Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

# A new methodology to implement the AES algorithm using partial and dynamic reconfiguration

José M. Granado-Criado *, Miguel A. Vega-Rodríguez, Juan M. Sánchez-Pérez, Juan A. Gómez-Pulido

*Department Technologies of Computers and Communications, University of Extremadura, Spain*

A R T I C L E  I N F O

*Article history:*
Received 4 April 2008
Received in revised form
27 May 2009
Accepted 30 May 2009

*Keywords:*
AES
Cryptography
FPGA
Efficiency
Reconfigurable computing
Run-time reconfiguration

A B S T R A C T

Wireless networks are very widespread nowadays, so secure and fast cryptographic algorithms are needed. The most widely used security technology in wireless computer networks is WPA2, which employs the AES algorithm, a powerful and robust cryptographic algorithm. In order not to degrade the Quality of Service (QoS) of these networks, the encryption speed is very important, for which reason we have implemented the AES algorithm in an FPGA, taking advantage of the hardware characteristics and the software-like flexibility of these devices. In this paper, we propose our own methodology for doing an FPGA-based AES implementation. This methodology combines the use of three hardware languages (Handel-C, VHDL and JBits) with partial and dynamic reconfiguration, and a pipelined and parallel implementation. The same design methodology could be extended to other cryptographic algorithms. Thanks to all these improvements our pipelined and parallel implementation reaches a very high throughput (24.922 Gb/s) and the best efficiency (throughput/area ratio) of all the related works found in the literature (6.97 Mb/s per slice).

## 1. Introduction

Wireless networks are very widespread nowadays. These networks are very useful, because they can cover an office building, a University campus, or a home with a very simple installation. However, wireless networks suffer from the problem that a non-authorised person can access them very easily. For this reason, wireless networks use cryptographic algorithms to encrypt the information. One of the first algorithms used was RC4 in the WEP protocol, a very simple and fast algorithm, but with the problem that it is very easy to break. For this reason, the current technology, WPA2, uses AES, which is much more secure than RC4, though slower than RC4. This fact along with the increasing wireless throughput requires very efficient implementations of the AES algorithm.

In this paper, we have implemented an AES-128 algorithm using parallelism, pipelining, and partial and dynamic reconfiguration. The most complex element of this algorithm is the multiplication modulo the irreducible polynomial $m(x) = x^8+x^4+x^3+x+1$. We find one possible implementation of this element in [36], where the *xtime* function is implemented to do the multiplication. In our case, we have implemented this element simply calculating the XOR operations needed to make the multiplication. Another important element is the KeyExpansion phase. It is very usual to implement this phase and we can find several implementations of it, like [42], where a hierarchical simultaneous key generation (HSKG) methodology is used. In our case, we use dynamic and partial reconfiguration to modify the LUTs which contain the sub-keys, so we do not re-calculate each sub-key within the FPGA and we do not use FPGA resources to implement this method. In addition, we employ Handel-C and VHDL, specifically VHDL to implement the AES elements and Handel-C to implement the communication system and the pipelining registers.

Taking all this into account, we achieve very good values for throughput (24.922 Gb/s), area (only 3576 slices) and latency (210.5 ns). In fact, the efficiency (that is, throughput/area ratio) of our circuit is the best of all the related works found in the literature (we obtain an efficiency of 6.97 Mb/s per slice).

This paper is structured as follows. Section 2 reviews the different techniques and approaches used in all the FPGA-based AES implementations that we have found in the literature. In Section 3 we describe the AES algorithm. Then, Section 4 briefly presents the different hardware languages that have been used in this work (Handel-C, VHDL and JBits). In this paper, in order to do the FPGA-based AES implementation, we propose our own methodology combining these three languages with partial and dynamic reconfiguration, and a pipelined and parallel implementation. In this regard, we explain our methodology and how exactly we have implemented the algorithm in Sections 5 and 6. Section 7 presents and analyzes the final results, and we state our conclusions in the last section.

* Corresponding author. Tel.: +34 616 557 935; fax: +34 927 257 202.
*E-mail addresses:* granado@unex.es (J.M. Granado-Criado), mavega@unex.es (M.A. Vega-Rodríguez), sanperez@unex.es (J.M. Sánchez-Pérez). jangomez@unex.es (J.A. Gómez-Pulido).

0167-9260/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.vlsi.2009.05.003

## 2. Related work

This section makes a background review, detailing the different techniques and approaches used by other FPGA-based implementations. This allows distinguishing our work from others.

Table 1 exposes all the papers we have found in the literature about FPGA-based implementations of the AES algorithm. For every work we detail the reference, the publication year, and if this work uses Pi (Pipelining) and/or BRAM (including the exact number of BRAMs). In this table, column "SBox" indicates the implementation techniques of the SBox tables: Slices (implemented by means of a distributed memory, including the exact number of slices between brackets), BRAM (the SBoxes are implemented by BRAMs) and GF2$^4$ (implemented by using the GF($2^4$) instead of GF($2^8$)) . Similarly, "MCP" indicates the MixColumns Phase implementation technique used: *xtime* (implemented by means of the *xtime* function), LUTs (precalculated tables which store all the possible multiplying results), TBOX (combining the MixColumn and SubByte operations) and GF2$^4$ (implemented by using the GF($2^4$) instead of GF($2^8$)). In the same way, column "KEP" shows the KeyExpansion Phase implementation technique used: PDR (implemented by means of partial and dynamic reconfiguration), hierarchical simultaneous key generation, separated clock domain implementation (SCD), online key generation (OKG), GF2$^4$ (implemented by using the GF($2^4$) instead of GF($2^8$)), and A&S (in advance and stored in a register file). Finally, we also say the Hardware Description Language used in each work (VHDL or Handel-C–H-C).

From Table 1 we can conclude that our work is the only one that combines all the techniques shown in this table (except the use of BRAM, in fact, the half of the references does not use BRAM). We have to highlight that no other work uses the Handel-C language (or its combination with VHDL) or partial and dynamic reconfiguration (we use this important technique in order to implement the KeyExpansion phase).

## 3. The AES-128 algorithm

The Advanced Encryption Standard (AES) algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits [9]. This standard is based on the Rijndael algorithm [4], a symmetric block cipher. As the AES algorithm may be used with three different key lengths, these three different "flavors" are generally referred to as "AES-128", "AES-192", and "AES-256". In our case, we have used a key with a length of 128 bits; therefore, we are using the AES-128.

