Implementation of AES on FPGA Using Application-Specific Instruction Processor

P.Kokila
Assistant Professor/ECE, Nandha Engineering College, Erode, Tamil Nadu, India.

.T.G.Dhaarani
Assistant Professor/ECE, Nandha Engineering College, Erode, Tamil Nadu, India.

S.Nandhini
Associate Professor/ECE, Nandha Engineering College, Erode, Tamil Nadu, India.

P.Premkumar
Assistant Professor/ECE, Nandha Engineering College, Erode, Tamil Nadu, India.

Abstract – This paper presents two designs for the advanced encryption standard on field-programmable gate arrays (FPGAs) which occupies low area. The first design is an 8-bit application-specific instruction processor, which supports key expansion (currently programmed for a 128-bit key), encipher and decipher. The design utilizes less than 60% of the resources of the smallest available Xilinx Spartan II FPGA (XC2S15). The average encipher-decipher throughput is 2.1 Mbps when clocked at 70 MHz The design has numerous applications where low area and low power are priorities. The second design, using the Xilinx PicoBlaze soft core is included to provide an embedded 8-bit microcontroller comparison baseline.

Index Terms – Advanced encryption standard (AES), Application-specific instruction processor (ASIP), field-programmable gate array (FPGA).

1. INTRODUCTION

In January 1997, the National Institute of Standards and Technology (NIST) set out to establish a new standard of cryptographic algorithm to protect sensitive computer information and telecommunications systems in the Federal Government. The new algorithm would replace the aging Data Encryption Standard (DES) cipher algorithm, developed by IBM in the early 1970's. As a FIPS standard, AES will officially be identified as an approved cipher algorithm that can be used by U.S. Government organizations to protect sensitive (unclassified) information. Those Government organizations will be able to use the other FIPS approved algorithms in addition to, or in lieu of, AES.

The AES and its implementation for both application-specific instruction processor (ASIC) and field-programmable gate array (FPGA) technologies has been the subject of much research and continues to be a topic of interest in both academic and commercial environments.

In recent years, there has been a trend towards using FPGA in the production versions of electronic systems. It is no longer true that FPGAs are only used for prototyping. Their inclusion in the final version, would at first appear more expensive, however the ability to update the design and reduced time to market are strong commercial drivers. This was furthered by the introduction by the FPGA manufacturers of effectively mask programmed standard-cell versions of their technologies. This has resulted in an increased demand on optimal FPGA designs.

The main contribution of this paper is an ASIP capable of performing AES encipher and decipher operations using a truly8-bit datapath. The design avoids use of LUTs and proposes use of composite field data path for the SubBytes and InvSubBytes transformations. In addition, the *MixColumns* and all remaining operations are performed using a dedicated 8-bit Galois Field multiply-accumulate architecture. An iterative approach to multiplication was taken by implementing hardware support for finite-field doubling or finite-field multiplication by two (*ffm2*), halving and modulo-two addition. The ASIP achieves an average encipher-decipher throughput of 2.1 Mbps and utilizes less than two thirds of the resources of the smallest Xilinx Spartan-II part (XC2S15). To complete the design space exploration, a second design is also presented in this paper which utilizes the *PicoBlaze* [10] complex state machine soft core processor to provide a comparative 8-bit embedded processor design. The ASIP is shown to offer a threefold speed advantage for a similar area. The structure of this paper is organized as Follows. Section II briefly describes the AES followed by a description of the design in Section III. Section IV details the ASIP hardware followed by, in Section V, the corresponding software. The Results for FPGA implementation are given in Section VI .The paper ends by drawing some conclusions in Section VII.

2. AES

The AES has been fully documented in the freely available U.S. government publication FIPS-197 [12]. The standard comprises three block Ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as Rijndael. Each AES cipher has a 128-bit block size, with key sizes of 128, 192 and 256 bits, respectively. The 128-bit data block is divided into 16 bytes. These bytes are mapped to a 4x4 array called the State, and all the internal operations of the AES algorithm are performed on the State. The structure of AES is shown in Fig. 1. Each of the component operations are described below together with their respective inverses required for decipherment.

