Low area implementation of AES on FPGA

Group Member:

Yukun Liu, Yuche Huang, Congbo Zhang, Jiahan Zhang

Abstract-

The goal of this project is to design a low-area AES encryption/decryption system written by team members in both C and VHDL languages. Unlike high-throughput designs, low-area designs do not pursue higher and faster throughput. However, for operating frequency, the pursuit of lower occupancy and power consumption. Ideally, the design should eventually be applied to the FPGA. The purpose of this report is to introduce the design ideas of our team and how each part works.

1. Introduction

The Advanced Encryption Standard (AES) is a symmetric cipher, also known as Rijndae cryptography in cryptography. It is a block cipher standard designed to replace the original DES standard. Published by the National Institute of Standards and Technology (NIST) in 2002. Unlike its predecessor, the standard DES, AES uses a replacement-replacement network

instead of a Feisty architecture. AES can be quickly encrypted and decrypted in both software and hardware. It is highly efficient and only occupies a small memory. As a new encryption standard, it has a good market prospect. In addition, as the rapidly growth of technology, FPGA prices begin to decline. Therefore, the AES encryption and decryption platform through FPGA design has become popular. Unlike CPLDs, the reconfigurability of FPGAs reduces development costs and makes debugging easier. Over the time, FPGA high-throughput and low-area AES systems have become a trend.

For this paper, AES system is divided into four parts - SubBytes (Yuzhe Huang), ShiftRows (Congbo Zhang), MixColumns (Yukun Liu), Keyschedule (Jiahan Zhang). Each of the group

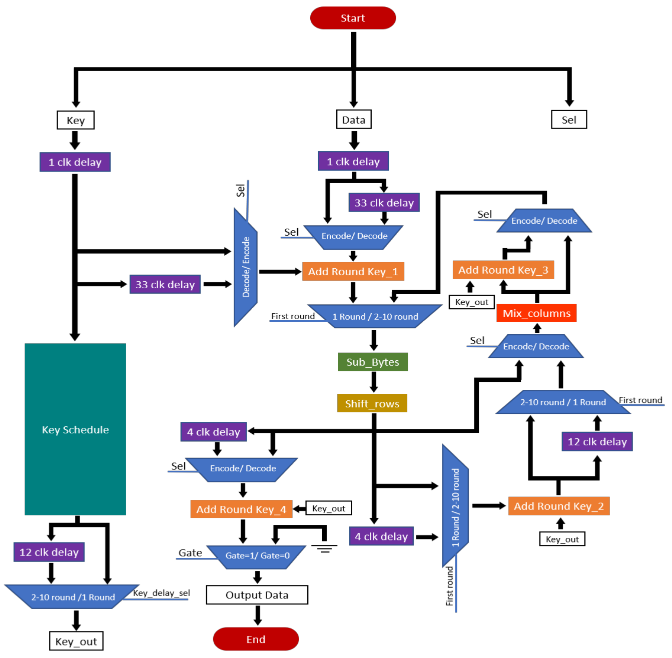


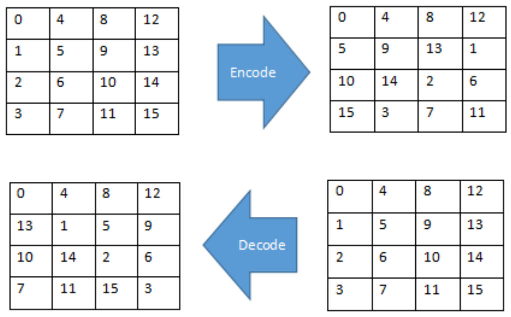
Figure 11. the designed architecture of AES

member is responsible for one part, and finally integrated by Huang, and completed the specific functions. The design ideas of each part will be explained in detail below. Theoretically, SubBytes uses a S-box to complete the byte-to-byte substitution of the packet. ShiftRows is a regular replacement of the matrix. MixColumns is replaced by the arithmetic properties of the Galois Field GF, while Keyschedule is a bitwise XOR of the current packet and part of the extended key.

I. IMPLEMENTAITON OF DESGINED ARCHITECTURE

3.1 ShiftRows

The ShiftRows section provides a simple, regular replacement that confuses the 4x4 matrix for encryption. The specific order of scrambling is shown below. The decryption part of ShiftRows is relatively simple. It is basically same as the inverse of the encryption step. It needs to restore the array that was scrambled in the previous encryption back to the previous state.



