Systolic Design Space Exploration of EEA-BASED INVERSION Over Binary and Ternary Fields

by

Ibrahim Hazmi
B.Sc., King Saud University, 2001
M.Sc., RMIT University, 2009

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Contents

List of Tables

List of Figures

Chapter 2

Finite Field Arithmetic Background

Finite field arithmetic is a branch of algebra that has diverse applications in many types of computers and digital communication systems. For example, computing point operations in Elliptic curve cryptographic systems requires different field operations, such as addition, subtraction, multiplication, squaring and division. TABLE 2.1 provides an example that illustrates the cost of point addition and doubling implementation, in terms of the required field operations, i.e., inversion (I), multiplication (M), squaring (S), and Cubing (C) [40, 41].

Table 2.1: Table of costs for different point operations

Operation	**GF(p) cost**	**Binary field cost**	**Ternary field cost**
P+Q	1I+2M+1S	1I+2M+1S	1I+2M+1S+0C
2P	1I+2M+2S	1I+2M+1S	1I+2M+1S+0C
2P +Q	1I+9M+2S	1I+9M+2S	-
3P	1I+7M+4S	1I+7M+4S	1I+4M+2S+5C
3P +Q	2I+9M+4S	2I+9M+3S	-
4P	1I+9M+9S	1I+8M+5S	-

Different algorithms with several techniques and architectures were proposed to perform inversion over finite field. In this research, the main concern is the implementation of finite field inversion using extended Euclidean algorithm. This chapter deliberates a brief background about related topics such as finite fields and the issue of post-quantum field arithmetic, in particular. Moreover, it revisits modular arithmetic over GF(p) including multiplication and division, in addition to introducing GF(3) representation. Furthermore, it presents polynomial arithmetic over GF(p^m), including Euclidean algorithm and EEA-based inversion. Finally, parallel computing and systolic architectures are exhibited at the end of this chapter.

2.2.1 Modular Arithmetic

Arithmetic operations between coefficients, e.g., $[(a+b) \mod p]$, $[(a-b) \mod p]$ and $[(a \times b) \mod p]$, are examples of modular arithmetic over GF(p). Operations over GF(2), GF(3), GF(5) and GF(7) are summarized in TABLES 2.2, 2.3, 2.4 and 2.5, respectively. Note that modular division (a/b) over GF(p) is defined as follows [62]:

$$a/b = a \times b^{-1} \mod p \tag{2.1}$$

GF(p)-division is declared accordingly in each table. It can be observed that GF(2)-addition and subtraction are equivalent to bitwise XOR, while GF(2)-multiplication is an AND gate. Meanwhile, division can be treated evenly with multiplication in GF(2) and GF(3).

Table 2.2: GF(2) Arithmetic

GF(2)-Addition

$a+b$		b = 0	b = 1
a	0	0	1
	1	1	0

GF(2)-Subtraction

$a-b$		b = 0	b = 1
a	0	0	1
	1	1	0

GF(2)-Multiplication

$a \times b$		b = 0	b = 1
a	0	0	0
	1	0	1

GF(2)-Division

$a/b =$ $a \times b^{-1}$	b	b = 0	b = 1
	b^{-1}	–	1
a	0	–	0
	1	–	1

Table 2.3: GF(3) Arithmetic

GF(3)-Addition

$a+b$		b = 0	b = 1	b = 2
a	0	0	1	2
	1	1	2	0
	2	2	0	1

GF(3)-Subtraction

$a-b$		b = 0	b = 1	b = 2
a	0	0	2	1
	1	1	0	2
	2	2	1	0

GF(3)-Multiplication

$a \times b$		b = 0	b = 1	b = 2
a	0	0	0	0
	1	0	1	2
	2	0	2	1

GF(3)-Division

$a/b =$ $a \times b^{-1}$	b	b = 0	b = 1	b = 2
	b^{-1}	–	1	2
a	0	–	0	0
	1	–	1	2
	2	–	2	1

2.2.2 Polynomial Arithmetic

Finite fields can be represented using polynomial, GNB, ONB, or dual bases. The elements of GF(p^m) are the polynomials of degree $m-1$, whose coefficients are in the field GF(p). In polynomial basis representation, an irreducible polynomial $G(x)$ of degree m, which cannot be factored as two polynomials, is defined over GF(p) as follows:

$$\boldsymbol{G}(x) = \sum_{i=0}^{m} g_i x^i, \quad \text{where } g_i \in \{0, 1, \cdots p-1\} \text{ and } g_m, g_0 \neq 0.$$

It is argued that choosing irreducible polynomial of low number of nonzero coefficients can result in more efficient implementation of the polynomial arithmetic. Therefore, it is suggested to be an irreducible trinomial, pentanomial, or equally-spaced polynomials. Note that for brevity, polynomials such as $\boldsymbol{G(x)}$ will be written as $\boldsymbol{G}$. Whereas, basic arithmetic operations and Boolean functions will be denoted as follows: addition ($+$), subtraction ($-$), multiplication ($\times$), AND ($\cdot$), OR ($|$), XOR ($\oplus$).

Polynomial addition and subtraction can be performed by adding/subtracting the corresponding coefficients. Consider the irreducible polynomial $\boldsymbol{G}$ and assume two polynomials $\boldsymbol{A}$ and $\boldsymbol{B}$ of degrees n and o as follows:

$\boldsymbol{A} = \sum_{i=0}^{n} a_i x^i$ and $\boldsymbol{B} = \sum_{i=0}^{o} b_i x^i$, where $o \leq n < m$, $\quad a_i, b_i \in \{0, 1, \cdots p-1\}$ and $a_n, b_o \neq 0$.

Addition/Subtraction can be defined as follows:

$$\boldsymbol{A} \pm \boldsymbol{B} = \sum_{i=0}^{o}((a_i \pm b_i) \mod p)x^i + \sum_{i=o+1}^{n} a_i x^i \tag{2.5}$$

Considering $o = n$, polynomial multiplication can be defined as follows:

$$\boldsymbol{A} \times \boldsymbol{B} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}((a_i \times b_i) \mod p)x^{i+j} \mod \boldsymbol{G} \tag{2.6}$$

At least, n^2 multiplications are required in order to multiply these two polynomials [62, 72]. If $n \geq m/2$, the product degree will be $> m$, which will be reduced. In this dissertation, modular and polynomial divisions are distinguished and denoted as ($/$) and ($\div$), respectively. Modular division, $\boldsymbol{B}/\boldsymbol{A}$, is defined as $\boldsymbol{B} \times \boldsymbol{A}^{-1} \mod \boldsymbol{G}$, where $\boldsymbol{A}^{-1}$ is the multiplicative inverse of $\boldsymbol{A}$ modulo $\boldsymbol{G}$. While, polynomial division, $\boldsymbol{G} \div \boldsymbol{A}$, produces $\boldsymbol{Q}$ of degree $(m-n)$, and $\boldsymbol{R}$ of degree $(< n)$, such that [2, 73]:

$$\boldsymbol{Q} = \sum_{i=0}^{m-n} q_i x^i, \ \boldsymbol{R} = \sum_{i=0}^{n-1} r_i x^i \quad \text{and} \quad \boldsymbol{G} = \boldsymbol{Q} \cdot \boldsymbol{A} + \boldsymbol{R}$$

The reduction in Equation (2.6) is finding the remainder of polynomial division, $(\boldsymbol{A} \times \boldsymbol{B}) \div \boldsymbol{G}$.

2.2.2.1 Polynomial Arithmetic over Binary Fields

Assume binary polynomials $\boldsymbol{G}$, $\boldsymbol{A}$ and $\boldsymbol{B}$ of degrees m, n and o, where $o \leq n < m$. If $o = n$, polynomial addition/subtraction and multiplication can be defined as follows:

$$\boldsymbol{A} \pm \boldsymbol{B} = \sum_{i=0}^{n} (a_i \oplus b_i) x^i \tag{2.7}$$

$$\boldsymbol{A} \times \boldsymbol{B} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (a_i \cdot b_i) x^{i+j} \mod \boldsymbol{G} \tag{2.8}$$

$$\text{where} g_i, a_i, b_i \in \{0, 1\} \text{ and } g_m, g_0, a_n, b_o = 1.$$

Let $\boldsymbol{G} = x^8+x^5+x^4+x^3+1$, $\boldsymbol{A} = x^5+x^4+x+1$ and $\boldsymbol{B} = x^5+x^4+x^3+x^2+1$ be binary polynomials. TABLE 2.7 presents polynomial addition/subtraction and multiplication. Whereas, Fig. 2.1 demonstrates GF(2^m) polynomial division, where $\boldsymbol{Q} = x^3+x^2+x$, and $\boldsymbol{R} = x^3+x+1$.

Table 2.7: Polynomial Addition and Multiplication in GF(2^m)

G	$x^8 + x^5 + x^4 + x^3 + 1$	**1 0 0 1 1 1 0 0 1**
A	$x^5 + x^4 + x + 1$	- 0 0 **1 1 0 0 1 1**
B	$x^5 + x^4 + x^3 + x^2 + 1$	- 0 0 **1 1 1 1 0 1**
A±B	$x^3 + x^2 + x$	- 0 0 **0 0 1 1 1 0**
A×B	$(x^{10} + x^5 + x^4 + x^2 + x + 1) \mod \boldsymbol{G}$	
	$x^7 + x^6 + x^4 + x + 1$	- **1 1 0 1 0 0 1 1**

						1	**1**	**1**	**0**	**=Q**	$m-n=3$
A= 51_{10}	①	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**G**= 313_{10}
A× **1**						⊕ **1**	**1**	**0**	**0**	**1**	1 0 0 0	<< 3
						0	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	
A× **1**						⊕ 0	**1**	**1**	**0**	**0**	**1**	**1**	0 0	<< 2
						0	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	
A× **1**						⊕ 0	0	**1**	**1**	**0**	**0**	**1**	**1**	0	<< 1
						0	0	**0**	**0**	**0**	**1**	**0**	**1**	**1**	
A× **0**						⊕ 0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	<< 0
						0	0	0	0	**0**	**1**	**0**	**1**	**1**	**=R**

Figure 2.1: GF(2^m) Polynomial Division Example ($\boldsymbol{G} \div \boldsymbol{A}, m = 8, n = 5$)

Note that elements of $\boldsymbol{Q}$ are produced by performing modular division ($\boldsymbol{g_m}/\boldsymbol{a_n}$), where $\boldsymbol{a_n}$ is the Most Significant Bit (MSB) of $\boldsymbol{A}$ and $\boldsymbol{g_m}$ is the MSB of $\boldsymbol{G}$ or XOR result.

2.2.2.4 The Euclidean Algorithm

Euclidean algorithm computes the greatest common divisor (gcd) of two polynomials, e.g., $\boldsymbol{G}$ and $\boldsymbol{A}$, based on the notion that:

$$\gcd[\boldsymbol{G}, \boldsymbol{A}] = \gcd[\boldsymbol{A}, (\boldsymbol{G} \mod \boldsymbol{A})] \tag{2.14}$$

As mentioned previously, the modulo operation ($\boldsymbol{G} \mod \boldsymbol{A}$) is basically polynomial division, where only remainder $\boldsymbol{R}$ is targeted. Euclidean algorithm repeats this modulo operation until the remainder becomes *zero*, e.g., $\boldsymbol{R}_{(n+1)} = 0$, which means $\boldsymbol{R}_n = \boldsymbol{R}_{(n-1)} =$. Then, $\gcd[\boldsymbol{G}, \boldsymbol{A}] = \boldsymbol{R}_n$, which complies with the following set of equations [62]:

$$\left.\begin{array}{cc} \boldsymbol{G}=\boldsymbol{Q}_1\cdot\boldsymbol{A}+\boldsymbol{R}_1 & 0<\boldsymbol{R}_1<\boldsymbol{A} \\ \boldsymbol{A}=\boldsymbol{Q}_2\cdot\boldsymbol{R}_1+\boldsymbol{R}_2 & 0<\boldsymbol{R}_2<\boldsymbol{R}_1 \\ \boldsymbol{R}_1=\boldsymbol{Q}_3\cdot\boldsymbol{R}_2+\boldsymbol{R}_3 & 0<\boldsymbol{R}_3<\boldsymbol{R}_2 \\ \cdots & \cdots \\ \boldsymbol{R}_{n-2}=\boldsymbol{Q}_n\cdot\boldsymbol{R}_{n-1}+\boldsymbol{R}_n & 0<\boldsymbol{R}_3<\boldsymbol{R}_2 \\ \boldsymbol{R}_{n-1}=\boldsymbol{Q}_{n+1}\cdot\boldsymbol{R}_n+\boldsymbol{0} & \gcd[\boldsymbol{G},\boldsymbol{A}]=\boldsymbol{R}_n. \end{array}\right\}$$

2.2.2.5 The Extended Euclidean Algorithm (EEA)

The EEA algorithm is an extension of the Euclidean algorithm, which finds the multiplicative inverse of $\boldsymbol{A}$ modulo $\boldsymbol{G}$. In addition to computing the $\gcd[\boldsymbol{G}, \boldsymbol{A}]$, it produces the coefficients of Bézout's identity; $\boldsymbol{S}$ and $\boldsymbol{T}$ such that:

$$\gcd[\boldsymbol{G}, \boldsymbol{A}] = (\boldsymbol{S} \cdot \boldsymbol{A} + \boldsymbol{T} \cdot \boldsymbol{G}) \mod \boldsymbol{G} = 1$$

Sine $(\boldsymbol{T} \cdot \boldsymbol{G}) \mod \boldsymbol{G} = 0$, $\boldsymbol{S} \cdot \boldsymbol{A} = 1 \mod \boldsymbol{G}$. Therefore [2]:

$$\boldsymbol{S} = \boldsymbol{A}^{-1} \mod \boldsymbol{G} \tag{2.15}$$

The multiplicative inverse A^{-1} can be computed only if $\boldsymbol{A}_{deg} < \boldsymbol{G}_{deg}$ and $\gcd[\boldsymbol{G}, \boldsymbol{A}] = 1$. The fast EEA scheme and its variants, e.g., Extended Stein's Algorithm (ESA) and Almost Inverse Algorithm (AIA), have the best known asymptotic complexity estimate for inversion. Therefore, they are highly preferred, especially in applications where inversion is used frequently [10, 12]. TABLE 2.10 illustrates how the traditional iterative EEA is performed. Whereas, Algorithm 1 presents it, showing how it clearly involves polynomial division and multiplication processes [7, 29].

Table 2.10: Polynomial EEA over Extension Fields

Extended Euclidean Algorithm for Polynomials [62]			
Calculate	**Which satisfies**	**Calculate**	**Which satisfies**
$R_{-1} = G$		$U_{-1} = 1; V_{-1} = 0$	$G = GU_{-1} + AV_{-1}$
$R_0 = A$		$U_0 = 0; V_0 = 1$	$A = GU_0 + AV_0$
$R_1 = G \mod A$ $Q_1 = \lfloor G/A \rfloor$	$G = Q_1 A + R_1$	$U_1 = U_{-1} - Q_1 U_0 = 1$ $V_1 = V_{-1} - Q_1 V_0 = -Q_1$	$R_1 = GU_1 + AV_1$
$R_2 = A \mod R_1$ $Q_2 = \lfloor A/R_1 \rfloor$	$A = Q_2 R_1 + R_2$	$U_2 = U_0 - Q_2 U_1$ $V_2 = V_0 - Q_2 V_1$	$R_2 = GU_2 + AV_2$
$R_3 = R_1 \mod R_2$ $Q_3 = \lfloor R_1/R_2 \rfloor$	$R_1 = Q_3 R_2 + R_3$	$U_3 = U_1 - Q_3 U_2$ $V_3 = V_1 - Q_3 V_2$	$R_3 = GU_3 + AV_3$
$\ddots$	$\ddots$	$\ddots$	$\ddots$
$R_n = R_{n-2} \mod R_{n-1}$ $Q_n = \lfloor R_{n-2}/R_{n-3} \rfloor$	$R_{n-2} = Q_n R_{n-1} + R_n$	$U_n = U_{n-2} - Q_n U_{n-1}$ $V_n = V_{n-2} - Q_n V_{n-1}$	$R_n = GU_n + AV_n$
$R_{n+1} = R_{n-1} \mod R_n = 0$ $Q_{n+1} = \lfloor R_{n-1}/R_{n-2} \rfloor$	$R_{n-1} = Q_{n+1} R_n + 0$	$U = U_n$ $V = V_n = A^{-1}$	$\gcd[G, A] = R_n$

Algorithm 1 Extended Euclidean Algorithm (EEA)

Input: Polynomial A of degree $(m-1)$ & irreducible polynomial G of degree (m).
Output: $A^{-1} \mod G$.

1: **Initialize**
2: $R^{(0)} \leftarrow G;\ R^{(1)} \leftarrow A;\ U^{(0)} \leftarrow 1;\ U^{(1)} \leftarrow 0;\ V^{(0)} \leftarrow 0;\ V^{(1)} \leftarrow 1;\ j \leftarrow 1$
3: **while** $R^{(j)} \neq 0$ (gcd $\neq 1$) **do**
4: **Divide:**
$Q^{(j)} \leftarrow \lfloor R^{(j-1)}/R^{(j)} \rfloor;$
$R^{(j+1)} \leftarrow R^{(j-1)} - Q^{(j)} \cdot R^{(j)};$
5: **Multiply:**
$U^{(j+1)} \leftarrow U^{(j-1)} - Q^{(j)} \cdot U^{(j)};$ $\{U^{(j+1)} = V^{(j)}\}$
$V^{(j+1)} \leftarrow V^{(j-1)} - Q^{(j)} \cdot V^{(j)};$
6: $j \leftarrow j + 1;$
7: **end while**
8: **return** $V^{(j)} = [A^{-1}]$

When designing a finite-field inverter, CPD, computation speed (throughput), computation delay (latency), and AT-complexity are compromised [78]. The number of cycles required for each internal process of Algorithm 1, i.e., polynomial *division* and *multiplication*, to be executed depends on the quotient *degree*. This unpredictable process results in complex control function and higher CPD, when EEA is directly implemented [29]. Therefore, a new reformulation of the algorithm is presented in this work and optimized to be suitable for systolic arrays implementation.

2.2.3 Modified EEA Algorithm

The drawbacks of implementing EEA in Algorithm 1 directly includes the dependency shadiness of polynomial multiplication on the results of division. Consequently, the latency will be doubled, and larger area will be required, accommodating each process individually. Moreover, polynomial multiplication may require extra area and time considering all possible combinations of multiplicands and multipliers. The first step in order to envision data flow in Algorithm 1, is to expand polynomial division and multiplication. This step will challenge the dependencies of these sequential processes within the algorithm, extending the space for parallelism and concurrency. In order to expand the algorithm, ***for*** loop is utilized considering our derivations in [2, 19], which will be studied elaborately in **Chaper** 3. Furthermore, $\boldsymbol{A}$ & $\boldsymbol{G}$ are co-primes and $\gcd[\boldsymbol{G}, \boldsymbol{A}] = 1$ since $\boldsymbol{G}$ is irreducible. So, the algorithm can be terminated one step earlier, i.e., one cycle before $\gcd$ reaches *zero*. It can also be observed that the exchange between variables in Algorithm 1 is invisible, which requires a sophisticated way to extract and exhibit them according to the algorithm logic. Expanded version of EEA, with a sufficient set of variables, is presented in Algorithm 2.

Algorithm 2 Expanded EEA over prime extension fields

Input: Polynomials $\boldsymbol{G}$ of degree m, and $\boldsymbol{A}$ of degree $n < m$
Output: $\boldsymbol{A}^{-1} \mod \boldsymbol{G}$

1: $\boldsymbol{U} \leftarrow 0; \boldsymbol{V} \leftarrow 1;$
2: **while** $R \neq 0$ ($\gcd \neq 1$) **do**
3: $\quad \boldsymbol{R} \leftarrow \boldsymbol{G}; \boldsymbol{P} \leftarrow 0; \boldsymbol{Q} \leftarrow 0;$
4: $\quad$ **for** $(i = G_{deg} - A_{deg}; i > 0; i--)$ **do** {*Division*}
5: $\quad\quad \boldsymbol{A} \leftarrow A << i;$ {*Left Shift A, i times*}
6: $\quad\quad \boldsymbol{Q(i)} \leftarrow R(i+n);$
7: $\quad\quad \boldsymbol{R} \leftarrow R \oplus (Q(i) \cdot A);$
8: $\quad$ **end for**
9: $\quad$ **for** $(i = Q_{deg}; i > 0; i--)$ **do** {*Multiplication*}
10: $\quad\quad \boldsymbol{V} \leftarrow V << i;$
11: $\quad\quad \boldsymbol{P} \leftarrow P \oplus (Q(i) \cdot V);$
12: $\quad$ **end for**
13: $\quad \boldsymbol{P} \leftarrow P + U;$
14: $\quad \boldsymbol{U} \leftarrow V;$
15: $\quad \boldsymbol{V} \leftarrow P;$
16: $\quad \boldsymbol{G} \leftarrow A;$
17: $\quad \boldsymbol{A} \leftarrow R;$
18: **end while**
19: **return** $\boldsymbol{V}$

For systolic EEA-based inversion, several compositions of polynomial division and multiplication have been experimented throughout the literature, including "*parallel division and multiplication*" in [8, 29], which was a variant of the work in [26]. In this research, concurrent polynomial division and multiplication over binary and ternary fields will be examined and design space will be explored in order to define the concurrency parameters and obtain competitive AT-complexity. In order to perform polynomial division and multiplication of EEA concurrently, the two consecutive *for* loops at lines 4-8 and 9-12 of Algorithm 2 can be combined into one *for* loop as shown in Algorithm 3.

