Design and Implementation of a UVM-Based AES-128 Verification Environment

Prepared by

Mahmoud Ahmed Mohamed

April 2025

Table of contents

Table of figures

Introduction

Overview of AES-128

AES-128 (Advanced Encryption Standard with a 128-bit key) is a symmetric encryption algorithm widely used for securing digital communications. It was established by the National Institute of Standards and Technology (NIST) and is known for its balance between security and efficiency. AES-128 operates on 128-bit data blocks and uses a series of transformations, including substitution, permutation, and mixing, to achieve strong encryption. The algorithm consists of multiple rounds of processing, where each round enhances the security of the plaintext until the final ciphertext is obtained.

Project Objectives

This project aims to implement AES-128 encryption in Verilog, focusing on achieving a functional and efficient hardware design. The main objectives include:

- Designing and implementing all core AES-128 operations, including SubBytes, ShiftRows, MixColumns, AddRoundKey, and Key Expansion.
- Developing a Finite State Machine (FSM) to control the encryption process.
- Ensuring proper integration of all modules to achieve a seamless encryption flow.
- Simulating and verifying the functionality of the AES-128 implementation using testbenches initially and then building a UVM verification environment to test it.
- Analyzing resource utilization and performance to optimize the design for hardware implementation.

By achieving these objectives, the project provides a practical and hardware-efficient AES-128 encryption implementation suitable for security applications.

AES-128 Architecture

AES-128 follows a well-defined encryption structure that consists of multiple rounds of transformation. The encryption process involves applying a series of operations on a 128-bit data block using a 128-bit key. Each round introduces confusion and diffusion to strengthen security. The main components of AES-128 encryption include SubBytes, ShiftRows, MixColumns, and AddRoundKey operations. Additionally, the Key Expansion module plays a crucial role in generating round keys from the original encryption key. The architecture ensures secure and efficient encryption suitable for hardware implementation.

Block Diagram

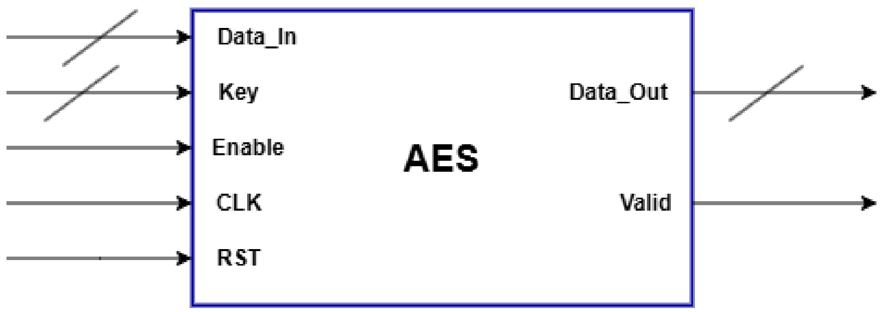


Figure 1: AES Block Diagram

I/O Ports Description

The AES-128 module interfaces with external components through a set of input and output ports. These ports allow the module to receive plaintext and encryption keys, process data synchronously, and output the encrypted ciphertext. The primary ports are described below:

Port Name	Direction	Width	Description
Data_In	Input	128 bits	Plaintext input block to be encrypted.
Key	Input	128 bits	128-bit secret key used for encryption.
Enable	Input	1 bit	Enable signal to start the encryption process.
Clock (CLK)	Input	1 bit	System clock signal to synchronize operations.
Reset (RST)	Input	1 bit	Active-low reset signal to initialize the module.
Data_Out	Output	128 bits	Ciphertext output after AES-128 encryption.
Valid	Output	1 bit	Indicates successful encryption operation.

AES Encryption Flow

AES-128 encryption follows a structured sequence of operations applied to a 128-bit data block using a 128-bit key. The process consists of multiple rounds, where each round increases security by introducing confusion and diffusion. The encryption relies on round keys derived from the original key using the **Key Expansion module**.

Encryption Process Steps:

1. **Initial AddRoundKey:**
   - The plaintext undergoes an initial XOR operation with the first-round key, ensuring the key is integrated from the start.
2. **Main Rounds (9 Rounds in AES-128):** Each round consists of the following transformations:
   - **SubBytes:** Non-linear substitution using an S-Box for confusion.
   - **ShiftRows:** Row-wise permutation of bytes to break byte dependencies.
   - **MixColumns:** Linear mixing of column values to diffuse bits across the block (**excluded in the final round**).
   - **AddRoundKey:** XOR operation with the corresponding round key to introduce key dependency.
3. **Final Round (10th Round):**
   - The same as the main rounds but without the **MixColumns** transformation.
4. **Ciphertext Output:**
   - The final transformed block is the encrypted ciphertext.

“The following diagram visually represents the encryption flow of AES-128, detailing the sequential transformations applied to the input data where Nr = number of rounds. “

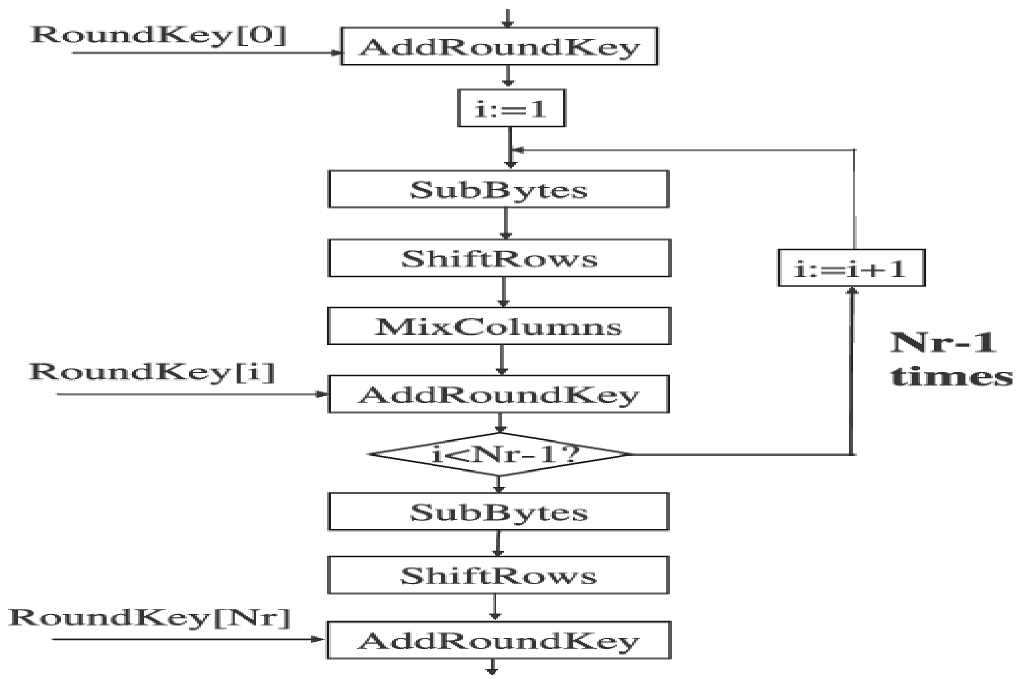


Figure 2: AES Encryption Flow Chart

Role of Key Expansion in AES-128

Key Expansion is a critical component of AES-128 encryption, responsible for generating **round keys** from the original 128-bit key. Since AES-128 consists of **10 encryption rounds**, a total of **11 round keys** (including the initial key) are required. This process ensures that each round applies a unique transformation, preventing attackers from easily deciphering the encrypted data.

