A Class of Scalable Unidirectional Bit Serial Systolic Architectures for Multiplicative Inversion and Division over GF(2^m)

Amir K. Daneshbeh and M.A. Hasan

Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada
E-mail: akdanesh@engmail.uwaterloo.ca, ahasan@ece.uwaterloo.ca

Abstract

A class of systolic architectures for multiplicative inversion and division over Galois field GF(2^m) is presented. The input field element to the inverter can be represented with a polynomial or triangular basis. The same is true for the divisor of the divider. The output of both the inverter and the divider is always returned in polynomial basis. These architectures are defined as universal because any field dimension to a maximum determined by the number of identical processing elements can be processed. Specifically, these scalable unidirectional counter-free bit serial systolic architectures are suitable for hardware implementations where the dimension of the field is large and may vary. This is the typical case for cryptographic applications. The time and area complexities are truly linear (no counters) and in spite of extra circuitry for flexibility and universality, the detailed complexity measures are equivalent or excel the best designs proposed so far. Also, since the data and control flow are unidirectional, not only are these architectures easily scalable but more importantly they are suitable for fault-tolerant implementation.

Keywords Division, extended Euclidean algorithm, finite fields, field arithmetic, inversion, systolic arrays, triangular basis.

1 Introduction

Finite field arithmetic has many applications in coding theory [1], and cryptography [17]. Among the basic arithmetic operations over finite fields the multiplicative inversion is the most costly one. Many hardware schemes to compute multiplicative inverses have been proposed which are mentioned next, but most of them suffer from a common problem. In applications where the dimension of the field is large or may vary, their implementation becomes cumbersome or even impractical. In particular, in cryptographic applications, the security strength of all cryptosystems is determined by the length of their keying material. For example, in public-key cryptosystems where the field dimension may exceed 4000 for discrete logarithm cryptosystems or 500 for elliptic curve cryptosystems [17], the common practice has been to avoid the costly inversion by devising complex and less inversion dependent algorithms.

In general, three major schemes to compute multiplicative inverses exist: Fermat's little theorem, variants of the extended Euclidean algorithm (EEA) or a solution of a set of linear equations. The first scheme is efficient if a fast squaring or multiplication unit is available [14]. Euclidean based schemes are the most efficient in time and area but may require variable size counters and/or pre or post computations [13, 18, 20] in which case they are not suitable for high-performance and scalable VLSI implementations. The third method is generally inefficient for large values of field dimension. However, it is shown that a set of linear equations formed upon a triangular basis representation can be solved by similar to EEA schemes [10].

Many systolic array proposals for multiplicative inversion or division over GF(2^m) exist [5, 7, 13, 21, 22, 23]. Systolic arrays by definition are modular, hence, scalable. But there are other limiting factors to the scalability. Specifically, the inversion schemes based on variants of Euclidean algorithm require a counter-like structure (incrementer-decrementer) to keep track of the difference of the degree of polynomials. Such counter-like structures are dependent on field dimension m and they may dominate the critical delay path of a systolic array. Only counter-free systolic architectures can be considered dimension independent and truly scalable. Also, among these, only unidirectional control and data flow structures can be easily extended to be a universal (variable field dimension) and fault tolerant design.

In this paper we expand the work in [10] on double basis (triangular-polynomial) multiplicative inversion scheme over GF(2^m), which is particularly important if other operations are implemented using triangular basis as well. Further, it will be shown that, in this scheme seen as a two step process, the cost of change of basis and precomputation step (both in time and area) is much less than EEA implementation for the polynomial updating step and these two can be pipelined. We present a common systolic implementation of a variant of EEA which computes inversion and division either with input in polynomial or triangular basis. Specifically, a class of unidirectional scalable counter-free bit serial systolic array architectures are introduced that can perform inversion and/or division of variable field dimensions with input in polynomial or triangular basis, and they can be easily scaled for large values of m.

This article is organized as follows. In Section 2, some mathematical background is reviewed. In Section 3, a variant of EEA is mentioned; and an efficient triangular basis inversion algorithm as in [10] is further investigated and its similarities to the EEA variant are highlighted. In Section 4, by imposing some constraints useful for systolization on these algorithms, a new variant of EEA is proposed. In Section 5, a bit serial unidirectional counter-free systolic architecture for inversion and division with polynomial or triangular basis inputs is proposed. In Section 6, a general structure for inversion/division over variable size fields and some optimization techniques are discussed. In Section 7, comparative designs are evaluated and we conclude with some remarks on future work in Section 8.

2 Preliminaries

The efficiency of algorithms to compute inversion, as well as other operations over finite fields, depends directly upon the representation of finite field elements. The description of a polynomial basis and its corresponding triangular basis representations will follow. For a general background of finite prime fields, binary extension fields and the theory of polynomials over finite fields, reader may refer to [16].

Given an irreducible polynomial $F(x)$ of degree m over the finite field GF(2), considering one of the roots of $F(x)$ such as ω, *i.e.*, $F(\omega) = 0$, it is well known that the set of elements $\Omega = \{1, \omega, \omega^2, \cdots, \omega^{m-1}\}$ is linearly independent and forms a basis referred to as a polynomial basis. Then, any element $A \in GF(2^m)$ can be represented as $A = \sum_{i=0}^{m-1} a_{\Omega i}\omega^i$, where $a_{\Omega i} \in \{0, 1\}$ are the coordinates of A with respect to the polynomial basis Ω, or in a column vector form $\underline{a}_\Omega = [a_{\Omega 0}, a_{\Omega 1}, \cdots, a_{\Omega m-1}]^T$ over GF(2).

Given the polynomial basis Ω, the set $\Delta = \{\delta_0, \delta_1, \cdots, \delta_{m-1}\}$ is linearly independent and forms a basis [12] if $\delta_i = \sum_{j=0}^{m-1-i} f_{i+j+1}w^j,\ 0 \leq i \leq m-1$, where f_i's are coefficients of $F(x)$. The set Δ is referred to as a triangular basis of Ω. Next, the element A of GF(2^m) can also be represented as $A = \sum_{i=0}^{m-1} a_{\Delta i}\delta_i$, where $a_{\Delta i}$ is the ith coordinate of A w.r.t. Δ.

The change of coordinates of element A between Ω and Δ bases is given by [12]

$$a_{\Omega j} = \sum_{i=0}^{m-j-1} f_{i+j+1} a_{\Delta i} \quad 0 \leq j \leq m-1 \tag{1}$$

and

$$a_{\Delta k} = \begin{cases} a_{\Omega m-1-k} & k = 0, \\ a_{\Omega m-1-k} + \sum_{i=0}^{k-1} f_{m-k+i} a_{\Delta i} & 1 \leq k \leq m-1. \end{cases} \tag{2}$$

The linear transformation, in Equation (2), from polynomial to triangular basis can be readily written in matrix form as $\underline{a}_\Delta = \mathbf{M}\underline{a}_\Omega$, with $\mathbf{M}$ as

$$\mathbf{M} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & t_1 \\ 0 & 0 & 0 & \dots & 1 & t_1 & t_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 1 & t_1 & \dots & t_{m-4} & t_{m-3} & t_{m-2} \\ 1 & t_1 & t_2 & \dots & t_{m-3} & t_{m-2} & t_{m-1} \end{pmatrix}, \tag{3}$$

where $t_i = \sum_{j=0}^{i-1} f_{m-i+j} t_j$, for $1 \leq i \leq m-1$, and $t_0 = 1$.

3 Inversion and Division By Euclidean Algorithm

In this section two variants of the basic extended Euclidean algorithm (EEA) is shown which can compute inversion or division using polynomial or triangular basis representation.

3.1 Inversion and Division over Polynomial Basis

First, let us recall inversion based on the basic EEA shown as Algorithm 1. The inputs are the irreducible polynomial $F(x)$ and the polynomial $A(x)$ corresponding to element A, in polynomial basis. Two aspects of this algorithm should be noted. First, the *exit* condition of Algorithm 1 is ensured by monotonically decreasing value of $R^{(i)}(x)$. Second, the invariant of EEA,

$$W^{(i)}(x)F(x) + U^{(i)}(x)A(x) = R^{(i)}(x), \tag{4}$$

is preserved at each iteration. In the following we ignore the $W^{(*)}(x)$ polynomials since they are not relevant to the inversion.

Algorithm 1 Inversion by Basic Extended Euclidean Algorithm (EEA).

input: $F(x)$, $A(x)$.

output: The inverse of $A(x)$.

Initialization:

$i \leftarrow 0;\quad R^{(-1)}(x) \leftarrow F(x);\quad R^{(0)}(x) \leftarrow A(x);$

$U^{(-1)}(x) \leftarrow 0;\quad U^{(0)}(x) \leftarrow 1;\quad W^{(-1)}(x) \leftarrow 1;\quad W^{(0)}(x) \leftarrow 0;$

Polynomial Updating:

while $(R^{(i)}(x) > 0)$ **do**

$i \leftarrow i + 1;$

$Q^{(i)}(x) \leftarrow \lfloor R^{(i-2)}(x)/R^{(i-1)}(x) \rfloor;$

$R^{(i)}(x) \leftarrow R^{(i-2)}(x) - Q^{(i)}(x)R^{(i-1)}(x);$

$U^{(i)}(x) \leftarrow U^{(i-2)}(x) - Q^{(i)}(x)U^{(i-1)}(x);$

$W^{(i)}(x) \leftarrow W^{(i-2)}(x) - Q^{(i)}(x)W^{(i-1)}(x);$

return $U^{(i-1)}(x)$

In Algorithm 1, the $Q^{(i)}(x)$ represents the quotient of a polynomial division [1]. The implementation of Algorithm 1 is inefficient mainly due to the polynomial division step to compute this quotient $Q^{(i)}(x)$. On the other hand, since not the quotient itself but the remainder, either exact or a series of partial remainders, is what we need, many variants of Algorithm 1 have been proposed, *e.g.*, [11, 2, 8, 7, 13, 23]. Basically, the polynomial updating step can be expanded by concatenating a series of multi-stepwise long division iterations where the invariant of Equation (4) should be preserved.

