An Area Optimized Reconfigurable Encryptor for AES-Rijndael

Monjur Alam Sonai Ray Debdeep Mukhopadhayay† Santosh Ghosh
Dipanwita RoyChowdhury Indranil Sengupta

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
† Indian Institute of Technology, Madras
{monjur, sonai, santosh, drc, isg}@cse.iitkgp.ernet.in
† {debdeep}@cse.iitm.ernet.in

Abstract

This paper presents a reconfigurable architecture of the Advanced Encryption Standard (AES-Rijndael) cryptosystem. The suggested reconfigurable architecture is capable of handling all possible combinations of standard bit lengths (128,192,256) of data and key. The fully rolled inner-pipelined architecture ensures lesser hardware complexity. The work develops a FSMD model based controller which is ideal for such iterative implementation of AES. S-boxes here have been implemented using combinational logic over composite field arithmetic which completely eliminates the need of any internal memory. The design has been implemented on Xilinx Vertex XCV1000 and 0.18μ CMOS technology. The performance of the architecture has been compared with existing results in the literature and has been found to be the most compact implementations of the AES algorithm.

1 Introduction

Rijndael block cipher algorithm [2] has been chosen by NIST as the new Advanced Encryption Standard (AES). It is a symmetric block cipher that can process 128, 192 and 256 bits message blocks and 128, 192, and 256 bits key lengths. Hardware implementation of AES is attractive since software implementation [1] is relatively slow. It is desirable to have reconfigurable AES architectures that can work under various combinations of block and key lengths.

Many FPGA [4, 9, 11] and ASIC [3, 5, 13, 14, 16] implementations for Rijndael have been reported to date. Without exploiting composite field arithmetic most of them have used look up tables to implement the non-linear S-box operations in their architecture, resulting in larger area requirements. Moreover, these implementations can only process blocks of 128 bits and keys of the same length. The effective application of composite field $GF((2^4)^2)$ arithmetic in S-box operation was proposed by Rudra *et al.* [5]. Among those who tried to produce a really compact implementation using composite field arithmetic, the works of Satoh *at el.* [6] and Wolkerstorfer *et al.* [7] can only process 128 bits block and key lengths. The reconfigurable AES Rijndael architecture proposed in this paper can process all nine combinations of key and data lengths. The effective use of composite field $GF((2^4)^2)$ helps to reduce the hardware complexity of the architecture.

The remainder of this paper is organized as follows. In Section 2, AES algorithm is briefly described. In Section 3 an overview of the architecture is presented. Section 4 discusses the encryption unit. Section 5 explains how round keys are generated. Section 6 explores the FSMD model to generate the control signals. Section 7 analyzes the results, followed by concluding remarks in section 8.

2 AES-Rijndael Algorithm

AES-Rijndael [2] can support variable block and key lengths of 128, 192 or 256-bits. The round transformation consists of four different transformations: *ByteSub, ShiftRow, MixColumn and AddRoundKey*. They are performed in this order with the exception of the final round which is slightly different. All transformations are based on byte-oriented operations. *AddRoundKey* consists of bitwise XOR operations. The transformations operate on the intermediate result, which is called the *State*. The *ByteSub* transformation is a non-linear byte substitution, also called S-box. The S-box is invertible and consists of the following two operations:

- Inversion in the $GF(2^8)$ field, modulo the irreducible polynomial m(x) = x^8 + x^4 + x^3 + x + 1.
- Affine transformation defined as: Y = AX^{-1} + B, where A is an 8×8 fixed matrix and B is an 8×1 vector-matrix.

978-3-9810801-2-4/DATE07 © 2007 EDAA

DataBlock	Row1	Row2	Row3
128	1-Byte	2-Byte	3-Byte
192	1-Byte	2-Byte	3-Byte
256	1-Byte	3-Byte	4-Byte

Table 1. Shift offsets for different blocks

In *ShiftRow*, the rows of the *State* are cyclically shifted over different offsets (Table 1 [2]); row 0 is not shifted.

