Small FPGA based Multiplication-Inversion Unit for Normal Basis Representation in $GF(2^m)$

Jérémy Métairie, Arnaud Tisserand, Emmanuel Casseau

▶ To cite this version:

Jérémy Métairie, Arnaud Tisserand, Emmanuel Casseau. Small FPGA based Multiplication-Inversion Unit for Normal Basis Representation in $GF(2^m)$. ISVLSI: IEEE Computer Society Annual Symposium on VLSI, Jul 2015, Montpellier, France. hal-01175712

HAL Id: hal-01175712
https://inria.hal.science/hal-01175712

Submitted on 16 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Small FPGA based Multiplication-Inversion Unit for Normal Basis Representation in GF(2^m)

Jérémy Métairie, Arnaud Tisserand, *Senior Member, IEEE,* and Emmanuel Casseau

***Abstract*—Halving methods have been proposed for parallel implementation of ECC primitives on multicore processors. In hardware, they can also provide protection against some side channel attacks (thanks to parallel independent operations). But they require affine coordinates for curve points and costly inversions. We propose a new combined multiplication-inversion unit for binary field extensions and halving based ECC methods optimized for FPGAs. We target small area solutions compared to very fast but costly ones from state-of-art. Our solution is based on permuted normal basis, Massey-Omura multiplication and Itoh-Tsujii inversion algorithms. Our FPGA implementations show better efficiency for large fields.**

***Index Terms*—finite-field arithmetic, binary fields, normal basis, multiplication, inversion, ECC, halving scalar multiplication**

I. Introduction

FINITE fields [1] are widely used in cryptography. Efficient arithmetic over finite fields is a key element for implementing public-key cryptosystems. Binary field extensions GF(2^m) are widely used in hardware implementations of elliptic curve cryptography (ECC [2]) due to their higher speed and smaller silicon area compared to GF(p) based solutions.

Scalar multiplication using *point halving* has been proposed in [3] to provide more parallelism. There, operations can be divided into two independent sequences and performed in parallel [4]. See [5] for recent work in this domain. This can also provide higher resilience against side channel attacks (see [6] for instance) since two shorter and independent scalar multiplications are performed in parallel. This requires the curve points to be represented using affine coordinates. Then, more inversion operations have to be computed than in projective coordinates [2, Sec. 3.2]. Representing field elements using normal basis will also provide very efficient square and square-root operations (*i.e.* circular shifts) which are also important for halving methods.

In this paper, we present a combined multiplication-inversion unit (MIU). State of art GF(2^m) inversion and related operations are recalled in Sec. II. Sec. III presents our GF(2^m) inversion algorithm. It uses a new representation for field elements called *permuted normal basis*, a modified Massey-Omura multiplication [7] which allows efficient inversion through Itoh-Tsujii algorithm [8]. Our proposed hardware unit presents a higher internal parallelism without the large area penalty used in state-of-art solutions. We target small solutions instead of faster but very costly operators since ECC primitives for high security levels (*e.g.* $m = 571$) may not be used frequently. It offers a new speed-area trade-off in arithmetic units with higher throughput for halving based ECC accelerators. We evaluated our solution using GF(2^m) fields recommended by NIST [9]. Our MIU architecture and its implementation results are presented in Sec. IV. Our implementations have been performed on a Spartan-6 LX75T and Virtex-4 LX100 FPGAs for comparison purpose. Finally, Sec. V concludes the paper.

J. Métairie and A. Tisserand are with CNRS, IRISA, University Rennes 1, INRIA in Lannion France.

E. Casseau is with University Rennes 1, IRISA, INRIA in Lannion France.

II. State of the Art

A. Representations of GF(2^m) Elements

There exist two popular representations: *normal basis* and *polynomial basis* [1]. In normal basis, $A \in$ GF(2^m) is encoded as $\sum_{i=0}^{m-1} a_i \beta^{2^i}$ where coefficients a_i belong to GF(2) and β is a special element. $A = [a_0, a_1, \ldots, a_{m-1}]$ denotes the vector of coefficients. Square and square-root operations can be easily computed using circular shifts on the vector of coefficients. In this work, we use normal basis.

In polynomial basis, A is encoded as $\sum_{i=0}^{m-1} a'_i x^i$ where coefficients $a'_i \in$ GF(2). Additions and multiplications are polynomial operations modulo the irreducible polynomial f in GF$(2)[x]$. There exist other representations with particular properties such as Dickson basis [10] or dual basis [11].

B. GF(2^m) Addition and Multiplication

GF(2^m) addition can be efficiently performed using parallel `XOR` gates on all coefficients without any carry propagation. In normal basis, most of multiplication algorithms are based on matrix-vector product formulations of the basic multiplication. This leads to larger multipliers compared to polynomial basis ones. Massey and Omura (MO) proposed a multiplication method (see Algo. 1 from [7]), which computes $P = A \times B$, based on matrix-vector product where the constant matrix M_0 is only composed of GF(2) elements. Notation $\texttt{ROL}(x, n)$ (`ROR`) stands for n-bit left (right) circular shift of x. Notation $P[i]$ is the i-th bit of P (starting with LSBs). At line 3 in Algo. 1, M_0 is a $m \times m$ binary, constant, symmetric and very sparse matrix. Then multiplying by M_0 just uses `XOR` trees. A serial-output MO multiplier requires a block of multiplication by M_0 (noted $\times M_0$), two `ROL` registers and a dot product operator. A last `ROL` register can be used to provide a parallel output if needed.

