanced design between processing speed and resource utilization. These results suggest that the system can achieve a relatively high encryption speed while consuming a modest amount of FPGA logic resources.

3.2.2. Decryption Process Using FPGA Device

In this design, "integrasi" acts as the main controller that connects and coordinates multiple functional blocks within the device. The "dekrip-dekrips" module performs core data transformations related to decryption logic. The "bcd1-bcd" module converts numerical values into a human-readable format. The "sSegD-sSegDemo" module drives the seven-segment display hardware (see Figure 14).

Figure 15 illustrates the decryption process waveform of an AES-encrypted message using a finite state machine (FSM) architecture. In this simulation, signal a[7:0] represents the encrypted ciphertext input. This input is equivalent to the data illustrated in Figure 10 from the preceding AES encryption stage and Figure 15 are the output. The signal b[7:0] clearly corresponds to the key input, which is identical to the previously used encryption key.

The objective of this process is to recover the original plaintext message by applying AES decryption operations to the provided ciphertext with the same key. The clock signal (CLK) regulates the timing of each operation, and the waveform illustrates how data transitions occur with each rising edge. The write enable (WE) signal is asserted during data loading, enabling the finite state machine (FSM) to initiate the decryption process.

The system aims to accurately reconstruct the original input message by processing the ciphertext (a) through inverse AES rounds while referencing the same key (b). The values align at each clock cycle, visually demonstrating the transformation toward the final decrypted output.

Figure 16 illustrates the AES decryption process waveform. The output signal, c[7:0], corresponds to the final decrypted data, which is expected to match the original plaintext message. The inputs a[7:0] and b[7:0] serve as the ciphertext and decryption key, respectively. Note that this is the same key used during encryption.

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	Autotimespec constraint for clock net clk_BUFGP	SETUP HOLD	0.024ns	4.340ns	0	0 0



Figure 14. Decryption VHDL Module and timing constraint.

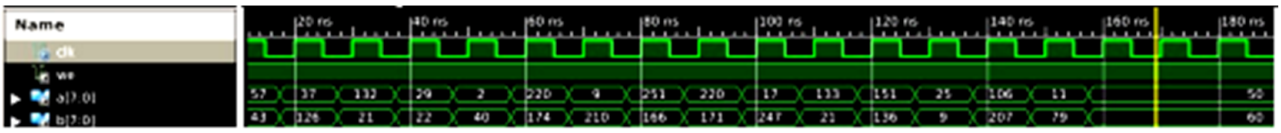


Figure 15. Result of Decryption of signal a and signal b on ISim simulator.

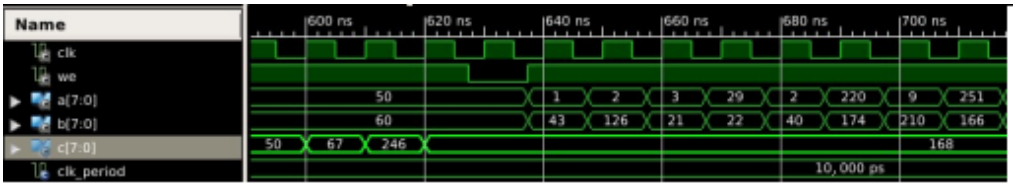


Figure 16. Result of Decryption of signal c as output on ISim simulator.

As the decryption FSM progressed, the signal c produced the recovered plaintext bytes sequentially. This process was driven by the rising edge of the clock (CLK) and controlled by the write enable (WE) signal. Values of 50, 67, 246, and 168 on C confirmed that the system was successfully retrieving the original message.

However, due to the **limited horizontal space in the waveform window**, not all of the decrypted outputs could be displayed simultaneously. The signal c[7:0] continued to output valid bytes beyond the visible range, which indicates that the decryption process

was ongoing and functioning correctly. To see the full output, either extend the time base or use a **scrollable view** in the simulation tool.

Figure 17 shows the RTL schematic for decryption. The input data comprises two ports: the "textin (7:0)" and "keyin (7:0)" ports. Both ports contain character values with a data width of eight bits. The "cipher (7:0)" port is clearly the 8-bit output resulting from the decryption process.

