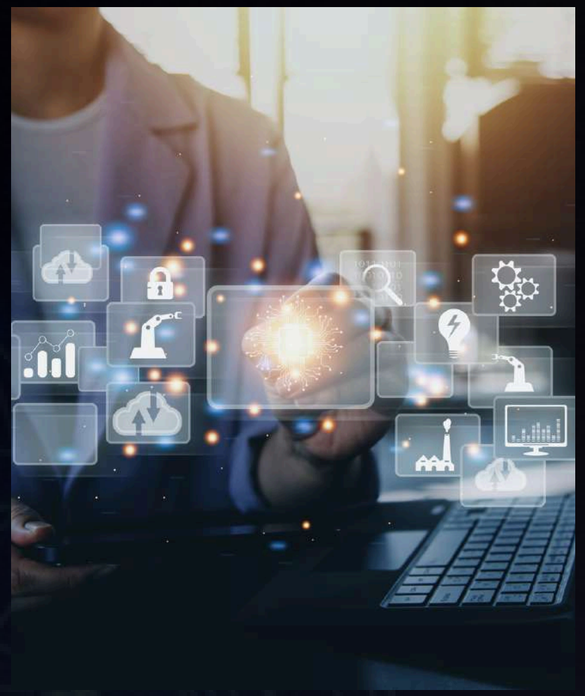
CRYPTOGRAPHIC

REPORT

Presented by
Nicolas, Fatih, Rafael, Yohana

TABLE OF CONTENTS



ABOUT THE PROJECT

BACKGROUND

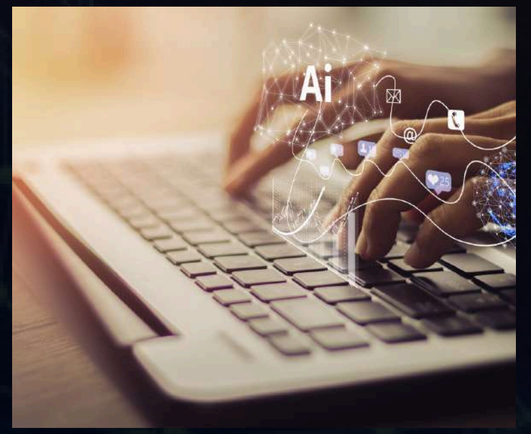
Cryptography is a category that aims at securing sensitive information such as passwords and identifications from unauthorized access that might use it for bad intentions. Cryptography is able to promote several key features which includes confidentiality, non-repudiation, integrity, adaptability, interoperability, and authentication.

One of the subcategories that contributes towards this category is AES. AES stands for Advanced Encryption Standard which has been an algorithm function that has widely been used for cryptography. The AES supports various lengths of inputs which are either 128, 192 or 256 bits.

PROJECT DESCRIPTION

This VHDL project implements a complete AES encryption core suitable for FPGA synthesis and RTL simulation.

The datapath is fully structural: key schedule, SubBytes (S-box), ShiftRows, MixColumns, and AddRoundKey are separate modules. A simple controller sequences rounds, exposes a clean streaming/bus interface, and a testbench verifies correctness with known answer vectors.



OBJECTIVES

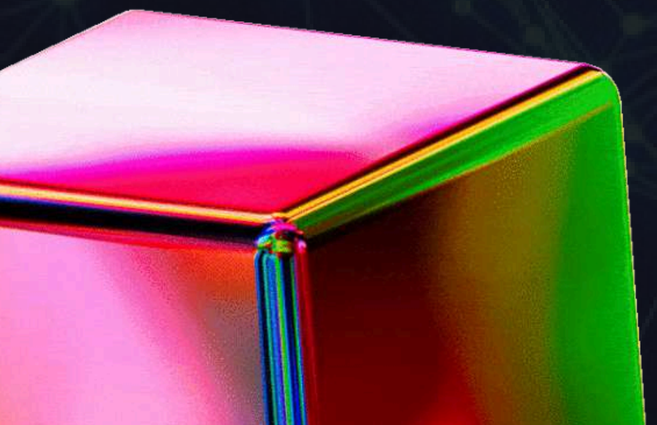
Primary Goal: To design and implement the complete AES-128 encryption algorithm

Platform: Utilizing the VHDL (VHSIC Hardware Description Language).

