Written by: Mohamed Sayed

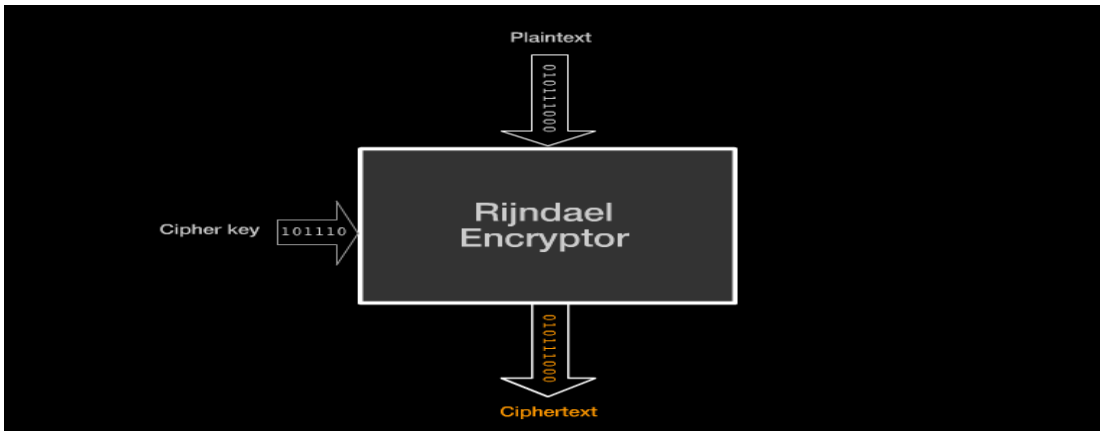
Design of AES (Advanced Encryption Standard) 128-Bits Based on (RIJNDAEL) Encryption Algorithm

ABSTRACT

In this document we will provide a detailed and easy to understand explanation of the implementation of the AES (RIJNDAEL) encryption algorithm and then we will show and discuss the results of our design. In order to fulfill the requirements of executing precise calculations and less power & area consumption, we performed many optimizations.

INTRODUCTION

To make a data in hidden form, it is necessary to change the data from its original form. Cryptography is the art of representation of data from its original form to another form which is not readable. For this purpose several algorithms are used in cryptography. AES is a cryptographic algorithm used to protect electronic data. AES is a symmetric block cipher which is capable of using cryptographic keys of 128, 192 and 256 bits to encrypt data block of 128 bits. In Symmetric key cryptography a shared key used for both encryption and decryption process. The AES encryption process of AES-128 bit consists of 10 rounds. Each round performs different operation such as shift rows, byte substitution, mix column step and addition of round key operations.



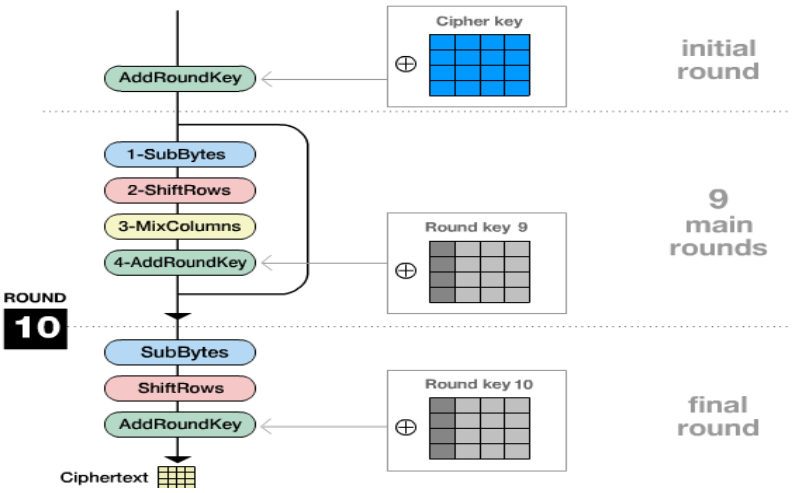
Brief History

Effective May 26, 2002 the National Institute of Science and Technology (NIST) has selected a block cipher called RIJNDAEL (named after its creators Vincent Rijmen and Joan Daemen) as the symmetric key encryption algorithm to be used to encrypt sensitive but unclassified American federal information.

RIJNDAEL was originally a variable block (16, 24, 32 bytes) and variable key size (16, 24, 32 bytes) encryption algorithm. NIST has however decided to define AES with a block size of 16 bytes while keeping their options open for future changes.

AES Algorithm

AES is an iterated symmetric block cipher. AES as well as most encryption algorithms is reversible. This means that almost the same steps are performed to complete both encryption and decryption in reverse order. The AES algorithm operates on bytes, which makes it simpler to implement and explain.



This key is expanded into individual sub keys, a sub keys for each operation round. This process is called KEY EXPANSION.

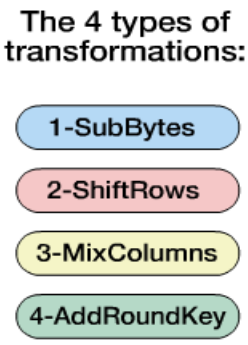
As mentioned before AES is an iterated block cipher. All that means is that the same operations are performed many times on a fixed number of bytes. These operations can easily be broken down to the following functions:

- ADD ROUND KEY
- BYTE SUB
- SHIFT ROW
- MIX COLUMN

Encryption

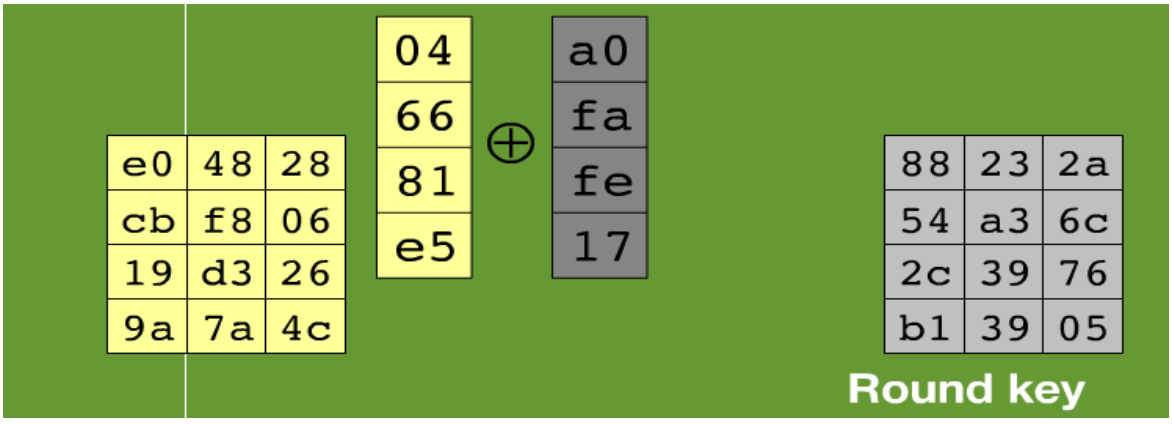
AES encryption cipher using a 16 byte key:

Round	Function
-	Add Round Key(State)
0	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Shift Row(Byte Sub(State)))



Add Round Key

Each of the 16 bytes of the state is XORed against each of the 16 bytes of a portion of the expanded key for the current round.



The first time Add Round Key gets executed:

state	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Round Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

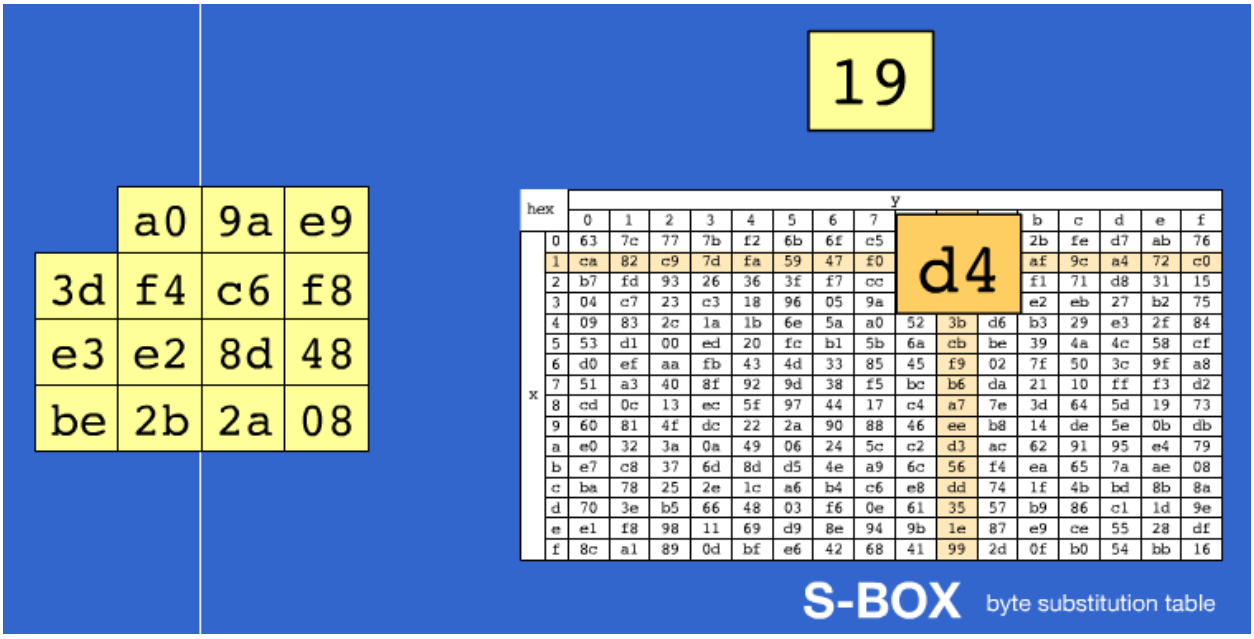
The second time Add Round Key is executed:

state	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Round Key	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

And so on for each round of execution.

SubBytes

During encryption each value of the state is replaced with the corresponding SBOX value.



AES S-Box Lookup Table:

	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

Shift Row

Arranges the state in a matrix and then performs a circular shift for each row. The circular shift just moves each byte one space over. The circular part of it specifies that the byte in the last position shifted one space will end up in the first position in the same row.
The state is arranged in a 4x4 matrix (square). Each row is then moved over (shifted) 1, 2 or 3 spaces over to the right, depending on the row of the state.

Row1	0
Row2 →	1
Row3	2
Row4	3

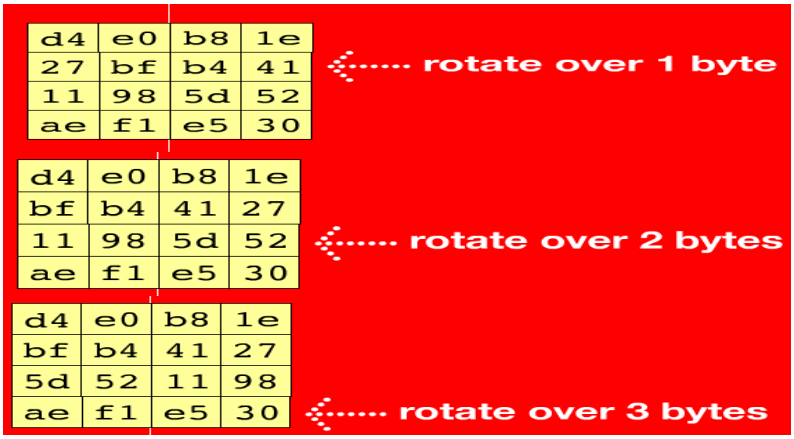
From

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

→

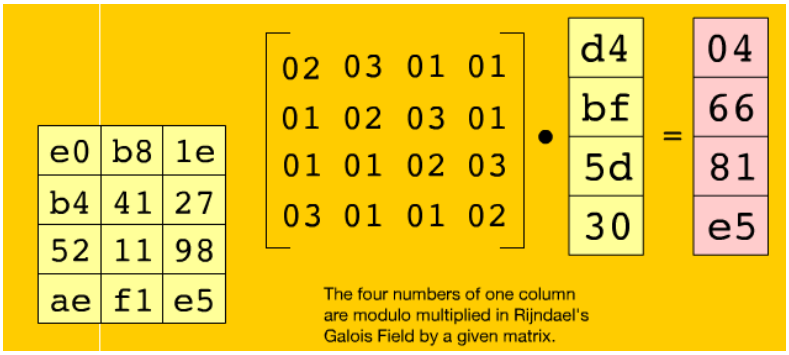
To

1	2	3	4
6	7	8	5
11	12	9	10
16	13	14	15



Mix Column

There are two parts to this step. The first will explain which parts of the state are multiplied against which parts of the matrix. The second will explain how this multiplication is implemented over what's called a Galois Field.