We use Gaussian normal basis (GNB, a specific type of normal basis) due to its very low hardware complexity for the implementation of multiplication by M_0 [12]. The number of “1” in M_0 is given by $C_m \leq t \times m - t + 1$ where t is the

Algorithm 1: Original Massey-Omura multiplication [7].

Operands: A, B in GF(2^m) represented in normal basis
Result: $P = A \times B$
1 $P \leftarrow 0$
2 **for** i **from** 0 **to** $m-1$ **do**
3 $\quad P[0] \leftarrow A \times M_0 \times B^T$
4 $\quad A \leftarrow \mathtt{ROL}(A,1)$; $B \leftarrow \mathtt{ROL}(B,1)$; $P \leftarrow \mathtt{ROL}(P,1)$
5 **return** P

TABLE I
FPGA AREA EVALUATION OF THE BLOCK "MULTIPLICATION BY M_0" (SPARTAN-6 LX75T).

m / t	163 / 4	233 / 2	283 / 6	409 / 4	571 /10
C_m /#LUT	645 / 159	465 / 232	1677 / 282	1629 / 408	5637 / 1128

field type (the smallest integer such that $p = m \times t + 1$ is prime and $\gcd(\frac{tm}{k}, m) = 1$ where k is the multiplicative order of $2 \bmod p$). The number of XOR gates in the $\times M_0$ block is $C_m - m$ (see Tab. I for a FPGA area evaluation).

The original Massey-Omura multiplier produces one result bit every clock cycle. Using two parallel $\times M_0$ blocks produces two result bits per clock cycle with an important area overhead. For instance, multiplication in GF(2^{233}) only requires $117 = \lceil 233/2 \rceil$ clock cycles with an area overhead close to 2.

In [13] some redundancies in d parallel copies of the $\times M_0$ block are used to provide multiple bits of the result at each clock cycle without the full area penalty. The number of XOR gates is then $d\big(t(m-(d+1)/2)+(d-1)/2\big)$.

Parallel-output multipliers have been proposed in normal basis [14], [15]. P is the final accumulation of the partial products which is available after m clock cycles. In [16], a serial-input/parallel-output multiplier using w-bit sub-words computes the result in $\lceil m/w \rceil$ clock cycles.

C. GF(2^m) Inversion Algorithms

Two main methods are used for GF(2^m) inversion: *Euclidean algorithm* and *Fermat's little theorem* (FLT). Up to now, Euclide based solutions are rarely used in hardware [17]. FLT states that $A^{2^m-1} = 1$, then $A^{-1} = A^{2^m-2}$. Thus, an exponentiation can compute the inverse of $A \in$ GF(2^m)* using the standard *square and multiply* algorithm.

Itoh and Tsujii (IT) proposed in [8] an efficient way to perform this exponentiation noting that $2^m - 2 = (111\cdots110)_2$ and using addition chains. An addition chain is a sequence of additions where all operands are selected among the previously computed terms [18, Sec. 4.6.3]. Each term is the Hamming weight of the exponent written in the binary representation. Tab. II provides an example in GF(2^7) with both the addition chain and the corresponding sequence of multiplication and square operations (where T_i is the i-th intermediate product). In practice, the efficiency of IT based inversion mostly relies on the multiplier efficiency.

TABLE II
INVERSION EXAMPLE OF A IN GF(2^7) USING IT ALGORITHM.

$T_0 = A^{(1)_2}$	$u_0 = 1$
$T_1 = T_0^2 \times T_0 = A^{(10)_2} \times A^{(1)_2} = A^{(11)_2}$	$u_1 = u_0 + u_0$
$T_2 = T_1^2 \times T_0 = A^{(110)_2} \times A^{(1)_2} = A^{(111)_2}$	$u_2 = u_1 + u_0$
$T_3 = T_2^{2^3} \times T_2 = A^{(111000)_2} \times A^{(111)_2} = A^{(111111)_2}$	$u_3 = u_2 + u_2$
$A^{-1} = T_3^2 = A^{(1111110)_2}$	

Recently in [19], a new IT inversion operator has been proposed using the hybrid multiplier from [16] which performs $A \times B \times C$ in $\lceil m/w \rceil + 1$ clock cycles with w-bit sub-words.

In [20], the authors generalize the concept of addition chains introducing the k-addition chains. They use such a tool to improve the work from [19]. Very recently in [21], a parallel-IT algorithm has been proposed to speed-up the IT-sequence using a second multiplier.