Skill Demonstration: To demonstrate a thorough understanding of hardware-based security algorithms.

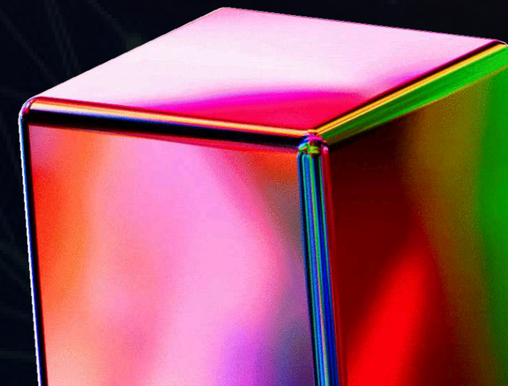
Deliverable: To produce an efficient AES-128 encryption core suitable for FPGA synthesis or RTL simulation.

Specific Focus: Implementing the core AES-128 functions, including the Key Schedule, SubBytes, ShiftRows, MixColumns, and AddRoundKey.



TOOLS

- VIVADO
- GITHUB, GIT
- GOOGLE DOCS
- CANVA



IMPLEMENTATION 01

SUBBYTES (256-ENTRY CONSTANT LOOKUP)

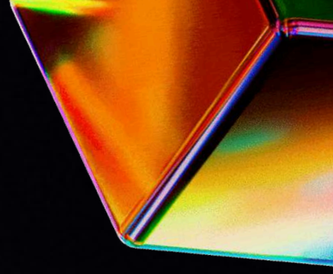
```
begin

    process(state_in)
        variable byte_val : std_logic_vector(7 downto 0);
        variable index : integer;
    begin
        for i in 0 to 15 loop
            byte_val := state_in(8*i+7 downto 8*i);
            index := to_integer(unsigned(byte_val));
            state_out(8*i+7 downto 8*i) <= SBOX(index);
        end loop;
    end process;

end Behavioral;
```

- The sbox_comb module implements the SubBytes transformation (AES step).
- It provides Confusion by performing a non-linear substitution on every byte.
- A constant lookup table named SBOX (256 entries) is defined, containing the pre-calculated substitution values.
- Operation: The VHDL code iterates 16 times, using each 8-bit input byte from state_in as an index to fetch the corresponding 8-bit substituted value from the SBOX.
- The module is combinational (acts as a direct lookup, driven by a process(state_in)).
- Ports:

1. state_in: 128-bit data state.
2. state_out: 128-bit substituted data state.

IMPLEMENTATION 02

SHIFTROWS COMBINATIONAL



```
architecture Behavioral of shiftrows is
begin
    -- Row 0: no shift
    state_out(127 downto 120) <= state_in(127 downto 120);
    state_out(95  downto 88 ) <= state_in(95  downto 88 );
    state_out(63  downto 56 ) <= state_in(63  downto 56 );
    state_out(31  downto 24 ) <= state_in(31  downto 24 );

    -- Row 1: shift left by 1
    state_out(119 downto 112) <= state_in(87  downto 80);
    state_out(87  downto 80 ) <= state_in(55  downto 48);
    state_out(55  downto 48 ) <= state_in(23  downto 16);
    state_out(23  downto 16 ) <= state_in(119 downto 112);

    -- Row 2: shift left by 2
    state_out(111 downto 104) <= state_in(47  downto 40);
    state_out(79  downto 72 ) <= state_in(15  downto 8 );
    state_out(47  downto 40 ) <= state_in(111 downto 104);
    state_out(15  downto 8  ) <= state_in(79  downto 72 );

    -- Row 3: shift left by 3
    state_out(103 downto 96 ) <= state_in(7   downto 0);
    state_out(71  downto 64 ) <= state_in(103 downto 96);
    state_out(39  downto 32 ) <= state_in(71  downto 64 );
    state_out(7   downto 0  ) <= state_in(39  downto 32 );
end Behavioral;
```

