Contents

List of Figures

List of Tables

Chapter I

Compact and Efficient Encryption/Decryption Module for FPGA Implementation of AES

Gaël Rouvroy, François-Xavier Standaert,
Jean-Jacques Quisquater, and Jean-Didier Legat [I.1]

Hardware implementations of the Advanced Encryption Standard (AES) Rijndael algorithm have recently been the object of an intensive evaluation. Several papers describe efficient architectures for ASICs (Application Specific Integrated Circuits) and FPGAs (Field Programmable Gate Array). In this context, the highest effort was devoted to high throughput (up to 20 Gbps) encryption-only designs, fewer works studied low area encryption-only architectures and only a few papers have investigated low area encryption/decryption structures. However, in practice, only a few applications need throughput up to 20 Gbps while flexible and low cost encryption/decryption solutions are needed to protect sensible data, especially for embedded hardware applications. This contribution proposes an efficient solution to combine Rijndael encryption and decryption in one FPGA design, with a strong focus on low area constraints. The proposed design fits into the smallest Xilinx FPGAs, deals with data streams of 208 Mbps, uses 163 slices and 3 RAM blocks and improves by 68% the best-known similar designs in terms of ratio $Throughput/Area$. We also propose implementations in other FPGA Families (Xilinx Virtex-II) and comparisons with similar DES, triple-DES and AES implementations.

[I.1] UCL Crypto Group, Laboratoire de Microélectronique, Université catholique de Louvain, Place du Levant, 3, Belgium, `rouvroy,standaert,quisquater,legat@dice.ucl.ac.be`, `www.dice.ucl.ac.be\crypto`.

I. ENCRYPTION/DECRYPTION MODULE FOR FPGA IMPLEMENTATION OF AES

I.1 Introduction

In October 2000, NIST (National Institute of Standards and Technology) selected Rijndael [4] as the new Advanced Encryption Standard (AES), in order to replace the old Data Encryption Standard (DES). The selection process included performance evaluation on both software and hardware platforms and many hardware architectures were proposed. However, most of these architectures simply transcribe the algorithm into hardware designs, without relevant optimizations and tradeoffs. Moreover, the throughput and area constraints considered are often unrealistic as shown by the recently published results.

First, many very high-speed ($\geq$ 10 Gbps) cipher hardware implementations have been published in the literature. These designs consists of FPGA implementations of a complete unrolled and pipelined cipher. The best such DES implementation is an encryptor/decryptor based on a new mathematical description. It can achieve data rates of 21.3 Gbps in Virtex-II FPGAs [18]. The encryption/decryption mode can be changed on a cycle-by-cycle basis with no dead cycles. For the AES, the best similar RAM-based solution unrolls the 10 cipher rounds and pipelines them in an encryption-only process. This implementation in a Virtex-E FPGA produces a throughput of 11.8 Gbps [20, 21] and allows the key to be changed at every cycle. This DES implementation reaches higher throughput than the corresponding AES implementation.

However, these speed efficient designs are not always relevant solutions. Many applications require smaller throughput (wireless communication, digital cinema, pay TV, ...). Sequential designs based on a 1-round loop may be judicious and attractive in terms of hardware cost for many embedded applications. Several such implementations have been published in the literature. For DES and triple-DES designs, the most efficient solution [19] encrypts/decrypts in 18 cycles with a fresh key. For AES, the best design based on 1-round loop [20, 21] produces a data rate of 1450 Mbps (Virtex-E) using 542 slices and 10 RAM blocks, but it does not support the decryption mode. Another efficient circuit [23] proposes a compact architecture that combines encryption and decryption. It executes 1 round in four cycles and produces a throughput of 166 Mbps (Spartan-II) using 222 slices and 3 RAM blocks.

The proposed design is also based on a quarter of round loop implementation and improves by 68% (in term of ratio *Throughput/Area*) the design detailed in [23]. We investigate a good combination of encryption/decryption and place a strong focus on a very low area constraints. The resulting design fits in the smallest Xilinx devices (e.g. the Spartan-3 XC3S50 and Virtex-II XC2V40), achieves a data stream of 208 Mbps (using 163 slices, 3 RAM blocks) and 358 Mbps (using 146 slices, 3 RAM blocks), respectively in Spartan-3 and Virtex-II devices. It attempts to create a bridge between throughput and cost requirements for embedded applications.

The structure of this chapter is organized as follows: section I.2 introduces the FPGAs architecture; the mathematical description of Rijndael is referred in section I.3; section I.4 describes our sequential AES encryptor/decryptor; finally, section I.5 concludes this chapter.

