LAB BASED PROJECT REPORT

On

AES Algorithm Using Verilog

Submitted in partial fulfilment of the

Requirements for the award of degree

Bachelor of Technology

In

Electronics and Communication Engineering

Submitted By

N.ESWAR – 180040256

K.Y SAGAR – 180040257

P.SRIVIDYA – 180040263

Under the guidance of

M. Vineetha Mam

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

KONERU LAKSHMAIAH EDUCATIONAL FOUNDATION

Green Fields, Vaddeswaram , Guntur District

K L E F

KONERU LAKSHMAIAH EDUCATION FOUNDATION

(Deemed to be university estd, u/s, 3 of the UGC Act, 1956)

(NAAC Accredited "A" Grade University)

CERTIFICATE

This is to certify that the major project entitled **"AES Algorithm Using Verilog"**, being submitted by **"N. Eswar – 180040256 , K. Y. Sagar – 180040257 , P. Srividya – 180040263"** in partial fulfillment for the award of degree of **Bachelor of Technology (B. Tech)** in Electronics and Communications Engineering is a record of confide work carried out by them under our guidance during the academic year **2020 - 2021** and it has been found worthy of acceptance according to the requirements of the university.

Signature of the Project Guide

Signature of Head of Department

Department of ECE

K L E F

KONERU LAKSHMAIAH EDUCATIONAL FOUNDATION

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

DECLARATION

we here by declare that this project based lab report entitled " **AES Algorithm Using Verilog**" has been prepared by us in partial fulfillment of the requirement for the award of degree **"BACHELOR OF TECHNOLOGY IN ELECTRONICS AND COMMUNICATIONS OF ENGINEERING"** during the academic year 2020-2021.

we also declare that this project-based lab report is of our own effort and it has not been submitted to any other university for the award of any degree.

Acknowledgement

We are greatly indebted to our KL University that has provided a healthy environment to drive us to achieve our ambitions and goals. We would like to express our sincere thanks to our project Uncharged T**.Sanath Kumar Sir** & **M. Vineetha Mam** for the guidance, support, and assistance they have provided in completing this project.

With immense pleasure, we would like to thank the Head of the Department, **Dr. M. SUMAN** sir for his valuable suggestions and guidance for the timely completion of this project.

We are very much glad for having the ssupport given by our principal, **K.Subba Rao** sir who inspired us with his words filled with dedication and discipline towards work.

We believe that **"Practical Leads A Man Towards Performance**".
Last but not the least, a special thanks goes to the Parents, staff and classmates who are helpful either directly or indirectly in completion of the Lab Based project

1. Abstract

With the advent of easily available high-speed computers there has been an increase in demand for effective methods to secure data despite what processing power an adversary may possess. Older cryptographic methods, such as DES, do not have a large enough key space to lend themselves to applications where high security is needed. To remedy this situation, the National Institute of Standards and Technology (NIST) created a competition to select the next generation encryption standard. The method that was selected was a design proposed by Joan Daemon and Vincent Rijman, which they called Rijndael.

The Rijndael algorithm, or as it is now known, the Advanced Encryption Standard (AES) provides a symmetric key cryptography that allows for the encryption and decryption of fixed size blocks of data. As a symmetric system, the secret key must be shared between the sender and receiver for communication to be possible.

2. Rijndael Algorithm Overview

The Rijndael algorithm that was implemented uses a 128-bit key and 128-bit data block. It operates on the data in several rounds; each round consisting of several mathematic operations designed to obscure the data. To accommodate the several rounds using a single 128-bit key, a key expansion algorithm is used. The major steps involved in each round of encryption are the ByteSub, ShiftRow, MixColumns, and AddRoundKey. Throughout the steps of encryption, the current value for the system is stored in the system state. The system state is a 128-bit variable that starts with the contents of the plaintext and ends with the contents of the ciphertext.

For a 128-bit key and data bock size ten rounds of encryption or decryption are required. The differences between performing encryption and decryption are minimal. The same basic mathematical operations and keys are used, only they are used in a different order with different parameters. To accommodate for the required key expansion, the system that was implemented also has a setup round that is used to perform the expansion as well as handle other initialization tasks.

