Two Approaches for a Single-Chip FPGA Implementation of an Encryptor/Decryptor AES Core

Nazar A. Saqib, Francisco Rodríguez-Henríquez and Arturo Díaz-Pérez

Computer Science Section, Electrical Engineering Department
Centro de Investigación y de Estudios Avanzados del IPN
Av. Instituto Politécnico Nacional No. 2508, México D.F.
nabbas@computacion.cs.cinvestav.mx
{francisco, adiaz}@cs.cinvestav.mx

Abstract. In this paper we present the design of a full encryptor/decryptor core for AES algorithm which fits in single-chip FPGA. Our design performs encryption, decryption and key scheduling. To increase performance, a pipelined architecture is being proposed. In addition, several modifications to standard AES algorithm's formulations have introduced that allow us to obtain a significant reduction in the total number of computations and the path delay associated to them. Two approaches have been followed to implement the most costly step of AES, multiplicative inverse in GF(2^8). The first approach uses pre-computed values stored in a lookup table. The second approach simplifies computations to reduce memory requirements at the cost of increasing time. The obtained results indicate that both designs are competitive with the fastest complete AES FGPA core implementation reported to date. Our first approach requires up to 11.8% less CLB slices, 21.5% less BRAMs and yields up to 18.5% higher throughput than the fastest one.

1. Introduction

Recently, Rijndael block cipher algorithm was chosen by NIST as the new Advanced Encryption Standard (AES) [2, 13]. Rijndael is a block cipher that can process blocks of 128, 192 and 256 bits and keys of the same lengths, but for the official AES version, the only legal block length is 128 bits.

FPGA AES implementations are attractive since costs of VLSI design and fabrication can be reduced. However, AES hardware implementation poses a challenge since encryption and decryption processes are not completely symmetrical [2, 7, 13]. Designing separated architectures for encryption and decryption processes would imply the allocation of a large amount of FPGA resources. Designs reported in [3, 4, 5] have considered only the encryption part of AES. Only a single FPGA implementation of a full encryptor/decryptor AES core has been reported yet [9]. Performance results for these designs are broadly variable; they range from 300 Mbit/s to 3.2 Gbit/s, approximately.

In this paper, we describe a fully pipelined AES implementation core for an FPGA device. It is a complete encryptor/decryptor core for which encryption, decryption and

key scheduling work efficiently. We propose two approaches to implement multiplicative inverse for GF(2^8) which is the most costly operation of AES. The first design use pre-computed values through a lookup table requiring fast memory access to obtain a good throughput. The second approach eliminates memory requirements at the cost of more FPGA resources. Obtained results show that both designs are competitive with the previous full AES core reported in [9].

In Section 2, AES algorithm is briefly described. Several transformations to AES algorithm's expressions, incorporated to gain performance in hardware, are explained in detail in Section 3. In Section 4, we discuss a three-stage computation to calculate multiplicative inverse in the finite field GF(2^8). In Section 5, the FPGA implementation of a fully pipelined AES algorithm is presented and performance results are provided. Concluding remarks are presented at the end.

2. The AES algorithm

The AES cipher treats the input 128-bit block as a group of 16 bytes organized in a 4×4 matrix called *State* matrix. Fig. 1 depicts the AES cipher algorithm flow for encrypting one block of input data utilizing a given user key.

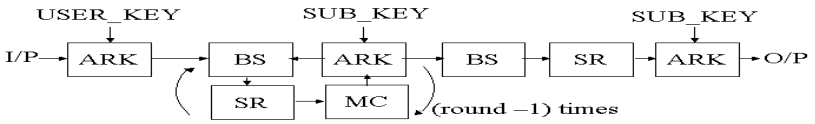

Figure 1. Basic AES encryption flow.

The algorithm consists of an initial transformation, followed by a main loop where nine iterations called *rounds* are executed. Each round is composed of a sequence of four transformations: Byte Substitution (BS), ShiftRows (SR), MixColumns (MC) and AddRoundKey (ARK). For each round of the main loop, a round key is derived from the original key through a process called *Key Scheduling*. Finally, a last round consisting of three transformations, BS, SR and ARK, is executed. The AES decryption algorithm operates similarly by applying the inverse of all the transformations described above in reverse order. In the rest of this section we will briefly describe the four AES round transformations BS, SR, MC and ARK.