I.2 The Spartan-3 CLB description

The Spartan-3 configurable logic blocks (CLBs) are organized in an array and are used to build combinatorial and synchronous logic designs. Each CLB element is tied to a switch matrix to access the general routing matrix, as shown in Figure I.1. A CLB element includes 4 similar slices, with fast local feedback within the CLB. The four slices are split into two columns of two slices with two independent carry logic chains and one common shift chain.

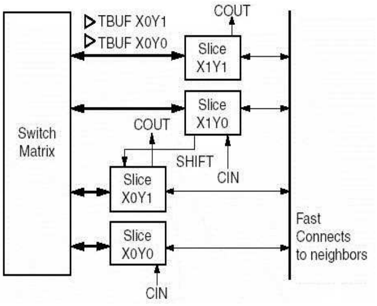

Figure I.1. The Spartan-3 CLB

Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, multiplexers and two storage elements. As shown in Figure I.2, each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed SelectRAM memory, or a 16-bit variable-tap shift register element. The output from the function generator in each slice drives both the slice output and the D input of the storage element.

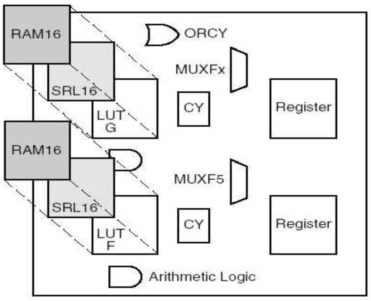

Figure I.2. The Spartan-3 slice

A specific feature of the slice is the 16-bit shift register configuration. The write operation is synchronous with a clock input and an optional clock enable. A dynamic read access is performed through the 4-bit address bus.

Table I.1. FPGA Resources

Component	XC3S50	XC2V40
CLB array: row × col.	16×12	8×8
Number of slices	768	256
Number of flip-flops	$1,536$	512
Number of LUTs	$1,536$	512
Max dist. selectRAM or shift reg. (bits)	$24,576$	$8,192$
Number of RAM blocks	4	4

Spartan-3 devices also incorporate 18-Kbit RAM blocks. These ones complement the distributed SelectRAM resources provided by the CLBs. Each RAM block is an 18-Kbit true dual-port RAM with two independently clocked and independently controlled synchronous ports that access a common storage area. Both ports are functionally identical.

Virtex-II devices exploit the same architecture as Spartan-3.

The XC3S50 and XC2V40 components are, respectively, the smallest Spartan-3 and Virtex-II components. Table I.1 illustrates the logic resources available in both components.

I.3 The AES algorithm

The Advanced Encryption Standard (AES, Rijndael) algorithm is a symmetric block cipher that processes data block of 128, 192 and 256 bits using, respectively, keys of the same length. In this paper, only the 128 bit encryption version is considered (AES-128). The 128-bit data block and key are considered as a byte array, respectively called *State* and *RoundKey*, with four rows and four columns.

Let a 128-bit data block in the i^{th} round be defined as:

$$data\_block^i = d^i_{15}|d^i_{14}|d^i_{13}|d^i_{12}|d^i_{11}|d^i_{10}|d^i_9|d^i_8|d^i_7|d_6|d^i_5|d^i_4|d^i_3|d^i_2|d^i_1|d^i_0$$

where d^i_{15} represents the most significant byte of the data block of the round i. The corresponding $State^i$ is:

$$State^i = \begin{bmatrix} d^i_{15} & d^i_{11} & d^i_7 & d^i_3 \\ d^i_{14} & d^i_{10} & d^i_6 & d^i_2 \\ d^i_{13} & d^i_9 & d^i_5 & d^i_1 \\ d^i_{12} & d^i_8 & d^i_4 & d^i_0 \end{bmatrix}$$

AES-128 consists of ten rounds. One AES encryption round includes four transformations: *SubByte*, *ShiftRow*, *MixColumn* and *AddRoundKey*. The first and last rounds differ from the other ones. Indeed there is an additional *AddRoundKey* transformation at the beginning of the first round and no *MixColumn* transformation is performed in the last round. This is done to facilitate the decryption process.