- The shiftrows module performs a byte-level cyclical shift on the data within the 128-bit state. This is a crucial step for Diffusion in the AES algorithm.
- The 128-bit state_in is conceptually treated as a 4×4 matrix of bytes (four rows). Each row shifts a different amount:

1. Row 0: Shift left by 0 bytes (no change).
2. Row 1: Shift left by 1 byte (8 bits).
3. Row 2: Shift left by 2 bytes (16 bits).
4. Row 3: Shift left by 3 bytes (24 bits).

The operation is performed by assigning specific ranges of the input vector (state_in) to new ranges in the output vector (state_out), as seen in the VHDL code.

- This module is combinational (no clock required) and accepts one 128-bit input:

1. state_in: The 128-bit state from the previous round (e.g., from SubBytes or MixColumns).
2. state_out: The resulting 128-bit state after the cyclical shifts.

IMPLEMENTATION 03

MIXCOLUMNS (USES GF MUL FUNCTIONS)

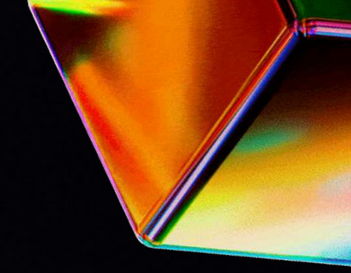
```
process(stateIn, mul2_out, mul3_out)
begin
    for column in 0 to 3 loop
        stateOut(0, column) <= mul2_out(0, column) xor
                               mul3_out(1, column) xor
                               stateIn(2, column) xor
                               stateIn(3, column);

        stateOut(1, column) <= stateIn(0, column) xor
                               mul2_out(1, column) xor
                               mul3_out(2, column) xor
                               stateIn(3, column);

        stateOut(2, column) <= stateIn(0, column) xor
                               stateIn(1, column) xor
                               mul2_out(2, column) xor
                               mul3_out(3, column);

        stateOut(3, column) <= mul3_out(0, column) xor
                               stateIn(1, column) xor
                               stateIn(2, column) xor
                               mul2_out(3, column);

    end loop;
end process;
```

This module, mixcolumns, implements the third step of the AES round function. It provides the final layer of Diffusion by creating a heavy dependency between the four bytes within each column.

- Function: The operation treats each 4-byte column of the state matrix as a vector and multiplies it by a fixed, known, and invertible polynomial matrix over the Galois Field GF(2^8)
- Mathematical Basis: This transformation is equivalent to polynomial multiplication modul x^4 + 1 with the fixed polynomial a(x) = (03)x^3 + (01)x^2 + (01)x + (02)
- Hardware Implementation: The VHDL uses a simpler approach for hardware: it relies on a separate utils component to pre-calculate the complex GF(2^8) multiplications by the constants] 2 and 3 for every byte.

IMPLEMENTATION 04

XOR ROUND KEY ONTO STATE



```
architecture Behavioral of addroundkey is
begin
    process(state_in, round_key)
    begin
        state_out <= state_in xor round_key;
    end process;
end Behavioral;
```

- The AddRoundKey.vhd module performs a direct 128-bit XOR operation on both the "state input" and the "round key."
- The XOR operation creates an amalgamation of both key material and current intermediate encrypted data as a function of bit-wise combination; therefore, it also includes elements of Diffusion and Confusion.
- This module is combinational (no clock required) and accepts two 128-bit inputs:
  - state_in: The state from MixColumns (or ShiftRows in the final round)
  - round_key: The corresponding round key from the Key Expansion module