Algorithm 1 can be extended to perform division B/A, where $A, B \in \mathrm{GF}(2^m)$, $A \neq 0$, by initializing $U^{(0)}(x)$ with the dividend polynomial $B(x)$ whose degree is less than or equal to $m-1$. In this case, while computing $U^{(i)}(x)$, a modulo reduction may be required, since the degree of $Q^{(i)}(x)U^{(i-1)}(x)$ may become m or more.

Algorithm 1a is a variant of Algorithm 1 following a long-division scheme where the quotient calculation is performed stepwise by computing d, the difference of the degree of two polynomials. The number of iterations and partial remainders $R^{(i)}(x)$ can be much more than Algorithm 1. On the other hand, this partial remainder computation requires the introduction of a *restoring* mechanism, **swap**, since in this variant, the degree of a newly computed $R^{(i)}(x)$ is not necessarily less than that of $R^{(i-1)}(x)$ as it was ensured to be monotonically decreasing in Algorithm 1. The *swap* instruction of Algorithm 1a enforces this condition.

[1] It is worth mentioning that this polynomial division is over the binary field GF(2) and it is different from our target division B/A over the extension field $\mathrm{GF}(2^m)$.

Algorithm 1a Polynomial Basis Inversion Algorithm (PBIA) using EEA.

input: $F(x)$, $A(x)$.

output: The inverse of $A(x)$.

Initialization:

$i \leftarrow 0;\ \ U^{(-1)}(x) \leftarrow 0;\ \ U^{(0)}(x) \leftarrow 1;\ \ R^{(-1)}(x) \leftarrow F(x);\ \ R^{(0)}(x) \leftarrow A(x);$

Polynomial Updating:

while ($\deg R^{(i)}(x) > 0$) **do**

$i \leftarrow i + 1;$

$d \leftarrow \deg R^{(i-1)}(x) - \deg R^{(i)}(x);$

$R^{(i)}(x) \leftarrow R^{(i-2)}(x) - x^d R^{(i-1)}(x);$

$U^{(i)}(x) \leftarrow U^{(i-2)}(x) - x^d U^{(i-1)}(x);$

if ($\deg R^{(i)}(x) \geq \deg R^{(i-1)}(x)$) **then**

swap ($R^{(i)}(x),\ R^{(i-1)}(x)$) **swap** ($U^{(i)}(x),\ U^{(i-1)}(x)$);

return $U^{(i)}(x)$

In addition, the **while** loop condition is reduced to a degree checking rather than the content. Algorithm 1a is further modified to end one iteration earlier when $R^{(i)}(x)$ reaches one vs. zero of Algorithm 1. Hence, the return value will be $U^{(i)}(x)$ which is the same as $U^{(i-1)}(x)$ in Algorithm 1. These last two modifications are made to show similarities between Algorithm 1a using polynomial basis with the one based on triangular basis as presented next in Section 3.2.

3.2 Inversion and Division Using Triangular Basis

In [10], an inversion algorithm is proposed which uses a triangular basis representation for input A and returns its inverse in the polynomial basis. It is said that the inversion can be seen as the solution to a set of linear equations written as

$$\underline{1}_\Delta = \mathbf{H}(\underline{a}_\Delta)\underline{b}_\Omega \tag{5}$$

where $\underline{1}_\Delta$ is a column vector representing the multiplicative identity element of GF(2^m) w.r.t. Δ and $\mathbf{H}(\underline{a}_\Delta) = [\underline{a}_\Delta, \underline{a\omega}_\Delta, \cdots, \underline{a\omega}_\Delta^{m-1}]$ is a Hankel matrix built upon $\underline{a}_\Delta$, the column vector representation of element A with entries as its triangular basis coordinates. Then, the coordinates of the inverse element B in the polynomial basis can be computed by solving Equation (5). The Hankel matrix in (5) has $2m-1$ constant coefficients. Let these be $h_i,\ 0 \leq i \leq 2m-2$, then the matrix entry at (i, j) is h_{i+j}. In Section 3.3, it is discussed how h_i's can be computed efficiently. In [10], an algorithm to solve Equation (5) is proposed which is repeated here as Algorithm 2.

Algorithm 2 Triangular Basis Inversion Algorithm (TBIA) using EEA.

input: $H(x) = \sum_{i=0}^{2m-2} h_i x^i$

output: The inverse element.

Initialization:

$i \leftarrow 0$; $U^{(-1)}(x) \leftarrow 0$; $U^{(0)}(x) \leftarrow 1$; $R^{(-1)}(x) \leftarrow x^{2m-1}$; $R^{(0)}(x) \leftarrow x^{2m-2}H(x^{-1})$;

Polynomial Updating:

while $(\deg R^{(i)}(x) > m-1)$ **do**

$i \leftarrow i+1$;

$d \leftarrow \deg R^{(i-2)}(x) - \deg R^{(i-1)}(x)$;

$R^{(i)}(x) \leftarrow R^{(i-2)}(x) - x^d R^{(i-1)}(x)$;

$U^{(i)}(x) \leftarrow U^{(i-2)}(x) - x^d U^{(i-1)}(x)$;

if $(\deg R^{(i)}(x) \geq \deg R^{(i-1)}(x))$ **then**

swap $(R^{(i)}(x), R^{(i-1)}(x))$; **swap** $(U^{(i)}(x), U^{(i-1)}(x))$;

return $U^{(i)}(x)$

Algorithms 2 and 1a are similar. Despite the differences of the input polynomials in the initialization step, the polynomial updating step of both algorithms are the same variant of the EEA. The apparent difference of the while loop condition represents the same number of degree reduction in both cases. Also here, the same generalization to perform division by applying a modulo reduction on $U^{(i)}(x)$, as in the case of polynomial basis, is possible which will be discussed later.

The difference between the initialization step of Algorithms 1a and 2 is essential. Solving Equation (5) and consequently Algorithm 2 could be considered as an effective two step process represented in block diagram in Figure 1. The Hankel matrix entry formation for the initialization step must start, pipelined or otherwise, before the polynomial updating step. Below it is shown that the computational complexity of the first step can be much less than the second in the specific (practical) cases.

3.3 Some Comments on Hankel Matrix Formation

In general, the formation of $2m-1$ entries of the Hankel matrix requires recursive formulae such as [10]

$$h_i = \begin{cases} a_{\Delta i} & 0 \leq i \leq m-1, \\ \sum_{j=0}^{m-1} h_{i-m+j} f_j & m \leq i \leq 2m-2. \end{cases} \tag{6}$$

Let us consider the entries of this Hankel matrix as a polynomial representation $H(x) = h_{2m-2}x^{2m-2} + h_{2m-3}x^{2m-3} + \cdots + h_1 x + h_0$. Here, we add some comments on the formation

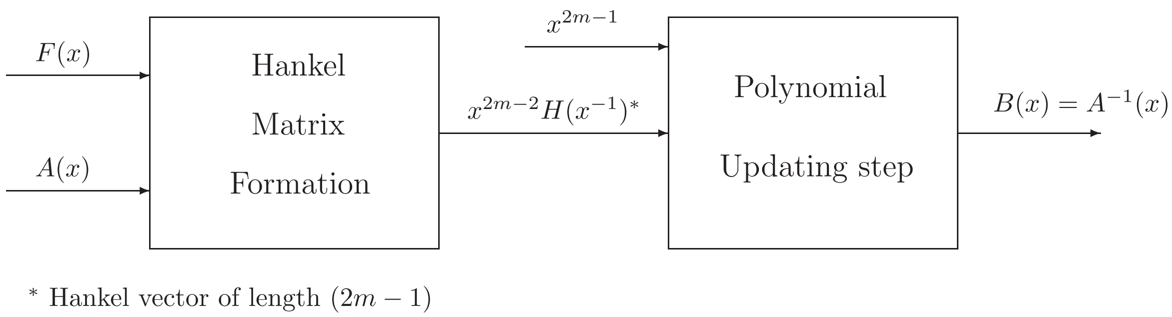

Figure 1: Two step block diagram of Algorithm TBIA.

of the coefficients of this polynomial.

- In [10], it is shown that by a suitable choice of irreducible polynomial, such as trinomials of type x^m+x+1, the formation of the entries of the Hankel matrix requires no recursion and they can be easily generated using a single row of XOR gates, with a delay path of one gate only. That is, since $f_j = 0$ for $2 \le j \le m-1$, and only $f_0 = f_1 = 1$, then $h_i = h_{i-m+1} + h_{i-m}$ for $m \le i \le 2m-2$.

- For all practical purposes, there exists an irreducible trinomial or pentanomial [19], hence a suitable choice of them will be useful. For general trinomials, $x^m + x^k + 1$, with $k \le \lfloor m/2 \rfloor$, each coefficient can be generated by 2 XOR gates at maximum (worst case delay path). With a similar reasoning as above, only $f_0 = f_k = 1$ and all other $f_i = 0$, then let $\Delta = m - k$, for $m \le i \le m+k-1$, $h_i = h_{i-m+k} + h_{i-m}$, with both right hand terms directly available; and for $m+k \le i \le 2m-2$, $h_i = h_{i-m+k} + h_{i-m} = (h_{i-\Delta}) + h_{i-\Delta-k}$ with $i - \Delta > m$ hence $h_i = (h_{i-2\Delta} + h_{i-2\Delta-k}) + h_{i-\Delta-k}$ with all three right hand terms directly available. These complexity figures and ones that follow can be computed in the same manner as those of entries for the multiplication matrix in Mastrovito multipliers [9].

