California Polytechnic State University Pomona

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Digital Circuit Design Using Verilog

ECE 3300-03 Fall 2023

Final Project Report

Advanced Encryption Standard

Prepared by

Jason Molina

Sean Abi Saab

Nicholas Heckers

Presented to

Dr. Mohamed El-Hadedy

December 11, 2023

Introduction

The Advanced Encryption Standard (AES) is a FIPS-approved cryptographic algorithm designed to safeguard electronic data. Functioning as a symmetric block cipher, AES can both encrypt (encipher) and decrypt (decipher) information. Encryption transforms data into ciphertext, an unintelligible form, while decryption reverses this process, restoring the data to its original plaintext. This specification outlines the Rijndael algorithm, which was designed for the processing of 128-bit data blocks. The algorithm accommodates cipher keys with lengths of 128, 192, and 256 bits.

This standard is suitable for Federal departments and agencies seeking cryptographic protection for sensitive (unclassified) information, as defined in P. L. 100-235. While other FIPS-approved cryptographic algorithms can be employed alongside or instead of AES, agencies using cryptographic devices for classified information protection may apply them to safeguard sensitive (unclassified) information in place of this standard. Non-Federal Government organizations are encouraged to adopt and use this standard if it meets their security requirements.

Implementation of the AES algorithm can take the form of software, firmware, hardware, or a combination thereof, contingent on factors like application, environment, and technology. It must be used alongside a FIPS-approved or NIST-recommended mode of operation, with Object Identifiers (OIDs) and associated parameters available at the Computer Security Objects Register (CSOR). Validated implementations, tested by an accredited laboratory, are considered compliant with this standard.

Recognizing that cryptographic security involves various factors beyond correct algorithm implementation, users, including Federal Government employees, should refer to NIST Special Publication 800-21, Guideline for Implementing Cryptography in the Federal Government, for comprehensive information and guidance.

ALGORITHM FUNCTIONS

AddRoundKey(): In the process of Cipher and Inverse Cipher transformation, a Round Key is incorporated into the State through an XOR operation. The Round Key's length corresponds to the size of the State, meaning that for Nb = 4, the Round Key length is 128 bits or 16 bytes.

InvMixColumns(): Transformation in the Inverse Cipher. Inverse of MixColumns().

InvShiftRows(): Transformation in the Inverse Cipher. Inverse of ShiftRows().

InvSubBytes(): Transformation in the Inverse Cipher. Inverse of SubBytes().

MixColumns(): Transformation in the Cipher where it independently mixes the data of each column in the State, resulting in the generation of new columns.

RotWord(): This function takes a four-byte word and executes a cyclic permutation on it. Employed by the Key Expansion routine.

ShiftRows(): Transformation in the Cipher that shifts the last three rows of the State by varying offsets in a cyclical manner.

SubBytes(): Transformation in the Cipher that applies a nonlinear byte substitution table (S-box) to the State, independently operating on each of its bytes.

SubWord(): Processes a four-byte input word by applying an S-box to each of the four bytes, resulting in an output word. Employed by the Key Expansion routine.

Parts Used

Nexys A7 100T FPGA Trainer Board

- Buttons
- Normal LEDs
- RGB LEDs
- 7-Segment Displays
- Switches
- VGA Display Output

Theory

The AES algorithm processes input and output sequences of 128 bits, referred to as blocks. The Cipher Key is a 128, 192, or 256-bit sequence. Bit indices range from 0 to one less than the sequence length ($0 \le i < 128$, $0 \le i < 192$, or $0 \le i < 256$) for blocks and keys. Other lengths are not allowed.

In AES, the fundamental processing unit is a byte, an eight-bit sequence. Input, output, and Cipher Key sequences are organized as byte arrays, denoted by "an" or "a[n]". The index "n" varies based on the key length:

- For a 128-bit key or a 128-bit block: $0 \le n < 16$
- For a 192-bit key: $0 \le n < 24$
- For a 256-bit key: $0 \le n < 32$

Byte values are presented as {b7, b6, b5, b4, b3, b2, b1, b0}. Hexadecimal notation is also employed, where each group of four bits is represented by a single character. For example, {01100011} is expressed as {63}. Certain finite field operations may include an extra bit (b8) to the left of an 8-bit byte. This additional bit is depicted as '{01}' preceding the 8-bit byte.

Bit Pattern	Character
0000	0
0001	1
0010	2
0011	3

Bit Pattern	Character
0100	4
0101	5
0110	6
0111	7

Bit Pattern	Character
1000	8
1001	9
1010	a
1011	b

Bit Pattern	Character
1100	c
1101	d
1110	e
1111	f

Figure 1. Hexadecimal representation of bit patterns.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	…
Byte number	0								1								2								…
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	…

Figure 2. Indices for Bytes and Bits.

an = {input8n, input8n+1, …, input8n+7}.