**Table 1**
Summary of techniques used in the FPGA-based implementations of AES.

| Ref. | Year | Techniques | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Pi | BRAM | SBox | MCP | KEP | VHDL | H-C |
| This work | 2008 | X | 0 | Slices (10,240) | *xtime* | PDR | X | X |
| [26] | 2001 | | 244 | ?? | LUTs | ?? | ?? | ?? |
| [34] | 2003 | X | 100 | BRAM | *xtime* | ?? | ?? | ?? |
| [17] | 2004 | X | 84 | GF2$^4$ | ?? | ?? | ?? | ?? |
| [42] | 2005 | X | 200 | BRAM | ?? | HSKG | ?? | ?? |
| [19] | 2005 | X | 4 | ?? | TBOX | SCD | X | |
| [31] | 2003 | X | 100 | ?? | ?? | OKG | X | |
| [43] | 2004 | X | 0 | Slices (12,037) | ?? | OKG | X | |
| [44] | 2004 | X | 0 | GF2$^4$ | *xtime* | OKG | ?? | ?? |
| [18] | 2003 | X | 0 | ?? | GF2$^4$ | GF2$^4$ | X | |
| [13] | 2001 | X | 80 | BRAM | ?? | ?? | X | |
| [14] | 2005 | X | 0 | Slices (7319) | ?? | OKG | X | |
| [8] | 2006 | X | 0 | GF2$^4$ | *xtime* | A&S | X | |

In this table, "??" means that those data were not found in the corresponding reference.

The AES algorithm is divided into four different phases, which are executed in a sequential way forming rounds. The encryption is achieved by passing the plaintext through an initial round, 9 equal rounds and a final round. In all of the phases of each round, the algorithm operates on a $4 \times 4$ array of bytes (called the State). In Fig. 1 we can see the structure of this algorithm.

Let us see every phase of the algorithm.

### 3.1. KeyExpansion phase

The AES algorithm takes the Master Key K, and performs a Key Expansion routine to generate a key schedule. The KeyExpansion generates a total of 11 sub-key arrays of 16 words of 8 bits, denoted $w_i$, taking into account that the first sub-key is the initial key. To calculate every $w_i$ (except $w_0$) the routine uses the previous $w_{i-1}$ and two tables, RCon and SBox. RCon[i] contains the values given by [$x^{i-1}$,{00},{00},{00}], with $x^{i-1}$ being powers of $x$ ($x$ is denoted as {02}) in the field GF($2^8$). On the other hand, SBox is a non-linear and invertible substitution table used to perform a one-by-one substitution of a byte value.

### 3.2. AddRoundKey phase

The AddRoundKey phase performs an operation on the State with one of the sub-keys. The operation is a simple XOR between each byte of the State and each byte of the sub-key.

### 3.3. SubByte phase

The SubByte transformation is a non-linear byte substitution that operates independently on each byte of the State using the SBox table.

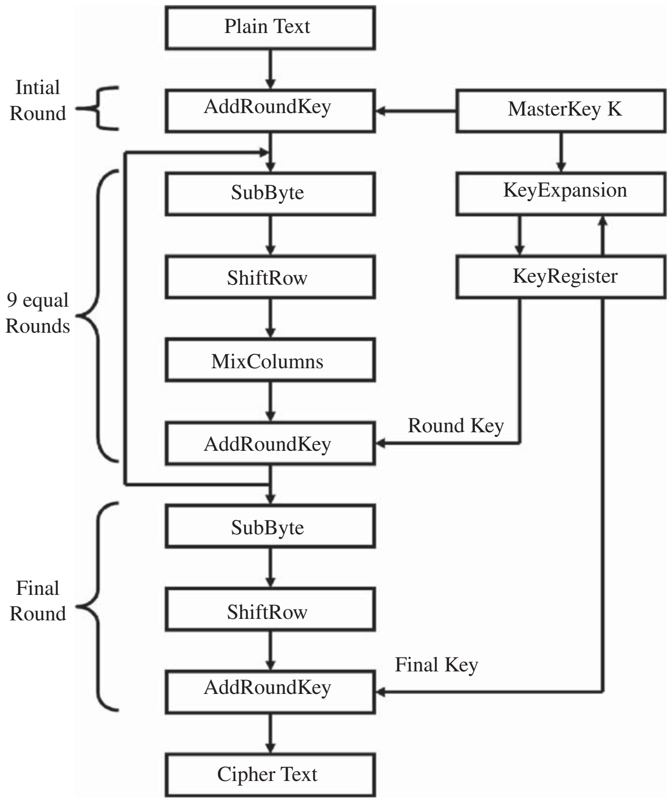

**Fig. 1.** Description of the AES cryptographic algorithm.

### 3.4. ShiftRow phase

In the ShiftRow transformation, the bytes in the last three rows of the State are cyclically shifted over 1, 2 and 3 bytes, respectively. The first row is not shifted.

### 3.5. MixColumns phase

The MixColumns transformation operates on the State column by column, treating each column as a four-term polynomial. The columns are considered as polynomials over GF($2^8$) and multiplied by a fixed polynomial $a(x)$ modulo $x^4$+1, given by

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

This can be written as a matrix multiplication as follows:

$$S'(x)=A(x)\otimes S(x)\begin{bmatrix} s'_{0,c}\\ s'_{1,c}\\ s'_{2,c}\\ s'_{3,c}\end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01\\ 01 & 02 & 03 & 01\\ 01 & 01 & 02 & 03\\ 03 & 01 & 01 & 02\end{bmatrix}\begin{bmatrix} s_{0,c}\\ s_{1,c}\\ s_{2,c}\\ s_{3,c}\end{bmatrix} \quad \text{for } 0\le c<4 \tag{2}$$

As a result of this multiplication, the four bytes in a column are replaced as follows:

$$\begin{aligned} S'_{0,c}&=(\{02\}\cdot S_{0,c})\oplus(\{03\}\cdot S_{1,c})\oplus S_{2,c}\oplus S_{3,c}\\ S'_{1,c}&=S_{0,c}\oplus(\{02\}\cdot S_{1,c})\oplus(\{03\}\cdot S_{2,c})\oplus S_{3,c}\\ S'_{2,c}&=S_{0,c}\oplus S_{1,c}\oplus(\{02\}\cdot S_{2,c})\oplus(\{03\}\cdot S_{3,c})\\ S'_{3,c}&=(\{03\}\cdot S_{0,c})\oplus S_{1,c}\oplus S_{2,c}\oplus(\{02\}\cdot S_{3,c})\end{aligned} \tag{3}$$

where $\oplus$ is the XOR operation and the $\cdot$ is a multiplication modulo the irreducible polynomial $m(x)=x^8$+$x^4$+$x^3$+$x$+1. Fig. 2 shows the implementation of the function $B=xtime(A)$, which will be used to make the multiplications of a number by 2 modulo $m(x)$. So, we will only have binary operations as follows:

$$\begin{aligned} \{02\}\cdot S'_{x,c}&=xtime(S'_{x,c})\\ \{03\}\cdot S'_{x,c}&=xtime(S'_{x,c})\oplus S'_{x,c}\end{aligned} \tag{4}$$

See [9] for a complete mathematical explanation of the AES algorithm.