0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

0	4	8	12
5	9	13	1
10	14	2	6
15	3	7	11

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Figure 1 Encoding and decoding

Whether it is encryption or decryption, the difference is only reflected in the address of the shift register. In order to achieve such a function, we need a 32-bit auto-addressable shift register, and then 8 such SLRCs are connected in parallel to share the address for output. The primitives of SRLC32 are directly available in the Spartan 6 device library, whereas in the later versions of FPGA Spartan 3 there are only SLRC16 components. Therefore, we need two SLRC16 plus two MUX cascades to form an SRLC32. Such a SLRC has six address terminals (A5A4A3A2A1A0), in which two address terminals of A1A0 are always 1. When the system is initialized, the ShiftRows counter is set to zero, the data begins to enter the register with each clock, and the counter is incremented by 1. When the 12th CLK is reached, the preload ends, the SLR outputs the first result, and then 16 outputs are continuously output. Then the counter resets and starts a new loop.

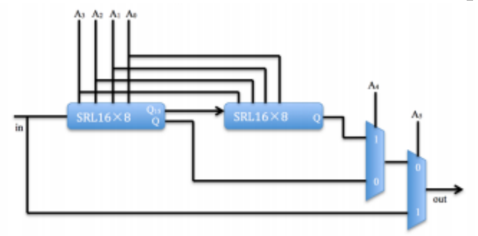


Figure 2 Structure of SLRC16[1]

Let's take a look at some of the improvements we made in the design process for this part of ShiftRows. In the initial version, I removed the second MUX and tried to reduce the area of this part of the circuit, because the only function of the A5 address (MUX2) is to directly output the value of the input signal at the 16th (12 4) CLK. Although this reduces the area, the CLK required for preloading changes from 12 to 13, which reduces the speed of the system after integrating the parts. So we added a second MUX later. The second thing is that we used SLRC16 in the device library before, but then we wrote a code for SLR16. Although we didn't have the simplicity of the component library, it can show our own design ideas.

3.2 SubBytes

The Substitution-box (S-Box) is a basic component which provides the symmetric algorithm to generate strong security to prevent data from cryptanalysis. The conventional method for S-box implementation suffers from an unbreakable delay and occupied a significant amount of memory allocation due to adopt a looking up table (TLB) which stores 8 bits for all 256 values in a ROM. Alternatively, to optimize the size of SubBytes on FPGA, this project exploits multiplicative inversion which is adopted by both encryption and decryption to share the function and using two multiplexers to select encoding and decoding path to contribute a low area application on FPGA implementation.

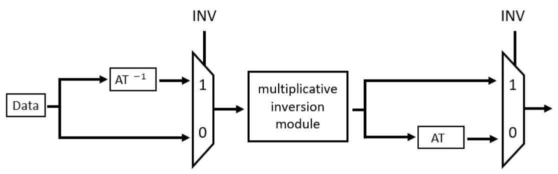


Figure 3. SubBytes Encryption/ Decryption Block

Figure 3 demonstrates the structure of SubBytes which is composed of three main blocks: the affine transformation, affine inversion and multiplicative inversion. According to the previous research [1], the work demonstrates the key to optimization the circuit space is the multiplicative inversion which is the most costly operation in S-box. As a result, the work exploits the composite field arithmetic and isomorphic to implement the multiplicative inversion to achieve a lower source utilization.

To implement the multiplicative inversion in GF(2^8), it occupies approximately 620 gates with larger critical path delay [2]. In order to achieve low area application, by using the composite field arithmetic, GF(2^8) operation is decomposed into $GF((2^4)^2)$ and $GF(((2^2)^2)^2)$ which provide lower gates count utilization to alleviate the issue.

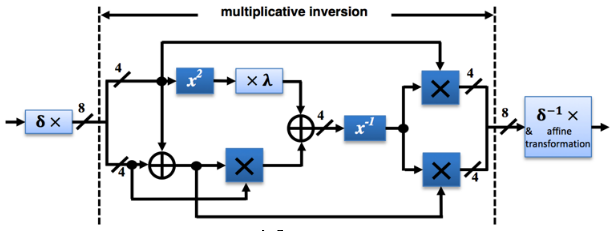


Figure 4. $GF((2^4)^2)$ Multiplicative Inversion

According to the previous work [3], it demonstrates the principle to implement the $GF((2^4)^2)$ multiplicative inversion by using Euclidian's Extended algorithm as shown in **Figure 4** Each block in this module can be achieved with simple logic block gates. Based on the characteristic, the GF (2^4) operation can be assembled by using GF (2^2) which is represented by some simple AND gate.