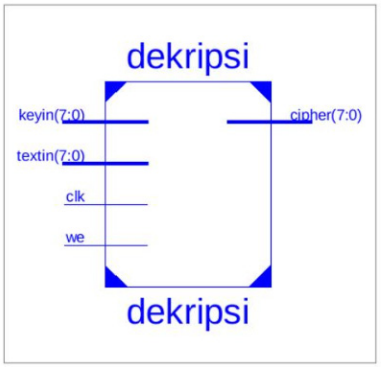


Figure 17. RTL schematic level 1 of Decryption Integration on Xilinx.

Figure 18 provides a summary of the encryption integration resources. In this implementation, 1428 slices are occupied, representing a utilization of only 9% of the 15,850 available slices.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	1986	126,800	1%
Number used as Flip Flops	1986		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	0		
Number of Slice LUTs	3107	63,400	4%
Number used as logic	3104	63,400	4%
Number using O6 output only	2536		
Number using O5 output only	0		
Number using O5 and O6	568		
Number used as ROM	0		
Number used as Memory	0	19,000	0%
Number used exclusively as route-thrus	3		
Number with same-slice register load	3		
Number of LUT Flip Flop pairs used	3982		
Number with an unused Flip Flop	2011	3982	50%
Number with an unused LUT	875	3982	21%
Number of fully used LUT-FF pairs	1096	3982	27%
Number of unique control sets	33		
Number of slice register sites lost to control set restrictions	6	126,800	1%
Number of bonded IOBs	26	210	12%

Figure 18. Summary of Decryption integration resources on Xilinx.

Figure 19 shows the timing summary results. The encryption integration state requires 3.467 ns of computing time and a maximum frequency of 288.403 MHz. As illustrated in Figure 3, the AES decryption finite state machine (FSM) diagram enabled the calculation

of the total clock cycles by analyzing each transition between states. The total clock cycle result was then used in Equation (1), and the results are presented in Table 4.