## 4. Hardware languages used

This section describes the three hardware languages (Handel-C, VHDL and JBits) used in order to implement our FPGA-based AES circuit.

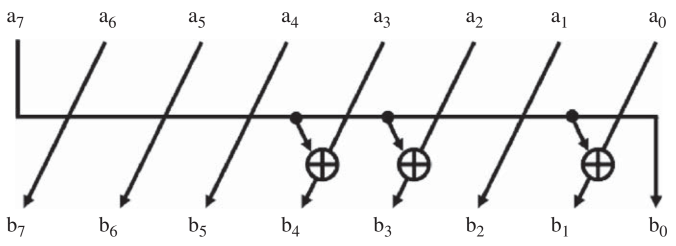

**Fig. 2.** The *xtime* function.

### 4.1. Handel-C

Handel-C is a high-level Hardware Description Language designed by Celoxica [2]. This language makes the hardware implementation of a circuit easier by using syntax based on conventional ANSI C, with the addition of statements that exploit the hardware parallelism easily. For example, if we want to implement two instructions in a parallel way we only need to include those instructions in a *par* statement.

Besides, Handel-C provides a set of libraries, which allows us to implement the Host-FPGA communication in a more direct way than if we only use VHDL. In conclusion, all these advantages reduce the design time of a circuit.

### 4.2. VHDL

Although Handel-C is a very useful language, it has some limitations. For example, in Handel-C it is not possible to define place-and-route constraints (needed to make the partial and dynamic reconfiguration). Therefore, we have used VHDL [30] in order to design the hardware elements, which will be run-time reconfigured.

Some important features of VHDL are: it is one of the most used HDL, it has a large and flexible syntax which allows us to describe a circuit by using different abstraction levels (structural, data flow, or hardware behaviour), it is possible to indicate low-level constraints (like place-and-route constraints), etc. All these features have motivated us to use VHDL.

### 4.3. JBits

JBits [35] is a set of Java classes which provide an Application Program Interface (API) into the Xilinx Virtex-II FPGA family bitstream. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. This provides the capability of designing, modifying and dynamically reconfiguring circuits in Xilinx Virtex-II series FPGA devices. The programming model used by JBits is a two-dimensional array of configurable logic blocks (CLBs). Each CLB is referenced by a row and column, and all configurable resources in the selected CLB may be set or probed. Additionally, control of all routing resources adjacent to the selected CLB is made available. Because the code is written in Java, compilation times are very fast, and because control is at the CLB level, bitstreams can typically be modified or generated very quickly.

We have used this API to perform the partial and dynamic reconfiguration of our circuit. Specifically, when the AES key is modified by the user, the sub-keys are calculated by software (JBits) and the FPGA (the LUTs storing the AES sub-keys) is run-time reconfigured by means of JBits.

## 5. Methodology

To implement the algorithm we have employed two different hardware description languages.

On the one hand, we have used VHDL [30], a Hardware Description Language that allows us to implement the AES components, including those which will be reconfigured in run-time by means of JBits [35] (a Java library used to perform this task). In order to do that, we have to locate the components manually, and then synthesize them.

On the other hand, the final implementation of AES has been done by means of Handel-C [2] (the other language used). This language is very close to the classical programmer (it is similar to

C language), and it allows us to reduce the final development time, because with this language it is very easy to implement and manage several elements of our design, such as pipelining registers, memories and the host–FPGA communication system.

Additionally, as we use two different languages, we have to join the two codes. To do that, we synthesize the VHDL components by means of XST tool Xilinx ISE 8.1i, and we reference them in Handel-C by means of interfaces.

Besides the languages used, we have followed a pipelined and parallel methodology.

In the pipelining, all the phases of the algorithm are executed in a pipeline way, that is, when the first phase finishes with the first block to be encrypted, the second block goes into that phase while the first block goes into the next operation.

Finally, we have used parallelism as much as possible, calculating, for example, the result of all bytes of the *state* in a parallel way in all phases of the AES algorithm.

Later we will see the pipelining and the parallelism in detail.

### 5.1. Performing the partial and dynamic reconfiguration

We have used JBits [35], which is a Java library designed by Xilinx, and the Xilinx constraints [39] to perform the dynamic and partial reconfiguration. We use partial and dynamic reconfiguration to calculate the sub-keys, that is, we implement the KeyExpansion phase using this technique. We do not use partial and dynamic reconfiguration in other phases/operations because AES is a very simple algorithm and it does not have complex operations (in our implementation, all of them are binary operations). In Section 6.2 we will see the advantages of employing partial and dynamic reconfiguration in the KeyExpansion phase.

Let us see how to perform the dynamic and partial reconfiguration necessary in the LUTs used to store the sub-keys.

#### 5.1.1. Synthesizing the LUTs

In order to use VHDL components in a Handel-C design, we have to synthesize those components, but before the synthesis can be made, we have to constrain the elements into the FPGA. Placement constraints are necessary because we need to know the exact location of these elements when we have to reconfigure them in run-time. In fact, all the constraints we have used in the design are on logic placement. No routing constraints are needed.

To do this, we have used the following constraints within the VHDL code:

- LUT_MAP constraint: This constraint indicates that an element must be synthesized using LUTs. In the AES algorithm, we use this constraint with each LUT employed in the storage of the sub-keys.
- LOC constraint: This constraint allows us to place a LUT in a specific slice of the FPGA.
- BEL constraint: This constraint will allow us to indicate whether the LUT F or LUT G is used (let us remember that the Virtex-II slices have 2 LUTs, LUT F and LUT G).

With all these constraints, we have defined a LUT placed in a particular LUT into a specific slice. If we repeat this process 128 times, we will define a 128-bit LUT.

When we have our 128 constrained LUTs, we have to synthesize all of them together into a 128-bit LUT by means of the Xilinx XST utility (we can do this in the Xilinx ISE Project Navigator). We will repeat this process for each 128-bit LUT in our design (the AES-128 has 11 sub-keys, that is, we must implement 11 128-bit LUTs). It is important to highlight that each LUT in our design has a different (LOC, BEL) value. If the (LOC, BEL) pair of two different LUTs were equal, we would be placing two LUTs in the same place and the final implementation would not work correctly (one of them would be placed randomly).

#### 5.1.2. Important details of the run-time reconfiguration

As we have said, we have implemented the KeyExpansion phase by means of partial and dynamic reconfiguration. This section describes how the reconfiguration is done.

