

# HARDWARE IMPLEMENTATION OF THE RIJNDAEL SBOX: A CASE STUDY

Marco Macchetti  
Guido Bertoni

Alari, Università della Svizzera Italiana  
Politecnico di Milano

The Rijndael algorithm was officially selected as the Advanced Encryption Standard in 2001 and will replace the DES in all applications, including Smart Card based products. For this kind of platform, a compact, area efficient hardware implementation of the algorithm is highly desirable. This paper describes such an implementation, which we have based on  $GF(2^8)$  finite field decomposition. We present our results from mappings on the STMicroelectronics ASIC technology library and discuss area, timing and power consumption figures.

## 1. INTRODUCTION

The National Institute for Standards and Technology (NIST) selected the Rijndael algorithm as the official Advanced Encryption Standard (AES) in 2001 [1]. The AES cryptographic module will, most likely, replace the old Data Encryption Standard (DES) in all application and remain the reference in symmetric key ciphers for many years to come. The Rijndael algorithm has a simple structure and can be implemented efficiently on a wide range of microprocessors, a very important feature when a fast software module is needed and when the algorithm is coded for Embedded platforms.

Although a software realization of the Rijndael cipher scheme can lead to high throughputs when compared to other block ciphers, a hardware implementation is desirable in many applications. Network cryptographic coprocessors are an example, where the primary concern is speed of implementation rather than area requirements and power consumption. Smart cards are another example of a platform that would benefit from the AES hardware module; while speed is important, the main concern is to reach minimum area requirements and limit power consumption.

This paper describes a compact implementation of the Rijndael byte substitution step (called Sbox) that can be useful for Smart Cards. The byte substitution step is crucial for the algorithm's

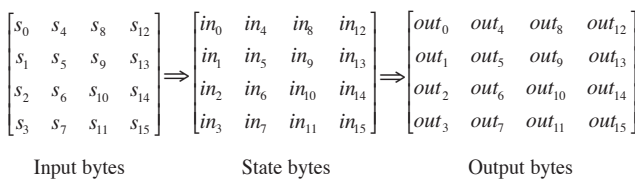
speed, as can be verified by profiling the software implementation; therefore, this component is the first candidate to be implemented in hardware. Sbox components are instantiated in the global encryption core and more specifically in the logic that calculates the algorithm's round transformation; for this reason, area savings for this component result in area saving for the whole algorithm hardware implementation.

We have organized our paper as follows: section 2 will provide a synthetic outline of the Rijndael algorithm; section 3 will describe a compact hardware implementation of the Sbox; section 4 will present findings from mappings of a real technology library; and section 5 concludes the paper.

## 2. DESCRIPTION OF THE RIJNDAEL ALGORITHM

Although the initial specification [2] of the algorithm includes 128-bits, 192-bits and 256-bits as possible lengths for both the plaintext blocks and for the key material, the approved standard will consider only 128-bit as legal block length.

The basic unit for processing in the AES algorithm is a byte, i.e., a sequence of eight bits treated as a single entity. The input, output and cipher key bit sequences are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits. Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State, which consists of four rows of bytes, each containing four bytes.



The four bytes in each column of the State array form 32-bit words; hence, the State can be interpreted as a one-dimensional array of 32-bit words (columns) where the column number provides an index into this array.

The Rijndael cipher operates in rounds, constant sets of transformations applied to the State. The number of these rounds is chosen depending on the key length and varies from 10 to 14.

Four transformations are applied in each round and correspond to the round key addition step, the non-linear, the dispersion step, and the diffusion step. We describe them as follows:

**AddRoundKey:** In this transformation, a round key is added to the State by a simple bit wise XOR operation (that is a sum in Galois Fields). Each round key consists of 4 words from the key schedule procedure.

**SubBytes:** This transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (Sbox). We construct this Sbox, which is reversible, by composing two transformations:

1. Taking the multiplicative inverse in the finite field GF(2<sup>8</sup>) with

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{1}$$

as irreducible polynomial; the element {00} is mapped onto itself.

2. Applying an affine (over GF(2)) transformation defined by:

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \tag{2}$$

for  $0 \leq i < 8$ , where  $b_i$  is the  $i$ th bit of the byte and  $c_i$  is the  $i$ th bit of a constant byte  $c$  with the value {63}.

