[[1]](#footnote-1)

A Low-Area Memoryless Implementation of the AES Algorithm on FPGA Boards

Robert W. Carpenter, Aeman S. K. Faruqui, Rolando A. May Alvarez, Mohammad R. Y. Almshafi, and Weiwei Zhang, Department of Electronic and Electrical Engineering, University of Sheffield

*Abstract*—In this paper an 8-bit memoryless FPGA implementation of the AES cipher is presented. The design requires 349 slices and can achieve throughputs of 31.4 Mbps on a Spartan 3 (XC3S50) FPGA, 210 slices with a throughput of 47.46 Mbps on a Spartan 6 (XC6SLX4), and 244 slices with a throughput of 91.5 Mbps on an Artix7.

*Index Terms*—Advanced Encryption Standard (AES), Field Programmable Gate Array (FPGA), low area, finite field

# INTRODUCTION

T

HE title of Advanced Encryption Standard (AES) was given to the winning algorithm of the 2000 US government competition to select a new standard symmetric-key cryptography algorithm. The winner in question is known as the Rijndael algorithm [1-3], of which there are three different versions, based on different key lengths. Herein, we will focus on the most basic version, AES-128, with a key and plaintext length of 128 bits (organised into 16 bytes) per block, and which is based on finite field arithmetic [4].

The algorithm is a recursive one, based on a cyclical structure with four basic operations: ShiftRows, SubBytes, MixColumns, and AddRoundKey. These are cycled through a number of times depending on the key length; in the AES-128 case, there are 10 rounds, of which the last does not include MixColumns, and also an extra “zeroth” round consisting only of an extra AddRoundKey. For the AddRoundKey operations, a set of round keys must be generated from the original key in another operation called KeyGeneration or KeySchedule: the add round key is a simple bitwise xor operation between the current state and the round key. ShiftRows involves reordering the bytes of the current state, SubBytes is a per-byte substitution stage, and MixColumns involves manipulating sets of 4 bytes at a time to produce another set of 4 bytes. Decryption involves the inverses of each of these modules, which typically can be equivalently realised using the same sets of Galois field operations but with the numbers modified.

Applications of this algorithm in hardware are generally divided into high-throughput applications, in which the main design consideration is the speed at which plaintext can be encrypted, and low-area applications, in which limiting the hardware and power consumption is the main constraint. FPGAs are a common hardware implementation method for both. One design aspect is the size of the data path: in our design this is 8 bits, but 128 or 32 are also common. High-throughput designs generally consist of loop-unrolled architectures relying on pipelining to increase their speed [5-9]. The focus of this report is low-area, where the main metric of quality is the number of FPGA slices used. An early influential paper was by Good & Benaissa [5], which detailed 8-bit data path designs for FPGAs based on ASIP and Picoblaze implementations, using 124 and 119 slices respectively, plus two BRAMS, on a Spartan II FPGA. Many incremental improvements have since been made to individual components and exploitation of aspects of FPGAs [10-15].

In this paper an 8-bit memoryless FPGA implementation of the AES cipher is presented. The design requires 349 slices and can achieve throughputs of 31.4 Mbps on a Spartan 3 (XC3S50) FPGA, 210 slices with a throughput of 47.46 Mbps on a Spartan 6 (XC6SLX4), and 244 slices with a throughput of 91.5 Mbps on an Artix7.

# Design

## **Top-level Architecture**

The overall architecture map is based on the previous work by Chu & Benaissa [12] and Hämäläinen et al. [16], with a data path of 8 bits. A simple diagram is presented in Fig. 1. The design relies on a cyclical process in which each component feeds its result immediately into the next, with the output of the last module feeding back in to the beginning of the loop. KeySchedule has two output lines, one of which is delayed and one of which is not, the latter being used for the final round. This is because the final round does not include the MixColumns step and so avoids a four-cycle delay, so the final round key must be effectively four cycles early, as compared to the other rounds. The delayer is included to account for this 4-period delay during the initial round; afterwards this delay is no longer an issue.