The *ShiftRows* operator is essentially a defined reordering of the bytes within the current state. The first row of the State does not change, while the second, third and fourth rows cyclically shift one byte, two bytes and three bytes to the left, respectively. The 128-bit data word, applied to *ShiftRows*, R(x) , and *InvShiftRows*, R^{-1} (x) , may be considered as a 4-by-4 matrix of 8-bit values. The byte order being ordered from x_{00} to x_{33}

$$x_{128BIT} = [x_{RC} : x_{00}, x_{10}, x_{20}, x_{30}, x_{01}, x_{11}, x_{21}, x_{31}, x_{02}, x_{12}, x_{22}, x_{32}, x_{03}, x_{13}, x_{23}, x_{33}] \quad (1)$$

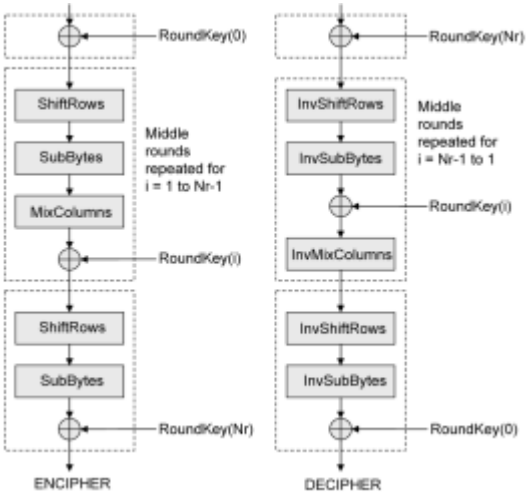

Fig. 1. Structure of AES

$$x = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$R(x) = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$R^{-1}(x) = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \quad (2)$$

SubBytes S(x) performs sixteen $GF(2^8)$ multiplicative inverse with irreducible polynomial $P(w) = w^8 + w^8 + w^8 + w + 1$,each inversion being followed by a specific affine transformation (Ax+B). The *InvSubBytes* operation $S^{-1}(x)$ can be similarly defined

$$x = x_0 + x_1 w + x_2 w^2 + x_3 w^3 + x_4 w^4 + x_5 w^5 + x_6 w^6 + x_7 w^7 \quad (3)$$

$$S(x) = A(x^{-1}) + B$$

$$S^{-1}(x) = [A(x+B)]^{-1} \quad (4)$$

Where,

$$A = w^0 \begin{bmatrix} 10001111 \\ 11000111 \\ 11100011 \\ 11110001 \\ 11111000 \\ 01111100 \\ 00111110 \\ 00011111 \end{bmatrix}_{w7} \qquad B = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}^{w0}_{w7}$$

The *MixColumns* operator m(x) performs a set of fixed-value GF multiplications

$$m(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

$$m^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \quad (5)$$

This may be conveniently written in matrix form for each column to give the *MixColumns* M(x) and *InvMixColumns* $M^{-1}(x)$ operations using GF multiplication modulo P(w) represented by the $\otimes$ symbol

$$M(x) = [M_0(x) M_1(x) M_2(x) M_3(x)]$$

$$M^{-1}(x) = [M_0^{-1}(x) M_1^{-1}(x) M_2^{-1}(x) M_3^{-1}(x)]$$

$$\text{where } M_c^{-1}(x) = \begin{bmatrix} 02\ 03\ 02\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \otimes \begin{bmatrix} x_{0c} \\ x_{1c} \\ x_{2c} \\ x_{3c} \end{bmatrix}$$

$$\text{and } M_c^{-1}(x) = \begin{bmatrix} 0e\ 0b\ 0d\ 09 \\ 09\ 0e\ 0b\ 0d \\ 0d\ 09\ 0e\ 0b \\ 0b\ 0d\ 09\ 0e \end{bmatrix} \otimes \begin{bmatrix} x_{0c} \\ x_{1c} \\ x_{2c} \\ x_{3c} \end{bmatrix} \quad (6)$$