Algorithm 3 Modified EEA over prime extension fields

Input: Polynomials $\boldsymbol{G}$ of degree m, and $\boldsymbol{A}$ of degree $n < m$
Output: $\boldsymbol{A}^{-1} \mod \boldsymbol{G}$
1: $\boldsymbol{U} \leftarrow 0; \boldsymbol{V} \leftarrow 1;$
2: **while** $R \neq 1$ (gcd $\neq 1$) **do**
3: $\quad \boldsymbol{R} \leftarrow \boldsymbol{G}; \boldsymbol{P} \leftarrow 0; \boldsymbol{Q} \leftarrow 0;$
4: $\quad$ **for** $(i = G_{deg} - A_{deg}; i > 0; i--)$ **do** {*Concurrent Division/Multiplication*}
5: $\quad\quad \boldsymbol{A} \leftarrow A << i;$
6: $\quad\quad \boldsymbol{Q(i)} \leftarrow R(i+n);$
7: $\quad\quad \boldsymbol{R} \leftarrow R + (Q(i) \times A);$
8: $\quad\quad \boldsymbol{V} \leftarrow V << i;$
9: $\quad\quad \boldsymbol{P} \leftarrow P + (Q(i) \times V);$
10: $\quad$ **end for**
11: $\quad \boldsymbol{P} \leftarrow P + U;$
12: $\quad \boldsymbol{U} \leftarrow V;$
13: $\quad \boldsymbol{V} \leftarrow P;$
14: $\quad \boldsymbol{G} \leftarrow A;$
15: $\quad \boldsymbol{A} \leftarrow R;$
16: **end while**
17: **return** $\boldsymbol{V}$

Although Algorithm 3 looks more optimized than Algorithm 2, several challenges are still confronted when hardware implementation is aimed. The main challenge, besides the unpredictability of terminating the inner and outer loops within the algorithm, is the varying alignments of variables. These uncertain digit-shifts involves degrees revaluation of each variable every outer iteration and inner cycle. These issues will be addressed in **Chaper** 3 and **Chaper** 4, resolving them by using systolic architectural methodology and maintaining reasonable AT-complexity and CPD.

Systolic Architectures

A "systolic array" is an array of simple computing cells that process and pass the data in a rhythmic manner. If a task can be decomposed into subtasks, then these subtasks can be performed in parallel if they are independed, or pipelined if they are dependent. Systolic architectures can provide massive concurrency by pipelining, parallelism, or both [80].

According to Kung [81], systolic architectures have simple and regular design, which results in cost-effectiveness, flexibility and scalability. In addition, they demonstrate reliable concurrency with regular and local communication and control, enabling efficient implementations. Moreover, they provide a computation rate that can be decomposed to minimize I/O requirements, balancing them with different performance parameters of a special-purpose system.

According to Gebali [18], the array dimension is based on the nature of the algorithm, and most of the well-known systolic processors require one-, two-, or three-dimensional arrays. In such architectures, data moves between the adjacent Processing Element (PE)s in one direction or more, all PEs usually perform the same process, and intermediate results should be stored in a small memory in each PE. Systolic arrays have been suggested to implement regular iterative algorithms with simple data dependencies, e.g. finite field arithmetic, digital filters, string search and pattern matching, solution of linear algebraic systems and motion estimation in video compression.

It has been reported that systolic field inversion architectures can be categorized into three types. First, two-dimensional arrays, which have area complexity of $O(m^2)$, are more suitable for high-throughput applications with small field size. Second, one-dimensional arrays, which have area complexity of $O(m)$, include bit-parallel, bit-serial and in-place bit-parallel architectures. Finally, digit-serial architectures, which compromise area complexity and throughput in their circuit implementation, have area complexity of $O(dm)$, where d is the digit size [30, 82].

In this work, one-dimensional systolic arrays are mainly considered, in particular, bit-parallel architectures. Meanwhile, one digit-serial architecture is proposed to be utilized for embedded applications.

Chapter 3

Systolic Architectures for Polynomial Division and Multiplication

The realization of EEA-based field inversion requires two sequential iterative processes, i.e., the GCD and multiplicative inverse computations. Therefore, performing these activities concurrently contributes significantly in achieving preferable throughput while maintaining reasonable critical path delay and AT-complexity. As stated earlier, these computations are simply repeated iterative polynomial divisions and multiplications. The nature of these processes brings many challenges including the varying alignments of variables before and during each iteration. Since the size of polynomial division and multiplication can only be determined during the computation, direct implementation of EEA is considered inefficient. The irregularity of these internal processes results in complex control function and higher CPD [25, 29]. Exploring the space of implementing such an algorithm on hardware requires reconsidering these internal sequential processes in order to allow for parallelism and/or concurrency.

In this chapter, polynomial division and multiplication are revisited, addressing their implementation challenges, and deriving suitable iterative equations for systolic architectures in order to reduce their expensive requirements. Starting with polynomial division conceptualization, its iterative equations over extension fields, GF(p^m), are derived. Then, the design space of polynomial division over extension fields is explored and examples of binary and ternary polynomial division are demonstrated. Similarly, polynomial multiplication is realized in systolic array architectures using the same systematic methodology.

3.1 Design Space Exploration of Polynomial Division

Polynomial division is found in applications such as error detection, Cyclic Redundancy Check (CRC) in particular, and data encoding for digital communication systems. In addition, it is employed in factoring polynomials and finding tangents to polynomial functions [83]. Furthermore, it implements EEA, which is used as the field inversion unit of ECC. Every iteration of EEA, a remainder and quotient are produced in order to compute the inverse, in which revaluation of variables' degrees is required each division cycle. Therefore, many reformulations for EEA have been proposed, avoiding polynomial division to suite VLSI implementation. However in this work, polynomial division is revisited to be implemented independently using the systolic arrays methodology in [18], before it is utilized to build VLSI-suitable EEA inverters in Chapter 4. In this section, polynomial division is analyzed and iterative equations that are suitable for systolic architectures are derived. Then, our design exploration methodology for such an algorithm is utilized. Finally, the obtained architectures are instanced for binary and ternary fields, comparing them with the existing related works. Reconsidering polynomials $\boldsymbol{G}$ and $\boldsymbol{A}$ of degrees m and $n<m$ as follows:

$$\boldsymbol{G} = \sum_{i=0}^{m} g_i x^i \text{ and } \boldsymbol{A} = \sum_{i=0}^{n} a_i x^i, \quad \text{where } g_i, a_i \in \{0, 1, \cdots p-1\} \text{ and } g_m, g_0, a_n \neq 0.$$

The division, $\boldsymbol{G} \div \boldsymbol{A}$, produces $\boldsymbol{Q}$ of degree ($w=m-n$), and $\boldsymbol{R}$ of degree ($< n$), such that:

$$\boldsymbol{Q} = \sum_{i=0}^{m-n} q_i x^i \text{ and } \boldsymbol{R} = \sum_{i=0}^{n-1} r_i x^i, \quad \text{where } q_i, r_i \in \{0, 1, \cdots p-1\}.$$

TABLE 3.1 illustrates the process of polynomial division over extension fields, where iteration number occupies the first column, and the results ($\boldsymbol{Q}$, $\boldsymbol{R}$) reside in the last row.

Table 3.1: Polynomial Division: Derivation Approach

It.#	$\boldsymbol{g_m}$	$\boldsymbol{g_{m-1}}$	$\boldsymbol{g_{m-2}}$	$\boldsymbol{g_{m-3}}$	$\cdots$	$\boldsymbol{g_{m-w}}$	g_{m-w-1}	$\cdots$	g_2	g_1	g_0
0	$a_n^{(0)}$	$\boldsymbol{a_{n-1}^{(0)}}$	$\boldsymbol{a_{n-2}^{(0)}}$	$\boldsymbol{a_{n-3}^{(0)}}$	$\cdots$	$\boldsymbol{a_{n-w}^{(0)}}$	$a_{n-w-1}^{(0)}$	$\cdots$	0	0	0
1		$a_n^{(1)}$	$\boldsymbol{a_{n-1}^{(1)}}$	$\boldsymbol{a_{n-2}^{(1)}}$	$\cdots$	$\boldsymbol{a_{n-w+1}^{(1)}}$	$a_{n-w}^{(1)}$	$\cdots$	0	0	0
2			$a_n^{(2)}$	$\boldsymbol{a_{n-1}^{(2)}}$	$\cdots$	$\boldsymbol{a_{n-w+2}^{(2)}}$	$a_{n-w+1}^{(2)}$	$\cdots$	0	0	0
3				$a_n^{(3)}$	$\cdots$	$\boldsymbol{a_{n-w+3}^{(3)}}$	$a_{n-w+2}^{(3)}$	$\cdots$	0	0	0
$\cdots$					$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$w-2$					$\cdots$	$\boldsymbol{a_{n-2}^{(w-2)}}$	$a_{n-3}^{(w-2)}$	$\cdots$	$a_0^{(w-2)}$	0	0
$w-1$					$\cdots$	$\boldsymbol{a_{n-1}^{(w-1)}}$	$a_{n-2}^{(w-1)}$	$\cdots$	$a_1^{(w-1)}$	$a_0^{(w-1)}$	0
w					$\cdots$	$a_n^{(w)}$	$a_{n-1}^{(w)}$	$\cdots$	$a_2^{(w)}$	$a_1^{(w)}$	$a_0^{(w)}$
$\boldsymbol{Q,R}$	$\boldsymbol{q_w}$	$\boldsymbol{q_{w-1}}$	$\boldsymbol{q_{w-2}}$	$\boldsymbol{q_{w-3}}$	$\cdots$	$\boldsymbol{q_0}$	$\boldsymbol{r_{n-1}}$	$\cdots$	$\boldsymbol{r_2}$	$\boldsymbol{r_1}$	$\boldsymbol{r_0}$

$a_n^{(0)} = a_n \times q_w$, $a_n^{(1)} = a_n \times q_{w-1}$, $a_n^{(2)} = a_n \times q_{w-2}$, $a_n^{(3)} = a_n \times q_{w-3}$, $a_n^{(w)} = a_n \times q_0$

Each cycle, elements of $\boldsymbol{A}$ are *multiplied* by $\boldsymbol{q}_{w-i}$ and *left-shifted* by $\boldsymbol{w-i}$, in order to align $\boldsymbol{a}_n^{(i)}$ and $\boldsymbol{g}_{m-i}$. Meanwhile, $\boldsymbol{q}_{w-i}$ is obtained by *subtracting* $\sum_{j=0}^{i-1}\{a_{n-(i-j)}^{(j)}\}$ from g_{m-i}. Elements of $Q(x)$ can be expressed as follows:

$$
\begin{aligned}
&\boldsymbol{q_w} = g_m/a_n,\\
&\boldsymbol{q_{w-1}} = (g_{m-1} - a_{n-1}^{(0)})/a_n,\\
&\boldsymbol{q_{w-2}} = (g_{m-2} - a_{n-2}^{(0)} - a_{n-1}^{(1)})/a_n,\\
&\boldsymbol{q_{w-3}} = (g_{m-3} - a_{n-3}^{(0)} - a_{n-2}^{(1)} - a_{n-1}^{(2)})/a_n,\\
&\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots\\
&\boldsymbol{q_0} = (g_{m-w} - a_{n-w}^{(0)} - a_{n-w+1}^{(1)} - a_{n-w+2}^{(2)} - \cdots - a_{n-1}^{(\text{w-1})})/a_n.\\
&\boldsymbol{q_{w-i}} = (g_{m-i} - a_{n-i}^{(0)} - a_{n-i+1}^{(1)} - a_{n-i+2}^{(2)} - \cdots - a_{n-w+i-1}^{(\text{i-1})})/a_n.
\end{aligned}
$$

Substituting $a_{n-i+k}^{(k)}$ with their equivalence, we found that:

$$
\begin{aligned}
&\boldsymbol{q_{w-1}} = (g_{m-1} - a_{n-1} \times q_w)/a_n,\\
&\boldsymbol{q_{w-2}} = (g_{m-2} - a_{n-2} \times q_w - a_{n-1} \times q_{w-1})/a_n,\\
&\boldsymbol{q_{w-3}} = (g_{m-3} - a_{n-3} \times q_w - a_{n-2} \times q_{w-1} - a_{n-1} \times q_{w-2})/a_n,\\
&\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots\\
&\boldsymbol{q_0} = (g_{m-w} - a_{n-w} \times q_w - a_{n-w+1} \times q_{w-1} - a_{n-w+2} \times q_{w-2} - \cdots - a_{n-1} \times q_1)/a_n.\\
&\boldsymbol{q_{w-i}} = (g_{m-i} - a_{n-i} \times q_w - a_{n-i+1} \times q_{w-1} - a_{n-i+2} \times q_{w-2} - \cdots - a_{n-w+i-1} \times q_{w-i+1})/a_n.
\end{aligned}
$$

Therefore, the ith element of $\boldsymbol{Q}$, $\boldsymbol{q_{w-i}}$, can be rewritten as follows:

$$
\boldsymbol{q_{w-i}} = (\boldsymbol{g_{m-i}} - \sum_{\boldsymbol{j=0}}^{\boldsymbol{i-1}} \{\boldsymbol{a_{n-(i-j)}} \times \boldsymbol{q_{w-j}}\})/\boldsymbol{a_n}, \quad \text{where } 0 \leq i \leq w \tag{3.1}
$$

All elements of $R(x)$ can be computed simultaneously with q_0 as follows:

$$
\begin{aligned}
&\boldsymbol{r_{n-1}} = g_{m-w-1} - a_{n-w-1} \times q_w - a_{n-w} \times q_{w-1} - a_{n-w+1} \times q_{w-2} - \cdots - a_{n-2} \times q_1 - a_{n-1} \times q_0,\\
&\boldsymbol{r_{n-2}} = g_{m-w-2} - a_{n-w-2} \times q_w - a_{n-w-1} \times q_{w-1} - a_{n-w} \times q_{w-2} - \cdots - a_{n-3} \times q_1 - a_{n-2} \times q_0,\\
&\boldsymbol{r_{n-3}} = g_{m-w-3} - a_{n-w-3} \times q_w - a_{n-w-2} \times q_{w-1} - a_{n-w-1} \times q_{w-2} - \cdots - a_{n-4} \times q_1 - a_{n-3} \times q_0,\\
&\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots\\
&\boldsymbol{r_0} = g_0 - a_0 \times q_w.\\
&\boldsymbol{r_{n-i}} = g_{m-w-i} - a_{n-w-i} \times q_w - a_{n-w-i+1} \times q_{w-1} - a_{n-w-i+2} \times q_{w-2} - \cdots - a_{n-i-1} \times q_1 - a_{n-i} \times q_0.
\end{aligned}
$$

From these expressions, it can be observed that:

$$
\boldsymbol{r_{n-i}} = \boldsymbol{g_{m-i}} - \sum_{\boldsymbol{j=0}}^{\boldsymbol{i-1}} \{\boldsymbol{a_{n-(i-j)}} \times \boldsymbol{q_{w-j}}\} \quad \text{where } w+1 \leq i \leq m \tag{3.2}
$$

3.1.1 Design Space Exploration of Polynomial Division over GF(p^m)

Considering the two iterative equations for $\boldsymbol{Q}$ and $\boldsymbol{R}$, and using Gebali's methodology [18], the DG, DAG and Projected DAG ($\overline{\text{DAG}}$) are developed to build our dividers.

3.1.1.1 Dependence Graph of Polynomial Division

Since Equations (3.1) and (3.2) include *two* indices, $\boldsymbol{i}$ and $\boldsymbol{j}$, they can be mapped into **2-D** graph. Instances of each variable can be interpreted as straight line equations in $\mathbb{D}$.

Inputs, $\boldsymbol{a}_{n-i+j}$ and $\boldsymbol{q}_{w-j}$ can be represented as follows:
$\boldsymbol{a}_{n-i+j}$: $\boldsymbol{i}-\boldsymbol{j} = n-k$, where $0 \leq k \leq n$,
$\boldsymbol{q}_{w-j}$: $\boldsymbol{j} = w-k$, where $0 \leq k \leq w$.

While inputs, $\boldsymbol{g}_{m-i}$, and outputs, $\boldsymbol{q}_{w-i}$ and $\boldsymbol{r}_{n-i}$, can be combined as follows:
$\boldsymbol{g}_{m-i}$: $\boldsymbol{i} = m-k$, where $0 \leq k \leq m$,
$\boldsymbol{q}_{w-i}$ and $\boldsymbol{r}_{n-i}$: $\boldsymbol{i} = m-k$, where $0 \leq k \leq m$.

Finally, the term $(/a_n)$ in Equation (3.1) can be represented as points (k, k), where $0 \leq k \leq w$. Figure 3.1 shows the DG of polynomial division, when $m=8$, and $n=5$.

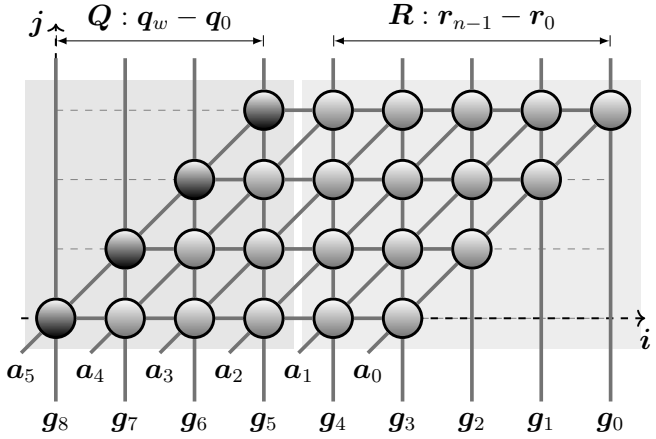


Figure 3.1: Polynomial Division **DG** $m = 8$, $n = 5$

Each node $(\boldsymbol{i} = \boldsymbol{j})$ in Fig. 3.1 accepts *two* inputs and produces *one* output, $\boldsymbol{q_i}$, performing *one* GF(p) operation. Whereas, each node $(\boldsymbol{i} \neq \boldsymbol{j})$ performs *two* GF(p) operations, accepting *three* inputs, and producing *one* output. However, each $(\boldsymbol{i}, \boldsymbol{j})$ node bypasses $\boldsymbol{a_k}$ to the node $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ each cycle. Nodes operations can be expressed as follows:

$$\boldsymbol{q}_{(j,j)} = (\boldsymbol{g}_m, \boldsymbol{r}_{(j-1,j-1)})/a_n, \quad \text{and} \quad \boldsymbol{r}_{(i,j)} = \boldsymbol{g}_k, \boldsymbol{r}_{(i-1,j-1)} - (\boldsymbol{a}_k \times \boldsymbol{q}_{j_k})$$

3.1.1.2 The Scheduling Function for Polynomial Division

Since the DG of polynomial division is obtained, different DAG's can be developed based on different scheduling vectors. In order to find a proper scheduling, the base vector of the dependence matrix of each variable, which defines its nullvector, should be determined. Based on the nullvectors and scheduling function, variables are *pipelined* or *broadcast*. From Equations (3.1) and (3.2), the base vectors of the dependence matrix and the nullvectors of all variables can be presented as follows:

$$\begin{aligned}
\boldsymbol{b}_a &= [1 \quad -1] \Rightarrow \boldsymbol{e}_a = [1 \;\; 1], \\
\boldsymbol{b}_{q_j} &= [0 \;\; 1] \Rightarrow \boldsymbol{e}_{q_j} = [1 \;\; 0], \\
\boldsymbol{b}_{g,q_i,r} &= [1 \;\; 0] \Rightarrow \boldsymbol{e}_{g,q_i,r} = [0 \;\; 1].
\end{aligned}$$

Since nodes at $(\boldsymbol{j})$ must wait until $(\boldsymbol{q_j})$ is available, the scheduling vectors for the design can be selected accordingly. The variable $\boldsymbol{q_j}$, which is shown as horizontal lines in Fig. 3.1, is generated from the first cell on the left, then *broadcast* to all cells on the right. Therefore, the time value associated with the cell $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ is greater than the time value associated with cell $(\boldsymbol{i}, \boldsymbol{j})$, i.e., $T_{p(\boldsymbol{i}+1,\boldsymbol{j}+1)} > T_{p(\boldsymbol{i},\boldsymbol{j})}$, which is represented in *affine scheduling* as follows:

$$\begin{bmatrix} s_1 & s_2 \end{bmatrix} \times \begin{bmatrix} \boldsymbol{i}+1 \\ \boldsymbol{j}+1 \end{bmatrix} - s > \begin{bmatrix} s_1 & s_2 \end{bmatrix} \times \begin{bmatrix} \boldsymbol{i} \\ \boldsymbol{j} \end{bmatrix} - s \tag{3.3}$$

Equation (3.3) indicates that the scheduling function can be rewritten as: $s_1 + s_2 > 0$, which means that the *three* possible scheduling vectors are:

$$\begin{aligned}
\boldsymbol{s}_1 &= [0 \;\; 1] \\
\boldsymbol{s}_2 &= [1 \;\; 0] \\
\boldsymbol{s}_3 &= [1 \;\; 1]
\end{aligned}$$

Based on the obtained nullvectors, inputs, $\boldsymbol{a}_k$, $\boldsymbol{g}_k$, and outputs, $\boldsymbol{q}_{i_k}, \boldsymbol{r}_k$, will be *pipelined*, while input $\boldsymbol{q}_{j_k}$ will be *broadcast* if the scheduling vector, $\boldsymbol{s}_1$, is selected. However, inputs, $\boldsymbol{q}_{j_k}$, $\boldsymbol{a}_k$, will be *pipelined*, and inputs, $\boldsymbol{g}_k$, and outputs, $\boldsymbol{q}_{i_k}, \boldsymbol{r}_k$ will be *broadcast* when $\boldsymbol{s}_2$ is chosen. On the other hand, all variables will be *pipelined* with $\boldsymbol{s}_3$ [18]. Design with scheduling vector, $\boldsymbol{s}_1$ is preferable since it complies with the time constraint on $\boldsymbol{q}_{j_k}$ and outperforms $\boldsymbol{s}_2$ and $\boldsymbol{s}_3$. For example when $m=8$, and $n=5$, polynomial division requires 4, 9 and 12 cycles to be executed with $\boldsymbol{s}_1$, $\boldsymbol{s}_2$ and $\boldsymbol{s}_3$, respectively. Selecting $\boldsymbol{s}_1$ as the scheduling vector is associating a point $p = (i, j)$ with the time value, $\boldsymbol{s} \times \boldsymbol{p} = [0 \;\; 1] \times [i \;\; j]^t = \boldsymbol{j}$. Therefore, nodes at $(\boldsymbol{j})$ produce partial results of all output samples every time step, requiring the availability of all input samples for the computations [18].

