# CRYPTOGRAPHIC PROCESSOR

Using: BASYS3, VHDL

## STUDENTS

Alexandru- Gabriel Brabete
Raul Bria
Thomas Botizan

# CONTAINED

1. Introduction.....3
2. AES (Advanced Encryption Standard) Algorithm Overview.....3
   2.1. History and standardization.....3
   2.2. Substitution-permutation network (SPN) model.....3
   2.3. Internal structure and number of rounds.....4
   2.4. Key Schedule Expansion.....4
   2.5. Modes of Operation.....4
   2.6. Security and performance.....5
3. FPGA board used.....5
4. Development environment: Vivado 2024.....5
5. Block diagram.....6
   5.1. Use of the Galois body.....7
   5.2. Synchronization entries and registers.....7
   5.3. Adding the initial key.....8
   5.4. AES Round Cycle.....8
   5.5. Key Schedule Generation.....8
   5.6. Output and End Signal.....8
6. Detailed explanation of VHDL files for AES co-processor.....9
   6.1. Round.vhd.....9
      - 6.1.1. Entity.....9
      - 6.1.2. Internal signals and code comments.....10
      - 6.1.3. Architecture with comments.....10

6.2. SubBytes.vhd and SBox.vhd....13

- 6.2.1. SubBytes: Entity and Byte Generation....13
- 6.2.2. Generation and reassembly....13
- 6.2.3. SBox: ROM and Sample Metrics....14

6.3. ShiftRows.vhd (Indexing Details)....15

6.4. GFMult.vhd (algorithm and comments)....17

6.5. MixColumns.vhd (Detailed Implementation)....18

6.6. AddRoundKey.vhd....19

6.7. Key_Expansion.vhd (fully commented)....20

6.8. AES_Encryptor.vhd (FSM and Synchronization)....23

6.9. enc_tb.vhd (Full Testbench)....23

- 6.9.1. Clock generation and reset....23
- 6.9.2. Test vectors and observations....23
- 6.9.3. Test Scenario....24

7. On-board deployment....26

7.1. AES_Wrapper.vhd (AXI-Lite Interface)....26

7.2. driver7seg.vhd (multiplexing and timing)....26

7.3. How does display work?....27

7.4. What role does display data play?....28

7.5. Recap....28

8. Areas of application of the AES co-processor....29

9. General conclusions....30

10. Bibliography....31

# 1. INTRODUCTION

The present documentation aims to describe the general problem of the project for the implementation in VHDL of a cryptographic co-processor based on the AES algorithm. Next, fundamental notions regarding AES will be defined, the FPGA board on which the project will be carried out and the development environment used will be presented.

# 2. AES (ADVANCED ENCRYPTION STANDARD) ALGORITHM - OVERVIEW

## 2.1. History and standardisation

The Advanced Encryption Standard (AES), originally called Rijndael, was proposed by Joan Daemen and Vincent Rijmen and approved in 2001 by the National Institute of Standards and Technology (NIST) of the United States, through the FIPS-197 specification. AES replaced the Data Encryption Standard (DES), providing increased resistance to cryptanalytic attacks and superior performance. Subsequently, the standard was adopted internationally through ISO/IEC 18033-3.

## 2.2. Substitution-permutation network (SPN) model

AES is a symmetric block cipher that operates on fixed blocks of 128 bits (16 bytes), repeatedly applying four fundamental operations:

1. **SubBytes** – non-linear substitution of each byte using a predefined S-box, built for invertibility and resistance to linear and differential cryptanalysis attack properties.
2. **ShiftRows** – cyclic permutation of the rows of the state array, so that the bytes of each row are offset by a specific offset (0, 1, 2, (3 steps) respectively), contributing to data scattering.
3. **MixColumns** – linear mixing of the bytes of each column through a transformation defined in the Galois GF(2^8) body, providing additional diffusion at the column level.
4. **AddRoundKey** – bitwise XOR operation between the state array and the derived round key, integrating key control over the encryption process.

The repetition of these steps defines a substitution–permutation (SPN) network, characterized by strong confounding and diffusion properties.

## 2.3. Internal structure and number of rounds

The number of AES rounds depends on the length of the key used (round 0 is not always considered as a round):

- **AES-128**: 10 / 11 rounds
- **AES-192**: 12 / 13 rounds
- **AES-256**: 14 / 15 rounds

The first round consists only of the AddRoundKey operation, the intermediate rounds apply all four SPN operations, and the last round skips the MixColumns step to ensure symmetric compatibility between encryption and decryption.

## 2.4. Key Schedule

The key extension procedure generates a set of round keys based on the initial key, using successive operations of:

- **RotWord** – rotation of bytes in a 32-bit column;
- **SubWord** – applying the S-box for each byte;
- **Rcon** – adding a round constant specific to each step.

This method ensures the independence of each round wrench and increases the resistance to partial compromise of the wrench.

## 2.5. Modes of Operation

For encrypting data streams larger than block size, AES is used in various operating modes:

- **ECB (Electronic Codebook)** – each block is encrypted independently (simple, but vulnerable to repeated patterns);
- **CBC (Cipher Block Chaining)** – each block is combined with the block previously encrypted by XOR, increasing entropy
- **CTR (Counter Mode)** – uses an encrypted meter, allowing parallel processing and encryption of data streams;
- **GCM (Galois/Counter Mode)** – combines CTR with authentication (MAC) through operations in GF(2^128), providing confidentiality and integrity. (as is our case)

### 2.6. Security and performance

To date, the AES has not been compromised practically in the full version; Theoretical attacks (bikes, multiple text-pure attacks) have exponential complexity relative to the length of the key. Hardware implementations on FPGAs allow for throughput and latency optimization by:

- **Round-level pipeline** – parallel execution of transformations;
- **Partial or full unroll implementation** of the round for maximum throughput;
- **Intensive use of LUTs and internal multipliers** to accelerate GF operations(2^8).

