ECE4/530 Digital Hardware Design

Final Project

Simplified Advanced Encryption Standard (AES)

Background

This project is centered around the security encryption and decryption technique called the Advanced Encryption Standard (AES) which is a specification for the encryption of electronic data established by NIST in 2001. For more in-depth information and the original specification as described, see FIPS publication 197. AES was adopted by the US government as the replacement for the Data Encryption Standard (DES) and is now commonly used worldwide in everything from archive and compression tools such as WinZip and WinRAR to VPNs like GlobalProtectVPN.

The AES cipher belongs to a cryptographic set of primitives known as block ciphers or, algorithms that encrypt data on a per-block basis (not a bit by bit). The blocks are measured in (number of bits) to determine the input of plaintext and the resulting output of ciphertext. For example, a 128-bit block of plaintext is utilized in AES and a 128-bit ciphertext is produced.

As with all other symmetric encryption algorithms, AES relies on the usage of keys for encryption and decryption. The key term, symmetric, refers to the usage of the same key for both the encrypting party and decrypting party. Depending on key length for AES as defined by the FIPS publication, the number of iterations required within a specified set of function changes, where either 10, 12, or 14 rounds are required for a 128-, 192-, or 256-bit key, respectively. The iteration through multiple rounds is necessary for AES to add additional resistance to cryptanalysis.

Security breaches have been recorded throughout time for AES but, AES has yet to be broken in the manner that DES was broken in 1999. Many cryptographers agree that, with current computer hardware, a successful brute-force attack on the AES algorithm would take quite some time. Recent works on attacking the hardness of AES rely on analytical techniques such as Tau-statistical analysis, based on the Kendall rank correlation coefficient (look this up if you are interested!).

Despite the impracticality of an attack on AES by purely data-analytic means, that does not mean that AES is completely secure. Forms of side-channel attack relying on the information gained from the physical implementation of a cryptosystem can still be exploited to attack a system encrypted with AES. Common examples of side-channel attacks are:

1. Timing Attack: Attack based on the measurement of time required for various known computations to perform.
2. Power-Monitoring Attack: Attack relying on the variability of power consumption during a known computation.

3. Electromagnetic Attack: Attack based on leaking electromagnetic interference (EMI) that can directly provide an attacker with plaintext or other information such as cryptographic keys being used during computation in a method similar to the NSA's TEMPEST experiments (Similar experiment carried out by Tel Aviv University: http://www.cs.tau.ac.il/~tromer/papers/radioexp.pdf).

Mathematical Background

Many of the operations performed in AES are defined and executed at the byte-level, where the bytes represent elements in the finite Galois field $GF(2^8)$, with other operations happening in terms of a 4-byte word. For any prime power, there is a single finite field, thus all representations of $GF(2^8)$ are isomorphic with coefficients 0 and 1. To better understand the implementation, a classical polynomial representation can be achieved for some byte, $\boldsymbol{b} = \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$, is considered a polynomial with coefficient in $\{0,1\}$: $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$. For example, $0x57 = 0b01010111$ will correspond with the polynomial $x^6 + x^4 + x^2 + x + 1$.

The addition of two polynomials or bytes would work as a simple XOR function: $0x57 \oplus 0x83 = 0xD4$ in hexadecimal byte notation or, in polynomial notation as: $(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$. In binary notation, this is $0b01010111 + 0b10000011 = 0b11010100$. Addition of polynomials is done modulo 2. Thus, $1 + 1 = 0\ ;\ 1 + 0 = 1\ ;\ 0 + 0 = 0$.

