International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue-01
January 2018

VLSI Implementation of AES Algorithm using Rijndael algorithm

M.Srujana & B.Parameshwari
Assistant Professor Department of Electronics and Communication Engineering, CJITS, Yeshwanthapur,T.S
E-mail:mittapallysrujana814@gmail.com & E-mail:parameshwari.bhoomigari@gmail.com

Abstract— *In this paper we present a high-performance, high throughput, and area efficient architecture for the VLSI implementation of the AES algorithm. The subkeys, required for each round of the Rijndael algorithm, are generated in real-time by the key-scheduler module by expanding the initial secret key, thus reducing the amount of storage for buffering. Moreover, pipelining is used after each standard round to enhance the throughput. A prototype chip implemented using 0.35µ CMOS technology resulted in a throughput of 232M bps for iterative architecture and 1.83ttbps for pipelining architecture.*

I. Introduction

Several techniques, such as cryptography, steganography, watermarking, and scrambling, have been developed to keep data secure, private, and copyright protected [1], [2]. Cryptography is an essential tool underlying virtually all networking and computer protection, traditionally used for military and espionage. However, the need for secure transactions in e-commerce, private networks, and secure messaging has moved encryption into the commercial realm [3].

Advanced encryption standard (AES) was issued as Federal Information Processing Standards (FIPS) by National Institute of Standards and Technology (NIST) as a successor to data encryption standard (DES) algorithms. In recent literature, a number of architectures for the VLSI implementation of AES Rijndael algorithm are reported [4], [5], [6], [7], [8]. It can be observed that some of these architectures are of low performance and some provide low throughput. Further, many of the architectures are not area efficient and can result in higher cost when implemented in silicon.

In this paper, we propose a high performance, high throughput and area efficient VLSI architecture for Rijndeal algorithm that is suitable for low cost silicon implementation. The proposed architecture is optimized for high throughput in terms of the encryption and decryption data rates using pipelining. Polynomial multiplication is implemented using XOR operation instead of using multipliers to decrease the hardware complexity. In the proposed architecture both the encryption and decryption modes use common hardware resources, thus making the design area efficient. Selective use of look-up tables and combinational logic further enhances the architecture's memory optimization, area, and performance. An important feature of our proposed architecture is an effective solution of online (real-time) round key generation needing significantly less storage for buffering.

II. Rijndael Algorithm

Rijndael algorithm is an iterated block cipher [9] supporting a variable data block and a variable key length of 128, 192 or 256 bits. The algorithm consists of three distinct phases: (i) an initial data/key addition, (ii) nine (128-bits), eleven (192-bits) or thirteen (256-bits) standard rounds, (iii) a final round which is a variation of a standard round. The number of standard rounds depends on the data block and key length. If the maximum length of the datablock or key is 128, 192 or 256, then the number of rounds is 10, 12 or 14, respectively. The initial key is expanded to generate the round keys, each of size equal to block length. Each round of the algorithm receives a new round key from the key schedule module.

Each standard round includes four fundamental algebraic function transformations on arrays of bytes. These transformations are: byte substitution, shift row, mix column, and round key addition. The final round of the algorithm is similar to the standard round, except that it does not have *MixColumn* operation. Decryption is performed by the application of the inverse transformations of the round functions. The sequence of operations for the standard round function differs from encryption. The computational performance differs between encryption and decryption because the inverse transformations in the round function is more complex than the corresponding transformation for encryption.

III. The Proposed VLSI Architecture for Rijndael

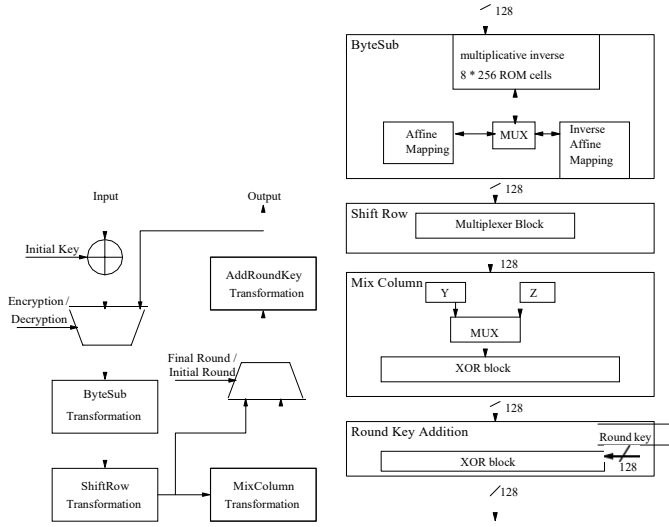
The proposed architecture showing the order of operation and control between the transformations is shown in Fig. 1(a).

