An ASIC Implementation of the AES SBoxes*

Johannes Wolkerstorfer[1], Elisabeth Oswald[1], and Mario Lamberger[2]

[1] Institute for Applied Information Processing and Communications,
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Johannes.Wolkerstorfer@iaik.at, http://www.iaik.at

[2] Department of Mathematics
Graz University of Technology, Steyrergasse 30, A-8010 Graz, Austria

Abstract. This article presents a hardware implementation of the S-Boxes from the Advanced Encryption Standard (AES). The SBoxes substitute an 8-bit input for an 8-bit output and are based on arithmetic operations in the finite field $GF(2^8)$. We show that a calculation of this function and its inverse can be done efficiently with combinational logic. This approach has advantages over a straight-forward implementation using read-only memories for table lookups. Most of the functionality is used for both encryption and decryption. The resulting circuit offers low transistor count, has low die-size, is convenient for pipelining, and can be realized easily within a semi-custom design methodology like a standard-cell design. Our standard cell implementation on a 0.6 μm CMOS process requires an area of only 0.108 mm^2 and has delay below 15 ns which equals a maximum clock frequency of 70 MHz. These results were achieved without applying any speed optimization techniques like pipelining.

Keywords: Advanced Encryption Standard (AES), finite field arithmetic, inversion, Application Specific Integrated Circuit (ASIC), standard-cell design, Very Large Scale Integration (VLSI), scalability, pipelining.

1 Introduction

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm. It will become a FIPS standard in Fall 2001 [1]. AES will replace the DES-algorithm in the coming years since it offers higher levels of security. AES supports key lengths of 128, 192, and 256 bits. It operates on 128-bit data blocks. The major building blocks of the AES algorithm are the non-linear SBoxes (SubByte-operation) and the MixColumn-operation. Both are based on finite field arithmetic and have an inverse function which is used for decryption.

* The work described originates from the European Commission funded Project *Secure Terminal IC (SETIC)* established under contract IST-2000-25167 resp. *Crypto Module with USB Interface (USB_CRYPT)* established under contract IST-2000-25169 in the Information Society Technologies (IST) Program.

B. Preneel (Ed.): CT-RSA 2002, LNCS 2271, pp. 67–78, 2002.

```
AESROUND () {
    SubByte(State);
    ShiftRow(State);
    MixColumn(State);
    AddRoundKey(State, RoundKey);
}
```

Fig. 1. Round function of the AES-algorithm

The AES-algorithm's operations are performed on a two-dimensional array of bytes called the *State*. The State consists of four columns and four rows of bytes. For both encryption and decryption, the AES-algorithm uses a round function that is composed of four different transformations which modify the State (see Fig. 1). First, the SubByte-function substitutes all bytes of the State using a lookup-table called SBox. SBox-table entries are calculated by inversion in the finite field GF(2^8) followed by a short final transformation. Second, the rows of the State are shifted by different offsets (ShiftRow-function). The MixColumn-function scrambles the columns of the State by multiplying a finite field constant. An addition of the State with the Roundkey – which is derived from the input key – concludes the round function which is executed ten times when 128-bit keys are used. The RoundKey is calculated by operations which are similar to those of the round function and require the SBox functionality too.

The efficiency of an AES hardware implementation in terms of die-size, throughput, and power consumption is mainly determined by the implementation of the MixColumn-operation and the SBoxes. The remaining operations are trivial: ShiftRow is a simple cyclic shift, and AddRoundKey is a XOR-operation of the State and the RoundKey. Up to 20 instances of AES-SBoxes are used to realize hardware for the AES round function. The exact number of SBoxes depends on the architecture's degree of parallelism and is determined by throughput requirements and the desired clock frequency. In case that the AES-module should also decrypt data, it has to be taken into account that the SBoxes used for decryption have a different functionality. The number of SBoxes and their style of implementation has important influence on the size and the speed of an AES hardware. For this reason, V. Rijmen (one of the AES inventors) suggests in [4] an alternative method for the computation of the AES-SBox. It consists essentially of a replacement of the SBox lookup-table by an efficient combinational logic for the computation of the inverse elements in GF(2^8). Therefore, another representation of the finite field GF(2^8) is used. This representation leads to an efficient implementation of the finite field arithmetic and was investigated in connection with the implementation of error correcting codes in C. Paar [7], Soljanin et al. [5], and Mastrovito [6]. In contrast to V. Rijmen's original proposal which additionally suggests the optimal normal basis representation of finite field elements (for a definition see [3]) we use the polynomial representation of finite field elements. The benefit of our method is that we have a far more flexible hardware architecture (in comparison to the possible architectures with a straightforward SBox implementation) without the necessity to do complex conversions from one