The DAG of polynomial division is illustrated in Fig. 3.2, where $m = 8$, $n = 5$, and $\boldsymbol{s}_1 = [0\ \ 1]$ is selected. With $\boldsymbol{s}_1$, number of cycles is proportional to the degree of $\boldsymbol{Q}$, which equals to $m-n+1$ precisely. This scheduling vector is advantageous over others because it basically complies with time constraint, which is highlighted by the horizontal gray stripes.

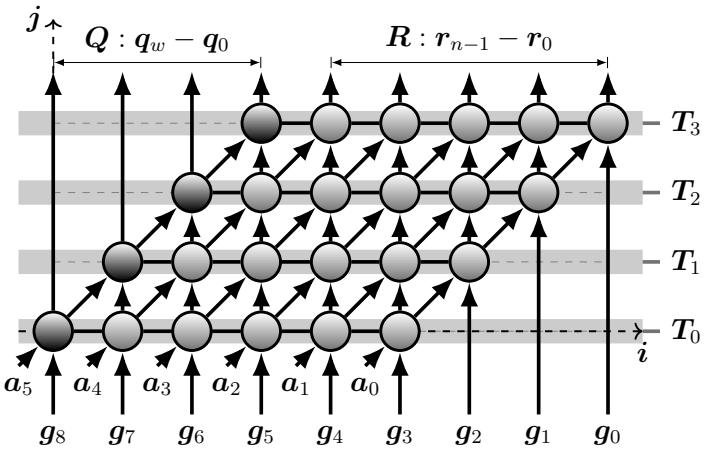


Figure 3.2: Polynomial Division **DAG**: $m = 8$, $n = 5$, $\boldsymbol{s}_1 = [0\ \ 1]$

However, the DAG of polynomial division, with $m = 8$, $n = 5$, and $\boldsymbol{s}_2 = [1\ \ 0]$, is depicted in Fig. 3.3. Number of cycles with $\boldsymbol{s}_2$ equals to the sum of $\boldsymbol{Q}_{deg}$ and nodes count, i.e., $m-n+n+1=m+1$. If nodes count is fixed at its maximum value, i.e., m, number of cycles with $\boldsymbol{s}_2$ equals to $m-n+m=2m-n$.

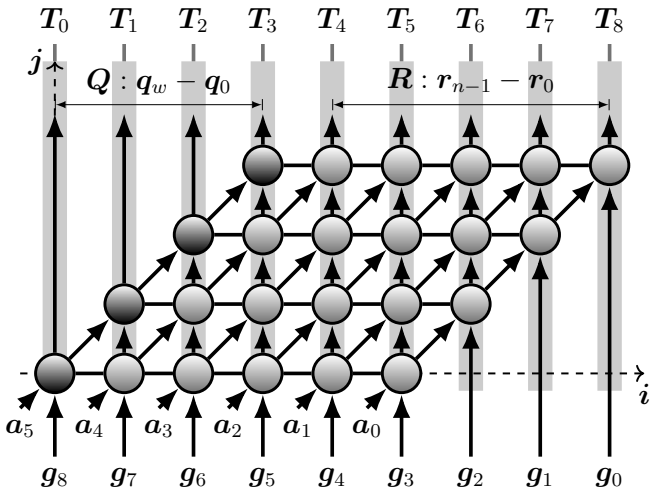


Figure 3.3: Polynomial Division **DAG**: $m = 8$, $n = 5$, $\boldsymbol{s}_2 = [1\ \ 0]$

Meanwhile, the DAG of polynomial division, with $m = 8$, $n = 5$, and $\boldsymbol{s}_3 = [1\ \ 1]$, is shown in Fig. 3.4. Number of cycles with $\boldsymbol{s}_3$ equals to $2(m-n)+n+1=2m-n+1$ exactly.

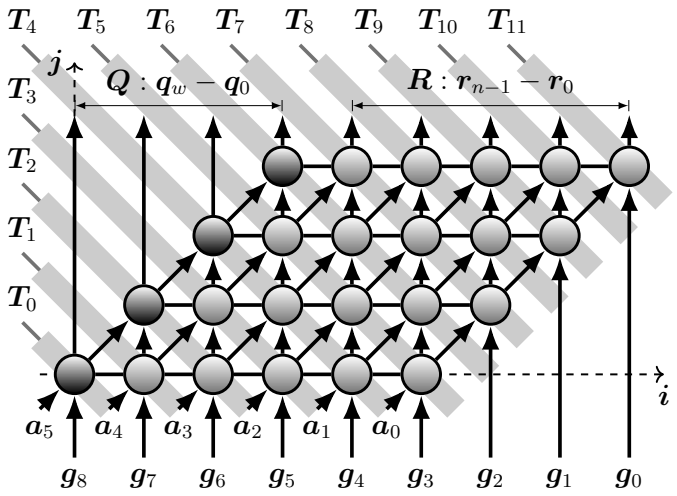


Figure 3.4: Polynomial Division **DAG**: $m = 8$, $n = 5$, $\boldsymbol{s}_3 = [1\ \ 1]$

3.1.1.3 Design Space Exploration when $\boldsymbol{s} = [0\ \ 1]$

Node projection transforms DAG in $\mathbb{Z}^c$ to $\overline{\text{DAG}}$ in $\mathbb{Z}^d$, where $d < c$, assigning a PE to each node and determining its activity. Projection direction ($\boldsymbol{d} = [d_1\ \ d_2]^t$) and its nullvector, projection matrix ($\boldsymbol{P} = [d_2\ \ -d_1]$), are vital to node projection. Since two nodes on an equitemporal plane in DAG should not be mapped into the same point in $\overline{\text{DAG}}$, projection direction must satisfy the inequality ($\boldsymbol{s} \times \boldsymbol{d} \neq 0$) [18]. Therefore, projection direction, $\boldsymbol{d}$, will have *three* choices with *three* corresponding matrices, when $\boldsymbol{s}=[0\ \ 1]$ as follows:

$\boldsymbol{d}_1 = [1\ \ 1]^t,\ \boldsymbol{P}_1 = [1\ \ -1]$, such that: $\Rightarrow \overline{\boldsymbol{p}} = \boldsymbol{i}-\boldsymbol{j}$,
$\boldsymbol{d}_2 = [-1\ \ 1]^t,\ \boldsymbol{P}_2 = [1\ \ 1]$, such that: $\Rightarrow \overline{\boldsymbol{p}} = \boldsymbol{i}+\boldsymbol{j}$,
$\boldsymbol{d}_3 = [0\ \ 1]^t,\ \boldsymbol{P}_3 = [1\ \ 0]$, such that: $\Rightarrow \overline{\boldsymbol{p}} = \boldsymbol{i}$.

Because the divisor coefficients $\boldsymbol{a}_{k(i,j)}$ are bypassed to nodes $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ in Fig. 3.2, they will be stored in the PE's, while $\boldsymbol{g}_k$'s and $\boldsymbol{q}_{j_k}$'s will be treated as inputs. For all $\boldsymbol{d}$ choices, there are $n+1$ PE's and the latency is $m-n+1$ cycles. There are two different PE structures with $\boldsymbol{d}_1$, where ($\boldsymbol{Q}$) is produced from PE_0, and ($\boldsymbol{R}$) is generated from $\text{PE}_{k>0}$. However, projection with $\boldsymbol{d}_2$ and $\boldsymbol{d}_3$ requires extra circuitry to *right-rotate* the stored variable ($\boldsymbol{a}_k$) and update the broadcast variable ($\boldsymbol{q}_{j_k}$) each cycle. In addition, their PE's occupy more area and require the initial inputs and the final outputs to be rearranged.

Design #1, with projection direction $d_1 = [1\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}_1$, when $m = 8$, $n = 5$, $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$ is presented in Fig. 3.5a. At time step i, each $\text{PE}_k^{(i)}$ accepts $\boldsymbol{g}_k$, or $\boldsymbol{r}_{k+1}^{(i-1)}$, producing $\boldsymbol{r}_k^{(i)}$, in which PE's activity in Fig. 3.5b is met. The details of PE's are illustrated in Figures 3.5c and3.5d.

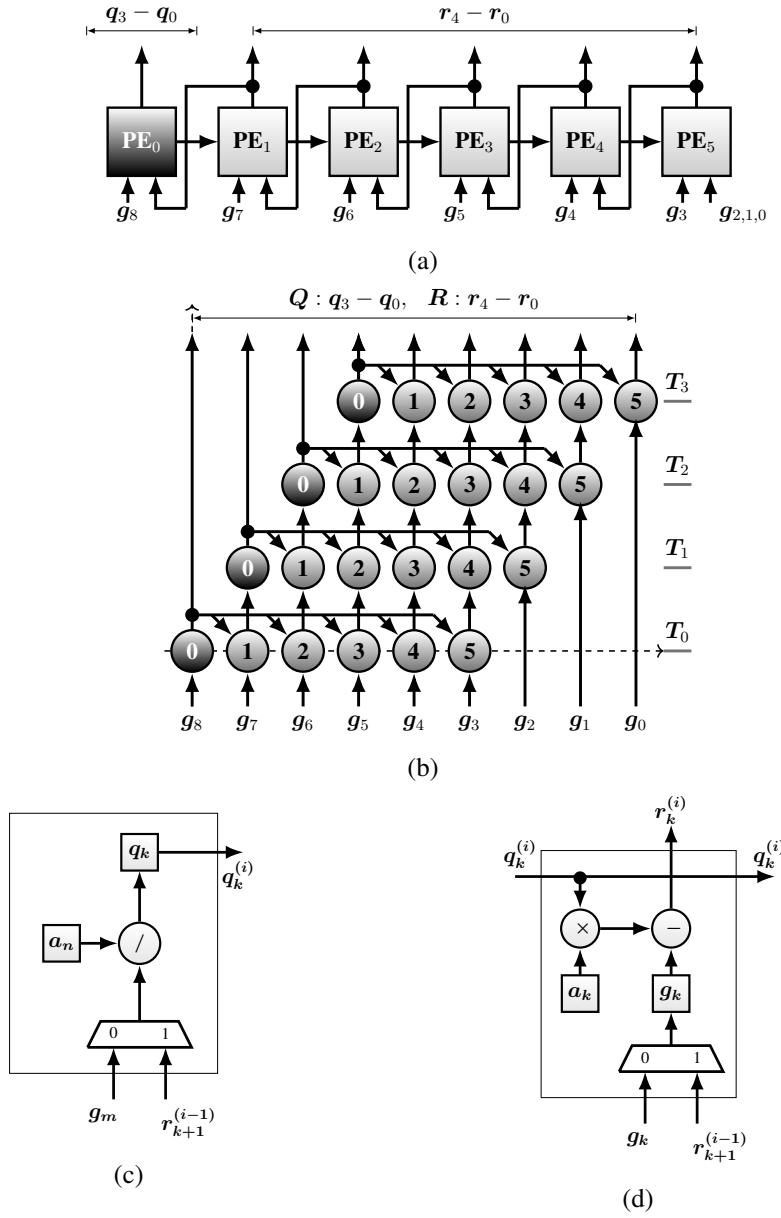


Figure 3.5: Polynomial Division, $\boldsymbol{d} = [1\ \ 1]^t$: (a) $\overline{\text{DAG}}$, (b) **PE** activity, (c) PE_0, (d) $\text{PE}_{k>0}$.

3.1.2 Design Space Exploration of Polynomial Division over GF(2^m)

Binary polynomial division appeared in [54, 84, 85] as part of their EEA proposals, and presented independently in [18, 86, 87] as systolic array architectures. In [2], a novel implementation of binary polynomial division in systolic arrays was presented. Assume binary polynomials $\boldsymbol{G}$ and $\boldsymbol{A}$ of degrees m and $n<m$ as follows:

$$\boldsymbol{G} = \sum_{i=0}^{m} g_i x^i \text{ and } \boldsymbol{A} = \sum_{i=0}^{n} a_i x^i, \quad \text{where } g_i, a_i \in \{0,1\} \text{ and } g_m, g_0, a_n = 1$$

The division, $\boldsymbol{G} \div \boldsymbol{A}$, produces $\boldsymbol{Q}$ and $\boldsymbol{R}$, such that:

$$\boldsymbol{Q} = \sum_{i=0}^{m-n} q_i x^i \text{ and } \boldsymbol{R} = \sum_{i=0}^{n-1} r_i x^i, \quad \text{where } q_i, r_i \in \{0,1\}.$$

Because $\boldsymbol{a_n} = 1$ always, the *two* polynomials, $\boldsymbol{Q}$ and $\boldsymbol{R}$ in Equations (3.1) and (3.2), can be concatenated in *one* polynomial, $\boldsymbol{F}$, such that: $\boldsymbol{F} = \boldsymbol{Q} \cdot x^n + \boldsymbol{R}$ as follows [2]:

$$\boldsymbol{f_i} = \boldsymbol{g_{m-i}} \oplus \sum_{\boldsymbol{j=0}}^{\boldsymbol{i-1}} \{\boldsymbol{a_{n-(i-j)}} \cdot \boldsymbol{q_{w-j}}\}, \text{ where } m \geq i \geq 0 \tag{3.4}$$

Note that logical XOR is the equivalent circuit for binary addition and subtraction over polynomials, while logical AND is the equivalent circuit for bits multiplication.

Considering Design #1 in Fig. 3.5, where $\boldsymbol{s} = [0 \;\; 1]$ and $\boldsymbol{d} = [1 \;\; 1]^t$, the PE's in Fig. 3.5d and Fig. 3.5d can be simplified for GF(2^m) division. The details of PE's in this case are depicted in Fig. 3.14.

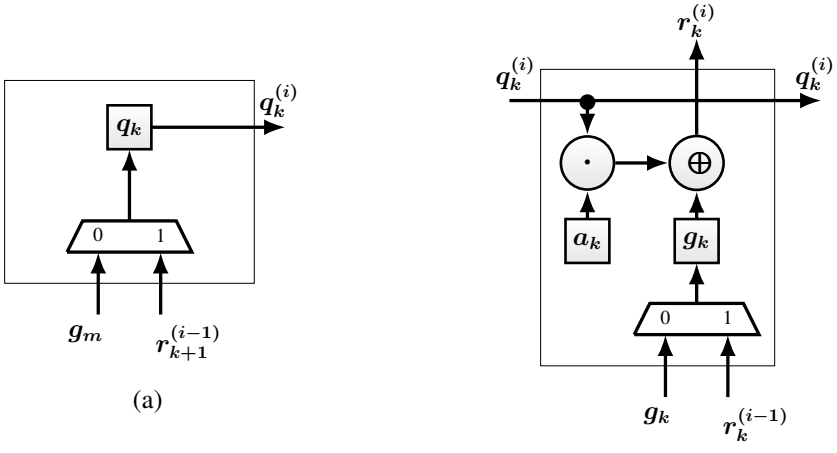


Figure 3.14: Binary Polynomial Division **PE**'s, $\boldsymbol{s} = [1 \;\; 0]^t$, $\boldsymbol{d} = [1 \;\; 1]$: (a) PE_0, (b) $PE_{k>0}$.

3.2 Exploring Polynomial Multiplication Design Space

Field polynomial multiplication has been discussed thoroughly in the literature, serving tremendous number of applications, with several algorithms presented and enormous amount of implementations proposed. However in this research, it is considered as an underlying process, which is attached to polynomial division in oder to compute the multiplicative inverse of a polynomial over finite fields based on EEA. It is important to note that if the multiplication operands are available simultaneously, the operation can be processed in any order or direction. This is not the case with our multiplication, in which its multiplier is generated sequentially from the polynomial division operation. Nevertheless, it can be performed concurrently with polynomial division, which satisfies the goal of this research.

Polynomial multiplication over finite fields is normally implemented as a modular arithmetic operation of polynomials, which requires regular polynomial multiplication and modular reduction [88]. However, it will be treated as a non-modular process in this work, because the degree of its product, $\boldsymbol{P}$, should not exceed $(m-1)$. In order to prove the non-modularity of our multiplier, the multiplication product, $\boldsymbol{P}$ can be observed in different case scenarios along with the multiplicand, $\boldsymbol{V}$ and the multiplier, $\boldsymbol{Q}$. Consider the variables involved in Algorithm 3, i.e., $\boldsymbol{G}$, $\boldsymbol{A}$, $\boldsymbol{R}$, $\boldsymbol{Q}$, $\boldsymbol{V}$, $\boldsymbol{U}$, and $\boldsymbol{P}$, with initial values as follows:

$\boldsymbol{G}_{deg}^{(0)} = \boldsymbol{m}$,
$\boldsymbol{A}_{deg}^{(0)} = \boldsymbol{n}$,
$\boldsymbol{R}_{deg}^{(0)} = \boldsymbol{m}$,
$\boldsymbol{Q}_{deg}^{(0)} = \boldsymbol{-1}$,
$\boldsymbol{V}_{deg}^{(0)} = \boldsymbol{0}$,
$\boldsymbol{U}_{deg}^{(0)} = \boldsymbol{-1}$,
$\boldsymbol{P}_{deg}^{(0)} = \boldsymbol{-1}$.

Then at any iteration k within the *while* loop, they can be as follows:
$\boldsymbol{G}_{deg}^{(k)} = \boldsymbol{A}_{deg}^{(k-1)}$,
$\boldsymbol{A}_{deg}^{(k)} = \boldsymbol{R}_{deg}^{(k-1)}$,
$\boldsymbol{R}_{deg}^{(k)} \leq \boldsymbol{A}_{deg}^{(k)} - 1$,
$\boldsymbol{Q}_{deg}^{(k)} = \boldsymbol{G}_{deg}^{(k)} - \boldsymbol{A}_{deg}^{(k)}$,
$\boldsymbol{V}_{deg}^{(k)} = \boldsymbol{P}_{deg}^{(k)}$,
$\boldsymbol{U}_{deg}^{(k)} = \boldsymbol{V}_{deg}^{(k-1)}$,
$\boldsymbol{P}_{deg}^{(k)} = \boldsymbol{Q}_{deg}^{(k)} + \boldsymbol{V}_{deg}^{(k-1)}$.

Table 3.7 presents behavioral examples of each variable in different extreme scenarios.

Table 3.7: The behavior of $\boldsymbol{P}_{deg}$ in different case scenarios

Case (a): $n=1$, where the termination is at iteration #1.

It#	$\boldsymbol{G}_{deg}$	$\boldsymbol{A}_{deg}$	$\boldsymbol{R}_{deg}$	$\boldsymbol{Q}_{deg}$	$\boldsymbol{V}_{deg}$	$\boldsymbol{P}_{deg}$	$\boldsymbol{U}_{deg}$
0	m	1	0	$m-1$	0	-1	-1
1	1	0	-1	1	1	1	0

Case (b): $n=m-1$, where $\boldsymbol{A}^{(i)}_{deg}=\boldsymbol{G}^{(i)}_{deg}-1$ and the termination at iteration #$k+1$.