The *MixColumn* transformation operates on each column of *State* individually. Each column of the *State* matrix is multiplied by a fixed matrix M and it is defined as:

$$\mathbf{M} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \quad (1)$$

3 Design Overview

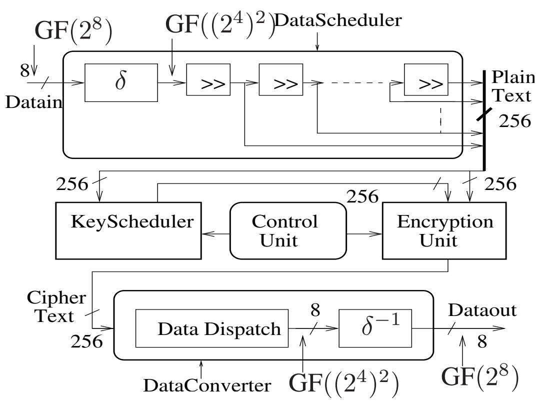

Figure 1. Top-Level Architecture

In the proposed work, some novel techniques are introduced for the implementation. The two main parts of AES algorithm, namely encryption and key scheduling, are considered for optimization. The major optimization criteria considered are maximization of hardware reduction and path delay reduction. For maximizing hardware reduction we focus on two things. Firstly, computation of non-linear operations like sub-bytes are implemented by combinational logic which makes sure that no internal memory is required. Moreover all computations involving S-boxes are done in $GF((2^4)^2)$ which reduces hardware and power consumption without imposing much penalty on the throughput [7]. The *DataScheduler* converts an 8-bit input element in $GF(2^8)$ to an element in $GF((2^4)^2)$ using an isomorphic function δ, while *DataConverter* does the reverse mapping (Figure 1). Secondly, reutilization of precomputed blocks are made as much as possible. For maximizing path delay reduction the architecture has adopted the concept of inner-pipelining. All operations in the *KeyScheduler* and the *Encryption Unit* are performed in $GF((2^4)^2)$.

4 Encryption Unit

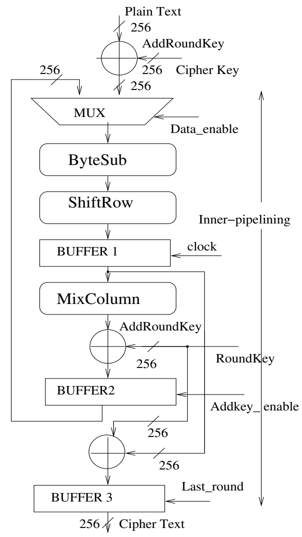

Figure 2. The Encryption Unit

During encryption, the data are organized conceptually in an 4x8 matrix of bytes. This organization is used for data block sizes of 256 bits. For smaller data block sizes (128 or 192 bits), the leftmost columns of the matrix are unused. The encryption unit is fully rolled which ensures a reduction in hardware. It has been implemented with inner pipelining technique for reduction of the combinational path delay. The encryption data path processes 32-byte block in parallel. A complete round transformation executes in two clock cycles. Each transformation is optimized appropriately for maximal performance. The data flow through various parts of the unit are controlled by three control signals, namely *Data_enable*, *Addkey_enable* and *Last_round*, that are generated by the control unit. Due to its rolling technique the design can support all standard modes of encryption operation including CBC, OFB etc.

4.1 S-box Design on Composite Fields

The major computation inside S-box is to find out the multiplicative inverse of an element in the finite field $GF(2^8)$. It is the most costly operation in terms of hardware and power [16]. For reducing cost associated with this operation, several authors have designed AES S-boxes based on composite field techniques [6, 7, 10]. The techniques use three-stage strategy (Figure 3):

- Map the element X $\in GF(2^8)$ to a composite field $F \in GF(2^4)^2)$ using an isomorphic function δ [6].
- Compute the multiplicative inverse over the field $F \in GF(2^4)^2)$.
- Finally map the computation result back to the original field using the inverse function δ^{-1}.

The primary drawback of these technique is that for processing 128-bit data in *ByteSub* operations, 32 isomorphic

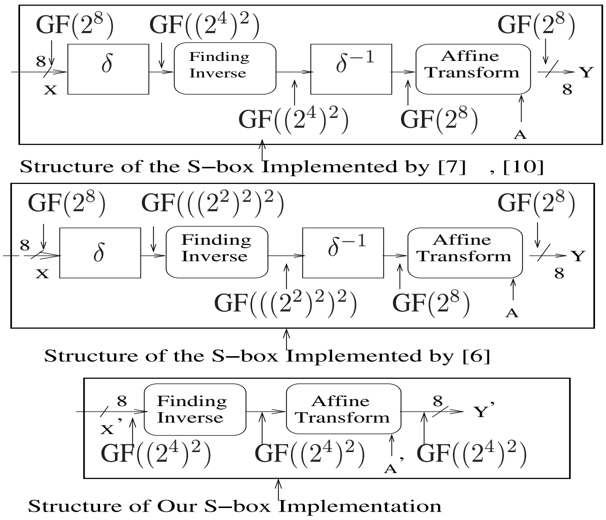

Figure 3. Structure of Different S-boxes

functions (δ and δ^{-1}) are to be used parallely, 16 for mapping $GF(2^8)$ elements to $GF((2^4)^2)$ elements and 16 for reverse conversion.

