Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

FPGA Implementation of a Recently Published Signature Scheme

Jean-Luc Beuchat ,
Nicolas Sendrier ,
Arnaud Tisserand ,
Gilles Villard

March 2004

Research Report N° RR2004–14

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

FPGA Implementation of a Recently Published Signature Scheme

Jean-Luc Beuchat , Nicolas Sendrier , Arnaud Tisserand , Gilles Villard

March 2004

Abstract

An algorithm producing cryptographic digital signatures less than 100 bits long with a security level matching nowadays standards has been recently proposed by Courtois, Finiasz, and Sendrier. This scheme is based on error correcting codes and consists in generating a large number of instances of a decoding problem until one of them is solved (about $9! = 362880$ attempts are needed). A careful software implementation requires more than one minute on a 2GHz Pentium 4 for signing. We propose a first hardware architecture which allows to sign a document in 0.86 second on an XCV300E-7 FPGA, hence making the algorithm practical.

Keywords: Cryptography, digital signature, code-based cryptosystems, FPGA implementation

Résumé

Courtois, Finiasz et Sendrier ont récemment proposé un algorithme produisant des signatures numériques de moins de 100 bits et satisfaisant les exigences de sécurité actuelles. Basé sur les codes correcteurs d'erreurs, cet algorithme consiste à générer un grand nombre d'instances d'un problème de décodage, jusqu'à ce que l'une d'elles admette une solution (environ $9! = 362880$ essais sont nécessaires). Une implantation logicielle optimisée nécessite plus d'une minute pour produire une signature à l'aide d'un Pentium 4 cadencé à 2 GHz. Nous proposons une première implantation matérielle permettant de signer un document en 0.86 seconde sur un FPGA XCV300E-7.

Mots-clés: Cryptographie, signature numérique, cryptosystèmes basés sur les codes correcteurs, implantation sur FPGA

FPGA Implementation of a Recently Published Signature Scheme *

Jean-Luc Beuchat[1], Nicolas Sendrier[2], Arnaud Tisserand[3,1], Gilles Villard[4,1]

[1] Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon, 46, Allée d'Italie, F–69364 Lyon Cedex 07
{first name.last name}@ens-lyon.fr
[2] Projet Codes, INRIA Rocquencourt BP 105, F–78153 Le Chesnay Cedex
Nicolas.Sendrier@inria.fr
[3] INRIA – Institut National de Recherche en Informatique et Automatique
[4] CNRS – Centre National de la Recherche Scientifique

1 Introduction

A cryptographic digital signature is a sequence of digits appended to any message which allows anyone to make sure both of the content and of the origin of the message. A signature scheme is proposed in [2], which allows the production of signatures less than 100 bits long with a security level matching nowadays standards. For software implementations, the relatively high signature time of the scheme could represent a main drawback. In this paper we demonstrate the applicability of the approach with the design of a dedicated hardware architecture. Indeed, a careful software implementation requires more than one minute on a 2GHz Pentium 4 to produce one signature. We are able to sign in about 0.86 second on an XCV300E-7 FPGA. A disavantage of the signature scheme may concern its large public key, which size is about one Mbyte. This may not be crucial with available low prices memories.

The signature scheme of [2] compares favourably with other existing approaches. To match the present security requirements, classical techniques (RSA, DSA, Elliptic Curves) produce digital signatures of length 320 to 1024 bits. Producing shorter signatures can be of great interest for some applications, typically when either the transmission channel or the physical data support has a limited capacity. For instance one might wish to transmit a signature by voice on a phone, or print an authentifier on a banknote or a letter, store many signatures on a small device like a smart card.

The signature algorithm we have implemented, briefly described in Section 2, is based on a 9-error correcting Goppa code of length 2^{16}. This corresponds to the secure parameters $t = 9$, $n = 2^{16}$ and $m = 16$ in [2]. The algorithm operates on elements of the finite field $\mathbb{F}_{2^{16}}$, and manipulates polynomials of one variable z whose coefficients are in $\mathbb{F}_{2^{16}}$. For the representation of the elements and the arithmetic operations in $\mathbb{F}_{2^{16}}$ we rely on the design of [5], of which key points are recalled in Section 3.