Key Expansion Process:

The Key Expansion module generates round keys using a recursive process based on the original 128-bit key. It involves **four major steps**:

1. Key Splitting into Words

- The 128-bit key is divided into **four 32-bit words** (**W[0] to W[3]**) as the first set of round keys.
- Each subsequent round key is derived by modifying the previous words.

2. RotWord (Byte Rotation)

- Every **fourth word** (W[i]) undergoes **RotWord**, which cyclically shifts its bytes left by one position.
  - Example: If W[3] = **{A1, B2, C3, D4}**, after RotWord: **{B2, C3, D4, A1}**

3. SubWord (Byte Substitution using S-Box)

- The rotated word then undergoes **SubWord**, replacing each byte with its corresponding value from the AES **S-Box** (Substitution Box).
  - Example: If the rotated word is **{B2, C3, D4, A1}**, each byte is replaced using the S-Box.

4. XOR Operations (Adding Non-Linearity & Round Constant)

- The transformed word is XORed with:
  1. **Previous Round Key (W[i-4])**
  2. **Round Constant (Rcon[j])**, which introduces round-specific uniqueness
  - **Rcon values** are predefined constants ensuring different round keys.
  - **Finally W[i] =** W[i−4] ⊕ SubWord(RotWord(W[i−1])) ⊕ Rcon[j] if (i mod4) = 0
- **Remaining words** (W[i+1] to W[i+3]) are computed by XORing them with the previous word:
  - W[i+1] = W[i] ⊕ W[i-3]
  - W[i+2] = W[i+1] ⊕ W[i-2]
  - W[i+3] = W[i+2] ⊕ W[i-1]

"The following diagram visually represents the key expansion process in AES-128, illustrating how the initial key is transformed into round keys through substitution, rotation, and XOR operations where g = SubWord(RotWord(W[i−1])) ⊕ Rcon[j]."

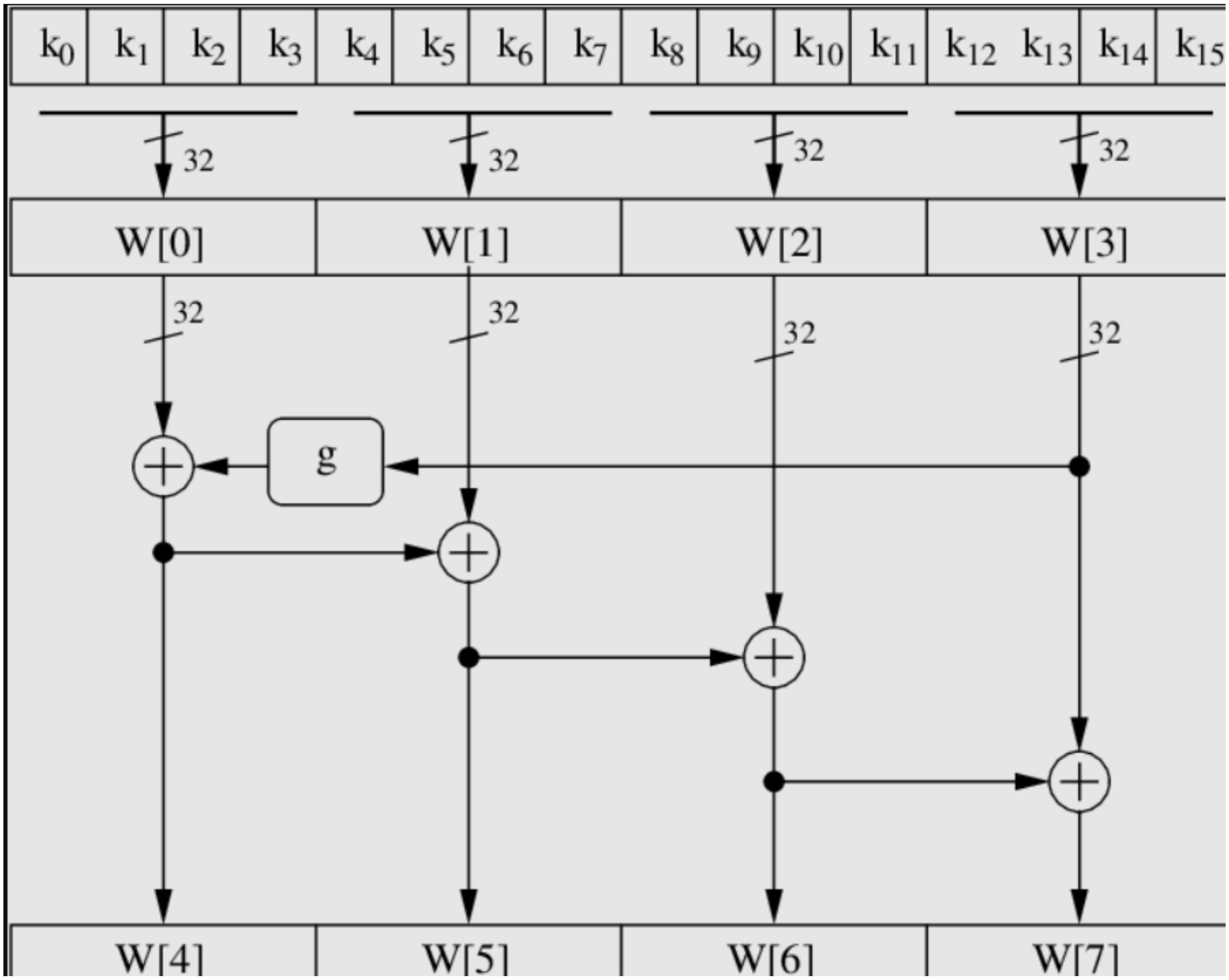


Figure 3: Key Expansion Flow Diagram

Implementation Details

The implementation of AES-128 encryption in hardware requires the design and integration of multiple modules that work together to perform encryption efficiently. Each module corresponds to a specific transformation in the AES encryption process. The key design considerations include modularity, efficient resource utilization, and optimizing for performance while ensuring correctness.

The implementation consists of the following major components:

- **Finite State Machine (FSM):** Controls the sequencing of encryption steps.
- **AddRoundKey:** Applies the round key to the state using bitwise XOR.
- **SubBytes:** Performs a non-linear byte substitution using the AES S-Box.
- **ShiftRows:** Rearranges the bytes within each row to increase diffusion.
- **MixColumns:** Performs matrix multiplication to enhance diffusion (not used in the final round).
- **Key Expansion:** Generates round keys from the initial cipher key for all encryption rounds.