The AES algorithm internally processes data in a two-dimensional array of bytes called the State. The State comprises four rows, each with Nb bytes (where Nb is the block length divided by 32). Each byte in the State is identified by its row number (r) and column number (c), allowing it to be referred to as $s_{r,c}$ or s[r,c]. In this standard, Nb is set to 4, meaning $0 \leq c < 4$. At the beginning of the Cipher and Inverse Cipher, the input array of bytes is copied into the State array. Subsequent Cipher or Inverse Cipher operations are performed on this State array, and the final result is then copied to the output array of bytes.

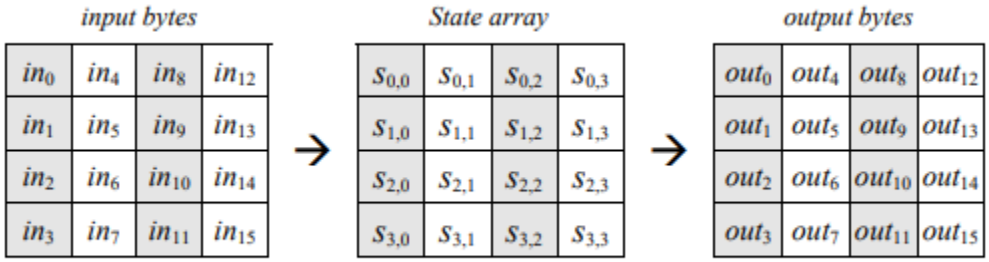


Figure 3. State array input and output.

In the initial stage of the Cipher or Inverse Cipher process, the input array, denoted as 'in,' is transferred to the State array using the following rule: $s[r,c]=in[r+4c]$ for $0 \leq r < 4$ and $0 \leq c < Nb$. At the conclusion of the Cipher and Inverse Cipher operations, the State is duplicated to the output array 'out' using the following rule: $out[r+4c]=s[r,c]$ for $0 \leq r < 4$ and $0 \leq c < Nb$. The State array consists of four bytes in each column, forming 32-bit words. Each row (indexed by r) represents the four bytes within a word. This arrangement allows the state to be viewed as a one-dimensional array of 32-bit words (columns). The column number serves as an index for accessing this array.

Implementation

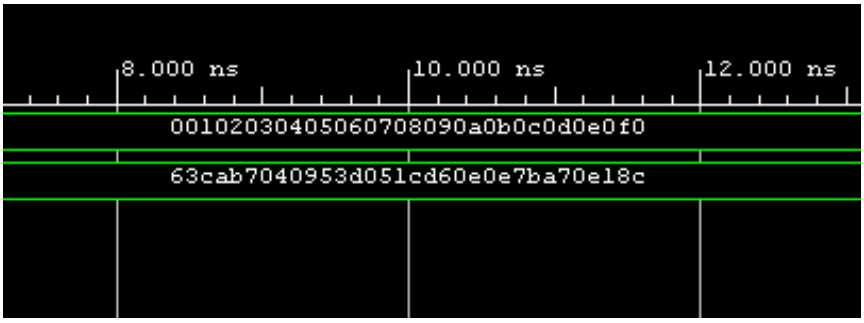
To implement this encryption algorithm, we can simply create HDL modules that perform the operations on their input data the same way that the functions described in the AES standard do. Since the cipher will be implemented for use in an FPGA, however, it is unnecessary to keep the state stored in memory somewhere, and we can instead send the state from module to module via Verilog wires.

SubBytes

The first function to implement was SubBytes, since it is used in other functions and depends on none of the others. The AES standard describes how the values in the S-box can be calculated, but for this use case we can simply use the precomputed values to implement SubBytes. The obvious way to go about this is simply to create a two dimensional register that is initiated with the proper values. After we have this, all that is left is to find which indices of the S-box to use in producing the desired output.

The index of the desired column of the S-box, or "y" in the AES standard, is the lower byte of the two byte input. The index of the desired row of the S-box, or "x", is the higher byte of the two byte input. Simple wire manipulation can be used to route these bits into acting as the indices of the S-box. This allows us to get the proper two byte output value for the S-box.

Since SubBytes is often called on a string of multiple bytes, it is beneficial to expand our module to take in an input longer than the minimum two bytes, as well as output data of the same length. This is a rather simple modification to our existing S-box module, where we repeat the process done on the first two bytes to the rest of the two byte blocks, and then combine their respective outputs into one.

