IMPLEMENTING AES-256 ON FPGA

5th June 2021

Ho Hai Cong Thuan

Student, Computer Engineering, University of Information Technology, HCMC, Vietnam

hohaicongthuan@gmail.com

Abstract—In the modern days, the amount of data grow exponentially, including classified and sensitive data that need to be kept secured. For this reason, many cryptographic techniques have been invented for the purpose. AES is one of them. It provides fast and secure data encryption which are the reasons this algorithm is chosen for this project.

The goal of this project is to implement a fully functional AES encryption and decryption system using 256-bit key on FPGA.

Keywords—AES-256; cryptography; data security; FPGA; encryption; decryption.

1 INTRODUCTION

The Advanced Encryption Standard (AES), also known as *Rijndael* is a specification for encrypting electronic data first introduced by the *U.S. National Institute of Standards and Technology (NIST)* in 2001. It provides a fast and secure way to encrypt data and uses symmetric keys encryption which means both the encryption and decryption processes using the same key. The key length for AES could be 128, 192 and 256 bits. This paper concentrates on AES using 256-bit key which will be refered to as AES-256 for the rest of this paper.

1.1 Concepts used in AES-256

Key expansion	Routine used to generate a series of Round Keys from the Cipher Key.
State	Intermediate Cipher result that can be pictured as a rectangular array of bytes, having four rows and **Nb** columns.
S-box	Non-linear substitution table used in several byte substitution transformation and in the Key Expansion routine to perform a one-for-one substitution of a byte value.
Word	A group of 32 bits that is treated either as a single entity or as an array of 4 bytes.

1.2 Abbreviations and Symbols used in AES-256

Nb	Number of columns (32-bit words) comprising the State. For this standard, **Nb** = 4.
Nk	Number of 32-bit words comprising the Cipher Key. For this standard, **Nk** = 8.
Nr	Number of rounds, which is a function of **Nk** and **Nb** (which is fixed). For this standard, **Nr** = 14.
XOR	Exclusive-OR operation
$\oplus$	Exclusive-OR operation
$\otimes$	Multiplication of two polynomials (each with degree < 4) modulo $x^4 + 1$
$\bullet$	Finite field multiplication

2 AES-256

2.1 Key Expansion

The Key Expansion routine in AES-256 takes a 256-bit cipher key and generate a set of $Nb(Nr + 1)$ (which is 60) words. These words are smaller parts that make up round keys, each round key has four words. These round keys involve in the *Add Round Key* in the encryption and decryption process.

There are 3 functions that participate in the key scheduling process:

RotWord	Takes a four-byte word $[a_0, a_1, a_2, a_3]$ and performs rotation one byte to the left and returns $[a_1, a_2, a_3, a_0]$.
SubWord	Takes a four-byte word and substitute each byte with the corresponding byte in the S-box.
Rcon	The round constants, which is given in the form $[rc_i, 00_{16}, 00_{16}, 00_{16}]$ with i starts from 1. rc_i is defined as in (1)

								Y									
		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
x	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

Figure 1: Substitution Box (S-Box) used in SubBytes transformation.

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

2.1.1 Algorithm

In AES-256, the first two round keys (first 8 words) are filled with the cipher key. For the rest, $w[i]$ word is generated using $w[i-1]$ word.

Loop through the following steps until we have generated $Nb(Nr+1)$ words.

If i is divisible by **Nk**, $w[i-1]$ is rotated by the function **RotWord** and then substituted by the function **SubWord**. The final result is **XOR**-ed with **Rcon[i/Nk]** and assigned to $w[i]$. Otherwise, if i dividing by **Nk** results in 4 as the remainder, only **SubWord** is performed on $w[i-1]$.

$w[i]$ will then be **XOR**-ed with $w[i-Nk]$ and the result is assigned back to itself. i is incremented by 1.

After finishing the algorithm, a set of $Nb(Nr+1)$ words is generated. Round Keys are created by grouping four words each sequentially. At this point, we have one round key for the initial round and 14 round keys for 14 rounds during the encryption or decryption processes, with the total of 15 round keys.