These strategies will be detailed in the chapter on VHDL architecture.

# 3. FPGA BOARD USED

For the realization of the cryptographic co-processor, the following FPGA boards can be chosen:

- **Basys3 (Digilent):** based on the Artix-7 XC7A35T FPGA, it offers 33,280 LUT elements, 1,800 kbit memory blocks and integrated test interfaces (USB, buttons, LEDs). It is suitable for educational prototyping and experiments of moderate complexity.
- **Artix-7 CPG236 (Xilinx):** Similar generation FPGAs with increased resource availability for commercial deployments. Detailed specifications can be found in the Xilinx datasheet.

The Basys3 **board will be used in this project**, thanks to the support of the community and extensive documentation.

# 4. DEVELOPMENT ENVIRONMENT: VIVADO 2024

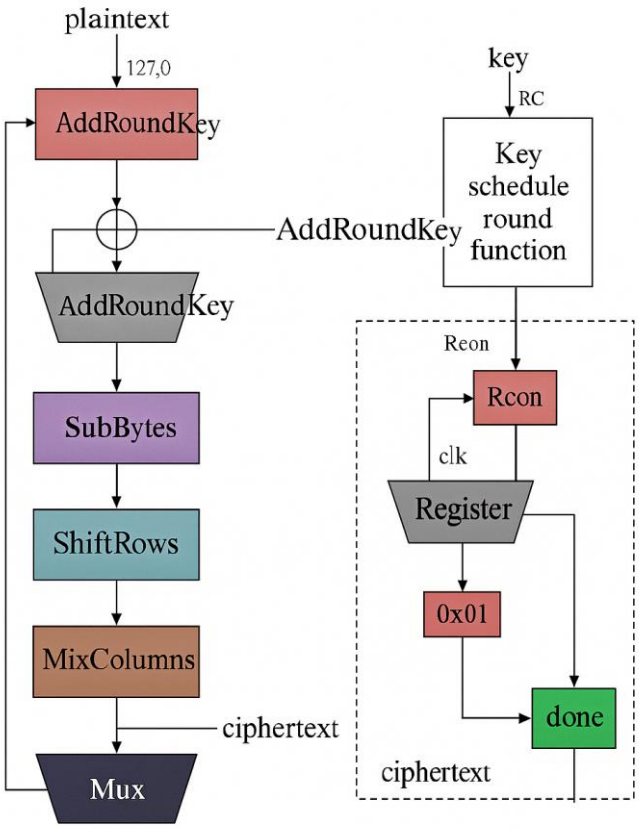
For the hardware description and synthesis of the project, **Xilinx Vivado Design Suite 2024** will be used. The main advantages of this version are:

- Modern graphical interface and register-transfer (RTA) level performance analysis tools.
- Full support for VHDL language (including VHDL-2008 syntax).
- Integration with predefined IPs (AES cores), automatic key block generation and resource usage reporting.

Vivado 2024 will allow the optimization of timing and resource consumption, adapting the circuit configuration for maximum performance.

In the following chapters, the VHDL architecture of the co-processor, how to integrate with the FPGA bus and the test and validation strategies will be detailed.

## 5. BLOCK SCHEME:




[1]Image 1: Block diagram

[1] Use of the Galois body: Galois GF(2) is used in the MixColumns module and in the x_time multiplication of GFMult

Legend:

- **Muxes**: dark gray
- **Register**: dark blue
- **AddRoundKey**: gray
- **SubBytes**: mov
- **ShiftRows**: Edge
- **MixColumns**: brown
- **Final Mux**: Black

## 5.1. Use of the Galois body

Galois GF(2^8) is used in:

1. **MixColumns mode**: the mixture coefficients (0x02 and 0x03) are multiplied by the bytes of the state in GF(2^8), ensuring vertical diffusion of the data.
2. **The xtime function in GFMult**: the polynomial doubling (multiplication by x) and the reduction of the generating polynomial $x^8 + x^4 + x^3 + x + 1$ is performed in GF(2^8).

These operations on the Galois body are essential for the security and diffusion properties of the AES.

## 5.2. Synchronization Inputs and Registers

- **Plaintext Mux (dark gray) & Register (dark blue)**
  - *Mux* selects between the reset signal (all 0 bits) and the 128-bit plaintext input.
  - *Register* captures this data at the rising front of clk.

- **Key Mux (dark grey) & Register (dark blue)**
  - It works identically for the 128-bit key, then feeding the subkey generation block.

## 5.3. Adding the initial key

- The status signal exits the plaintext register and enters an **AddRoundKey block (bordo),** where the XOR operation is performed with the first sub-key.