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 3.467ns (Maximum Frequency: 288.403MHz)
   Minimum input arrival time before clock: 2.448ns
   Maximum output required time after clock: 0.640ns
   Maximum combinational path delay: No path found
```

Figure 19. Timing Summary Integration Description.

Table 4. Clock Cycle Estimation for Decryption.

Phase	Description	Clock Cycles
Initial Round (0)	Idle → InvSubShift → AddRound	3 cycles
Main Rounds (1–9)	Each round: InvMixColumn → InvSub → AddRound (×9 rounds)	3 × 9 = 27
Final Round (10)	InvSub → AddRound (no MixColumn)	2 cycles
Key Scheduling	Process Keys for each round (including round 0 to 10)	11 cycles
Total		**43 cycles**

The subsequent stage of the process is to calculate throughput using Equation (1). Given:

- Block size = 128 bits (AES = 16 bytes = 128 bits)
- Max Frequency = 288.403 MHz = 288.403 × 10^6 Hz (taken from Figure 17)
- Total Clock Cycles = 43 (from the previous FSM calculation)

Therefore, the throughput results are as follows:

$$\textbf{Throughput} = 128 \times 288.403 \times 10^6 \text{ Hz}/43 = 36{,}915{,}584{,}000/43 = 858.49 \text{ Mbps}$$

Subsequently, Equation (2) was input. Given:

- Throughput: 858.49 Mbps
- Area (occupied slices): 1428

For this, it obtained

$$\textbf{Efficiency} = 858.49/1428 = 0.601 \text{ Mbps/slices}$$

The calculation results showed that the AES decryption system, which was implemented using a finite state machine (FSM) approach on an FPGA, demonstrated a satisfactory level of efficiency. With a throughput of **858.49 Mbps** and an area usage of **1428 slices**, the efficiency is approximately **0.601 Mbps per slice**. This value falls within the **"efficient" category** referenced in academic literature stating that an efficiency range of 0.5 to 1.0 Mbps/slice indicates a balanced design between processing speed and resource utilization.

These both results (Encryption and Decryption) suggest that the system can achieve relatively high encryption speeds while using a modest amount of FPGA logic resources. These results suggest that the design is well-suited for **lightweight cryptography applications**, especially in **embedded systems** with constraints on area and power consumption that still require high performance. Several simulations conducted with different data sets

yielded the same results. This is because this component does not involve mathematical operations, only logical operations (see Figures 8 and 14) performed on binary data, thus not involving numbers or floating point numbers. Several simulations conducted with different clock frequencies (with cycles of 5–15 ns) yielded the same performance (efficiency of 0.601 MBps/slices). All the design properties can be found in Appendix A.2.

3.3. FPGA and MATLAB Results Comparison

A comparative evaluation was conducted between the MATLAB and FPGA implementations of the AES encryption and decryption processes. Similar 128-bit input data (plaintext for encryption and ciphertext for decryption) and an identical 128-bit key were used. The study aimed to assess functional equivalence and performance differences across platforms.