representation (of finite field elements) to another. The main advantages of our architecture are:

- lower transistor count and die-size than a ROM-based approach,
- a short critical path to achieve a high operational frequency,
- easier implementation within a semi-custom design methodology since all computations can be done with standard-cells,
- flexibility for speed optimization: pipelining techniques can trade throughput for latency.
- suitability for a full-custom implementation: a few leaf-cells using an appropriate logic-style could increase speed or decrease power consumption.

The remainder of this article provides the mathematical background of the finite field arithmetic and the computation of the AES SBoxes in Sect. 2. The building blocks of an SBox and the according formulas are given in Sect. 3. Section 4 presents the implementation.

2 Mathematical Background

This article uses the same notation and conventions as the AES specification [1]. All notations and mathematical operations required for the SBox-operation are presented in a condensed form.

Bytes. The basic data unit of AES are Bytes $a = \{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\}$ each holding eight bits. A Byte can be interpreted as an element of the Galois-Field GF(2^8) in polynomial representation:

$$a(x) = \sum_{i=0}^{7} a_i x^i = a_7 x^7 + a_6 x^6 + a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0.$$

The coefficients a_i of a polynomial $a(x)$ are bits. Bytes can be written in different notations. For example, the binary value $\{01100011\}$ is $\{63\}$ in hexadecimal notation and represents the polynomial $x^6 + x^5 + x + 1$.

Addition. The addition of two Bytes representing polynomials $a(x), b(x) \in GF(2^8)$ is achieved by adding their corresponding coefficients modulo 2 which is a XOR-operation usually denoted with $\oplus$.

$$a(x) \oplus b(x) = \sum_{i=0}^{7} a_i x^i \oplus \sum_{i=0}^{7} b_i x^i = \sum_{i=0}^{7} (a_i \oplus b_i) x^i \tag{1}$$

The additive inverse of a Byte is the Byte itself: $-b(x) = b(x)$ and therefore subtraction is identical with addition: $a(x) - b(x) = a(x) + b(x)$.

Multiplication. The multiplication of $a(x), b(x) \in GF(2^8)$ – denoted with $a(x) \otimes b(x)$ – requires an irreducible polynomial of degree 8. For the AES-algorithm it is defined as

$$m(x) = x^8 + x^4 + x^3 + x + 1 = 1\{00011011\} \text{ (bin)} = 1\{1b\} \text{ (hex)}.$$

The multiplication $q(x) = a(x) \cdot b(x)$ in GF(2^8) is done by multiplying the polynomials $a(x)b(x)$ which yields a polynomial $p(x)$ with degree less than 15. This step is followed by a modular reduction step $q(x) = p(x)\ mod\ m(x)$ to ensure that the result is an element of GF(2^8).

A convenient method to multiply in the finite field GF(2^8) is to generate eight partial products: $P_i(x) = a(x) \cdot x^i$ and to add those partial products where the according bit b_i of the multiplier $b(x)$ is 1: $q(x) = \sum_{i=0}^{7} P_i b_i$. The partial products can be calculated efficiently by iterating a multiplication by x: $P_i(x) = P_{i-1}(x) \cdot x\ mod\ m(x), P_0(x) = a(x)$. A multiplication by x is termed *xtimes* and is given by

$$q(x) = xtimes(a) = a(x)x\ mod\ m(x) \tag{2}$$

$$q_0 = a_7, \quad q_1 = a_0 \oplus a_7, \quad q_2 = a_1, \quad q_3 = a_2 \oplus a_7$$
$$q_4 = a_3 \oplus a_7, \quad q_5 = a_4, \quad q_6 = a_5, \quad q_7 = a_6.$$

Xtimes can be implemented by a shift left operation of the input Byte a and a conditional addition of the irreducible polynomial $m(x)$ if the most significant bit (a_7) of a is set. This ensures a Byte as result.

