AN FPGA OPTIMIZED IMPLEMENTATION OF THE ADVANCED ENCRYPTION STANDARD ALGORITHM

Nikos Moshopoulos Kostas Marinis Lefteris Chaniotakis Kiamal Pekmestzi

{nikos, kmarinis, lchaniot, pekmes}@microlab.ntua.gr

Microprocessors and Digital Systems Laboratory, Department of Electrical and Electronic Engineering, National Technical University of Athens, 9 Iroon Polytechneiou, Zografou, 157 73, Athens, Greece

Key words: AES, Rijndael, Cryptography, FPGA

ABSTRACT

In this paper, an FPGA implementation of the Advanced Encryption Standard (AES) algorithm is presented. Several options regarding the use of on-chip RAM and internal pipelining are examined. The proposed architecture combines high throughput with low area consumption.

I. INTRODUCTION

The rapid evolution of the e-business during the last two decades has augmented the need for security in information and networking systems. The role of single-key cryptography has become even more prevalent in the infrastructure of secure systems. The recently announced Advanced Encryption Standard (AES) is expected to replace DES in the near future, in terms of cryptographic functionality. This introduces a small overhead for software platforms since the encryption module can be easily updated. However, applications with higher throughput requirements, require hardware support and in many cases special purpose ASICs for their efficient realization. The adoption of the new standard in the already installed hardware platforms is inevitable. Moreover, the limited amount of information and the lack of reports on cryptanalytic attacks regarding the new standard, as well as the small number of tested and verified hardware designs and implementations, increase the risk of an ASIC fabrication. On the other hand, Field Programmable Gate Array (FPGA) devices can provide the hardware performance required along with flexibility, low cost and small time-to-market margins.

The incorporation of reconfigurable logic devices into modern hardware platforms requires an in-depth study of the specifications in terms of speed, cost and expandability with respect to the application. All these aspects are examined and discussed in this paper. In section II, an overview of the Rijndael algorithm is presented. Both the algorithm core and the key scheduling are analyzed. The internal architectural blocks are well defined, taking into account the hardware resources and specific features of the target devices. In section III, each unit is carefully mapped to the available hardware resources. Several architectural approaches are discussed in section IV, varying in terms of area, speed and throughput, for different FPGA parts. Section V includes implementation results and comparisons with previous work done in this area.

II. THE RIJNDAEL ALGORITHM

The Rijndael algorithm [1] is a proposal of Joan Daemen and Vincent Rijmen of Proton World and the University of Leuven respectively. The algorithm parameters discussed in this paper are:

- ⇒ 128-bit data blocks
- ⇒ 128-bit user provided key
- ⇒ 10 rounds for encryption / decryption.

The algorithm consists of two basic functional blocks: the core and the key scheduling. The core realizes the actual process of ciphering and de-ciphering, while the key scheduling is responsible for the expansion of the user-supplied key and the generation of the intermediate round keys.

THE ALGORITHM CORE

The core consists of four discrete processing stages:

- ⇒ The BYTE_SUB transformation, which is a non-linear byte substitution operating on each of the data block bytes independently.
- ⇒ The SHIFT_ROW transformation that mixes the data block on a byte level.
- ⇒ The MIX_COLUMN transformation, which denotes a long word modulo multiplication over GF(2^8).
- ⇒ The ROUND_KEY addition that corresponds to a bit wise XOR operation of the data block with the round key.

Each stage has a 128-bit wide input and a 128-bit wide output. The structure of the algorithm core for one round of encryption/decryption is illustrated in Fig. 1.