The multiplication of the two polynomials may result in a polynomial outside of the $GF(2^8)$ field. In order to ensure that the result of the multiplication is still within the field $GF(2^8)$, it must be reduced by division with an irreducible polynomial (analogous to a prime number in arithmetic) of degree 8, the remainder of which will be taken as the final result. In AES this polynomial is chosen to be $m(x) = x^8 + x^4 + x^3 + x + 1$, or in hexadecimal notation $(0x11B)$

Considering the multiplication of our corresponding polynomials, $0x57 * 0x83 = 0x15BD$ as follows:

$$(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

Taking the result modulo the AES polynomial will give the result by:

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1\ modulo\ x^8 + x^4 + x^3 + x + 1 = \boldsymbol{x^7 + x^6 + 1}$$

Or in hexadecimal notation, $0x15BD\ mod\ 0x11B = 0x15BD\ XOR\ 0x11B = 0xC1$

Simplified AES

The standard AES algorithm is quite long and complex for those unfamiliar with the internal operations, thus an academic version of AES, named Mini-AES, devised by R. Phan (https://www.tandfonline.com/doi/abs/10.1080/0161-110291890948) will be described in this section. The Mini-AES algorithm was developed as an exercise and is simplified by having all parameters significantly reduced while still preserving the original structure as that of AES.

Mini-AES works on plaintext blocks of 16-bits each, with a key of 16-bits in a method known as Electronic Codebook Mode (ECB) since a large message may be divided into many blocks, in our case 16-bits, then encrypted individually. If a message is not divisible by 16-bits, the last block will contain the remainder of the message with 0-padding. The disadvantage of this method is known as a lack of diffusion, which will be discussed in detail in the future. A figure from Phan's paper of the method described can be seen in Figure 1.

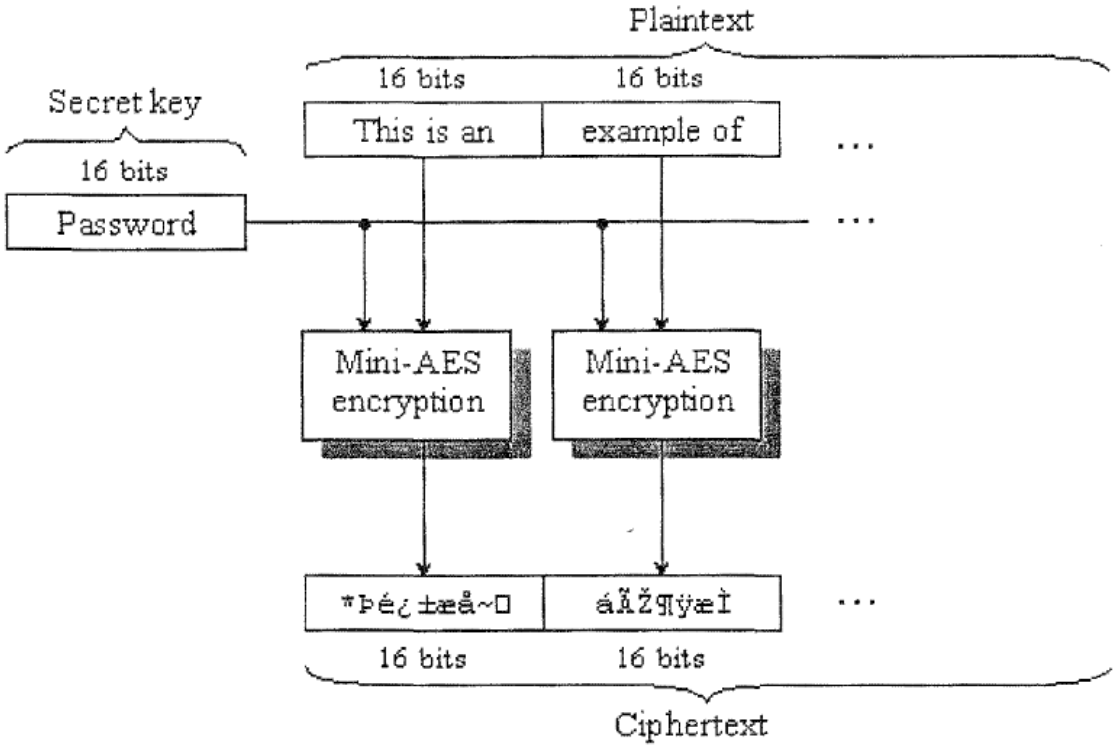


Figure 1: Method of encrypting many blocks with Mini-AES.

Before getting into the operation of Mini-AES, we will discuss the mathematical operation of standard AES, which is available in the original specification set by Rijndael (https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf).

