Efficient Hardware implementation of Advanced Encryption Standard (AES)

A

Project Report

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

Bachelor of Engineering
in
Electronics & Communication Engineering

Submitted by

Sachleen Singh Chani

Roll No. 101506143

Mentor

Dr. Vijaypal Singh Rathor

Assistant Professor

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
TIET, PATIALA-147004, PUNJAB
INDIA

June 2021

Certificate

Certified that project entitled **"*Efficient Hardware implementation of Advanced Encryption Standard (AES)*"** which is being submitted by ***Sachleen Singh Chani*** **(University Registration No.** 101506143**)** to the **Department of Electronics and Communication Engineering, TIET, Patiala, Punjab,** is a record of project work carried out by him under guidance and supervision of **Dr. Vijaypal Singh Rathor**. The matter presented in this project report does not incorporate without acknowledgement any material previously published or written by any other person except where due reference is made in the text.

Sachleen Singh Chani

101506143

Mentor

Dr. Vijaypal Singh Rathor
Assistant Professor

Acknowledgement

I feel grateful and privileged in taking this opportunity to express my gratitude to many people who helped and contributed to the completion of this project. Working on this project was a source of great knowledge and experience.

First, I would like to express my appreciation to Dr. Vijaypal Singh Rathor (mentor) for his support, continuous guidance, and his most valuable insights.

I would like to thank Dr. Alpana Agarwal (HOD) for giving me this opportunity to work on a project like this and learn.

Sachleen Singh Chani

Roll no. 101506143

Abstract

With increasing need for data security, and the growing world of data exchange, a fast, reliable, and efficient cryptographic algorithm is essential. Hardware implementation are physically secure and can achieve higher performance which is much required in the world today.

This project presents an efficient implementation of this Advanced Encryption Standard (Rijndael) on hardware using Xilinx Virtex-7 FPGA board. Reprogrammable devices such as field-programable gate arrays (FPGAs) provide physical security, agility, and better performance. There are three different formats for the AES algorithm, these are, AES-128, AES-192 and AES-256. In this project, AES-128 encryption is implemented. Each module of the implementation has been designed in Verilog-HDL and the simulations have been run on the Xilinx Vivado Design Suit 2020.2. The synthesis of the project has been done on Virtex-7 XC7VX690T target device.

For improving the efficiency of the implementation, pre-calculated Look-up tables (LUTs) have been used for the S-box and the Galois field, GF(28), multiplication in mix-columns transformations. This gives an area utilization of 0.93% (<1%) with a throughput of 0.974 Gbps and 0.33W power consumption. By using this approach, the area utilized is reduced by 62.61% comparing to the approach of the algorithm implementation in [1].

List of Figures

List of Tables

Contents

Chapter 1
Introduction

1.1 Motivation

As the world moves from the paper-based to the electronic-based data exchange and storage, there is an increasing need for the protection of that data, this has led to the development of various cryptographic algorithms which helps in secure transmission of the information.

Encryption algorithm is used to change the plain message into cipher text ready to be sent over any unsecure network, and on the receiver end decryption algorithm is used to obtain the plain message back. This process of encryption and decryption can be done by using identical keys, which is known as symmetric encryption. Another method where two different public and private keys and used to encrypt and decrypt the message respectively, this is called asymmetric/public-key encryption.

Advanced Encryption Standard (AES) is a symmetric block cryptosystem. Rijndael algorithm was designed by two people, Joan Daemen and Vincent Rijmen and was selected as the AES algorithm by the National institute of Standards and Technology (NIST) in 2001.

To not create a bottleneck on the network, algorithms must at least equivalent speeds to data transmission. These higher speeds can be achieved by using Field-programable gate arrays (FPGAs) reliably while providing algorithm agility and reliability. Although software implementation gives ease of use and upgrade, flexibility, but there are downsides to it as well such as limited physical security. To counter with the higher flexibility in software, FPGAs give an efficient alternative and they are by nature more secure from physical attacks [2].

1.2 Objective

With various possible approaches to the implementation of AES on hardware, this project explores an effective solution to decrease the resource utilization in the FPGA.

For this implementation, mixcolumn round operation was chosen to optimize as the most processing was needed to perform this operation and by the use of simple look-up tables, this processing can be reduced, which would give faster results and would reduce the area [1].

In this report, Chapter 2 describes the Background on AES, Chapter 3 goes into detail about the LUTs implementation methodology. The simulation results are presented in Chapter 4 with the conclusion is described in Chapter 5.

Chapter 2

Background and Related work

2.1 AES Algorithm

The AES algorithm is a symmetric block cipher using the Rijndael algorithm as it uses identical keys for encryption and decryption of data, it has a 128-bit block size with 128, 192 and 256 bit key sizes chosen independently [3]. The algorithm consists of several rounds, the number of which depends on the chosen length of the key, the number of rounds can be 10, 12 or 14 depending on the key length being 128-bit, 192-bit or 256-bit respectively. Each round performs internal operations in the 128-bit block data, this data is called the state. As the operations are performed on a block data, thus this algorithm in a block cipher. The state matrix [3] of a round is grouped as a 16 byte, 4x4 matrix upon which iterative operations are performed.

AES	Key Length (Nk[a] Words)	Block Size (Nb[a] Words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Table 1 Relation between key length and number of rounds. a : number of 32-bit words [1].

Each round of the algorithm has four operations, namely ‘Subbytes’, ‘Shiftrows’, ‘MixColumns’ and ‘Addroundkey’, other than the last round of encryption which doesn’t have the ‘Mixcolumns’ operation.

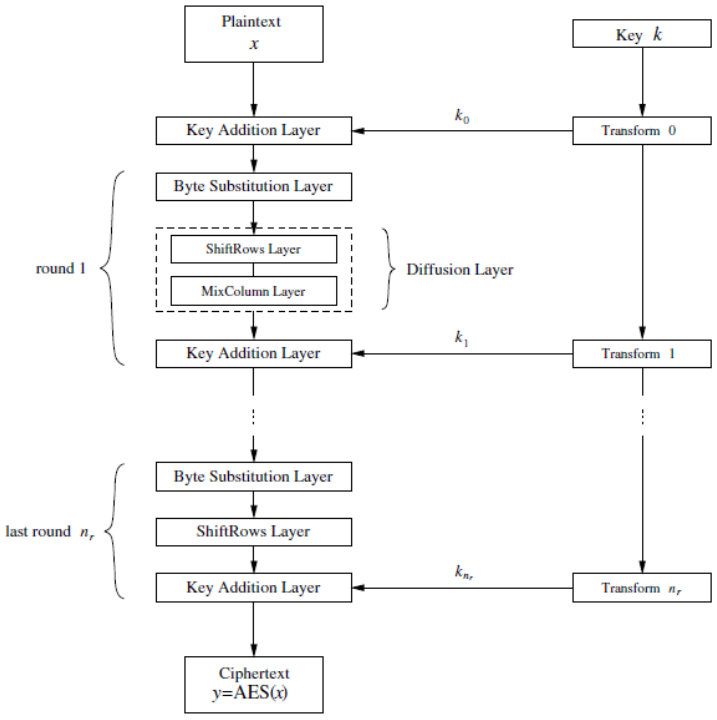

Figure 1 AES encryption block diagram [4]

The input plain text is arranged in the same 4x4 16-byte matrix. This is the input state matrix. Before the round operations start, the 4-word key is added to the input plain text and then this state matrix goes through the 10 rounds (AES-128) to give the cipher text after the last round operations.

Another process called the “Key expansion” which generates a 44-word (176 bytes) key from the original 4-word (16 bytes) key. Every 4-work round key is added to the state matrix at the end of each round in the ‘Addroundkey’ operation.