The final operation *AddRoundKey* is simply the bitwise exclusive-or (XOR) of the current state and the *RoundKey*. The *Key-Expansion* utilizes four *SubBytes* operations followed by GF addition to yield the set of *RoundKeys*. Unfortunately, the order of use of the *RoundKeys* is reversed for the decipher data path thus it is necessary to compute the final *RoundKey* before deciphering data can proceed. The only method of doing this is to commence with the initial key and run through all the intermediate *RoundKeys* to reach the final (starting) value. The expansion operation also incorporates a byte-wise rotation and addition of a round specific constant, *Rcon*. These constants can be derived using *ffm2*. For 128-bit key, the *i*th *RoundKey* k^i is composed of k^i_C columns of byte values $k^i_{C,R}$ and is defined by the following equation:

$$k_0^i = \begin{bmatrix} s(k_{0,3}^{i-1}) + f^i(01) + k_{3,0}^{i-1} \\ s(k_{0,0}^{i-1}) + k_{3,1}^{i-1} \\ s(k_{0,1}^{i-1}) + k_{3,2}^{i-1} \\ s(k_{0,2}^{i-1}) + k_{3,3}^{i-1} \end{bmatrix}$$

and

$$k_j^i = k_j^{i-1} + k_{j-1}^i, \quad \text{for j = 1,2,3.} \quad (7)$$

3. ASIP DESIGN

The first decision was to select an appropriate datapath width for the processor. As already described in the introduction, a number of the previous low-resource designs had opted for a 32-bit datapath. Examination of the AES mathematics revealed the possibility of using an 8-bit datapath which had not been previously explored. Using less than 8 bits is believed to be impractical as the AES predominately uses 8-bit Galois Field arithmetic.

The design of the ASIP was an iterative process. The design was conceptually split into three principal areas: the hardware, the instruction set and the application program. The definition of the instruction set effectively formed a design partition between the software and hardware aspects. A number of design iterations were followed. This is the classical hardware-software co-design issue.

From the initial stages of the design, three key issues were identified which contributed to most of the area. The first concerned the computation of *SubBytes*, for which existing implementations vary from look-up tables to computing the function mathematically. The second, was the definition of a suitable primitive operation (namely *ffm*-accumulate) to efficiently perform the Galois Field mathematics in the AES *MixColumns*, *AddRoundKey*, and *KeyExpansion* operations. The final issue was program ROM size reduction for which the two traditional techniques of iteration and subroutines were considered. These three issues are discussed in detail in the following sections.

A. Low-Area Sub Bytes

The most obvious method for implementing the *SubBytes* operation on FPGA was using a look-up table (LUT) based around a block memory (the "S-box"). The table for the forward and inverse transformation would require 512 bytes (4kbits). Given the dual port nature of Xilinx block memories this ROM could be used for two simultaneous operations. Here, an alternative, lower area, solution was required. A number of existing works [1]–[10] demonstrated how *Sub-Bytes* may be computed using Composite Field mathematics rather than a LUT.

For a composite field value T in $GF((2^n)^2)$

$T = t_{Hx} + t_L$ where $t_{Hx} + t_L \in GF(2^n)$ (8)

With a primitive polynomial $P_{nm}(x)$

$P_{nm}(x) = x^2+x+\lambda, \quad \lambda \in GF(2^n)$ (9)

Then letting $\varphi=(t_H^2\lambda+t_L(t_L+t_H))^{-1}$, we have the inverse

$$T^{-1}=t_H\varphi x+(t_L+t_H)\varphi, \quad \text{where } t_H,t_L,\lambda,\varphi \in GF(2^n) \quad (10)$$