Module Descriptions

Finite State Machine (FSM)

The **FSM** controls the sequence of AES operations by managing different states, such as:

1. **Idle:** The module waits for an enable signal to start encryption.
2. **Key Expansion:** It’s the first state to generate all round keys before encryption begins.
3. **Initial AddRoundKey:** XORs the plaintext with the first-round key.
4. **Main Rounds (1–9):** Performs SubBytes, ShiftRows, MixColumns, and AddRoundKey sequentially.
5. **Final Round:** Executes SubBytes, ShiftRows, and AddRoundKey (without MixColumns).

```
//////////////////////////////////////////////////////
// Define States
localparam IDLE         = 3'b000;
localparam KEY_EXPANSION = 3'b001;
localparam ADD_ROUND_KEY = 3'b010;
localparam SUB_BYTES     = 3'b011;
localparam SHIFT_ROWS    = 3'b100;
localparam MIX_COLUMNS   = 3'b101;

//////////////////////////////////////////////////////
// Internal Connections
reg  [0:2] current_state, next_state;

//////////////////////////////////////////////////////
// State Register Update
always @(posedge CLK or negedge RST) begin
    if (!RST) begin
        current_state   <= IDLE;
        round_count     <= 4'b0000;
    end
    else begin
        current_state   <= next_state;
        if (current_state == ADD_ROUND_KEY && add_key_done) begin
            round_count <= round_count + 1;
        end
        else if (current_state == IDLE) begin
            round_count <= 4'b0000;
        end
    end
end
```

Figure 4: FSM Module Snippet 1

```
    case (current_state)
        IDLE: begin
            if (Enable) begin
                next_state    = KEY_EXPANSION;
            end
        end

        KEY_EXPANSION: begin
            key_expan_en = 1'b1;
            if (key_expan_done) begin
                next_state = ADD_ROUND_KEY;
            end
        end

        ADD_ROUND_KEY: begin
            add_roundkey_en = 1'b1;
            // Select input for AddRoundKey
            if (round_count == 4'd0) begin
                data_sel_init = 1'b1;      // Use original Data_In
            end
            else if (round_count == 4'd10) begin
                data_sel_final = 1'b1;     // Use ShiftRows output
            end
            if (add_key_done) begin
                if (round_count == 4'd10) begin
                    Data_Out_VLD     = 1'b1;
                    data_out_en      = 1'b1;       // Output is now valid
                    next_state       = IDLE;
                end
                else begin
                    next_state = SUB_BYTES;
                end
            end

        end

        SUB_BYTES: begin
            sub_bytes_en = 1'b1;
            if (sub_bytes_done) begin
                next_state = SHIFT_ROWS;
            end
        end
```

Figure 5: FSM Module Snippet 2

Comment: The round counter in the AES-128 state machine controls the encryption flow by tracking the number of rounds and ensuring correct state transitions. It starts at round 0 with the initial AddRoundKey operation and increments after each round. During the ShiftRows state, the counter determines whether the encryption has reached round 10; if so, it directs the transition to AddRoundKey while skipping MixColumns, which is omitted in the final round. The counter ensures that encryption does not exceed the required 10 rounds, and upon completion, it signals the end of the process, allowing the state machine to transition back to IDLE, ready for the next encryption operation.

In addition to controlling state transitions, the round counter also determines the **input data to the AddRoundKey module** by selecting from three different sources: the original plaintext input during round 0, the output of the MixColumns module during intermediate rounds, and the output of the ShiftRows module during the final round. This ensures that the correct data is processed at each encryption stage. The round counter also plays a critical role in validating the **final encrypted output (Data_Out)**, which is only assigned once the final AddRoundKey operation (round 10) is completed. This mechanism guarantees that AES-128 follows the correct sequence of transformations, maintaining both encryption accuracy and cryptographic security.

AddRoundKey

- module performs a bitwise **XOR** between the current state and the corresponding round key.
- Since XOR is computationally simple, this module operates efficiently in a single clock cycle.

```
module add_round_key(
    input           [127:0]     data_in,
    input           [127:0]     round_key,
    input                       enable,
    input                       CLK,
    input                       RST,
    output reg      [127:0]     data_out,
    output reg                  done_flag        // done signal flag
);

reg      [127:0]     data_out_c;
reg                  done_flag_c;
always @(posedge CLK or negedge RST) begin
        if (!RST) begin
            data_out    <= 0    ;
            done_flag   <= 1'b0 ;
        end
        else if (enable) begin
            data_out    <= data_out_c   ;
            done_flag   <= done_flag_c ;
        end
        else begin
            done_flag   <= 1'b0 ;
        end
    end

always @(*) begin

    data_out_c  = data_in ^ round_key;
    done_flag_c = 1'b1;

end

endmodule
```

Figure 6: Add Round Key Module Snippet

SubBytes

- Implements a **non-linear substitution** using the AES **S-Box**.
- Each byte in the 128-bit block is replaced using a precomputed lookup table.
- This operation enhances security by introducing confusion in the data.

```
module sub_bytes (
    input       [0:127]         bytes,
    input                       enable,       // Enable Signal
    input                       CLK,
    input                       RST,
    output reg  [0:127]         sub_bytes,
    output reg                  done_flag     // done signal flag
);

wire      [0:127]     sub_bytes_c;

always @(posedge CLK or negedge RST) begin
        if (!RST) begin
            sub_bytes   <= 0   ;
            done_flag  <= 1'b0 ;
        end
        else if (enable) begin
            sub_bytes   <= sub_bytes_c ;
            done_flag   <= 1'b1 ;
        end
        else begin
            done_flag  <= 1'b0 ;
        end
    end

genvar i;
generate
    for (i = 0; i < 128; i = i + 8) begin
        assign sub_bytes_c[i +: 8] = sbox(bytes[i +: 8]);
    end
endgenerate
```

Figure 7: SubBytes Module Snippet

ShiftRows

- A **row-wise permutation** applied to the state matrix:
  - **Row 0:** No shift.
  - **Row 1:** Shift left by 1 byte.
  - **Row 2:** Shift left by 2 bytes.
  - **Row 3:** Shift left by 3 bytes.
- This transformation increases diffusion across columns.

```
always @(posedge CLK or negedge RST) begin
        if (!RST) begin
            shifted   <= 0     ;
            done_flag  <= 1'b0 ;
        end
        else if (enable) begin
            shifted    <= shifted_c   ;
            done_flag  <= done_flag_c ;
        end
        else begin
            done_flag  <= 1'b0 ;
        end
    end

always @(*) begin
    // First row (r = 0) is not shifted
    shifted_c[0+:8] = in[0+:8];
    shifted_c[32+:8] = in[32+:8];
    shifted_c[64+:8] = in[64+:8];
    shifted_c[96+:8] = in[96+:8];

    // Second row (r = 1) is cyclically left shifted by 1 offset
    shifted_c[8+:8] = in[40+:8];
    shifted_c[40+:8] = in[72+:8];
    shifted_c[72+:8] = in[104+:8];
    shifted_c[104+:8] = in[8+:8];

    // Third row (r = 2) is cyclically left shifted by 2 offsets
    shifted_c[16+:8] = in[80+:8];
    shifted_c[48+:8] = in[112+:8];
    shifted_c[80+:8] = in[16+:8];
    shifted_c[112+:8] = in[48+:8];

    // Fourth row (r = 3) is cyclically left shifted by 3 offsets
    shifted_c[24+:8] = in[120+:8];
    shifted_c[56+:8] = in[24+:8];
    shifted_c[88+:8] = in[56+:8];
    shifted_c[120+:8] = in[88+:8];

    done_flag_c = 1'b1;
end


endmodule
```