The 128-bit data input is viewed as a byte matrix with four rows and four columns. Some of the operations defined for the algorithm operate on rows or columns of this data matrix, while other operations deal with individual data bytes.

2.1 ByteSub

The ByteSub operation performs a byte substitution on every byte of data in the current state. This substitution is defined as operations within a Gallois Field (GF). The operations required are to find the multiplicative inverse of a byte within GF(256) and then apply a fixed affine transformation on the data over GF(2).The results of performing these operations were computed for all 256 possible values and the operation was implemented using a look up table known as an “s-box”. A representation of the ByteSub operation can be seen in figure 1.

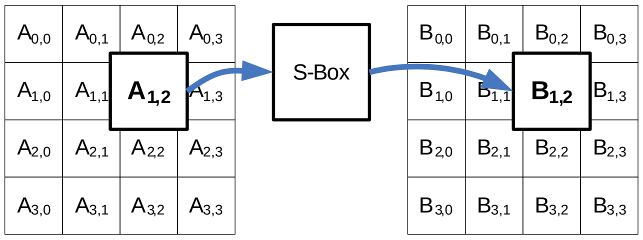

$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
S-Box
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

Figure 1: S-box Operates on Individual Bytes

For decryption, the “s-box” must be inverted so that the application of the inverse s-box to data received from the normal s-box will result in the original data. This function was also implemented using a look-up table.

2.2 ShiftRow

The ShiftRow operation affects each of the four rows of data individually. Each row is rotated by a different amount. The first row is unchanged and rows two through four are rotated by one, two, and three bytes, respectively. For encryption, a right rotation is performed while for decryption a left rotation is used. The result of the shift on the data matrix representing the data can be seen in figure 2.

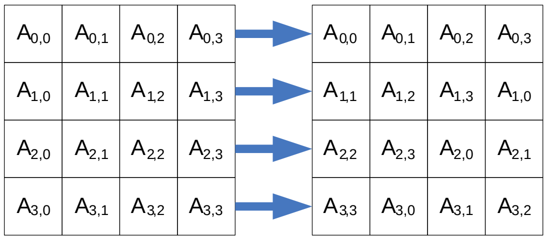

$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{1,0}$
$A_{2,2}$ $A_{2,3}$ $A_{2,0}$ $A_{2,1}$
$A_{3,3}$ $A_{3,0}$ $A_{3,1}$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

Figure 2: Left Rotation Used for Decryption

2.3 MixColumns

The MixColumns operation transforms the data in the current state by operating on each of the four columns independently. For this operation each column of the data is treated as a polynomial over GF(256). The polynomial is multiplied by a fixed polynomial and the result is taken modulo $x^4 + 1$. The fixed polynomial used for decryption is the inverse of the polynomial used for encryption in the field GF(256) modulo $x^4 + 1$. The encryption polynomial is $3x^3 + 1x^2 + 1x + 2$. The inverse of this polynomial is $11x^3 + 13x^2 + 9x + 14$.

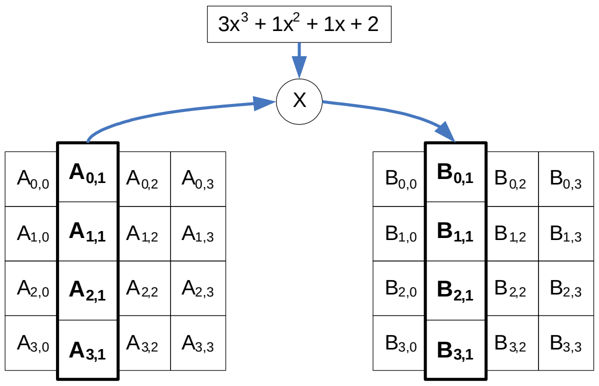

$3x^3 + 1x^2 + 1x + 2$
X
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

Figure 3: MixColumn Operation

The encryption and decryption for the MixColumns operation were implemented in different ways. The polynomial used for encryption was specifically selected to make the computation very easy to implement. Because of this, the encryption portion of the design implements the multiplication within GF(256) for the small coefficients. The implementation uses a succession of bit-shifts and the xtimes() operator to implement this multiplication. The xtimes() operator is defined by the authors of the Rijndael algorithm and is designed to allow for higher powers of x to computed in smaller steps. Each '1' bit within a byte being multiplied requires a shift and conditional XOR operation to complete the multiplication using the xtimes() operator. The encryption coefficients were intentionally chosen to be small so that the maximum number of repeated applications of the xtimes() operator that would be needed is two.