The JBits functions *read* and *writePartial* will allow us to read and write the corresponding bitstream. Once we have located the LUTs which will store the sub-keys, we use JBits [35] to modify the LUTs contents in run-time. The first step is to generate all the sub-keys, implementing the KeyExpansion algorithm in Java. Next, we modify the corresponding LUT with each bit of the corresponding sub-key by means of the *setCLBBits* JBits function [35]. In order to perform the reconfiguration, we have to take into account that each sub-key will be stored in a 64-slice column, using both LUTs of each slice.

Finally, it is important to stand out that the *LOC* constraint makes reference to a concrete slice but the *setCLBBits* function references a concrete CLB and the 4 slices of this CLB in the form of a $2\times 2$ matrix.

### 5.2. The union between VHDL and Handel-C

We have mixed VHDL and Handel-C in our design. This mix is made in two phases: a first step in which all the VHDL components must be synthesized, and a second step in which these components will be used in Handel-C by means of interfaces. The first step has been explained previously (in Section 5.1.1).

In the second step we will employ the synthesized VHDL elements in our Handel-C code by means of interfaces. This utilization is made in two parts:

1. *Interface definition*: In this first phase the interface of all elements must be set up. Furthermore, we will establish the inputs to and the outputs from the interfaces. The name of each element's interface must be the same as the name of that element in the synthesized VHDL entity.
2. *Interface access*: After the interface is defined, we can access it by means of its instance name. It is important to emphasize that two instances of the same interface are different and they will be implemented separately. In order to access the output data of an instance we have only to write the name of the instance followed by the output port, separated by a dot.

## 6. The AES implementation

The final implementation was made in a Xilinx Virtex-2 6000 FPGA [40] included in a Celoxica [3] ADMXRC2 board. We used this FPGA because it has a very large number of resources (to be exact, a total of 33,792 slices), and it allows us to perform partial and dynamic reconfiguration.

AES does not have complex operations; however, it has a high memory cost. If we want to do the transformation of all bytes of the State in the SubByte phase in a parallel way, we must implement 16 SBox per phase. In addition, if we want to implement a pipelined version of the algorithm, we must use 10 SubByte phases, each of which has 16 SBox tables. Furthermore, as we want to implement a pipelined version, we must define 11, $4\times 4$ 1-byte sub-keys, because we have 11 AddRoundKey phases. All these storage elements give us a total of 329,088 bits ( $=10$ SubBytePhases$*$16 SBoxTablesPerSubBytePhase$*$256

ElementsPerSBox∗8 BitsPerElement+11 AddRoundKeyPhases∗16 ElementsPerAddRoundKeyPhase∗8 BitsPerElement). Moreover, we also have 41,128-bit pipelining registers (Fig. 4).

### 6.1. *MixColumns phase implementation*

As we have said, the AES algorithm has no complex phases and the implementation of all of them is very simple. However, the MixColumns phase implementation deserves special consideration.

In Section 3, we saw a brief mathematical description of this phase and now we will see how to implement it. To explain the implementation we will take the calculation of the element $S'_{0,0}$ of the $S'(x)$ matrix (Eq. (2)). The equation to solve this element is as follows:

$$S'_{0,0} = (\{02\} \cdot S_{0,0}) \oplus (\{03\} \cdot S_{1,0}) \oplus S_{2,0} \oplus S_{3,0} \tag{5}$$

Let us remember that the $\cdot$ operation is done by means of the *xtime* function (Eq. (4)). So, Eq. (5) changes to

$$S'_{0,0} = xtime(S_{0,0}) \oplus (xtime(S_{1,0}) \oplus S_{1,0}) \oplus S_{2,0} \oplus S_{3,0} \tag{6}$$

And, finally, taking into account the representation of the *xtime* function in Fig. 2, we will obtain the final result described in Table 2 (the result of bit *n* will be the XOR among the four columns of row *n*).

In conclusion, in order to implement the MixColumns phase we only need XOR gates.

### 6.2. *The KeyExpansion phase*

If we look at Fig. 1, we can see that the KeyExpansion phase calculates the sub-keys which will be used by the different AddRoundKey phases. However, we do not implement this method; instead we will use partial reconfiguration. We reconfigure, by means of JBits, the LUTs which will be used in the AddRoundKey phases, that is, the LUTs which store the sub-keys (see Section 5.1.2).

But, why do we use this technique? It is easy to think that we do not need to use this technique since, at first glance, we do not achieve a clear improvement of the performance (we do not have complex operations), but this question is automatically answered if we see the data described in Table 3. In this table we can see the result of implementing the AES algorithm with and "without" (using partial and dynamic reconfiguration) KeyExpansion phase. "Without KeyExpansion" means that we replace the hardware used in the KeyExpansion by the use of run-time reconfiguration (in both cases, the storage hardware of the sub-keys remains unaltered).

**Table 2**
Calculation of the state's $S'_{0,0}$ byte in the MixColumns phase.

| $S'_{0,0}$ bit | Operations to calculate the resulting bit | | | |
|---|---|---|---|---|
| | *xtime* $(S_{0,0})$ | *xtime*$(S_{1,0}) \oplus S_{1,0}$ | $S_{2,0}$ | $S_{3,0}$ |
| 7 | $S_{0,0}[6]$ | $S_{1,0}[7] \oplus S_{1,0}[6]$ | $S_{2,0}[7]$ | $S_{3,0}[7]$ |
| 6 | $S_{0,0}[5]$ | $S_{1,0}[6] \oplus S_{1,0}[5]$ | $S_{2,0}[6]$ | $S_{3,0}[6]$ |
| 5 | $S_{0,0}[4]$ | $S_{1,0}[5] \oplus S_{1,0}[4]$ | $S_{2,0}[5]$ | $S_{3,0}[5]$ |
| 4 | $S_{0,0}[7] \oplus S_{0,0}[3]$ | $S_{1,0}[7] \oplus S_{1,0}[4] \oplus S_{1,0}[3]$ | $S_{2,0}[4]$ | $S_{3,0}[4]$ |
| 3 | $S_{0,0}[7] \oplus S_{0,0}[2]$ | $S_{1,0}[7] \oplus S_{1,0}[3] \oplus S_{1,0}[2]$ | $S_{2,0}[3]$ | $S_{3,0}[3]$ |
| 2 | $S_{0,0}[1]$ | $S_{1,0}[2] \oplus S_{1,0}[1]$ | $S_{2,0}[2]$ | $S_{3,0}[2]$ |
| 1 | $S_{0,0}[7] \oplus S_{0,0}[0]$ | $S_{1,0}[7] \oplus S_{1,0}[1] \oplus S_{1,0}[0]$ | $S_{2,0}[1]$ | $S_{3,0}[1]$ |
| 0 | $S_{0,0}[7]$ | $S_{1,0}[7] \oplus S_{1,0}[0]$ | $S_{2,0}[0]$ | $S_{3,0}[0]$ |

On the one hand, as we can observe, the space occupied is reduced by some 50.44%, or in concrete terms by 3639 slices. This reduction is due to the elimination of several elements, that is, 16 SBox tables and the RCon table [9], and also the slices associated with the calculation of every sub-key.

