Very Compact FPGA Implementation of the AES Algorithm

Paweł Chodowiec and Kris Gaj

George Mason University, MS1G5, 4400 University Drive, Fairfax, VA 22030, USA
{pchodow1, kgaj}@gmu.edu
http://ece.gmu.edu/crypto-text.htm

Abstract. In this paper a compact FPGA architecture for the AES algorithm with 128-bit key targeted for low-cost embedded applications is presented. Encryption, decryption and key schedule are all implemented using small resources of only 222 Slices and 3 Block RAMs. This implementation easily fits in a low-cost Xilinx Spartan II XC2S30 FPGA. This implementation can encrypt and decrypt data streams of 150 Mbps, which satisfies the needs of most embedded applications, including wireless communication. Specific features of Spartan II FPGAs enabling compact logic implementation are explored, and a new way of implementing *MixColumns* and *InvMixColumns* transformations using shared logic resources is presented.

1 Introduction

The National Institute of Standards and Technology (NIST) selected the Rijndael algorithm as the new Advanced Encryption Standard (AES) [29] in 2001. Numerous FPGA [2] [15] [16] [17] [18] [19] [20] [24] [25] [26] [27] [28] and ASIC [4] [6] [7] [8] [10] [11] implementations of the AES were previously proposed and evaluated. To date, most implementations feature high speeds and high costs suitable for high-end applications only.

The need for secure electronic data exchange will become increasingly more important. Therefore, the AES must be extended to low-end customer products, such as PDAs, wireless devices, and many other embedded applications. In order to achieve this goal, the AES implementations must become very inexpensive.

Most of the low-end applications do not require high encryption speeds. Current wireless networks achieve speeds up to 60 Mbps. Implementing security protocols, even for those low network speeds, significantly increases the requirements for computational power. For example, the processing power requirements for AES encryption at the speed of 10 Mbps are at the level of 206.3 MIPS [12]. In contrast, a state-of-the-art handset processor is capable of delivering approximately 150 MIPS at 133 MHz, and 235 MIPS at 206 MHz.

This paper attempts to create a bridge between performance and cost requirements of the embedded applications. As a result, a low-cost AES implementation for FPGA devices, capable of supporting most of the embedded applications, was developed and evaluated.

2 Related Work

Early AES designs were mostly straightforward implementations of various loop unrolled and pipelined architectures [24] [25] [26] [27] [28] with limited number of architectural optimizations, which resulted in poor resource utilization. For example, AES 8x8 S-boxes were implemented on LUTs as huge tables left for synthesizers to optimize.

Later FPGA implementations demonstrate better utilization of FPGA resources. Several architectures using dedicated on-chip memories implementing S-boxes and T-boxes were developed [15] [17] [18] [19] [20].

Recent research focused on fast pipelined implementations in both FPGA [2] [3] [14] [18] [19] [20] and ASIC [4] [6] [7] [11] worlds. Unfortunately, most of those implementations are too costly for practical applications.

The first significant step in compacting the AES implementation was made when V. Rijmen proposed an AES S-box implementation based on composite fields [31]. A similar solution was proposed by J. Wolkerstorfer [13]. Rijmen's idea has already been implementated in FPGA [2], and in ASICs [4] [6] [8]. S. Morioka et al. [10] went even farther and proposed a low-power compact S-box design suited for ASIC designs.

3 Architecture of the Compact Implementation

We began the design of the compact architecture by analyzing the basic architecture, as introduced in [26]. The basic architecture unrolls only one full cipher round, and iteratively loops data through this round until the entire encryption or decryption transformation is completed. Only one block of data is processed at a time making it equally suited for feedback and non-feedback modes of operation.

The structure of the AES round for encryption is shown in Fig. 1. The decryption round looks very similar, except it employs inverted operations in the following order: *InvShiftRows*, *InvSubBytes*, *AddRoundKey* and *InvMixColumns*. The *SubBytes* and *ShiftRows* operations in Fig. 1 are reordered compared to the cipher round depicted in the standard [29]. Their order is not significant because *SubBytes* operates on single bytes, and *ShiftRows* reorders bytes without altering them. This feature was used in our implementation.

The AES round shown in Fig. 1 reveals a great deal of parallelism. The data bytes are ordered from the most significant (byte 0) to the least significant (byte F) assuming big-endian representation. The round is composed of 16 8-bit S-boxes computing *SubBytes*, and four 32-bit *MixColumns* operations, working independent of each other. The only operation that spans throughout the entire 128-bit block is *ShiftRows*.