AES Specifications

AES is an iterated block cipher with variable block length, with an independent specification of 128, 192, or 256 key bit-lengths. The transformations of the input message operate on the intermediate result, known as the *state*. The state can be visualized as a 2D array of bytes with four rows and a number of columns (*Nb*) determined by the block length divided by 32. The cipher key is also an array with four rows and number of columns (*Nk*) equal to the key length divided by 32. Figure 2 shows an illustration for State of a block cipher of 192 bit-length and a key of 128-bit length. The number of rounds needed for the transformation depends upon *Nb* and *Nk*. In this case 12.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Figure 2: Example of State (with Nb = 6 words) and Cipher Key (with Nk = 4 words) layout.

Round transformations are composed of four different transformations in pseudocode we have:

```
Round(State,RoundKey) {
                        ByteSub(State);
                        ShiftRow(State);
                        MixColumn(State);
                        AddRoundKey(State,RoundKey);
                      }
```

Then the final round of the cipher is slightly different and is defined by:

```
FinalRound(State,RoundKey) {
                             ByteSub(State) ;
                             ShiftRow(State);
                             AddRoundKey(State,RoundKey);
                           }
```

The ByteSub transformation is non-linear byte substitution, operating on each of the state's bytes independently by using a substitution table (S-Box) which is invertible and constructed by the composition of two transformations:

1. First by taking the multiplicative inverse of $GF(2^8)$ where $0x00$ is mapped onto itself.
2. Applying an affine cipher over $GF(2)$ with a transformation defined by:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The result can be shown as:

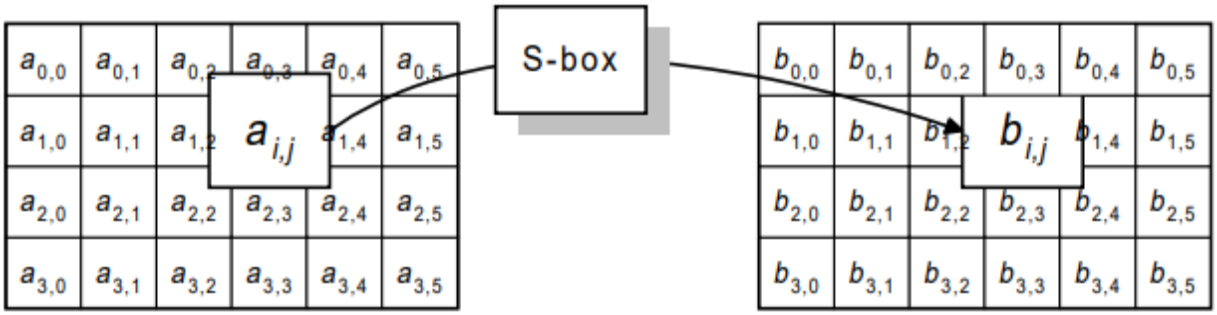
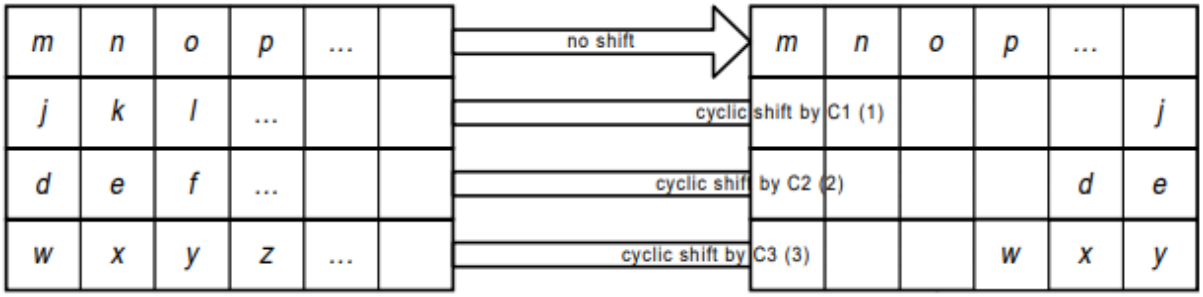


Figure 3: ByteSub acting on individual bytes of the state.