III. PROPOSED SOLUTION

In ECC halving based scalar multiplication, there is an inversion for each scalar bit equal to "1": on average every 0.5 key bit without key recoding or every 0.33 key bit using non-adjacent form (NAF) recoding, see [2, Sec. 3.3]. Then a standalone inversion unit would lead to underused silicon. We designed a new *combined multiplication-inversion unit* (MIU) to support both multiplication and inversion GF(2^m) operations in GNB. It uses the IT method, then only multiplications and squares are required for field inversion. As squaring is very cheap in GNB (*i.e.* just a ROR), implementing a MIU requires a field multiplier, a ROR, a local register and a small controller. Our MIU has two operation modes: i) standard GF(2^m) multiplication where operands A and B are multiplied to produce $P = A \times B$; ii) IT based GF(2^m) inversion where multiplier and ROR blocks are used to compute the multiplications and squares (with intermediate values in the local register). We improved the inversion speed by using optimized multiplications for specific terms of the IT exponentiation. Our MIU produces two result bits per clock cycle with only one MO multiplication block. The obtained speed-up is about 20 % and not 100 % for two reasons: first, not all terms in the IT exponentiation sequence have this specific form; second, some of result bits are produced several times.

The GF(2^m) operand(s) and result of our MIU, for $P = A \times B$ or $R = A^{-1}$ operations, are represented using a variation of GNB with permuted coefficients proposed in Sec. III-C. Some values are stored using w-bit sub-words.

A. Optimizing Large Shifter in IT Exponentiation on FPGA

During the IT exponentiation [8], the intermediate products in GF(2^m) must be shifted by several shift amounts (*e.g.* for $m = 571$ there are 13 different shift amounts). This requires a very large m-bit barrel shifter ($m \in [163, 571]$ bits in ECC). As an example, for GF(2^{571}) on a Spartan 6 FPGA, a 571-bit shifter (optimized for the 13 shift amounts) requires 3425 LUTs while a Massey-Omura multiplier requires 2246 LUTs. Furthermore, the shift amounts are different for each field size m, then one must use a specific shifter for each m.

We replace all these large shifters by one hardwired block RAM (BRAM) of the FPGA. Instead of storing each bit of

Algorithm 2: w-bit words Massey-Omura multiplication.

Operands: $A, B \in \text{GF}(2^m)$
Result: $P = A \times B$
1 $G \leftarrow 0$
2 **for** i **from** 0 **to** $m + w - 2$ **do**
3 | $G \leftarrow \texttt{SHL}(G, 1)$; $G[w-1] \leftarrow A \times M_0 \times B^T$
4 | **if** $i \geq (w-1)$ **then**
5 | | $P[i - w + 1] \leftarrow G$ (*write in BRAM*)
6 | $A \leftarrow \texttt{ROL}(A, 1)$; $B \leftarrow \texttt{ROL}(B, 1)$
7 **return** P

the intermediate m-bit product P in a register, we duplicate them w times in the BRAM using the following patterns: $[p_0, p_1, \ldots, p_{w-1}]$ at address @$=0$; $[p_1, p_2, \ldots, p_w]$ at @$=1$; $[p_2, p_3, \ldots, p_{w+1}]$; $\ldots$; $[p_{m-1}, p_0, \ldots, p_{w-2}]$ at @ $= m - 1$. BRAMs in recent FPGAs are large enough to support the $m \cdot w$ bits. For instance, with $w = 32$ (a typical width for BRAMs), 7456 bits are required for $m = 233$ and 18272 bits for $m = 571$. In a low-cost Spartan-6 FPGA, 18Kb BRAMs are available, then for the largest field $m = 571$, one needs two BRAMs. The MO multiplier is a sequential operator with m clock cycles for producing each intermediate P. We use a w-bit temporary register to store $[p_i, p_{i+1}, \ldots, p_{i+w-1}]$ to feed the BRAM, and this register is shifted by 1 bit for the next cycle. The total cycle count for each product in the IT sequence is $m+w$. Using our BRAM based solution, shifting is performed by reading the words at addresses $(i + \alpha w) \bmod m$ for $\alpha \in \{0, 1, 2, \ldots, \lfloor m/w \rfloor\}$, where i is the shift amount. In Algo. 2, we adapt the MO multiplication algorithm to our BRAM based shifting solution (where SHL is a left shift).

This method seems to be equivalent to a simple shift register iteratively used over m clock cycles. But it will allow us to provide, for all m parameters, two shifted values for two different shift amounts at each clock cycle using a dual-port BRAM (DP-BRAM).