- For pentanomials, $x^m + x^k + x^j + x^i + 1$, where $m > k > j > i > 0$, it can be shown that the critical path is dependent on $\Delta = m - k$. More precisely, the number of XOR gates required to generate each entry are always less than or equal to $C = \frac{m-2}{\Delta} + 3$, with a worst case delay path $\lceil \log_2 C \rceil$. Specifically, the worst case corresponds to pentanomials of type $x^m + x^{m-1} + x^j + x + 1$. However, in [19], it is shown that for pentanomials up to degree 10000, for the degrees where no trinomial exists, the value of k can be quite low, and it is suggested that it is close or below the real solution t to the equation $m = t(t-1)(t-2)/6$. Under these circumstances, very low values of the critical path can be achieved.

- In [4], Furness et al. have shown that for trinomials, the triangular basis is a simple permutation of the polynomial basis, and for pentanomials this conversion requires a minimal hardware and a reordering of the coordinates of the element. For general irreducible polynomials, the conversion between a polynomial basis representation and its corresponding triangular basis representation can be systolized with techniques similar to the one used in bit serial dual basis systolic multipliers [3]; this may be useful when other computations such the polynomial updating step of Algorithm 2 are systolized as well.

We conclude that the Hankel matrix formation has a lower computational complexity compared to the polynomial updating step of Algorithm 2, and in the worst case, these two can be pipelined. In what follows we concentrate only on the implementation of the polynomial updating step of Algorithms 1a or 2 which are basically the same, and mention the effect of the initialization step whenever necessary.

4 Systolization of the Polynomial Updating Step

For large values of m, the complexity of a centralized control implementation of the polynomial updating step of Algorithm 2 is dominated by the presence of counter-like structures and long control interconnects. In such cases, a systolic implementation would be more suitable for VLSI design.

Systolic structures are mainly proposed to increase throughput and decrease the data transfer between the storage and processing elements (PE) by using bit level pipelining among similar PEs which results in extremely high clock rates. There are different systolic architectures which trade-off throughput, latency, delay and area. Many systolic proposals to compute multiplicative inverses over GF(2^m) use $\lceil \log_2 m \rceil$ bit counters or necessitate bidirectional control flows to keep track of the degree of the polynomials. If one of the goals of the systolization is to eliminate a centralized control for large values of m, then a systolic architecture with distributed control (*i.e.*, counter free) architecture is necessary. Moreover, if the structure is unidirectional then it will be suitable for VLSI designs where fault-tolerance is of prime concern [15].

4.1 Tasks of the Polynomial Updating Step

From a scheduling point of view, in each iteration, Algorithm 1a must perform three tasks:

- Task 1. Check for the *exit* condition: $degR(x) > 0$.

- Task 2. *Align* two polynomials $R^{(i-2)}(x)$ and $R^{(i-1)}(x)$ by shifting the latter d times and compute a new polynomial $R^{(i)}(x)$.
- Task 3. Check for the *swap* condition by computing the difference of the degree of polynomials $R^{(i)}(x)$ and $R^{(i-1)}(x)$ which is the value of d for the next iteration.

These tasks as described are not suitable for a direct mapping onto hardware. For example, Tasks 1 and 3 either require counter-like structures, or in order to implement each one of three tasks in a single cycle, an access mechanism to all coefficients of polynomials at each iteration should be available which may not be either desirable or even practical.

In a practical hardware implementation for large values of m and reasonable clock rates, in particular in a systolic architecture, these tasks must be implemented in a *stepwise* fashion, *e.g.*, digit or bit serial. Specifically, the shifting of a polynomial in Task 2, *i.e.*, $x^d R^{(i-1)}(x)$, should be performed stepwise, *e.g.*, in d iterations. In this case all tasks defined above which are tightly intertwined must be redefined.

4.2 Tasks 1 and 2: Long division with Shifted Divisor

A *stepwise* "long division" operation is described by many authors, *e.g.*, [2, 23]. We will consider a variant of a "long division" operation which requires, as a pre-condition, the degree of its dividend to be greater than the degree of its divisor. As the post-condition, it ensures that the degree of its remainder is less than the degree of its divisor at its termination. This variant of "long division" operation is shown as Algorithm 3.

Algorithm 3 Shifted Polynomial Long Division Algorithm.

input: r and s polynomials with $\deg(r) > \deg(s)$.
output: $x^{\delta}(r \pmod{s})$, $x^{\delta}s$, where $\delta = \deg(r) - \deg(s)$.

```
  δ ← 0
  while (deg(r) > deg(s)) do
    δ ← δ + 1
    s ← xs;
  r ← (r − s);      (*first alignment, this one always occur*)
  while (δ > 0) do
    δ ← δ − 1
    r ← xr;
    if (deg(r) = deg(s)) then
      r ← (r − s);
return r, s
```

In Algorithm 3, the pre- and post-condition as defined above are satisfied, however, a δ times shifted version of final remainder and the divisor are returned. Initially, the degree of dividend is greater than the degree of divisor by δ units. For δ iterations a divisor polynomial is left shifted, *i.e.*, multiplied by x. At a first alignment iteration where the degree of shifted divisor is reached the degree of dividend, a remainder polynomial is computed which may have a degree greater, equal or less than the pre-shifted divisor, but always its degree is less than the degree of the shifted divisor. Next, a "virtual" restoring of divisor starts. That is the degree of the shifted divisor is not decreased, but rather in the next δ iterations, the remainder or partial remainders are left shifted where none, one or more other alignments may occur and new partial remainders may be computed. After exactly δ iterations, the pre-shifted divisor is "virtually" restored, and in all these δ iterations the remainder polynomial is continuously updated to a degree less than the shifted divisor. Thus, at the termination of each "long division" operation, its post-condition is ensured.

Let us consider a stepwise "virtually" restoring implementation of Algorithm 1a where a concatenation of many "long division" operations occurs and the pre-condition of the first "long division" is satisfied by $\deg F(x) > \deg A(x)$. It will be shown that the last "long division" returns 0 and x^m, as its output. Furthermore, the following facts always hold; Facts 1, 2, and 4 are straightforward and are stated without proof.

Fact 1. *Each "long division" requires 2δ iterations to complete where δ is the difference of the degree of dividend and divisor. Specifically, δ iterations are required for its divisor to be aligned with its dividend and δ more iterations are needed to "virtually" restore the divisor. The algorithm starts and ends with $\delta = 0$.*

Fact 2. *Each "long division", except the first, starts with the previous shifted divisor as its dividend and the previous remainder as its divisor. The first "long division" has $F(x)$ and $A(x)$ as dividend and divisor respectively.*

Fact 3. *At the end of each "long division", i.e., 2δ iterations, the degree of a shifted divisor is "virtually" restored, but its shifted degree is always m.*

Proof. At initialization of Algorithm 1a and the first "long division", the degree of the dividend is $\deg(R^{(-1)}(x)) = \deg(F(x)) = m$. Let the difference of degree of dividend and divisor be δ, then, after δ iterations the first *alignment* condition occurs which can be easily recognized by the degree of the divisor reaching m. Afterward, for the next δ iterations up to the termination of "long division", the degree of shifted divisor is not changed and remains m, but its relative degree with respect to remainders is decreased by δ units. Next, according to Fact 2, at the start of the next "long division", this shifted divisor with degree m will be used as a new dividend and this process is repeated. $\square$

Fact 4. *In each iteration of a "long division" and consequently a stepwise "virtually" restoring implementation of Algorithm 1a a single left shift operation occurs.*

The first advantage of a stepwise "virtually" restoring implementation and a direct consequence of Fact 4 is the possibility to compute as exact upper bound of the number of iterations of Algorithm 1a, and to implement Task 1 defined in the beginning of this section at no cost. At initialization of Algorithm 1a, $\deg(R^{(-1)}(x)) = \deg(F(x)) = m$. In a single shift per iteration scheme, in the worst case, at $2m-1$st iteration both polynomials $R^{(2m-2)}(x) = R^{(2m-1)}(x) = 1$, assuming they are effectively restored. However, in a "virtually" restoring implementation, at each iteration, one of the polynomials $R^{(*)}(x)$ is left shifted by exactly 1, and its lower order coefficients are zeroed one at a time. Hence, according to Fact 3, at $i = 2m$th iteration, before the last iteration of the last "long division" operation is executed,

$$R^{(2m-2)}(x) = R^{(2m-1)}(x) = x^m.$$

That is all lower order coefficients are zeroed. At this point the invariant previously defined by Equation (4) will become

$$x^m W^{(i)}(x)F(x) + x^m U^{(i)}(x)A(x) = x^m. \tag{7}$$

It is mentioned that the last "long division" ends returning 0 and x^m.

In a bit serial systolic architecture where each iteration is mapped onto a different PE, Task 1, and its **while** loop, can be defined as a **for** loop of $2m$ iterations and simply implemented by the presence of $2m$ PEs. (Implementations with $2m-1$ PEs, or even $2m-2$ PEs with a shifted initialization, may suffice since the final result, inverse of a multiplicative inversion or quotient of a field division, is computed at $2m-1$st iteration but in these cases other information regarding where and how to access the result may be required.)

