Advanced Encryption Standard (AES)

Overview

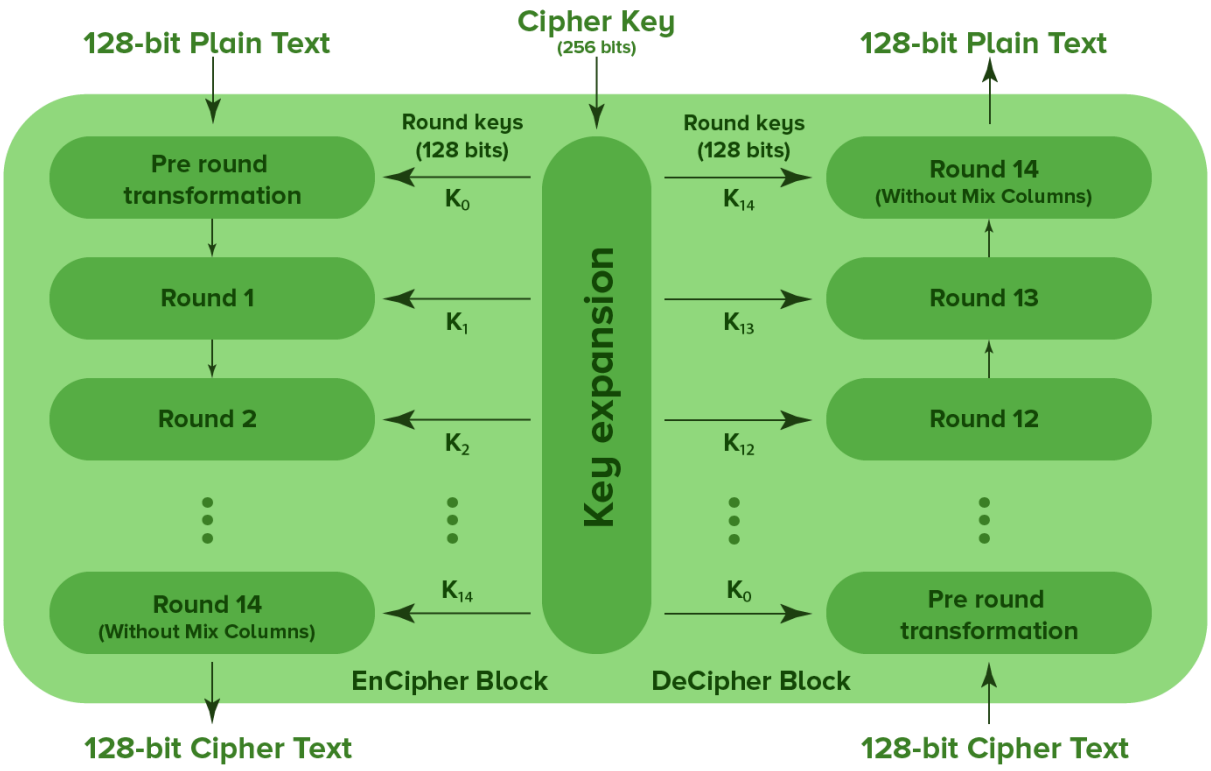


Figure 3: Overview of AES-256

AES 256 aims to encrypt plaintext of 128 bits into ciphertext, with the help of 256 bits secret key. The ciphertext is transferred to the receiver, and using the same secret key, the ciphertext is decrypted to give back the same plaintext.