Fig. 1. A simplified view of the overall architecture. [12]

There are two AddRoundKey operations, one adding the delayed keys to the initial data or to the output of MixColumns, the other again only used in the final round, adding the final key to the output of the S-box for that round. Two S-box components were also included: one as part of the KeyExpansion, the other as its own independent module.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **t** | **R15** | **R14** | **R13** | **R12** | **R11 …. R4** | **R3** | **R2** | **R1** | **R0** | **Operations** |
| 0 | a15 | a14 | **a13** | a12 | a11 …. a4 | a3 | a2 | a1 | a0 | b0= a0 ⊕ **S(a13)** ⊕ Rcon, b4 = a4 ⊕ b0 |
| 1 | **b0** | a15 | **a14** | a13 | a12 … a5 | **b4** | a3 | a2 | a1 | b1= a1 ⊕ **S(a14)** , b5 = a5 ⊕ b1 |
| 2 | **b1** | b0 | **a15** | a14 | a13 … a6 | **b5** | b4 | a3 | a2 | b2= a2 ⊕ **S(a15)** , b6 = a5 ⊕ b2 |
| 3 | **b2** | b1 | b0 | a15 | a14 … a7 | **b6** | b5 | b4 | a3 | b3= a3 ⊕ **S(a12)** , b7 = a7 ⊕ b3 |
| 4 | **b3** | b2 | b1 | b0 | a15 … a8 | **b7** | b6 | b5 | b4 | b8= a8 ⊕ b4 |
| 5 | b4 | b3 | b2 | b1 | b0 … a9 | **b8** | b7 | b6 | b5 | b9= a9 ⊕ b5 |
| 6 | b5 | b4 | b3 | b2 | b1 … a10 | **b9** | b8 | b7 | b6 | b10= a10 ⊕ b6 |
| 7 | b6 | b5 | b4 | b3 | b2 … a11 | **b10** | b9 | b8 | b7 | b11= a11 ⊕ b7 |
| 8 | b7 | b6 | b5 | b4 | b3 … a12 | **b11** | b10 | b9 | b8 | b12= a12 ⊕ b8 |
| 9 | b8 | b7 | b6 | b5 | b4 … a13 | **b12** | b11 | b10 | b9 | b13= a13 ⊕ b9 |
| 10 | b9 | b8 | b7 | b6 | b5 … a14 | **b13** | b12 | b11 | b10 | b14= a14 ⊕ b10 |
| 11 | b10 | b9 | b8 | b7 | b6 … a15 | **b14** | b13 | b12 | b11 | b15= a15 ⊕ b11 |
| 12 | b11 | b10 | b9 | b8 | b7 … b6 | b15 | b14 | b13 | b12 | - |
| 13 | b12 | b11 | b10 | b9 | b8 … b1 | b0 | b15 | b14 | b13 | - |
| 14 | b13 | b12 | b11 | b10 | b9… b2 | b1 | b0 | b15 | b14 | - |
| 15 | b14 | b13 | b12 | b11 | b10 …b3 | b2 | b1 | b0 | b15 | - |

Table 1. The order of operations for the KeyExpansion [16]

The control signals of the architecture are generated by a simple 160-cycle counter process. The individual requirements of the components were the main factor behind its exact structure: MixColumns required its ce signal one clock period before input, as did the delay, ShiftRows and KeySchedule modules. These thus occurred 15 periods after the initial input for MixColumns, one period before for KeySchedule and the delayer, and two periods after for ShiftRows. The multiplexer also needed to switch its select line after the final delayed input: 20 periods after the initial input. The delayer is included to account for the 4-period delay from the initial round of the KeySchedule, which has two output lines, one of which is delayed and one of which is not, the latter being used for the final round. There are two AddRoundKey operations, one adding the delayed keys to the initial data or to the output of MixColumns, the other again only used in the final round, adding the final key to the output of the SubBytes for that round. Each signal was given a “reset” value in the 0 period.

## **KeyExpansion**

The KeyExpansion unit used is based on the architecture described in [16] [12]. It makes use of an 8-bit data path and a dedicated compact SubBytes component. This helps to generate round keys on-the-fly without the need of storing the keys which helps in saving space. Our architecture occupies 91 slices on a Spartan 3 with a highest possible clock frequency of 39.342 MHz which is very close to the 81 slices used by KeySchedule module in [12] having a clock frequency of 46.035 MHz Fig. 2 shows the architecture for KeyExpansion module and table 1 describes step by step operations and data flow to generate a new key. A new round key is generated from the previous one in 16 clock cycles. There are two outputs taken out from this module. The key\_out is fed into the AddRoundKey module and the Key\_delay\_out is used in the last round.