The decryption algorithm uses several look-up tables instead of implementing the multiplication in GF(256) for the larger numbers. Repeated application of the xtimes() operator for the much larger coefficients that are used during decryption would be harder to implement and would run much slower. Since the four coefficients needed for decryption are fixed, four lookup tables are needed to implement the multiplication in GF(256). Each table has 256 entries, one for each possible byte value that could be used in the multiplication. This implementation requires the use of memory which has a higher space requirement but is easier to implement and operates much faster.

2.4 AddRoundKey

The AddRoundKey operation performs a simple XOR between the current state data values and the round key for the current round. The round key is obtained from the key selector block based on the value of the expanded key and the current round.

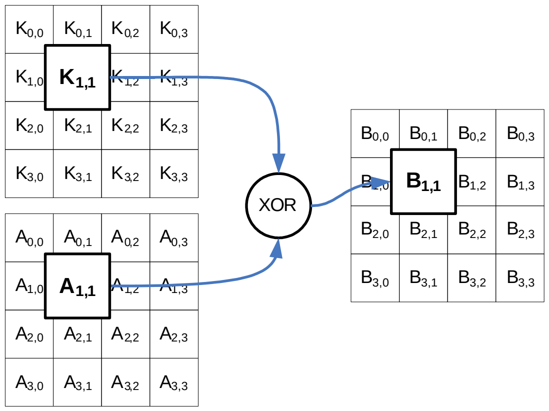

$K_{0,0}$ $K_{0,1}$ $K_{0,2}$ $K_{0,3}$
$K_{1,0}$ $K_{1,1}$ $K_{1,2}$ $K_{1,3}$
$K_{2,0}$ $K_{2,1}$ $K_{2,2}$ $K_{2,3}$
$K_{3,0}$ $K_{3,1}$ $K_{3,2}$ $K_{3,3}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
XOR
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

Figure 4: Round Key Bytes (K) XOR with Data Bytes (A)

3. Verilog Structure

The Verilog structure was designed to have function blocks that closely resembled the separate operational units within the algorithm. To this end, there is a separate module dedicated to the ByteSub, ShiftRow, MixColumns, and AddRoundKey operations. Additionally, there is a special module for both the key expansion and key selection operations, called KeyExpander and KeySelector. There are also two mapping modules. These modules are used to map the data from the input format as defined in the Rijndael specification to the data format used by this implementation. All these modules are tied together using the Rijndael control module, which provides the external interfaces for the project and implements the controller policies.

