Department of Electrical Engineering
The University of Mississippi

HARDWARE IMPLEMENTATION OF REAL TIME COMMUNICATION USING UART WITH AES 128 ENCRYPTION/DECRYPTION IN VERILOG HDL

Prepared by

Dhungel,Anurag
Paudel, Ayush
Regmi,Khum

Department of Electrical Engineering
The University of Mississippi

May 1, 2021

for

Dr. Md Sakib Hasan
Department of Electrical Engineering
The University of Mississippi

Abstract

This report gives details on the implementation of AES-128 bit encryption and decryption on Verilog Hardware Description Language and the modelling of real time communication between two NEXYS4 DDR FPGA boards. Advanced Encryption Standard(AES) is a symmetric block cipher defined in US Federal Information Processing Standard(FIPS). Encryption and Decryption has been an integral part of communication in the modern world, where a number of exchanges of valuable information take place through different private and public networks. Secure communication has become a crucial need for our technological infrastructures and this project intends to explore our industry standard security measures and explore low-cost embedded applications.

Contents

List of Figures

List of Tables

1. Introduction

With the advent of the digital age, data became the definition of the world around us describing not only the ambient surrounding but also our personal information. Each data shared between two parties could be easily accessible bestowing utmost transparency of concerned parties. Hence, the requirement to securely transfer data became inevitable. Several encryption techniques utilizing the substitution of each letter to a known and reversible data field came into existence. But, computing power allowed illegal personnel to decrypt data without authorization. Hence, powerful techniques were required which were immune to attacks but simpler to process and exchange files. Encryption techniques with personal keys with the aid of already defined abstract algebra came into popularity. People with keys could easily extract information from jargon and people without keys would be unable to decrypt despite huge processing power.

Advanced Encryption Standard(AES) is a symmetric block cipher defined in US Federal Information Processing Standard (FIPS). AES comes in three versions; AES_128, AES_192, and AES_256. The personal key and the data slice taken are represented by the numeric part of the AES naming system[5]. AES is one of the most widely used and proven encryption systems used all around the world. AES has resisted all crypto-analysis and proved unbreakable since its introduction. This paper explores what makes AES such a successful cryptography technique.

AES encryption system consists of four main operations in encryption and a similar set of operations in decryption which equalizes the four operations of encryption. Each operation involves different hardware configurations. Verilog, a hardware description language was used for producing the synthesizable design and its verification. These hardware configurations were written in a modular format and are reused throughout each process described in the report.

The other important part of the project was the real-time communication of the Field Programmable Gate Arrays(FPGA) boards using AES Encryption and Decryption. The communication method of choice implemented between the devices was UART, which stands for Universal Asynchronous Receiver/Transmitter. UART, an asynchronous serial data transmission protocol was chosen for this particular project because of its simplicity as well as its minimalist techniques. The basic unit of operation is an 8-bit method that is 8-bits of data is sent or received at a time, and which is expanded to 16 rounds to make it compatible with 128-bit AES. To conclude, whilst plain data see 8-bit data transfer, encrypted/decrypted data works with 16 8-bit data transfers in tandem.

This project is successful in creating a secure real-time communication link between FPGAs, each with AES-128 bits encryption and decryption. This paper is wrapped around the details of the AES, each sub-process involved in encryption and decryption techniques, UART protocol and implementation, communicating different FPGAs as well as with a computer, describing hardware schematics as well as documenting the results after each process and the final outcome.

2. Advanced Encryption System 128

2.1. Overview

AES is based on a design principle known as a substitution–permutation network, and is efficient in both software and hardware[key1]. AES takes in data in blocks of 128 bits and encrypts them using the input 128 bit cipher key. The basic processing unit for AES algorithm is byte and all operations are done in the Galios finite field (2^8). The finite field operations are performed on two dimensional 4*4 array of bytes which is known as State.

Table 1: AES Transformation Method

Each box represent 8 bit out of 128 bit													
in_00	in_04	in_08	in_12	→	S_00	S_04	S_08	S_12	→	out_00	out_04	out_08	out_12
in_01	in_05	in_09	in_13	→	S_01	S_05	S_09	S_13	→	out_01	out_05	out_09	out_13
in_02	in_06	in_10	in_14	→	S_02	S_06	S_10	S_14	→	out_02	out_06	out_10	out_14
in_03	in_07	in_11	in_15	→	S_03	S_07	S_11	S_15	→	out_03	out_07	out_11	out_15
Input				→	Process				→	Output			

The input plain-text is pasted into state as shown above and all operations are done into the state. The end state is then taken as output. Here's the main breakdown of all operations involved with AES Encryption and Decryption.

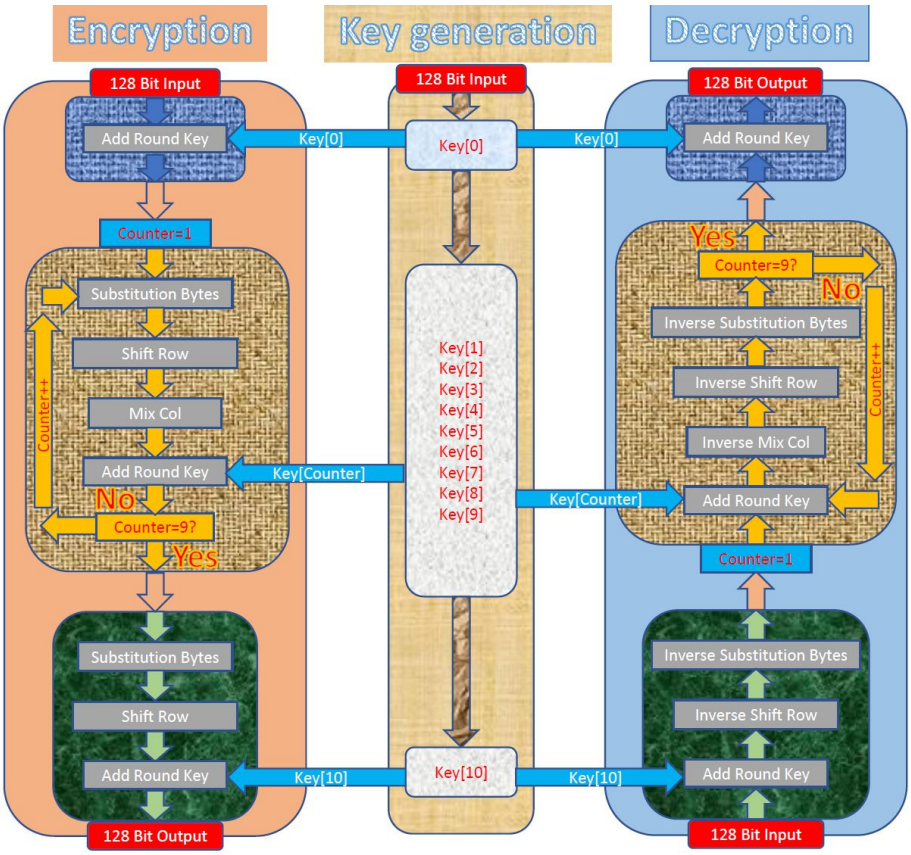

Figure 1: AES-128

2.2. Algorithm

1. Encryption Algorithm
   (a) Key Expansion : Eleven Round Keys are generated from the input cipher key using Key Expansion algorithm.
   (b) Addition of Initial Round Key: A bitwise-XOR operation of initial roundkey (k0) is carried out with the input data of 128 bits.
   (c) 9 Rounds of Transformations: Following transformations take place in each round, for a total of nine rounds. As a round is completed, it goes to the beginning of round until completion of 9 rounds. The key changes in each round from (k1) all the way to k(9).
      - SubBytes : Each byte of the state is substituted non-linearly using Rijndael S-box.
      - ShiftRows: The 16 byte data is arranged in 4 by 4 grid, and shiftRows transformation ( ) is carried out in the grid.
      - MixColumns: The columns of the 4×4 matrix is multiplied to a mix_colmns matrix to obtain output.
      - AddRoundKey: A bitwise-XOR operation of the expanded roundkey is carried out with the data of 128 bits.
   (d) 10th Round:
      - SubBytes : Each byte of the state is substituted non-linearly using Rijndael S-box.
      - ShiftRows: The 16 byte data is arranged in 4 by 4 grid, and shiftRows transformation ( ) is carried out in the grid.
      - AddRoundKey: A bitwise-XOR operation of the expanded roundkey is carried out with the data of 128 bits.
2. Decryption Algorithm
   (a) Initial round:
      - AddRoundKey: A bitwise-XOR operation with last part of the expanded roundkey(k10) is carried out with the encrypted data of 128 bits.
      - invShiftRows: Iverse shift row() transformation is done on the output from the step above.
      - invSubBytes : Each byte of the state is substituted non-linearly using Rijndael S-box.
   (b) 9 Rounds of Transformations: Following transformations take place in each round, for a total of nine rounds. As a round is completed, it goes to the beginning of round until completion of 9 rounds. The key changes in each round for (k9) all the way to k(1).
      - AddRoundKey: A bitwise-XOR operation of the expanded roundkey is carried out with the data of 128 bits.
      - invMixColumns: The columns of the 4×4 matrix is multiplied to a inv_mix_colmns matrix to obtain output. The inv_mix_colmns matrix is the inverse of mix_colmns matrix in $GF(2^8)$.
      - invShiftRows: Iverse shift row() transformation is done on the output from the step above.
      - invSubBytes : Each byte of the state is substituted non-linearly using Rijndael S-box.
   (c) 10th Round: The ouput from the above step is now XOR'ed with the intial round key (k0) generated through key expansion algorithm.

3. Key addition and inversion

3.1. Add Round Key

Figure 2: Inverse Add Round Key Pseudo Circuit

In Galios Field(2^8) the addition operator is realized by bitwise $\bigoplus$ operation. AES encryption is based on Galios Field(2^8). The add round key operation is performed by using bitwise $\bigoplus$ operator. This operation defines the relationship between the key used in the encryption and the data. With the initial decided key that is shared by two concerned organization, both organization employ key expansion methods to produce 10 more keys.[4] Add round key is employed 11 different times to mask the data to be encrypted.

- Pseudo Code for Add Round Key

$$\textbf{\textit{Output of Add Round Key} = \textit{128-Bit Input Data} } \bigoplus \textbf{\textit{ 128-Bit Key}}$$

The 128-Bit data and 128-Bit key are both arranged in 4×4 matrix for graphical representation. Each element in the matrix cell in 8-Bit data. Corresponding element from key matrix and data matrix are taken and bit wise $\bigoplus$ operation is performed to obtain the output as shown in the table 18.In hardware level, the linear busses are arranged for the required configuration instead of creating a 2D busses.

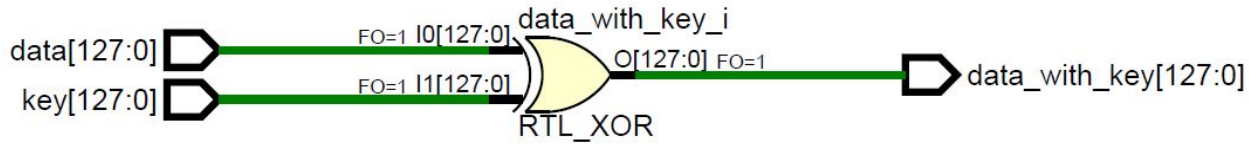

Figure 3: Add Round Key Schematics.

Table 2: Add Round Key

D0	D4	D8	D12	$\bigoplus$	K0	K4	K8	K12	=	A0	A4	A8	A12
D1	D5	D9	D13		K1	K5	K9	K13		A1	A5	A9	A13
D2	D6	D10	D14		K2	K6	K10	K14		A2	A6	A10	A14
D3	D7	D11	D15		K3	K7	K11	K15		A3	A7	A11	A15
Input Data					Key					Output			

3.2. *Inverse Add Round Key*

Figure 4: Add Round Key Pseudo Circuit

Bitwise $\bigoplus$ toggles the bits of input data, whenever there is '1', is the corresponding bit of the Key used in add round key. Hence, repeating the process using the same key again toggles the same bits to produce the original data. In terms of abstract algebra, in Galois Field(2^8) each number is its own additive inverse.[4] Hence, the addition operator is realized by bitwise $\bigoplus$ operation and the subtraction operator is also realized by the bitwise $\bigoplus$ operation. The inverse add round key operation is performed by using bitwise $\bigoplus$ operator between the cryptic data and the same key used to create the cryptic data from the original one.

- Pseudo Code for Add Round Key

$$\textbf{\textit{Original Data}} = \textbf{\textit{Cryptic Data}} \bigoplus \textbf{\textit{128-Bit Key}}$$

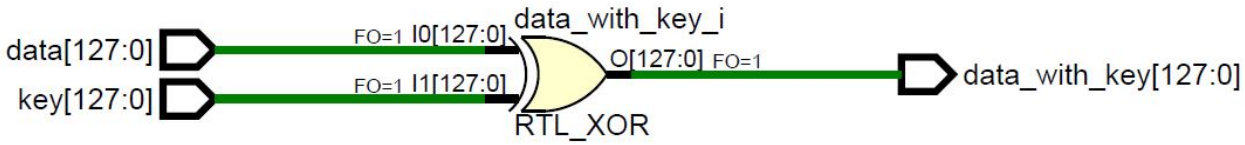

Figure 5: Inverse Add Round Key Schematics.

Table 3: Inverse Add Round Key

A0	A4	A8	A12	$\bigoplus$	K0	K4	K8	K12	=	D0	D4	D8	D12
A1	A5	A9	A13		K1	K5	K9	K13		D1	D5	D9	D13
A2	A6	A10	A14		K2	K6	K10	K14		D2	D6	D10	D14
A3	A7	A11	A15		K3	K7	K11	K15		D3	D7	D11	D15
Cryptic Data					Key					Original Data			

4. Substitution Operation

4.1. Substitution bytes( ) Transformation

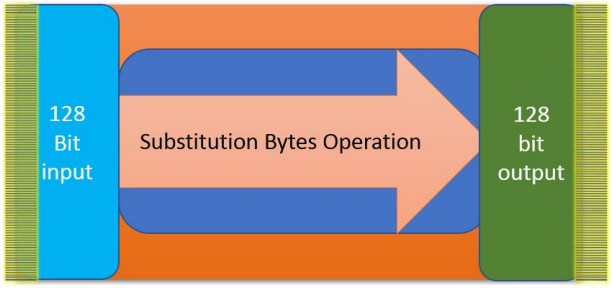

Figure 6: Substitution Bytes Pseudo Circuit

The Sub-bytes Transformation( ) is a byte substitution operation performed on each individual byte of the 128-bit data. Each byte is substituted to a different byte using a well-defined substitution box called S-box. S-box contains a mapping of each byte from 00 to FF.

fig: AES-Substitution Box

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

The S-box is constructed by performing the following transformation:

- Calculate the multiplicative inverse in the finite field GF(2^8) of the byte.
- Apply the following transformation to the byte:
  $b_i^{'} = b_i \bigoplus b_{(i+4)mod8} \bigoplus b_{(i+5)mod8} \bigoplus b_{(i+6)mod8} \bigoplus b_{(i+7)mod8} \bigoplus c_i$
  Here, b_i represents i^th bit of the byte and c_i represents i^th of a constant byte with a value of 63.

The second operation is performed in each bit of the entire byte b_0 through b_7[[5]]. As the above two transformations are performed in each byte ranging from 00 to FF, a S-box is generated which contains the unique mapping of each byte as shown below:

And, the substitution byte for the input byte is determined by looking at the intersection of row and column. For example, the substitution byte of 59 will the intersection of a row with index '5' and column with index '9' which is cb. It can be located in the substitution box below.

- Pseudo code for substitution byte transformation :

  S-box(data[0:7], substituted-data[0:7])

  This is for the first byte and it goes all the way down to 16^{th} byte where a byte is mapped to a different byte using the S-box.

Table 4: Sub bytes () Transformation

A0	A4	A8	A12	=	S0	S4	S8	S12
A1	A5	A9	A13		S1	S5	s9	S13
A2	A6	A10	A14		S2	S6	S10	S14
A3	A7	A11	A15		S3	S7	S11	S15

The schematics of sub-bytes( ) transformation is shown below:

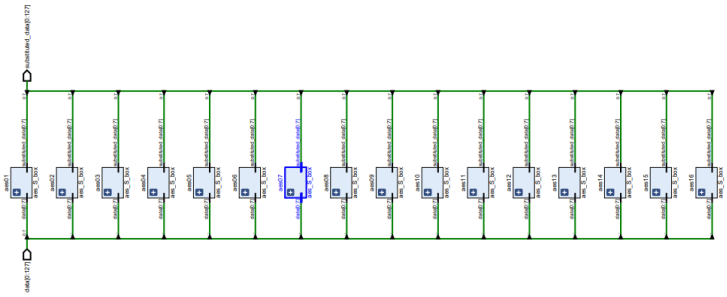

Figure 7: Sub-bytes Schematic View

4.2. Inverse Sub-bytes( ) Transformation

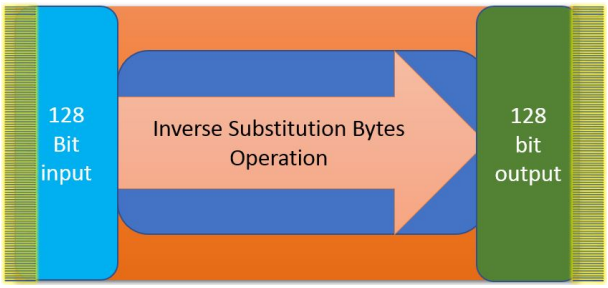

Figure 8: Inverse Substitution Bytes Pseudo Circuits

The Inverse Sub-bytes( ) Transformation is also a byte substitution operation performed on the individual byte of the 128-bit data during decryption. It is similar to sub-bytes( ) Transformation where each byte is substituted to a different byte using a well-defined substitution box called Inverse S-box. Just like S-box, this also contains a mapping of each byte from 00 to FF and is inverse of the S-box.

And, the substitution byte for the input byte is determined by looking at the intersection of row and column. For example, the substitution byte of cb will the intersection of a row with index 'c' and column with index 'b' which is 59. This can be located in the Inverse Substitution box below.

fig: AES-Inverse Substitution Box

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
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

- Pseudo code for Inverse substitution byte transformation :

  Inverse-S-box(data[0:7], inverse-data[0:7])
  This is for the first byte and it goes all the way down to 16^{th} byte where a byte is mapped to a different byte using inverse S-box.

Table 5: Inverse Sub bytes () Transformation

S0	S4	S8	S12		A0	A4	A8	A12
S1	S5	S9	S13		A1	A5	A9	A13
S2	S6	S10	S14	=	A2	A6	A10	A14
S3	S7	S11	S15		A3	A7	A11	A15

The schematics of inverse sub-bytes( ) transformation is shown below:

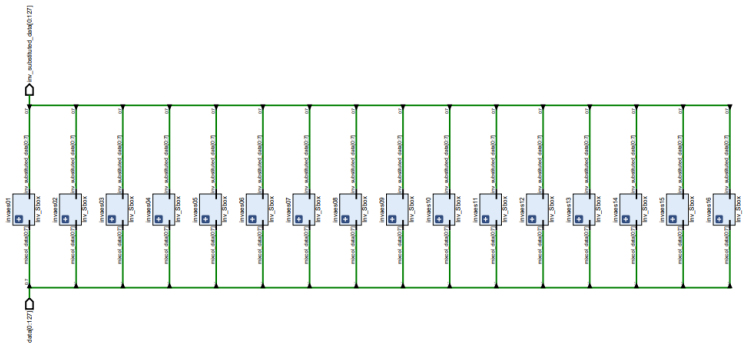

Figure 9: Inverse Sub-bytes Schematic View

5. Shifting Operation

5.1. Shift-Rows() Transformation

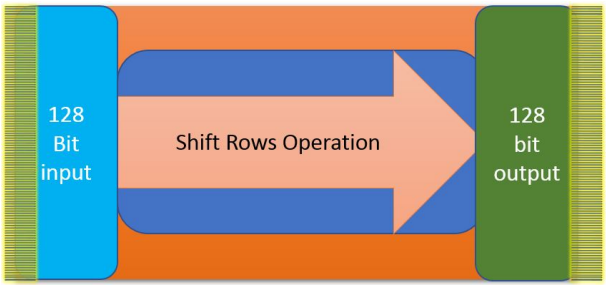

Figure 10: Shift Rows Pseudo Circuits

The shift rows() transformation is about shifting bytes in each row of a matrix by a certain offset as determined in the algorithm. The data are arranged in a 4 by 4 grid which is shown in the figure below. The following transformation is applied in each row[5]:

- The first row is left unchanged.
- The bytes in the second row are shifted one position to the left.
- The bytes in the second row are shifted two positions to the left.
- The bytes in the second row are shifted three positions to the left.

The array of bytes before and after transformation is shown below:

Table 6: Shift Rows () Transformation

S0	S4	S8	S12		S0	S4	S8	S12
S1	S5	S9	S13		S5	S9	s13	S1
S2	S6	S10	S14	=	S10	S14	S2	S6
S3	S7	S11	S15		S15	S3	S7	S11

- Pseudo code for shift rows( ) transformation :

  shifted byte[0:7] = substituted byte[0:7]
  shifted byte[8:15] = substituted byte [104:11]
  shifted byte[16:23] = substituted byte [80:87]
  shifted byte [24:31] = substituted byte [53:63]
  This is for the first column, and the pattern is similar for all the other columns as shown in figure above.

5.2. *Inverse Shift-Rows() Transformation*

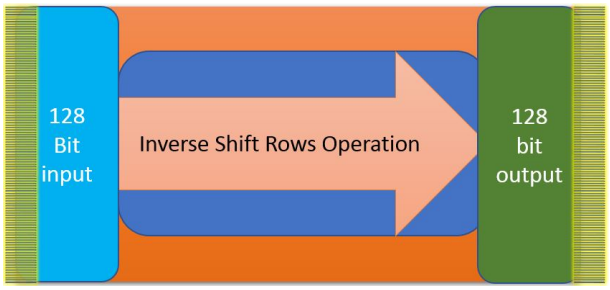

Figure 11: Inverse Shift Rows Pseudo Circuits

The Inverse shift rows() transformation is about shifting bytes in each row of a matrix by a certain offset as determined in the algorithm during decryption. The following transformation is applied in each row:

- The first row remains unchanged.
- The bytes in the second row are shifted one position to the right.
- The bytes in the second row are shifted two positions to the right.
- The bytes in the second row are shifted three positions to the right.

The array of bytes before and after transformation is shown below:

Table 7: Inverse Shift Rows () Transformation

S0	S4	S8	S12		S0	S4	S8	S12
S1	S5	S9	S13		S13	S1	s5	S9
S2	S6	S10	S14	=	S10	S14	S2	S6
S3	S7	S11	S15		S7	S11	S15	S3

- Pseudo code for inverse shift rows/( ) transformation :
  inverse shifted byte[0:7] = inverse mix-column byte[0:7]
  inverse shifted byte[8:15] = inverse mix-column byte [40:47]
  inverse shifted byte[16:23] = inverse mix-column byte [80:87]
  inverse shifted byte [24:31] = inverse mix-column byte [120:127]

  This is for the first column, and the pattern is similar for all the other columns as shown in figure above.

6. Column Operations

6.1. Mix Column

Figure 12: Mix Column Pseudo Circuits

Mix Column operation is a crucial part of AES Encryption Decryption. Mix Column utilizes finite field arithmetic in Galois Field(2^8). 128-bit data is accepted in each mix column operation and it returns 128-bit data[5]. The 128-bit data entered is arranged in a 4×4 matrix. The 4×4 matrix contains 16 elements as shown in table 8. Each element is of size 8 bit which can be represented by 2 hexadecimal characters.

Table 8: Mix Col Matrix and Input Data Matrix

Mix Col Matrix			
02	03	01	01
01	02	03	01
01	01	02	03
01	01	02	03

and

Input Data Matrix			
D0	D4	D8	D12
D1	D5	D9	D13
D2	D6	D10	D14
D3	D7	D11	D15

After the input data is arranged in 4×4 matrix, it undergoes matrix multiplication with the Mix Col Matrix as shown is table 9.

Table 9: Mix Column Operation

Mix Col Matrix			
02	03	01	01
01	02	03	01
01	01	02	03
01	01	02	03

×

Input Data Matrix			
D0	D4	D8	D12
D1	D5	D9	D13
D2	D6	D10	D14
D3	D7	D11	D15

=

Output Data Matrix			
O0	O4	O8	O12
O1	O5	O9	O13
O2	O6	O10	O14
O3	O7	O11	O15

Column Wise multiplication is visualized in table 10.