The case of $2m$ iterations is an upper bound, *i.e.*, worst-case. In all other cases, the condition $R^{(i-2)}(x) = R^{(i-1)}(x) = x^m$ occurs when $i < 2m$, then a modified version of Algorithm 1a must ensure that the value of $x^m U^{(i)}(x)$ are not modified in consecutive iterations up to $2m$. In Sections 4.4 and 4.5, it is discussed how to recover the correct result $U^{(i)}(x)$ from $x^m U^{(i)}(x)$.

4.3 Task 3: Swapped Shifted Long Division Algorithm

The main advantage of a stepwise "virtually" restoring implementation of Algorithm 1a is that Task 3 can be implemented efficiently with no swapping of polynomials, but rather a selective overwriting of one of them. This also means that the same polynomial is always shifted. In order to highlight this, Algorithm 3 can be rewritten as in Algorithm 4.

Algorithm 4 Swapped Shifted Polynomial Long Division Algorithm.

input: r and s polynomials with $\deg(r) > \deg(s)$.
output: $x^{\delta}(r \pmod{s}),\ x^{\delta}s$, where $\delta = \deg(r) - \deg(s)$.

```
δ ← 0
while (deg(r) > deg(s)) do
    δ ← δ + 1
    s ← xs;
[r, s] ← [s, (r − s)];      (*first alignment, and overwriting of r*)
while (δ > 0) do
    δ ← δ − 1
    s ← xs;
    if (deg(r) = deg(s)) then
        s ← (r − s);
return r, s
```

Algorithm 4 indicates that the same shifting polynomial s which represents the remainder of one "long division" operation will be the divisor of the next operation. As a consequence, no *swapping* of polynomials at the end of a "long division" is needed. That is Fact 2 is satisfied automatically. Thus, in this case, the end of a "long division" uniquely corresponds to a reset of δ, and δ is needed only to distinguish between an overwriting condition (or a first alignment case) and other alignment cases.

Let d be defined as a common up/down counter among all "long division" steps. That is $d = \delta$ in each "long division" step which counts the number of shifts of the shifting polynomial s. Then, Task 3 is simply an *overwriting* check which occurs at an *alignment* if d was incrementing. After an *overwriting*, d starts decrementing to return to zero again. This definition is particularly advantageous for hardware implementation.

Algorithmically, a more illustrative way to express d as a signed value exists. At an *alignment* where $d > 0$, an overwriting occurs and the sign of d is negated. This signed d can be kept increasing at all iterations. Hence a positive sign of this d indicates that the counter d was incremented, while a negative sign of it shows that the counter d was decremented. The $d = 0$ indicates that it was decremented and it will start incrementing next. In this sense, the sign of $d = 0$ is assumed to be negative. Now, Fact 1 can be extended as

Fact 5. *A stepwise "virtually" restoring implementation of Algorithm 1a is a concatenation of n "long division" operations which requires $2m = \sum_{i=1}^{n} 2\delta_i$ iterations to complete. Further, the counter d which corresponds to δs of Algorithm 3 or 4, is incrementing and decrementing exactly m times each.*

4.4 Shifted Result Problem

An inherent drawback of a stepwise "virtually" restoring implementation of Algorithm 1a is what we may call the *shifted result* problem. That is while the use of shifted pair of polynomials $R^{(*)}(x)$ are advantageous for the implementation of Tasks 1, 2, and 3, the same is not true for computation of pair of polynomials $U^{(*)}(x)$ which becomes a *shifted result* and this pair of polynomials must be restored. This problem was already shown in Equation (7) where the result polynomial was written as $x^m U^{(i)}(x)$.

Different strategies to solve this problem, which is to perform $x^m U^{(i)}(x)/x^m$, have been proposed: initializing $U^{(0)} = x^{-m}$ [13]; initializing $U^{(-1)}(x) = x^m$, $U^{(0)}(x) = 0$, but at each iteration divide $U^{(i-1)}(x)$ instead of multiply such that $U^{(i)}(x) \leftarrow U^{(i-2)}(x) - U^{(i-1)}(x)/x^d$ ([1], Section 2.3), [23]; multiplying or dividing $U^{(i-1)}(x)$ alternatively [2]; or using an auxiliary polynomial [5]. The first method requires pre-computation for each different irreducible polynomial. The second and third proposals compute only inverses. The fourth method requires an auxiliary polynomial.

The architecture described in [2] is the least efficient of all, specifically, in a centralized control design. In this case, the main inefficiency is that, even for an inversion, it required both left and right shifts over a constraint $(m + 1)$ bit register with the consequence that modulo reductions were needed when either the highest order or lowest order coefficients of a result sequence were being shifted out.

On the other hand, by allowing the result polynomials to expand in both directions, *i.e.*, to exceed x^m (left shifted) and x^{-1} (right shifted), a new method will be proposed which conceptually follows that of [2] but appears to be the most efficient scheme when implemented in a bit serial systolic structure and computes the division as well. In fact, in the inversion case this implementation requires no modulo reduction, and no auxiliary polynomials are used in the case of field division. By using double delay elements in a bit serial systolic structure a stepwise restoring of $U^{(*)}(x)$ can be performed with no need for a modulo reduction. This restoring transformation method will be described in the following variant of EEA.

4.5 A Stepwise Restoring Transformation Inversion Algorithm

In Algorithm 5, a new variant of Algorithm 1a with four major modifications is proposed. Specifically, the *exit* condition has an exact number of iterations; a bivariable renaming of temporally defined polynomials into physically implemented ones is used; the newly defined signed integer d is introduced where the sign of d is used for overwriting and to select the restoring $U^{(*)}(x)$ transformations.

Algorithm 5 Inversion with Single Shift, Bivariable Renaming, Overwriting and Restoring.

input: $F(x)$, $A(x)$.

output: The inverse of $A(x)$.

Initialization:

$d \leftarrow 0;\ \ U(x) \leftarrow 0;\ \ V(x) \leftarrow 1;\ \ R(x) \leftarrow F(x);\ \ S(x) \leftarrow A(x);$

Polynomial Updating:

for $(i = 1$ to $2m)$ **do**

  if $(s_m = 1\ \&\ d > 0)$ **then**

    $d \leftarrow -d;$ (*the overwriting case*)

    $[R(x),\ S(x)] \leftarrow [S(x),\ x(R(x) - S(x))];$

    $[U(x),\ V(x)] \leftarrow [V(x),\ x(U(x) - V(x))];$

  else if $(s_m = 1\ \&\ d \leq 0)$ **then**

    $[R(x),\ S(x)] \leftarrow [R(x),\ x(R(x) - S(x))];$

    $[U(x),\ V(x)] \leftarrow [U(x),\ x(U(x) - V(x))];$

  else if $(s_m = 0\ \&\ d \geq 0)$ **then**

    $[R(x),\ S(x)] \leftarrow [R(x),\ xS(x)];$

    $[U(x),\ V(x)] \leftarrow [(U(x))/x,\ V(x)];$

  else if $(s_m = 0\ \&\ d < 0)$ **then**

    $[R(x),\ S(x)] \leftarrow [R(x),\ xS(x)];$

    $[U(x),\ V(x)] \leftarrow [U(x),\ xV(x)];$

  $d \leftarrow d + 1;$

return $U(x)$

In this variant, we distinguish between two consecutive pairs of polynomials $R^{(*)}(x)$ and $U^{(*)}(x)$ of Algorithm 1a which have orderly indexes during two consecutive iterations. The pair of $R^{(i-2)}(x)$ and $R^{(i-1)}(x)$ are renamed as a pair of distinct polynomials $[R(x),\ S(x)]$. Specifically, $S(x)$ will be the shifting polynomial where the newly formed divisor with a degree less than m is assigned and consequently shifted to be aligned with $R(x)$ which will be the dividend, non-shifting polynomial, with the degree equals to m always. Similarly, the polynomials $U^{(i-2)}(x)$ and $U^{(i-1)}(x)$ may be renamed into a pair of polynomials $[U(x), V(x)]$. In this setting, the *alignment* is simply $(s_m = 1)$, where s_m is the mth degree coefficient of $S(x)$ and the *overwriting* occurs at $(s_m = 1\ \&\ d > 0)$.

The main innovation of Algorithm 5 is the restoring transformation of $[U(x),\ V(x)]$ versus the non-restoring transformation of pair $[R(x),\ S(x)]$. It is said that the aim of this restoring process is to perform $x^m U(x)/x^m$. Fact 5 in Section 4.1 asserts that, in Algorithm 5, the counter d is incrementing and decrementing for exactly m times each. Hence, a stepwise restoring of $x^m U(x)/x^m$ can be performed in exactly m step while $(s_m = 0\ \&\ d \geq 0)$.

Algorithm 5 ensures that the the same transformation is applied to both pair of polynomials in spite of the restoring action. At an alignment, *i.e.*, ($s_m = 1$), or while ($s_m = 0$ & $d < 0$) they follow exactly the same transformation. Only when ($s_m = 0$ & $d \geq 0$), the pair $[U(x),\ V(x)]$ follows a delayed (right shifted) transformation of the pair $[R(x),\ S(x)]$ but their relative difference of degree follows that of $[R(x),\ S(x)]$, (except in the initial iterations up to the first *overwriting* while the degree of polynomial $U(x) = 0$ does not change even though its all zero coefficients are being right shifted according to the same polynomial transformation).

