B. TECH. PROJECT REPORT

On

An Ultra-Low Power AES Architecture for IoT

BY

Mohammad Fayaz Ahmed

DISCIPLINE OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

December 2018

An Ultra-Low Power AES Architecture for IoT

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the degrees

of
BACHELOR OF TECHNOLOGY
in
ELECTRICAL ENGINEERING

Submitted by:
Mohammad Fayaz Ahmed

Guided by:
Dr. Santosh Kumar Vishvakarma,
Assistant Professor,
Electrical Engineering,
Indian Institute of Technology Indore

INDIAN INSTITUTE OF TECHNOLOGY INDORE
December 2018

CANDIDATE'S DECLARATION

I hereby declare that the project entitled **"An Ultra-Low Power AES Architecture for IoT "** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in **Electrical Engineering** completed under the supervision of **Dr. Santosh Kumar Vishvakarma, Assistant Professor, Electrical Engineering, IIT Indore** is an authentic work.

Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

Mohammad Fayaz Ahmed
1500002021
Discipline of Electrical Engineering
Indian Institute of Technology Indore

CERTIFICATE by BTP Guide

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Dr. Santosh Kumar Vishvakarma,
Assistant Professor,
Discipline of Electrical Engineering,
Indian Institute of Technology Indore

Preface

This report on “ **An Ultra-Low Power AES Architecture for IoT** " is prepared under the guidance of Dr. Santosh Kumar Vishvakarma, Assistant Professor, Electrical Engineering, IIT Indore

Throughout this report, detailed description of the technologies that have been used to design and implement the low power AES module is provided. The implemented low power AES module is tested for its different inputs and results are presented in a clear and concise manner. I have tried to the best of my ability and knowledge to explain the content in a lucid manner. I have also added figures to make it more illustrative.

Acknowledgements

I would like to thank my B.Tech Project supervisor **Dr. Santosh Kumar Vishvakarma** for his constant support in structuring the project and for his valuable feedback throughout the course of this project. He gave me an opportunity to discover and work in such an interesting domain. His guidance proved really valuable in all the difficulties I faced in the course of this project.

I am really grateful to **Mr. Sajid Khan** who also provided valuable guidance and helped with the problems while working on various technologies. He provided initial pathway for starting the project in the right manner and provided useful directions to proceed along whenever necessary.

I am also thankful to all my family members, friends and colleagues who were a constant source of motivation. I am really grateful to Dept. of Electrical Engineering, IIT Indore for providing with the necessary hardware utilities to complete the project. I offer sincere thanks to everyone who else knowingly or unknowingly helped me to complete this project.

Mohammad Fayaz Ahmed

1500002021

Discipline of Electrical Engineering

Indian Institute of Technology Indore

Abstract

Internet of Things (IoT) is now became a part of our life. Many devices are already connected and more are expected to be deployed in next coming years. To provide a practical solution for security, privacy and trust is the main concern for IoT. To provide security in IoT, cryptography is needed. AES algorithm is a well known, highly secure and symmetric key algorithm.

In this project we have presented an ultra-low power AES architecture for IoT applications. The proposed architecture has been implemented on SCL 180 nm technology. I have used 4-bit SerDes (serializer and deserializer) to send and receive 128-bit data. The designed AES architecture uses a 32-bit data path in SubByte transformation, it requires 44 clock cycles for encryption of 128-bit plaintext with 128 bit cipher key. To deserialize 128- bit plaintext and cipher key our architecture requires 32 clock cycles and same to serialize 128-bit cipher text and these 32 clock cycles are overlapped by 44 clock cycles required by AES module, hence once after the first 32 clock cycles, the use of SerDes does not affect the throughput of system.

The goal is to present an ultra-low power AES architecture for IoT applications.

Table of contents

List of Figures

List of Tables

Chapter 1: Introduction

This chapter highlights the background and motivation for the project. The problem statement has been described of the project and the importance of the results is also clearly portrayed. Towards the end, the objectives are briefly outlined and the future scope is also discussed.

1.1 Background

Internet of things (IoT) refers to the ever-growing network and is expected to use in variety of emerging applications, such as, smart cities, wearable electronics, remote health care, agriculture, and many more.

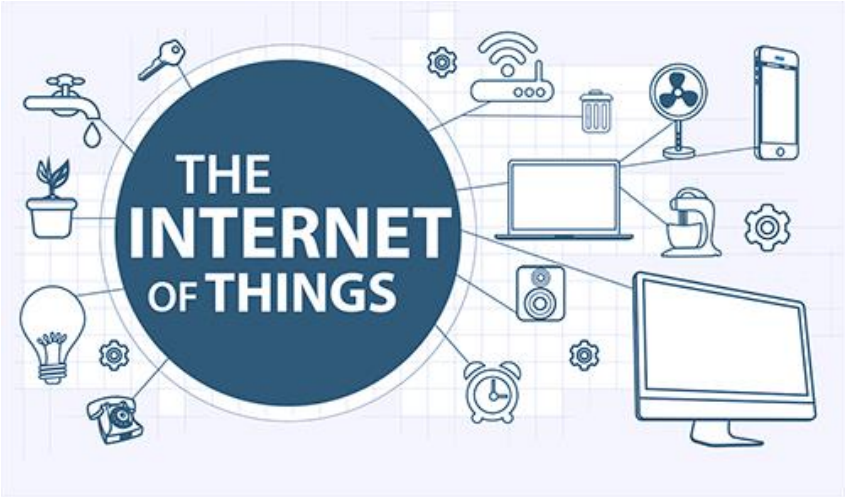

Fig. 1.1 Internet of Things

This adds a level of intelligence to the devices, enabling them to communicate even without the involvement of a human being. It utilizes real-time analysis with machine learning method to process data and make decisions. Security failure of these devices can affect billions of lives as well as huge financial loss with privacy invasion. Thus, avoiding security failure is considered as a serious issue in the effective and meaningful deployment of IoT devices on a large scale. To secure the sensitive data, cryptography provides an efficient solution. However, the hardware cost and power budget of conventional cryptography implementations limits the use of conventional cryptography algorithm in IoT.

In 2001, AES(Advanced Encryption Standard) was formally approved as a US federal standard. The Current issue on security of the IoT's can be solved by using the available AES cipher algorithm. It is considered as a strong and secure cipher. But majority of the IoT devices are battery powered.

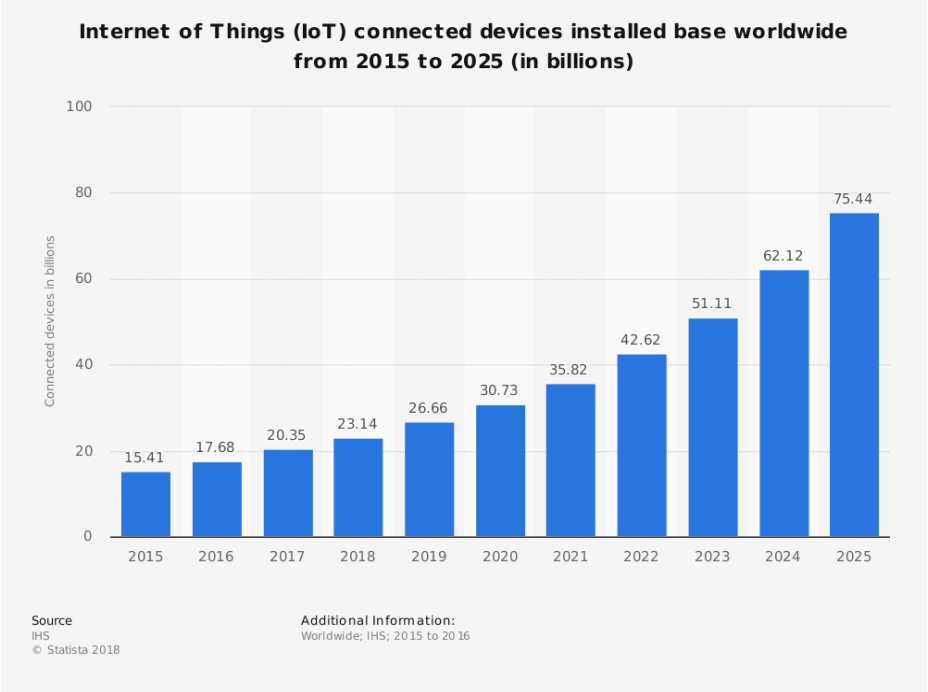

Fig. 1.2 IoT's growth from past, present and expected growth in the coming future

Cryptography algorithms consume notable amount of computing resources, such as CPU time and memory, that makes battery power draining fast and that is the problem for mobile devices. Thus, we need a way to reduce power usage by cryptography algorithms.

1.2 Advantages and disadvantages of using IoT

Advantages

Here are some advantages of IoT:

1. **Data:** The more the information, the easier it is to make the right decision. Knowing what to get from the grocery while you are out, without having to check on your own, not only saves time but is convenient as well.

2. **Tracking:** The computers keep a track both on the quality and the viability of things at home. Knowing the expiration date of products before one consumes them improves safety and quality of life. Also, you will never run out of anything when you need it at the last moment.

3. **Time:** The amount of time saved in monitoring and the number of trips done otherwise would be tremendous.

4. **Money:** The financial aspect is the best advantage. This technology could replace humans who are in charge of monitoring and maintaining supplies.

Disadvantages

Here are some disadvantages of IoT:

1. **Compatibility:** As of now, there is no standard for tagging and monitoring with sensors. A uniform concept like the USB or Bluetooth is required which should not be that difficult to do.

2. **Complexity:** There are several opportunities for failure with complex systems. For example, both you and your spouse may receive messages that the milk is over and both of you may end up buying the same. That leaves you with double the quantity required. Or there is a software bug causing the printer to order ink multiple times when it requires a single cartridge.

3. **Privacy/Security:** Privacy is a big issue with IoT. All the data must be encrypted so that data about your financial status or how much milk you consume isn't common knowledge at the work place or with your friends.

4. **Safety:** There is a chance that the software can be hacked and your personal information misused. The possibilities are endless. Your prescription being changed or your account details being hacked could put you at risk. Hence, all the safety risks become the consumer's responsibility.

1.3 Motivation of the work

Every day, often without us even being aware of it, encryption keeps our personal data private and secure. Encryption is a vault that secures our personal information that is held by businesses and government agencies. But at the same time, encryption should not create any design and cost overheads. So trading off between these two and coming out with the optimized solution is the main motivation for this project.

Chapter 2: About AES Algorithm

2.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is the most widely used symmetric cipher today. AES is a symmetric algorithm because it uses the same key for encryption and decryption both. Even though the term “Standard” in its name only refers to US government applications, the AES block cipher is also mandatory in several industry standards and is used in many commercial systems. Among the commercial standards that include AES are the Internet security standard IPsec, TLS, the Wi-Fi encryption standard IEEE 802.11i, the secure shell network protocol SSH (Secure Shell), the Internet phone Skype and numerous security products around the world. To date, there are no attacks better than brute-force known against AES.

In this chapter, AES internal structure is discussed.
The four stages of AES, namely:

1. SubByte
2. ShiftRows
3. MixColumn
4. AddRoundKey

Each round contains this 4 steps. Before the first round, AddRoundKey is executed, and in the last round, MixColumn is dropped.