Inversion. The multiplicative inverse a^{-1} of an element $a \in GF(2^8)$ has the property that $\forall a \in GF(2^8) \setminus \{0\} : a \otimes a^{-1} = \{1\}$. Calculating the inverse of a Byte is even more costly than multiplying Bytes. A widely used algorithm for inversion is the *extended Euclidean algorithm* described in [2]. Unfortunately, this algorithm is not suitable for a hardware implementation.

2.1 GF(2^8) as an Extension of GF(2^4)

Usually, the field GF(2^8) is seen as a field extension of GF(2) and therefore its elements can be represented as Bytes. An isomorphic – but for hardware implementations far better suited – representation is to see the field GF(2^8) as a quadratic extension of the field GF(2^4). In this case, an element $a \in$ GF(2^8) is represented as a linear polynomial with coefficients in GF(2^4),

$$a \cong a_h x + a_l, \qquad a \in \text{GF}(2^8), \qquad a_h, a_l \in \text{GF}(2^4) \tag{3}$$

and will be denoted by the pair $[a_h, a_l]$. Both coefficients of such a polynomial have four bits. All mathematical operations applied to elements of GF(2^8) can also be computed in this representation which we call *two-term polynomials*. Two-term polynomials are added by addition of their corresponding coefficients

$$(a_h x + a_l) \oplus (b_h x + b_l) = (a_h \oplus b_h)x + (a_l \oplus b_l). \tag{4}$$

Multiplication and inversion of two-term polynomials require a modular reduction step to ensure that the result is a two-term polynomial too. The irreducible polynomial needed for the modular reduction is given by

$$n(x) = x^2 + \{1\}x + \{e\}. \tag{5}$$

The coefficients of $n(x)$ are elements in GF(2^4) and are written in hexadecimal notation. Their particular values are chosen to optimize the finite field arithmetic.

The multiplication of two-term polynomials involves multiplication of elements in GF(2^4) which requires an irreducible polynomial of degree 4 which is given by

$$m_4(x) = x^4 + x + 1. \tag{6}$$

Deriving formulas for multiplication in GF(2^4) is similar to Byte-multiplication. Multiplication in GF(2^4) is given by

$$q(x) = a(x) \otimes b(x) = a(x) \cdot b(x) \ mod \ m_4(x), \quad a(x), b(x), q(x) \in \mathrm{GF}(2^4) \tag{7}$$

$$a_A = a_0 \oplus a_3, \qquad a_B = a_2 \oplus a_3$$

$$q_0 = a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3 \qquad q_1 = a_1b_0 \oplus a_Ab_1 \oplus a_Bb_2 \oplus (a_1 \oplus a_2)b_3$$

$$q_2 = a_2b_0 \oplus a_1b_1 \oplus a_Ab_2 \oplus a_Bb_3 \quad q_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_Ab_3.$$

Squaring in GF(2^4) is a special case of multiplication and is given by

$$q(x) = a(x)^2 \ mod \ m_4(x), \qquad q(x), a(x) \in \mathrm{GF}(2^4) \tag{8}$$

$$q_0 = a_0 \oplus a_2, \qquad q_1 = a_2, \qquad q_2 = a_1 \oplus a_3, \qquad q_3 = a_3.$$

