Indian Institute of Technology Tirupati
Electrical CAD LAB

ADVANCED ENCRYPTION STANDARD(AES)

ALEKHYA GEDAM - EE17B012
SHRADDHA SAHOO - EE17B027

1) HISTORY:

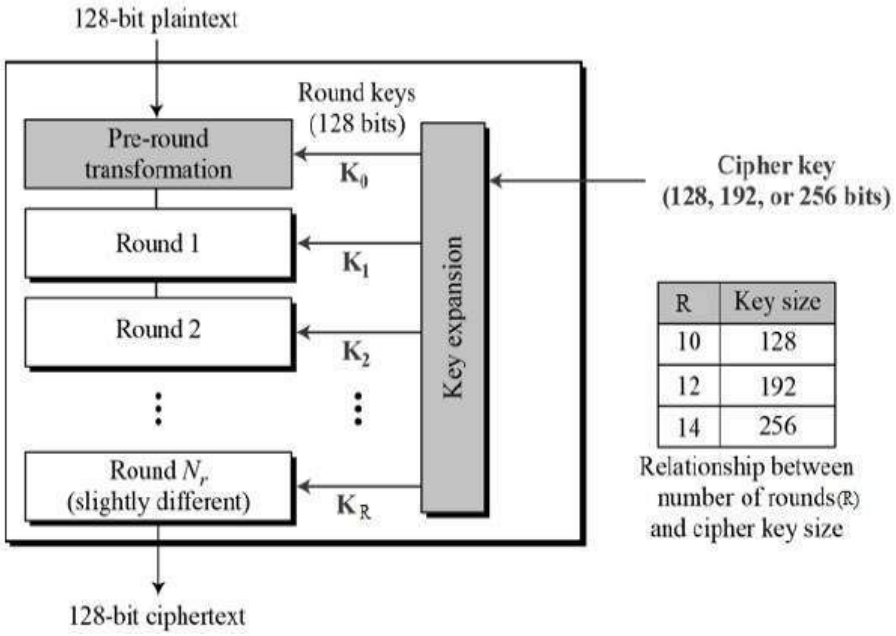
- On January 2, 1997, NIST announced the initiation of an effort to develop the AES and made a formal call for algorithms on September 12, 1997.
- The call indicated NIST's goal that the AES would specify an unclassified, publicly disclosed encryption algorithm, available royalty-free, worldwide.
- The algorithm would have to implement symmetric key cryptography as a block cipher and support a block size of 128 bits and key sizes of 128, 192, and 256 bits.
- On August 20, 1998, NIST announced 15 AES candidate algorithms at the First AES Candidate Conference (AES1) and put in for public comments on the candidates.
- Industry and academia submitters from twelve countries proposed the fifteen algorithms. A Second AES Candidate Conference (AES2) was held in March 1999 to discuss the results of the analysis that was conducted by the international cryptographic community on the candidate algorithms.
- In August 1999, NIST announced its selection of five finalist algorithms from the fifteen candidates. The selected algorithms were MARS, RC6, Rijndael, Serpent and Twofish.
- Out of five, one was selected and today AES is based on the Rijndael cipher developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen.

2) OVERVIEW OF IMPLEMENTATION:

- **For Encryption**

AES encryption requires 2 inputs, plain text of 128 bits and key(128 bits/ 192 bits/256 bits) to give an output of 128 bits cipher text.

Every key size has a different number of rounds. For example key size of 128 bits requires 10 rounds. It can be seen in the small table below. There is a pre round as well so that makes 1+10 = 11 rounds for 128 bits key. Similarly for other key sizes. Each round requires a key which is in fact the modification of the original key. How the key is modified is given by the key expansion box in the figure below.



R	Key size
10	128
12	192
14	256

Relationship between number of rounds(R) and cipher key size

Fig 1

There are two main things that happen in AES:

- Key Generation
- Rounds

In key generation:

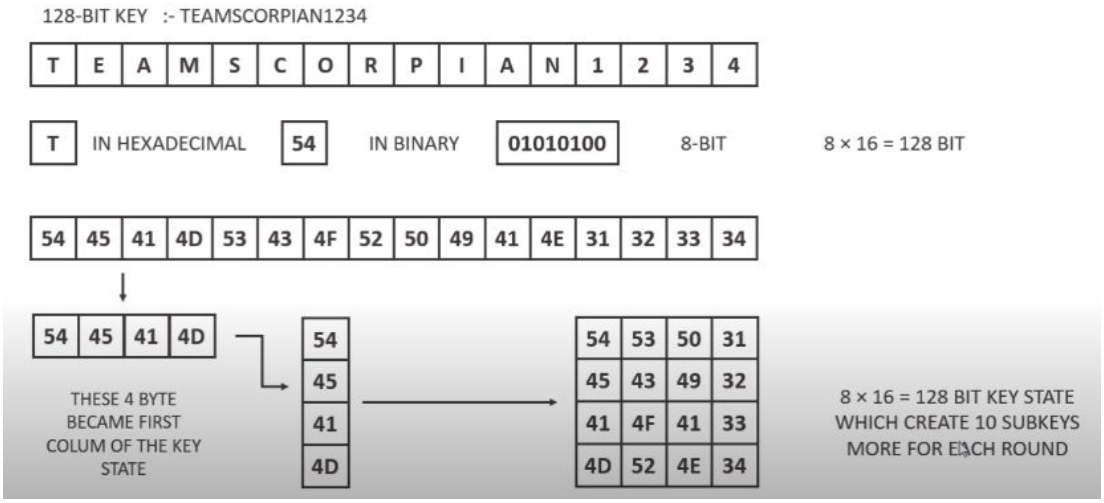


Fig 2

For a key size of 128 bits, that means it has 16 characters. Each character is converted into hexadecimal as shown in the above figure. This matrix makes the key state 0.

