128-bit AES (Advanced Encryption Standard) (Decryption)

Aditya Gupta (20116003)

Akash Jha (20116005)

Apoorva Verma (20115018)

Apurba Prasad Padhy (20310005)

Tushar Sahu (20116102)

Outline

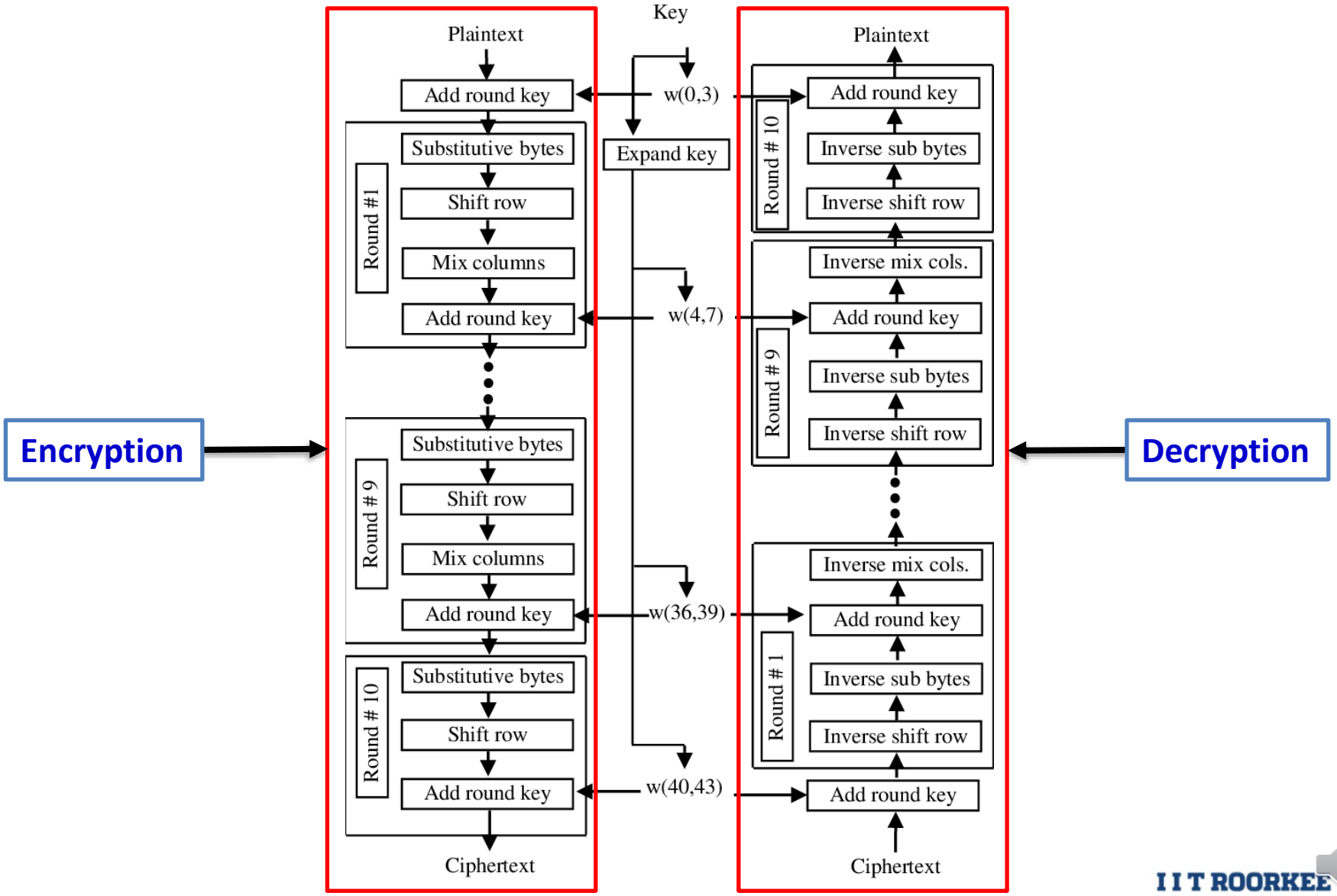
- Problem Statement
- Block Diagram
- Flow Chart
- Algorithms/Steps
- RTL Verilog Code
- Schematic of Sub-blocks
- Simulation Result 1
- Simulation Result 2
- Simulation Result 3
- Summary

Problem Statement

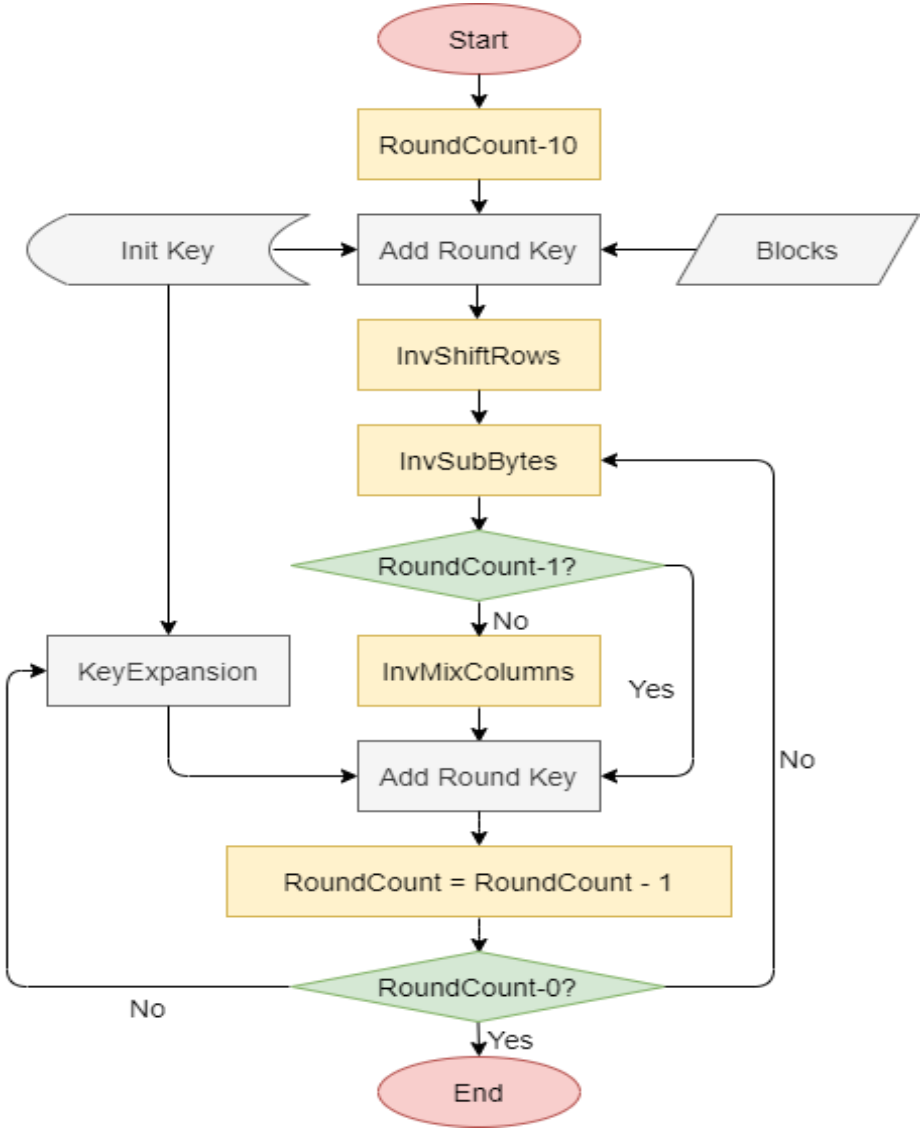
- Maintaining confidentiality and ensuring security of sensitive data is an open problem in today's age of information technology.
- It is also necessary to make sure that the transmitted data is not modified, intentionally or unintentionally, and that the authenticity of the sender is verified correctly.
- In this context, various cryptographic specifications have been developed to protect data from being accessed by anyone other than the sender and the intended receiver.
- The AES (Advanced Encryption Standard) is a block-cipher algorithm known for its considerable improvements over its predecessors especially with respect to portability and security.