The ShiftRow transformation cylindrically shifts rows of the state, to the left, by differing offsets.



The MixColumns transformation considers the columns of the state as polynomials over $GF(2^8)$ multiplied modulo $x^4 + 1$ with a fixed polynomial, $c(x) = 03x^3 + 01x^2 + 01x + 02$.

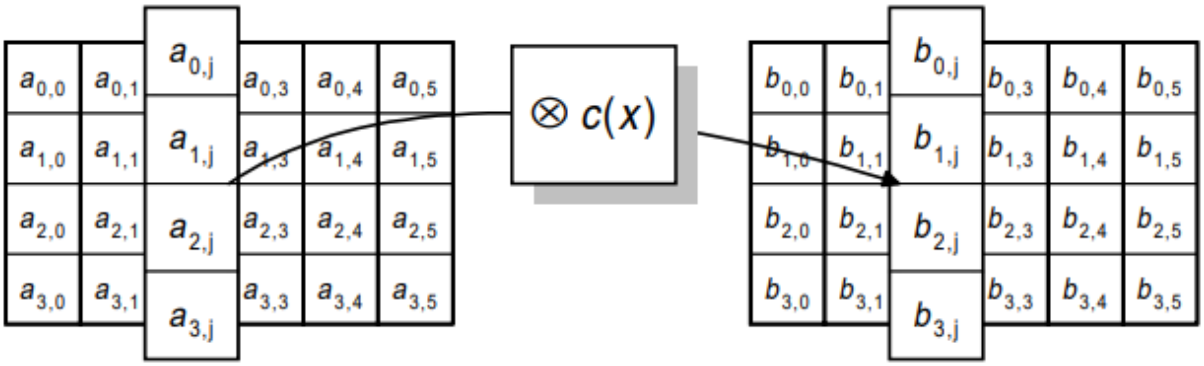


Figure 4: MixColumns operation for the columns of the state.

The round key addition works where the round key is applied to the state by a simple XOR. The round key is derived from the cipher key by means of a key schedule, the round key length is exactly equal to the block length.

Figure 5: Round key addition.

Mini-AES

Mini-AES works on a 16-bit block instead of a larger block as in the standard AES but the operations remain the same. The definition for a block is 16-bits represented as a matrix of 2 rows and 2 columns.

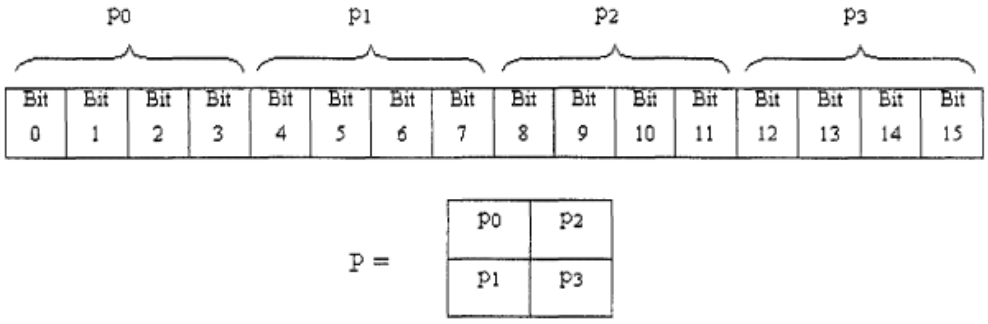


Figure 6: Representation of 16-bit block.

Each section of the block is a 4-bit nibble. The data being operated on in each component of the Mini-AES will operate on the individual nibbles.

