Implementation of AES-128 Encryption in SystemVerilog

Leandro Borzyk

March 24, 2025

Abstract

This document presents an in-depth overview of a project that implements the AES-128 encryption algorithm in SystemVerilog. The design is modular and breaks the encryption process into distinct functional components, each described in detail.

1 Introduction

AES-128 is a symmetric encryption algorithm that transforms a 128-bit plaintext into a 128-bit ciphertext using a 128-bit key through a series of well-defined transformation steps. The encryption process uses 10 rounds; it begins with an initial AddRoundKey operation, where the plaintext is combined with the key using a bitwise XOR. This is followed by nine rounds of processing; each round consists of four steps: SubBytes, ShiftRows, MixColumns, and AddRoundKey, while in the final (tenth) round the MixColumns step is omitted. This order of events is shown in 1.

The design presented in this document incorporates both combinational and clocked logic. Combinational logic is heavily used in arithmetic operations such as the bitwise XOR in the `AES_AddRoundKey` module and the finite field multiplications in the `AES_MixColumns` module. In contrast, clocked logic is employed in the `AES_Core` module, for instance, which uses a finite state machine (FSM) implemented with clocked logic to control the sequencing of operations across the encryption rounds. This FSM, which transitions through states such as IDLE, INIT, ROUND, and FINISH, synchronizes the activation of modules like `AES_AddRoundKey`, `AES_SubBytes`, `AES_ShiftRows`, and `AES_MixColumns`.

Testing and verification of these modules are conducted using dedicated SystemVerilog testbenches. Each testbench provides precomputed input vectors and expected outputs extracted from an external AES step-by-step reference available at `https://legacy.cryptool.org/en/cto/aes-step-by-step` to ensure correct functionality. For example, the testbench for `AES_KeyExpansion` validates the generation of round keys, while those for `AES_MixColumns` and `AES_ShiftRows` verify that the arithmetic operations and data reordering are performed correctly. Together, these design and verification techniques—including the use of ROM for fixed lookup tables, combinational arithmetic blocks, and clocked FSMs for control logic—ensure that the AES-128 implementation adheres strictly to the encryption standard.

Having established an overview of the AES-128 encryption process and its core transformations, we now delve into the detailed implementation of each functional module. The following section presents the modular structure of the design, highlighting the role and operation of each component in the encryption pipeline.

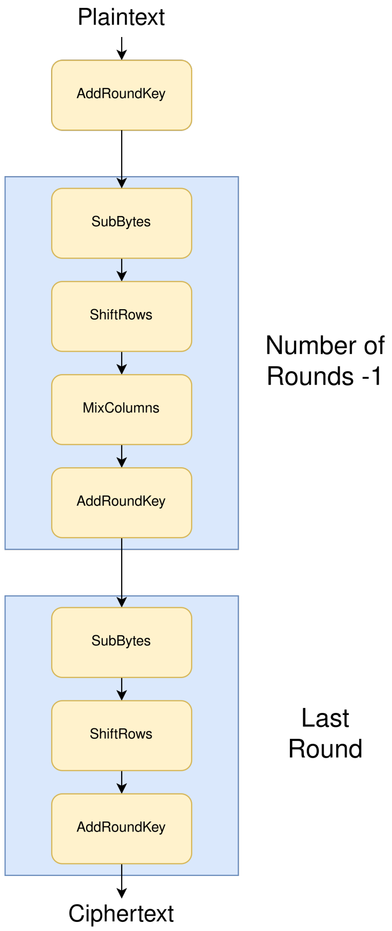

Figure 1: Overview of the AES-128 encryption process

2 Module Descriptions and Operation

The AES-128 encryption design is implemented through a collection of SystemVerilog modules, each responsible for a specific transformation in the encryption process. The design emphasizes modularity and reusability, with clear interfaces and well-defined data flows. Each of the modules—`AES_AddRoundKey`, `AES_KeyExpansion`, `AES_MixColumns`, `AES_ShiftRows`, `AES_SBOX`, `AES_SubBytes`, and `AES_Core`—is designed to focus on a specific function within the encryption process. Together, they form a cohesive system that leverages memory for lookup tables, arithmetic units for critical operations, and structured control logic to meet the AES-128 standard.