Depending on the Key size the number of rounds for AES are classified into three :

1. AES-128: 128 bit key, 10 rounds
2. AES-192: 192 bit key, 12 rounds
3. AES-256: 256 bit key, 14 rounds

The transformation of plaintext to ciphertext is clearly shown in figure 2.1.

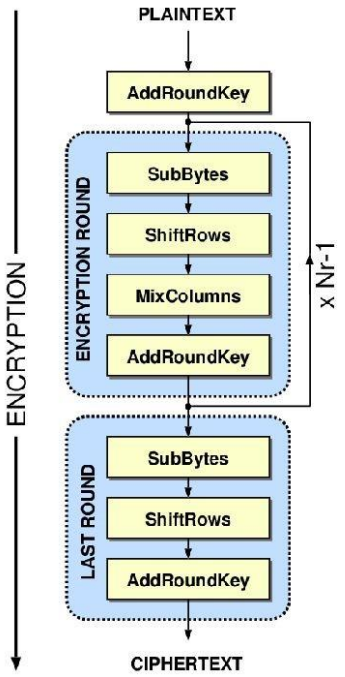

Fig. 2.1 Standard AES algorithm

2.2 Internal Structure of AES

In the following, we examine the internal structure of AES. Figure 2.2 shows the graph of a single AES round. The 16-byte input $A_0, \ldots, A_{15}$ is fed byte-wise into the S-Box. The 16-byte output $B_0, \ldots, B_{15}$ is permuted byte-wise in the ShiftRows layer and mixed by the MixColumn transformation c(x). Finally, the 128-bit subkey k_i is XORed with the intermediate result. We note that AES is a byte-oriented cipher.

This is in contrast to DES, which makes heavy use of bit permutation and can thus be considered to have a bit-oriented structure. In order to understand how the data moves through AES, we first imagine that the state A (i.e., the 128-bit data path) consisting of 16 bytes $A_0, A_1, \ldots, A_{15}$ is arranged in a four-by-four byte matrix:

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

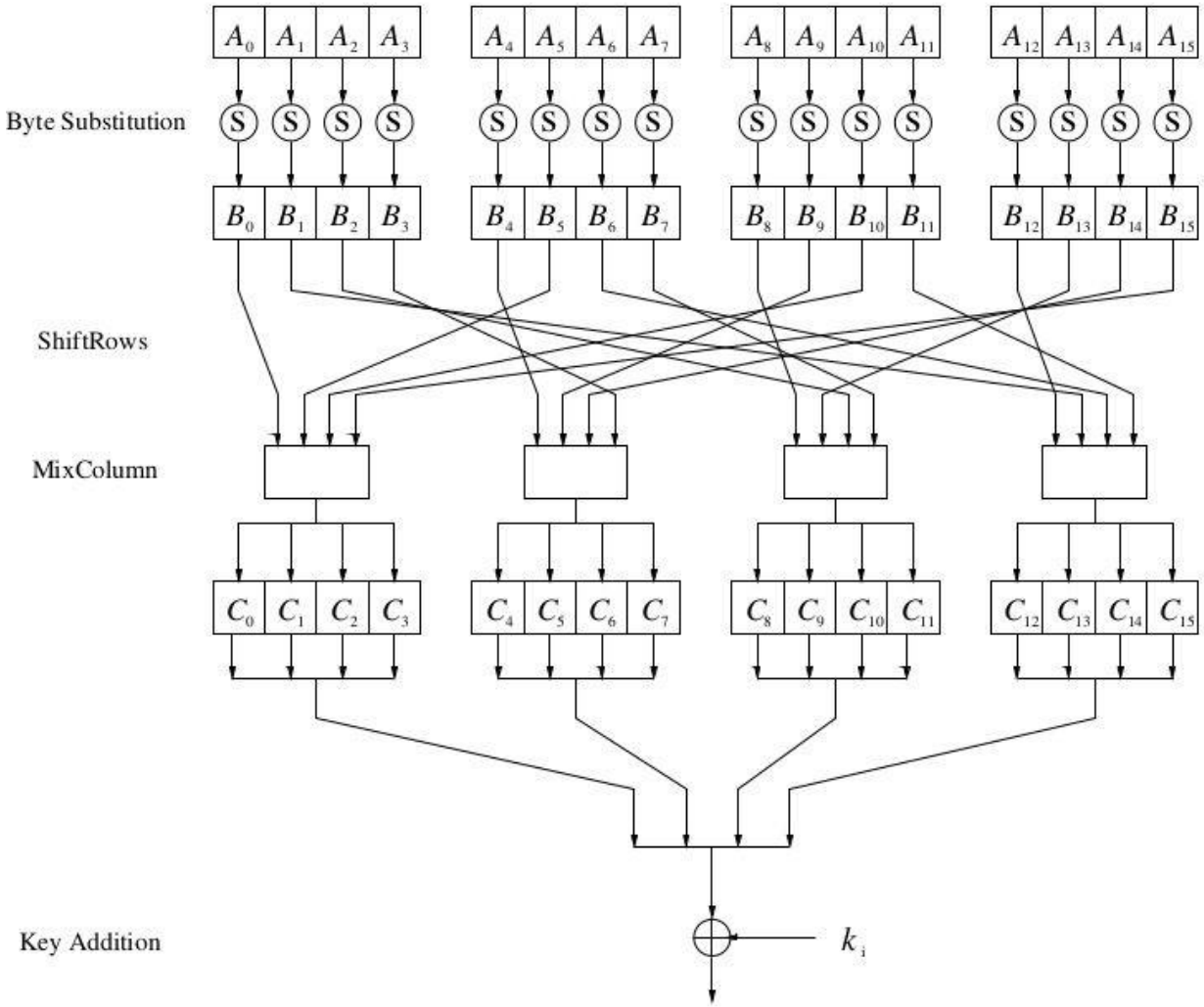

Fig. 2.2 AES round function for rounds 1, 2, . . ., n_{r-1}

As we will see in the following, AES operates on elements, columns or rows of the current state matrix. Similarly, the key bytes are arranged into a matrix with four rows and four (128-bit key), six (192-bit key) or eight (256-bit key) columns. Here is, as an example, the state matrix of a 192-bit key:

k_0	k_4	k_8	k_{12}	k_{16}	k_{20}
k_1	k_5	k_9	k_{13}	k_{17}	k_{21}
k_2	k_6	k_{10}	k_{14}	k_{18}	k_{22}
k_3	k_7	k_{11}	k_{15}	k_{19}	k_{23}

We discuss now what happens in each of the steps.

2.2.1 The SubByte Step

As shown in Fig. 2.3, the first layer in each round is the Byte Substitution layer. The Byte Substitution layer can be viewed as a row of 16 parallel S-Boxes, each with 8 input and output bits. Note that all 16 S-Boxes are identical, unlike DES where eight different S-Boxes are used. In the layer, each state byte A_i is replaced, i.e., substituted, by another byte B_i :

$$S(A_i) = B_i .$$

The S-Box is the only nonlinear element of AES,

ByteSub(A) + ByteSub(B) ≠ ByteSub(A + B) for two states A and B.

The S-Box substitution is a bijective mapping, i.e., each of the $2^8 = 256$ possible input elements is one-to-one mapped to one output element. This allows us to uniquely reverse the S-Box, which is needed for decryption. In software implementations the S-Box is usually realized as a 256-by-8 bit lookup table with fixed entries, as given in Table 2.3.

	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

Fig. 2.3 AES S-Box: Substitution values in hexadecimal notation for input byte

2.2.2 The ShiftRows Step

The ShiftRows transformation cyclically shifts the second row of the state matrix by three bytes to the right, the third row by two bytes to the right and the fourth row by one byte to the right. The first row is not changed by the ShiftRows transformation. The purpose of the ShiftRows transformation is to increase

the diffusion properties of AES. If the input of the ShiftRows sublayer is given as a state matrix B = (B_0, B_1, . . . , B_{15}):

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

Fig. 2.4 Before ShiftRows transformation

The output is the new state:

B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	⟵ one position left shift
B_{10}	B_{14}	B_2	B_6	⟵ two positions left shift
B_{15}	B_3	B_7	B_{11}	⟵ three positions left shift

Fig. 2.5 After ShiftRows transformation

2.2.3 The MixColumn Step

The MixColumn step is a linear transformation which mixes each column of the state matrix. Since every input byte influences four output bytes, the MixColumn operation is the major diffusion element in AES. The combination of the ShiftRows and MixColumn layer makes it possible that after only three rounds every byte of the state matrix depends on all 16 plaintext bytes. In the following, we denote the 16-byte input state by B and the 16-byte output state by C: MixColumn(B) = C, where B is the state after the ShiftRows operation as given in figure 2.5. Now, each 4-byte column is considered as a vector and multiplied by a fixed 4 × 4 matrix. The matrix contains constant entries. Multiplication and addition of the coefficients is done in GF(2^8). As an example, we show how the first four output bytes are computed:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

The second column of output bytes (C_4, C_5, C_6, C_7) is computed by multiplying the four input bytes (B_4, B_9, B_{14}, B_3) by the same constant matrix, and so on. Figure 2.3 shows which input bytes are used in each of the four MixColumn operations. We discuss now the details of the vector–matrix multiplication which forms the MixColumn operations. We recall that each state byte C_i and B_i is an 8-bit value representing an element from GF(2^8). All arithmetic involving the coefficients is done in this Galois field. For the constants in the matrix a hexadecimal notation is used: "01" refers to the GF(2^8) polynomial with the coefficients (0000 0001), i.e., it is the element 1 of the Galois field; "02" refers to the polynomial with the bit vector (0000 0010), i.e., to the polynomial x; and "03" refers to the polynomial with the bit vector (0000 0011), i.e., the Galois field element x + 1.

The additions in the vector–matrix multiplication are GF(2^8) additions, that is simple bitwise XORs of the respective bytes. For the multiplication of the constants, we have to realize multiplications with the constants 01, 02 and 03. These are quite efficient, and in fact, the three constants were chosen such that software implementation is easy. Multiplication by 01 is multiplication by the identity and does not involve any explicit operation. Multiplication by 02 and 03 can be done through table look-up in two 256-by-8 tables. As an alternative, multiplication by 02 can also be implemented as a multiplication by x, which is a left shift by one bit, and a modular reduction with $P(x) = x^8 + x^4 + x^3 + x + 1$. Similarly, multiplication by 03, which represents the polynomial (x + 1), can be implemented by a left shift by one bit and addition of the original value followed by a modular reduction with P(x).

2.2.4 The AddRoundKey Step

The two inputs to the Key Addition layer are the current 16-byte state matrix and a subkey which also consists of 16 bytes (128 bits). The two inputs are combined through a bitwise XOR operation. Note that the XOR operation is equal to addition in the Galois field GF(2).

2.2.5 The Key Schedule

The key schedule takes the original input key (of length 128, 192 or 256 bit) and derives the subkeys used in AES. Note that an XOR addition of a subkey is used both at the input and output of AES. This process is sometimes referred to as key whitening. The number of subkeys is equal to the number of rounds plus one, due to the key needed for key whitening in the first key addition layer, cf. Fig. 2.1. Thus, for the key length of 128 bits, the number of rounds is $n_r = 10$, and there are 11 subkeys, each of 128 bits. The AES with a 192-bit key requires 13 subkeys of length 128 bits, and AES with a 256-bit key has 15 subkeys. The AES subkeys are computed recursively, i.e., in order to derive subkey k_i, subkey k_{i-1} must be known, etc.