Table 10: Column wise multiplication in Mix Col Operation

Mix Col Matrix			
02	03	01	01
01	02	03	01
01	01	02	03
01	01	02	03

×

Input Col
D_n
D_{n+1}
D_{n+2}
D_{n+3}

=

Output Col
O_n
O_{n+1}
O_$n+2$
O_{n+3}

$n \in \{0, 4, 8, 12\}$

The matrix multiplication only involves multiplication by 01,02 and 03 in Galios Field(2^8) .

- Irreducible Polynomial
  The irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ is utilized if our multiplication operation exceeds 8 bits. Since, our multiplication only involves multiplication by 01,02 and 03, the irreducible operation only needs to be subtracted once. This will provides us the remainder when our 9 bit output is divided by the irreducible polynomial. But, since we onlyt need 8 bit as our output, we can ignore the ninth bit from the beginning and just utilize $x^4 + x^3 + x + 1$ as our irreducible polnomial.
- Example of Remainder

Table 11: Remainder of 100000000 when divided by irreducible polynomial (2^8)

Binary Position	8	7	6	5	4	3	2	1	0
8	1	0	0	0	0	0	0	0	0
(8+4+3+1+0) * 0	1	0	0	0	1	1	0	1	1
XOR Remainder	0	0	0	0	1	1	0	1	1
Binary	0	0	0	0	1	1	0	1	1
Hex	1 B								

- Multiplication by 01
  Multiplication by one returns the original input as nothing is changed.

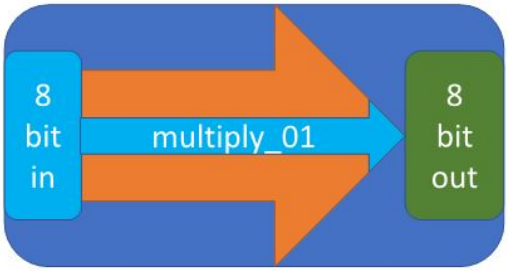

Figure 13: Pseudo Circuit for multiplication by 1

- Multiplication by 02

Table 12: Algorithm for Multiplication by 2

Input	a	b	c	d	e	f	g	h	
	is a=1 or 0								
temp	b	c	d	e	f	g	h	0	Shift 1 to left
	if a is 1								
Output	temp $\oplus$ irreducible polynomial								
	if a is 0								
Output	temp								

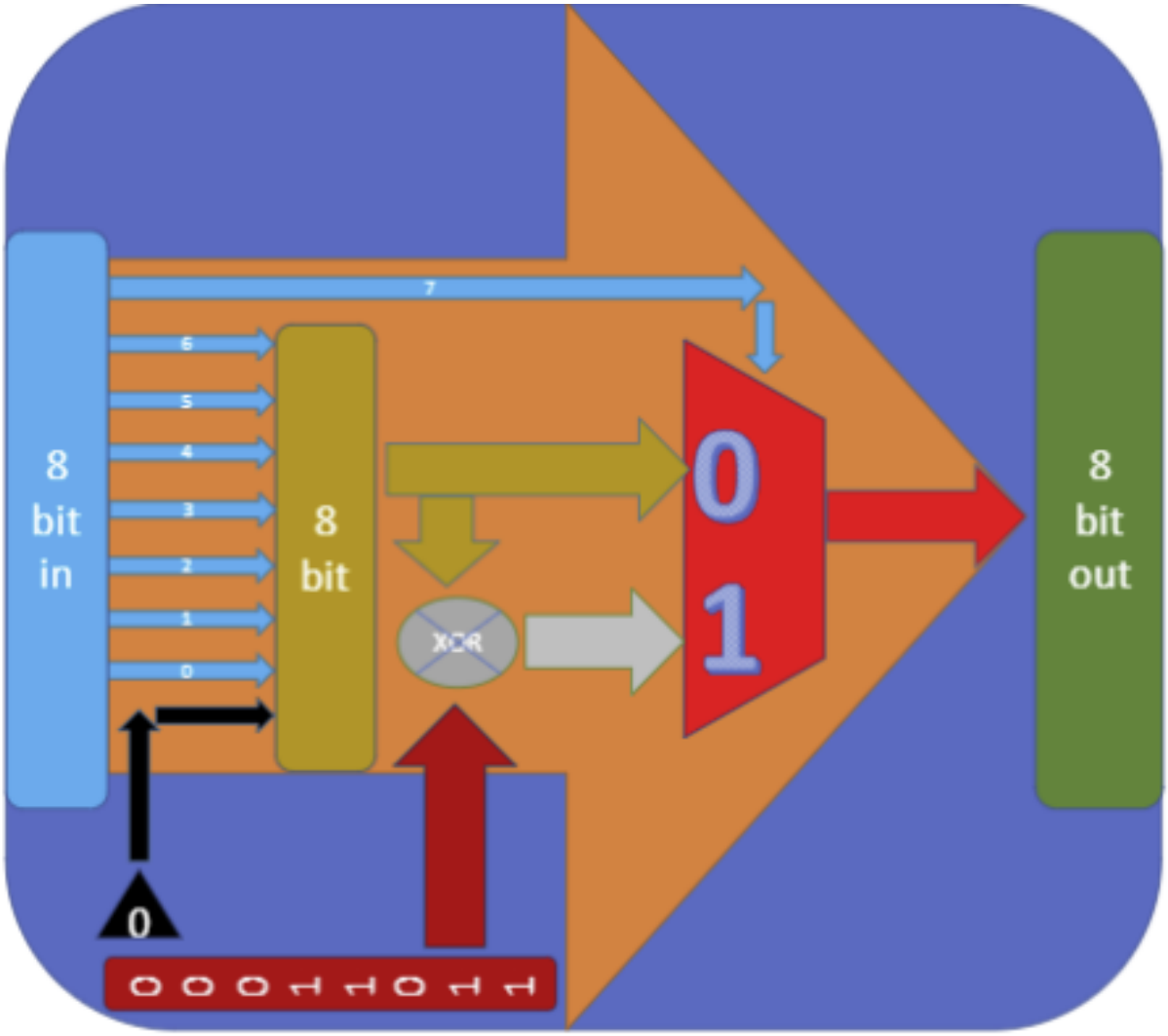

Figure 14: Pseudo Circuit for multiplication by 2

- Multiplication By 03
  In Galois Field(2^8) , multiplication is distributive over addition. Hence multiplication by 03 can be converted to multiplication by one and two.

  $$input \times 03 = input \times (11) = input \times (10 \oplus 01) = input \times 10 \oplus input \times 01 = input \times 2 \oplus input$$

  Thus, utilizing multiplication by 02, we can achieve multiplication by 03.

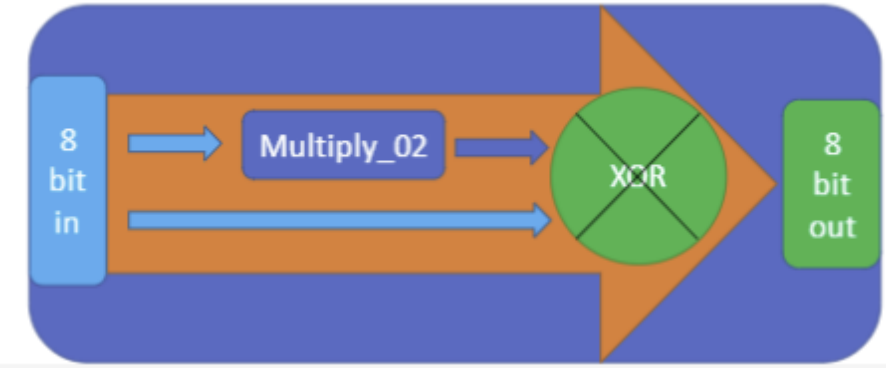

Figure 15: Pseudo Circuit for multiplication by 3

- Addition
  Addition in GF(2^8) is simply bitwise xor operation.

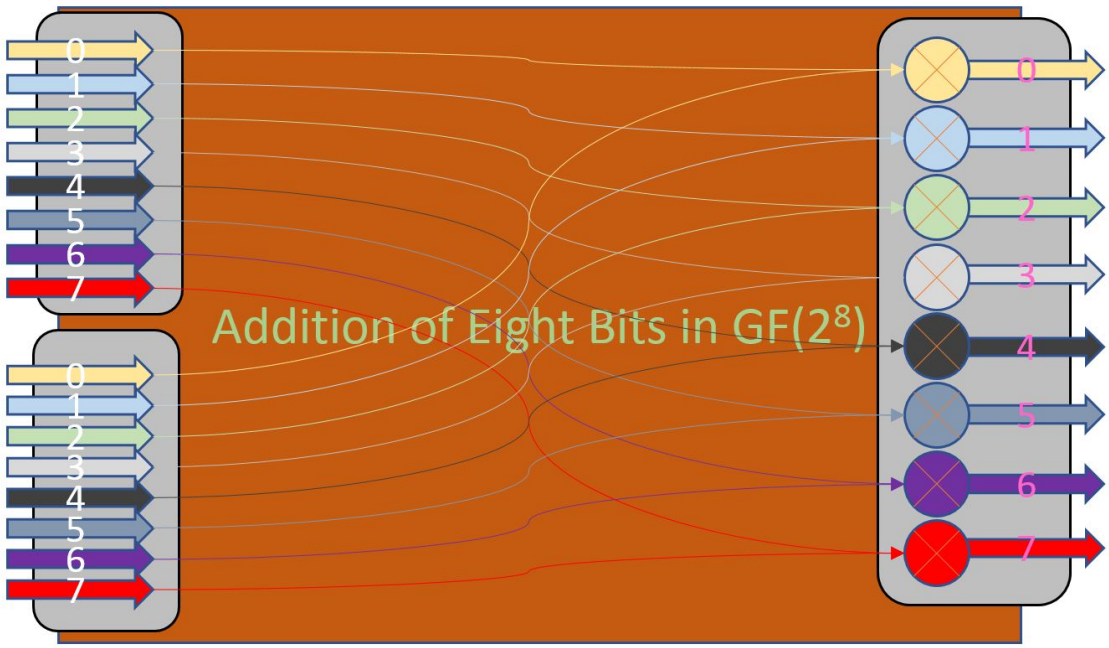

Figure 16: Pseudo Circuit for multiplication by 3

Hence, Mix Column operation is conducted in GF(2^8) feild using the above principle of multiplication and addition.

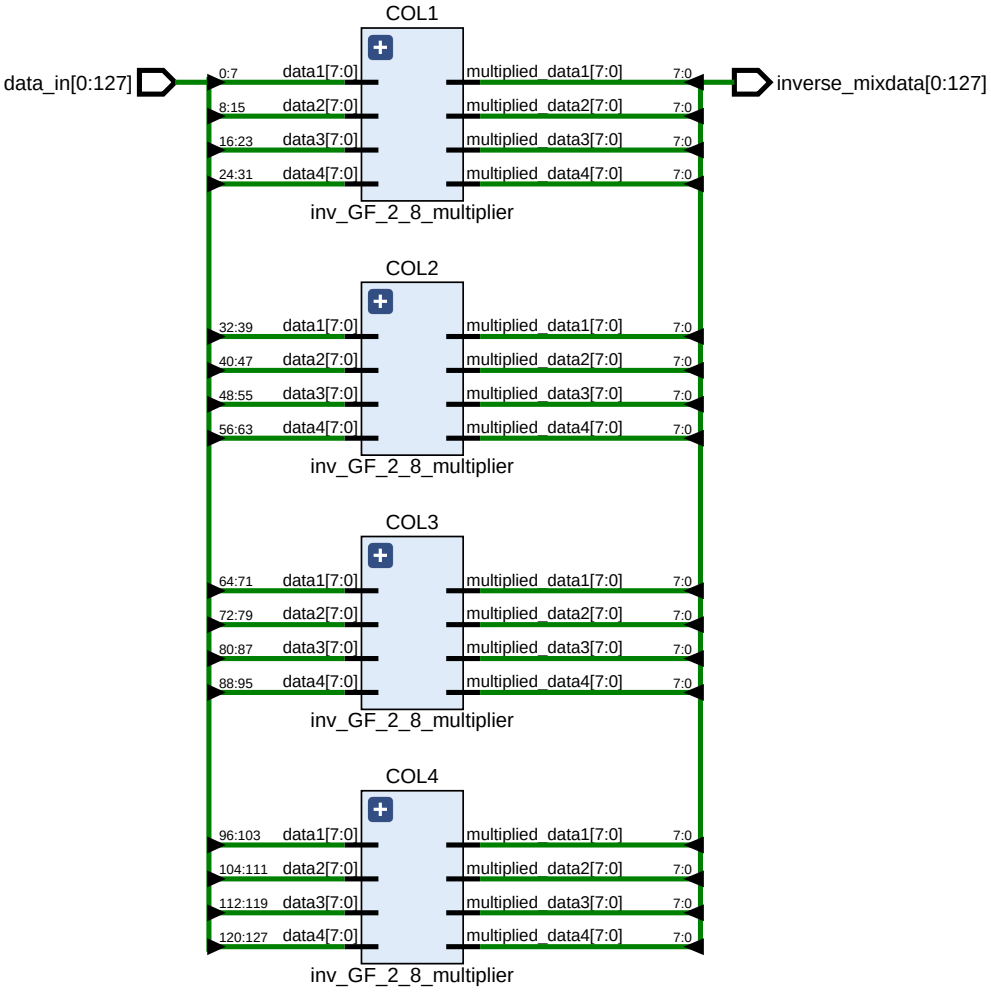

Figure 17: Schematic View of Mix Columns

6.2. *Inverse Mix Column*

Figure 18: Inverse Mix Column Pseudo Circuits

Inverse Mix Column operation inverts the changes performed to Mix Column. [4] It utilizes finite field arithmetic in Galois Field(2^8). 128-bit data is accepted in each inverse mix column operation and it returns 128-bit data. The 128-bit data entered is arranged in a 4×4 matrix. The 4×4 matrix contains 16 elements as shown in table 13. Each element is of size 8 bit which can be represented by 2 hexadecimal characters.

Table 13: Inverse Mix Col Matrix and Input Data Matrix

Inv Mix Col Matrix			
0E	0B	0D	09
09	0E	0B	0D
0D	09	0B	0D
0B	0D	09	0E

and

Input Data Matrix			
D0	D4	D8	D12
D1	D5	D9	D13
D2	D6	D10	D14
D3	D7	D11	D15

After the input data is arranged in 4×4 matrix, it undergoes matrix multiplication with the Inv Mix Col Matrix as shown is table 14.

Table 14: Inv Mix Column Operation

Inv Mix Col Matrix			
0E	0B	0D	09
09	0E	0B	0D
0D	09	0B	0D
0B	0D	09	0E

×

Input Data Matrix			
D0	D4	D8	D12
D1	D5	D9	D13
D2	D6	D10	D14
D3	D7	D11	D15

=

Output Data Matrix			
O0	O4	O8	O12
O1	O5	O9	O13
O2	O6	O10	O14
O3	O7	O11	O15

Column Wise multiplication is visualized in table 15.

Table 15: Column wise multiplication in Inv Mix Col Operation

Inv Mix Col Matrix			
0E	0B	0D	09
09	0E	0B	0D
0D	09	0B	0D
0B	0D	09	0E

×

Input Col
D_n
D_{n+1}
D_{n+2}
D_{n+3}

=

Output Col
O_n
O_{n+1}
O_$n+2$
O_{n+3}

$n \in \{0, 4, 8, 12\}$

The matrix multiplication only involves multiplication by 0E,0B,0D, and 09 in Galois Field(2^8).

- Multiplication by 09

$$input \times 09 = input \times (00001001) = input \times (1000 \oplus 0001) = (input \times 1000) \oplus (input \times 0001)$$
$$input \times 1000 = input \times (10) \times (10) \times (10) = input \times 2 \times 2 \times 2$$
$$input \times 0001 = input$$

Hence,

$$input \times 09 = (input \times 2 \times 2 \times 2) \oplus (input)$$

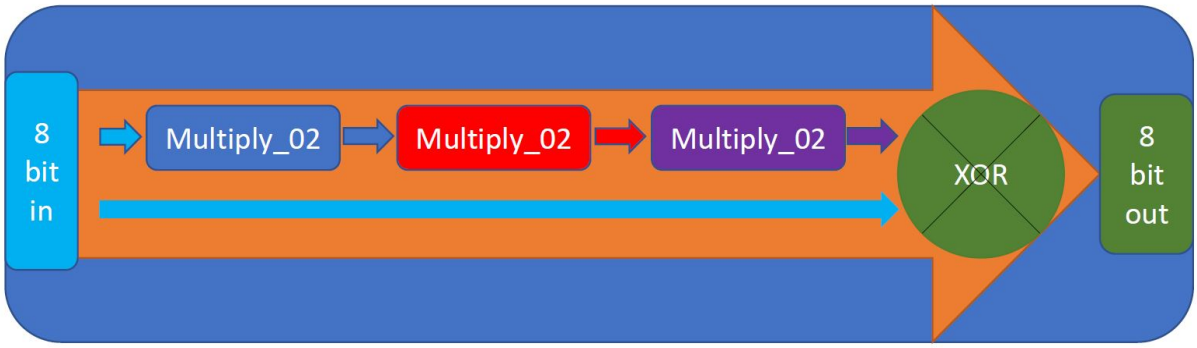

Figure 19: Pseudo Circuit for multiplication by 09

- Multiplication by 0B

$$input \times 0B = input \times (00001011) = input \times (1000 \oplus 0010 \oplus 0001) = (input \times 1000) \oplus (input \times 0010) \oplus (input \times 0001)$$
$$input \times 0B = (input \times 2 \times 2 \times 2) \oplus (input \times 2) \oplus (input)$$

Smart Implementation by reducing repetition:

$$input \times 0B = \ (((input \times 2 \times 2) \oplus input) \times 2) \oplus input$$

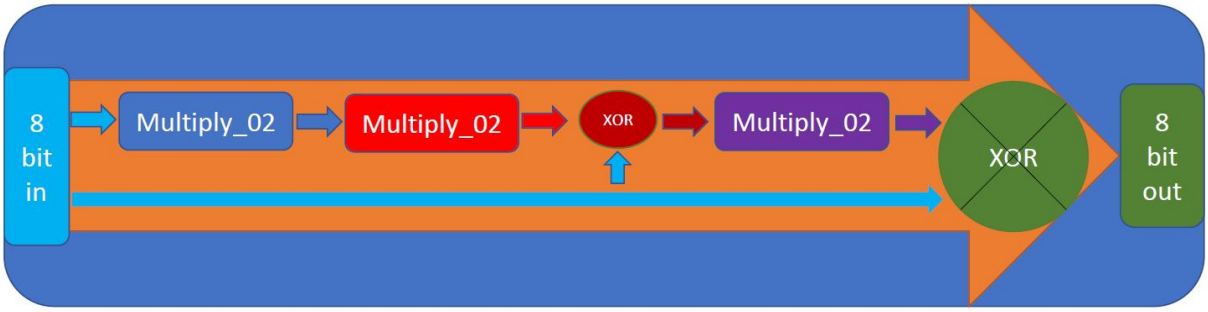

Figure 20: Pseudo Circuit for multiplication by 0B

- Multiplication by 0D

$$input{\times}0D = input{\times}(00001101) = input{\times}(1000{\oplus}00100{\oplus}0001) = (input{\times}1000){\oplus}(input{\times}0100){\oplus}(input{\times}0001)$$

$$input \times 0D = (input \times 2 \times 2 \times 2) \oplus (input \times 2 \times 2) \oplus (input)$$

Smart Implementation by reducing repetition:

$$input \times 0D = \ (((input \times 2) \oplus input) \times 2 \times 2) \oplus input$$

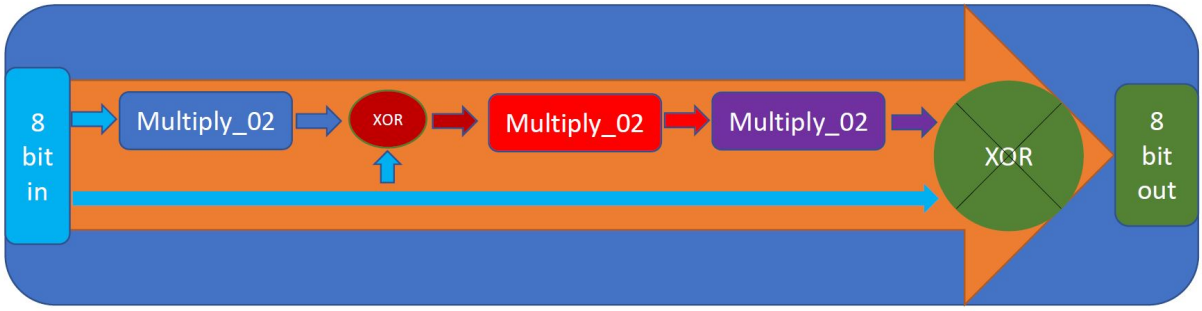

Figure 21: Pseudo Circuit for multiplication by 0D

- Multiplication by 0E

$$input{\times}0E = input{\times}(00001110) = input{\times}(1000{\oplus}00100{\oplus}0010) = (input{\times}1000){\oplus}(input{\times}0100){\oplus}(input{\times}0010)$$

$$input \times 0E = (input \times 2 \times 2 \times 2) \oplus (input \times 2 \times 2) \oplus (input \times 2)$$

Smart Implementation by reducing repetition:

$$input \times 0E = \ (((input \times 2) \oplus input) \times 2) \oplus input) \times 2$$

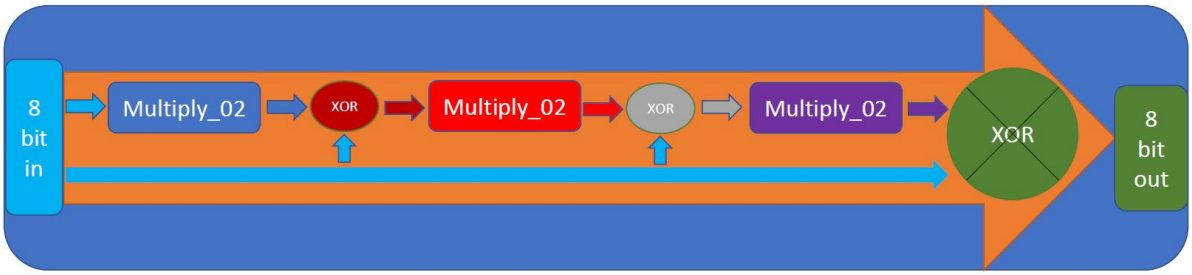

Figure 22: Pseudo Circuit for multiplication by 0E

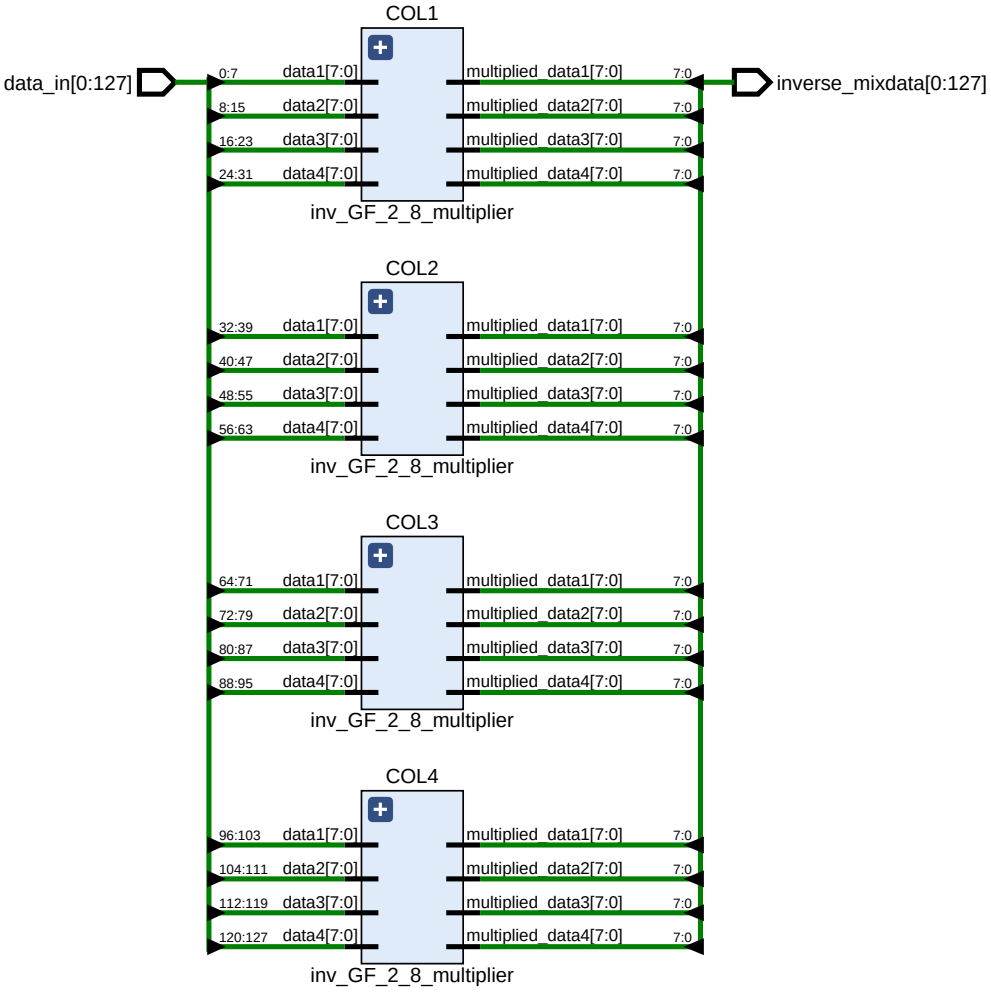

Figure 23: Schematic View of Inverse Mix Columns

7. Key Expansion

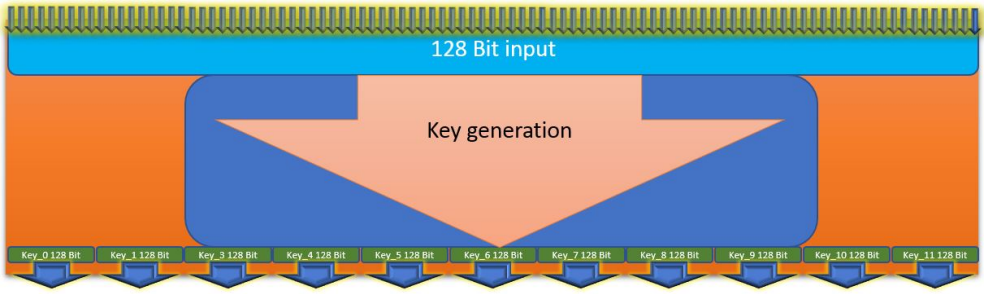

Figure 24: Key Generation pseudo circuit

Overview : Round Keys required for each AddRoundKey iteration is derived from a single Cipher Key by the means of the KeyExpansion algorithm. In AES_128, we have 11 total key additions that require 11 keys[3]. Each key is 128 bit and is divided into 4 words of 1 byte length to perform mathematical operation. The first four words i.e w[0],w[1],w[2],w[3] contains the input cipher key. The key expansion Pseudocode is given below:

```
KeyExpansion( [127:0] cipher_key, [127:0] expanded_key[0:10])

{
If (nr == 0)

W[0] = cipher_key(127: 96)
w[1] = cipher_key(95:64)
w[2] = cipher_key(63: 32)
w[3] = cipher_key(31:0)

expanded_key = {w[0],w[1],w[2],w[3]}
else
for(i = 1; i < 11, nr = 4*i ; i++)

{

temp = W[ nr - 1];

temp = SubByte (RotByte(temp)) ^ Rcon[];

W[nr] = W(nr -4) ^ temp;
W[nr+1] = W(nr ) ^ w(nr - 3) ;
W[nr+2] = W(nr +1) ^ w(nr - 2) ;
W[nr+3] = W(nr +2) ^ w(nr - 1) ;
expanded_key = {W[nr],W[nr+1],W[nr+2],W[nr+3]}
}

}
```