Matrix Multiplication

The multiplication is performed one column at a time (4 bytes). Each value in the column is eventually multiplied against every value of the matrix (16 total multiplications). The results of these multiplications are XORed together to produce only 4 result bytes for the next state. Therefore 4 bytes input, 16 multiplications 12 XORs and 4 bytes output. The multiplication is performed one matrix row at a time against each value of a state column.

Multiplication Matrix:

2	**3**	**1**	**1**
1	**2**	**3**	**1**
1	**1**	**2**	**3**
3	**1**	**1**	**2**

16 byte State

b1	**b5**	**b9**	**b13**
b2	**b6**	**b10**	**b14**
b3	**b7**	**b11**	**b15**
b4	**b8**	**b12**	**b16**

```
b1 = (b1 * 2) XOR (b2*3) XOR (b3*1) XOR (b4*1)

b2 = (b1 * 1) XOR (b2*2) XOR (b3*3) XOR (b4*1)

b3 = (b1 * 1) XOR (b2*1) XOR (b3*2) XOR (b4*3)

b4 = (b1 * 3) XOR (b2*1) XOR (b3*1) XOR (b4*2)
```

Galois Field Multiplication

The multiplication mentioned above is performed over a Galois Field which can be done quite easily with the use of the following two tables in (HEX).

E Table

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

L Table

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

D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

The result of the multiplication is simply the result of a lookup of the **L** table, followed by the addition of the results, followed by a lookup to the **E** table. The addition is a regular mathematical addition represented by +.

Example: **AF * 8**

1- L(AF)=B7 , L(08)=4B

2- if (B7 + 4B > FF) -> result = ((B7 + 4B) – FF =03

*3- E(03)= **0F***

One last exception is that any number multiplied by one is equal to its self and does not need to go through the above procedure. For example: FF * 1 = FF

Therefore the result of multiplying AF * 8 over a Galois Field is 0F

Mix Column Example During Encryption:

Input = D4 BF 5D 30

*Output(0) = (D4 * 2) XOR (BF*3) XOR (5D*1) XOR (30*1)*

= E(L(D4) + L(02)) XOR E(L(BF) + L(03)) XOR 5D XOR 30

= E(41 + 19) XOR E(9D + 01) XOR 5D XOR 30

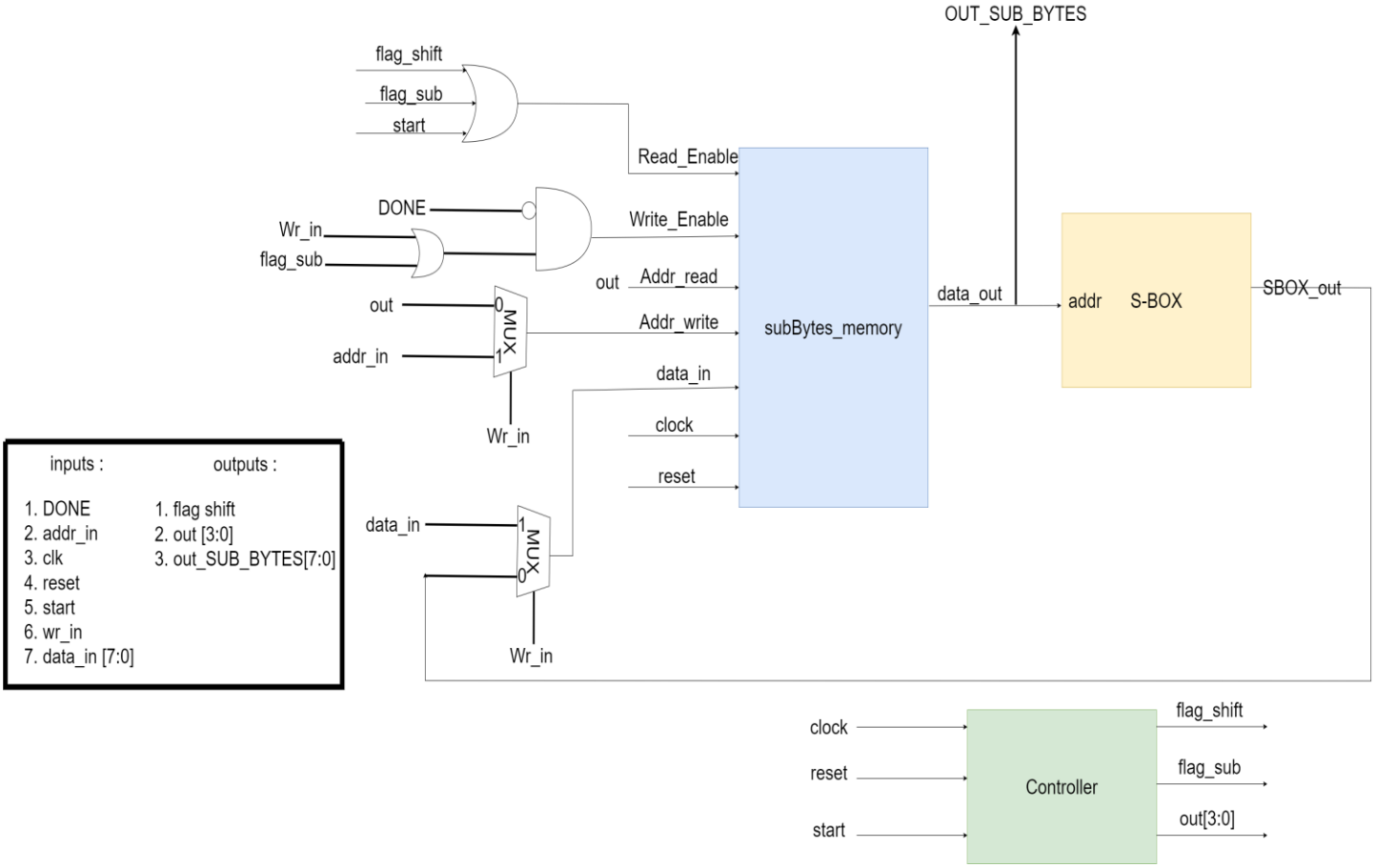
= E(5A) XOR E(9E) XOR 5D XOR 30

= B3 XOR DA XOR 5D XOR 30

*= **04***

Hardware Design

SubBytes STATE

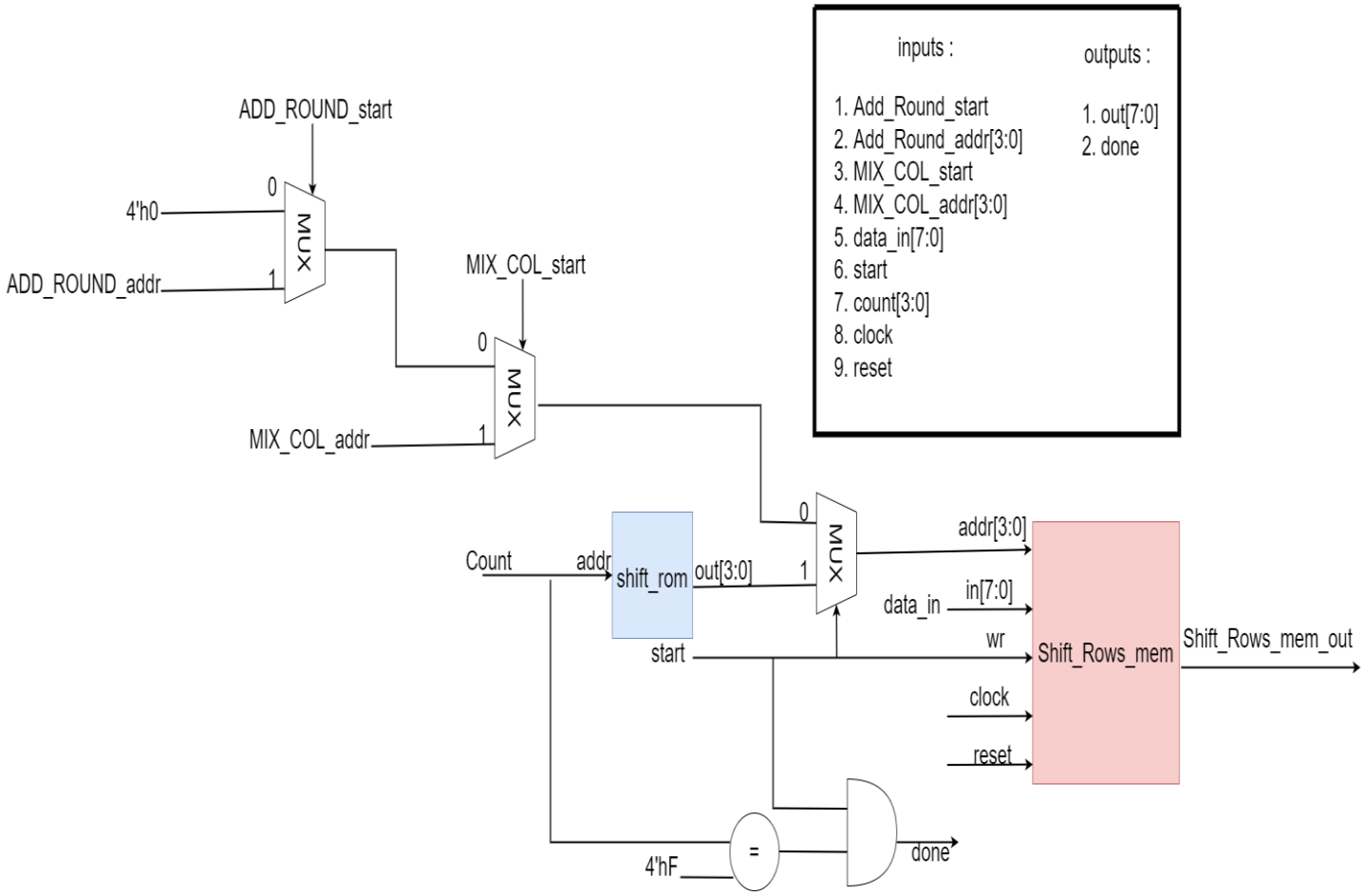


As shown in figure, we used two main blocks to build the SubBytes STATE architecture:

- SubBytes memory
- S-BOX Rom

The inputs to the state are stored at the memory at the beginning and then we pass them from the memory to the S-BOX to map them to the required values and then we store them again at the memory, we control these processes using 4-bits counter acts as a controller and the state starts when the START signal comes and then we passes the outputs to the next state.