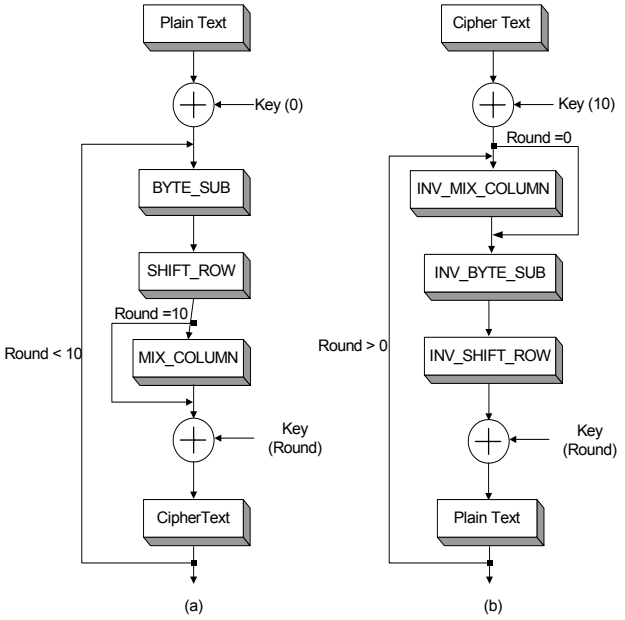

Figure 1. Rijndael algorithm flow for (a) encryption, and (b) decryption

The symbol ⊕ denotes the ADD_ROUND_KEY stage, which is a bitwise XOR between two 128-bit data sets. The round keys are provided by the key scheduling unit.
The BYTE_SUB transformation requires two types of S-Boxes (an 8x256 matrix), one for encryption and another for decryption. It is a non-linear byte substitution.
In the Forward and Inverse SHIFT_ROW transformations the 128-bit data bus is transposed on a byte basis. Considering Data[1…16] as an array of 16 bytes, the SHIFT_ROW transformation derives a new array $Data_{ShiftRow}$ according to the following algorithm:

Input : Data[1..16], 16 bytes array
Output : $Data_{ShiftRow}$ [1..16], 16 bytes array
Algorithm : Shift_Row(Data, $Data_{ShiftRow}$)
for (i = 1; i < 17; i++) {
If encryption $Data_{ShiftRow}[i] = Data[5 \cdot i \bmod 16]$;
else $Data_{ShiftRow}[i] = Data[13 \cdot i \bmod 16]$;
}

In the MIX_COLUMN transformation the 128-bit input is organized into a 4x4 byte matrix. Each column consists of four bytes, and it is considered a polynomial with coefficients in GF(2^8).
Each column is multiplied by a fixed polynomial, which is $c(x)='03'x^3+'01'x^2+'01'x+'02'$ in the forward transform, and $d(x)='0B'x^3+'0D'x^2+'09'x+'0E'$ in the inverse transform. All multiplications are modulo x^4+1, which is represented by '1B' in hexadecimal notation.
The encryption and decryption flows are very similar in terms of operations performed, as illustrated in Fig. 1. This property of the Rijndael algorithm can be used to advantage when integrating both flows into a unified data path, allowing a higher degree of resource sharing between the two flows. The resulting data path is illustrated in Fig. 2.

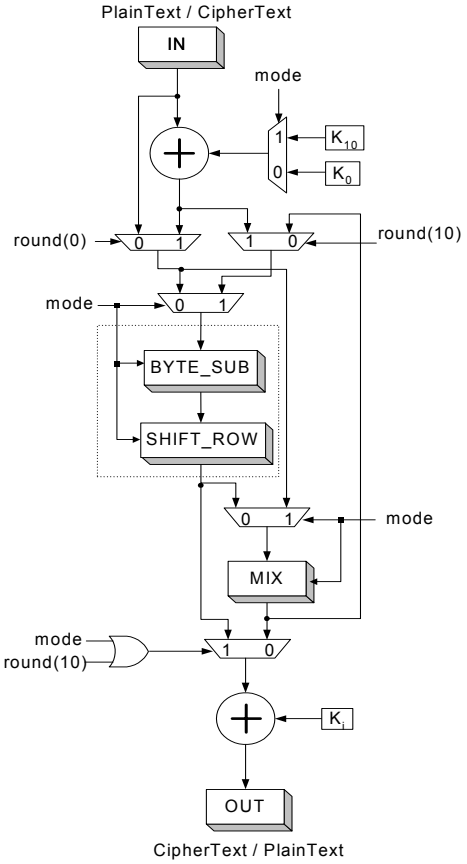

Figure 2. Combined encryption/decryption data path

The data flow for encryption/decryption is controlled by multiplexers. The "mode" control signal indicates the type of operation to be performed ('0' for encryption, '1' for decryption), while the "round_sel" signal indicates whether the current round is the first one ('0' for first round, '1' otherwise). In both modes, the pre-addition (first XOR) is executed only during the first round. The MIX_COLUMN stage is bypassed in the first round during decryption and in the final round during encryption. A different round key is used in the pre- and post-addition stages depending on the current mode and round. Since a single set of keys is used for both encryption and decryption, a unified memory scheme has been adopted; the round keys are stored in a single memory bank, and are then retrieved in ascending order (K_0 to K_{10}) during encryption and in descending order (K_{10} to K_0) during decryption. K_0 is used in the pre-addition stage during the first round of encryption, and K_{10} during the first round of decryption. In the post-addition stage, K_1 to K_{10} are used during encryption, and K_9 to K_0 are used during decryption.
The BYTE_SUB and SHIFT_ROW operations have been enclosed in a dashed-line box to indicate that their

ordering is indifferent: SHIFT_ROW simply transposes the bytes and has no effect on the byte value, while BYTE_SUB operates on individual bytes.