Originally, the shift register was adopted to implement both Affine transformation its inversion due to the next element of the row can be generated by shifting with the previous one **Figure 5** However, with this methodology, the source utilization increases evidently due to exploit a considerable number of logic gates when using registers to store the constant shifting data. Alternatively, by describing the operation directly with Boolean algebra for calculating each bit. It provides more intuitive and efficient for understanding and source utilization.

$$AT(a)=\begin{pmatrix}1&1&1&1&1&0&0&0\\0&1&1&1&1&1&0&0\\0&0&1&1&1&1&1&0\\0&0&0&1&1&1&1&1\\1&0&0&0&1&1&1&1\\1&1&0&0&0&1&1&1\\1&1&1&0&0&0&1&1\\1&1&1&1&0&0&0&1\end{pmatrix}\times\begin{pmatrix}a_7\\a_6\\a_5\\a_4\\a_3\\a_2\\a_1\\a_0\end{pmatrix}\oplus\begin{pmatrix}0\\1\\1\\0\\0\\0\\1\\1\end{pmatrix}\quad AT^{-1}(a)=\begin{pmatrix}0&1&0&1&0&0&1&0\\0&0&1&0&1&0&0&1\\1&0&0&1&0&1&0&0\\0&1&0&0&1&0&1&0\\0&0&1&0&0&1&0&1\\1&0&0&1&0&0&1&0\\0&1&0&0&1&0&0&1\\1&0&1&0&0&1&0&0\end{pmatrix}\times\begin{pmatrix}a_7\\a_6\\a_5\\a_4\\a_3\\a_2\\a_1\\a_0\end{pmatrix}\oplus\begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\1\end{pmatrix}$$

Figure 5 Affine Transformation Operation

In order to increase throughput, initially, the s-Box was divided into two stages which is set in multiplicative inversion module for pipelining. However, these delays caused by the D-type flip-flop will increase the complexity of the circuit design when combining into AES system due to the CLK cycle for key schedule generating the new around key is the same when data pass through ShiftRow and MixColumn. Therefore, we finally decide to abandon using registers for pipelining in S-Box.

3.3 Mixcolumns

REGISTER	CLOCK PULSE			
	t=0	t=1	t=2	t=3
R_0	x_0	$x_0 \oplus x_1$	$\{03\}x_0 \oplus x_1 \oplus x_2$	$\{02\}x_0 \oplus \{03\}x_1 \oplus x_2 \oplus x_3$
R_1	x_0	$\{03\}x_0 \oplus x_1$	$\{02\}x_0 \oplus \{03\}x_1 \oplus x_2$	$x_0 \oplus \{02\}x_1 \oplus \{03\}x_2 \oplus x_3$
R_2	$\{03\}x_0$	$\{02\}x_0 \oplus \{03\}x_1$	$x_0 \oplus \{02\}x_1 \oplus \{03\}x_2$	$x_0 \oplus x_1 \oplus \{02\}x_2 \oplus \{03\}x_3$
R_3	$\{02\}x_0$	$x_0 \oplus \{02\}x_1$	$x_0 \oplus x_1 \oplus \{02\}x_2$	$\{03\}x_0 \oplus x_1 \oplus x_2 \oplus \{02\}x_3$

Table 1. the behavior of each colck cycle

Figure 6 shows the architecture of Mixcolumns transformation. It works in a byte systolic manner with serial input data. The system totally needs 4 clock cycles to generate a new column as every cycle has one bytes is fed into the system, it results in a full block Mixcolumn transformation is accomplished every 16 clock cycles with 4 stage pipelining. After multiplication, the data goes through the cyclically shifted process and the intermediate results are stored in the 4 registers (R0~R3). The intermediate results are an addition of the new byte and a cyclic shift of previous bytes (en1=logic 1). Those processes continue until the fourth clock, while the parallel to serial shift register will be enabled and start to send out the stored data. Pin "En2" control the shift register, while Pin "En1" control the feedback function, and "Decryption" decide the system to do encryption or decryption (encryption=0 and decryption=1). Technically, the operation of Encryption and Decryption are using same hardware layout and process, but with different value of Galois field multiplication, while the encryption is applied $\{02\}_H$ and$\{03\}_H$. F$\{09\}_H$, $\{0d\}_H$, $\{11\}_H$and $\{14\}_H$. According to the process, the behavior of each cycle is shown in **Table 1,** $x0{\sim}x3$ represents the input data in different cycle. Finally, the equation of the process can be determined as t=3.