2.2 Bytes Substitution

Bytes Substitution transformation is denoted by *SubBytes* function. This independently replaces all the bytes in the *State* with the corresponding bytes using a *substituion box* (or *S-Box*), which is shown in the figure 1.

The higher 4 bits determine the coordinate of the row and lower 4 bits determine the coordinate of the column. For example, the byte **5A**, we have $x = 5$ & $y = A$, these values point to the new value in the table, which is **BE**. So **5A** will be substituted by **BE**.

2.3 Shift Rows

In the *ShiftRows* transformation, the last three row of the *State* will be rotated to the left with different byte offsets.

The first row of the *State* is unaffected. The second row will be rotated to the left by one byte, two bytes for the third row, and three bytes for the fourth row. Therefore,

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix} \quad (2)$$

will become

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_5 & a_9 & a_{13} & a_1 \\ a_{10} & a_{14} & a_2 & a_6 \\ a_{15} & a_3 & a_7 & a_{11} \end{bmatrix} \quad (3)$$

after *ShiftRows* transformation.

2.4 Mix Columns

MixColumns transformation operates column-by-column on the *State*. Each column is multiplied with a fixed matrix which results in a new column with new values.

The fixed matrix used in *MixColumns* is shown below:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \quad (4)$$

2.5 Add Round Keys

Round Keys generated from *Key Expansion* routine will be used in this transformation. Round Key is added to the *State* by a bitwise XOR operation with the corresponding bytes.
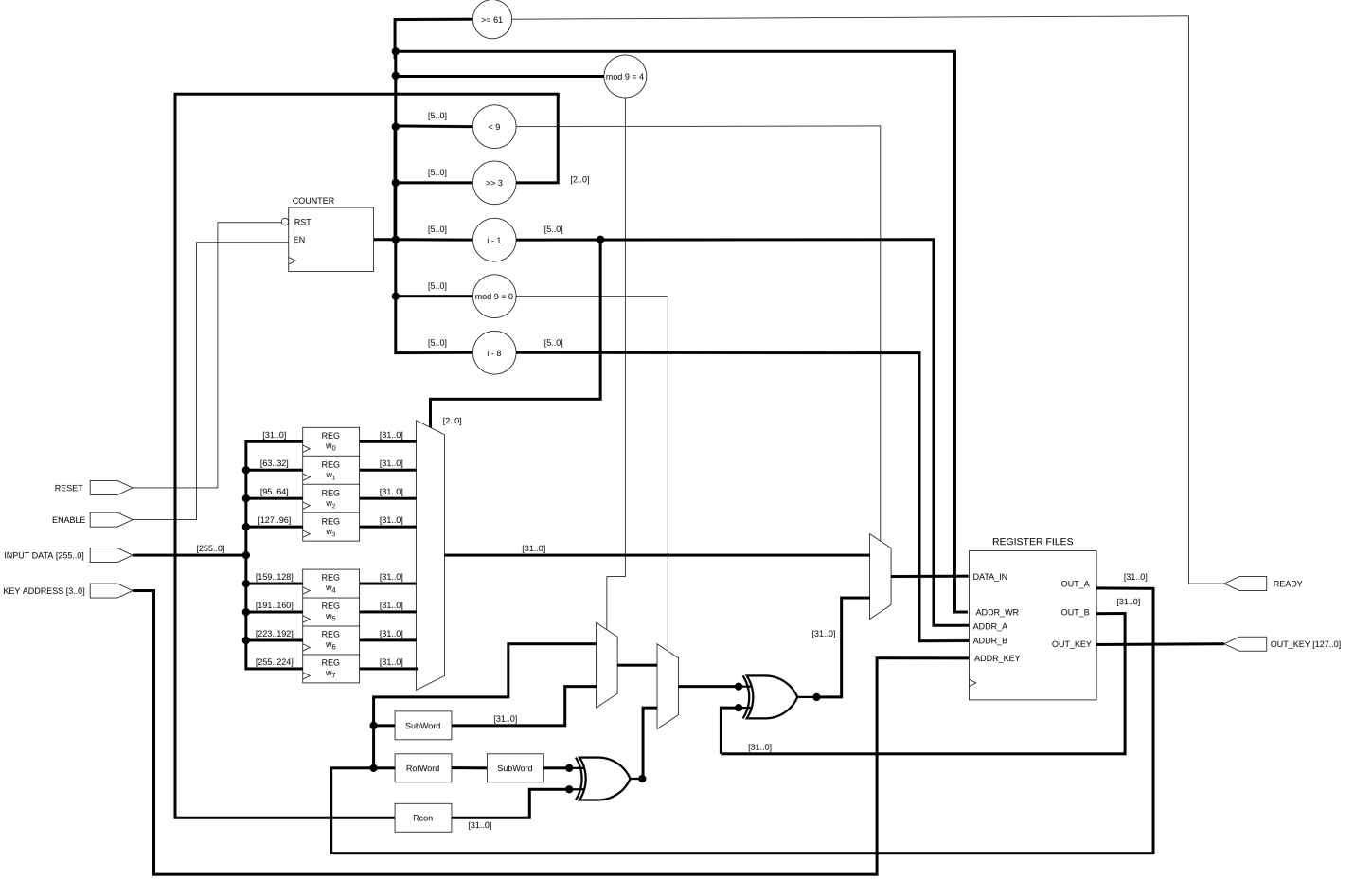
Figure 2: Diagram of Key Expansion.