B. Support of Specific Patterns in the Exponentiation Chain

In the Itoh-Tsujii exponentiation sequence, the specific multiplication pattern (SMP) $A^{2^k} \times A$ frequently appears. It corresponds to the term $U_i = U_j + U_j$ in the addition chain. We modify the MO multiplication algorithm, and operator, to support both standard multiplications $A \times B$ as well as optimized SMPs. During the SMP computation, we notice that $\texttt{ROL}(A, -i) \times M_0$ and $M_0 \times \texttt{ROL}(A, -i)^T$ are computed. Since these two values are equal (one is the transpose of the other), we save one matrix-vector multiplication for each iteration of the algorithm. At every clock cycle, we compute $V = M_0 \times \texttt{ROL}(A, -i)^T$ and return $(p_i = \texttt{ROL}(A, k - i) \times V, p_{i+k} = \texttt{ROL}(A, -i - k) \times V)$. For instance, if $k = 1$ then the outputs will be at CC(0): (p_0, p_1); CC(1): (p_1, p_2); $\ldots$; CC($m - 2$): (p_{m-2}, p_{m-1}) where CC(r) is clock cycle number r. We produce the output bits serially, like the original MO, but 2 bits at each clock cycle. This allows to efficiently overlap successive computations as in [19]. Producing most

Algorithm 3: Massey-Omura Multiplication in PNB.

Operands: $A, B \in \text{GF}(2^m)$ in PNB, and k
Result: $P = A \times B$
1 $C \leftarrow \texttt{ROL}(A, -k)$; $G \leftarrow 0$; $H \leftarrow 0$; $j \leftarrow k \cdot \theta^{-1} \bmod m$
2 **for** i **from** 0 **to** $(w + N(k) - 2)$ **do**
3 | $G \leftarrow \texttt{SHL}(G, 1)$; $H \leftarrow \texttt{SHL}(H, 1)$
4 | $V \leftarrow A \times M_0'$
5 | $G[w-1] \leftarrow V \times B^T$; $H[w-1] \leftarrow V \times C^T$
6 | **if** $i \geq (w-1)$ **then**
7 | | $P[i - w + 1] \leftarrow G$ (*write in DP-BRAM*)
8 | | $P[j + i - w + 1] \leftarrow H$ (*write in DP-BRAM*)
9 | $A \leftarrow \texttt{ROL}(A, 1)$; $B \leftarrow \texttt{ROL}(B, 1)$; $C \leftarrow \texttt{ROL}(C, 1)$
10 **return** P

of the result bits twice will not be a problem using the trick presented in the next subsection.

C. Multiplication and Inversion using Permuted Normal Basis

During the computation of a SMP in the IT sequence, using the modified MO from Sec. III-B leads to some redundancies in the produced bits. Using the example from last paragraph of Sec. III-B, a SMP with $k = 1$ produces $2m - 2$ bits instead of m for the result of $A^{2^k} \times A$ (redundancy ≈ 2). This is due to the constant shift amount equal to 1 used in the MO algorithm.

We propose a modified MO algorithm with a new constant shift amount $\theta > 1$ leading to a lower overall redundancy level during a complete IT exponentiation. We wrote a Python program to find θ which minimizes the inversion time for a given field GF(2^m). We compute an addition chain for m using the basic binary method: $U_i = U_{i-1} + U_{i-1}$ or $U_i = U_{i-1} + U_0$ with $U_0 = 1$. Then our program exhaustively tests the m possible shift amounts. It returns θ leading to the smallest number of clock cycles for the whole inversion based on all SMPs $A^{2^k} \times A$ in the IT sequence.

Addition chains produced by the basic binary method are not always the shortest ones but they lead to a smaller number of intermediate m-bit registers. In practice, all our chains are the shortest or at most 1-term longer than the shortest chains from [22].

We introduce the *permuted normal basis* (PNB) representation where element $A = [a_0, a_1, a_2, \ldots, a_{m-1}]$ is represented by $A' = [a_0, a_\theta, a_{2\theta \bmod m}, \ldots, a_{(m-1)\theta \bmod m}]$. Our adaptation of the MO multiplication to PNB is detailed at Algo. 3 where $N(k)$ denotes the clock cycle count in SMP $A^{2^k} \times A$.

In our modified Algo. 3, matrix M_0' is the adapted matrix M_0 for PNB representation. M_0' is also symmetric and is a permutation of the rows and columns of M_0. Row i in M_0 is at row $i\theta^{-1} \bmod m$ in M_0' and column j in M_0 is at column $j\theta^{-1} \bmod m$ in M_0' (where θ^{-1} is the modular inverse of $\theta \bmod m$). An example is given in Fig. 1 for GF(2^7).

Our algorithm produces the two result bits $(p_{i\theta}, p_{i\theta+k})$ at every clock cycle. Since m is prime, the indexes $i\theta \bmod m$ generate GF(m) seen as an additive group. Consequently, there exists for each $i < m$ an integer $j < m$ such that $(i\theta + k) \bmod m = j\theta \bmod m$. We sequentially write the result bits into two w-bit words in a DP-

BRAM with at $@ = i, [p_{i\theta}, p_{(i+1)\theta}, \ldots, p_{(i+w-1)\theta}]$ and $@ = j, [p_{j\theta}, p_{(j+1)\theta}, \ldots, p_{(j+w-1)\theta}]$ (all the indexes are computed modulo m).