In this project, we aim to implement 128-bit AES algorithm on an FPGA board using Verilog hardware coding (only decryption).

Block Diagram

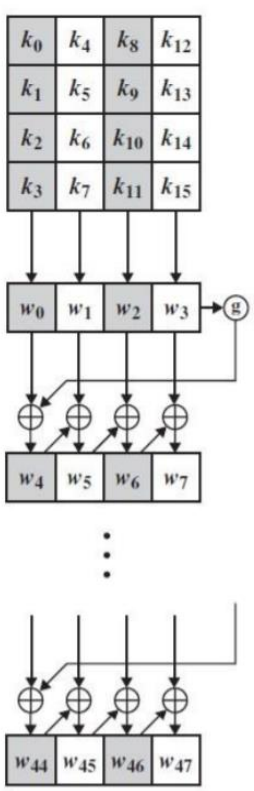


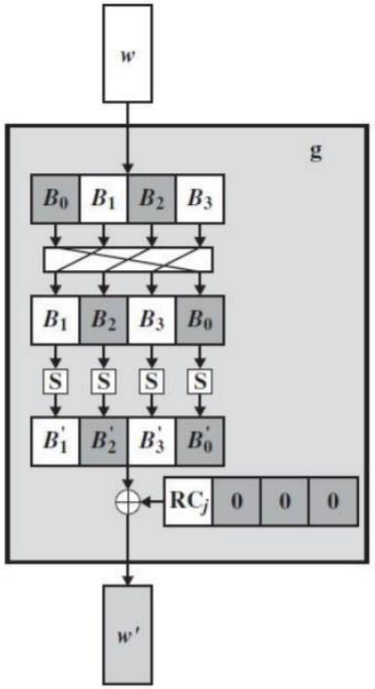
Flow Chart for Decryption



Step 1: Key Expansion

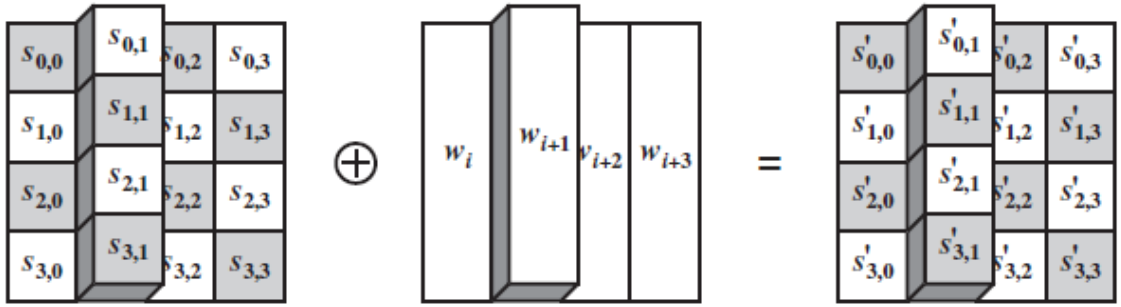
- Before beginning the decryption process, the key is passed through a key expansion function, which produces 11 round keys, which are used as one of the inputs at every ‘Add Round Key’ stage.
- In the key expansion process, the previous key is used to obtain next key, to get the next key from the previous one, the last column of the previous key is passed as argument to some function, let us call this function 'g'.
- The function g involves following steps:
  - This function accepts 1 word (4 bytes) perform left shift operation and a substitute byte transformation.
  - The matrix is then bitwise XORed with the round constant.
  - The matrix thus obtained is the output of the function.
- The output of the function is then XORed with the first column (first word) of the previous key to get the first column (first word) of the next key. The obtained word is then XORed with the second column of the previous key to get the second column, and so on. Thus, we get the next key.





Step 2: Pre-Round Transformation

- In this round, we bitwise XOR the original data (128 bits) i.e., 4 words with the last key of 128 bits generated during the key expansion process.
- The output of this round is input for round 1.



Step 3: Inverse Shift Row

- The inverse shift row transformation performs circular shifts in such a way that the i^{th} row undergoes (i-1) 1-byte right shifts.

- The rationale behind the Shift Rows operation was to spread out 4 bytes of one column to different columns and introduce a degree of randomness in the cipher.

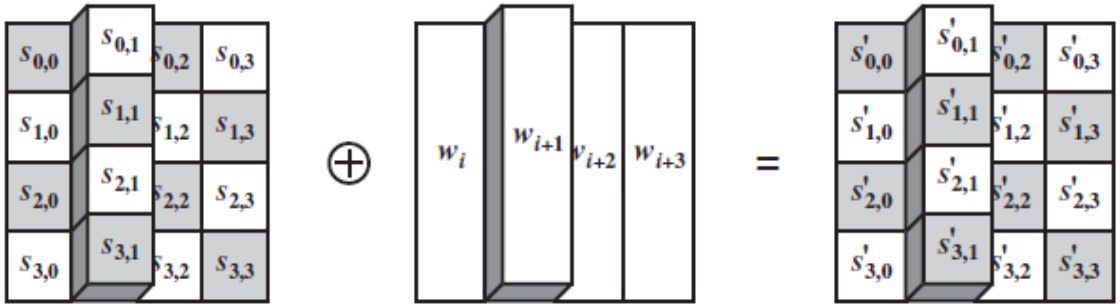
Step 4: Inverse Sub Bytes

- This transformation uses the Inverse S-Box to find one-to-one replacement bytes for all possible bytes in the input state array. The table is derived from the inverse function over GF(2^8).

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

Step 5: Add Round Key

- In the AddRoundKey transformation, the key is bitwise XORed with the state matrix.
- In AES decryption process, the last key (of the expanded keys) is used in the preround transformation. The second last key is used in the first round and so on.
- Finally, the initial key is used in the last round. This is done to exactly reverse the encryption process.