3 IMPLEMENTATION

3.1 Key Expansion

The design of the Key Expansion routine is shown in figure 2. As in the design, the total of 15 round keys, including round key for the initial round, will be created firstbefore they could involve in the **Add Round Key** function in the encryption and decryption process.

This circuit will generate words that make up the Round Key one-by-one and store them in the *Register File*. The *Register File* has the total amount of 64 registers but only 60 of them are used because we only need to generate 60 words. The module *Rcon* and *SubWord* are implemented in the form of look-up tables (LUTs).

In order for the circuit to work, the *reset* signal must be pulled low (the *reset* signal is active low) and then the *enable* signal is pulled high. The cipher key can be passed through *Input Data*. After reset, the counter's value will be set to zero and increased by 1 after each clock cycle. In the meantime, the inputted cipher key will be broken up into words, which are 32 bits each, and stored in eight registers from w_0 to w_7.

In the next eight clock cycles each of the word will be saved into the *Register File* sequentially from the address 1 to 8. After that, to generate the ith word, $(i-1)^{th}$ and $(i-Nk)^{th}$ words are required, which can be accessed through Register File's Out_A & Out_B ports respectively, with the

addresses are selected by $i-1$ & $i-8$ blocks in the circuit.

Depend on which word is generating, *RotWord*, *SubWord* can all be performed or just *SubWord* is performed or neither of those. This is controlled by signals from *mod 9 = 4* & *mod 9 = 0* blocks. Basically, the *mod 9 = 4* block checks if a input number dividing by 9 has 4 as the remainder. If it is, the output will be pulled high and pulled low if it is not. Same thing with the *mod 9 = 0* block but with 0 as the remainder. The design of *mod 9 = 0* & *mod 9 = 4* are shown in figure 3 and figure 4, respectively.
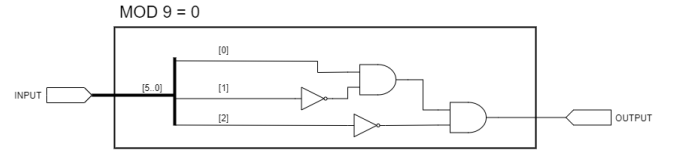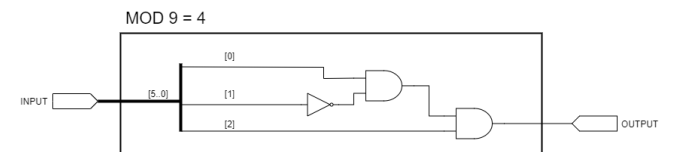

Figure 3: Circuit of *mod 9 = 0* block

Figure 4: Circuit of *mod 9 = 4* block

All the words will have been generated after sixty-one

clock cycles. The counter will shut itself and the *ready* signal is set indicating all round keys have been generated. After done generating and the *ready* signal is set, round keys can be accessed by inputting the appropriate address through *Key Address* input, which have the value range from 0 to 14.
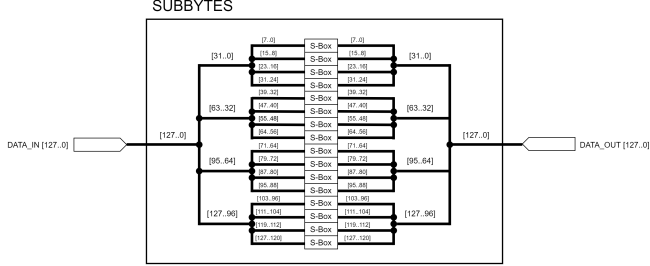
3.2 Bytes Substitution

Figure 5: SubBytes circuit diagram.