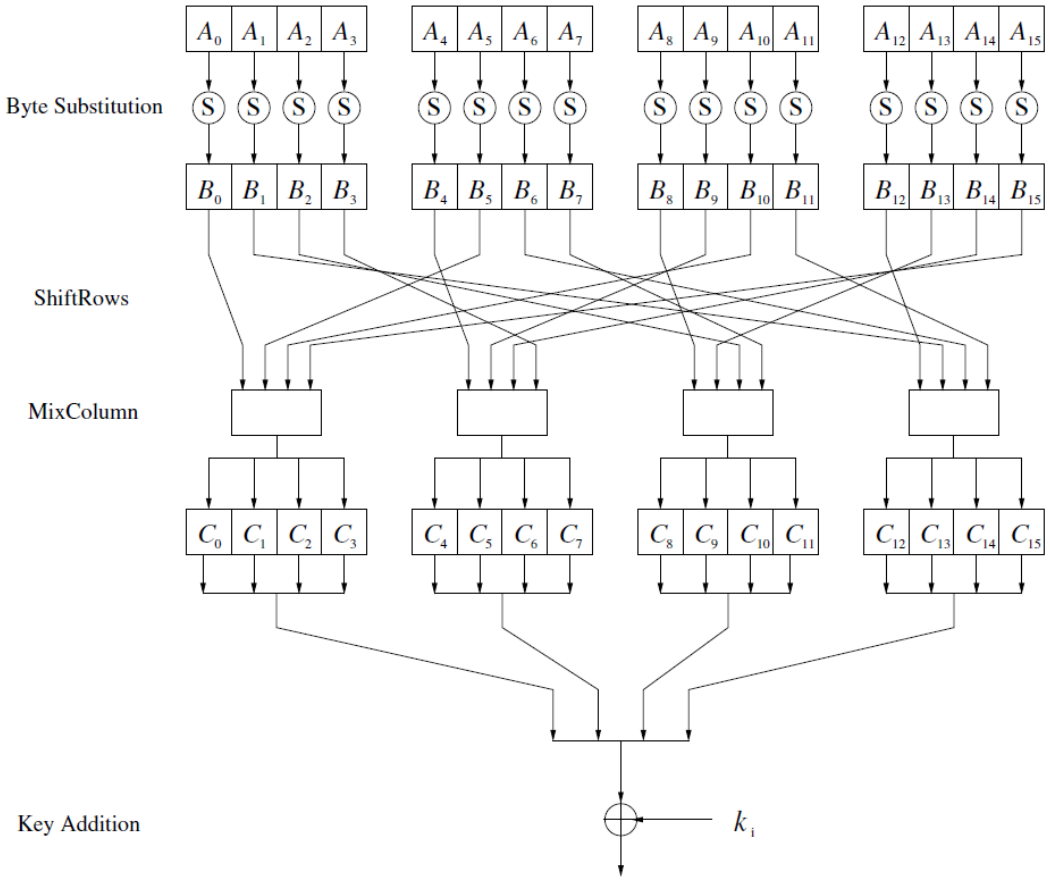

Figure 2 AES round functions [4]

2.2 Subbytes Operation

For encryption, in this operation each byte of the state matrix is replaced with a byte from the Rijndael S-box [3]. This S-box is a look-up table of 16x16 1 byte values, which could be computed by deriving the multiplicative inverse in GF(2^8) and then adding a vector to it [5]. The subbytes layer adds non-linearity in the cipher making it a tougher for targeted cryptanalysis.

The first 4 bits tell which row to select and the last 4 bits give the column to select. The new state matrix obtained after byte substitution goes to the next operation. Fig. 3 shows the S-box used in the subbyte function in the AES.

		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 3 S-box: Substitution values in base 16 [3]

2.3 Shiftrows Operation

Shiftrow operation shifts each byte in a row cyclically to the left by a certain amount depending on which row it is. The first row remains unchanged, the second is shifted to the left by an offset of 1 byte, third row is shifted to the left by 2 bytes and the fourth is shifted by an offset of 3 bytes to the left. It doesn't take any hardware area as the input wires and connected to the output by offsetting them.

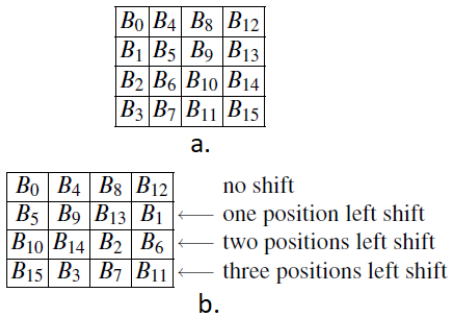

Figure 4 a) before Shift row function, b) after the function

For example, referring to the Fig. 4, the state matrix value for the 3rd row and 2nd column was S_{32}, but now after the operation is done, this new value in this place would be S_{34}.

2.4 Mixcolumns Operation

In this operation the state matrix is linear transformation which as the name suggests mixes each column of the state matrix. The state matrix is left-multiplied by another 4x4 matrix and the result gives the new state matrix.

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

Figure 5 Mixcolumn multiplication for each column [4]

Referring to Fig. 5, C_0, C_1, C_2 and C_3 are the values for the new state matrix and B_0, B_5, B_{10}, B_{15} are the input from after the shift row function was performed, referring to the Fig. 4 b).

2.5 AddroundKey Operation

In this operation a bitwise exclusive-OR (XOR) operation is performed between the output state matrix from the mixcolumn function and the round key obtained from key expansion.

2.6 Key Expansion

The input key is 128-bit in length, which is expanded to 11 round keys each of 128-bit in length.

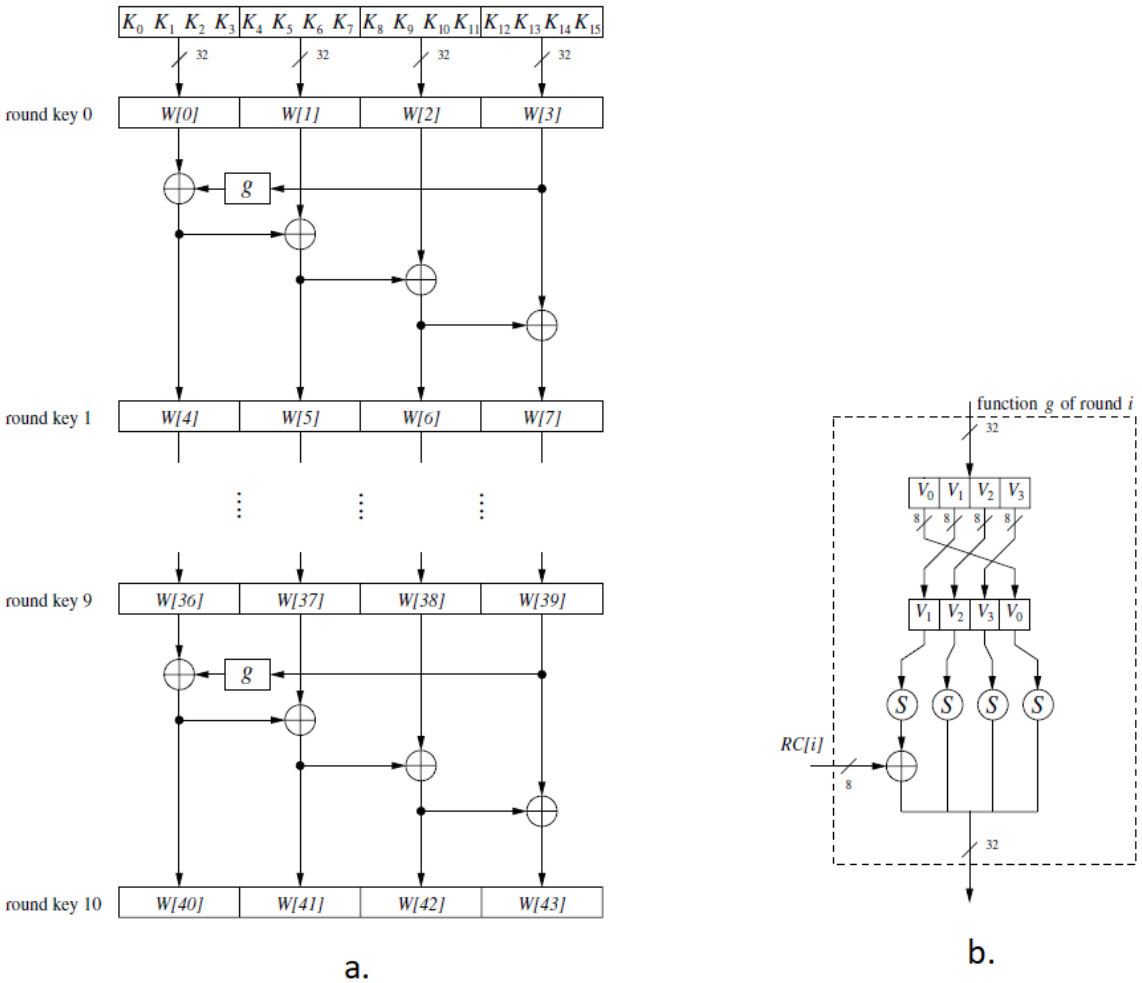

Figure 6 AES key expansion [4]

The key is taken the same into the first 4 words, the next 4 words are computed as shown in the Fig.6 a). The g-function, Fig.6 b), takes 4 bytes, and does 3 operations namely, RotWord which rotates one byte to the left, subbyte, in which we use the S-box to substitute bytes and the a RoundConstant is added. These three operations are performed to compute,

$$W[4i] = W[4(i-1)] + g(W[4i-1]) \qquad \text{where } i = 1,2,\ldots,10$$

And the remaining 3 words for the round key are computed as,

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j] \qquad \text{where } i = 1,\ldots,10 \text{ and } j = 1,2,3$$

The round constant value varies from round to round, but it is always an element of the Galois field GF(2^8). The round constant values change according to the rule:

$$RC[1] = x^0 = (00000001)_2 = 01$$
$$RC[2] = x^1 = (00000010)_2 = 02$$
$$RC[2] = x^2 = (00000100)_2 = 04$$
$$\vdots$$
$$RC[10] = x^9 = (00110110)_2 = 02$$

2.7 Related Work

To satisfy the need for various application requirements AES has been implemented on the hardware in numerous ways and methods. Verbauwhede et al. elaborates on the first AES implementation on silicon [6], for which the throughput was 2.29 Gbps using a nonpipeline architecture. This design was improved upon in Mukhopadhyay et al. by using a pipeline architecture [7], which improved the throughput to 8 Gbps.

One implementation [8], optimizes high performance by optimizing the hardware throughput. The methodology used here is the implementation of operations in parallel to increase the throughput. Through this application of multi core approach the throughput obtained was 2.21 Gbps at a frequency of 1.2 GHz.