Figure 25: Key Generation pseudo code

Initially, the input cipher key is divided into four 32 bit words; w[0], w[1], w[2], and w[3]. We call this round zero. We use the word which contains the least significant bits of the previous key (w[3], w[7], w[11]..) and passes it sequentially to ROTWORD, then SUBBYTE, and then bitwise XOR with RCON. Then, the temp result and other previous words are used to calculate further words as we go along. At i = 10, we get our final word, w[43], and thus 11 keys are generated. The key expansion is designed in such a way that if a single bit in the input key sequence changes, it would affect every other key generated afterward. We explain the main operations; ROTWORD, SUBBYTE, and RCON below:

7.1. SubByte, Rotword and RoundConstant (Rcon)

RotWord

RotWord performs a one-byte circular left shift on a word. This means that an input word [B0, B1, B2, B3] is transformed into [B1, B2, B3, B0].[3]

Subword

SubByte performs a byte substitution on each byte of its input word, using the Rijndael S-box as described above in Section 5.1. [3]

Rcon

Rcon, as described by the Rijndael documentation, is the exponentiation of 2 to a user-specified value. This operation is not performed with regular integers, rather in Rijndael's finite field. RCON could be expressed as:

$$rc_i = \begin{cases} 1 & \text{if } i = 1 \\ 2 \cdot rc_{i-1} & \text{if } i > 1 \text{ and } rc_{i-1} < 80_{16} \\ (2 \cdot rc_{i-1}) \oplus 11\mathrm{B}_{16} & \text{if } i > 1 \text{ and } rc_{i-1} \geq 80_{16} \end{cases}$$

Figure 26: Round Constant(RCON)

where multiplication is defined in the Galius Field (GF(2^8)). [3]

Round	1	2	3	4	5	6	7	8	9	10
Hex	01	02	04	08	10	20	40	80	1b	36
Dec	01	02	04	08	16	32	64	128	27	54

Let look at Rcon(9) ,

$$Rcon(9) = 2 * Rcon(9-1)$$

$$Rcon(9) = 2 * Rcon(8)$$

Since the mulitplication is over ($GF(2^8)$),

$$Rcon(9) = 2 * Rcon(8) mod(x^8 + x^4 + x^3 + x + 1)$$

$$Rcon(9) = 256 mod(x^8 + x^4 + x^3 + x + 1)$$

Now, 256 in polynomial form is $1 * x^8$ where x = 2. So our operation returns as :

$$Rcon(9) = 8 mod(8 + 4 + 3 + 1 + 0)$$

where the numbers represent the powers of the binary number. We illustrate this operation below:

Table 16: Round Constant(RCON) Generation Over Galios Field (2ˆ8)

Binary Position	8	7	6	5	4	3	2	1	0
8	1	0	0	0	0	0	0	0	0
(8+4+3+1+0) * 0	1	0	0	0	1	1	0	1	1
XOR Remainder	0	0	0	0	1	1	0	1	1
Binary	0	0	0	0	1	1	0	1	1
Hex	1 B								

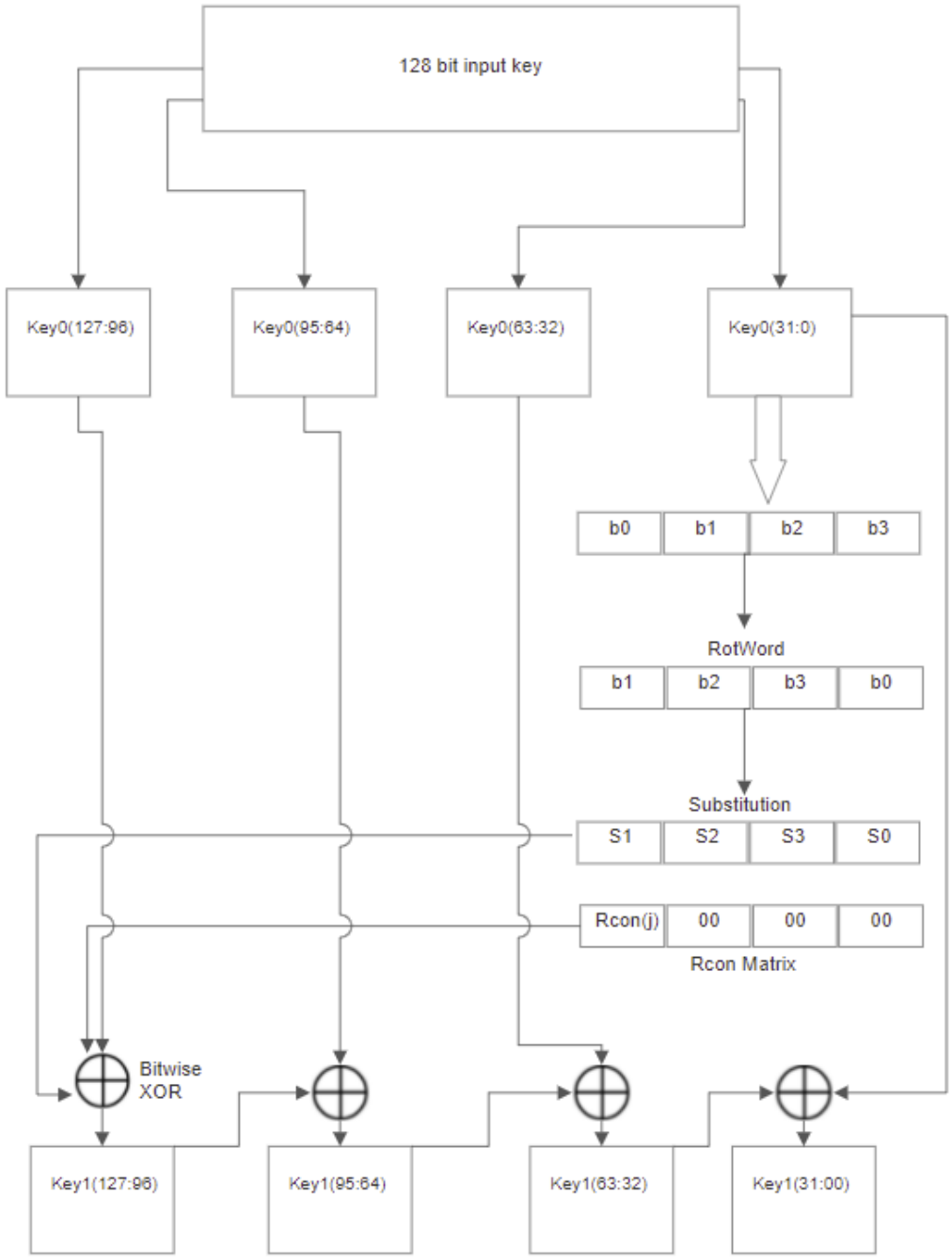

Figure 27: AES_128 Key Schedule

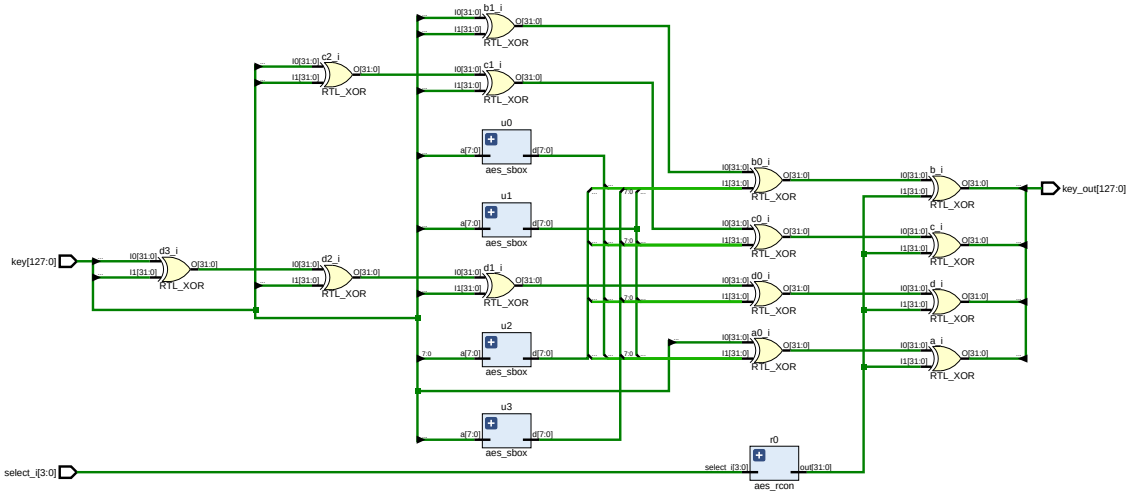

Figure 28: AES_128 Key Expansion Schematics

8. UART

8.1. UART for 8 bit Data Transfer

When we talk about connecting our phone to a computer or vice-versa for any kind of file transfers, USB is the most widely used interface which allows from charging our phones to sending/receiving files at a high speed to and from another device. In a similar manner, for hardware communication purposes, or simply communicating FPGA with a computer, we need an interface and protocols that govern the transfer and receiving of data. UART is one of the simplest and widely used methods of talking to the FGPA using a computer and also communicating different FPGAs.

UART stands for Universal Asynchronous Receiver/Transmitter. By definition, it is a hardware communication protocol that uses asynchronous serial communication with configurable speed where asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end. A UART is an interface that sends out usually a byte at a time over a single wire. Embedded systems, micro-controllers, and computers mostly use UART as a form of device-to-device hardware communication protocol as well as a way to talking to the FPGAs. Two UARTs directly communicate with each others using two wires for their transmitting and receiving ends.

The two signals of each UART device are named Transmitter(Tx) and Receiver(Rx). The main motive of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication. The UART transmitter and receiver have to agree on some parameters, such as:

- Baud Rate: 9600
- Number of data bits: 8
- Parity Bit : 0
- Stop Bits: 1
- Flow Control: None

Baud rate: The frequency at which the receiver captures the incoming bits and colloquially, the frequency at which the transmitter sends the outgoing bits.

The number of data bits: It is set to eight bits at a time. That means, in each round, a byte is sent over the channel.

Parity bit: It is to check the validity of the transmitted data over UART, and is appended after the data is sent which is calculated by doing an XOR operation on all of the bits.

Stop Bits: A stop is always set is 1, and it indicates the end of transmission of a byte.

Flow Control: It is likely to be set to None, and is not widely used in present days applications.

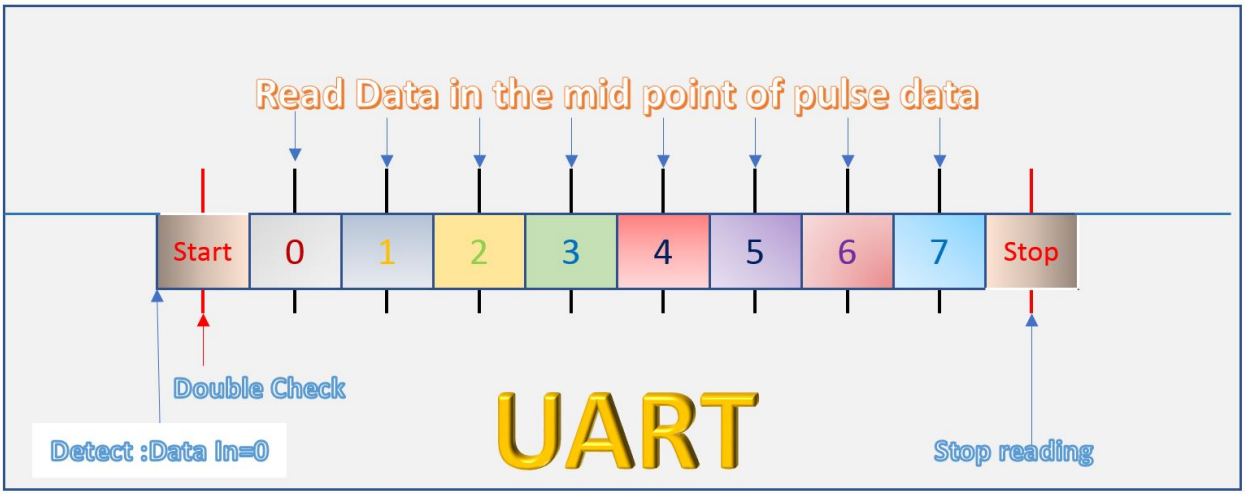

Figure 29: UART

Working Mechanism:
1) Uart receives parallel data from the data bus and creates a serial packet and adds a start bit stop bit and a parity bit.
2) Start bit pulls UART line to 0 (usually kept at 1), which indicates the receiver to capture incoming data while stop bit will stop communication by pulling the line back to 1.
3) The receiver will ignore start, stop, and parity bits and captures data into the parallel stream, and sends it to the data buses. [1]

Now, we will talk about the UART transmitter and UART receiver in details.

9. UART 128 bits

9.1. Baud Rate

Baud rate is the rate at which information is transferred in a communication channel that includes the number of bits exchanged per second in serial communication between two devices. It is quite important in the UART channel as UART protocol is asynchronous and thus, is necessary for the transmitter and receiver to transfer and receive the data at the same speed. In the serial port, the baud rate of "9600 baud" means a maximum of 9600 bits of data can be transferred per second.

For example, the FPGA board has a 100 Mhz clock cycle, with a baud rate of 9600 bits per seconds, each bit is being transferred and received every 10417 cycles (i.e. 100000000 cycles/sec — 9600 bits/sec). And, each bit is being captured exactly in the midway of 10417 cycles for the consistency and accuracy of the data.

The UART line stays at logic high '1', and a transmitter is activated, with start bit '0', the line is active and transmits 8 bits serially, the end recognized by the stop bit '1', and then the UART line stays at high logic '1' and wait for the next stream of data. A similar procedure occurs in the receiver side, which remains at logic high '1' and as it receives the start bit '0', it captures the next 8 bits of data at the baud rate serially and remains logic high '1' at the completion and wait for the next stream of bits. [2]

Table 17: Baudrate and clocks per bit selection

FPGA clock speed	100MHZ
Baud_rate	9600 bits per seconds
clock_per_bits	10417 clock cycles per bit

9.2. Uart Transmitter

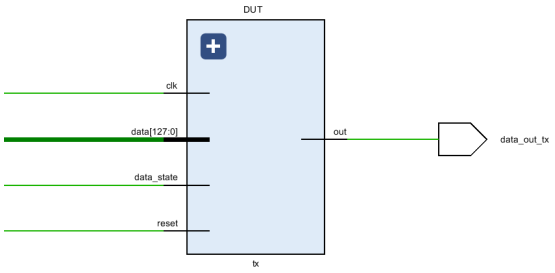

Figure 30: UART-128 bit Transmitter Input Output

Table 18: Input/Output wire for Uart Transmitter

I/O	Wire	Size(bit)	Utilization
	clock	1	Clock for operations
	data	128	data that needs to be sent
	data_state	1	Data_ state is a flag that initiates the transmitter module.
Input	reset	1	Resets the transmitter to state Init.
Output	data_out_tx	1	Output pin.

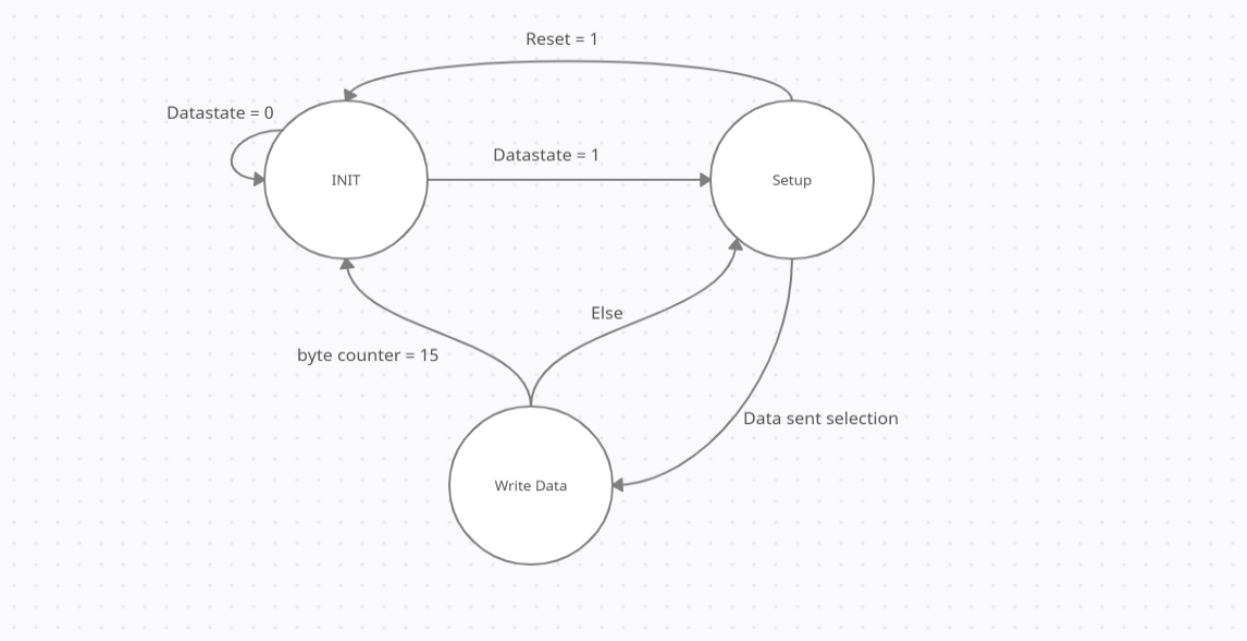

Figure 31: UART-128 bit Transmitter State Diagram

1. Init: In this state is the transmitter goes to state setup if the data_state flag from the receiver is 1.

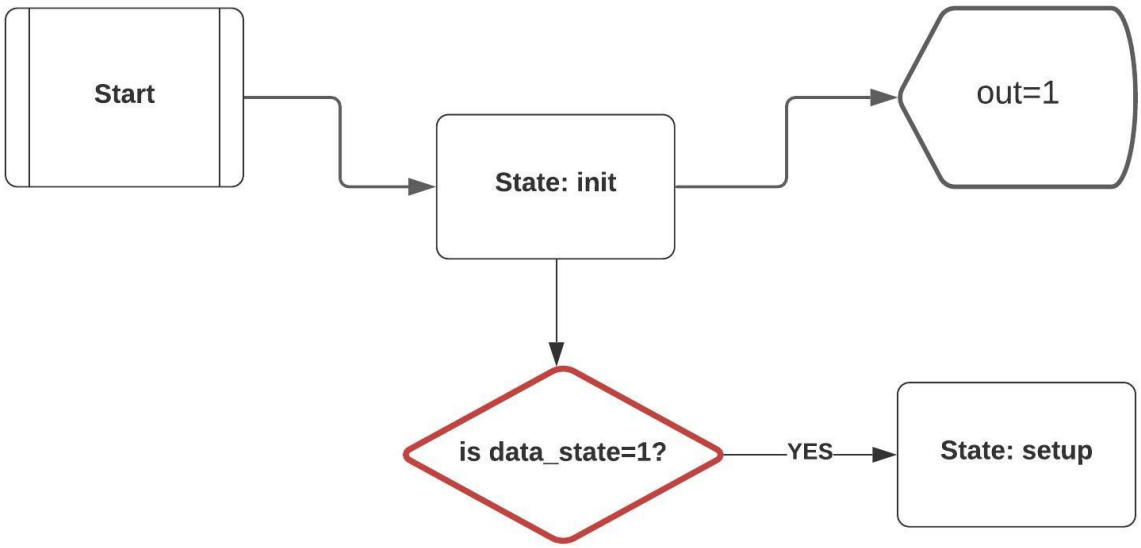

Figure 32: State Init

2. Setup:In this state, the transmitter breaks the 128 bit data into 16 8bit chunks, assigns start and stops bits making it 10 bits of data and passes it to the next state i.e. Write Data.

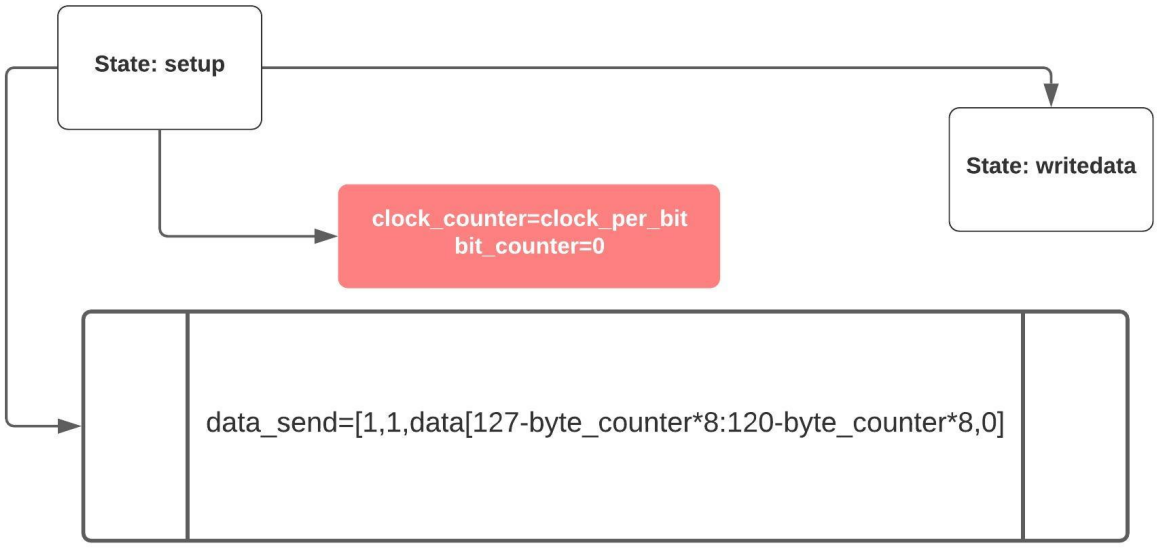

Figure 33: State Setup

3. Write Data:In this state, the transmitter sends the chunk of data received from Setup state and assigns it to the output transmitter pin every clocks_per_bits number of cycles. If the number of bytes sent is 16 then it goes to state Init. If the number of bytes sent is not 16 , it increments the number of bytes sent counter by 1 ignoring the start and stop bits and goes to state Setup again.