It#	$\boldsymbol{G}_{deg}$	$\boldsymbol{A}_{deg}$	$\boldsymbol{R}_{deg}$	$\boldsymbol{Q}_{deg}$	$\boldsymbol{V}_{deg}$	$\boldsymbol{P}_{deg}$	$\boldsymbol{U}_{deg}$
0	m	$m-1$	$m-2$	1	0	-1	-1
1	$m-1$	$m-2$	$m-3$	1	1	1	0
2	$m-2$	$m-3$	$m-4$	1	2	2	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$k-1$	$m-k+1$	$m-k$	1	1	$k-1$	$k-1$	$k-2$
k	$m-k$	1	0	$m-k-1$	$m-2$	$m-2$	$k-1$
$k+1$	1	0	-1	1	$m-1$	$k+1$	k

Case (c): $n=m-1$, where $\boldsymbol{A}^{(i)}_{deg}=\boldsymbol{G}^{(i)}_{deg}-1$ and the termination at iteration #$m-1$.

It#	$\boldsymbol{G}_{deg}$	$\boldsymbol{A}_{deg}$	$\boldsymbol{R}_{deg}$	$\boldsymbol{Q}_{deg}$	$\boldsymbol{V}_{deg}$	$\boldsymbol{P}_{deg}$	$\boldsymbol{U}_{deg}$
0	m	$m-1$	$m-2$	1	0	-1	-1
1	$m-1$	$m-2$	$m-3$	1	1	1	0
2	$m-2$	$m-3$	$m-4$	1	2	2	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
k	$m-k$	$m-k-1$	$m-k-2$	1	k	k	$k-1$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$m-3$	3	2	1	1	$m-3$	$m-3$	$m-4$
$m-2$	2	1	0	1	$m-2$	$m-2$	$m-3$
$m-1$	1	0	-1	1	$m-1$	$m-1$	$m-2$

Case (d): $0\leq n\leq m-1$, where the termination is at iteration #l.

It#	$\boldsymbol{G}_{deg}$	$\boldsymbol{A}_{deg}$	$\boldsymbol{R}_{deg}$	$\boldsymbol{Q}_{deg}$	$\boldsymbol{V}_{deg}$	$\boldsymbol{P}_{deg}$	$\boldsymbol{U}_{deg}$
0	m	n	$n-a$	$m-n$	0	-1	-1
1	n	$n-a$	$n-a-b$	a	a	a	0
2	$n-a$	$n-a-b$	$n-a-b-c$	b	$a+b$	$a+b$	a
3	$n-a-b$	$n-a-b-c$	$n-\cdots-d$	c	$a+b+c$	$a+b+c$	$a+b$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
l–2	$n-\cdots-x$	$n-\cdots-x-y$	1	y	$a+\cdots+y$	$a+\cdots+y$	$a+\cdots+x$
l–1	$n-\cdots-y$	1	0	$n-\cdots-y$–1	$n-1$	$n-1$	$a+\cdots+y$
l	1	0	-1	1	n	n	$n-1$

The algorithm terminates if $\boldsymbol{R}_{deg} = -1$, which occurs when $\boldsymbol{V}_{deg}$ reaches its maximum. Obviously from TABLE 3.7, $\boldsymbol{P}_{deg}^{max} = m-1$ for Cases (a), (b) and (c). In order to examine the correctness of this figure for all cases, the general Case (d) is considered as follows:
At iteration #$l-2$:

$$\boldsymbol{V}_{deg}^{(l-2)} = a + b + c + d + \cdots + y.$$

At iteration #$l-1$:

$$\boldsymbol{Q}_{deg}^{(l-1)} = \boldsymbol{G}_{deg}^{(l-1)} - \boldsymbol{A}_{deg}^{(l-1)} = n - a - b - c - d - \cdots - y - 1.$$

Since $\boldsymbol{P}_{deg}^{(k)} = \boldsymbol{Q}_{deg}^{(k)} + \boldsymbol{V}_{deg}^{(k-1)}$, it can be observed that:

$$\begin{aligned}\boldsymbol{P}_{deg}^{(l-1)} &= \boldsymbol{Q}_{deg}^{(l-1)} + \boldsymbol{V}_{deg}^{(l-2)} \\ &= n - \cancel{a} - \cancel{b} - \cancel{c} - \cancel{d} - \cdots - \cancel{y} - 1 \\ &\quad + \cancel{a} + \cancel{b} + \cancel{c} + \cancel{d} + \cdots + \cancel{y}.\end{aligned}$$

Because $\boldsymbol{V}_{deg}^{(l-1)} = \boldsymbol{P}_{deg}^{(l-1)}$, it can be ascertained that: $\boldsymbol{V}_{deg}^{(l-1)} = n - 1$.
As the algorithm terminates at iteration #l, where $\boldsymbol{Q}_{deg}^{(l)} = 1$, it is obvious that:

$$\boldsymbol{P}_{deg}^{max} = \boldsymbol{P}_{deg}^{(l)} = \boldsymbol{Q}_{deg}^{(l} + \boldsymbol{V}_{deg}^{(l-1)} = n.$$

Since $n^{max} = m - 1$, it can be derived that:

$$\boldsymbol{P}_{deg}^{max} = m - 1$$

Furthermore, it is worth noticing that the summation of all $\boldsymbol{Q}_{deg}$ has a fixed figure as follows:

$$\begin{aligned}\sum_{k=0}^{l} \boldsymbol{Q}_{deg} &= m - \cancel{n} + \cancel{a} + \cancel{b} + \cancel{c} + \cancel{d} + \cdots + \cancel{y} \\ &\quad + \cancel{n} - \cancel{a} - \cancel{b} - \cancel{c} - \cancel{d} - \cdots - \cancel{y} + \cancel{1} - \cancel{1} \\ &= m.\end{aligned}$$

This figure will be highly useful when analyzing the complexity of the systolic architectures in Chapter 4. However, it is crucial to remember that the algorithm can be terminated one step earlier as stated in 2.2.3. Therefore, the latency of our EEA inverter is proportional to $m - 1$. Finally, despite the non-modularity of our multiplier, it can be utilized for regular polynomial multiplication considering two polynomials of degrees $\leq m/2$.

Having the quotient polynomial $\boldsymbol{Q}$ of degree $w = m - n$ and assuming a polynomial $\boldsymbol{V}$ of degree y as follows:

$$\boldsymbol{V} = \sum_{i=0}^{y} v_i x^i \text{ and } \boldsymbol{Q} = \sum_{i=0}^{w} q_i x^i, \text{ where } v_i, q_i \in \{0, 1, \cdots p-1\}.$$

$\boldsymbol{V}$ multiply $\boldsymbol{Q}$ generates $\boldsymbol{P}$ of degree, $z = y + w$, so that $0 \leq z \leq m-1$, as follows:

$$\boldsymbol{P} = \sum_{i=0}^{z} p_i x^i, \text{ where } p_i \in \{0, 1, \cdots p-1\}$$

Table 3.8: GF(2^m) Polynomial Multiplication Process

	$\boldsymbol{v_y}$	$\boldsymbol{v_{y-1}}$	$\boldsymbol{v_{y-2}}$	$\cdots$	$\boldsymbol{v_2}$	$\boldsymbol{v_1}$	$\boldsymbol{v_0}$	
$\boldsymbol{q_0}$	$v_y \times q_0$	$v_{y-1} \times q_0$	$v_{y-2} \times q_0$	$\cdots$	$v_2 \times q_0$	$v_1 \times q_0$	$v_0 \times q_0$	$\boldsymbol{p_0}$
$\boldsymbol{q_1}$	$v_y \times q_1$	$v_{y-1} \times q_1$	$v_{y-2} \times q_1$	$\cdots$	$v_2 \times q_1$	$v_1 \times q_1$	$v_0 \times q_1$	$\boldsymbol{p_1}$
$\boldsymbol{q_2}$	$v_y \times q_2$	$v_{y-1} \times q_2$	$v_{y-2} \times q_2$	$\cdots$	$v_2 \times q_2$	$v_1 \times q_2$	$v_0 \times q_2$	$\boldsymbol{p_2}$
$\boldsymbol{\cdots}$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\boldsymbol{q_{w-2}}$	$v_y \times q_{w-2}$	$\cdots$	$\cdots$	$\cdots$	$v_2 \times q_{w-2}$	$v_1 \times q_{w-2}$	$v_0 \times q_{w-2}$	$\cdots$
$\boldsymbol{q_{w-1}}$	$v_y \times q_{w-1}$	$v_{y-1} \times q_{w-1}$	$\cdots$	$\cdots$	$v_2 \times q_{w-1}$	$v_1 \times q_{w-1}$	$v_0 \times q_{w-1}$	$\cdots$
$\boldsymbol{q_w}$	$v_y \times q_w$	$v_{y-1} \times q_w$	$v_{y-2} \times q_w$	$\cdots$	$v_2 \times q_w$	$v_1 \times q_w$	$v_0 \times q_w$	$\cdots$
	$\boldsymbol{p_z}$	$\boldsymbol{p_{z-1}}$	$\boldsymbol{p_{z-2}}$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

TABLE 3.8 illustrates how the elements of $\boldsymbol{P}$ are computed, presenting them in the last row and last column. Following our coloring system in the table, it can be observed that:

$p_z = v_y \times q_w$,
$p_{z-1} = v_{y-1} \times q_w + v_y \times q_{w-1}$,
$p_{z-2} = v_{y-2} \times q_w + v_{y-1} \times q_{w-1} + v_y \times q_{w-2}$,
$\cdots \quad \cdots \quad \cdots$,
$p_2 = v_0 \times q_2 + v_1 \times q_1 + v_2 \times q_0$,
$p_1 = v_0 \times q_1 + v_1 \times q_0$,
$p_0 = v_0 \times q_0$.

Therefore, the ith element, i.e., p_{z-i}, can be obtained as follows:

$$\boldsymbol{p_{z-i}} = v_{y-i} \times q_w + v_{y-(i-1)} \times q_{w-1} + v_{y-(i-2)} \times q_{w-2} + \cdots + v_{y-(i-j)} \times q_{w-j} + v_{y-j} \times q_{w-(i-j)} + \cdots + v_{y-2} \times q_{w-(i-2)} + v_{y-1} \times q_{w-(i-1)} + v_y \times q_{w-i}$$

Which can be rewritten as follows:

$$\boldsymbol{p_{z-i}} = \sum_{\boldsymbol{j=0}}^{\boldsymbol{i}} \{\boldsymbol{v_{y-(i-j)}} \times \boldsymbol{q_{w-j}}\}, \quad \text{where } 0 \leq i \leq z \tag{3.5}$$

3.2.1 Design Space Exploration of Polynomial Multiplication over GF(p^m)

Similar to polynomial division, the DG, DAG and Projected DAG ($\overline{\text{DAG}}$) are developed to build our multipliers, considering Equation (3.5) and using Gebali's methodology [18].

.

3.2.1.1 Dependence Graph of Polynomial Multiplication

In Equation (3.5), *two* iterations are described over *two* indices, $\boldsymbol{i}$ and $\boldsymbol{j}$. Which means that polynomial multiplication can be interpreted into **2-D** graph. Instances of each variable can be described by straight line equations in $\mathbb{D}$ as follows:

Inputs, $\boldsymbol{v}_{y-i+j}$ and $\boldsymbol{q}_{w-j}$ can be represented as follows:
$\boldsymbol{v}_{y-i+j}$: $\boldsymbol{i}-\boldsymbol{j} = y-k$, where $0 \leq k \leq y$,
$\boldsymbol{q}_{w-j}$: $\boldsymbol{j} = w-k$, where $0 \leq k \leq w$.

While outputs, $\boldsymbol{p}_{z-i}$ can be presented as follows:
$\boldsymbol{p}_{z-i} : \boldsymbol{i} = z-k$, where $0 \leq k \leq z$,

Figure 3.17 shows the DG of polynomial multiplication, when $m=8$, $n=5$, $y=4$, and $w=3$.

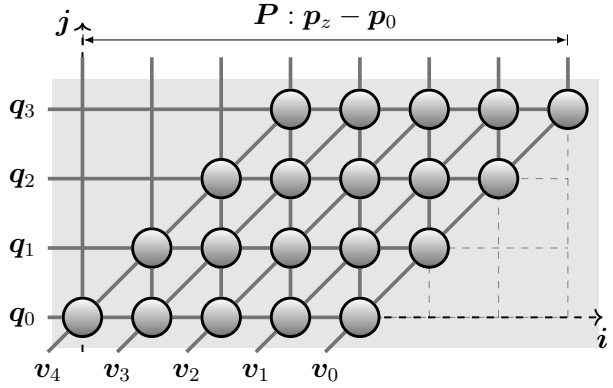


Figure 3.17: Polynomial Multiplication Dependency Graph (**DG**) $m=8$, $n=5$, $y=4$, $w=3$

Each node $(\boldsymbol{i}, \boldsymbol{j})$ in Fig. 3.17 accepts *two* inputs, and produces *one* output, $\boldsymbol{p_i}$, performing *two* GF(p) operation and bypassing $\boldsymbol{v_k}$ to the node $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ each cycle, such that:

$$\boldsymbol{p}_k(i) = \boldsymbol{p}_k(i-1) \oplus (\boldsymbol{v}_k \cdot \boldsymbol{q}_k),$$

3.2.1.2 The Scheduling Function for Polynomial Multiplication

Equivalent to polynomial division, Equation (3.5) indicates that the base vectors of the dependence matrix and the nullvectors of all variables can be presented as follows:

$$\begin{aligned} \boldsymbol{b}_v &= [1\ \ -1] \Rightarrow \boldsymbol{e}_v = [1\ \ 1], \\ \boldsymbol{b}_q &= [0\ \ 1] \Rightarrow \boldsymbol{e}_q = [1\ \ 0], \\ \boldsymbol{b}_p &= [1\ \ 0] \Rightarrow \boldsymbol{e}_p = [0\ \ 1]. \end{aligned}$$

The input $\boldsymbol{q}_k$, presented as horizontal lines in Fig. 3.17, is produced sequentially from the polynomial division circuit. Therefore, the time value associated with cells $(\boldsymbol{i}, \boldsymbol{j})$ is always smaller than the time value associated with cell $(\boldsymbol{i}{+}1, \boldsymbol{j}{+}1)$, i.e., $T_{p(\boldsymbol{i}+1,\boldsymbol{j}+1)} > T_{p(\boldsymbol{i},\boldsymbol{j})}$, which can be written similarly to Equation (3.3), i.e., $s_1 + s_2 > 0$ using *affine scheduling*. Thus, the *three* possible scheduling vectors are:

$$\begin{aligned} \boldsymbol{s}_1 &= [0\ \ 1], \\ \boldsymbol{s}_2 &= [1\ \ 0], \\ \boldsymbol{s}_3 &= [1\ \ 1]. \end{aligned}$$

Knowing that $\boldsymbol{e}_v = [1\ \ 1]$, $\boldsymbol{e}_q = [1\ \ 0]$, and $\boldsymbol{e}_p = [0\ \ 1]$, the scheduling vector, $\boldsymbol{s}_1$, will result in *pipelined* input ($\boldsymbol{v}_k$) and output ($\boldsymbol{p}_k$), and *broadcast* input variable ($\boldsymbol{q}_k$). The DAG of polynomial multiplication is illustrated in Fig. 3.18, where $y{=}4$, $w{=}3$, and $\boldsymbol{s}_1{=}[0\ \ 1]$ is selected. With $\boldsymbol{s}_1$, number of cycles is proportional to $\boldsymbol{Q}_{deg}$, which equals to $w{+}1$ precisely. This scheduling vector is advantageous over others because it basically complies with time constraint, which is highlighted by the horizontal gray stripes.

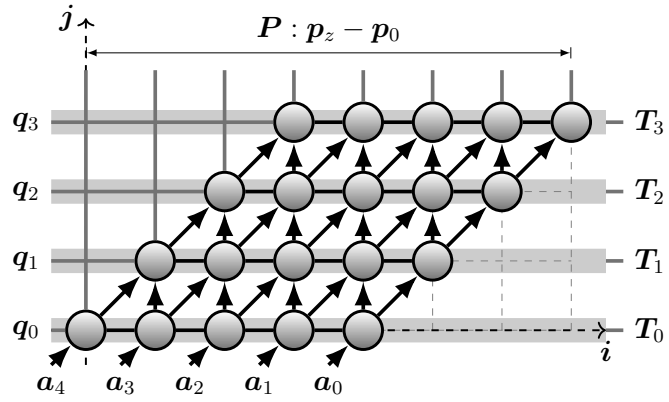


Figure 3.18: Polynomial Multiplication **DAG**: $m{=}8$, $n{=}5$, $y{=}4$, $w{=}3$, $\boldsymbol{s}_1 = [0\ \ 1]$

However, inputs, $\boldsymbol{q}_k$, $\boldsymbol{v}_k$, will be *pipelined*, and outputs, $\boldsymbol{p}_k$ will be *broadcast* when $\boldsymbol{s}_2$ is selected. Figure 3.19 depicts the DAG of polynomial multiplication, with $m=8$, $n=5$, $y=4$, $w=3$, and $\boldsymbol{s}_2=[1\ \ 0]$. Number of cycles with $\boldsymbol{s}_2$ equals to the sum of $\boldsymbol{Q}_{deg}$ and nodes count, i.e., $w+y+1$. The maximum number of PE's that work simultaneously is $m/2$.

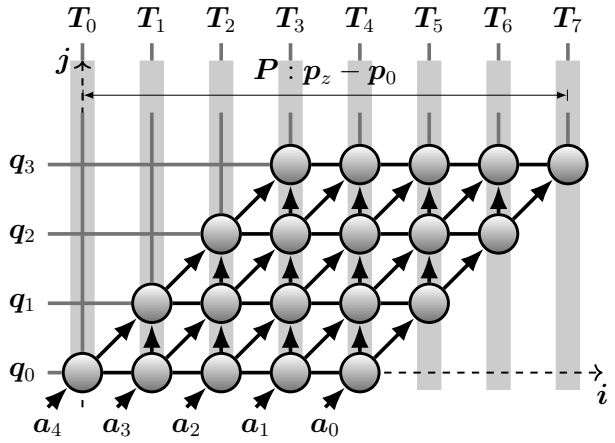


Figure 3.19: Polynomial Multiplication **DAG**: $m=8$, $n=5$, $y=4$, $w=3$, $\boldsymbol{s}_2 = [1\ \ 0]$

On the other hand, all variables will be *pipelined* when $\boldsymbol{s}_3$ is chosen. Figure 3.20 presents the DAG of polynomial multiplication, with $m=8$, $n=5$, $y=4$, $w=3$, and $\boldsymbol{s}_3=[1\ \ 1]$. Number of cycles with $\boldsymbol{s}_3$ equals to double $\boldsymbol{Q}_{deg}$ plus nodes count, i.e., $=2w+y+1$ exactly. The maximum number of PE's that work simultaneously is $m/4+1$.

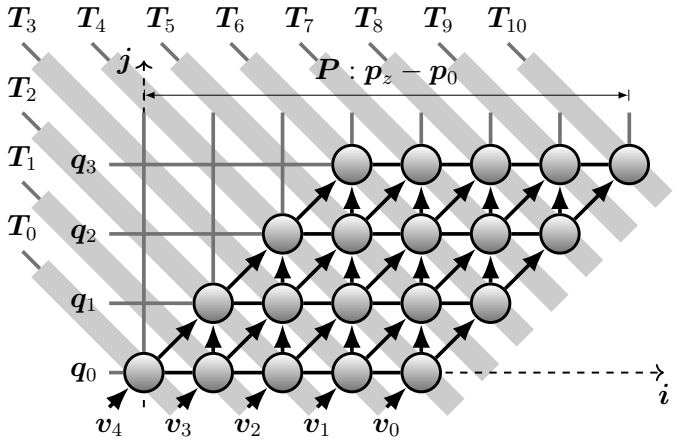


Figure 3.20: Polynomial Multiplication **DAG**: $m=8$, $n=5$, $y=4$, $w=3$, $\boldsymbol{s}_3=[1\ \ 1]$

3.2.1.3 Design Space Exploration of Polynomial Multiplication when $s = [0\ \ 1]$

Equivalent to polynomial division, projection direction, $\boldsymbol{d}$, will have *three* choices with *three* corresponding matrices, when $\boldsymbol{s}=[0\ \ 1]$ as follows:

$$\boldsymbol{d}_1 = [1\ \ 1]^t,\ \boldsymbol{P}_1 = [1\ \ -1], \quad \boldsymbol{d}_2 = [-1\ \ 1]^t,\ \boldsymbol{P}_2 = [1\ \ 1], \quad \boldsymbol{d}_3 = [0\ \ 1]^t,\ \boldsymbol{P}_3 = [1\ \ 0].$$

Since the multiplicand coefficients $\boldsymbol{v}_{k(i,j)}$ are bypassed to nodes $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ in Fig. 3.18, they will be stored in the PE's, while the multiplier $\boldsymbol{q}_j$'s will be treated as inputs.

Design #1, with projection direction $d_1 = [1\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}_1$, when $m=8$, $n=5$, $y=4$, $w=3$, $\boldsymbol{s}=[0\ \ 1]$ and $\boldsymbol{d}=[1\ \ 1]^t$ is presented with the activity and details of PE's in Fig. 3.21.