On the contrary our design uses only two isomorphic functions δ and δ^{-1}: one in *DataScheduler* and another in *DataConverter* (Figure 1). The function δ converts an element x in $GF(2^8)$ to an element in $GF((2^4)^2)$ [8]. The function δ is defined as $\delta(x)$ = T.x, where T is a transformation matrix and defined as:

$$\mathbf{T} = \begin{pmatrix} 1&0&1&1&1&0&1&1\\ 0&1&0&1&0&0&0&0\\ 0&1&0&0&1&0&1&0\\ 0&1&1&0&0&0&1&1\\ 0&0&0&0&1&1&1&0\\ 0&1&0&0&1&0&1&1\\ 0&0&1&1&0&1&0&1\\ 0&0&0&0&0&1&0&1 \end{pmatrix} \quad (2)$$

The function δ^{-1} does the reverse conversion. We have not used any transformation function (δ or δ^{-1}) in the S-box operation (Figure 3). The elements of standard affine matrix used in S-box is defined over the composite field $GF(2^8)$ [2]. The *ByteSub* (S-box) transformation is carried out in our design over the composite field $GF((2^4)^2)$. It is possible without imposing additional hardware overheads using the following technique:

- As discussed at Section 2, Y = AX^{-1} + B. In the composite field,

$$\begin{aligned} \delta(Y) &= \delta(AX^{-1}) + \delta(B) \\ &= \delta(A\delta^{-1}(\delta(X^{-1}))) + \delta(B) \\ &= A^{'}X^{'} + \delta(B), \end{aligned}$$

where $A^{'}$ = $\delta A \delta^{-1}$ and $X^{'}$ = $(\delta(X))^{-1}$. $A^{'}$ becomes:

$$\mathbf{A}^{'} = \begin{pmatrix} 0&0&1&0&1&0&0&0\\ 0&1&0&1&0&0&0&1\\ 0&0&1&0&0&1&0&0\\ 0&0&0&0&0&0&0&1\\ 1&0&1&0&0&1&0&0\\ 0&1&0&0&0&1&0&1\\ 0&0&1&0&1&0&1&0\\ 0&0&0&1&0&0&0&0 \end{pmatrix} \quad (3)$$

- Every $X \in GF((2^4)^2)$ can be represented as $(ax+b)$. The multiplicative inverse for an arbitrary polynomial $(ax+b)$ is given by: $(ax+b)^{-1}$ = a$(a^2\lambda$ + ab + $b^2)^{-1}$x + (a + b)$(a^2\lambda$ + ab + $b^2)^{-1}$, where λ is a primitive element of $GF(2^4)$

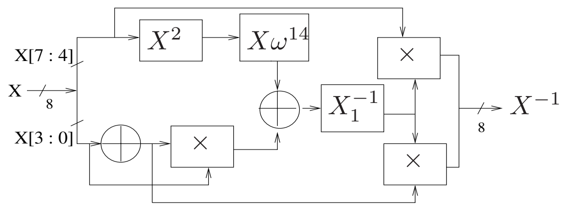

Figure 4. Structure of Inversion Calculation

Figure 4 depicts the corresponding block-diagram of three stage inverse multiplier. The primary goal of such design is to reduce hardware. The field polynomial used for the computations in $GF(2^4)$ is $(x^4 + x + 1)$. The multiplication employs modulo arithmetic of an irreducible polynomial $(x^2 + x + \lambda)$, where λ is a primitive in $GF(2^4)$. There are four such polynomials, for each of which there are seven different transformation matrices (T) [5]. We choose $\lambda = \omega^{14}$ for the best case result, where $\omega = (0010)_2$ is an element in $GF(2^4)$. The result is obtained out of logic synthesis with 0.18μ technology with minimum gate-count of 273 gates and maximum latency of 2.93 ns.