2.1 AES_AddRoundKey Module

The `AES_AddRoundKey` module is a critical component of the AES encryption algorithm, performing the integration of a 128-bit round key with a 128-bit state using a bitwise XOR operation. This operation is essential for the security of the AES algorithm, as it ensures that the round key is mixed into the state, creating the desired level of diffusion and making it more difficult to reverse the encryption process. The module is designed by instantiating a parameterized Arithmetic Logic Unit (ALU) that is specifically configured for XOR operations. The ALU is a versatile module capable of supporting multiple logical and arithmetic operations, but in this case, it is configured to perform only the

XOR operation, which is the key operation for the AddRoundKey transformation. The ALU takes two 128-bit operands, which are the current state and the round key, and applies the XOR operation between corresponding bits of these operands. The result is stored in the output state. The ALU is parameterized with a width of 128 bits, ensuring that it can handle the full 128-bit input and output required by AES. The operation is controlled through the `alu_op` signal, which is set to 2'b10 to specify the XOR operation. Additionally, the ALU's interface includes a `zero` output, which in this context is not used but is included to maintain the ALU's full functionality. This modular approach, with the clear and simple interface of the `AES_AddRoundKey` module, facilitates its seamless integration into the broader AES encryption process, ensuring that the round key is effectively combined with the state at each round of encryption.

2.2 AES_KeyExpansion Module

The `AES_KeyExpansion` module is responsible for generating the eleven round keys required for AES encryption from a given 128-bit cipher key. The process starts by dividing the 128-bit key into four 32-bit words. These words are then processed iteratively in a loop to generate the full set of round keys. Initially, the first four words are directly assigned from the input cipher key, forming the seed for the expansion. For each subsequent word, the module performs a series of operations: it copies the previous word into a temporary variable, applies a rotation (via the `RotWord` function) if the index is a multiple of four, and substitutes the bytes using the `SubWord` function, which uses an S-box stored in a ROM initialized from an external file. Additionally, a round constant is added to every fourth word via the `Rcon` function. The round constant values are predefined in a constant table. Each new word is then derived by XORing the rotated and substituted word with the word four places before it, ensuring that the expansion is dependent on previous values and promoting diffusion across all rounds. After expanding all 44 words, the 11 round keys are constructed by concatenating four words into each 128-bit key and stored in the `round_keys` array. These keys are then used in the AES encryption rounds to provide cryptographic strength and security. This method ensures that each round of AES encryption uses a unique key derived from the original 128-bit cipher key, contributing to the algorithm's resistance to attacks.

2.3 AES_MixColumns Module

The `AES_MixColumns` module is an essential part of the AES encryption algorithm, designed to achieve diffusion by processing the four 32-bit columns that make up the 128-bit state. This operation ensures that the output is a highly mixed version of the input, with each byte in the state being influenced by several other bytes. The module operates on the state by transforming each of its columns using arithmetic operations defined within the finite field GF(2^8), a crucial concept for AES. In GF(2^8), each element (or byte) is represented as a polynomial of degree 7, and all operations are carried out modulo an irreducible polynomial, which in the case of AES is given by $x^8 + x^4 + x^3 + x + 1$. This field is used for all arithmetic, including multiplication, addition, and inversion, making the operations more efficient and secure.

To implement the finite field multiplication, the module uses a helper function, `xtime`, which multiplies a byte by 2 in GF(2^8). The `xtime` function shifts the bits of a byte left by one position (essentially multiplying by 2) and applies an XOR operation with the constant $0x1B$ if the most significant bit is set, which handles the modular reduction by the irreducible polynomial. This step ensures that the result of the multiplication is within the field, taking into account the finite field arithmetic.

The core transformation of the module occurs in the `mix_column` function, which applies the MixColumns operation to each 32-bit column of the state. Each 32-bit column is further split into four 8-bit bytes, which are processed individually. The function computes a new byte for each position in the column by performing a series of multiplications and XOR operations on the bytes. Specifically, for each byte in the column, the function multiplies other bytes by 2 (using `xtime`) and combines the results with XOR operations. This transformation mixes the bytes in a non-linear way, contributing to the diffusion property of the AES encryption. For example, the new byte at position b_0 is derived from the first byte multiplied by 2, XORed with the second byte multiplied by 2 and XORed with the third and fourth bytes, and so on for the other positions.

Once the new column values are computed, the module reassembles the transformed columns into a 128-bit output state, which is still in column-major order, and passes it on for the next stage of the AES algorithm. This modular approach, where the arithmetic operations are encapsulated in helper functions like `xtime`, helps simplify the overall implementation of the MixColumns transformation while maintaining efficient, clear, and effective processing.