The AES key schedule is word-oriented, where 1 word = 32 bits. Subkeys are stored in a key expansion array W that consists of words. There are different key schedules for the three different AES key sizes of 128, 192 and 256 bit, which are all fairly similar. We introduce the 128-Bit key schedules in the following.

Key Schedule for 128-Bit Key AES

The 11 subkeys are stored in a key expansion array with the elements W[0], . . . ,W[43]. The subkeys are computed as depicted in Fig. 2.6. The elements $K_0, \ldots, K_{15}$ denote the bytes of the original AES key.

First, we note that the first subkey k_0 is the original AES key, i.e., the key is copied into the first four elements of the key array W. The other array elements are computed as follows. As can be seen in the figure, the leftmost word of a subkey W [4i], where i = 1, . . . , 10, is computed as:

$$W[4i] = W[4(i-1)] + g(W[4i-1]).$$

Here g( ) is a nonlinear function with a four-byte input and output. The remaining three words of a subkey are computed recursively as:

$$W[4i + j] = W[4i + j - 1] + W[4(i-1) + j],$$

where i = 1, . . . , 10 and j = 1, 2, 3. The function g( ) rotates its four input bytes, performs a byte-wise S-Box substitution, and adds a round coefficient RC to it. The round coefficient is an element of the Galois field GF(2^8), i.e, an 8-bit value. It is only added to the leftmost byte in the function g( ). The round coefficients vary from round to round according to the following rule:

$$RC[1] = x^0 = (0000\ 0001)_2,$$
$$RC[2] = x^1 = (0000\ 0010)_2,$$
$$RC[3] = x^2 = (0000\ 0100)_2,$$
$$\vdots$$
$$RC[10] = x^9 = (0011\ 0110)_2 .$$

The function g( ) has two purposes. First, it adds nonlinearity to the key schedule. Second, it removes symmetry in AES. Both properties are necessary to thwart certain block cipher attacks.

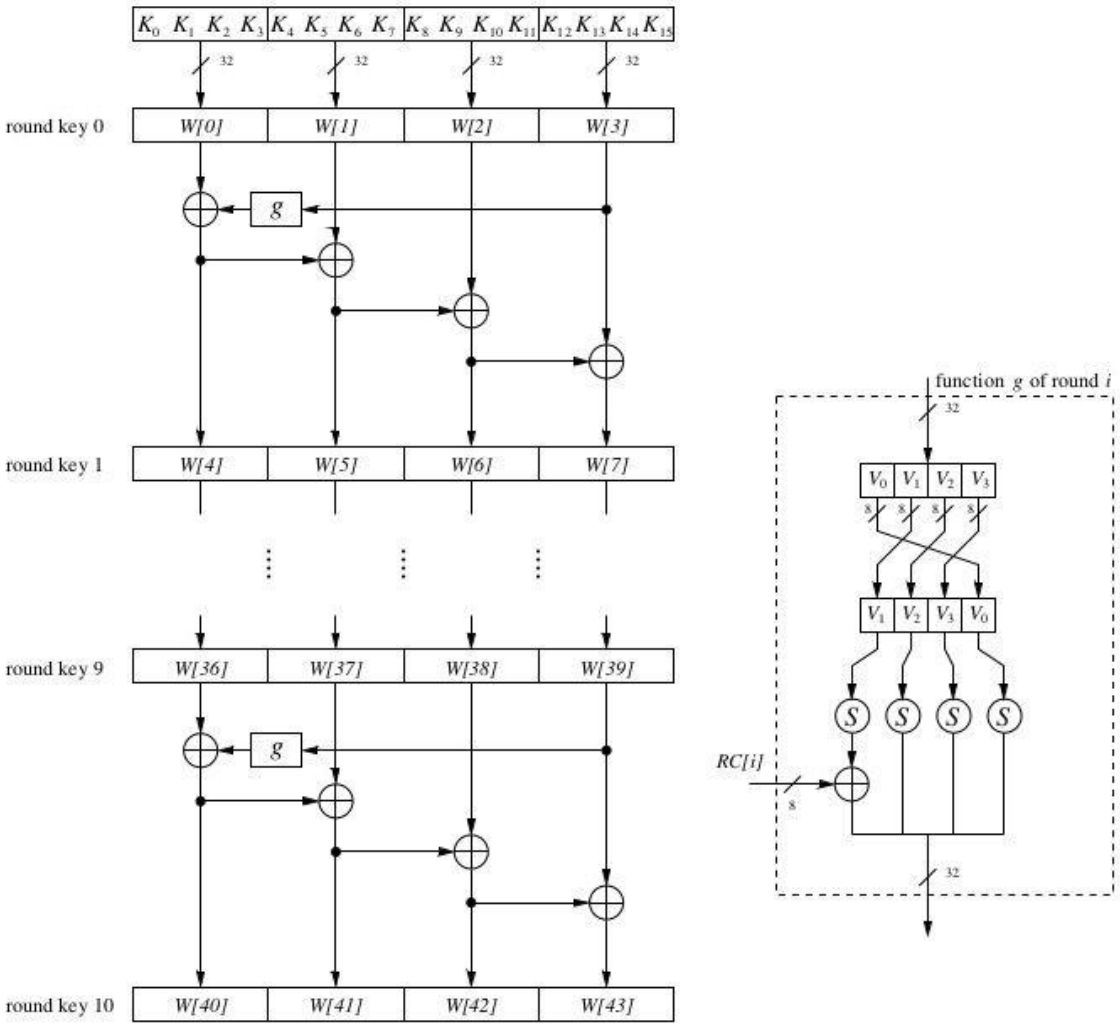

Fig. 2.6 AES key schedule for 128-bit key size

Chapter 3: Design of Module and Tools used

Note: The Results from each tool are discussed in the next Chapter. In this chapter how we used the tool are explained

In this chapter the design of AES IP for IoT constrained processor is explained clearly. In this the power consumption is a major issue, so for power optimization the hardware is reduced. For IoT high throughput is not required hence pipelined and unrolled architectures are not the suitable candidate for IoT applications. We have generated all the keys on-the-fly to prevent hardware redundancy. The architecture shares hardware for different rounds, this reduces the hardware as well as power overhead. As we all know memory is a power hungry circuit, the keys are generated on-the-fly. The AES runs for ten rounds using the same hardware for all the rounds instead of replicating the same hardware multiple times.

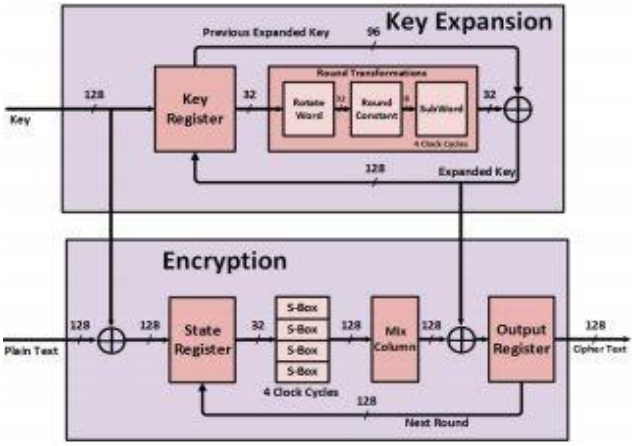

Fig. 3.1 Architecture of our AES implementation.

In the designed AES architecture all the blocks are combinational except the SubByte, the SubByte block is sequential block. The SubByte block uses 32bit data path with four S-Boxes and takes four clock cycles to substitute 128 bit data. After four clock cycles all the 128 bits are passed to ShiftRows blocks. Since data is available after three clock cycles, all the other blocks are disabled for the three clock cycles to prevent unnecessary transitions and power reduction. For further reduction we have not used any circuit for ShiftRows block, ShiftRows block is implemented using wires only.

Since a new round key is required after four clock cycles, the key generation block only consists of a single S-Box instead of four hence the SubWord takes four clock cycles for substitution of a word which

results in a less area and power overhead. To keep the memory usage minimum, all the Round Constants are generated on-the-fly instead of storing them into an LUT.

3.1 Overview of the project

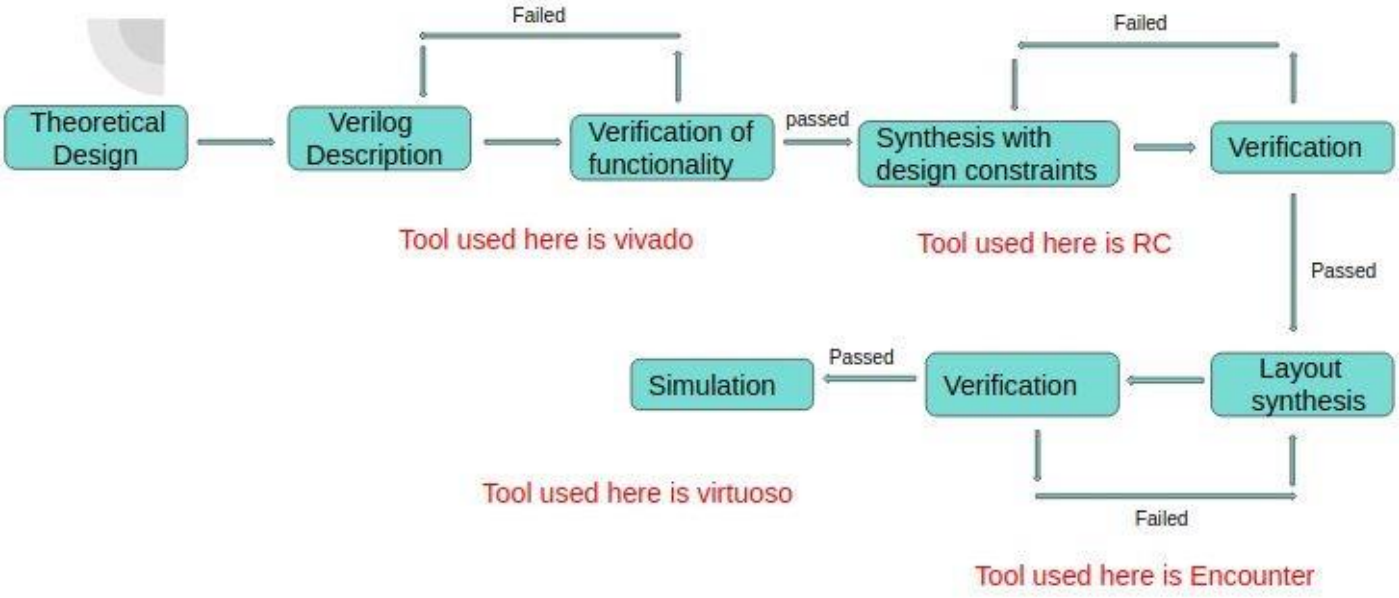

Fig. 3.2 Project Overview

3.2 Design of AES Algorithm

The AES algorithm is designed as explained below.

3.2.1 Serializer and Deserializer

128-Bit input to AES and 128-Bit output from AES at a time makes the chip size large so we used the deserializer for input and serializer for the output.