- *SubByte* (SB) is a non-linear byte substitution. It operates with every byte of the *State* separately. The substitution box (S-box) is invertible and consists of two transformations:

  1. Multiplicative inverse in $GF(2^8)$. The zero element is mapped to itself.
  2. An affine transform over $GF(2)$.

  The *SubByte* transformation applied to the *State* can be represented as follows:

$$SB(State^i) = \begin{bmatrix} SB(d^i_{15}) & SB(d^i_{11}) & SB(d^i_{7}) & SB(d^i_{3}) \\ SB(d^i_{14}) & SB(d^i_{10}) & SB(d^i_{6}) & SB(d^i_{2}) \\ SB(d^i_{13}) & SB(d^i_{9}) & SB(d^i_{5}) & SB(d^i_{1}) \\ SB(d^i_{12}) & SB(d^i_{8}) & SB(d^i_{4}) & SB(d^i_{0}) \end{bmatrix}$$

  The inverse transformation is defined *InvSubByte* (ISB).

- $ShiftRow$ (SR) performs a cyclical left shift on the last three rows of the *State*. The second row is shifted of one byte, the third row is shifted of two bytes and the fourth row is shifted of three bytes. Thus, the $ShiftRow$ transformation proceeds as follows:

$$SR(SB(State^i)) = \begin{bmatrix} SB(d^i_{15}) & SB(d^i_{11}) & SB(d^i_{7}) & SB(d^i_{3}) \\ SB(d^i_{10}) & SB(d^i_{6}) & SB(d^i_{2}) & SB(d^i_{14}) \\ SB(d^i_{5}) & SB(d^i_{1}) & SB(d^i_{13}) & SB(d^i_{9}) \\ SB(d^i_{0}) & SB(d^i_{12}) & SB(d^i_{8}) & SB(d^i_{4}) \end{bmatrix}$$

  The inverse $ShiftRow$ operation ($InvShiftRow$ (ISR)) is trivial.

- $MixColumn$ (MC) operates separately on every column of the *State*. A column is considered as a polynomial over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with the fixed polynomial $c(x)$:

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

  In the polynomial representation, multiplications in $GF(2^8)$ corresponds with multiplications of polynomials modulo an irreducible binary polynomial of degree 8. A polynomial is irreducible if it has no divisors other than 1 and itself. For Rijndael, this polynomial is called m(x) and given by $m(x) = x^8 + x^4 + x^3 + x + 1$. It follows that multiplication by $'02'$ can be implemented at byte level as a left shift and a subsequent conditional bitwise EXOR with $'1B'$ (hexadecimal).

  This can be represented as a matrix multiplication:

$$R^i = MC(SR(SB(State^i))) =$$
$$\begin{bmatrix} '02' & '03' & '01' & '01' \\ '01' & '02' & '03' & '01' \\ '01' & '01' & '02' & '03' \\ '03' & '01' & '01' & '02' \end{bmatrix} \otimes \begin{bmatrix} SB(d^i_{15}) & SB(d^i_{11}) & SB(d^i_{7}) & SB(d^i_{3}) \\ SB(d^i_{10}) & SB(d^i_{6}) & SB(d^i_{2}) & SB(d^i_{14}) \\ SB(d^i_{5}) & SB(d^i_{1}) & SB(d^i_{13}) & SB(d^i_{9}) \\ SB(d^i_{0}) & SB(d^i_{12}) & SB(d^i_{8}) & SB(d^i_{4}) \end{bmatrix}$$

To achieve the inverse operation ($InvMixColumn$ (IMC)), every column is transformed by multiplying it with a specific multiplication polynomial $d(x)$, defined by

$$c(x) \otimes d(x) = '01', \; d(x) = '0B'x^3 + '0D'x^2 + '09'x + '0E' \tag{I.2}$$

- *AddRoundKey* (AK) performs an addition (bitwise XOR) of the $State^i$ with the $RoundKey^i$:

$$AK(R^i) = \begin{bmatrix} R^i_{15} & R^i_{11} & R^i_7 & R^i_3 \\ R^i_{14} & R^i_{10} & R^i_6 & R^i_2 \\ R^i_{13} & R^i_9 & R^i_5 & R^i_1 \\ R^i_{12} & R^i_8 & R^i_4 & R^i_0 \end{bmatrix} \oplus \begin{bmatrix} rk^i_{15} & rk^i_{11} & rk^i_7 & rk^i_3 \\ rk^i_{14} & rk^i_{10} & rk^i_6 & rk^i_2 \\ rk^i_{13} & rk^i_9 & rk^i_5 & rk^i_1 \\ rk^i_{12} & rk^i_8 & rk^i_4 & rk^i_0 \end{bmatrix}$$

The inverse operation ($InvAddRoundKey$ (IAK)) is trivial.