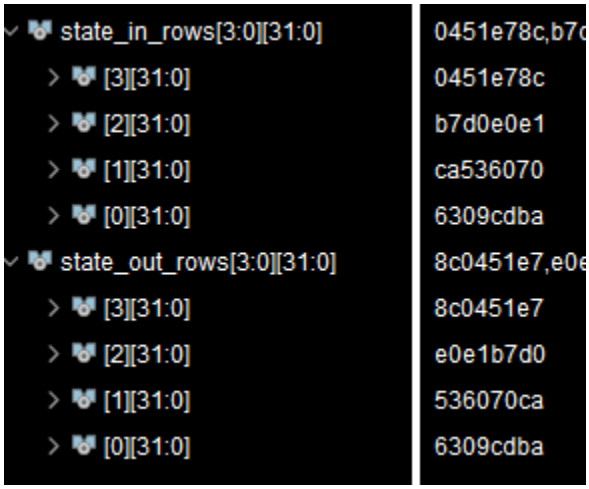


Output (bottom) of the S-box module based on the input (top)

ShiftRows

Shift rows is another relatively simple function used in the AES cipher. Shift row takes the State, which has four rows, and rotates them left by 0, 1, 2, and 3 bytes respectively. This could be implemented in a variety of ways, the route we decided to take was to use a previously designed n-bit barrel shifter module to do the necessary rotating.

The existing barrel shifter module is capable of rotating an n-bit wire n-bits to the right. While at first glance this isn't what we want to do in the ShiftRows module, it is all we need in order to achieve the desired functionality. With the barrel shifter set to rotate rather than shift, we can specify the number of bits to shift in multiples of 8, to act as though it is shifting whole bytes rather than individual bits. With this in mind, we can rotate the input data right until we have achieved the equivalent rotate left.