It is possible to implement only four *SubBytes* and one *MixColumns* in order to compact the AES implementation. Ideally, the resources should be cut by four, while execution of one round should take four clock cycles. This approach

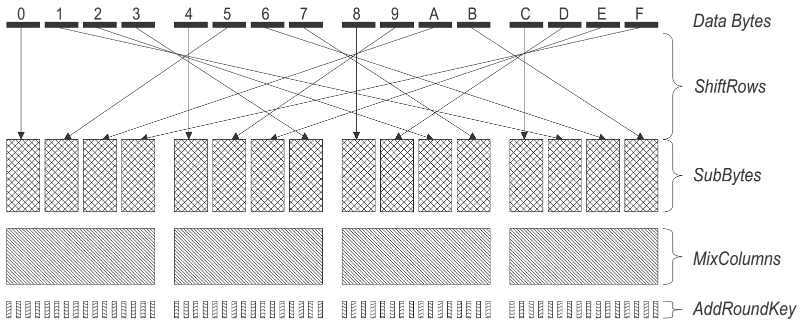

Fig. 1. Operations within AES encryption round

would result in approximately four times lower performance than for the basic architecture.

Cutting the resources by 75% may not appear easy. The *folded round*, as we call the modified round, still must transform 128 bits, and storage for all 128 bits of the data block must exist. Another complication is related to the implementation of the *ShiftRows* operation. The data bytes processed in the AES round cannot return to the same positions in the block register because it would not execute the *ShiftRows* operation. On the other hand, those same bytes cannot be placed into locations indicated by *ShiftRows* because those locations are occupied by other bytes that have not yet been processed. Therefore, additional bits of intermediate results must be stored, and more logic resources are needed.

One of the possible architectures for a folded implementation is shown in Fig. 2a. This architecture requires one 128-bit register, one 96-bit register and one 32-bit wide 4-to-1 multiplexer on top of the main cipher operations. The multiplexer becomes even bigger when both *ShiftRows* and *InvShiftRows* are implemented using same logic resources. The execution of one round takes four clock cycles. We believe that this, or very similar architecture, was implemented by A. Satoh et al. [23], but we cannot be sure since the authors do not provide enough detail. Their results show that the 4-cycle round takes 50% of the resources required by the 1-cycle round, and yields four times lower throughput.

Another possible architecture is shown in Fig. 2b. The 96-bit register is implemented as three 32-bit registers inserted into round operations creating a pipeline. In the case of FPGAs, those 32-bit registers will most likely be placed in the same Slices as logic operations yielding better resource utilization. The critical path is also shortened which permits the execution at a higher clock rate; however, the execution of the entire round requires seven, instead of four, clock cycles. We believe that this architecture was implemented by S. McMillan et al. [21], but again, we cannot be certain since the authors did not provide enough detail. S. McMillan et al. reported only slight difference of 48 Slices (16%), and large difference of 24 Block RAMs (75%), between 1-round unrolled and folded architecture.

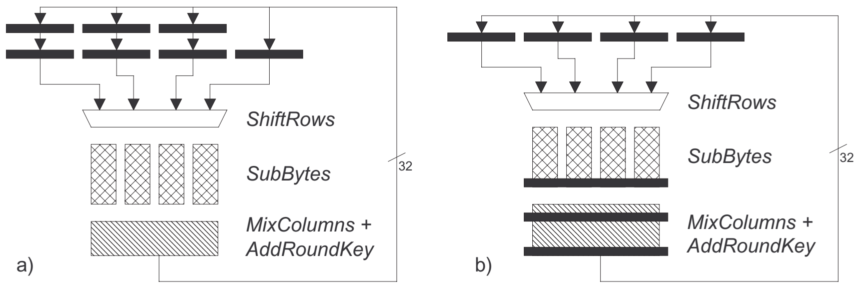

Fig. 2. Folded architecture. a) by A. Satoh et al. [23]; b) by S. McMillan et al. [21]

3.1 Implementation of a Folded Register

The two folded architectures described above are very straightforward and resulted in small logic savings. In order to create a folded architecture with better parameters, we decided to explore fine details of FPGA devices. We arranged data bytes into rows as shown in Fig. 3. This data arrangement is consistent with a *state* introduced in [30]. The following exercise can now be executed in steps:

1. Read input bytes: 0, 5, A, F; execute *SubBytes*, *MixColumns* and *AddRoundKey* on them; write results to the output at locations: 0, 1, 2, 3. This step is highlighted in the Fig. 3.
2. Repeat above operations for input bytes: 4, 9, E, 3; write results at output locations: 4, 5, 6, 7.
3. Repeat above operations for bytes: 8, D, 2, 7; write results at locations: 8, 9, A, B.
4. Repeat above operations for bytes: C, 1, 6, B; write results at locations: C, D, E, F. Output now becomes input for the next step.