Figure 8: Shift Rows Module Snippet

MixColumns

- A matrix transformation that mixes bytes within each column using **Galois Field (GF) multiplication**.
- It provides additional diffusion by spreading changes across the entire data block.
- This operation is omitted in the **final round** to maintain structure.

```
always @(posedge CLK or negedge RST) begin
        if (!RST) begin
            mixed      <= 0    ;
            done_flag  <= 1'b0 ;
        end
        else if (enable) begin
            mixed      <= mixed_c     ;
            done_flag  <= 1'b1 ;
        end
        else begin
            done_flag  <= 1'b0 ;
        end
    end

genvar col;

generate

    for (col = 0; col < 4; col = col + 1) begin
    assign mixed_c[(col*32)   +: 8] = multiply_by_two(in[(col*32)   +: 8]) ^ multiply_by_three(in[(col*32+8)  +: 8]) ^ in[(col*32+16)  +: 8] ^ in[(col*32+24)  +: 8];
    assign mixed_c[(col*32+8) +: 8] = in[(col*32)   +: 8] ^ multiply_by_two(in[(col*32+8)  +: 8]) ^ multiply_by_three(in[(col*32+16) +: 8]) ^ in[(col*32+24)  +: 8];
    assign mixed_c[(col*32+16)+: 8] = in[(col*32)   +: 8] ^ in[(col*32+8)  +: 8] ^ multiply_by_two(in[(col*32+16) +: 8]) ^ multiply_by_three(in[(col*32+24) +: 8]);
    assign mixed_c[(col*32+24)+: 8] = multiply_by_three(in[(col*32)   +: 8]) ^ in[(col*32+8)  +: 8] ^ in[(col*32+16) +: 8] ^ multiply_by_two(in[(col*32+24) +: 8]);
    end

endgenerate
```

Figure 9: Mix Columns Module Snippet 1

```
// Function to multiply by 2 in GF(2^8)
function [7:0] multiply_by_two;
    input [7:0] byte;

    begin
        if (byte[7] == 1'b1) begin
            multiply_by_two = (byte << 1) ^ 8'h1b;
        end
        else begin
            multiply_by_two = (byte << 1);
        end
    end

endfunction
```

Figure 11: Mix Columns Module Snippet 3

```
// Function to multiply by 3 in GF(2^8)
function [7:0] multiply_by_three;
    input [7:0] byte;


    begin
        multiply_by_three = multiply_by_two(byte) ^ byte;
    end


endfunction
```

Figure 10: Mix Columns Module Snippet 2

Comment: The MixColumns module applies the AES MixColumns transformation, ensuring diffusion by mixing each column of the state matrix. This is achieved using two key functions: multiply_by_two and multiply_by_three, which perform finite field multiplication in GF(2^8). The multiply_by_two function shifts the input left by one bit, and if the most significant bit (MSB) was set before the shift, it conditionally XORs the result with 0x1B (the AES irreducible polynomial) to keep the result within GF(2^8). The multiply_by_three function computes multiplication by 0x03 as multiply_by_two(in) ^ in, combining both transformations. These functions enable the MixColumns operation by performing matrix multiplication on each byte, enhancing security by ensuring that small changes in the input propagate across the state.

Key Expansion

- Derives **round keys** from the initial 128-bit key for all encryption rounds.
- Uses **byte substitution, rotation, and XOR operations** to generate new keys iteratively as mentioned above.

- The expanded keys ensure that each encryption round uses a different key, strengthening security.

```
always @(*)  begin
    // Load initial key into w[0] to w[3]
    w[0] = key[0+:32];
    w[1] = key[32+:32];
    w[2] = key[64+:32];
    w[3] = key[96+:32];

    // Generate round keys
    for (i = 4; i < 44; i = i + 1) begin
        if (i % 4 == 0) begin
            w[i] = w[i-4] ^ sub_word(rot_word(w[i-1])) ^ rcon(i >> 2);
        end else begin
            w[i] = w[i-4] ^ w[i-1];
        end
    end

    // Store in round_keys
    for (i = 0; i < 11; i = i + 1) begin
        round_keys_c[i*128 +: 128] = {w[i*4], w[i*4+1], w[i*4+2], w[i*4+3]};
    end
    done_flag_c = 1'b1;
end
endmodule
```

Figure 12: Key Expansion Module Snippet 1

```
// Rotate Left (RotWord)
function [31:0] rot_word;
    input [31:0] word;
    begin
        rot_word = {word[23:0], word[31:24]};
    end
endfunction

// SubWord (Apply S-box)
function [31:0] sub_word;
    input [31:0] word;
    begin
        sub_word = {sbox(word[31:24]), sbox(word[23:16]), sbox(word[15:8]), sbox(word[7:0])};
    end
endfunction

// Round Constants (Rcon)
function [31:0] rcon;
    input [3:0] round;
    begin
        case (round)
            4'h1: rcon = 32'h01000000;
            4'h2: rcon = 32'h02000000;
            4'h3: rcon = 32'h04000000;
            4'h4: rcon = 32'h08000000;
            4'h5: rcon = 32'h10000000;
            4'h6: rcon = 32'h20000000;
            4'h7: rcon = 32'h40000000;
            4'h8: rcon = 32'h80000000;
            4'h9: rcon = 32'h1b000000;
            4'hA: rcon = 32'h36000000;
            default: rcon = 32'h00000000;
```

Figure 13: Key Expansion Module Snippet 2

AES-128 Encryption RTL Design Synthesis

The AES-128 encryption RTL design was synthesized using Xilinx Vivado, targeting the Artix-7 FPGA (xc7a200tffg1156-2), selected for its large number of I/O pins and a clock period of **30ns**. The synthesis process successfully met all timing constraints, and the power consumption for the design was estimated.

Synthesized Design Schematic:

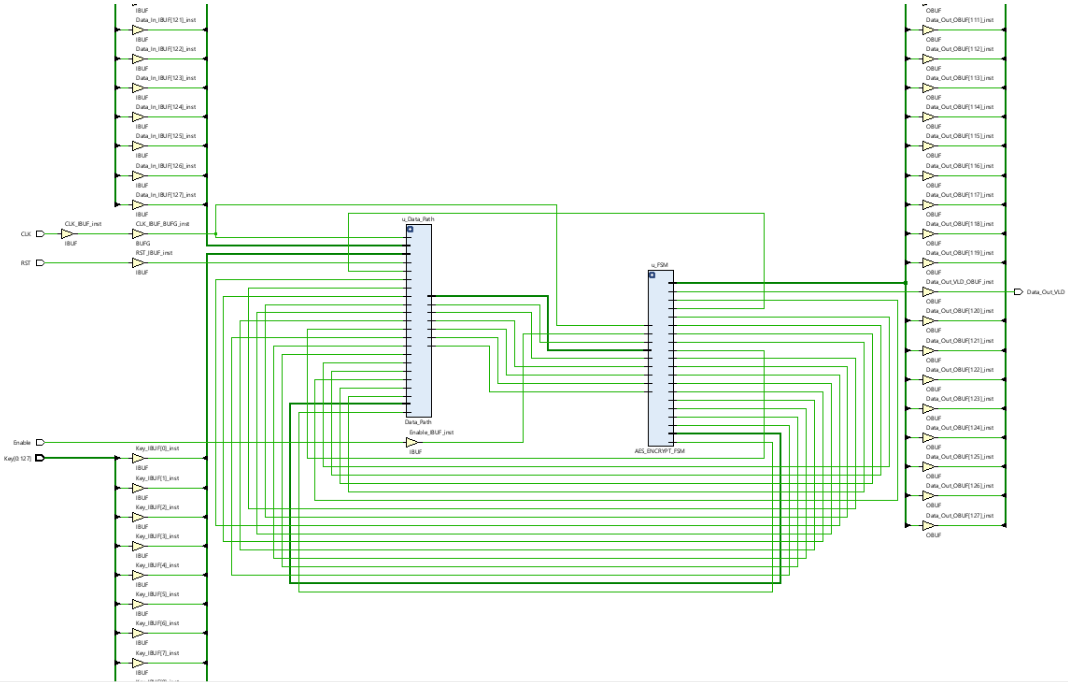


Figure 14: Design Schematic

Timing Analysis:

- **Target Frequency:** 33.33 MHz (30 ns clock period)
- **Setup Slack: +1.764 ns**
- **Hold Slack: 0.038 ns**

Design met all timing constraints.