In our design, the S-box table requires 32 instances in the encryption unit and 8 instances in the key scheduling part. This makes area optimization of a single S-box instance an important factor in terms of the overall hardware overhead. The factors controlling the size of an S-box are a combination of the design and the logic-synthesis effort. Elements of the transformation matrix T are changed depending upon the values of λ. So, choosing the appropriate values of λ is another factor to optimize the size of the S-box. The total number of ‘1‘ entries in the fixed affine matrix A [2] is equal to 40. The total number of ‘1‘ entries in our new defined affine matrix $A^{'}$ is equal to 18. Implementing the matrices in a straightforward way, the number of XORs would be equal to the number of ‘1‘ entries minus the number of rows in the matrices. This would lead to an XOR gate count of 10, which results in a reduction of 22 XOR gates. Figure 5 illustrates the area-latency curves of different existing S-box implementations, along with our suggested one. The different area-latency values for a particular design are represented by the symbol $(\cdot)$. Three dots in the curve corresponding to our design has $\lambda = (\omega^{14}, \omega^{12}, \omega^{11})$. Our implementation is 1.047 times (286/273) better than the smallest one (Satoh *at el.* [6]) in terms of area.

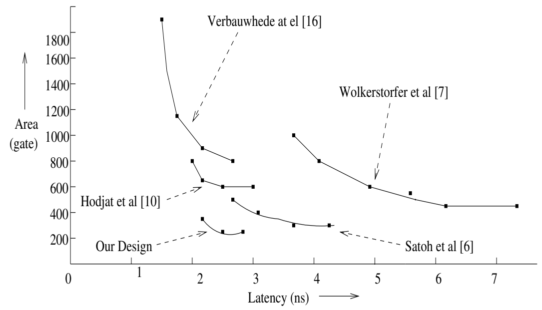

Figure 5. Area-Latency tradeoff for S-box

4.2 Subfield Implementation of MixColumn

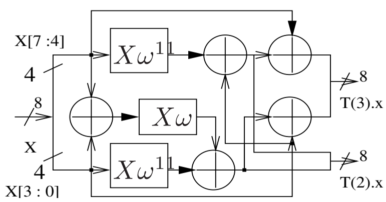

Figure 6. Structure of the MixColumn

In *Mixcolumn* transformation, the elements of the fixed matrix M belongs to $GF(2^8)$. Since, according to our design, all operations are in $GF((2^4)^2)$, all elements of M $\in GF(2^8)$ must be mapped into the elements of $M' \in GF((2^4)^2)$. This is done using a transform matrix T (equation 2). For example, T(2) = 00101110 = $(\omega^{11} + x\omega)$, where ω is the primitive element of $GF(2^4)$ and the irreducible polynomial for $GF((2^4)^2)$ is $(x^2 + x + \omega^{14})$. Figure 6 shows the structure of the *MixColumn* transformation.

5 Key Schedule Optimization

All computations on key expansion occur wordwise (32-bit). Let N_k denotes the length of the key divided by 32 and N_b denotes the length of the data block divided by 32. Let W[0],...,W[N_k-1] be the N_k columns of the original key. These N_k columns can be recursively expanded to obtain N_k*N_r more columns (*RoundKey*), where N_r is the number of round. The values of N_r are determined from the Table 2 [2]. Suppose that all columns up to W[i-1] have been expanded. The next column W[i] can be constructed as:

$$W[i] = \begin{cases} \text{W[i-}N_k\text{]}\oplus\text{T(W[i-1])} & \text{if } i \bmod N_k = 4 (N_k = 8) \\ \text{W[i-}N_k\text{]}\oplus\text{T(W[i-1],Rcon)} & \text{if } i \bmod N_k = 0 \\ \text{W[i-}N_k\text{]}\oplus\text{W[i-1]}) & \textit{otherwise} \end{cases}$$