A. Architecture of the Data Unit

The data unit consists of: the initial round of key addition, $N_r - 1$ standard rounds, and a final round. The architecture for a standard round composed of four basic blocks is shown in Fig. 1(b). For each block, both the transformation and the inverse transformation needed for encryption and decryption, respectively are performed using the same hardware resources. This implementation generates one set of subkey and reuses it for calculating all other subkeys in real-time.

1) ByteSub: In this architecture each block is replaced by its substitution in an S-Box table consisting of the multiplicative inverse of each byte of the block state in the finite field *ttF* (2^8). In order to overcome the performance bottleneck,

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue-01
January 2018



(a) Rijndael Algorithm Data and Control Flow (b) Architecture for the Standard Round in the Data Unit

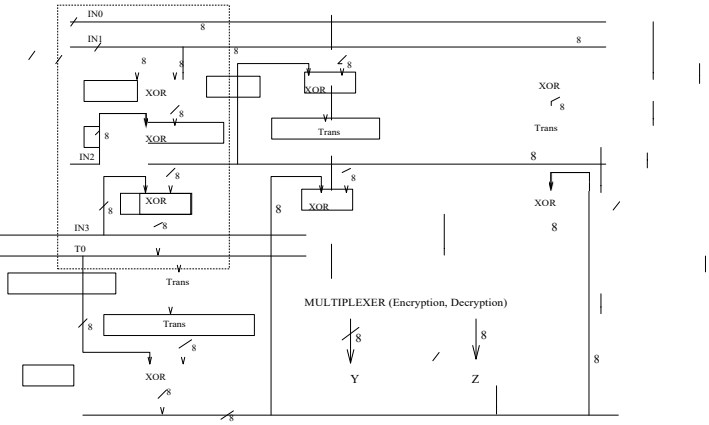
Fig. 1. Top Level View of the Rijndael

the implementation of multiplicative inverses is carried out using look-up tables (stored in a table of 8×256). The implementation includes the affine mapping of the input in both encryption and decryption processes as follows:
Affine Mapping: $Out[i] = In[i] \oplus In[(i+4) mod 8] \oplus In[(i+5) mod 8] \oplus In[(i+6) mod 8] \oplus In[(i+7) mod 8] \oplus CE[i]$, where $CE = 01100011$ is a constant, leftmost bit the being MSB.
Inverse Affine Mapping: $Out[i] = In[(i+2) mod 8] \oplus In[(i+5) mod 8] \oplus In[(i+7) mod 8] \oplus CD[i]$, where $CD = 00000101$ is a constant, leftmost bit being the MSB.

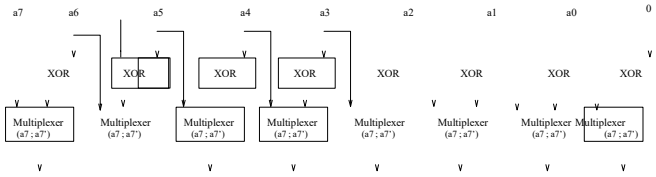
2) ShiftRow: In this transformation the rows of the block state are shifted over different offsets. The amount of shifts is determined by the block length. The proposed architecture implements the shift row operation using combinational logic considering the offset by which a row should be shifted.

3) MixColumn: In this transformation each column of the block state is considered as a polynomial over $ttF\ (2^8)$. It is multiplied with a constant polynomial $C(x)$ or $D(x)$ over a finite field in encryption or decryption, respectively. In hardware, the multiplication by the corresponding polynomial is done by XOR operations and multiplication of a block by X. This is implemented using a multiplexer, the control being the MSB is 1 or 0. The equations implemented in hardware for *MixColumn* in encryption and decryption are as follows.
In encryption process,
$Y = In0 \oplus In1 \oplus In2 \oplus In3$ and $Z = Y$.
In decryption process, $T0 = In0 \oplus In1 \oplus In2 \oplus In3$, $T1 = T0 \oplus [In2Trans(In2Trans(T0))]$,
$Y = T1 \oplus [In2Trans(In2Trans(In0 \oplus In2))]$, and $Z = T1 \oplus [In2Trans(In2Trans(In1 \oplus In3))]$.
$Out0 = In0 \oplus [Y \oplus In2Trans(In0 \oplus In1)]$, $Out1 = In1 \oplus [Z \oplus In2Trans(In1 \oplus In2)]$
$Out2 = In2 \oplus [Y \oplus In2Trans(In2 \oplus In3)]$, and $Out3 = In3 \oplus [Z \oplus In2Trans(In3 \oplus In0)]$.