Let us present a complete example for the toy field GF(2^7). $m = 7$ leads to the chain $(U) = (1, 2, 3, 6)$ (see Tab. II). In this chain, there are two SMPs: $k = 1$ (for $2 = 1+1$) and $k = 3$ (for $6 = 3+3$). Our program returns $\theta = 2$. For SMP $k = 1$, the cycles are: CC(0): $(\underline{p_0}, p_1)$; CC(1): (p_2, p_3); CC(2): (p_4, p_5); CC(3): $(p_6, \underline{p_0})$ and $N(1) = 4$ (redundancies are underlined). For SMP $k = 3$, the cycles are: CC(0): $(\underline{p_0}, p_3)$; CC(1): $(\underline{p_2}, p_5)$; CC(2): $(\underline{p_4}, \underline{p_0})$; CC(3): $(p_6, \underline{p_2})$; CC(4): $(p_1, \underline{p_4})$; and $N(3) = 5$. Using our PNB MO multiplication, a GF(2^7) inversion requires $4 + 7 + 5 = 16$ clock cycles instead of $7 + 7 + 7 = 21$ for the classical MO algorithm.

For the cryptographic field GF(2^{163}), the addition chain is $(U) = (1, 2, 4, 5, 10, 20, 40, 80, 81, 162)$ and our program returns $\theta = 72$. For this field, the SMPs lead to $N(1) = 120$, $N(2) = 86$, $N(5) = 111$, $N(10) = 104$, $N(20) = 118$, $N(40) = 90$ and $N(81) = 103$.

D. Cost Estimation

We denote N_{SMP} the number of SMPs in the IT sequence. We compute the number of clock cycles in an inversion as $C_{\text{inv}} = C_{\text{IO}} + C_{\text{SMP}} + C_{\text{nonSMP}}$ where:

- $C_{\text{IO}} = 2\lceil \frac{m}{w} \rceil$ is time for reading the operand A and writing the result $R = A^{-1}$;
- $C_{\text{SMP}} = D + 2N_{\text{SMP}}\lceil \frac{m}{w} \rceil$, with D the computation time in the SMPs evaluated by our program and reported in Tab. III, and the second term the duration of internal memory transfers;
- $C_{\text{nonSMP}} = (|(U)| - N_{\text{SMP}}) \cdot (m + \lceil \frac{m}{w} \rceil)$ for computations and internal memory transfers of non-SMP terms in the IT sequence ($|(U)|$ is the length of chain (U)).

In Tab. III, we report θ parameters, the estimated inversion time (in clock cycles) of both original and PNB algorithms for finite fields recommended by NIST [9] and parameter $w = 32$ (selected for our target FPGAs, see Sec. IV). For the largest fields, our Python program generates θ in less than 5 minutes on a mid-range computer (3 GHz Xeon processor with 8 GB).

Using PNB, IT-based inversion and modified MO multiplication from Algo. 3, we obtain 20 % faster GF(2^m) inversions compared to the original IT [8]. For PNB GF(2^m) multiplication, our MIU is as fast as the original MO with a small area overhead. Addition in GF(2^m) has exactly the same time and area costs in both representations (GNB and PNB). Squaring in PNB GF(2^m) is also a constant shift but where the shift amount is θ instead of 1. Conversion between PNB and GNB (for both directions) is a permutation of the coefficients. In parallel architectures this can be done using routing. In sequential implementations with several chunks per field element, this can be done using several registers accesses and routing. In practice, no conversion is required during the scalar multiplication.

$$M_0 = \begin{pmatrix} 0&1&0&0&0&0&0 \\ 1&0&1&0&0&1&1 \\ 0&1&0&1&1&1&0 \\ 0&0&1&0&0&1&0 \\ 0&0&1&0&0&0&1 \\ 0&1&1&1&0&0&1 \\ 0&1&0&0&1&1&1 \end{pmatrix} \quad \Rightarrow \quad M_0' = \begin{pmatrix} 0&0&0&0&0&1&0 \\ 0&0&0&1&1&0&0 \\ 0&0&1&0&1&1&1 \\ 0&1&0&0&1&1&1 \\ 0&1&1&1&0&1&0 \\ 1&0&1&1&1&0&0 \\ 0&0&1&1&0&0&0 \end{pmatrix}$$

Fig. 1. M_0 and the corresponding M_0' in PNB for GF(2^7) and $\theta = 3$.

TABLE III
INVERSION DETAILS AND COMPARISON FOR NIST FIELDS.

m	θ	D	inversion time		speed-up
			original	PNB	
163	72	732	1702	1335	$\approx 21\%$
233	36	1046	2667	2082	$\approx 21\%$
283	28	1431	3522	2761	$\approx 21\%$
409	35	2263	5090	4185	$\approx 17\%$
571	171	3221	8282	5973	$\approx 27\%$

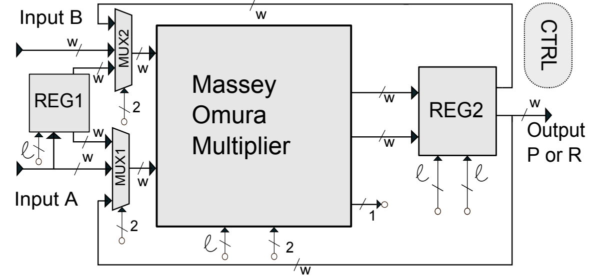

Fig. 2. Architecture of our multiplication-inversion unit (MIU).