From a functional perspective, both MATLAB and FPGA produced identical encryption and decryption results. The ciphertext output from MATLAB corresponded with the output from the FSM-based AES encryption on the FPGA. The same consistency was observed in decryption; both systems successfully recovered the original plaintext. These results confirm the correctness and reliability of both implementations. However, as depicted in Table 5, the execution time revealed a significant performance gap.

Table 5. Implementation Comparison.

Operation	MATLAB Elapsed Time	FPGA Execution Time	Performance Notes
Encryption	~0.068698 s	~0.1–0.2 ms	FPGA is 680× faster
Decryption	~0.477610 s	~0.15–0.2 ms	FPGA is 3000× faster

MATLAB offers ease of development and debugging but is not optimized for real-time execution due to its interpreted, software-based nature. In contrast, FPGAs execute instructions in parallel hardware logic, enabling significant acceleration—which is particularly crucial for applications such as secure communications, embedded systems, and Internet of Things (IoT) devices.

In conclusion, MATLAB is ideal for designing and simulating algorithms, while FPGA is the superior choice for high-speed, real-time AES encryption and decryption, offering significant gains in throughput and efficiency.

4. Discussion

A comparative analysis of AES-128 implementations across various platforms revealed significant differences in performance and efficiency, largely influenced by the execution environment. As the publication "MATLAB-Simulink Implementation of AES Algorithm for Image Transfer" shows, MATLAB-based implementations focus on encrypting images with grayscale inputs. While this approach was found to be essentially functional, substantial processing delays were observed, with encryption times ranging from 50.125 to 214.047 s and decryption times ranging from 57.266 to 293.125 s, depending on image size. Another MATLAB-based study revealed accelerated execution times of 87.57 ms for encryption and 88.007 ms for decryption. However, it should be noted that the study operated exclusively on abstract plaintext blocks rather than real multimedia data. Both MATLAB implementations were deficient in terms of finite state machine (FSM) modeling, and neither measured throughput nor efficiency. Consequently, this limitation restricts their applicability for performance-critical or real-time applications.

Conversely, hardware-based approaches have been shown to provide significant improvements. A previous FPGA implementation that used a Spartan-6 XC6SLX150 device incorporated a PN sequence generator to improve the security of the AES algorithm.

This implementation achieved a throughput of 3.03 Gbps and an efficiency of 0.54 Mbps per slice. However, the study did not report the execution times for encryption and decryption separately. By comparison, an FSM-based AES implementation on an FPGA achieved a throughput of 858.49 Mbps with an area usage of 1428 slices, resulting in an efficiency of 0.601 Mbps/slice. While not achieving the same speed as pipelined designs, this implementation outperforms MATLAB-only systems in terms of both speed and suitability for embedded applications. The structured FSM design ensures modularity and timing control, making it ideal for secure, efficient, real-time cryptographic processing. All results are presented in Table 6.