NibbleSub operates as a S-Box shown below:

Input	Output		Input	Output
0000	1110		1000	0011
0001	0100		1001	1010
0010	1101		1010	0110
0011	0001		1011	1100
0100	0010		1100	0101
0101	1111		1101	1001
0110	1011		1110	0000
0111	1000		1111	0111

Figure 7: 4x4 S-Box

Where the input block is transformed with individual bytes substituted as:

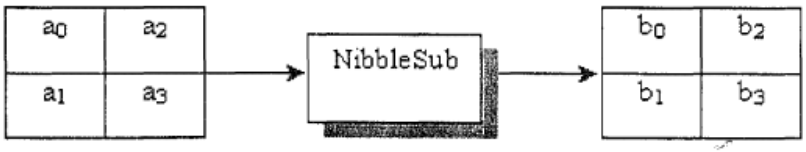


Figure 8: Transformation from S-Box.

The ShiftRows transformation does similarly a rotate of the input block to the left by differing nibble amounts. The first row is unchanged, but the second row is rotated by a single nibble.

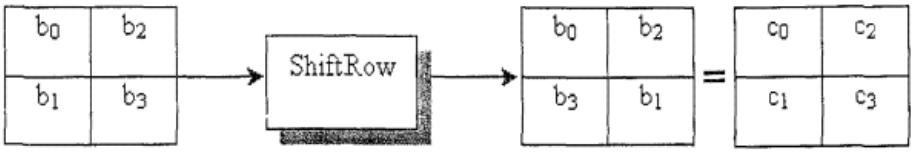


Figure 9:ShiftRows operation changing position of data in rows.

The MixColumns step is similar to its full AES counterpart where the input block is effectively multiplied by a constants matrix as defined by the Phan text, based on a simplification of the AES constants matrix.

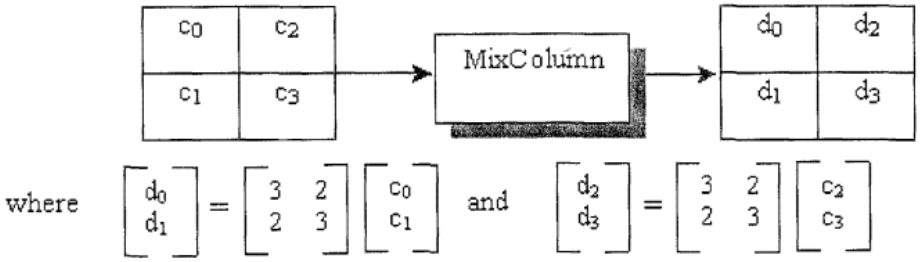


Figure 10: MixColumns operation in Mini-AES.

Following, the KeyAddition step is a simple XOR of the generated round-key with the data. The round key is derived from the secret key, K by using the key schedule, which will be described next.

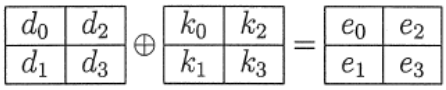


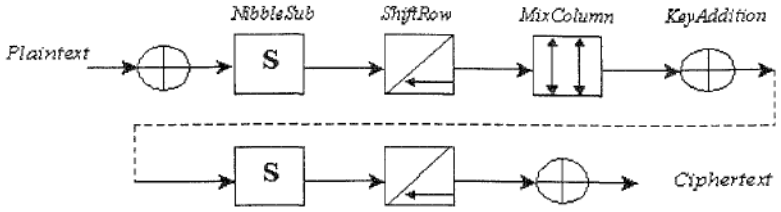
Figure 11: KeyAddition Operation.

The 16-bit secret key is *passed* through a key-schedule to produce one 16-bit round key, K_o to be used prior to the first round, and a 16-bit round key, K_i for use in each round of Mini-AES. Mini-AES encryption is defined to have 2 rounds, hence three round keys, K_0, K_1 and K_2 are generated. The generation of round keys is shown in the following table for 3 rounds. Note that in each round, round constants rcon(i) are used, where rcon(l) = 0001 and rcon(2) = 0010.

Round	Round Key Values
0	$w_0 = k_0$ $w_1 = k_1$ $w_2 = k_2$ $w_3 = k_3$
1	$w_4 = w_0 \oplus \text{NibbleSub}(w_3) \oplus \text{rcon}(1)$ $w_5 = w_1 \oplus w_4$ $w_6 = w_2 \oplus w_5$ $w_7 = w_3 \oplus w_6$
2	$w_8 = w_4 \oplus \text{NibbleSub}(w_7) \oplus \text{rcon}(2)$ $w_9 = w_5 \oplus w_8$ $w_{10} = w_6 \oplus w_9$ $w_{11} = w_7 \oplus w_{10}$

Figure 12: Key Generation.