A point must be highlighted. The δ iterations where d is incrementing always anticipates the δ iterations where d is decrementing. Hence, the coefficients of $U^{(*)}(x)$ are always right shifted or delayed, before those of $V(x)$ are being left shifted. This ensures that in the case of inversion the highest order coefficient of the result polynomial never reaches the mth position. On the other hand, as mentioned above, in the initial iterations up to the first *overwriting*, the polynomial $U(x) = 0$ always, hence there are no nonzero coefficients which may lag the lowest order coefficient of other polynomials. This is an important issue for pipelined computations in a systolic architecture. In all other iterations, the coefficients of $U(x)$ are spread over double delay elements discussed in Section 5.

Last, it must be noted that because of the similarities between Algorithms 1a and 2, Algorithm 5 can be used for inversion using triangular basis as well with the corresponding changes in the initialization step. An example of an inversion according to Algorithm 5 will be shown next.

4.6 An Example of Inversion with Polynomial Basis Input

In Table 1, the inverse of element $A(x) = 1 + x$ in GF(2^3) defined by $F(x) = 1 + x + x^3$ is $x + x^2$ which is returned in $U^{(6)}(x)$ if a restoring is applied. Otherwise, its shifted version by exactly m position is computed. In this example, the overwriting iterations are recognized at $i = 2$ and 5.

In the non-restoring case, $[U(x),\ V(x)]$ transformations follow that of $[R(x),\ S(x)]$. In the restoring one, at overwriting iteration and while $d < 0$, $V(x)$ is shifted similar to $S(x)$. Otherwise, $U(x)$ is divided by x while $V(x)$ is not shifted. Thus the relative degree of pair $[U(x),\ V(x)]$ follows that of pair $[R(x),\ S(x)]$ always. As counter d increases and decreases exactly m times, hence, after $2m$ iterations, the pair $[U(x),\ V(x)]$ is divided by exactly x^m.

The example in Table 1 is simple but rather a special case where the correct result is written in $U(x)$ exactly at $i = 2m$ iteration, since $d = 1$ at $i = 5$. In general this is not the case, and the last *overwriting* may occur at $i < 2m$, with $d > 1$. Afterward, the sign of d is

i	d	$R(x)$	$S(x)$	*non-restoring* $U(x)$	*non-restoring* $V(x)$	*restoring* $U(x)$	*restoring* $V(x)$
0	0	$1+x+x^3$	$1+x$	0	1	0	1
1	1	$1+x+x^3$	$x+x^2$	0	x	0	1
2	2	$1+x+x^3$	x^2+x^3	0	x^2	0	1
3	-1	x^2+x^3	$x+x^2+x^3$	x^2	x^3	$1=x^0$	x
4	0	x^2+x^3	x^2	x^2	x^3+x^4	x^0	$x+x^2$
5	1	x^2+x^3	x^3	x^2	x^4+x^5	x^{-1}	$x+x^2$
6	0	x^3	x^3	x^4+x^5	$x^3+x^5+x^6$	$x+x^2$	$x^0+x^2+x^3$

Table 1: Example of inversion over GF(2^3).

negated, and till it reaches zero no further *overwriting* occurs. Hence, Algorithm 5 does not modify $U(x)$ anymore by branching to the last **else if** case repeatedly. This ensures that the correct result is always returned after $2m$ iterations.

In a bit serial systolic architecture, a double delay element may be used to implement a stepwise $(U(x)/x)$ while the pair $[U(x),\ V(x)]$ are updated. In the next section, a class of bit-serial unidirectional architectures based on Algorithm 5 is proposed.

5 Bit Serial Unidirectional Systolic Architectures

In Figure 2, a generalized bit serial unidirectional counter-free systolic architecture for inversion and/or division over both polynomial and triangular basis is shown. In all the descriptions of this section we refer to this maximum $2m$ identical PE model. However, in the next section optimizations are described which result in fewer number of PEs. Note that no external synchronization latches in the systolic architecture of Figure 2 is shown; this is intentional since there are none.

There are three sets of inputs to each PE: the *datapath* consisting of $r,\ s,\ u,\ v,\ f$; the *state* consisting of *dseq*, *dec*, *update* and one *control* signal *start*. For division using polynomial basis (*i.e.*, Algorithm 1a), both r and f inputs to PE_0 consist of the coefficients of $F(x)$. On the other hand, for division using triangular basis (*i.e.*, Algorithm 2), the r input consists of coefficients of the monomial x^{2m-1}. For inversion the input f is not needed (used to perform modulo reduction in the division case only) and the input r is as described before. The highest order coefficients, degree m (or degree $2m-1$ for the triangular basis respectively), of polynomials enter first. In a bit serial systolic structure, each PE must be initialized once per each computation round. The *start* signal is used to set up the *state*

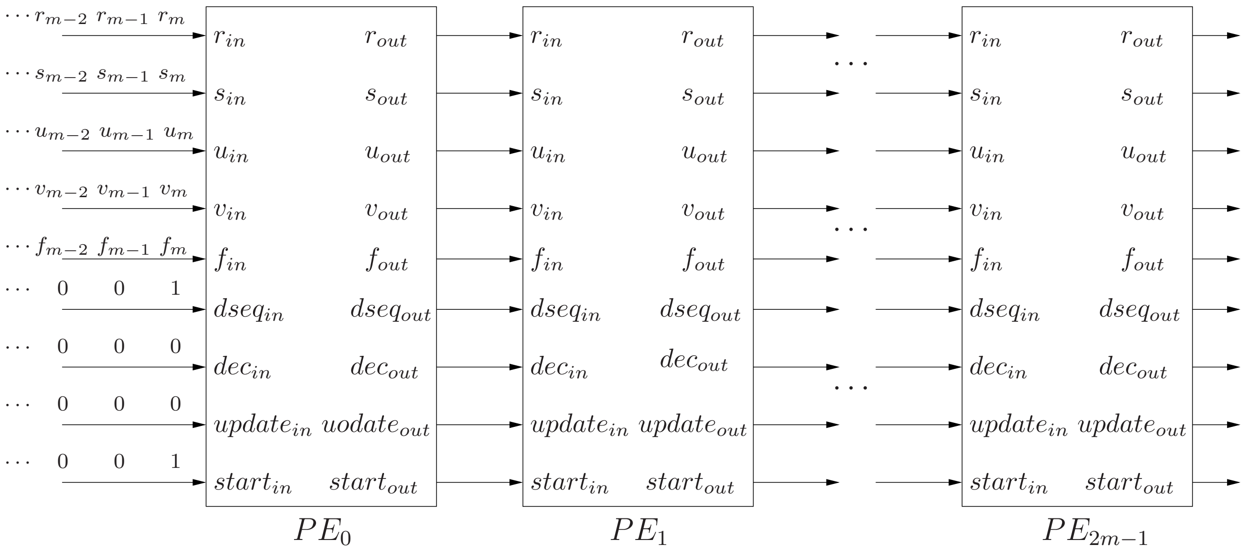

Figure 2: Bit serial unidirectional systolic architecture for Inversion/Division.

signals in this initialization cycle. The *start* is defined as a one following by m zeroes. It takes 2 cycles for the *start* signal to pass through each PE and after $4m$ cycles, the coefficient of the highest degree, *i.e.*, $m-1$, of the result synchronized with ($start = 1$) exits the last PE; $m-1$ more cycles and all the coefficients of the result are available. Thus, the latency is $5m-2$. The throughput of this bit serial architecture depends on the size of the inputs. In the polynomial basis inversion model, $R(x) = F(x)$, hence, 1 inverse per $m+1$ cycles, can be computed. In the triangular basis model where $R(x) = x^{2m-1}$, one inverse per $2m$ cycles is achievable. In the following some specifics of PE design for different cases with different inputs are presented.

5.1 Processing Element Design for Inversion

Figure 3 represents the single type PE based on Algorithm 5. Note that no external latches are shown in Figure 2. All the latches of the systolic structure are represented internal to a PE, shown as circles in Figure 3. In a serial-in systolic structure where each PE contains both the datapath and the control which uses latches as *state* and as operators (shifter, restoring transformation), it is more illustrative to describe all latches including output synchronization ones internally.

Two of the *control* signals, *update* and *dec* are precomputed and saved for the next PE. The *update* is the latched value of incoming $s_{in} = s_m$ at ($start = 1$). The precomputation of *update* serves to reduce the delay of the datapath. The sign of d as defined before must

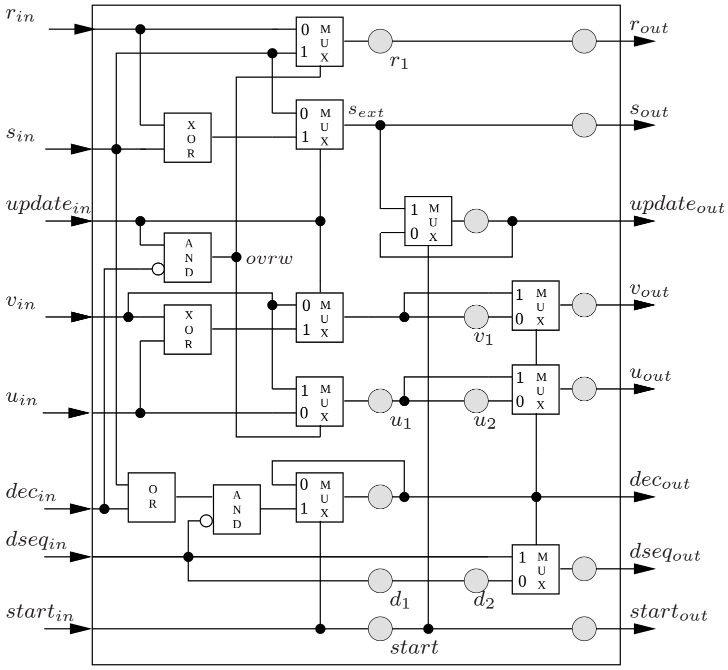

Figure 3: Processing element for the inverter.

be computed separately, and is called *dec* as described next.