KEY SCHEDULING

The key scheduling is responsible for the expansion of the user-supplied key. Ten more 128-bit round keys are generated from the initial key resulting in a total of 1408 bits. The round keys are stored in a 32x44 memory bank. The key expansion algorithm is presented below.

Input : 128-bit user key organized in 16 bytes.
Output: Key memory (32x44 bits) containing the round keys.
Algorithm : Key_Sched_original(user_key, key_mem)

```
for (i = 0; i < 4; i++)
    key_mem[i] := (user_key[4·i], user_key[4·i + 1],
                   user_key[4·i + 2], user_key[4·i + 3]);
for (i = 4; i < 44; i++) {
    temp := key_mem[i –1];
    if (i mod 4 == 0)
        temp := BYTE_SUB(ROT_BYTE(temp))
                xor RCON[i /4];
    key_mem[i] := key_mem[i – 4] xor temp;
}
```

The "key_mem" variable denotes the memory bank of the round keys, while "temp" is an intermediate variable. The BYTE_SUB function has been described in the previous section. The ROT_BYTE function performs a left byte-rotation and RCON is a 4-byte constant.
Eight read accesses at the key memory are required in order to generate a new 128-bit round key, to be used by the algorithm core. These memory accesses have a severe impact on the design performance. The following modification to the algorithm eliminates this bottleneck.

Input : 128 bits wide user key organized in 16 bytes.
Output: Key memory (128x11 bits) containing the round keys.
Algorithm : Key_Sched_modified(user_key, key_mem)

```
key_mem[0] := (user_key[0], user_key[1], user_key[2],
               user_key[3]);
previous_key := key_mem[0];
for (i = 1; i < 11; i++) {
    current_key[0]:=BYTE_SUB(ROT_BYTE
                    (previous_key[3])) xor RCON[i];
    for (j = 1; j < 4; j++) {
        current_key[j] := current_key[j-1] xor
                          previous_key[j];
    }
    previous_key := current_key;
    key_mem[i] := current_key;
}
```

In this modified version of the algorithm, quad-word registers are used for the temporary storage of the previous and the current keys (variables "previous_key" and "current_key", respectively). The key memory is organized in 11 locations of 128-bit words. This organization eliminates the need of read accesses for the generation of a new round key, since all operations are performed between these two registers. Hence, a significant increase in operating speed is achieved.

III. MAPPING OF FUNCTIONAL MODULES ONTO FPGA BLOCKS

The units described in section II are modeled in RTL VHDL for the functional verification of the design. The specific features of the target technology have been taken into account in order to achieve an optimal design in terms of speed and area utilization. Each functional block has been thoroughly analyzed and mapped onto technology primitives. The implementations were targeted at the Xilinx Virtex FPGA device family [5].

MAPPING OF THE ALGORITHM CORE

The BYTE_SUB transformation: The Virtex devices incorporate several 4096-bit memory blocks, which may be used in various configurations. To take full advantage of this feature an 8x512 Block SelectRAM+ [5] configuration was selected, which allowed the storage of both an encryption and decryption S-Box in a single block of RAM.
The BYTE_SUB transformation can be applied on all 16 bytes of the input array in parallel for maximum speed. This would require 16 copies of the S-Box, i.e. 16 blocks of Block SelectRAM+. The smallest device in the Virtex family that contains a minimum of 16 Block SelectRAM+ blocks is the XCV300. The Block SelectRAM+ memory blocks are organized in two columns, one along each vertical edge of the device.
We also experimented with a different configuration, using Distributed SelectRAM+. In this configuration, the S-Boxes were implemented using look-up tables (LUTs) within the FPGA. Using the LUTs as 16x1 bit synchronous single port RAM cells, an array of 8x32 LUTs would be required for each S-Box, plus an additional amount of LUTs for multiplexing. The complete set of 16 S-Boxes would then require a total of 4096 LUTs. The Distributed SelectRAM+ has significantly lower access times compared to Block SelectRAM+. The main drawback of this configuration, however, was that the resource utilization in the device would increase dramatically. Both configurations have been implemented, tested and analyzed. The results are presented and discussed in section V.
The Forward and Inverse SHIFT_ROW transformation: Only routing resources are required for the implementation of both the forward and inverse transforms. However, a multiplexer is necessary for the selection of the appropriate transform considering the unified encryption/decryption data path of Fig. 2. A 256-

to-128 multiplexer is expected to utilize 64 LUTs, since each LUT can be used as a 4-input function generator.