Our main objective is to prove the feasibility and the interest of the signature scheme on FPGA boards. Due to packaging reasons, we further impose a size constraint corresponding to an XCV300E-7 FPGA. As we shall see in Section 4, this size constraint does not allow a straightforward implementation of the algorithm. In particular the two main steps —the Berlekamp-Massey algorithm and a polynomial divisibility test— cannot be implemented as separate blocks. Our solution is to show that both steps can share the same hardware resources, and can be implemented by designing a specific processor.

2 The Signature Algorithm

The signature algorithm is based on a public key McEliece/Niederreiter cryptosystem. Its underlying code is a 9-error correcting binary Goppa code of length $n = 2^{16}$, and dimension $n - tm =$

* This work is supported by the *ACI "Sécurité Informatique"* of the French *Ministère délégué à la Recherche et aux Nouvelles Technologies* and by the Swiss National Science Foundation.

$2^{16} - 9 \times 16$ [2]. The code is defined using the 2^{16} elements of $\mathbb{F}_{2^{16}}$, and the public key is a $tm \times n = 144 \times 2^{16}$ parity check matrix. The security of the signature scheme relies on the difficulty of decoding. Indeed, for a document D, the signature is obtained from the decryption (an n-bit word of weight 9) of a syndrome $s^{(i)}$ (a 144-bit word) associated to D. The computational signing task is to find a decodable syndrome to be used. For two hashing functions h and h_c, one considers $s = h(D)$ and $s^{(i)} = h_c(D, i)$, $i \geq 0$. The algorithm essentially increments the index i until $s^{(i)}$ is decodable. We have implemented the most time consuming part of the process:

while $s^{(i)}$ is not decodable **do**
  1: Computation a double syndrome S (a 288-bit word) from $s^{(i)} = h_c(s, i)$;
  2: Berlekamp-Massey algorithm for a candidate error locator $\sigma(z) \in \mathbb{F}_{2^{16}}[z]$;
  3: Divisibility test on $\sigma(z)$ that determines whether $s^{(i)}$ is decodable;
  $i \leftarrow i + 1$;
end while

The signature cost is essentially the cost of this while loop. About $9! = 362880$ decoding attempts are required on the average (with a very small standard deviation) [2]. We shall detail the architecture design of the three steps of the loop in Section 4. As commonly done for the syndrome elements and the error location, we manipulate polynomials $S(z) = \sum_{i=0}^{17} S_i z^i$ and $\sigma(z) = \sum_{i=0}^{9} \sigma_i z^i$ over $\mathbb{F}_{2^{16}}$. Step 2 and step 3 are briefly detailed below, the polynomial σ is computed using the Berlekamp-Massey algorithm for solving the key equation $S(z) = \omega(z)/\sigma(z) \mod z^{2t}$. Then, the decodability of $s^{(i)}$ is checked through a divisibility test, which determines whether σ has $t = 9$ distinct roots in $\mathbb{F}_{2^{16}}$ that situate the errors.

Berlekamp-Massey Algorithm. The Berlekamp-Massey algorithm efficiently solves the key equation and compute σ [3,1]. Searching for a polynomial of degree $t = 9$ exactly and assuming that the discrepancies are non-zero, slightly simplify the implementation. We compute $\sigma^{(2t-1)} = \sigma$ using the following $2t$-step iterative and division-free scheme:

$$\sigma^{(i)}(z) = \delta^{(i-1)}\sigma^{(i-1)}(z) - \Delta^{(i)} z \rho^{(i-1)}(z), \tag{1}$$

$$\rho^{(i)}(z) = \begin{cases} \sigma^{(i-1)}(z) & \text{if } i \text{ is even,} \\ z\rho^{(i-1)}(z) & \text{if } i \text{ is odd,} \end{cases} \tag{2}$$

$$\delta^{(i)} = \begin{cases} \Delta^{(i)} & \text{if } i \text{ is even,} \\ \delta^{(i-1)} & \text{if } i \text{ is odd,} \end{cases} \tag{3}$$