IMPLEMENTATION 05

KEY SCHEDULE (GENERATES ROUND KEYS)

```
-- Round 0: initial key
round_keys(0) <= key_in;

-- Split into words
w0 := key_in(127 downto 96);
w1 := key_in(95 downto 64);
w2 := key_in(63 downto 32);
w3 := key_in(31 downto 0);

-- Generate rounds 1-10
for round in 1 to 10 loop
    -- Apply g() function to previous round's last word
    temp := RotWord(w3);
    temp := SubWord(temp);
    temp(31 downto 24) := temp(31 downto 24) xor RCON(round);

    -- Generate new words
    w0 := w0 xor temp;
    w1 := w1 xor w0;
    w2 := w2 xor w1;
    w3 := w3 xor w2;

    -- Store round key
    round_keys(round) <= w0 & w1 & w2 & w3;
end loop;
```

- The module sequentially generates 11 round keys from a 128-bit initial AES key using a clock-driven process.
- It starts by storing the input key as Round Key 0 and splits it into four 32-bit words (w0–w3).
- For each round (1–10), the last word (w3) undergoes RotWord, SubWord, and XOR with Rcon, then is XORed with w0 to form the first word of the new round key.
- The remaining three words are generated through successive XOR operations with the previous words, forming a complete 128-bit round key stored in an internal array.
- A done signal is raised upon completion, and any round key can be accessed via the round_num input for use in the corresponding encryption round.

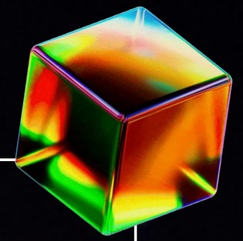
IMPLEMENTATION 06

GF(2) HELPERS (MUL2, MUL3), CONVERSIONS

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity utils is
  Port (
    byte_in  : in std_logic_vector(7 downto 0);
    mul2_out : out std_logic_vector(7 downto 0);
    mul3_out : out std_logic_vector(7 downto 0)
  );
end utils;