The composite field multiplication AB of two values A and B

$$A=A_Hx+A_L \text{ and } B=B_Hx+B_L \quad (11)$$

may be represented in terms of subfield arithmetic as

$$AB=x(A_HB_H+A_LB_H+A_HB_L) +\lambda A_HB_H+A_LB_L, \quad (12)$$

A further optimization can be made by describing this multiplication in Mastrovito form as

$$AB=x(A_L+A_H)(B_L+B_H)+A_LB_L) \quad (13)$$

$$+\lambda A_H B_H + A_L B_L .$$

In order to perform an equivalent inversion in composite field arithmetic additional isomorphic transformations are required. These can be found using the method described in Paar [7]. The composite field theory was applied a number of times to construct a set of fields starting with the base field and building up successively to reach. Each stage has its own primitive trinomial and binary value format. Table I summarizes the field construction.

The objective was to perform the multiplicative inverse of the supplied value in GF(((2²)²)²) over a number of cycles sharing the composite field multiplier. Here, the input byte is split into two 4-bit nibbles V=Az + B. The inversion is then given by the following equation:

$$V^{-1} = A\varphi z + (A+B)\varphi$$
$$\text{where } \varphi = (A^2\lambda + B(A+B))^{-1} \quad (14)$$

TABLE I

COMPOSITE FILD ARITHMETIC

Field	Polynomial	Bit Representation, $a_0 \ldots a_7$
$GF(2)$	n/a	a_0
$GF(2^2)$	$P_x(x) = x^2 + x + 1$	$a_1 x + a_0$
$GF((2^2)^2)$	$P_y(y) = y^2 + y + \phi$, $\phi \in GF(2^2)$	$y(a_3 x + a_2) + a_1 x + a_0$
$GF(((2^2)^2)^2)$	$P_z(z) = z^2 + z + \lambda$, $\lambda \in GF((2^2)^2)$	$z(a_7 xy + a_6 y + a_5 x + a_4) + a_3 xy + a_2 y + a_1 x + a_0$

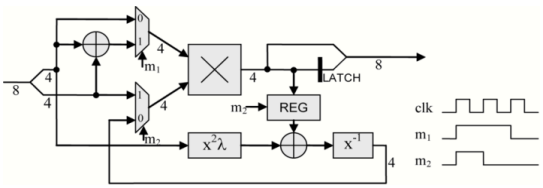

Fig.2. Block diagram of multiplicative inverse in GF(((2²)²)²).

The computational path of *SubBytes* was relatively long and this would dominate the cycle time of the entire processor. As the *SubBytes* operation was not the dominant operation in terms of quantity (as a fraction of the total instructions needed to perform the AES) this would have unduly limited the performance. Thus, *SubBytes* was split further into a total of five cycles to remove it from the critical path (Fig. 3).

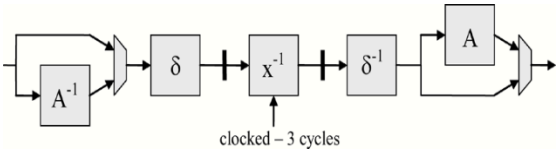

Fig.3. Block diagram of new subbytes circuit

This approach reduced the total forward and inverse *SubBytes* circuit to 42 slices on an XC2S15, a reduction in size of 27% compared to the original high-throughput version [9].

B. *8-BitffmAccumulate*

The AES *MixColumns* operator is fundamentally a 32-bit and there have been a number of designs [5][6] based around a 32-bit datapath. Only one design [9], for ASIC, was found which reported using an 8-bit datapath. However, the design married a 32-bit *MixColumns* to the 8-bit datapath by successively loading three 8-bit input registers in sequence to form the required 32-bit word with a similar process at the output. Here, a truly 8-bit alternative is sought with the corresponding area saving.