The inverse a^{-1} of an element $a \in GF(2^4)$ can be derived by solving the equation $a(x) \cdot a^{-1} \ mod \ m_4(x) = 1$ as follows

$$q(x) = a(x)^{-1} \ mod \ m_4(x), \qquad q(x), a(x) \in \mathrm{GF}(2^4) \tag{9}$$

$$a_A = a_1 \oplus a_2 \oplus a_3 \oplus a_1a_2a_3$$

$$q_0 = a_A \oplus a_0 \oplus a_0a_2 \oplus a_1a_2 \oplus a_0a_1a_2$$

$$q_1 = a_0a_1 \oplus a_0a_2 \oplus a_1a_2 \oplus a_3 \oplus a_1a_3 \oplus a_0a_1a_3$$

$$q_2 = a_0a_1 \oplus a_2 \oplus a_0a_2 \oplus a_3 \oplus a_0a_3 \oplus a_0a_2a_3$$

$$q_3 = a_A \oplus a_0a_3 \oplus a_1a_3 \oplus a_2a_3.$$

The concatenation of two bits a_ia_j in Equations 7 and 9 represents a binary multiplication which is an AND-operation. In contrast to inversion in GF(2^8), inversion in GF(2^4) is suitable for a hardware implementation using combinational logic since all Boolean equations depend only on four input bits.

Inversion of Two-Term Polynomials. Inversion of two-term polynomials is the equivalent operation to inversion in GF(2^8). A multiplication of a two-term polynomial with its inverse yields the 1-element of the field: $(a_hx + a_l) \otimes (a'_hx +$

$a'_l) = \{0\}x + \{1\}, \quad a_h, a_l, a'_h, a'_l \in GF(2^4)$. From this definition the formula for inversion can be derived:

$$(a_h x + a_l)^{-1} = a'_h x + a'_l = (a_h \otimes d)x + (a_h \oplus a_l) \otimes d \qquad (10)$$
$$d = ((a_h^2 \otimes \{e\}) \oplus (a_h \otimes a_l) \oplus a_l^2)^{-1}.$$

Inversion of two-term polynomials involves only operations in GF(2^4) which are suitable for a hardware implementation using combinational logic. Most of the functionality is used to calculate the term d which is used to calculate both coefficients of the inverted two-term polynomial.

Transition between Representations of GF(2^8). The finite field GF(2^8) is isomorphic to the finite field GF($(2^4)^2$) which means that for each element in GF(2^8) there exists exactly one element in GF($(2^4)^2$). The bijection from an element $a \in GF(2^8)$ to a two-term polynomial $a_h x + a_l$ where $a_h, a_l \in GF(2^4)$ is given by the function *map*:

$$a_h x + a_l = map(a), \quad a_h, a_l \in \mathrm{GF}(2^4), \quad a \in \mathrm{GF}(2^8) \qquad (11)$$
$$a_A = a_1 \oplus a_7, \quad a_B = a_5 \oplus a_7, \quad a_C = a_4 \oplus a_6$$
$$a_{l0} = a_C \oplus a_0 \oplus a_5, \quad a_{l1} = a_1 \oplus a_2, \quad a_{l2} = a_A, \quad a_{l3} = a_2 \oplus a_4$$
$$a_{h0} = a_C \oplus a_5, \quad a_{h1} = a_A \oplus a_C, \quad a_{h2} = a_B \oplus a_2 \oplus a_3, \quad a_{h3} = a_B.$$

The inverse transformation (map^{-1}) converts two-term polynomials $a_h x + a_l$, $a_h, a_l \in GF(2^4)$ back into elements $a \in GF(2^8)$. It is given by

$$a = map^{-1}(a_h x + a_l), \quad a \in \mathrm{GF}(2^8), \quad a_h, a_l \in \mathrm{GF}(2^4) \qquad (12)$$
$$a_A = a_{l1} \oplus a_{h3}, \quad a_B = a_{h0} \oplus a_{h1},$$
$$a_0 = a_{l0} \oplus a_{h0}, \quad a_1 = a_B \oplus a_{h3}$$
$$a_2 = a_A \oplus a_B, \quad a_3 = a_B \oplus a_{l1} \oplus a_{h2}$$
$$a_4 = a_A \oplus a_B \oplus a_{l3}, \quad a_5 = a_B \oplus a_{l2}$$
$$a_6 = a_A \oplus a_{l2} \oplus a_{l3} \oplus a_{h0}, \quad a_7 = a_B \oplus a_{l2} \oplus a_{h3}$$

Both transformations can be derived by following the procedure given in C. Paar's PhD thesis [7]. They differ from the transformations given in [5] because the irreducible polynomial $m(x)$ for GF(2^8) is different.