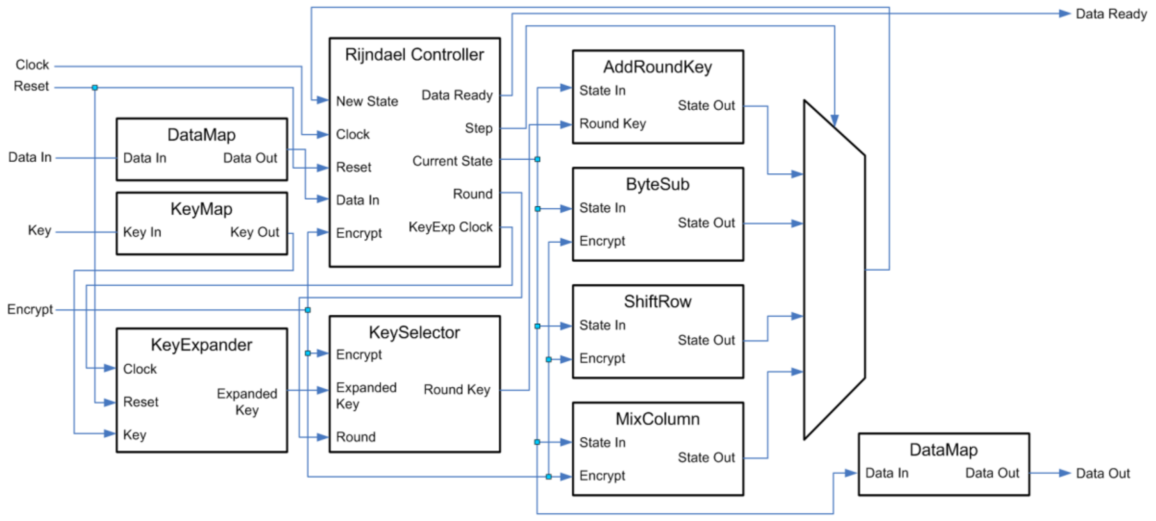


Figure 5: Verilog Structure Block Diagram

The ByteSub, ShiftRow, MixColumns, and AddRoundKey modules each implement a different feature as described in sections 2.1 through 2.4. Each takes as input the current system state as a 128-bit value and has a 128-bit output value containing the result of performing the appropriate operation. Each one also has a single bit input that controls whether encryption or decryption should be performed. Additionally, each of these modules also has a clock input that is used to control when the result of the operation moves to the output port. The AddRoundKey operation has one additional 128-bit input that contains the round key for the particular round.

The KeyExpander module is one of the most complicated portions of the system. The key expansion algorithm starts with the 128-bit input key and then computes the rest of the key iteratively in 32-bit chunks. The entire key size is 1408-bits. This provides enough key space for ten separate round keys and one extra key used prior to starting the first round (this key is the input key). The KeyExpander has three input signals and one output. There is a clock and reset input. The clock is used to control the KeyExpander. The entire operation of performing the key expansion takes several repeated iterations. These repeated iterations are handled using a built-in controller within the key expander algorithm which requires a clock signal to control the process of stepping through the key expansion. The reset signal is used to start the key expansion process. Reset can also be used to change to a new key after a previous key expansion has already been computed. This module also has an input that contains the 128-bit key from the user. The only output from this module is the resulting 1408-bit expanded key. The key expansion operation is the same for both encryption and decryption. The only differences for the keys come from the key scheduling, which feeds the keys into the system in a reverse order for decryption.

The key selector is a very simple module that implements the key scheduling algorithm. Based on the current round and whether the system is doing encryption or decryption, the key selector chooses the appropriate round key from the expanded key.

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

Figure 6: Data Matrix Representation

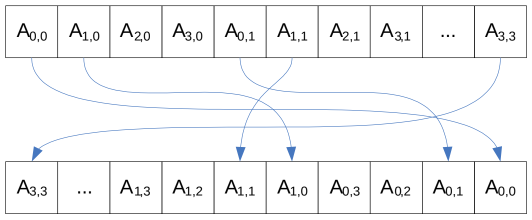

A0,0 A1,0 A2,0 A3,0 A0,1 A1,1 A2,1 A3,1 ... A3,3
A3,3 ... A1,3 A1,2 A1,1 A1,0 A0,3 A0,2 A0,1 A0,0

Figure 7: Data Mapping Scheme

The two mapping modules are very trivial. The input data from the user is specified to be in a “by column”, big endian format. For the module implementations the data is expected to be in a “by row”, little endian format. This transformation is achieved via a simple mapping between byte locations in the two formats. The data mapping can be seen in figure 7. The key mapping is used for a similar purpose. In the case of the key, the input is in a big-endian format and the mapping is used to reverse the byte order into a little-endian format.

4. Data and Control Paths

The controller for this implementation is contained in the Rijndael module. This module contains the external interface for the project. This includes inputs for the clock, reset, encrypt/decrypt selection, key, and data. The outputs are the result and a data ready signal. The clock signal is used to allow the system to progress through the required steps. The reset signal is used to initialize the system when started and can also be used later to begin a new encryption or decryption cycle. Once the system has finished performing the encryption or decryption, the data ready signal is set, and the result is placed on the output port.

The controller operates as a state machine, progressing from one state to the next in a fixed order based on whether the system is performing an encryption or decryption. The state of the system is tracked with two separate state variables. The first state variable is the round, this variable may be any of the ten encryption or decryption rounds as well as a special initialization round or a special output round. The second state variable is the step, this tracks what step is currently being performed in the current round.