Verilog implementation of pre-round and 14 rounds is shown below.

```
if ((Done == 1'b0))//(Start == 1'b1) & (Done == 1'b0)
   begin
   if (count == 5'h0)
      begin
         Ki_0 = Key0;
         Mr0 = Msg_in; Kr0 = Key0[0:127];
      end
   else if (count == 5'h1)
      begin
         K1 = Ki_1; K2 = Ki_2; K3 = Ki_3; K4 = Ki_4; K5 = Ki_5; K6 = Ki_6;
         K7 = Ki_7; K8 = Ki_8; K9 = Ki_9; KA = Ki_A; KB = Ki_B; KC = Ki_C;KD = Ki_D;
         Mr1 = Emr0; Kr1 = Key0[128:255];
      end
   else if (count == 5'h2)
      begin
         Mr1 = Emr1; Kr1 = K1;
      end
   ...
   else
      begin
         Msg_out = Emr2;
         Done = 1'b1;
      end
   count = count + 5'h1;
   end
end
```

Pre-round transformation is done at count = 0. All the required keys are calculated in the next cycle. Emr1 is the output of the round module and the same is given as input to the round in the next cycle.

Encryption

Plaintext of **128** bits, along with the secret key of **256** bits is converted into the encrypted text of 128 bits.

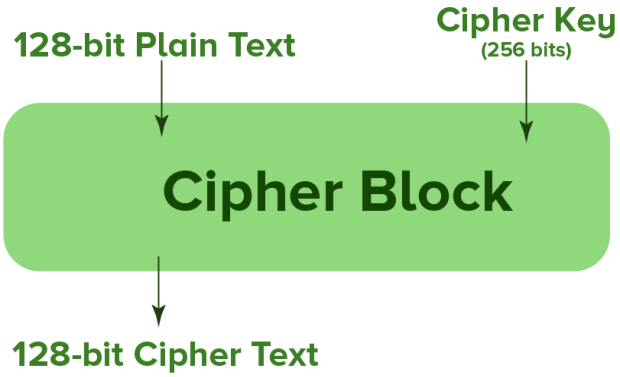


Figure 4: High-level abstraction of encryption

AES is an iterative algorithm, carried out in **rounds**, which involve replacing, shuffling and adding bytes. The number of rounds is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and **14 rounds for 256-bit keys**.

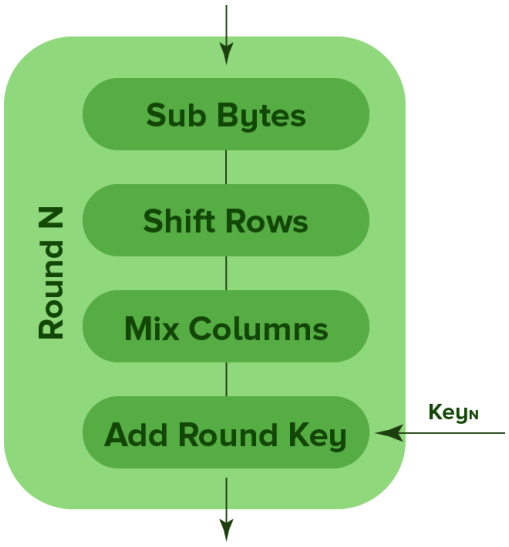


Figure 5: Abstraction of general round n

Each round comprises 4 sub-processes. The **last round** comprises only three sub-processes (without Mix Columns). For a **general round**, (Round N, not the final round) every subprocess is shown in fig. 5.

Verilog implementation:

The main round (general round) is as described.

```
Sub_byter      r_A (C_Msg_in, C_Msg_1);
Shift_rowr     r_B (C_Msg_1, C_Msg_2);
Mix_columnr    r_C (C_Msg_2, C_Msg_3);
Add_round_keyr r_S (C_Msg_3, R_key, C_Msg_out);
```

The latter part differentiates the last round from other rounds. The individual modules are elaborated on in the later sections. The suffix ‘r’ after the submodule names represent either row or column words.

Round subprocess

AES operates on 128 bits of data. The input data undergoes various modifications, yet, is 128-bit data every time. This intermediate data is called **state**. The state of 128 bits (i.e, 16 bytes) is arranged as a 4x4 byte matrix, hence known as the state matrix.