## 5.4. AES Round Cycle

- **SubBytes (purple):** each of the 16 bytes of the state passes through the S-box, performing a non-linear substitution.
- **ShiftRows (teal):** Bytes are circularly permuted on the rows of the 4×4 matrix (offsets of 0,1,2,3) for horizontal diffusion.
- **MixColumns (brown):** this is where **the Galois GF(2^8) body** comes in — each 4-byte column is multiplied by the constant array {02,03,01,01}. Multiplications by 0x02 and 0x03 are performed in GF(2^8) by the **GFMult module**, using the xtime function for polynomial doubling and XOR for reduction. The result is the vertical diffusion of the state.
- **Final Mux (black):** In the last round, it bypasses MixColumns (signal is_last) and goes directly to AddRoundKey, then exits as ciphertext.

## 5.5. Generating Subkeys (Key Schedule)

- **The Key Schedule Round Function** continuously produces the subkeys needed for each round.
- An internal control block (framed with dashes) contains:
  - **Rcon (bordo)** – round constants (0x01, 0x02, ..., 0x36).
  - **Register (dark blue)** and a **Mux (dark gray)** to iterate Rcon on each clock cycle.
- The current subkey outputs through the subkey signal and powers the AddRoundKey.

## 5.6. Output and End Signal

- After the last round, the final state resulting from the AddRoundKey is delivered as **ciphertext(C).**
- The control block compares the round index with Nr and, at the end, activates the **done signal (green),** along with the display of the last constant Rcon (0x36), to mark the end of the encryption.

# 6. DETAILED EXPLANATION OF VHDL FILES FOR AES CO-PROCESSOR

This documentation provides an in-depth and commented description of each VHDL module in the AES co-processor. Each section includes:

1. **Module interface**: entity (ports and generics)
2. **Internal signals** and data types used
3. **Architectural Blocks**: Processes and Instantiators with Code Comments
4. **Signal flow** and connectivity between components
5. **Algorithmic functionality**: step-by-step explanations and examples of intermediate values

## 6.1. Round.vhd

### 6.1.1. Entity

```
entity Round is
 port(
   clk : in std_logic;                    -- Main Clock
   rst_n : in std_logic;                  -- Active asynchronous reset to 0
   state_in : in std_logic_vector(127 downto 0);      -- Input state (4x4 16-byte array)
   round_key : in std_logic_vector(127 downto 0);     -- Sub-key of the current round
   is_last : in std_logic;                -- '1' for the last round (skip MixColumns)
   state_out : out std_logic_vector(127 downto 0) -- Updated status output
 );
end entity;
```

- **state_in** and **round_key** are mapped as linear vectors, but represent 4×4 matrices.
- **is_last** controls a combinatorial process for omitting MixColumns.

### 6.1.2. Internal signals and code comments

*signal sb_state: std_logic_vector(127 downto 0);  -- after SubBytes*

*signal sr_state: std_logic_vector(127 downto 0);  -- after ShiftRows*

*signal mc_state: std_logic_vector(127 downto 0);  -- after MixColumns (or directly sr_state if is_last)*

*signal addrk_state: std_logic_vector(127 downto 0);  -- after AddRoundKey*

- **sb_state**: Each byte replaced by the S-box.
- **sr_state**: Rearrange bytes on rows.
- **mc_state**: Linear mixing on columns.
- **addrk_state**: XOR of mc_state with round_key.

### 6.1.3. Architecture with comments

*architecture RTL of Round is*

***-- Declare external components***

*SubBytes port(...) component; end component;*

*component ShiftRows port(...); end component;*

*MixColumns component port(...); end component;*

*AddRoundKey component port(...); end component;*

*begin*

```
-- 1. Apply SubBytes
u_subbytes: SubBytes port map(
 data_in => state_in,
 data_out => sb_state
);

-- 2. Apply ShiftRows
u_shiftrows: ShiftRows port map(
  data_in => sb_state,
  data_out => sr_state
);

-- 3. MixColumns conditional
mix_proc: process(sr_state, is_last)
begin
  if is_last = '1' then
    mc_state <= sr_state;  -- last round skip MixColumns
  else
    u_mix: MixColumns port map(
      data_in => sr_state,
      data_out => mc_state
    );
  end if;
end process;
```

```
-- 4. AddRoundKey
u_addrk: AddRoundKey port map(
  state_in => mc_state,
  round_key => round_key,
  state_out => addrk_state
);

-- 5. Synchronous Output Recording
sync_proc: process(clk, rst_n)
begin
  if rst_n = '0' then
    state_out <= (others => '0');  -- Initialize at Reset
  elsif rising_edge(clk) then
    state_out <= addrk_state;
  end if;
 end process;
end architecture;
```

- **Comment**: In architecture, pure combinatorial components are separated from the synchronous process of writing to the register.

## 6.2. SubBytes.vhd and SBox.vhd

### 6.2.1. SubBytes: Entity and Byte Generation

```
entity SubBytes is
 port(
  data_in : in std_logic_vector(127 downto 0);
  data_out : out std_logic_vector(127 downto 0)
 );
end SubBytes;
```

- Decompose data_in into 16 bytes using a for generator.
- Instantiate 16 SBoxes in parallel.

### 6.2.2. Generation and reassembly

```
gen_bytes: for i in 0 to 15 generate
 -- Select byte i (MSB to LSB)
 bytes_in(i) <= data_in(127 - 8*i downto 120 - 8*i);

 -- Byte-level substitution
 u_sbox: SBox port map(
  a => bytes_in(i),
  y => bytes_out(i)
 );
```

```
  -- Reassembly in the output vector
  data_out(127 - 8*i downto 120 - 8*i) <= bytes_out(i);
 generated ends;
```

- **MSB** mapping: i=0 → the byte at position 127..120.