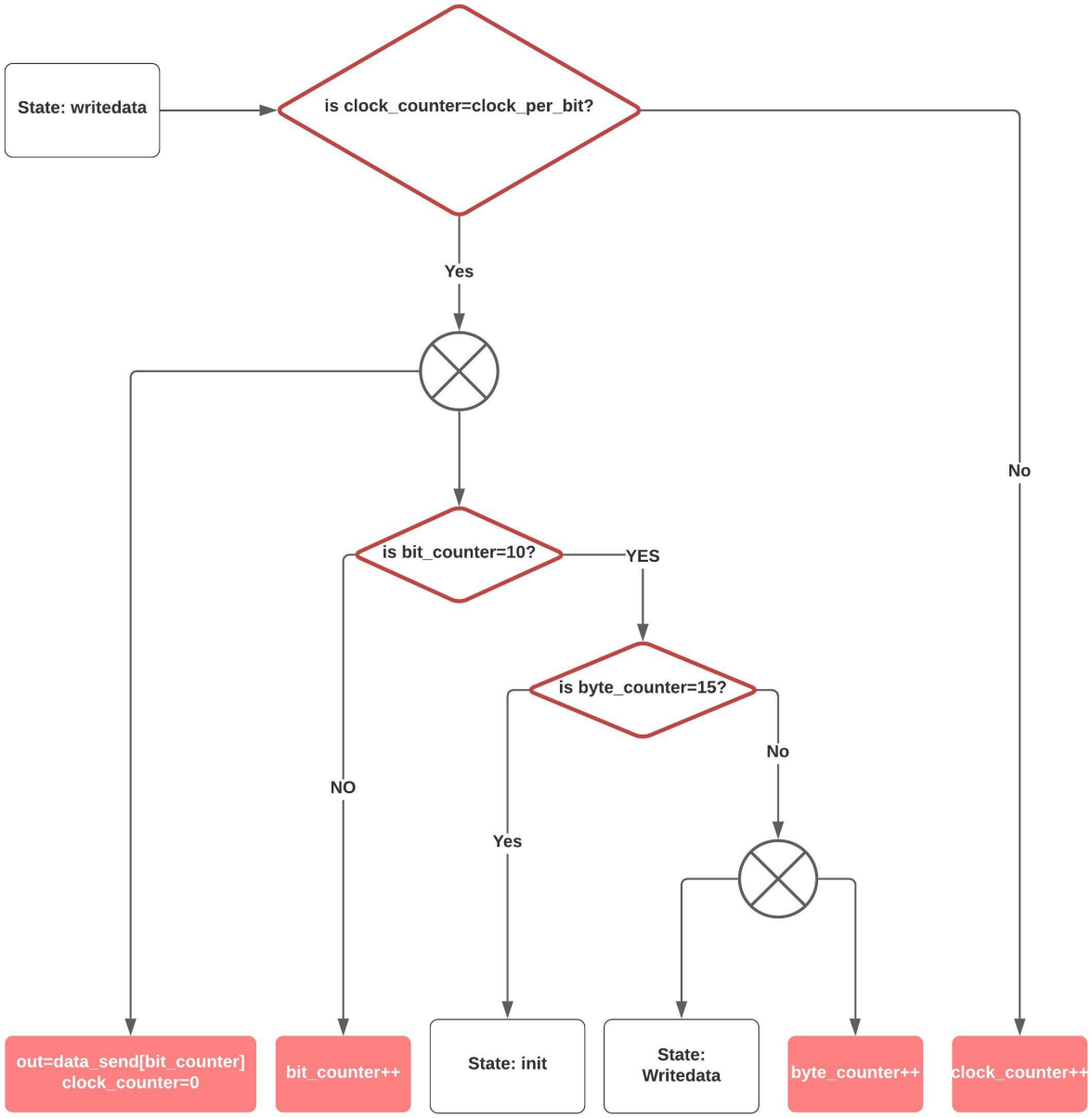

Figure 34: State Writedata

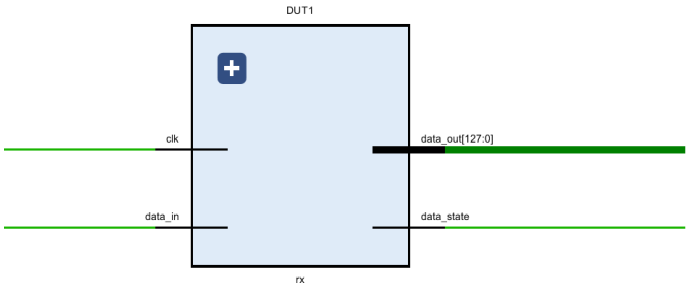

Figure 35: UART-128 bit Receiver Input Output

Table 19: Input/Output wire for Uart Reciever

I/O	Wire	Size(bit)	Utilization
Input	clock	1	Clock for operations
	data_in	1	It represents the pin that acts as reciever for FPGA.
Output	data_out	128	Output pin.
	data_state	1	Flag that represents new set of 128 bit has been received.

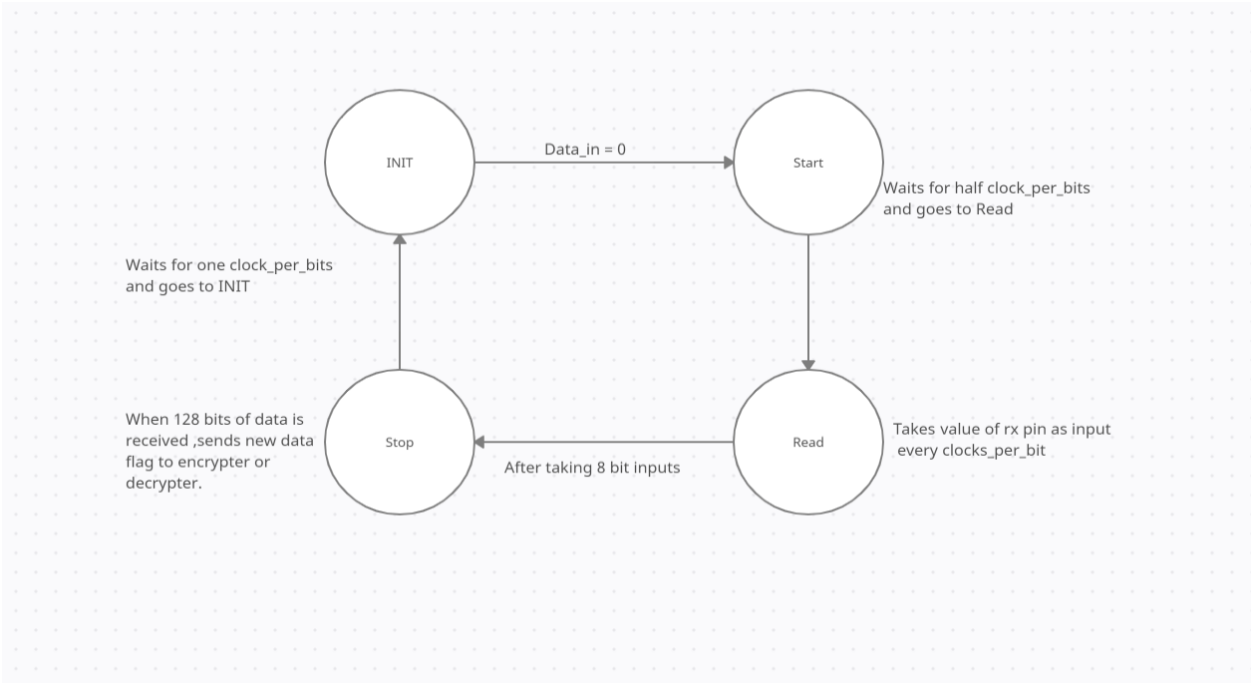

Figure 36: UART-128 bit Receiver State Diagram

1. Init: The FPGA waits until Data_in = 0 and then goes to Start.

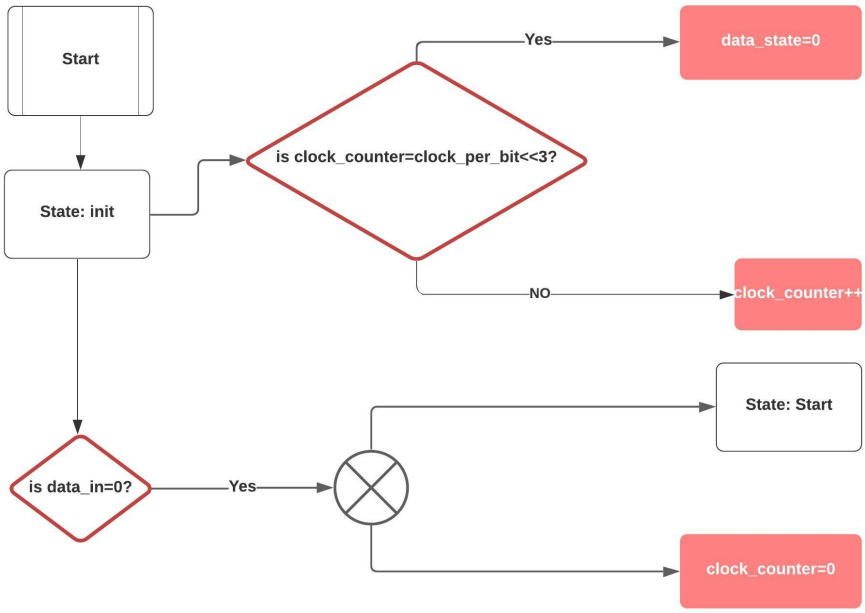

Figure 37: State Init

2. Start: It waits for half clocks_per_bits and checks if the Data_in is still 0. If Data_in is still 0, it proceeds to next state i.e. Read.

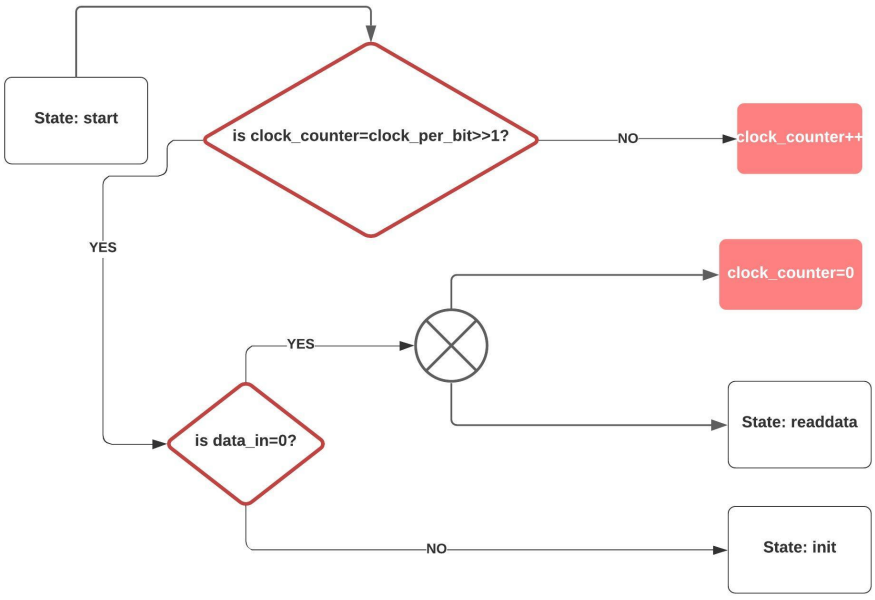

Figure 38: State Start

3. Read:In this state, the receiver reads 8 bits of data from the Data_in pin every clock_per_bits cycle and increments the number of bytes received by 1. Then, it proceeds to next state i.e. Stop.

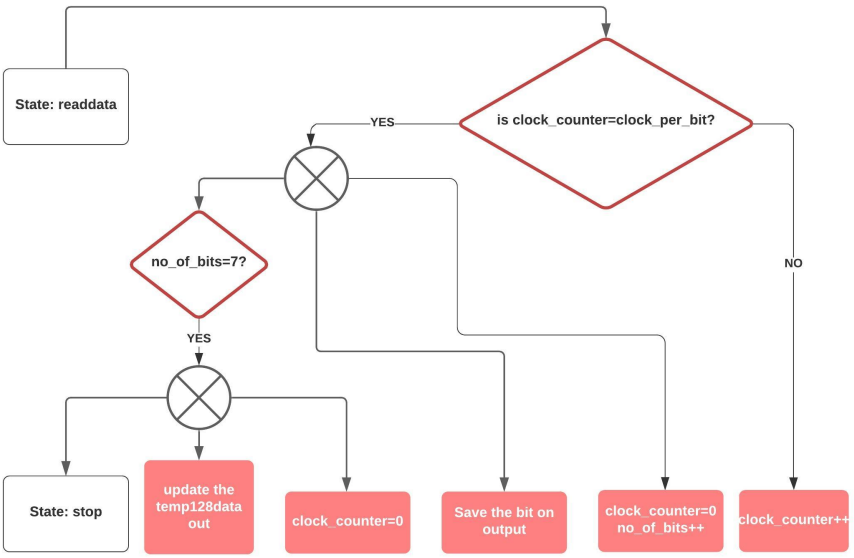

Figure 39: State Read data

4. Stop:In this state, the receiver checks is the number of bytes received is 16 or not. If it has received 16 bytes of data, it updates the data_out register and also sends a flag that new data has been received.Then, it proceeds to next state i.e. Init.

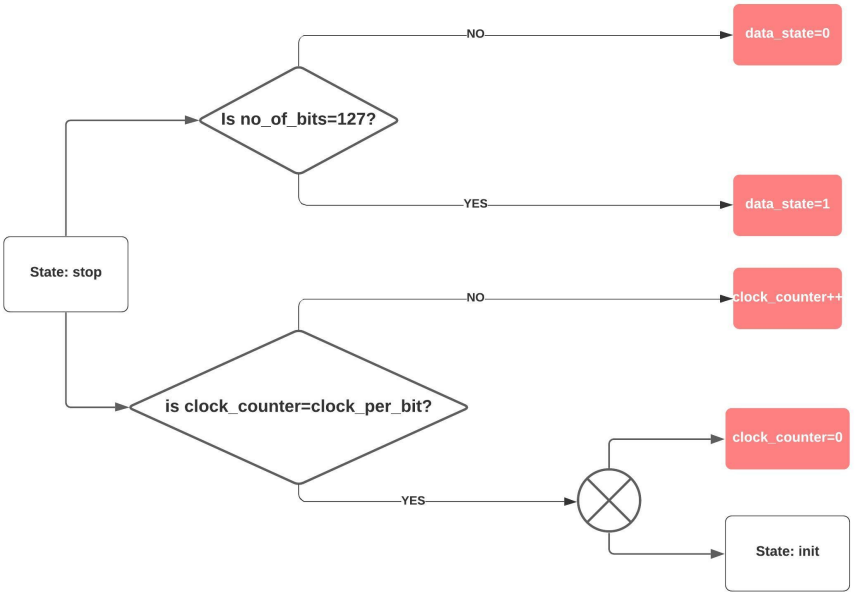

Figure 40: State Stop

5. Every Clock Cycle: Every rising edge of FPGA clock, the device checks if the state is stop or init. If the state is stop, then it saves the 8 bit acquired data to a temporary input. It also check if the total acquired data is 128 data and released the 128 bit to the output of receiver.
   If the state is init then, it clears the output of data after waiting for clock_per_bit times 160 clock cycles.

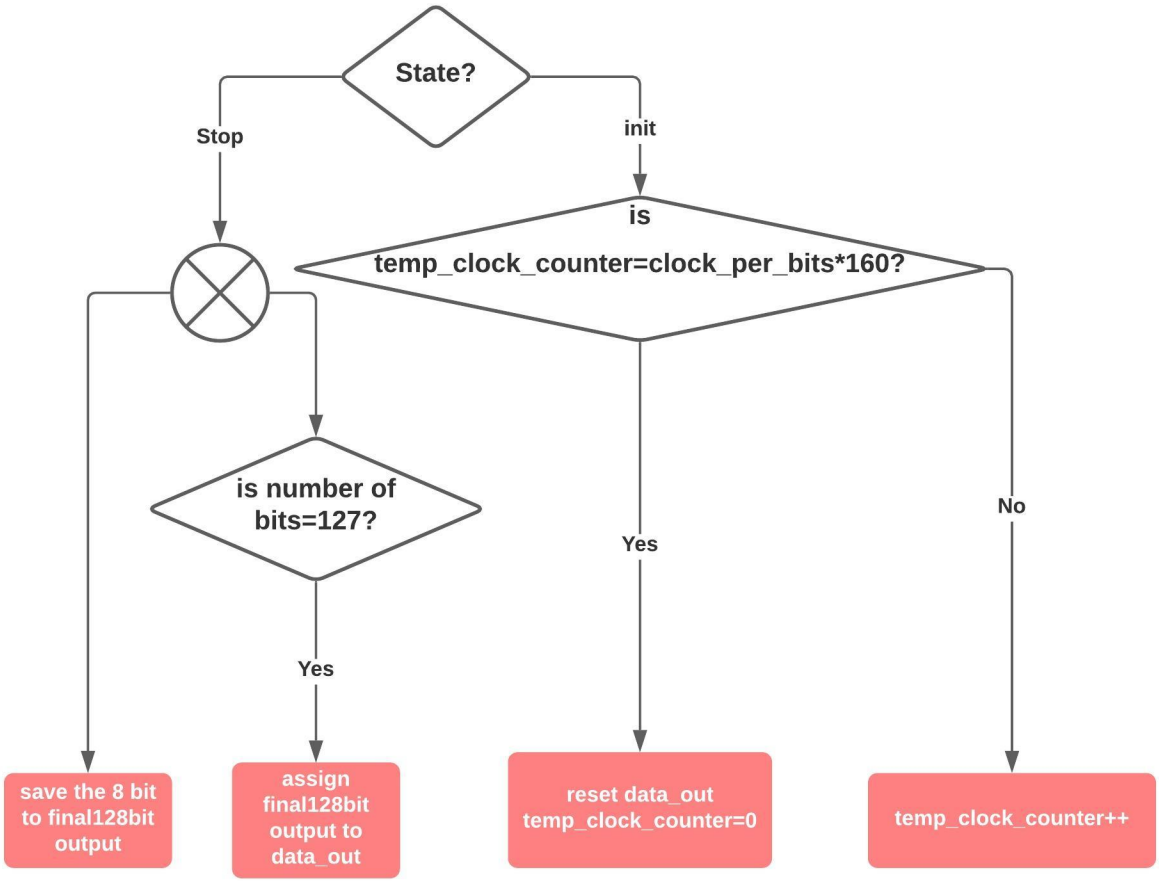

Figure 41: Every rising edge of FPGA clock

10. Pipelined Implementation

The entire process of encryption and decryption is broken down to small steps which are pipelined for efficient transmission of files.

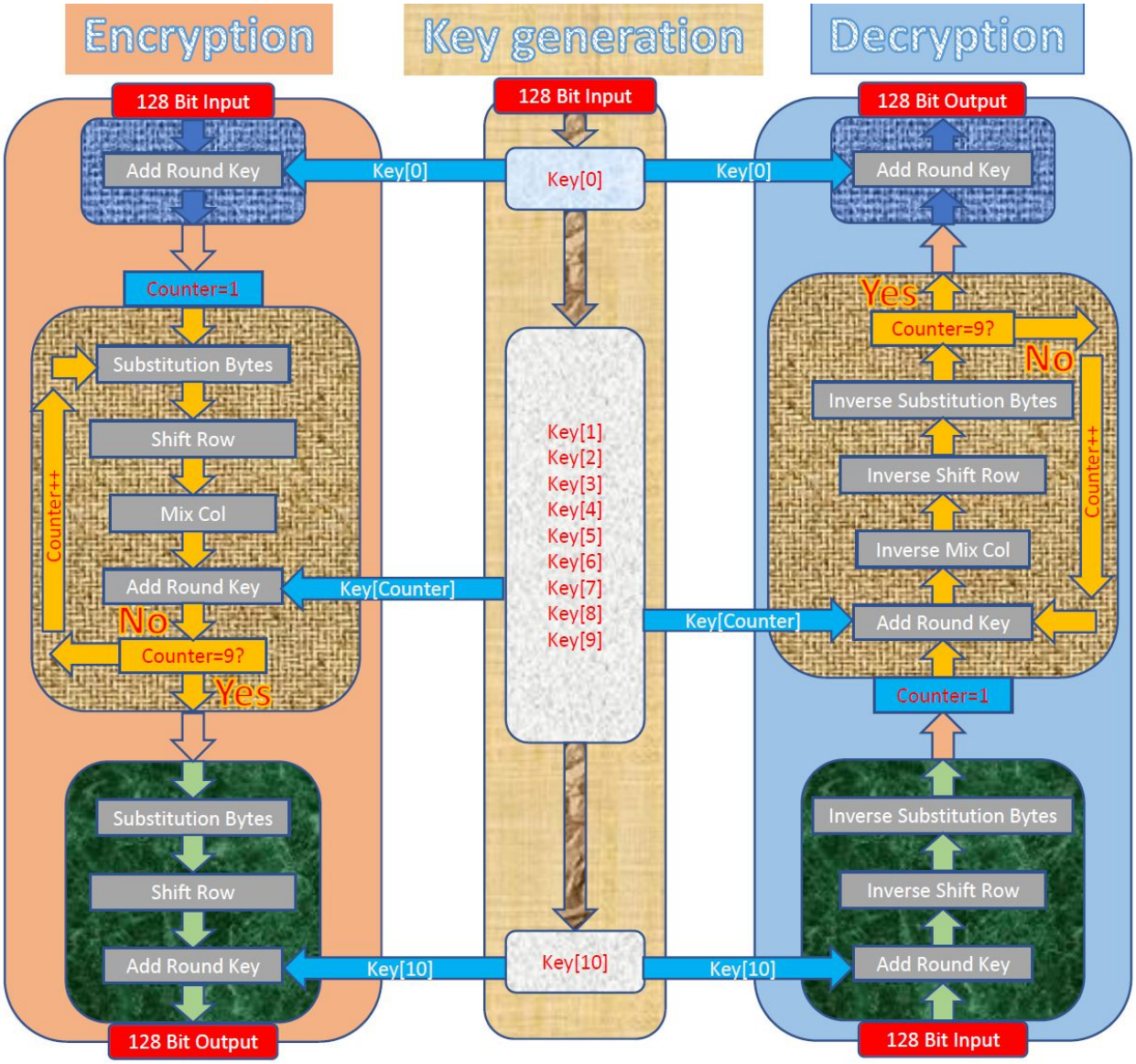

Figure 42: Full AES Encryption, Decryption and Key Generation.

10.1. Encryption Pipeline

This is the encryption process. Key generation is a circuit without registers so they are ignored. Appropriate keys are passed at appropriate circuits.

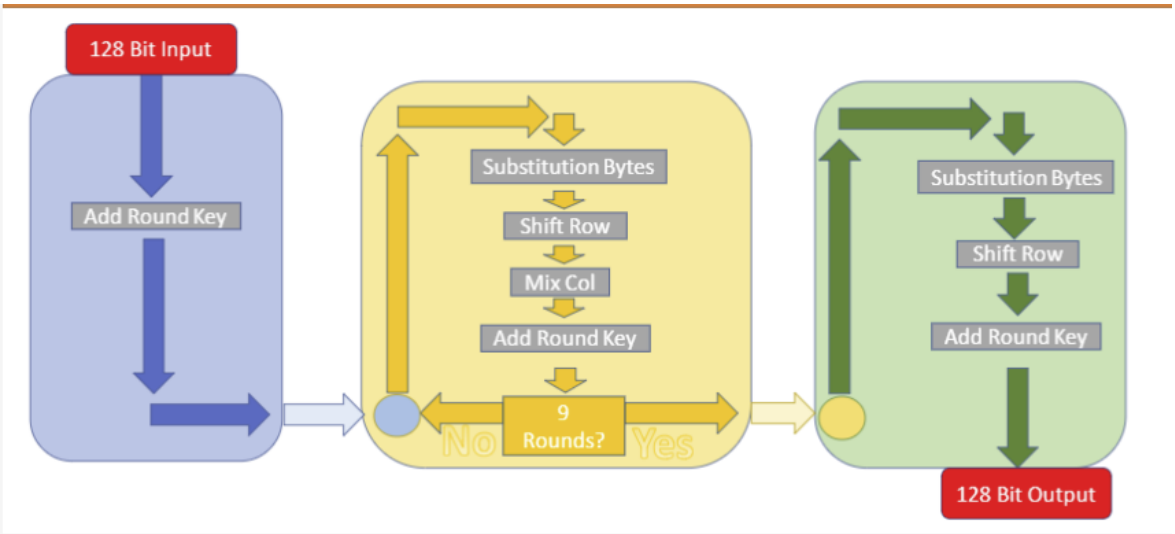

Figure 43: Full AES_128 bit Encryption.