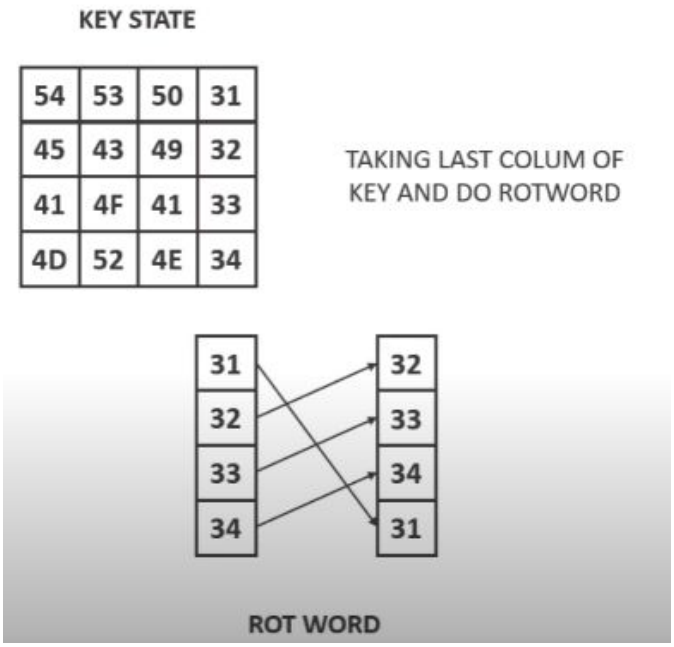


Fig 3

Last column of the key state is taken and its ROT word is found out by shifting the column by one byte as shown above.Then each byte is replaced by its corresponding sub byte.

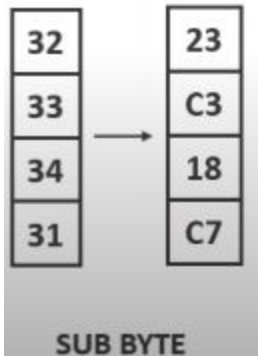


Fig 4

This table is used for this purpose which is called the Rijndael S-box.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig 5

After doing the subbyte operation, (if it is the 1st round) it is XORed with the first column of the Rcon table.

01	02	04	08	10	20	40	80	1B	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Fig 6

Then it is XORed with the first column of the key state to give the first column of the key state 1. Then the first column of the key state 1 is XORed with the second column of key state 0 to give the second column of the key state 1. The third column of key state 0 is XORed with the second column of the key state 1 to form the third column of the key state 1 and so on. This happens till the 4th column of the key state 1. Sub byte operation happens with that column. The result is then XORed with the 5th column of the key state 0 to form the 5th column of key state 0 and so on.

Note this happens with a key size of 256 bits where the number of columns is 8. For a 128 bit key, the process stops at column 4 of the key state 1.

This was for the key for the first round. In the key state 2 , similar things happens with key state 1 using the second column of the rcon table and so on.
For a 256 bit key, the key is divided into two 128 bit keys and then used in the add round operation.

In rounds, there are 3 types of rounds:

- Initial round
- Main round
- Last round

In the initial round, the message block is XORed with Key 0 and the output is passed to the main round. This process is also called the add Round key.

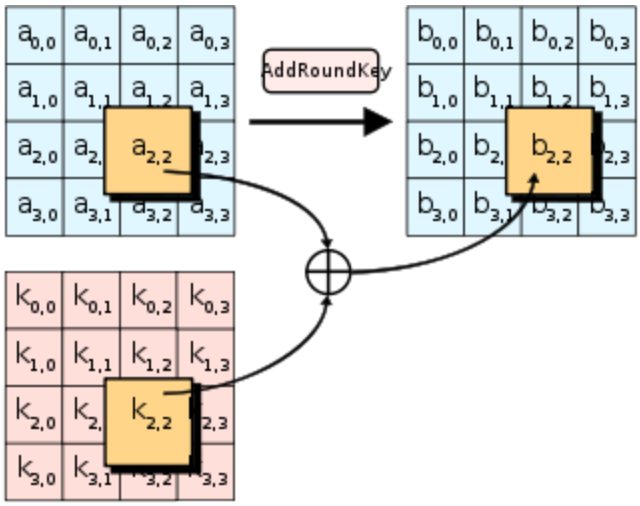


Fig 7

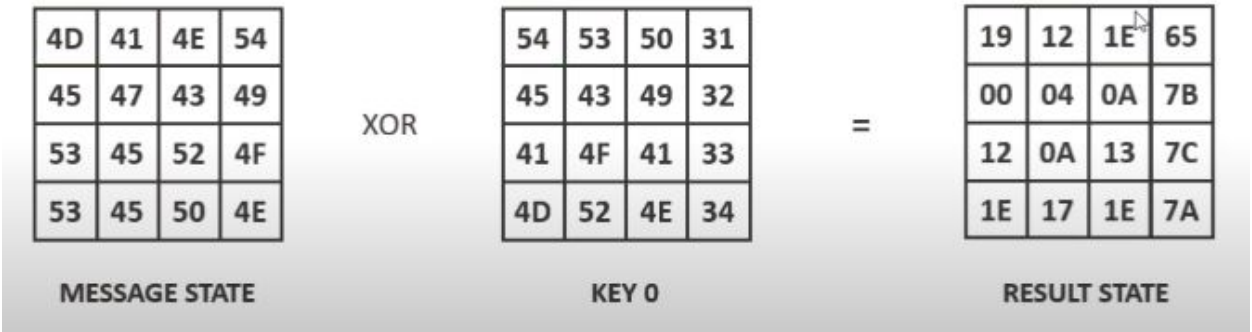


Fig 8

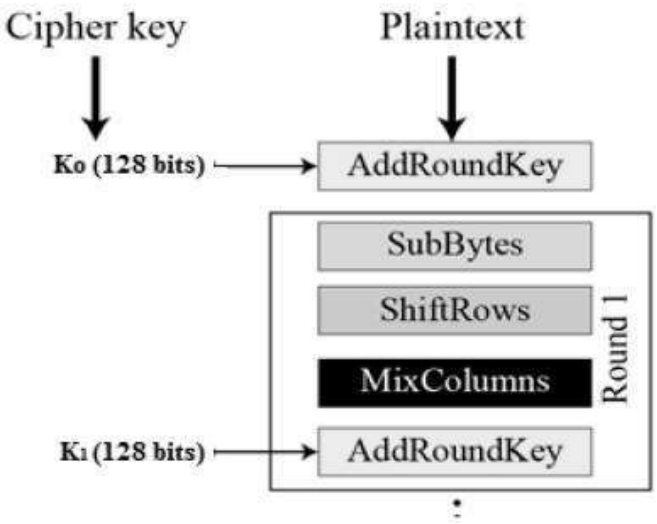


Fig 9

In the main round, the operations are done in the following manner(from round 1 to N-1 round):

- Sub byte
- Shift rows
- Mix columns
- Add round key

In sub byte, every byte from the output of the previous round is replaced with its subbyte.