where $0 \leq i < 2t$, $\sigma^{(-1)}(z) = \rho^{(-1)}(z) = 1$, $\Delta^{(0)} = S_0$, and $\delta^{(-1)} = 1$. At each step $i < 2t - 1$, the new discrepancy $\Delta^{(i+1)}$ is computed as $\Delta^{(i+1)} = \sum_{j=0}^{\min(t,i+1)} \sigma_j^{(i)} S_{i+1-j}$. If $\Delta^{(i+1)}$ is equal to zero (this happens only with small probability), the computation is aborted, and we go for another decoding attempt. For simplifying the last step of the signature scheme, we make the error locator polynomial $\sigma^{(2t-1)}$ monic. The operation is $\sigma(z) = \sigma^{(2t-1)}(z)/\sigma_t^{(2t-1)}$, and is implemented as one inversion and eight multiplications, with the sole inversion of the body of the while loop.

Divisibility/Decodability Test. Step 3 of the signature scheme consists in checking if the error locator polynomial σ splits completely into linear factors, i.e. if σ has 9 distinct roots in $\mathbb{F}_{2^{16}}$. Since Fermat's little theorem states that $z^{2^m} - z = \prod_{a \in \mathbb{F}_{2^{16}}}(z - a)$, the polynomial σ splits over $\mathbb{F}_{2^{16}}$ iff $z^{2^{16}} \bmod \sigma(z) = z$ [2].

Assume that the four polynomials defined by $\varphi_i(z) = z^{2i} \bmod \sigma(z)$, $5 \leq i \leq 8$, are precomputed, and let p be a polynomial of degree 8. Then,

$$p(z)^2 \bmod \sigma(z) = \left(\sum_{i=0}^{8} p_i^2 z^{2i}\right) \bmod \sigma(z) = \sum_{i=5}^{8} p_i^2 \varphi_i(z) + \sum_{i=0}^{4} p_i^2 z^{2i}. \tag{4}$$

Consequently, $z^{2^{16}} \bmod \sigma(z)$ can be evaluated in 12 steps by repeated squaring, starting with $\varphi_8(z) = z^{2^4} \bmod \sigma(z)$. For the precomputation of the φ_i's, notice that for a polynomial $p(z)$ of degree 8, since $\sigma(z)$ is monic, we have:

$$zp(z) \bmod \sigma(z) = p_8\sigma(z) + zp(z). \tag{5}$$

Therefore, the four polynomials $\varphi_i(z)$ can be iteratively computed according to (5), starting from $p(z) = z^8$.

3 Composite Arithmetic in the Finite Field $\mathbb{F}_{2^{16}}$

The efficiency of the signature algorithm relies heavily on the efficiency of the underlying arithmetic in the field $\mathbb{F}_{2^{16}}$. The operation count is dominated by the number of additions and multiplications. We have followed the composite field approach of [5], whose performance is superior, in our case, to other techniques like standard base arithmetic operators. The field $\mathbb{F}_{2^m}$, $m = 16$, is built recursively from $\mathbb{F}_{2^{m/2}}$ (using a field extension of degree 2). An element $a \in \mathbb{F}_{2^m}$ is represented by an m-bit array $[a_1, a_0]$ of two elements in $\mathbb{F}_{2^{m/2}}$ [4,5].

For the architecture in Section 4 we shall use multiply-and-add operators. With above representation, the operation also is defined recursively. For a, b and c in $\mathbb{F}_{2^m}$, $d = [d_1, d_0] = a \times b + c$ is given by:

$$\begin{cases} d_1 &= (a_0 + a_1)(b_0 + b_1) + a_0 b_0 + c_1, \\ d_0 &= a_0 b_0 + a_1 b_1 \omega_{m/2} + c_0 \end{cases}$$

where $\omega_{m/2}$ is a constant in $\mathbb{F}_{2^m}$. For this work we have stopped the recursion at the level of $\mathbb{F}_{2^4}$ or $\mathbb{F}_{2^2}$, and implemented 1-input or 2-input operators using look-up tables with four address bits. We have proceeded in a similar way for the inverter, we refer to the algorithm of [5, §2.2.2] and references therein.

These choices lead to a Multiply-and-Add operator using 98 slices and to an inverter using 109 slices (including an internal pipeline stage). Both operators have a critical path of about 10ns. For a general comparison, note that a 16 bits to 32 bits integer multiplier would require about 140 slices.