Implementation methods in [9], [10], and [11] achieved a throughput approx. between 20 to 30 Gbps by pipelining and applying loop unrolling. The approach taken in [12] demonstrated a 73.7 Gbps throughput with a frequency of 570 MHz on a virtex-5 FPGA.

Constating to the previously mentioned implementations, a compact, low-cost approach is investigated in [13] which uses an inexpensive Spartan II FPGA and was able to achieve a data throughput of 166 Mbps with only 50% of the logic resources used.

Chapter 3

Proposed efficient implementation of AES

3.1 Methodology

The mixcolumn operation, which is the most computation heavy module of the algorithm [2] was chosen to optimize by using a different approach than the traditional method of matrix multiplication. The implementation uses ROMs to store the Galois field multiplications by 2 and 3 in encryption. By optimizing the mixcolum operation through the use of LUTs the area and power utilization is significantly improved while also improving the throughput. Fig. 7 shows the block diagram for the proposed implementation.

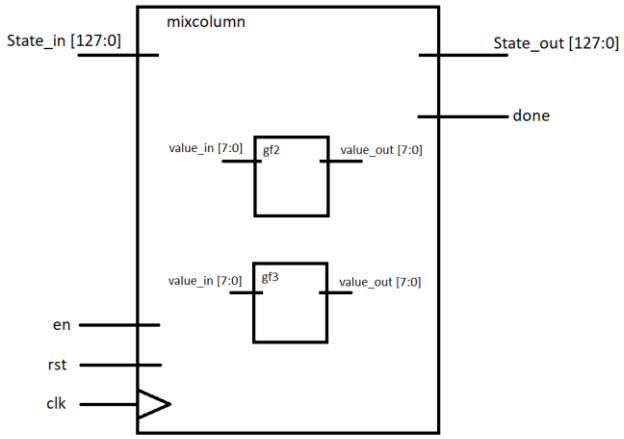

Figure 7 Mixcolumn block diagram

The Proposed design implementation is aimed at reducing the area and increasing the speed by using LUTs and ROMs for the implementation of the S-box and Mixcolumn operations. The LUT based implementation is easy to apply and after the synthesis is run, the area occupied is considerably less. [14]. For the implementation of the mixcolums as a LUT, two different modules were made, one for the Galois field GF(2^8) multiplication by '2' and the other for Galois field GF(2^8) multiplication by '3' for the encryption process. As these are the values that are computed during a multiplication in mixcolumn operation referring to Fig. 5.

Each round operation is implemented in a separate module, and all these are instantiated in the aes_encryption module which takes the plaintext and key as input and gives ciphertext as output along with a single wire signal indicating the cipher text is ready to be read.

To implement the encryption algorithm, a Finite State Machine (FSM) was designed to perform the iterative round operations.

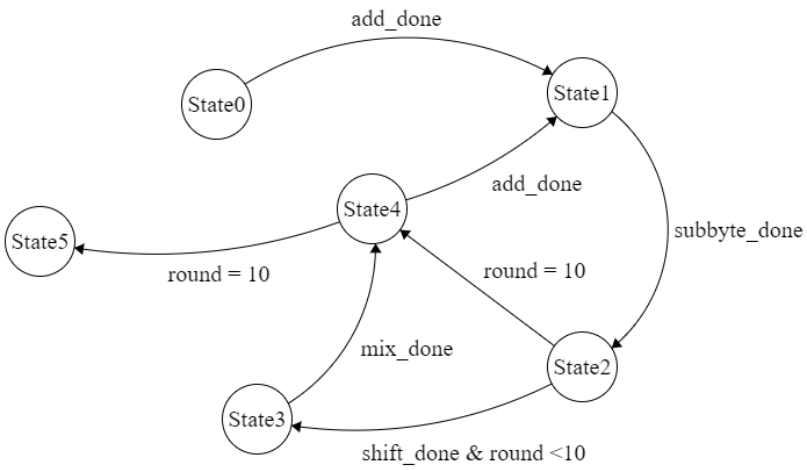

Figure 8 AES encryption FSM

State0 – Add first round key
State1 – Subbytes
State2 – Shift rows
State3 – mix columns
State4 – Add round keys
State5 –Complete

In Fig. 8, the FSM stays in the state until the *_done signal is not high indicating the function is done and the FSM can move to the next operation.

Fig. 9 shows the Top-level block diagram of the encryption algorithm implementation.

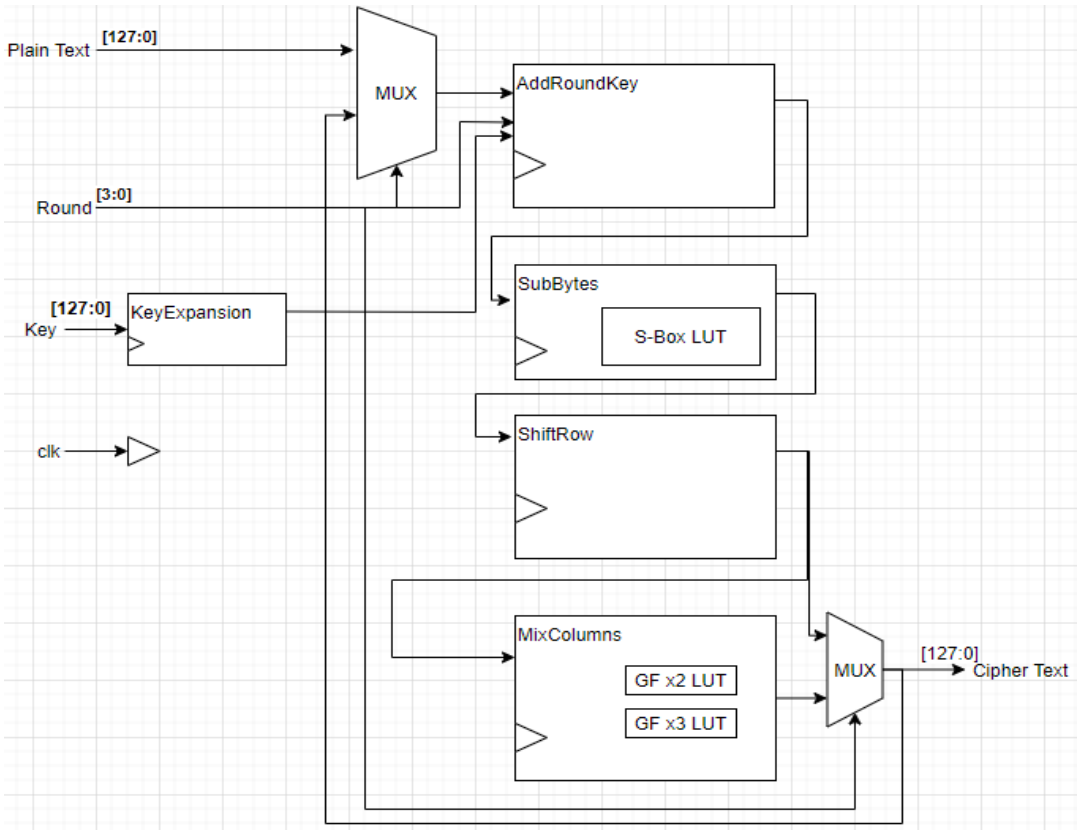

Figure 9 AES Encryption block diagram

The output of the mixcolumn and shiftrow goes into a MUX which selects which output to take depending on the current round.

The aes_encription module, the wrapper module, takes 128 bit plain text, 128 bit key along with clock, reset and enable signals as inputs and gives 128 bit ciphertext and done signal as outputs to the module. The round value is kept track by an internal counter in this same aes_encription module.

In the final round of the AES algorithm, the mixcolumn operation is not performed, to incorporate the functionality, the afore mentioned MUX is used.

For the implementation of the S-box LUT, a separate module is written as a ROM with 32-bit input and a 32-bit output, Fig. 3 shows the s-box table. This s-box is implemented as a ROM and uses 4 parallel MUXs each of 8 bits.