The dataflow for Mini-AES then can be put together as:



Datapath

The flowchart design for Mini-AES is straightforward, see the below figure:

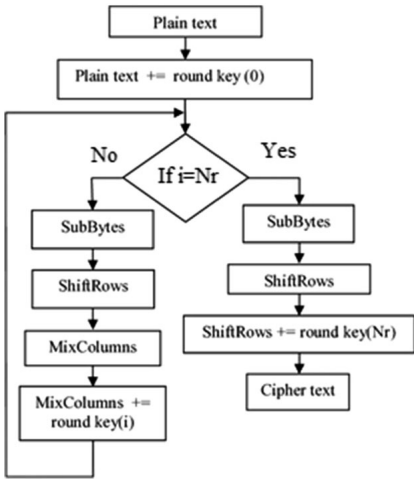


Figure 13: Datapath for Mini-AES

In the shown flowchart, there is a 16-bit *key*, 16-bit data block (*plain text*), a 16-bit initialization vector (*IV*) inputs. The 16-bit *cipher text* result is generated and stored in a result register, a finished signal should also be generated to indicate a valid result is ready. Not shown is an XOR operation of the plain text and the IV which is done prior to the rounds for added security, a counter, which will be critical as feedback to the rounds since the order of operations changes depending on if the round is the last round or not.

Project Descriptions

The goal of this project is to develop a Mini-AES encryption machine with a parameterized number of rounds (1 to 15).

Inputs:

- Low-active Reset signal
- High-active Start signal
- 16-bit initialization vector
- 16-bit message value (plain text)
- 16-bit encryption key
- 4-bit representing number of rounds

Outputs:

- 16-bit encrypted message

Design Modules:

1. Design the NibbleSub module as a ROM.
2. Design the shiftRow module as a combinatorial module.
3. Design the MixCol module as multiple instantiations of the PolyMult (polynomial multiplication) described in the attached write-up. The multiplication should be designed as FSM with registers to store the two input numbers and the multiplication output. Test this module individually via simulation to ensure the correct functionality.
4. Design a K-Exp (Key Expansion) module to generate a round key specified by a 4-bit input i, as explained in Figure 12. The module should expand an input key (16-bit) to a (16-bit) round key for any identified round.
5. Design an AES_Cntr (Controller) to handle the encryption process in the rounds using a FSM. Generate control signals as needed to activate any of the designed modules
6. Design the AES_Min top module as instantiations of all the designed modules.
7. Synthesize the top module and extract the resource utilizations. Make sure there are no latches.

Testing:

Use Example 9 as explained in the Phan paper! The initialization vector for that example is 0 would be zero. A sample testing procedure that requires a user input from the keyboard is as follows:

1. Reset the system. Display: Push “R” to start: <accept a user input>
2. Enter the initialization vector: <accept input from the user>
3. Enter the message in the same manner: <accept input from the user>
4. Enter the number of rounds from (1-15) in a 4-bit binary number: <accept input from the user>
5. Activate a start signal for the machine to calculate the encryption.
6. Once encryption calculation has finished, a done signal is generated with the output be the encrypted value as 0x????: a print out in the form DONE: 0x???
7. To do another encryption, go to Step 1.

Submission:

- Submit all Verilog design modules well commented to explain what you are intended to design and how, at the beginning of each file as a comment. Attach all FSM bubble diagrams. With an explanation of each state.
- Submit the Verilog file of the Testbench. Describe the testbench inputs and outputs. Attach simulation waveforms with comments.
- Summary of the resource utilization with comments.
- You will sign-up for a 30-minute meeting with me to demo your project any day prior to December 09, 2022. (Sign in the sheet posted on Canvas). Good Luck
- Graduate students are to write a 4-page IEEE paper about own project submitted by end of day Sunday before the final's week. (December 11, 2022)