Where, $In2Trans(K)$ is the multiplication of the byte by X (hexadecimal value 02) over $ttF\ (2^8)$. $In0$ is the least significant 8 bits of a column of a matrix. Architecture of different units are shown in Fig. 2 and the architecture of *MixColumn* transformation is shown in the Fig. 3.



(a) For Computation of Y, Z



(b) For Multiplication by X (hex "02")

Fig. 2. Architecture for Units used in Mix Column Transformation

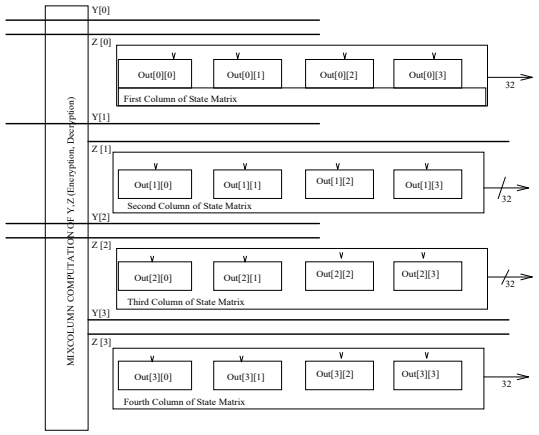


Fig. 3. Architecture for Mix Column Transformation for 128 bits

4) AddRoundKey: In this transformation (architecture represented in Fig. 4), the round key obtained from the key scheduler is XORed with the block state obtained from the *MixColumn* transformation or *ShiftRow* transformation based on the type of round being implemented. In the standard round, the round key is XORed with the output obtained from the *MixColumn* transformation. In the final round the round key is XORed with the output obtained from the *ShiftRow* transformation. In the initial round, bitwise XOR operation is

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue-01
January 2018

performed between the initial round key and the initial state block.

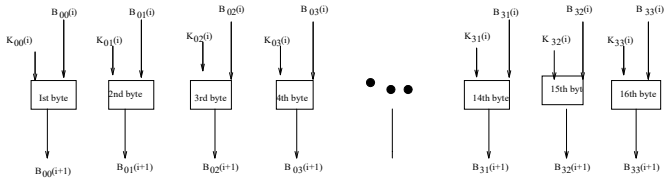


Fig. 4. Architecture for Round Key Addition Transformation

B. Architecture for Key Scheduling

In the key scheduling module (Fig. 5), the initial key is expanded and the generated round keys are stored in four 32-bit registers. Both the forward and reverse key scheduling are done in the same device. The *ByteSub* required in the key expansion unit is implemented using the S-Boxes. Four S-Boxes are needed for a 128-bit key and 128-bit data block implemented using 8×256 ROM cells. Multiplexers are used as a control signal to distinguish between the initial key and the round key (obtained from the initial key using a "key expansion unit"). The least significant 32 bits of the 128-bit key is cyclically shifted to the left by a byte, implemented using combinational logic. The resulting word after the left shift operation is sent through the S-boxes and the affine mapping operation, in order to perform *ByteSub*. The key resulting from the *ByteSub* is XORed with the Round Constant (RCON). In this architecture, the round constant is generated using the combinational logic. The round constant should be symmetric with the round key being generated.

