IMPLEMENTING AES-256 ON FPGA

25th June 2021

Ho Hai Cong Thuan
Student, Computer Engineering, University of Information Technology, HCMC, Vietnam
hohaicongthuan@gmail.com

***Abstract*—In 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 this 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 keys 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 keys which will be referred to as AES-256 for the rest of this paper.

1.1 Concepts used in AES-256

Key expansion	Routine used to generate a set 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	Substitution table used in byte substitution transformation and in the Key Expansion routine to perform a one-for-one replacement 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)

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

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 $\boldsymbol{Nk}$, $w[i-1]$ 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 $\boldsymbol{Nk}$ 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.

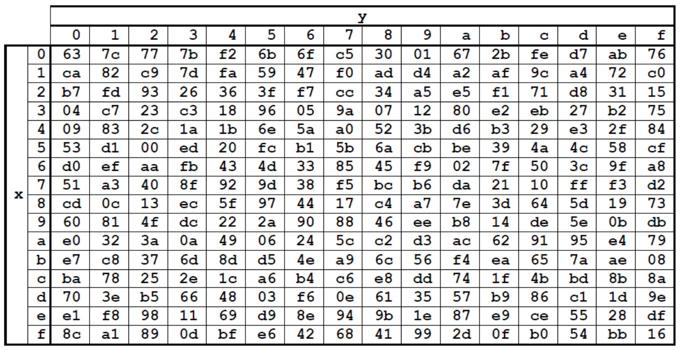
After finishing the algorithm, a set of $Nb(Nr+1)$ words is generated. Round Keys are created by grouping four words each sequentially. At this point, we have one round key for the initial round and 14 round keys for 14 rounds during the encryption or decryption processes, with a total of 15 round keys.

2.2 AES-256 Encryption

The AES-256 Encryption process comprises of fourteen rounds plus the initial round. In the initial round, the input data will be added with the initial round key. In the next thirteen rounds, all the *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* transformations will be performed on the *State*, respectively. In the last round, only *SubBytes*, *ShiftRows*, and *AddRoundKey* are performed.

2.2.1 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.



		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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

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

The higher 4 bits determine the coordinate of the row and the 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**.

2.2.2 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,

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

will become

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

after *ShiftRows* transformation.

2.2.3 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:

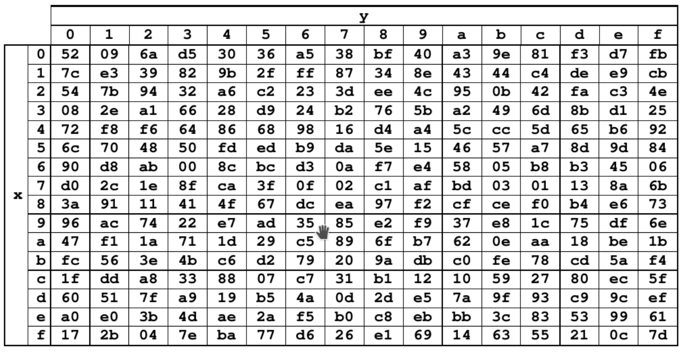
$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \tag{4}$$

The multiplication operation performed in this *MixColumns* transformation is not ordinary integer multiplication but multiplication in *GF*(2^8), which can be performed at byte level as left shift operation followed by the conditional *XOR* operation with the number **1B** in hexadecimal.

2.3 AES-256 Decryption

The AES-256 Decryption process is the inverse of the AES-256 Encryption process. Basically, it is the same as the encryption process but in reverse. Round keys used in the decryption process are also in reverse order. However, the *SubBytes*, *ShiftRows*, and *MixColumns* transformations are replaced with *InvSubBytes*, *InvShiftRows*, and *InvMixColumns*, respectively. Details of those transformations are discussed in the following sub-sections.