**ShiftRows:** in this transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, row 0, is not shifted. Row 1 of the State is left shifted by 1 byte position; row 2 is left shifted by 2 byte positions; row 3 is left shifted by 3 byte positions.

Notice that the bytes are wrapped around the rows (rows, in fact, are rotated, not shifted).

**MixColumns:** This transformation operates on the State column-by-column, treating each column as a four-term polynomial over  $GF(2^8)$ . These polynomials are multiplied modulo  $(x^4 + 1)$  with a fixed polynomial  $a(x)$ , specified in the standard.

### 3. HARDWARE IMPLEMENTATION

From the previous section, we see that the Rijndael algorithm has a simple structure and can be coded efficiently for a wide range of microprocessors. However, when considering embedded applications, a hardware implementation of the cipher is desirable. Such hardware implementation should maintain flexibility of algorithm parameters, such as the key length, but at the same time result in small silicon area, effectively lowering the global cost of the system and addressing the strict area constraints typical for embedded systems and smart card platforms.

In the following, we analyze the four Rijndael transformations focusing on their possible hardware realization:

1. **AddRoundKey:** Since this transformation is a simple bit wise XOR between the round key and the cipher state, the hardware implementation consists of 128 two-inputs XOR gates.
2. **ShiftRows:** This transformation consists of a permutation of the bytes inside the cipher State. For this reason, it does not consume any silicon area and is sufficient to switch the wires corresponding to the correct bytes.
3. **MixColumns:** Even if this transformation includes finite field multiplications, there is no need to allocate full finite field multipliers. In fact, bytes from the state are always multiplied times fixed bytes specified in the algorithm description. This means the transformation is able to allocate constant multipliers. An overall size estimation for the whole MixColumns step is 560 two-inputs XOR gates.

4. **SubBytes:** Several hardware implementations of the non-linear step are possible, and there is no evident best general solution. We could exploit the particular features of the ASIC technology library or FPGA platform to limit area occupation for the Sbox component, which is critical to the success of the implementation.

Because the Sbox is based on an operation of inversion in the finite field  $GF(2^8)$ , we can propose different architectures. A broad classification divides all the possible implementations in two main categories: serial architectures and parallel architectures. While it is true that serial architectures could lead to compact circuits, in the following, we will focus on parallel ones, assuming that the dedicated round logic computes one round of the algorithm in one clock cycle with 16 Sboxes instantiated, one per each State byte. Even if this is not the case, we can pipeline a completely combinatorial implementation by inserting registers to obtain an enhanced throughput, multicycle architecture. Paper [3] shows that such pipelined combinatorial architectures tend to be very efficient in terms of both area requirements and latency when mapped to commercial FPGA units.

More simply, we could instantiate a small asynchronous ROM for each Sbox. With an ROM made up of 256 bytes, we feed the address lines with the incoming byte and read the desired output byte from data lines. Usually ASIC vendors provide memory models separately. A ROM approach is effective for large data structures more than several Kbytes long. The compiler substitutes smaller ones with combinatorial logic and maps them in an optimized way. Indeed, by writing behavioral VHDL code that describes the function we want to obtain and by letting the silicon compiler map and optimize the design according to the desired constraint(s), we achieve effective results.

This kind of implementation is very general and obviously does not exploit the mathematical structure of the particular transformation. Instead, it depends on the compiler capability of optimising circuits on the basis of the peculiar technology library.

It is possible to design more efficient structures for the Rijndael Sbox and inverse Sbox. For instance, we can implement the finite field inversion step, present in both blocks, using Fermat's Theorem, where we obtain inversion by raising the element to the 254th power; Jing et al proposes a circuit for performing such a task in [4]. This circuit contains 3 general multipliers and several blocks that perform the operations of raising one  $GF(2^8)$  element to the 2<sup>nd</sup>, 3<sup>rd</sup>, and 4<sup>th</sup> powers. The overall area occupation is 386 XOR gates, 234 AND gates, and the latency is 38 gate delays. The disadvantages of this implementation are that the area is at least three times that of the finite field multiplier, and the latency is quite high.

Schroepel offers another alternative with the implementation of the Almost Inverse Algorithm [5]. Although this is an optimised algorithm, it is more suited for fast software systems than hardware platforms; its sequential nature (underlined by the presence of loops) requires several logic circuits to be built, such as a comparator to evaluate if  $F(x)=1$ , a circuit which compares the polynomial degrees with the other complex blocks. Logic performing the Rijndael affine transformation at byte level should also be added to the inversion circuit.