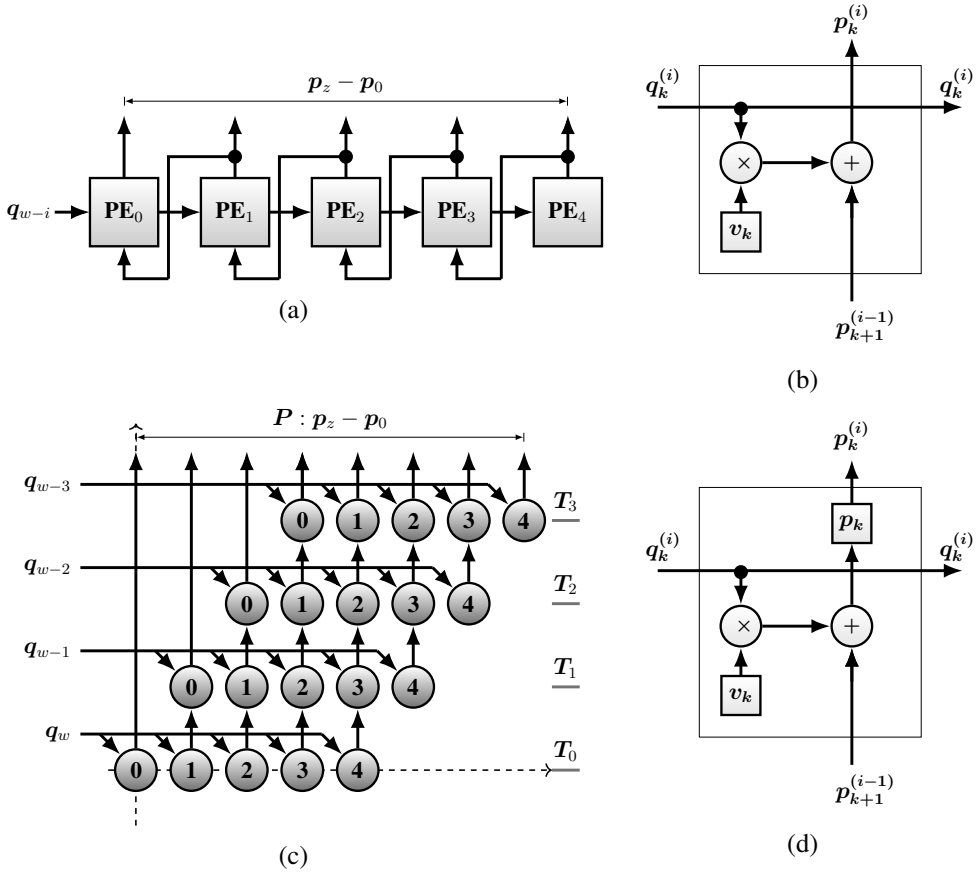


Figure 3.21: Poly. Multiplication, $\boldsymbol{d}=[1\ \ 1]^t$: (a) $\overline{\text{DAG}}$, (b) PE$_0$, (c) **PE** activity, (d) PE$_{k>0}$.

3.2.2 Design Space Exploration of Polynomial Multiplication over GF(2^m)

In [20], binary polynomial multiplication was revisited briefly in order to develop systolic array multipliers as part of EEA inversion circuit. Similar to division, systolic based binary polynomial multipliers are discussed in this subsection. Assume polynomials $\boldsymbol{V}$ and $\boldsymbol{Q}$ of degrees y and w as follows:

$$\boldsymbol{V} = \sum_{i=0}^{y} v_i x^i \text{ and } \boldsymbol{Q} = \sum_{i=0}^{w} q_i x^i, \text{ where } v_i, q_i \in \{0,1\}.$$

$\boldsymbol{V}$ multiply $\boldsymbol{Q}$ generates $\boldsymbol{P}$ of degree, $z = y+w$, so that $0 \leq z \leq m-1$, as follows:

$$\boldsymbol{p_{z-i}} = \sum_{\boldsymbol{j=0}}^{\boldsymbol{i}} \{\boldsymbol{v_{y-(i-j)}} \cdot \boldsymbol{q_{w-j}}\}, \qquad \text{where } p_i \in \{0,1\} \tag{3.6}$$

Note that logical XOR is the equivalent circuit for binary addition over polynomials.
Example:

$$\boldsymbol{Q} = q_3x^3 + q_2x^2 + q_1x + q_0, \text{ and } \boldsymbol{V} = v_4x^4 + v_3x^3 + v_2x^2 + v_1x + v_0,$$

TABLE 3.10 illustrates how ($\boldsymbol{P} = \sum_{i=0}^{7} p_i x^i = \boldsymbol{V} \times \boldsymbol{Q}$) is computed.

Table 3.10: Polynomial Multiplication ($\boldsymbol{P} = \boldsymbol{V} \times \boldsymbol{Q}$) Example , $y = 4, w = 3$

	$\boldsymbol{v_4}$	$\boldsymbol{v_3}$	$\boldsymbol{v_2}$	$\boldsymbol{v_1}$	$\boldsymbol{v_0}$	
$\boldsymbol{q_0}$	$v_4 \cdot q_0$	$v_3 \cdot q_0$	$v_2 \cdot q_0$	$v_1 \cdot q_0$	$v_0 \cdot q_0$	
$\boldsymbol{q_1}$	$v_4 \cdot q_1$	$v_3 \cdot q_1$	$v_2 \cdot q_1$	$v_1 \cdot q_1$	$v_0 \cdot q_1$	$\boldsymbol{p_0}$
$\boldsymbol{q_2}$	$v_4 \cdot q_2$	$v_3 \cdot q_2$	$v_2 \cdot q_2$	$v_1 \cdot q_2$	$v_0 \cdot q_2$	$\boldsymbol{p_1}$
$\boldsymbol{q_3}$	$v_4 \cdot q_3$	$v_3 \cdot q_3$	$v_2 \cdot q_3$	$v_1 \cdot q_3$	$v_0 \cdot q_3$	$\boldsymbol{p_2}$
		$\boldsymbol{p_7}$	$\boldsymbol{p_6}$	$\boldsymbol{p_5}$	$\boldsymbol{p_4}$	$\boldsymbol{p_3}$

Following the diagonal gray coloring system in the table, it can be observed that:

$p_7 = v_4 \cdot q_3$
$p_6 = v_4 \cdot q_2 \oplus v_3 \cdot q_3$
$p_5 = v_4 \cdot q_1 \oplus v_3 \cdot q_2 \oplus v_2 \cdot q_3$
$p_4 = v_4 \cdot q_0 \oplus v_3 \cdot q_1 \oplus v_2 \cdot q_2 \oplus v_1 \cdot q_3$
$p_3 = v_3 \cdot q_0 \oplus v_2 \cdot q_1 \oplus v_1 \cdot q_2 \oplus v_0 \cdot q_3$
$p_2 = v_2 \cdot q_0 \oplus v_1 \cdot q_1 \oplus v_0 \cdot q_2$
$p_1 = v_1 \cdot q_0 \oplus v_0 \cdot q_1$
$p_0 = v_0 \cdot q_0$

Considering Design #1 for polynomial multiplication in Fig. 3.21, where $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$, the PE can be simplified for GF(2^m) multiplication as depicted in Fig. 3.30.

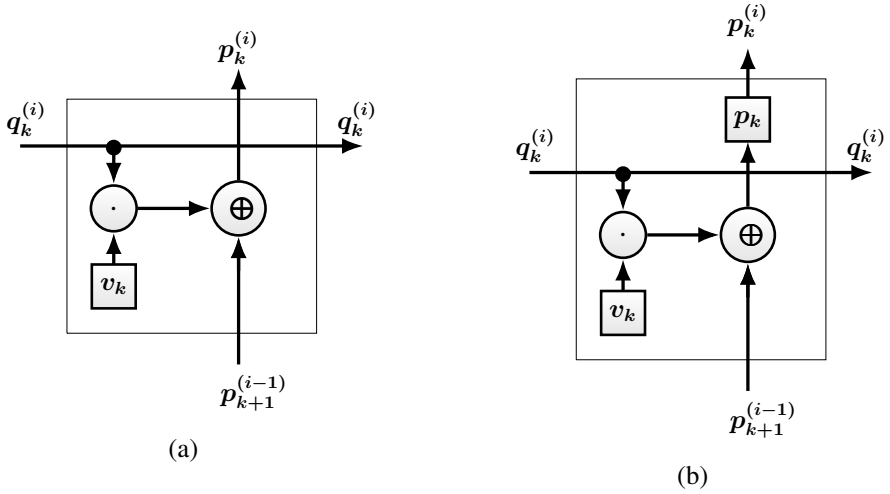


Figure 3.30: Binary Poly. Multiplication **PE**'s, $\boldsymbol{s}=[1\ \ 0]^t$, $\boldsymbol{d}=[1\ \ 1]$: (a) PE_0, (b) $\text{PE}_{k>0}$.

AT-complexity of our *seven* architectures in Subsection 3.2 can be estimated for binary multiplication as shown in TABLE 3.11. T_A and T_X denote the delay of AND and XOR logic gates, respectively.

Table 3.11: AT requirements of different Binary Polynomial Multipliers

Design	**#PEs**	$(+)$	$(\times)$	**MUX**	**Flip-Flop**	**Latency**	**CPD**
#1	m	m	m	0	$2m-1$	$w+1$	$\text{T}_\times$+T_+
#2,#3	m	m	m	0	$2m$	$w+1$	$\text{T}_\times$+T_+
#4	m	m	m	0	$2m-1$	$w+y+1$	$\text{T}_\times$+T_+
#5	m	m	m	0	$2m-1$	$2w+y+1$	$\text{T}_\times$+T_+
#6	m	$2m$	$2m$	0	$3m$	$(w+1)/2$	$\text{T}_\times$+T_+
#7	$m/2$	m	m	0	$2m$	$w+1$	$\text{T}_\times$+T_+

3.3 Generalized Polynomial Division and Multiplication

In order to determine the maximum number of nodes and cycles in the worst case scenarios of Equations (3.1), (3.2) and (3.5), the following extreme cases will be considered:

- $\boldsymbol{n} = n_{max} = m-1$, $\boldsymbol{w} = m-n = 1$: There are m instances of input $\boldsymbol{a}_k$ ($k \leq m-1$) and *two* instances of input $\boldsymbol{q}_{j_k}$ ($k \leq 1$), with *nine* instances of input $\boldsymbol{g}_k$ and output $\boldsymbol{q}_{i_k}\&\boldsymbol{r}_k$.
- $\boldsymbol{n} = n_{min} = 1$, $\boldsymbol{w} = m-n = m-1$: There are m instances of input $\boldsymbol{q}_{j_k}$ ($k \leq m-1$) and *two* instances of input $\boldsymbol{a}_k$ ($k \leq 1$), with *nine* instances of input $\boldsymbol{g}_k$ and output $\boldsymbol{q}_{i_k}\&\boldsymbol{r}_k$.
- $\boldsymbol{y} = m-1$: There are m instances of input $\boldsymbol{v}_k$ and *eight* instances of output $\boldsymbol{p}_k$.

3.3.1 Generalized DAG for Polynomial Division when $s = [0\ \ 1]$

With $m = 8$ and $\boldsymbol{s} = [0\ \ 1]$, the DAG for generalized polynomial division in Fig. 3.33 is obtained, where number of cells always $= m$. There are two criteria to look at our DAG. First, a fully generalized, where number of cycles is fixed at m, $\boldsymbol{a_0}$ is stored in PE_{m-1} and $\boldsymbol{g_m}$ is applied to PE_{m-n}. The second is semi-generalized, where number of cycles $= \boldsymbol{Q}_{deg}$ and both $\boldsymbol{a_n}$ and $\boldsymbol{g_m}$ are stored/applied to PE_0 each GCD iteration. In this research, we are only interested in the second criteria, where number of cycles is variable every iteration.

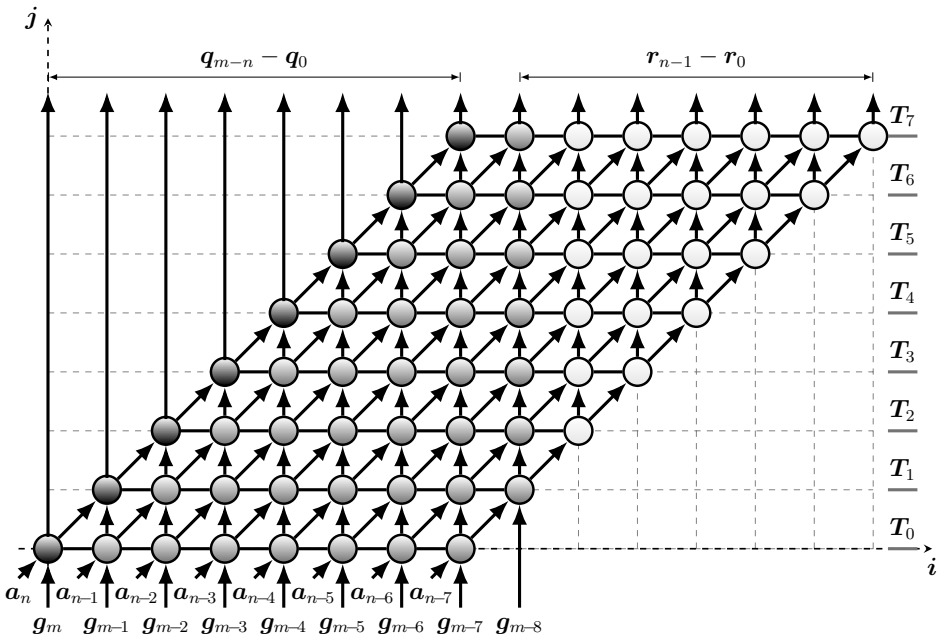
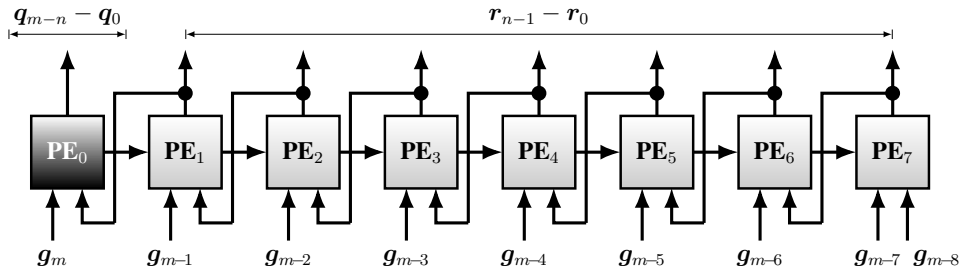


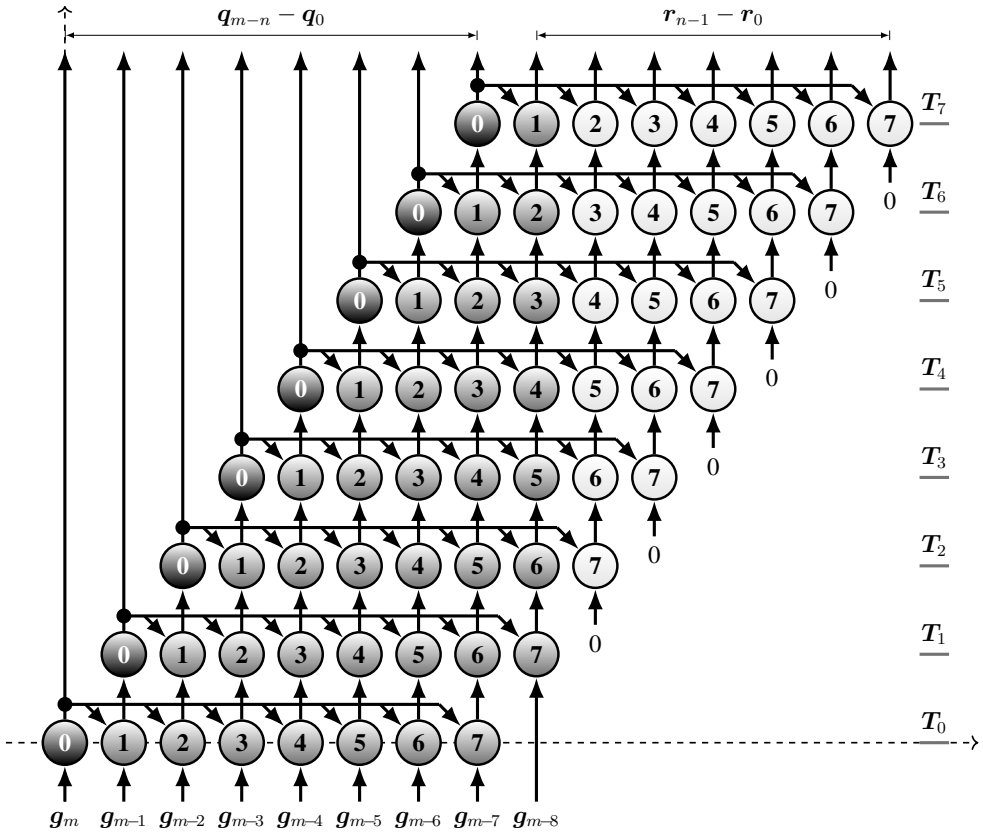
Figure 3.33: Generalized Polynomial Division **DAG**, when $m = 8$ and $\boldsymbol{s}_1 = [0\ \ 1]$

Design #1, when $s = [0\ \ 1]$ and $d_1 = [1\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to the matrix $\boldsymbol{P}_1$ consists of $(m = 8)$ PE's as illustrated in Fig.3.34a. While *PE* activity is shown in Fig. 3.34b.



(a)



(b)

Figure 3.34: Generalized Polynomial Division when $m = 8$, $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$: (a) The $\overline{\text{DAG}}$, (b) **PE** activity of the resulting array.

3.3.2 Generalized DAG for Polynomial Multiplication when $s = [0\ \ 1]$

Similar to division, the DAG for generalized polynomial multiplication with m PE's when $m = 8$ and $\boldsymbol{s} = [0\ \ 1]$is depicted in Fig. 3.35. However, a mix of fully and semi generalized criteria is utilized with our polynomial multiplication, where $\boldsymbol{v_0}$ is stored in PE_{m-1} and number of cycles $= \boldsymbol{Q}_{deg}$. In other words, the maximum of $\boldsymbol{V}_{deg}$ is considered, i.e., $y = m-1$, and multiplication is completed simultaneously with division.

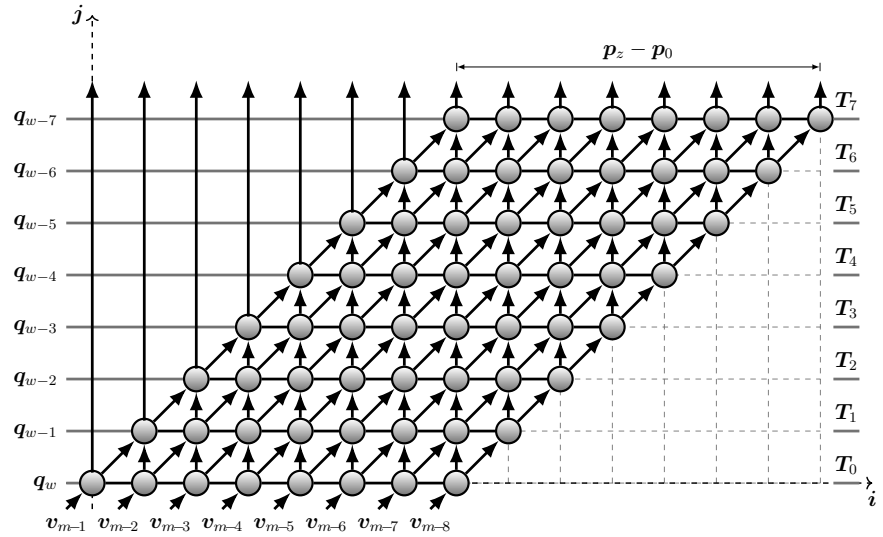


Figure 3.35: Generalized Polynomial Multiplication **DAG** $m = 8$ and $\boldsymbol{s} = [0\ \ 1]$

Design #1, when $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}$ consists of ($m = 8$) PE's as illustrated in Fig.3.36.

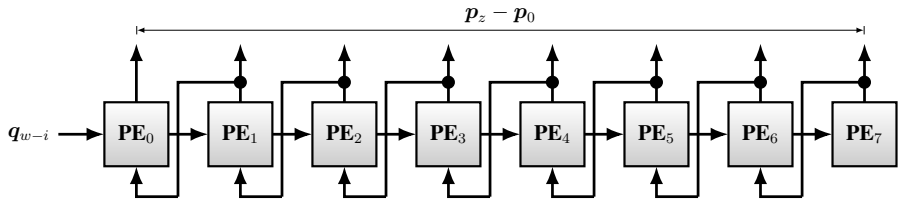


Figure 3.36: Generalized Polynomial Multiplication $\overline{\text{DAG}}$, $m = 8$, $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [1\ \ 1]^t$.

Chapter 4

Systolic Architectures for Concurrent EEA over Extension Fields

Designing a finite field inversion unit, with desirable performance and area-time complexity, requires exploring the design space of the algorithm to examine its potential for concurrency improvement. The systolic finite fields inverters developed in this work are based on our reformulation of EEA in Algorithm 3, accommodating the methodology in [18].