2.3.1 Inverse Bytes Substitution



		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 2: Inverse S-Box

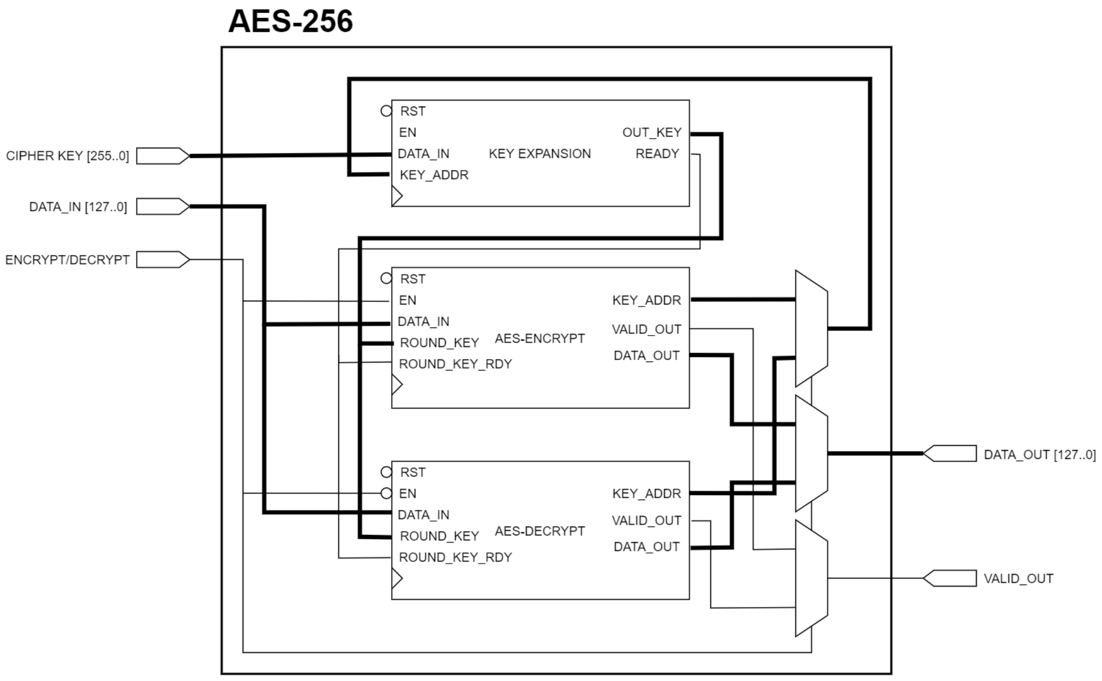


Figure 3: Top-level design of AES-256.

In the *Inverse Bytes Substitution* transformation, which is denoted as *InvSubBytes*, all bytes from the *State* will be replaced by the corresponding bytes from the inverse S-Box, which is shown in figure 2.

2.3.2 Inverse Shift Rows

Inverse Shift Rows denoted as *InvShiftRows* is the inverse of *ShiftRows* transformation. Each row from the *State* is rotated with different offsets to the right instead of left. According to that, the first row is untouched, the second row is rotated one byte to the right, rotated two bytes to the right for the third row, and three bytes for the fourth row. Suppose (2) as the example input, after *InvShiftRows* transformation, the input will become

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_{13} & a_1 & a_5 & a_9 \\ a_{10} & a_{14} & a_2 & a_6 \\ a_7 & a_{11} & a_{15} & a_3 \end{bmatrix} \tag{5}$$

2.3.3 Inverse Mix Columns

Inverse Mix Columns denoted as *InvMixColumns* is the inverse of *MixColumns* transformation. Each column from the *State* is multiplied with the inverse matrix of the matrix (4), which is shown in (6), using the multiplication in $GF(2^8)$. Note that all numbers in the matrix are written in hexadecimal.

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \tag{6}$$

2.4 Add Round Keys

Round keys generated from *Key Expansion* routine will be used in this transformation. It takes two 128-bit input data and output 128-bit data. Round keys are added to the *State* by bitwise *XOR* operations with the corresponding bytes.

3 IMPLEMENTATION

The top-level design for AES-256, including encryption and decryption, is shown in figure 3. The implementation of smaller modules is discussed in the following sub-sections.

3.1 Key Expansion

The design of the Key Expansion routine is shown in figure 4. As in the design, the total of 15 round keys, including the round key for the initial round, will be created first before 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 a 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 words will be saved into the *Register File* sequentially from the address