architecture Behavioral of utils is
    function mul2(x : std_logic_vector(7 downto 0)) return std_logic_vector is
        variable byte_result : std_logic_vector(7 downto 0);
    begin
        byte_result(7) := x(6);
        byte_result(6) := x(5);
        byte_result(5) := x(4);
        byte_result(4) := x(3) xor x(7);
        byte_result(3) := x(2) xor x(7);
        byte_result(2) := x(1);
        byte_result(1) := x(0) xor x(7);
        byte_result(0) := x(7);
        return byte_result;
    end function;
begin
    mul2_out <= mul2(byte_in);
    mul3_out <= mul2(byte_in) xor byte_in;
end Behavioral;
```

- The utils module implements the helper functions for MixColumns (multiplication by 2 and 3).
- It performs arithmetic over the Galois Field GF(2^8, where addition is the XOR operation.
- Multiplication by 2 is implemented as a left shift followed by a conditional reduction using the irreducible polynomial.
- Operation: Multiplication by 3 is calculated simply as (2 * x) XOR (1 * x) , which in VHDL is mul2_out xor byte_in.
- The module is combinational (the output is immediately determined by the input).
- Ports:
- byte_in: 8-bit input byte from the state.
- mul2_out: 8-bit result of input {02\}.
- mul3_out: 8-bit result of input {03\}

TESTING

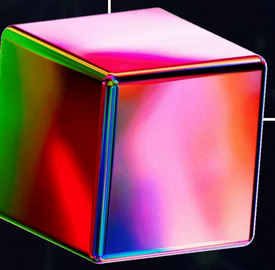


- Correctness of both AES Modules (AddRoundKey and Key Expansion) was confirmed through the testing phase. The simulation of the outputs in Modelsim/QuestaSim confirmed correctness relative to the AES-128 (FIPS-197) Standard.

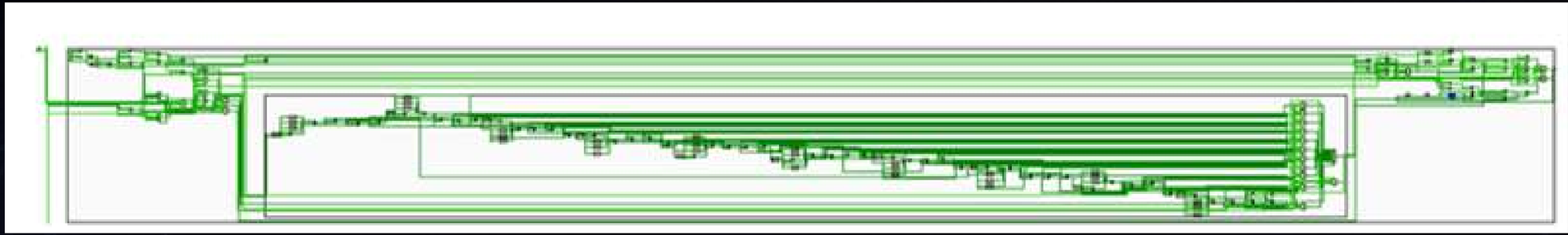