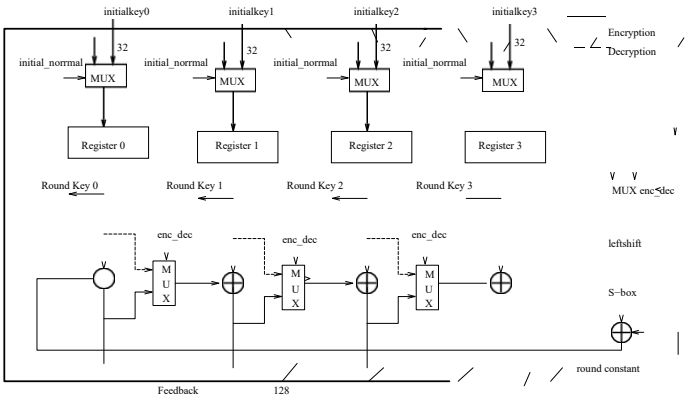


Fig. 5. Architecture for Key Scheduling Unit

The total number of round constants that need to be generated is equal to the number of rounds. The round constant is obtained in real-time by multiplying the previous round constant by X. This is amenable for implementation in the hardware using XOR operations. For the reverse key scheduling, the last round key should be generated with forward key scheduling for the first time. The last round key is expanded to generate the reverse round keys. Decryption requires more cycles than encryption because it needs pre-scheduling to generate the last key value. Since the Rijndael algorithm allows different key lengths and block lengths, each round key is carefully set to have the same length as the data block. In the case where key length and the block length are not equal, previous, current and also the next round keys are needed in order to generate the appropriate set of round keys that are fed into the encryption module, which is performed by a "key alignment unit".

IV. A Propotype Chip Implementation for Rijndael

We now present the methodology used to design, simulate, and verify the proposed architecture.

A. Implementation Analysis

It is evident that the Rijndael's S-Boxes are the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is an 8-bit$\times$8-bit look-up-table, requiring more hardware resources. However, the remaining components of the Rijndael round function – byte swapping, constant multiplication by an element of Galois Field , and key addition – were found to be simpler in structure, resulting in these elements of the round function requiring fewer hardware resources. Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA [10]. Partially pipelined implementation with one sub-pipeline stage provided one area-optimized solution. As compared to a one-stage implementation with no sub-pipelining, the addition of a sub-pipeline stage afforded the synthesis tool greater flexibility in its optimizations, resulting in a more area efficient implementation. The 2-stage loop unrolling was found to yield the highest throughput when operating in FeedBack (FB) mode.

B. Memory Optimization

Since the design is based on one clock cycle for each encryption round, the memory modules had to be duplicated. For example, in the *ByteSub*, the S-boxes need to be duplicated 16 times. Consequently, the choice of memory architecture is very critical. Since all the table entries are fixed and defined in the standard, the usage of ROM is preferred. Specifically, the architecture requires several small ROM modules instead of one large module, since each lookup will only be based on a maximum of 8-bit address, which translates to 256 entries. We implemented the multiplicative inverse function using the look-up table of size 8×256. We have a total of 20 copies of the S-boxes in our design; 16 of them in encryption module and 4 in the key scheduling module.

C. Design Flow

The proposed architecture was implemented using the CADENCE virtuoso layout design tool. The method adopted was a custom designed at the transistor level based on a custom cell library of 0.35μ CMOS primitive standard cells. A hierarchical

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 05 Issue-01
January 2018

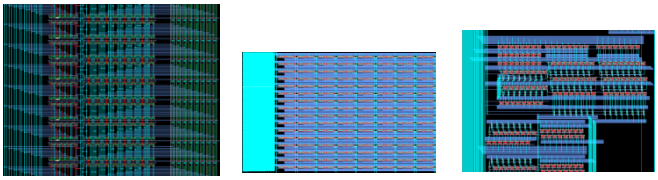
TABLE I
COMPONENTS OF THE AES-128 MODULE

Module/Component	Number of Components Proposed Architecture	Mangard et. al. Architectutre [7]
DATA UNIT		
S-Boxes	16	16
32-bit Registers using D-cells	8	16
Multiplexers	240	384
32-bit Multiplexers	180	NA
128-bit Multiplexers	60	NA
Multipliers	0	16
KEY UNIT		
S-Boxes	4	NA
32-bit Registers using D-cells	4	NA
32-bit Multiplexers	4	NA

TABLE II
SUMMARY OF THE PERFORMANCE OF THE AES-128 MODULE

Architecture	Clock cycles	Throughput [*Mbps*]
Proposed Architecture	11	232
Mangard et. al. [7]-Standard	64	128
Mangard et. al. [7]-High Performance	34	241

approach was followed in the implementation. The layout for each module was generated and later integrated to obtain the final chip. The generated netlist was then simulated with HSPICE using the MOSIS CMOS model. Once the creation of layout design is finished, the I/O pins have to be added to the circuit. The design layouts of different architectural units are shown in the Fig. 6.