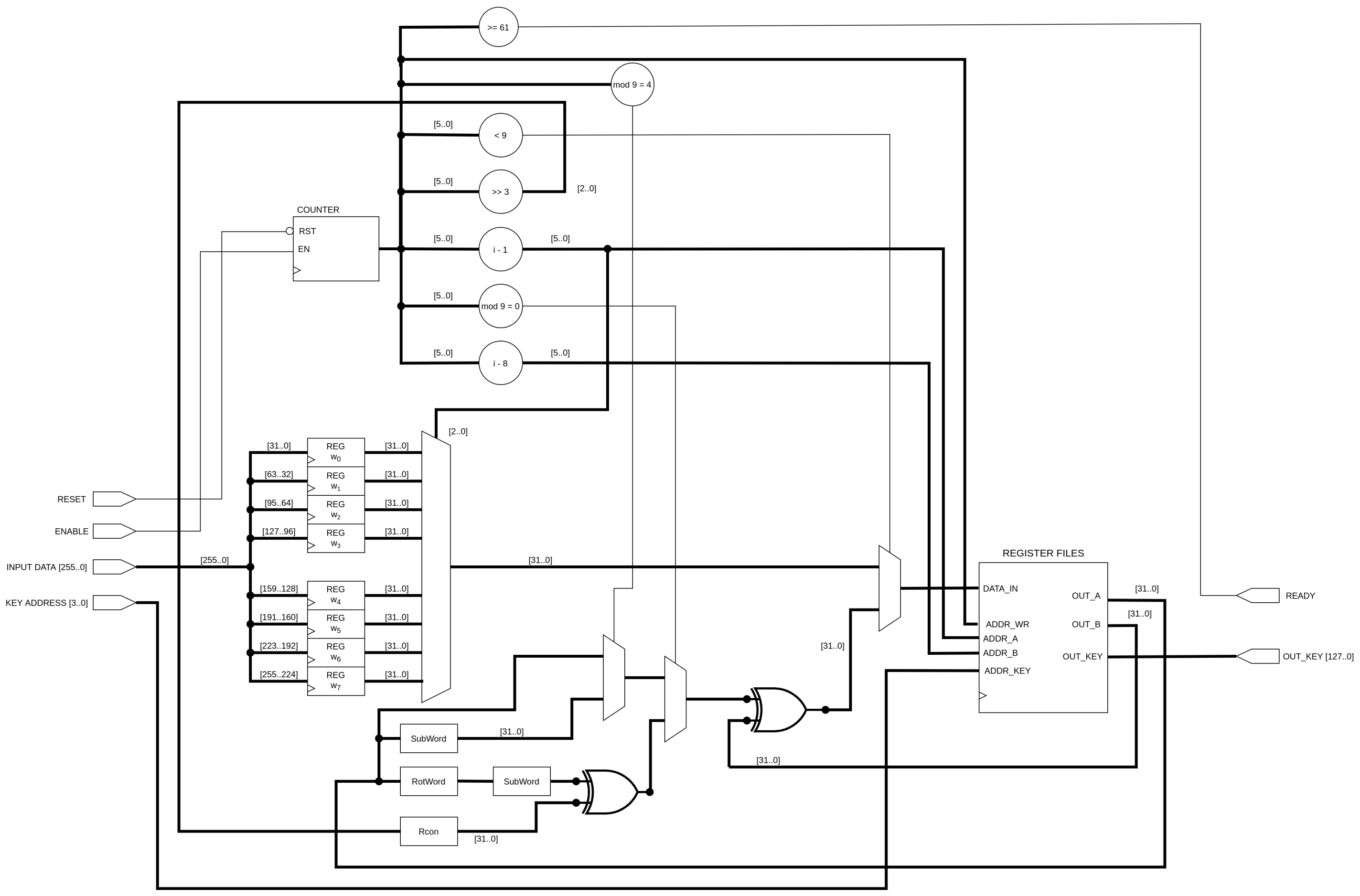


Figure 4: Diagram of Key Expansion.

1 to 8. After that, to generate the i^{th} 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 5 and figure 6, respectively.

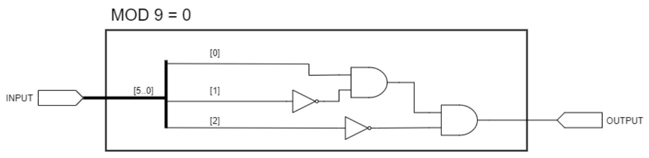


Figure 5: Circuit of *mod 9 = 0* block

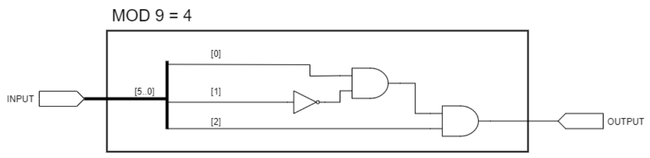


Figure 6: Circuit of *mod 9 = 4* block

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 has a value range from 0 to 14.

3.2 AES-256 Encryption

Figure 7 shows the design of AES Encryption circuit. This circuit takes 128-bit data in and a set of round keys to perform AES encryption on the inputted data.

A reset signal must be inputted in order for the circuit to work properly. A counter will keep track of which round and round key the encryption process is currently at. In the first clock cycle, inputted data will be stored in a register and added with the first round key and saved in another register that acts as the *State*, completing the initial round. In the next thirteen clock cycles, the circuit performs thirteen rounds comprising *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* transformations in each round on the *State*. In the last round, the *MixColumns* transformation is opted out. The encryption process will be completed after fifteen clock cycles and the *done* signal will be pulled high.

Details on smaller circuits in this AES Encryption circuit like *SubBytes*, *ShiftRows*, *MixColumns*, etc. are represented in the following sub-sections.

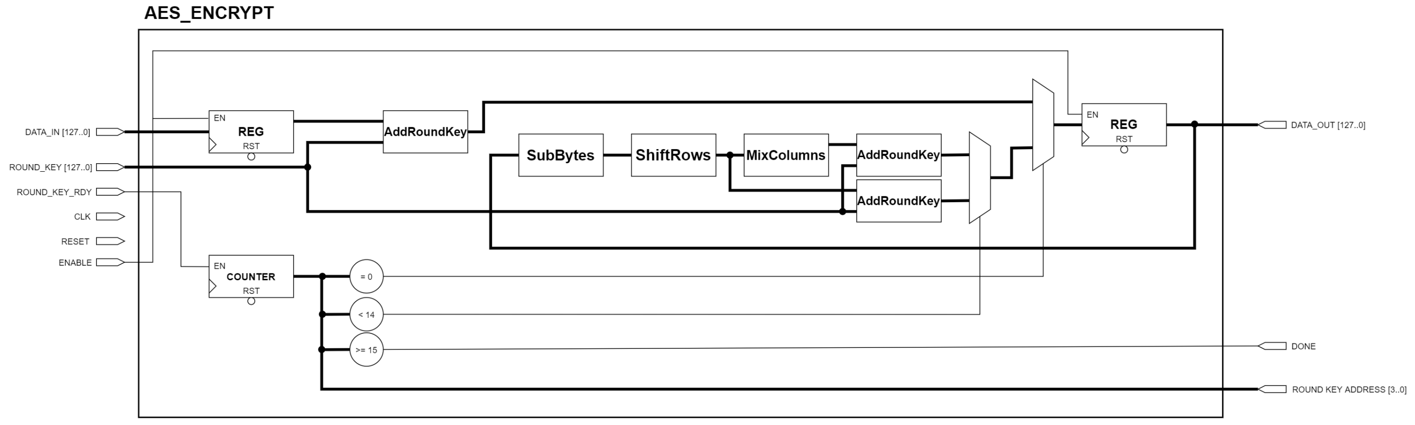


Figure 7: Diagram of AES Encryption Circuit.