3 SBox Building Blocks

The SubByte transformation operates independently on each Byte of the State using a substitution table (SBox). An AES-SBox is composed of two transformations:

1. Calculate the multiplicative inverse in the finite field GF(2^8). The element {00} is mapped to itself. Table 1 presents the inversion.

Table 1. Inversion of a Byte $\{xy\} \in GF(2^8)$ in hexadecimal notation.

$x \backslash y$	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	00	01	8d	f6	cb	52	7b	d1	e8	4f	29	c0	b0	e1	e5	c7
1	74	b4	aa	4b	99	2b	60	5f	58	3f	fd	cc	ff	40	ee	b2
2	3a	6e	5a	f1	55	4d	a8	c9	c1	0a	98	15	30	44	a2	c2
3	2c	45	92	6c	f3	39	66	42	f2	35	20	6f	77	bb	59	19
4	1d	fe	37	67	2d	31	f5	69	a7	64	ab	13	54	25	e9	09
5	ed	5c	05	ca	4c	24	87	bf	18	3e	22	f0	51	ec	61	17
6	16	5e	af	d3	49	a6	36	43	f4	47	91	df	33	93	21	3b
7	79	b7	97	85	10	b5	ba	3c	b6	70	d0	06	a1	fa	81	82
8	83	7e	7f	80	96	73	be	56	9b	9e	95	d9	f7	02	b9	a4
9	de	6a	32	6d	d8	8a	84	72	2a	14	9f	88	f9	dc	89	9a
a	fb	7c	2e	c3	8f	b8	65	48	26	c8	12	4a	ce	e7	d2	62
b	0c	e0	1f	ef	11	75	78	71	a5	8e	76	3d	bd	bc	86	57
c	0b	28	2f	a3	da	d4	e4	0f	a9	27	53	04	1b	fc	ac	e6
d	7a	07	ae	63	c5	db	e2	ea	94	8b	c4	d5	9d	f8	90	6b
e	b1	0d	d6	eb	c6	0e	cf	ad	08	4e	d7	e3	5d	50	1e	b3
f	5b	23	38	34	68	46	03	8c	dd	9c	7d	a0	cd	1a	41	1c

2. Apply the affine transformation which is given by Equation 13.

$$
\begin{aligned}
q &= \text{aff\_trans}(a) \qquad (13)\\
a_A &= a_0 \oplus a_1, a_B = a_2 \oplus a_3,\\
a_C &= a_4 \oplus a_5, a_D = a_6 \oplus a_7\\
q_0 &= \overline{a_0} \oplus a_C \oplus a_D\\
q_1 &= \overline{a_5} \oplus a_A \oplus a_D\\
q_2 &= a_2 \oplus a_A \oplus a_D\\
q_3 &= a_7 \oplus a_A \oplus a_B\\
q_4 &= a_4 \oplus a_A \oplus a_B\\
q_5 &= \overline{a_1} \oplus a_B \oplus a_C\\
q_6 &= \overline{a_6} \oplus a_B \oplus a_C\\
q_7 &= a_3 \oplus a_C \oplus a_D.
\end{aligned}
\qquad
\begin{aligned}
q &= \text{aff\_trans}^{-1}(a) \qquad (14)\\
a_A &= a_0 \oplus a_5, a_B = a_1 \oplus a_4,\\
a_C &= a_2 \oplus a_7, a_D = a_3 \oplus a_6\\
q_0 &= \overline{a_5} \oplus a_C\\
q_1 &= a_0 \oplus a_D\\
q_2 &= \overline{a_7} \oplus a_B\\
q_3 &= a_2 \oplus a_A\\
q_4 &= a_1 \oplus a_D\\
q_5 &= a_4 \oplus a_C\\
q_6 &= a_3 \oplus a_A\\
q_7 &= a_6 \oplus a_B.
\end{aligned}
$$

Overlined bits in Equation 13 and 14 denote inverted bits. Decryption requires the inverse function of SubByte (*InvSubByte*) which reverses the SBox-operation by applying the inverse affine transformation first (Equation 14). Then, the multiplicative inverse in the finite field GF(2^8) is calculated.