ShiftRows STATE

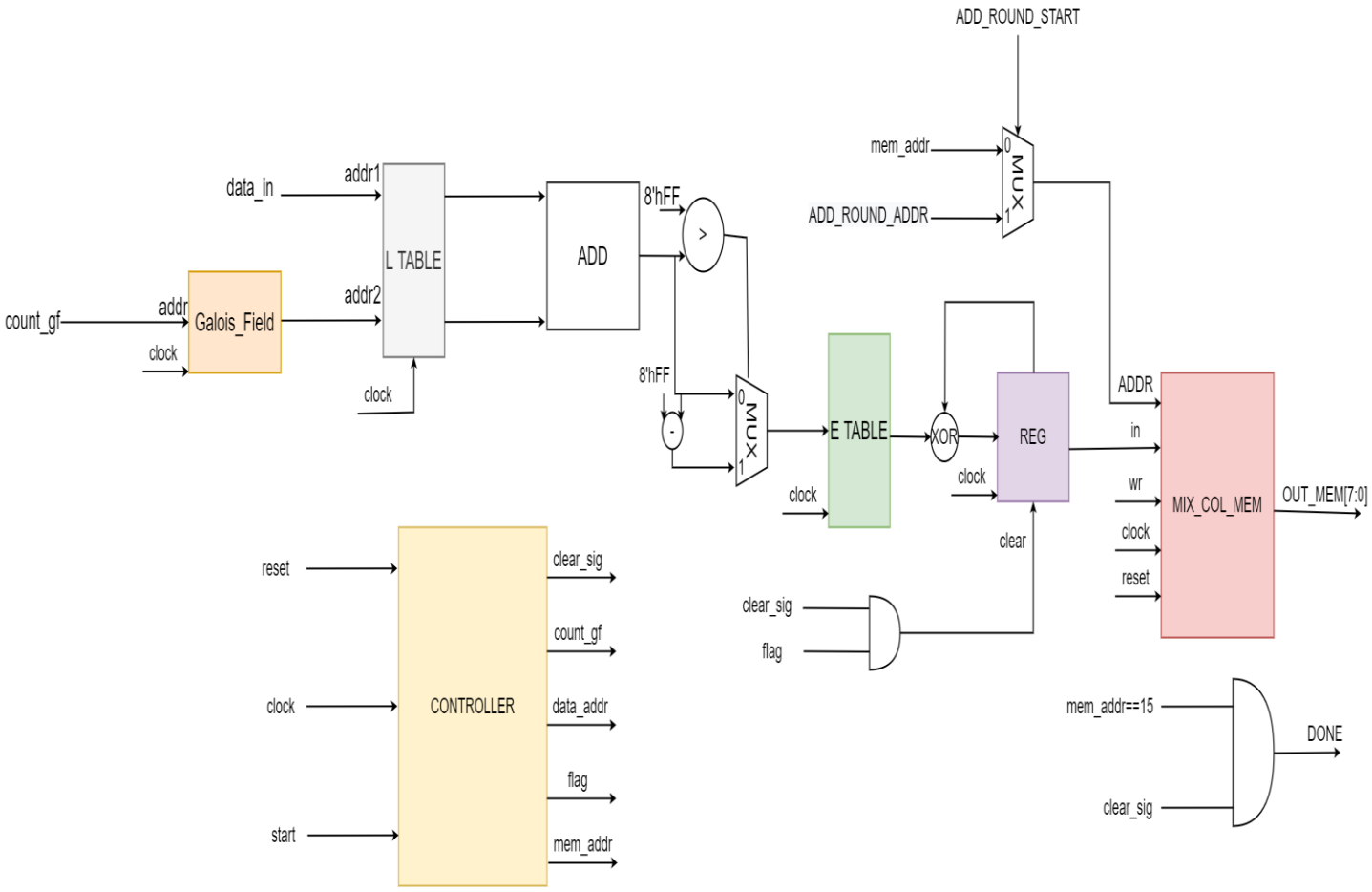


As shown in figure, we used two main blocks to build the ShiftRows STATE architecture:

- Shift_rom : we store the shift values at it.
- Shift_Rows_mem : we store the data after shifting in it.

We control the state using the same controller of the SubBytes STATE to reduce the area, but it makes the logic more complicated. The state starts when START signal comes and when the state processes are done the DONE signal is high to make the next state starts and to send the output to the next state.

MixColumn STATE

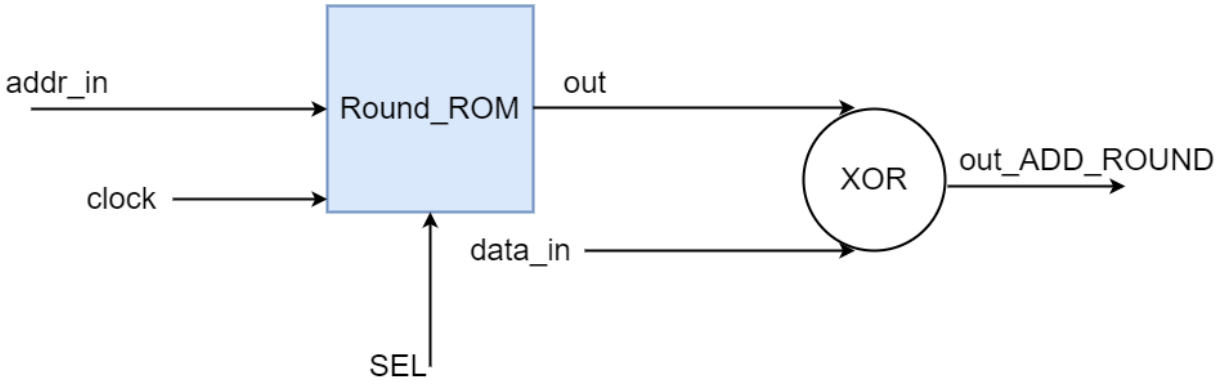


As shown in figure, we used five main blocks to build the MixColumn STATE architecture:

- Galois_Field ROM
- L-TABLE ROM
- E-TABLE ROM
- Register with clear: acts as an accumulator but instead of using adder we use XOR.
- MIX_COL memory