In those four steps the entire AES round was executed including *ShiftRows* operation. At each step only one byte was read from each input row, and one byte was written to each output row. A similar exercise with identical conclusions can be executed for decryption transformation. Each row can be viewed as an addressable 8-bit wide memory. The correct execution of *ShiftRows* and *InvShiftRows* is now resolved to the proper addressing of each of the memories at the consecutive clock cycles. At the fourth clock cycle output memories become input memories and vice versa.

Dual-Port RAM Based Implementation. Each CLB Slice in Spartan II FPGA contains two look-up tables (LUT), which are the primary resources for logic implementation. Typically LUTs are configured as small 16x1 ROM tables

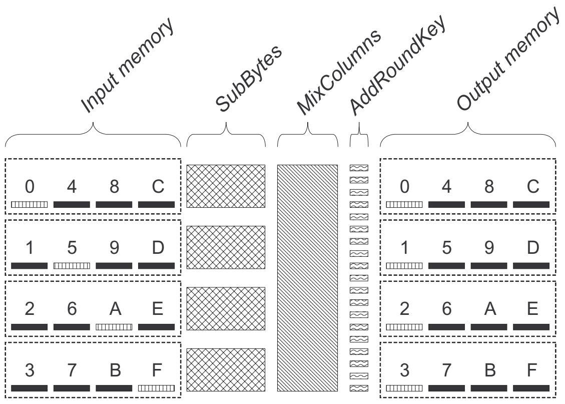

Fig. 3. Data arrangement in the folded architecture. Data bytes involved in the first step of calculation are highlighted

implementing logic functions of up to four inputs; however, other configurations are also possible. Two LUTs within the same Slice can implement a 16x1 Dual-Port RAM. An 8-bit wide Dual-Port RAM can be implemented using eight CLB Slices. This memory can be divided into two banks; each addressed by a different port. One port is used for reading data from the memory, while the other one for writes results back to the same memory. The switching between banks can be achieved by fliping one address bit in both ports every fourth clock cycle.

The Dual-Port RAM based solution has major advantages over solutions presented in Fig. 2:

– The logic resources required for storing intermediate results are far smaller.
– The multiplexer used before for *ShiftRows* and *InvShiftRows* is no longer needed.
– The complicated routing resulting from implementation of *ShiftRows* and *InvShiftRows* is avoided, yielding better performance.

Shift Register Based Implementation. A better solution may result from the following observation: all bytes from the output of *AddRoundKey* are written into consecutive locations in the output memory in consecutive clock cycles. Therefore, we could use a simple shift-register to shift computed data in without generating any addresses. Fortunately, LUTs can also be configured as 16-bit shift registers with variable taps, as shown in Fig. 4. Four Slices can implement an 8-bit wide, 16-bit long shift register. The input of the shift register is used for shifting results in while the output, selected dynamically by changing tap address, is used for reading data out. This solution encompasses all of the advantages of the Dual-Port RAM based solution, and requires less than a half of the logic resources than the Dual-Port RAM.

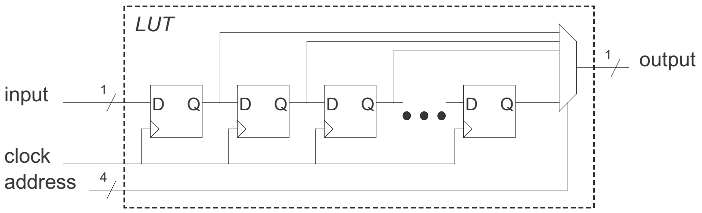

Fig. 4. Look-Up Table (LUT) configured as a shift register

3.2 Implementation of the *SubBytes* and *InvSubBytes*

Various area efficient implementations of AES S-boxes were proposed in [2] [4] [6] [8] [10] [13] [22] [23] [31]. All of those implementations are based on an idea of transforming the original $GF(2^8)$ field into a composite of smaller fields $GF\big((2^4)^2\big)$. It is a very attractive solution especially from the perspective of an ASIC because its implementation occupies a smaller area than a ROM. In the case of FPGAs, S-boxes can be mapped into dedicated Block RAMs treated as ROMs, or into LUTs. The latter approach could utilize the idea of composite fields. We decided to keep a good balance between utilization of LUTs and Block RAMs for the entire design, and implemented our S-boxes on dedicated Block RAMs.