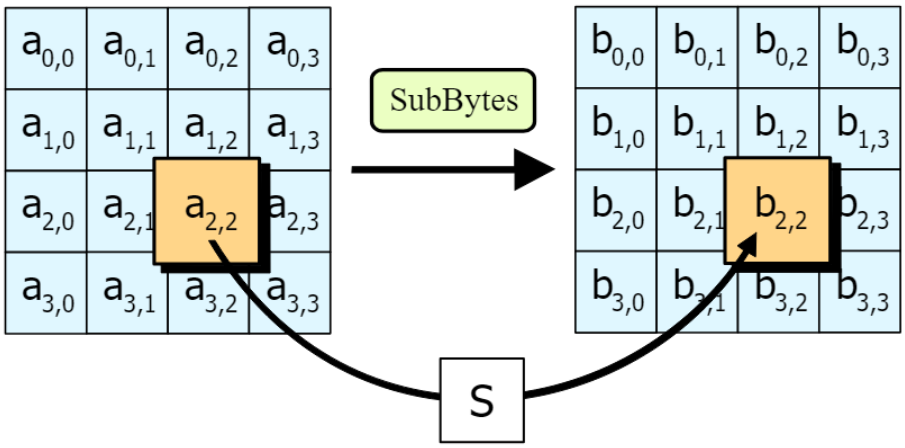


Fig 10

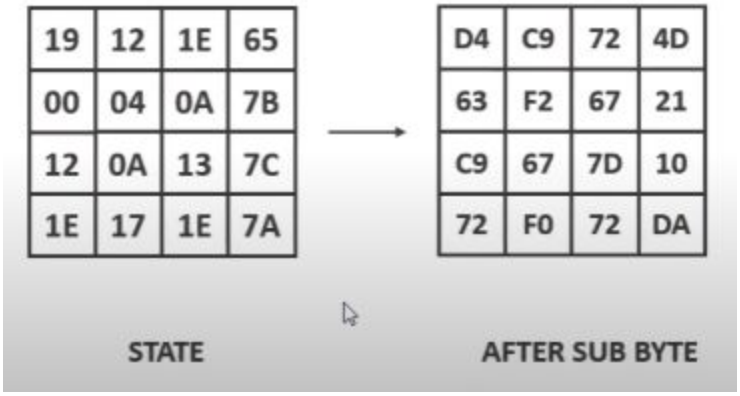


Fig 11

In shift rows, each row is shifted by some particular bytes.

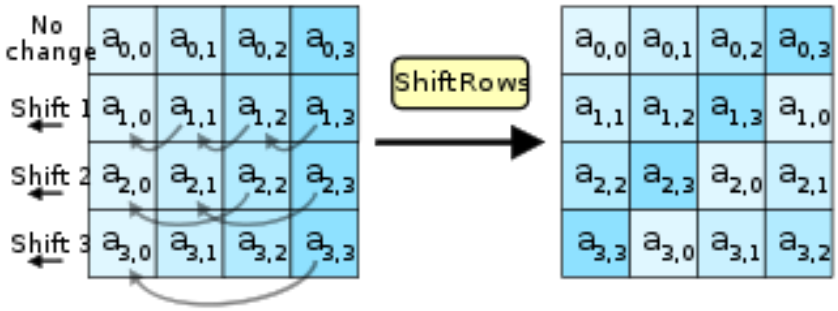


Fig 12

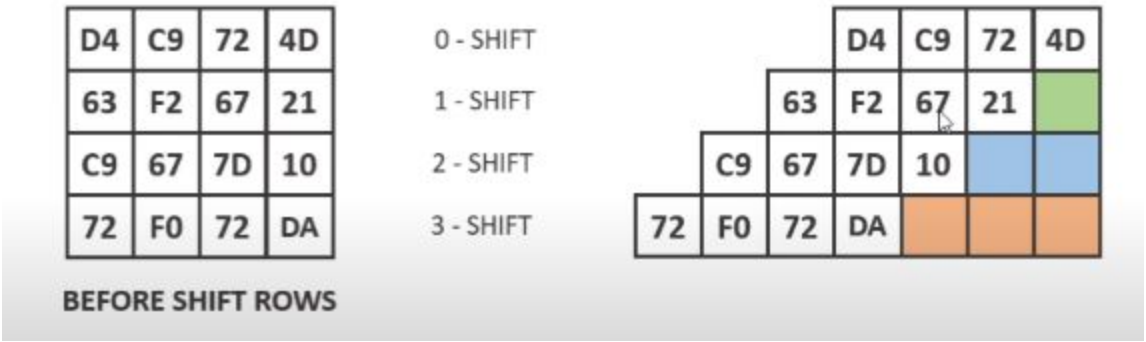


Fig 13

In mix columns, each column is multiplied with a predefined matrix.