4.1 Literature Survey on EEA-based Field inversion

Generally, many systolic architectures have been proposed to realize EEA and its variants showing feasible AT-complexities. At the beginning, Brent and Kung [25] proposed a new reformulation of the iterative Euclidean GCD algorithm and suggested that their systolic architecture can be extended to perform EEA-based inversion. They built their GF(2^m)-GCD reformulation on a degree reduction technique called *GCD-preserving Transformation* and resolved the algorithm irregularity by replacing the comparison of polynomial degrees with a simple counter [39, 91]. In contrast to the traditional algorithms, their algorithm operates on the Least Significant Bit (LSB) first, maintaining the same number of iterations, $O(n)$, and having the next iteration starts when the LSB of the previous iteration is found. During the GCD computation, their $(m+n+1)$ array architecture tests only the mth coefficients of two polynomials, performing two operations in parallel [92]. Accordingly, Araki et al. [93] and Doyle et al. [94] proposed their versions of polynomial EEA-based inversion systolic architectures over GF(2^m), manipulation polynomial division and multiplication processes dependently on the size of finite field [84].

4.2 Concurrent EEA-Based Inversion over Extension Fields

Having polynomial division and multiplication realized as systolic architectures, a concurrent divider/multiplier consisting of ($2m = 16$) PE's can be built by connecting the DAG/$\overline{\text{DAG}}$'s of polynomial divider and multiplier as shown in Fig. 4.1.

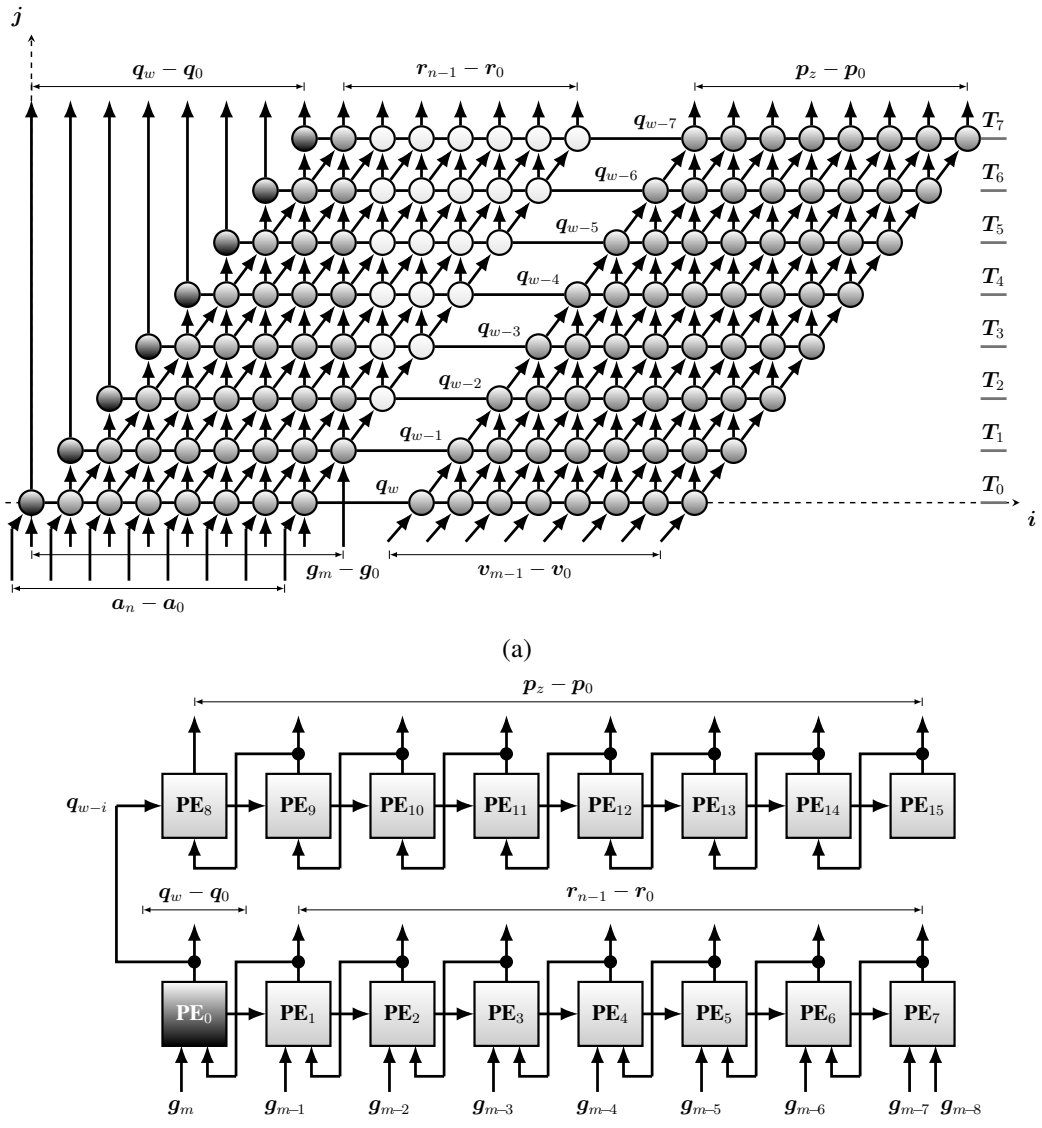


Figure 4.1: Concurrent Polynomial Divider/Multiplier, $m = 8$: (a) **DAG** when $\boldsymbol{s}_1 = [0\ \ 1]$, (b) $\overline{\text{DAG}}$ when $\boldsymbol{d} = [1\ \ 1]^t$

However, our concurrent divider/multiplier will be developed by applying the methodology in [18] on the iterative equations derived for polynomial devision and multiplication conjointly. The resulting systolic architectures will be utilized to build EEA inverters based on Algorithm 3, which includes an extra process of polynomial addition followed by data exchange and alignment in conjunction with division and multiplication every iteration. Before proceeding with our design process, let us recal Algorithm 3, combining polynomial addition in step **11** with the multiplication inside *for* loop by initializing $\boldsymbol{P}$ in step **3** with $\boldsymbol{U}$. Algorithm 4 presents the multiplier–accumulator (MAC) version of our modified EEA-based inversion over prime extension fields.

Algorithm 4 Modified EEA over prime extension fields: MAC version

Input: Polynomials $\boldsymbol{G}$ of degree m, and $\boldsymbol{A}$ of degree $n < m$
Output: $\boldsymbol{A}^{-1} \mod \boldsymbol{G}$
1: $\boldsymbol{U} \leftarrow 0; \boldsymbol{V} \leftarrow 1;$
2: **while** $R \neq 1$ ($\gcd \neq 1$) **do**
3: $\quad \boldsymbol{R} \leftarrow \boldsymbol{G}; \boldsymbol{P} \leftarrow U; \boldsymbol{Q} \leftarrow 0;$
4: $\quad$ **for** $(i = G_{deg} - A_{deg}; i > 0; i--)$ **do** {*Concurrent Division/Multiplication*}
5: $\quad\quad \boldsymbol{A} \leftarrow A << i;$
6: $\quad\quad \boldsymbol{Q(i)} \leftarrow R(i+n);$
7: $\quad\quad \boldsymbol{R} \leftarrow R + (Q(i) \times A);$
8: $\quad\quad \boldsymbol{V} \leftarrow V << i;$
9: $\quad\quad \boldsymbol{P} \leftarrow P + (Q(i) \times V);$
10: $\quad$ **end for**
11: $\quad \boldsymbol{U} \leftarrow V;$
12: $\quad \boldsymbol{V} \leftarrow P;$
13: $\quad \boldsymbol{G} \leftarrow A;$
14: $\quad \boldsymbol{A} \leftarrow R;$
15: **end while**
16: **return** $\boldsymbol{V}$

Systolic inversion architectures, which accommodate all these processes, will be developed in this section, accordingly. First, a concurrent divider/MAC will be constructed to perform a single iteration of *wile* loop in Algorithm 4. The resulted circuit will replace the *for* loop, which occupies steps **4** to **10**, and $\boldsymbol{P}$ initialization in step **3**. Then, data exchange and alignment will be accommodated, allowing for the algorithm to be carried through based on a technique we call it "shift and inverse".

4.2.1 Concurrent Polynomial Division and MAC over GF(p^m)

Reconsidering the iterative equations derived for polynomial devision and including the accumulation process with multiplication, our divider/MAC iterative equations can be written as follows:

$$\boldsymbol{q_{w-i}} = (\boldsymbol{g_{m-i}} - \textstyle\sum_{j=0}^{i-1}\{\boldsymbol{a_{n-(i-j)}} \times \boldsymbol{q_{w-j}}\})/\boldsymbol{a_n}, \quad \text{where } 0 \le i \le w \tag{4.1}$$

$$\boldsymbol{r_{n-i}} = \boldsymbol{g_{m-i}} - \textstyle\sum_{j=0}^{i-1}\{\boldsymbol{a_{n-(i-j)}} \times \boldsymbol{q_{w-j}}\} \quad \text{where } w+1 \le i \le m \tag{4.2}$$

$$\boldsymbol{p_{z-i}} = \boldsymbol{u_{z-i}} + \textstyle\sum_{j=0}^{i}\{\boldsymbol{v_{y-(i-j)}} \times \boldsymbol{q_{w-j}}\}, \quad \text{where } 0 \le i \le m-1 \tag{4.3}$$

Correspondingly, the DG, DAG and Projected DAG ($\overline{\text{DAG}}$) are developed to build our divider/MAC systolic architectures.

4.2.1.1 Dependence Graph of the Divider/MAC

Since there are *two* indices, $\boldsymbol{i}$ and $\boldsymbol{j}$, Equations (4.1), (4.2) and (4.3) can be interpreted into **2-D** graph. Instances of each variable are described by line equations in $\mathbb{D}$ as follows:

Inputs $\boldsymbol{g}_{m-i}$ and Outputs $\boldsymbol{q}_{w-i}$ & $\boldsymbol{r}_{n-i}$: $\boldsymbol{i}=m-k$, $(0\le k\le m)$
Inputs $\boldsymbol{q}_{w-j}$: $\boldsymbol{j} = w-k$ $(0 \le k \le w)$,
Inputs $\boldsymbol{a}_{n-i+j}$ and $\boldsymbol{v}_{y-i+j}$: $\boldsymbol{i}-\boldsymbol{j} = n, y-k$ $(0 \le k \le n, y)$,
Inputs $\boldsymbol{u}_{z-i}$ and Outputs $\boldsymbol{p}_{z-i}$: $\boldsymbol{i} = z-k$ $(0 \le k \le z)$.

Finally, the term $(/a_n)$ in Equation (4.1) can be interpreted as points (k,k), where $0 \le k \le w$. The DG of our concurrent divider/MAC, when $m=8$, and $n=5$ is illustrated in Fig. 4.2. Note that #PE$_{max} = m$, in which PE$_{m-1}$ accepts $\boldsymbol{g}_1, \boldsymbol{g}_0$ or 0 producing $\boldsymbol{p}_0^{(i)}$, $\boldsymbol{r}_1^{(i)}$ and $\boldsymbol{r}_0^{(i+1)}$.

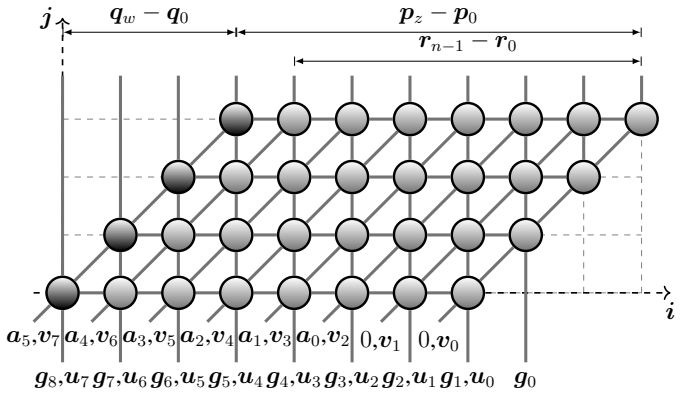


Figure 4.2: Concurrent Div/Mul **DG** $m=8$, $n=5$, $y=2$

Each node $(\boldsymbol{i}=\boldsymbol{j})$ in Fig. 4.2 receives *four* inputs and produces *two* outputs, $\boldsymbol{q}_{i_k}$ and $\boldsymbol{p}_k$, performing *three* $GF(p)$ operations. While, each node $(\boldsymbol{i} \neq \boldsymbol{j})$ accepts *five* inputs and produces *two* outputs, $\boldsymbol{r}_k$ and $\boldsymbol{p}_k$, performing *four* $GF(p)$ operations. Each node $(\boldsymbol{i}, \boldsymbol{j})$ bypasses $\boldsymbol{a_k}$, $\boldsymbol{v}_k$ and $\boldsymbol{u}_k$ to the node $(\boldsymbol{i}+1, \boldsymbol{j}+1)$. Whereas, input $\boldsymbol{q}_{j_k}$ is broadcast from nodes $(\boldsymbol{j}, \boldsymbol{j})$ to nodes $(\boldsymbol{i}>\boldsymbol{j}, \boldsymbol{j})$. Nodes operations can be expressed as follows:

$$\begin{aligned}
\boldsymbol{q}_{i_k}(j,j) &= (\boldsymbol{g}_m, \boldsymbol{r}_k(j-1, j-1))/a_n, \\
\boldsymbol{r}_k(i,j) &= \boldsymbol{g}_k, \boldsymbol{r}_k(i-1, j-1) - (\boldsymbol{a}_k \times \boldsymbol{q}_{j_k}), \\
\boldsymbol{p}_k(i,j) &= \boldsymbol{u}_k, \boldsymbol{p}_k(i-1, j-1) + (\boldsymbol{v}_k \times \boldsymbol{q}_{j_k}).
\end{aligned}$$

4.2.1.2 The Scheduling Function for the Divider/MAC

From Equations (4.1), (4.2) and (4.3), the base vectors of the dependence matrix and the nullvectors of all variables can be presented as follows:

$$\begin{aligned}
\boldsymbol{b}_{a,v} = [1 \quad -1] &\Rightarrow \boldsymbol{e}_{a,v} = [1 \;\; 1], \\
\boldsymbol{b}_{q_j} = [0 \;\; 1] &\Rightarrow \boldsymbol{e}_{q_j} = [1 \;\; 0], \\
\boldsymbol{b}_{g,u,q_i,r,p} = [1 \;\; 0] &\Rightarrow \boldsymbol{e}_{g,u,q_i,r,p} = [0 \;\; 1].
\end{aligned}$$

Variable $\boldsymbol{q}_{j_k}$, which is represented horizontally in Fig. 4.2, is produced from nodes $(\boldsymbol{j}, \boldsymbol{j})$, and broadcast to nodes $(\boldsymbol{i}>\boldsymbol{j}, \boldsymbol{j})$. Moreover, nodes in a row $\boldsymbol{j+1}$ must wait until nodes in row $\boldsymbol{j}$ are executed. Thus, the time value associated with the node $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ is greater than the time value associated with node $(\boldsymbol{i}, \boldsymbol{j})$, i.e., $T_{(\boldsymbol{i}+1,\boldsymbol{j}+1)} > T_{(\boldsymbol{i},\boldsymbol{j})}$, which can be expressed in *affine scheduling* as follows:

$$\begin{bmatrix} s_1 & s_2 \end{bmatrix} \times \begin{bmatrix} \boldsymbol{i}+1 \\ \boldsymbol{j}+1 \end{bmatrix} - s > \begin{bmatrix} s_1 & s_2 \end{bmatrix} \times \begin{bmatrix} \boldsymbol{i} \\ \boldsymbol{j} \end{bmatrix} - s \tag{4.4}$$

Equation (4.4) indicates that the scheduling function of our Divider/MAC can be declared as: $s_1 + s_2 > 0$, in which there are *three* possible scheduling vectors as follows:

$$\boldsymbol{s}_1 = [0 \;\; 1], \qquad \boldsymbol{s}_2 = [1 \;\; 0], \qquad \boldsymbol{s}_3 = [1 \;\; 1].$$

Based on the obtained nullvectors, if the scheduling vector $\boldsymbol{s}_1$ is selected, variables $\boldsymbol{a}_k$, $\boldsymbol{g}_k$, $\boldsymbol{v}_k$, $\boldsymbol{u}_k$, $\boldsymbol{q}_{i_k}$, $\boldsymbol{r}_k$ and $\boldsymbol{p}_k$ are *pipelined*, while variable $\boldsymbol{q}_{j_k}$ is *broadcast*. Whereas, selecting $\boldsymbol{s}_2$ results in *pipelined* variables $\boldsymbol{q}_{j_k}$, $\boldsymbol{a}_k$ and $\boldsymbol{v}_k$, and *broadcast* variables $\boldsymbol{g}_k$, $\boldsymbol{u}_k$, $\boldsymbol{q}_{i_k}$, $\boldsymbol{r}_k$ and $\boldsymbol{p}_k$. Meanwhile, all variables will be *pipelined* if $\boldsymbol{s}_3$ is selected [18]. Design with $\boldsymbol{s}_1$ is preferable since it matches the time constraint on $\boldsymbol{q}_{j_k}$ and outperforms $\boldsymbol{s}_2$ and $\boldsymbol{s}_3$ in terms of latency.

Figure 4.3 illustrates the DAG of concurrent divider/MAC, when $m = 8$, $n = 5$ and $\boldsymbol{s}_1$ is selected. Note that with $\boldsymbol{s}_1$, number of cycles is relative to $\boldsymbol{Q}_{deg}$, i.e., $=m-n+1$.

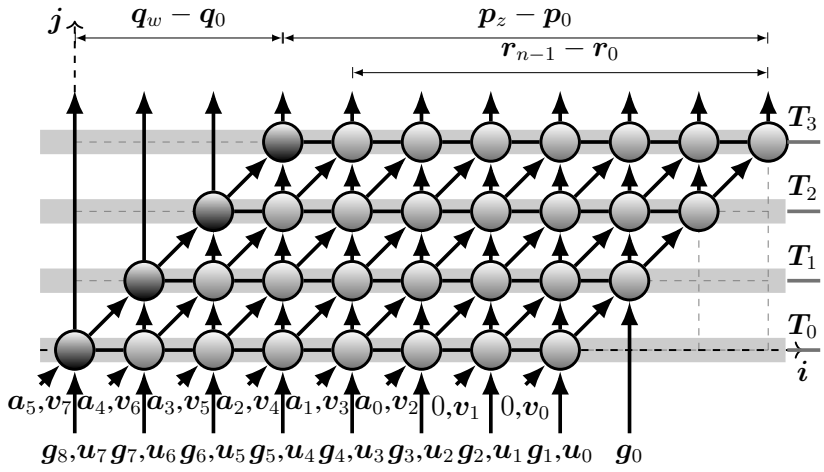


Figure 4.3: **DAG** of Concurrent Divider/MAC, when $\boldsymbol{s} = [0\ \ 1]$, $m{=}8$, $n{=}5$

Nonetheless, Fig. 4.4 depicts the DAG of our concurrent divider/MAC when $m = 8$, $n = 5$ and $\boldsymbol{s}_2 = [1\ \ 0]$ is selected. Accordingly, number of cycles is the sum of $\boldsymbol{Q}_{deg}$ and nodes count, i.e., $m-n+m=2m-n$.

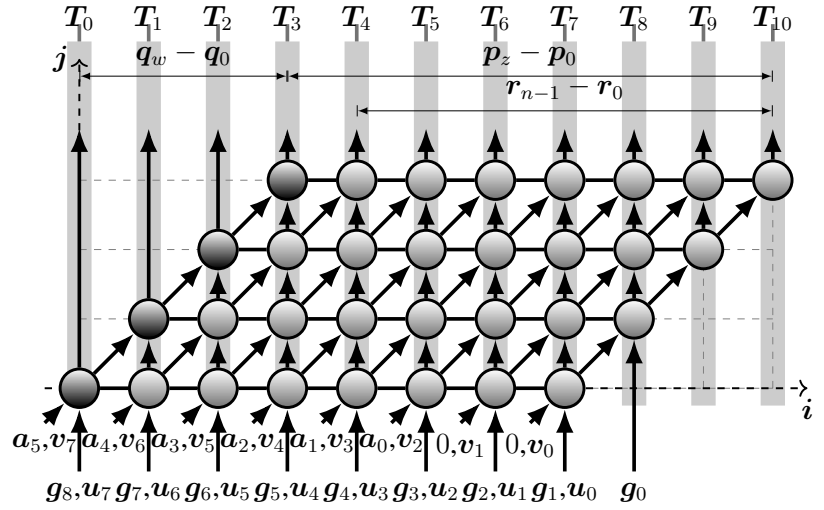


Figure 4.4: Concurrent Divider/MAC **DAG**: $m = 8$, $n = 5$, $\boldsymbol{s}_2 = [1\ \ 0]$

The DAG of our concurrent divider/MAC with $m = 8$ and $n = 5$, selecting $\boldsymbol{s}_3 = [1\ \ 1]$, is shown in Fig. 4.5, where number of cycles is $2(m-n)+m=3m-2n$.