In *ByteSubstitution* (BS), Each input byte of the State matrix is independently replaced by another byte from a look-up table called S-box. The AES S-box is a 256-entry table composed of two transformations: First each input byte is replaced with its multiplicative inverse in $GF(2^8)$ with the element {00} being mapped onto itself; followed by an affine transformation over $GF(2)$ [2, 13]. For decryption, inverse S-box is obtained by applying inverse affine transformation followed by multiplicative inversion in $GF(2^8)$ [13]. *ShiftRows* (SR): is a cyclic shift operation where each row is rotated cyclically to the left using 0,1,2 and 3-byte offset for encryption while for decryption, rotation is applied to the right. In *MixColums*(MC) transformation, each column of the *State* matrix is multiplied by a constant fixed matrix as follows,

$$\begin{bmatrix} c'_{0,i} \\ c'_{1,i} \\ c'_{2,i} \\ c'_{3,i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,i} \\ c_{1,i} \\ c_{2,i} \\ c_{3,i} \end{bmatrix} \qquad i = 0,1,2,3 \tag{1}$$

Similarly, for the decryption, we compute Inverse MixColumns, by multiplying each column of the State matrix by a constant fixed matrix as shown below

$$\begin{bmatrix} c'_{0,i} \\ c'_{1,i} \\ c'_{2,i} \\ c'_{3,i} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} c_{0,i} \\ c_{1,i} \\ c_{2,i} \\ c_{3,i} \end{bmatrix} \qquad i = 0,1,2,3 \tag{2}$$

Finally, in the *AddRoundKey* (ARK), the output of MC is XOR-ed with the corresponding round sub-key derived from the user key. The ARK step is essentially the same for the AES encryption and decryption processes.

3. Novel Computational Expressions for Implementing AES on FPGA Devices

In this section, we introduce some novel techniques for implementing AES algorithm on FPGA devices. The three main AES algorithms, key schedule, encryption and decryption, are considered for optimization. Our optimization criteria are mainly based on three main factors: To maximize path delay reductions, reutilization of pre-computed blocks and to exploit the full resources of the target device. In addition, we have tried to use 4-input logic gates wherever is possible, since it has been demonstrated good use of CLBs.

3.1 AES algorithm optimizations

S-box and Inverse S-box implementation. S-box and the inverse S-box are required to compute BS step of AES. Both may be computed by implementing the affine (AF) and inverse affine (IAF) transformations together with a look-up table for Multiplicative Inverse (MI). In this way, the combination MI + AF provides S-box for encryption, while IAF + MI computes the Inverse S-box needed for decryption. To use only one MI module, a multiplexer is used to switch the data path for encryption/decryption, as shown in Fig. 2. Two approaches to implement MI are discussed in Section 4.

SR and ISR Implementation: These steps do not require FPGA resources as they can be implemented by rewiring. For the sake of symmetry, ISR step is embedded into IAF while SR and AF steps are joined together.

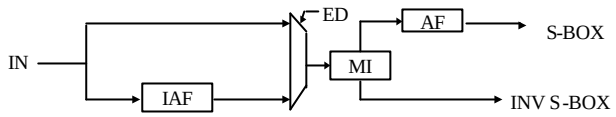

Figure 2. S-box and inverse S-box implementation.

MC and IMC Implementation: We review these two transformations in deep. Encryption's MC can be efficiently computed by using only 3 steps [1]: a sum step, a doubling step and a final sum step. Further optimization consists on embedding ARK step to fully exploit the 4-input FPGA slice resources. Let the elements of State matrix's column one be $a[0]$, $a[1]$, $a[2]$, $a[3]$, and let $k[0]$, $k[1]$, $k[2]$ and $k[3]$ represent the first four bytes of a key block, then the transformed matrix (MC + ARK) column $a'[0]$, $a'[1]$, $a'[2]$ and a'[3] can be efficently obtained as shown in Table 1.

Table 1. The modified MC expression with ARK.

Step 1	Step 2	Step 3
$v = a[1] \oplus a[2] \oplus a[3]$	$xt_0 = xtime(a[0])$	$a'[0] = k[0] \oplus v \oplus xt_o \oplus xt_1$
$v = a[0] \oplus a[2] \oplus a[3]$	$xt_1 = xtime(a[1])$	$a'[1] = k[1] \oplus v \oplus xt_1 \oplus xt_2$
$v = a[0] \oplus a[1] \oplus a[3]$	$xt_2 = xtime(a[2])$	$a'[2] = k[2] \oplus v \oplus xt_2 \oplus xt_3$
$v = a[0] \oplus a[1] \oplus a[2]$	$xt_3 = xtime(a[3])$	$a'[3] = k[3] \oplus v \oplus xt_3 \oplus xt_0$

Here $xtime(v)$ represents the finite field multiplication of $02 \times v$, where 02 stands for the constant polynomial x in $GF(2^8)$. Note that computations in the same column can be performed in parallel.