### 6.2.3. SBox: ROM and sample values

```
entity SBox is
 port(
  a : in std_logic_vector(7 downto 0);
  y: out std_logic_vector(7 downto 0)
 );
end SBox;
architecture RTL of SBox is
 -- Substitution table (first 8 values)
 constant TABLE: array(0 to 255) of std_logic_vector(7 downto 0) := (
  x"63", -- 0x00 → 0x63
  x"7c", -- 0x01 → 0x7C
  x"77", -- 0x02 → 0x77
  x"7b", -- 0x03 → 0x7B
  x"f2", -- 0x04 → 0xF2
  x"6b", -- 0x05 → 0x6B
```

```
    x"6f", -- 0x06 → 0x6F
    x"c5", -- 0x07 → 0xC5
    -- ... up to 0xFF
    x"16" -- 0xFF → 0x16
  );
begin
  -- Index conversion and ROM access
  and <= TABLE(to_integer(unsigned(a)));
RTL end;
```

- **Comment**: ROM synthesized with LUTs; latency=LUT depth.

## 6.3. ShiftRows.vhd (indexing details)

```
architecture Dataflow of ShiftRows is
  type state_array is array(0 to 3, 0 to 3) of std_logic_vector(7 downto 0);
  signal s_in, s_out: state_array;
begin
  -- Linear mapping → 4×4 matrix
  for r in 0 to 3 generate
    for c in 0 to 3 generate
      s_in(r,c) <= data_in(127 - 8*(4*r + c) downto 120 - 8*(4*r + c));
```

```
  generated ends;
 generated ends;

 -- Movements in rows:
 s_out(0,0) <= s_in(0,0);
 s_out(0.1) <= s_in(0.1);
 -- ... identical for row 0

 -- Row 1 (offset 1):
 for c in 0 to 3 generate
   s_out(1,c) <= s_in(1,(c+1) mod 4);
 generated ends;

 -- Row 2 (offset 2), Row 3 (offset 3)

 -- Reassembly in vector:
 for r in 0 to 3 generate
   for c in 0 to 3 generate
     data_out(127 - 8*(4*r + c) downto 120 - 8*(4*r + c)) <= s_out(r,c);
   generated ends;
 generated ends;
end Dataflow;
```

- **Variable offset** used with the mode operator.

## 6.4. GFMult.vhd (algorithm and comments)

```
architecture RTL of GFMult is
 -- Polynomial doubling x·a in GF(2^8)
 function xtime(x: std_logic_vector(7 downto 0)) return std_logic_vector is
 begin
  return (x(6 downto 0) & '0') xor
        ("00011011" and ("00000001" & x(7 downto 1)));  -- polynomial 0x1B
 End;
begin
 process(a, b)
  variable p : std_logic_vector(7 downto 0) := (others => '0');
  variable temp : std_logic_vector(7 downto 0) := a;
 begin
  -- Loop on the bits of b for polynomial multiplication
  for i in 0 to 7 loop
   if b(i) = '1' then
    p := p xor temp;            -- conditionally add polynomial x^i
   end if;
   temp := xtime(temp);         -- double temp → x^(i+1)
  end loop;
  product <= p;                 -- the final result
 end process;
RTL end;
```

- **Comment**: xtime applies polynomial reduction after shift.

## 6.5. MixColumns.vhd (Detailed Implementation)

```
architecture RTL of MixColumns is
 component GFMult port(
  a : in std_logic_vector(7 downto 0);
  b : in std_logic_vector(7 downto 0);
  product : out std_logic_vector(7 downto 0)
 ); end component;

 type col_t is array(0 to 3) of std_logic_vector(7 downto 0);
 Signal S: col_t;
 R signal: col_t;
begin
 -- Extract column i
 s(0) <= data_in(127 downto 120);
 s(1) <= data_in(119 downto 112);
 s(2) <= data_in(111 downto 104);
 s(3) <= data_in(103 downto 96);

 -- Required multiplications:
 -- r0 = 2·s0 ⊕ 3·s1 ⊕ s2 ⊕ s3
```

```
u_m0_2: GFMult port map(a => x"02", b => s(0), product => open);

u_m0_3: GFMult port map(a => x"03", b => s(1), product => open);

-- ... instantiators for each coefficient and s(2), s(3)


mix_proc: process(all)

begin

  r(0) <= GFMult_02_s0 xor GFMult_03_s1 xor s(2) xor s(3);

  -- r(1), r(2), r(3) similar

end process;


-- Reassembly:

data_out(127 downto 120) <= r(0);

-- ... the other bytes

RTL end;
```

- **Comment**: For each column, 8 GFMult products are generated simultaneously, followed by XOR.

## 6.6. AddRoundKey.vhd

```
architecture Dataflow of AddRoundKey is

begin

  -- Bitwise XOR between the current state and the round key

  state_out <= state_in round_key;
```

```
end Dataflow;
```

- Note: Pipeline as synchronized signal for higher throughput.