Euclid's algorithm, used to find the greatest common divisor of two polynomials, can help us find the inverse of a polynomial. Brunner et al presents an inversion architecture which implements an efficient algorithm based on Euclid's [6]. The expected area occupation for such an architecture working on  $GF(2^8)$  is 51 flip-flops, 51 AND gates, 51 XOR gates, and 128 multiplexers, while the latency is 24 clock cycles.

We target the above three solutions to more general finite fields  $GF(2^n)$ , where  $n$  is equal to a prime number greater than 100. In these cases, solutions based on LUTs or combinatorial logic fail due to the high complexity of the inversion logic function.

A better alternative in the specific case is to use architectures based on composite fields. The basic idea behind this implementation is that the finite field  $GF(2^8)$  can be

considered as composite and substituted with the homomorphic field  $GF((2^4)^2)$ . In this representation, each element of the field  $GF(2^8)$  is mapped to a polynomial of the first degree with coefficients belonging to the field  $GF(2^4)$ . A linear transformation matrix  $T$ , whose coefficients can be obtained with simple calculations, governs this mapping. We know the mathematical theory of composite finite fields [7], [8] and [9] with Rudra et al first discussing application to the Rijndael algorithm [10]. Rijmen suggest the use of Optimal Normal Basis (ONB) for composite field Sbox architecture [11]; however, we have decided to use the standard polynomial basis specified for the Rijndael algorithm in our implementation.

The big advantage of the composite fields implementation is that the problem of finding the inverse of a field element is reduced from the original finite field  $GF(2^8)$  to the ground field  $GF(2^4)$ ; to give an idea of the benefit, consider that a 4096 bits look-up table can describe the inverse function in the original field, while a mere 64 bits look-up table can describe inversion in the ground field. Moreover, from the composite fields mathematical theory, we have a choice among 8 different homomorphisms between the original field  $GF(2^8)$  and the composite field  $GF((2^4)^2)$ , with each of these homomorphic mappings potentially leading to a slightly different circuit with distinct area and timing characteristics. We will choose the most area efficient circuit after the mapping on the real technology library.

Following are the details on how we have built the composite field. We choose the irreducible polynomial of the ground field  $GF(2^4)$  as

$$y^4 + y + 1 = \text{"0001 0011"} = 0x13. \quad (3)$$

$\omega^{14}$  is the 14<sup>th</sup> power of the generator element (of the sub field  $GF(2^4)$ ).  $\omega$  itself is equal to  $\text{"(0000)0010"}$  and its 14<sup>th</sup> power is equal to  $\text{"(0000)1001"}$ . We use this element to define the irreducible polynomial for the upper field, which is chosen as  $x^2 + x + \omega^{14}$ . This particular choice of the polynomial leads to an efficient implementation of the needed constant multiplier, as

explained in the following.  $a$  is equal to “0001 0000” and represents a root of the irreducible polynomial used in the upper field  $(x^2 + x + \omega^{14})$ . Thus we have that

$$a^2 + a + \omega^{14} = 0 \quad (4)$$

or

$$(0001\ 0000)^2 \oplus (0001\ 0000) \oplus (0000\ 1001) = (0000\ 0000).$$

We identify the eight different isomorphisms with different powers of  $\alpha$ ; correct powers are:

$$\alpha^5, \alpha^{10}, \alpha^{20}, \alpha^{40}, \alpha^{65}, \alpha^{80}, \alpha^{130}, \alpha^{160}$$

For instance, we describe the linear transformation which results from considering the 5<sup>th</sup> power of  $\alpha$  using 8x8  $T$  matrix, whose coefficients are elements of the finite field GF(2):

$$T = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}. \quad (5)$$

The inverse mapping is trivial and is described by the inverse of the  $T$  matrix; we have used a calculus package (Mathworks Matlab) to compute the inverse matrix but note that all the obtained coefficients have to be taken modulo 2.