IV. Implementation Details and Results on FPGAs

The architecture of our combined multiplication-inversion unit (MIU) is presented at Fig. 2 where $\ell = \lceil \log_2 m \rceil$ is the address size for field elements at bit level. Our MIU supports both SMP $A^{2^k} \times A$ and standard multiplications $A \times B$. In the MIU, the internal address computations (addition and reduction modulo m) are pipelined to reach higher frequencies. Its main computation block is our modified MO multiplier for PNB representation from Algo. 3 and illustrated at Fig. 3. This block requires two dot product operators (the central rectangles in Fig. 3) instead of one (in the original MO multiplier) as well as three m-bit shift registers instead of two. It also requires two small w-bit registers for temporary transfers to the BRAM. In Fig. 2, the m-bit register REG1 stores A during the complete IT sequence when computing A^{-1}. REG2, in BRAM, stores all intermediate/final products of the IT sequence. We used $w = 32$ to fit actual BRAM sizes in our target FPGAs. Our MIU operator is scalable for larger w (for other targets). Our solution is optimized for FPGAs with BRAMs. In case of ASIC implementation, we need to study an appropriate architecture.

All internal operations and transfers in our MIU are scheduled by the high-level finite states machine (FSM) presented at Fig. 4. It schedules the complete IT algorithm with:

- Initial state `LOAD_OP` loads the operand(s) A in PNB GF(2^m) for inversion (and possibly B for multiplication) sequentially using w-bit sub-words;
- `MULT` starts a standard MO multiplication $A \times B$;
- `DONE` is reached when the computation is finished (and the user can serially read the result);
- `REG×REG` starts the computation of the first product $A \times A$ in the IT sequence using the modified MO multiplier

from Algo. 3. The result A^2 is stored in the BRAM using the redundant representation from Sec. III-A.
- `BRAM×BRAM` starts the computation of a SMP $A^{2^k} \times A$;
- `REG×BRAM` starts the computation of a non-SMP multiplication $A \times B$;
- `WAIT` is reached at the end of one step in the IT sequence;
- `NEXT` determines what is the next step in the IT sequence (or the end of the computation).

Tab. IV and Tab. V report implementation results of GF(2^m) inversions using our MIU on Spartan-6 LX75T and Virtex-4 LX100 FPGAs respectively. Three inversion algorithms have been fully implemented and validated: MO1 uses a single $\times M_0$ block with the original MO multiplier from [7]; RM2 uses two $\times M_0$ blocks with the modified multiplier proposed in [13] (which eliminates t redundant `XOR` gates and produces two output w-bit words at addresses $(i, i + \lceil m/2 \rceil)$); In Tab. IV, V and VI, lines with a star (*) correspond to state-of-art algorithms for multiplication and inversion from [7] and [13] we implemented and optimized on the same FPGA than our PNB solution. PNB corresponds to our MIU architecture at Fig. 2 with one $\times M_0$ block from Algo. 3 (Fig. 3) in PNB representation. The three versions have been implemented using the architecture of Fig. 2 with the corresponding internal multiplier. The inversion duration is reported in column named "Time". Bottom of Tab. V presents comparisons with state-of-art algorithms from [19] and [21]. Our solution is about 10 times slower but it is more than 10 times smaller. On a Virtex-4 LX100 (a mid-range device), there are 98304 LUTs. Then state-of-art solutions from [19] and [21] require 86 and 57 % of the device just for multiplication and inversion respectively.

The RM2 solution (with a large internal multiplier) leads to important frequency reduction compared to the original MO1 solution. Our PNB solution has a smaller impact on the circuit frequency. This can be an advantage when designing complete halving based accelerators. Our MIU with PNB representation seems to be interesting for the larger fields. For instance, in GF(2^{571}), we obtain a better area–speed trade-off than the RM2 for a cost similar to the simple MO1. Comparing various computation units is not an easy task when multiple area–speed trade-offs are possible. The efficiency of our MIU also depends on the targeted FPGA: the best results are obtained for the most recent one (this is probably due to a better use of 6-input LUTs and more important routing resources).

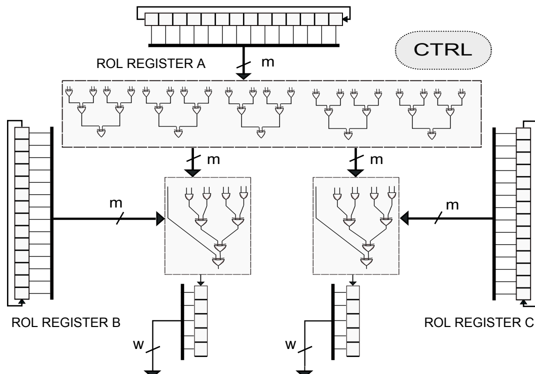

Fig. 3. High-level architecture of our modified Massey-Omura multiplier for PNB representation.