RoundKeys are calculated with the key schedule for every *AddRoundKey* transformation. In AES-128, the original cipher key is the first $RoundKey^0$ (rk^0) used in the additional *AddRoundKey* at the beginning of the first round. $RoundKey^i$, where $0 < i \leq 10$, is calculated from the previous $RoundKey^{i-1}$. Let $p(j)$ ($0 \leq j \leq 3$) be the column j of the $RoundKey^{i-1}$ and let $w(j)$ be the column j of the $RoundKey^i$. Then the new $RoundKey^i$ is calculated as follows:

$$\begin{aligned} w(0) &= p(0) \oplus (Rot(Sub(p(3))) \oplus rcon^i \\ w(1) &= p(1) \oplus w(0) \\ w(2) &= p(2) \oplus w(1) \\ w(3) &= p(3) \oplus w(2) \end{aligned}$$

Rot is a function that takes a four byte input $[a0; a1; a2; a3]$ and rotates them as $[a1; a2; a3; a0]$. The function *Sub* applies the substitution box (S-box) to four bytes. The round constant $rcon^i$ contains values $[('02')^{i-1}; '00'; '00'; '00']$.

I.4 Our sequential AES implementation

Some designs propose an implementation based on one complete round, and iteratively loop data through this round until the entire encryption or decryption is achieved. Only one $State^i$ is processed in one cycle. These designs are suited for feedback and non-feedback modes of operation.

As mentioned in [23], the AES round offers various opportunities of parallelism. The round is composed of 16 S-boxes and four 32-bit $MixColumn$ operations, working on independent data. Only $ShiftRow$ needs to deal with the entire 128-bit $State$.

Based on this observation, we propose an implementation using four S-boxes and one $MixColumn$ in order to compact the design. This decreases the area by a factor of four but increases the time of one round to four cycles. In practice, only the time-space tradeoff is modified. A similar approach was proposed in [23].

I.4.1 Implementation of ShiftRow/InvShiftRow

In our design, the way to access the $State^i$, for the first quarter of the round, is described in Figure I.3. We read d^i_{15},d^i_{10},d^i_5,d^i_0 in parallel from the memory, and apply *SubByte*, *MixColumn* and *AddRoundKey* operations. Then we write results d^{i+1}_{15},d^{i+1}_{14},d^{i+1}_{13}, d^{i+1}_{12} to a different location in this memory. The second, third, and fourth quarters of the round are managed in a similar manner, depending on $ShiftRow$.

The best FPGA solution to implement such simultaneous read and write memory accesses is proposed in [23]. The solution is based on a shift register design. As described above, all calculations from the *AddRoundKey* are written into adjacent locations of the output memory in consecutive cycles. We store first d^{i+1}_{15},d^{i+1}_{14},d^{i+1}_{13}, d^{i+1}_{12} in parallel, then d^{i+1}_{11},d^{i+1}_{10},d^{i+1}_{9}, d^{i+1}_{8} in parallel, and so on. Therefore we can store the consecutive round results into shift registers (one shift register per row of the *State*, four shift registers for four rows). Xilinx FPGAs propose a very space efficient solution to achieve a 16-bit shift register with a dynamic variable access. Four slices can implement an 8-bit wide, 16-bit long shift register. The four dynamic variable accesses are used to read the input memory content at correct positions into the rows. Four 8-bit wide shift register are needed, which corresponds to 16 slices.

$$\begin{bmatrix} \mathbf{d^i_{15}} & d^i_{11} & d^i_7 & d^i_3 \\ d^i_{14} & \mathbf{d^i_{10}} & d^i_6 & d^i_2 \\ d^i_{13} & d^i_9 & \mathbf{d^i_5} & d^i_1 \\ d^i_{12} & d^i_8 & d^i_4 & \mathbf{d^i_0} \end{bmatrix} \Rightarrow SB + MC + AK \Rightarrow \begin{bmatrix} \mathbf{d^{i+1}_{15}} & d^{i+1}_{11} & d^{i+1}_7 & d^{i+1}_3 \\ \mathbf{d^{i+1}_{14}} & d^{i+1}_{10} & d^{i+1}_6 & d^{i+1}_2 \\ \mathbf{d^{i+1}_{13}} & d^{i+1}_9 & d^{i+1}_5 & d^{i+1}_1 \\ \mathbf{d^{i+1}_{12}} & d^{i+1}_8 & d^{i+1}_4 & d^{i+1}_0 \end{bmatrix}$$

Figure I.3. Memory accesses involved in the AES round

The $InvShiftRow$ operation can be done using the same shift registers modifying the way to access the read port of the memory.