The Add_Round_Key Transformation: 128 LUTs are required for the implementation, since it is only a bitwise XOR of two 128-bit values.

The forward and inverse MIX_COLUMN Transformations: Addition and subtraction of polynomials in GF(2^8) corresponds to a bitwise XOR of the coefficients of the same order. Multiplication of a polynomial by x implies a left shift, modulo x^4+1 (or '1B' hex). This implies that the product of each of the coefficients of c(x) and d(x) with a polynomial in GF(2^8), can be implemented by computing the product of the polynomial with '0001', '0010', '0100' and '1000' and performing a bitwise XOR operation between the appropriate results. The conditional reduction can be carried out by a bit wise AND of each polynomial with the value '1B' hex, and then a bitwise XOR with the shifted polynomial. A "Multiply-Byte" block has been implemented that computes the products of all four coefficients in parallel and then selects the appropriate output using a multiplexer. Four copies of this block have been used to multiply one column of the matrix in parallel, while 16 copies are needed for the multiplication of all four columns. The "Multiply-Byte" is a purely combinational block that produces an 8-bit output from an 8-bit input using simple AND, XOR and shift operations.

MAPPING OF THE KEY SCHEDULING

Using the modified algorithm presented in section II, a finite state machine is required for the implementation of the key scheduling. The key memory can be realized using either Distributed or Block SelectRAM+ modules incorporated in the Virtex devices. Minimum area has been the main goal in the implementation of the key expansion unit. A block diagram of the design is presented in Fig. 2.

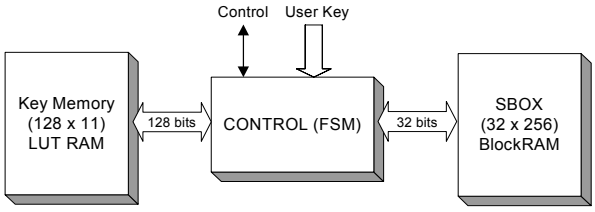

Figure 3. The key scheduling unit block diagram

The maximum-width/minimum-depth configuration allowed using Block SelectRAM+ is 16x256. Thus, the key memory would require 8 Block SelectRAM+ with a total loss of approximately 30 Kbits. If the key memory is mapped onto Distributed SelectRAM+, the expected utilization is 128 LUTs, with 640 unused bits. The Distributed SelectRAM+ solution is apparently more attractive, also considering that its access time is half that of the Block SelectRAM+.

However, in the case of the BYTE_SUB transformation, needed in the key scheduling, a Distributed SelectRAM+ based approach would not be the optimal solution: 512 LUTs would be required to store the S-Boxes, plus some additional logic for multiplexing. On the contrary, using Block SelectRAM+, two 16x256 blocks would be needed, thus saving on LUT and routing resources.

The generation of a round key is performed in five clock cycles, hence 51 cycles are required to complete the key scheduling.

IV. ARCHITECTURAL APPROACHES

All the implementations of the algorithm are based on a single-round architecture. This architecture was selected because it yields the best results in terms of throughput and area utilization. The implementations were targeted at the XCV400BG560 and the XVC1000BG560 parts of the Xilinx Virtex family. The former device contains the minimum number of Block SelectRAM+ required for the realization of both the algorithm core and the key scheduling. The latter device provides a much higher amount of hardware and routing resources thus permitting a more efficient placement and routing of the circuit. Both the Rijndael core and the key scheduling were implemented as stand-alone modules. They were also combined to form the complete chip.

Timing analysis of a combinational implementation has indicated increased delays in the INV_MIX_COLUMN transformation when using Block SelectRAM+ for the implementation of the BYTE_SUB function. To overcome this drawback, three approaches with varying levels of internal pipelining, in an effort to achieve the highest possible throughput, have been tested:

1. Registered output of the INV_MIX_COLUMN block (IMix_Reg_1).
2. Registered input and output of the INV_MIX_COLUMN block (IMix_Reg_2).
3. Registered input and output of the INV_MIX_COLUMN block plus an internal register (IMix_Reg_3).

All the results are presented in section V, along with comparisons with other implementations of the algorithm.