4 Hardware Implementation

We now detail our architecture for the three steps of the signature while loop. We show that parallel structures for the Berlekamp-Massey and for the decodability stages can be based on similar resources. From (1) and (4) we identify the key operation $p(z) \leftarrow a \times q(z) + r(z)$ where p, q and r are three polynomials of degree 9 over $\mathbb{F}_{2^{16}}$, and where a is an element of $\mathbb{F}_{2^{16}}$. Designing a specific processor for the latter polynomial operator (which actually can be seen as a vector Axpy), leads to an efficient implementation of the whole while loop that respect our board size constraint.

4.1 Computation of a Double Syndrome

Figure 1 describes the operator computing a double syndrome from the 144-bit hashed document s and the counter i. The seed of the 25-stage linear feedback shift register (LFSR) is set to i during an initialization step. Then, at each clock cycle, the circuit performs the XOR of s_j and the jth bit of the pseudo-random sequence generated by the LFSR to obtain $s_j^{(i)}$. The jth row of the 144×288 matrix V (stored in 18 Block Select RAM memories) is then multiplied by $s_j^{(i)}$, which gives the 18 coefficients of a new double syndrome (in the 16×18 flip-flops):

$$S = \left(\sum_{j=0}^{143} s_j^{(i)} v_{0,j}\right) + \left(\sum_{j=0}^{143} s_j^{(i)} v_{1,j}\right) z + \ldots + \left(\sum_{j=0}^{143} s_j^{(i)} v_{287,j}\right) z^{17} = \sum_{i=0}^{17} S_i z^i,$$

where $S_i \in \mathbb{F}_{2^{16}}$. The double syndrome block and its control unit require 204 slices, that is to say 6.7% of the available resources of an XCV300E FPGA.

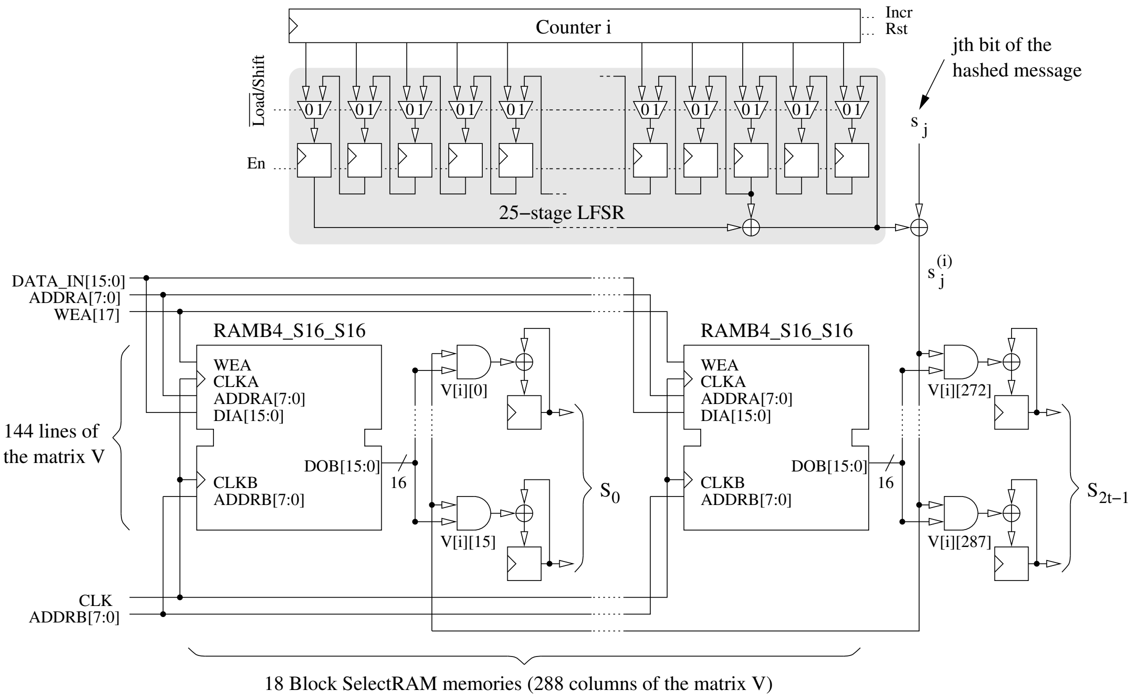

Fig. 1. Computation of the double syndrome S.