Pre round transformation

This step is bitwise XOR of plaintext and key 0. Required keys (key2 to key14) are provided by the key expansion process.

Byte Substitution

The 16 input bytes are substituted by looking up a fixed table (S-box). The output is also a matrix of four rows and four columns.

For example,

31	64	a8	ed	bytes are substituted in the S-box to get output	c7	43	c2	55

Byte 31 ⟹ Row number 3 (i.e, 30) and Column number 1 (i.e, 01) in the box contains the Byte C7.

This means 8 bits 00110001 (31 in hexadecimal) is replaced with 11000111 (C7) after SUB operation.

Verilog implementation of S - box is as follows,

```
always@(Sin)
begin
   case(Sin)
   //R0
   8'h00: Sout = 8'h63; 8'h01: Sout = 8'h7C; 8'h02: Sout = 8'h77; 8'h03: Sout = 8'h7B;
   8'h04: Sout = 8'hF2; 8'h05: Sout = 8'h6B; 8'h06: Sout = 8'h6F; 8'h07: Sout = 8'hC5;
   8'h08: Sout = 8'h30; 8'h09: Sout = 8'h01; 8'h0A: Sout = 8'h67; 8'h0B: Sout = 8'h2B;
   8'h0C: Sout = 8'hFE; 8'h0D: Sout = 8'hD7; 8'h0E: Sout = 8'hAB; 8'h0F: Sout = 8'h76;
   ...
   endcase
end
```

Here, ‘Sin’ and ‘Sout’ are one byte before and after the substitution respectively.

ShiftRows

After BYTEs substitution, the 128-bit state matrix undergoes shift rows operation.

Each of the four rows of the matrix is shifted to the left in a **circular fashion**, respectively.

- The first row is not shifted (Shifted zero bytes to the left).
- The second row is shifted one position (one byte) to the left.
- The third row is shifted two positions (two bytes) to the left.
- The fourth row is shifted three positions (three bytes) to the left, circularly.
- The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.

Before					After			
00	44	88	CC	For a simple example of a 4x4 state matrix, **shift row** operation is shown.	00	44	88	CC
11	55	99	DD		55	99	DD	11
22	66	AA	EE		AA	EE	22	66
33	77	BB	FF		FF	33	77	BB

Verilog implementation of Shift rows is as follows,

```
input [0:127] Stin;
output [0:127] Stout;

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

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

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

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

State ‘Stin’ is before and ‘Stout’ is after ShiftRows operations. The whole state matrix is recognised as BYTEs. Each byte is assigned its new position. Without this process, ciphertext would have been treated as four independent cipher columns.

MixColumns

Each column of four bytes is now transformed individually, using a special mathematical function. This function takes one column as input and outputs four completely new column bytes (as shown below), which replace the original column. The result is another new matrix consisting of 16 new bytes.

> It must be noted that this step is **not performed in the last round**.

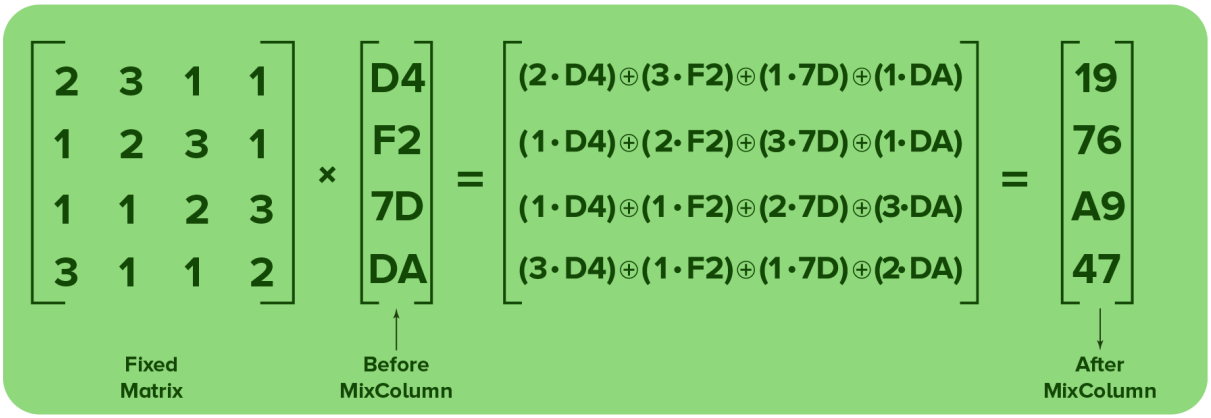


Figure 6: Mix column operation functioning

Verilog implementation of Mix columns round is:

```
Mod2 m1 (Min[0:7], tempr1[0:7]);
Mod1 m2 (Min[0:7], tempr1[8:15]);
Mod1 m3 (Min[0:7], tempr1[16:23]);
Mod3 m4 (Min[0:7], tempr1[24:31]);
assign Mout[0:7]   = tempr1[0:7]^tempr2[0:7]^tempr3[0:7]^tempr4[0:7];
```

Where 'Mout' is the output of the process. 'tempr1' wire contains the corresponding **X**2 and **X**3 of the first byte of a column performing functions. These multiplications of inputs with 2 and 3 are in the Galois field, realized using Mod2 and Mod3 modules described below.

```
always@ (Sin)
begin
case (Sin[7])
   1'b0: Sin_x2 = {Sin[6:0], 1'b0};
   1'b1: Sin_x2 = {Sin[6:0], 1'b0}^8'h1b;
endcase
end
```

```
always@ (Sin)
begin
case (Sin[7])
   1'b0: Sin_x3 = {Sin[6:0], 1'b0}^Sin[7:0];
   1'b1: Sin_x3 = {Sin[6:0], 1'b0}^8'h1b^Sin[7:0];
endcase
end
```

The Mod2 function involves shifting the BYTE towards the left by 1 bit and optional XOR with standard 0001 1011, based on the MSB of the BYTE. The Mod3 function of a byte B involves the XOR of B and Mix2(B).

Addroundkey

The 16 bytes of the matrix are now considered as 128 bits and are XORed with the 128 bits of the round key (key n). Similar rounds are carried 14 times in total. (Excluding Mix Columns in the final round)

Key expansion

We require 15 128-bit keys for encryption/decryption, two of them are already two halves of the main cipher key, so 13 more keys are to be created. The **Expanded key** module generates a new key of 256 bits, from its previous 256-bit key. We now generate 7 new 256-bit keys from the cipher key one after the other. The new keys are halved and used as the round keys as shown in fig. 8.

> **It is very important to note that each of these 256 bits keys is sufficient for 2 rounds, as each round requires a 128 bits key.**

For example, **our cipher key** is "**WADHWANI_ELECTRONICS_LABORATORY.**" of 32 Bytes or 256bits (a 'word' contains 32 bits, so the words **column** and **word** are used interchangeably). Converting each letter into hexadecimal (ASCII to hex), gives 2hex characters, each hexadecimal character corresponds to 4 bits ⇒ 2x4 = 8 bits

W	A	D	H	W	A	N	I	_	E	L	E	C	T	R	O	N	I