In the project we used two 4-bit deserializer to receive 128-bit plaintext and cipher key. Here we have used two 128-bit temporary registers to store plaintext and cipher key. Once all the 128-bit received from

plaintext or cipher key serializer AES module start working. Fig. 3.3 shows the architecture of deserializer used for cipher key and plaintext. To overlap the clock cycles of serializer after 12 clock cycles, nextReady signal is asserted and wrapper module is ready to capture new plaintext and data. Similarly once the ciphertext is ready, cipherReady signal is asserted and 128-bit ciphertext is delivered in 32 clock cycles. Hence without affecting the throughput of system, our proposed architecture is able to perform encryption at very low power.

In the design we have used two 4-bit Deserializer to deserialize 128-bit plaintext and cipher key and one serializer to again serialize the 128-bit ciphertext in 32 clock cycles as shown in Fig. 3.3.

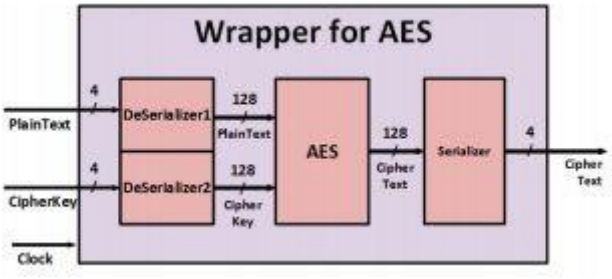

Fig. 3.3 SerDes wrapper for our AES architecture.

3.2.2 SubByte Transformation

SubByte transformation is the only nonlinear transformation in AES. It consists of Substitutions box (S-Box) which maps an eight bit input to an eight bit output and has 256 combinations. These 256 values can either be generated on-the-fly or can be implemented using LUT.

We can do the process using S-Box-on-the-fly approach instead of S-Box using LUT approach

We have implemented S-Box using LUT approach, which takes eight bit input and provide its corresponding eight bit S-Box value. The SubByte block uses 32 bit data path with four S-Boxes and takes four clock cycles to substitute 128 bit data. After four clock cycles all the 128 bits are passed to ShiftRows blocks. Since data is available after three clock cycles, all the other blocks are disabled for the three clock cycles to prevent unnecessary transitions and power reduction.

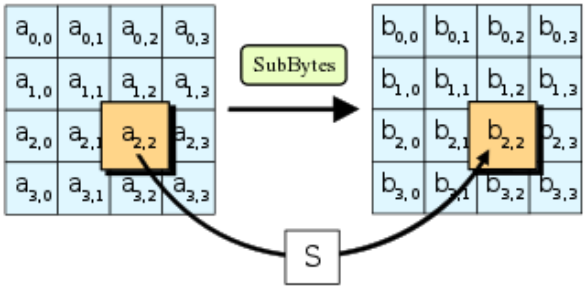

Fig. 3.4 SubByte Transformations

We have used four S-Boxes for SubByte transformation and it requires four clock cycles to substitute all the 128-bit.

3.2.3 ShiftRows Transformation

In ShiftRows transformation each row of state matrix is shifted to left with the offset of 0, 1, 2 and 3 for first, second, third and froth row respectively. In our implementation we have implemented ShiftRows without any circuit, we have used wires only. In this approach we have reduced the power and hardware overhead.

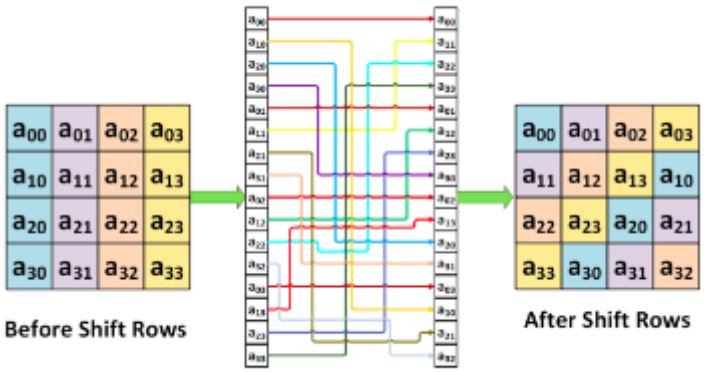

Fig. 3.5 ShiftRows Transformations

3.2.4 MixColumn transformation

MixColumn transformation is linear transformation applied to the columns of state matrix. It is a multiplication of state matrix columns with a polynomial $a(x) = 3x^3 + x^2 + 1x + 2$ using the modulo $(x^4 + 1)$ in $GF(2^8)$. To multiply input by 2 it has been left shifted by 1-bit and then XORed with '1B' (in hexadecimal) if shifting generates a carry. Similarly, to multiply by 3, first it is multiplied by 2 and then XORed with itself.

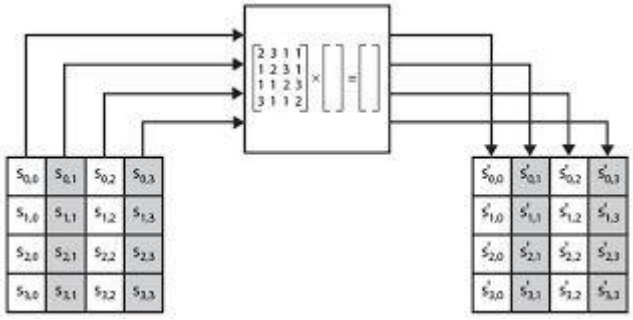

Fig. 3.6 MixColumn Transformations

Since MixColumn transformation is required at the fourth clock cycle of every round, we disabled it for the rest three clock cycles to save power.

3.2.5 Key Expansion

Using Key Expansion all round keys are generated. Key Expansion includes four operations : RotWords, RoundConstant, SubWord and XORing. In SubWord 32-‘bit is substituted using S-Box. Since key is required after four clock cycles, in our architecture Key Expansion generates a new round key after four clock cycles. We have used a single S-Box for substitution of 32-bit in four clock cycles, which results low hardware overhead and low-power.

3.2.6 Xilinx Vivado Tool

AES (Advanced Encryption Standard) algorithm is coded in Verilog. I got few errors and i debugged the code. When we are finally satisfied with the functional description, the VERILOG code is Simulated in **Xilinx Vivado** and we got correct results in Behavioral Simulation and Post-Synthesis Functional Simulation and Post -Synthesis Timing Simulation.

3.3 RC Tool for synthesis

The **Verilog code** is simulated in **Vivado**. So, After implementing the code, it is the time to synthesis. So, for synthesis we used **RC (Cadence tool)** .

As a result, a gate_level netlist of the circuit using the logic library of the circuit manufacture is generated. The gate_level netlist just tells how to connect different standard cells to realize the desired functionality, such that design constraints are met. Now it’s time to move to layout generation.

Before layout generation, we have to make sure that the result after synthesis matches with the original description using **Formality (Synopsys tool)**

3.4 Encounter Tool for Layout Generation

Before layout generation the result after synthesis and the original description should be cross checked. We achieved correct output in all the above stages and now it's time to generate the layout.

For layout generation **Encounter (Cadence tool)** is used.

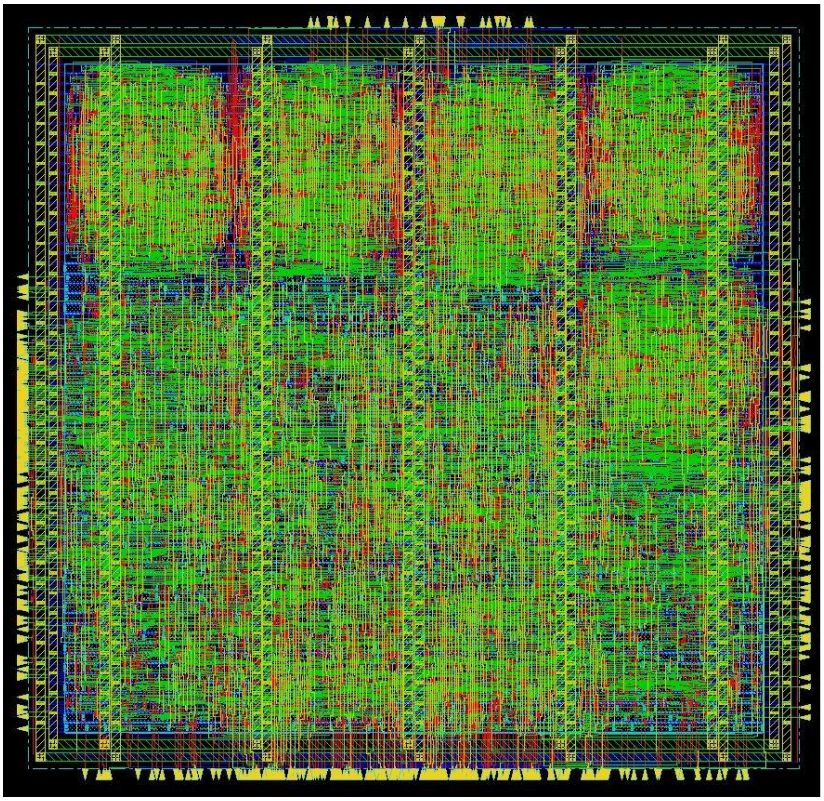

Fig. 3.7 Layout Generation using Encounter Tool

3.5 Virtuoso Tool for Simulation

The layout generation is completed. Now it's time for simulation. I used **Virtuoso (Cadence Tool)** for simulations. To simulate we created a testbench of the chip and connected the circuit with the input and obtained the output. The results which I achieved are correct.

The results are discussed in the next chapter.

Chapter 4: Results

4.1 Result from Xilinx Vivado Tool

Once I am done with the theoretical design it is time to write the Verilog code for AES algorithm. So I wrote the code and we simulated in Xilinx Vivado. I achieved correct results in Behavioral Simulation and Post-Synthesis Functional Simulation and Post -Synthesis Timing Simulation.

NOTE: The Verilog Code is Provided in the Appendix A.

In the below given example the Plaintext and Key are Inputs and the Ciphertext is the Output

Example:

Plaintext in Hex (128 bits): 54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F

Key in Hex (128 bits): 54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75

Ciphertext (128 bits): 29 C3 50 5F 57 14 20 F6 40 22 99 B3 1A 02 D7 3A

NOTE: IN FIGURE BLOCK (PLAINTEXT) AND BLOCK_NEW (CIPHERTEXT)

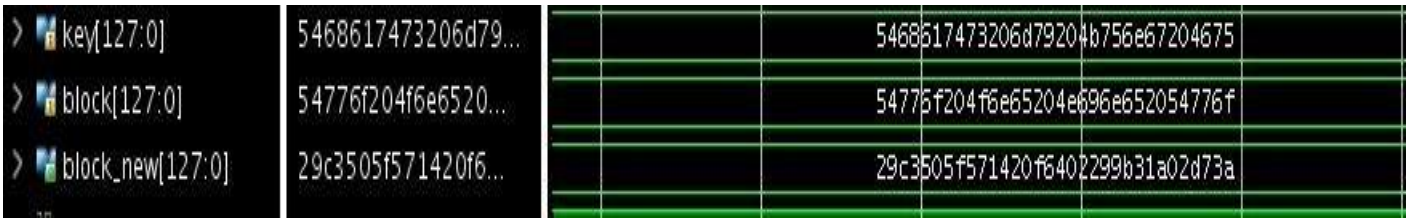

Fig. 4.1 Result from the Xilinx Vivado

4.2 Result from RTL Compiler Tool

After implementing the code, it is the time to synthesize. So, for synthesis we used **RC (Cadence tool).** As a result, a gate level netlist of the circuit using the standard cell library of the SCL PDK is generated. The gate level netlist just tells how different standard cells are connected to realize the desired functionality, such that design constraints are met. Now it's time to proceed to layout generation.