I.4.2 Implementation of SubByte/MixColumn and InvSubbyte/InvMixColumn

Compared to the paper [23], we propose a more efficient combination of *SubByte* and *MixColumn* operations, i.e. we use less resources than separated block implementations. Our solution takes advantage of specific features of the new Xilinx devices and perfectly fits into the Spartan-3 or Virtex-II technologies[I.2].

The Spartan-3 and Virtex-II FPGAs have both dedicated 18-Kbit dual-port RAM blocks[I.3], that can be used to store tables for the combination of *SubByte* and $MixColumn$.

As also mentioned in [4], the consecutive *SubByte* and *MixColumn* operations on the first quarter of the round can be expressed as $e^i_{15..12}$:

[I.2] It is not the case with Spartan-II.

[I.3] The Spartan-II has dedicated 4-Kbit dual-port RAM blocks.

$$\begin{bmatrix} e^i_{15} \\ e^i_{14} \\ e^i_{13} \\ e^i_{12} \end{bmatrix} = \begin{bmatrix} '02' & '03' & '01' & '01' \\ '01' & '02' & '03' & '01' \\ '01' & '01' & '02' & '03' \\ '03' & '01' & '01' & '02' \end{bmatrix} \otimes \begin{bmatrix} SB(d^i_{15}) \\ SB(d^i_{10}) \\ SB(d^i_5) \\ SB(d^i_0) \end{bmatrix}$$

that is also equivalent to:

$$\begin{bmatrix} '02' \\ '01' \\ '01' \\ '03' \end{bmatrix} \otimes SB(d^i_{15}) \oplus \begin{bmatrix} '03' \\ '02' \\ '01' \\ '01' \end{bmatrix} \otimes SB(d^i_{10}) \oplus \begin{bmatrix} '01' \\ '03' \\ '02' \\ '01' \end{bmatrix} \otimes SB(d^i_5) \oplus \begin{bmatrix} '01' \\ '01' \\ '03' \\ '02' \end{bmatrix} \otimes SB(d^i_0)$$

If we define four tables (T_0 to T_3) with 256 4-byte data as:

$$T_0(a) = \begin{bmatrix} '02' \bullet SB(a) \\ SB(a) \\ SB(a) \\ '03' \bullet SB(a) \end{bmatrix} T_1(a) = \begin{bmatrix} '03' \bullet SB(a) \\ '02' \bullet SB(a) \\ SB(a) \\ SB(a) \end{bmatrix}$$

$$\mathrm{T}_2(a) = \begin{bmatrix} SB(a) \\ '03' \bullet SB(a) \\ '02' \bullet SB(a) \\ SB(a) \end{bmatrix} T_3(a) = \begin{bmatrix} SB(a) \\ SB(a) \\ '03' \bullet SB(a) \\ '02' \bullet SB(a) \end{bmatrix}$$

The combination of *SubByte* followed by *MixColumn* can be expressed as:

$$\begin{bmatrix} e^i_{15} \\ e^i_{14} \\ e^i_{13} \\ e^i_{12} \end{bmatrix} = T_0(d^i_{15}) \oplus T_1(d^i_{10}) \oplus T_2(d^i_5) \oplus T_3(d^i_0)$$

The size of one T_i table is 8 Kbits for encryption. The corresponding similar table for decryption also takes 8 Kbits (IT_0 to IT_3). It is therefore possible to achieve the complete *SubByte/MixColumn* and *InvSubByte/InvMicolumn* operations using two dual-port 18-Kbit RAM blocks.

The proposed solution significantly reduces the resources used in [23].

I.4.3 Encryption/Decryption design choices

One of the inconveniences of AES comes from the fact that the *AddRoundKey* is executed after *MixColumn* in the case of encryption and before *InvMixColumn* in the case of decryption. Such encryption/decryption implementation will therefore require additional switching logic to select appropriate data paths, which can also affects the time performance. The paper [23] mentions this problem but chooses to design like that anyway.