## 6.7. Key_Expansion.vhd (fully commented)

```
entity Key_Expansion is
 generic(
   Nk : integer := 4;  -- number of words in the initial key (4 for 128 bits)
   Nb : integer := 4;  -- words in the status block
   No. : integer := 10 -- number of rounds for AES-128
 );
 port(
   clk : in std_logic;
   rst_n : in std_logic;
   cipher_key : in std_logic_vector(32*Nk-1 downto 0);
   round_index : in integer range 0 to Nr;
   round_key_out : out std_logic_vector(32*Nb-1 downto 0)
 );
End;
architecture RTL of Key_Expansion is
 -- Memory for all extended words (4*(Nr+1))
 type w_array is array(0 to Nb*(Nr+1)-1) of std_logic_vector(31 downto 0);
 W signal: w_array;
```

```
-- Round constant (Rcon), first values:
constant Rcon: array(1 to 10) of std_logic_vector(31 downto 0) := (
  x"01000000", -- for i=1
  x"02000000", -- i=2
  x"04000000", -- i=3
  -- ... up to x"36000000"
);

-- RotWord and SubWord
function RotWord(word : std_logic_vector(31 downto 0)) return std_logic_vector is
begin
  return word(23 downto 0) & word(31 downto 24);
End;
function SubWord(word : std_logic_vector(31 downto 0)) return std_logic_vector is
begin
  -- Apply SBox to each byte
  return TABLE(to_integer(unsigned(word(31 downto 24)))) &
         TABLE(to_integer(unsigned(word(23 downto 16)))) &
         TABLE(to_integer(unsigned(word(15 downto 8)))) &
         TABLE(to_integer(unsigned(word(7 downto 0))));
End;
begin
 init_proc: process(clk, rst_n)
```

```
begin
  if rst_n = '0' then
    -- Load the initial key in the first Nk words
    for i in 0 to Nk-1 loop
      w(i) <= cipher_key(32*(Nk-1-i)+31 downto 32*(Nk-1-i));
    end loop;
  elsif rising_edge(clk) then
    for i in Nk to Nb*(Nr+1)-1 loop
      if (i mod Nk) = 0 then
        w(i) <= SubWord(RotWord(w(i-1))) xor w(i-Nk) xor Rcon(i/Nk);
      else
        w(i) <= w(i-1) xor w(i-Nk);
      end if;
    end loop;
  end if;
end process;

-- Extract the current sub-key
round_key_out <= w(32*round_index+31 downto 32*round_index);
RTL end;
```

- **Comment**: The expansion is on two levels: RotWord+SubWord+Rcon and simple XOR.

## 6.8. AES_Encryptor.vhd (FSM and Synchronization)

| State | Action | Transition |
|---|---|---|
| **IDLE** | Wait for start = '1' and rst_n='1' | → KEYEXP |
| **KEYEXP** | Round keys are prepared by default by Key_Expansion (index=0) | → INIT |
| **INIT** | st <= plain_in xor first_round_key; round_cnt<=1 | → ROUND |
| **ROUND** | Call Round with is_last='0' and round_cnt < No; increases round_cnt | If round_cnt = No. → FINAL otherwise → ROUND |
| **FINAL** | Call Round with is_last='1' | → DONE |
| **DONATED** | Put cipher_out <= st; done<='1'; Cleans round_cnt | → IDLE |

Table 1: WSF Details

- **Comment**: FSM implemented in two processes (sync/comb separation) for clarity.

## 6.9. enc_tb.vhd (Full Testbench)

### 6.9.1. Clock Generation and Reset

*clk_tb <= '0'; wait for 5 ns;*

*clk_tb <= '1'; wait for 5 ns;*

*-- Reset*

*rst_tb <= '0'; wait for 20 ns; rst_tb <= '1';*

### 6.9.2. Test Vectors and Observations

*plain_tb <= x"00112233445566778899aabbccddeeff";*

*key_tb <= x"000102030405060708090a0b0c0d0e0f";*

```
cipher_exp <= x"69c4e0d86a7b0430d8cdb78070b4c55a";
```

- **Comment**: The vectors come from the FIPS-197 benchmark tests.

### 6.9.3. Test Scenario

```
stim_proc: process
begin
  wait until rising_edge(clk_tb) and rst_tb = '1';
  start_tb <= '1';
  wait until rising_edge(clk_tb);
  start_tb <= '0';

  -- Wait for completion and check every bit
  wait until done_tb = '1';
  assert cipher_out_tb = cipher_exp
    report "AES-128 Test Vector Failure" severity error;
  report "Test Vector AES-128 OK" severity note;
  wait;
end process;
```

- **Includes success report** for clarity.