4.3 Result from Encounter Tool

I got correct output in all the above stages and now it's time to generate the layout. For layout generation **Encounter (Cadence tool)** is used.

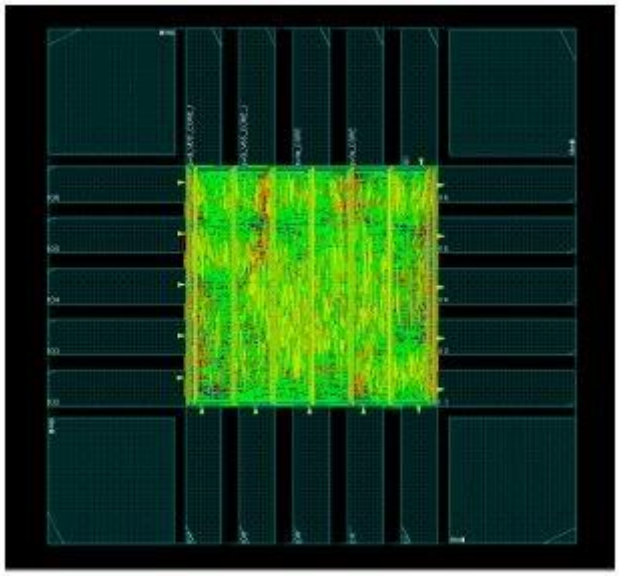

Fig. 4.2 Layout of the Designed Architecture

4.4 Result from Virtuoso Tool

The Virtuoso tool is used for post layout simulation. We created testbench and connected to the circuit.

We gave inputs plaintext and key and observed the output.

In the below given example the Plaintext and Key are Inputs and the Cipher text is the Output

Example:

Plaintext in Hex (128 bits): 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Key in Hex (128 bits): 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Ciphertext (128 bits): 66 e9 4b d4 ef 8a 2c 3b 88 4c fa 59 ca 34 2b 2e

The Plaintext, Key and Ciphertext are in hexadecimal. So, each digit is of four bits. It means "2e" in Ciphertext are the **first eight bits(7-0)** and " 66" in the Ciphertext are the **last eight bits(127-120).** The numbering of bits in the example starts from right to left.

So, in the figure 4.3 we get the output when the ready signal is 1(high). The output of the Ciphertext is from bits 0-7 in the figure 4.3. In the example, the 7-0 bits is “2e”. In binary it is written as 0010 1110. Which is same as in the figure 4.3.

The order of signals in the figure 4.3 is

wave of 7th bit

wave of 6th bit

wave of 5th bit

wave of 4th bit

wave of 3rd bit

wave of 2nd bit

wave of 1st bit

wave of 0th bit

wave of the ready signal

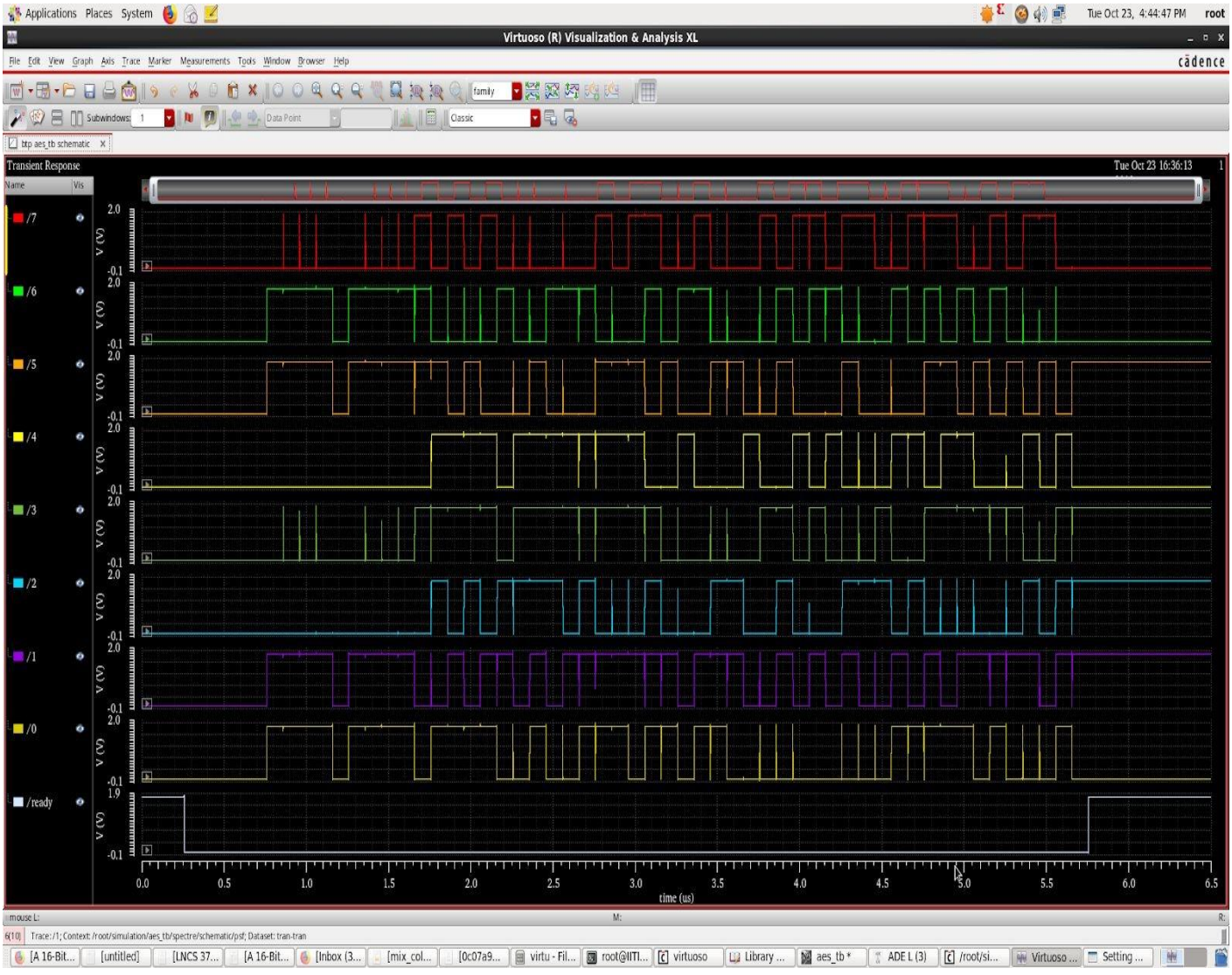

Fig 4.3 Simulation of the Designed Architecture using Virtuoso

So, in the figure 4.4 we get the output when the ready signal is 1(high). The output of the Ciphertext is from bits 120-127 in the figure 4.4. In the example, the 127-0 bits is “66”. In binary it’s written as 0110 0110. Which is same as in the figure 4.4.

The order of signals in the figure is

wave of 127th bit

wave of 126th bit

wave of 125th bit

wave of 124th bit

wave of 123rd bit

wave of 122nd bit

wave of 121st bit

wave of 120th bit

wave of the ready signal

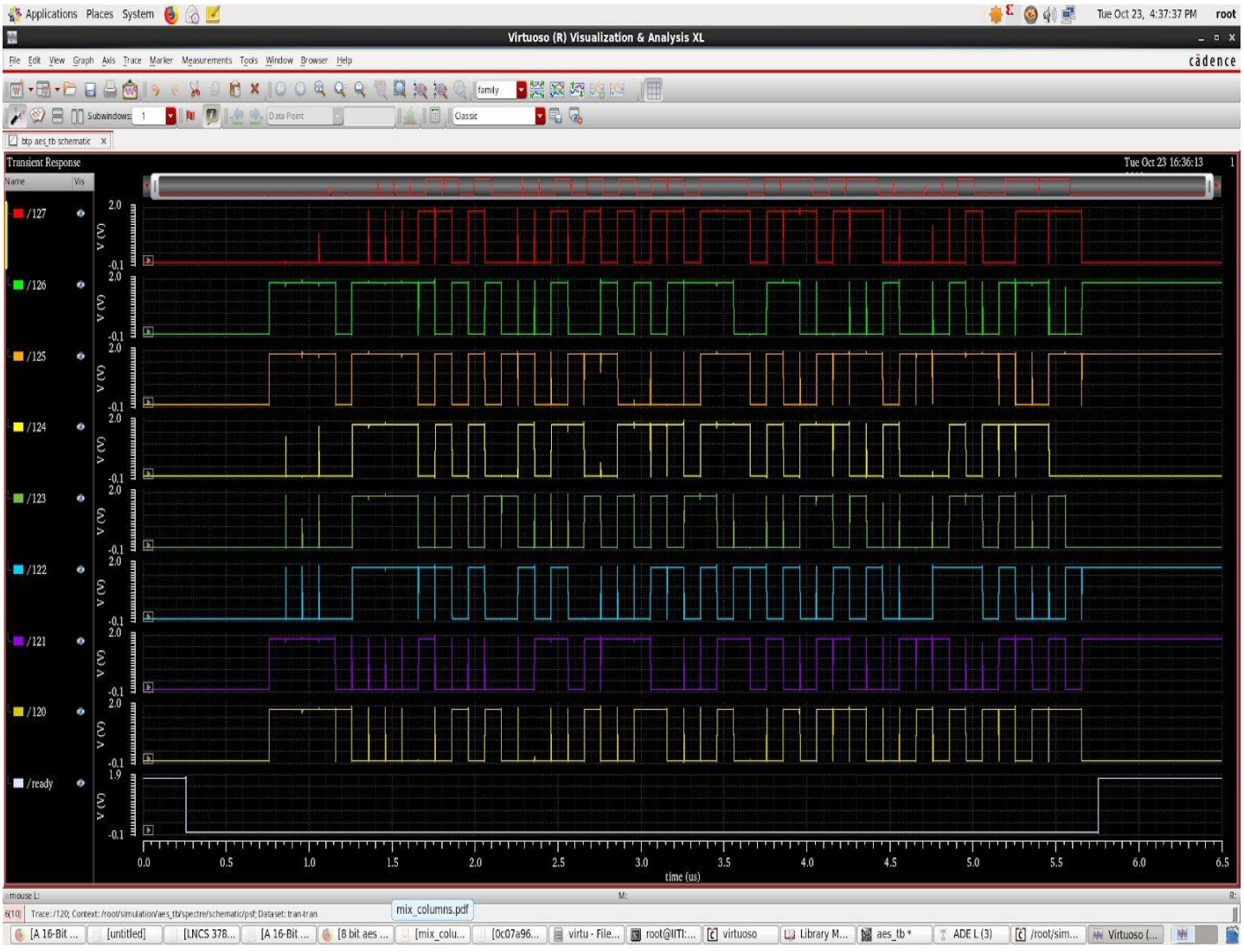

Fig 4.4 Simulation of the Designed Architecture using Virtuoso

Hence, the designed AES module is working perfectly.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

In this project i have presented a low-power architecture of AES suitable for IoT applications implemented on SCL 180 nm technology. The proposed architecture provides moderate throughput, which is sufficient for all of the IoT nodes. The post layout simulation results show that the complete design consumes about 194.7 µW but after applying voltage scaling the design consumes 52.2 µW at 1 V, 77.35 µW at 1.2 V, 108.7µW at 1.4 V and 142.5 µW at 1.6 V. The future work direction of this work is to further reduce the power by investigating different low-power design techniques.