AES decryption nevertheless allows *InvMixColumn* and *AddRoundKey* to be reordered if we perform an additional *InvMixColumn* operation on most of the *RoundKeys* (except the first and the last *RoundKeys*). More details about such scheduling of operations can be found in [4, 5]. At first sight, *InvMixColumn* could seem to require much more area than the switching logic. This is especially true

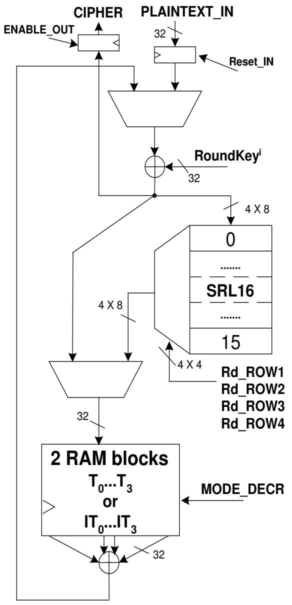

Figure I.4. Our AES data path round

if the *InvMixColumn* of the round is narrowly combined with the *InvSubByte* in RAM blocks. Nevertheless, the subsection I.4.4 proposes a solution using very few additional resources but some extra cycles to generate all inverse *Roundkeys* (*InvRoundKeys*). Figure I.4 summarizes our design choices concerning the data path round.

I.4.4 Implementation of the key schedule

The implementation of our AES key schedule is based on precomputing *RoundKeys* and *InvRoundKey* in advance and storing them in one RAM block. The difficult computation of the *InvRoundkeys* on-the-fly[I.4] completely justifies this approach.

Our implementation of the key schedule is shown in Figure I.5. First, it computes 32-bits of all the $RoundKey^i$ per clock cycle. The results are stored in one dual-port block RAM, thanks to the first port. This step takes 44 clock cycles. In the same time, we also store $SB(RoundKeys)$ data in the same RAM using the other port. It corresponds to the first step of the calculation process of *InvRoundKeys*. As mentioned in the subsection I.4.3, $InvRoundKey^i$ equals to $IMC(RoundKey^{10-i})$, except for the first $InvRoundKey^0$ and last $InvRoundKey^{10}$, that equal to respectively $RoundKey^{10}$ and $RoundKey^0$.

If we need decryption, a second step has to be applied to $SB(RoundKeys)$. We start to calculate $ISB(SB(RoundKey^{10}))$ and store it as $InvRoundKey^0$. Then, we evaluate $IMC(ISB(SB(RoundKey^{10-i})))$ that equals to $IMC(RoundKey^{10-i})$

[I.4] It is a real weak aspect of AES algorithm.

and we store it as $InvRoundKey^i$. $InvRoundKey^{10}$ is generated as $InvRoundKey^0$. This optional decryption process takes 48 cycles to generate all $InvRoundKeys$.

Due to $InvRoundKey^0$ and $InvRoundKey^{10}$, tables (T_0 to T_3) need to be changed. $InvSubByte$ has to replace the duplicated $SubByte$. We define new 16-Kbit tables (CT_0 to CT_3) combined with (IT_0 to IT_3). CT_0 is illustrated below as an example:

$$CT_0(a) = \left[\begin{array}{c|c} '02' \bullet SB(a) & '0E' \bullet SB(a) \\ SB(a) & '09' \bullet SB(a) \\ ISB(a) & '0D' \bullet SB(a) \\ '03' \bullet SB(a) & '0B' \bullet SB(a) \end{array}\right]$$

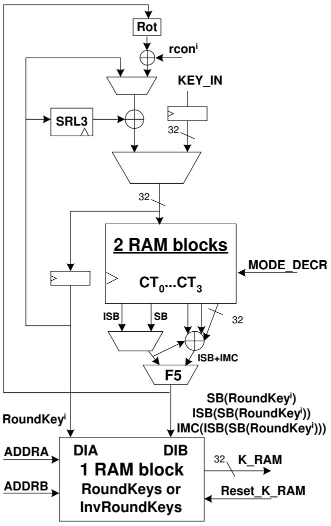

Figure I.5. Our AES key schedule

I.4.5 Implementation and results of our complete AES

Our final AES design combines the data path part and the key schedule part. Since the key schedule is done with precomputation, this part does not work simultaneously with the encryption/decryption process. It is therefore possible to share resources between both circuits. Both parts of the circuit were thought to perfectly fuse together without additional slices[I.5] and tri-state buffers. This allows reaching higher frequency than in [23]. The global design is shown in Figure I.6. We fused the key and plaintext inputs to one register. The input and output registers are packed into IOBs to improve the resources used and the global frequency of the design.

[I.5] Only new F5 multiplexers are required.