N_r	N_b=4	N_b=6	N_b=8
N_k=4	10	12	14
N_k=6	12	12	14
N_k=8	14	14	14

Table 2. Numbers of rounds (N_r) as a function of the block and key length

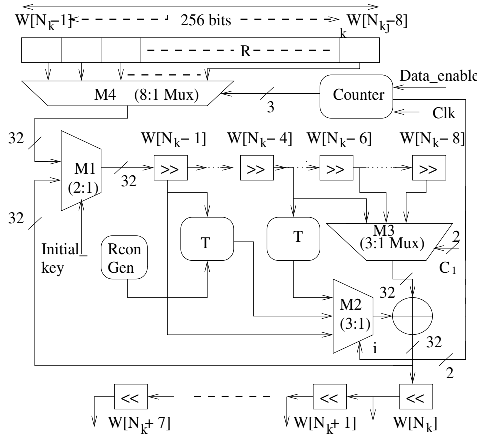

Figure 7. KeyScheduler

Here T(W[i-1],Rcon) is a non-linear transformation based on the application of the S-box to the four bytes of the column, and the addition of a round constant (Rcon) for symmetry elimination [2]. T(W[i-1]) represents the same without addition of Rcon. In Figure 7, the *W*'s are the 32-bit shift registers. *Initial_key* is generated from control unit, C_1 is user-specific and *i* is generated from an 8-bit up counter which is reset to value 1 by the *Data_enable* signal. The *Data_enable* signal is set when data blocks are ready to be processed. R is a 256-bit register to store initial key. When *Data_enable* signal is set, a single word (32 bits) among 8 words goes through the shift registers W[N_k-8],...,W[N_k-1] at every clock. The word is selected by 3 bits control signals generated by the counter. After N_k cycles all registers W[N_k-8],...,W[N_k-1] are occupied by the initial key stored at register R. Now at every clock single word of *RoundKey* is made. It goes through another set of 32-bit shift registers W[N_k],...,W[N_k+7] as round key and the same key is fed back to the register W[N_k-1] through the multiplexer M1 for generating the next round key. The architecture takes N_b clock cycles to generate single round key. A maximum of $8 \times 14 + 8 = 120$ clock cycles are required to generate complete round key (for 256 bits data and key). The primary goal of this architecture is to have a drastic reduction in hardware as compared to [5, 6, 10, 16].

6 The Control Unit

A Finite State Machine with Data-path (FSMD) is a universal specification model [12] which is used in the core of the present work with the addition of a reset state. This reset state can be treated as the start state of the FSMD.

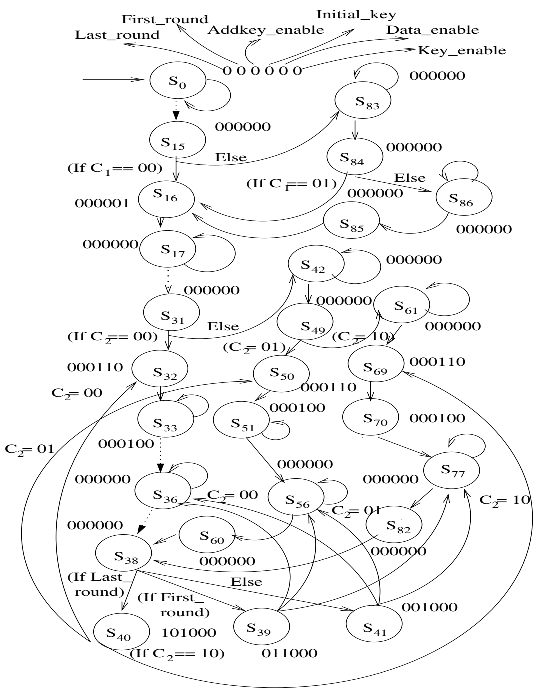

Figure 8. FSMD of Control Unit

In the Figure 8, $\{S_0, S_1, S_2, S_3, . . . S_{86}\}$ are the set of control states, S_0 = reset state, $\{clock, reset, C_2C_1\}$ = input signals, $\{O_0, O_1, O_2, O_3, O_4, O_5\}$ are the output signals where O_0 = *Key_enable*, O_1 = *Data_enable*, O_2 = *Initial_key*, O_3 = *Addkey_enable*, O_4 = *First_round* and O_5 = *Last_round*. C_2C_1 is a 4-bit control signal specified by user. The lower 2-bit C_1 specifies the key length and higher 2-bit C_2 specifies the block length. The *Key_enable* signal loads the initial key into the register R shown in Figure 7. *Data_enable* signal select 256 bits block among two 256 bits blocks, one coming from Plain Text and other generated as intermediate cipher shown in Figure 2. This signal is also used to initiate the value of the *Counter* shown in Figure 7. Figure 2 takes two clock cycles to complete a single round, but *KeyScheduler* (Figure 7) takes a minimum of 4 cycles for generating single round key. This is why *Addkey_enable* signal is used as an enable signal of BUFFER2 to get valid data from xor operation (Figure 2). Similarly, *Last_round* signal is used as an enable signal of BUFFER3 to get valid Cipher Text.