Inversion in the finite field GF(2^8) is needed to calculate the SubByte-function as well as InvSubByte. It makes sense, to merge the encryption SBox with the decryption SBox in order to reuse the finite field inversion circuit for decryption. Figure 2 depicts this approach. The control signal *enc* switches between encryption and decryption. If encryption is chosen (*enc*=1), the inverse affine transformation (*aff_trans*$^{-1}$) is bypassed and the input a is directly fed

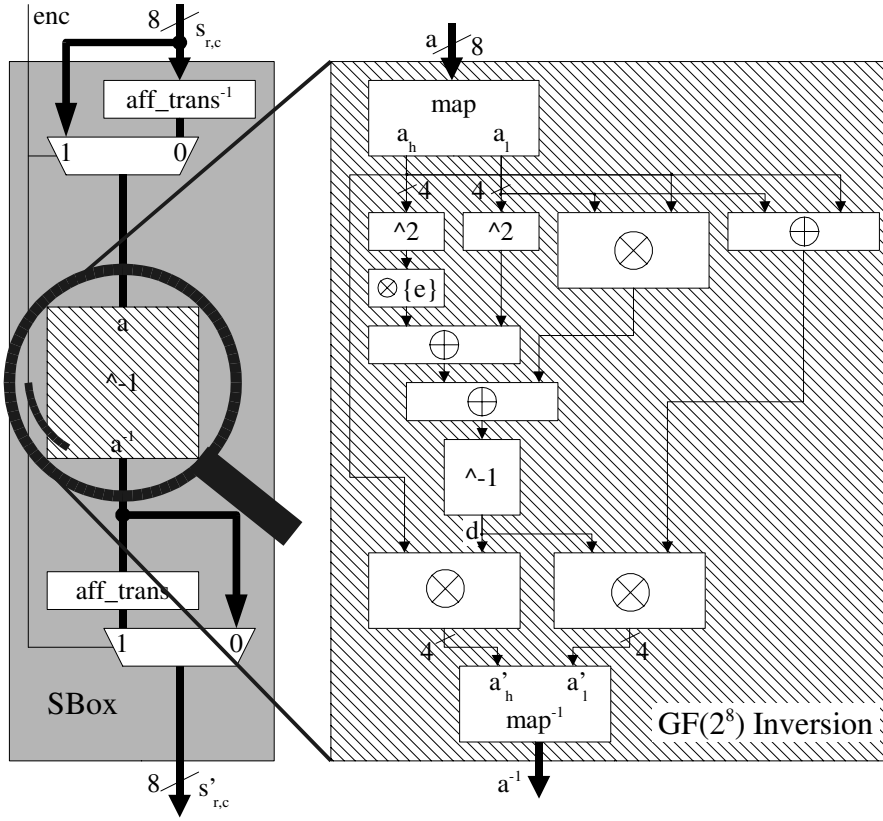

Fig. 2. Architecture of the AES-SBox

into the inversion circuit. The output of the inversion circuit is modified by the affine transformation block which calculates the result of the SubByte-function. During decryption (enc=0), the inverse affine transformation is active and the affine transformation is bypassed to calculate InvSubByte. The delay for encryption and decryption is essentially the same because the circuit's complexity for the affine transformation and its inverse are equal.

The circuit for inversion of elements in GF(2^8) covers most of the SBox functionality. In our approach the inversion is calculated with combinational logic and is based on Equation 10 which operates in GF(2^4). The operations occurring in this equation correspond to the function blocks shown in Fig. 2. Furthermore, this function block has to convert data from GF(2^8) to two-term polynomials and vice versa. The blocks map resp. map^{-1} provide this functionality based on Equation 11 and 12. Addition of elements in GF(2^4) is accomplished by a bitwise XOR-operation. Squaring relies on Equation 9. Multiplication of an element in GF(2^4) with the constant $\{e\}$ is given by

$$q = a \otimes \{e\} \tag{15}$$

$$\begin{aligned} a_A &= a_0 \oplus a_1,\ a_B = a_2 \oplus a_3 \\ q_0 &= a_1 \oplus a_B \\ q_1 &= a_A \\ q_2 &= a_A \oplus a_2 \\ q_3 &= a_A \oplus a_B. \end{aligned}$$

Multiplication and inversion in GF(2^4) are the most complex function blocks and rely on Equations 7 and 9.