Once we find all the possible linear direct and inverse mappings, we design the inner optimized circuit that will calculate the inverse in the composite field, basing our theoretic structure on the following fact. If we denote the element to be inverted with  $A(x) = a_1x + a_0$ , the inverse  $B(x) = b_1x + b_0$  is equal to:

$$B(x) = b_1x + b_0 = \frac{a_1x + (a_1 + a_0)}{a_0(a_1 + a_0) + a_1^2\omega^{14}} \quad (6)$$

In this case we perform all arithmetic operations in the ground field GF(2<sup>4</sup>). The required operations are 1 inversion, 3 general multiplications, 2 additions, 1 constant multiplication with  $\omega^{14}$  and 1 squaring. The basic circuits needed are:

- An inverter in the field GF(24), described in pure behavioral code as a simple 16 nibbles look up table. The silicon compiler will optimize the resulting circuit and will provide an efficient solution.
- Three general multipliers. We use the basic Mastrovito architecture [12] for these circuits. Each multiplier is made up of 15 two-inputs XOR gates and 16 two-inputs AND gates.
- Two adders. These components are simple bit wise XORs, consisting of 4 two-inputs XOR gates each.
- A constant multiplier. Multiplication by  $\omega^{14}$  requires 1 two-inputs XOR gate.

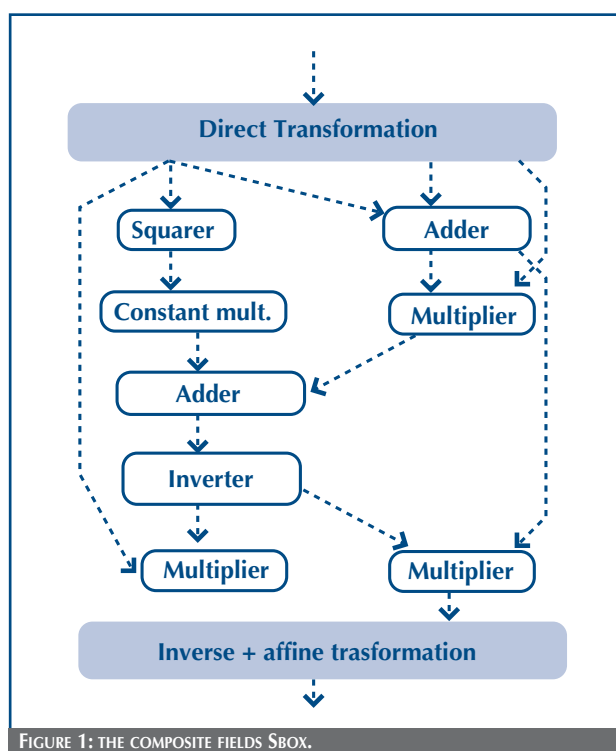


FIGURE 1: THE COMPOSITE FIELDS SBOX.

- A squaring component, which requires 2 two-inputs XOR gates.
- A component which performs the homomorphic mapping between the finite fields  $GF(2^8)$  and  $GF((2^4)^2)$ .
- A component that performs the inverse homomorphic mapping and the Rijndael affine transformation, all in one.

Fig. 1 shows the block structure of the entire circuit.

Note that we can collapse the inverse homomorphic mapping and the affine transformation into one component. It is sufficient to multiply the two  $8 \times 8$  matrices (coefficients of the result must always be taken modulo 2) and add four inverters to respect the functionality of the Rijndael affine transformation (addition of the element  $0x63$ ). This is true also in the case of the inverse Sbox, where we join the homomorphic mapping with the inverse affine transformation, thus allowing us to expect that area occupation (and saving) of the inverse Sbox will be very similar to that of the Sbox.

#### 4. RESULTS

Fig. 2 shows area and latency figures for a number of mappings of the Sbox subcomponent; all these mappings start from a high level functional description of the Sbox byte substitution step and have been obtained with the STMicroelectronics HCCMOS 0.25 $\mu$ m 1.8V technology library using Synopsys Design Compiler framework version 2001.08.