For implementing the mixcolumn operation, two different modules were made for the Galois field multiplication by 2 and 3, namely gf2.v and gf3.v. The code for this implementation can be found in the appendix.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
1	20	22	24	26	28	2A	2C	2E	30	32	34	36	38	3A	3C	3E
2	40	42	44	46	48	4A	4C	4E	50	52	54	56	58	5A	5C	5E
3	60	62	64	66	68	6A	6C	6E	70	72	74	76	78	7A	7C	7E
4	80	82	84	86	88	8A	8C	8E	90	92	94	96	98	9A	9C	9E
5	A0	A2	A4	A6	A8	AA	AC	AE	B0	B2	B4	B6	B8	BA	BC	BE
6	C0	C2	C4	C6	C8	CA	CC	CE	D0	D2	D4	D6	D8	DA	DC	DE
7	E0	E2	E4	E6	E8	EA	EC	EE	F0	F2	F4	F6	F8	FA	FC	FE
8	1B	19	1F	1D	13	11	17	15	0B	09	0F	0D	03	01	07	05
9	3B	39	3F	3D	33	31	37	35	2B	29	2F	2D	23	21	27	25
A	5B	59	5F	5D	53	51	57	55	4B	49	4F	4D	43	41	47	45
B	7B	79	7F	7D	73	71	77	75	6B	69	6F	6D	63	61	67	65
C	9B	99	9F	9D	93	91	97	95	8B	89	8F	8D	83	81	87	85
D	BB	B9	BF	BD	B3	B1	B7	B5	AB	A9	AF	AD	A3	A1	A7	A5
E	DB	D9	DF	DD	D3	D1	D7	D5	CB	C9	CF	CD	C3	C1	C7	C5
F	FB	F9	FF	FD	F3	F1	F7	F5	EB	E9	EF	ED	E3	E1	E7	E5

Figure 10 Galois Multiplication LUT for multiply by 2 [1]

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	00	03	06	05	0C	0F	0A	09	18	1B	1E	1D	14	17	12	11
1	30	33	36	35	3C	3F	3A	39	28	2B	2E	2D	24	27	22	21
2	60	63	66	65	6C	6F	6A	69	78	7B	7E	7D	74	77	72	71
3	50	53	56	55	5C	5F	5A	59	48	4B	4E	4D	44	47	42	41
4	C0	C3	C6	C5	CC	CF	CA	C9	D8	DB	DE	DD	D4	D7	D2	D1
5	F0	F3	F6	F5	FC	FF	FA	F9	E8	EB	EE	ED	E4	E7	E2	E1
6	A0	A3	A6	A5	AC	AF	AA	A9	B8	BB	BE	BD	B4	B7	B2	B1
7	90	93	96	95	9C	9F	9A	99	88	8B	8E	8D	84	87	82	81
8	9B	98	9D	9E	97	94	91	92	83	80	85	86	8F	8C	89	8A
9	AB	A8	AD	AE	A7	A4	A1	A2	B3	B0	B5	B6	BF	BC	B9	BA
A	FB	F8	FD	FE	F7	F4	F1	F2	E3	E0	E5	E6	EF	EC	E9	EA
B	CB	C8	CD	CE	C7	C4	C1	C2	D3	D0	D5	D6	DF	DC	D9	DA
C	5B	58	5D	5E	57	54	51	52	43	40	45	46	4F	4C	49	4A
D	6B	68	6D	6E	67	64	61	62	73	70	75	76	7F	7C	79	7A
E	3B	38	3D	3E	37	34	31	32	23	20	25	26	2F	2C	29	2A
F	0B	08	0D	0E	07	04	01	02	13	10	15	16	1F	1C	19	1A

Figure 11 Galois Multiplication LUT for multiply by 3 [1]

Fig. 10 and Fig. 11 are the LUT for the outcomes of the Galois filed multiplication operations performed in the mixcolumn function.

As an example, from Fig. 5 we can see that the value of $C_0 = \{02\}B_0 + \{03\}B_5 + \{01\}B_{10} + \{01\}B_{15}$. Assuming arbitrary values for the $B_0 = 35$, $B_5 = F5$, $B_{10} = 8C$ and $B_{15} = DA$. So, checking these values in the Fig. 9 and Fig. 10 we get,

$\{02\}B_0 = \{02\}35 = 6A$,
$\{03\}B_5 = \{03\}F5 = 04$.

This makes finding the multiplication values easier without having to compute the multiplication in the $GF(2^8)$ and uses minimum number of slices in the FPGA.

Chapter 4

Implementation and Simulation results

The proposed algorithm is simulated and synthesized using the Xilinx Vivado Design Suit 2020.2. The target Device is Virtex-7 xc7vx690tffg1761-1. All the individual operations were coded in separate modules and tested for their correct functionality. The top-level module, aes_encryption has all the logic for encryption along with the instantiations of the individual modules.

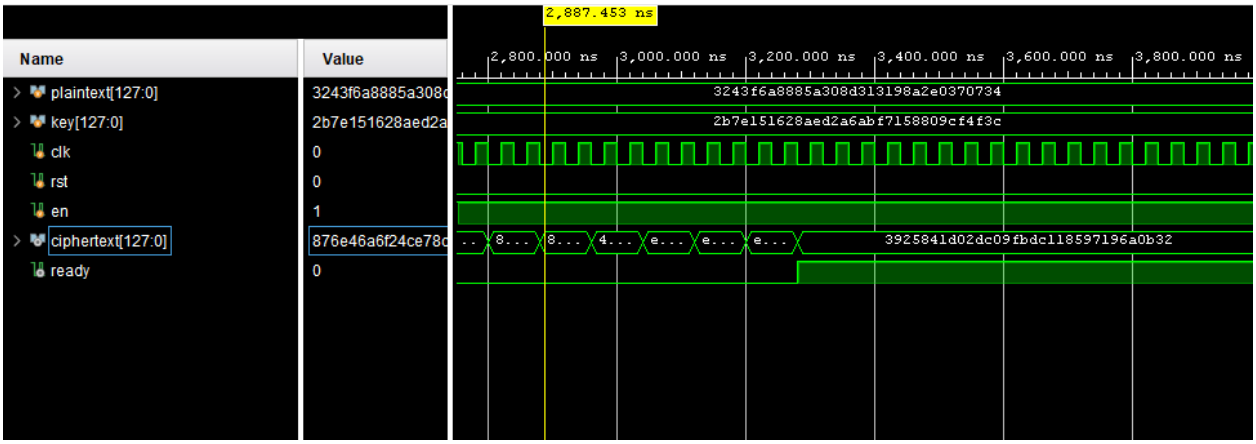

Figure 12 Encryption Timing Diagram

The ready signal goes high when the encryption is complete and the ciphertext is ready to be read. The value of plaintext and key shown in the waveform diagram in Fig. 12 have been taken from [1].

Table 2 FPGA implementation results using proposed method after optimizing the mixcolumn block

Slice Logic Utilization			
Parameters	***Used***	***Available***	***Utilization***
No. of Slice Registers	2206	866400	0.25%
No. of Slice LUTs	4027	433200	0.93%

Table 3 FPGA implementation results using the existing method [1]

Slice Logic Utilization			
Parameters	***Used***	***Available***	***Utilization***
No. of Slice Registers	3760	866400	0.43%
No. of Slice LUTs	10773	433200	2%

On comparing the results in Table 2 and Table 3 [1], we see a clear reduction in the number of Slice Registers and number of Slice LUTs being utilized.

Table 4 shows the percent reduction in each value.

Table 4 Comparison FPGA implementation results of proposed and existing methods

Resource/Parameters	***Available Resources on-board***	***Resource Utilization***		***Percent reduction compared to existing method***
		Existing method [1]	***Proposed method***	
No. of Slice Registers	866400	3760	2206	41.32%
No. of Slice LUTs	433200	10773	4027	62.61%

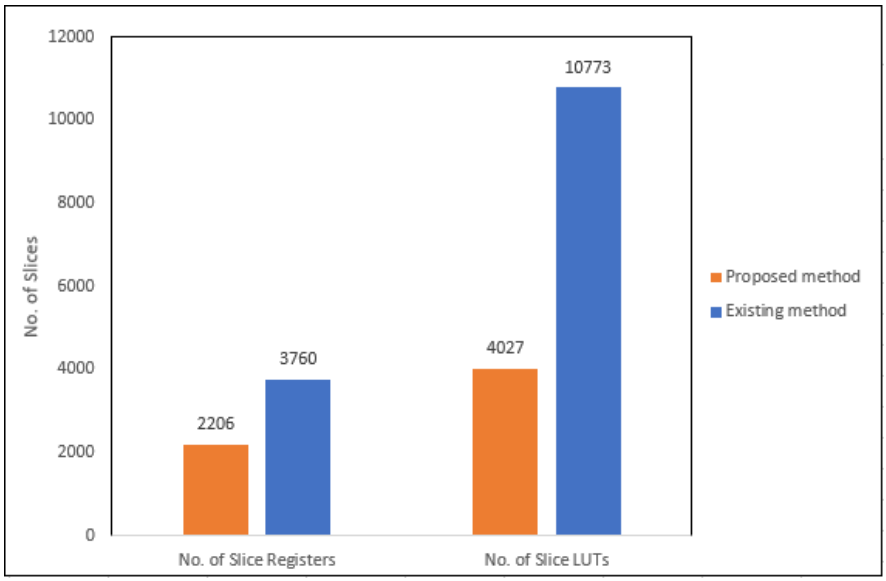

Figure 13 Result comparison between proposed method and existing method [1]

Fig. 13 shows a graphical comparison to the values obtained through the proposed implementation in this project (blue) to the results obtained for AES-128 encryption in [1] (orange)

The results obtained on the slice registers used and slice LUTs used, we can see that the utilization is 0.25% and 0.93% which is a significant reduction from [1]. The power consumption on the design is 0.33W. With 30 clock cycles per encryption to complete the operation. The maximum path delay of the design is 4.380 ns, which gives the maximum frequency of 228.31MHz. The throughput can be calculated is 0.974 Gbps using the following formula.

$$Throughput = \frac{128\ \text{bits} * \text{frequency}}{\text{clock cycles}}$$

Chapter 5

Conclusion and Future Scope

In this project, effective hardware implementation of AES is presented. AES-128 is implemented on the Virtex-7 board. It is well designed to meet the requirements of efficient design. Look-up table (LUT) based approach saves processing time and reduces the complex architecture. With the subbyte and mixcolumn operations as LUTs, the throughput of 0.974 Gbps was achieved, which is comparable to [1] AES-128 throughput. The utilization is 0.93% (<1%) with the power consumption of 0.33W.