According to our design, the state transitions of the controller take place at every clock cycle. If the controller waits for j ($j > i$) clock cycles at state S_i then the next state is S_j and the controller self loops at S_i for (j-i) clock cycles. The transitions make several branches depending upon the control signal C_2C_1. Values of C_2C_1 for different values of key and data length are shown in Table 3. We need total (N_b*N_r+N_k) states, i.e, 8x14+8 = 120 states for 256 bits block and key length. But due to short of scope only 86 states are shown in the figure, through which it can be explained how does the FSMD work. Let us take an example.

Example 1 *In the example let us take 128 bits data and 128 bits key (i.e C_2C_1 = 0000).*

- *The controller starts at S_0 with the positive edge of reset. The consecutive 15 clock cycles states $\{S_0, S_1, . . . S_{15}\}$ have same status and in those cases all outputs are 0.*
- *At the next state S_{16} (16th cycle) Key_enable signal becomes 1 and rest are 0. It signifies that 128 bits data are stored in register R as initial key. As we are taking 8 bits data from I/O at every cycle, after 16 cycles we can get 128 bits data.*
- *In next cycle (state S_{17}) all output signals are set to 0. Similarly after 15 cycles at state S_{32} Data_enable and Initial_key are set to 1. In consecutive 4 cycles Initial_key signal sets to 1. At these stages Mux M1 selects 4 words from R (Figure 7) and those words are stored in shift registers W[N_k-4],...,W[N_k-1], i.e, W[0],...,W[3]. Now the words of the key are ready to be expanded.*
- *At next 4 cycles 128 bits round key is made. At S_{38} there are three branching S_{39}, S_{40} and S_{41}. Those branching take place depending upon the signals Last_round and First_round.*
- *All those states except S_{40} come back to S_{36} in the next cycle signifying that the expansion of next round keys can start. State S_{40} comes back to S_{32}. It signifies that all 10 rounds key generation are completed and ready to store initial keys from register R to the shift registers W[N_k-4],...,W[N_k-1], i.e, W[0],...,W[3] to generate next 10 round key for next cipher text.*

C_2C_1		Key		
		128	192	256
Data	128	0000	0100	1000
	192	0001	0101	1001
	256	0010	0110	1010

Table 3. C_2C_1 for different Key and Data

7 Implementation Results and Comparison

The proposed design has been implemented on Xilinx Vertex XCV1000 hardware platform and simulated by ModelSim8.1i. The same design has been compiled in 0.18 μ CMOS using Synopsys design tool (Design Compiler). The performances (throughput and frequency) are shown at Table 4 and Table 5. Throughput (τ) is calculated as:
τ = ($\eta \times f$)/(ψ), where η, f and ψ stand for block length, clock frequency and number of clock cycle respectively.
In our design N_b clock cycles are needed to generate single round intermediate cipher.
τ = ($N_b \times 32 \times f$)/($N_b \times N_r$) = ($32 \times f$)/N_r.

Clock Frequency = 37.73 MHz			
Throughput (Mb/s)	N_b=4	N_b=6	N_b=8
N_k=4	120.74	100.61	86.24
N_k=6	100.61	100.61	86.24
N_k=8	86.24	86.24	86.24

Table 4. Throughput in FPGA

Clock Frequency = 120 MHz			
Throughput (Mb/s)	N_b=4	N_b=6	N_b=8
N_k=4	384	320	274.28
N_k=6	320	320	274.28
N_k=8	274.28	274.28	274.28

Table 5. Throughput in 0.18μ CMOS

	E	D	*Key*			*Data*		
			128	192	256	128	192	256
[4]	•	•	•			•		
[11]	•		•			•		
0.3μ Technology								
[14]	•	•	•			•		
0.18μ Technology								
[13]	•		•	•	•	•		
[16]	•		•	•	•	•	•	•
Our	•		•	•	•	•	•	•

Table 6. Features of compared Architectures

In Table 6 and Table 7 we show comparative analysis of different existing AES architectures, along with our suggested one. The symbol • signifies that the design does offer the choice. E and D stand for Encryption and Decryption respectively.

We do not take into account any implementation based on fully-pipelining [3, 5, 6, 10] as they give greater throughput at the expense of larger area. We have not shown any comparative analysis of performance in FPGA as it can be misleading when dedicated memories and RAMs are present in the design. Moreover, there is no FPGA based reconfigurable AES-Rijndael which is capable of handling all possible combinations (128,192,256) of data and key.