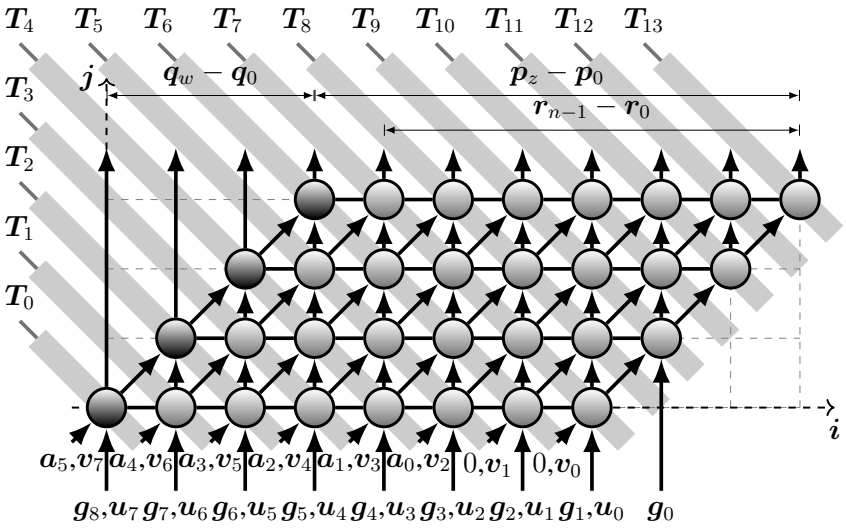


Figure 4.5: Concurrent Divider/MAC **DAG**: $m = 8$, $n = 5$, $\boldsymbol{s}_3 = [1\ \ 1]$

4.2.1.3 Design Space Exploration when $s = [0\ \ 1]$

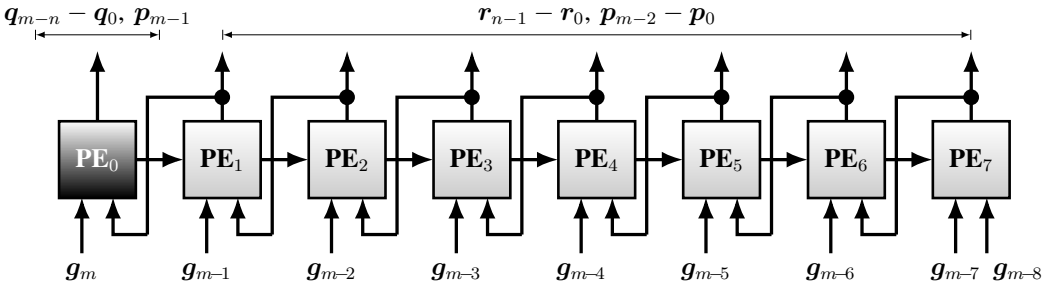
Since two nodes on one equitemporal plane can not be mapped to the same point in $\overline{\text{DAG}}$, i.e., $\boldsymbol{s} \times \boldsymbol{d} \neq 0$ [18], projection directions $\boldsymbol{d}$ can have *three* choices with *three* corresponding matrices ($\boldsymbol{P}$), when $\boldsymbol{s}=[0\ \ 1]$ as follows:

$\boldsymbol{d}_1 = [1\ \ 1]^t,\ \boldsymbol{P}_1 = [1\ \ -1]$,
$\boldsymbol{d}_2 = [-1\ \ 1]^t,\ \boldsymbol{P}_2 = [1\ \ 1]$,
$\boldsymbol{d}_3 = [0\ \ 1]^t,\ \boldsymbol{P}_3 = [1\ \ 0]$.

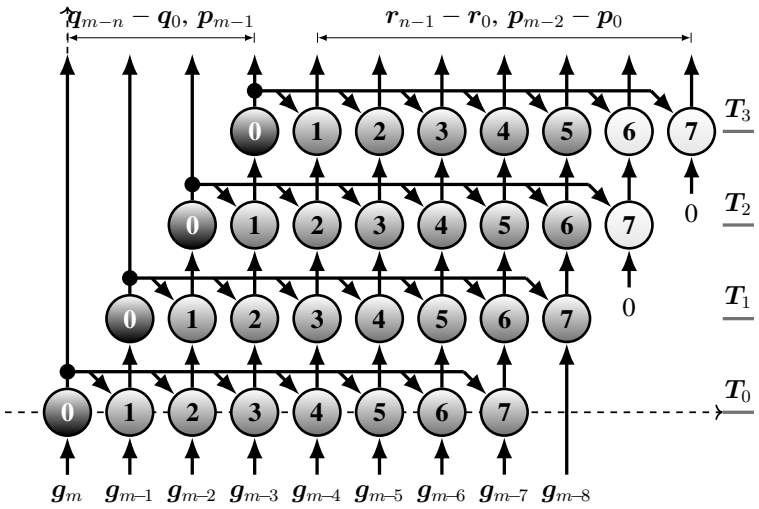
Because inputs $\boldsymbol{a}_{(i,j)}$, $\boldsymbol{v}_{(i,j)}$ and $\boldsymbol{u}_{(i,j)}$ are bypassed to cells $(\boldsymbol{i}+1, \boldsymbol{j}+1)$ in Fig. 4.3, they are stored in the PE, while $\boldsymbol{g}_k$ and $\boldsymbol{q}_{j_k}$ will be treated as inputs. For all $\boldsymbol{d}$ choices, there are m PE's and the latency is $m-n+1$ cycles. There are *three* different PE structures with $\boldsymbol{d}_1$ and only *one* with $\boldsymbol{d}_2$ and $\boldsymbol{d}_3$. However, projection with $\boldsymbol{d}_2$ and $\boldsymbol{d}_3$ requires extra circuitry to *right-rotate* the stored variables and update the broadcast variable ($\boldsymbol{q}_{j_k}$) each cycle. In addition, their PE occupy more area and require the initial inputs and the final outputs to be rearranged. Therefore, design with $\boldsymbol{d}_1$ outperforms $\boldsymbol{d}_2$ or $\boldsymbol{d}_3$ in terms of AT-complexity.

Design #1, with projection direction $d_1 = [1\ \ 1]^t$

Selecting $\boldsymbol{d}_1$, the $\overline{\text{DAG}}$ of concurrent divider/MAC corresponding to $\boldsymbol{P}_1$, when $m=8$, $n=5$, $\boldsymbol{s}=[0\ \ 1]$ and $\boldsymbol{d}=[1\ \ 1]^t$, is depicted in Fig. 4.6a. At time step i, each $\text{PE}^{(i)}_{k<m-1}$ accepts $\boldsymbol{g}_k$, or $\boldsymbol{r}^{(i-1)}_{k+1}$ and $\boldsymbol{p}^{(i-1)}_{k+1}$, producing $\boldsymbol{r}^{(i)}_k$ and $\boldsymbol{p}^{(i)}_k$, in which PE activity in Fig. 4.6b is realized. Only $\text{PE}^{(i)}_{m-1}$ accepts $\boldsymbol{g}_1$, $\boldsymbol{g}_0$ or 0, producing $\boldsymbol{r}^{(i)}_0$ and $\boldsymbol{p}^{(i)}_0$.



(a)



(b)

Figure 4.6: Concurrent Divider/MAC, $\boldsymbol{d}=[1\ \ 1]^t$: (a) $\overline{\text{DAG}}$, (b) **PE** activity.

This concurrent divider/MAC can perform only *one* iteration of the EEA algorithm. In order to execute EEA-based inversion completely, the concurrent division/MAC process is performed repeatedly until the inverse is produced. Therefore, the processing details of PE_0, PE_i, and PE_{m-1} are developed according to their activities and enhanced to facilitate data flow and processes synchronization. They exchange variables and align them avoiding degrees revaluation, despite the decrease of data size after each EEA iteration.

The details of PE_0 are illustrated in Fig. 4.7, where registers $\boldsymbol{a}_k$, $\boldsymbol{v}_k$ and $\boldsymbol{u}_k$ will be initialized with $\boldsymbol{a}_{in_n}$, $\mathbf{0}$ and $\mathbf{0}$, in the first iteration of the EEA algorithm.

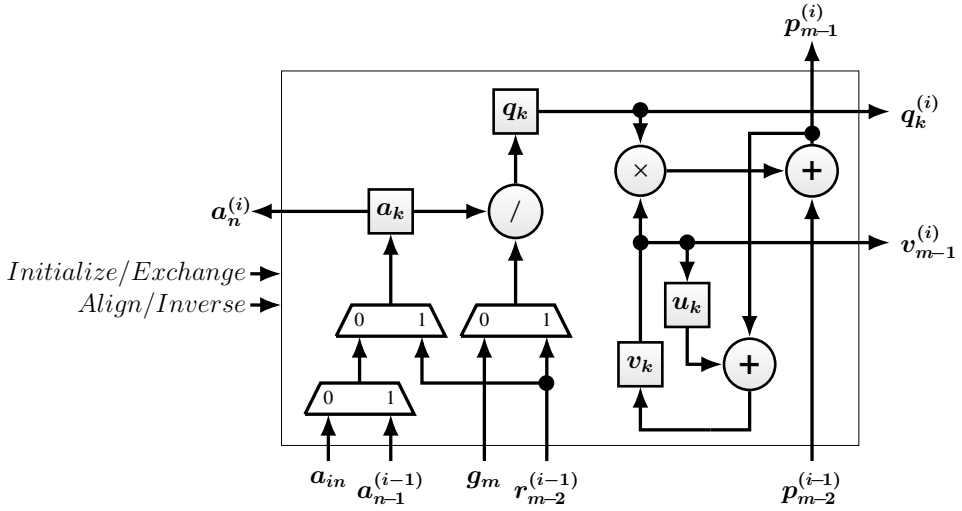


Figure 4.7: **EEA PE**$_0$ Details when $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$.

Whereas, the details of $PE_{0<k<m-1}$ are presented in Fig. 4.8, where registers $\boldsymbol{g}_k$, $\boldsymbol{a}_k$, $\boldsymbol{v}_k$ and $\boldsymbol{u}_k$ will be initialized with $\boldsymbol{g}_{in_{m-k}}$, $\boldsymbol{a}_{in_{n-k}}$, $\mathbf{0}$ and $\mathbf{0}$, in the first iteration of the algorithm.

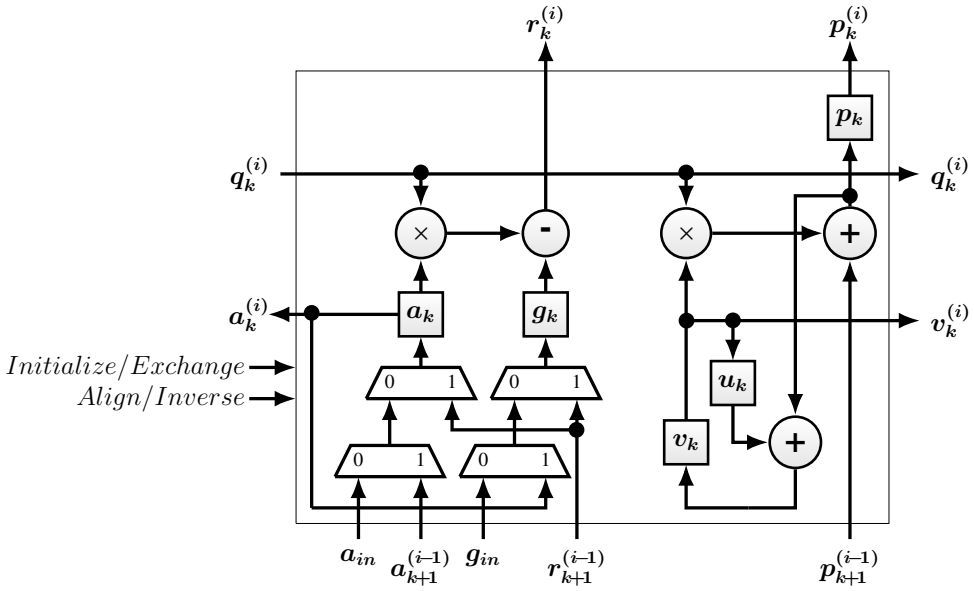


Figure 4.8: **EEA PE**$_{0<k<m-1}$ Details when $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$

However, the details of PE_{m-1} are shown in Fig. 4.9, where registers $\boldsymbol{g}_k$, $\boldsymbol{a}_k$, $\boldsymbol{v}_k$ and $\boldsymbol{u}_k$ will be initialized with $\boldsymbol{g}_{in_1}$, $\boldsymbol{a}_{in_0}$, $\mathbf{1}$ and $\mathbf{0}$, in the first iteration of the EEA algorithm.

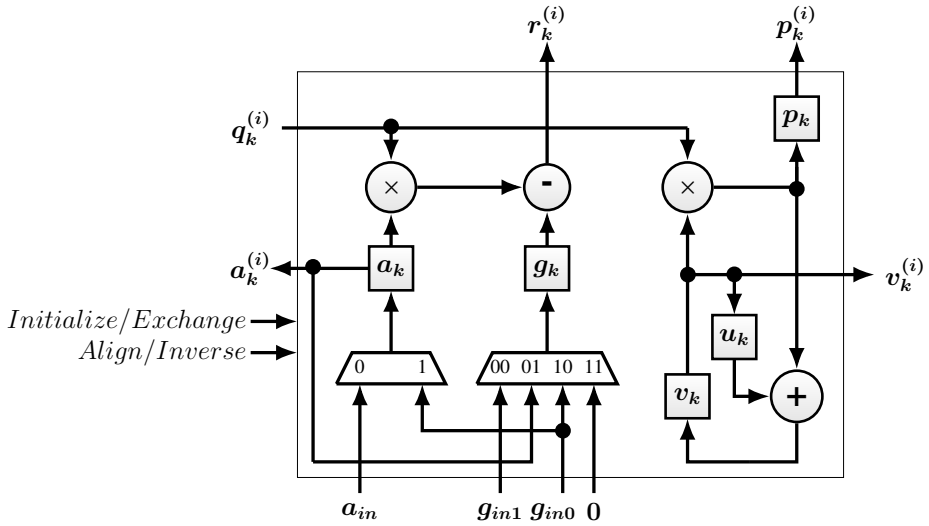


Figure 4.9: **EEA** $\textbf{PE}_{m-1}$ Details when $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d}=[1\ \ 1]^t$

Control Mechanism

During each iteration of the concurrent inversion, only one bit of $\boldsymbol{Q}$ is active each cycle. Therefore, the b[1]-bit register $\boldsymbol{q_k}$ in Fig. 4.7 satisfies the requirement. Moreover, registers $\boldsymbol{g_k}$, $\boldsymbol{a_k}$, $\boldsymbol{u_k}$, $\boldsymbol{v_k}$ and $\boldsymbol{p_k}$ can accommodate $\boldsymbol{G}$, $\boldsymbol{A}$, $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{P}$. Initially at $(i = 0)$, $\boldsymbol{g_k}$'s store $\boldsymbol{g_{in}}$'s and $\boldsymbol{a_k}$'s store $\boldsymbol{a_{in}}$'s, while $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{P}$ are reset to 0, 1 and 0, respectively. Subsequently, $\boldsymbol{g_k}$'s store $\boldsymbol{a_k}$'s as the new dividend, while $\boldsymbol{a_k}$'s store *left-shift*ed $\boldsymbol{R}$ as the new divisor, at the beginning of each iteration $(i > 0)$. Meanwhile, $\boldsymbol{u_k}$'s store $\boldsymbol{v_k}$'s and $\boldsymbol{v_k}$'s store the MAC result ($\boldsymbol{P}+\boldsymbol{U}$). After each exchange, $\boldsymbol{p_k}$'s are reset and $\boldsymbol{A}$ is *left-shift*ed to be aligned with $\boldsymbol{G}$, in which PE_0 accepts $\boldsymbol{g_m}$ always. Each iteration of the inversion takes the same number of cycles required to align $a_n^{(i)}$ with g_{m-i}.

We name the control technique utilized in our design "***Align and Inverse***", where couple of signals control MUXs and registers in each PE. Every time $\boldsymbol{a_k}$'s are updated, the *left-shift* process starts immediately, and continues until $\boldsymbol{a_n} = \mathbf{1}$ in PE_0. Consequently, the inversion process is performed for the same number of cycles. Furthermore, a simple ***up-down counter*** is employed to control the switch between *align* and *inverse* operation modes.

[1] b = number of bits representing each coefficient, e.g., $b=1$ for $GF(2)$ and $b = 2$ for $GF(3)$.

Control signals generator (CSG) is illustrated in Fig. 4.10, where signals, *Align/Inverse* and *Exchange/Initialize*, control MUXs selections and registers enabling in the PE's.

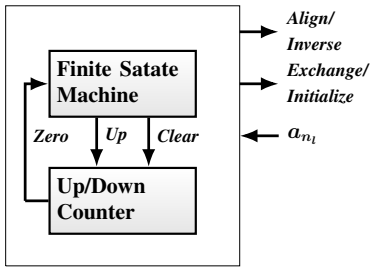


Figure 4.10: Control Signals ($Align/Inverse$) Generator (CSG),

The control mechanism of our design works as follows:

At T_0, PE's accepts the system inputs, $\boldsymbol{g}_{in_{m\rightarrow 1}}$ and $\boldsymbol{a}_{in_{n\rightarrow 0}}$, while $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{P}$ are initialized to 0, 1 and 0, respectively.

At T_1, $\boldsymbol{a_n}$ is zero-checked and $\boldsymbol{A}$ is *left-shifted* until $\boldsymbol{a_n} \neq 0$, counting-up number of shifts.

At $T_{w=m-n}$, ***Align*** mode ends and ***Inverse*** starts, producing $\boldsymbol{q_w}$, $\boldsymbol{r_k^{(1)}}$ and $\boldsymbol{p_k^{(1)}}$, selecting g_0 as an input to PE$_{m-1}$, and signaling the counter to decrement.

At T_{w+1}, the second cycle of ***Inverse*** starts, producing $\boldsymbol{q_{w-1}}$, $\boldsymbol{r_k^{(2)}}$ and $\boldsymbol{p_k^{(2)}}$, while selecting 0 as the input to PE$_{m-1}$. The ***Inverse*** operation mode continues until the counter is *Zero*, generating $\boldsymbol{R}$ and $\boldsymbol{P}$ of the first EEA iteration.

At T_{2w}, the exchange, $\boldsymbol{G} \leftarrow \boldsymbol{A}$ and $\boldsymbol{A} \leftarrow \boldsymbol{R}$, is executed, enabling $\boldsymbol{v_k}$'s and $\boldsymbol{u_k}$'s to store $\boldsymbol{P}+\boldsymbol{U}$ and $\boldsymbol{V}_{\text{old}}$, respectively, and boosting the counter as $\boldsymbol{r_{m-2}}$ is stored in $\boldsymbol{a_k}$ of PE$_0$. Moreover, $\boldsymbol{P}$ is reset to be ready for the second EEA iteration.

At T_{2w+1}, the process at T_1 is repeated to start the alignment for the second EEA iteration.

At T_{2m-2}, the algorithm is terminated, resulting in $\boldsymbol{G}=1$ and $\boldsymbol{V}=\boldsymbol{A^{-1}}$.

This figure, $2m-2$, is resulted because the number of cycles required to complete one iteration of inversion and number of shifts required to align $\boldsymbol{A}$ with $\boldsymbol{G}$, both equal to $\boldsymbol{Q}_{deg}$.

Latency $= 2[$ Total of $\boldsymbol{Q}_{deg}$'s$]$,
Total of $\boldsymbol{Q}_{deg}$'s $= Q_{deg}^{max} = m-1$ (see section 3.2),
Thus, *Latency* $=$ *Completion Time* $= 2m-2$.

For the following designs we will be interested in $\overline{\text{DAG}}$ and PE activity, considering PE details in **Chapter** 3 and assuming the same CSG as Design #1.

Design #2, with projection direction $d_2 = [-1\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}_2$, when $m=8$, $n=5$, $\boldsymbol{s}=[0\ \ 1]$ and $\boldsymbol{d}=[-1\ 1]^t$, is illustrated in Fig. 4.11a. However, the *eleven* PE's in Fig. 4.11b are reduced to *eight* in Fig. 4.11c, eliminating inactive cells from each row by applying ($\bmod 8$). It is possible to get $n=1$ at any EEA iteration, in which all PE's may act as PE_0 and $\text{PE}_{k>0}$ through the algorithm.