The same strategy applied above for MC would yield up to seven steps to compute IMC (four sum steps and three doubling steps). The difference is due to the fact that coefficients in equation (2) have a higher Hamming weight than the ones in equation (1). In order to overcome this drawback we use the strategy depicted in Table 2 where IMC manipulation is re-structured and seven steps are cut to five steps with an additional step to include IARK.

Table 2. The modified IMC expression with IARK.

Step 1	Step 2	Step 3	Step 4	Step 5
$t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]$			$u = xtime(u)$	$t' = t \oplus u$
		$u = s'_0 \oplus s'_1 \oplus s'_2 \oplus s'_3$	Step 6	
$s_0 = xtime(a[0])$	$s'_0 = xtime(s_0)$	$v = s_0 \oplus s_1 \oplus s'_0 \oplus s'_2$	$a'[0] = a[0] \oplus t' \oplus v \oplus k[0]$	
$s_1 = xtime(a[1])$	$s'_1 = xtime(s_1)$	$v = s_1 \oplus s_2 \oplus s'_1 \oplus s'_3$	$a'[1] = a[1] \oplus t' \oplus v \oplus k[1]$	
$s_2 = xtime(a[2])$	$s'_2 = xtime(s_2)$	$v = s_2 \oplus s_3 \oplus s'_0 \oplus s'_2$	$a'[2] = a[2] \oplus t' \oplus v \oplus k[2]$	
$s_3 = xtime(a[3])$	$s'_3 = xtime(s_3)$	$v = s_3 \oplus s_0 \oplus s'_1 \oplus s'_3$	$a'[3] = a[3] \oplus t' \oplus v \oplus k[3]$	

ARK Implementation: As explained above, ARK step is embedded into the MC transformation. For final round (Round 10), MC and IMC steps are not executed, therefore, a separated implementation of ARK step is made.

The ideas discussed in this section can be implemented as shown in Fig. 3, where the block-diagram represents the proposed architecture for implementing the AES encryption/decryption processes on FPGA devices.

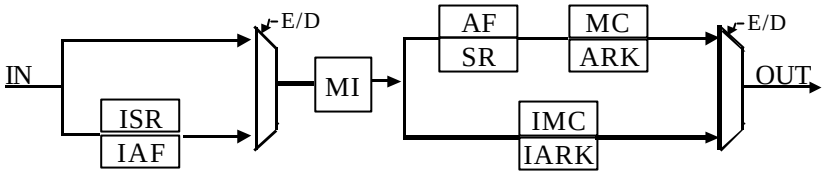

Figure 3. AES algorithm encryptor/decryptor implementation.

The proposed datapath for encryption and decryption can be as follows:
Encryption: MI+AF+ SR+MC+ARK
Decryption: ISR+IAF+MI+IMC+IARK

3.2 Key Schedule Optimization

The original user key consists of 128 bits arranged as a 4 x 4 matrix of bytes. Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be the four columns of the original key. Then, those four columns can be recursively expanded to obtain 40 more columns, as follows: Let the columns up to $w[i-1]$ have been defined then,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if } i \bmod 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \textit{otherwise} \end{cases} \quad (3)$$

Where $T(w[i-1])$ is a non-linear transformation based on the application of the S-box to the four bytes of the column, an additional cyclic rotation of the bytes within the column and the addition of a round constant (*rcon*) for symmetry elimination [2]. Let $[k_0, k_4, k_8, k_{12}]$, $[k_1, k_5, k_9, k_{13}]$, $[k_2, k_6, k_{10}, k_{14}]$ and $[k_3, k_7, k_{11}, k_{15}]$ be the first four columns of the key. By combining and parallelizing expressions, the first column of the new key $[k'_0, k'_4, k'_8, k'_{12}]$ can be calculated as shown in Table 3.

Table 3. Modified expressions for key schedule.

Step 1	Step 2
$k'_0 = k_0 \oplus Sbox(k_{13}) \oplus rcon$	$k'_4 = k_4 \oplus k'_0$ $k'_8 = k_4 \oplus k_8 \oplus k'_0$ $k'_{12} = k_{4.} \oplus k_8 \oplus k_{12} \oplus k'_0$

Sbox(k_{13}) refers to the data obtain by substituting k_{13}. The same process is used for calculating other 3 columns of the new key. In the same manner, next nine keys are obtained.