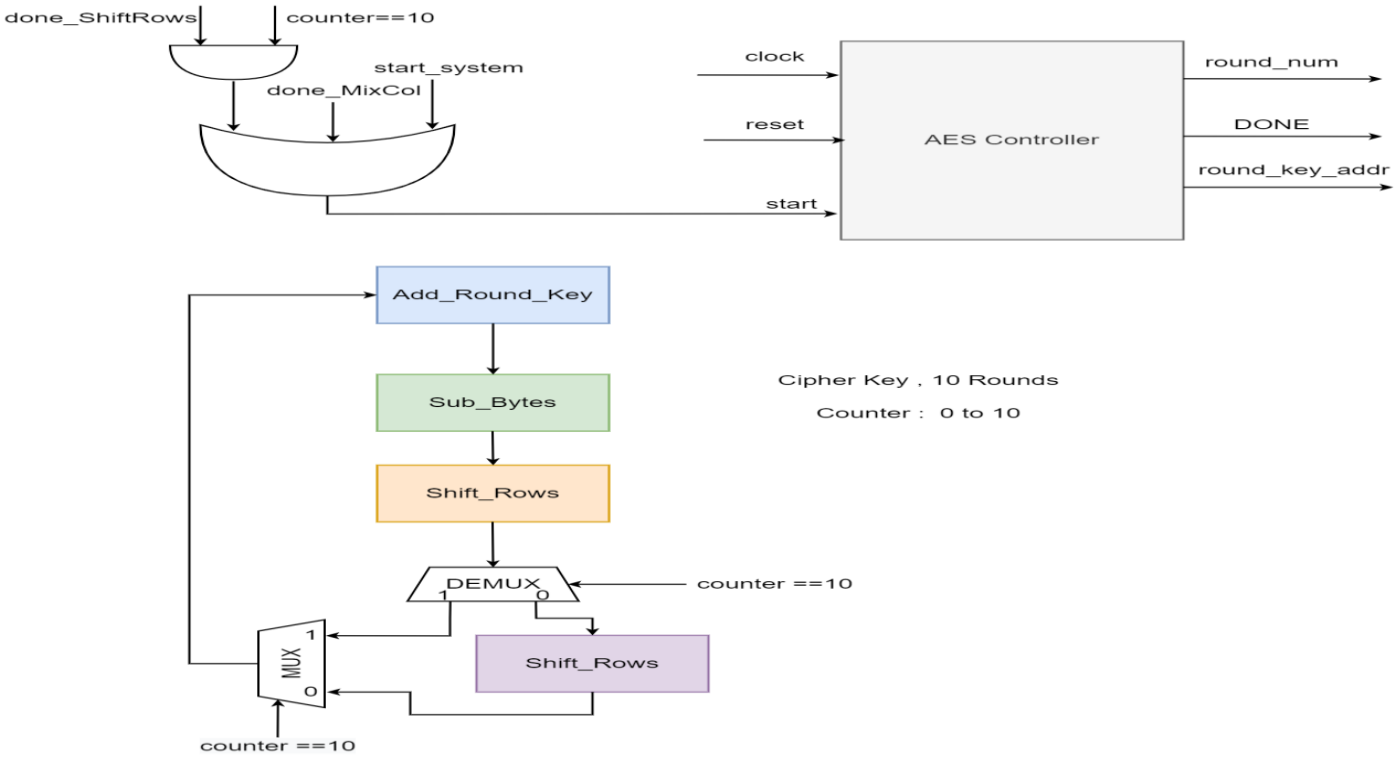
Instead of performing matrix multiplication, we used Galois Field Multiplication which can be done easily using two tables, L-TABLE and E-TABLE as mentioned before. This method has much reduced area and power, but it's more complicated.

AddRoundKey STATE



In this state we make XORing between the ROUNDKEYS and the input data, we have 10 Round ROMS for the 10 Rounds and one Cipher-key Rom for Round 0, So we used 11 ROM, we didn't use shared resources and use one ROM instead of 11 as we may in the future make our target is throughput.

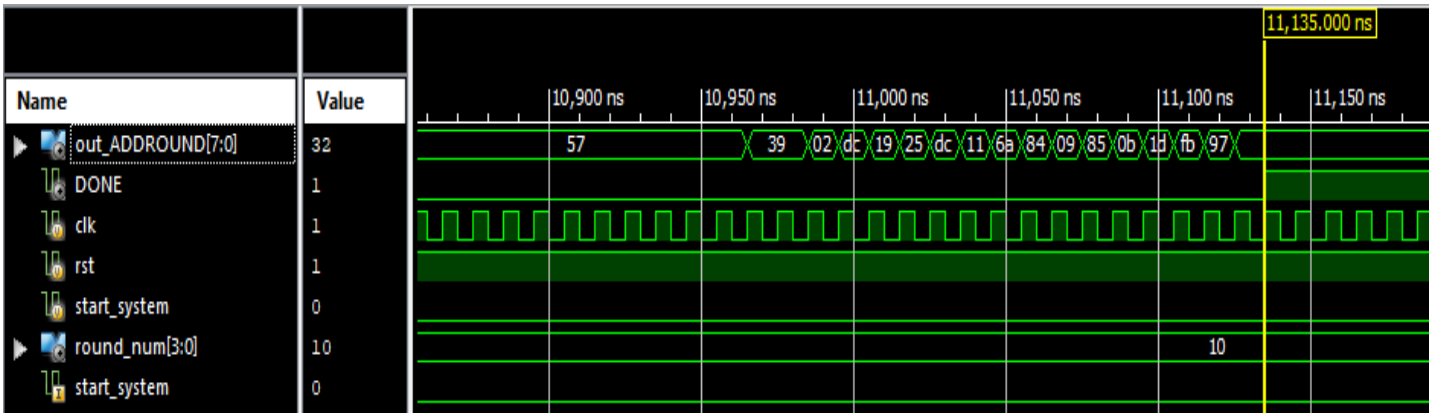
AES STATES



We have 4 main states and 10 rounds so the process is iterative, after the 10 rounds the 128-bit input data are completely encrypted and stored in memory.

Results

Simulation Results



For a frequency of 100MHz all output appear after 11.135 usec which means 11,135 clock cycles.

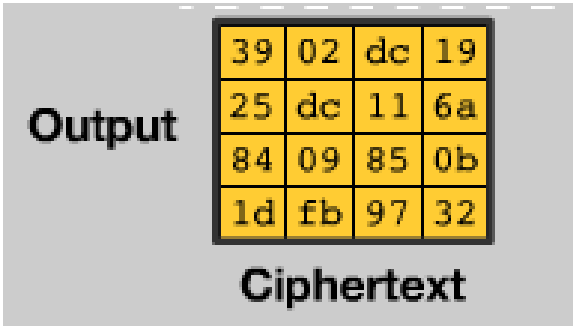
We put the same input as our reference to compare with it the output results:

	Start of round			
Input	32	88	31	e0
	43	5a	31	37
	f6	30	98	07
	a8	8d	a2	34

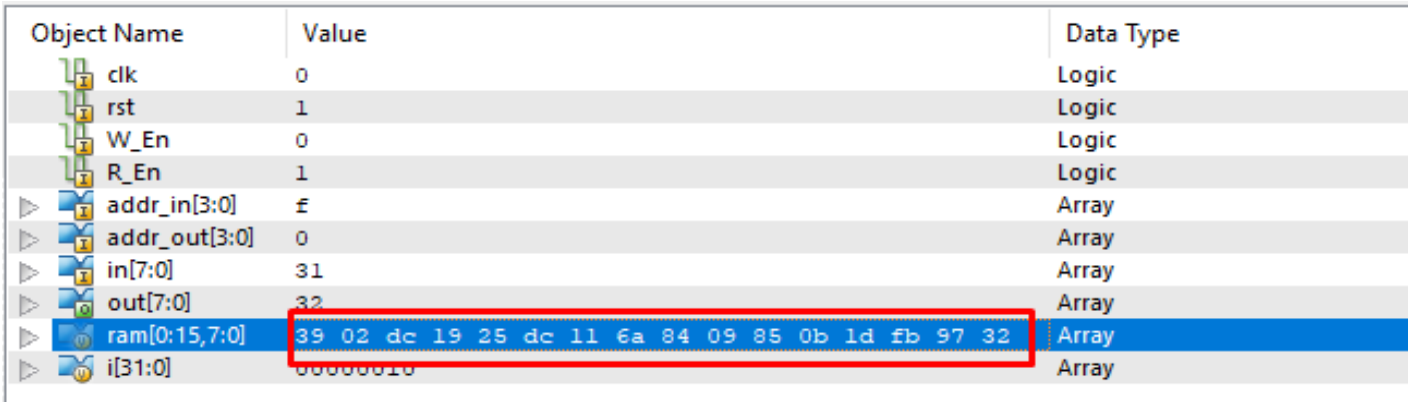
The states output after each state is shown in the following figure:

	Start of round				After SubBytes				After ShiftRows				After MixColumns				Round key			
Input	32	88	31	e0													2b	28	ab	09
	43	5a	31	37													7e	ae	f7	cf
	f6	30	98	07													15	d2	15	4f
	a8	8d	a2	34													16	a6	88	3c
Round 1	19	a0	9a	e9	d4	e0	b8	1e	d4	e0	b8	1e	04	e0	48	28	a0	88	23	2a
	3d	f4	c6	f8	27	bf	b4	41	bf	b4	41	27	66	cb	f8	06	fa	54	a3	6c
	e3	e2	8d	48	11	98	5d	52	5d	52	11	98	81	19	d3	26	fe	2c	39	76
	be	2b	2a	08	ae	f1	e5	30	30	ae	f1	e5	e5	9a	7a	4c	17	b1	39	05
Round 2	a4	68	6b	02	49	45	7f	77	49	45	7f	77	58	1b	db	1b	f2	7a	59	73
	9c	9f	5b	6a	de	db	39	02	db	39	02	de	4d	4b	e7	6b	c2	96	35	59
	7f	35	ea	50	d2	96	87	53	87	53	d2	96	ca	5a	ca	b0	95	b9	80	f6
	f2	2b	43	49	89	f1	1a	3b	3b	89	f1	1a	f1	ac	a8	e5	f2	43	7a	7f
Round 3	aa	61	82	68	ac	ef	13	45	ac	ef	13	45	75	20	53	bb	3d	47	1e	6d
	8f	dd	d2	32	73	c1	b5	23	c1	b5	23	73	ec	0b	c0	25	80	16	23	7a
	5f	e3	4a	46	cf	11	d6	5a	d6	5a	cf	11	09	63	cf	d0	47	fe	7e	88
	03	ef	d2	9a	7b	df	b5	b8	b8	7b	df	b5	93	33	7c	dc	7d	3e	44	3b
Round 4	48	67	4d	d6	52	85	e3	f6	52	85	e3	f6	0f	60	6f	5e	ef	a8	b6	db
	6c	1d	e3	5f	50	a4	11	cf	a4	11	cf	50	d6	31	c0	b3	44	52	71	0b
	4e	9d	b1	58	2f	5e	c8	6a	c8	6a	2f	5e	da	38	10	13	a5	5b	25	ad
	ee	0d	38	e7	28	d7	07	94	94	28	d7	07	a9	bf	6b	01	41	7f	3b	00
Round 5	e0	c8	d9	85	e1	e8	35	97	e1	e8	35	97	25	bd	b6	4c	d4	7c	ca	11
	92	63	b1	b8	4f	fb	c8	6c	fb	c8	6c	4f	d1	11	3a	4c	d1	83	f2	f9
	7f	63	35	be	d2	fb	96	ae	96	ae	d2	fb	a9	d1	33	c0	c6	9d	b8	15
	e8	c0	50	01	9b	ba	53	7c	7c	9b	ba	53	ad	68	8e	b0	f8	87	bc	bc

And the outputs should be as the following figure:



And that's the same output from the wave form. Also it's stored at the output memory correctly as shown in the next figure.



Resources Results

For Virtex-7 using ISE Design SUITE 14.7:

The used building blocks are shown in the following figure:

```
HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 16
 16x4-bit single-port Read Only RAM                   : 1
 16x8-bit single-port Read Only RAM                   : 12
 256x8-bit dual-port Read Only RAM                    : 1
 256x8-bit single-port Read Only RAM                  : 2
# Adders/Subtractors                                  : 10
 2-bit adder                                          : 2
 4-bit adder                                          : 6
 8-bit subtractor                                     : 1
 9-bit adder                                          : 1
# Registers                                           : 63
 1-bit register                                       : 22
 128-bit register                                     : 3
 2-bit register                                       : 2
 4-bit register                                       : 16
 8-bit register                                       : 20
# Comparators                                         : 1
 9-bit comparator greater                             : 1
# Multiplexers                                        : 70
 1-bit 2-to-1 multiplexer                             : 1
 4-bit 2-to-1 multiplexer                             : 11
 8-bit 13-to-1 multiplexer                            : 1
 8-bit 16-to-1 multiplexer                            : 3
 8-bit 2-to-1 multiplexer                             : 54
# Xors                                                : 2
 8-bit xor2                                           : 2
```

The total resources used for Virtex-7 FPGA:

```
Selected Device : 7vx690tffg1761-2


Slice Logic Utilization:
 Number of Slice Registers:              507  out of  866400     0%
 Number of Slice LUTs:                   762  out of  433200     0%
    Number used as Logic:                762  out of  433200     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     822
   Number with an unused Flip Flop:      315  out of     822    38%
   Number with an unused LUT:             60  out of     822     7%
   Number of fully used LUT-FF pairs:    447  out of     822    54%
   Number of unique control sets:         13

IO Utilization:
 Number of IOs:                          20
 Number of bonded IOBs:                  20  out of     850     2%
    IOB Flip Flops/Latches:               9

Specific Feature Utilization:
 Number of Block RAM/FIFO:                7  out of    1470     0%
    Number using Block RAM only:          7
 Number of BUFG/BUFGCTRLs:                1  out of      32     3%
```

We used only 762 LUTs, 822 LUT-FF pairs and 7 BRAMS. We can use distributed ram instead of BRAM and that will reduce the overhead area.