Examining the AES algorithm, a set of primitive operations were determined which cover the remaining operations of *ShiftRows*, *mixColumns*, and *KeyExpansion*. These were found to be *ffm2* and XOR. For this design, the decipher function was also required and as it is undesirable to store the entire set of *RoundKeys*, a further operation of finite-field halving or finite-field division by two (*ffd2*) was needed for reverse *KeyExpansion*. The *ShiftRows* operator was implemented as a set of 8-bit data moves between memory locations. Hardware implementation of the *ffm2* and halving is described by the following equations:

$$ffm2(d_{7-0}) = [d_6, d_5, d_4, d_3 \oplus d_7, d_1, d_0 \oplus d_7, d_7]$$
$$ffd2(d_{7-0}) = [d_0, d_7, d_6, d_5, d_4 \oplus d_0, d_3 \oplus d_0, d_2, d_1 \oplus d_0] \quad (15)$$

There are numerous examples in the *MixColumns* and *Key-Expansion* calculations where the result of an 8-bit operation was further acted upon. This was either in terms of repeated finite-field addition or repeated *ffm2*s. Thus, the inclusion in the datapath of an accumulator reduced the demands placed on the data memory. These requirements led to the development of a multiply-accumulate architecture capable of supporting moving 8-bit data 8-bit finite-field addition (XOR) and multiplication and division by two in GF(2^8). An execution unit specific to this type of operation was developed and its circuit is presented in Fig. 4.

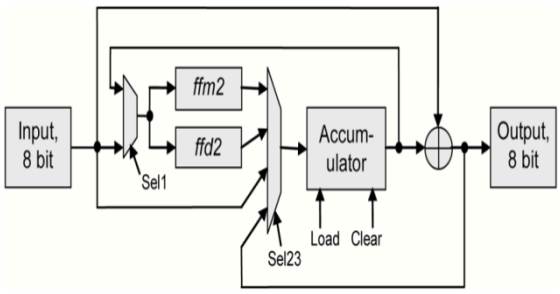

Fig. 4. Circuit diagram for "multiply-accumulate" functions.

C. Program ROM Size Reduction

One of the critical design decisions was which looping constructs, if any, were to be supported. A very simple processor could be constructed which only permitted execution of linear code. However, once the cost of the large program ROM size was balanced against the area and performance penalties for implementing even the most limited forms of iteration then linear code was no longer a viable option. The standard techniques for reducing the size of a program are iteration and subroutines. However, both techniques require specialist support from the processor hardware thus their inclusion would increase the area cost and complexity of the processor.

The final ASIP hardware provided support for one level of subroutines and two levels of iteration with one of the loop counters being used to conditionally provide indexed addressing. This enabled programming of the entire AES cipher process using only a few hundred instructions from an instruction set consisting of only 15 instructions.

4. ASIP HARDWARE

TABLE II

PROCESSOR INSTRUCTION SET

Opcode	Src. Addr.	Dst. Addr.	Operation
0000 nop	—	—	no operation
0001 unused	—	—	
0010 jsr	00 | A_L	00 | A_H	$rtn \leftarrow (pc)+1, pc \leftarrow A_H A_L$
0011 retn	—	111111	$pc \leftarrow (rtn)$
0100 ldloop	n 0 | cnt	111111	$\{x\|_{n=0}, y\|_{n=1}\} \leftarrow cnt$
0101 ldflag	RXZEIO	111111	$(flags) \leftarrow RXZEIO$
0110 djnz	n 0 | A_L	0 0 | A_H	$N \leftarrow N-1, N = \{x\|_{n=0}, y\|_{n=1}\}$ $if N \neq 0, then\ pc \leftarrow A_H A_L$
0111 outp	src	111111	$output \leftarrow (src)$
1000 sboxs	src	111111	$sb \leftarrow (src)$
1001 sboxc	src	dst	$(dst) \leftarrow sb$ repeat ♯4 times
1010 ldi	imm	dst	$(dst) \leftarrow imm$
1011 inp	—	dst	$(dst) \leftarrow (input)$
1100 ffmsrc	src	dst	$acc \leftarrow ffm2(src)$ $(dst) \leftarrow (src)\ xor\ acc_{prev}$
1101 ffmacc	src	dst	$acc \leftarrow ffm2(acc_{prev})$ $(dst) \leftarrow (src)\ xor\ acc_{prev}$
1110 xorsrc	src	dst	$acc \leftarrow (src)\ xor\ acc_{prev}$ $(dst) \leftarrow (src)\ xor\ acc_{prev}$
1111 xoracc	src	dst	$(dst) \leftarrow (src)\ xor\ acc_{prev}$