4. AES S-Box Design Based on Composite Field Techniques

The most costly operation in the BS transformation described in section 2.1, is the computation of the inverse multiplicative of a byte in the Finite field $GF(2^8)$ defined by the AES cipher. In an effort to reduce the costs associated to this operation, several authors have designed AES S-Box designs based on the composite field technique reported first in [10, 11, 12]. That technique uses a three-stage strategy: Map the element $A \in GF(2^8)$ to a composite field F using a isomorphism function δ. Compute the multiplicative inverse over the field F and finally map the computations results back to the original field.

In [6] an efficient method to compute the inverse multiplicative based on Fermat's little theorem was outlined. That method is useful because it allows us to compute the multiplicative inverse over a composite field $GF((2^m)^n)$ as a combination of operations over the ground field $GF(2^m)$. It is based on the following theorem:

Theorem [7,12]: The multiplicative inverse of an element A of the composite field $GF((2^n)^m)$, $A \neq 0$, can be computed by

$$A^{-1} = \left(A^r\right)^{-1} A^{r-1} \bmod P(x),$$
$$\text{Where } A^r \in GF\left(2^n\right) \text{and } r = \frac{2^{nm}-1}{2^n-1} \qquad (4)$$

An important observation of the above theorem is that the element A^r belongs to the ground field $GF(2^n)$. This remarkable characteristic can be exploited to obtain an efficient implementation of the inverse multiplicative over the composite field. By selecting m =4 and n = 2 in the above theorem we obtain r = 17 and,

$$A^{-1} = \left(A^r\right)^{-1} \cdot A^{r-1} = \left(A^{17}\right)^{-1} \cdot A^{16} \qquad (5)$$

In the case of AES, it is possible to construct a suitable composite field F, by using two degree-two extensions based on the following irreducible polynomials [10]:

$$\begin{aligned} F_1 &= GF\left(2^2\right): & P_0(x) &= x^2 + x + 1 \\ F_2 &= GF((2^2)^2): & P_1(y) &= y^2 + y + f \\ F_3 &= GF(((2^2)^2)^2): & P_2(z) &= z^2 + z + l \end{aligned} \qquad (6)$$

where $\varphi = \{10\}_2$, $\lambda = \{1100\}_2$.

The inverse multiplicative over the composite field F_2 defined in the equation (6), can be found as follows. Let $A \in F_2 = GF((2^2)^2)$ be defined in polynomial basis as $A = A_H y + A_L$, and let the the Galois field F_1, F_2, and F_3 be defined as shown in equation (5). Then it can be shown that

$$A^{16} = A_H\, y + (A_H + A_L);$$
$$A^{17} = A^{16} \cdot A = 0 \cdot y + \left(l A_H^{16} A_H + A_L^{16} A_L\right) = l A_H^2 + A_L^{16} A_L \quad (7)$$

Fig. 4 depicts the block-diagram to the three-stage inverse multiplier represented by equations (5) and (7). The circuits shown in Fig. 4 and Fig. 5 present a gate-level implementation of the aforementioned strategy.

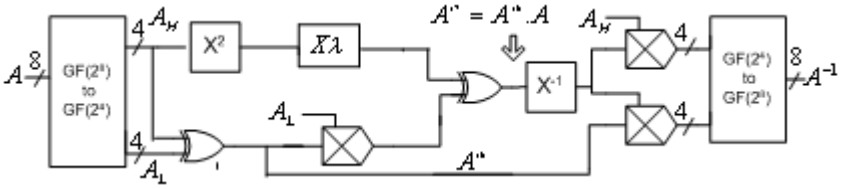

Figure 4. Three-stage strategy to compute multiplicative inverse in composite fields.

As we explained above, in order to obtain the multiplicative inverse of the element $A \in F = GF(2^8)$, we first map A to its equivalent representation (A_H, A_L) in the isomorphic field $F_2 = GF((2^2)^2)$ using the isomorphism d (and its corresponding inverse d^{-1}) given by [10]:

$$d = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}; d^{-1} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (8)$$

In order to map a given element A from the finite field F to its isomorphic composite field F_2 and vice versa, we only need to compute the matrix multiplication of the said element A, by the isomorphic functions shown in equation (8) After the transformation of the element A. Notice also that that taking advantage of the fact that A^{17} is an element of F_2, the final operation $\left(A^{17}\right)^{-1} \cdot A^{16}$ of equation (7), can be easily computed with further gate reduction. Last stage of the algorithm consists of mapping the computed value in the composite field, back to the field $F = GF(2^8)$.