Design	Block Size (bit)	Key Size (bit)	Technology (nm)	#Clock Cylces	Frequency (MHz)	Power (μW)	Throughput (Mbps)	Energy (pJ)
This work	128	128	180	44	10	52.2 @ 1V 194.7 @ 1.8V	28	5.2 @ 1V 19.47 @ 1.8V
[8]	128	128, 192, 256	28	44, 52, 60	10	20 @ 0.6V	28	2
[9]	128	128	90	44	10	-	28	-
[10]	128	128	40	337	122	100 @ 0.45V	46.2	0.8196
[11]	128	128	65	160	32	61.7 @ 0.6V	25.6	1.92
[12]	128	128	22	336	76	170 @ 0.34V	29	2.2368

Table 5.1 Analyzing the Power consumption of our design with previous designs

5.2 Future Work

It is about 20 years since AES has been proposed and there are several AES implementations existing for a variety of applications ranging from IoT to high-performance computing. The challenge is to implement AES designs which are not only low power and compact but also side-channel resistant.

In computer security, a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs). Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.

The future work direction of this work is to further reduce the power by investigating different low-power design techniques and also including the side-channel-resistant.

References and Image Sources

- AES Datapath Optimization Strategies forLow-Power Low-Energy Multisecurity-Level Internet-of-Things Applications by Duy-Hieu Bui, Student Member, IEEE, Diego Puschini, Simone Bacles-Min,Edith Beigné, Senior Member, IEEE, and Xuan-Tu Tran, Senior Member, IEEE
- Practical Implementation of Rijndael S-Box using combinational Logic by Edwin NC Mui, Custom R&D Engineer, Texco Enterprise Ptd.Ltd
- A Low Cost DPA –Resistant 8-bit AES Core Based on Ring Oscillators by Hsing-ping Fu, Ju-Hung Hsiao, Po-chun Liu, Hsie-chia Chang, and Chen-Yi-Lee
- ELEC-E3540 Digital Microelectronics II L Implementation instructions by Enrico Roverato
- Understanding Cryptography (A Textbook for Students and Practitioners) by Christof Paar . jan Pelzl
- Ieeexplore.ieee.org
- www.wikipedia.com
- https://www.zcorum.com/the-hold-up-with-the-internet-of-things/
- https://www.newgenapps.com/blog/iot-statistics-internet-of-things-future-research-data
- https://crypto.stackexchange.com/questions/2711/does-the-mixcolumns-step-come-before-or-after-addroundkey-in-aes-decryption
- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://captanu.wordpress.com/tag/aes/
- https://www.researchgate.net/figure/Mix-column-transformation2_fig4_304141002

Appendix A Implemented Code

Three files are there

1.aes_encipher_block.v
2.aes_key_mem.v
3.aes_sbox.v

1.

```
module aes_encipher_block (
                                input wire              clk,
                                input wire              reset_n,
                                input wire [127 : 0]    key,
                                input wire [127 : 0]    block,
                                output reg [127 : 0]    block_new,
                                output wire             ready
                                );

wire[127:0]  round_key;
reg [1 : 0]     sword_ctr_reg;
reg [1 : 0]     sword_ctr_new;

wire[31:0] sboxw, new_sboxw;
aes_sbox sbox_inst (.sboxw(sboxw), .new_sboxw(new_sboxw));
aes_key_mem keymem(.clk(clk),.reset_n(reset_n),.key(key),.key_sajid(round_key));

//----------------------------------------------------------------
// Round functions with sub functions.
//----------------------------------------------------------------
function [7 : 0] gm2(input [7 : 0] op);
        begin
        gm2 = {op[6 : 0], 1'b0} ^ (8'h1b & {8{op[7]}});
        end
endfunction // gm2

function [7 : 0] gm3(input [7 : 0] op);
        begin
        gm3 = gm2(op) ^ op;
        end
endfunction // gm3

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

```
        b3 = w[07 : 00];
        mb0 = gm2(b0) ^ gm3(b1) ^ b2        ^ b3;
        mb1 = b0        ^ gm2(b1) ^ gm3(b2) ^ b3;
        mb2 = b0        ^ b1     ^ gm2(b2) ^ gm3(b3);
        mb3 = gm3(b0) ^ b1   ^ b2     ^ gm2(b3);
        mixw = {mb0, mb1, mb2, mb3};
        end
endfunction // mixw

function [127 : 0] mixcolumns(input [127 : 0] data);
        reg [31 : 0] w0, w1, w2, w3;
        reg [31 : 0] ws0, ws1, ws2, ws3;
        begin
        w0 = data[127 : 096];
        w1 = data[095 : 064];
        w2 = data[063 : 032];
        w3 = data[031 : 000];
        ws0 = mixw(w0);
        ws1 = mixw(w1);
        ws2 = mixw(w2);
        ws3 = mixw(w3);
        mixcolumns = {ws0, ws1, ws2, ws3};
        end
endfunction // mixcolumns

function [127 : 0] shiftrows(input [127 : 0] data);
        reg [31 : 0] w0, w1, w2, w3;
        reg [31 : 0] ws0, ws1, ws2, ws3;
        begin
        w0 = data[127 : 096];
        w1 = data[095 : 064];
        w2 = data[063 : 032];
        w3 = data[031 : 000];
        ws0 = {w0[31 : 24], w1[23 : 16], w2[15 : 08], w3[07 : 00]};
        ws1 = {w1[31 : 24], w2[23 : 16], w3[15 : 08], w0[07 : 00]};
        ws2 = {w2[31 : 24], w3[23 : 16], w0[15 : 08], w1[07 : 00]};
        ws3 = {w3[31 : 24], w0[23 : 16], w1[15 : 08], w2[07 : 00]};
        shiftrows = {ws0, ws1, ws2, ws3};
        end
endfunction // shiftrows

function [127 : 0] addroundkey(input [127 : 0] data, input [127 : 0] rkey);
        begin
        addroundkey = data ^ rkey;
        end
endfunction // addroundkey

//----------------------------------------------------------------
// Registers including update variables and write enable.
//----------------------------------------------------------------
reg             sword_ctr_we;
reg             sword_ctr_inc;
reg             sword_ctr_rst;
```

```
reg [3 : 0]     round_ctr_reg;
reg [3 : 0]     round_ctr_new;
reg             round_ctr_we;
reg             round_ctr_rst;
reg             round_ctr_inc;
reg [31 : 0]    block_w0_reg;
reg [31 : 0]    block_w1_reg;
reg [31 : 0]    block_w2_reg;
reg [31 : 0]    block_w3_reg;
reg             block_w0_we;
reg             block_w1_we;
reg             block_w2_we;
reg             block_w3_we;
reg             ready_reg;
reg             ready_new;
reg             ready_we;
reg [2 : 0]     enc_ctrl_reg;
reg [2 : 0]     enc_ctrl_new;
reg             enc_ctrl_we;
//----------------------------------------------------------------
// Wires.
//----------------------------------------------------------------
reg [2 : 0]   update_type;
reg [31 : 0] muxed_sboxw;
//----------------------------------------------------------------
// Concurrent connectivity for ports etc.
//----------------------------------------------------------------

assign sboxw       =   muxed_sboxw;
assign new_block =  {block_w0_reg, block_w1_reg, block_w2_reg, block_w3_reg};
assign ready        =   ready_reg;

always @ (posedge clk or negedge reset_n)
        begin: reg_update
        if (!reset_n)
        begin
        block_w0_reg  <= 32'h0;
        block_w1_reg  <= 32'h0;
        block_w2_reg  <= 32'h0;
        block_w3_reg  <= 32'h0;
        sword_ctr_reg  <= 2'h0;
        round_ctr_reg   <= 4'h0;
        ready_reg        <= 1'b1;
        enc_ctrl_reg     <= 3'h0;
        end
        else
        begin
        if (block_w0_we)
        block_w0_reg <= block_new[127 : 096];
        if (block_w1_we)
        block_w1_reg <= block_new[095 : 064];
        if (block_w2_we)
        block_w2_reg <= block_new[063 : 032];
```

```
        if (block_w3_we)
        block_w3_reg <= block_new[031 : 000];
        if (sword_ctr_we)
        sword_ctr_reg <= sword_ctr_new;
        if (round_ctr_we)
        round_ctr_reg <= round_ctr_new;
        if (ready_we)
        ready_reg <= ready_new;
        if (enc_ctrl_we)
        enc_ctrl_reg <= enc_ctrl_new;
        end
        end

always @*
        begin : round_logic
        reg [127 : 0] old_block, shiftrows_block, mixcolumns_block;
        reg [127 : 0] addkey_init_block, addkey_main_block, addkey_final_block;
        block_new   = 128'h0;
        muxed_sboxw = 32'h0;
        block_w0_we = 1'b0;
        block_w1_we = 1'b0;
        block_w2_we = 1'b0;
        block_w3_we = 1'b0;
        old_block       = {block_w0_reg, block_w1_reg, block_w2_reg, block_w3_reg};
        shiftrows_block         = shiftrows(old_block);
        mixcolumns_block   = mixcolumns(shiftrows_block);
        addkey_init_block  = addroundkey(block, round_key);
        addkey_main_block  = addroundkey(mixcolumns_block, round_key);
        addkey_final_block = addroundkey(shiftrows_block, round_key);

        case (update_type)
        3'h1:
        begin
        block_new      = addkey_init_block;
        block_w0_we  = 1'b1;
        block_w1_we  = 1'b1;
        block_w2_we  = 1'b1;
        block_w3_we  = 1'b1;
        end
        3'h2:
        begin
        block_new = {new_sboxw, new_sboxw, new_sboxw, new_sboxw};
        case (sword_ctr_reg)
        2'h0:
                begin
                muxed_sboxw = block_w0_reg;
                block_w0_we = 1'b1;
                end
        2'h1:
                begin
                muxed_sboxw = block_w1_reg;
                block_w1_we = 1'b1;
                end
```