The first round that the system goes through is the initialization round. The main task of the initialization round is for the controller to have the KeyExpander perform the key expansion routine. In addition to this, the initial XOR of the data with the first key is performed. The first few clock cycles after a reset are used to reset the KeyExpander and begin a new expansion based on the input key. Several cycles are needed at this point to finish the expansion process. Once this is complete, the controller stores the value of the expanded key and uses the key selector to retrieve the first key for the encryption or decryption process. The first key is XORed with the input data, and the system is then ready to begin the several rounds needed for encryption or decryption.

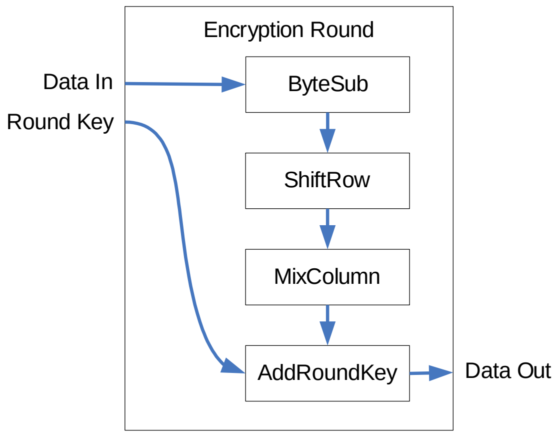


Figure 8: Encryption Round Operation Order

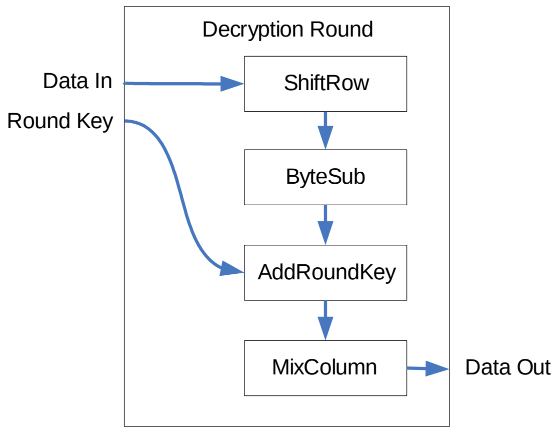


Figure 9: Decryption Round Operation Order

The order or operations for encryption and decryption is different. In either case, there are four steps that need to be completed during each round. The controller stores the value of the current state. This value is used as the input for every sub-module in the system. This means that at every step the value of the ByteSub, ShiftRow, MixColumns, and AddRoundKey operations are computed. These are all computed in parallel, so there is no performance penalty for computing all these values even though only one is needed. At each step, the controller chooses the output from the appropriate module and saves this as the new current state. Also, it updates the current round key in the step prior to the AddRoundKey operation. The order of operations for encryption and decryption can be seen in figures 8 and 9.

After all the required rounds have been completed the controller outputs the result and sets the data ready signal. After this point, the system will remain in a state outputting the result until a reset signal is received.

5.Verilog Modules / Code:

1. **AddRoundKey.v** -> Implementation of AddRoundKey module, used to XOR data with the round key
2. **ByteSub.v** -> Implementation of ByteSub module, used to perform the s-box substitutions
3. **Constants.v** -> Contant definitions, such as number of data blocks and number or rounds
4. **DataMap.v** -> Implementation of DataMap module, used to conver data format
5. **KeyExpander.v** -> Implementation of KeyExpander module, used to expand 128-bit input key to 1408-bits
6. **KeyMap.v** -> Implementation of KeyMap module, used to reverse byte order of input key

7. **KeySelector.v** -> Implementation of KeySelector module, used to perform the key scheduling
8. **MixColumn.v** -> Implementation of MixColumn module, used to perform the multiplication and modulus in GF(256)
9. **Rijndael.v** -> Implementation of the Rijndael module, this is the main module that contains the controller
10. **ShiftRow.v** -> Implementatino of the ShiftRow module, this module perform byte rotation on the rows of data
11. **Testbench.v**