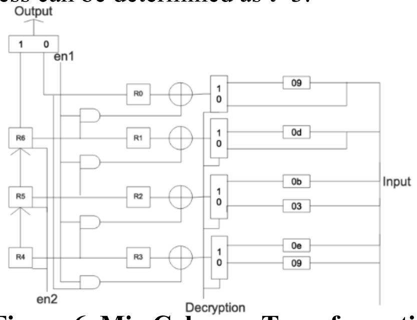


Figure 6. Mix Columns Transformation

3.3.1 Discussion

During the design stage, the first proposed architecture was not compatible to the AES system, as it took 5 clock cycle to generate each column and 20 clock cycle for a full Mixcolumn transformation (**Figure 7**), because it consists in two components. It results in synchronization issue, while key schedule needs 16 clock cycle and shift row requires 12 clock cycle. Therefore, the architecture needs to be reduced to 4 clock cycle for each column. The way to achieve this was by making the parallel to serial stage as a component, and utilized “for loop” to complete the cyclically shifted stage (**Figure 8**). Finally, the clock cycle not only can be reduced to 4 for each column, but also shirked the occupied slices, achieved higher throughput.

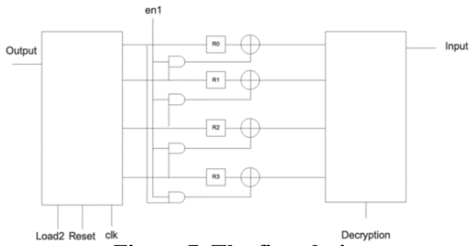


Figure 7. The first design

The design shows a good performance, the occupied slices is 24, which is slightly better than the standard number.

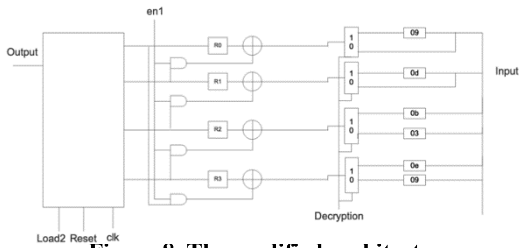


Figure 8. The modified architecture

Key schedule

This Keyschedule architecture is based on the ASIC design from [3] and it is shown in **Figure 9.** In this Keyschedule architecture, there are 17 shift registers which compose 5 register units, and 4 multiplexers. 16 of shift registers are a SRL16 and the other register is to change the order of some data. It need 16 clock cycles to preload the input key and store them in SRL16 before the output key is ready.

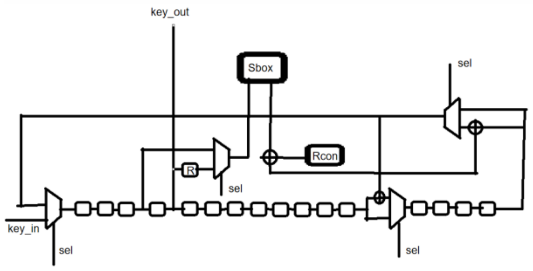


Figure 9. The structure of forward Keyschedule process.

The inverse Keyschedule architecture is based on the Keyschedule architecture, and it is shown in **Figure 10**. The new structure need an inverse register to reverse the order of input key. Because the inverse operation is started with the last byte of the key, the inverse register can reverse the input key to satisfy the structure requirement and reverse the generated key to satisfy the reading requirement. The total preload time becomes 48 clock cycles, where 16 for reverse input key, 16 cycles for preload key and 16 cycles for reverse output key.

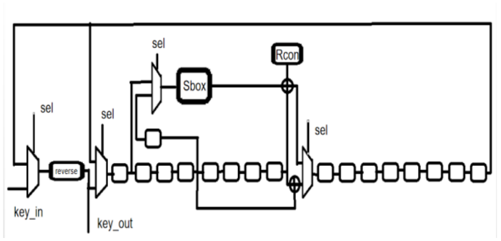


Figure 10. The structure of inverse Keyschedule process

This design of whole KeySchedule process occupies 201 slices on Artix7 XC7A100T device and the highest possible operating frequency is 175.386MHz. The respect area is 81 slices on Spartan III XC3S50 device and the highest possible operating frequency is 46.035MHz, but it only includes the forward Keyschedule part. So the whole structure might have a good performance.

There are still some part that can improve the performance, such as the Rcon generation part can save more area if we use an advance algorithm.