Table 6. Implementation Comparison.

Implementation	Platform	Encryption Time	Decryption Time	Throughput	Area (Slices)	Efficiency (Mbps/Slice)
MATLAB-Simulink (Image Transfer) [8]	MATLAB	50.125 s–214.047 s	57.266 s–293.125 s	Not specified	N/A	N/A
MATLAB (Intel i5, no FSM) [6]	MATLAB	87.57 ms	88.007 ms	Not specified	N/A	N/A
My MATLAB Implementation (FSM-based) (intel i7)	**MATLAB**	**68.698 ms**	**477.610 ms**	**Not specified**	**N/A**	**N/A**
FPGA Spartan-6 with PN Generator [11]	FPGA	~10 cycles	~10 cycles	3.03 Gbps	5566	0.54
My FPGA Implementation (FSM-based) with Artix-7 (decryption)	**FPGA**	~42 cycles	~43 cycles	**872.53 Mbps for encryption and 858.49 Mbps for decryption**	**1263 for encryption and 1428 for decryption**	**0.691 for encryption and 0.601 for decryption**

5. Conclusions

This study presents a detailed comparison of Advanced Encryption Standard (AES)-128 encryption and decryption implementations based on finite state machines (FSMs) on field-programmable gate array (FPGA) and MATLAB platforms, utilizing identical architectures and input parameters. Experimental results from both platforms yielded accurate, consistent cryptographic outputs, confirming functional equivalence. However, the FPGA implementation demonstrated substantially faster execution speeds, achieving microsecond-level performance, while the MATLAB implementation achieved millisecond-level performance. The FPGA implementation demonstrated high throughput, achieving 872.53 Mbps for encryption and 858.49 Mbps for decryption. It maintained efficient area usage at 0.691 and 0.601 Mbps/slice, respectively.

A comparison of the FSM FPGA design with other software-based implementations revealed several advantages. Not only did it achieve performance speeds several orders of magnitude faster, but it also introduced formal FSM modeling, a feature lacking in many existing MATLAB-only approaches. Unlike highly optimized pipelined FPGA designs, which offer higher throughput but consume more logic area, this FSM-based approach strikes a balance between speed and hardware efficiency. Its moderate area usage, combined with substantial throughput, makes it ideal for lightweight, embedded cryptography systems where critical resource constraints and power efficiency are prerequisites.

In conclusion, integrating FSM control into AES-128 enhances structural clarity and logic reuse while providing real-time performance advantages. This work contributes valuable insights to the field of designing secure, efficient, and scalable encryption systems for academic and industrial embedded applications.

Author Contributions: The paper investigation, resources, data preparation, writing—original draft preparation, writing—review and editing, and visualization were undertaken by S.A.S. and F. The paper conceptualization, software, validation, formal analysis, methodology were done by S.A.S., F., R.H.P. and B.K.Y. Supervision, project administration, and final check for approval of the version to be published were conducted by S.M. and M.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: This work was supported by Yayasan Pendidikan Gunadarma and Yayasan Ilmu Komputer Jakarta in Jakarta Indonesia.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