On the other hand, as we can see in Table 3, the clock cycle passes from 14.657 to 5.136 ns, resulting in an important improvement in the throughput (from 8.733 to 24.922 Gb/s). This clock cycle improvement is easy to explain if we recall that the KeyExpansion phase is strictly sequential (the complete algorithm of this phase can be seen in the AES official description [9]): to calculate one column of one sub-key we need the previous column, and to calculate one sub-key we need the previous sub-key, as we can see in Fig. 3. Fig. 3(a) shows the block diagram which calculates the first sub-key column (the other columns calculation is simpler). Observing Fig. 3(a), we can deduce, and Table 3 confirms, that the clock cycle of this column calculation module is much higher than the clock cycle of the slowest AES phase. Fig. 3(b) shows how to obtain the four columns of the corresponding sub-key at the same time that the AES loop is running. In this figure, the block "Column 1" internally includes the diagram of Fig. 3(a) and the blocks "Column i" implement the calculation of the corresponding column. The first round does not need to calculate a sub-key (the first sub-key is the original key) and, the final round is similar to Fig. 3(b), but excluding the MixColumn phase.

### 6.3. *The AES pipelining*

To implement the pipelining of the AES algorithm, we perform a phase level pipelining since in the AES algorithm all the operations of one phase (remember that in the AES algorithm a round has several phases, see Fig. 1) are done at the same time. This pipelining can be seen in Fig. 4.

## 7. Results

Xilinx ISE 8.1i was used for the synthesis, place-and-route, and timing analysis. After obtaining the *.BIT* file, our implementation uses a total of 3576 slices, only 11% of the FPGA resources. Furthermore, thanks to our pipelined implementation, we reach a clock cycle of 5.136 ns (194.7 MHz of operating frequency). From these base statistics we calculated the resultant maximum throughput by using Eq. (7). Please remember that 128 is the block size

$$\text{Throughput} = \frac{128 * \text{Blocks}_{\text{per}}\text{Cycle}}{\text{Clock}_{\text{Period}}} \tag{7}$$

Finally, the efficiency of an implementation can be obtained by analyzing the metrics as follows:

$$\text{Efficiency} = \frac{\text{Throughput}}{\text{Number}_{\text{of}}\text{Slices}} \tag{8}$$

As we can see, this metrics divides the obtained throughput into the area occupied by that circuit, giving a value in Mb/s per

**Table 3**
Comparison between different implementations of the KeyExpansion phase.

| Implementation | Clock cycle (ns) | Throughput (Gb/s) | Area (Slices) |
|---|---|---|---|
| With KeyExpansion | 14.657 | 8.733 | 7215 |
| "Without" KeyExpansion (using partial and dynamic reconfiguration) | 5.136 | 24.922 | 3576 |

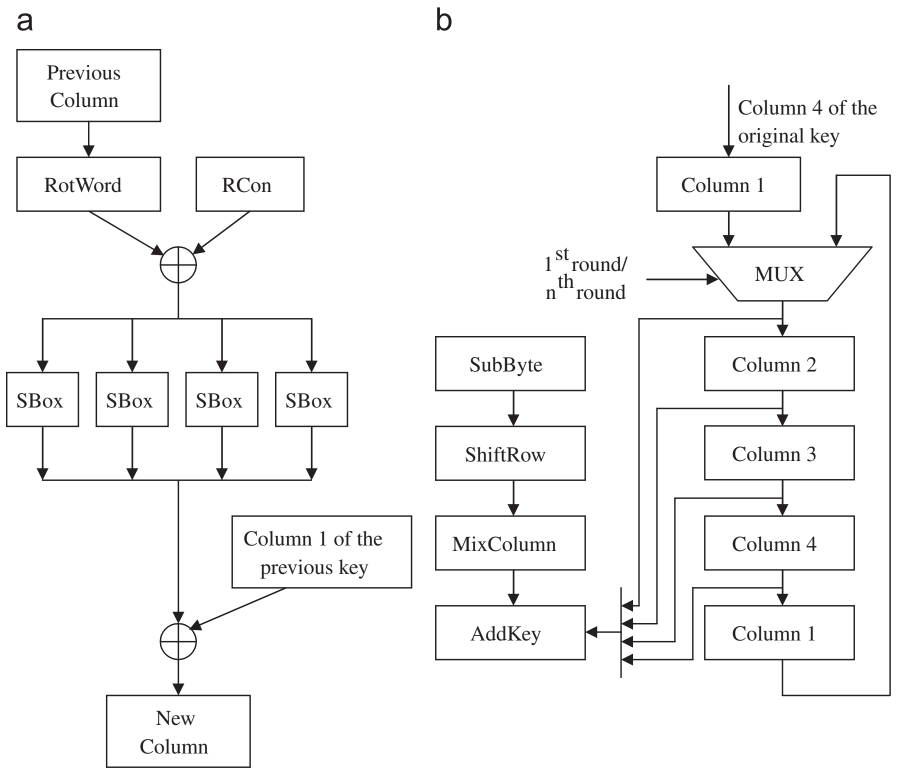

**Fig. 3.** (a) Block diagram of the first column calculation of the KeyExpansion phase and (b) block diagram of the AES loop together with the sub-key calculation.

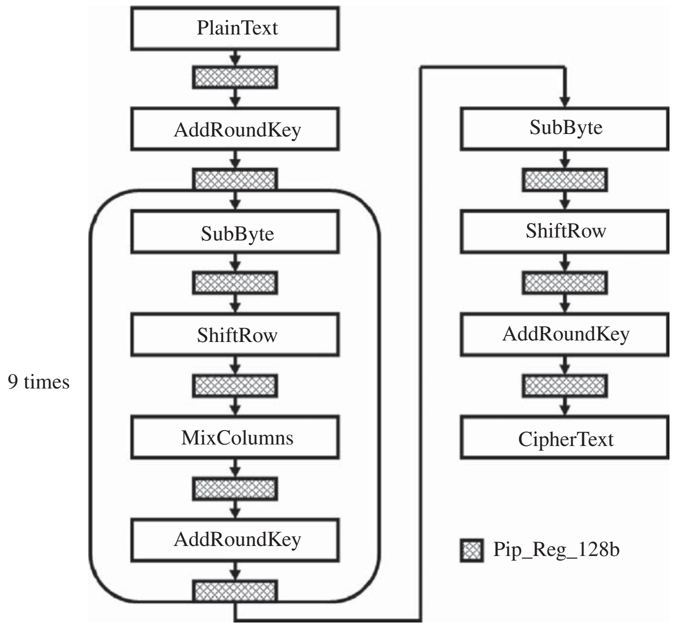

**Fig. 4.** The phase level pipelining of the AES algorithm.

slice. In fact, sometimes this measure is known as throughput per slice (TPS) [6,7]. Note that the efficiency metrics behaves inversely to the classical time–area (TA) product.