For DE0-CV using Quartus Prime 21.1:

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Aug 16 20:36:51 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	aes
Top-level Entity Name	AES
Family	Cyclone V
Device	5CEBA4F23C7
Timing Models	Final
Logic utilization (in ALMs)	355 / 18,480 (2 %)
Total registers	636
Total pins	20 / 224 (9 %)
Total virtual pins	0
Total block memory bits	4,096 / 3,153,920 (< 1 %)
Total DSP Blocks	0 / 66 (0 %)

We used only 355 LUT, 636 register and 4.096 BRAM bits which means that the resources on DE0-CV is much less than the resources on Virtex-7.

Power Results

For DE0-CV using Quartus Prime 21.1:

Power Analyzer Status	Successful - Tue Aug 16 20:42:00 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	aes
Top-level Entity Name	AES
Family	Cyclone V
Device	5CEBA4F23C7
Power Models	Final
Total Thermal Power Dissipation	197.73 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	193.90 mW
I/O Thermal Power Dissipation	3.83 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

The total dissipated power is approximately 3.83 mW and that’s due to using less hardware resources.

Timing Information

For Virtex-7 using ISE Design SUITE 14.7:

```
Timing Summary:
---------------
Speed Grade: -2

   Minimum period: 7.002ns (Maximum Frequency: 142.807MHz)
   Minimum input arrival time before clock: 1.572ns
   Maximum output required time after clock: 3.281ns
   Maximum combinational path delay: No path found
```

The maximum frequency we can use is 142.807 MHz.

For DE0-CV using Quartus Prime 21.1:

Slow model:

Slow 1100mV 85C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	141.12 MHz	141.12 MHz	clk	

The maximum frequency we can use is 141.12 MHz.

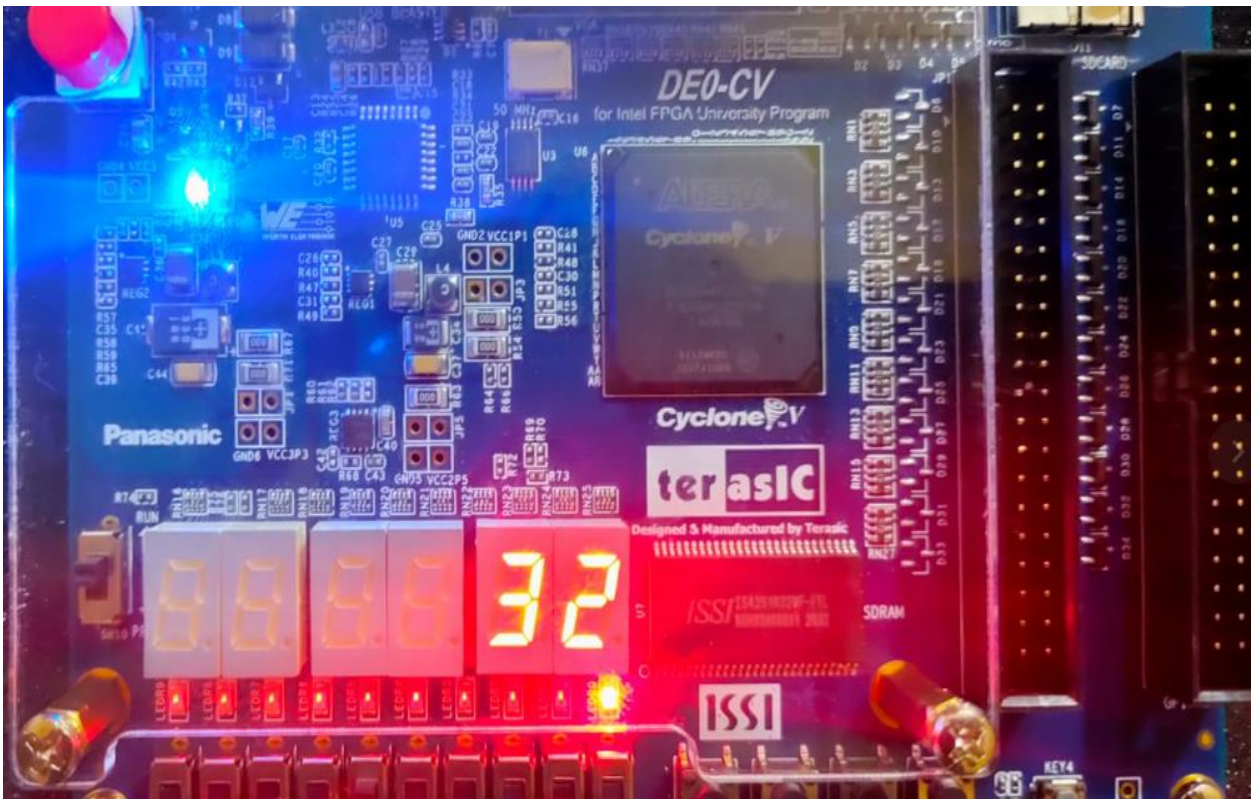
Fast model:

Fast 1100mV 0C Model

	Fmax	Restricted Fmax	Clock Name	Note
1	320.31 MHz	315.06 MHz	clk	limit due to minimum period restriction (tmin)

The maximum frequency we can use is 315 MHz.

FPGA Results using DE0-CV



We used the first two 7-Segments to display the outputs by using BCD decoder and 7segment leds decoder, and the first LEDR to display the DONE flag.

Tools used

For ***Virtex-7*** FPGA on VC-709 Board: we used *ISE DESIGN SUITE14.7* and *VIVADO2018.3* to make the synthesis, simulation and get all the needed results.

For ***DE0-CV*** FPGA : we used *Quartus Prime21.1* to complete the FPGA flow and *QuestaSim* to check the functionality by making the behavioral simulation.