In this work, a new serial-in counter-free implementation of the difference of degree computation is proposed. A mathematical formulation similar to [21] but simpler follows. In a serial-in counter-free implementation, the signed integer d may be represented by a sign bit, *dec*, and by a $(m+1)$-bit sequence *dseq* where $dseq = 2^{|d|}$. At *initialization* $d = 0$, hence, $dec = 0$, and $dseq = 00\cdots001$. At each shift of $S(x)$ while $dec = 0$, *dseq* is left shifted, that's a delay element is inserted. At overwriting iteration, *dec* is set, and *dseq* is right shifted, that's a delay element is bypassed. In all consecutive iterations until *dseq* returns back to 1, *dec* remains set. It will reset at $dseq = 1$. Hence, *dec* represents the signal to decrement the position of the single one in the *dseq* sequence. The *dec* computation circuit in Figure 3 represents a Boolean simplification of the set-reset mechanism described, exactly, $dec \leftarrow \overline{dseq_{in}} \; \& \; (dec_{in} \mid s_{in})$ where the overline represents a NOT logic operator, & an AND operator, and | an OR operator.

It is said that *dec* is needed to check for an *overwriting* event. An internal *state* signal $ovrw \leftarrow (update_{in} \; \& \; \overline{dec_{in}})$ is computed accordingly. It is not latched since both its inputs are latched. In Figure 3, a single delay element, r_1 causes the $xS(x)$ action. Double delay elements u_1, u_2 synchronized with v_1, while *dec* is set, provide the restoring action as described. On the other hand, *dec* selects whether the position of single 1 in the *dseq* to be left-shifted or right-shifted by inserting or bypassing double delay elements d_1 and d_2. The maximum gate delay of Figure 3 is

$$t_g = max(2T_{A2}, T_{X2} + T_{M2}) + T_{M2},$$

where T_{A2}, T_{X2} and T_{M2} represent the delay of a 2-input AND gate or OR gate, a 2-input XOR gate and a 2-input MUX gate respectively. On the other hand, this can be further shortened at the expense of two extra latches and the propagation of the internal *ovrw* signal. Specifically, the output of OR and AND gates on the dec_{in} path can be precomputed and two latches added: one on the path of *dec*, the other the *ovrw* signal, which are synchronized with all other outputs. In this case, the critical delay path will be as short as $t_g = T_{A2} + T_{M2}$. However, in general, such a reduction may not be a substantial improvement which justifies the trade-off. The critical delay path of an architecture required to compute the effective clock period depends not only on the gate delay shown above but also on the delay of the latches, interconnect and the clock network as well. In a standard synchronous design, the *setup* time of latches (flip-flop) and the clock skew may be much larger than single gate delay and may dominate the effective clock period especially where such reduced logic paths exist. In the following we only consider the gate delays which are architecture dependent and not the critical path delay.

5.1.1 Bit Serial Systolic Inverter in Triangular Basis

The bit serial systolic architecture of Figure 2 can be used to perform inversion in triangular basis promptly and without any modification or extra control signals. Specifically, all *state* and *control* signals are the same and only the datapath polynomials at input should be defined as in Algorithm 2: $R(x) = x^{2m-1}$; $S(x) = x^{2m-2}H(x^{-1})$; $U(x) = 0$; $V(x) = 1$. It can be noted that even though the number of PEs to compute the inversion in either triangular or polynomial basis input variants is the same, but $\deg(R(x))$, the largest input polynomial in the triangular basis, is almost twice as large as that in polynomial basis input. We will show that this fact does not change the latency but rather the throughput of this architecture.

Mathematically, according to Algorithm 2, $V(x)$ must be equal to 1. But from an implementation point of view this is not the most suitable value. In fact, the latency of this architecture depends on the number of PEs and the input $V(x)$. The number of PEs required for inversion in both polynomial and triangular basis is the same. But, initializing $V(x)$ with 1, while $R(x)$ is initialized with x^{2m-1} will increase the latency of this architecture with respect to a polynomial basis input by $m-1$ cycles. However a reduced latency is achievable. Since the result of the inversion with the triangular basis input according to

Algorithm 2 is always returned as a polynomial with degree at most $m-1$, in polynomial basis, then the first $m-1$ coefficients of the result, exiting highest order first, are always equal to zero. Also, it is easy to note that if the initial value of $V(x)=1$ is shifted by $m-1$, *i.e.*, $V(x)=x^{m-1}$, a shifted result $x^{m-1}U^{(i)}(x)$ is computed. The highest order coefficient of $x^{m-1}U^{(i)}(x)$ exits the $2m$th PE at $4m$th cycle synchronized with ($start=1$). In this case the latency of the architecture for both polynomial and triangular basis will be the same. This is also convenient since no control circuitry at system level should be changed. On the other hand, in a triangular basis input with $\deg(R(x))=2m-1$, the number of cycles per inversion is $2m$.

5.2 Processing Element Design for Division

It is said that Algorithm 1 can also be used for computing division over GF(2^m) by initializing its $U^{(0)}(x)$ with dividend instead of 1 and at the cost of adding a modulo reduction to the $U^{(i)}(x)$ computation. This is true for Algorithms 1a, 2 and 5 as well.

In Algorithm 5, in order to perform a division, at initialization, $V^{(0)}(x)$ is initialized with the dividend. Next, at each iteration where a new value of $V^{(i)}(x)$ is computed, specifically at overwriting iteration and while $d<0$, a modulo reduction may be needed. Let us point out that, in Algorithm 5, $U^{(i)}(x) \leftarrow U^{(i-1)}(x)/x$ is used only for restoring and never requires a modulo reduction. Up to the first overwriting its value is zero, and always the number of division of $U(x)$ is compensated by number of multiplication of $V(x)$ at overwriting or while $d<0$. In Table 1, rows 5 and 6, in the restoring case, show an example why no modulo reduction on $U^{(5)}(x)$ is required and its shifted (compensated) value is used for next $V(x)$ computation.

The *reduction* condition for the division algorithm occurs when both the leading coefficient of $V(x)$, *i.e.*, $v_{in}=v_m=1$ and also $dec_{in}=1$, *i.e.*, $reduce = v_{in}\ \&\ dec_{in}\ \&\ start_{in}$. Then $\underline{V}^{(i)}(x) \leftarrow V^{(i)}(x)+F(x)$, where $+$ is an XORing with the underline representing a modulo reduced representation.

Figure 4 shows as augmented version of Figure 3 with added circuitry to check the *reduction* condition, and to perform modulo reduction accordingly. However a direct hardware mapping of the conditional computation of $\underline{V}^{(i)}(x)$, as defined above, generates a long gate delay path. In order to reduce the gate delay path, at the expense of extra logic but not latches, a balanced delay path for the input of three latches u_1, v_1, and $reduce_{lat}$ is computed, assuming similar delays for AND and XOR gates, as shown in Figure 4. Accordingly, the maximum gate delay is *max* $(3T_{A2},\ 2T_{A2}+T_{X2},\ T_{A2}+2T_{X2})$ $+2T_{M2}$, where T_{A2}, T_{M2} and T_{X2} were defined before. In the *max* function the first term represents the path from dec_{in} to

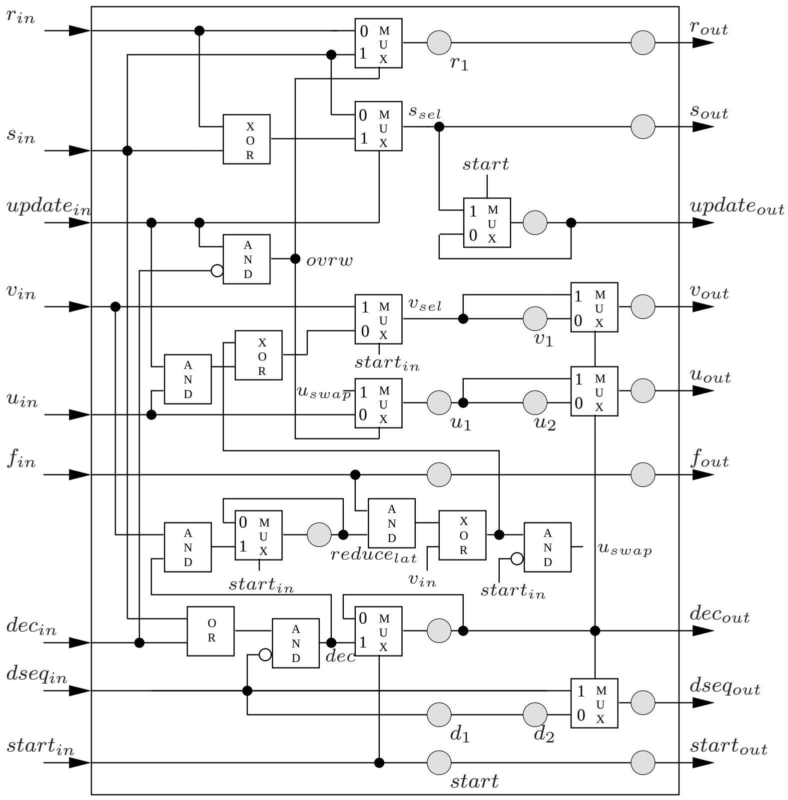

Figure 4: Processing element for the divider.

$reduce_{lat}$ and the second and third that from $reduce_{lat}$ to delay elements v_1, u_1 respectively.