4.2 Specific Processor for Error Locating and Decodability Testing

Let us first look at the resources asked by the Berlekamp-Massey algorithm with input S and output σ. The computation (1) of $\sigma^{(i)}$ may be rewritten as follows:

$$p(z) = -\Delta^{(i)}\rho^{(i-1)}(z) = -\sum_{j=0}^{t} \Delta^{(i)}\rho_j^{(i-1)} z^j, \tag{6}$$

$$\sigma^{(i)}(z) = \delta^{(i-1)}\sigma^{(i-1)}(z) + zp(z) = \sum_{j=0}^{t}\left(\delta^{(i-1)}\sigma_j^{(i-1)} + p_{j+1}\right) z^j, \tag{7}$$

where $p \in \mathbb{F}_{2^{16}}[z]$. Thus $\sigma^{(i)}$ can be evaluated in two clock cycles by means of $t+1$ multiply-and-add units. The computation of the new discrepancy $\Delta^{(i+1)}$ requires up to $t+1$ products $\sigma_j^{(i)} S_{i+1-j}$ and a multi-operand addition. Finally, σ is obtained by multiplying the coefficients of the polynomial $\sigma^{(2t-1)}$ by the inverse of $\sigma_t^{(2t-1)}$.

This is summarized in Table 1 which provides an estimate of the corresponding circuit area on an XCV300E FPGA.

A brief study of the divisibility test indicates that its hardware implementation involves t multiply-and-add units and a few registers (see Table 2 for details). Indeed, from (5), the four polynomials φ_i, $5 \le i \le 8$, are obtained using t multiply-an-add operations in $\mathbb{F}_{2^{16}}$. The computation (4) is then realized in five steps:

$p(z) \leftarrow \sum_{i=0}^{4} p_i^2 z^{2i}$, $q_0 = p_5^2$, $q_1 = p_6^2$, $q_2 = p_7^2$, and $q_3 = p_8^2$;
for $j = 0$ to 3 **do**
  $p(z) \leftarrow q_j\varphi_{j+5}(z) + p(z)$;
end for

Table 1. Hardware resources required for the Berlekamp-Massey algorithm.

Components	Area (% of available slices)
Ten Multiply-and-add operators	980 slices (~ 32.0%)
An inverter to compute $1/\sigma_t^{(2t-1)}$	109 slices (~ 3.5%)
A multi-operand adder to compute $\Delta^{(i+1)}$	32 slices (~ 1.0%)
Four 160-bit registers to store $\sigma^{(i)}(z)$, $\rho^{(i)}(z)$, up to ten coefficients of the double syndrome, and the polynomial $p(z)$	320 slices (~ 10.4%)
Two 16-bit registers for $\delta^{(i)}$ and $\Delta^{(i)}$	16 slices (~ 0.5%)
Total:	1457 slices (~ 47.4%)

Table 2. Hardware resources required for the divisibility test.

Components	Area (% of available slices)
Nine Multiply-and-add operators	882 slices (~ 28.7%)
Five 144-bit registers to store intermediate results and the polynomials $\varphi_5(z)$, $\varphi_6(z)$, $\varphi_7(z)$, and $\varphi_8(z)$	360 slices (~ 11.7%)
Four 16-bit registers for p_5^2, p_6^2, p_7^2, and p_8^2	32 slices (~ 1.0%)
Total:	1274 slices (~ 41.4%)

According to our area estimates (Tables 1 and 2), an XCV300E FPGA does not contain enough slices for implementing a double syndrome block, a Berlekamp-Massey block, a divisibility test block, and their respective control units. However, the above analysis shows that the Berlekamp-Massey algorithm and the divisibility test can share the same hardware resources. Our solution is the design of a specific processor for programming these algorithms (Figure 2). It consists of ten multiply-and-add operators, a multi-operand adder, an inverter, six 160-bit registers, and four 16-bit registers (Table 3). We have seen the arithmetic units in Section 3, and we only describe the main features of the registers:

– If inputs A and B of the multiply-and-add block are equal to $p(z)$, then the SQR unit provides a routing mechanism for storing the polynomial $\sum_{i=0}^{4} p_i^2 z^{2i}$ of degree 8 in R_0.
– The APPEND unit updates the double syndrome sequence involved in the computation of the discrepancy as follows: $R_1 \leftarrow (zR_1 + S_i) \bmod z^{2t}$.
– For reducing the size of the multiplexer selecting the A input of the multiply-and-add block, R_2, R_3, R_4, and R_5 form a FIFO. The feedback mechanism allows to copy R_4 or R_5, and to shift R_4 (multiplication by z). For instance this allows to update $\rho^{(i)}(z)$ according to (2). A more detailed example is given in Appendix A.

If we apply loop unrolling, no conditional instructions are needed in Berlekamp-Massey algorithm and the divisibility test. Furthermore, each instruction simply consists of the 28 control bits described on Figure 2. This allows the design of a small control unit which increments the program counter, checks if $\Delta^{(i+1)} = 0$, and returns $z^{2^{16}} \bmod \sigma(z)$. The Berlekamp-Massey algorithm and the divisibility test respectively require 74 and 72 instructions (see Appendix A for more details). Our specific processor, including the program ROM, requires 2209 slices (72% of the slices available in an XCV300E FPGA) and allows to implement the signature scheme on our target FPGA.

5 Results and Analysis

A VHDL description of the signature scheme has synthesized by XST 5.2.03i and placed-and-routed for an XCV300E-7 FPGA. 2593 slices (88% of the available resources) and 18 Block SelectRAM memories implement this first prototype which runs at $f = 62$ MHz.

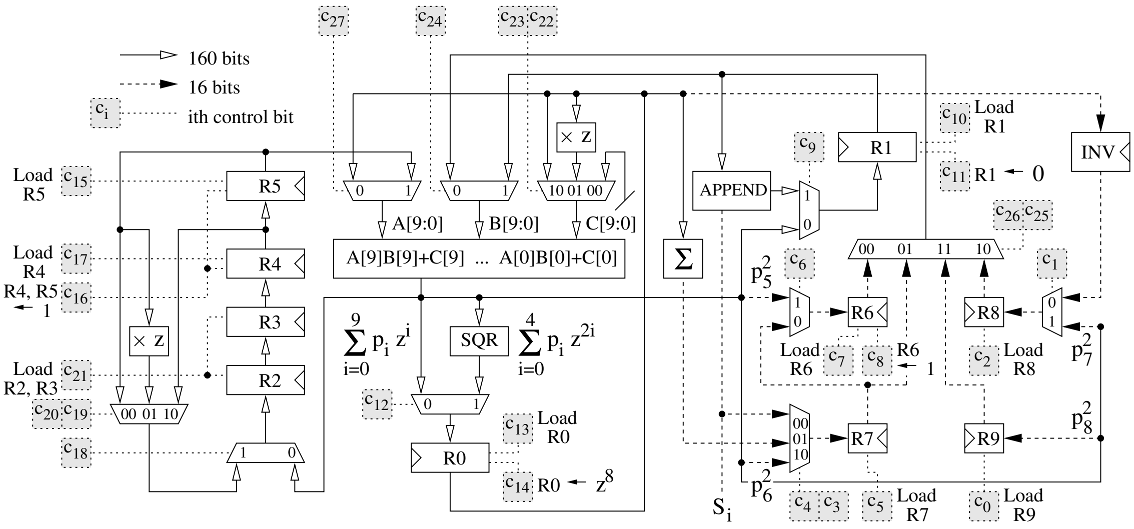

Fig. 2. Application-specific processor for error locating and decodability testing.

Table 3. Register assignment for the Berlekamp-Massey and the divisibility test.