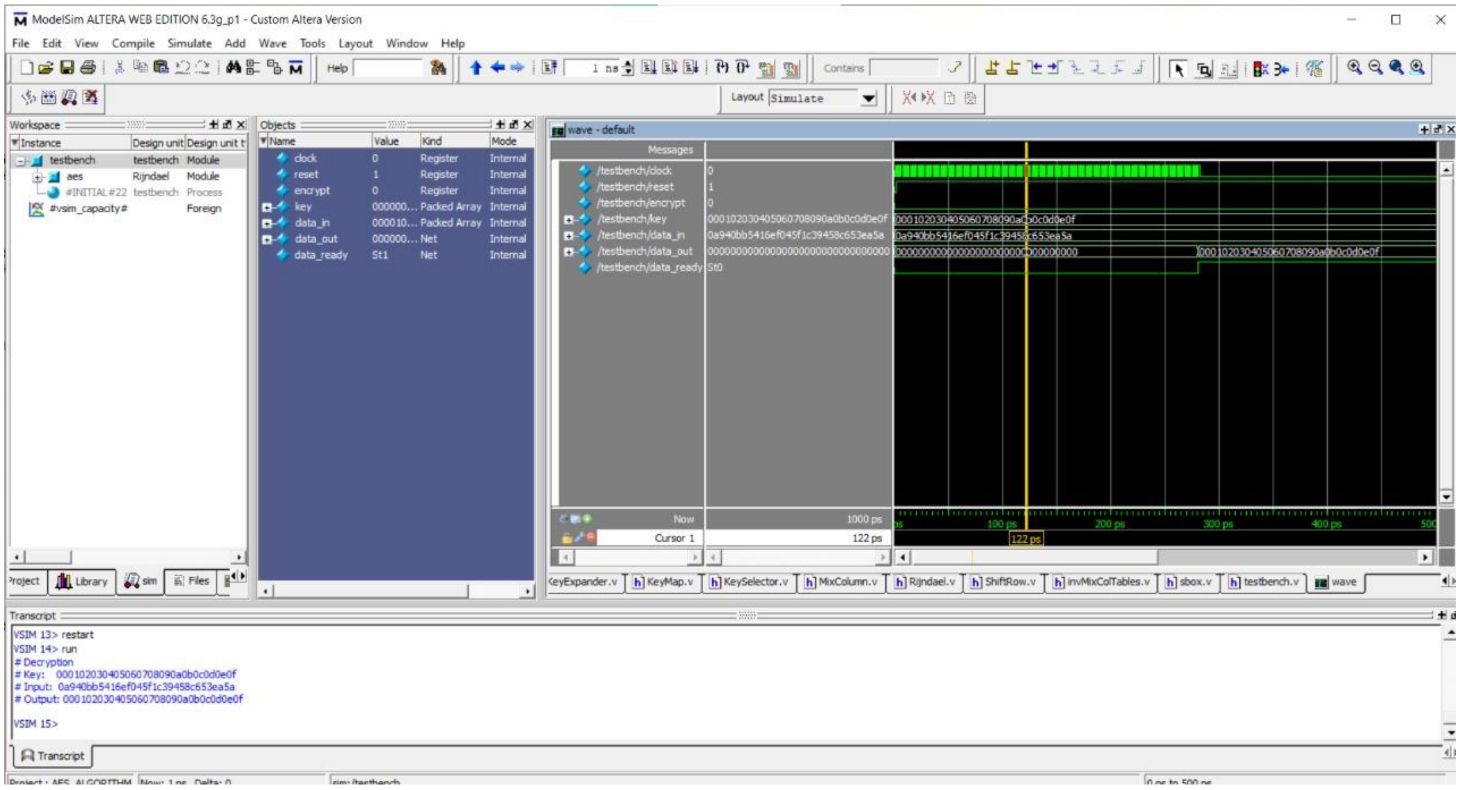
5. Code:-

```
`include "Constants.v"

/**************************************************************************
* testbench - This is a non-synthesizable module that can be used to run a
*            test bench operation on the Rijndael encryption method that
*            was implemented. It simply sets the input data and key, sends
*            a reset signal, and then supplies the required clock signal
*            to allow the encryption/decryption to take place. Once the
*            operations are done, the results can be verified by the user.
**************************************************************************/
module testbench;
  reg clock, reset, encrypt;
  reg[`KeyBits - 1:0] key;
  reg[`DataBits - 1:0] data_in;
  wire[`DataBits - 1:0] data_out;
  wire data_ready;
  reg[3:0] i;
  //This is the instatiatoin of the main AES encryption module (Rijndael)
  Rijndael aes(clock, reset, encrypt, key, data_in, data_out, data_ready);
  initial
  begin
    //Set the data/key to use
    data_in = 128'h0A940BB5416EF045F1C39458C653EA5A;
    //data_in = 128'h000102030405060708090A0B0C0D0E0F;
    key =     128'h000102030405060708090A0B0C0D0E0F;
    //Cause the system to reset
    reset = 0;
    clock = 0;
    //Select encryption/decryption
    encrypt = 0;
    //One clock cycle to allow the reset to occur, then clear the reset
    #1 clock = 1;
    #1 reset = 1;
    clock = 0;
    //Provide a clock signal until the operation is completed
    while (data_ready == 0)
    begin
      #1 clock = 1;
      #1 clock = 0;
    end
    //Show the results of the operation
    if (encrypt == 1)
      $display("Encryption");
    else
      $display("Decryption");
    $display("Key:    %h", key);
    $display("Input:  %h", data_in);
    $display("Output: %h", data_out);
  end
endmodule
```