Each Block RAM represents a dual-port memory of 4096 bits. Each port can be independently configured for different width and depth [34]. We selected a 512x8 configuration for each port, which provides access to the same memory space in the same way from both ports. A single *SubBytes* or *InvSubBytes* implementation requires a 256x8 ROM. A Block RAM has enough space to implement both *SubBytes* and *InvSubBytes*, as shown in Fig. 5. Each port has access to the entire memory space, and can perform a *SubBytes* or *InvSubBytes* transformation independently of each other. The folded architecture requires only 2 Block RAMs to implement four *SubBytes* and four *InvSubBytes* operations all together.

The Block RAM is a fully synchronous memory. Reading from it requires supplying the address one clock cycle before the data appears at the output. This feature can be viewed as a pipeline stage introducing a delay of one clock cycle. Execution of the entire round in such a circuit would take five clock cycles; however, a simple modification can be applied to maintain the execution rate at the level of four clock cycles per round. The trick is based on the fact that the folded register, described in section 3.1, does not transform data bytes in any

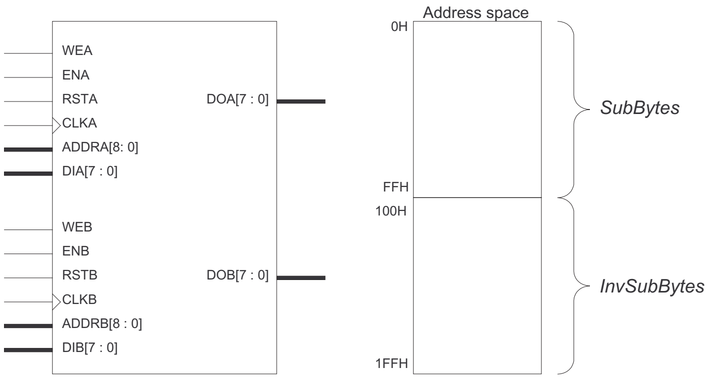

Fig. 5. Block RAM based implementation of *SubBytes* and *InvSubBytes*

other way than just reordering them. Therefore, this stage can be safely skipped if necessary. It apperars that forwarding of only one byte from the input to the folded register to the input of S-boxes is sufficient to maintain the execution rate of four clock cycles per round. Unfortunately, different bytes are forwarded in the case of encryption and in the case of decryption, as shown in Fig. 7.

3.3 Implementation of the *MixColumns* and *InvMixColumns*

The 32-bit input to the *MixColumns* transformation is represented as a polynomial of the form $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$, with coefficients in $GF(2^8)$. The coefficients of $a(x)$ are also polynomials of the form $b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$, with their own coefficients in $GF(2)$.

The *MixColumns* multiplies the input polynomial by a constant polynomial

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{1}$$

modulo $x^4 + 1$. The coefficients in $GF(2^8)$ are multiplied modulo $x^8 + x^4 + x^3 + x + 1$. The *InvMixColumns* multiplies the input polynomial by another constant polynomial:

$$d(x) = c^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \tag{2}$$

The implementation of the *MixColumns* is very simple because the coefficients of $c(x)$ are small. On the other hand, the *InvMixColumns* is far more complex and occupies larger area.

A. Satoh et al. [23] proposed an implementation based on the following idea:

$$d(x) = c(x) + e(x) + f(x) \tag{3}$$

where

$$e(x) = \{08\}x^3 + \{08\}x^2 + \{08\}x + \{08\} \tag{4}$$

$$f(x) = \{04\}x^2 + \{04\} \tag{5}$$

This implementation yields logic optimizations since *InvMixColumns* shares logic resources with *MixColumns*.

We propose a different method for exploring resource sharing. Our implementation is derived as follows:

$$c(x) \bullet d(x) = \{01\} \tag{6}$$

If we multiply both sides of the equation (6) by $d(x)$ we obtain:

$$c(x) \bullet d^2(x) = d(x) \tag{7}$$

where

$$d^2(x) = \{04\}x^2 + \{05\} \tag{8}$$

Note that two of the coefficients of the $d^2(x)$ are equal to $\{00\}$.

The *MixColumns* and *InvMixColumns* can be implemented using shared logic resources as shown in Fig. 6.