	Berlekamp-Massey	**Divisibility test**
R_0	Intermediate results	
R_1	Double syndrome sequence	Copy of R_0 (for squaring)
R_2 to R_5	$\sigma^{(i)}(z)$ and $\rho^{(i)}(z)$	$\sigma(z)$, then $\varphi_5(z)$, $\varphi_6(z)$, $\varphi_7(z)$, and $\varphi_8(z)$
R_6	$\delta^{(i)}$	p_5^2
R_7	$\Delta^{(i)}$	p_6^2
R_8	–	p_7^2
R_9	–	p_8^2

The computation of the first double syndrome involves 145 clock cycles (an initialization step and 144 computation steps). The Berlekamp-Massey algorithm and the divisibility test require 148 additional clock cycles (146 instructions and 2 clock cycles devoted to control tasks), a new double syndrome is evaluated in parallel. Hence the signature time is about (on the average with a very small standard deviation):

$$T_{\text{signature}} = \frac{1}{f} \cdot (145 + 148 \cdot 9!) \approx 0.86 \text{ second},$$

whereas a software implementation requires more than one minute to achieve the same task on a 2GHz Pentium 4 processor.

Though our circuit outperforms software implementations, future studies should improve its efficiency. In particular, the arithmetic operators of the specific processor are not fully exploited. Among the 146 instructions executed, there are 17 multi-operand additions ($\sim$ 11.6% of the total number of instructions), a single inversion ($\sim$ 0.7%), and 122 multiply-and-add operations ($\sim$ 83%). Furthermore, the tenth multiply-and-add unit is only involved in the very last steps of the Berlekamp-Massey algorithm. Further theoretical and architectural studies should therefore allow to shorten the signature time.

References

1. E.R. Berlekamp. *Algebraic Coding Theory.* Second Edition, Aegean Park Press, Laguna Hills, California, 1984.

2. N. Courtois, M. Finiasz, and N. Sendrier. How to achieve a McEliece-based digital signature scheme. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, number 2248 in Lecture Notes in Computer Science, pages 157–174. Springer, 2001.
3. J.L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. Inform. Theory*, IT-15:122–127, 1969.
4. C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.
5. C. Paar and M. Rosner. Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware. In *5th IEEE Symposium on FPGA-based Custom Computing Machines (FFCM'97), Napa Valley, CA*, April 1997.

A Implementation of the Berlekamp-Massey Algorithm

This Appendix describes the instructions which allow to implement the Berlekamp-Massey algorithm on our specific processor (Table 4), and provides a coding example:

- Instruction I_0 is responsible for the initialization step of the Berlekamp-Massey algorithm.
- We define two instructions I_1 and I_2 to compute $\sigma^{(i)}(z)$, $\rho^{(i)}(z)$, and $\delta^{(i)}$ when i is even. I_1 implements Equations (2) and (6), and shifts the FIFO so that the A input of the multiply-and-add block is $\sigma^{(i-1)}(z)$. Then, I_2 evaluates in parallel the ith step error locator polynomial and $\delta^{(i)}$ (Equations (3) and (6)), shifts the FIFO, and updates the double syndrome sequence. Instructions I_3 and I_4 perform the same task when i is odd.
- Finally, instructions I_5 and I_6 compute the new discrepancy according to the algorithm described in Section 4.2, and shift the FIFO so that R_5 and R_4 respectively store $\rho^{(i)}(z)$ and $\sigma^{(i)}(z)$. This state of the FIFO allows to start the computation of the $(i+1)$th step error locator polynomial.

Table 5 illustrates the programming of the Berlekamp-Massey Algorithm. Instruction I_0 properly initializes the registers. Then, four instructions implement the ith step of the algorithm. Note that it is useless to compute a new discrepancy when $i = 2t - 1$ (see Section 2). Consequently, the algorithm requires $1 + (2t - 1) \cdot 4 + 2 = 71$ instructions. Three clock cycles are then required to make $\sigma(z)$ monic (an inversion and a multiplication), which gives the total of 74 clock cycles mentioned in Section 4.2.

Table 4. Operations involved in the Berlekamp-Massey Algorithm.