3.2.1 Bytes Substitution

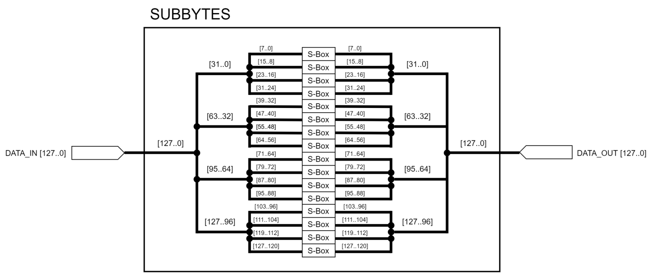


Figure 8: SubBytes circuit diagram.

The design for *SubBytes* block is shown in figure 8. 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 circuit takes 128-bit data and splits it 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.

3.2.2 Shift Rows

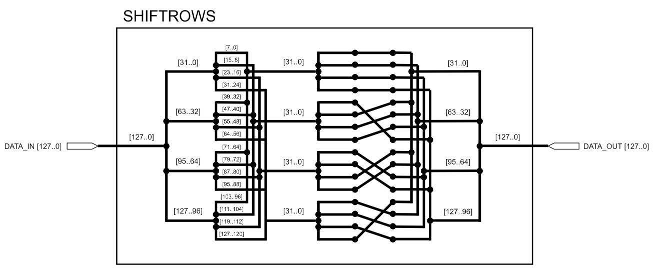


Figure 9: ShiftRows circuit diagram.

The circuit diagram for *ShiftRows* is shown in figure 9. This circuit takes 128-bit data in, performs shifting and outputs 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 2.2.2.

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

3.2.3 Mix Columns

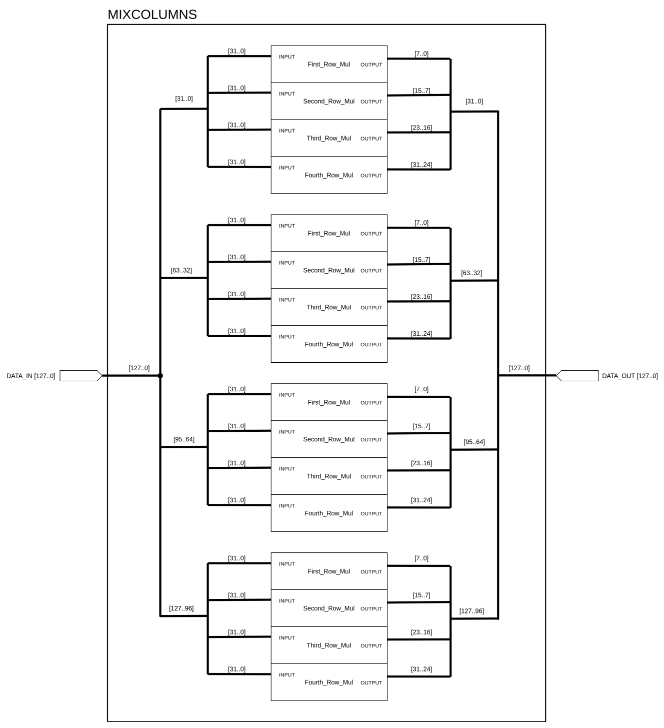


Figure 10: MixColumns circuit diagram.

In figure 10 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 2.2.3. 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.

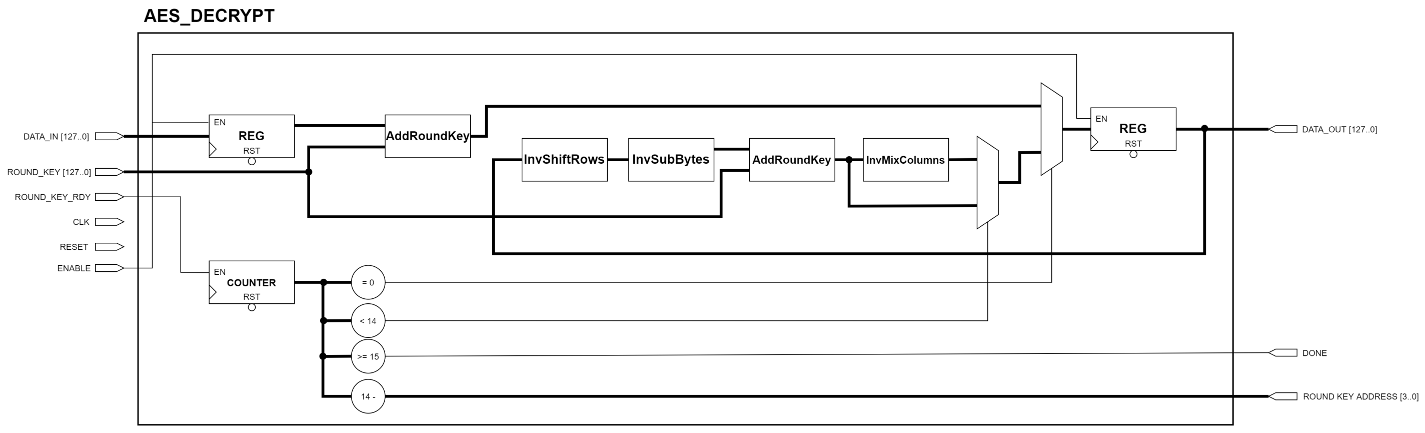


Figure 11: Diagram of AES Decryption Circuit.