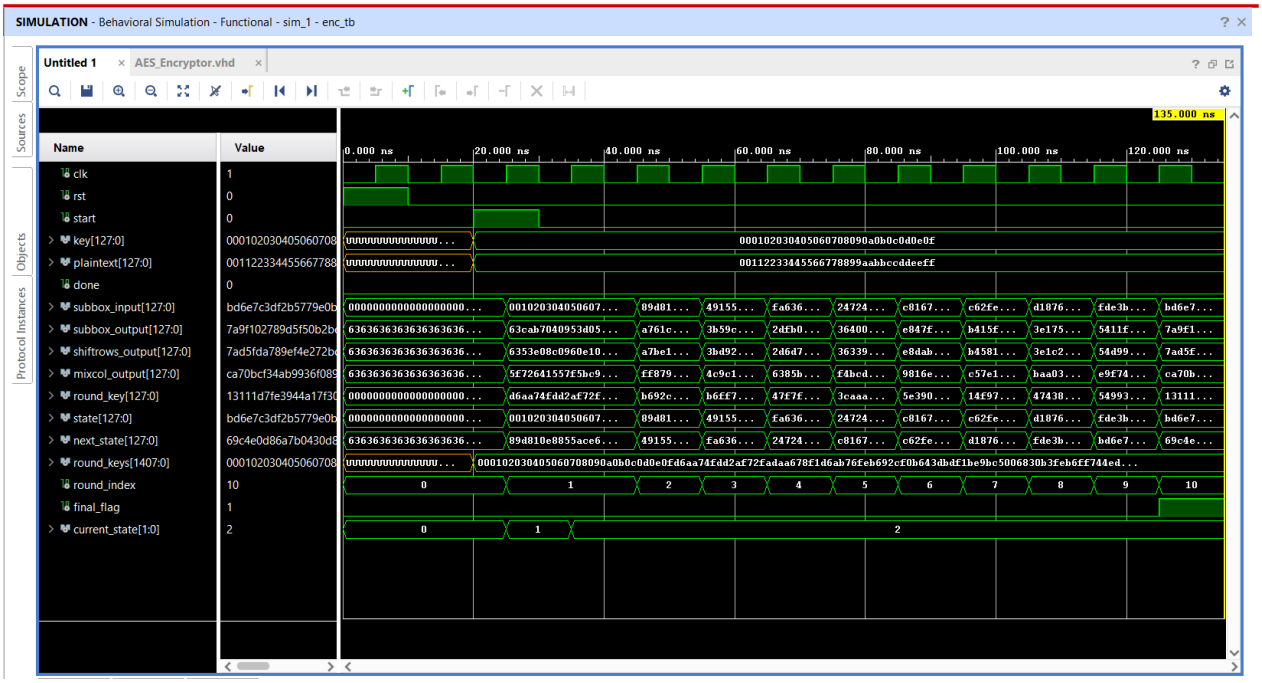



Image 2: Simulation Chart

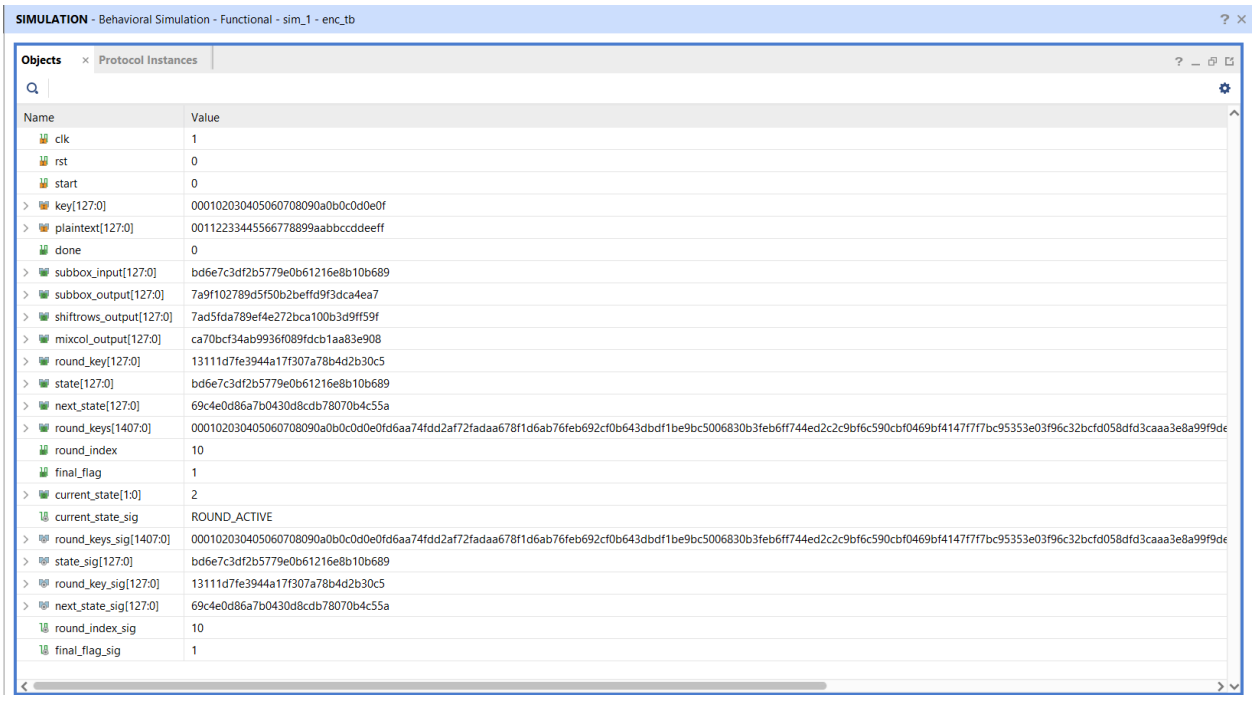



Image 3: Simulation results table

Through the additional comments and examples of values, we have detailed each internal step of the VHDL code: from the initial byte mapping, access to the S-box, the multiplication algorithm in GF(2^8), to the organization of the FSM and the AXI interface.

# 7. IMPLEMENTATION ON THE BOARD

## 7.1. AES_Wrapper.vhd (AXI-Lite Interface)

- **Register**:
  - [0x00] Plaintext[31:0]
  - [0x04] Plaintext[63:32]
  - [0x08] Plaintext[95:64]
  - [0x0C] Plaintext[127:96]
  - [0x10] Key[31:0]
  - ... up to [0x1C]
  - [0x20] Control (bit 0 = start)
  - [0x24] Status (bit 0 = done)
  - [0x28] Cipher[31:0]
  - ... etc.