Design Timing Summary

Setup	
Worst Negative Slack (WNS):	1.764 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	5959

Hold	
Worst Hold Slack (WHS):	0.038 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	5959

Pulse Width	
Worst Pulse Width Slack (WPWS):	14.500 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	1946

All user specified timing constraints are met.

Figure 15: Timing Report

Resource Utilization:

- **LUTs: 3961**
- **FFs: 1945**
- **I/O pins: 388**

Summary

Resource	Utilization	Available	Utilization %
LUT	3961	134600	2.94
FF	1945	269200	0.72
IO	388	500	77.60

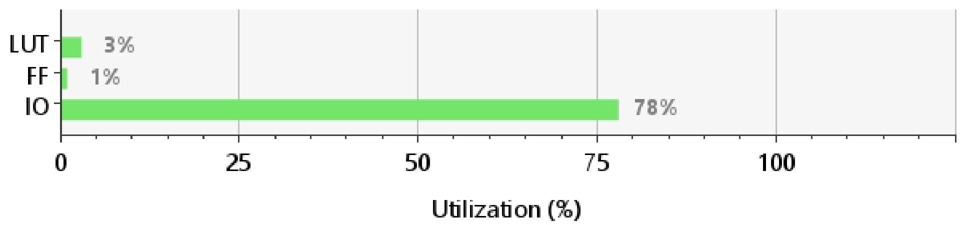


Figure 16: Utilization Report

Power Consumption (Estimated):

- **Static Power: 0.131 W**
- **Dynamic Power: 0.216 W**
- **Total Power: 0.347 W**

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.347 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (40.7 W)
Effective ϑJA:	1.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

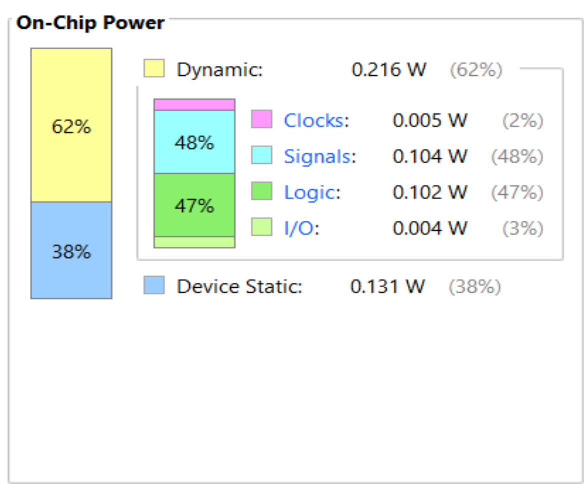


Figure 17: Power Report

Verification Strategy

Initial Testbench and Results

Testbench Structure

To validate the basic functionality of the AES-128 encryption module, a simple testbench was developed. The testbench initializes the AES module, provides test vectors for encryption, and captures the output. It operates in a simulation environment using **QuestaSim**.

Test Vectors and Expected Outputs

The testbench applies two sets of plaintext and key inputs to the AES module. The expected behavior is that after a specific number of clock cycles, the module should produce a valid encrypted output, with the Valid signal indicating when the output is ready.

Testbench Code

The following Verilog testbench was used to validate the AES encryption module:

```verilog
// Testbench initial block to provide stimulus
initial begin
    // Initialize signals
    CLK = 1'b0;            // Initial clock value
    RST = 1'b0;            // Reset is initially low (inactive)

    // Apply reset signal
    #5
    RST = 1'b1;            // Assert reset for 1 time unit

    // Monitor input and output signals during simulation
    $monitor("in= %h, key= %h ,out= %h ,valid=%h", In, Key, Out, Valid);

    // Set test vector 1: Plaintext and Key
    In = 128'h3243f6a8_885a308d_313198a2_e0370734;  // Sample input data
    Key = 128'h2b7e1516_28aed2a6_abf71588_09cf4f3c; // Sample key

    // Enable AES encryption
    Enable = 1'b1;

    // Run for some time to let the encryption process happen
    #20
    Enable = 1'b0;  // Disable the Enable signal after 20 time units

    // Set test vector 2: Another set of plaintext and key for testing
    #820;
    In = 128'h01234567_89abcdef_01234567_89abcdef;   // New input data
    Key = 128'h1642268b_1b156c52_62c70b10_4ad7ef34;  // New key

    // Enable AES encryption again for the new test vector
    Enable = 1'b1;

    // Run the simulation for a longer time
    #850

    // End the simulation after running the test cases
    $finish;
end
```

Figure 18: Testbench Snippet

Simulation Results and Observations

The waveform captured from **QuestaSim** verifies the correct behavior of the AES encryption module. The key observations include:

- The input plaintext and key are correctly applied at the start of the encryption process.
- After a certain delay, the module outputs the encrypted data.

- The Valid signal is asserted, confirming that the output is ready.

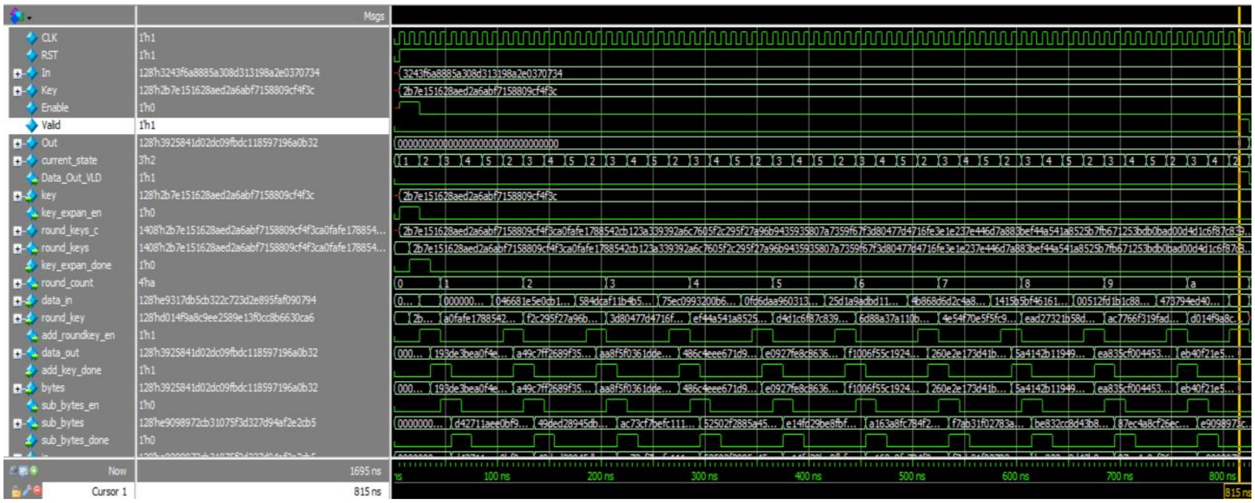


Figure 19: Results Waveform

Limitations of the Initial Testbench

This testbench provides a basic validation of the AES encryption module but has some limitations:

- No extensive corner case testing was performed.
- No automated checking of expected outputs against known AES test vectors.

This initial validation serves as a foundation for more comprehensive verification using a **UVM-based environment**, which will be discussed in the next section

Verification Plan and Methodology

The verification strategy follows a structured approach, transitioning from a basic directed testbench to a more robust UVM-based test environment.