Fig. 2. A block diagram of the KeyExpansion [12].

## **ShiftRows**

ShiftRows process is relocating each byte, in the rows of the block text, cylindrically in the left direction as shown in Fig. 3a. The first row of the block will stay as it is, the second row of the block will be shifted by one cyclical shift to the left, the third row will be shifted by two cyclical shifts to the left, the final row will be shifted by three cyclical shifts to the left [17].

The architecture of ShiftRows consists of two sets of SRL 16 and two multiplexers, shown in Fig. 3b, eight of this architecture are constructed to work in parallel and share the address state machine taps to perform the shifting operation. The data will go through the shift registers and while is being shifted the address taps will select the data required to the output. The ShiftRows process in this architecture will start to produce the first output after 12 clock cycles. This ShiftRows construction occupies 43 slices in Spartan 6 FPGA, and 44 slices in Artix 7 FPGA, and 59 slices in Spartan 3 FPGA.

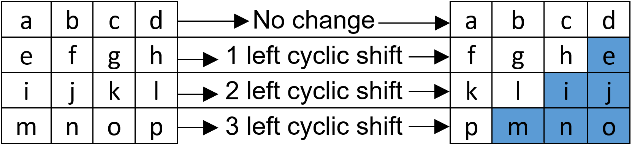


Fig. 3a. A visualization of the ShiftRows process [17].



Fig. 3b. A block diagram of the ShiftRows module [18].

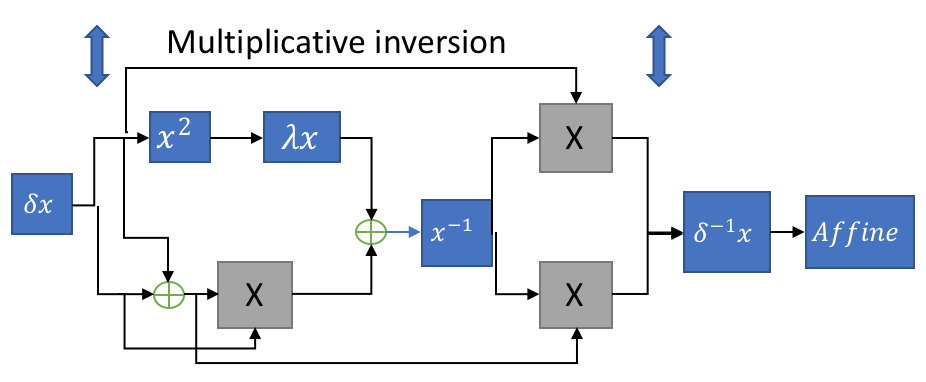


Fig. 4a: Sub Bytes implementation. [6]