- **AXI Handshake**:
  - AW/W for writing input registers.
  - B for confirmation.
  - AR/R for reading state and output registers.

- **Flow**:
  1. The CPU writes the 8 words of plaintext and key.
  2. Writing to the Control registry sets internal start.
  3. At the end (done → 1), the wrapper populates the Cipher and Status register.

## 7.2. driver7seg.vhd (multiplexing and timing)

- **Signals**:
  - digit: The current digit (4 bits).
  - Anodes: Select the active display from 4.
  - Segments: LED segment A–G.

- DP: decimal point.

- **Function**:
  1. Internal cyclic timer based on clk for multiplexing (approx. 1 kHz each anode).
  2. Digit conversion → segments by combinatorial lookup.

## 7.3. How does display work?

### 1. AES_Encryptor produces next_state

- next_state is a 128-bit vector (std_logic_vector(127 downto 0)).
- It is the result of AES-128 encryption.

### 2. The scrolling process

A VHDL process controls the display in 4-bit steps:

- Every **5 seconds**, the tick_5s signal becomes 1.
- Take 4 bits from the next_state, starting at bit_index.
- Each bit is converted to "0" or "1" to be displayed.
- Then bit_index is increased by 4.

### 3. Convert Bit to Digit

Each bit is displayed as follows:

- 0 → shown as binary digit 0000
- 1 → shown as binary number 0001

Thus, you will only see the digits 0 and 1 on the display, but depending on their position in the group of 4.

### Concrete example

- Suppose the first 4 bits of the next_state are:bit127 = 0, bit126 = 1, bit125 = 1, bit124 = 0→ group is 0110
- On the 4-digit display we will see:

| Digit | Bit index |
|---|---|
| DIG3 | 127 |
| DIG2 | 126 |
| DIG1 | 125 |
| DIG0 | 124 |

**Full Show**

- The system will display a total of 32 such groups (128 / 4 = 32).
- Each group appears for 5 seconds.
- After the last one (bits 3..0), the system goes back to the beginning (127..124).

**Reset (btnC)**

- If you click on btnC (which gives rst = '1'):
  - bit_index is reset to 0
  - The 5-second timer is reset
  - The display starts again at bit 127

## 7.4. What role does display_data play?

In code: display_data(3 downto 0) <= "0001" when bit = '1' else "0000";

Construct the final value sent to the display for each of the 4 digits.

## 7.5. Recap

| Component | Function |
|---|---|
| next_state | AES Result (128-bit) |
| bit_index | Current display position (0 → 124) |
| display_data | Contains 4 7-sec digits with 1 bit/digit |
| driver7seg | Displays cyclical display_data on the display |
| rst (btnC) | Restart the display from the beginning |