- **Verification Goals:** Ensure the AES-128 encryption operates correctly across various scenarios.
- **Methodology:**
  - **Directed Testing** (as in the initial testbench).
  - **Constrained Random Testing** to explore a wide range of inputs.
  - **Coverage-Driven Verification** to ensure all features are exercised.
- **Simulation Tools Used:** QuestaSim for waveform analysis and debugging.

Test Case Table

This section outlines the test cases, their scope, expected results, and pass/fail criteria for verifying the features listed above.

Test Case	Purpose and Scope	Expected Results	Pass/Fail Criteria
Key Expansion Test	Verify that key expansion generates the correct round keys.	Round keys match the expected expanded values for a given input key.	Pass if all round keys match precomputed reference values.
Initial AddRoundKey Test	Ensure correct initial XOR operation with the first round key.	The output matches the bitwise XOR of plaintext and round key.	Pass if output is correctly computed for all test vectors.
SubBytes Test	Validate correct S-Box substitution for each byte.	Output bytes match S-Box substitution values.	Pass if every byte substitution matches expected values.
ShiftRows Test	Ensure correct shifting of state matrix rows.	State matrix is modified according to AES row shifting rules.	Pass if shifted state matches expected output.
MixColumns Test	Verify correct matrix transformation using $GF(2^8)$ multiplication.	Output columns are transformed as per AES MixColumns rules.	Pass if all output columns match expected transformation results.
Round Key Addition Test	Ensure correct round key XOR operation in each round.	State matrix correctly updates after each round key addition.	Pass if the XOR operation is performed correctly for all rounds.
Final Round Test	Verify that the last round executes correctly without MixColumns.	The final transformed state should match expected values.	Pass if the final state matches the expected AES ciphertext.
Ciphertext Verification Test	Ensure that the final ciphertext output is correct.	Encrypted output matches the expected result for given plaintext and key.	Pass if ciphertext matches expected values from a verified AES implementation.
Edge Case Handling Test	Test AES behavior on special cases like all-zero and all-one inputs.	AES encrypts special cases correctly without errors.	Pass if outputs are correct and no failures occur.

UVM-Based Verification Environment

The class-based environment, while functional, had limitations in terms of scalability and maintainability. To address these limitations, the environment was transitioned to a **UVM-based** approach, which introduced standardized constructions and more robust communication mechanisms.

UVM-Based Environment Block Diagram

Below is a block diagram representing the structure of the UVM-based verification environment:

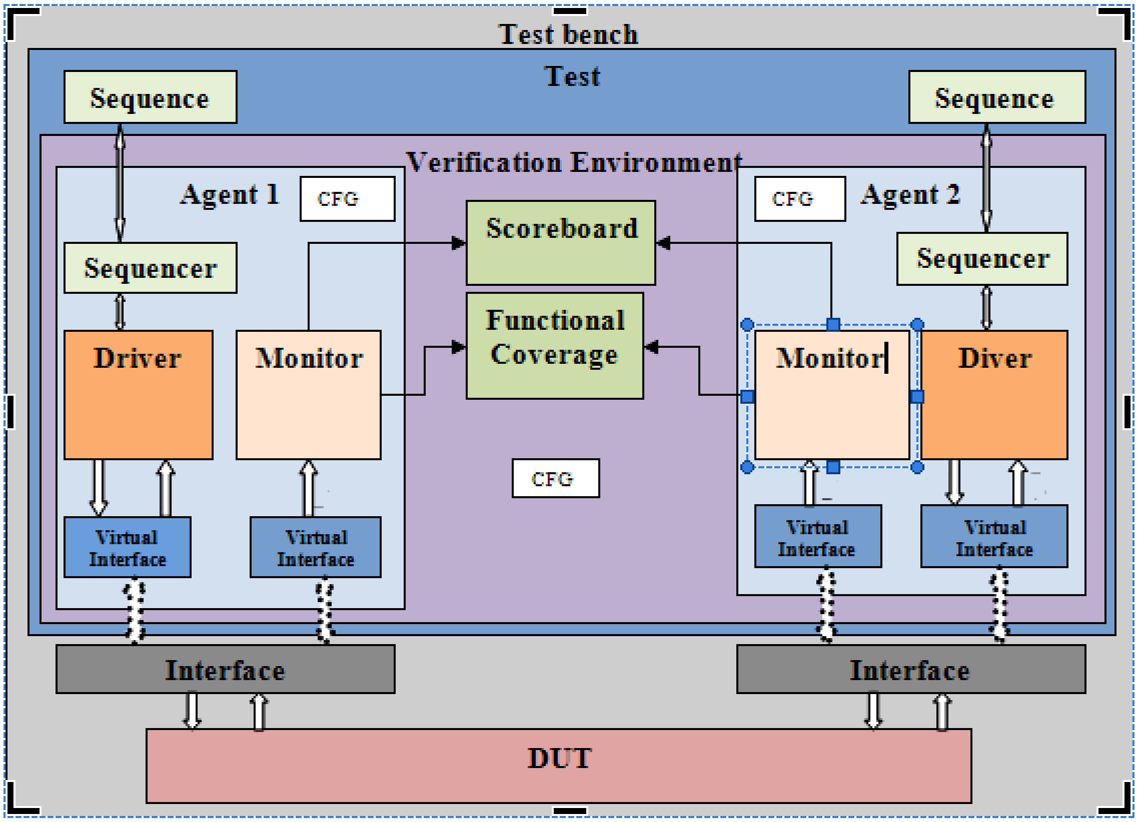


Figure 20: UVM-based Environment

Components of the UVM-Based Environment: -

1. **My_Sequence_item Class (Object Class)**:
   - The **sequence_item class** was used to encapsulate the data and operations for each transaction. This class defined the various fields required for memory operations, such as the Address, Data_in, WrEn, RdEn, and other relevant control signals.
   - The **sequence_item** class allowed the **sequence class** to generate data in a structured format, which was then passed to the **driver** for sending to the DUT.
2. **My_Sequence Class (Object Class):**
   - The **sequence** defines the logic for generating and randomizing transactions (my_sequence_item). In the my_sequence class, transactions are created, randomized, and handed to the driver. Using the start_item and finish_item methods But should connect the sequencer port to driver port in the connect phase of agent class(my_agent).

3. **My_Sequencer Class**:

   - The **sequencer** acts as the central component to manage transaction generation. It collaborates with sequences to create and provide multiple transactions to the driver in a controlled manner.

4. **My_Driver Class:**

   - The **driver** retrieves transactions (sequence items) from the sequencer using the get_next_item method. Each transaction's fields, such as Data_in, Address, WrEn, and RdEn, are driven to the DUT using a virtual interface (v_intf). After completing the transaction, the driver invokes the item_done method, signaling that the transaction has been executed. The my_driver class efficiently handles synchronization, ensuring accurate signal delivery to the DUT.

5. **My_Monitor Class**:

   - The **monitor** observed DUT's outputs by sampling the interface signals. It created transactions (sequence items) with these values and sent them using the put_port_monitor.write() method to connected components, such as the **scoreboard** and **subscriber**, via an **uvm_analysis_port** (put_port_monitor). But should connect the monitor port to both subscriber & scoreboard ports in the connect phase of env class.

6. **My_Agent Class**:

   - The **agent** combined the sequencer, driver and monitor into a modular component. Each agent was responsible for managing the transaction flow and communication between components, ensuring that the testbench was both reusable and scalable.

7. **My_Scoreboard Class**:

   - The **scoreboard** in UVM received the transactions (my_sequence_item) from **monitor** using an **uvm_analysis_import** (get_port) and write() method which is defined in the my scoreboard class to compare the actual outputs with the expected results and tracked the success or failure of each transaction.