Step 6: Inverse Mix Columns

- The state matrix is represented as column polynomials over GF(2^8) and the transformation consists of matrix multiplication of the state with a polynomial over a finite field.
- This works after the AddRoundKey stage to ensure that all parts of the block affect each other. For a data block of 128 bits, the state matrix has 4 rows. Therefore, the columns of the state matrix are each viewed as the polynomial of degree 8 over GF(2^8) and multiplied modulo (x^4 +1).

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

$$\begin{aligned} y_0 &= 0e * x_0 \oplus 0b * x_1 \oplus 0d * x_2 \oplus 09 * x_3 \\ y_1 &= 09 * x_0 \oplus 0e * x_1 \oplus 0b * x_2 \oplus 0d * x_3 \\ y_2 &= 0d * x_0 \oplus 09 * x_1 \oplus 0e * x_2 \oplus 0b * x_3 \\ y_3 &= 0b * x_0 \oplus 0d * x_1 \oplus 09 * x_2 \oplus 0e * x_3. \end{aligned}$$

Step 7: Final Round

- The aforementioned steps are now repeated 8 times.
- In the final round (10^{th} round), we skip the 'InvMixColumns' operation (that is, we only perform InvShiftRows, InvSubBytes, AddRoundKey).
- At the end of these 10 rounds, we obtain the plaintext from the ciphertext.

RTL Verilog Code

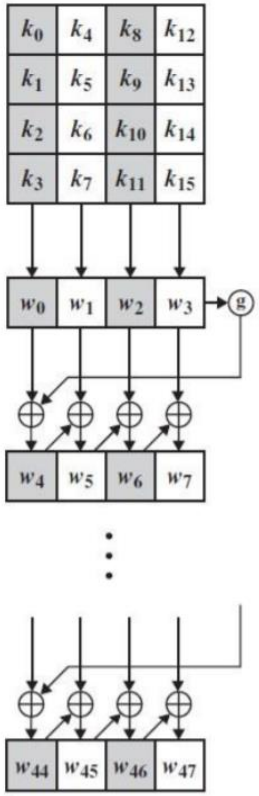
```
`timescale 1ns / 1ps

module AES_Decrypter(CipherText, key, PlainText);
    input wire[127:0] CipherText,key;
    output reg [127:0] PlainText;
```

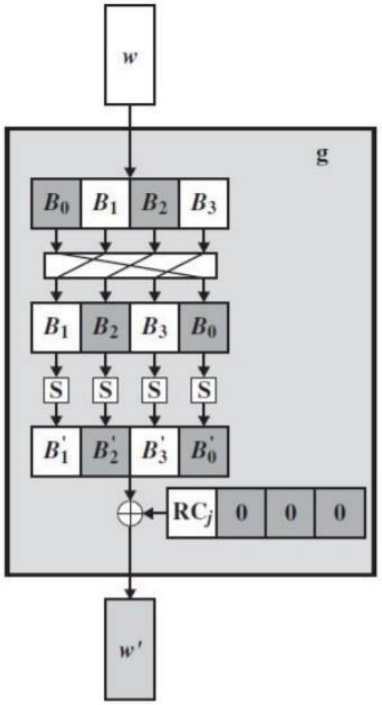
RTL Verilog Code (S-Box)

```
// Substitute Byte Transformation
function [127:0] SubBytes(input [127:0] State);
    integer i;
    reg [7 : 0] sbox [0 : 255];
    begin
        sbox[8'h00] = 8'h63;
        sbox[8'h01] = 8'h7c;
        sbox[8'h02] = 8'h77;
        sbox[8'h03] = 8'h7b;
        sbox[8'h04] = 8'hf2;
        sbox[8'h05] = 8'h6b;
        sbox[8'h06] = 8'h6f;
        sbox[8'h07] = 8'hc5;
        sbox[8'h08] = 8'h30;
        sbox[8'h09] = 8'h01;
        sbox[8'h0a] = 8'h67;
        sbox[8'h0b] = 8'h2b;
        sbox[8'h0c] = 8'hfe;
        sbox[8'h0d] = 8'hd7;
        sbox[8'h0e] = 8'hab;
        sbox[8'h0f] = 8'h76;
        sbox[8'h10] = 8'hca;
        sbox[8'h11] = 8'h82;
        sbox[8'h12] = 8'hc9;
        sbox[8'h13] = 8'h7d;
        sbox[8'h14] = 8'hfa;
        sbox[8'h15] = 8'h59;
        sbox[8'h16] = 8'h47;
        sbox[8'h17] = 8'hf0;
        sbox[8'h18] = 8'had;
        sbox[8'h19] = 8'hd4;
        sbox[8'h1a] = 8'ha2;
        sbox[8'h1b] = 8'haf;
        sbox[8'h1c] = 8'h9c;
        sbox[8'h1d] = 8'ha4;
        sbox[8'h1e] = 8'h72;
        sbox[8'h1f] = 8'hc0;
        sbox[8'h20] = 8'hb7;
        sbox[8'h21] = 8'hfd;
        sbox[8'h22] = 8'h93;
        sbox[8'h23] = 8'h26;
        sbox[8'hdf] = 8'h9e;
        sbox[8'he0] = 8'he1;
        sbox[8'he1] = 8'hf8;
        sbox[8'he2] = 8'h98;
        sbox[8'he3] = 8'h11;
        sbox[8'he4] = 8'h69;
        sbox[8'he5] = 8'hd9;
        sbox[8'he6] = 8'h8e;
        sbox[8'he7] = 8'h94;
        sbox[8'he8] = 8'h9b;
        sbox[8'he9] = 8'h1e;
        sbox[8'hea] = 8'h87;
        sbox[8'heb] = 8'he9;
        sbox[8'hec] = 8'hce;
        sbox[8'hed] = 8'h55;
        sbox[8'hee] = 8'h28;
        sbox[8'hef] = 8'hdf;
        sbox[8'hf0] = 8'h8c;
        sbox[8'hf1] = 8'ha1;
        sbox[8'hf2] = 8'h89;
        sbox[8'hf3] = 8'h0d;
        sbox[8'hf4] = 8'hbf;
        sbox[8'hf5] = 8'he6;
        sbox[8'hf6] = 8'h42;
        sbox[8'hf7] = 8'h68;
        sbox[8'hf8] = 8'h41;
        sbox[8'hf9] = 8'h99;
        sbox[8'hfa] = 8'h2d;
        sbox[8'hfb] = 8'h0f;
        sbox[8'hfc] = 8'hb0;
        sbox[8'hfd] = 8'h54;
        sbox[8'hfe] = 8'hbb;
        sbox[8'hff] = 8'h16;

        for(i=0;i<128;i=i+8)
        begin
            SubBytes[i+:8] = sbox[State[i+:08]];
        end
    end
endfunction
```