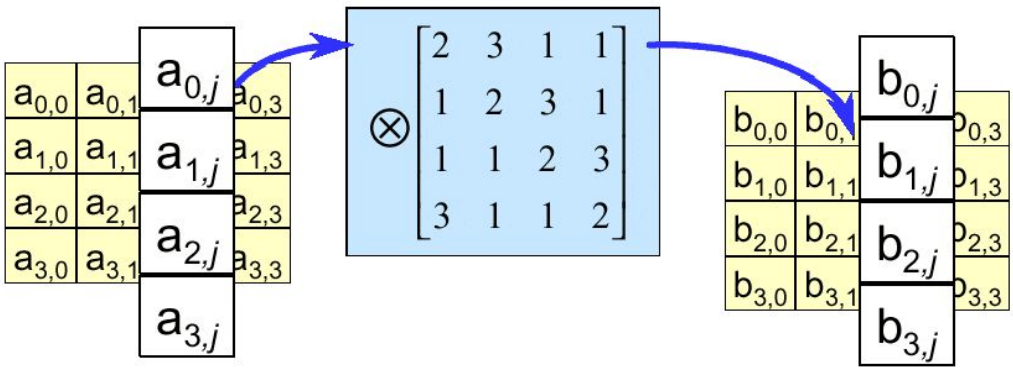


Fig 14

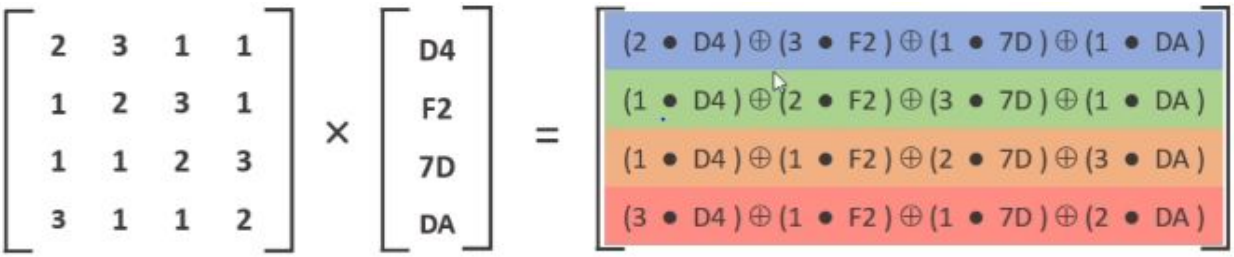


Fig 15

If a product is bigger than a byte, then it is reduced by using a reduce polynomial. That is simply by doing XOR with 100011011 till we get the result as less or equal to 8 bits.

In the last round, operations are done in the following manner:

- Sub byte
- Shift rows
- Add last round key

- **For Decryption**

AES decryption is similar to AES encryption where it takes the cipher text and the same key to give the original message.

Following changes are seen in AES decryption:

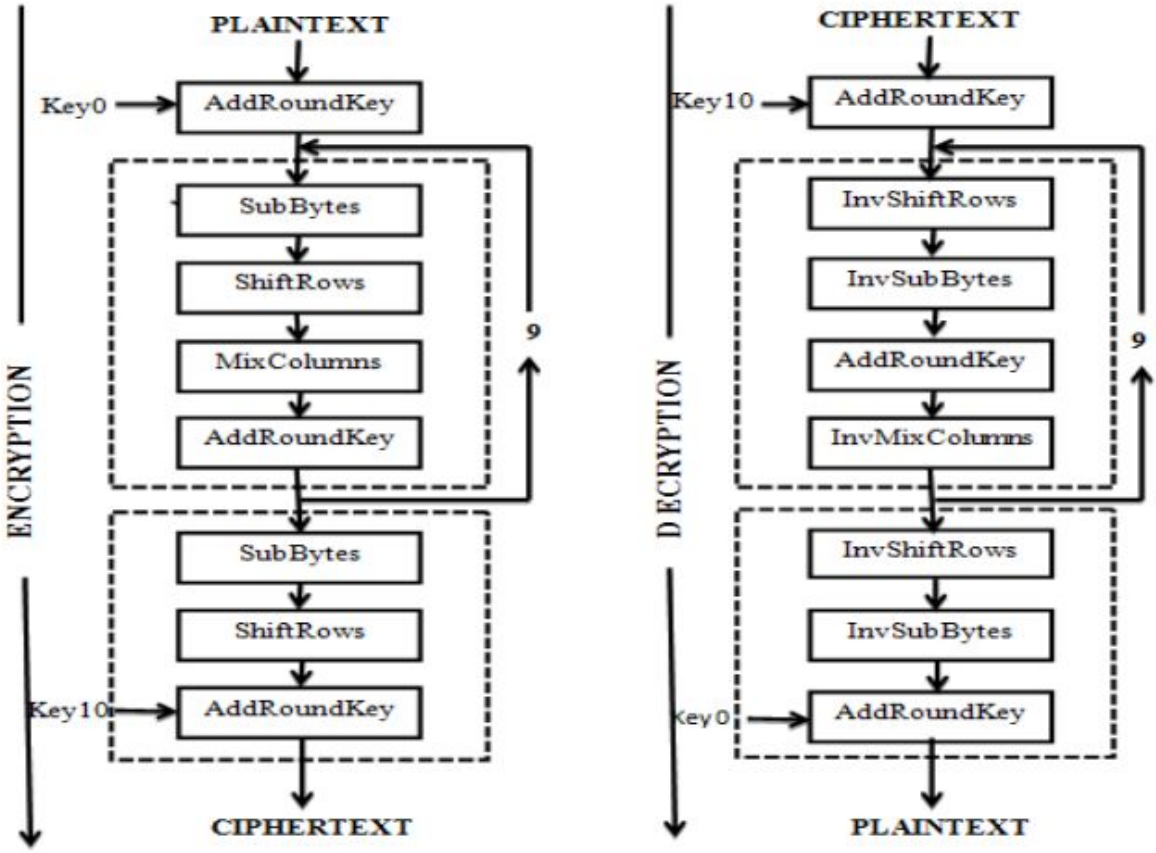


Fig 16

- The figure shows how an encryption is different from decryption.
- Here the same keys(Key0,Key1....) are used but in reverse order. For the first add round key step, key 14 is used.
- For 256 bits key, there are 14 rounds similarly for 128 bits key(10 rounds).
- Main round of decryption has the following steps in order:
    - InvShiftRows
    - InvSubByte
    - AddRoundKey
    - InvMixColumns
- Last round has the following steps in order:
    - InvShiftRows
    - InvSuByte
    - AddRoundKey

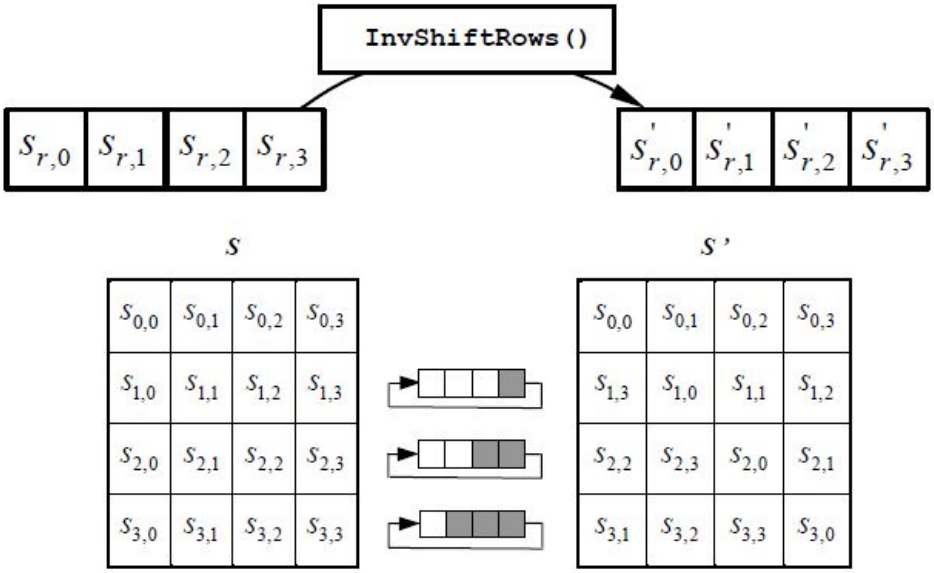


Fig 17

In invShiftRows, each row is shifted by some particular bytes in the opposite direction as compared to shiftRows.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Fig 18

In invSubByte, every byte from the output of the previous round is replaced with its inverse Subbyte. Fig 18 shows the inverse SubByte table.