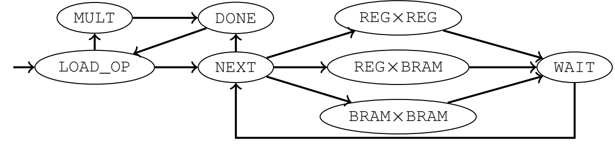

Fig. 4. Finite-state machine of the proposed multiplication-inversion unit.

TABLE IV
FPGA IMPLEMENTATION RESULTS FOR OUR MULTIPLICATION-INVERSION UNIT WITH VARIOUS INTERNAL MULTIPLIERS (SPARTAN 6 LX75T).

m	Algo.	Area Slices (LUT, FF)	Freq. MHz	Time μs	#block RAM
163	MO1 [7]*	337 (1004, 763)	217	7.84	1
	RM2 [13]*	423 (1348, 862)	140	6.04	1
	our PNB	469 (1411, 1034)	196	5.56	1
233	MO1 [7]*	420 (1231, 966)	229	11.6	1
	RM2 [13]*	569 (1765, 1078)	138	9.66	1
	our PNB	526 (1703, 1296)	181	10.0	1
283	MO1 [7]*	484 (1659, 1107)	160	22.0	1
	RM2 [13]*	668 (2164, 1209)	131	13.4	1
	our PNB	719 (2230, 1498)	159	15.3	1
409	MO1 [7]*	647 (2178, 1501)	163	31.2	1
	RM2 [13]*	917 (2768, 1610)	139	18.3	1
	our PNB	980 (3167, 1993)	159	22.3	1
571	MO1 [7]*	941 (3336, 1968)	109	75.0	2
	RM2 [13]*	1656 (4911, 2727)	80	53.8	2
	our PNB	1190 (4422, 2634)	94	63.5	2

Our solution actually factorizes some matrix-vector products in the SMPs. For large values of t, our solution leads to more factorizations of sub-expressions. We believe that the PNB solution could be used for fields whose type t is greater or equal to 10.

We estimated the cost and performance of various halving based ECC scalar multiplication algorithms. Paper [3] presents a rough estimation using r-NAF key recoding: $mI/(r+1) + mM(1 + 3/(r+1))\,\mu s$ where I is the inversion time and M the multiplication time (given in microseconds). We estimated the area cost and the computation time of all operators used in halving based methods (field addition, field multiplication-inversion, field square and square-root, trace, and other small specific operators). We compared several algorithms, the corresponding results are reported in Tab. VI. To compare the efficiency of various algorithms and area–time trade-offs, we also report the *area–time product* (ATP): the number of LUTs multiplied by the scalar multiplication time. The area results are very similar for both NAF and 3-NAF key recoding units (the area difference is about a few LUTs). Tab. VI shows that our PNB solution is more efficient when considering both computation time and silicon area. We think our approach can be used in applications where high security level is required on small circuits.

All algorithms have been validated using intensive functional simulation (at bit level) in Maple. All architectures have been validated using VHDL simulations using Modelsim.

TABLE V

FPGA IMPLEMENTATION RESULTS FOR OUR MULTIPLICATION-INVERSION UNIT WITH VARIOUS INTERNAL MULTIPLIERS (VIRTEX-4 LX100).

m	Algo.	Area Slices (LUT, FF)	Freq. MHz	Time μs	#block RAM
163	MO1 [7]*	906 (1636, 743)	250	6.80	1
	RM2 [13]*	1220 (2068, 808)	210	3.99	1
	our PNB	1227 (2274, 1058)	247	4.12	1
233	MO1 [7]*	1256 (2233, 1009)	202	13.2	1
	RM2 [13]*	1430 (2435, 1016)	204	6.53	1
	our PNB	1654 (2792, 1329)	212	8.22	1
283	MO1 [7]*	1577 (2839, 1149)	191	18.4	1
	RM2 [13]*	1924 (2435, 1147)	165	9.80	1
	our PNB	2073 (3741, 1525)	186	12.8	1
409	MO1 [7]*	2283 (2839, 1149)	169	31.1	1
	RM2 [13]*	2729 (4833, 1532)	150	15.9	1
	our PNB	2482 (4627, 2024)	155	21.4	1
571	MO1 [7]*	3378 (5615, 2016)	125	64.4	2
	RM2 [13]*	4976 (9445, 2090)	107	38.7	2
	our PNB	4308 (5928, 2650)	125	47.7	2
571	Hybrid ($d = 13$) [19]	#LUTs = 85268	74	4.98	–
	Parallel ($d = 13$) [21]	#LUTs = 56657	82	5.00	–

TABLE VI

COST/PERFORMANCE ESTIMATIONS FOR VARIOUS HALVING BASED ECC SCALAR MULTIPLICATIONS AND $m = 571$ ON A VIRTEX-4.