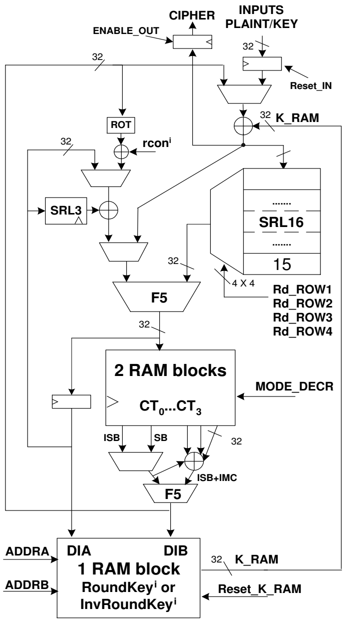

Figure I.6. Our complete AES design

The synthesis of our complete design was done using Synpllify Pro 7.2 from Synplicity. The placing and routing were done using Xilinx ISE 6.1.02i software. The final results are given in Table I.2 for Spartan-3 and Virtex-II.

As a comparison, we also set up a comparison table with the previous AES [23], DES and 3-DES [19] results. Table I.3 shows the results of these compact encryption/decryption circuits. Like others papers, we also define a ratio $Throughput/Area$ to facilitate comparisons. We finally achieve an implementation of AES which is 68% better in terms of $Throughput/Area$ assuming that Spartan-II and Spartan-3 are equivalent.

In comparison with the most efficient compact 3-DES circuits in XC2V40-6, we

Table I.2. Final results of our complete sequential AES

Device	XC3S50-4	XC2V40-6
LUTs used	293	288
Registers used	126	113
Slices used	163	146
RAM blocks	3	3
Latency (cycles)	46	46
Out. every (cycles)	1/44	1/44
Frequency	71.5 MHz	123 MHz

Table I.3. Comparisons with other sequential block cipher implementations

Algorithm	Gaj's AES	**Our AES**	**Our AES**	**Our DES**	**Our 3-DES**
Device	XC2S30-6	XC3S50-4	XC2V40-6	XC2V40-6	XC2V40-6
Slices	222	163	146	189	227
Throughput (Mbps)	166	208	358	974	326
RAM blocks	3	3	3	0	0
Throughput/Area (Mbps/slices)	0.75	**1.26**	**2.45**	**5.15**	**1.44**

can conclude that AES is more effective if we do not care about the use of three internal RAM blocks. However, 3-DES remains interesting for applications that need to regularly change the key for encryption or decryption. Indeed, our AES design takes 92 cycles, in the worst case, to calculate a new complete *InvRoundKeys*.

I.5 Conclusion

In this paper, we propose solutions for a very compact and effective FPGA implementation of AES. We combine narrowly the non-linear S-boxes and the linear diffusion layer thanks to specific features of recent Xilinx devices. We also propose a low-cost solution to deal with the subkeys computed during the decryption step. In addition, we merge efficiently the key schedule and the data path parts.

The resulting implementations fits in a very inexpensive Xilinx Spartan-3 XC3S50 FPGA, for which the cost starts below $10 per unit. This implementation can encrypt and decrypt a throughput up to 208 Mbps, using 163 slices. The design also fits in Xilinx Virtex-II XC2V40 and produces data streams up to 358 Mbps, using 146 slices. In comparison with 3-DES, AES is more efficient if we do not care about the use of three internal FPGA RAM blocks.

The throughput, low-cost and flexibility of our solution make it perfectly practical for cryptographic embedded applications.

Bibliography

[1] J.M. Rabaey, Digital Integrated Circuits, Prentice Hall, 1996

[2] Xilinx, The Spartan-3 and Virtex-II field programmable gate arrays data sheets, available from http://www.xilinx.com

[3] National Bureau of Standards, FIPS PUB 46, The Data Encryption Standard. U.S. Departement of Commerce, Jan 1977

[4] J. Daemen and V. Rijmen, The Block Cipher RIJNDAEL, NIST's AES home page, available from http://www.nist.gov/aes

[5] P.Baretto, V.Rijmen, The KHAZAD Legacy-Level Block Cipher, Submission to NESSIE project, available from http://www.cosic.esat.kuleuven.ac.be/nessie/

[6] M. Davio, Y. Desmedt, M. Fossprez, R. Govaerts, J. Hulsbosch, P. Neutjens,P. Piret, J.J. Quisquater, J. Vandewalle and P. Wouters, Analytical Characteristics of the DES, in David Chaum, editor, Advances in Cryptology - Crypto '83, pages 171–202, Berlin, 1983. Springer-Verlag

[7] FreeIP, available from http://www.free-ip.com/DES/index.html

[8] C. Patterson, High performance DES encryption in Virtex FPGAs using Jbits, in the proceedings of FCCM'01, IEEE Computer Society, 2000