The design of *First_Row_Mul*, *Second_Row_Mul*, *Third_-Row_Mul*, and *Fourth_Row_Mul* blocks is shown in figure 12, 13, 14, and 15, respectively. Note that all the addition operations (the plus symbol) in these circuits are all *XOR* operations.

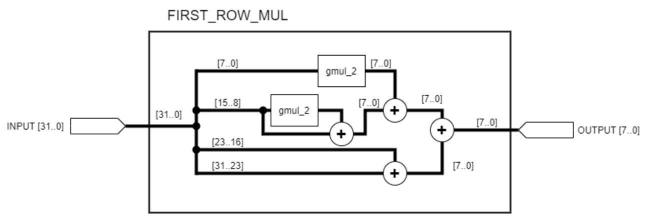


Figure 12: First_Row_Mul.

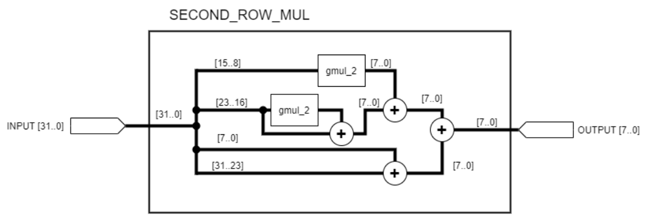


Figure 13: Second_Row_Mul.

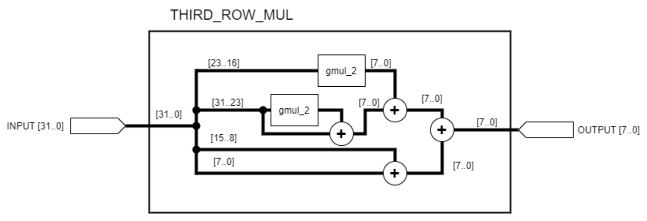


Figure 14: Third_Row_Mul.

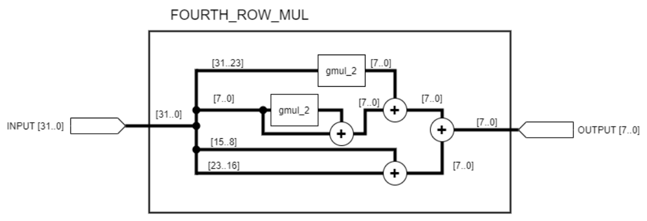


Figure 15: Fourth_Row_Mul.

The implementation of the multiplication by 2 in *GF(*2^8*)*, which is denoted as *gmul_2*, is shown in figure 16. This multiplication operation is performed at byte level as shift operation one bit to the left followed by a conditional *XOR* operation with the number *1B* in hexadecimal.

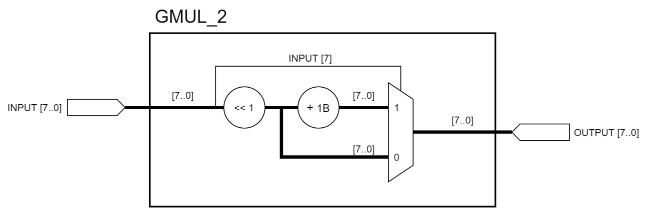


Figure 16: gmul_2.

3.3 AES-256 Decryption

3.3.1 Inverse Byte Substitution

Both circuit design for *InvSubBytes* and *SubBytes* are similar. The only difference is *InvSBox* is used instead of *SBox* in *InvSubBytes* transformation. Note that both *SBox* and *InvSBox* are implemented as LUTs.

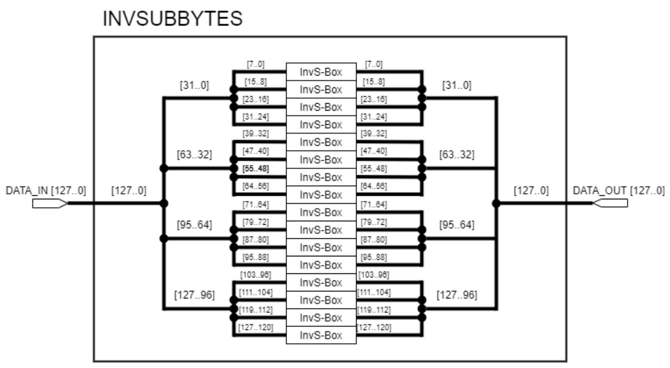


Figure 17: InvSubBytes circuit diagram.

3.3.2 Inverse Shift Rows

The circuit design for *InvShiftRows* transformation is similar to the design of *ShiftRows* transformation shown in figure 9. However, the wiring is different in the second row and the fourth row. Details can be seen in figure 18.

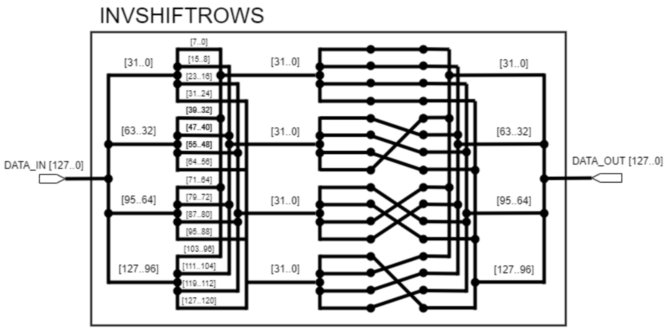


Figure 18: InvShiftRows circuit diagram.