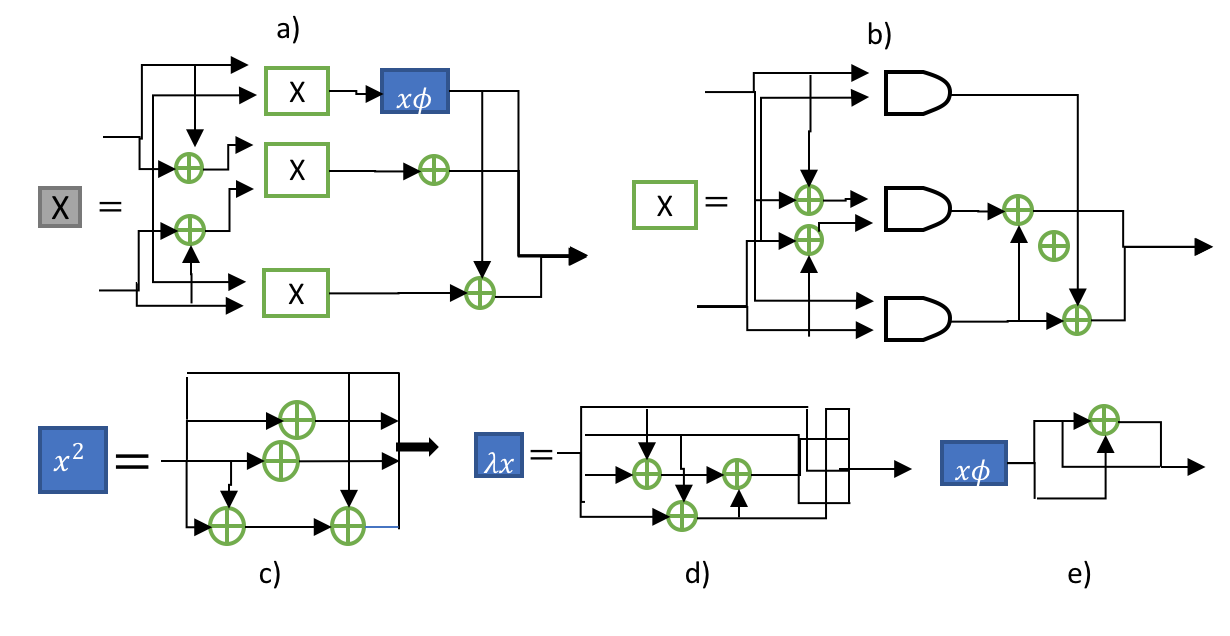


Fig. 4b: Implementation of each block: a) Multiplication in b) Multiplication in c) square in d) Multiplication with constant e) Multiplication with constant [6]

## **SubBytes**

The SubBytes process consists of updating each of the input bytes using the Rijndael S-box: this will allow the system to include non-linearity and thus generates a more secure cypher. Commonly this procedure is achieved using look up table that is stored in the devices memory, but the project design was built under low area criteria (using as little hardware as possible) this means that the box must be generated using the logical gates available on the FPGA. To achieve this purpose a multiplicative inverse in GF () must be applied followed by the affine transformation. For the Inverse SubBytes, the same procedure can be implemented only with different order: first the Inverse Affine Transformation and then the Multiplicative Inverse.

Since the data can’t be processed in the composite field GF() an isomorphic mapping function must be applied (), once the multiplicative inverse has finished the inverse mapping function must be implemented to map it back from its composite field representation using **.**  It must kept in mind that the isomorphic and its inverse transformation are built under the assumption that p(x) is defined as [19]|,[6].

The multiplicative inverse in the was built iteratively form lower order fields of , and to achieve this, the following irreducible polynomials [6][14] are used:

Where:

In the Fig. 4a is show the general implementation of the whole design for the SubBytes, from left to right it can be seen the isomorphic function then the square in the block label in the picture is the multiplication with the constant after the addition in with the result of the multiplication in label as the grey block with an X, followed with the multiplicative inverse in with two multiplication finally the inverse isomorphic transformation followed by the inverse Affine transformation. The same procedure is implemented for the Inverse SubBytes but using the inverse affine transformation, since is the same structure a chip select can be implemented to choose which process is needed without significantly increasing the area. In Fig. 4b the implementation of each of the blocks is shown.

## **MixColumns**

The theory of MixColumns is as shown below:

(1)

In this part, four registers () are used to store the numbers respectively, where a new byte will be fed in after every clock cycle. Therefore, every byte in the registers can be regarded as a unit of a continuous data flow, and indexes are introduced to obtain the byte needed. The storage status in four registers can be sketched out in chart 1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| clock | 0 | 1 | 2 | **3** | 4 | 5 | 6 | index |
| t | 2d | 2c | 2b | 2a |  |  |  | 3 |
| t+1 | 2e | 2d | 2c | 2b | 2a |  |  | 3 |
| t+2 | 2f | 2c | 2d | 2c | 2b | 2a |  | 3 |
| t+3 | 2g | 2f | 2e | 2d | 2c | 2b | 2a | 3 |

Table 2(a). Storage status in

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| clock | 0 | 1 | **2** | 3 | 4 | 5 | **6** | index |
| t | 3d | 3c | 3b | 3a |  |  |  | 2 |
| t+1 | 3e | 3d | 3c | 3b | 3a |  |  | 2 |
| t+2 | 3f | 3c | 3d | 3c | 3b | 3a |  | 2 |
| t+3 | 3g | 3f | 3e | 3d | 3c | 3b | 3a | 6 |

Table 2(b). Storage status in

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| clock | 0 | **1** | 2 | 3 | 4 | **5** | 6 | index |
| t | d | c | B | a |  |  |  | 1 |
| t+1 | e | d | C | b | a |  |  | 1 |
| t+2 | f | c | D | c | b | a |  | 5 |
| t+3 | g | f | E | d | c | b | a | 5 |