57	41	44	48	57	41	4E	49	5F	45	4C	45	43	54	52	4F	4E	49

C	S	_	L	A	B	O	R	A	T	O	R	Y	.
43	53	5F	4C	41	42	4F	52	41	54	4F	52	59	2E

Arranging these bytes are represented as words of 32 bits as shown below.

57	57	5F	43	4E	5F	4F	4F
41	41	45	54	49	4C	52	52
44	4E	4C	52	43	41	41	59
48	49	45	4F	53	42	54	2E
W1	W2	W3	W4	W5	W6	W7	W8

Every alternate round requires a key of size 256 bits and there are 14 rounds along with a pre-round, so **15 rounds** in total. Round constant (**RCON**) is provided by the following seven-column table, with one column for each round of key expansion.

Table 1. Round constants for key generation

01	02	04	08	10	20	40
00	00	00	00	00	00	00
00	00	00	00	00	00	00
00	00	00	00	00	00	00

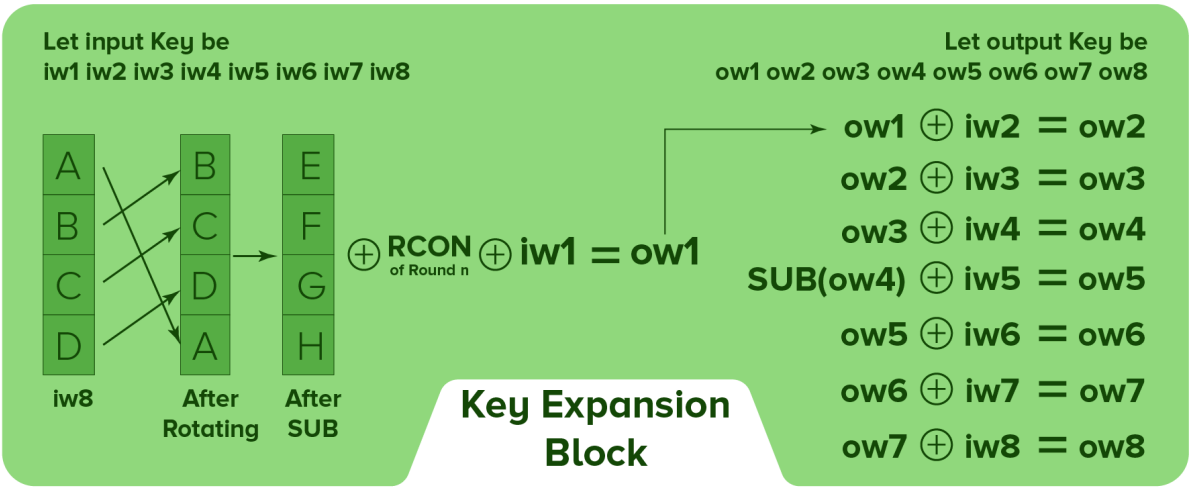


Figure 7: Process of key expansion

Now, we must extract **15** keys of 128 bits each, from the cipher key (main key) of 256 bits as shown in fig. 7 by following the below steps 7 times.

1. Rotate the last column of the keyn. (as shown below)
2. Substitute bytes of the rotated column in the forward S-box
3. Bitwise XOR of RCON, substituted byte and first column of keyn, becomes the first column of keyn+1 (next round key).
4. XOR of the 2nd column of keyn and the above-obtained column becomes the 2nd column of keyn+1. This step is repeated 2 more times till the 4th column of keyn+1 is obtained.
5. All 4 BYTEs of the 4th column are now Substituted in the Forward S-box.

6. The new column/word, on XORing with the 5th column of keyn gives the 5th column of keyn+1. This step is repeated 3 more times.

Now, with the knowledge of Key expansion/generation, we store all the 15 required keys in the expanded key block. This block readily supplies the required keys to each round. This block only requires the cipher key and the round number.

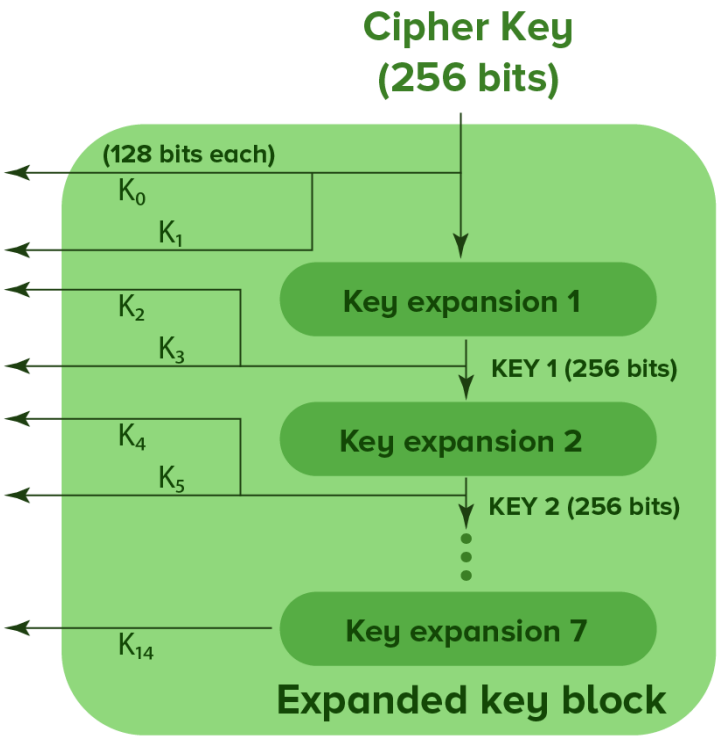


Figure 8: Abstraction of key generation process

Verilog implementation of the Expanded key module is as follows:

K1, K2, K3s are 256-bit keys, obtained from each key expansion module and k0, k1, k2s are of 128 bits, which are to be supplied to the corresponding rounds.