RTL Verilog Code (Key Expansion)

```
// Key Expansion
function [1279:0] key_expansion(input [127:0] key);
    reg [127 : 0] sub_key;
    reg [127 : 0] prev_key;
    reg [31:0] temp [0:3];
    reg [7:0] j[1:10];
    reg [1279:0] final_key;
    integer i;

    begin
        j[1]=8'h01;
        j[2]=8'h02;
        j[3]=8'h04;
        j[4]=8'h08;
        j[5]=8'h10;
        j[6]=8'h20;
        j[7]=8'h40;
        j[8]=8'h80;
        j[9]=8'h1b;
        j[10]=8'h36;
        prev_key = key;

        for(i=1;i<=10;i=i+1)
        begin
            sub_key = SubBytes(prev_key);
            temp[0] = prev_key[127:96]^ Shift_Column(sub_key[31:0]) ^ {j[i],24'h0};
            temp[1] = prev_key[95:64] ^ temp[0];
            temp[2] = prev_key[63:32] ^ temp[1];
            temp[3] = prev_key[31:0] ^ temp[2];
            prev_key = {temp[0], temp[1], temp[2], temp[3]};
            final_key[1279-(i-1)*128-:128] = prev_key;
        end

        key_expansion = final_key;
    end
endfunction

function [31:0] Shift_Column(input [31:0] column);
    begin
        Shift_Column = {column[23:16],column[15:8],column[7:0],column[31:24]};
    end
 endfunction
```



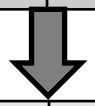


RTL Verilog Code (Add Round Key & Inverse Shift Rows)

```
// Add Round Key Transformation
function [127:0] Add_Round_key(input [127:0] State,input [127:0]  Round_key);
    begin
    Add_Round_key = State ^ Round_key;
    end
endfunction
```

```
// Inverse Shift Row Transformation
function [127 : 0] Inv_ShiftRow(input [127 : 0] state);
    reg [31 : 0] c0, c1, c2, c3;
    reg [31 : 0] sc0, sc1, sc2, sc3;
    begin
        c0 = state[127 : 096];
        c1 = state[095 : 064];
        c2 = state[063 : 032];
        c3 = state[031 : 000];

        sc0 = {c0[31 : 24], c3[23 : 16], c2[15 : 08], c1[07 : 00]};
        sc1 = {c1[31 : 24], c0[23 : 16], c3[15 : 08], c2[07 : 00]};
        sc2 = {c2[31 : 24], c1[23 : 16], c0[15 : 08], c3[07 : 00]};
        sc3 = {c3[31 : 24], c2[23 : 16], c1[15 : 08], c0[07 : 00]};

        Inv_ShiftRow = {sc0, sc1, sc2, sc3};
    end
endfunction
```

127-120	95-88	63-56	31-24
119-112	87-80	55-48	23-16
111-104	79-72	47-40	15-08
103-96	71-64	39-32	07-00



127-120	95-88	63-56	31-24
23-16	119-112	87-80	55-48
47-40	15-08	111-104	79-72
71-64	39-32	07-00	103-96

RTL Verilog Code (Inverse S-box)

```
// Inverse Substitute Bytes Transformation
function [127:0] InvSubBytes(input [127:0] State);
    integer i;
    reg [7 : 0] inv_sbox [0 : 255];
    begin
        inv_sbox[8'h00] = 8'h52;
        inv_sbox[8'h01] = 8'h09;
        inv_sbox[8'h02] = 8'h6a;
        inv_sbox[8'h03] = 8'hd5;
        inv_sbox[8'h04] = 8'h30;
        inv_sbox[8'h05] = 8'h36;
        inv_sbox[8'h06] = 8'ha5;
        inv_sbox[8'h07] = 8'h38;
        inv_sbox[8'h08] = 8'hbf;
        inv_sbox[8'h09] = 8'h40;
        inv_sbox[8'h0a] = 8'ha3;
        inv_sbox[8'h0b] = 8'h9e;
        inv_sbox[8'h0c] = 8'h81;
        inv_sbox[8'h0d] = 8'hf3;
        inv_sbox[8'h0e] = 8'hd7;
        inv_sbox[8'h0f] = 8'hfb;
        inv_sbox[8'h10] = 8'h7c;
        inv_sbox[8'h11] = 8'he3;
        inv_sbox[8'h12] = 8'h39;
        inv_sbox[8'h13] = 8'h82;
        inv_sbox[8'h14] = 8'h9b;
        inv_sbox[8'h15] = 8'h2f;
        inv_sbox[8'h16] = 8'hff;
        inv_sbox[8'h17] = 8'h87;
        inv_sbox[8'h18] = 8'h34;
        inv_sbox[8'he9] = 8'heb;
        inv_sbox[8'hea] = 8'hbb;
        inv_sbox[8'heb] = 8'h3c;
        inv_sbox[8'hec] = 8'h83;
        inv_sbox[8'hed] = 8'h53;
        inv_sbox[8'hee] = 8'h99;
        inv_sbox[8'hef] = 8'h61;
        inv_sbox[8'hf0] = 8'h17;
        inv_sbox[8'hf1] = 8'h2b;
        inv_sbox[8'hf2] = 8'h04;
        inv_sbox[8'hf3] = 8'h7e;
        inv_sbox[8'hf4] = 8'hba;
        inv_sbox[8'hf5] = 8'h77;
        inv_sbox[8'hf6] = 8'hd6;
        inv_sbox[8'hf7] = 8'h26;
        inv_sbox[8'hf8] = 8'he1;
        inv_sbox[8'hf9] = 8'h69;
        inv_sbox[8'hfa] = 8'h14;
        inv_sbox[8'hfb] = 8'h63;
        inv_sbox[8'hfc] = 8'h55;
        inv_sbox[8'hfd] = 8'h21;
        inv_sbox[8'hfe] = 8'h0c;
        inv_sbox[8'hff] = 8'h7d;

        for(i=0;i<128;i=i+8)
            begin
            InvSubBytes[i+:8] = inv_sbox[State[i+:08]];
            end
    end
endfunction
```