10.1.1. Division of encryption circuit into three parts

The full encryption shown in figure 43 is broken down to three chunks as shown in figure 44. These three chunks as combined to form three circuits as shown in figure 45.

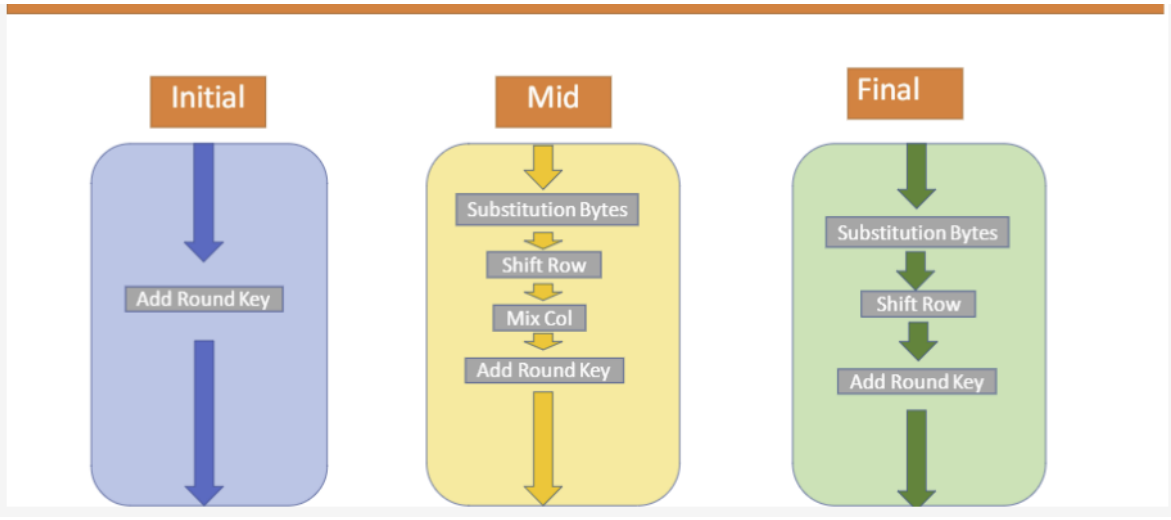

Figure 44: Encryption Broken down to different parts that can be in turn implemented to state machine.

10.1.2. Three circuits for Encryption

Each of these circuit take in 128 bit data and output 128 bit data. Combination of these three circuits represents complete encryption. The circuit First has three parts as the inital circuit has only add roundkey whilst Between circuit and Final Circuit are more complex.

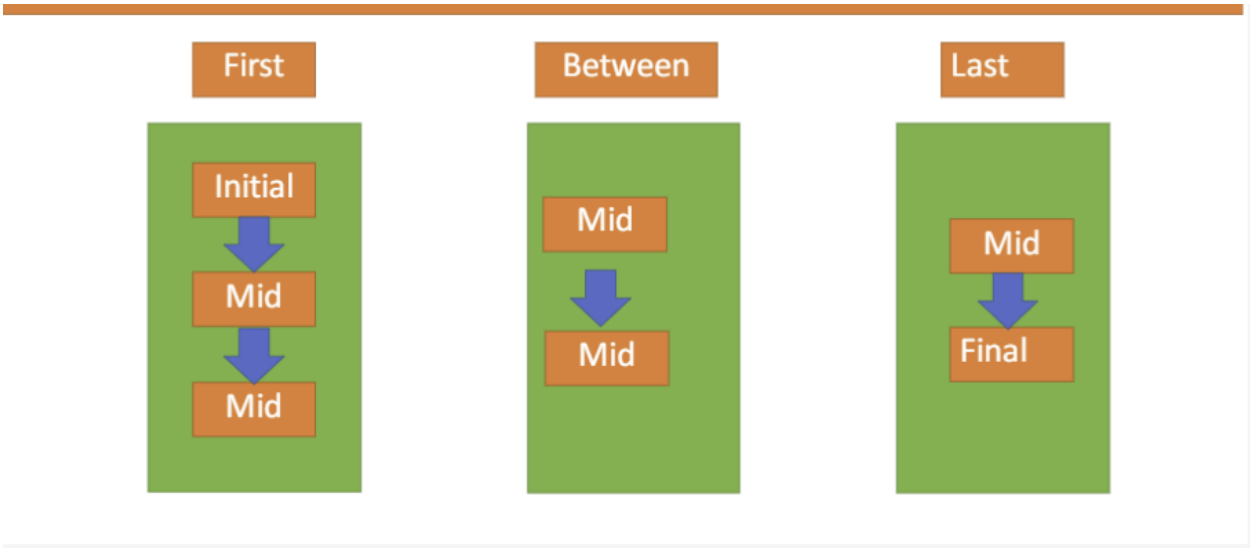

Figure 45: Three main circuit for Encryption.

Each of these three circuit are combine with a state machine circuit. The state machine circuit algorithm is shown in figure 46. The FSM circuit takes a data_state and 128 bit data as input.The FSM assigns the 128 bit data to the circuit associated with it. After waiting for clock_per_bit times 16 clock cycles, the FSM changes the output of to the new 128 bit data which is the output of the device associated with it. It also sends a completion flag (data_state)as output.

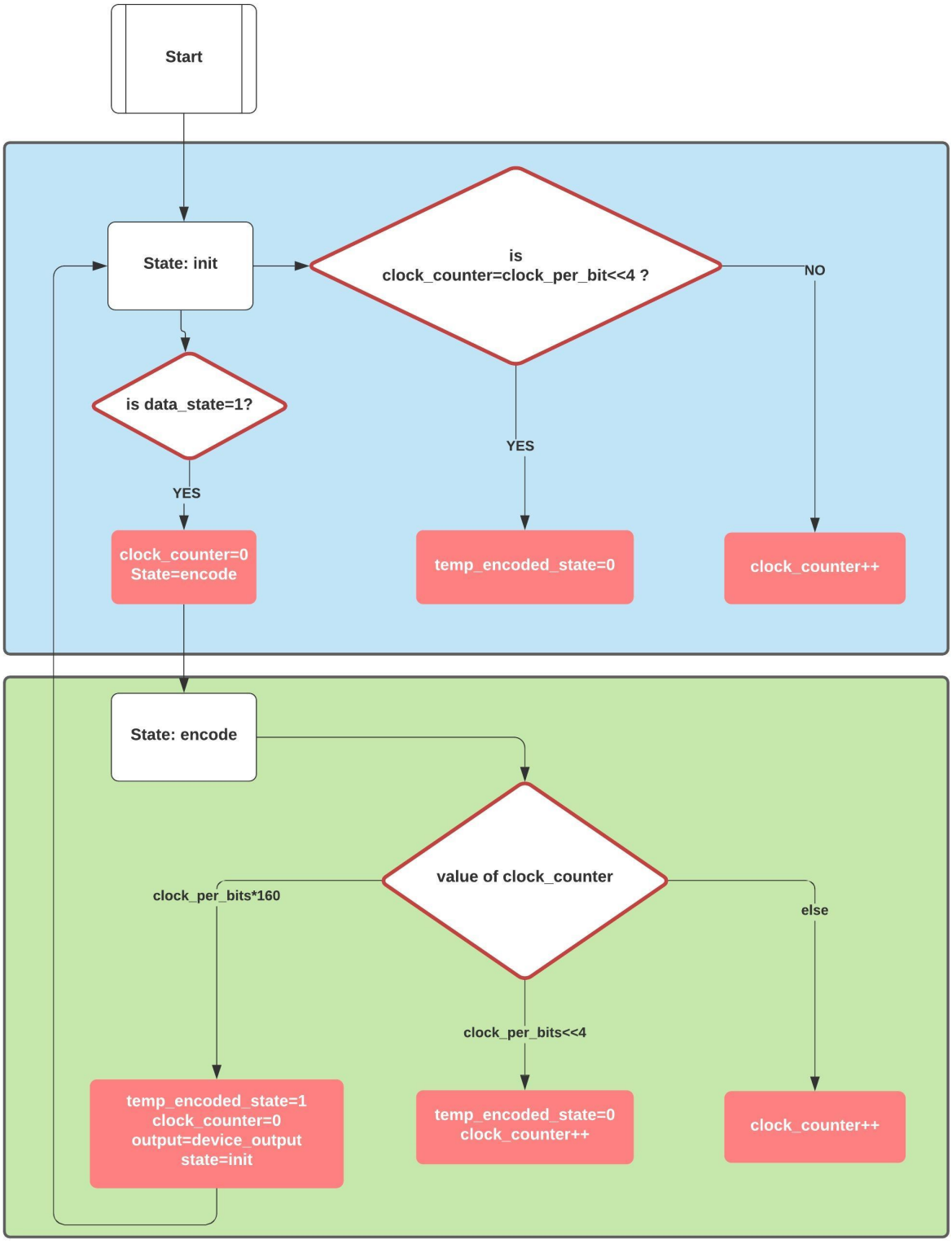

Figure 46: State machine implementation of different part of Encryption.

10.1.3. *Final Encryption Pipeline*

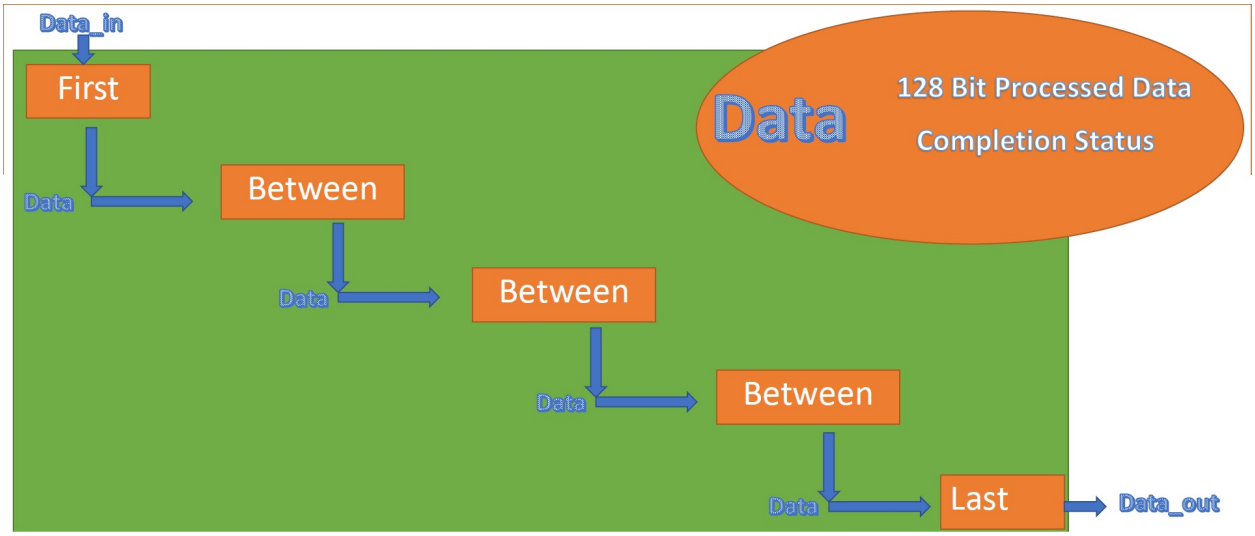

Figure 47: Final Encryption Pipeline

Each of these devices above i.e. First, Between and Last has the following input and output as shown in the table 20. The encode_state is connected to data_state of the next circuit and the data_out is connected to the data_in of the next circuit.

Table 20: Input/Output wires for First, Between and last

I/o	Wire	Size
Input	data_in	128
	data_state	1
Output	data_out	128
	encode_state	1
These are wires which are connected to internal registers		

10.2. Decryption Pipeline

This is the decryption process. Key generation is a circuit without registers so they are ignored. Appropriate keys are passed at appropriate circuits.

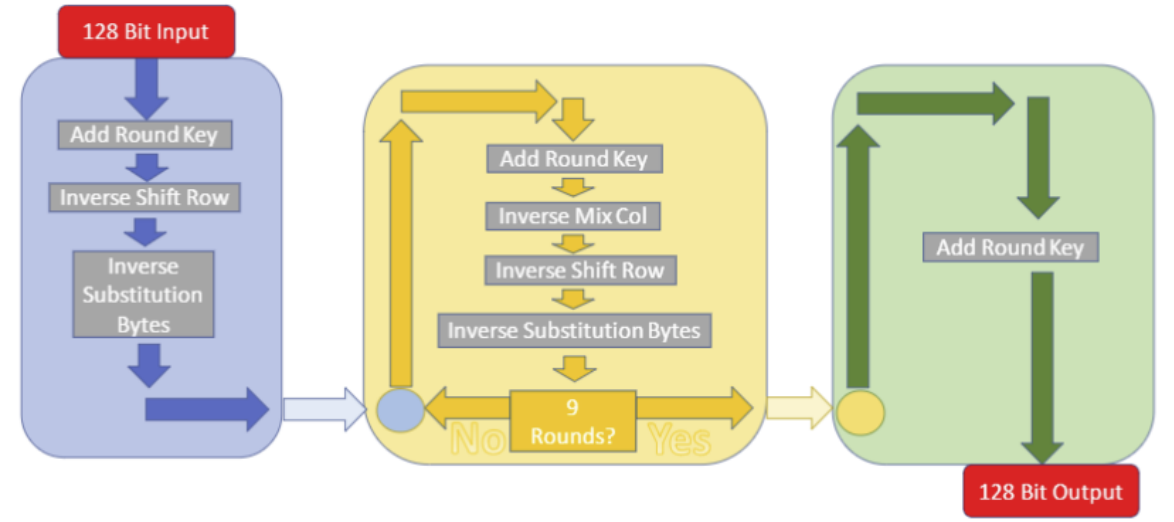

Figure 48: Full AES_128 bit Decryption.

10.2.1. Division of decryption circuit into three parts

The full decryption shown in figure 48 is broken down to three chunks as shown in figure 49. These three chunks as combined to form three circuits as shown in figure 50.

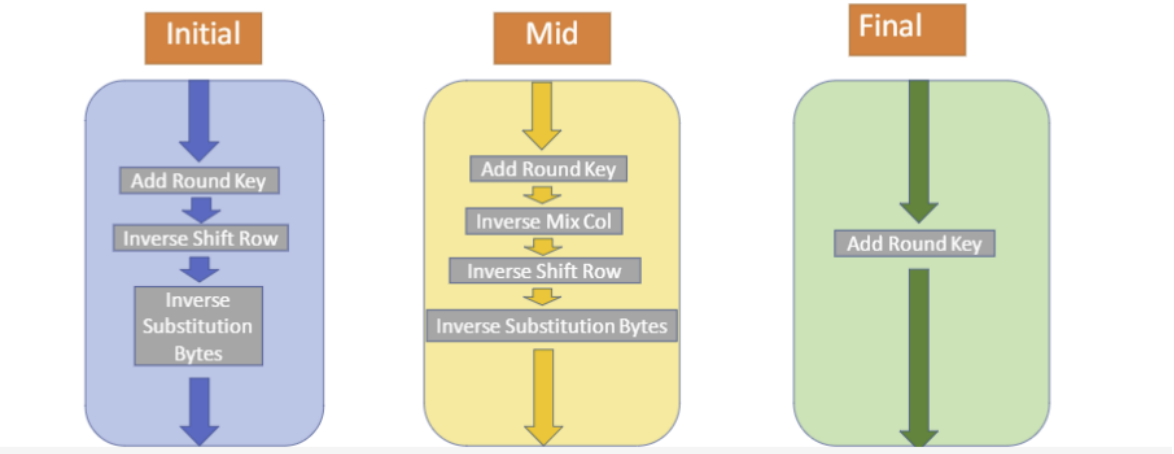

Figure 49: Decryption broken down to different parts that can be in turn implemented to state machine.

10.2.2. Three circuits for Decryption

Each of these circuit take in 128 bit data and output 128 bit data. Combination of these three circuits represents complete encryption. The circuit Last has three parts as the final circuit has only add roundkey whilst between circuit and First Circuit are more complex.

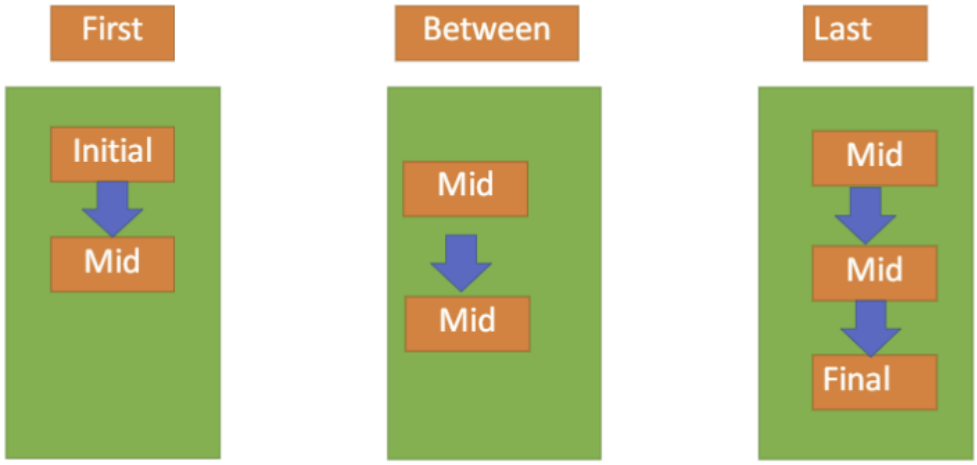

Figure 50: Three main circuit for Decryption.

Each of these three circuit are combined with a state machine circuit. The state machine circuit algorithm is shown in figure 51. The FSM circuit takes a data_state and 128 bit data as input.The FSM assigns the 128 bit data to the circuit associated with it. After waiting for clock_per_bit times 16 clock cycles, the FSM changes the output of to the new 128 bit data which is the output of the device associated with it. It also sends a completion flag (data_state)as output.

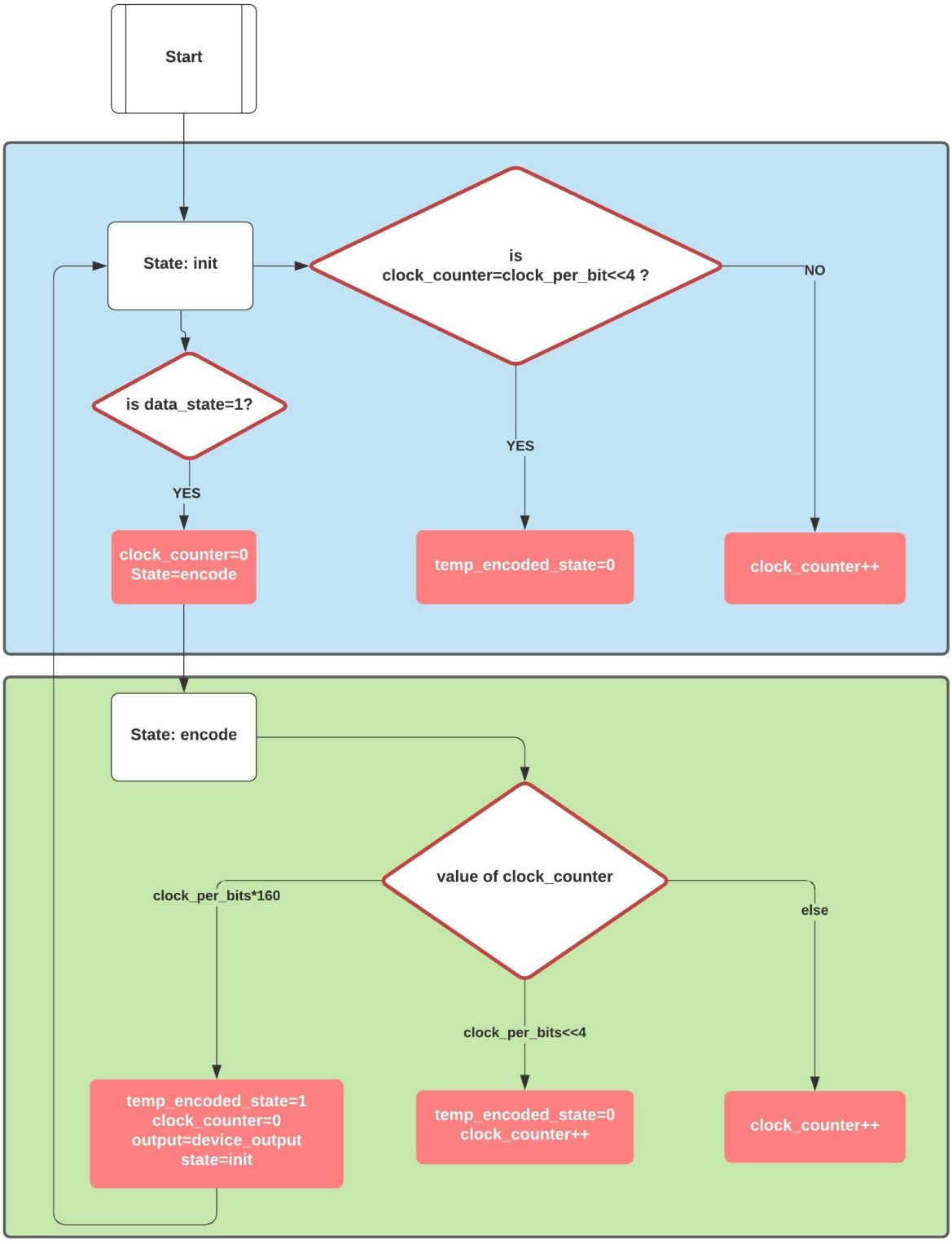

Figure 51: State machine implementation of different part of Decryption.

10.2.3. *Final Decryption Pipeline*

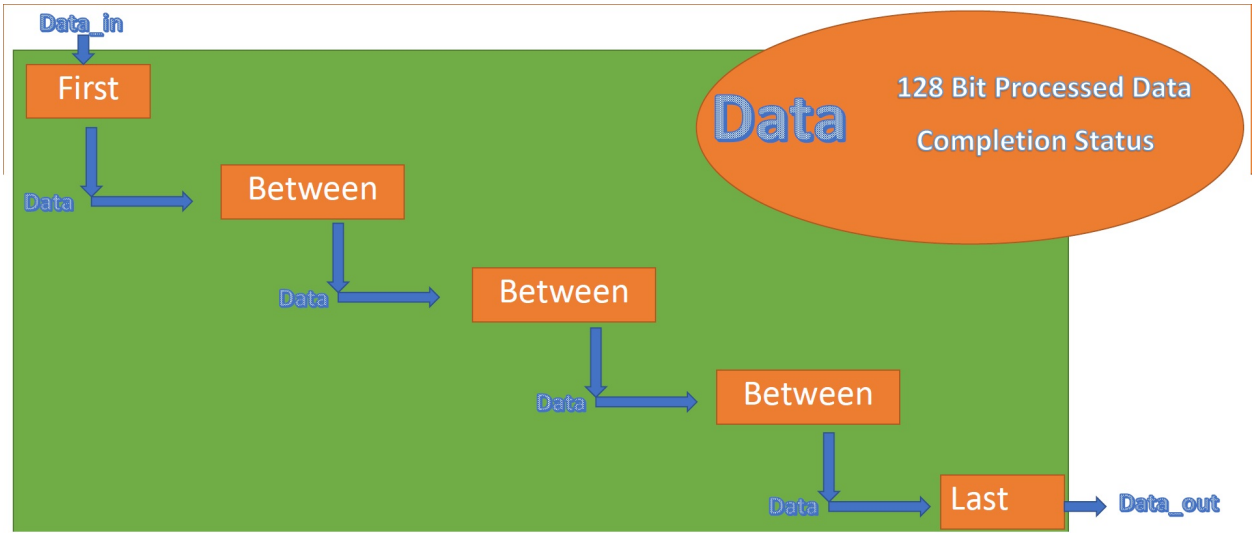

Figure 52: Final Decryption Pipeline

Each of these devices above i.e. First, Between and Last has the following input and output as shown in the table 21. The encode_state is connected to data_state of the next circuit and the data_out is connected to the data_in of the next circuit.