```
wire [127:0] k0,k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11,k12,k13,k14;
wire [255:0] K1,K2,K3,K4,K5,K6,K7;
assign k0 = Key[255:128];
assign k1 = Key[127:0];

KeyExpansion KExp1 (Key,0,K1);
assign k2 = K1[255:128];
assign k3 = K1[127:0];

KeyExpansion KExp2 (K1,1,K2);
assign k4 = K2[255:128];
assign k5 = K2[127:0];

KeyExpansion KExp3 (K2,2,K3);
assign k6 = K3[255:128];
assign k7 = K3[127:0];
```

The below code snippet selects the required round key, based on the round number.

```
// Round number in binary is s3s2s1s0
wire [1023:0] c1;
assign c1 = s3 ? {k8,k9,k10,k11,k12,k13,k14,k14}:{k0,k1,k2,k3,k4,k5,k6,k7};

wire [511:0] c2;
assign c2 = s2 ? c1[511:0]:c1[1023:512];

wire [255:0] c3;
assign c3 = s1 ? c2[255:0]:c2[511:256];

wire [127:0] c4;
assign c4 = s0 ? c3[127:0]:c3[255:128];

assign Keyout = c4;
```

Implementation of the Key expansion/generation block is as shown beside

ColSUB function gives the column after substituting in the forward S-box.

As specified in the chart above, ow1 and ow2 represent words/columns of the output key, and w1, w2 are of the input key.