	Mathematical description	**Instruction**	**Comment**
I_0	$\rho^{(-1)}(z) = 1$	$R_5 \leftarrow 1$	$R_5 = \rho^{(-1)}(z)$
	$\sigma^{(-1)}(z) = 1$	$R_4 \leftarrow 1$	$R_4 = \sigma^{(-1)}(z)$
	$\Delta^{(0)} = S_0$	$R_1, R_7 \leftarrow S_0$	$R_7 = \Delta^{(0)}$ and R_1 stores the current double syndrome sequence
	$\delta^{(-1)} = 1$	$R_6 \leftarrow 1$	$R_6 = \delta^{(-1)}$
I_1	$p(z) \leftarrow -\Delta^{(i)}\rho^{(i-1)}(z)$	$R_0 \leftarrow R_7R_5$	
	$\rho^{(i)}(z) \leftarrow \sigma^{(i-1)}(z)$	$R_2 \leftarrow R_4$	$R_2 = \rho^{(i)}(z) = \sigma^{(i-1)}(z)$
		$R_5 \leftarrow R_4$	$R_5 = \sigma^{(i-1)}(z)$
I_2	$\sigma^{(i)}(z) \leftarrow \delta^{(i-1)}\sigma^{(i-1)}(z) + zp(z)$	$R_0, R_2 \leftarrow R_6R_5 + zR_0$	$R_2 = \sigma^{(i)}(z)$
		$R_3 \leftarrow R_2$	$R_3 = \rho^{(i)}(z)$
	$\delta^{(i)} \leftarrow \Delta^{(i)}$	$R_6 \leftarrow R_7$	$R_6 = \delta^{(i)}$
		$R_1 \leftarrow zR_1 + S_{i+1}$	New syndrome sequence
I_3	$p(z) \leftarrow -\Delta^{(i)}\rho^{(i-1)}(z)$	$R_0 \leftarrow R_7R_5$	
	$\rho^{(i)}(z) \leftarrow z\rho^{(i-1)}(z)$	$R_2 \leftarrow zR_5$	$R_2 = \rho^{(i)}(z) = z\rho^{(i-1)}(z)$
		$R_5 \leftarrow R_4$	$R_5 = \sigma^{(i-1)}(z)$
I_4	$\sigma^{(i)}(z) \leftarrow \delta^{(i-1)}\sigma^{(i-1)}(z) + zp(z)$	$R_0, R_2 \leftarrow R_6R_5 + zR_0$	$R_2 = \sigma^{(i)}(z)$
		$R_3 \leftarrow R_2$	$R_3 = \rho^{(i)}(z)$
		$R_1 \leftarrow zR_1 + S_{i+1}$	New syndrome sequence
I_5	$p(z) \leftarrow \sum_{j=0}^{\min(t,i+1)} \sigma_j^{(i)} S_{i+1-j} z^j$	$R_0 \leftarrow R_0R_1$	
		$R_4 \leftarrow R_3$	$R_4 = \rho^{(i)}(z)$
		$R_3 \leftarrow R_2$	$R_3 = \sigma^{(i)}(z)$
I_6	$\Delta^{(i+1)} \leftarrow \sum_{j=0}^{\min(t,i+1)} p_j$	$R_7 \leftarrow \sum R_0$	$R_7 = \Delta^{(i+1)}$
		$R_5 \leftarrow R_4$	$R_5 = \rho^{(i)}(z)$
		$R_4 \leftarrow R_3$	$R_4 = \sigma^{(i)}(z)$

Table 5. First steps of the Berlekamp-Massey algorithm.

	I_0	I_1	I_2	I_5	I_6	I_3
R_0	–	$\Delta^{(0)}\rho^{(-1)}(z)$	$\sigma^{(0)}(z)$	$\sigma_1^{(0)}S_0z + \sigma_0^{(0)}S_1$	–	$\Delta^{(1)}\rho^{(0)}(z)$
R_1	S_0		$S_0z + S_1$			
R_2	–	$\rho^{(0)}(z) = \sigma^{(0)}(z)$	$\sigma^{(0)}(z)$	–	–	$\rho^{(1)}(z) = z\rho^{(0)}(z)$
R_3	–	–	$\rho^{(0)}(z)$	$\sigma^{(0)}(z)$	–	–
R_4	$\sigma^{(-1)}(z)$	–	–	$\rho^{(0)}(z)$	$\sigma^{(0)}(z)$	–
R_5	$\rho^{(-1)}(z)$	$\sigma^{(-1)}(z)$	–	–	$\rho^{(0)}(z)$	$\sigma^{(0)}(z)$
R_6	$\delta^{(-1)}$		$\delta^{(0)}$			
R_7	$\Delta^{(0)}$				$\Delta^{(1)}$	