2.4 AES_ShiftRows Module

The `AES_ShiftRows` module is an important part of the AES encryption algorithm, tasked with reordering the state by cyclically shifting its rows. This operation contributes to the overall diffusion of the state, making it harder to reverse-engineer the original data, and plays a crucial role in the security of the AES encryption process. The input state, provided in column-major order, is first decomposed into individual bytes. This process is based on the column-major ordering, where the state is organized into four columns, with each column containing four 8-bit bytes. Specifically, the first column (Column0) consists of the bytes s_{00}, s_{10}, s_{20}, and s_{30}, the second column (Column1) contains s_{01}, s_{11}, s_{21}, and s_{31}, and so on for the third and fourth columns. After the state is broken down into individual bytes, it is regrouped into rows, where Row0 is formed from s_{00}, s_{01}, s_{02}, and s_{03}, Row1 from s_{10}, s_{11}, s_{12}, and s_{13}, and similarly for Rows 2 and 3. The key transformation here is the cyclic shifting of these rows, which is fundamental to achieving diffusion in AES encryption. The first row (Row0) remains unchanged, while the second, third, and fourth rows undergo shifts to the left by one, two, and three bytes, respectively. Specifically, Row1 is shifted left by one byte (8 bits), Row2 is shifted left by two bytes (16 bits), and Row3 is shifted left by three bytes (24 bits). These cyclic shifts are achieved by moving the bits from the leftmost positions to the rightmost, preserving the order of bytes in a cyclic fashion. After the rows are shifted accordingly, the transformed rows are reassembled into the output state, maintaining the original column-major order. This reassembly ensures that the result of the transformation is well-defined and easily verifiable, allowing each byte in the state to be located at a predictable position.

2.5 AES_SBOX and AES_SubBytes Modules

The `AES_SBOX` module is responsible for implementing the substitution step in the AES encryption algorithm, which is a critical non-linear transformation that provides confusion in the ciphertext. This substitution is achieved using a ROM-based lookup table, where each possible 8-bit input is mapped to a corresponding 8-bit output based on a predefined S-box. The values for the S-box are initialized from an external file using the $readmemh system task, which allows for efficient loading of the S-box data from a hexadecimal file. This approach provides a fixed, static dataset that can be quickly accessed during the encryption process, ensuring high performance in hardware implementations. The `AES_SBOX` module accepts an 8-bit input byte, referred to as `sbox_in`, and performs a combinational lookup operation in the S-box ROM, where the input byte directly indexes into the ROM to retrieve the substituted output byte, `sbox_out`. This lookup operation is simple and efficient, providing the desired substitution as defined by the AES standard.

The `AES_SubBytes` module enhances throughput by applying the substitution transformation concurrently across all 16 bytes of the AES state. This is achieved by instantiating 16 parallel instances of the `AES_SBOX` module, one for each byte in the 128-bit input state. Each instance of `AES_SBOX` takes an 8-bit slice of the input state, performs the substitution using the ROM-based lookup, and then outputs the substituted byte. The parallelization of the substitution process is facilitated by the `genvar` and `generate` constructs, which iterate over the 16 bytes of the input state and instantiate a separate `AES_SBOX` module for each byte. Specifically, for each iteration, the appropriate byte of the input state is extracted using a bit-select operation (`state_in[127-i*8-:8]`), passed to the corresponding `AES_SBOX` instance, and the resulting substituted byte is placed in the corresponding position of the output state (`state_out[127-i*8-:8]`). This parallel substitution approach ensures that the AES-128 SubBytes step can be performed in a single clock cycle, significantly improving the throughput of the encryption process and aligning with the performance requirements of the AES-128 standard.

2.6 AES_Core Module

The `AES_Core` module serves as the central component of the AES encryption system, orchestrating the entire encryption process by integrating the various AES transformations: key expansion, SubBytes, ShiftRows, MixColumns, and AddRoundKey. The module operates in accordance with the AES round structure, coordinating these transformations in a precise sequence to ensure that the encryption process is carried out correctly. The process is managed by a finite state machine (FSM) that controls the progression through different stages: the initial state, multiple encryption rounds, and the final encryption stage. The FSM operates synchronously with the system clock and ensures that each operation is triggered in the correct sequence, facilitating smooth data exchanges between the various components. The FSM also ensures that the encryption process begins when the `start` signal is received and concludes when the `done` signal is asserted. The `done` signal indicates the completion of the encryption process, at which point the resulting ciphertext is available at the `ciphertext` output.

The AES algorithm begins in the `IDLE` state, where the system awaits the `start` signal to initiate the encryption process. Upon receiving the `start` signal, the FSM transitions to the `INIT` state, where the key expansion module is used to generate the round keys. The `AES_KeyExpansion` module computes the round keys based on the input cipher key, which are then stored in the `round_keys` array. These round keys are critical for the AddRoundKey transformation at each round of encryption. In the `INIT` state, the input plaintext is transformed with the first round key through the `AES_AddRoundKey` module, which produces the initial state for the encryption process. After this, the FSM transitions to the ROUND state, where the AES transformations are applied repeatedly.