ShiftRows

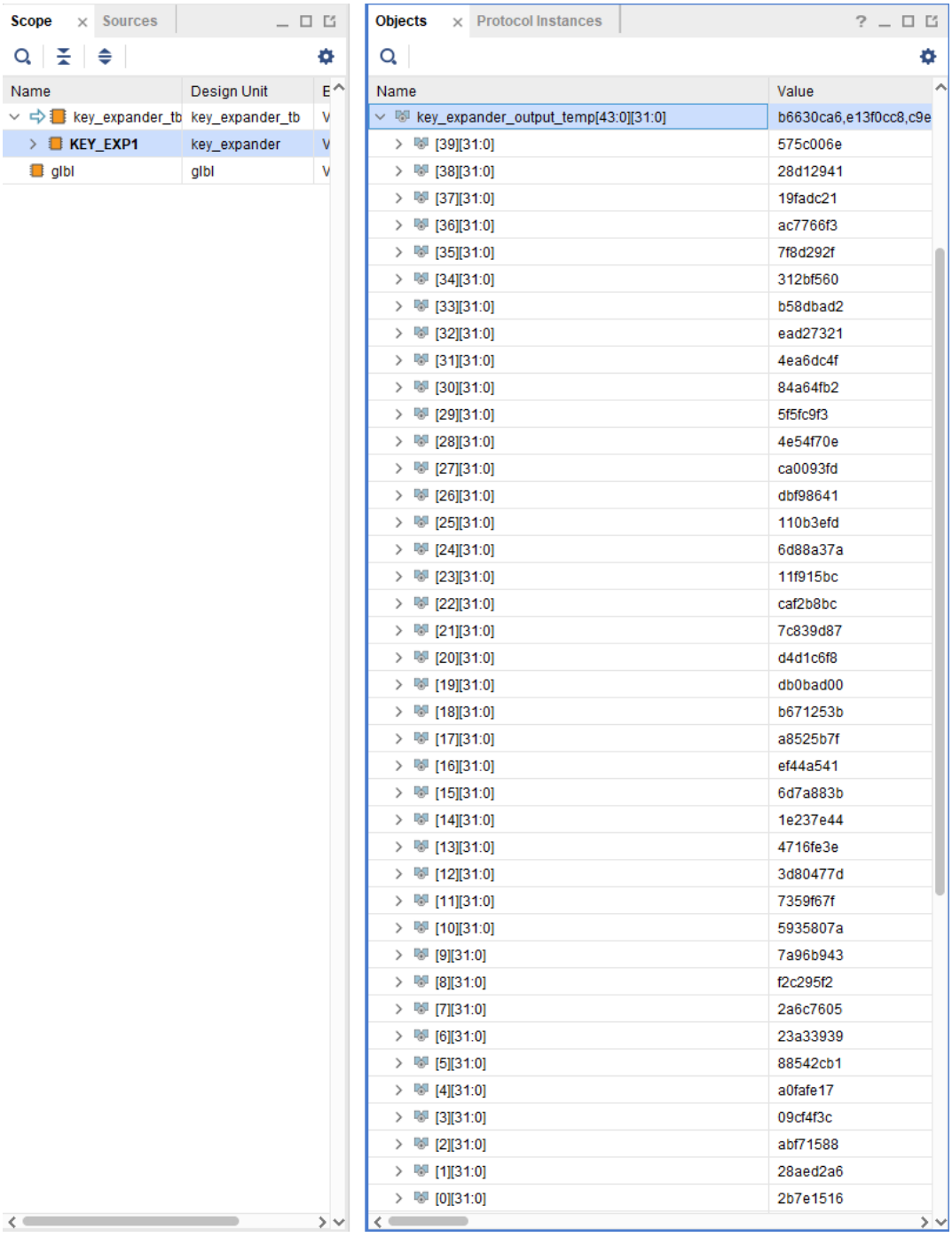
MixColumns

While MixColumns appears to be a few simple mathematical operations performed on

the input data, it can be one of the trickier functions to implement. This is because it involves multiplication in a Galois field. Fortunately, a general multiplication module is not necessary. In the case of MixColumns, only multiplication by 2 and by 3 are performed. This means that these more specific and thus simpler operations are necessary. After the multiplication by 2 and by 3 are added, these can be used with regular XOR in the pattern described in the AES standard to complete the module for this function.

One of the most vital parts of the AES cipher is the key expander, which takes a 128 bit input key and, in the case of AES-128, outputs 44 128 bit keys for use mainly in AddRoundKey. Most of the operations performed during the key expansion can be achieved using the prior described functions. One of the new elements used is Rcon, the round constant, which is a precomputed value for each round. This is not a challenge to implement, and can be achieved by creating a 2 dimensional register, with each row having the round constant inside.

The rest of the key expansion routine is also straightforward to implement, and is mainly just connecting existing modules in the right order as described in the standard's provided pseudocode. This module was slightly more difficult to verify it was properly working due to the sheer amount of data it produces for its output.



Key Expansion testbench words

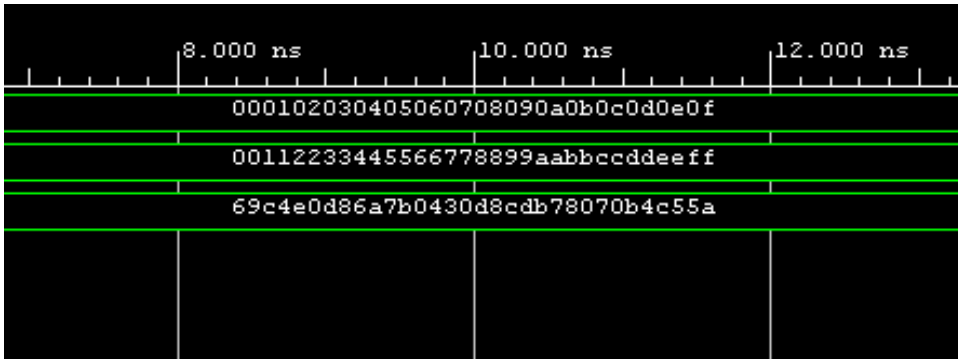
AddRoundKey

AddRoundKey is one of the simplest functions in the AES cipher. This module only requires an XOR between its input data and a word from the previously generated key schedule,

as described in the AES standard.

Cipher

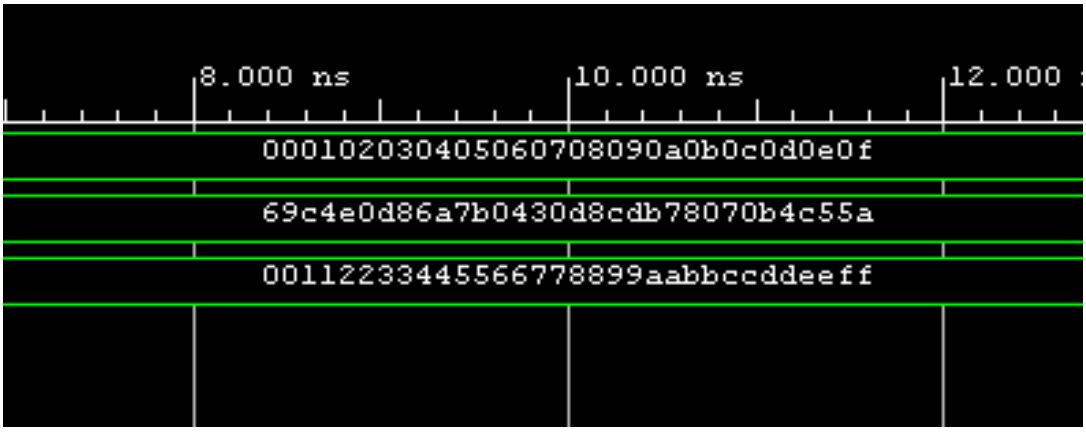
The culmination of the previous modules, the AES cipher is another function that is straightforward to implement as an HDL module after creating the necessary functions used. We simply direct the wire containing the current State through the proper modules in order. After the data reaches the final AddRoundKey, we have the completed encrypted data.