- During initial tests, the results of the encryption were inconsistent, indicating that there was a problem with how an internal transformation was implemented. Therefore, debug reports were utilized to record the state of data after each step in order to identify the source of the error.

- The debugging process identified the cause of the error as being in the ShiftRows Process. The way that data was being shifted was not row-wise (as written) but was column-wise, which caused the misalignment of the bytes. The implementation of this correction resolved this issue.

- Following those corrections, the simulation of the AES interpretation matched that of the AES Reference Implementation.



RESULTS

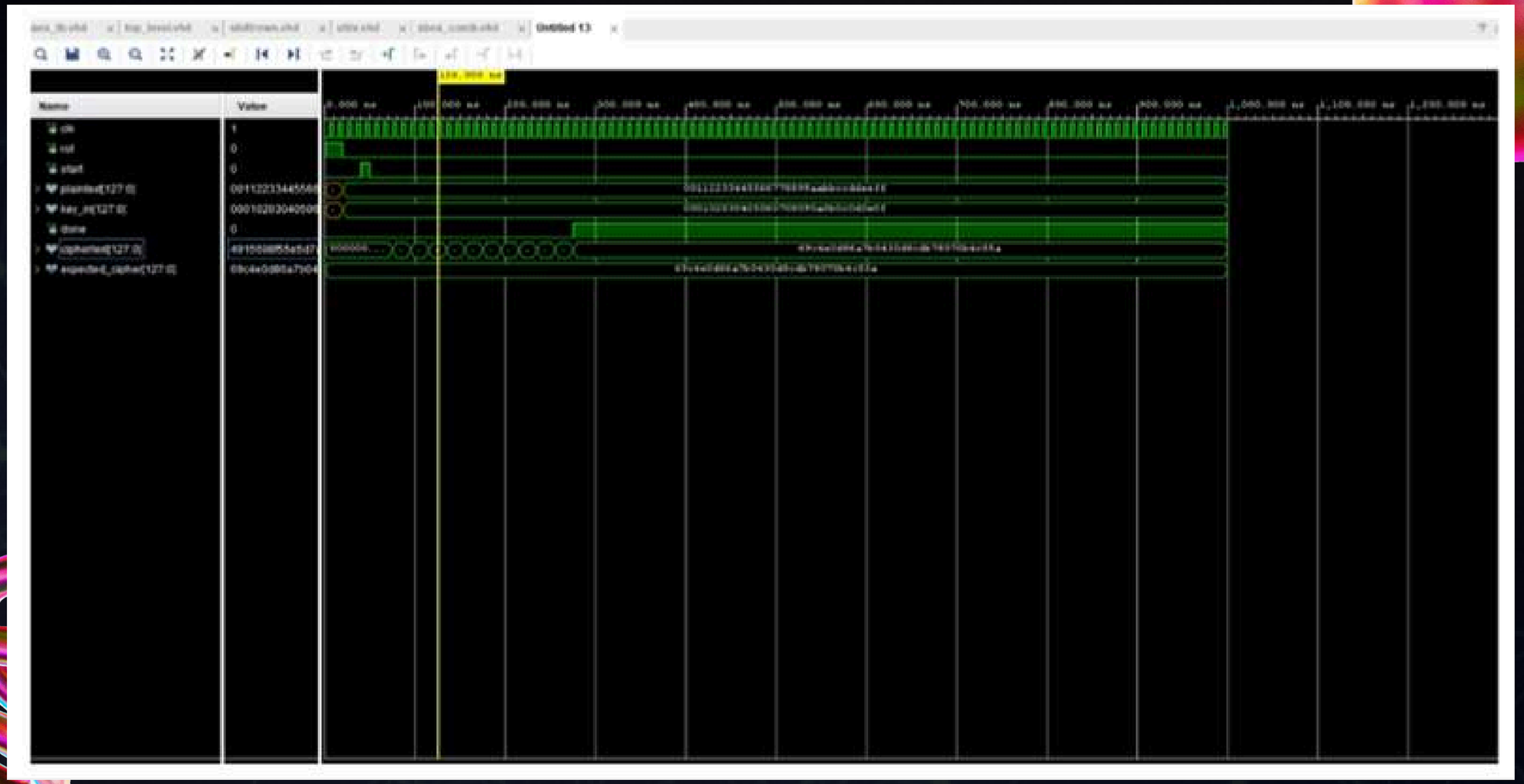


Project Schematic

- After the correction to the ShiftRows operation, a complete re-simulation of the AES modules is re-run. All round outputs now provide a match with the official AES-128 test vector documents, thus confirming that this system has been verified against their design criteria.

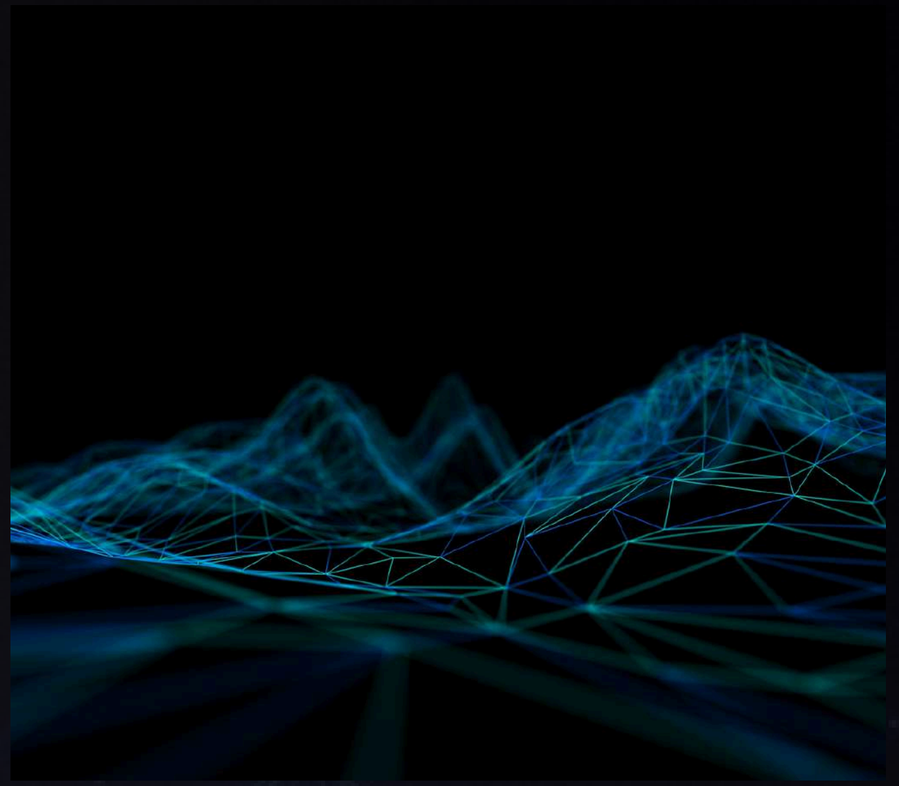
-This also provided verification of the Testability and Debug Capability of the new system; the verified AES-128 core will reliably produce valid ciphertext output for all input data.

RESULTS



ANALYSIS

From the results it can be observed that the AES encryption process performs correctly across all stages. The simulation clearly displays 10 full rounds of AES-128 encryption, beginning with the initial AddRoundKey (Round 0) and followed by Rounds 1 through 10. Each round includes the expected sequence of transformations: SubBytes, ShiftRows, MixColumns (Rounds 1–9), and AddRoundKey.