The overall design is found to have food efficiency in terms of power, area, and throughput. This throughput is equivalent to several modern wired and wireless systems and this implementation can be incorporated in larger designs due to its small area utilization. The delay is very less and is reasonable enough to not create a bottleneck should this be incorporated at transmitter and receiver ends without affecting the data transmission speeds.

The design can be further improved in performance by using parallel processes that can be performed until they don’t require an input and can be run along with the other processes.

References

[1] N. S. S. Srinivas and M. Akramuddin, "FPGA based hardware implementation of AES Rijndael algorithm for Encryption and Decryption," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 3-5 March 2016 2016, pp. 1769-1776, doi: 10.1109/ICEEOT.2016.7754990.

[2] R. Doud, "Hardware crypto solutions Boost VPN," *Electron Eng. Times,* pp. 57-64, 1999.

[3] N. James *et al.*, "Advanced Encryption Standard (AES)," (in - en), *Journal of Research of the National Institute of Standards and Technology,* vol. 106, 3, pp. 511-577, 2001.

[4] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. 2010.

[5] P. V. S. Shastry, A. Agnihotri, D. Kachhwaha, J. Singh, and M. S. Sutaone, "A combinational logic implementation of S-box of AES," in *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 7-10 Aug. 2011 2011, pp. 1-4, doi: 10.1109/MWSCAS.2011.6026559.

[6] I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and performance testing of a 2.29-GB/s Rijndael processor," *IEEE Journal of Solid-State Circuits,* vol. 38, no. 3, pp. 569-572, 2003, doi: 10.1109/JSSC.2002.808300.

[7] D. Mukhopadhyay and D. RoyChowdhury, "An efficient end to end design of Rijndael cryptosystem in 0.18 /spl mu/ CMOS," in *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, 3-7 Jan. 2005 2005, pp. 405-410, doi: 10.1109/ICVD.2005.49.

[8] B. Liu and B. M. Baas, "Parallel AES Encryption Engines for Many-Core Processor Arrays," *IEEE Transactions on Computers,* vol. 62, no. 3, pp. 536-547, 2013, doi: 10.1109/TC.2011.251.

[9] A. Hodjat and I. Verbauwhede, "A 21.54 Gbits/s fully pipelined AES processor on FPGA," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 20-23 April 2004 2004, pp. 308-309, doi: 10.1109/FCCM.2004.1.

[10] J. Criado, M. A. Vega-Rodríguez, J. Sánchez-Pérez, and J. A. Gomez-Pulido, "A new methodology to implement the AES algorithm using partial and dynamic reconfiguration," *Integration,* vol. 43, pp. 72-80, 01/31 2010, doi: 10.1016/j.vlsi.2009.05.003.

[11] C. Chi-Jeng, H. Chi-Wu, C. Kuo-Huang, C. Yi-Cheng, and H. Chung-Cheng, "High throughput 32-bit AES implementation in FPGA," in *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, 30 Nov.-3 Dec. 2008 2008, pp. 1806-1809, doi: 10.1109/APCCAS.2008.4746393.

[12] S. Qu, G. Shou, Y. Hu, Z. Guo, and Z. Qian, "High Throughput, Pipelined Implementation of AES on FPGA," in *2009 International Symposium on Information Engineering and Electronic Commerce*, 16-17 May 2009 2009, pp. 542-545, doi: 10.1109/IEEC.2009.120.

[13] P. Chodowiec and K. Gaj, *Very compact FPGA implementation of the AES algorithm*. 2003, pp. 319-333.

[14] W. McLoone and J. V. McCanny, "Rijndael FPGA implementation utilizing look-up tables," vol. 34, pp. 349-360, 2001, doi: 10.1109/SIPS.2001.957363.

Appendix

Mixcolumn Module (mixcolumn.v)

```
`timescale 1ns / 1ps

module mixcolumn(
            input wire [127:0] state_in,
            input wire clk, rst, en,

            output wire [127:0] state_out,
            output wire done
    );


     reg [127:0] state_ff, state_nxt;
     wire [127:0] state_temp;
     reg done_ff, done_nxt;
     integer i;

     wire [7:0] gf [0:31];


     //registers
     always@(negedge clk, posedge rst) begin
        if(rst) begin
            state_ff <= 128'h0;
            done_ff <= 128'b0;
        end else begin
            state_ff <= state_nxt;
            done_ff <= done_nxt;
        end
    end

    gf2 b0 (.in(state_in[127:120]), .out(gf[0]));
    gf3 b1 (.in(state_in[119:112]), .out(gf[1]));

    gf2 b2 (.in(state_in[95:88]), .out(gf[2]));
    gf3 b3 (.in(state_in[87:80]), .out(gf[3]));

    gf2 b4 (.in(state_in[63:56]), .out(gf[4]));
    gf3 b5 (.in(state_in[55:48]), .out(gf[5]));

    gf2 b6 (.in(state_in[31:24]), .out(gf[6]));
    gf3 b7 (.in(state_in[23:16]), .out(gf[7]));


    gf2 b8 (.in(state_in[119:112]), .out(gf[8]));
    gf3 b9 (.in(state_in[111:104]), .out(gf[9]));

    gf2 ba (.in(state_in[87:80]), .out(gf[10]));
    gf3 bb (.in(state_in[79:72]), .out(gf[11]));

    gf2 bc (.in(state_in[55:48]), .out(gf[12]));
    gf3 bd (.in(state_in[47:40]), .out(gf[13]));

    gf2 be (.in(state_in[23:16]), .out(gf[14]));
    gf3 bf (.in(state_in[15:8]), .out(gf[15]));


    gf2 b16 (.in(state_in[111:104]), .out(gf[16]));
```

```
    gf3 b17 (.in(state_in[103:96]), .out(gf[17]));

    gf2 b18 (.in(state_in[79:72]), .out(gf[18]));
    gf3 b19 (.in(state_in[71:64]), .out(gf[19]));

    gf2 b20 (.in(state_in[47:40]), .out(gf[20]));
    gf3 b21 (.in(state_in[39:32]), .out(gf[21]));

    gf2 b22 (.in(state_in[15:8]), .out(gf[22]));
    gf3 b23 (.in(state_in[7:0]), .out(gf[23]));


    gf2 b24 (.in(state_in[103:96]), .out(gf[24]));
    gf3 b25 (.in(state_in[127:120]), .out(gf[25]));

    gf2 b26 (.in(state_in[71:64]), .out(gf[26]));
    gf3 b27 (.in(state_in[95:88]), .out(gf[27]));

    gf2 b28 (.in(state_in[39:32]), .out(gf[28]));
    gf3 b29 (.in(state_in[63:56]), .out(gf[29]));

    gf2 b30 (.in(state_in[7:0]), .out(gf[30]));
    gf3 b31 (.in(state_in[31:24]), .out(gf[31]));


    assign state_temp[127:120] = gf[0] ^ gf[1] ^ state_in[111:104] ^ state_in[103:96];
    assign state_temp[119:112] = state_in[127:120] ^ gf[8] ^ gf[9] ^ state_in[103:96];
    assign state_temp[111:104] = state_in[127:120] ^ state_in[119:112] ^ gf[16] ^
gf[17];
    assign state_temp[103:96]  = gf[25] ^ state_in[119:112] ^ state_in[111:104] ^
gf[24];

    assign state_temp[95:88] = gf[2] ^ gf[3] ^ state_in[99:72] ^ state_in[71:64];
    assign state_temp[87:80] = state_in[95:88] ^ gf[10] ^ gf[11] ^ state_in[71:64];
    assign state_temp[79:72] = state_in[95:88] ^ state_in[87:80] ^ gf[18] ^ gf[19];
    assign state_temp[71:64] = gf[27] ^ state_in[87:80] ^ state_in[79:72] ^ gf[26];

    assign state_temp[63:56] = gf[4] ^ gf[5] ^ state_in[47:40] ^ state_in[39:32];
    assign state_temp[55:48] = state_in[63:56] ^ gf[12] ^ gf[13] ^ state_in[39:32];
    assign state_temp[47:40] = state_in[63:56] ^ state_in[55:48] ^ gf[20] ^ gf[21];
    assign state_temp[39:32] = gf[29] ^ state_in[55:48] ^ state_in[47:40] ^ gf[28];

    assign state_temp[31:24] = gf[6] ^ gf[7] ^ state_in[15:8] ^ state_in[7:0];
    assign state_temp[23:16] = state_in[31:24] ^ gf[14] ^ gf[15] ^ state_in[7:0];
    assign state_temp[15:8] = state_in[31:24] ^ state_in[23:16] ^ gf[22] ^ gf[23];
    assign state_temp[7:0] = gf[31] ^ state_in[23:16] ^ state_in[15:8] ^ gf[30];



    //next state logic
    always@* begin
         if(en) begin
            for(i=0;i<=15;i=i+1) begin
                state_nxt[i*8+:8] = state_temp[i*8+:8];
            end
            done_nxt = 1'b1;
         end else begin
            done_nxt = 1'b0;
         end

    end
```

```
    //output logic
    assign state_out = state_ff;
    assign done = done_ff;

endmodule
```