8 Conclusions

We have presented AES encryptor core design in rolling and inner-pipelined fashion. The design strikes an optimal balance between area and frequency of operation. Results and comparisons with existing works have been furnished. The paper demonstrates that the proposed architecture outperforms prior results with respect to the parameter throughput per area. Implementation of encryptor/decryptor core designs into a single-chip exploiting extended composite subfield $GF(((2^2)^2)^2)$ may be included as a future work.

	Frequency (MHz)	Throughput (Mb/s)	Gate (K)	Throughput per kilo gates
[14]	80	9.9	3.4	2.91
[15]	50	70	7	10
[16]	125	2290	173	13.23
[13]	132	2400	149	16.17
Our	**120**	**384**	**21**	**18.21**

Table 7. Performances of compared cores

References

[1] Guido Bertoni et al: *Efficient Software Implementation of AES on 32-bits Platforms*: CHES 2002, Revised Papers, LNCS Vol. 2523, pp.159-171, Springer-Verlag.

[2] J.Daemen and V.Rijmen (2002); *The Design of Rijndael: AES-The Advanced Encryption Standard*: Springer-Verlag Berlin Heidelberg, New York, 2002.

[3] D. Mukhopadhyay, D. RoyChowdhury: *An Efficient End to End Design of Rijndael Cryptosystem in 0.18μ CMOS*: The 18th International Conference on VLSI Design and The 4th International Conference on Embedded Systems (VLSID'05), pp.405-410.

[4] V. Fischer and M. Drutarovsky *Two Methods of Rijndael Implementation in reconfigurable Hardware*: CHES 2001, LNCS Vol. 2162, pp.77-92, Springer-Verlag.

[5] A.Rudra et al; *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic*: CHES 2001, LNCS Vol. 2162, pp.171-184, Springer-Verlag 2001.

[6] A. Satoh, S. Morioka, K. Takona and S. Munetoh: *A Compact Rijndael Hardware Architecture with S-Box optimization*: Proceedings of Advances in Cryptography-ASIACRYPT 2001, LNCS Vol. 2248, pp.239-254, Springer-Verlag.

[7] J. Wolkerstorfer, E.Oswald and M. Lamberger: *An ASIC Implementation of the AES S-boxes*: in Proc. RSA Conference (CT-RSA) 2002, LNCS Vol. 2271, pp.67-78, San Jose, CA, Feb. 2002.

[8] C. Paar: *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*: PhD thesis, Institute of Experimental Mathematics, University of Essen, Germany, 1994.

[9] P. Chodowiec and K. Gaj: *Very Compact FPGA Implementation of the AES Algorithm*: CHES 2003, LNCS Vol. 2779, pp.319-333, Springer-Verlag.

[10] A. Hodjat, I. Verbauwhede: *Area Throughput Tradeoffs for Fully Pipelined 30 to 70 Gbits/s Aes Processors*: IEEE Transaction on Computers, Vol. 55, No. 4, pp.83-88, 2006.

[11] A. J. Elbirt, W. Yip, B. Chetwynd, C. Paar: *An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists*: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9, No. 4, pp.545-557, 2001.

[12] D. D. Gajaki and L. Ramachandran: *Introduction to high-level synthesis*: IEEE Transaction on Design and Test of Computers, Vol. 11, No. 4, pp.44-54, Dec, 1994.

[13] R. Sever, A. Neslin, Y. Tekmen, M. Asker *A High Speed ASIC Implementation of the Rijndael Algorithm*: International Symphosium of Circuit and System (ISCAS-2004), IEEE Vol.2, pp.541-4.

[14] M. Feldhofer, J. Wolkerstorfer, J. Rijmen *AES implementation on a grain of sand*: IEE Procidings in Information Security, July, 2005.

[15] N. Praustaller, S. Mangard, S, Dominikus, J. Wolkerstorfer *Efficient AES implementation on ASIC's and FPGA's*: Proc. Fourth Workshop on the Advanced Encryption Standard (AES 2004), LNCS Vol. 3373, pp.98-112, Springer-Verlag, 2004.

[16] I. Verbauwhede, P. Schaumont, H. Kuo *Design and Performance Testing of a 2.29-GB/s Rijndael Processor*: IEEE Journal of Solid State Circuit, Vol. 38, No. 3, pp.569-572, March 2003.