4 Implementation

Our implementation is based on the architecture described in Sect. 3. It has a maskable affine transformation block, a maskable inverse affine transformation block and a block which calculates the inverse in the finite field GF(2^8). Most of the functionality can be implemented with XOR-gates. Additionally, inverters, AND-gates, and 2-to-1 multiplexers are required, but they can be neglected for performance analysis purposes.

Table 2. Complexity of the SBox

block	d_{XOR}	XORs	instances	sum XORs
aff_trans	3	16	1	16
aff_trans^{-1}	2	12	1	12
map	2	11	1	11
map^{-1}	3	15	1	15
$\oplus$	1	4	3	12
$\wedge^2$	1	2	2	4
$\otimes\{e\}$	2	5	1	5
$\otimes$	2	12	3	36
$\wedge^{-1}$	3	12	1	12
	Max 15			Sum 123

Table 3. Pipelining of the SBox

stages	flipflops	frequency	area
0	0	100%	100%
1	12	178%	111%
3	28	205%	151%

Table 2 lists the resources of all blocks – measured in number of XORs. The overall amount of gates are 123 XOR-gates with two inputs, 16 2-to-1 multiplexers and a dozen of inverters and AND-gates. If XOR-gates with three inputs are available, the number of gates can be reduced. The delay of the blocks is measured in numbers of XOR-gates in series (d_{XOR}). The critical path for encryption is composed of 15 XOR-gates in series. For decryption it is 14 because the inverse of the affine transformation has lower complexity. XOR-gates with three inputs will shorten the critical path and improve performance.

A feature that can be exploited to gain higher throughput is pipelining. Pipelining is a technique which subdivides the critical path by insertion of storing elements (flipflops). Subdividing the SBox functionality into a number of stages

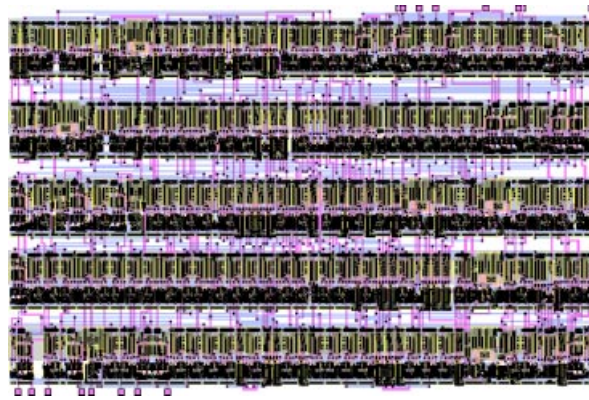

Fig. 3. Layout of the AES-SBox

is easy to accomplish since flipflops can be inserted nearly anywhere when SBoxes are implemented with combinational logic. Pipelining introduces latency but the additional clock cycles are made up by an increased clock frequency as shown in Table 3. This technique will offer best results if the number of SBox instances used for an AES implementation is kept low (e.g. 4), otherwise latency will consume more time than it is saved by shortening the critical path.

Our implementation of the AES-SBox which combines the SubByte-function and InvSubByte-function from the AES algorithm is a standard-cell circuit on a 0.6 μm CMOS process from AMS using two metal layers. It has an area of 0.108 mm^2 and contains 1624 transistors (406 NAND equivalents) when no pipelining is used. A layout is shown in Fig. 3. Layout simulation with typical mean parameters considering parasitics yields a delay below 14.2 ns which equals a maximum clock frequency of 70 MHz for both encryption and decryption. A pipelined version with one stage is slightly bigger (0.120 mm^2), contains nearly 2000 transistors (500 NAND equivalents) and yields a delay below 8 ns (125 MHz). Further increasing the number of pipeline stages will not improve throughput that strong and has a worse ratio of performance benefit to area penalty. It should be considered when the maximum clock frequency is of utmost interest.