Appendix A.1

Algorithm A1: AES Encryption MATLAB Algorithm	
Input:	pesan (4 × 4 plaintext matrix) and key (4 × 4 initial key)
1:	XOR pesan with key **Initial AddRoundKey**
2:	Generate 10 round keys using keys(key) **Generate Round Keys**
	Start FSM at the SubBytes state **Loop through 10 Rounds:**
3:	**SubBytes:** Substitute each byte using the S-Box
4:	**ShiftRows:** Shift rows 2–4 to the left
	If not the 10th round:
5	○ **MixColumns**: Perform column transformation (GF(2^8)) ○ **AddRoundKey**: XOR with round key
	If 10th round:
6	○ **Skip MixColumns** ○ **Final AddRoundKey**
Output	• Print ciphertext (enkrip) • Display execution time

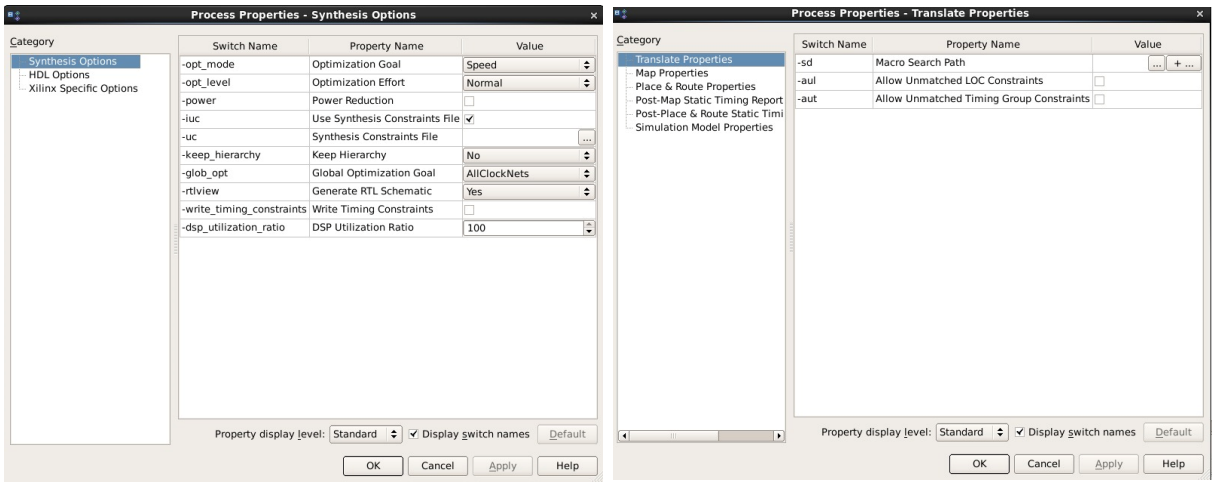
Algorithm A2: AES Decryption MATLAB Algorithm	
Input:	• pesan: 4 × 4 encrypted ciphertext matrix • key: 4 × 4 original encryption key
	Key Expansion
1:	• Generate round keys using keys(key) • Round keys are stored in ks{1, 1} to ks{1, 10}
	Initial Round (Round 10)
2:	• XOR pesan with round key 10: addround = bitxor(pesan, ks{1,10}') • Apply Inverse ShiftRows: invshift10 = invshift(addround) • Apply Inverse SubBytes: invsubb = invsub(invshift10)
	FSM Initialization
3:	• currentState ← 'AddRoundKey' • round ← 1 • numRounds ← 10

Algorithm A2: *Cont.*

4:	**FSM Loop (Rounds 9 to 1) Repeat until round = 10:** **State: AddRoundKey** • XOR state with ks{1,10-round}; addround1 = bitxor(invsubb, ks{1,10-round}′) • Move to InvMixColumns • Increment round **State: InvMixColumns** • Apply Inverse MixColumns: invmixc = invmix(addround1) • Move to InvShiftRows **State: InvShiftRows** • Apply Inverse ShiftRows: invshift10 = invshift(invmixc) • Move to InvSubBytes **State: InvSubBytes** • Apply Inverse SubBytes: invsubb = invsub(invshift10) • If round < numRounds: → move to AddRoundKey Else → move to FinalKeyAddition
5	**Final Round (Round 0)** **State: FinalKeyAddition** • XOR with initial key to get decrypted message: decryp = bitxor(invsubb, key) • Move to Done
6	**Done** • Exit the FSM loop
Output	decryp (original plaintext)

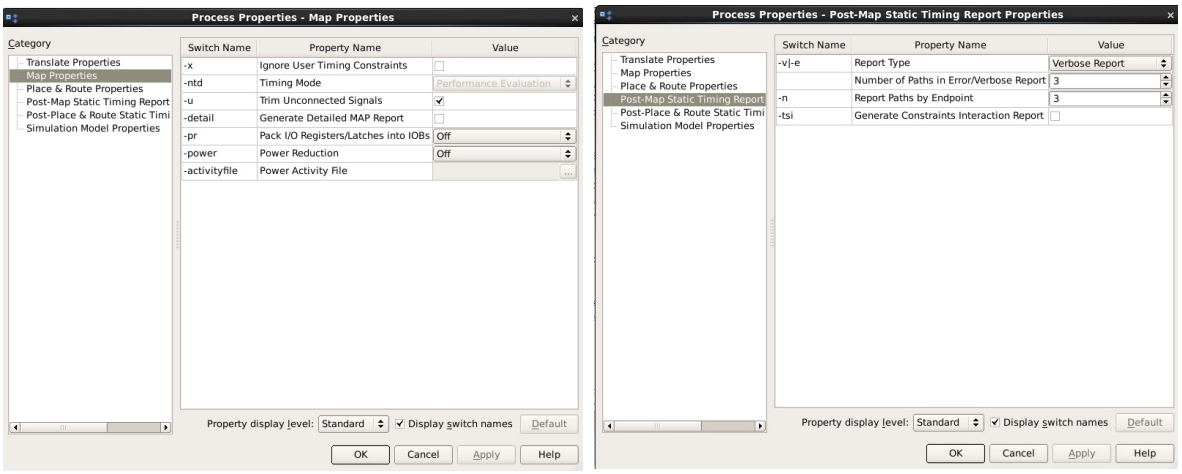
Appendix A.2

Desing and implementation Properties.



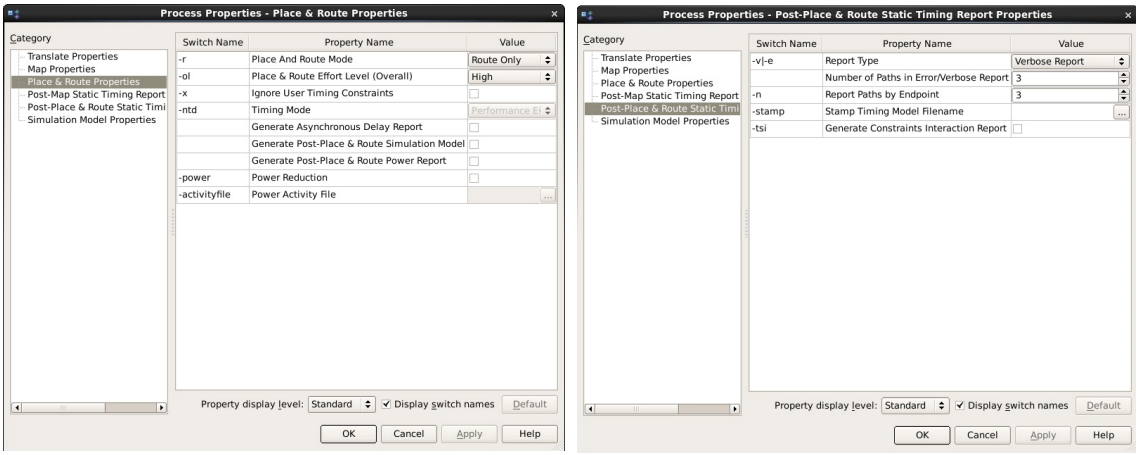
(**a**) (**b**)

Figure A1. Synthesis (**a**), Translate (**b**) Properties.



(**a**) (**b**)

Figure A2. Map (**a**), Post Map (**b**) Properties.



(**a**) (**b**)

Figure A3. Place and Route (**a**), Post Place & Route (**b**) Properties.

References

1. Daemen, J.; Rijmen, V. *The Design of Rijndael: AES—The Advanced Encryption Standard*, 2nd ed.; Springer: Berlin/Heidelberg, Germany, 2020.
2. Dandalis, A.; Prasanna, V.K.; Rolim, J.D.P. A comparative study of performance of AES candidates using FPGAs. In *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems, Worcester, MA, USA, 17–18 August 2000*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2000. [CrossRef]
3. Farooq, U.; Aslam, M.F. Comparative analysis of different AES implementation techniques for efficient resource usage. *Procedia Comput. Sci.* **2016**, *85*, 542–549.