Galois field multiplication module for multiply by 2 (gf2)

```
`timescale 1ns / 1ps

module gf2(
            input wire [7:0] in,
            output wire [7:0] out
    );


     wire [7:0] mul_2 [0:255];


    assign out = mul_2[in];


    assign mul_2[8'h00] = 8'h00;
    assign mul_2[8'h01] = 8'h02;
    assign mul_2[8'h02] = 8'h04;
    assign mul_2[8'h03] = 8'h06;
    assign mul_2[8'h04] = 8'h08;
    assign mul_2[8'h05] = 8'h0a;
    assign mul_2[8'h06] = 8'h0c;
    assign mul_2[8'h07] = 8'h0e;
    assign mul_2[8'h08] = 8'h10;
    assign mul_2[8'h09] = 8'h12;
    assign mul_2[8'h0a] = 8'h14;
    assign mul_2[8'h0b] = 8'h16;
    assign mul_2[8'h0c] = 8'h18;
    assign mul_2[8'h0d] = 8'h1a;
    assign mul_2[8'h0e] = 8'h1c;
    assign mul_2[8'h0f] = 8'h1e;
    assign mul_2[8'h10] = 8'h20;
    assign mul_2[8'h11] = 8'h22;
    assign mul_2[8'h12] = 8'h24;
    assign mul_2[8'h13] = 8'h26;
    assign mul_2[8'h14] = 8'h28;
    assign mul_2[8'h15] = 8'h2A;
    assign mul_2[8'h16] = 8'h2C;
    assign mul_2[8'h17] = 8'h2E;
    assign mul_2[8'h18] = 8'h30;
    assign mul_2[8'h19] = 8'h32;
    assign mul_2[8'h1a] = 8'h34;
    assign mul_2[8'h1b] = 8'h36;
    assign mul_2[8'h1c] = 8'h38;
    assign mul_2[8'h1d] = 8'h3A;
    assign mul_2[8'h1e] = 8'h3C;
    assign mul_2[8'h1f] = 8'h3E;
    assign mul_2[8'h20] = 8'h40;
    assign mul_2[8'h21] = 8'h42;
    assign mul_2[8'h22] = 8'h44;
    assign mul_2[8'h23] = 8'h46;
    assign mul_2[8'h24] = 8'h48;
    assign mul_2[8'h25] = 8'h4a;
    assign mul_2[8'h26] = 8'h4c;
```

```
assign mul_2[8'h27] = 8'h4e;
assign mul_2[8'h28] = 8'h50;
assign mul_2[8'h29] = 8'h52;
assign mul_2[8'h2a] = 8'h54;
assign mul_2[8'h2b] = 8'h56;
assign mul_2[8'h2c] = 8'h58;
assign mul_2[8'h2d] = 8'h5a;
assign mul_2[8'h2e] = 8'h5c;
assign mul_2[8'h2f] = 8'h5e;
assign mul_2[8'h30] = 8'h60;
assign mul_2[8'h31] = 8'h62;
assign mul_2[8'h32] = 8'h64;
assign mul_2[8'h33] = 8'h66;
assign mul_2[8'h34] = 8'h68;
assign mul_2[8'h35] = 8'h6a;
assign mul_2[8'h36] = 8'h6c;
assign mul_2[8'h37] = 8'h6e;
assign mul_2[8'h38] = 8'h70;
assign mul_2[8'h39] = 8'h72;
assign mul_2[8'h3a] = 8'h74;
assign mul_2[8'h3b] = 8'h76;
assign mul_2[8'h3c] = 8'h78;
assign mul_2[8'h3d] = 8'h7a;
assign mul_2[8'h3e] = 8'h7c;
assign mul_2[8'h3f] = 8'h7e;
assign mul_2[8'h40] = 8'h80;
assign mul_2[8'h41] = 8'h82;
assign mul_2[8'h42] = 8'h84;
assign mul_2[8'h43] = 8'h86;
assign mul_2[8'h44] = 8'h88;
assign mul_2[8'h45] = 8'h8a;
assign mul_2[8'h46] = 8'h8c;
assign mul_2[8'h47] = 8'h8e;
assign mul_2[8'h48] = 8'h90;
assign mul_2[8'h49] = 8'h92;
assign mul_2[8'h4a] = 8'h94;
assign mul_2[8'h4b] = 8'h96;
assign mul_2[8'h4c] = 8'h98;
assign mul_2[8'h4d] = 8'h9a;
assign mul_2[8'h4e] = 8'h9c;
assign mul_2[8'h4f] = 8'h9e;
assign mul_2[8'h50] = 8'ha0;
assign mul_2[8'h51] = 8'ha2;
assign mul_2[8'h52] = 8'ha4;
assign mul_2[8'h53] = 8'ha6;
assign mul_2[8'h54] = 8'ha8;
assign mul_2[8'h55] = 8'haa;
assign mul_2[8'h56] = 8'hac;
assign mul_2[8'h57] = 8'hae;
assign mul_2[8'h58] = 8'hb0;
assign mul_2[8'h59] = 8'hb2;
assign mul_2[8'h5a] = 8'hb4;
assign mul_2[8'h5b] = 8'hb6;
assign mul_2[8'h5c] = 8'hb8;
assign mul_2[8'h5d] = 8'hba;
assign mul_2[8'h5e] = 8'hbc;
assign mul_2[8'h5f] = 8'hbe;
assign mul_2[8'h60] = 8'hc0;
assign mul_2[8'h61] = 8'hc2;
assign mul_2[8'h62] = 8'hc4;
assign mul_2[8'h63] = 8'hc6;
assign mul_2[8'h64] = 8'hc8;
assign mul_2[8'h65] = 8'hca;
```

```
assign mul_2[8'h66] = 8'hcc;
assign mul_2[8'h67] = 8'hce;
assign mul_2[8'h68] = 8'hd0;
assign mul_2[8'h69] = 8'hd2;
assign mul_2[8'h6a] = 8'hd4;
assign mul_2[8'h6b] = 8'hd6;
assign mul_2[8'h6c] = 8'hd8;
assign mul_2[8'h6d] = 8'hda;
assign mul_2[8'h6e] = 8'hdc;
assign mul_2[8'h6f] = 8'hde;
assign mul_2[8'h70] = 8'he0;
assign mul_2[8'h71] = 8'he2;
assign mul_2[8'h72] = 8'he4;
assign mul_2[8'h73] = 8'he6;
assign mul_2[8'h74] = 8'he8;
assign mul_2[8'h75] = 8'hea;
assign mul_2[8'h76] = 8'hec;
assign mul_2[8'h77] = 8'hee;
assign mul_2[8'h78] = 8'hf0;
assign mul_2[8'h79] = 8'hf2;
assign mul_2[8'h7a] = 8'hf4;