Table 2(c). Storage status in

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| clock | **0** | 1 | 2 | 3 | **4** | 5 | 6 | index |
| t | d | c | B | a |  |  |  | 0 |
| t+1 | e | d | C | b | a |  |  | 4 |
| t+2 | f | c | D | c | b | a |  | 4 |
| t+3 | g | f | E | d | c | b | a | 4 |

Table 2(d). Storage status in

Noting that the input bytes were dealt by the function “”, “”, “”, “” over respectively before entering the registers. The indexes in four registers are defined as:

(2a)

(2b)

(2c)

(2d)

The value of is changing over time

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Authors***  ***Parameters*** | **Chodowiec & Gaj [11]** | **Rouvroy et al [13]** | **Pramstaller et al [10]** | **T.Good & M.Benaissa [5]** | **Picoblaze based [5]** | **Yong Sung Jeon et al [20]** | **J. Chu & M. Benaissa [12]** | **Ours** | **Sasdrich &**  **Güneysu [15]** |
| **FPGA** | Spartan II XC2S30-6 | Spartan III XC3S50-4 | Virtex-E XCV1000E | Spartan II XC2S15-6 | Spartan II XC2S15-6 | Spartan II XC2S30-6 | Spartan III XC2S30-5 | Spartan VI  XC6SLX4 | Spartan VI XC6SLX4 |
| **Clk Freq (MHz)** | 60 | 71 | 161 | 67 | 90 | 66 | 45.642 | 59.328 | 105 |
| **Data Path** | 32 | 32 | 32 | 8 | 8 | 8 | 8 | 8 | 8 |
| **No of clk cycles** | 44 | 46 | 92 | 3691 | 13546 | 352 | 160 | 160 | 1471 |
| **Slices** | 222 | 163 | 1125 | 124 | 119 | 258 | 184 | 210 | 21 |
| **No of Block RAM** | 4 | 18 | 0 | 4 | 4 | 0 | 0 | 0 | 0 |
| **Total Eq. slices** | 522 | 1231 | 1125 | 264 | 452 | 258 | 184 | 210 | 21 |
| **Throughput (Mbps)** | 318 | 208 | 215 | 2.2 | 0.71 | 24 | 36.5 | 47.5 | 9.12 |
| **Thro/slice (kbps/slice)** | 318 | 169 | 191 | 8.3 | 1.9 | 93 | 198 | 226 | 0.43 |
| **Summary** | Excellent speed/area |  | Highest Speed |  |  |  | Compact with good throughput |  | Most compact |

Table 3. A comparison of previous work with ours. [12]

when clock=t, ;

when t+1, ;

when t+2, ;

when t+3, ;

thus, every MixColumns can be finished after four clocks cycle.

# Results

Synthesis reports were generated for the design on each of three FPGA boards: Spartan 3, Spartan 6, and Artix 7. The sizes of the individual components and of the total design are summarised in Table 3. Table 4 contains a comparison of several designs, including some with 32-bit and some with 8-bit data paths. We have achieved a thoughput of 226 kbps/slice using Spartan 6 with a total of 210 of slices occupied based on the architecture used by Chu & Benaissa [12] which gives a throughput of 198 kbps/slice using only 184 slices on Spartan 3. Our control overhead is a possible reason for the difference in the slice count affecting the total performance.

|  |  |  |  |
| --- | --- | --- | --- |
| Components | Spartan6 | Spartan3 | Artix7 |
| Sbox | 26 | 42 | 35 |
| ShiftRows | 43 | 59 | 44 |
| MixColumns | 36 | 77 | 42 |
| KeyExpansion (incl. Sbox) | 33 | 91 | 53 |
| Controller | 72 | 80 | 70 |
| Sum of components | 138 | 269 | 174 |
| No of occupied slices | 210 | 349 | 244 |
| Clk Freq (MHz) | 59.3 | 39.3 | 114.4 |
| Data Rate (kbps) | 226 | 90 | 375 |

Table 4. A comparison of the component and total slice sizes and other metrics on different FPGA devices