The point on the extreme left of the graph represents the effect of a strict timing constraint: we forced the compiler to map the fastest possible circuit for the specified function using the SET\_MAX\_DELAY command. The result is a 1.53 ns delay component, which is, however, quite area consuming. The point on the extreme right of the graph represents the effect of a strict area constraint, where we forced the compiler to map the smallest possible circuit for the specified function using the SET\_MAX\_AREA command. The result is a circuit

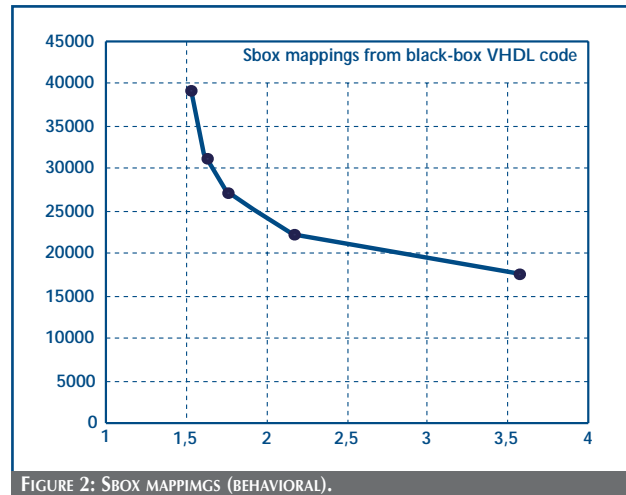


FIGURE 2: SBOX MAPPINGS (BEHAVIORAL).

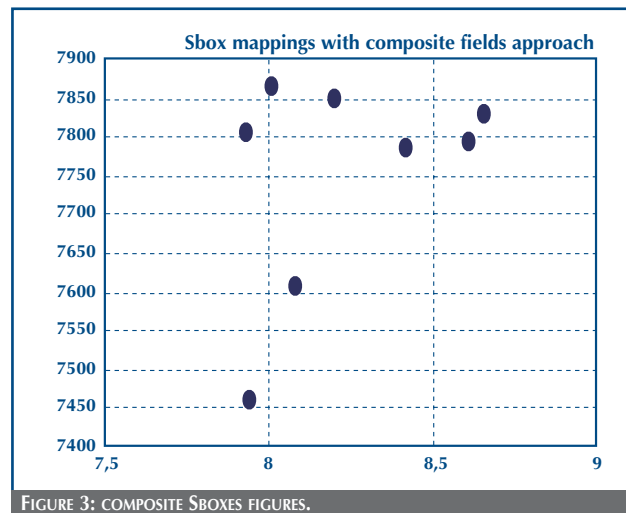


FIGURE 3: COMPOSITE SBOXES FIGURES.

which has a delay of 3.57 ns and occupies 17658 area units, i.e. only 45% the area of the previous one. We obtain points in the middle with intermediate timing constraints, clearly showing an area-latency trade-off. We have described circuits resulting from the composite fields in structural VHDL and validated their functionality. The silicon compiler further flattens and optimizes the circuits, once elaborated, for area occupation. All these implementations are very compact when compared to the ones obtained by behavioral description. Fig. 3 shows the area - timing figures for all the eight composite fields circuits; these circuits correspond to the eight

possible choices in the power of  $\alpha$ , as described in section 3. Latencies are in the order of 8 ns, while area occupation is in the order of 7700 units. The most compact circuit is based on the element  $\alpha^5$  and has a latency of 7.94 ns and an area occupation of 7461 area units, only 42% the area of the most compact circuit that we obtained from a pure behavioral VHDL code.

We have performed a complete synthesis of the round logic and observed that Sboxes are responsible for the 85% of area occupation. By using the composite fields approach, and, in particular the circuit based on the  $\alpha^5$  mapping, area saving for the whole round logic amounts to 50% when we compare this with the behavioral, high level description.

While area is certainly an important circuit cost function, we must also pay attention to power consumption of the implemented circuits, which we can measure using the Synopsys Power Compiler in conjunction with a power characterized technology library such as the aforementioned STMicroelectronics library.

The testbench used to evaluate the power consumption should replicate as much as possible the real usage of the circuit. In

the case of the Sbox circuit, we have used a statistically "good" series of random byte values to simulate and estimate power. We have implemented George Marsaglia's Mother of All Random Number Generator in VHDL and used it as a tester block. This particular RNG has a very long period and has a good statistical characterization [13].

Fig. 4 summarizes our results, which show the comparison with the power consumption figures from the behavioral mappings we previously discussed. (We did exclude some mapping in the first graph due to pareto considerations.) We obtained these power figures by testing the circuits at their maximum speed with the random byte sequence mentioned above. The number of test vectors varies from 50000 to 250000. Power consumption for the composite fields circuit is good compared with that obtained from the behavioral code. Latency is not in favor of the composite fields approach, but this implementation is useful when speed is not a primary concern. The big advantage of the composite fields solution is that it allows very compact implementation of the Sbox component and of the whole round logic; for this reason it is particularly indicated for smart cards and low cost products, and for all the applications where area saving is a primary goal.