5. Performance Results

To achieve high throughput, we have designed a pipelined architecture for AES as it is shown in Fig. 7. Eleven AES rounds have been unrolled to state a stage of the pipeline design. The design is symmetric in the sense that the same steps used for

encryption are re-used for decryption. The corresponding round-keys for encryption or decryption are generated from de input key accordingly eleven stages of the pipeline. Each stage is clock triggered and data is transferred to next stage at the rising-edge of the clock. The data blocks are accepted at each clock cycle and then after 11 cycles, output encrypted/decrypted blocks appear at the output at consecutive clock cycles.

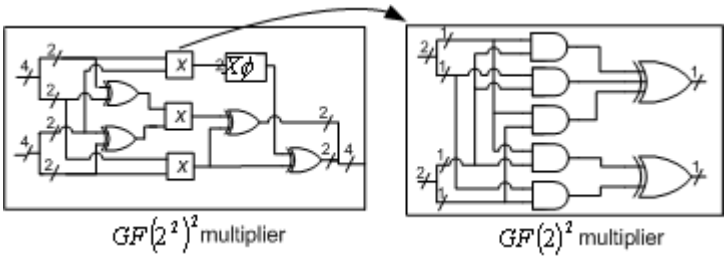

Figure 5. $GF((2^2)^2)$ and $GF(2^2)$ multipliers

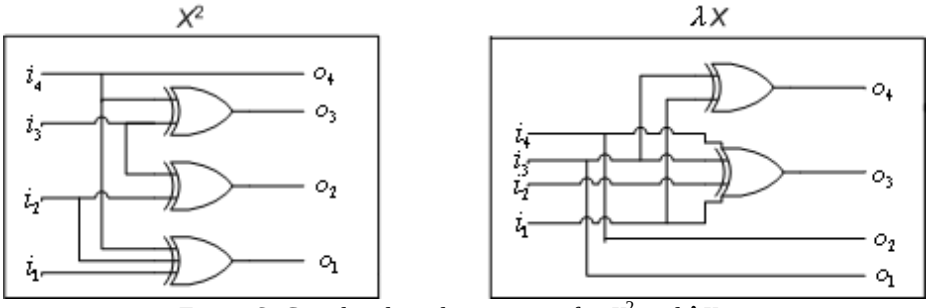

Figure 6. Gate-level implementation for X^2 and λX

AES memory requirements could be too high, especially for a pipeline design approach. To overcome this requirement, the design is implemented on a XCV2600E Xilinx Virtex-E. The Virtex and Virtex-E family of devices contains more than 280 BRAMs which are well suited for fast memory access [14]. A dual port BRAM can be configured into two single port BRAMs with independent data access. We also used Xilinx Foundation Series F4.1i for design entry, verification and synthesis.

Both approaches to implement MI, previously discused, follow the same pipeline architecture. The first design uses 43 BRAMs (43%) used for MI and occupies 386 I/O Blocks (48%) and 5677 slices (22.3%). The system runs at 30 MHz and data is processed at 3840 Mbits/s. The second design, three-stage MI computation, occupies 13416 CLB slices (52%) and 386 I/O Blocks(48%); no BRAMs are required here. The allowed system frequency is 24.5 MHz and the design achieves a throughput of 3136 Mbits/s. Table 4 details the total area required for each block shown in Figure 3.

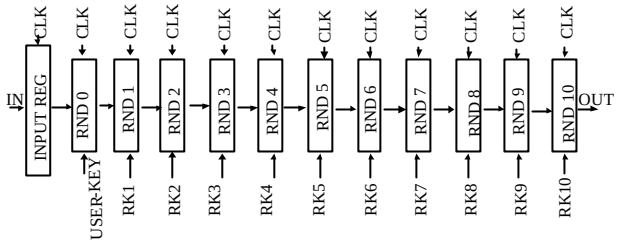

Figure 7. Pipeline approach for 128 bit AES Encryption/decryption

McLoone and J.V. McCanny [9] have reported an encryptor/decryptor core in FPGAs. It was implemented on a VirtexE XCV3200 FPGA device, uses 7576 CLB slices, 102 BRAMs and achieves a throughput of 3239 Mbits/s. The design uses same BRAMs for encryption/decryption and consumes 256 cycles to prepare BRAMs for encryption or decryption. The design also occupies 20 BRAMs for key scheduling. To the best of our knowledge, the design in [9] had been the best encryptor/decryptor AES core design for FPGA. Our first design compared with McLoone's has reduced the area requirements up to 11.8% while the expected throughput has been increased more than 18.5%. For the second design, area requirements are high but throughput is competitive with [9]. An advantage of this design could be that portability is possible since it does not uses BRAMs.