The design for *SubBytes* block is shown in figure 5. The *SBox* used in this circuit is implemented as LUT and is the same as *SBox* from figure 1 and *SubWord* in *Key Expansion* routine.

This ciruit takes 128-bit data and splits into words and then into individual bytes. Each byte will be fed into *SBox* and the corresponding byte will be outputted. The outputted bytes will then be combined into words and eventually into 128-bit output data.

3.3 Shift Rows
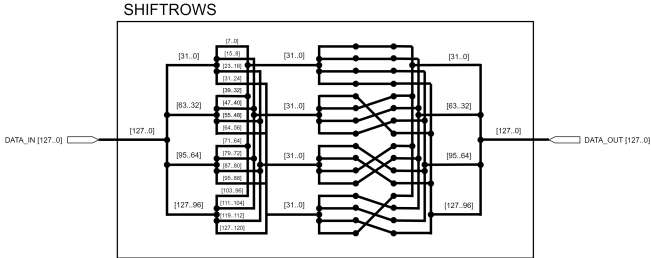
Figure 6: ShiftRows circuit diagram.

The circuit diagram for *ShiftRows* is shown in figure 6. This circuit takes 128-bit data in, perform shifting and output 128-bit data.

The inputted data will first be split into 32-bit words, note that each word corresponds to a column in the *State*. The first row is formed by taking the first byte in each word, and the second byte in each word for the second row and so on for the third and fourth rows. After dividing the input into rows, the shifting operation is done by wiring each row with the corresponding offsets which discussed in section 2.3.

Finally, all the rows will be split and combined into correct columns and rows format of the *State*.

3.4 Mix Columns

In figure 7 is the circuit design for *MixColumns* transformation. The 128-bit data input fed into the

circuit is divided into four words. Each word is fed into *First_Row_Mul*, *Second_Row_Mul*, *Third_Row_Mul*, and *Fourth_Row_Mul* blocks, which is basically multiplying that word using the multiplication in $GF(2^8)$ with first, second, third, and fourth rows of the matrix (4) from section 2.4. Each of the four blocks will output a new byte after the multiplication. Four bytes will be combined into a word. And the 128-bit output is formed by combining four words.
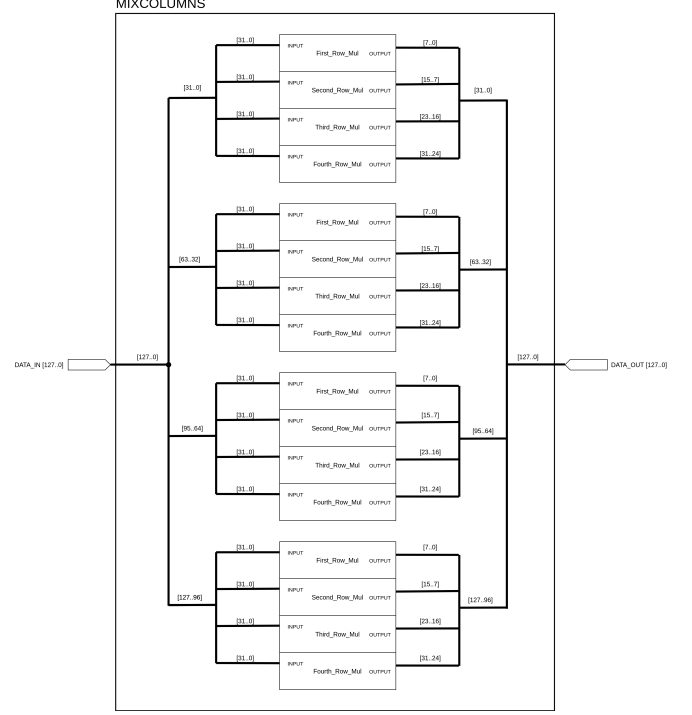

Figure 7: MixColumns circuit diagram.