assign mul_2[8'h7b] = 8'hf6;
assign mul_2[8'h7c] = 8'hf8;
assign mul_2[8'h7d] = 8'hfa;
assign mul_2[8'h7e] = 8'hfc;
assign mul_2[8'h7f] = 8'hfe;
assign mul_2[8'h80] = 8'h1b;
assign mul_2[8'h81] = 8'h19;
assign mul_2[8'h82] = 8'h1f;
assign mul_2[8'h83] = 8'h1d;
assign mul_2[8'h84] = 8'h13;
assign mul_2[8'h85] = 8'h11;
assign mul_2[8'h86] = 8'h17;
assign mul_2[8'h87] = 8'h15;
assign mul_2[8'h88] = 8'h0b;
assign mul_2[8'h89] = 8'h09;
assign mul_2[8'h8a] = 8'h0f;
assign mul_2[8'h8b] = 8'h0d;
assign mul_2[8'h8c] = 8'h03;
assign mul_2[8'h8d] = 8'h01;
assign mul_2[8'h8e] = 8'h07;
assign mul_2[8'h8f] = 8'h05;
assign mul_2[8'h90] = 8'h3b;
assign mul_2[8'h91] = 8'h39;
assign mul_2[8'h92] = 8'h3f;
assign mul_2[8'h93] = 8'h3d;
assign mul_2[8'h94] = 8'h33;
assign mul_2[8'h95] = 8'h31;
assign mul_2[8'h96] = 8'h37;
assign mul_2[8'h97] = 8'h35;
assign mul_2[8'h98] = 8'h2b;
assign mul_2[8'h99] = 8'h29;
assign mul_2[8'h9a] = 8'h2f;
assign mul_2[8'h9b] = 8'h2d;
assign mul_2[8'h9c] = 8'h23;
assign mul_2[8'h9d] = 8'h21;
assign mul_2[8'h9e] = 8'h27;
assign mul_2[8'h9f] = 8'h25;
assign mul_2[8'ha0] = 8'h5b;
assign mul_2[8'ha1] = 8'h59;
assign mul_2[8'ha2] = 8'h5f;
assign mul_2[8'ha3] = 8'h5d;
assign mul_2[8'ha4] = 8'h53;
```

```
assign mul_2[8'ha5] = 8'h51;
assign mul_2[8'ha6] = 8'h57;
assign mul_2[8'ha7] = 8'h55;
assign mul_2[8'ha8] = 8'h4b;
assign mul_2[8'ha9] = 8'h49;
assign mul_2[8'haa] = 8'h4f;
assign mul_2[8'hab] = 8'h4d;
assign mul_2[8'hac] = 8'h43;
assign mul_2[8'had] = 8'h41;
assign mul_2[8'hae] = 8'h47;
assign mul_2[8'haf] = 8'h45;
assign mul_2[8'hb0] = 8'h7b;
assign mul_2[8'hb1] = 8'h79;
assign mul_2[8'hb2] = 8'h7f;
assign mul_2[8'hb3] = 8'h7d;
assign mul_2[8'hb4] = 8'h73;
assign mul_2[8'hb5] = 8'h71;
assign mul_2[8'hb6] = 8'h77;
assign mul_2[8'hb7] = 8'h75;
assign mul_2[8'hb8] = 8'h6b;
assign mul_2[8'hb9] = 8'h69;
assign mul_2[8'hba] = 8'h6f;
assign mul_2[8'hbb] = 8'h6d;
assign mul_2[8'hbc] = 8'h63;
assign mul_2[8'hbd] = 8'h61;
assign mul_2[8'hbe] = 8'h67;
assign mul_2[8'hbf] = 8'h65;
assign mul_2[8'hc0] = 8'h9b;
assign mul_2[8'hc1] = 8'h99;
assign mul_2[8'hc2] = 8'h9f;
assign mul_2[8'hc3] = 8'h9d;
assign mul_2[8'hc4] = 8'h93;
assign mul_2[8'hc5] = 8'h91;
assign mul_2[8'hc6] = 8'h97;
assign mul_2[8'hc7] = 8'h95;
assign mul_2[8'hc8] = 8'h8b;
assign mul_2[8'hc9] = 8'h89;
assign mul_2[8'hca] = 8'h8f;
assign mul_2[8'hcb] = 8'h8d;
assign mul_2[8'hcc] = 8'h83;
assign mul_2[8'hcd] = 8'h81;
assign mul_2[8'hce] = 8'h87;
assign mul_2[8'hcf] = 8'h85;
assign mul_2[8'hd0] = 8'hbb;
assign mul_2[8'hd1] = 8'hb9;
assign mul_2[8'hd2] = 8'hbf;
assign mul_2[8'hd3] = 8'hbd;
assign mul_2[8'hd4] = 8'hb3;
assign mul_2[8'hd5] = 8'hb1;
assign mul_2[8'hd6] = 8'hb7;
assign mul_2[8'hd7] = 8'hb5;
assign mul_2[8'hd8] = 8'hab;
assign mul_2[8'hd9] = 8'ha9;
assign mul_2[8'hda] = 8'haf;
assign mul_2[8'hdb] = 8'had;
assign mul_2[8'hdc] = 8'ha3;
assign mul_2[8'hdd] = 8'ha1;
assign mul_2[8'hde] = 8'ha7;
assign mul_2[8'hdf] = 8'ha5;
assign mul_2[8'he0] = 8'hdb;
assign mul_2[8'he1] = 8'hd9;
assign mul_2[8'he2] = 8'hdf;
assign mul_2[8'he3] = 8'hdd;
```

```
    assign mul_2[8'he4] = 8'hd3;
    assign mul_2[8'he5] = 8'hd1;
    assign mul_2[8'he6] = 8'hd7;
    assign mul_2[8'he7] = 8'hd5;
    assign mul_2[8'he8] = 8'hcb;
    assign mul_2[8'he9] = 8'hc9;
    assign mul_2[8'hea] = 8'hcf;
    assign mul_2[8'heb] = 8'hcd;
    assign mul_2[8'hec] = 8'hc3;
    assign mul_2[8'hed] = 8'hc1;
    assign mul_2[8'hee] = 8'hc7;
    assign mul_2[8'hef] = 8'hc5;
    assign mul_2[8'hf0] = 8'hfb;
    assign mul_2[8'hf1] = 8'hf9;
    assign mul_2[8'hf2] = 8'hff;
    assign mul_2[8'hf3] = 8'hfd;
    assign mul_2[8'hf4] = 8'hf3;
    assign mul_2[8'hf5] = 8'hf1;
    assign mul_2[8'hf6] = 8'hf7;
    assign mul_2[8'hf7] = 8'hf5;
    assign mul_2[8'hf8] = 8'heb;
    assign mul_2[8'hf9] = 8'he9;
    assign mul_2[8'hfa] = 8'hef;
    assign mul_2[8'hfb] = 8'hed;
    assign mul_2[8'hfc] = 8'he3;
    assign mul_2[8'hfd] = 8'he1;
    assign mul_2[8'hfe] = 8'he7;
    assign mul_2[8'hff] = 8'he5;

endmodule
```