Table 21: Input/Output wires for First, Between and last

I/o	Wire	Size
Input	data_in	128
	data_state	1
Output	data_out	128
	encode_state	1
These are wires which are connected to internal registers		

10.3. Full Implementation

Finally each of these devices i.e. receiver, encrypter/decrypter and transmitter are connected. Receiver takes in data, completes receiving 128 bit data and outputs the 128 bit data and a completion flag. When the encrypter/decrypter sees a completion flag, it starts to encrypt/decrypt the data. Once encrypter/decrypter completes encryption/decryption, it outputs the encrypted/decrypted data along with a completion flag. Once the transmitter sees a completion flag, it starts to transmit the 128 bit data which is the output of encrypter/decrypter. The python codes is created with encryption/decryption requirement of 128 bit data.

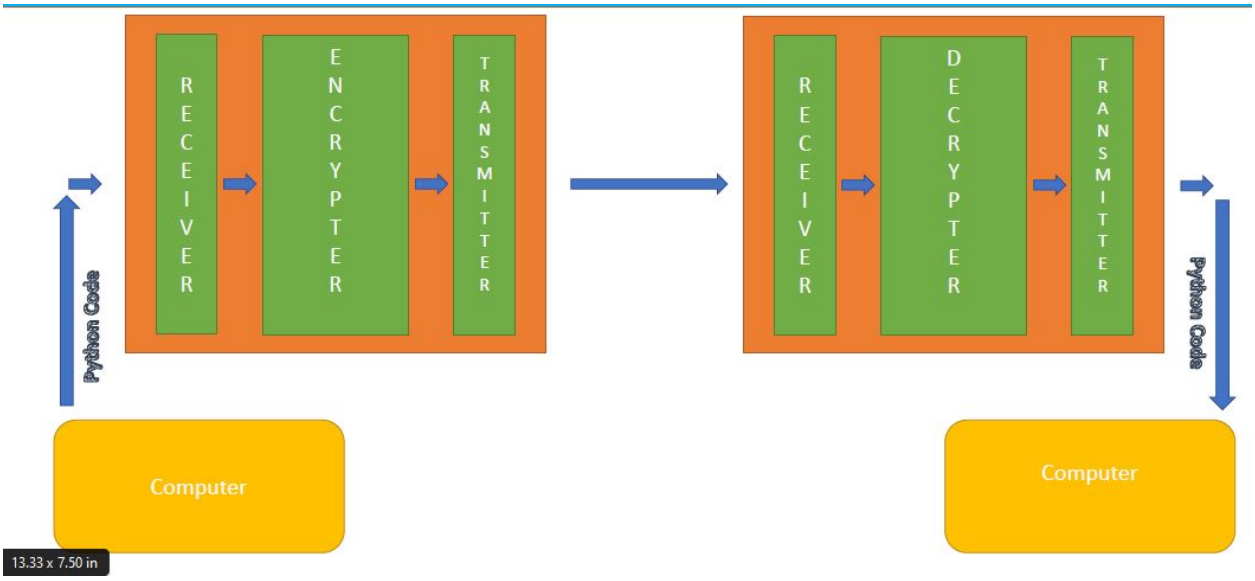

Figure 53: State machine implementation of different parts of Encryption

Python data_sender code reads a file, and makes it a multiple of 128 bit by adding 8'b11111111 multiple times at the end of the data. Similarly, the python receives code, reads data and ends reading data whenever it encounter 8'b1111111. Hence, the data used to convert out data to a multiple of 128 bit is also used as end flag by receiver.

11. Result

11.1. Simulation for encryption with uart

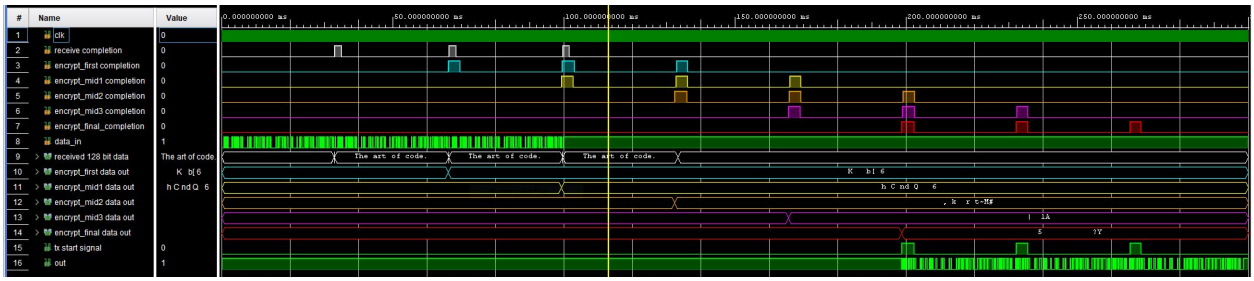

Figure 54: Final implementation, Computer to FPGA with encryption

11.2. Simulation for decryption with uart

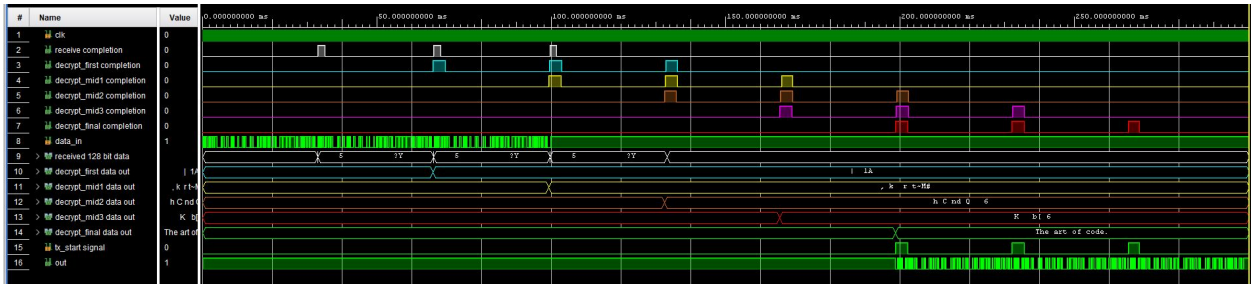

Figure 55: Final implementation, FPGA to Computer with decryption

11.3. FPGA Implementation

A large sized text is sent from computer and the another computer receives the data. The data is received without corruption. The encryption and decryption process is hidden from the user. There is a latency of 30 ms due to receiving and transmitting process along with encryption and decryption process.

12. Challenges

1. **Timing Challenges**
   Hardware Description Language describes hardware. It was quite challenging to grasp concurrent programming intuition. It was also hard to think parallel. Timing error arised due to pre-release of data output by a circuit which was less than the setup time.

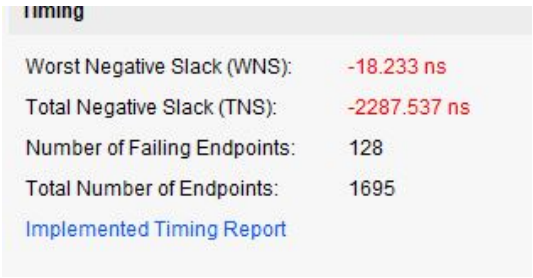

Figure 56: Timing error without pipeline of encryption/decryption

Figure 57: Timing error with pipeline of encryption/decryption: 2 circuits

Figure 58: Timing validation with pipeline of encryption/decryption with 5 circuits

Similarly, error occurred due to mis-matched receiver and transmitter. This is because with real time communication, there is no storing of data and data are passed from registers to registers until it is processed.

2. **Time management**
   Just as every other group project, it was tiresome to toggle schedules based on everyone's need.

3. **Need Knowledge of Abstract Algebra**
   Strong knowledge of Abstract Algebra was required to implement the AES algorithm on hardware level. Extra study on the subject material was required.

13. Conclusion

The project was successful in realizing the objectives set at the very beginning. The FPGA Implementation of AES 128 algorithm is capable of accepting and transmitting data, regardless of size, from either a computer or another FPGA, perform encryption/decryption and communicate using the UART communication channel. Its features and cost makes it well suited for any low cost embedded application in the field of data security.

14. Verilog : Encryption(CODE)

14.1. encryption_main.v

```
module encryption_main(
clk,
data,
rx_state,
key,
encrypted_data,
encrypted_data_state
    );
input clk;
input rx_state;
input  [0:127] key;
input  [0:127] data;
output [0:127] encrypted_data;
output encrypted_data_state;


wire  [0:127] key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10;

key keygen(key,key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10);

main_encrypt encode(clk,data,rx_state,encrypted_data,encrypted_data_state,
            key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10);

endmodule
```

14.2. main_encrypt.v

```
`timescale 1ns / 1ps

module main_encrypt(
                        clk,
                        data,
                        rx_state,
                        encrypted_data,
                        encrypted_data_state,
                        key_0,key_1,key_2,key_3,key_4,
                        key_5,key_6,key_7,key_8,key_9,
                        key_10
                        );
input          clk,rx_state;
input  [0:127] data;
output [0:127] encrypted_data;
output         encrypted_data_state;
input  [0:127] key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10;

wire   [0:127] encoded_data_1,encoded_data_2,encoded_data_3,encoded_data_4;
wire   encoded_state_1, encoded_state_2, encoded_state_3, encoded_state_4;
```

```
encrypt_first  first(clk,        data  ,key_0,key_1,key_2,
                    rx_state       ,encoded_data_1,encoded_state_1     );
encrypt_mid    mid1 (clk,encoded_data_1,key_3,key_4      ,
                    encoded_state_1,encoded_data_2,encoded_state_2     );
encrypt_mid    mid2 (clk,encoded_data_2,key_5,key_6      ,
                    encoded_state_2,encoded_data_3,encoded_state_3     );
encrypt_mid    mid3 (clk,encoded_data_3,key_7,key_8      ,
                    encoded_state_3,encoded_data_4,encoded_state_4     );
encrypt_final  last (clk,encoded_data_4,key_9,key_10     ,
                    encoded_state_4,encrypted_data,encrypted_data_state);

endmodule
```

14.3. encrypt_first.v

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps

module encrypt_first(
            clk,
            data,
            key1,key2,key3,
            data_state,
            encoded_data,
            encoded_state
    );
input [127:0] data,key1,key2,key3;
input         clk,data_state;
output [127:0]encoded_data;
output        encoded_state;


//change these to change baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg state=1'b0;//STATES
localparam init=1'b0,encode=1'b1;//states

reg  [127:0] temp_encoded_data=0;
reg temp_encoded_state=0;;
reg[24:0] clock_counter=25'b0;//counts how many clock cycles passed


wire [127:0] tempered_data1,tempered_data2,tempered_data3;
```

```
main_first   DUM1(data,tempered_data1,key1);
main_mid   DUM2(tempered_data1,tempered_data2,key2);
main_mid DUM3(tempered_data2,tempered_data3,key3);


always@(posedge clk) begin
case (state)
init:
    begin
    if (clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
    end
    else
    begin
        clock_counter<=clock_counter+1;
    end

    if (data_state==1)
    begin
        clock_counter<=0;
        state<=encode;
    end
    end
encode:
    begin
    if (clock_counter==clock_per_bit*159)
    begin
       temp_encoded_state<=1'b1;
       temp_encoded_data<=tempered_data3;
       clock_counter<=0;
       state<=init;

    end
    else if(clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
        clock_counter<=clock_counter+1;
    end
    else
    begin
        clock_counter<=clock_counter+1;

    end
    end
endcase
end
assign encoded_state=temp_encoded_state;
assign encoded_data=temp_encoded_data;
endmodule
```

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps

module encrypt_mid(
            clk,
            data,
            key1,key2,
            data_state,
            encoded_data,
            encoded_state
    );
input [127:0] data,key1,key2;
input         clk,data_state;
output [127:0]encoded_data;
output        encoded_state;


//change these to change baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg state=1'b0;//STATES
localparam init=1'b0,encode=1'b1;//states


reg  [127:0] temp_encoded_data=0;
reg temp_encoded_state=0;;
reg[24:0] clock_counter=25'b0;//counts how many clock cycles passed



wire [127:0] tempered_data1,tempered_data2;

main_mid   DUM1(data,tempered_data1,key1);
main_mid   DUM2(tempered_data1,tempered_data2,key2);


always@(posedge clk) begin
case (state)
init:
    begin
    if (clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
    end
    else
```

```
    begin
        clock_counter<=clock_counter+1;
    end

    if (data_state==1)
    begin
        clock_counter<=0;
        state<=encode;
    end
    end
encode:
    begin
    if (clock_counter==clock_per_bit*159)
    begin
       temp_encoded_state<=1'b1;
       temp_encoded_data<=tempered_data2;
       clock_counter<=0;
       state<=init;
    end
    else if(clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
        clock_counter<=clock_counter+1;
    end
    else
    begin
        clock_counter<=clock_counter+1;

    end
    end
endcase
end
assign encoded_state=temp_encoded_state;
assign encoded_data=temp_encoded_data;
endmodule
```

14.5. encrypt_final.v

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps

module encrypt_final(
            clk,
            data,
            key1,key2,
            data_state,
            encoded_data,
            encoded_state
    );
input [127:0] data,key1,key2;
input         clk,data_state;
output [127:0]encoded_data;
output        encoded_state;
```

```
//change these to change baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg state=1'b0;//STATES
localparam init=1'b0,encode=1'b1;//states


reg  [127:0] temp_encoded_data=0;
reg temp_encoded_state=0;;
reg[24:0] clock_counter=25'b0;//counts how many clock cycles passed


wire [127:0] tempered_data1,tempered_data2;


main_mid   DUM1(data,tempered_data1,key1);
main_final DUM2(tempered_data1,tempered_data2,key2);


always@(posedge clk) begin
case (state)
init:
    begin
    if (clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
    end
    else
    begin
        clock_counter<=clock_counter+1;
    end

    if (data_state==1)
    begin
        clock_counter<=0;
        state<=encode;
    end
    end
encode:
    begin
    if (clock_counter==clock_per_bit*159)
    begin
       temp_encoded_state<=1'b1;
       temp_encoded_data<=tempered_data2;
       clock_counter<=0;
       state<=init;
```

```
    end
    else if(clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
        clock_counter<=clock_counter+1;
    end
    else
    begin
        clock_counter<=clock_counter+1;

    end
    end

endcase
end
assign encoded_state=temp_encoded_state;
assign encoded_data=temp_encoded_data;
endmodule
```

14.6. main_first.v

```
`timescale 1ns / 1ps
module
main_first(
data,
tempered_data,
key_0
    );
input  [0:127] data;
output [0:127] tempered_data;
input  [0:127] key_0;

addRoundKey RFAK (data,key_0,tempered_data);
endmodule
```

14.7. main_mid.v

```
`timescale 1ns / 1ps
module
main_mid(
data,
tempered_data,
key
    );
input  [0:127] data;
output [0:127] tempered_data;
input  [0:127] key;

wire   [0:127] data_after_subs;
wire   [0:127] data_after_mixcol;
```

```
wire   [0:127] data_after_shift;



substitution RiSB(data,data_after_subs);
shift_rows   RiSR(data_after_subs,data_after_shift);
mix_col      RiMC(data_after_shift,data_after_mixcol);
addRoundKey  RiRK (data_after_mixcol,key,tempered_data);

endmodule
```

14.8. main_final.v

```
`timescale 1ns / 1ps
module
main_final(
data,
tempered_data,
key_10
    );
input  [0:127] data;
output [0:127] tempered_data;
input  [0:127] key_10;

wire   [0:127] data_after_subs;
wire   [0:127] data_after_shift;

substitution RFSB(data,data_after_subs);
shift_rows   RFSR(data_after_subs,data_after_shift);
addRoundKey  RFAk (data_after_shift,key_10,tempered_data);
endmodule
```

14.9. addRoundKey.v

```
`timescale 1ns / 1ps

module addRoundKey(
    input [127:0] data,
    input [127:0] key,
    output [127:0] data_with_key
    );

assign data_with_key=data^key;
endmodule
```

14.10. substitution.v

```
module substitution(
 data,
substituted_data
```

```
);
input [0:127] data;
output [0:127] substituted_data;

aes_S_box aes01( data[000:007] , substituted_data[000:007] );
aes_S_box aes02( data[008:015] , substituted_data[008:015] );
aes_S_box aes03( data[016:023] , substituted_data[016:023] );
aes_S_box aes04( data[024:031] , substituted_data[024:031] );
aes_S_box aes05( data[032:039] , substituted_data[032:039] );
aes_S_box aes06( data[040:047] , substituted_data[040:047] );
aes_S_box aes07( data[048:055] , substituted_data[048:055] );
aes_S_box aes08( data[056:063] , substituted_data[056:063] );
aes_S_box aes09( data[064:071] , substituted_data[064:071] );
aes_S_box aes10( data[072:079] , substituted_data[072:079] );
aes_S_box aes11( data[080: 87] , substituted_data[080:087] );
aes_S_box aes12( data[088: 95] , substituted_data[088:095] );
aes_S_box aes13( data[096:103] , substituted_data[096:103] );
aes_S_box aes14( data[104:111] , substituted_data[104:111] );
aes_S_box aes15( data[112:119] , substituted_data[112:119] );
aes_S_box aes16( data[120:127] , substituted_data[120:127] );
endmodule
```

14.11. aes_S_box.v

```
module aes_S_box(
data,
substituted_data
);

input [0:7] data;
output reg [0:7] substituted_data;
reg [0:7] c;
always @(data)
begin
case(data)
8'h00:c =8'h63;8'h01:c =8'h7c;8'h02:c =8'h77;8'h03:c =8'h7b;8'h04:c =8'hf2;
8'h05:c =8'h6b;8'h06:c =8'h6f;8'h07:c =8'hc5;8'h08:c =8'h30;8'h09:c =8'h01;
8'h0a:c =8'h67;8'h0b:c =8'h2b;8'h0c:c =8'hfe;8'h0d:c =8'hd7;8'h0e:c =8'hab;
8'h0f:c =8'h76; //0

8'h10:c =8'hca;8'h11:c =8'h82;8'h12:c =8'hc9;8'h13:c =8'h7d;8'h14:c =8'hfa;
8'h15:c =8'h59;8'h16:c =8'h47;8'h17:c =8'hf0;8'h18:c =8'had;8'h19:c =8'hd4;
8'h1a:c =8'ha2;8'h1b:c =8'haf;8'h1c:c =8'h9c;8'h1d:c =8'ha4;8'h1e:c =8'h72;
8'h1f:c =8'hc0; //1

8'h20:c =8'hb7;8'h21:c =8'hfd;8'h22:c =8'h93;8'h23:c =8'h26;8'h24:c =8'h36;
8'h25:c =8'h3f;8'h26:c =8'hf7;8'h27:c =8'hcc;8'h28:c =8'h34;8'h29:c =8'ha5;
8'h2a:c =8'he5;8'h2b:c =8'hf1;8'h2c:c =8'h71;8'h2d:c =8'hd8;8'h2e:c =8'h31;
8'h2f:c =8'h15; //2

8'h30:c =8'h04;8'h31:c =8'hc7;8'h32:c =8'h23;8'h33:c =8'hc3;8'h34:c =8'h18;
8'h35:c =8'h96;8'h36:c =8'h05;8'h37:c =8'h9a;8'h38:c =8'h07;8'h39:c =8'h12;
8'h3a:c =8'h80;8'h3b:c =8'he2;8'h3c:c =8'heb;8'h3d:c =8'h27;8'h3e:c =8'hb2;
```

```
8'h3f:c =8'h75; //3

8'h40:c =8'h09;8'h41:c =8'h83;8'h42:c =8'h2c;8'h43:c =8'h1a;8'h44:c =8'h1b;
8'h45:c =8'h6e;8'h46:c =8'h5a;8'h47:c =8'ha0;8'h48:c =8'h52;8'h49:c =8'h3b;
8'h4a:c =8'hd6;8'h4b:c =8'hb3;8'h4c:c =8'h29;8'h4d:c =8'he3;8'h4e:c =8'h2f;
8'h4f:c =8'h84; //4

8'h50:c =8'h53;8'h51:c =8'hd1;8'h52:c =8'h00;8'h53:c =8'hed;8'h54:c =8'h20;
8'h55:c =8'hfc;8'h56:c =8'hb1;8'h57:c =8'h5b;8'h58:c =8'h6a;8'h59:c =8'hcb;
8'h5a:c =8'hbe;8'h5b:c =8'h39;8'h5c:c =8'h4a;8'h5d:c =8'h4c;8'h5e:c =8'h58;
8'h5f:c =8'hcf; //5

8'h60:c =8'hd0;8'h61:c =8'hef;8'h62:c =8'haa;8'h63:c =8'hfb;8'h64:c =8'h43;
8'h65:c =8'h4d;8'h66:c =8'h33;8'h67:c =8'h85;8'h68:c =8'h45;8'h69:c =8'hf9;
8'h6a:c =8'h02;8'h6b:c =8'h7f;8'h6c:c =8'h50;8'h6d:c =8'h3c;8'h6e:c =8'h9f;
8'h6f:c =8'ha8; //6

8'h70:c =8'h51;8'h71:c =8'ha3;8'h72:c =8'h40;8'h73:c =8'h8f;8'h74:c =8'h92;
8'h75:c =8'h9d;8'h76:c =8'h38;8'h77:c =8'hf5;8'h78:c =8'hbc;8'h79:c =8'hb6;
8'h7a:c =8'hda;8'h7b:c =8'h21;8'h7c:c =8'h10;8'h7d:c =8'hff;8'h7e:c =8'hf3;
8'h7f:c =8'hd2; //7

8'h80:c =8'hcd;8'h81:c =8'h0c;8'h82:c =8'h13;8'h83:c =8'hec;8'h84:c =8'h5f;
8'h85:c =8'h97;8'h86:c =8'h44;8'h87:c =8'h17;8'h88:c =8'hc4;8'h89:c =8'ha7;
8'h8a:c =8'h7e;8'h8b:c =8'h3d;8'h8c:c =8'h64;8'h8d:c =8'h5d;8'h8e:c =8'h19;
8'h8f:c =8'h73; //8

8'h90:c =8'h60;8'h91:c =8'h81;8'h92:c =8'h4f;8'h93:c =8'hdc;8'h94:c =8'h22;
8'h95:c =8'h2a;8'h96:c =8'h90;8'h97:c =8'h88;8'h98:c =8'h46;8'h99:c =8'hee;
8'h9a:c =8'hb8;8'h9b:c =8'h14;8'h9c:c =8'hde;8'h9d:c =8'h5e;8'h9e:c =8'h0b;
8'h9f:c =8'hdb; //9

8'ha0:c =8'he0;8'ha1:c =8'h32;8'ha2:c =8'h3a;8'ha3:c =8'h0a;8'ha4:c =8'h49;
8'ha5:c =8'h06;8'ha6:c =8'h24;8'ha7:c =8'h5c;8'ha8:c =8'hc2;8'ha9:c =8'hd3;
8'haa:c =8'hac;8'hab:c =8'h62;8'hac:c =8'h91;8'had:c =8'h95;8'hae:c =8'he4;
8'haf:c =8'h79; //a

8'hb0:c =8'he7;8'hb1:c =8'hc8;8'hb2:c =8'h37;8'hb3:c =8'h6d;8'hb4:c =8'h8d;
8'hb5:c =8'hd5;8'hb6:c =8'h4e;8'hb7:c =8'ha9;8'hb8:c =8'h6c;8'hb9:c =8'h56;
8'hba:c =8'hf4;8'hbb:c =8'hea;8'hbc:c =8'h65;8'hbd:c =8'h7a;8'hbe:c =8'hae;
8'hbf:c =8'h08; //b

8'hc0:c =8'hba;8'hc1:c =8'h78;8'hc2:c =8'h25;8'hc3:c =8'h2e;8'hc4:c =8'h1c;
8'hc5:c =8'ha6;8'hc6:c =8'hb4;8'hc7:c =8'hc6;8'hc8:c =8'he8;8'hc9:c =8'hdd;
8'hca:c =8'h74;8'hcb:c =8'h1f;8'hcc:c =8'h4b;8'hcd:c =8'hbd;8'hce:c =8'h8b;
8'hcf:c =8'h8a; //c

8'hd0:c =8'h70;8'hd1:c =8'h3e;8'hd2:c =8'hb5;8'hd3:c =8'h66;8'hd4:c =8'h48;
8'hd5:c =8'h03;8'hd6:c =8'hf6;8'hd7:c =8'h0e;8'hd8:c =8'h61;8'hd9:c =8'h35;
8'hda:c =8'h57;8'hdb:c =8'hb9;8'hdc:c =8'h86;8'hdd:c =8'hc1;8'hde:c =8'h1d;
8'hdf:c =8'h9e; //d

8'he0:c =8'he1;8'he1:c =8'hf8;8'he2:c =8'h98;8'he3:c =8'h11;8'he4:c =8'h69;
8'he5:c =8'hd9;8'he6:c =8'h8e;8'he7:c =8'h94;8'he8:c =8'h9b;8'he9:c =8'h1e;
```