6. Simulation Results

The correctness of the Verilog model was tested using simulation in both Quartus II and ModelSim. The testing was done using the test vectors provided by the NIST. Both the encryption and decryption functions were tested. The results from one test encryption are shown in figure 10.


ModelSim ALTERA WEB EDITION 6.3g_p1 - Custom Altera Version

Name	Value	Kind	Mode
clock	0	Register	Internal
reset	1	Register	Internal
encrypt	0	Register	Internal
key	000000...	Packed Array	Internal
data_in	000010...	Packed Array	Internal
data_out	000000...	Net	Internal
data_ready	St1	Net	Internal

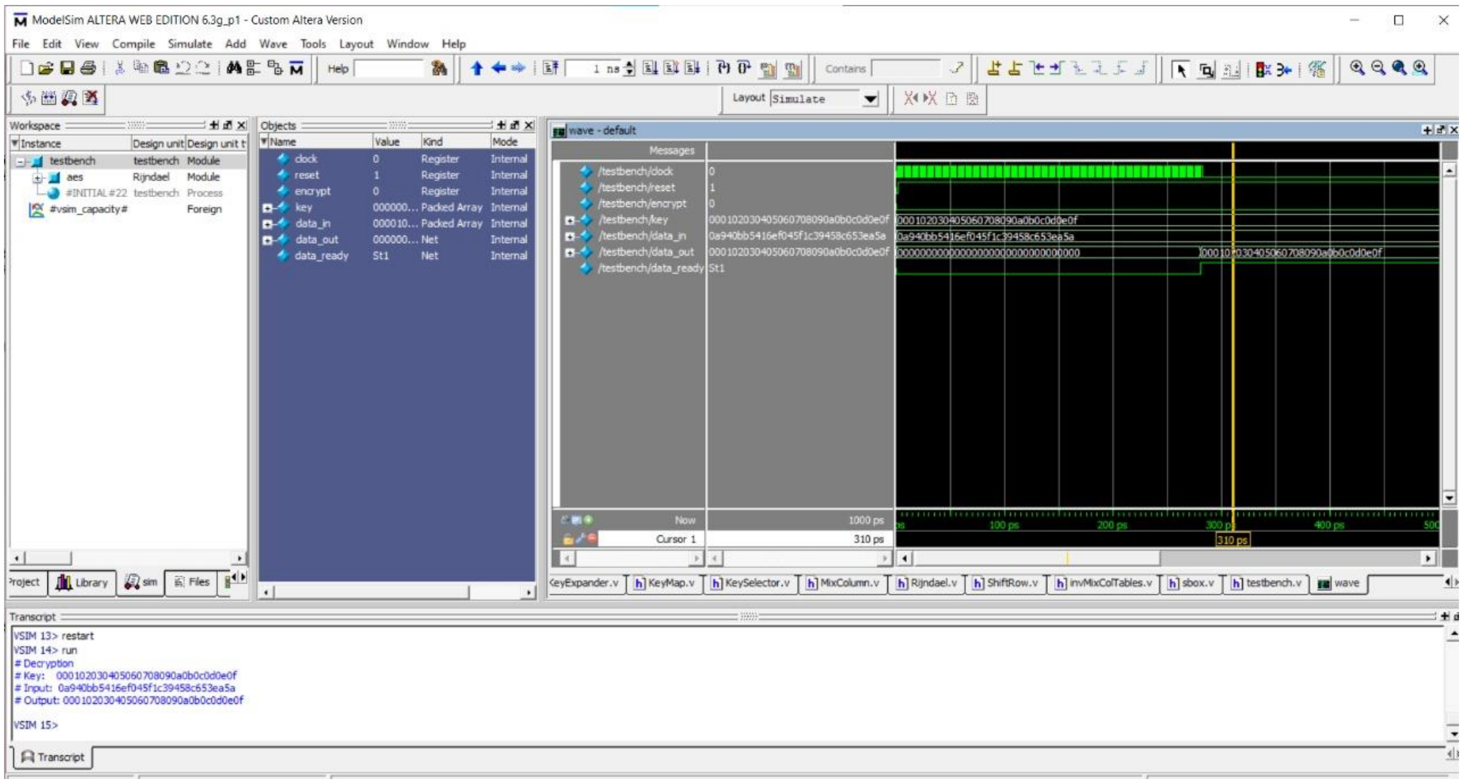
Messages	
/testbench/clock	0
/testbench/reset	1
/testbench/encrypt	0
/testbench/key	000102030405060708090a0b0c0d0e0f
/testbench/data_in	0a940bb5416ef045f1c39458c653ea5a
/testbench/data_out	00000000000000000000000000000000
/testbench/data_ready	St0