Table 4: Summary of features for AES encryptor/decryptor cores.

	E/D $GF(2^8)$	**E/D $GF(2^4)$**
BLOCK	**CLB SLICES**	**CLB SLICES**
KeyBlock	1278	1278
MI (10 rounds)	(80 BRAMS)	6676
SR	------	------
AF (10 rounds)	800	800
IAF (10 rounds)	640	640
MC + ARK (combined) (9 rounds)	1368	1368
IMC + IARK (combined) (9 rounds)	2088	2088
ARK (1^{st} + last round)	128	128
Misc. (Timing + I/O registers etc.)	374	438
TOTAL	6676 + (80 BRAMs)	13416 (NO BRAMs)
Throughput (Mbits/s)	3840	3136
Throughput/Area (Mbits/s/Slices)	0.58	0.24

6. Conclusions

We have presented two AES encryptor/decrypt core designs following a pipelined architecture. Both designs take advantage of using only one Sbox (to compute multiplicative inverse). The difference lies in the implementation of MI. In the first design, 80 BRAMs are utilized for MI pre-computed values. In the second design, a

three-stage architecture has been adopted for MI where calculations are made in $GF(2^4)^2$ and $GF((2^2)^2)^2$ instead of $GF(2^8)$. The goal was to reduce memory requirements as less as possible, then this design does not require BRAMs. The encryptor/decryptor starts reporting results only after 11 cycles needed to latch the round keys. We have re-organized MC and IMC computations to reduce data-path and to take advantage of four-input/one-output organization of CLBs. The ARK step has been embedded with MC and IMC step to enhance the timing performance.

Our designs have been implemented on Virtex-E family of devices (XCV2600) using Xilinx Foundation Series F4.1i. Our results have shown competetive behavior compared to the only full AES FPGA encryptor/decryptor known core. In the first design, we have outperformed design reported in [10]. The second design can be implemented on any FPGA family of devices. Future work includes extending the design for variable key and block lengths for AES algorithm.

7. References

1. Guido Bertoni et al: Efficient Software Implementation of AES on 32-bits Platforms: CHES2002, LNCS 2523, Springer-Verlag, 2002.
2. Joan Daemen, Vincent Rijmen: The Design of Rijndael, AES-The Advanced Encryption Standard: Springer-Verlag Berlin Heidelberg, New York, 2002.
3. A. Dandalis, V.K. Prasanna, J.D.P. Rolim: A Comparitive Study of Performance of AES Candidates Using FPGAs: The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.
4. J. Elbirt, W. Yip, B. Chetwynd and C. Paar: A FPGA implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists: The Third AES3 Candidate Conference, 13-14 April 2000, New York.
5. K. Gaj, P. Chodowiec: Comparison of the Hardware Performance of the AES Candidates using Reconfigurable Hardware: The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.
6. Brian Gladman: The AES Algorithm (AES) in C and C++: URL: http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm, April 2001.
7. Guajardo, C. Paar: Efficient Algorithms for Elliptic Curve Cryptosytems: CRYPTO '97, August 17-21, Santa Barbara, CA, USA.
8. T. Ichikawa, T. Kasuya, M. Matsui: Hardware Evaluation of the AES Finalists: The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.
9. Maire McLoone and J.V McCanny: High Performance FPGA Rijndael Algorithm Implementations: C. Koc, D. Naccache, and C.paar(Eds): CHES2001, LNCS 2162, pp. 65-76, Springer-Verlag, 2001.
10. S. Morioka and A. Satoh: An Optimized S-Box Circuit Architecture for Low Power AES Design: CHES2002, LNCS 2523, Springer-Verlag, 2002.
11. C. Paar: Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields: PhD thesis: Universitat GH Essen, VDI Verlag, 1994.
12. A. Rudra et al: Efficient Rijndael Encryption Implementation with Composed Field Arithmetic: CHES2001, LNCS 2162, pp. 65-76, Springer-Verlag, 2001.
13. W. Trappe and L. C. Washington: Introduction to Cryptography with Coding Theory: Prentice-Hall, Upper Saddle River, 2002.
14. Xilinx Virtex TM-E 1.8V Field Programmable Gate Arrays, URL: www.xilinx.com, November 2000.