3.3.3 Inverse Mix Columns

The circuit diagram for *InvMixColumns* transformation, which is shown in figure 19, is a bit more sophisticated than the one for *MixColumns* transformation, yet the top-level designs are still identical.

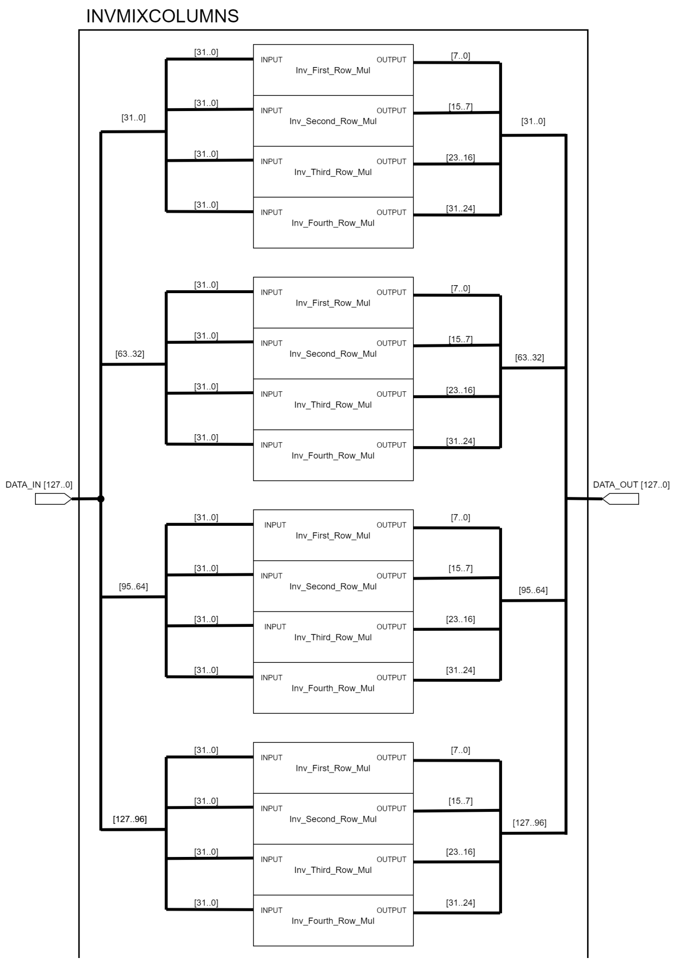


Figure 19: InvMixColumns circuit diagram.

Inv_First_Row_Mul, *Inv_Second_Row_Mul*, *Inv_Third_Row_Mul*, and *Inv_Fourth_Row_Mul* multiply a column (or a word) from the *State* with the first, second, third, and fourth row of the matrix (6) mentioned in section 2.3.3, respectively, using multiplication operation in *GF(*2^8*)*. The circuit diagrams of those blocks can be seen in figure 20, 21, 22, 23, respectively.

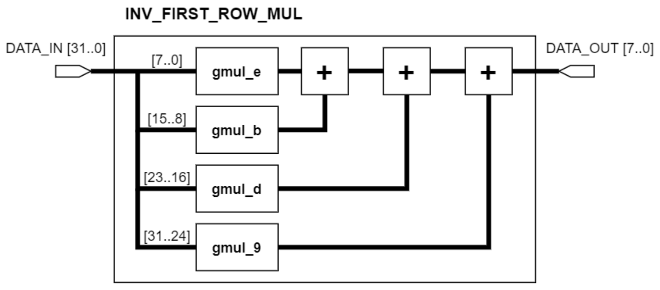


Figure 20: Inv_First_Row_Mul circuit diagram.

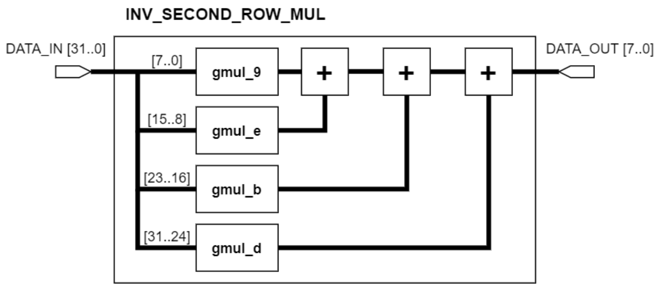


Figure 21: Inv_Second_Row_Mul circuit diagram.

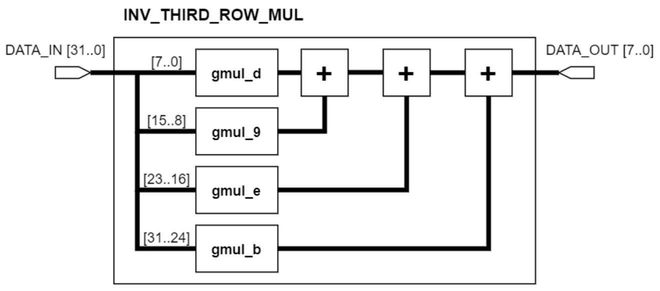


Figure 22: Inv_Third_Row_Mul circuit diagram.