For comparison, a lookup-table based approach requires a 4k-bit ROM to store the 256 8-bit entries of SubByte and InvSubByte. An implementation on the same process technology uses about 0.200 mm^2 and has an estimated maximum frequency of 100 MHz [9].

For an SBox implementation using a full-custom design methodology we suggest to use a differential logic style [8]. The logic functions of an SBox based on combinational logic are dominated by XOR-gates and differential XOR-gates offer good performance and have a moderate transistor count. We assume that a full-custom design could halve the required chip area because an SBbox is a small module and does not require the excessive driving capability offered by

standard cells. Output transistors of gates can be dimensioned smaller and this will in turn make it possible to scale all transistors down without deteriorating performance. At least three leaf cells are needed: a XOR-gate, an AND-gate and an inverter. To develop these cells will be an rewarding task if the resulting performance gain and area saving is pictured.

5 Related Work

Several AES-implementations have been presented recently. For comparison with our work, only ASIC circuits [11] or circuits exploiting a more efficient finite field arithmetic are of interest [10]. The approach followed in IBM implementation [10] is also based on a conversion of elements in $GF(2^8)$ into two-term polynomials. In contrast to our approach, they calculate the whole round function in this representation. Therefore, they choose the conversion function $map()$ in a way that minimizes the overall gate count. Our primary focus on choosing $map()$ was to minimize the critical path of the complete SBox and secondarily to keep the gate count low. Our $map()$-function has a shorter critical path compared to the IBM implementation, the critical path of $map^{-1}()$ is identical.

6 Conclusion

This article presented an ASIC implementation of the SBoxes from the Advanced Encryption Standard (AES). It is based on finite field arithmetic rather than using lookup-tables. This approach offers higher flexibility. Area requirements can be traded for the maximum clock frequency. The architecture can be easily implemented within a standard-cell design methodology because it completely relies on combinational logic. It is also well suited for a full-custom implementation since it uses only a few leaf cells. We implemented the AES-SBox on 0.6 μm CMOS process with standard-cells. The most promising configuration of design parameters we found is a single stage pipeline architecture. It has a silicon area of 0.12 mm^2 and a maximum clock frequency of 125 MHz. This configuration has a latency of one clock cycle like a ROM based approach. In comparison to ROMs, it offers better performance on smaller area and can even be improved by exploiting better suited logic styles like differential logic.

References

1. NIST, *Advanced Encryption Standard (AES)*, FIPS PUBS 197, National Institute of Standards and Technology, November 2001.
2. A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997.
3. R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, Cambridge, 1986.
4. V. Rijmen, *Efficient Implementation of the Rijndael SBox*, `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/` .

5. E. Soljanin, R. Urbanke, *An Efficient Architecture for Implementation of a Multiplier and Inverter in GF(2^8)*, Lucent Technologies.
6. E. D. Mastrovito, *VLSI Architectures for Computations in Galois Fields*, PhD thesis, Linköping University, Linköping, Sweden, 1991.
7. C. Paar, *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*, PhD thesis, Universität Essen, 1994.
8. J. B. Kuo, J. H. Lou, *Low-Voltage VLSI Circuits*, John Wiley, New York, Jan. 1999.
9. AMS, *Memory Compiler for Diffusion Programmable ROM in 0.6 μm CMOS*, `http://www.amsint.com/databooks/`.
10. A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, P. Rohatgi, *Efficient Rijndael Encryption Implementation with Composite Field Arithmetic*, Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, France, 2001, to be published in Springer LNCS.
11. I. Verbauwhede, H. Kuo, *Architectural Optimization for a 1.82 Gbits/sec VLSI Implementation of the AES Rijndael Algorithm*, Proceedings of Workshop on Cryptographic Hardware and Embedded Systems, France, 2001, to be published in Springer LNCS.