Now 1000 ps
Cursor 1 122 ps

```
VSIM 13> restart
VSIM 14> run
# Decryption
# Key:    000102030405060708090a0b0c0d0e0f
# Input:  0a940bb5416ef045f1c39458c653ea5a
# Output: 000102030405060708090a0b0c0d0e0f

VSIM 15>
```

Until the data_ready bit is logic Zero Encryption took place and the cipher text generated is internally passed to the input of the Decryption part that start after data_ready bit goes logic high



Name	Value	Kind	Mode
clock	0	Register	Internal
reset	1	Register	Internal
encrypt	0	Register	Internal
key	000000...	Packed Array	Internal
data_in	000010...	Packed Array	Internal
data_out	000000...	Net	Internal
data_ready	St1	Net	Internal

```
/testbench/clock       0
/testbench/reset       1
/testbench/encrypt     0
/testbench/key         000102030405060708090a0b0c0d0e0f
/testbench/data_in     0a940bb5416ef045f1c39458c653ea5a
/testbench/data_out    000102030405060708090a0b0c0d0e0f
/testbench/data_ready  St1
```

```
VSIM 13> restart
VSIM 14> run
# Decryption
# Key:    000102030405060708090a0b0c0d0e0f
# Input:  0a940bb5416ef045f1c39458c653ea5a
# Output: 000102030405060708090a0b0c0d0e0f

VSIM 15>
```

Figure 10: Encryption Test Waveform

The timing analysis revealed that the maximum clock frequency on this device would be 46.82 MHz. Since a complete cryptographic operation requires 140 clock cycles, the device is capable of a maximum throughput of 42.57 Mbps operating on 128-bit blocks of data at a time. It would be simple to modify the design to achieve a much faster encryption rate for encrypting or decryption multiple blocks with the same key by modifying the system to skip the key expansion phase on subsequent operations with the same initial key. This would eliminate almost 1/3 of the computation time.

7. References

[1] C. Kaufman, R. Perlman, M and Speciner, *“Network Security: PRIVATE communication in a PUBLIC world”*, pg.206-208, Prentice-Hall, 2002.

[2] National Institute of Standards and Technology (NIST), AES Test Vectors, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/rijndael-vals.zip.

[3] J. Daemon and V. Rijman, *“AES Proposal: Rijndael”*, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.