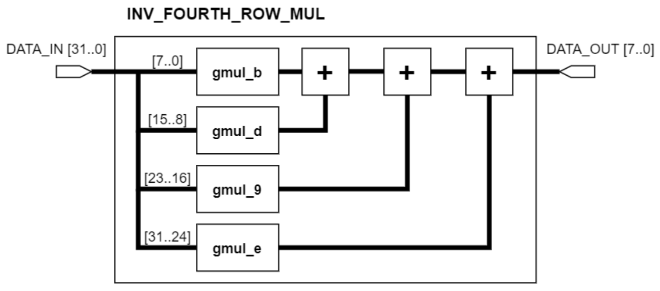


Figure 23: Inv_Fourth_Row_Mul circuit diagram.

gmul_9, *gmul_e*, *gmul_b*, and *gmul_d* multiply an input number with 9, e, b, and d, respectively, using multiplication in *GF(*2^8*)*. Note that all numbers in this section are in hexadecimal. Multiplying a number x with the number 9 in *GF(*2^8*)* can be broken down as

$$\begin{aligned} x \bullet 9 &= (x \bullet 8) \oplus x \\ &= (x \bullet 4 \bullet 2) \oplus x \\ &= (x \bullet 2 \bullet 2 \bullet 2) \oplus x \end{aligned} \tag{7}$$

Since we have created the multiplier by 2 in *GF(*2^8*)*, denoted as *gmul_2* and shown in figure 16, so we can use that to create smaller components that make up the *gmul_9* as shown in figure 24, 25, and 26.

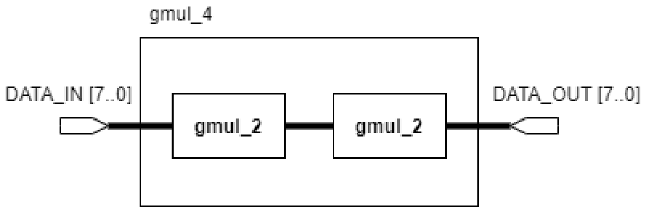


Figure 24: gmul_4 circuit diagram.

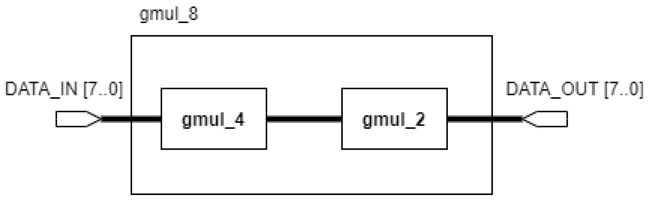


Figure 25: gmul_8 circuit diagram.

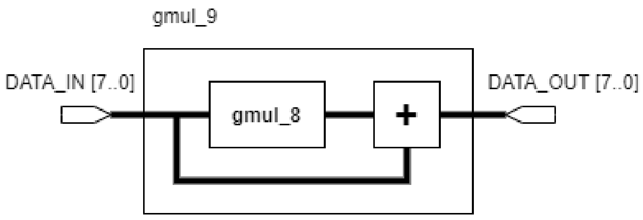


Figure 26: gmul_9 circuit diagram.

Similarly, multiplying a number x with the number b can be broken down as

$$\begin{aligned} x \bullet b &= (x \bullet 9) \oplus (x \bullet 2) \\ &= (x \bullet 8) \oplus x \oplus (x \bullet 2) \end{aligned} \tag{8}$$

Therefore, we can use the previously made *gmul_9* to make *gmul_b*.

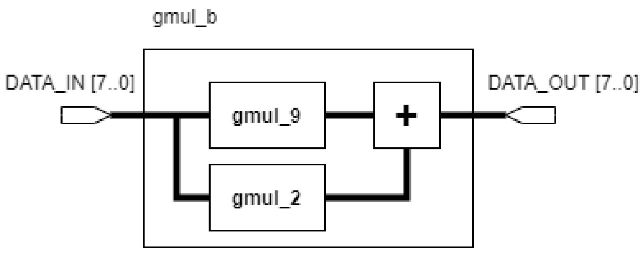


Figure 27: gmul_b circuit diagram.

Equivalently, multiplying by d and e can be rewritten as in (9) and (10), in the order mentioned.

$$x \bullet d = (x \bullet 9) \oplus (x \bullet 4) \tag{9}$$

$$x \bullet e = (x \bullet 8) \oplus (x \bullet 4) \oplus (x \bullet 2) \tag{10}$$

As a result, we can totally use previously made circuits to make *gmul_d* and *gmul_e*.

Note that all the plus signs in all the diagrams in this section represent the *XOR* operation.

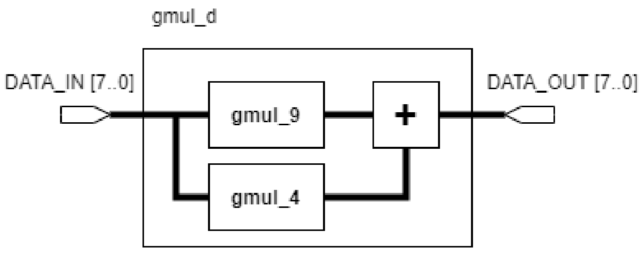


Figure 28: gmul_d circuit diagram.

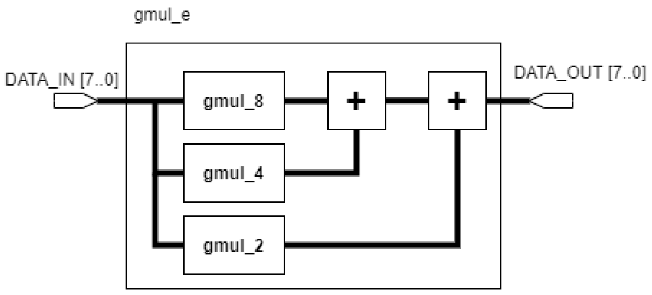


Figure 29: gmul_e circuit diagram.