```
8'hea:c =8'h87;8'heb:c =8'he9;8'hec:c =8'hce;8'hed:c =8'h55;8'hee:c =8'h28;
8'hef:c =8'hdf; //e

8'hf0:c =8'h8c;8'hf1:c =8'ha1;8'hf2:c =8'h89;8'hf3:c =8'h0d;8'hf4:c =8'hbf;
8'hf5:c =8'he6;8'hf6:c =8'h42;8'hf7:c =8'h68;8'hf8:c =8'h41;8'hf9:c =8'h99;
8'hfa:c =8'h2d;8'hfb:c =8'h0f;8'hfc:c =8'hb0;8'hfd:c =8'h54;8'hfe:c =8'hbb;
8'hff:c =8'h16; //f
endcase
end
assign substituted_data=c;
endmodule
```

14.12. shift_rows.v

```
`timescale 1ns / 1ps


module shift_rows(
sub_data,
data_in
);
input  [0:127] sub_data;
output  [0:127] data_in;

assign data_in[ 0   : 7   ] = sub_data[ 0   : 7   ];
assign data_in[ 8   : 15  ] = sub_data[ 104 : 111 ];
assign data_in[ 16  : 23  ] = sub_data[ 80  : 87  ];
assign data_in[ 24  : 31  ] = sub_data[ 53  : 63  ];

assign data_in[ 32  : 39  ] = sub_data[ 32  : 39  ];
assign data_in[ 40  : 47  ] = sub_data[ 8   : 15  ];
assign data_in[ 48  : 55  ] = sub_data[ 112 : 119 ];
assign data_in[ 56  : 63  ] = sub_data[ 88  : 95  ];

assign data_in[ 64  : 71  ] = sub_data[ 64  : 71  ];
assign data_in[ 72  : 79  ] = sub_data[ 40  : 47  ];
assign data_in[ 80  : 87  ] = sub_data[ 16  : 23  ];
assign data_in[ 88  : 95  ] = sub_data[ 120 : 127 ];

assign data_in[ 96  : 103 ] = sub_data[ 96  : 103 ];
assign data_in[ 104 : 111 ] = sub_data[ 72  : 79  ];
assign data_in[ 112 : 119 ] = sub_data[ 48  : 55  ];
assign data_in[ 120 : 127 ] = sub_data[ 24  : 31  ];

endmodule
```

14.13. mix_col.v

```
`timescale 1ns / 1ps
```

```
module mix_col(
data_in,
mix_data
);
input  [0:127] data_in;
output  [0:127] mix_data;
GF_2_8_multiplier COL1(data_in[0:7],data_in[8:15],data_in[16:23],data_in[24:31],
                  mix_data[0:7],mix_data[8:15],mix_data[16:23],mix_data[24:31]);

GF_2_8_multiplier COL2(data_in[32:39],data_in[40:47],data_in[48:55],data_in[56:63],
                  mix_data[32:39],mix_data[40:47],mix_data[48:55],mix_data[56:63]);

GF_2_8_multiplier COL3(data_in[64:71],data_in[72:79],data_in[80:87],data_in[88:95],
                  mix_data[64:71],mix_data[72:79],mix_data[80:87],mix_data[88:95]);

GF_2_8_multiplier COL4(data_in[96:103],data_in[104:111],data_in[112:119],
                     data_in[120:127],mix_data[96:103],mix_data[104:111],
                                   mix_data[112:119],mix_data[120:127]);

endmodule
```

14.14. GF_2_8multiplier.v

```
`timescale 1ns / 1ps

module GF_2_8_multiplier(
data1,
data2,
data3,
data4,
multiplied_data1,
multiplied_data2,
multiplied_data3,
multiplied_data4
);

input [7:0] data1,data2,data3,data4;
output [7:0] multiplied_data1, multiplied_data2, multiplied_data3, multiplied_data4;
/*Multiplication Matrix for Mix Col
        |02  03  01  01|   |data1|
        |01  02  03  01|   |data2|
output= |01  01  02  03| * |data3|
        |03  01  01  02|   |data4|
*/
assign multiplied_data1=
multiply_02(data1)^multiply_03(data2)^multiply_01(data3)^multiply_01(data4);

assign multiplied_data2=
multiply_01(data1)^multiply_02(data2)^multiply_03(data3)^multiply_01(data4);

assign multiplied_data3=
multiply_01(data1)^multiply_01(data2)^multiply_02(data3)^multiply_03(data4);
```

```
assign multiplied_data4=
multiply_03(data1)^multiply_01(data2)^multiply_01(data3)^multiply_02(data4);


function [7:0] multiply_01(input [7:0]a);
    begin
    multiply_01=a;
    end
endfunction
function [7:0] multiply_02(input [7:0]a);
    begin
    if (a[7]==0) begin
        multiply_02=a<<1;
    end else begin
        multiply_02=(a<<1)^8'b00011011;
    end
    end
endfunction

function [7:0] multiply_03(input [7:0]a);
    reg [7:0]temp;
    begin
    temp=multiply_02(a);
    multiply_03=temp^a;
    end
endfunction

endmodule
```

15. Verilog : Decryption(CODE)

15.1. decryption_main.v

```
`timescale 1ns / 1ps

module decryption_main(
clk,
data,
rx_state,
key,
decrypted_data,
decrypted_data_state
    );
input clk;
input rx_state;
input  [0:127] key;
input  [0:127] data;
output [0:127] decrypted_data;
output decrypted_data_state;


wire  [0:127] key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10;
```

```
key keygen(key,key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10);

inverse_main_decrypt decode(clk,data,rx_state,decrypted_data,decrypted_data_state,
               key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10);

endmodule
```

15.2. inverse_main_decrypt.v

```
`timescale 1ns / 1ps

module inverse_main_decrypt(
                        clk,
                        data,
                        rx_state,
                        decrypted_data,
                        decrypted_data_state,
                        key_0,key_1,key_2,key_3,key_4,
                        key_5,key_6,key_7,key_8,key_9,
                        key_10
                        );
input           clk,rx_state;
input  [0:127] data;
output [0:127] decrypted_data;
output         decrypted_data_state;
input  [0:127] key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10;

wire   [0:127] key[0:10];
wire   [0:127] encoded_data_1,encoded_data_2,encoded_data_3,encoded_data_4;
wire   encoded_state_1, encoded_state_2, encoded_state_3, encoded_state_4;



decrypt_first  first(clk,data,key_10,key_9,
                          rx_state,encoded_data_1,encoded_state_1);
decrypt_mid    mid1(clk,encoded_data_1,key_8,key_7,
                     encoded_state_1,encoded_data_2,encoded_state_2);
decrypt_mid    mid2(clk,encoded_data_2,key_6,key_5,
                     encoded_state_2,encoded_data_3,encoded_state_3);
decrypt_mid    mid3(clk,encoded_data_3,key_4,key_3,
                     encoded_state_3,encoded_data_4,encoded_state_4);
decrypt_final  last(clk,encoded_data_4,key_2,key_1,key_0,
               encoded_state_4,decrypted_data,decrypted_data_state);


endmodule
```

15.3. decrypt_first.v

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps
module decrypt_first(
            clk,
            data,
            key1,key2,
            data_state,
            encoded_data,
            encoded_state
    );
input [127:0] data,key1,key2;
input         clk,data_state;
output [127:0]encoded_data;
output        encoded_state;


//change these to change baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg state=1'b0;//STATES
localparam init=1'b0,encode=1'b1;//states


reg  [127:0] temp_encoded_data=0;
reg temp_encoded_state=0;;
reg[24:0] clock_counter=25'b0;//counts how many clock cycles passed


wire [127:0] tempered_data1,tempered_data2;


inverse_main_first DUM1(data,tempered_data1,key1);
inverse_main_mid   DUM2(tempered_data1,tempered_data2,key2);


always@(posedge clk) begin
case (state)
init:
    begin
    if (clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
    end
    else
    begin
```

```
        clock_counter<=clock_counter+1;
    end

    if (data_state==1)
    begin
        clock_counter<=0;
        state<=encode;
    end
    end
encode:
    begin
    if (clock_counter==clock_per_bit*159)
    begin
       temp_encoded_state<=1'b1;
       temp_encoded_data<=tempered_data2;
       clock_counter<=0;
       state<=init;

    end
    else if(clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
        clock_counter<=clock_counter+1;
    end
    else
    begin
        clock_counter<=clock_counter+1;

    end
    end
endcase
end
assign encoded_state=temp_encoded_state;
assign encoded_data=temp_encoded_data;
endmodule
```

15.4. decrypt_mid.v

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps

module decrypt_mid(
            clk,
            data,
            key1,key2,
            data_state,
            encoded_data,
            encoded_state
    );
input [127:0] data,key1,key2;
input         clk,data_state;
output [127:0]encoded_data;
output        encoded_state;
```

```
//change these to change baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg state=1'b0;//STATES
localparam init=1'b0,encode=1'b1;//states


reg  [127:0] temp_encoded_data=0;
reg temp_encoded_state=0;;
reg[24:0] clock_counter=25'b0;//counts how many clock cycles passed



wire [127:0] tempered_data1,tempered_data2;

inverse_main_mid   DUM1(data,tempered_data1,key1);
inverse_main_mid   DUM2(tempered_data1,tempered_data2,key2);


always@(posedge clk) begin
case (state)
init:
    begin
    if (clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
    end
    else
    begin
        clock_counter<=clock_counter+1;
    end

    if (data_state==1)
    begin
        clock_counter<=0;
        state<=encode;
    end
    end
encode:
    begin
    if (clock_counter==clock_per_bit*159)
    begin
       temp_encoded_state<=1'b1;
       temp_encoded_data<=tempered_data2;
       clock_counter<=0;
       state<=init;
```

```
    end
    else if(clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
        clock_counter<=clock_counter+1;
    end
    else
    begin
        clock_counter<=clock_counter+1;

    end
    end
endcase
end
assign encoded_state=temp_encoded_state;
assign encoded_data=temp_encoded_data;
endmodule
```

15.5. decrypt_final.v

```
`timescale 1ns / 1ps
`timescale 1ns / 1ps

module decrypt_final(
            clk,
            data,
            key1,key2,key3,
            data_state,
            encoded_data,
            encoded_state
    );
input [127:0] data,key1,key2,key3;
input         clk,data_state;
output [127:0]encoded_data;
output        encoded_state;


//change these to change baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg state=1'b0;//STATES
localparam init=1'b0,encode=1'b1;//states

reg  [127:0] temp_encoded_data=0;
reg temp_encoded_state=0;;
reg[24:0] clock_counter=25'b0;//counts how many clock cycles passed
```

```
wire [127:0] tempered_data1,tempered_data2,tempered_data3;

inverse_main_mid   DUM1(data,tempered_data1,key1);
inverse_main_mid   DUM2(tempered_data1,tempered_data2,key2);
inverse_main_final DUM3(tempered_data2,tempered_data3,key3);


always@(posedge clk) begin
case (state)
init:
    begin
    if (clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
    end
    else
    begin
        clock_counter<=clock_counter+1;
    end

    if (data_state==1)
    begin
        clock_counter<=0;
        state<=encode;
    end
    end
encode:
    begin
    if (clock_counter==clock_per_bit*159)
    begin
       temp_encoded_state<=1'b1;
       temp_encoded_data<=tempered_data3;
       clock_counter<=0;
       state<=init;

    end
    else if(clock_counter==clock_per_bit<<4)
    begin
        temp_encoded_state<=1'b0;
        clock_counter<=clock_counter+1;
    end
    else
    begin
        clock_counter<=clock_counter+1;

    end
    end
endcase
end
assign encoded_state=temp_encoded_state;
assign encoded_data=temp_encoded_data;
endmodule
```

15.6. inverse_main_first.v

```
`timescale 1ns / 1ps
module
inverse_main_first(
data,
tempered_data,
key_10
    );
input  [0:127] data;
output [0:127] tempered_data;
input  [0:127] key_10;

wire   [0:127] data_with_key;
wire   [0:127] data_after_shift;



addRoundKey      RFAK (data,key_10,data_with_key);
inv_shift_rows   RFSR (data_with_key,data_after_shift);
inv_substitution RFSB (data_after_shift,tempered_data);
endmodule
```

15.7. inverse_main_mid.v

```
`timescale 1ns / 1ps
module
inverse_main_mid(
data,
tempered_data,
key
    );
input  [0:127] data;
output [0:127] tempered_data;
input  [0:127] key;

wire   [0:127] data_with_key;
wire   [0:127] data_after_mixcol;
wire   [0:127] data_after_shift;



addRoundKey      RiAk(data,key,data_with_key);
inverse_mixcol   RiMC(data_with_key,data_after_mixcol);
inv_shift_rows   RiSR(data_after_mixcol,data_after_shift);
inv_substitution RiSB(data_after_shift,tempered_data);

endmodule
```

15.8. *inverse_main_final.v*

```
`timescale 1ns / 1ps
module
inverse_main_final(
data,
tempered_data,
key_0
    );
input  [0:127] data;
output [0:127] tempered_data;
input  [0:127] key_0;

addRoundKey RFAK (data,key_0,tempered_data);
endmodule
```

15.9. *addRoundKey.v*

```
`timescale 1ns / 1ps

module addRoundKey(
    input [127:0] data,
    input [127:0] key,
    output [127:0] data_with_key
    );

assign data_with_key=data^key;
endmodule
```

15.10. *Inv_substitution.v*

```
module inv_substitution(
 mixcol_data,
inv_substituted_data
);
input [0:127] mixcol_data;
output [0:127] inv_substituted_data;

aes_invS_box invaes01( mixcol_data[000:007] , inv_substituted_data[000:007] );
aes_invS_box invaes02( mixcol_data[008:015] , inv_substituted_data[008:015] );
aes_invS_box invaes03( mixcol_data[016:023] , inv_substituted_data[016:023] );
aes_invS_box invaes04( mixcol_data[024:031] , inv_substituted_data[024:031] );
aes_invS_box invaes05( mixcol_data[032:039] , inv_substituted_data[032:039] );
aes_invS_box invaes06( mixcol_data[040:047] , inv_substituted_data[040:047] );
aes_invS_box invaes07( mixcol_data[048:055] , inv_substituted_data[048:055] );
aes_invS_box invaes08( mixcol_data[056:063] , inv_substituted_data[056:063] );
aes_invS_box invaes09( mixcol_data[064:071] , inv_substituted_data[064:071] );
aes_invS_box invaes10( mixcol_data[072:079] , inv_substituted_data[072:079] );
aes_invS_box invaes11( mixcol_data[080: 87] , inv_substituted_data[080:087] );
aes_invS_box invaes12( mixcol_data[088: 95] , inv_substituted_data[088:095] );
aes_invS_box invaes13( mixcol_data[096:103] , inv_substituted_data[096:103] );
```

```
aes_invS_box invaes14( mixcol_data[104:111] , inv_substituted_data[104:111] );
aes_invS_box invaes15( mixcol_data[112:119] , inv_substituted_data[112:119] );
aes_invS_box invaes16( mixcol_data[120:127] , inv_substituted_data[120:127] );
endmodule
```

15.11. aes_invS_box.v

```
module aes_invS_box(
mixcol_data,
inv_substituted_data
);

input [0:7] mixcol_data;
output reg [0:7] inv_substituted_data;
reg [0:7] c;
always @(mixcol_data)
begin
case(mixcol_data)
8'h63:c =8'h00;8'h7c:c =8'h01;8'h77:c =8'h02;8'h7b:c =8'h03;
8'hf2:c =8'h04;8'h6b:c =8'h05;8'h6f:c =8'h06;8'hc5:c =8'h07;
8'h30:c =8'h08;8'h01:c =8'h09;8'h67:c =8'h0a;8'h2b:c =8'h0b;
8'hfe:c =8'h0c;8'hd7:c =8'h0d;8'hab:c =8'h0e;8'h76:c =8'h0f; //0

8'hca:c =8'h10;8'h82:c =8'h11;8'hc9:c =8'h12;8'h7d:c =8'h13;
8'hfa:c =8'h14;8'h59:c =8'h15;8'h47:c =8'h16;8'hf0:c =8'h17;
8'had:c =8'h18;8'hd4:c =8'h19;8'ha2:c =8'h1a;8'haf:c =8'h1b;
8'h9c:c =8'h1c;8'ha4:c =8'h1d;8'h72:c =8'h1e;8'hc0:c =8'h1f; //1

8'hb7:c =8'h20;8'hfd:c =8'h21;8'h93:c =8'h22;8'h26:c =8'h23;
8'h36:c =8'h24;8'h3f:c =8'h25;8'hf7:c =8'h26;8'hcc:c =8'h27;
8'h34:c =8'h28;8'ha5:c =8'h29;8'he5:c =8'h2a;8'hf1:c =8'h2b;
8'h71:c =8'h2c;8'hd8:c =8'h2d;8'h31:c =8'h2e;8'h15:c =8'h2f; //2

8'h04:c =8'h30;8'hc7:c =8'h31;8'h23:c =8'h32;8'hc3:c =8'h33;
8'h18:c =8'h34;8'h96:c =8'h35;8'h05:c =8'h36;8'h9a:c =8'h37;
8'h07:c =8'h38;8'h12:c =8'h39;8'h80:c =8'h3a;8'he2:c =8'h3b;
8'heb:c =8'h3c;8'h27:c =8'h3d;8'hb2:c =8'h3e;8'h75:c =8'h3f; //3

8'h09:c =8'h40;8'h83:c =8'h41;8'h2c:c =8'h42;8'h1a:c =8'h43;
8'h1b:c =8'h44;8'h6e:c =8'h45;8'h5a:c =8'h46;8'ha0:c =8'h47;
8'h52:c =8'h48;8'h3b:c =8'h49;8'hd6:c =8'h4a;8'hb3:c =8'h4b;
8'h29:c =8'h4c;8'he3:c =8'h4d;8'h2f:c =8'h4e;8'h84:c =8'h4f; //4

8'h53:c =8'h50;8'hd1:c =8'h51;8'h00:c =8'h52;8'hed:c =8'h53;
8'h20:c =8'h54;8'hfc:c =8'h55;8'hb1:c =8'h56;8'h5b:c =8'h57;
8'h6a:c =8'h58;8'hcb:c =8'h59;8'hbe:c =8'h5a;8'h39:c =8'h5b;
8'h4a:c =8'h5c;8'h4c:c =8'h5d;8'h58:c =8'h5e;8'hcf:c =8'h5f; //5

8'hd0:c =8'h60;8'hef:c =8'h61;8'haa:c =8'h62;8'hfb:c =8'h63;
8'h43:c =8'h64;8'h4d:c =8'h65;8'h33:c =8'h66;8'h85:c =8'h67;
8'h45:c =8'h68;8'hf9:c =8'h69;8'h02:c =8'h6a;8'h7f:c =8'h6b;
8'h50:c =8'h6c;8'h3c:c =8'h6d;8'h9f:c =8'h6e;8'ha8:c =8'h6f; //6
```

```
8'h51:c =8'h70;8'ha3:c =8'h71;8'h40:c =8'h72;8'h8f:c =8'h73;
8'h92:c =8'h74;8'h9d:c =8'h75;8'h38:c =8'h76;8'hf5:c =8'h77;
8'hbc:c =8'h78;8'hb6:c =8'h79;8'hda:c =8'h7a;8'h21:c =8'h7b;
8'h10:c =8'h7c;8'hff:c =8'h7d;8'hf3:c =8'h7e;8'hd2:c =8'h7f; //7

8'hcd:c =8'h80;8'h0c:c =8'h81;8'h13:c =8'h82;8'hec:c =8'h83;
8'h5f:c =8'h84;8'h97:c =8'h85;8'h44:c =8'h86;8'h17:c =8'h87;
8'hc4:c =8'h88;8'ha7:c =8'h89;8'h7e:c =8'h8a;8'h3d:c =8'h8b;
8'h64:c =8'h8c;8'h5d:c =8'h8d;8'h19:c =8'h8e;8'h73:c =8'h8f; //8

8'h60:c =8'h90;8'h81:c =8'h91;8'h4f:c =8'h92;8'hdc:c =8'h93;
8'h22:c =8'h94;8'h2a:c =8'h95;8'h90:c =8'h96;8'h88:c =8'h97;
8'h46:c =8'h98;8'hee:c =8'h99;8'hb8:c =8'h9a;8'h14:c =8'h9b;
8'hde:c =8'h9c;8'h5e:c =8'h9d;8'h0b:c =8'h9e;8'hdb:c =8'h9f; //9

8'he0:c =8'ha0;8'h32:c =8'ha1;8'h3a:c =8'ha2;8'h0a:c =8'ha3;
8'h49:c =8'ha4;8'h06:c =8'ha5;8'h24:c =8'ha6;8'h5c:c =8'ha7;
8'hc2:c =8'ha8;8'hd3:c =8'ha9;8'hac:c =8'haa;8'h62:c =8'hab;
8'h91:c =8'hac;8'h95:c =8'had;8'he4:c =8'hae;8'h79:c =8'haf; //a

8'he7:c =8'hb0;8'hc8:c =8'hb1;8'h37:c =8'hb2;8'h6d:c =8'hb3;
8'h8d:c =8'hb4;8'hd5:c =8'hb5;8'h4e:c =8'hb6;8'ha9:c =8'hb7;
8'h6c:c =8'hb8;8'h56:c =8'hb9;8'hf4:c =8'hba;8'hea:c =8'hbb;
8'h65:c =8'hbc;8'h7a:c =8'hbd;8'hae:c =8'hbe;8'h08:c =8'hbf; //b

8'hba:c =8'hc0;8'h78:c =8'hc1;8'h25:c =8'hc2;8'h2e:c =8'hc3;
8'h1c:c =8'hc4;8'ha6:c =8'hc5;8'hb4:c =8'hc6;8'hc6:c =8'hc7;
8'he8:c =8'hc8;8'hdd:c =8'hc9;8'h74:c =8'hca;8'h1f:c =8'hcb;
8'h4b:c =8'hcc;8'hbd:c =8'hcd;8'h8b:c =8'hce;8'h8a:c =8'hcf; //c

8'h70:c =8'hd0;8'h3e:c =8'hd1;8'hb5:c =8'hd2;8'h66:c =8'hd3;
8'h48:c =8'hd4;8'h03:c =8'hd5;8'hf6:c =8'hd6;8'h0e:c =8'hd7;
8'h61:c =8'hd8;8'h35:c =8'hd9;8'h57:c =8'hda;8'hb9:c =8'hdb;
8'h86:c =8'hdc;8'hc1:c =8'hdd;8'h1d:c =8'hde;8'h9e:c =8'hdf; //d

8'he1:c =8'he0;8'hf8:c =8'he1;8'h98:c =8'he2;8'h11:c =8'he3;
8'h69:c =8'he4;8'hd9:c =8'he5;8'h8e:c =8'he6;8'h94:c =8'he7;
8'h9b:c =8'he8;8'h1e:c =8'he9;8'h87:c =8'hea;8'he9:c =8'heb;
8'hce:c =8'hec;8'h55:c =8'hed;8'h28:c =8'hee;8'hdf:c =8'hef; //e

8'h8c:c =8'hf0;8'ha1:c =8'hf1;8'h89:c =8'hf2;8'h0d:c =8'hf3;
8'hbf:c =8'hf4;8'he6:c =8'hf5;8'h42:c =8'hf6;8'h68:c =8'hf7;
8'h41:c =8'hf8;8'h99:c =8'hf9;8'h2d:c =8'hfa;8'h0f:c =8'hfb;
8'hb0:c =8'hfc;8'h54:c =8'hfd;8'hbb:c =8'hfe;8'h16:c =8'hff; //f
endcase
end
assign inv_substituted_data=c;
endmodule
```

15.12. inv_shift_rows.v