RTL Verilog Code (Inverse Mix Column)

```
// Inverse Mix Column Transformation
function [7 : 0] mul_2(input [7 : 0] in);
    begin
        mul_2 = {in[6 : 0], 1'b0} ^ (8'h1b & {8{in[7]}});
    end
endfunction

function [7 : 0] mul_4(input [7 : 0] in);
    begin
        mul_4 = mul_2(mul_2(in));
    end
endfunction

function [7 : 0] mul_8(input [7 : 0] in);
    begin
        mul_8 = mul_2(mul_4(in));
    end
endfunction

function [7 : 0] mul_9(input [7 : 0] in);
    begin
        mul_9 = mul_8(in) ^ in;
    end
endfunction

function [7 : 0] mul_11(input [7 : 0] in);
    begin
        mul_11 = mul_8(in) ^ mul_2(in) ^ in;
    end
endfunction

function [7 : 0] mul_13(input [7 : 0] in);
    begin
        mul_13 = mul_8(in) ^ mul_4(in) ^ in;
    end
endfunction
```

```
function [7 : 0] mul_14(input [7 : 0] in);
    begin
        mul_14 = mul_8(in) ^ mul_4(in) ^ mul_2(in);
    end
endfunction

function [31 : 0] inv_mixc(input [31 : 0] c);
    reg [7 : 0] b0, b1, b2, b3;
    reg [7 : 0] mb0, mb1, mb2, mb3;
    begin
        b0 = c[31 : 24];
        b1 = c[23 : 16];
        b2 = c[15 : 08];
        b3 = c[07 : 00];

        mb0 = mul_14(b0) ^ mul_11(b1) ^ mul_13(b2) ^ mul_9(b3);
        mb1 = mul_9(b0) ^ mul_14(b1) ^ mul_11(b2) ^ mul_13(b3);
        mb2 = mul_13(b0) ^ mul_9(b1) ^ mul_14(b2) ^ mul_11(b3);
        mb3 = mul_11(b0) ^ mul_13(b1) ^ mul_9(b2) ^ mul_14(b3);

        inv_mixc = {mb0, mb1, mb2, mb3};
    end
endfunction

function [127 : 0] inv_mixcolumns(input [127 : 0] info);
    reg [31 : 0] c0, c1, c2, c3;
    reg [31 : 0] cs0, cs1, cs2, cs3;
    begin
        c0 = info[127 : 096];
        c1 = info[095 : 064];
        c2 = info[063 : 032];
        c3 = info[031 : 000];

        cs0 = inv_mixc(c0);
        cs1 = inv_mixc(c1);
        cs2 = inv_mixc(c2);
        cs3 = inv_mixc(c3);

        inv_mixcolumns = {cs0, cs1, cs2, cs3};
    end
endfunction
```

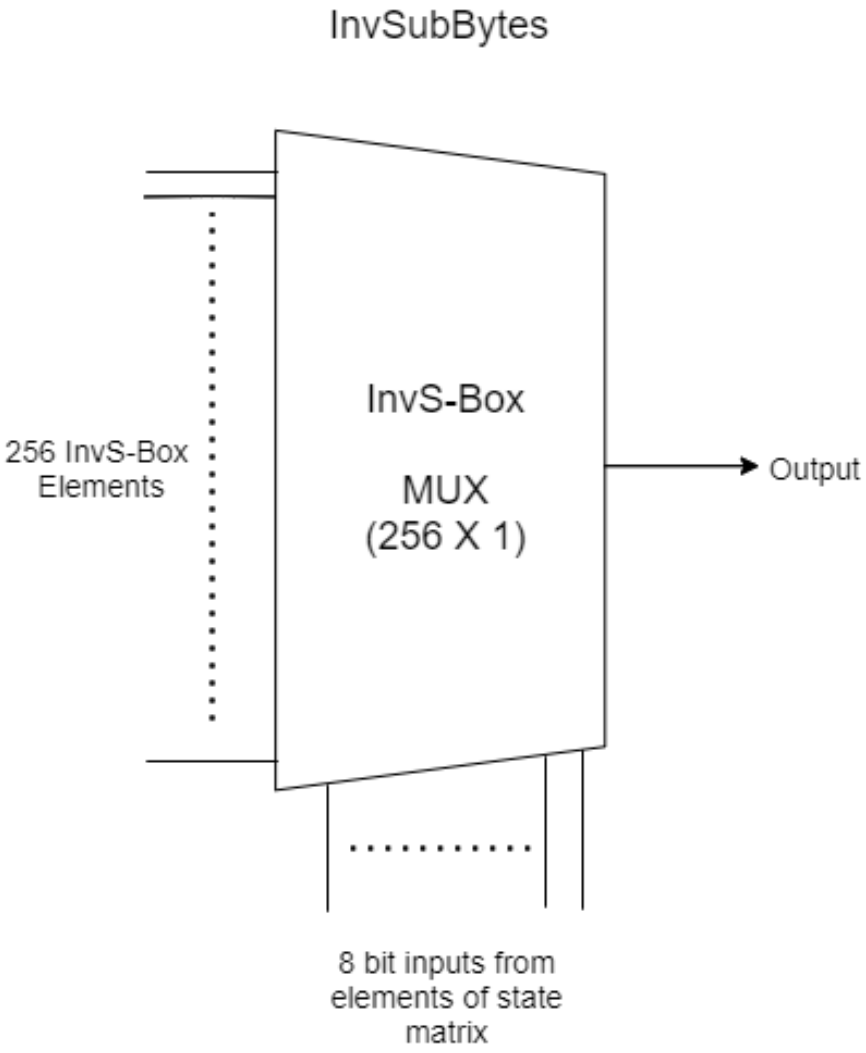
RTL Verilog Code (AES Decryption Rounds)

```
// AES Decryption Rounds
integer a;
always @*
    begin : implement
        reg [127 : 0] temp;
        reg [1279 : 0] full_key_expansion;
        full_key_expansion = key_expansion(key);
        temp = Add_Round_key(CipherText, full_key_expansion[127:0]);
        temp = Inv_ShiftRow(temp);
        temp = InvSubBytes(temp);

        for(a=9;a>0;a=a-1)
        begin
            temp = Add_Round_key(temp, full_key_expansion[1279-(a-1)*128-:128]);
            temp = inv_mixcolumns(temp);
            temp = Inv_ShiftRow(temp);
            temp = InvSubBytes(temp);
        end

        PlainText = Add_Round_key(temp,key);
    end
endmodule
```

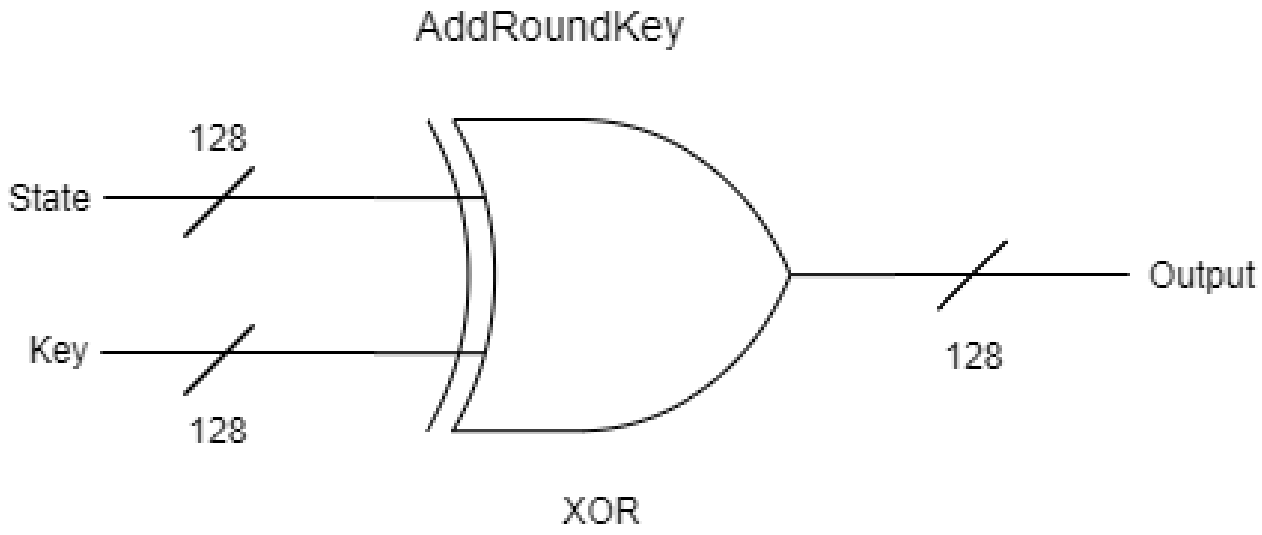
RTL Verilog Code (Test Bench)

```
`timescale 1ns / 1ps

module TB;
    reg [127:0] Test_State;
    reg [127:0] Test_Key;
    wire [127:0] Test_Result;

    AES_Decrypter a(Test_State,Test_Key,Test_Result);

    initial
    begin
       Test_State = 128'hff0b844a0853bf7c6934ab4364148fb9;
       Test_Key = 128'h0f1571c947d9e8590cb7add6af7f6798;
    end
endmodule
```