In invMixColumns, each column is multiplied with a predefined matrix.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Fig 19

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

Fig 20

3) VERILOG IMPLEMENTATION:

In our implementation of the AES algorithm, we have made the following modules using verilog:

- main
- aesE
- initialRound
- mainRound
- lastRound
- KeyGenerate
- addRoundkey
- mixColumns
- shiftRows
- subByte2
- galois

- rcon
- sbox
- Subbyte
- aesD
- invGalois
- invLastRound
- invMainRound
- invMixColumns
- invSbox
- invShiftRows
- invSubByte2
- invSubByte

Please note that in addition to the above modules we have also added an extra python code that converts any given text into hexadecimal. The hexadecimal converted message can be then used in our code for encryption. Name of the python code : **text_to_hexa.py**

If you want to convert hexadecimal converted message back to text you can use this python code: **hex_to_text.py**

3.1) main

It is the main file where we give our input message of 128 bits and key of 256 bits in hexadecimal form. It returns the cipher text of 128 bits. It uses *aesE* module. Then we give the cipher text to get back our original message this is done by *aesD* module.

3.2) aesE

It is the module in which carries out the encryption of the message. It takes the message and key as input and gives the cipher text as output. During the starting of code, it generates all the keys that are required in the (1 + 14) rounds. It uses the *KeyGenerate* module for this action. The keys are then passed to the initial, main and the last rounds. Initial, main and last round have their separate modules.

3.3) initialRound

It takes the message and key as input and gives an output which is passed to the main round. It performs the add round key action on the message using the key. Add round key is done using the *addRoundkey* module.

3.4) mainRound

It takes the output from the initial round or the previous main round and the key for that particular round as input and gives an output for the next main round. It first performs the sub byte operation, next shift rows, next mix columns and then finally add round key operation. Sub byte, shift rows,mix columns and add round keys have their own modules: *subByte2, shiftRows, mixColumns* and *addRoundkey* module respectively.

3.5) lastRound

It is the final round(14th) where it takes the output of the previous main round and the last key as input and gives the final cipher text as the output. It first performs the sub byte operation, next shift rows and then finally add round key operation. Sub byte, shift rows and add round keys have their own modules: *subByte2, shiftRows* and *addRoundkey* module respectively.

3.6) KeyGenerate

In this module first it divides the 128 bits into 8 columns(parts) with 32 bits each. It takes the last column, finds its rot word and replaces each byte with sub byte using *subbyte* module. The result is XORed with the column of Rcon table of that particular round. The result is then XORed with the first column. This forms the first column of the output. The second column of input is XORed with the first column of the output to form the second column of the output. The third column of input is XORed with the second column of the output to form the third column of the output and so on. This happens till the 4th column of the output. Sub byte operation happens with the column. The result is then XORed with the 5th column of the input to form the 5th column of the output and so on.

3.7) addRoundkey

It takes the 128 bit input and performs XOR operation with the input key and gives the result as the output which is again of 128 bits.

3.8) mixColumns

This performs the mix columns operation on the 128 bits inputs and gives the result as output. It divides the 128 bits into 4 columns of 32 bits each. Matrix multiplication takes place between the predefined matrix and the 4 bytes present in each column. This matrix multiplication happens with all the columns.

3.9) shiftRows

It performs the shift row operation on the 128 bits input and gives the result as output. It divides the 128 bits into 16 parts with 8 bits in each part. Then it takes each part and places it in its respective position in the output. For example from 80th to 87th bit of the input (assuming LSB is at 0) is given to 112th to 119th bit of the output.

3.10) subByte2

It performs the sub byte operation on the 128 bits input and gives the result as output. It divides the 128 bits into 16 parts with 8 bits in each part. Then it takes each part and replaces it with its corresponding value in the Sub byte table. It uses the *subbyte* module for its operation.

3.11) galois

During the mix columns operation, when each byte of the 128 bit is multiplied with 1,2 and 3, the expected output for that byte is picked from this *galois* module. For every possible combination of a byte what would be the output is stored in this module. So when you give an input byte, it returns the expected byte.

3.12) rcon

This module stores the rcon table values. So when you give a x as input, it returns the corresponding column values of that xth column.

3.13) sbox

This module stores the values that are present in a Sub byte table. So when you give an input byte, it returns its expected sub byte.

3.14) subbyte

It performs the sub byte operation on the 32 bits input and gives the result as output. It divides the 32 bits into 4 parts with 8 bits in each part. Then it takes each part and replaces it with its corresponding value in the Sub byte table. It is different from subByte2 operation because the former uses this particular module 4 times to replace each byte in a 128 bits input . This module uses the *sbox* module.

3.15) aesD

It is the module in which carries out the decryption of the cipher text. It takes the ciphertext and key as input and gives the actual message as output. During the starting of code, it generates all the keys that are required in the (1 + 14) rounds. It uses the *KeyGenerate* module for this action. The keys are then passed to the initial, inverse main and the inverse last rounds. Initial, inverse main and inverse last round have their separate modules.

3.16) invGalois

During the inverse mix columns operation, when each byte of the 128 bit is multiplied with 9,11,13 and 14, the expected output for that byte is picked from this *invGalois* module. For every possible combination of a byte what would be the output is stored in this module. So when you give an input byte, it returns the expected byte.

3.17) invLastRound

It is the final round(14th) where it takes the output of the previous inverse main round and the zeroth key as input and gives the final message as the output. It first performs the inverse sub byte operation, next inverse shift rows and then finally add round key operation. Inverse sub byte, inverse shift rows and add round keys have their own modules: inv*SubByte2, invShiftRow*s and *addRoundkey* module respectively.