The traditional microcontroller architecture was adopted with separate program and data memories (i.e., Harvard Architecture).Two levels of looping were supported using two dedicated four bit counters X and Y. The loading of these was performed using the LDLOOP instruction and a single instruction DJNZ decreases a specified counter and performs a conditional jump if the value was nonzero. It was decided that a single four bit index, conditionally applied to source and destination RAM addresses, and associated with a loop counter (Y) was optimal. This addressing was enabled by the X flag and only operates on the lower 32 addresses (i.e., those associated with key and data and not the temporary storage). The value of the index was added modulo 16 or if the R flag was set, negated prior to the addition. The use of a single level of subroutines was supported using a dedicated return address register associated with the program counter. The JSR instruction calls a subroutine and the RETN instruction resumes execution at the instruction immediately after the previous JSR.

The complete instruction set for the processor is summarized in Table II. Fig. 5 shows the architecture of the processor. It should be noted that due to the clocking requirement of block memories, instructions take multiple cycles.

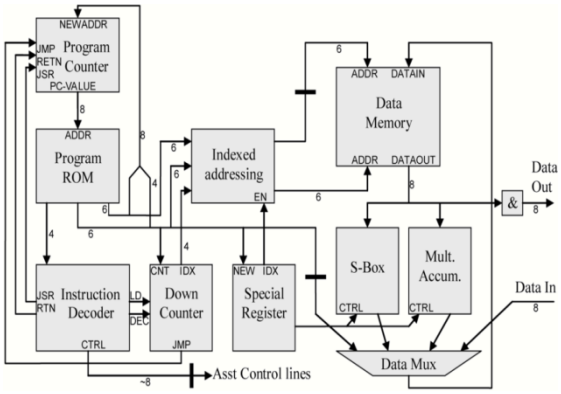

Fig. 5. ASIP Architecture

5. ASIP SOFTWARE

A. Forward Key Expansion

The key expansion, defined in the AES specification, can expressed as a set of operations which are performed each round to generate the next *RoundKey*.

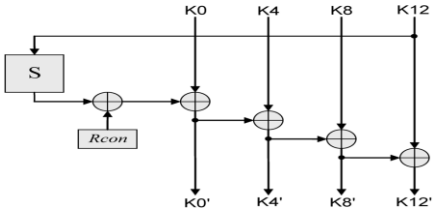

Fig. 6. Forward Key Expansion.

B. Reverse Key Expansion

Reverse key expansion was approached using a similar method to the forward key expansion. However, the process starts with ($k3^{'}, k7^{'}, k11^{'}, k15^{'}$) and works backwards to finally yield ($k0^{'}, k4^{'}, k8^{'}, k12^{'}$). This time (Fig. 7), the *Rcon* value was propagated in the reverse direction using *ffd2*.

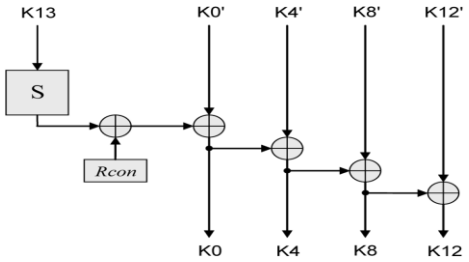

Fig. 7. Reverse Key Expansion

6. FPGA IMPLEMENTATION RESULTS

Fig. 8 shows that the placement of this design fits comfortably into the smallest Spartan-II device (XC2S15) occupying about 60% of the resources. The design required 145 slices (depending on user constraints) and two block memories. The block memory used as the register file was only partially utilized (360 bits) which gives rise to an alternative implementation using distributed memory with a cost of 42 additional slices and saving one of the block memories. No comparable 8-bit FPGA designs were found so comparison was made against the best 32-bit designs. Additionally, a second design was developed using the freely available Xilinx *PicoBlaze* core. This was done to provide a small embedded software baseline for comparison in terms of throughput and area. A concession was made in terms of implementing *SubBytes* as a ROM based lookup table.