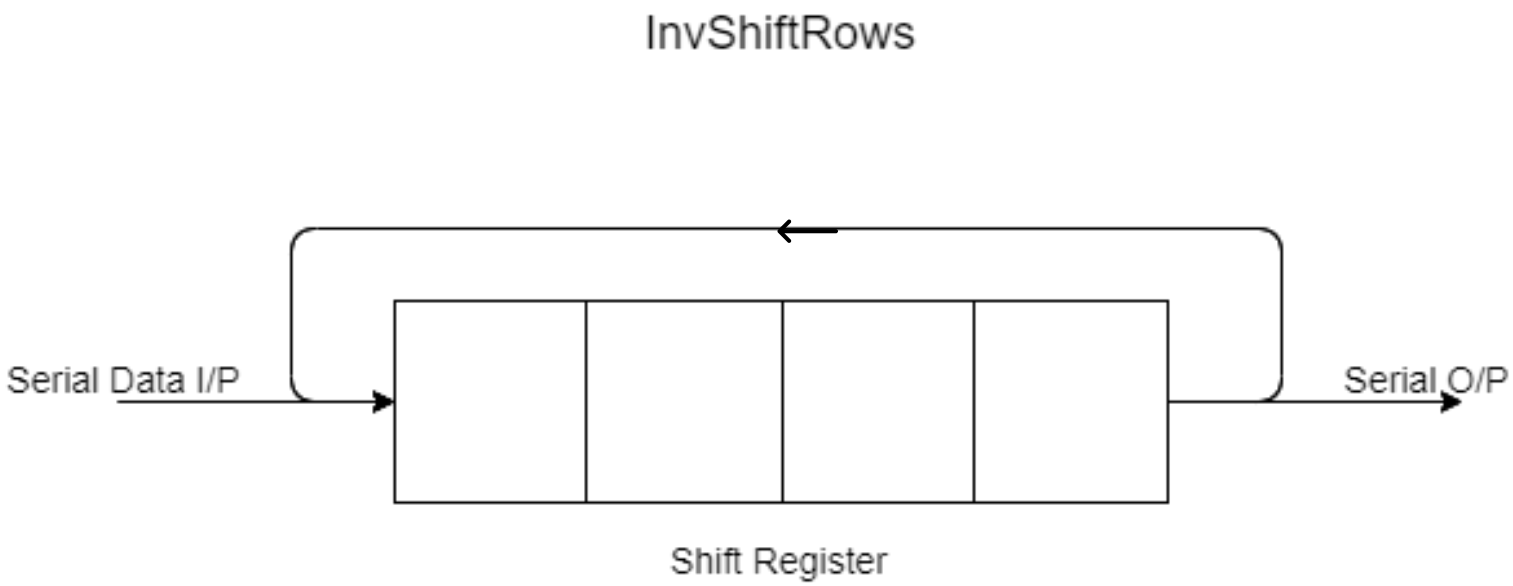
Schematic of InvSubBytes



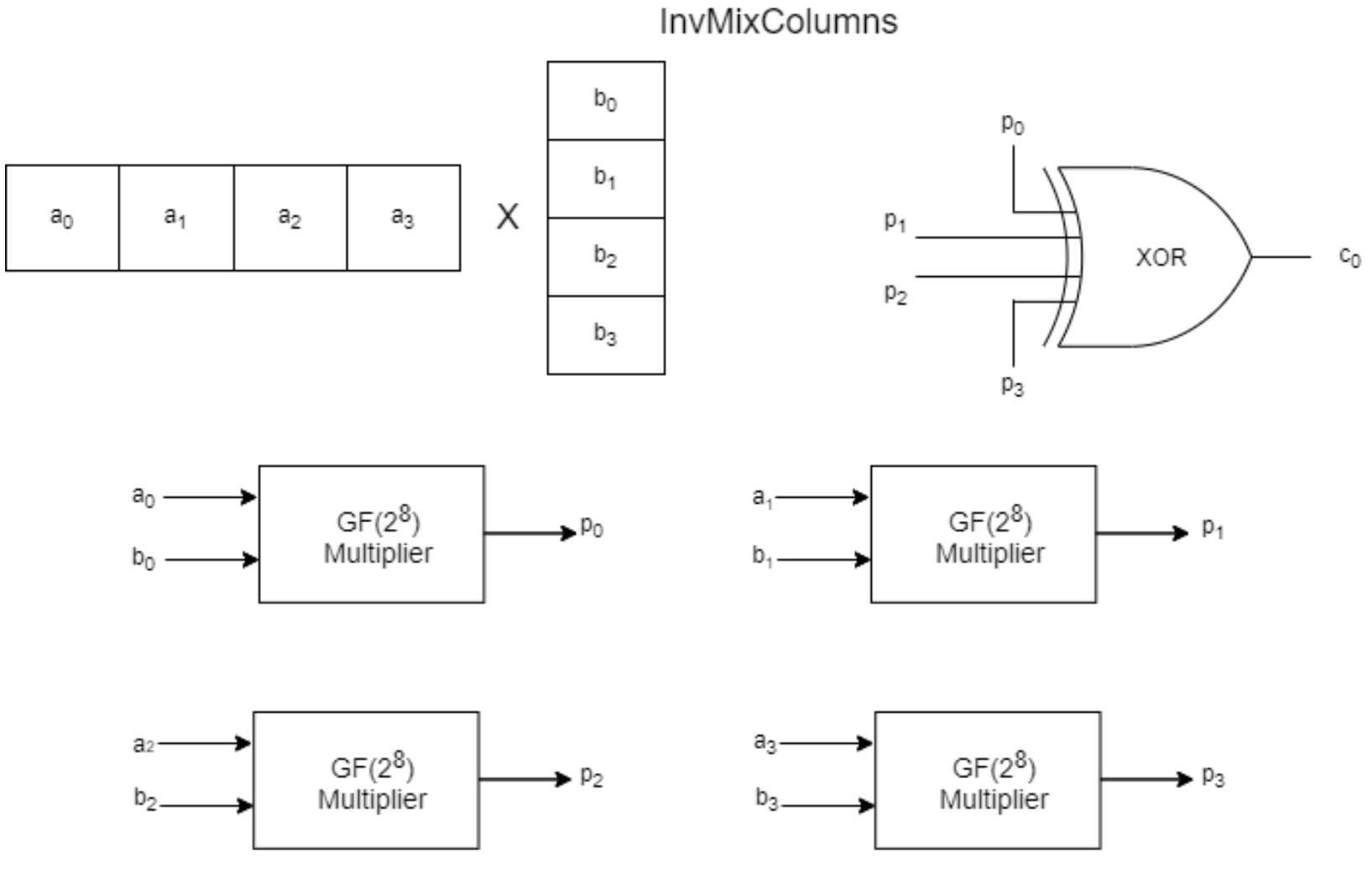
Schematic of AddRoundKey



Schematic of InvShiftRows



Schematic of InvMixColumns

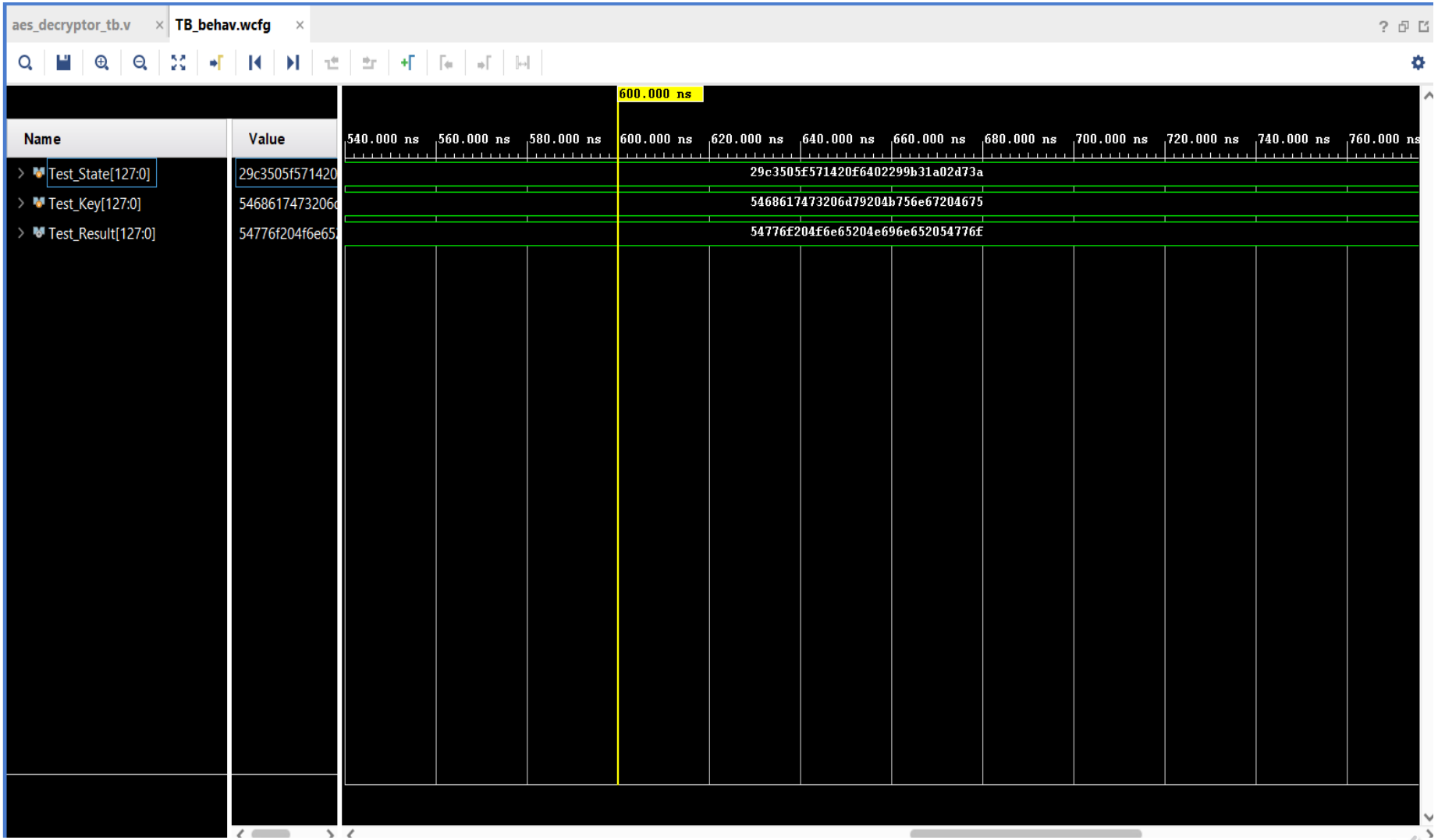


Simulation Results

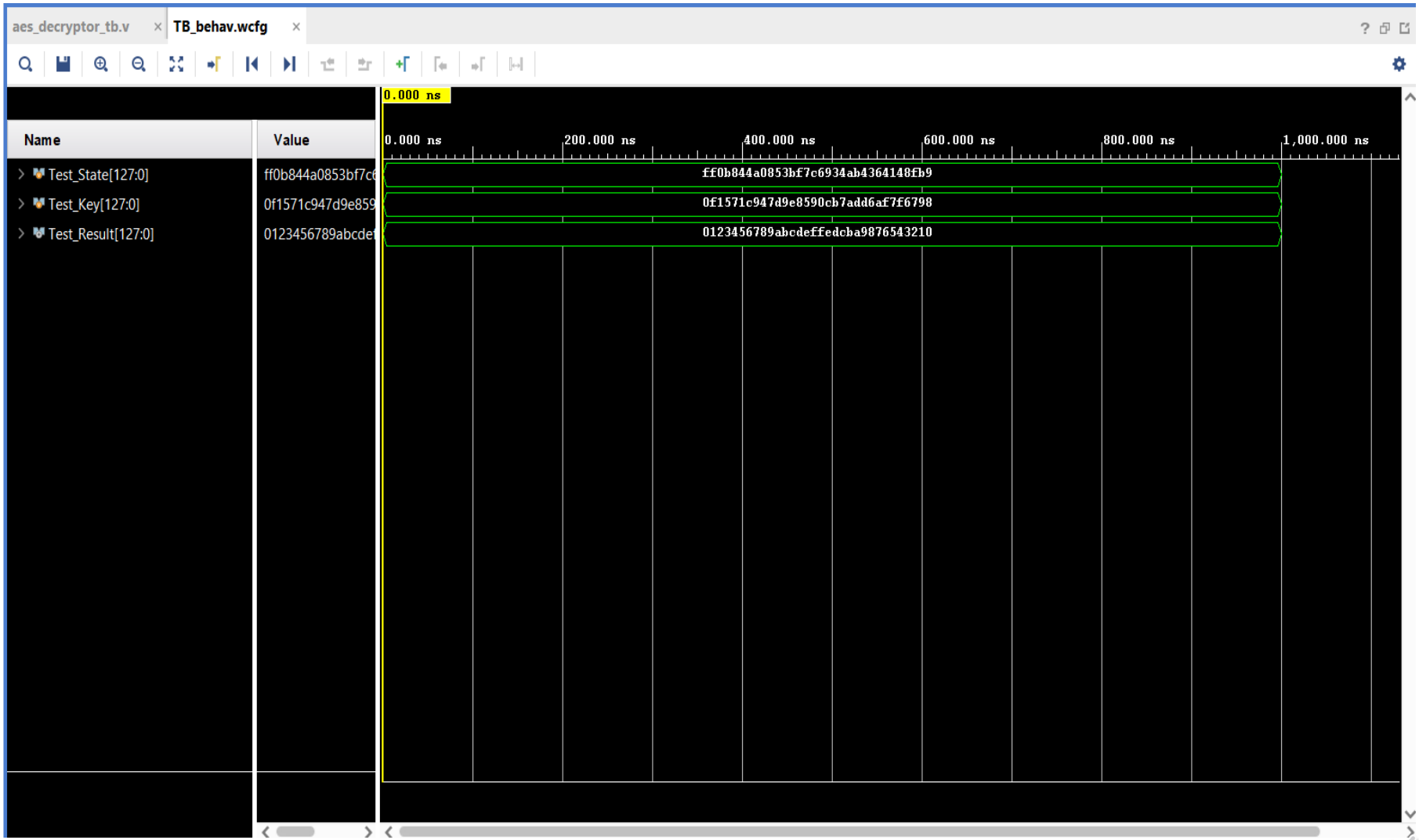
S.No.	Ciphertext	Key	Expected Plaintext
1	29c3505F571420f6402299b31a02d73a	5468617473206d79204b756e67204675	54776f204f665204e696e652054776f
2	ff0b844a0853bf7c6934ab4364148fb9	0f1571c947d9e8590cb7add6af7f6798	0123456789abcedffedcba9876543210
