York University

AES Module for the Hardware-Accelerated Local Encryption for Secure Data Storage

Group:
Syed Ali Raza Rizvi - //
Sachin Satahoo - //

Professor - Navid Mohaghegh
Course - //

Introduction

This report will focus on the aes128_iterative module written in system verilog in aes_128.sv. The AES-128 (Advanced Encryption Standard) algorithm encrypts or decrypts a 16 byte block (128 bits) with its 128 bit cipher key. In our project we have used it to encrypt a 10 digit phone number to receive ciphertext. Or decrypt 32 HEX characters to receive the plaintext in HEX characters, which is 32 characters, where 2 characters represent one decimal digit. The plaintext ASCII printable character results should be in the range of 0x30 to 0x39 (0-9).

System Architecture

We are utilizing the Pico 2W as the master to send and request 128 bits at a time.

For **encryption**, the Pico will be zero padding the 10 digit number to 128 bits so that the DE10-lite can perform a full block encryption with the mode switch SW0 set to 0 (LED9 is off). Example: Entering a 10 digit phone number (1234567890), will send 31323334353637383930303030303030 over SPI, then receive the 32 HEX characters representing the encryption.

For **decryption**, The Pico will send a length of 32 HEX characters to send the 128 bits to the DE10-lite to perform the decryption with the mode switch SW0 set to 1 (LED9 is on). Then the Pico will receive 32 HEX characters of the decrypted number with 10 digits being the phone number and the remaining 6 digits being zeros(0x30). 16 digits in the range of 0x30 to 0x39.

Design and Implementation

The AES-128 algorithm goes through 10 rounds for encrypting and performs the inverse for encryption.
The 128 bit data is viewed in column major form in a state array.
We have 4 main functions that will be affecting the 4x4 byte state array.
Substitute Bytes, Shift Rows, Mix Columns, and Add Round Key.

Starting with **Add Round Key**, this is the simplest, all this does is perform an XOR on the state array with a round key depending on which round it is on. Round keys are generally generated with the cipherkey in key expansion. However, for this project, due to testing constraints, we hardcoded the round keys for the cipher key "~~YYYYYYYYYYYYYY~~".

Next, **Substitute Bytes** uses a 16x16 lookup table, SBOX, each index of this lookup table is 8

bytes. We are looking at the bytes in the state array and substituting them with the new values. The upper nibble gives the row index and the lower nibble gives the column index of the lookup table. We also have the **Inverse Substitute Bytes** for **decryption** which does the same thing, but with INV_SBOX.

Then, **Shift Rows** is a matrix transformation.

State array		After Shift
[b0][b4][b8][b12]		[b0][b4][b8][b12], no shifts for row 0
[b1][b5][b9][b13]	=>	[b5][b9][b13][b1], shift left once row 1
[b2][b6][b10][b14]		[b10][b14][b2][b6], shift left twice row 2
[b3][b7][b11][b15]		[b15][b3][b7][b11], shift left thrice row 3

For **decryption**, we have **Inverse Shift Rows** which undoes the shifts by shifting right instead of left for each of the rows for the same amount of times.

Finally, **Mix Columns**, the most complicated step, involves matrix multiplication in polynomial representation in GF(2^8) (Galois Field). Each byte is a polynomial over GF(2^8), columns are considered as polynomials over GF(2^8), and remainders modulo "overflow" beyond degree 7. The FIPS 197, Advanced Encryption Standard book shows the fixed matrix a(x) like so:

$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$

For **encryption** we use the fixed matrix:

02 03 01 01
01 02 03 01
01 01 02 03
03 01 01 02

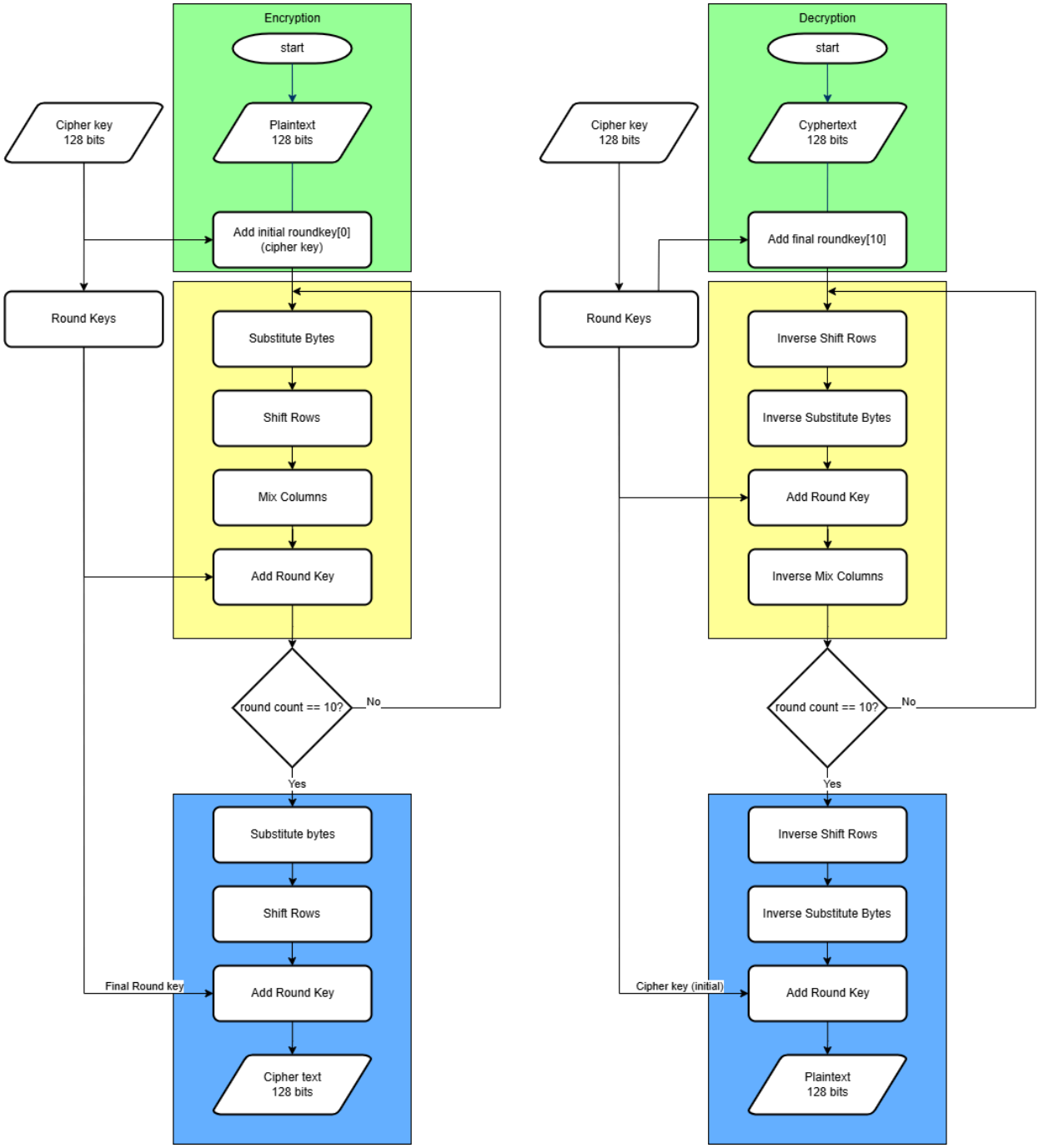
For **decryption** we use the fixed matrix(inverse of AES matrix):