5.2.1 Bit Serial Systolic Divider in Triangular Basis

The systolic architecture based on the divider PE of Figure 4 can perform division over triangular basis as well. Once again, the inputs are defined as in Algorithm 2 with the exception of $V(x)$ which will be the dividend, plus the extra input $F(x)$.

It is interesting to note that $F(x)$, the field defining polynomial, is the same used for the polynomial basis divider. The dividend must be in polynomial basis as well. In fact, at each *reduction* condition as defined above, the intermediate $\underline{V}^{(i)}(x) \leftarrow V^{(i)}(x)+F(x)$ computation is assumed to be done in polynomial basis.

Also, with a reasoning similar to Section 5.1.1, to reduce the latency and to have a common system control for the divider both in the polynomial and triangular basis, a shifted version of the dividend $V(x)$, *i.e.*, $x^mV(x)$ is more suitable to be used. Further, this shifted version $V(x)$ must be used and be synchronized with $F(x)$ coefficients, if f_m enters the first PE at ($start = 1$). Otherwise both should be shifted accordingly. In the former case, with the same reasoning as before, the shifted result, *i.e.*, $x^m U(x)$, is returned starting at the $4m$th cycle. Hence, the latency of the triangular basis divider would be the same as triangular basis inverter which is $5m - 2$.

6 Generalization, Optimization and Complexity Issues

The unidirectional bit serial systolic architecture described in Figure 2 not only is independent of the defining irreducible polynomial of the field but also it can compute the inverses in binary fields of any dimension. Two aspects of this assertion are discussed next.

6.1 Universal Bit Serial Systolic Inverter/Divider

This architecture due to its unidirectionality may process any field size up to the size defined by the number of the PEs where no external buffering is available. Let us recall the *exit* condition as defined in Section 4. It is said that in systolic architectures the *exit* condition can be checked implicitly by a mapping to an exact number of PEs. Now we generalize this assertion. Let N represents the number of PEs in the architecture. In a bit serial unidirectional systolic architecture as described, it is easy to see that if $N \geq 2m$, where m represents the field dimension, the coefficients of the inverse element start to appear at the output of the $2m$th PE after $4m$ cycles. Hence, using the architecture in Figure 2, it is possible to process any field dimension $m \leq N/2$ as far as the correct result can be captured at the right cycle. Here two such schemes are described.

Either the output of the $2m$th PE should be accessible directly, or a mechanism should be in place to de-activate all remaining PEs from $2m + 1$ to N, and to capture the result at the output of Nth PE always. The former does not require any inner PE modification and provides the optimal latency. One such scheme using a selector circuitry over a common tristate output line is shown in Figure 5.

One of the selector latches can be set such that only the specific output corresponding to this latch is passed to the common output, and all other outputs are kept in tristate. Figure 5 represents a serial-in mechanism to set the latch corresponding to the field dimension. A parallel load is possible as well, since the serial-in *selector* must not be used synchronized with *start*. In fact, the given latch is set only once per each change of field dimension. Note that only the output of every other PE requires a selector latch.

The latter provides a uniform, even though not optimal, latency and delay, and it does not require any external circuitry. This is possible if a de-activation mechanism based on the *exit* condition checking is included within each PE. By the de-activation we mean to prevent

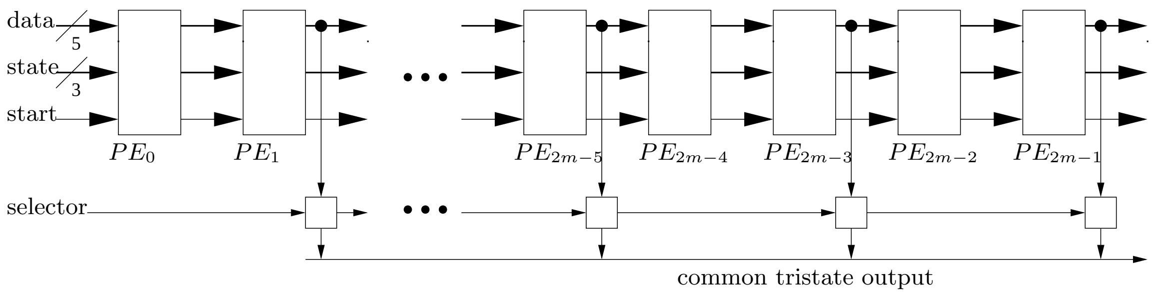

Figure 5: Variable dimension divider using selectors over common tristate output.

the ($start = 1$) signal to propagate to the next PE which means this PE is not initialized, and two *state* signals, *dec* and *update* remain zero which cause *ovrw* stays zero as well. In this case, $u_{out} \leftarrow u_{in}$ every three cycles according to Figures 3 or 4.

For the latter case itself two solutions exit. Either using an explicit *control* or *state* signal to act as a distributed counter-like structure to count the total number of shifts of initial polynomial $R^{(0)}(x)$, or at the expense of extra logic, to use a modified *start* signal to act both as synchronization and a down counter for the total number of shifts. In either case, the $u_{out} \leftarrow u_{in}$ path must be modified (an added MUX) to occur every other cycle to provide a uniform latency for any field dimension.

6.2 Trading off Throughput for Area

If some external buffering is used, it is easy to see that most bit serial systolic architectures including the one proposed here are capable to perform inversion/division over variable size fields. In this case, at the expense of throughput, the same PEs are initialized more than once for each inversion computation to process different blocks of data. A limiting case is of particular interest where zero external buffering is used. Once this is discussed a generalization may follow.

In Section 4, it is assumed that $2m$ PEs are required to perform the inversion but this is a case where a minimum number of $(m + 1)$ cycles per inversion is a design requirement. On the other hand, if this constraint is relaxed, and no back to back inversion is required, it is possible to use exactly m PEs to perform an inversion without any external buffering. This is achievable through a MUXing mechanism at the input of the first PE. Let us recall, that the largest input polynomial in the case of triangular basis inversion/division is $R(x) = x^{2m-1}$, and more importantly, in the worst case of a polynomial or triangular basis inversion/division, all coefficients of polynomials require $2m$ cycles at most to spread over m PEs. Hence, at the $2m$th cycle, by the time the ($start = 1$) signal exits the mth PE, all input coefficients have already entered the first PE. In the next cycle, *i.e.*, $2m+1$, it is possible to feedback the outputs of the mth PE synchronized with ($start = 1$) into the first PE which will be re-initialized for a second time. In this case, throughput is sacrificed, and the input of a new set of input should be delayed by $2m$ cycles either in polynomial or triangular basis case. A generalization of this scheme using external buffering elements and further lowering the number of PEs is possible. Multiple alternatives exits whether the number of PEs used are an exact divisor of m or not. However, this is not covered here.

6.3 Area and Latency Optimization without Throughput Loss

A slight optimization in area and latency of the architecture in Figure 2 is achievable without degrading throughput. In Sections 4, 5, it is always assumed that the highest order coefficient of polynomials to be processed are the mth for the polynomial, and the $(2m-1)$th for the triangular basis. In fact, among all data inputs of Algorithm 5 only $R(x)$ has a nonzero coefficient of this degree. As a result the inputs to the first PE of the proposed architecture is always the same, most importantly, $s_{in} = s_m = 0$, $dseq_{in} = 1$, and according to Algorithm 5, the first PE always branches to same **else if** ($s_m = 0$ & $d \geq 0$). Hence, it is easy to compute the output of PE_0, and initialize the algorithm with inputs of PE_1 directly. Specifically, a new architecture with only $2m-1$ PEs with the following inputs to its new first PE can be devised: $start = 1$, $update = 1$, $dec = 0$, $dseq = 00\cdots010$, f_m, and data inputs r_m, s_{m-1}, v_{m-1}, u_{m-1} for the polynomial basis, or r_{2m-1}, s_{2m-2}, v_{2m-2}, u_{2m-2} for the triangular basis model. Hence, r, u, f, $start$, and dec are not changed (u is initially always zero). Also, s_m and v_m (or s_{2m-1}, v_{2m-1} in the triangular basis) are always zero. Thus, only two *state* signals *dseq* and *update* will have modified values. In this $2m-1$ PE architecture, the latency is reduced to $5m-4$. Since most systolic proposals for inversion/division use such a model, we refer to this optimized model for comparison purposes.

6.4 Implementation Results

Exhaustive test patterns for values of m up to thirteen for inputs and outputs of either polynomial or triangular basis case using C models are generated. Specifically, the triangular basis conversion, Hankel matrix formation and initialization vectors for the triangular basis inverter/divider with their corresponding outputs are modeled in C. Next, eight variants of the architecture (inverter and divider, basic or optimized, polynomial and triangular) are coded in Verilog-HDL language which has been simulated and verified. The Verilog model has been synthesized using Synopsys with a standard CMOS 0.18 library for a clock period of 1.1ns for the inverter and 1.4ns for the divider. The average setup time of the flip-flops was 0.32ns and the clock skew set at 10%. The area reports, including the flip-flops, are equivalent to 90 and 120 2-input NAND gates for each inverter and divider of Figure 3, 4. It should be pointed out that non-combinational area, as defined by Synopsys, is the dominant part. For the divider, it is more than 75%, and for the inverter more than 80%.

7 Comparison

Among many systolic inversion algorithms, some of them are based on variants of EEA algorithm [8, 7, 5, 13, 21, 23]. Here more recent and relevant ones to this work are mentioned and compared. Please note, systolic architectures which can compute more than inversion/division, in particular those performing $AB^2 + C$ are not compared here. Clearly, they have very poor AT complexity figures with respect to our proposal. In the following, as before, T_{X2}, T_{M2}, T_{A2} represent a 2-input XOR, 2-input MUX and a 2-input AND/OR delay time, respectively.