In the ROUND state, the encryption process proceeds through a series of transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The AES_SubBytes module substitutes each byte of the state using the AES S-box, providing non-linearity to the cipher. The AES_ShiftRows module then performs a cyclic shift of the rows of the state, further contributing to diffusion. After that, the AES_MixColumns module mixes the columns of the state, ensuring that the bits in each column are diffused across all rows. The round key for the current round is then applied to the transformed state using the AES_AddRoundKey module, completing one round of the AES encryption process. The process repeats for 9 rounds, with the state being updated at each round. The AES_AddRoundKey module is used again in the final round to add the last round key, but without the MixColumns step, as the final state is ready to be outputted as the ciphertext.

The FSM ensures that the process progresses through these stages in the correct order, with the state being updated appropriately after each transformation. When the final round is complete, the FSM transitions to the FINISH state, where the done signal is asserted, indicating that the encryption process is complete and the ciphertext is available. This hierarchical and well-coordinated approach ensures that the AES encryption algorithm is implemented efficiently and accurately, with the FSM managing the sequencing of operations and data flow between modules. The AES_Core module thus acts as the central controller for the encryption process, providing a clear and organized structure for AES encryption.

With the design modularized into distinct functional blocks, it is essential to ensure the correctness and reliability of each module. The next section discusses the testing and verification strategy, detailing how each module and the overall encryption system are rigorously validated against expected outputs.

3 Testing and Verification

A comprehensive verification strategy has been implemented to ensure that every module of the AES-128 encryption system functions as specified by the standard. The testbenches utilize precomputed outputs from a detailed AES step-by-step guide (available at https://legacy.cryptool.org/en/cto/aes-step-by-step) to rigorously compare module outputs against expected values. These testbenches not only check basic functionality but also provide detailed error reporting to quickly pinpoint any discrepancies during simulation.

The AES_AddRoundKey testbench verifies the bitwise XOR operation between a 128-bit input state and a round key. The expected output is precomputed, and the testbench checks for any deviation, reporting errors immediately. The AES_KeyExpansion module is tested by initializing a 128-bit cipher key and generating eleven round keys, each of which is compared against known expected values from the AES standard. Any mismatch is flagged, ensuring that cyclic rotation, S-box substitution, and round constant addition are correctly implemented.

To validate the AES_MixColumns transformation, a test vector with a single active column is used, allowing precise debugging of the GF(2^8) arithmetic operations. The testbench ensures that the column transformation aligns with expected mathematical results. The AES_ShiftRows module is tested by providing a column-major ordered state, applying row shifts, and then checking the final output against a manually computed reference. This guarantees that the byte reordering step is correctly executed.

For the AES_SubBytes module, two sets of test vectors are used: one where all input bytes are 0x00 (expecting an output of 0x63 for each byte) and another using incremental values from 0x00 to 0x0F, ensuring that the S-box performs correct substitutions. Any errors are logged with detailed messages. Finally, at the system level, the AES_Core testbench simulates the full encryption process. This includes clock generation, reset handling, and start signal management. A plaintext and key are provided, and the resulting ciphertext is compared against a precomputed expected value, verifying the correct integration of all AES operations.

These testbenches collectively validate the correctness of the AES-128 implementation, ensuring compliance with cryptographic standards and robustness in encryption functionality.

4 Conclusion

The project demonstrates a robust implementation of the AES-128 encryption algorithm in SystemVerilog using a modular design approach. Each module is carefully structured to perform a distinct function within the encryption process, ensuring clarity, reusability, and ease of verification. The design incorporates both combinational and sequential logic, including finite state machines (FSMs) for control flow, arithmetic operations for finite field computations, and ROM-based substitution tables to optimize efficiency.

Detailed explanations of each module reveal how the encryption process—from key expansion to round transformations—is executed in a structured manner. The key expansion module correctly derives round keys using cyclic transformations, S-box lookups, and round constants, ensuring proper diffusion of key material throughout the encryption rounds. The core AES transformation steps—SubBytes, ShiftRows, MixColumns, and AddRoundKey—are implemented with strict adherence to the AES standard, leveraging column-major data organization and efficient finite field arithmetic.

Comprehensive testing at both the module and system levels confirms that the design meets AES specifications. The verification process includes unit tests for individual modules and integrated system-level simulations, with expected outputs cross-checked against reference implementations and external AES computation tools. The simulation results validate the correctness of key expansion, data transformations, and encryption output, ensuring reliability and compliance with the AES standard.

Overall, this implementation highlights the effectiveness of modular hardware design in cryptographic applications, demonstrating the feasibility of AES-128 encryption in hardware environments. The design principles and verification methodologies used in this project provide a strong foundation for future optimizations, such as pipelined or hardware-accelerated AES implementations, making it a valuable reference for secure embedded systems and FPGA-based cryptographic applications.