0e 0b 0d 09
09 0e 0b 0d
0d 09 0e 0b
0b 0d 09 0e

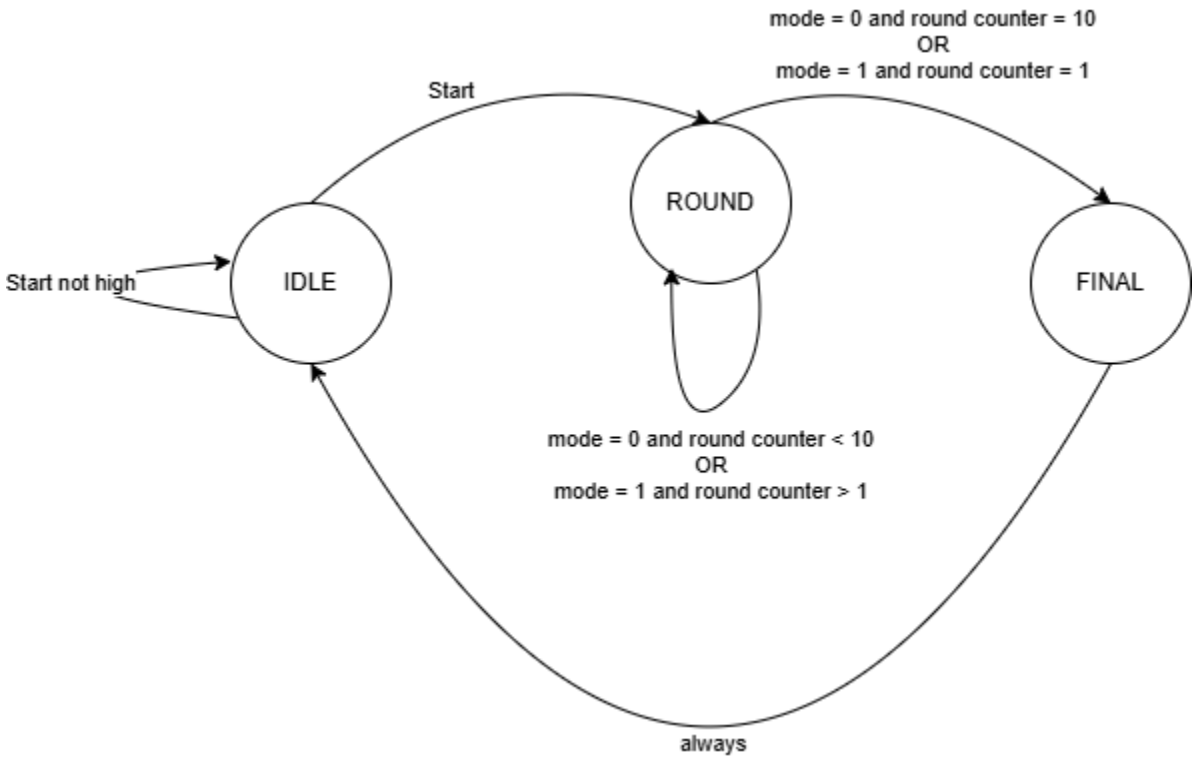
For each column, we will perform the multiplication like so:

[b0']	| 02 03 01 01|		[b0]
[b1']	|01 02 03 01 |	*	[b1]
[b2'] =	|01 01 02 03 |		[b2]
[b3']	|03 01 01 02 |		[b3]

Below is a flowchart and FSM diagram for the aes128_iterative module.



In the aes128_iterative module, we have 3 states. IDLE(green), ROUND(yellow), and FINAL(blue)



For the decryption, we simply use the inverse functions, and the round counter starts from 10 instead. In the flowchart, it says “round count == 10” in the decision block for decryption simply to show that they both go through 10 rounds.

The FSM has three states: **S_IDLE**, **S_ROUND**, and **S_FINAL**. This will sequence the AES rounds. In **S_IDLE,** we wait for the start signal, then load data_in (XOR’d with either the first or last round key depending on mode being set to on or off), sets busy, and initializes round_cnt up or down depending on encryption or decryption. In **S_ROUND** it applies the middle‑round transform (or its inverse) and updates round_cnt until the last round, then moves to **S_FINAL**, performs the final SubBytes/ShiftRows + AddRoundKey (no MixColumns), asserts done, clears busy, and returns to **S_IDLE**. With data_out being assigned to the state_reg (the state matrix being passed through).

Conclusion

In this project we implemented the aes128_iterative module in SystemVerilog on the DE10-Lite FPGA and verified all ten rounds of AES-128 encryption and decryption (SubBytes, ShiftRows,

MixColumns, AddRoundKey and their inverses) using hard-coded round keys. Functional tests show that encryption and decryption are working, confirming the core's correctness. SPI communication with the Pico 2W still requires debugging.

Next steps include replacing hard-coded keys with a dynamic key-expansion unit and fixing the SPI interface. With these enhancements and support for any 128-bit cipherkey over the HTTP socket, the system will become a complete hardware-accelerated AES-128 solution for secure data storage.