II. Combination

This combined architecture is based on the fast AES architecture and it is shown in **Figure 11**. In order to save area, we come up with the new structure to combine the forward and reverse process, so that two multiplexers are added around the MixColumn operation to decide whether to execute AddRoundKey operation first or later. As mentioned before, the forward KeySchedule process needs 16 clock cycles to preload the key and the reverse KeySchedule process needs 48 clock cycles. So there are two multiplexers to match the key and data with 33 clock cycles delay, which means that the encoding process can operate at start time and the decoding process need the 33 clock cycles delay before operating. And the role of other multiplexers is to differentiate the first round from other rounds. Finally, we control the output data path to generate the right data only by adding a multiplexer.

This whole AES design occupies 572 slices on Artix7 XC7A100T device and its highest possible operating frequency is 69.925MHz. Because the throughput is 55.59Kbps, the performance which defined as throughput/slices is 97.18 kbps per slice. Compared with other groups, this performance might not be very well, because we did not make every part optimized. Another reason is that to simplify the circuit and simulation, we did not combine all SubByte part together. That may waste some area.

	This design	Junfeng Chu [1]	Yong Sung Jeon et al [2]	Picoblaze Based [3]	T.Good & M.Benaissa [3]
FPGA	**Artix7 XC7A100T**	Spartan III XC3S50-5	Spartan II XC2S30-6	Spartan II XC2S15-6	Spartan II XC2S15-6
Clock Frequency (MHz)	**69.925**	45.64	66	90	67
Data path	**8**	8	8	8	8
No. of Clock cycles for Encryption	**161**	160	352	13546	3691
No. of Clock cycles for Decryption	**207**	-	-	-	-
Total Equivalent slices	**572**	184	258	452	264
Throughput (Mbps)	**55.59**	36.5	24	0.71	2.2
Throughput/slices (kbps/slice)	**97.18**	198	93	1.9	8.3
Summary	-	Smallest	-	Fastest	-

Table 2. Comparison of different designs

III. Conclusion

This paper has presented the AES studying with FPGA implementation. **Table 2** shows the comparison with different design. Our designed architecture shows an optimistic result. Although, the occupied slices are not comparable, while we are operating in different FPGA board, we still can see the Throughput performance, whereas the design of Chu **[1]** achieved 36.5 Mbps speed, and our design shows 55.59 Mbps. Also, out clock frequency of our design is higher than other 4 designs. However, the decryption part is not comparable, as the decryption was not included in their design. For future improvement, due to the clock cycles are mainly taken by Keyschedule part and Shiftrow, it will be more efficient to optimize these two parts.

IV. Reference

[1] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization," pp. 239–254, 2007.

[2] F. Field, "The Design of A Fast Inverse Module," pp. 298–303.

[3] S.Sheffield,"LOWAREA MEMORY-FREE FPGA IMPLEMENTATION OF THE AES ALGORITHM Junfeng Chu Department of Electronic and Electrical Engineering University of Sheffield Mohammed Benaissa Department of Electronic and Electrical Engineering University of Sheffield Sheffield S," pp. 623–626, 2012.

[4] E. N. C. Mui, "Practical Implementation of Rijndael S-Box Using Combinational Logic."

[5] Alex Biryukov and Dmitry Khovratovich, *Related-key Cryptanalysis of the Full AES-192 and AES-256*, "Archived copy". Archived *from the original on 2009-09-28*. Retrieved 2010-02-16.

[6] Daemen, Joan; Rijmen, Vincent (March 9, 2003). "AES Proposal: Rijndael". *National Institute of Standards and Technology. p. 1.* Archived *from the original on 5 March 2013*. Retrieved 21 February 2013.

[7] Westlund, Harold B. (2002). "NIST reports measurable success of Advanced Encryption Standard". *Journal of Research of the National Institute of Standards and Technology*. Archived from the original on 2007-11-03.

[8] *Bruce Schneier; John Kelsey; Doug Whiting; David Wagner; Chris Hall; Niels Ferguson; Tadayoshi Kohno; et al. (May 2000).* "The Twofish Team's Final Comments on AES Selection". Archived *from the original on 2010-01-02.*

[9] *Wisniewski, Remigiusz (2009).* *Synthesis of compositional microprogram control units for programmable devices*. *Zielona Góra: University of Zielona Góra. p. 153.* ISBN 978-83-7481-293-1.

[10] T. Good and M. Benaissa, "AES on FPGA from the Fastest to the Smallest,"LectureNotesinComputerScience,vol.3659,pp.427 - 440, Sep. 2005.

[11] X. Zhang, K. K. Parhi, High-speed VLSI architectures for the AES algorithm, IEEE Trans. VLSI Systems, Vol. 12, Iss. 9, pp. 957 - 967, Sept. 2004.