In our case, as we encrypt a data block per cycle, we obtain a throughput of 24.922 Gb/s and an efficiency of 6.97 Mb/s per slice. Table 4 compares our result with those obtained by 14 other FPGA-based AES implementations. It has been impossible to find other FPGA-based implementations using exactly the same type of device. For this reason, we have included all the FPGA-based implementations with good results that we have found, and we use the efficiency as the main comparison parameter. At present, efficiency is the most suitable and reliable basis for making a fair comparison of two FPGA-based implementations of a cryptographic algorithm, because it is focused not solely on a high throughput, but additionally on the fact that the hardware resources required to achieve this throughput are also a critical parameter. As an example, our circuit occupies only 11% of our FPGA. Therefore, we could replicate our circuit several times in the FPGA, enormously increasing its throughput. For example, if we replicate our circuit three times, we will encrypt three data blocks per cycle, and following Eq. (7), our throughput will be 74.77 Gb/s (greatly surpassing any of the throughputs shown in Table 4). However, our efficiency will continue to be around 6.97 Mb/s per slice, because our area (number of slices) will be also multiplied by 3. In conclusion, efficiency is the most invariant and reliable measurement for making comparisons between different implementations.

Furthermore, it is important to emphasize that, in column “# of BRAMs” of Table 4 (number of BRAMs which are used in the SBox storage), in order to make the comparisons we have taken into account that a dual-port 256 $\times$ 8 BlockRAM (BRAM), that is, the BRAM configuration used to implement the SBox tables, can be replaced by a distributed memory composed of 256 LUTs (128 slices) [32]. In conclusion, 10,240 slices (number of slices that we use to implement the SBox tables, see Table 1) are equivalent to 80 BRAMs, 12,037 slices (which are used to implement the SBox tables in [43], see Table 1) are equivalent to 95 BRAMs and 7319 slices (which are used to implement the SBox tables in [14], see Table 1) are equivalent to 58 BRAMs. At this point, it is important to emphasize that, although all the referenced FPGAs have BRAMs with bigger sizes than one SBox table (2 kb), one BRAM can only store two SBox tables, due to the number of access that a dual-port BRAM admits at the same time.

Finally, as we have already said, in Table 4 we only compare our implementation with the best AES implementations found in the

**Table 4**
Comparative results table of the different FPGA-based AES implementations.

| Device | Efficiency (Mb/s per Slice) | Throughput (Gb/s) | Area (Slices) | # of BRAMs | BRAM Size (kb) | Latency (ns) | Ref. |
|---|---|---|---|---|---|---|---|
| XC2V6000-6 | **6.97** | 24.92 | 3576 | 80 | 18 | 210.5 | This work |
| XCV812E | **6.01** | 12.02 | 2000 | 244 | 4 | 106 | [26] |
| XC2V4000 | **4.81** | 23.57 | 4901 | 95 | 18 | 162.9 | [43] |
| XCV3200E-8 | **4.23** | 11.78 | 2784 | 100 | 4 | ?? | [34] |
| XC2VP20-7 | **4.16** | 21.54 | 5177 | 84 | 18 | 292.8 | [17] |
| XC2VP70-7 | **3.84** | 29.77 | 7761 | 200 | 18 | 253.8 | [42] |
| XCV3200E-8 | **3.70** | 18.56 | 5016 | 100 | 4 | 496 | [34] |
| XCV2000E-8 | **3.49** | 20.30 | 5810 | 100 | 4 | 323 | [31] |
| XC2V6000-6 | **2.83** | 49.40 | 17,479 | –; | 18 | 158 | [8] |
| XCV2000E-8 | **2.52** | 23.65 | 9374 | 58 | 4 | 379 | [14] |
| XC3S2000-5 | **2.48** | 25.11 | 10,106 | 58 | 18 | 357 | [14] |
| XCV1000E-8 | **1.96** | 21.56 | 11,022 | –; | 4 | 422 | [44] |
| XCV1000E-8 | **1.66** | 17.80 | 10,750 | –; | 4 | 318 | [18] |
| XCV812E-8 | **1.27** | 11.97 | 9406 | –; | 4 | 332 | [44] |
| XCV1000-6 | **0.97** | 12.20 | 12,600 | 80 | 18 | 1768.8 | [13] |

literature. Other FPGA-based implementations with lesser throughput have not been included, for example: [31] (8.90 Gb/s), [1] (7.68 Gb/s), [25] (7 Gb/s), [27] (6.95 Gb/s), [33] (3.65 Gb/s), [29] (2.96 Gb/s), [6] (1.94 Gb/s), [36] (1.94 Gb/s), [19] (1.91 Gb/s), [7] (1.88 Gb/s), [37] (1.75 Gb/s), [30] (1.60 Gb/s), [36] (1.60 Gb/s), [7] (0.98 Gb/s), [28] (0.90 Gb/s), [37] (0.69 Gb/s), [19] (0.39 Gb/s) or [5] (0.35 Gb/s).

In conclusion, our results achieved for the throughput, area (number of slices used), number of BRAMs, and latency are very good. We always obtain close to the best result for these four parameters. Moreover, if we compare our result with all the ones displayed in Table 4, we can conclude that we reach the best efficiency, surpassing the results obtained by the other authors.

We obtain this very good efficiency because we combine a very high throughput with a very low occupation (that is, we reach a good balance between the two). Another important advantage of this low occupation is that we could implement our high-throughput circuit in very small and cheap FPGAs.