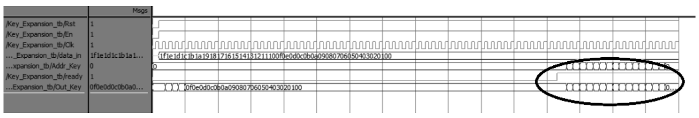
3.4 Add Round Keys

AddRoundKeys can simply be performed by the *XOR* operation with the *State* and a round key, which both have the length of 128 bits. The output is 128-bit.

4 RESULTS

4.1 Key Expansion

Figure 30 shows simulation waveform result using *ModelSim*. As can be seen in the figure, after 61 clock cycles since the last *reset* signal, the *ready* signal is set indicating round keys have been generated. Therefore, they can be retrieved correctly, which is the part circled in the figure.



```
                                    Msgs
/Key_Expansion_tb/Rst       1
/Key_Expansion_tb/En        1
/Key_Expansion_tb/Clk       1
..._Expansion_tb/data_in    1f1e1d1c1b1a1...   1f1e1d1c1b1a191817161514131211100f0e0d0c0b0a09080706050403020100
...xpansion_tb/Addr_Key     0                  0
/Key_Expansion_tb/ready     1
...Expansion_tb/Out_Key     0f0e0d0c0b0a0...   0f0e0d0c0b0a09080706050403020100
```

Figure 30: Key Expansion simulation result in *ModelSim*.

The circuit is tested with different keys from the example section in [1] and the web-based tool *https://www.cryptool.org/en/cto/aes-step-by-step*. The results produced by the circuit are the same as the expected results in each test case.

The design is synthesised and compiled on Quartus II 64-Bit for Cyclone III EP3C40F780C6 device. The result is shown in table 1.

Table 1: Synthesis & Compilation results for Key Expansion.

Total logic elements	4 942/39 600 (12%)
Total combinational functions	4 684/39 600 (12%)
Dedicated logic registers	2 310/39 600 (6%)

4.2 AES-256

Both encryption and decryption units have been tested and run simulation on *ModelSim*. The results waveform are shown in figure 31 for encryption and figure 32 for decryption.

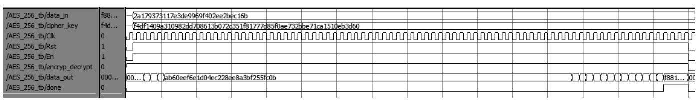

/AES_256_tb/data_in f88... 2a179373117e3de9969f402ee2bec16b
/AES_256_tb/cipher_key f4d... f4df1409a310982dd708613b072c351f81777d85f0ae732bbe71ca1510eb3d60
/AES_256_tb/Clk 0
/AES_256_tb/Rst 1
/AES_256_tb/En 1
/AES_256_tb/encryp_decrypt 0
/AES_256_tb/data_out 000... 00... ab60eef6e1d04ec228ee8a3bf255fc0b f881... 00
/AES_256_tb/done 0

Figure 31: AES-256 encryption simulation result in *ModelSim*.

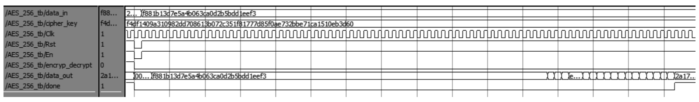

/AES_256_tb/data_in f88... 2... f881b13d7e5a4b063ca0d2b5bdd1eef3
/AES_256_tb/cipher_key f4d... f4df1409a310982dd708613b072c351f81777d85f0ae732bbe71ca1510eb3d60
/AES_256_tb/Clk 1
/AES_256_tb/Rst 1
/AES_256_tb/En 1
/AES_256_tb/encryp_decrypt 0
/AES_256_tb/data_out 2a1... 00... f881b13d7e5a4b063ca0d2b5bdd1eef3 e... 2a17...
/AES_256_tb/done 1

Figure 32: AES-256 decryption simulation result in *ModelSim*.

As can be seen in the two figures above, the done signal (the last line in both figures) is set when the circuit has done the encryption/decryption process indicating the current result is the final result. There are a couple of clock cycles needed for *Key Expansion* routine to generate round keys before the encryption/decryption process can actually start. That is the reason why it takes a fair amount of clock cycles to spit out the final output in the two figures.

The design is synthesised and compiled on Quartus II 64-Bit for Cyclone III EP3C40F780C6 device. The result is shown in table 2.

Table 2: Synthesis & Compilation results for AES-256.

Total logic elements	13 776/39 600 (35%)
Total combinational functions	12 687/39 600 (32%)
Dedicated logic registers	2 698/39 600 (7%)

5 CONCLUSION

This paper has represented the design of AES-256 that can do basic functionalities like encryption and decryption. Nevertheless, this design has not been well-optimised and thus manifested some limitations as well as other things that could be improved in future research.

The goal of this project is to make a fully functional AES-256 encryption and decryption system on FPGA. For this reason, optimising for speed and resources have not yet been taken into consideration, so the performance might be poor. Another downside of this design that it has to be reset for each time it performs encryption or decryption which leads to the re-calculation of round keys, even when the same cipher key is used for encryption and decryption, and thus wasting time as well as slowing the encryption/decryption speed down.

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.*