3	ae7614a563d5f29824182476369ffff	f1571c9546ae74863ef9752467983344	63881d081750d77786fc1c42c0717c93

- We have used Xilinx Vivado v2020.2 (64-bit) for running our simulations.
- The ciphertext-key-plaintext triplets required for validation have been generated using an online tool that can be found here.

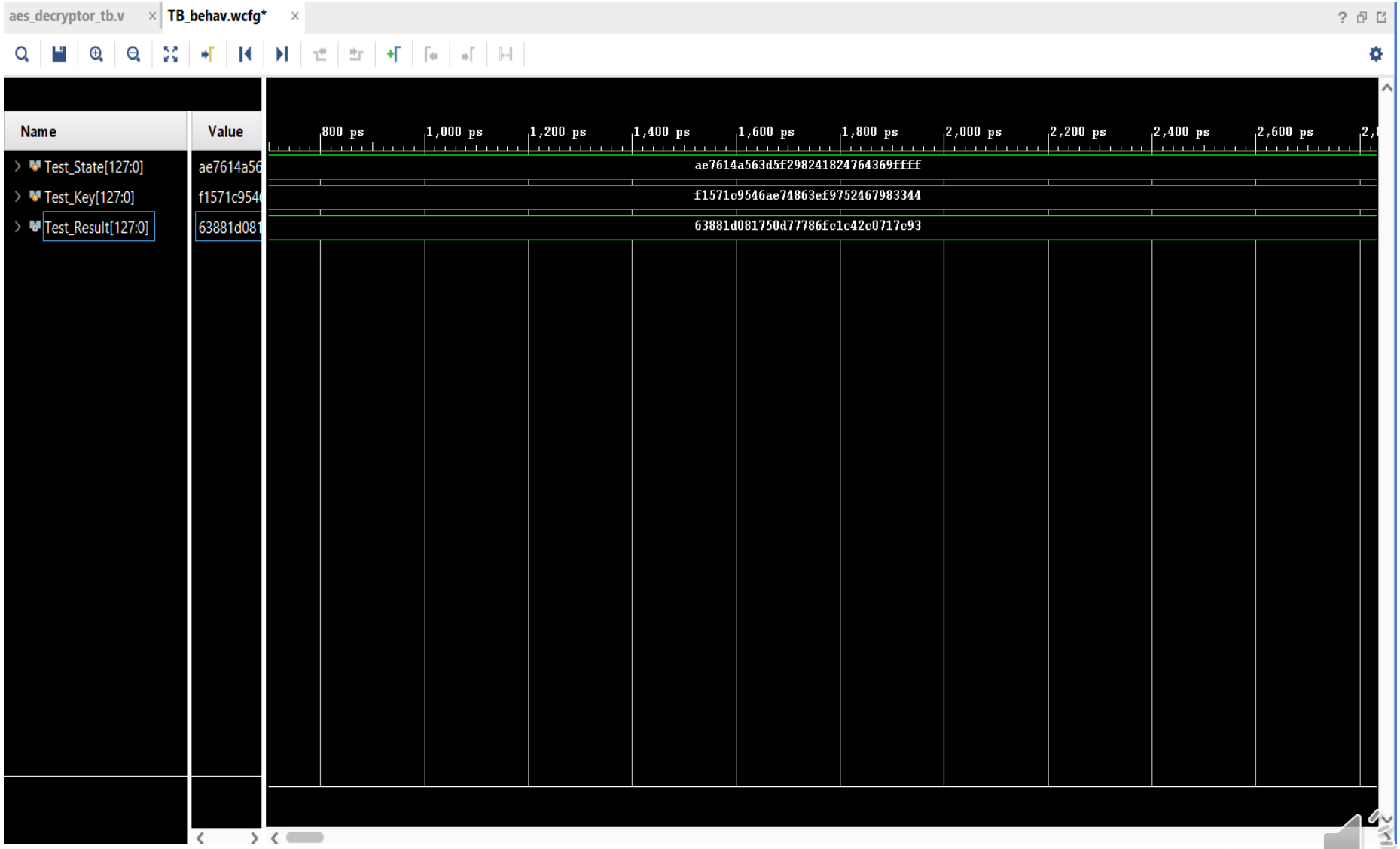
Simulation Result 1



Simulation Result 2



Simulation Result 3



Summary

- The Advanced Encryption Standard (AES) algorithm makes vast improvements over its predecessors in both security and speed.
- In this project, we have seen how this algorithm can be implemented on hardware (using Verilog).
- The four stages involved in encryption are entirely reversible with the help of the key. Thus, we can obtain the desired plaintext from ciphertext easily.

With no apparent mathematical weakness discovered yet, we can be reasonably confident that AES will keep our data secure for the years to come.