V. IMPLEMENTATION RESULTS AND COMPARISONS

The implementations of the algorithm core and the key scheduling have been realized with Xilinx Foundation 2.1. Constraints have been applied at the critical paths to reduce routing delays. 16 Block SelectRAM+ have been used for the BYTE_SUB transformation in the algorithm core. The core implementation results regarding the architectures discussed in section IV are summarized in Tables I and II. These results concern the use of the proposed architecture in ECB mode.

Table I: Core implementation results for the XCV400

Architecture	Slices	Clock Freq. (MHz)	Cycles per block	Throughput (TP) (Mbits/s)	TP/ #Slice (TPS)*
IMix_Reg_1	1039	37.490	10	479.872	0.462
IMix_Reg_2	1039	56.013	10	716.966	0.690
IMix_Reg_3	1049	65.612	10	839.834	0.801

(*) The TPS metric was introduced in [2]

In the LUT RAM implementation of the BYTE_SUB transformation, registers were added at the input and output of the block, since it appeared to have the longest combinational delay. As shown in Table II, the use of LUT RAM results in significantly higher area consumption. The low operating frequency is attributed to the large number of slices used, which results in poor routing.

Table II: Core implementation results for the XCV1000

Architecture	Slices	Clock Freq. (MHz)	Cycles per block	Throughput (TP) (Mbits/s)	TP/ #Slices (TPS)
IMix_Reg_2	1039	58,913	10	754.086	0.726
IMix_Reg_2*	3643	44.906	10	574.797	0.157
IMix_Reg_3	1049	92,370	10	1182.336	1.127

IMix_Reg_2: LUT RAM for the BYTE_SUB transformation*

The highest operating frequency and throughput is achieved by the implementation of the IMix_Reg_3 architecture for the XCV1000 part. These high results can be attributed to the use of internal pipelining.
The key scheduling implementation parameters are shown in Table III.

Table III: Key schedule implementation results

Key Schedule	Slices	Clock Frequency (MHz)	Cycles per block
XCV400	403	53,433	5
XCV1000	410	39,307	5

Compared to the key scheduling implementation in [4], the area is reduced by a factor of 3. This is mainly due to the use of Block SelectRAM+ for the BYTE_SUB transformation. Finally, a comparison of the IMix_Reg_3 core implementation with the implementations of [2], [3] and [4] is given in Fig. 4.

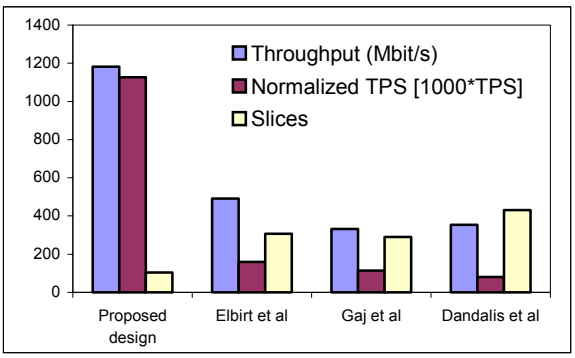

Figure 4. Comparison results for the Rijndael core implementation on the Virtex family

The use of Block SelectRAM+ resulted in more efficient area utilization, while the partitioning of the critical path via the use of registers increased the operating frequency. Thus, higher throughput has been achieved, with minimum hardware usage.

VI. CONCLUSIONS

In this paper, a hardware design of the Advanced Encryption Standard algorithm is presented. The design has been implemented in FPGA technology targeting the Xilinx Virtex family. The careful mapping of the algorithm blocks on the available hardware resources has yielded increased throughput and low area consumption. In particular, a 50% increase in the throughput has been achieved, compared to the next best results reported in the literature, while the reduction in the area is about 36%.

REFERENCES

1 J. Daemen, V. Rijmen: "The Rijndael Block Cipher", AES Proposal, September 1999, http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf

2 A. J. Elbirt, W. Yip, B. Chetwynd, C. Paar: "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", Proc. 3rd AES Conf., pp. 13 – 27, April 2000, New York, USA.

3 K. Gaj, P. Chodowiec: "Comparison of the hardware performance of the AES candidates using reconfigurable hardware", Proc. 3rd AES Conf., pp. pp. 40 – 54, April 2000, New York, USA.

4 A. Dandalis, V. K. Prasanna, J. D. P. Rolim: "A comparative study of performance of AES final candidates using FPGAs", 3rd AES Conf., April 2000, New York, USA, http://csrc.nist.gov/encryption/aes/round2/conf3/papers/23-adandalis.pdf

5 Xilinx Inc.: "Virtex 2.5V Field Programmable Gate Arrays", http://www.xilinx.com/partinfo/ds003.pdf