Table 5 shows a different comparison. In this table, we can observe several implementations of AES algorithm based on graphic processor units (GPUs), digital signal processors (DSPs) and application-specific integrated circuits (ASICs). We have included all works we have found in the literature but, as we can conclude, there are very few works about implementing AES algorithm using GPUs and DSPs devices. Anyway, Table 5 shows that the DSP-based implementation is not very good, reaching a low throughput. On the other hand, GPUs could be better to implement AES as we can see in [24], where a throughput of 8.28 Gb/s is reached. However, if we compare this datum with the throughput achieved by all the FPGA-based implementations shown in Table 4 (taking into account the differences between FPGAs and GPUs), we can conclude that FPGAs are more suitable for implementing the AES cryptographic algorithm than GPUs. In the same way, we can observe that ASIC-based implementations achieve good results but, equally, these works do not reach the results obtained by FPGA-based implementations. We would like to note that it was not possible to include other parameters (like area or power) in Table 5, because these data have not been found or appear in a no comparable way (with a different platform) in the papers referred in Table 5 (this table includes very different platforms).

Finally, another interesting datum is the reconfiguration time. This includes the time needed to rewrite the configuration file (around 141 ms) and the time of the FPGA configuration (4.65 ms, because our implementation needs 3576 slices, where one CLB includes four slices, and the time to configure one CLB for Virtex-II FPGAs is 5 μs [40,41]). This time may seem too high, but the

**Table 5**
GPUs, DSPs and ASICs implementations of the AES cryptographic algorithm.

| Device | Throughput (Gb/s) | Reference |
|---|---|---|
| XC2V6000-6 FPGA | 24.92 | This work |
| NVIDIA GeForce 8800GTX GPU | 8.28 | [24] |
| NVIDIA GeForce 7900GT GPU | 0.87 | [16] |
| TMS320C6x DSP | 0.14 | [38] |
| 0.35 μm CMOS ASIC | 2.38 | [21] |
| 0.6 μm CMOS ASIC (2 × 128 datapaths) | 2.16 | [22] |
| 0.25 μm CMOS ASIC | 2.12 | [15] |
| 0.35 μm CMOS ASIC | 1.95 | [10] |
| 0.18 μm CMOS ASIC | 1.82 | [23] |
| 0.35 μm CMOS ASIC | 1.33 | [20] |
| 0.25 μm CMOS ASIC | 1.15 | [31] |

reconfiguration is done when the user introduces the key and so he/she does not realize about this time (when the user push the “configure” button, the reconfiguration is already done). On the other hand, if every plaintext block comes with a different key, we have to do an FPGA reconfiguration every block (that is, every 128 bits). Clearly, this is an extreme case. Our FPGA-based implementation is focused on using the same key with many different blocks (as other authors do [8]). This is, for example, the typical situation in a wireless access point.

## 8. Conclusions

In this work, we have presented an implementation of the AES cryptographic algorithm using partial and dynamic reconfiguration. To implement this algorithm, we have employed two different languages, namely Handel-C and VHDL, and we have combined the two to achieve a very high-throughput implementation.

The implementation is pipelined and parallel, and we use JBits to reconfigure, in run-time, the elements involved with the keys, that is, the LUTs which store them. In summary, we have proposed our own methodology combining Handel-C, VHDL, JBits, partial and dynamic reconfiguration, and a pipelined and parallel implementation. This methodology is different to all the other proposals found in the literature (some proposals employ some of the characteristics we mix in this work, but there are not papers that employ our methodology completely). Furthermore, this same design methodology could also be extended to other cryptographic algorithms.

The results achieved for the throughput, area (number of slices used) and latency are very good. In fact, when we compare our result with more than 30 FPGA-based AES implementations (developed by other authors), we obtain the best efficiency (throughput/area), this parameter being the most reliable one for purposes of comparison.

Finally, the speed of the new 802.3 Ethernet standards, like 802.3ae (10 Gb/s) [11] or 802.3ap (4-lane 10 Gb/s) [12], makes necessary a high throughput in order to avoid bottlenecks. In conclusion, the use of FPGAs and partial and dynamic reconfiguration is a very good idea for implementing cryptographic algorithms, in particular AES, because we can obtain efficiency of around 7 Mb/s per slice. This efficiency (with a throughput of around 25 Gb/s) is in accordance with the new communication standards for wired and wireless networks.

## Acknowledgements

This work was partially funded by the Spanish Ministry of Education and Science and FEDER under contracts TIN2005-08818-C04-03 (the OPLINK project) and TIN2008-06491-C04-04/TIN (the MSTAR project).