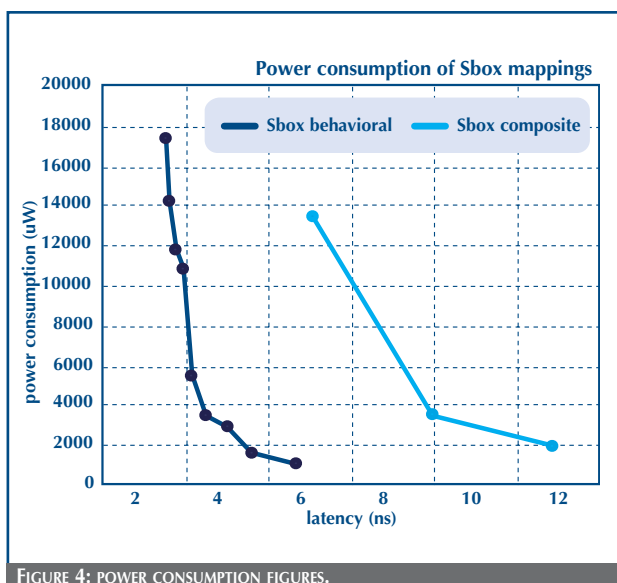


FIGURE 4: POWER CONSUMPTION FIGURES.

## 5. CONCLUSIONS

We have discussed an area efficient implementation of the Rijndael non-linear byte substitution step. We have tested this implementation, particularly indicated for Smart Card applications, using the STMicroelectronics ASIC technology library and have obtained an area saving of about 50% for the complete round logic with regard to the mapping that started from behavioral description.

## REFERENCES

- [1] "ANNOUNCING THE ADVANCED ENCRYPTION STANDARD (AES)," - Federal Information Processing Standards Publication, 2001

- [2] J. Daemen, V. Rijmen,  
“AES PROPOSAL: RIJNDAEL,” 1999
- [3] G. C. Ahlquist, B. Nelson, M. Rice,  
“OPTIMAL FINITE FIELD MULTIPLIERS FOR FPGAs,”  
Proceedings of the 9th International  
Workshop on Field-Programmable  
Logic and Applications, 1999
- [4] M. H. Jing, Y. H. Chen, Y. T. Chang, C. H. Hsu,  
“THE DESIGN OF A FAST INVERSE MODULE IN AES”
- [5] R. Schroepel, H. Orman, S. O’Malley,  
“FAST KEY EXCHANGE WITH ELLIPTIC CURVE SYSTEMS,”  
Lecture Notes in Computer Science, 1995
- [6] H. Brunner, A. Curiger, M. Hofstetter,  
“ON COMPUTING MULTIPLICATIVE INVERSES IN  $GF(2^m)$ ,”  
IEEE Transactions on Computers, 1993
- [7] C. Paar, “EFFICIENT VLSI ARCHITECTURES  
FOR BIT PARALLEL COMPUTATION IN GALOIS FIELDS,”  
PhD Thesis, 1994
- [8] J. Guajardo, C. Paar,  
“ITO-H-TSUJII INVERSION IN STANDARD BASIS  
AND ITS APPLICATION IN CRYPTOGRAPHY AND CODES,” 2001
- [9] J. Guajardo, C. Paar, “FAST INVERSION IN COMPOSITE  
GALOIS FIELDS  $GF((2^n)m)$ ,” ISIT, 1998
- [10] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar,  
J. R. Rao, P. Rohatgi,  
“EFFICIENT RIJNDAEL ENCRYPTION IMPLEMENTATION  
WITH COMPOSITE FIELD ARITHMETIC,” CHES 2001
- [11] V. Rijmen,  
“EFFICIENT IMPLEMENTATION OF THE RIJNDAEL S-BOX”
- [12] E. D. Mastrovito,  
“VLSI ARCHITECTURES FOR COMPUTATION  
IN GALOIS FIELDS,” PhD thesis, PhD dissertation,  
Linkping Univ., Linkping, Sweden, 1991
- [13] Mother of all RNG,  
<http://www.agner.org/random/mother/>
- CONTACT: ST.JOURNAL@ST.COM ■