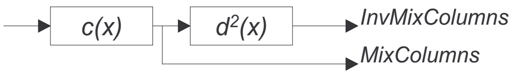

Fig. 6. Implementation of *MixColumns* and *InvMixColumns*

The multiplication by $\{04\}$ and $\{05\}$ lead to following equations:

$$b(x) \bullet \{04\} = b_5x^7 + b_4x^6 + (b_7 + b_3)x^5 + (b_7 + b_6 + b_2)x^4 +$$
$$+ (b_6 + b_1)x^3 + (b_7 + b_0)x^2 + (b_7 + b_6)x + b_6 \tag{9}$$

$$b(x) \bullet \{05\} = (b_7 + b_5)x^7 + (b_6 + b_4)x^6 + (b_7 + b_5 + b_3)x^5 + (b_7 + b_6 + b_4 + b_2)x^4 +$$
$$+ (b_6 + b_3 + b_1)x^3 + (b_7 + b_2 + b_0)x^2 + (b_7 + b_6 + b_1)x + (b_6 + b_0) \tag{10}$$

Their implementation appears area efficient since 4-input XOR gates are the widest gates involved in computations, and they get efficiently implemented in 4-input LUTs of the FPGA.

At the time this paper was written we learned that this technique was first discovered and proposed for software implementations by P. Barreto [5]. V. Fischer and F. Gramain were the first to apply it in hardware [1].

3.4 Encryption/Decryption Unit

Our circuit is capable of performing encryption and decryption. The AES encryption and decryption rounds substantially differ from the point of view of hardware implementations. One of the inconveniences arises from the fact that the *AddRoundKey* is executed after *MixColumns* in the case of encryption, and before *InvMixColumns* in the case of decryption. Therefore, a switching logic is required to select appropriate data paths, which affects the performance, as shown in Fig. 7.

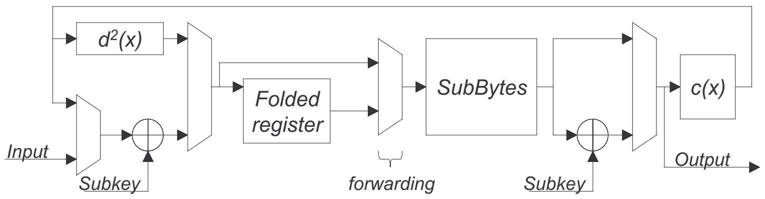

Fig. 7. Implementation of the encryption/decryption unit

It is possible to reorder the *InvMixColumns* and *AddRoundKey* and avoid some of the switching. In this case, the key schedule would need to perform additional *InvMixColumns* transformation on most of the subkeys. The *InvMix-Columns* requires much more area than the switching logic. Our implementation delivers sufficient performance with the switching logic in place, therefore we implemented the architecture shown in Fig. 7.

3.5 Implementation of the Key Schedule

The key schedule is typically implemented using one of the two methods: computing keys on-the-fly for every block of encrypted data, or precomputing them in advance and storing. The computation of keys on-the-fly has an obvious advantage of changing keys fast with low or no delay. This performance comes for a price of increased power consumption as the key schedule computes over and over again for each data block.

In the case of the AES it is easy to perform key schedule transformations in the forward direction, and this is the order the round keys are applied in the case of encryption. In the case of decryption round keys are applied in the reversed order. The key schedule could compute round keys in the backward direction, but it is possible only by starting from the last key, not the main key. Unfortunately, the last key can be obtained from the main key only by computing the entire key schedule in the forward direction first. For this reason, the key schedule computing keys on-the-fly completely looses its advantage when decryption is performed.

Our AES implementation is designed to perform encryption and decryption. Since we did not see any advantage in computing round keys on-the-fly, we selected to implement the key schedule that precomputes all round keys. The implementation of the key schedule is shown in Fig. 8. It computes 32-bits of the key material per clock cycle, therefore, full key schedule execution takes 44 clock cycles. The computed round keys are stored in a single Block RAM.

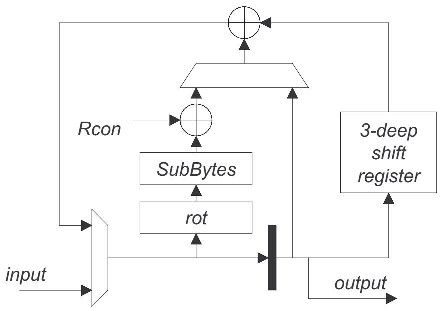

Fig. 8. Implementation of the key schedule

The key schedule uses *SubBytes* operation that is identical to the one used in the encryption circuit. Since key schedule does not work simultaneously with the encryption unit, it is possible to time share S-boxes between both circuits. This approach saves two Block RAMs at the expense of additional switching logic, and degraded performance. The performance is affected by the presence of the switching logic in the critical path, and by slightly more complicated floorplanning and routhing, as encryption/decryption and key schedule units are no longer separated. We implemented the switching logic using tri-state buffers in order to minimize its influence on the overall performance; however, this solution may not be the most desired for various reliability and testability related reasons. In the case when tri-state buffers are not allowed in the design, a multiplexer should be used for switching.