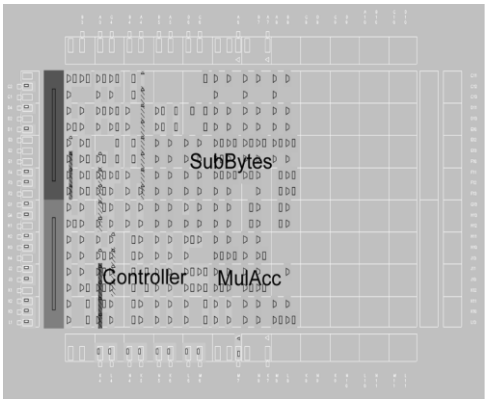

Fig. 8. Placement on XC2S15 FPGA

TABLE III

IMPLEMENTATION RESULTS

Design & FPGA (device)	ASIP(Spartan II XC2S15)	Picoblaze Spartan II (XC2S15)
Max.Clock Freq.(Mhz)	73	90
Datapath Bits	8	8
Slices	145	127
No.of BRAMS used	2	2
BRAM Size	4	4
Bits of BRAM used	4,580	10,676
Eqiv. slices for memory	140	333
Total Eqiv. Slices	250	451
Ave. Throughput(Mbps)	2.175	0.80
Performance, Typ. Throughput per slice	8.3	1.5

7. CONCLUSION

Both the ASIP and *PicoBlaze* based designs are the smallest known FPGA implementations to date. Such designs have application across a wide range of areas especially those needing a short time to market and relatively low power.

REFERENCES

[1] X. Zhang and K. K. Parhi, "High-speed VLSI architectures for the AES algorithm," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 12, no. 9, pp. 957–967, Sep. 2004.

[2] A. Hodjat and I. Verbauwhede, "A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA," in Proc. FCCM'04, Apr. 2004, pp. 308–309.

[3] J. Zambreno, D. Nguyen, and A. Choudhary, "Exploring Area/Delay Trade-Offs in an AES FPGA Implementation," in Proc. LNCS FPL'04, Antwerp, Belgium, 2004, vol.3203,pp.575-585.

[4] P. Chodowiec and Gaj, "Very Compact FPGA Implementation of the AES Algorithm," in Proc. LNCS'03,2003,vol.2779,pp.319-333.

[5] G. Rouvroy F. X. Standaert, J. J. Quisquater, and J. D.Legat," Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications," in Proc. ITCC'04, Apr. 2004, vol. 2, pp. 583–587.

[6] V. Fischer and M. Drutarovsky, "Two Methods of Rijndael Implementation in in Reconfigurable Hardware," in Proc. CHES'01, 2001, vol. 2162, pp. 77–92.

[7] F. X. Standaert, G. Rouvroy, J. Quisquater, and J. Legat, "A Methodology to Implement Block Ciphers in Reconfigurable Hardware and its Application to Fast and Compact AES RIJNDAEL," in Proc. ACMFPGA'03, Monterey, CA, 2003, pp. 216–224.

[8] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, "Strong Authention for RFID Systems Using the AES Algorithm," in Proc. LNCSCHES'04, 2004, pp. 357–370.

[9] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A Compact Rijndael Hardware ArchitectureWith S-Box Optimization," in Proc. LNCSASIACRYPT'01, Dec. 2001, vol. 2248, pp. 239–254.