8. **My_Subscriber Class**:

   - The **subscriber** received the transactions (sequence items) from **monitor** using an **uvm_analysis_import** (analysis_export) which is defined in uvm_subscriber and write() method which is defined in the my_subscriber class and overridden the write() method in uvm_subscriber class and reported the final results. It allowed for efficient handling of test cases and made the testbench more adaptable to different types of tests.

9. **My_env Class:**

   - My_env class is the top-level environment responsible for creating and connecting all components in the UVM-based verification environment.

   - Instances of the agent, scoreboard, and subscriber are created using the UVM factory in build phase.

- establishing communication between components in connect phase. The agent's analysis port is connected to both the scoreboard and the subscriber, enabling data flow for verification and coverage collection.

10. **My_test Class:**
    - The my_test class is derived from uvm_test and serves as the top-level component to orchestrate the testbench execution.
    - In the build_phase, the test class instantiates the environment (my_env) and the sequence (my_sequence). These components are created using the UVM factory.
    - During the run_phase, the test class initiates the sequence on the sequencer within the environment's agent. It raises a UVM objection at the start of the phase to prevent the simulation from ending prematurely and drops the objection after the sequence execution completes. This ensures controlled execution of the sequence within the simulation.

Communication Mechanisms in UVM-Based Environment

- In the UVM environment, the sequence class initiated and completed transactions using start_item and finish_item. These methods were essential in the sequencer-to-driver communication, with start_item initiating the transaction and finish_item signaling its completion.
- The driver used the get_next_item method to retrieve transactions from the sequencer, ensuring a synchronized flow of data. After completing the transaction, the driver called item_done to notify the sequencer that the transaction was processed. These methods were crucial for maintaining the communication and synchronization between the sequencer and driver, ensuring that transactions were handled one at a time in a controlled manner.
- The monitor used an analysis port (uvm_analysis_port) and write() method to transmit transaction data (sequence items) to downstream components, such as the scoreboard and subscriber.
- The scoreboard implemented an analysis export (uvm_analysis_imp) to receive and process transactions.
- The subscriber collected transactions (sequence items) through the monitor's analysis port.

Interface

The **interface** served as a shared communication medium between the DUT and the components of the environment. It simplified signal management and ensured proper synchronization in the design.

The interface also included two distinct clocking blocks:

1. **Driver Clocking Block (drv_cb):**

   This clocking block was used to drive DUT’s input signals during test execution. It ensured controlled timing for signal transitions, reducing race conditions and improving the stability of the testbench.

2. **Monitor Clocking Block (mon_cb):**

   This clocking block was used to sample the DUT's output and input signals. It ensured accurate sampling synchronized with the clock, improving the reliability and accuracy of the verification process.

Communication Between environment components and Interface

In the **UVM-based environment**, communication between the testbench components (driver, monitor, etc.) and the **interface** was achieved using **virtual interfaces** passed through the **uvm_config_db** mechanism. The **uvm_config_db** served as a central configuration database, enabling the testbench components to retrieve the interface dynamically during runtime in their build phase.

Results

Coverage Reports

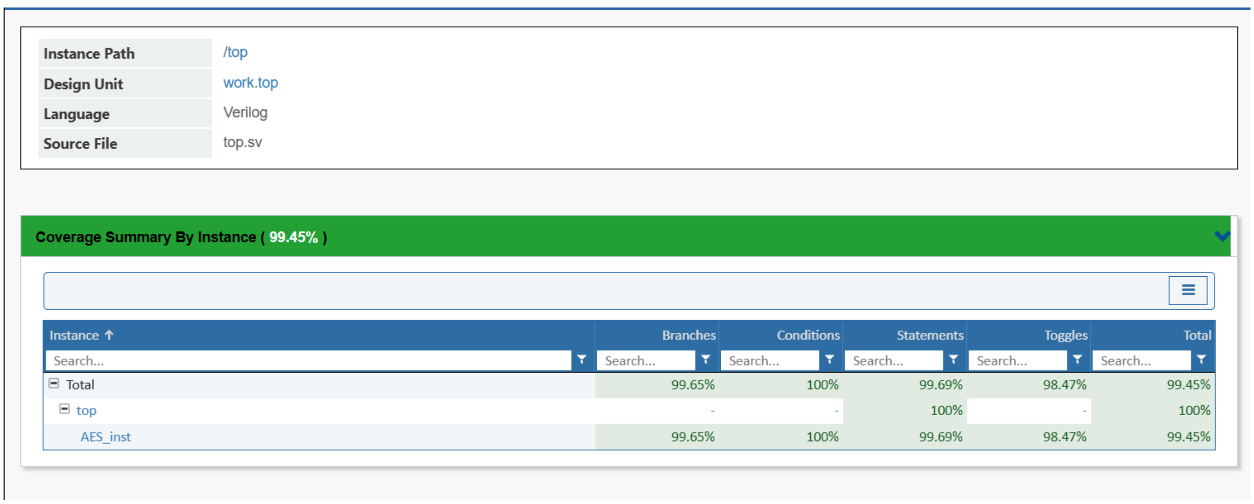
1. Code Coverage

Code coverage provides insights into the portions of the AES-128 DUT code exercised during simulation. This includes metrics such as statement coverage, branch coverage, condition coverage, and toggle coverage.

Below is a figure showing the code coverage report:

Summary:

- Achieved **99.65% branch coverage (not 100% due to default statements in cases)**.
- Achieved **100% condition coverage**.
- Achieved **99.69% statement coverage (not 100% due to default statements in cases)**.
- Achieved **98.47% toggle coverage**.



Instance Path	/top
Design Unit	work.top
Language	Verilog
Source File	top.sv

Coverage Summary By Instance (99.45%)

Instance	Branches	Conditions	Statements	Toggles	Total
Total	99.65%	100%	99.69%	98.47%	99.45%
top	-	-	100%	-	100%
AES_inst	99.65%	100%	99.69%	98.47%	99.45%

Figure 21: Code Coverage For DUT

These results confirm that the DUT code was thoroughly exercised under the test scenarios.

2. Functional Coverage

Validate the correctness and completeness of AES-128 encryption input and output behavior, functional coverage was applied using SystemVerilog covergroups. These covergroups monitor and categorize important test patterns and edge cases for inputs (Data_In, Key, and Enable) and outputs (Data_Out, Data_Out_VLD).