References

1. National Institute of Standards and Technology (NIST), *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197 (FIPS PUB 197), Nov. 2001.
2. B. Gladman, *A Specification for Rijndael, the AES Algorithm*, May 2002.
3. J. Daemen, V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, Springer, 2002
4. R.W. Ward, T.C.A. Molteno,*Efficient Hardware Calculation of Inverses in GF(28)*, Proceedings of ENZCon'03, University of Waikato, NZ, September 2003.
5. T. Good, M. Benaissa,*AES on FPGA from the fastest to smallest*,   
   CHES 2005, LNCS 3659, pp. 427–440, Springer-Verlag, 2005.
6. 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.
7. K.U. Järvinen, M.T. Tommiska, J.O. Skytta, *A fully pipelined memoryless 17.8 Gbps AES-128 encryptor*, Proc. Int. Symp. Field-Programmable Gate Arrays (FPGA 2003), Monterey, CA, pp. 207–215, Feb. 2003.
8. R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa. Reconfigurable memory based AES co-processor. In 20th International Parallel and Distributed Processing *Symposium (IPDPS 2006), Proceedings,* 2006.
9. S. Drimer, T. Giineysu, and C. Paar. DSPs, BRAMs and a Pinch of Logic: New Recipes for AES on FPGAs. In *16th IEEE International Symposium on* Field-Programmable Custom Computing Machines, FCCM 2008, 2008.
10. N. Pramstaller, J. Wolkerstorfer, *A Universal and efficient AES co-processor for Field Programmable Logic Arrays*,   
    FPL 2004, LNCS Vol. 3203, pp. 565-574, Springer-Verlag, 2004.
11. P. Chodowiec, K. Gaj,*Very Compact FPGA Implementation of the AES Algorithm*,   
    Cryptographic Hardware and Embedded Systems (CHES 2003), LNCS Vol. 2779, pp. 319 – 333, Springer-Verlag, Oct. 2003.
12. J. Chu and M. Benaissa, *Low area memory-free FPGA implementation of the AES algorithm,*Conference on Field Programmable Logic and Applications (FPL), Oslo, 2012, pp. 623-626.
13. G. Rouvroy, F.X. Standaert, J.J. Quisquater, J.D. Legat, *Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications*,   
    Procedings of the international conference on Information Technology: Coding and Computing 2004 (ITCC 2004), pp. 583 – 587, Vol.2, April 2004.
14. A. Satoh, S. Morioka, K. Takano, S.Munetoh, A Compact Rijndael Hardware Architecture with S-Box Optimization,   
    Proceedings of ASIACRYPT 2001, LNCS Vol. 2248, pp. 239 - 254, Springer-Verlag, Dec. 2001.
15. P. Sasdrich, T. Güneysu, A Grain in the Silicon: SeA-Protected AES in Less than 30 Slices, 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 25-32.
16. [P. Hamalainen](https://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=%22Authors%22:.QT.P.%20Hamalainen.QT.&newsearch=true); [T. Alho](https://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=%22Authors%22:.QT.T.%20Alho.QT.&newsearch=true); [M. Hannikainen](https://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=%22Authors%22:.QT.M.%20Hannikainen.QT.&newsearch=true); [T. D. Hamalainen](https://ieeexplore.ieee.org/search/searchresult.jsp?searchWithin=%22Authors%22:.QT.T.%20D.%20Hamalainen.QT.&newsearch=true), Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core, [9th EUROMICRO Conference on Digital System Design (DSD'06)](https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=11117), 2006, pp. 577-583.
17. Al-Wattar, A.H. et al., 2015. A New DNA-Based Approach of Generating Key-dependent ShiftRows Transformation.
18. Soltani & Sharifian, 2015. An ultra-high throughput and fully pipelined implementation of AES algorithm on FPGA. Microprocessors and Microsystems, 39(7), pp.480–493.
19. Edwin NC Mui, Xess.com, 2018. [Online]. Available: http://www.xess.com/static/media/projects/Rijndael\_SBox.pdf. [Accessed: 26- Apr- 2018].
20. Y. S. Jeon, Y. J. Kim, and D. H. Lee, “A Compact MemoryFree Architecture for the Aes Algorithm Using Resource Sharing Methods,” Journal of Circuits, Systems, and Computers, vol. 19, no. 5, p. 1109, 2010.

1. [↑](#footnote-ref-1)