```
wire [31:0]temp1,temp2,temp3;
assign temp1 = rcon(rc);
assign temp2 = temp^temp1;
assign ow8 = temp2^w1;
assign ow7 = temp2^w1^w2;
assign ow6 = temp2^w1^w2^w3;
assign ow5 = temp2^w1^w2^w3^w4;
ColSUB CS2(temp2^w1^w2^w3^w4,temp3);
assign ow4 = temp3^w5;
assign ow3 = temp3^w5^w6;
assign ow2 = temp3^w5^w6^w7;
assign ow1 = temp3^w5^w6^w7^w8;
```

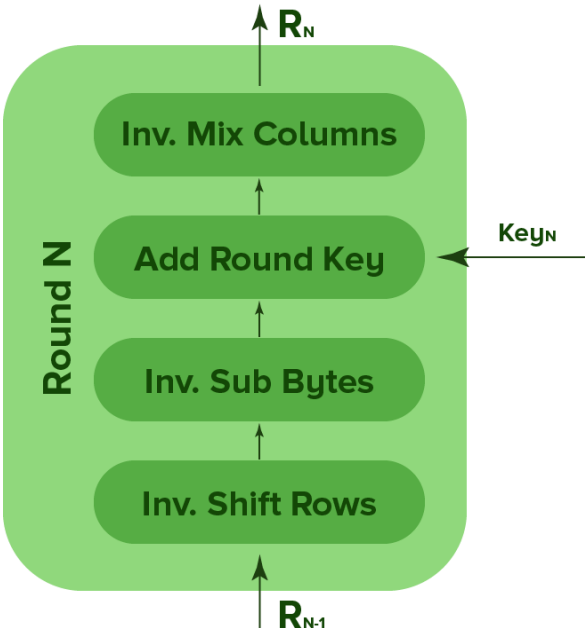
Decryption

Symmetric ciphers like AES assume that both the sender and receiver share the common secret key. So, the same 256-bit secret key is present with the receiver also.

Key expansion

In decryption, keys (128-bit) supplied to the rounds are the same as in encryption, but the order is reversed. Key0 of decryption is key14 of encryption, similarly, i^{th} key of decryption is the $(14-i)^{th}$ key of encryption.

Rounds



This process is inverse of the Encryption. Similar to Encryption, this procedure consists of 14 rounds.

Each round consists of 4 sub-processes which slightly differ and are elaborated below. The order of the sub-processes also **differs** from that of encryption.

Inverse shiftrow

The very first sub-process of the round. Each of the four rows of the matrix is shifted to the **right** in a **circular fashion**.

- The first row is not shifted (Shifted zero bytes to the right).
- The second row is shifted one position (one byte) to the right.
- The third row is shifted two positions (two bytes) to the right.
- The fourth row is shifted three positions (three bytes) to the right, circularly.

Inverse SUB bytes

The 16 input bytes are substituted by looking up the Inverse S-box. The output is also a matrix of four rows and four columns.

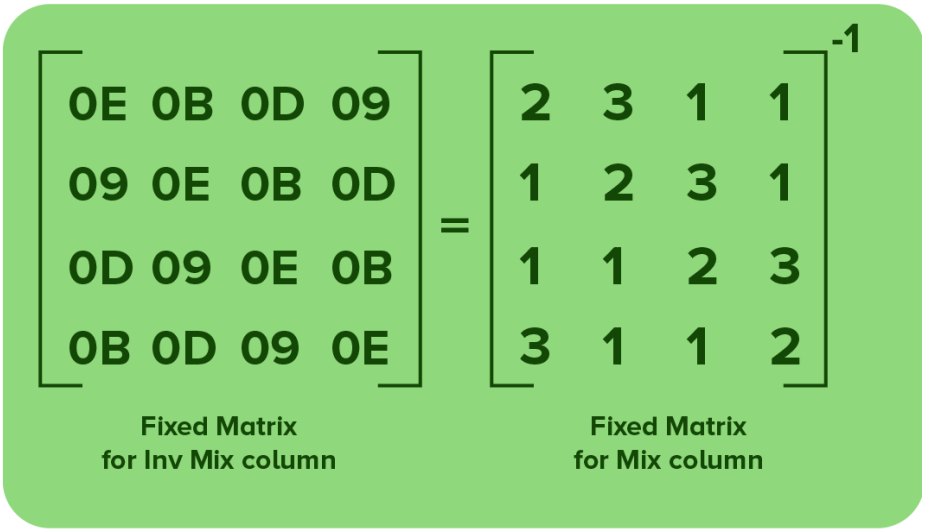
Inverse add round key

The 128-bit output from inverse SUB bytes is performed bitwise XOR with the 128-bit round key (Obtained from the inverse key expansion block).

Inverse MixColumns

Output matrix of inverse add round key is performed inverse mix columns operation.

The difference between mix columns and inverse mix columns is only the matrix with which the column is multiplied.



> It must be noted that this step is **not performed in the last round**.