The progression of the internal state in the waveform demonstrates that the round keys generated by the Key Expansion module are applied at the correct clock cycles and in the correct order. No timing violations or unexpected intermediate values are present after the correction of the ShiftRows implementation.

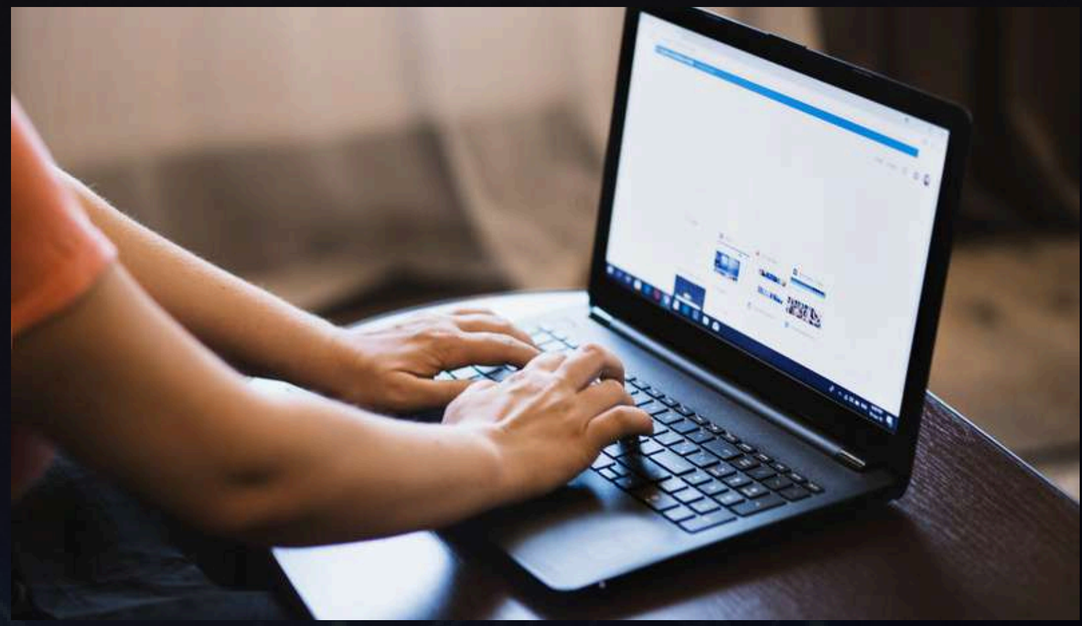
In the final stage of the simulation, the module outputs a 128-bit ciphertext that exactly matches the official AES reference output for the provided plaintext and key. Therefore, the correctness of the final ciphertext directly reflects the correctness of the AES core itself. The analysis confirms that the design is stable, logically correct, and fully compliant with AES-128 ECB encryption behavior.

RESULTS

```
# run 1000ns
Note: AES-128 TEST PASSED! Cipher = 69C4E0D86A7B0430D8CDB78070B4C55A
Time: 295 ns  Iteration: 0  Process: /aes_tb/stim_proc  File: C:/Users/Rafael/Downloads/AND_GATE.vhdl
INFO: [USF-XSim-96] XSim completed. Design snapshot 'aes_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:05 ; elapsed = 00:00:11 . Memory (MB): peak = 1075.449 ; gain = 52.090
```

CONCLUSION

- The project delivers a standard-compliant, testable, and functional AES-128 encryption core in VHDL.
- It fulfills all course requirements and provides a strong foundation for understanding hardware-based cryptography.
- The process demonstrated essential digital design skills: modular implementation, systematic debugging, and standards-based verification.
- Final Validation: Post-correction simulations confirmed a perfect match with all reference test vectors.



THANK YOU