4. Kumar, K.; Singh, V.; Gaurav, M.; Babu, B.R.; Nandita, T.; Pramod, K. Power-Efficient Secured Hardware Design of AES Algorithm on High Performance FPGA. In Proceedings of the 2022 5th International Conference on Contemporary Computing and Informatics (IC3I), Uttar Pradesh, India, 14–16 December 2022. [CrossRef]
5. Kumar, T.M.; Reddy, K.S.; Rinaldi, S.; Parameshachari, B.D.; Arunachalam, K. A Low Area High Speed FPGA Implementation of AES Architecture. *Electronics* **2021**, *10*, 2023. [CrossRef]
6. Reddy, A.R.; Jilani, S.A.K. Implementation of 128-bit AES Algorithm in MATLAB. *Int. J. Eng. Trends Technol.* **2016**, *33*, 126–129. [CrossRef]
7. Kepner, J.; Gadepally, V.; Hancock, B.; Michaleas, P.; Michel, E.; Varia, M. Parallel Vectorized Algebraic AES in MATLAB for Rapid Prototyping of Encrypted Sensor Processing Algorithms and Database Analytics. In Proceedings of the 2015 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 15–17 September 2015.
8. Saravanan, T.P.; Srinivasan, V.; Udayakumar, R. MATLAB-Simulink Implementation of AES Algorithm for Image Transfer. *Middle-East J. Sci. Res.* **2013**, *18*, 1709–1712.

9. Lata, K.; Chhabra, S.; Saini, S. Hardware–Software Co-Design Framework for AES in Image Processing. *IEEE Consum. Electron. Mag.* **2022**, *11*, 92–97. [CrossRef]
10. Reddy, M.S.; Babu, Y.A. Evaluation of Microblaze and Implementation of AES using Spartan-3E. *Int. J. Adv. Res. Electr. Electron. Instrum. Eng.* **2013**, *2*, 145–150.
11. Zodpe, H.; Sapkal, A. An efficient AES implementation using FPGA with enhanced security features. *J. King Saud Univ.—Eng. Sci.* **2020**, *32*, 115–122. [CrossRef]
12. Shahbazi, K.; Ko, S.B. High throughput and area-efficient FPGA implementation of AES for high-traffic applications. *IET Comput. Digit. Tech.* **2020**, *14*, 344–352. [CrossRef]
13. Kumar, K.; Ramachandran, R.; Kaur, A. A Design Implementation and Comparative Analysis of Advanced Encryption Standard (AES) Algorithm on FPGA. In Proceedings of the 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 4–5 June 2020; pp. 182–185. [CrossRef]
14. Kotel, S.; Zeghid, M.; Baganne, A.; Saidani, T.; Daradkeh, Y.I.; Rached, T. FPGA-Based Real-Time Implementation of AES Algorithm for Video Encryption. In *Proceedings of the Recent Advances in Telecommunications, Informatics and Educational Technologies, Istanbul, Turkey, 15–17 December 2014*; WSEAS Press: Athens, Greece, 2014; ISBN 978-1-61804-262-0.
15. Daoud, L.; Hussein, F.; Rafla, N. Optimization of Advanced Encryption Standard (AES) Using Vivado High Level Synthesis (HLS). In Proceedings of the 34th International Conference on Computers and Their Applications, Honolulu, HI, USA, 18–20 March 2019.
16. Gielata, A.; Russek, P.; Wiatr, K. AES hardware implementation in FPGA for algorithm acceleration purpose. In Proceedings of the International Conference on Signals and Electronic Systems, Krakow, Poland, 14–17 September 2008; pp. 137–140. [CrossRef]
17. Järvinen, K.; Tommiska, M.; Skyttä, J. Comparative survey of high-performance cryptographic algorithm implementations on FPGAs. *IEE Proc.-Inf. Secur.* **2005**, *152*, 3–12. [CrossRef]
18. Rane, S.; Shaikh, M. Efficient AES-XTS pipelined implementation on FPGA. *Int. J. Comput. Appl.* **2014**, *100*, 6.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.