4 Targeted Device, Synthesis, and Implementation Results

The goal for this design was to create a low-cost implementation of AES in the FPGA targeted for real life applications. Much of the previous research targets state-of-the-art technologies forgetting that the individual cost of those devices ranges in hundreds of US dollars. We shifted our attention to older technologies and smaller devices. Xilinx Inc. produces two low-cost families of devices called Spartan II, and Spartan IIE. Pricing for Spartan II FPGAs starts from less than \$10 per unit [35].

Spartan II FPGAs are manufactured in 0.22μm CMOS process. Their architecture is derived from a bigger family of Virtex devices. Spartan IIE are based on a newer VirtexE family, and are manufactured in 0.18μm CMOS process. The smallest device from the Spartan IIE family was too large for our needs. The device we selected for our implementation is Spartan II XC2S30; second smallest in its family.

The synthesis of our design was done using Synplify Pro 7.2 from Synplicity. We set the constraints for target clock frequency to 60MHz, fanout guide to 100, and enabled resourse sharing. We performed synthesis for speed grades -5 and -6.

The mapping, placing and routing was done using Xilinx ISE 5.2i package. Mapper optimized circuit for area, and router worked with effort level 5.

The results are given in the Table 1. The maximum frequencies come from static timing analysis only. The performance is nearly equally affected by logic and routing. The routing vs. logic delays ratio in the critical path is 54/46. Better results could be demonstrated with manual floorplanning.

Table 1. Implementation results

Device	Area		Max. clock	Throughput
	CLB Slices	Block RAMs	frequency [MHz]	[Mbps]
XC2S30-5	222	3	50	139
XC2S30-6	222	3	60	166

5 Comparison with Other Designs

Despite our intensive search we encountered suprisingly few compact implementations of the AES algorithm in FPGAs. There exist commercial compact cores from Amphion [32] and Helion [33] companies. Both companies provide compact cores in encryption or decryption version only, and a 128-bit key schedule. We also encountered a JBits implementation by S. McMillan et al. [21]. Their implementation uses JBits to tailor the bitstream for particular key, and encryption or decryption operation. Therefore, encryption and decryption are never simultaneously present in the circuit, and the key schedule is not implemented in the hardware.

We also collected information about other existing architectures capable of encrypting or decrypting data in feedback modes of operation [15] [16] [17] [24] [26]. We did not take into account any implementations based on *T-boxes* as they give greater throughput at the expense of much larger area. The basic features of all the implementations are collected in Table 2, and their performance characteristics in Table 3.

Table 2. Basic features of compared architectures

	Device	Encryption	Decryption	Key Schedule 128	192	256
			0.22μm			
Our	**Spartan II-6**	●	●	●		
P. Chodowiec et al. [15]	Virtex-6	●	●	●	●	●
A. Dandalis et al. [24]	Virtex-6	●	●	●		
A.J. Elbirt et al. [16]	Virtex-6	●				
V. Fischer et al. [17]	FLEX 10KE-1	●	●			
	ACEX 1K-1	●	●			
K. Gaj et al. [26]	Virtex-6	●	●			
S. McMillan et al. [21]	Virtex	●				
			0.18μm			
Amphion CS5220XV [32]	VirtexE-8	●		●		
CS5230XV	VirtexE-8	●		●	●	●
Helion compact [33]	Spartan IIE-6	●		●		
fast	VirtexE-8	●	●	●		
V. Fischer et al. [17]	APEX 20KE-1	●	●			
			0.15μm			
Amphion CS5220XV[32]	Virtex2-5	●		●		
CS5230XV	Virtex2-5	●		●	●	●
Helion fast [33]	Virtex2-5	●	●	●		

Among compact architectures, our design is one of the smallest and offers richer functionality than cores from Amphion and Helion because it supports both encryption and decryption. Both commercial cores are faster than ours; however, they are implemented in a better, thus more expensive technology. The implementation by S. McMillan et al. is also very compact and fast; however, it benefits from the JBits application which is not likely to work in an embedded environment.

We notice large differences among results for basic architecture. The implementation by P. Chodowiec et al. offers the most complete functionality and has nearly identical size with the fast Helion core. The implementation by V. Fischer et al. on FLEX and APEX also have similar parameters, but do not include key schedule. The Amphion core CS5230XV is the smallest implementation in the basic architecture, but does not support decryption.