3.18) invMainRound

It takes the output from the initial round or the previous main round and the key for that particular round as input and gives an output for the next main round. It first performs the inverse sub byte operation, next inverse shift rows, next add round key and then finally

inverse mix columns operation. Inverse sub byte, inverse shift rows, inverse mix columns and add round keys have their own modules: *invSubByte2, invShiftRows, invMixColumns* and *addRoundkey* module respectively.

3.19) invMixColumns

This performs the inverse mix columns operation on the 128 bits inputs and gives the result as output. It divides the 128 bits into 4 columns of 32 bits each. Matrix multiplication takes place between the predefined matrix and the 4 bytes present in each column. This matrix multiplication happens with all the columns.

3.20) invSbox

This module stores the values that are present in an inverse Sub byte table. So when you give an input byte, it returns its expected sub byte.

3.21) invShiftRows

It performs the inverse shift row operation on the 128 bits input and gives the result as output. It divides the 128 bits into 16 parts with 8 bits in each part. Then it takes each part and places it in its respective position in the output. For example from 80th to 87th bit of the input (assuming LSB is at 0) is given to 48th to 55th bit of the output.

3.22) invSubByte2

It performs the inverse sub byte operation on the 128 bits input and gives the result as output. It divides the 128 bits into 16 parts with 8 bits in each part. Then it takes each part and replaces it with its corresponding value in the inverse Sub byte table. It uses the *invSubByte* module for its operation.

3.23) invSubByte

It performs the inverse sub byte operation on the 32 bits input and gives the result as output. It divides the 32 bits into 4 parts with 8 bits in each part. Then it takes each part and replaces it with its corresponding value in the inverse Sub byte table. It is different from invSubByte2 operation because the former uses this particular module 4 times to replace each byte in a 128 bits input . This module uses the *invSbox* module.

4) EXPERIMENTAL RESULT:

The following inputs we gave to our main.v file.

PLAINTEXT: **00112233445566778899aabbccddeeff**

KEY: **000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f**

FINAL OUTPUT 1(CIPHER TEXT): **8ea2b7ca516745bfeafc49904b496089**

FINAL OUTPUT 2(ORIGINAL PLAIN TEXT): **00112233445566778899aabbccddeeff**

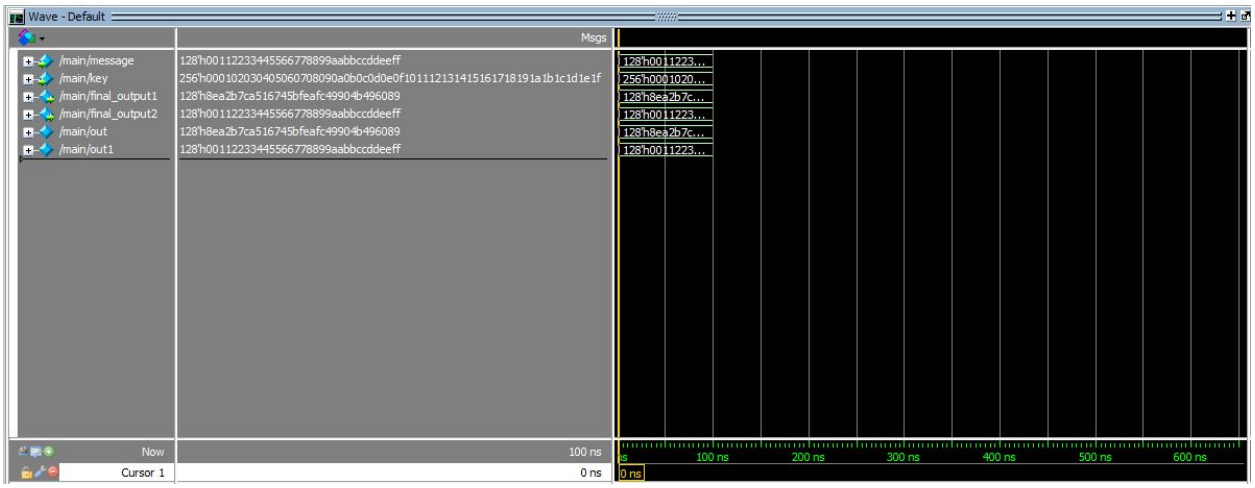


Fig 21. Final Output

Key Generation simulations:

Key 0: **000102030405060708090a0b0c0d0e0f**

Key 1: **101112131415161718191a1b1c1d1e1f**

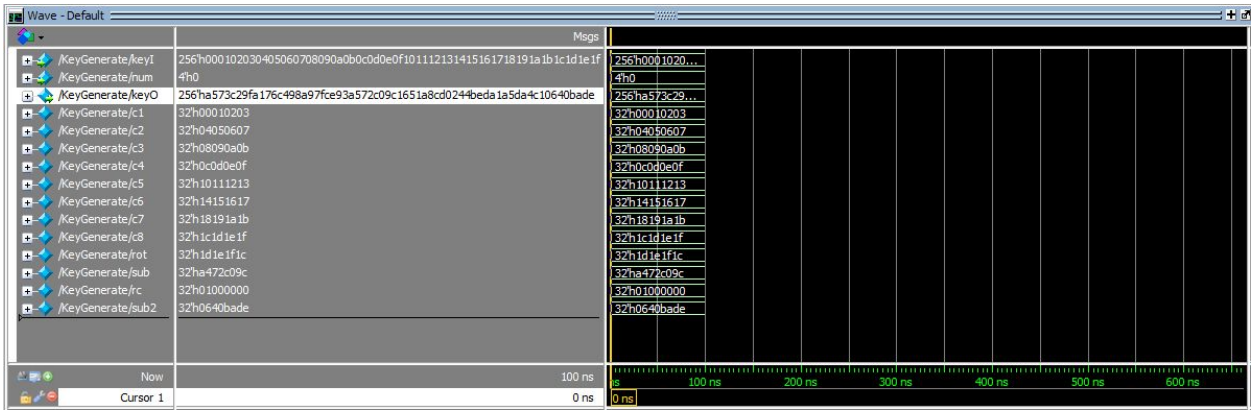


Fig 22. Output for Key 2 and Key 3

Key 2: **a573c29fa176c498a97fce93a572c09c**

Key 3: **1651a8cd0244beda1a5da4c10640bade**

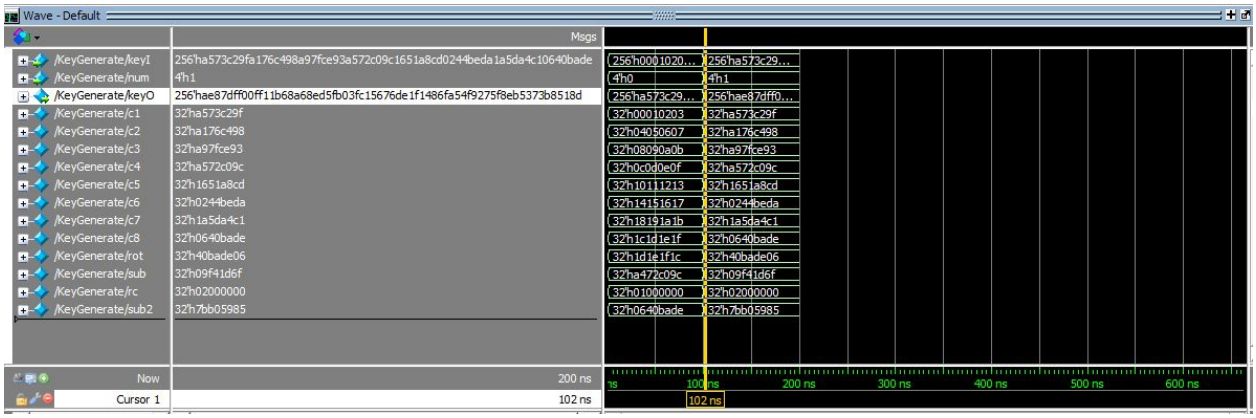


Fig 23. Output for Key 4 and Key 5

Key 4: **ae87dff00ff11b68a68ed5fb03fc1567**

Key 5: **6de1f1486fa54f9275f8eb5373b8518d**

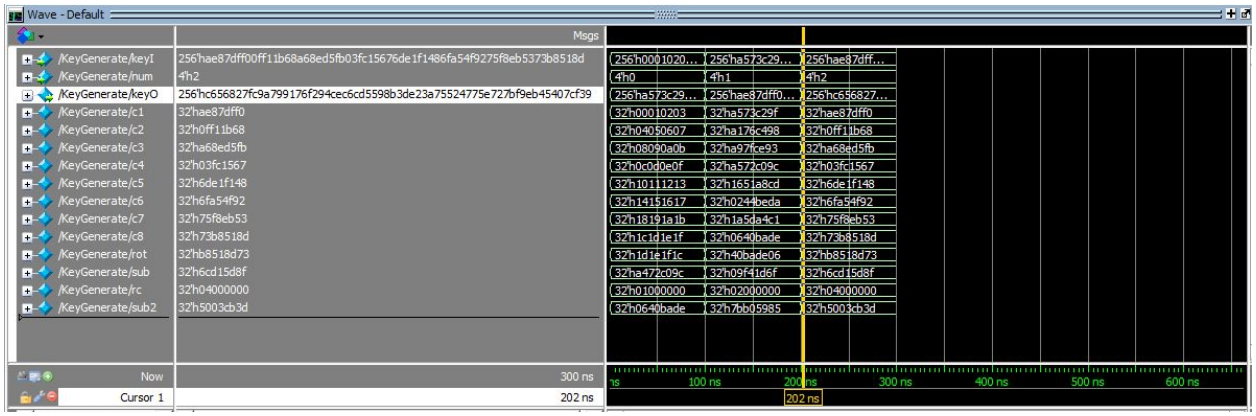


Fig 24. Output for Key 6 and Key 7

Key 6: **c656827fc9a799176f294cec6cd5598b**

Key 7: **3de23a75524775e727bf9eb45407cf39**

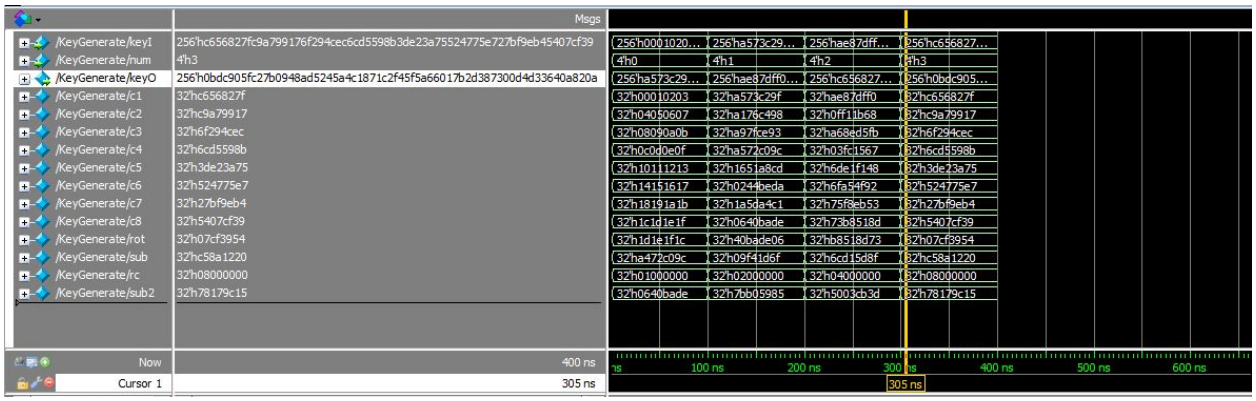


Fig 25. Output for Key 8 and Key 9

Key 8: **0bdc905fc27b0948ad5245a4c1871c2f**

Key 9: **45f5a66017b2d387300d4d33640a820a**

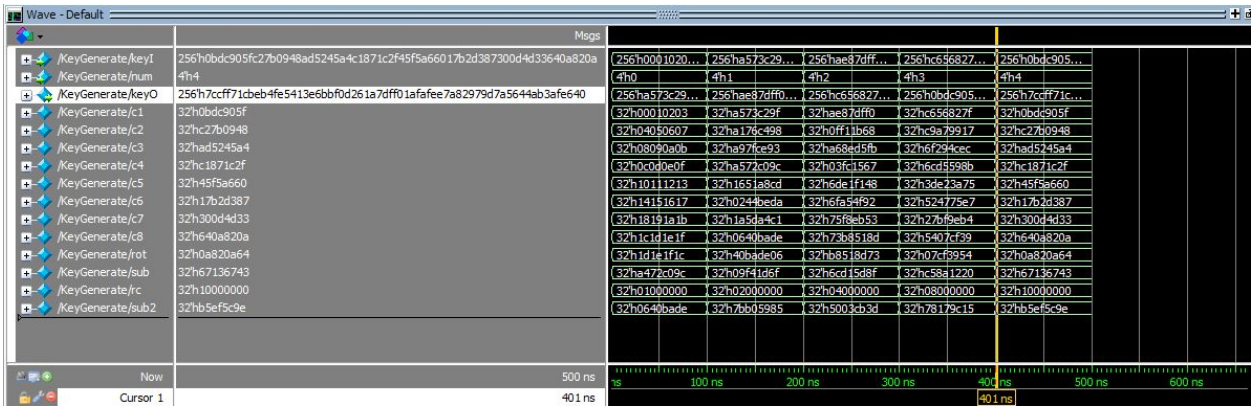


Fig 26. Output for Key 10 and Key 11

Key 10: **7ccff71cbeb4fe5413e6bbf0d261a7df**

Key 11: **f01afafee7a82979d7a5644ab3afe640**

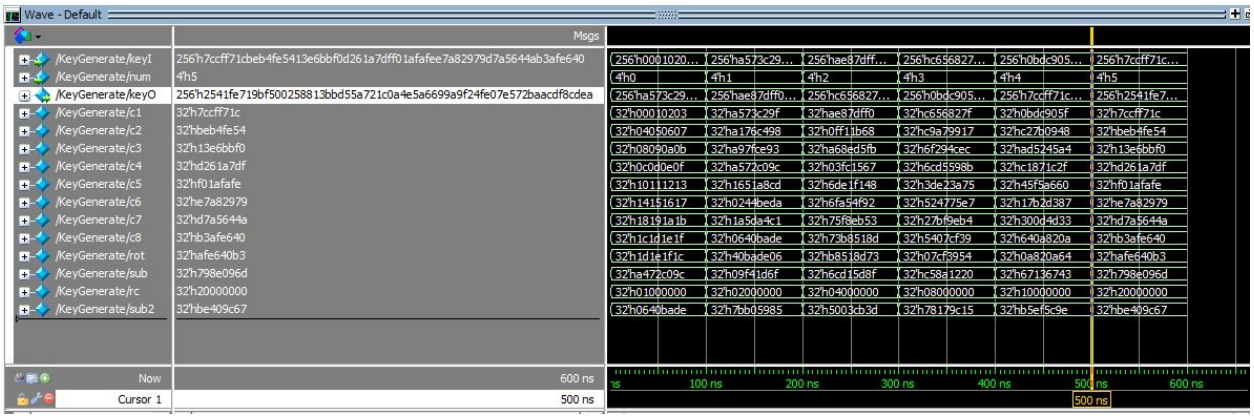


Fig 27. Output for Key 12 and Key 13

Key 12: **2541fe719bf500258813bbd55a721c0a**

Key 13: **4e5a6699a9f24fe07e572baacdf8cdea**

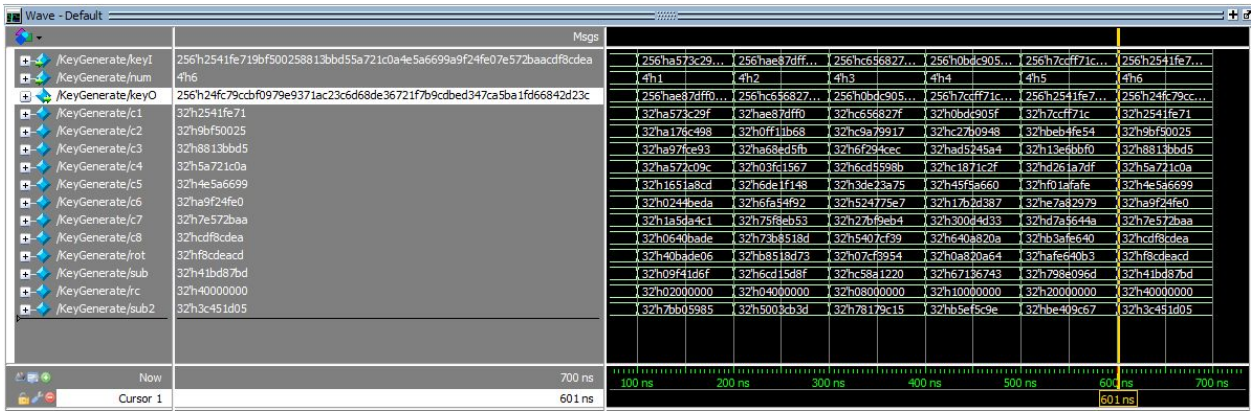


Fig 28. Output for Key 14

Key 14: **24fc79ccbf0979e9371ac23c6d68de36**

5) CONCLUSION:

- **Security:**

Before the Advanced Encryption Standard(AES) was designed, the mainly used algorithm was the Data Encryption Standard (DES). DES has a smaller key size which makes it less secure. DES is much slower as compared to AES. AES is way better and more widely used nowadays because of its higher security and faster working.

- **Application:**

As we are using many technologies, so there should be a secure encryption system for their security. The US National Security Agency authorizes the transmission of classified data at the TOP SECRET level through AES. Professor Christof Paar of the Ruhr University of Bochum, Germany has estimated that AES used to encrypt over 50% of all data globally. Usually, a hardware-based AES-128 bit helps to attain high speed, low power consumption, and silicon area Optimization.

Eg: Encrypted state for all bits in flight and at rest.
For robust cloud computing security, you really would like to get to homomorphic encryption standards so that the computes can take place in the encryption space.

It is also implemented in secure file transfer protocols like FTPS, HTTPS, SFTP, AS2, WebDAVS and OFTP.

- **Future scope:**

One could select larger size keys which can make this algorithm more secure for future attacks. So such a secure algorithm can have an ideal application such as more secure multimedia communications. So AES can be updated such that it becomes more difficult to decrypt any encrypted thing. The isomorphic mapping process of composite S-Box can be concentrated more in the future for further improvement in the system performance with the reduction of area and power utilization.

Note: The images used in overview of implementation section are used from google images and from some other sources.