```
        2'h2:
                begin
                muxed_sboxw = block_w2_reg;
                block_w2_we = 1'b1;
                end
        2'h3:
                begin
                muxed_sboxw = block_w3_reg;
                block_w3_we = 1'b1;
                end
        endcase
        end
        3'h3:
        begin
        block_new      = addkey_main_block;
        block_w0_we  = 1'b1;
        block_w1_we  = 1'b1;
        block_w2_we  = 1'b1;
        block_w3_we  = 1'b1;
        end
        3'h4:
        begin
        block_new      = addkey_final_block;
        end
        default:
        begin
        end
        endcase
        end

always @*
        begin : sword_ctr
        sword_ctr_new = 2'h0;
        sword_ctr_we  = 1'b0;
        if (sword_ctr_rst)
        begin
        sword_ctr_new = 2'h0;
        sword_ctr_we  = 1'b1;
        end
        else if (sword_ctr_inc)
        begin
        sword_ctr_new = sword_ctr_reg + 1'b1;
        sword_ctr_we  = 1'b1;
        end
        end

always @*
        begin : round_ctr
        round_ctr_we  = 1'b0;

        if (round_ctr_rst)
        begin
        round_ctr_new = 4'h0;
```

```
        round_ctr_we  = 1'b1;
        end
        else if (round_ctr_inc)
        begin
        round_ctr_new = round_ctr_reg + 1'b1;
        round_ctr_we  = 1'b1;
        end
        end

always @*
        begin: encipher_ctrl
        reg [3 : 0]  num_rounds;
        sword_ctr_inc = 1'b0;
        sword_ctr_rst = 1'b0;
        round_ctr_inc = 1'b0;
        round_ctr_rst = 1'b0;
        ready_new      = 1'b0;
        ready_we       = 1'b0;
        update_type   = 3'h0;
        enc_ctrl_new  = 3'h0;
        enc_ctrl_we   = 1'b0;
        case(enc_ctrl_reg)
        3'h0 :
                begin
                ready_new      = 1'b0;
                ready_we       = 1'b1;
                enc_ctrl_new  = 3'h5;
                enc_ctrl_we   = 1'b1;
                end
        3'h5 :
        begin
                ready_new      = 1'b0;
                ready_we       = 1'b1;
                enc_ctrl_new  = 3'h6;
                enc_ctrl_we   = 1'b1;
        end
        3'h6 :
                begin
                ready_new      = 1'b0;
                ready_we       = 1'b1;
                enc_ctrl_new  = 3'h7;
                enc_ctrl_we   = 1'b1;
                end
                        3'h7 :
                begin
                        ready_new      = 1'b0;
                        ready_we       = 1'b1;
                        enc_ctrl_new  = 4'h1;
                        enc_ctrl_we   = 1'b1;
                end
                3'h1 :
                begin
                        ready_new      = 1'b0;
```

```
                    ready_we        = 1'b1;
                    enc_ctrl_new  = 3'h2;
                    enc_ctrl_we   = 1'b1;
            end
        3'h2 :
        begin
        round_ctr_inc = 1'b1;
        sword_ctr_rst = 1'b1;
        update_type   = 3'h1;
        enc_ctrl_new  = 3'h3;
        enc_ctrl_we   = 1'b1;
        end
        3'h3:
        begin
        sword_ctr_inc = 1'b1;
        update_type   = 3'h2;
        if (sword_ctr_reg == 2'h3)
        begin
            enc_ctrl_new  = 3'h4;
            enc_ctrl_we   = 1'b1;
        end
        end
        3'h4:
        begin
        sword_ctr_rst = 1'b1;
        if (round_ctr_reg < 4'ha) // number of rouunds for aes 128
        begin
            round_ctr_inc = 1'b1;
            update_type   = 3'h3;
            enc_ctrl_new  = 3'h3;
            enc_ctrl_we   = 1'b1;
        end
        else
        begin
            update_type  = 3'h4;
            ready_new      = 1'b1;
            ready_we       = 1'b1;
            enc_ctrl_new = 3'h4;
            enc_ctrl_we  = 1'b1;
            round_ctr_inc = 1'b0;
        end
        end
        default:
        begin
        // Empty. Just here to make the synthesis tool happy.
        end
        endcase
        end

endmodule // aes_encipher_block
```

2.

```
module aes_key_mem (
                        input wire              clk,
                        input wire              reset_n,
                        input wire [127 : 0]    key,
                        output reg [127 : 0]    key_sajid
                    );
  wire [31 : 0]  sbox_out, sbox_in;
  aes_sbox sbox_inst_sw(.sboxw(sbox_out), .new_sboxw(sbox_in));

  reg [127 : 0]  key_mem_new;
  reg            key_mem_we;
  reg [127 : 0]  prev_key1_reg;
  reg [127 : 0]  prev_key1_new;
  reg            prev_key1_we;
  reg [3 : 0]    round_ctr_reg;
  reg [3 : 0]    round_ctr_new;
  reg            round_ctr_rst;
  reg            round_ctr_inc;
  reg            round_ctr_we;
  reg [2 : 0]    key_mem_ctrl_reg;
  reg [2 : 0]    key_mem_ctrl_new;
  reg            key_mem_ctrl_we;
  reg [7 : 0]    rcon_reg;
  reg [7 : 0]    rcon_new;
  reg            rcon_we;
  reg            rcon_set;
  reg            rcon_next;
  reg[2:0]       sbox_count_new,sbox_count;
  reg [31 : 0]   tmp_sboxw;
  reg            round_key_update;
  reg [3 : 0]    num_rounds;
  reg [127 : 0]  tmp_round_key;

  assign sbox_out = tmp_sboxw;

  always @ (posedge clk or negedge reset_n)
        begin: reg_update
        integer i;
        if (!reset_n)
        begin
        key_sajid <=key;
        rcon_reg        <= 8'h0;
        round_ctr_reg <= 4'h0;
        key_mem_ctrl_reg <= 3'h0;
        rcon_reg<=8'h8d;
        end
        else
        begin
        if (round_ctr_we)
        round_ctr_reg <= round_ctr_new;
        sbox_count<= sbox_count_new;
```

```
        if (rcon_next)
        rcon_reg <= rcon_new;
        if (key_mem_we)
        key_sajid <= key_mem_new;
        if (prev_key1_we)
        prev_key1_reg <= prev_key1_new;
        if (key_mem_ctrl_we)
        key_mem_ctrl_reg <= key_mem_ctrl_new;
        end
        end // reg_update

always @*
        begin: round_key_gen
        reg [31 : 0] w0, w1, w2, w3, w4, w5, w6, w7;
        reg [31 : 0] k0, k1, k2, k3;
        reg [31 : 0] rconw, rotstw, tw, trw;
        // Default assignments.
        key_mem_new   = 128'h0;
        key_mem_we  = 1'b0;
        prev_key1_new = 128'h0;
        prev_key1_we  = 1'b0;
        k0 = 32'h0;
        k1 = 32'h0;
        k2 = 32'h0;
        k3 = 32'h0;
        rcon_next  = 1'b0;
        w4 = prev_key1_reg[127 : 096];
        w5 = prev_key1_reg[095 : 064];
        w6 = prev_key1_reg[063 : 032];
        w7 = prev_key1_reg[031 : 000];
        rconw = {rcon_reg, 24'h0};
        tmp_sboxw = w7;
        rotstw = {sbox_in[23 : 00], sbox_in[31 : 24]};
        trw = rotstw ^ rconw;
        tw = sbox_in;
        if (round_key_update)
        begin
        rcon_set   = 1'b0;
        key_mem_we = 1'b1;
        if (round_ctr_reg == 0)
                begin
                key_mem_new   = key_sajid[127 : 0];
                prev_key1_new  = key_sajid[127 : 0];
                prev_key1_we    = 1'b1;
                rcon_next          = 1'b1;
                end
                else
                begin
                k0 = w4 ^ trw;
                k1 = w5 ^ w4 ^ trw;
                k2 = w6 ^ w5 ^ w4 ^ trw;
                k3 = w7 ^ w6 ^ w5 ^ w4 ^ trw;
                key_mem_new   = {k0, k1, k2, k3};
```

```
                prev_key1_new = {k0, k1, k2, k3};
                prev_key1_we  = 1'b1;
                rcon_next       = 1'b1;
                end
        end
        end // round_key_gen

always @*
        begin : rcon_logic
        reg [7 : 0] tmp_rcon;
        rcon_new = {rcon_reg[6 : 0], 1'b0} ^ (8'h1b & {8{rcon_reg[7]}});
        end

always @*
        begin : round_ctr
        round_ctr_new = 4'h0;
        round_ctr_we  = 1'b0;
        if (round_ctr_rst)
        begin
        round_ctr_new = 4'h0;
        round_ctr_we  = 1'b1;
        end
        else if (round_ctr_inc)
        begin               round_ctr_new = round_ctr_reg + 1'b1;
        round_ctr_we  = 1'b1;
        end
        end

always @*
        begin: key_mem_ctrl
        round_key_update = 1'b0;
        round_ctr_rst   = 1'b0;
        round_ctr_inc  = 1'b0;
        key_mem_ctrl_new = 3'h0;
        key_mem_ctrl_we  = 1'b0;
        case(key_mem_ctrl_reg)
        3'h0:
        begin
                key_mem_ctrl_new = 3'h1;
                key_mem_ctrl_we  = 1'b1;
                sbox_count_new=3'h5;
        end
        3'h1:
        begin
        round_ctr_rst   = 1'b1;
        key_mem_ctrl_new = 3'h2;
        key_mem_ctrl_we  = 1'b1;
        sbox_count_new=0;
        end
        3'h2:
        begin
        if(sbox_count==3'h0) begin
                round_ctr_inc  = 1'b1;
```

```
                round_key_update = 1'b1;
                end
        if (round_ctr_reg == 4'hb) // number for rounds for AES128
        begin
                key_mem_ctrl_new = 3'h3;
                key_mem_ctrl_we  = 1'b1;
        end
        if(sbox_count==3'h4)
                sbox_count_new=0;
        else
                sbox_count_new = sbox_count+1;
        end
        3'h3:
        begin
        key_mem_ctrl_new = 3'h3;
        key_mem_ctrl_we  = 1'b1;
        sbox_count_new=3'h6;
        end
        default:
        begin
        end
        endcase
        end

endmodule // aes_key_mem
```

3.

```
module aes_sbox (
                        input wire [31 : 0]  sboxw,
                        output wire [31 : 0] new_sboxw
                        );
  wire [7 : 0] sbox [0 : 255];
  assign new_sboxw[31 : 24] = sbox[sboxw[31 : 24]];
  assign new_sboxw[23 : 16] = sbox[sboxw[23 : 16]];
  assign new_sboxw[15 : 08] = sbox[sboxw[15 : 08]];
  assign new_sboxw[07 : 00] = sbox[sboxw[07 : 00]];

  assign sbox[8'h00] = 8'h63;
  assign sbox[8'h01] = 8'h7c;
  assign sbox[8'h02] = 8'h77;
  assign sbox[8'h03] = 8'h7b;
  assign sbox[8'h04] = 8'hf2;
  assign sbox[8'h05] = 8'h6b;
  assign sbox[8'h06] = 8'h6f;
  assign sbox[8'h07] = 8'hc5;
  assign sbox[8'h08] = 8'h30;
  assign sbox[8'h09] = 8'h01;
  assign sbox[8'h0a] = 8'h67;
  assign sbox[8'h0b] = 8'h2b;
  assign sbox[8'h0c] = 8'hfe;
  assign sbox[8'h0d] = 8'hd7;
  assign sbox[8'h0e] = 8'hab;
  assign sbox[8'h0f] = 8'h76;
  assign sbox[8'h10] = 8'hca;
  assign sbox[8'h11] = 8'h82;
  assign sbox[8'h12] = 8'hc9;
  assign sbox[8'h13] = 8'h7d;
  assign sbox[8'h14] = 8'hfa;
  assign sbox[8'h15] = 8'h59;
  assign sbox[8'h16] = 8'h47;
  assign sbox[8'h17] = 8'hf0;
  assign sbox[8'h18] = 8'had;
  assign sbox[8'h19] = 8'hd4;
  assign sbox[8'h1a] = 8'ha2;
  assign sbox[8'h1b] = 8'haf;
  assign sbox[8'h1c] = 8'h9c;
  assign sbox[8'h1d] = 8'ha4;
  assign sbox[8'h1e] = 8'h72;
  assign sbox[8'h1f] = 8'hc0;
  assign sbox[8'h20] = 8'hb7;
  assign sbox[8'h21] = 8'hfd;
  assign sbox[8'h22] = 8'h93;
  assign sbox[8'h23] = 8'h26;
  assign sbox[8'h24] = 8'h36;
  assign sbox[8'h25] = 8'h3f;
```