Cipher (Top to bottom: key, data, encrypted data)

InverseCipher

After completing the cipher for the encryption of the data, the inverse cipher is not very challenging. The inverse cipher itself is very similar to the cipher, except with the functions in a slightly different order, which mainly just requires changing which outputs go to which inputs for the modules used in place of the functions described in the standard.



Inverse Cipher (Top to bottom: key, encrypted data, decrypted data)

InvShiftRows

Inverse shift rows does the same thing as the regular ShiftRows, except it rotates the rows right instead of left. As this is what the barrel shifter described in the ShiftRows section did by

default, we just need to change the number of bits to rotate by to get the desired results.

InvSubBytes

InvSubBytes is the same as SubBytes, but with different values in the S-box. This means we can use the same code used for SubBytes, while only replacing the data in the two dimensional registers used for the S-box.

InvMixColumns

InvMixColumns follows the same principles used in InvMixColumns in its design, but uses multiplication by {0e}, {0b}, {0d}, and {09}. This leaves the code unchanged overall from MixColumns for the most part, besides the new multiplication functionality.

Top File

With AES-128 encryption and decryption done, all that is left is to combine them into one file to be uploaded to the FPGA development board along with additional functionality. Rather than a static module that only encrypts the data when a button is pressed. We went for the approach of storing the data to be operated on in a register for future use throughout time. With the initial key and data stored in this register, a switch can be switched to select encryption mode, and a button pressed to perform the operation and to update the data stored in the register. Switching the mode select switches to decryption mode and pressing the button will return the data to the initial unencrypted data.

To see the data that is stored in the data register, we have used the seven segment displays to display 4 bytes, or 8 hexadecimal characters, at a time. One switch is used to switch between displaying the current key and the current data. Another switch is used to enable or disable a feature that is used to automatically scroll and loop through which part of the data is currently on the seven segment displays, resulting in the effect of scrolling through the entire data stored before looping back to the start. Two buttons can be used for more fine control while viewing the data or key, with one shifting the viewed data 1 character right, and the other shifting it 1 character left.

A PWM controlled RGB LED is used to signify the encryption state of the data. If the data currently in the register matches the initial unencrypted data, it will appear red, signifying the data is unencrypted. If the current data does not match the initial data on the other hand, the LED will appear green, signifying the data is currently encrypted. For a more notable method of telling if the data is safely encrypted or if it is left unencrypted, a VGA monitor can be used to check with the VGA output.

CONCLUSION

This project was a valuable learning experience, both for the fundamentals of encryption, and for translating an algorithm for use in an FPGA. While many of the desired outcomes were achieved, there is still a lot of room for improvement. There are a number of ways this project could be improved, both simple and achievable ways, and more difficult ways.

While this project ultimately ended up solely working with AES-128, the design of our project would allow it to not be too difficult to accept longer keys as described in the AES standard. The key expander would have to be modified in order to produce the longer key schedules, and then more rounds of the cipher and inverse cipher would have to be added to achieve this goal. We could also make it so that the cipher module could operate on multiple blocks of data using the key, rather than only one.

During the development of the project, UART was intended to be used as the means of inputting the key and data, rather than hard coding it. Due to difficulties in implementing this feature, along with time constraints, we opted to preprogram this information and focus on finishing other features for the project.

In accordance to our abstract, recognizing the growing important of dedicated processors for common tasks like data encryption and decryption, the ultimate path for this concept after this project would be to add other common encryption algorithms that could be chosen, along with more efficient ways of sending encryption keys and data to the FPGA chip such as through a PCIe connector. This is the most likely application of something like our project, where an encryption coprocessor is used to alleviate the strain on main CPUs from common tasks, and to allow the CPU to focus on the more important tasks.