```
module inv_shift_rows(
inv_sub_data,
```

```
data_in
);
input  [0:127] inv_sub_data;
output  [0:127] data_in;

assign data_in[ 0   : 7   ] = inv_sub_data[ 0   : 7   ];
assign data_in[ 8   : 15  ] = inv_sub_data[ 40  : 47 ];
assign data_in[ 16  : 23  ] = inv_sub_data[ 80  : 87  ];
assign data_in[ 24  : 31  ] = inv_sub_data[ 120 : 127  ];

assign data_in[ 32  : 39  ] = inv_sub_data[ 32  : 39  ];
assign data_in[ 40  : 47  ] = inv_sub_data[ 72   : 79  ];
assign data_in[ 48  : 55  ] = inv_sub_data[ 112 : 119 ];
assign data_in[ 56  : 63  ] = inv_sub_data[ 24  : 31  ];

assign data_in[ 64  : 71  ] = inv_sub_data[ 64  : 71  ];
assign data_in[ 72  : 79  ] = inv_sub_data[ 104 : 111  ];
assign data_in[ 80  : 87  ] = inv_sub_data[ 16  : 23  ];
assign data_in[ 88  : 95  ] = inv_sub_data[ 56 : 63 ];

assign data_in[ 96  : 103 ] = inv_sub_data[ 96  : 103 ];
assign data_in[ 104 : 111 ] = inv_sub_data[ 8   : 15  ];
assign data_in[ 112 : 119 ] = inv_sub_data[ 48  : 55  ];
assign data_in[ 120 : 127 ] = inv_sub_data[ 88  : 95  ];

endmodule
```

15.13. inverse_mix_col.v

```
`timescale 1ns / 1ps


module inverse_mixcol(
data_in,
inverse_mixdata
);
input  [0:127] data_in;
output  [0:127] inverse_mixdata;
inv_GF_2_8_multiplier COL1(data_in[0:7],data_in[8:15],data_in[16:23],
data_in[24:31],
inverse_mixdata[0:7],inverse_mixdata[8:15],inverse_mixdata[16:23],
inverse_mixdata[24:31]);

inv_GF_2_8_multiplier COL2(data_in[32:39],data_in[40:47],data_in[48:55],
data_in[56:63],
inverse_mixdata[32:39],inverse_mixdata[40:47],inverse_mixdata[48:55],
inverse_mixdata[56:63]);

inv_GF_2_8_multiplier COL3(data_in[64:71],data_in[72:79],data_in[80:87],
data_in[88:95],
inverse_mixdata[64:71],inverse_mixdata[72:79],inverse_mixdata[80:87],
inverse_mixdata[88:95]);
```

```
inv_GF_2_8_multiplier COL4(data_in[96:103],data_in[104:111],data_in[112:119],
data_in[120:127],
inverse_mixdata[96:103],inverse_mixdata[104:111],inverse_mixdata[112:119],
inverse_mixdata[120:127]);

endmodule
```

15.14. inv_GF_2_8 multiplier.v

```
`timescale 1ns / 1ps

module inv_GF_2_8_multiplier(
data1,
data2,
data3,
data4,
multiplied_data1,
multiplied_data2,
multiplied_data3,
multiplied_data4,
                        );

input [7:0] data1,data2,data3,data4;
output [7:0] multiplied_data1, multiplied_data2, multiplied_data3, multiplied_data4;
/*Multiplication Matrix for Mix Col
        |0E  0B  0D  09|   |data1|
        |09  0E  0B  0D|   |data2|
output= |0D  09  0E  0B| * |data3|
        |0B  0D  09  0E|   |data4|
*/
assign multiplied_data1=
multiply_0E(data1)^multiply_0B(data2)^multiply_0D(data3)^multiply_09(data4);
assign multiplied_data2=
multiply_09(data1)^multiply_0E(data2)^multiply_0B(data3)^multiply_0D(data4);
assign multiplied_data3=
multiply_0D(data1)^multiply_09(data2)^multiply_0E(data3)^multiply_0B(data4);
assign multiplied_data4=
multiply_0B(data1)^multiply_0D(data2)^multiply_09(data3)^multiply_0E(data4);

function [7:0] multiply_09(input [7:0]a);
    begin
    multiply_09=(multiply_02(multiply_02(multiply_02(a)))^ a);
    end
endfunction
function [7:0] multiply_0B(input [7:0]a);
    begin
    multiply_0B=(multiply_02(multiply_02(multiply_02(a))^a)^a);
    end
endfunction
function [7:0] multiply_0D(input [7:0]a);
    begin
    multiply_0D=(multiply_02(multiply_02((multiply_02(a)^a)))^a);
    end
```

```
endfunction
function [7:0] multiply_0E(input [7:0]a);
    begin
    multiply_0E=multiply_02(multiply_02(multiply_02(a)^a)^a);
    end
endfunction

function [7:0] multiply_01(input [7:0]a);
    begin
    multiply_01=a;
    end
endfunction
function [7:0] multiply_02(input [7:0]a);
    begin
    if (a[7]==0) begin
        multiply_02=a<<1;
    end else begin
        multiply_02=(a<<1)^8'b00011011;
    end
    end
endfunction

function [7:0] multiply_03(input [7:0]a);
    reg [7:0]temp;
    begin
    temp=multiply_02(a);
    multiply_03=temp^a;
    end
endfunction

endmodule
```

16. Verilog : Key Expansion(CODE)

16.1. key.v

```
`timescale 1ns / 1ps



module key( key_in,key_0,key_1,key_2,key_3,key_4,key_5,key_6,key_7,key_8,key_9,key_10);

input [127:0] key_in;
output [127:0] key_0;
output [127:0] key_1;
output [127:0] key_2;
output [127:0] key_3;
output [127:0] key_4;
output [127:0] key_5;
output [127:0] key_6;
output [127:0] key_7;
output [127:0] key_8;
```

```
output [127:0] key_9;
output [127:0] key_10;
wire [127:0] key_bus [0:10];
wire [39:0] select_i;
assign key_bus[0] = key_in;
assign select_i = 40'h9876543210;

genvar i;
generate
for( i=0; i<10; i= i+1) begin
aes_key_expand_128 k0(
.select_i(select_i[4*(i+1)-1 : 4*i]),.key(key_bus[i]), .key_out(key_bus[i+1]));

end
endgenerate
assign key_0 = key_bus[0];
assign key_1 = key_bus[1];
assign key_2 = key_bus[2];
assign key_3 = key_bus[3];
assign key_4 = key_bus[4];
assign key_5 = key_bus[5];
assign key_6 = key_bus[6];
assign key_7 = key_bus[7];
assign key_8 = key_bus[8];
assign key_9 = key_bus[9];
assign key_10 = key_bus[10];




endmodule
```

16.2. key_expand.v

```
`timescale 1ns / 1ps



module aes_key_expand_128( select_i, key, key_out);

input [3:0] select_i;
input       [127:0]       key;
output [127:0] key_out;


wire        [31:0]        w[3:0];
wire        [31:0]        tmp_w;
wire        [31:0]        subword;
wire        [31:0]        rcon;
wire [31:0] a,b,c,d;
```

```
 assign w[0] =  key[127:096];

 assign w[1] =  key[095:064];

 assign w[2] =  key[063:032];

 assign w[3] = key[031:000] ;



assign a =   w[0]^subword^rcon;

assign b =  w[0]^w[1]^subword^rcon;

assign c =   w[0]^w[2]^w[1]^subword^rcon;

assign d =  w[0]^w[3]^w[2]^w[1]^subword^rcon;
assign tmp_w = w[3];
aes_sbox u0(        .a(tmp_w[23:16]), .d(subword[31:24]));
aes_sbox u1(        .a(tmp_w[15:08]), .d(subword[23:16]));
aes_sbox u2(        .a(tmp_w[07:00]), .d(subword[15:08]));
aes_sbox u3(        .a(tmp_w[31:24]), .d(subword[07:00]));

aes_rcon r0(  .select_i(select_i),  .out(rcon));

assign key_out ={a,b,c,d};



endmodule
```

16.3. aes_sbox.v

```
`timescale 1ns / 1ps

module aes_sbox(a,d);
input        [7:0]        a;
output        [7:0]        d;
reg        [7:0]        d;

always @(a)
        case(a)
           8'h00: d=8'h63;8'h01: d=8'h7c;8'h02: d=8'h77;8'h03: d=8'h7b;8'h04: d=8'hf2;
           8'h05: d=8'h6b;8'h06: d=8'h6f;8'h07: d=8'hc5;8'h08: d=8'h30;8'h09: d=8'h01;
           8'h0a: d=8'h67;8'h0b: d=8'h2b;8'h0c: d=8'hfe;8'h0d: d=8'hd7;8'h0e: d=8'hab;
           8'h0f: d=8'h76;8'h10: d=8'hca;8'h11: d=8'h82;8'h12: d=8'hc9;8'h13: d=8'h7d;
           8'h14: d=8'hfa;8'h15: d=8'h59;8'h16: d=8'h47;8'h17: d=8'hf0;8'h18: d=8'had;
           8'h19: d=8'hd4;8'h1a: d=8'ha2;8'h1b: d=8'haf;8'h1c: d=8'h9c;8'h1d: d=8'ha4;
           8'h1e: d=8'h72;8'h1f: d=8'hc0;8'h20: d=8'hb7;8'h21: d=8'hfd;8'h22: d=8'h93;
           8'h23: d=8'h26;8'h24: d=8'h36;8'h25: d=8'h3f;8'h26: d=8'hf7;8'h27: d=8'hcc;
           8'h28: d=8'h34;8'h29: d=8'ha5;8'h2a: d=8'he5;8'h2b: d=8'hf1;8'h2c: d=8'h71;
```

```
            8'h2d: d=8'hd8;8'h2e: d=8'h31;8'h2f: d=8'h15;8'h30: d=8'h04;8'h31: d=8'hc7;
            8'h32: d=8'h23;8'h33: d=8'hc3;8'h34: d=8'h18;8'h35: d=8'h96;8'h36: d=8'h05;
            8'h37: d=8'h9a;8'h38: d=8'h07;8'h39: d=8'h12;8'h3a: d=8'h80;8'h3b: d=8'he2;
            8'h3c: d=8'heb;8'h3d: d=8'h27;8'h3e: d=8'hb2;8'h3f: d=8'h75;8'h40: d=8'h09;
            8'h41: d=8'h83;8'h42: d=8'h2c;8'h43: d=8'h1a;8'h44: d=8'h1b;8'h45: d=8'h6e;
            8'h46: d=8'h5a;8'h47: d=8'ha0;8'h48: d=8'h52;8'h49: d=8'h3b;8'h4a: d=8'hd6;
            8'h4b: d=8'hb3;8'h4c: d=8'h29;8'h4d: d=8'he3;8'h4e: d=8'h2f;8'h4f: d=8'h84;
            8'h50: d=8'h53;8'h51: d=8'hd1;8'h52: d=8'h00;8'h53: d=8'hed;8'h54: d=8'h20;
            8'h55: d=8'hfc;8'h56: d=8'hb1;8'h57: d=8'h5b;8'h58: d=8'h6a;8'h59: d=8'hcb;
            8'h5a: d=8'hbe;8'h5b: d=8'h39;8'h5c: d=8'h4a;8'h5d: d=8'h4c;8'h5e: d=8'h58;
            8'h5f: d=8'hcf;8'h60: d=8'hd0;8'h61: d=8'hef;8'h62: d=8'haa;8'h63: d=8'hfb;
            8'h64: d=8'h43;8'h65: d=8'h4d;8'h66: d=8'h33;8'h67: d=8'h85;8'h68: d=8'h45;
            8'h69: d=8'hf9;8'h6a: d=8'h02;8'h6b: d=8'h7f;8'h6c: d=8'h50;8'h6d: d=8'h3c;
            8'h6e: d=8'h9f;8'h6f: d=8'ha8;8'h70: d=8'h51;8'h71: d=8'ha3;8'h72: d=8'h40;
            8'h73: d=8'h8f;8'h74: d=8'h92;8'h75: d=8'h9d;8'h76: d=8'h38;8'h77: d=8'hf5;
            8'h78: d=8'hbc;8'h79: d=8'hb6;8'h7a: d=8'hda;8'h7b: d=8'h21;8'h7c: d=8'h10;
            8'h7d: d=8'hff;8'h7e: d=8'hf3;8'h7f: d=8'hd2;8'h80: d=8'hcd;8'h81: d=8'h0c;
            8'h82: d=8'h13;8'h83: d=8'hec;8'h84: d=8'h5f;8'h85: d=8'h97;8'h86: d=8'h44;
            8'h87: d=8'h17;8'h88: d=8'hc4;8'h89: d=8'ha7;8'h8a: d=8'h7e;8'h8b: d=8'h3d;
            8'h8c: d=8'h64;8'h8d: d=8'h5d;8'h8e: d=8'h19;8'h8f: d=8'h73;8'h90: d=8'h60;
            8'h91: d=8'h81;8'h92: d=8'h4f;8'h93: d=8'hdc;8'h94: d=8'h22;8'h95: d=8'h2a;
            8'h96: d=8'h90;8'h97: d=8'h88;8'h98: d=8'h46;8'h99: d=8'hee;8'h9a: d=8'hb8;
            8'h9b: d=8'h14;8'h9c: d=8'hde;8'h9d: d=8'h5e;8'h9e: d=8'h0b;8'h9f: d=8'hdb;
            8'ha0: d=8'he0;8'ha1: d=8'h32;8'ha2: d=8'h3a;8'ha3: d=8'h0a;8'ha4: d=8'h49;
            8'ha5: d=8'h06;8'ha6: d=8'h24;8'ha7: d=8'h5c;8'ha8: d=8'hc2;8'ha9: d=8'hd3;
            8'haa: d=8'hac;8'hab: d=8'h62;8'hac: d=8'h91;8'had: d=8'h95;8'hae: d=8'he4;
            8'haf: d=8'h79;8'hb0: d=8'he7;8'hb1: d=8'hc8;8'hb2: d=8'h37;8'hb3: d=8'h6d;
            8'hb4: d=8'h8d;8'hb5: d=8'hd5;8'hb6: d=8'h4e;8'hb7: d=8'ha9;8'hb8: d=8'h6c;
            8'hb9: d=8'h56;8'hba: d=8'hf4;8'hbb: d=8'hea;8'hbc: d=8'h65;8'hbd: d=8'h7a;
            8'hbe: d=8'hae;8'hbf: d=8'h08;8'hc0: d=8'hba;8'hc1: d=8'h78;8'hc2: d=8'h25;
            8'hc3: d=8'h2e;8'hc4: d=8'h1c;8'hc5: d=8'ha6;8'hc6: d=8'hb4;8'hc7: d=8'hc6;
            8'hc8: d=8'he8;8'hc9: d=8'hdd;8'hca: d=8'h74;8'hcb: d=8'h1f;8'hcc: d=8'h4b;
            8'hcd: d=8'hbd;8'hce: d=8'h8b;8'hcf: d=8'h8a;8'hd0: d=8'h70;8'hd1: d=8'h3e;
            8'hd2: d=8'hb5;8'hd3: d=8'h66;8'hd4: d=8'h48;8'hd5: d=8'h03;8'hd6: d=8'hf6;
            8'hd7: d=8'h0e;8'hd8: d=8'h61;8'hd9: d=8'h35;8'hda: d=8'h57;8'hdb: d=8'hb9;
            8'hdc: d=8'h86;8'hdd: d=8'hc1;8'hde: d=8'h1d;8'hdf: d=8'h9e;8'he0: d=8'he1;
            8'he1: d=8'hf8;8'he2: d=8'h98;8'he3: d=8'h11;8'he4: d=8'h69;8'he5: d=8'hd9;
            8'he6: d=8'h8e;8'he7: d=8'h94;8'he8: d=8'h9b;8'he9: d=8'h1e;8'hea: d=8'h87;
            8'heb: d=8'he9;8'hec: d=8'hce;8'hed: d=8'h55;8'hee: d=8'h28;8'hef: d=8'hdf;
            8'hf0: d=8'h8c;8'hf1: d=8'ha1;8'hf2: d=8'h89;8'hf3: d=8'h0d;8'hf4: d=8'hbf;
            8'hf5: d=8'he6;8'hf6: d=8'h42;8'hf7: d=8'h68;8'hf8: d=8'h41;8'hf9: d=8'h99;
            8'hfa: d=8'h2d;8'hfb: d=8'h0f;8'hfc: d=8'hb0;8'hfd: d=8'h54;8'hfe: d=8'hbb;
            8'hff: d=8'h16;
        endcase

endmodule
```

17. Verilog : UART(CODE)

17.1. rx.v

```
`timescale 1ns / 1ps

module rx(
    clk,//clock of fpga
    data_in,// the reciver pin in fpga
    data_state,
    data_out // the data obtained
    );
input clk,data_in;
output [127:0] data_out;
output  data_state;

//change these to change the baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit


//states
reg[1:0] state=2'b0;//STATES
localparam init=2'd0,start=2'd1,readdata=2'd2,stop=2'd3;//states

//registers
reg[24:0] clock_counter=24'b0;//counts how many clock cycles passed
reg[24:0] temp_clock_counter=24'b0;//counts how many clock cycles passed
reg[7:0] output_array=8'b0;//stores the recieved bits

reg[6:0] no_data_recieved=7'b0,temp_no_data_recieved=3'b0;
//counts how many data bits recieved

reg[127:0] temp_temp_data_out=128'b0,temp_data_out=128'b0,data_out128=128'b1;
//holds 128 bit data


reg temp_data_state=0;
reg [18:0] data_state_counter=0;

always@(posedge clk) begin
case (state)
//stays in idle state
init:
    begin
    if (clock_counter==clock_per_bit<<3)
    begin
    temp_data_state<=1'b0;
    clock_counter<=0;
    end
```

```
    else
    begin

        clock_counter<=clock_counter+1;
    end

    if(data_in==0)//check for the first 0
    begin
        clock_counter<=0;
        state<=start;//goto next state
    end
    end
//starts when first 0 is recieved
start:
    begin

    if(clock_counter==(clock_per_bit>>1))
    // check if half the clock cycles assignes has passed
    begin
        if(data_in==0)//if the input is still 0 or not
        begin
            clock_counter<=0;//reset clock
            state<=readdata;//go to next state
        end
        else
        begin
            state<=init;//else go to the first state
        end
    end
    else
    begin
        clock_counter<=clock_counter+1;//count to wait the required clock cycles

    end
    end
//starts to read data until 8 datas  points are obtained
readdata:
    begin
    if(clock_counter==clock_per_bit)
        begin
            clock_counter<=0;
            output_array<={data_in,output_array[7:1]};// assign the input data
            //uart offers mirror image of data,
            //the trick above helps us to recreate the original data
            no_data_recieved<=temp_no_data_recieved;//counts the total data recieved
            if (&no_data_recieved[2:0])//check if data recieved is seven or not
            begin
                clock_counter<=0;//clears clock
                temp_temp_data_out<=temp_data_out;
                //calculates once 8 bit data is received

                state<=stop;//proceeds to next state
            end
        end
```

```
    else
        begin
        clock_counter<=clock_counter+1;//counts till required clock time
        end
    end
//once 8 bit data are obtained rests until the last enter data settles
stop:
    begin
    if(~(|no_data_recieved[6:0]))//7 times 8 bit data is input
        begin
        temp_data_state<=1'b1;
        end
    else
        begin
        temp_data_state<=1'b0;
        end

    if(clock_counter==clock_per_bit)//the 10 is used as temporary wait time to proceed to next input
    begin
         clock_counter<=0;//resets clock

        state<=init;//goes to sleep state and awakes if the next input is 0
    end
    else
    begin
    clock_counter<=clock_counter+1;//counts clock cycles
    end

    end
//default
default:
    state<=init;
endcase
end
//happens every clock cycles
always@(posedge clk)
    begin
    temp_no_data_recieved<=no_data_recieved+1;//increments the data collected


    if (state==stop)
        begin
        temp_data_out<={temp_temp_data_out[119:0],output_array};
        //at stop state assignes the collected seven bit data
        //else continues to carry its previoud value

        if(~(|no_data_recieved[6:0]))//7 times 8 bit data is input
            begin
            data_out128<=temp_data_out;//output data

            end
        end

    if (state==init)
```

```
    begin
        if(temp_clock_counter==clock_per_bit*160)
        begin
            data_out128<=128'b1;
            temp_clock_counter<=0;
        end
        else
        begin
            temp_clock_counter<=temp_clock_counter+1;
        end
    end
    else
        begin
            temp_clock_counter<=0;
        end

end
assign data_state=temp_data_state;
assign data_out=data_out128;

endmodule
```

17.2. tx.v

```
`timescale 1ns / 1ps



module tx(
    data_state,
    clk,
    data,
    reset,
    out
    );
input data_state;
input clk;
input reset;
input [127:0] data;
output out;

reg [2:0] state=2'd0;
reg [10:0] data_send;
localparam init=2'd0,setup=2'd1,writedata=2'd2;//states

parameter rest=1;

//change these to change the baud rate
parameter clock_speed=100000000;//speed of the FPGA clock
parameter baud_rate=9600;//the number of bits you need per second

parameter clock_per_bit=10417;
// number of clock cycles your hardware can afford per bit
```

```
reg[16:0] clock_counter=13'b0;//counts how many clock cycles passed

reg temp_out=rest;
reg [3:0] bit_counter=0,temp_bit_counter=0;;
reg [3:0] byte_counter;
always@(posedge clk)
begin
case(state)
init:
begin
    temp_out<=1'b1;
    byte_counter<=0;
    if (data_state==1)
    begin
        state<=setup;
    end
end

setup:
begin
    if (reset==1)
    begin
        state<=init;
    end
    else
    begin
      case(byte_counter)
       0:data_send = {1'b1,1'b1,data[127:120],1'b0};
       1:data_send = {1'b1,1'b1,data[119:112],1'b0};
       2:data_send = {1'b1,1'b1,data[111:104],1'b0};
       3:data_send = {1'b1,1'b1,data[103: 96],1'b0};
       4:data_send = {1'b1,1'b1,data[ 95: 88],1'b0};
       5:data_send = {1'b1,1'b1,data[ 87: 80],1'b0};
       6:data_send = {1'b1,1'b1,data[ 79: 72],1'b0};
       7:data_send = {1'b1,1'b1,data[ 71: 64],1'b0};
       8:data_send = {1'b1,1'b1,data[ 63: 56],1'b0};
       9:data_send = {1'b1,1'b1,data[ 55: 48],1'b0};
      10:data_send = {1'b1,1'b1,data[ 47: 40],1'b0};
      11:data_send = {1'b1,1'b1,data[ 39: 32],1'b0};
      12:data_send = {1'b1,1'b1,data[ 31: 24],1'b0};
      13:data_send = {1'b1,1'b1,data[ 23: 16],1'b0};
      14:data_send = {1'b1,1'b1,data[ 15:  8],1'b0};
      15:data_send = {1'b1,1'b1,data[  7:  0],1'b0};
      endcase

        state<=writedata;
        clock_counter<=clock_per_bit;
        bit_counter<=0;
    end
end

writedata:
begin
```

```
if (clock_counter==clock_per_bit)
    begin
        temp_out<=data_send[bit_counter];
        clock_counter<=0;
        if (bit_counter==10)
        begin
        if (byte_counter==15)
            begin
                state<=init;

            end
        else
            begin
                state<=setup;
                byte_counter=byte_counter+1;
            end
        end
        else
        begin
            bit_counter<=bit_counter+1;
        end
     end
else
    begin
        clock_counter<=clock_counter+1;
    end
end
default:
begin
    state<=init;
end
endcase
end
assign out=temp_out;
endmodule
```

18. Device Compilation Code

18.1. main.v for FPGA with encryption

```
`timescale 1ns / 1ps

module main(
    clk,
    data_in,
    data_out_tx
    );
input clk,data_in;
output data_out_tx;

wire data_state,encoded_data_state;
wire [127:0] data_out_rx;//data_out_rx_test;
```

```
wire [127:0] encrypted_data;
reg  [127:0] key=128'h5468617473206d79204b756e67204675;


rx         DUT1(clk,data_in,data_state,data_out_rx);
encryption_main DUM(clk,data_out_rx,data_state,key,encrypted_data,encoded_data_state);
tx         DUT3( encoded_data_state, clk, encrypted_data,1'b0, data_out_tx);
endmodule
```

18.2. main.v for FPGA with decryption

```
`timescale 1ns / 1ps

module main(
    clk,
    data_in,
    data_out_tx
    );
input clk,data_in;
output data_out_tx;

wire data_state,encoded_data_state;
wire [127:0] data_out_rx;//data_out_rx_test;
wire [127:0] decrypted_data;
reg  [127:0] key=128'h5468617473206d79204b756e67204675;


rx         DUT1(clk,data_in,data_state,data_out_rx);
decryption_main DUM(clk,data_out_rx,data_state,key,decrypted_data,encoded_data_state);
tx         DUT3( encoded_data_state, clk, decrypted_data,1'b0, data_out_tx);

endmodule
```

References

[1] "A review on serial communication by UART". http://ijarcsse.com/Before_August_2017/docs/papers/Volume_3/1_January2013/V3I1-0220.pdf Date Accessed = "2021-04-20".

[2] "Design of a 9-bit UART module based on Verilog HDL". 2012 10th IEEE International Conference on Semiconductor Electronics (ICSE), 2012, pp. 570-573, doi: 10.1109/SMElec.2012.6417210. Date Accessed = "2020-11-20".

[3] "The Rijndael Block Cipher". "https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf"DateAccessed="2020-20-20".

[4] Ako Abdullah. Advanced encryption standard (aes) algorithm to encrypt and decrypt data. 06 2017.

[5] "Hakhamaneshi, Bahram and Arad, Behnam". "A Hardware Implementation of the Advanced Encryption Standard (AES) Algorithm using SystemVerilog", "01" "2010".