Table 2: Summary table

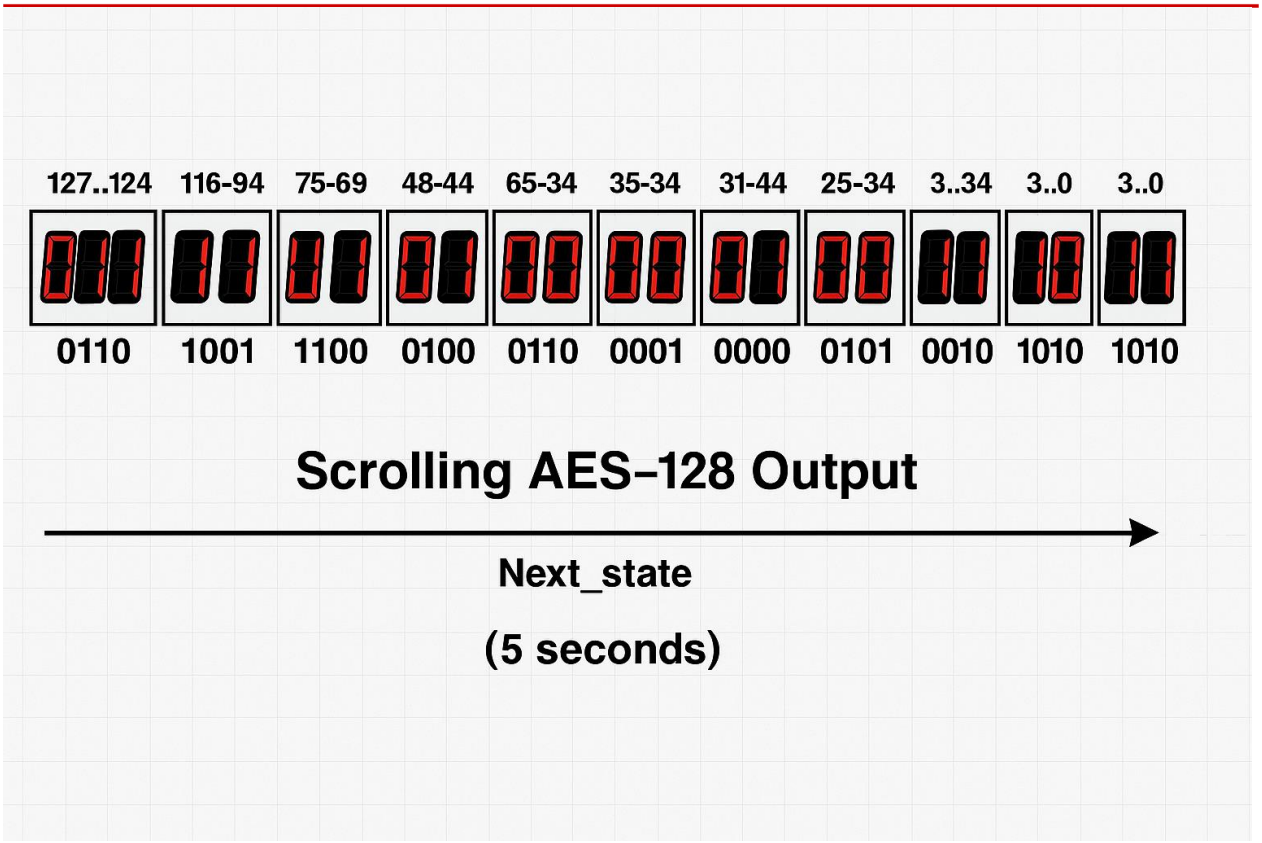



Image 4: Scroll to AES 128 Outbut (from AES_Wrapper)

# 8. SCOPE OF APPLICATION OF THE AES CO-PROCESSOR

The hardware implementation of the AES algorithm on the FPGA, in the form of a modular cryptographic co-processor, offers high performance and security, being relevant in numerous areas:

- **Cybersecurity**: protecting communications in corporate and government networks (VPN, TLS/SSL), performing encryption/decryption at high speeds at border points.
- **Medicine and health**: Encrypting patient data (electronic medical records, X-ray imaging or MRI protected), complying with GDPR/HIPAA requirements for privacy.
- **Finance and payments**: acceleration of EMV and ATM transactions, smart-card bank card systems, hardware wallets for cryptocurrencies, ensuring resistance to side-channel attacks.
- **Internet of Things (IoT):** securing communications between IoT devices at the edge, including industrial sensors and smart homes, where hardware resources are limited.
- **Automotive and transport**: V2X (vehicle-to-everything) data protection, ECU (Electronic Control Units) authentication and telemetry encryption for autonomous systems.
- **Telecommunications and infrastructure**: real-time encryption of voice/data streams for 5G networks, backbone equipment and military communications.
- **Cloud computing and storage**: server-level hardware encryption and SSD devices for data protection at rest, integrated into HSM (Hardware Security Module) solutions.
- **Digital identity and smartcard**: deployment on eID cards, SIMs and authentication devices, ensuring the secure storage and processing of cryptographic keys.

# 9. GENERAL CONCLUSIONS

1. **Modularity and maintenance**: the module-based VHDL architecture (SubBytes, ShiftRows, MixColumns, AddRoundKey, KeyExpansion, Round, FSM) facilitates point-by-point testing and optimization, allowing selective updates (e.g. extension to AES-192/256).
2. **Performance and pipeline**: the possibility of round-level or even partially unrolling pipelines provides high encryption throughputs, according to the requirements of streaming and real-time communication applications.
3. **Hardware security**: implementation on FPGAs allows the implementation of counter-countermeasures to side-channel attacks (glitch, fault injection), as well as integration into certified HSMs.
4. **Scalability**: the design is parameterized (generic key size, number of rounds) and can be ported to larger FPGAs (Artix-7, Kintex, Zynq) for high-volume commercial applications.
5. **Interoperability**: integration with interface standards (AXI-Lite) ensures easy communication with embedded processors (ARM, MicroBlaze) or SoC systems.
6. **Contribution to research and education**: Commented and structured code serves as teaching support for hardware cryptography courses and lab projects.

Overall, this AES co-processor represents a robust, flexible and high-performance solution for any system that requires symmetric encryption at the hardware level, adapting to security, throughput and resource consumption requirements.

# 10. BIBLIOGRAPHY

[1] National Institute of Standards and Technology (NIST), *FIPS Publication 197: Advanced Encryption Standard (AES),* November 2001.

[2] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer-Verlag, 2002.

[3] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 7th ed., Pearson, 2017.

[4] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*, 2nd ed., Springer, 2010.

[5] IEEE Std 1076-2008™, *IEEE Standard VHDL Language Reference Manual*, IEEE, December 2008.

[6] Xilinx, Inc., *Vivado Design Suite User Guide: Synthesis (UG910, Vivado 2024.1),* 2024.

[6] Xilinx, Inc., *Vivado Design Suite User Guide: Implementation (UG974, Vivado 2024.1),* 2024.

[7] Digilent Inc., *Basys3 Reference Manual*, Rev. B, 2015.

[8] D. Canright, "A very compact S-box for AES," in *Proceedings of CHES 2005*, vol. 3659, pp. 441–455, 2005.

[9] P. S. L. M. Barreto and V. Rijmen, "Efficient FPGA Implementation of Rijndael Round Function," in *IEEE Trans. on Computers*, vol. 52, no. 4, pp. 440–447, April 2003.

[10] J. López, "High-Speed AES Encryption on FPGA Platforms," *IEEE Micro*, vol. 24, no. 6, pp. 35–45, Nov.–Dec. 2004.

[11] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, Springer, 2007.

[12] A. Morioka and B. Preneel, "On the Hardware Implementation of AES," in *Rijndael 2002 Conference Proceedings*, 2002.