Relating the results for our compact implementation to the implementations in the basic architecture, we can see that the goal of reducing the required logic resources by 75% was achieved. Moreover, the throughput of our design is higher than the 25% of the best throughput reported for the basic architecture in the same technology.

Table 3. Performance of all compared cores

	Area		Throughput	clock cycles
	CLB Slices	Block RAMs	[Mbps]	per round
0.22μm				
Our	**222**	**3**	**166**	**4**
P. Chodowiec et al.	~1230	18	577	1
A. Dandalis et al.	5673	0	353	1
A.J. Elbirt et al.	3528	0	294.2	1
V. Fischer et al. FLEX	2530 LE[1]	24 EAB[2]	451	1
ACEX	2923 LE[1]	12 EAB[2]	212	1
K. Gaj et al.	2902	0	331.5	1
S. McMillan et al.	240	8	250	7
0.18μm				
Amphion CS5220XV	421	4	294	4
CS5230XV	573	10	1061	1
V. Fischer et al. APEX	2493 LE[1]	50 ESB[2]	612	1
Helion compact	392 LUT[1]	3	223	4
fast	2259 LUT[1]	18	1001	1
0.15μm				
Amphion CS5220XV	403	4	350	4
CS5230XV	573	10	1323	1
Helion fast	2259 LUT[1]	18	1408	1

[1] 2 LE $\approx$ 2 LUT $\approx$ 1 Slice [2] 1 EAB = 2 ESB = 1 BRAM

We intentionally did not provide Throughput/Area ratios for any of the compared designs as this measure can be very misleading when dedicated memories are present in the design.

6 Conclusions

In this paper the feasibility of creating a very compact, low-cost FPGA implementation of the AES was examined. The proposed folded architecture achieves good performance and occupies less area than previously reported designs. This compact design was developed by thorough examination of each of the components of the AES algorithm and matching them into the architecture of the FPGA.

The demonstrated implementation fits in a very inexpensive, off-the-shelf Xilinx Spartan II XC2S30 FPGA, which cost starts below $10 per unit. Only 50% of the logic resources available in this device were utilized, leaving enough area for additional glue logic. This implementation can encrypt and decrypt data streams up to 166 Mbps. The encryption speed, functionality, and cost make this solution perfectly practical in the world of embedded systems and wireless communication.

References

1. Fischer V. and Gramain F.: *Resource sharing in a Rijndael implementation based on a new MixColumn and InvMixColymn relation*, submitted to Electronic Letters, reference number: ELL 39 395, April 14, 2003
2. Järvinen K.U., Tommiska M.T., Skyttä J.O.: *A fully pipelined memoryless 17.8 Gbps AES-128 encryptor*, International Symposium on Field-Programmable Gate Arrays (FPGA 2003), Monterey, CA, 2003
3. Standaert F.X., Rouvroy G., Quisquater J.J., Legat J.D., *A methodology to implement block ciphers in reconfigurable hardware and its application to fast and compact AES RIJNDAEL*, International Symposium on Field-Programmable Gate Arrays (FPGA 2003), Monterey, CA, 2003
4. Verbauwhede I., Schaumont P., Kuo H.: *Design and performance testing of a 2.29-GB/s rijndael processor*, IEEE Journal of Solid-State Circuits, Volume: 38 Issue: 3, March 2003
5. Daemen J. and Rijmen V.: *The design of Rijndael: AES – The Advanced Encryption Standard*, Springer-Verlag, ISBN 3-540-42580-2, 2002
6. Lin T.F., Su C.P., Huang C.T., Wu C.W.: *A high-throughput low-cost AES cipher chip*, IEEE Asia-Pacific Conference on ASIC, 2002
7. Lutz A.K., Treichler J., Gürkaynak F.K., Kaeslin H., Basler G., Erni A., Reichmuth S., Rommens P., Oetiker S., Fichtner W., *2Gbit/s Hardware Realizations of RIJNDAEL and SERPENT: A Comparative Analysis*, Cryptographic Hardware and Embedded Systems (CHES 2002), San Francisco Bay, CA, 2002
8. Mayer U., Oelsner C., Kohler T.: *Evaluation of different rijndael implementations for high end servers*, IEEE International Symposium on Circuits and Systems (ISCAS 2002), 2002
9. Mitsuyama Y., Andales Z., Onoye T., Shirakawa I.: *Burst mode: a new acceleration mode for 128-bit block ciphers*, IEEE Custom Integrated Circuits Conference, 2002
10. Morioka S. and Satoh A., *An Optimized S-Box Circuit Architecture for Low Power AES Design*, Cryptographic Hardware and Embedded Systems (CHES 2002), San Francisco Bay, CA, 2002
11. Morioka S. and Satoh A.: *A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture*, IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2002
12. Ravi S., Raghunathan A., Potlapally N.: *Securing Wireless Data: System Architecture Challenges*, Symposium on System Synthesis, 2002
13. Wolkerstorfer J., Oswald E., Lamberger M.: *An ASIC Implementation of the AES SBoxes*, The Cryptographer's Track at the RSA Conference, San Jose, CA, 2002
14. Chodowiec P., Khuon P., Gaj K.: *Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining*, International Symposium on Field-Programmable Gate Arrays (FPGA 2001), Monterey, CA, 2001
15. Chodowiec P., Gaj K., Bellows P., Schott B.: *Experimental Testing of the Gigabit IPSec-Compliant Implementations of Rijndael and Triple DES Using SLAAC-1V FPGA Accelerator Board*, Information Security Conference (ISC 2001), Malaga, Spain, 2001
16. Elbirt A.J., Yip W., Chetwynd B., Paar C.: *An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 9 Issue: 4, August 2001
17. Fischer V. and Drutarovský M.: *Two Methods of Rijndael Implementation in Reconfigurable Hardware*, Cryptographic Hardware and Embedded Systems (CHES 2001), Paris, France, 2001