## References

[1] M. Alam, W. Badawy, G. Jullien, A novel pipelined threads architecture for AES encryption algorithm, in: The IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2002, pp. 296–302.
[2] Celoxica: "Handel-C Language Reference Manual, version 3.1", 2005.
[3] Celoxica, 〈http://www.celoxica.com〉, 2008.
[4] J. Daemen, V. Rijmen, The block cipher Rijndael, Smart Card Research and Applications (2000) 288–296.
[5] A. Dandalis, V.K. Prasanna, J.D.P. Rolim, A comparative study of performance of AES candidates using FPGAs, in: The Third Advanced Encryption Standard (AES3) Candidate Conference, 2000, pp. 125–134.
[6] A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists, in: The Third Advanced Encryption Standard (AES3) Candidate Conference, 2000, pp. 13–27.
[7] 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 9 (4) (2001) 545–557.
[8] M. Fayed, M.W. El-Kharashi, F. Gebali, A high-speed, fully-pipelined VLSI architecture for real-time AES, in: 8th International Conference on Information & Communications Technology (ICICT), 2006, pp. 1–2.
[9] Federal Information Processing Standards Publication 197 (FIPS 197), 〈http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf〉, 2001.
[10] T. Ichikawa, T. Kasuya, M. Matsui, Hardware evaluation of the AES finalists, in: 3rd AES Candidate Conference, 2000, pp. 279–285.
[11] IEEE Std 802.3ae, 〈http://www.ieee802.org/3/ae/index.html〉, 2002.
[12] IEEE Std 802.3ap, 〈http://www.ieee802.org/3/ap/index.html〉, 2007.
[13] K. Gaj, P. Chodowiec, Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using field programmable gate arrays, CT-RSA (2001) 84–99.
[14] T. Good, M. Benaissa, AES on FPGA from the fastest to the smallest, in: 7th Cryptographic Hardware and Embedded Systems (CHES), 2005, pp. 427–440.
[15] F.K. Gürkaynak, A. Burg, D. Gasser, F. Hug, N. Felber, H. Kaeslin, W. Fichtner, A 2 Gb/s balanced AES crypto-chip implementation, in: Great Lakes Symposium on VLSI, 2004, pp. 39–44.
[16] O. Harrison, J. Waldron, AES encryption implementation and analysis on commodity graphics processing units, Cryptographic Hardware and Embedded Systems (CHES) (2007) 209–226.
[17] A. Hodjat, I. Verbauwhede, A 21.54 Gbits/s fully pipelined AES processor on FPGA, in: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 308–309.
[18] K.U. Jarvinen, M.T. Tommiska, J.O. Skytto, A fully pipelined memoryless 17.8 Gbps AES-128 encryptor, in: Proceedings of the 11th International Symposium on Field Programmable Gate Arrays, 2003, pp. 207–215.
[19] A. Labbé, A. Pérez, AES implementations on FPGA: time-flexibility tradeoff, in: 12th Field Programmable Logic and Applications (FPL), 2002, pp. 836–844.
[20] M.-H. Li, A Gbps AES cipher, Master Thesis, Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, 2001.
[21] T-F. Lin, C-P. Su, C-T. Huang, C-W. Wu, A high-throughput low-cost AES cipher chip, in: 3rd IEEE Asia-Pacific Conference on ASICs (AP-ASIC), 2002, pp. 85–88.
[22] A.K. Lutz, J. Treichler, F.K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, W. Fichtner, 2 Gb/s hardware realizations of Rijndael and SERPENT: a comparative analysis, Cryptographic Hardware and Embedded Systems (CHES), Lecture Notes in Computer Science 2523 (2002) 144–158.
[23] H. Kuo, I. Verbauwhede, Architectural optimization for a 1.82 Gbits/s VLSI implementation of the AES Rijndael algorithm, Cryptographic Hardware and Embedded Systems (CHES) (2001) 51–64.
[24] S.A. Manavski, CUDA compatible GPU as an efficient hardware accelerator for AES cryptography, in: IEEE International Conference on Signal Processing and Communications (ICSPC), 2007, pp. 65–68.
[25] M. McLoone, J.V. McCanny, Single-chip FPGA implementation of the Advanced Encryption Standard algorithm, in: 11th Field Programmable Logic and Applications (FPL), 2001, pp. 152–161.
[26] M. McLoone, J.V. McCanny, Rijndael FPGA implementation utilizing look-up tables, in: IEEE Workshop on Signal Processing Systems, 2001, pp. 349–360.
[27] M. McLoone, J.V. McCanny, High performance single-chip FPGA Rijndael algorithm implementations, in: 3rd Cryptographic Hardware and Embedded Systems (CHES), 2001, pp. 65–76.
[28] S. McMillan, C. Patterson, JBits$^{TM}$ implementation of the Advanced Encryption Standard (Rijndael), in: 11th Field Programmable Logic and Applications (FPL), 2001, pp. 162–171.
[29] I. Papaefstathiou, V. Papaefstathiou, C. Sotiriou, Design-space exploration of the most widely used cryptography algorithms, Microprocessors and Microsystems 28 (2004) 561–571.
[30] V.A. Pedroni, Circuit Design with VHDL, MIT Press, Cambridge, MA, 2004.
[31] N. Pramstaller, F.K. Gürkaynak, S. Haene, H. Kaeslin, N. Felber, W. Fichtner, DPA resistant AES crypto-chip design, in: European Solid-State Circuits Conference (ESSCIRC), 2004, pp. 307–310.
[32] G.P. Saggese, A. Mazzeo, N. Mazzoca, A.G.M. Strollo, An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm, in: 13th Field Programmable Logic and Applications (FPL), 2003, pp. 292–302.
[33] N. Sklavos, O. Koufopavlou, Architectures and VLSI implementations of the AES-proposal Rijndael, IEEE Transactions on Computers 51 (2002) 1454–1459.
[34] F.X. Standaert, G. Rouvroy, J.J. Quisquater, J.D. Legat, Efficient implementation of Rijndael encryption in reconfigurable hardware: improvements and design tradeoffs, in: 5th Cryptographic Hardware and Embedded Systems (CHES), 2003, pp. 334–350.
[35] Sun Microsystems, JBits User Guide, 2004.
[36] S.-S. Wang, W.-S. Ni, An efficient FPGA implementation of Advanced Encryption Standard algorithm, in: IEEE International Symposium on Circuits and Systems (ISCAS), 2004, pp. 597–600.
[37] N. Weaber, J. Wawrzynek, Very high performance, Compact AES implementation in Xilinx FPGAs, 〈http://www.cs.berkeley/edu/~nweaver/sfra/rijndael.pdf〉, 2002.
[38] T.J. Wollinger, M. Wang, J. Guajardo, C. Paar, How well are high-end DSPs suited for the AES algorithms? In: 3rd Advanced Encryption Standard Candidate Conference, 2000, pp. 94–105.
[39] Xilinx, Xilinx Constraints Guide 8.1i, 2005.
[40] Xilinx, Virtex-II Platform FPGAs: Complete Data Sheet, 2005.
[41] Xilinx, Virtex-II Platform User Guide, 2005.
[42] S.-M. Yoo, D. Kotturi, D.W. Pan, J. Blizzard, An AES crypto chip using a high-speed parallel pipelined architecture, Microprocessors and Microsystems 29 (7) (2005) 317–326.
[43] J. Zambreno, D. Nguyen, A. Choudhary, Exploring area/delay tradeoffs in an AES FPGA implementation, in: 14th Field Programmable Logic and Applications (FPL), 2004, pp. 575–585.
[44] X. Zhang, K.K. Parhi, High-speed VLSI architectures for the AES algorithm, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12 (9) (2004) 957–967.

**José M. Granado-Criado** is a professor of Telematic Engineering in the Department of Engineering of Informatic and Telematic Systems, University of Extremadura, Spain. He received a Ph.D. degree in Computer Science from the University of Extremadura in 2009. His main research interests are FPGAs in custom-computing applications, and more concretely, applications of reconfigurable hardware to cryptography.

**Miguel A. Vega-Rodríguez** is a professor of Computer Architecture in the Department of Technologies of Computers and Communications, University of Extremadura, Spain. He received a Ph.D. degree in Computer Science from the University of Extremadura. Dr. Vega-Rodríguez has authored or co-authored more than 260 publications including journal papers, book chapters and peer-reviewed conference proceedings. Further-

more, he is editorial board member and reviewer of several international journals. Dr. Vega-Rodriguez's main research interests are reconfigurable computing (FPGAs), and parallel and distributed computing.

**Juan M. Sánchez-Pérez** is Professor of Computer Architecture in the Department of Technologies of Computers and Communications, University of Extremadura, Spain. He received a Ph.D. degree in Physics from the Complutense University of Madrid in 1976. His research interests are applications of reconfigurable hardware, logic design and modern computer architectures.

**Juan A. Gómez-Pulido** is a professor of Computer Architecture in the Deprtment of Technologies of Computers and Communications, University of Extremadura, Spain. He received a Ph.D. degree in Computer Science from the Complutense University of Madrid in 1993. His main research interests are applications of reconfigurable hardware to different fields of signal processing.