In [8], Guo and Wang have proposed a systolic implementation of a modified Brunner et al. [2] inversion algorithm based on a variant of EEA. They enhanced that algorithm by eliminating the asymmetric looping due to alternating division or multiplication of the algorithm to solve the shifted result problem. This modification allows the possibility of a regular systolization of the algorithm at the expense of extra circuitry to restore the shifted degree of resulting inverse value. This bit serial proposal required 3 type of cells and a bit reversing buffer (FILO).

In [7], Guo and Wang have proposed an improved version of an in-place parallel-in parallel-out systolic inverter/divider. It is a row of 2 types of cells with bidirectional signals. This proposal requires less number of cells and iteration loops since no extra steps for degree restoring is required. The trade-off is the use of an extra polynomial and an extra control signal. The reported area complexity is said to be $O(m)$ but in fact it is $O(m \log_2 m)$ due to inner cell adder/subtracter.

In [5], Guo and Wang further improved the bit serial implementation of the above design by introducing a single cell type unidirectional systolic design. Once again the shifted result problem is solved by using an extra polynomial as above. Also, in this design a distributed degree tracking control mechanism is introduced. This implementation has comparable complexity figures as the one described in this work.

In [13], Huang and Wu proposed a variant of EEA which does not require any *swap* action but the number of iterations can be asymptotically $O(m^2)$. They implemented a systolic parallel-in parallel-out inverter with m^2 rows based on it which requires a pre-computed value of α^{-m} where α is a root of the irreducible polynomial to solve the shifted result problem.

Their design is unidirectional and they proposed a C-testable fault-tolerant structure.

In [21], Wu et al. proposed a parallel-in parallel-out systolic divider based on a modified Stein's algorithm (also known as binary GCD). It has an area-time (AT) complexity of $O(m^2)$. It requires 3 types of cells, and its maximum gate delay is $T_{A2} + T_{X3}$. This design is a counter free structure.

In [23], Yan and Sarwate proposed two parallel-in systolic inversion architectures, centralized and distributed control, based on a modified EEA. The distributed control architecture which has a bidirectional degree tracking mechanism requires 2 control signals and a $2m$ bits register, but has only two gates on its critical path. These are inversion only architectures.

Table 2 shows a comparison of the divider with PE of Figure 4 and the inverter with the PE of Figure 3 with two representative designs reviewed above. The design in [5] is chosen since it is a divider with best overall characteristics and also in the same category as ours (bit serial unidirectional counter free systolic using variants of EEA). On the other hand, a same category inverter (which only does the inversion) could not be found. The second design [8] is chosen to show why single row parallel-in designs in particular when they use counter-like structures are so poor for large values of m.

8 Conclusion

In this article a class of bit serial unidirectional systolic architectures for multiplicative inversion and division over finite field with input either in polynomial or triangular basis representation has been presented. The complexity issues related to the initialization step of an efficient triangular basis inversion as described in [10] is investigated. A variant of EEA optimized for counter-free unidirectional systolization is devised. It is shown that the same algorithm and architecture can be used to perform inversion and division where the inputs are either in polynomial or triangular basis. Main contribution is the restoring mechanism introduced in Algorithm 5 and its implementation using double delay elements. Also, a simpler distributed counter structure is proposed. This architecture is suitable for applications where the field dimension may be large or variable (such as cryptographic applications).

This bit serial structure can be transformed into a digit serial one, by folding techniques as used in [6], if higher throughput is a requirement. On the other hand, if throughput is not a major requirement, the alternative designs with a number of PEs less than m using external buffering can be explored. Algorithm 5 and the architecture based on it can be easily transformed to compute inversion based on binary GCD (Stein's) method where lowest order coefficients are processed first. This architecture is well suitable for extremely high clock rates. It is seen that delay and area are highly dependent on the delay elements used. Delay

Circuits	Guo et al. [8][a]	Guo et al. [5][b]	Figure 4	Figure 3
Time Complexity	$O(m)$	$O(m)$	$O(m)$	$O(m)$
Area Complexity	$O(m \log_2 m)$	$O(m)$	$O(m)$	$O(m)$
# of cycles per inversion/division	m	$m+1$	$m+1$[c]	$m+1$
Latency	$8m-1$	$5m-4$	$5m-4$[d]	$5m-4$
IO pins	$O(m)$	10	9	8
Unidirectional	yes	yes	yes	yes
Max Gate Delay	$2T_{A2}+T_{X3}$ $+2T_{M2}$	$T_{A2}+2T_{X2}$ $+T_{M2}$	$2T_{A2}+T_{X2}$ $+2T_{M2}$	$2T_{A2}+T_{M2}$
Single Cell	no	yes	yes	yes
Gate Count:	total[e]	per cell	per cell	per cell
IO Latch[f]	$4m$	-	-	-
inner Latch[g]	$46m+$ $4m\lceil \log_2 m+1 \rceil$	19	18	15
MUX	$35m+2$	21	10	9
XOR	$11m$	5	3	3
AND/OR	$26m$	7	7	3
others	adder, zero-check	-	-	-

[a]single row parallel-in architecture, with adder
[b]counter-free, distributed control
[c]for polynomial basis input
[d]for a $2m-1$ PEs architecture
[e]different type of cells
[f]IO Synchronization and Reordering
[g]inner and inter cell latches

Table 2: Comparison of bit serial systolic dividers and inverter.

or area optimization by using special latch design may be explored further as well.

References

[1] E.R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill Book Company, 1968.

[2] H. Brunner, A. Curiger, and M. Hofstetter. “On Computing Multiplicative Inverses in GF(2^m)”. *IEEE Transactions on Computers*, 42(8):1010–1015, Aug. 1993.

[3] S.T.J. Fenn, M. Benaissa, and D. Taylor. "Bit-serial Dual Basis Systolic Multipliers for GF(2^m)". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'95*, 3:2000–2003, 1995.

[4] R. Furness, M. Benaissa, and S.T.J. Fenn. "GF(2^m) Multiplication over Triangular Basis for Design of Reed-Solomon Codes". *IEE Proceedings. Computers and Digital Techniques*, 145(6):437–443, Nov. 1998.

[5] J.-H. Guo and C.-L. Wang. "Bit-serial Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in $GF(2^m)$". *Proceedings of Technical Papers, International Symposium on VLSI Technology, Systems, and Applications*, pages 113–117, June 1997.

[6] J.-H. Guo and C.-L. Wang. "Novel Digit-serial Systolic Array Implementation of Euclid's algorithm for Division in $GF(2^m)$". *Proceedings of the IEEE International Symposium on Circuits and Systems, 1998. ISCAS'98*, 2:478–481, 1998.

[7] J.-H. Guo and C.-L. Wang. "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)". *IEE Proceedings Computers and Digital Techniques*, 145(4):272–278, July 1998.

[8] J.-H. Guo and C.-L. Wang. "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)". *IEEE Transactions on Computers*, 47(10):1161–1167, Oct. 1998.

[9] A. Halbutogullari and C.K. Koc. "Mastrovito Multiplier for General Irreducible Polynomials". *IEEE Transactions on Computers*, 49(5):503–518, May 2000.

[10] M.A. Hasan. "Double-Basis Multiplicative Inversion over GF(2^m)". *IEEE Transactions on Computers*, 47(9):960–970, Sep. 1998.

[11] M.A. Hasan. "Efficient Computation of Multiplicative Inverses for Cryptographic Applications". *Proceedings of the 15th. IEEE Symposium on Computer Arithmetic, Arith-15*, pages 66–72, June 2001.

[12] M.A. Hasan and V.K. Bhargava. "Architecture for a Low Complexity Rate-Adaptive Reed-Solomon Encoder". *IEEE Transactions on Computers*, 44(7):938–942, July 1995.

[13] C-T. Huang and C-W. Wu. "High-Speed C-Testable Systolic Array Design for Galois-field Inversion". *Proceedings of the European Design and Test Conference*, Paris:342–346, March 1997.

[14] T. Itoh and S. Tsujii. "A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Basis". *Information and Computation*, 78:171–177, 1988.

[15] J.V. McCanny, R.A. Evans, and J.G. McWhirter. "Use of Unidirectional Data Flow in Bit-level Systolic Array Chips". *Electronics Letters*, 22:540–541, May 1986.

[16] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[17] A.J. Menezes, P. van Ooreschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[18] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. "Fast Key Exchange with Elliptic Curve Systems". *Advances in Cryptology, EUROCRYPT '95, LNCS 921, Springer-Verlag*, pages 43–56, 1995.

[19] G. Seroussi. "Table of Low-weight Irreducible Polynomials over F_2". *Hewlett-Packard Laboratories Technical Report No. HPL-98-135*, August 1998.

[20] Y. Watanabe, N. Takagi, and K. Takagi. "A VLSI Algorithm for Division in GF(2^m) Based on Extended Binary GCD Algorithm". *Transactions of The Institute of Electronics, Information and Communication Engineers, 2002. IEICE'02*, E85(5):994–999, 2002.

[21] C-H. Wu, C-M. Wu, M-D. Shieh, and Y-T. Hwang. "Systolic VLSI Realization of a Novel Iterative Division Algorithm over GF(2^m): A High Speed Low-Complexity Design". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'01*, 4:33–36, 2001.

[22] C-H. Wu, C-M. Wu, M-D. Shieh, and Y-T. Hwang. "An Area-Efficient Systolic Division Circuit over GF(2^m) for Secure Communication". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'02*, 5:733–736, 2002.

[23] Z. Yan and D.V. Sarwate. "Systolic Architectures for Finite Field Inversion and Division". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'02*, V:789–792, June 2002.