18. McLoone M. and McCanny J.V.: *High Performance Single-Chip FPGA Rijndael Algorithm Implementations*, Cryptographic Hardware and Embedded Systems (CHES 2001), Paris, France, 2001
19. McLoone M. and McCanny J.V.: *Single-Chip FPGA Implementation of the Advanced Encryption Standard Algorithm*, Field-Programmable Logic and Applications (FPL 2001), Belfast, Northern Ireland, UK, 2001
20. McLoone W., McCanny J.V.: *Rijndael FPGA implementation utilizing look-up tables*, IEEE Workshop on Signal Processing Systems, 2001
21. McMillan S. and Patterson C.: *JBits Implementations of the Advanced Encryption Standard (Rijndael)*, Field-Programmable Logic and Applications (FPL 2001), Belfast, Northern Ireland, UK, 2001
22. Rudra A., Dubey P.K., Jutla C.S., Kumar V., Rao J.R., Rohatgi P.: *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic*, Cryptographic Hardware and Embedded Systems (CHES 2001), Paris, France, 2001
23. Satoh A., Morioka S., Takano K., Munetoh S.: *A Compact Rijndael Hardware Architecture with S-Box Optimization*, Theory and Application of Cryptology and Information Security (ASIACRYPT 2001), Gold Coast, Australia, 2001
24. Dandalis A., Prasanna V.K., Rolim J.D.: *A Comparative Study of Performance of AES Final Candidates Using FPGAs*, Cryptographic Hardware and Embedded Systems Workshop (CHES 2000), Worcester, Massachusetts, 2000
25. Elbirt A.J., Yip W., Chetwynd B., Paar C.: *An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*, Third Advanced Encryption Standard (AES3) Candidate Conference, New York, 2000
26. Gaj K. and Chodowiec P.: *Comparison of the hardware performance of the AES candidates using reconfigurable hardware*, Third Advanced Encryption Standard (AES3) Candidate Conference, New York, 2000
27. Gaj K. and Chodowiec P.: *Hardware performance of the AES finalists-survey and analysis results*, Technical Report, George Mason University, 2000, available at `http://ece.gmu.edu/crypto/AES_survey.pdf`
28. Ichikawa T. and Matsui T.: *Hardware Evaluation of the AES Finalists*, Third Advanced Encryption Standard (AES3) Candidate Conference, New York, 2000
29. National Institute of Standards and Technology: *FIPS 197: Advanced Encryption Standard*, November 2001
30. Daemen J. and Rijmen V.: *AES Proposal: Rijndael*, `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf`
31. Rijmen V.: *Efficient Implementation of the Rijndael S-box*, available at: `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf`
32. Amphion: `http://www.amphion.com/`
33. Helion: `http://www.heliontech.com/`
34. Xilinx, Inc.: *Spartan II Data Sheet*, available at: `http://www.xilinx.com`
35. Xilinx, Inc.: *The New Spartan-II FPGA Family: Kiss Your ASIC Good-bye*, XCELL Journal, Q1, 2000, available at: `http://www.xilinx.com/xcell/xl35/xl35_5.pdf`