Galois field multiplication module for multiply by 3 (gf3)

```
`timescale 1ns / 1ps
module gf3(
            input wire [7:0] in,
            output wire [7:0] out
    );


     wire [7:0] mul_3 [0:255];


    assign out = mul_3[in];

   assign mul_3[8'h00] = 8'h00;
   assign mul_3[8'h01] = 8'h03;
   assign mul_3[8'h02] = 8'h06;
   assign mul_3[8'h03] = 8'h05;
   assign mul_3[8'h04] = 8'h0c;
   assign mul_3[8'h05] = 8'h0f;
   assign mul_3[8'h06] = 8'h0a;
   assign mul_3[8'h07] = 8'h09;
   assign mul_3[8'h08] = 8'h18;
   assign mul_3[8'h09] = 8'h1b;
   assign mul_3[8'h0a] = 8'h1e;
   assign mul_3[8'h0b] = 8'h1d;
   assign mul_3[8'h0c] = 8'h14;
   assign mul_3[8'h0d] = 8'h17;
   assign mul_3[8'h0e] = 8'h12;
   assign mul_3[8'h0f] = 8'h11;
   assign mul_3[8'h10] = 8'h30;
   assign mul_3[8'h11] = 8'h33;
```

```
assign mul_3[8'h12] = 8'h36;
assign mul_3[8'h13] = 8'h35;
assign mul_3[8'h14] = 8'h3c;
assign mul_3[8'h15] = 8'h3f;
assign mul_3[8'h16] = 8'h3a;
assign mul_3[8'h17] = 8'h39;
assign mul_3[8'h18] = 8'h28;
assign mul_3[8'h19] = 8'h2b;
assign mul_3[8'h1a] = 8'h2e;
assign mul_3[8'h1b] = 8'h2d;
assign mul_3[8'h1c] = 8'h24;
assign mul_3[8'h1d] = 8'h27;
assign mul_3[8'h1e] = 8'h22;
assign mul_3[8'h1f] = 8'h21;
assign mul_3[8'h20] = 8'h60;
assign mul_3[8'h21] = 8'h63;
assign mul_3[8'h22] = 8'h66;
assign mul_3[8'h23] = 8'h65;
assign mul_3[8'h24] = 8'h6c;
assign mul_3[8'h25] = 8'h6f;
assign mul_3[8'h26] = 8'h6a;
assign mul_3[8'h27] = 8'h69;
assign mul_3[8'h28] = 8'h78;
assign mul_3[8'h29] = 8'h7b;
assign mul_3[8'h2a] = 8'h7e;
assign mul_3[8'h2b] = 8'h7d;
assign mul_3[8'h2c] = 8'h74;
assign mul_3[8'h2d] = 8'h77;
assign mul_3[8'h2e] = 8'h72;
assign mul_3[8'h2f] = 8'h71;
assign mul_3[8'h30] = 8'h50;
assign mul_3[8'h31] = 8'h53;
assign mul_3[8'h32] = 8'h56;
assign mul_3[8'h33] = 8'h55;
assign mul_3[8'h34] = 8'h5c;
assign mul_3[8'h35] = 8'h5f;
assign mul_3[8'h36] = 8'h5a;
assign mul_3[8'h37] = 8'h59;
assign mul_3[8'h38] = 8'h48;
assign mul_3[8'h39] = 8'h4b;
assign mul_3[8'h3a] = 8'h4e;
assign mul_3[8'h3b] = 8'h4d;
assign mul_3[8'h3c] = 8'h44;
assign mul_3[8'h3d] = 8'h47;
assign mul_3[8'h3e] = 8'h42;
assign mul_3[8'h3f] = 8'h41;
assign mul_3[8'h40] = 8'hc0;
assign mul_3[8'h41] = 8'hc3;
assign mul_3[8'h42] = 8'hc6;
assign mul_3[8'h43] = 8'hc5;
assign mul_3[8'h44] = 8'hcc;
assign mul_3[8'h45] = 8'hcf;
assign mul_3[8'h46] = 8'hca;
assign mul_3[8'h47] = 8'hc9;
assign mul_3[8'h48] = 8'hd8;
assign mul_3[8'h49] = 8'hdb;
assign mul_3[8'h4a] = 8'hde;
assign mul_3[8'h4b] = 8'hdd;
assign mul_3[8'h4c] = 8'hd4;
assign mul_3[8'h4d] = 8'hd7;
assign mul_3[8'h4e] = 8'hd2;
assign mul_3[8'h4f] = 8'hd1;
assign mul_3[8'h50] = 8'hf0;
```

```
assign mul_3[8'h51] = 8'hf3;
assign mul_3[8'h52] = 8'hf6;
assign mul_3[8'h53] = 8'hf5;
assign mul_3[8'h54] = 8'hfc;
assign mul_3[8'h55] = 8'hff;
assign mul_3[8'h56] = 8'hfa;
assign mul_3[8'h57] = 8'hf9;
assign mul_3[8'h58] = 8'he8;
assign mul_3[8'h59] = 8'heb;
assign mul_3[8'h5a] = 8'hee;
assign mul_3[8'h5b] = 8'hed;
assign mul_3[8'h5c] = 8'he4;
assign mul_3[8'h5d] = 8'he7;
assign mul_3[8'h5e] = 8'he2;
assign mul_3[8'h5f] = 8'he1;
assign mul_3[8'h60] = 8'ha0;
assign mul_3[8'h61] = 8'ha3;
assign mul_3[8'h62] = 8'ha6;
assign mul_3[8'h63] = 8'ha5;
assign mul_3[8'h64] = 8'hac;
assign mul_3[8'h65] = 8'haf;
assign mul_3[8'h66] = 8'haa;
assign mul_3[8'h67] = 8'ha9;
assign mul_3[8'h68] = 8'hb8;
assign mul_3[8'h69] = 8'hbb;
assign mul_3[8'h6a] = 8'hbe;
assign mul_3[8'h6b] = 8'hbd;
assign mul_3[8'h6c] = 8'hb4;
assign mul_3[8'h6d] = 8'hb7;
assign mul_3[8'h6e] = 8'hb2;
assign mul_3[8'h6f] = 8'hb1;
assign mul_3[8'h70] = 8'h90;
assign mul_3[8'h71] = 8'h93;
assign mul_3[8'h72] = 8'h96;
assign mul_3[8'h73] = 8'h95;
assign mul_3[8'h74] = 8'h9c;
assign mul_3[8'h75] = 8'h9f;
assign mul_3[8'h76] = 8'h9a;
assign mul_3[8'h77] = 8'h99;
assign mul_3[8'h78] = 8'h88;
assign mul_3[8'h79] = 8'h8b;
assign mul_3[8'h7a] = 8'h8e;
assign mul_3[8'h7b] = 8'h8d;
assign mul_3[8'h7c] = 8'h84;
assign mul_3[8'h7d] = 8'h87;
assign mul_3[8'h7e] = 8'h82;
assign mul_3[8'h7f] = 8'h81;
assign mul_3[8'h80] = 8'h9b;
assign mul_3[8'h81] = 8'h98;
assign mul_3[8'h82] = 8'h9d;
assign mul_3[8'h83] = 8'h9e;
assign mul_3[8'h84] = 8'h97;
assign mul_3[8'h85] = 8'h94;
assign mul_3[8'h86] = 8'h91;
assign mul_3[8'h87] = 8'h92;
assign mul_3[8'h88] = 8'h83;
assign mul_3[8'h89] = 8'h80;
assign mul_3[8'h8a] = 8'h85;
assign mul_3[8'h8b] = 8'h86;
assign mul_3[8'h8c] = 8'h8f;
assign mul_3[8'h8d] = 8'h8c;
assign mul_3[8'h8e] = 8'h89;
assign mul_3[8'h8f] = 8'h8a;
```

```
assign mul_3[8'h90] = 8'hab;
assign mul_3[8'h91] = 8'ha8;
assign mul_3[8'h92] = 8'had;
assign mul_3[8'h93] = 8'hae;
assign mul_3[8'h94] = 8'ha7;
assign mul_3[8'h95] = 8'ha4;
assign mul_3[8'h96] = 8'ha1;
assign mul_3[8'h97] = 8'ha2;
assign mul_3[8'h98] = 8'hb3;
assign mul_3[8'h99] = 8'hb0;
assign mul_3[8'h9a] = 8'hb5;
assign mul_3[8'h9b] = 8'hb6;
assign mul_3[8'h9c] = 8'hbf;
assign mul_3[8'h9d] = 8'hbc;
assign mul_3[8'h9e] = 8'hb9;
assign mul_3[8'h9f] = 8'hba;
assign mul_3[8'ha0] = 8'hfb;
assign mul_3[8'ha1] = 8'hf8;
assign mul_3[8'ha2] = 8'hfd;
assign mul_3[8'ha3] = 8'hfe;
assign mul_3[8'ha4] = 8'hf7;
assign mul_3[8'ha5] = 8'hf4;
assign mul_3[8'ha6] = 8'hf1;
assign mul_3[8'ha7] = 8'hf2;
assign mul_3[8'ha8] = 8'he3;
assign mul_3[8'ha9] = 8'he0;
assign mul_3[8'haa] = 8'he5;
assign mul_3[8'hab] = 8'he6;
assign mul_3[8'hac] = 8'hef;
assign mul_3[8'had] = 8'hec;
assign mul_3[8'hae] = 8'he9;
assign mul_3[8'haf] = 8'hea;
assign mul_3[8'hb0] = 8'hcb;
assign mul_3[8'hb1] = 8'hc8;
assign mul_3[8'hb2] = 8'hcd;
assign mul_3[8'hb3] = 8'hce;
assign mul_3[8'hb4] = 8'hc7;
assign mul_3[8'hb5] = 8'hc4;
assign mul_3[8'hb6] = 8'hc1;
assign mul_3[8'hb7] = 8'hc2;
assign mul_3[8'hb8] = 8'hd3;
assign mul_3[8'hb9] = 8'hd0;
assign mul_3[8'hba] = 8'hd5;
assign mul_3[8'hbb] = 8'hd6;
assign mul_3[8'hbc] = 8'hdf;
assign mul_3[8'hbd] = 8'hdc;
assign mul_3[8'hbe] = 8'hd9;
assign mul_3[8'hbf] = 8'hda;
assign mul_3[8'hc0] = 8'h5b;
assign mul_3[8'hc1] = 8'h58;
assign mul_3[8'hc2] = 8'h5d;
assign mul_3[8'hc3] = 8'h5e;
assign mul_3[8'hc4] = 8'h57;
assign mul_3[8'hc5] = 8'h54;
assign mul_3[8'hc6] = 8'h51;
assign mul_3[8'hc7] = 8'h52;
assign mul_3[8'hc8] = 8'h43;
assign mul_3[8'hc9] = 8'h40;
assign mul_3[8'hca] = 8'h45;
assign mul_3[8'hcb] = 8'h46;
assign mul_3[8'hcc] = 8'h4f;
assign mul_3[8'hcd] = 8'h4c;
assign mul_3[8'hce] = 8'h49;
```

```
    assign mul_3[8'hcf] = 8'h4a;
    assign mul_3[8'hd0] = 8'h6b;
    assign mul_3[8'hd1] = 8'h68;
    assign mul_3[8'hd2] = 8'h6d;
    assign mul_3[8'hd3] = 8'h6e;
    assign mul_3[8'hd4] = 8'h67;
    assign mul_3[8'hd5] = 8'h64;
    assign mul_3[8'hd6] = 8'h61;
    assign mul_3[8'hd7] = 8'h62;
    assign mul_3[8'hd8] = 8'h73;
    assign mul_3[8'hd9] = 8'h70;
    assign mul_3[8'hda] = 8'h75;
    assign mul_3[8'hdb] = 8'h76;
    assign mul_3[8'hdc] = 8'h7f;
    assign mul_3[8'hdd] = 8'h7c;
    assign mul_3[8'hde] = 8'h79;
    assign mul_3[8'hdf] = 8'h7a;
    assign mul_3[8'he0] = 8'h3b;
    assign mul_3[8'he1] = 8'h38;
    assign mul_3[8'he2] = 8'h3d;
    assign mul_3[8'he3] = 8'h3e;
    assign mul_3[8'he4] = 8'h37;
    assign mul_3[8'he5] = 8'h34;
    assign mul_3[8'he6] = 8'h31;
    assign mul_3[8'he7] = 8'h32;
    assign mul_3[8'he8] = 8'h23;
    assign mul_3[8'he9] = 8'h20;
    assign mul_3[8'hea] = 8'h25;
    assign mul_3[8'heb] = 8'h26;
    assign mul_3[8'hec] = 8'h2f;
    assign mul_3[8'hed] = 8'h2c;
    assign mul_3[8'hee] = 8'h29;
    assign mul_3[8'hef] = 8'h2a;
    assign mul_3[8'hf0] = 8'h0b;
    assign mul_3[8'hf1] = 8'h08;
    assign mul_3[8'hf2] = 8'h0d;
    assign mul_3[8'hf3] = 8'h0e;
    assign mul_3[8'hf4] = 8'h07;
    assign mul_3[8'hf5] = 8'h04;
    assign mul_3[8'hf6] = 8'h01;
    assign mul_3[8'hf7] = 8'h02;
    assign mul_3[8'hf8] = 8'h13;
    assign mul_3[8'hf9] = 8'h10;
    assign mul_3[8'hfa] = 8'h15;
    assign mul_3[8'hfb] = 8'h16;
    assign mul_3[8'hfc] = 8'h1f;
    assign mul_3[8'hfd] = 8'h1c;
    assign mul_3[8'hfe] = 8'h19;
    assign mul_3[8'hff] = 8'h1a;


endmodule
```