	Algorithm	halving ms	area #LUTs	ATP $\times 10^{-3}$
NAF	MO1 [7]*	17.3	5742	95
	RM2 [13]*	13.0	9572	122
	our PNB	**14.3**	6055	**82**
	Parallel IT (d=13) [21]	1.59	56784	90
	Hybrid IT (d=13) [19]	1.60	85395	136
3-NAF	MO1 [7]*	14.6	similar	79
	RM2 [13]*	8.95		76
	our PNB	**11.3**		**65**
	Parallel IT (d=13) [21]	1.34		74
	Hybrid IT (d=13) [19]	1.40		119

V. CONCLUSION

We proposed a new hardware unit, called MIU, combining GF(2^m) multiplication and inversion operations for halving based ECC cryptosystems. We proposed a new representation of the field elements called *permuted normal basis* (PNB), modifications of the Massey-Omura algorithm for multiplication, and the Itoh-Tsujii algorithm for inversion. Our solution leads to about 20 % theoretical speed-up over previous works. It has been implemented on FPGAs and seems to be interesting for the larger fields. It also leads to a higher frequency compared to parallel solutions from state-of-art. On halving based ECC scalar multiplication, our PNB solution leads to a better area-time efficiency for large fields.

In the future, we plan to study extensions with multiple internal multiplication blocks, mixing our PNB and parallel-IT [21] solutions and also try to adapt our algorithm (especially the optimization of very large shifters) to ASIC targets. We believe that PNB representation and associated algorithms could be efficiently used for finite fields with a type $t \geq 10$.

ACKNOWLEDGMENT

This work has been partially supported by the PAVOIS project (ANR 12 BS02 002 01).

REFERENCES

[1] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications*, 2nd ed. Cambridge University Press, 1994.

[2] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[3] E. W. Knudsen, "Elliptic scalar multiplication using point halving," in *Proc. Int. Conf. Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Nov. 1999, pp. 135–149.

[4] T. Oliveira, D. F. Aranha, J. Lopez, and F. Rodriguez-Henriquez, "Fast point multiplication algorithms for binary elliptic curves with and without precomputation," IACR ePrint, Tech. Rep. 427, Jun. 2014.

[5] C. Negre and J.-M. Robert, "New parallel approaches for scalar multiplication in elliptic curve over fields of small characteristic," LIRMM, University of Perpignan UPVD, Tech. Rep., 2013.

[6] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[7] J. K. Omura and J. L. Massey, "Computational method and apparatus for finite field arithmetic," US Patent US4587627 A, May 1986.

[8] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, Sep. 1988.

[9] NIST, "FIPS 186-2, digital signature standard (DSS)," 2000.

[10] M. A. Hasan and C. Negre, "Low space complexity multiplication over binary fields with dickson polynomial representation," *IEEE Transactions on Computers*, vol. 60, no. 4, pp. 602–607, Apr. 2011.

[11] H. W. Chang, W.-Y. Liang, and C. W. Chiou, "Low cost dual-basis multiplier over GF(2^m) using multiplexer approach," in *Knowledge Discovery and Data Mining*. Springer, 2012, pp. 185–192.

[12] Q. Liao, "The Gaussian normal basis and its trace basis over finite fields," *J. Number Theory*, vol. 132, no. 7, pp. 1507–1518, Jul. 2012.

[13] A. Reyhani-Masoleh, "Efficient algorithms and architectures for field multiplication using Gaussian normal bases," *IEEE Transactions on Computers*, vol. 55, no. 1, pp. 34–47, 2006.

[14] G.-L. Feng, "A VLSI architecture for fast inversion in GF(2^m)," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1383–1386, Oct. 1989.

[15] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone, "An implementation for a fast public-key cryptosystem," *Journal of Cryptology*, vol. 3, no. 2, pp. 63–79, Jan. 1991.

[16] R. Azarderakhsh and A. Reyhani-Masoleh, "Low-complexity multiplier architectures for single and hybrid-double multiplications in Gaussian normal bases," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 744–757, Apr. 2013.

[17] W. Drescher, K. Bachmann, and G. Fettweis, "VLSI architecture for non-sequential inversion over GF(2^m) using the euclidean algorithm," in *Proc. Conf. Signal Processing Applications and Technology*, 1997.

[18] D. E. Knuth, *Seminumerical Algorithms*, 3rd ed., ser. The Art of Computer Programming. Addison-Wesley, 1997, vol. 2.

[19] R. Azarderakhsh, K. Jarvinen, and V. Dimitrov, "Fast inversion in GF(2^m) with normal basis using hybrid-double multipliers," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 1041–1047, Apr. 2014.

[20] P. Jarvinen, S. Dimitrov, and R. Azarderakhsh, "A generalization of addition chains and fast inversions in binary fields," *IEEE Transactions on Computers*, 2015, accepted paper.

[21] J. Hu, W. G. J. Wei, and R. Cheung, "Fast and generic inversion architectures over GF(2^m) using modified Itoh-Tsujii algorithms," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2015, accepted paper.

[22] M. D. Bielefeld University, "Shortest addition chains," Website, 2011, http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html.

[23] D. Hankerson, J. Lopez, and A. Menezes, "Field inversion and point halving revisited," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 1047–1059, 2004.