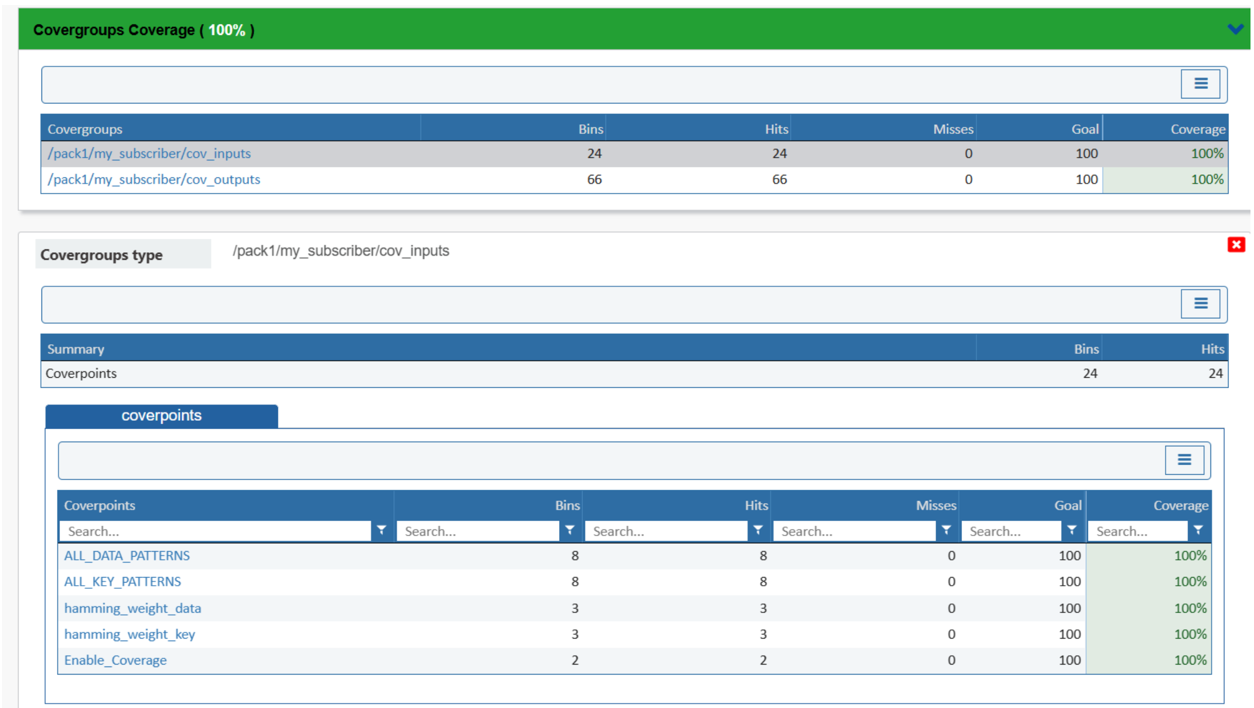
The cov_inputs covergroup was designed to verify key input patterns to ensure robust stimulus generation:

- **Data Patterns on Data_In**
  Covered known edge cases such as:
  - All zeros, all ones
  - Alternating bits (0xAA, 0x55)
  - MSB/LSB set

- Incremental/decremental patterns
- Random data

- **Key Patterns on Key**
  Mirrored the same pattern types as data input to ensure encryption key variability was thoroughly tested.
- **Hamming Weight**
  Both Data_In and Key were verified for different ranges of Hamming weights:
  - Low (0–32 bits set)
  - Medium (33–95 bits set)
  - High (96–128 bits set)
- **Enable Signal**
  The coverage on Enable ensured that encryption only occurred when explicitly triggered.

Result: All defined bins for Data_In, Key, Hamming weights, and Enable were hit during simulation, achieving **100% coverage**.



Covergroups Coverage (100%)

Covergroups	Bins	Hits	Misses	Goal	Coverage
/pack1/my_subscriber/cov_inputs	24	24	0	100	100%
/pack1/my_subscriber/cov_outputs	66	66	0	100	100%

Covergroups type /pack1/my_subscriber/cov_inputs

Summary	Bins	Hits
Coverpoints	24	24

coverpoints

Coverpoints	Bins	Hits	Misses	Goal	Coverage
Search...	Search...	Search...	Search...	Search...	Search...
ALL_DATA_PATTERNS	8	8	0	100	100%
ALL_KEY_PATTERNS	8	8	0	100	100%
hamming_weight_data	3	3	0	100	100%
hamming_weight_key	3	3	0	100	100%
Enable_Coverage	2	2	0	100	100%

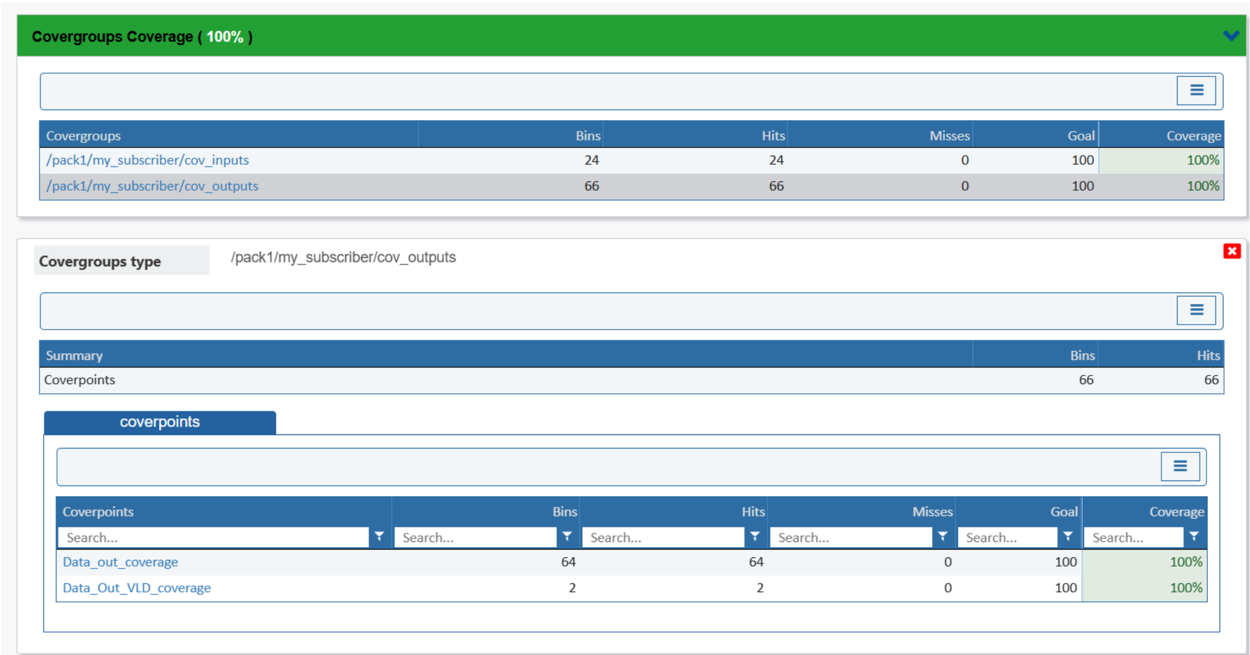
Figure 22: Functional Coverage For Input Signals

The cov_outputs covergroup checked the behavior of encryption results:

- **Data Output (Data_Out)**
  This coverpoint verified that the encrypted output changes in response to various input conditions and key variations.

- **Data Valid Flag (Data_Out_VLD)**
  Ensured that the output valid signal was asserted correctly after round 10.

Result: All bins for both Data_Out and Data_Out_VLD were hit, achieving **100% output coverage**.



Covergroups Coverage (100%)

Covergroups	Bins	Hits	Misses	Goal	Coverage
/pack1/my_subscriber/cov_inputs	24	24	0	100	100%
/pack1/my_subscriber/cov_outputs	66	66	0	100	100%

Covergroups type /pack1/my_subscriber/cov_outputs

Summary	Bins	Hits
Coverpoints	66	66

coverpoints

Coverpoints	Bins	Hits	Misses	Goal	Coverage
Search...	Search...	Search...	Search...	Search...	Search...
Data_out_coverage	64	64	0	100	100%
Data_Out_VLD_coverage	2	2	0	100	100%

Figure 23: Functional Coverage For Output Signals

Conclusion

The complete functional coverage ensures that all significant scenarios and edge cases related to AES encryption input and output behaviors have been thoroughly validated. This guarantees confidence in the functional correctness of the AES encryption module across a wide range of input conditions and key types.