The implementation of the *First_Row_Mul*, *Second_Row_Mul*, *Third_Row_Mul*, and *Fourth_Row_Mul* blocks is shown in figure 8, 9, 10, and 11, respectively. Note that all the addition operations (the plus symbol) in these circuits are all *XOR* operations.
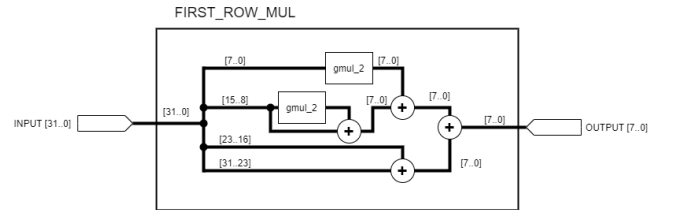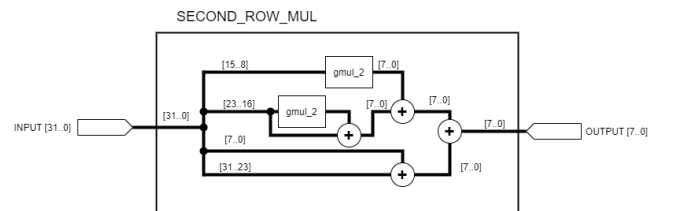

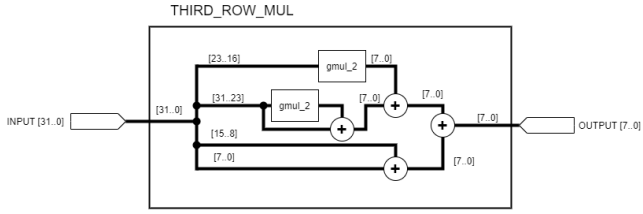Figure 8: First_Row_Mul.

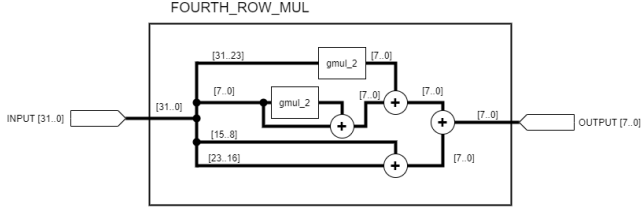Figure 9: Second_Row_Mul.

Figure 10: Third_Row_Mul.

Figure 11: Fourth_Row_Mul.

The implementation of the multiplication by 2 in $GF(2^8)$, which is denoted as *gmul_2*, is shown in figure 12. This multiplication operation is performed at the byte level as left shift operation followed by a conditional *XOR* operation with the number *1B* in hexadecimal.
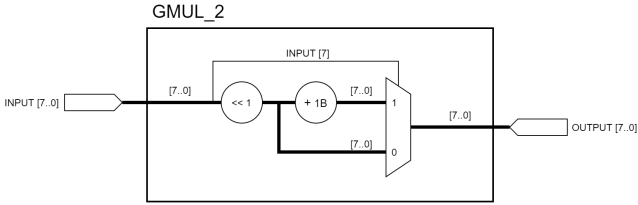

Figure 12: gmul_2.

3.5 Add Round Keys

Text here.

4 RESULTS

(N/A)

5 CONCLUSION

(N/A)

References

[1] Federal Information Processing Standard (FIPS) 197. *Advanced Encryption Standard (AES)* 26 November 2001.

[2] Sam Trenholme. *Rijndael's key schedule.* https://samiam.org/key-schedule.html.

[3] Sam Trenholme. *Rijndael's mix column stage.* https://samiam.org/mix-column.html.

[4] Sam Trenholme. *AES' Galois field.* https://samiam.org/galois.html.

[5] Sam Trenholme. *Rijndael's S-Box.* https://samiam.org/s-box.html.

[6] Kit Choy Xintong. *Understanding AES Mix-Columns Transformation Calculation.*