(a) Multiplicative Inverse (b) Affine Mapping (c) Round Constants

Fig. 6. Layout of Different Architectural Units

D. Performance Evaluation

An AES-128 encryption / decryption of a 128-bit block was done in 11 clock cycles using the feedback logic. In each clock cycle, one transformation is executed and, at the same time, the appropriate key for the next round is calculated. The whole process concludes after 10 rounds of transformations. The analysis of the components used for the proposed architecture is shown in the Table I. The architecture proposed by Mangard et. al. [7] uses multipliers for the implementation of the *MixColumn*, while ours uses XOR, multiplexors, inverters etc. to reduce the complexity. Kuo et. al. [4] uses lookup table for the implementation of the shift row module and for the generation of the round constants in the key scheduling module, but we used the combinational logic instead of the look-up tables, thus reducing the area.

The frequency of the external clock with which the architecture operates was $20MHz$; the critical delay being $50ns$. The throughput is calculated as: Throughput $= \frac{\text{block size} * \text{clock frequency}}{\text{total clock cycles}} = \frac{128 * 20MHz}{11} = 232.7Mbps$, [where, clock frequency = 1 / clock period for the critical path]. When the pipelining technique is used instead of the iterative feedback logic, the standard rounds are duplicated for N_r times cascaded by the pipelining registers. This increases the effective area. At a particular clock cycle, N_r blocks of data can be encrypted or decrypted using the pipelining technique. Based on the critical path obtained using our implementation, the throughput achieved with pipelining can be calculated as: Throughput $= \frac{\text{block size}}{\text{total delay}} = \frac{128}{70ns} = 1.83Tbps$, [where, total delay is the delay of the single round including the delays caused by the pipelined registers]. The summary of the performance in the Table II shows that our proposed architecture minimizes the needed number of clock cycles and achieves high throughput.

V. DISCUSSIONS AND CONCLUSIONS

We have presented a VLSI architecture for the Rijndael AES algorithm that performs both the encryption and decryption. S-boxes are used for the implementation of the multiplicative inverses and shared between encryption and decryption. The round keys needed for each round of the implementation are generated in real-time. The forward and reverse key scheduling is implemented on the same device, thus allowing efficient area minimization. Although the algorithm is symmetrical, the hardware required is not, with the encryption algorithm being less complex than the decryption algorithm. The implementation of the key unit in the proposed architecture, can be scaled for the keys of length 192 and 256 bits easily.

REFERENCES

[1] S. P. Mohanty, K. R. Ramakrishnan, and M. S. Kankanhalli, "A DCT Domain Visible Watermarking Technique for Images," in *Proc of the IEEE International Conf on Multimedia and Expo*, 2000, pp. 1029–1032.
[2] M. S. Kankanhalli and T. T. Guan, "Compressed-Domain Scrambler / Descrambler for Digital Video," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 356–365, May 2002.
[3] B. M. Macq and J. J. Quisquater, "Cryptography for Digital TV Broadcasting," *Proceedings of the IEEE*, vol. 83, no. 6, pp. 944–957, Jun 1995.
[4] H. Kuo and I. Verbauwhede, "Architectural Optimization for a 1.82 Gbits/sec VLSI Implementation of the AES Rijndael Algorithm," in *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, 2001, vol. 2162, pp. 51–64.
[5] M. McLoone and J. V. McCanny, "Rijndael FPGA Implementation Utilizing Look-up Tables," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2001, pp. 349–360.
[6] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization," in *Proceedings of Advances in Cryptology - ASIACRYPT 2001*, 2001, pp. 171–184.
[7] S. Mangard, M. Aigner, and S. Dominikus, "A Highly Regular and Scalable AES Hardware Architecture," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 483–491, April 2003.
[8] T. Sodon O. J. Hernandez and M. Adel, "Low-Cost Advanced Encryption Standard (AES) VLSI Architecture: A Minimalist Bit-Serial Approach," in *Proc of IEEE Southeast Conference*, 2005, pp. 121–125.
[9] J. Daemen and V. Rijmen, *The Design of Rijndael*, Springer-Verlag, 2002.
[10] A. J. Elbirt, W. Yip, B. Chetwynd, and Christof Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," in *Proceedings of the Third Advanced Encryption Standard (AES) Candidate Conference*, 2000, pp. 13–27.