```
assign sbox[8'h26] = 8'hf7;
assign sbox[8'h27] = 8'hcc;
assign sbox[8'h28] = 8'h34;
assign sbox[8'h29] = 8'ha5;
assign sbox[8'h2a] = 8'he5;
assign sbox[8'h2b] = 8'hf1;
assign sbox[8'h2c] = 8'h71;
assign sbox[8'h2d] = 8'hd8;
assign sbox[8'h2e] = 8'h31;
assign sbox[8'h2f] = 8'h15;
assign sbox[8'h30] = 8'h04;
assign sbox[8'h31] = 8'hc7;
assign sbox[8'h32] = 8'h23;
assign sbox[8'h33] = 8'hc3;
assign sbox[8'h34] = 8'h18;
assign sbox[8'h35] = 8'h96;
assign sbox[8'h36] = 8'h05;
assign sbox[8'h37] = 8'h9a;
assign sbox[8'h38] = 8'h07;
assign sbox[8'h39] = 8'h12;
assign sbox[8'h3a] = 8'h80;
assign sbox[8'h3b] = 8'he2;
assign sbox[8'h3c] = 8'heb;
assign sbox[8'h3d] = 8'h27;
assign sbox[8'h3e] = 8'hb2;
assign sbox[8'h3f] = 8'h75;
assign sbox[8'h40] = 8'h09;
assign sbox[8'h41] = 8'h83;
assign sbox[8'h42] = 8'h2c;
assign sbox[8'h43] = 8'h1a;
assign sbox[8'h44] = 8'h1b;
assign sbox[8'h45] = 8'h6e;
assign sbox[8'h46] = 8'h5a;
assign sbox[8'h47] = 8'ha0;
assign sbox[8'h48] = 8'h52;
assign sbox[8'h49] = 8'h3b;
assign sbox[8'h4a] = 8'hd6;
assign sbox[8'h4b] = 8'hb3;
assign sbox[8'h4c] = 8'h29;
assign sbox[8'h4d] = 8'he3;
assign sbox[8'h4e] = 8'h2f;
assign sbox[8'h4f] = 8'h84;
assign sbox[8'h50] = 8'h53;
assign sbox[8'h51] = 8'hd1;
assign sbox[8'h52] = 8'h00;
assign sbox[8'h53] = 8'hed;
assign sbox[8'h54] = 8'h20;
assign sbox[8'h55] = 8'hfc;
assign sbox[8'h56] = 8'hb1;
```

```
assign sbox[8'h57] = 8'h5b;
assign sbox[8'h58] = 8'h6a;
assign sbox[8'h59] = 8'hcb;
assign sbox[8'h5a] = 8'hbe;
assign sbox[8'h5b] = 8'h39;
assign sbox[8'h5c] = 8'h4a;
assign sbox[8'h5d] = 8'h4c;
assign sbox[8'h5e] = 8'h58;
assign sbox[8'h5f] = 8'hcf;
assign sbox[8'h60] = 8'hd0;
assign sbox[8'h61] = 8'hef;
assign sbox[8'h62] = 8'haa;
assign sbox[8'h63] = 8'hfb;
assign sbox[8'h64] = 8'h43;
assign sbox[8'h65] = 8'h4d;
assign sbox[8'h66] = 8'h33;
assign sbox[8'h67] = 8'h85;
assign sbox[8'h68] = 8'h45;
assign sbox[8'h69] = 8'hf9;
assign sbox[8'h6a] = 8'h02;
assign sbox[8'h6b] = 8'h7f;
assign sbox[8'h6c] = 8'h50;
assign sbox[8'h6d] = 8'h3c;
assign sbox[8'h6e] = 8'h9f;
assign sbox[8'h6f] = 8'ha8;
assign sbox[8'h70] = 8'h51;
assign sbox[8'h71] = 8'ha3;
assign sbox[8'h72] = 8'h40;
assign sbox[8'h73] = 8'h8f;
assign sbox[8'h74] = 8'h92;
assign sbox[8'h75] = 8'h9d;
assign sbox[8'h76] = 8'h38;
assign sbox[8'h77] = 8'hf5;
assign sbox[8'h78] = 8'hbc;
assign sbox[8'h79] = 8'hb6;
assign sbox[8'h7a] = 8'hda;
assign sbox[8'h7b] = 8'h21;
assign sbox[8'h7c] = 8'h10;
assign sbox[8'h7d] = 8'hff;
assign sbox[8'h7e] = 8'hf3;
assign sbox[8'h7f] = 8'hd2;
assign sbox[8'h80] = 8'hcd;
assign sbox[8'h81] = 8'h0c;
assign sbox[8'h82] = 8'h13;
assign sbox[8'h83] = 8'hec;
assign sbox[8'h84] = 8'h5f;
assign sbox[8'h85] = 8'h97;
assign sbox[8'h86] = 8'h44;
assign sbox[8'h87] = 8'h17;
```

```
assign sbox[8'h88] = 8'hc4;
assign sbox[8'h89] = 8'ha7;
assign sbox[8'h8a] = 8'h7e;
assign sbox[8'h8b] = 8'h3d;
assign sbox[8'h8c] = 8'h64;
assign sbox[8'h8d] = 8'h5d;
assign sbox[8'h8e] = 8'h19;
assign sbox[8'h8f] = 8'h73;
assign sbox[8'h90] = 8'h60;
assign sbox[8'h91] = 8'h81;
assign sbox[8'h92] = 8'h4f;
assign sbox[8'h93] = 8'hdc;
assign sbox[8'h94] = 8'h22;
assign sbox[8'h95] = 8'h2a;
assign sbox[8'h96] = 8'h90;
assign sbox[8'h97] = 8'h88;
assign sbox[8'h98] = 8'h46;
assign sbox[8'h99] = 8'hee;
assign sbox[8'h9a] = 8'hb8;
assign sbox[8'h9b] = 8'h14;
assign sbox[8'h9c] = 8'hde;
assign sbox[8'h9d] = 8'h5e;
assign sbox[8'h9e] = 8'h0b;
assign sbox[8'h9f] = 8'hdb;
assign sbox[8'ha0] = 8'he0;
assign sbox[8'ha1] = 8'h32;
assign sbox[8'ha2] = 8'h3a;
assign sbox[8'ha3] = 8'h0a;
assign sbox[8'ha4] = 8'h49;
assign sbox[8'ha5] = 8'h06;
assign sbox[8'ha6] = 8'h24;
assign sbox[8'ha7] = 8'h5c;
assign sbox[8'ha8] = 8'hc2;
assign sbox[8'ha9] = 8'hd3;
assign sbox[8'haa] = 8'hac;
assign sbox[8'hab] = 8'h62;
assign sbox[8'hac] = 8'h91;
assign sbox[8'had] = 8'h95;
assign sbox[8'hae] = 8'he4;
assign sbox[8'haf] = 8'h79;
assign sbox[8'hb0] = 8'he7;
assign sbox[8'hb1] = 8'hc8;
assign sbox[8'hb2] = 8'h37;
assign sbox[8'hb3] = 8'h6d;
assign sbox[8'hb4] = 8'h8d;
assign sbox[8'hb5] = 8'hd5;
assign sbox[8'hb6] = 8'h4e;
assign sbox[8'hb7] = 8'ha9;
assign sbox[8'hb8] = 8'h6c;
```

```
assign sbox[8'hb9] = 8'h56;
assign sbox[8'hba] = 8'hf4;
assign sbox[8'hbb] = 8'hea;
assign sbox[8'hbc] = 8'h65;
assign sbox[8'hbd] = 8'h7a;
assign sbox[8'hbe] = 8'hae;
assign sbox[8'hbf] = 8'h08;
assign sbox[8'hc0] = 8'hba;
assign sbox[8'hc1] = 8'h78;
assign sbox[8'hc2] = 8'h25;
assign sbox[8'hc3] = 8'h2e;
assign sbox[8'hc4] = 8'h1c;
assign sbox[8'hc5] = 8'ha6;
assign sbox[8'hc6] = 8'hb4;
assign sbox[8'hc7] = 8'hc6;
assign sbox[8'hc8] = 8'he8;
assign sbox[8'hc9] = 8'hdd;
assign sbox[8'hca] = 8'h74;
assign sbox[8'hcb] = 8'h1f;
assign sbox[8'hcc] = 8'h4b;
assign sbox[8'hcd] = 8'hbd;
assign sbox[8'hce] = 8'h8b;
assign sbox[8'hcf] = 8'h8a;
assign sbox[8'hd0] = 8'h70;
assign sbox[8'hd1] = 8'h3e;
assign sbox[8'hd2] = 8'hb5;
assign sbox[8'hd3] = 8'h66;
assign sbox[8'hd4] = 8'h48;
assign sbox[8'hd5] = 8'h03;
assign sbox[8'hd6] = 8'hf6;
assign sbox[8'hd7] = 8'h0e;
assign sbox[8'hd8] = 8'h61;
assign sbox[8'hd9] = 8'h35;
assign sbox[8'hda] = 8'h57;
assign sbox[8'hdb] = 8'hb9;
assign sbox[8'hdc] = 8'h86;
assign sbox[8'hdd] = 8'hc1;
assign sbox[8'hde] = 8'h1d;
assign sbox[8'hdf] = 8'h9e;
assign sbox[8'he0] = 8'he1;
assign sbox[8'he1] = 8'hf8;
assign sbox[8'he2] = 8'h98;
assign sbox[8'he3] = 8'h11;
assign sbox[8'he4] = 8'h69;
assign sbox[8'he5] = 8'hd9;
assign sbox[8'he6] = 8'h8e;
assign sbox[8'he7] = 8'h94;
assign sbox[8'he8] = 8'h9b;
assign sbox[8'he9] = 8'h1e;
```

```
  assign sbox[8'hea] = 8'h87;
  assign sbox[8'heb] = 8'he9;
  assign sbox[8'hec] = 8'hce;
  assign sbox[8'hed] = 8'h55;
  assign sbox[8'hee] = 8'h28;
  assign sbox[8'hef] = 8'hdf;
  assign sbox[8'hf0] = 8'h8c;
  assign sbox[8'hf1] = 8'ha1;
  assign sbox[8'hf2] = 8'h89;
  assign sbox[8'hf3] = 8'h0d;
  assign sbox[8'hf4] = 8'hbf;
  assign sbox[8'hf5] = 8'he6;
  assign sbox[8'hf6] = 8'h42;
  assign sbox[8'hf7] = 8'h68;
  assign sbox[8'hf8] = 8'h41;
  assign sbox[8'hf9] = 8'h99;
  assign sbox[8'hfa] = 8'h2d;
  assign sbox[8'hfb] = 8'h0f;
  assign sbox[8'hfc] = 8'hb0;
  assign sbox[8'hfd] = 8'h54;
  assign sbox[8'hfe] = 8'hbb;
  assign sbox[8'hff] = 8'h16;

endmodule // aes_sbox
```

assign