[9] S. Trimberger, R. Pang and A. Singh, A 12 Gbps DES encryptor/decryptor core in an FPGA, in the proceedings of CHES'00, LNCS, pages 156–163. Springer, 2000

[10] Xilinx, V. Pasham and S. Trimberger, High-Speed DES and Triple DES Encryptor/Decryptor, available from http://www.xilinx.com/xapp/xapp270.pdf, Aug 2001

[11] Helion Technology, High Performance DES and Triple-DES Core for XILINX FPGA, available from http://www.heliontech.com

[12] CAST, Inc., Triple DES Encryption Core, available from http://www.cast-inc.com

[13] CAST, Inc., DES Encryption Core, available from http://www.cast-inc.com

[14] inSilicon, X_3 DES Triple DES Cryptoprocessor, available from http://www.insilicon.com

[15] inSilicon, X_DES Cryptoprocessor, available from http://www.insilicon.com

[16] P. Chodowiec, K. Gaj, P. Bellows and B. Schott, Experimental Testing of the Gigabit IPSec-Compliant Implementations of RIJNDAEL and Triple DES Using SLAAC-1V FPGA Accelerator Board, in the proceedings of ISC 2001: Information Security Workshop, LNCS 2200, pp.220-234, Springer-Verlag

[17] J.P. Kaps and C. Paar, Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine, in the proceedings of SAC'98: Selected Areas in Cryptography, LNCS 1556, pp. 234-247, Springer-Verlag

[18] G. Rouvroy, FX. Standaert, JJ. Quisquater, JD. Legat. Efficient Uses of FPGA's for Implementations of DES and its Experimental Linear Cryptanalysis, in IEEE Transactions on Computers, Special CHES Edition, pp. 473-482, April 2003

[19] G. Rouvroy, FX. Standaert, JJ. Quisquater, JD. Legat. Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and TripleDES. In the proceedings of FPL 2003, Lecture Notes in Computer Science, vol 2778, pp. 181-193, Springer-Verlag

[20] FX. Standaert, G. Rouvoy, JJ. Quisquater, JD. Legat, A Methodology to Implement Block Ciphers in Reconfigurable Hardware and its Application to Fast and Compact AES Rijndael, in the proceedings of FPGA 2003, pp. 216-224, ACM

[21] FX. Standaert, G. Rouvoy, JJ. Quisquater, JD. Legat, Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs, in the proceedings of CHES 2003, Lecture Notes in Computer Science, vol 2779, pp. 334-350, Springer-Verlag

[22] P. Chodowiec and K. Gaj., Comparison of the Hardware Performance of the AES Candidates using Reconfigurable Hardware, The Third Advanced Encryption Standard (AES3) Candidate Conference, April 13-14 2000, New York, USA

[23] K. Gaj and P. Chodowiec, Very Compact FPGA Implementation of the AES Algorithm, in the proceedings of CHES 2003, Lecture Notes in Computer Science, vol 2779, pp. 319-333, Springer-Verlag

[24] V. Fischer and M. Drutarovsky, Two Methods of RIJNDAEL Implementation in Reconfigurable Hardware, in the proceedings of CHES 2001: The Third International CHES Workshop, Lecture Notes In Computer Science, LNCS2162, pp 65-76, Springer-Verlag

[25] A. Rudra et al., Efficient RIJNDAEL Encryption Implementation with Composite Field Arithmetic, in the proceedings of CHES 2001: The Third International CHES Workshop, Lecture Notes In Computer Science, LNCS2162, pp 65-76, Springer-Verlag

[26] M. McLoone and J.V. McCanny, High Performance Single Ship FPGA RIJNDAEL Algorithm Implementations, in the proceedings of CHES 2001: The Third International CHES Workshop, Lecture Notes In Computer Science, LNCS2162, pp 65-76, Springer-Verlag

[27] M. McLoone and J.V. McCanny, Single-Chip FPGA Implementation of the Advanced Encryption Standard Algorithm, in the proceedings of FPL 2002: The Field Programmable Logic Conference, Lecture Notes in Computer Science, LNCS 2147, pp. 152

[28] K.U. Jarvinen, M.T. Tommiska and J.O. Skytta, A fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor, in the proceedings of FPGA 2003: Symposium on Field-Programmable Gate Arrays, pp. 207-215, ACM

[29] N. Weaver and J. Wawrzynek, High Performance Compact AES Implementations in Xilinx FPGAs, available from http://www.cs.berkeley.edu/ nweaver/Rijndael.

Index

Author Index