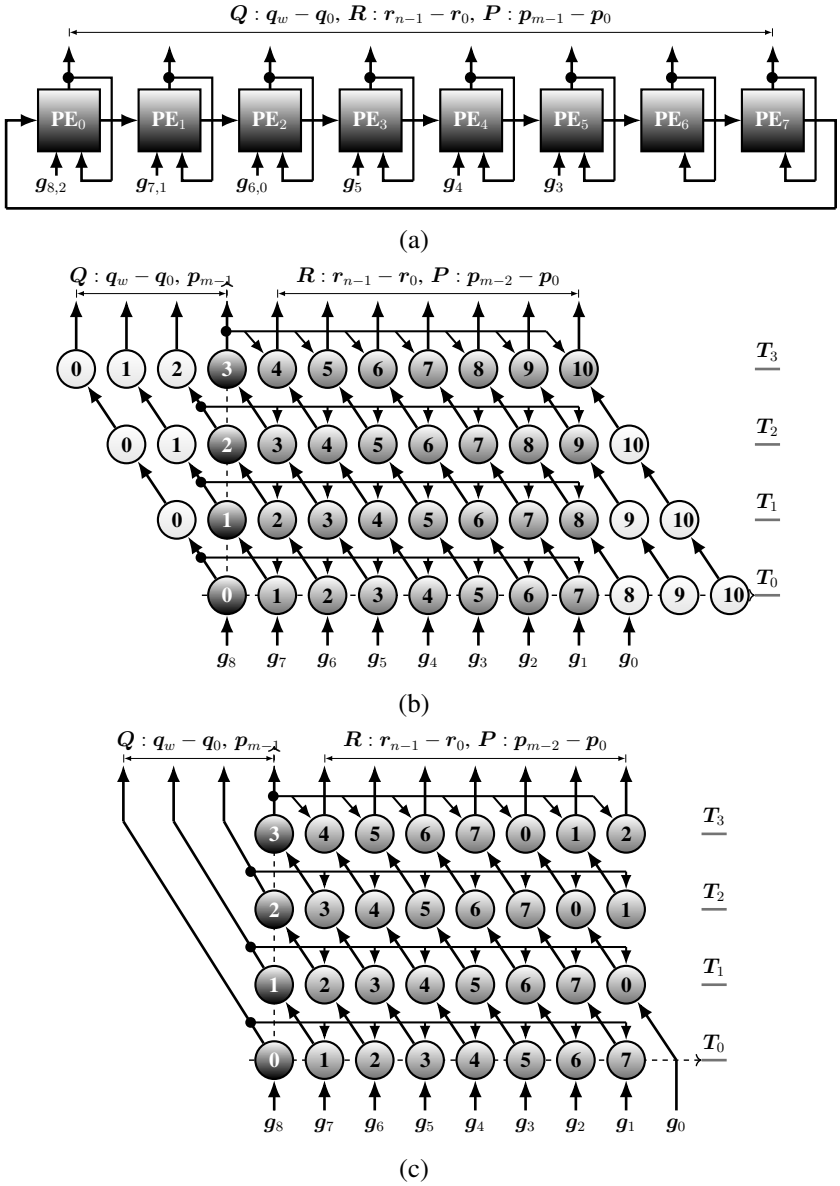


Figure 4.11: Concurrent Divider/MAC, $\boldsymbol{d}=[-1\ 1]^t$: (a) $\overline{\text{DAG}}$, (b,c) 11 **PE**-activity.

Design #3, with projection direction $d_3 = [0\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}_3$, when $m=8$, $n=5$, $\boldsymbol{s}=[0\ \ 1]$ and $\boldsymbol{d}=[0\ \ 1]^t$, is illustrated in Fig. 4.12a. Equivalent to Design #2, the *eleven* nodes of PE-activity in Fig. 4.12b are deducted to *eight* in Fig. 4.12c. For both Design #2 and #3, PE details is the same as PE_k of Design #1 plus 1-**Flip-Flop**, 1-**MUX** and 1-$(/)$.

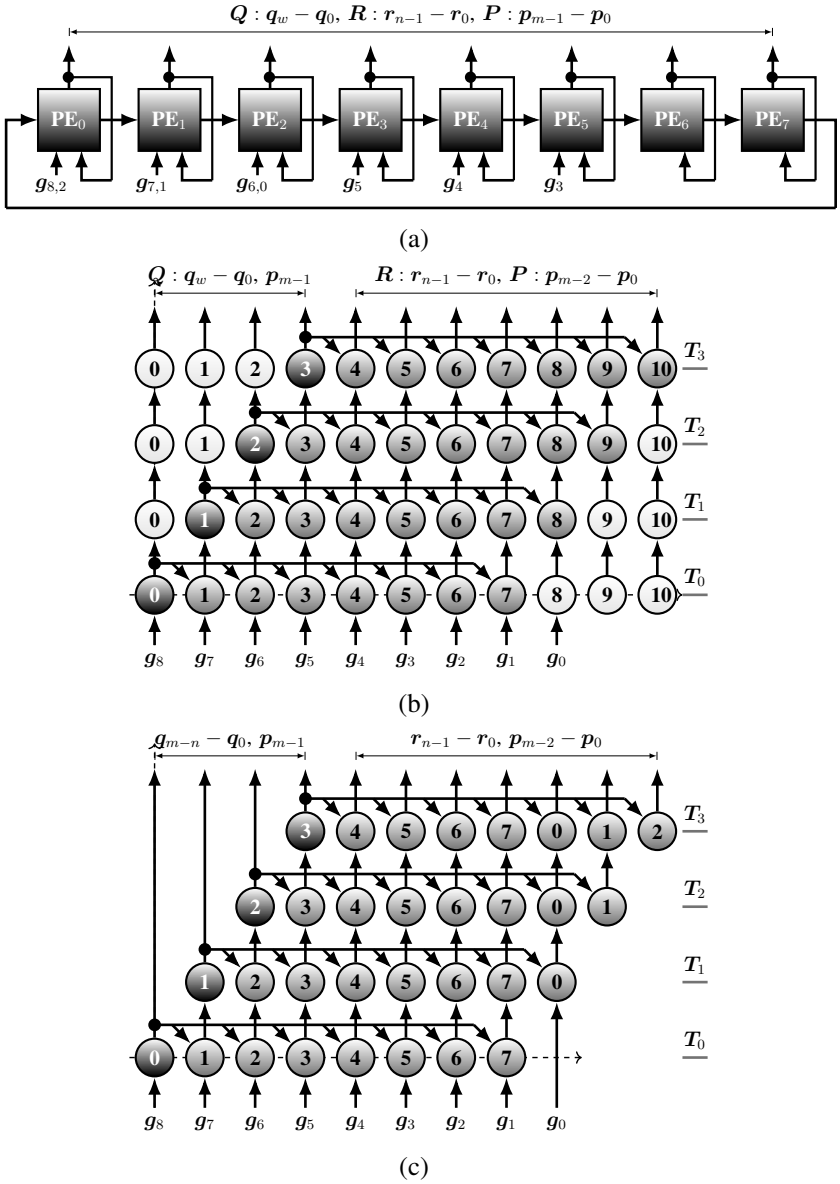


Figure 4.12: Concurrent Divider/MAC, $\boldsymbol{d}=[0\ \ 1]^t$: (a) $\overline{\text{DAG}}$ (b,c) **PE**-activity.

4.2.1.4 Design Space Exploration when $s = [1\ \ 0]$

Similar to design with $\boldsymbol{s} = [0\ \ 1]$, projection direction when $\boldsymbol{s} = [1\ \ 0]$ must satisfy the inequality ($\boldsymbol{s} \times \boldsymbol{d} \neq 0$), in which $\boldsymbol{d}$ will have *three* alternatives as follows [18]:

$$\boldsymbol{d}_1 = [1\ \ 1]^t,\ \boldsymbol{P}_1 = [1\ \ -1], \quad \boldsymbol{d}_2 = [-1\ \ 1]^t,\ \boldsymbol{P}_2 = [1\ \ 1], \quad \boldsymbol{d}_3 = [1\ \ 0]^t,\ \boldsymbol{P}_3 = [0\ \ -1].$$

For all $\boldsymbol{d}$ choices, there is a maximum of (m) PE's and latency of ($2m-n$) cycles.

Design #4, with projection direction $d_1 = [1\ \ 1]^t$

The $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}_1$ and PE's details, when $m=8$, $n=5$, $\boldsymbol{s}=[1\ \ 0]$ and $\boldsymbol{d}=[1\ \ 1]^t$ are equivalent to the ones in Fig. 4.6. However, the activity of PE's is presented in Fig. 4.13 according to the DAG in Fig. 4.4.

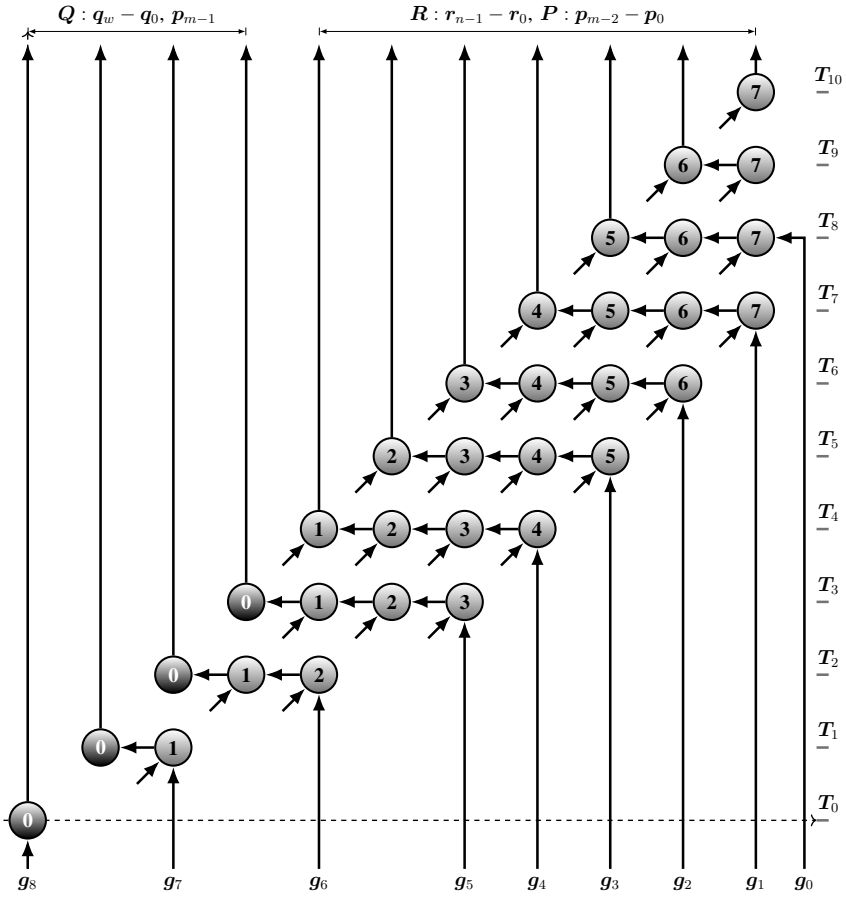


Figure 4.13: Concurrent Divider/MAC, **PE** activity, $m=8$, $n=5$, $\boldsymbol{s}=[1\ \ 0]$, $\boldsymbol{d}=[1\ \ 1]^t$.

4.2.1.5 Design Space Exploration when $s = [1\ \ 1]$

Similarly, projection direction $\boldsymbol{d}$ will have *three* options as follows:

$\boldsymbol{d}_1 = [1\ \ 1]^t,\ \boldsymbol{P}_1 = [1\ \ -1], \quad \boldsymbol{d}_2 = [0\ \ 1]^t,\ \boldsymbol{P}_2 = [1\ \ 0], \quad \boldsymbol{d}_3 = [1\ \ 0]^t,\ \boldsymbol{P}_3 = [0\ \ -1].$

For all $\boldsymbol{d}$ choices, there is a maximum of (m) PE's and latency of $(3m-2n)$ cycles.

Design #5, with projection direction $d_1 = [1\ \ 1]^t$

Equivalently, Fig. 4.6 depicts the $\overline{\text{DAG}}$ corresponding to $\boldsymbol{P}_1$ and PE's details, when $\boldsymbol{s}=[1\ \ 0]$ and $\boldsymbol{d}=[1\ \ 1]^t$. However according to the DAG in Fig. 4.5, Fig. 4.14 presents PE's activity.

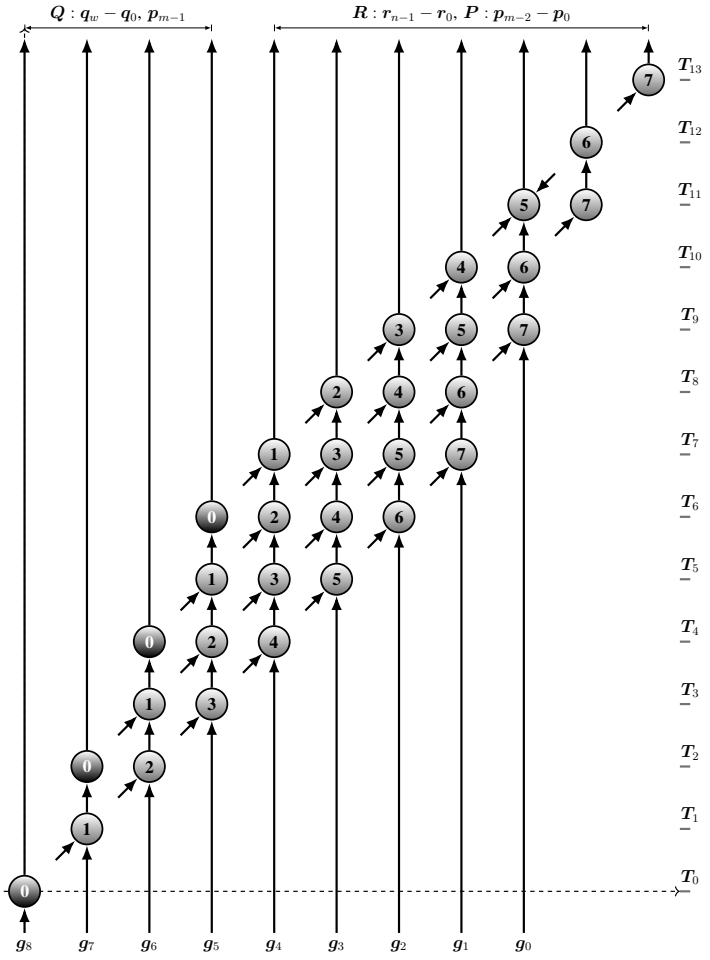


Figure 4.14: Concurrent Divider/MAC, **PE** activity, $m=8$, $n=5$, $\boldsymbol{s}=[1\ \ 1]$, $\boldsymbol{d}=[1\ \ 1]^t$.

4.2.1.6 Nonlinear Scheduling (Design #6)

Nonlinear scheduling can reduce the latency of our design to $(m-n+1)/L$ cycles.The DAG when level of aggregation $L=2$ is illustrated in Fig. 4.15, merging PE(i) and PE($i+1$).

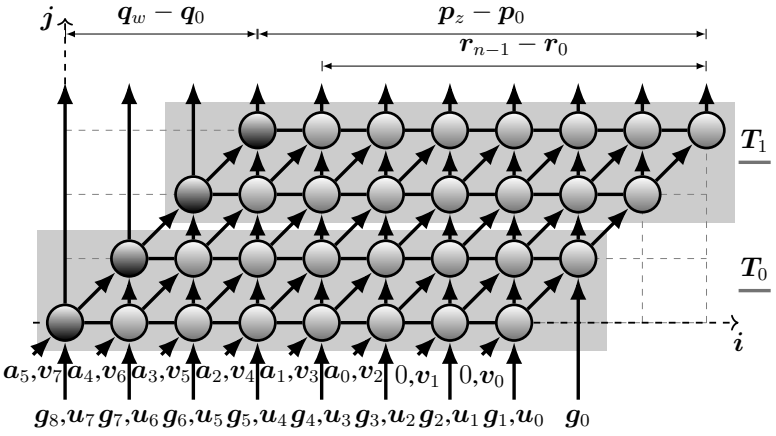


Figure 4.15: Non-Linear Scheduling $\boldsymbol{s} = [0\ \ 1]$ and $L = 2$

4.2.1.7 Nonlinear Projection (Design #7)

Nonlinear projection can reduce the number of PE's to $(n+1)/L$. Figure 4.16 depicts PE activity with $L = 2$, in which PE$_k$ and PE$_{k+1}$ are merged.

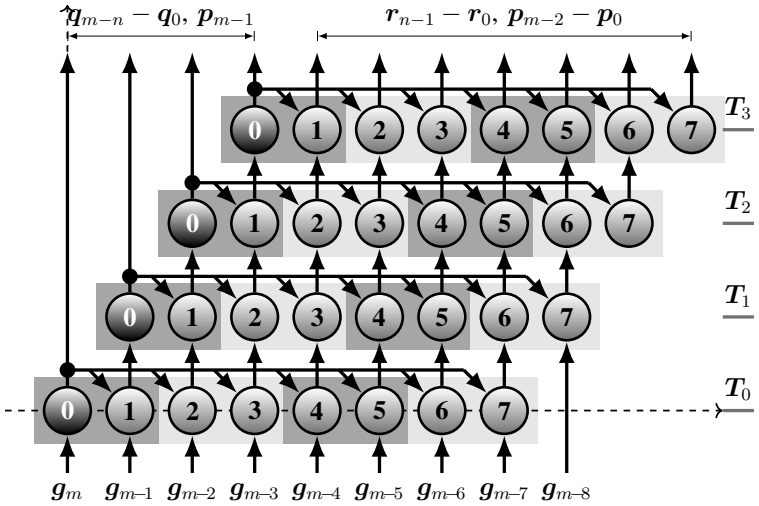


Figure 4.16: Non-Linear Projection $\boldsymbol{d} = [1\ \ 1]$ and $L = 2$

4.2.1.8 Scalable EEA-based Inversion when $s = [0\ \ 1]$ and $d_1 = [1\ \ 1]^t$ (Design #8)

A scalable design is defined as digit-serial implementation, where the word size is varied. Another function for *Nonlinear scheduling* is to downsize the design by employing a level of aggregation $L<1$, which magnifies the latency to $(m-n+1)\times\frac{1}{L}$. This technique can be applied to target resource-constrained embedded applications, which require small area and low power consumption [30]. Figure 4.17 shows the $\overline{\text{DAG}}$ and PE activity when $L=0.5$.

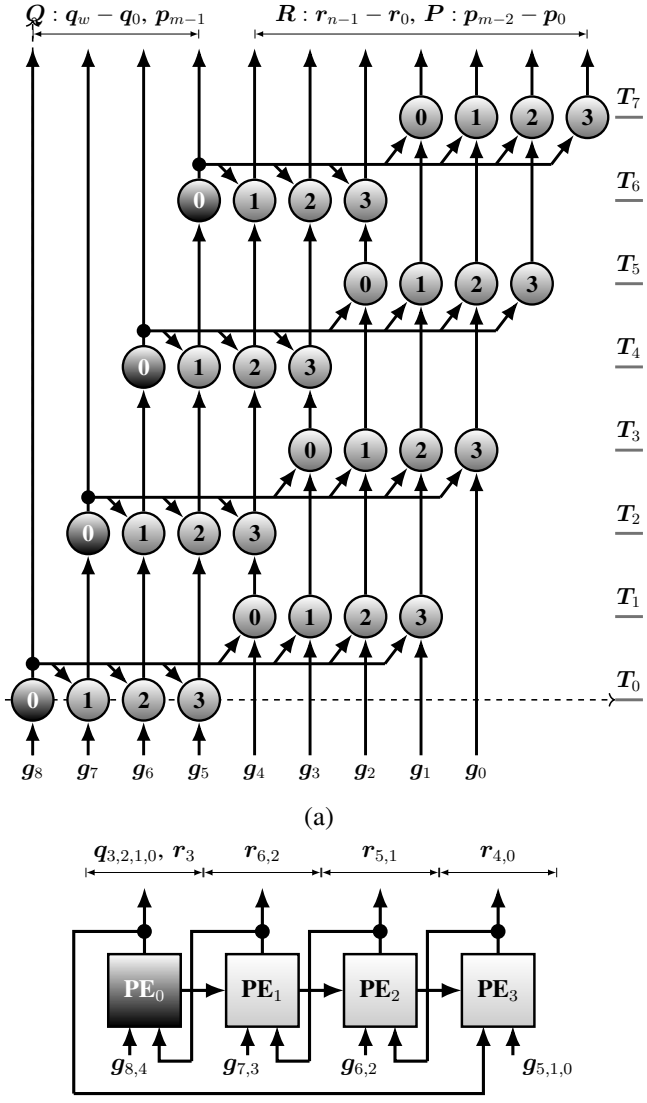


Figure 4.17: Scalable Divider/MAC $s=[1\ \ 0]$, $d=[1\ \ 1]^t$: (a) $\overline{\text{DAG}}$, (b) **PE** activity.

4.2.1.9 Complexity Analysis

The AT-complexity of the design can be affected by scheduling and projection selections and techniques. For example selecting $\boldsymbol{s}_1 = [0\ \ 1]$ as a scheduling vector, our inverter is faster than design with $\boldsymbol{s}_2 = [1\ \ 0]$ or $\boldsymbol{s}_3 = [1\ \ 1]$. Meanwhile, design with projection direction, $\boldsymbol{d}_1 = [1\ \ 1]$, offers smaller area and CPD comparing to $\boldsymbol{d}_2 = [-1\ 1]$ or $\boldsymbol{d}_3 = [0\ \ 1]$.

AT-complexity of the following *eight* designs is estimated in TABLE 4.1, denoting the delay of $(\times, /)$, $(+, -)$, and 2x1 Multiplexer by $\mathrm{T}_{\times,/}$, $\mathrm{T}_{+,-}$ and T_M, respectively.

Design #1: $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [1\ \ 1]^t$, Fig. 4.3, 4.6,
Design #2: $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [-1\ \ 1]^t$, Fig. 4.11,
Design #3: $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [0\ \ 1]^t$, Fig. 4.12,
Design #4: $\boldsymbol{s} = [1\ \ 0]$, $\boldsymbol{d} = [1\ \ 1]^t$, Fig. 4.4,
Design #5: $\boldsymbol{s} = [1\ \ 1]$, $\boldsymbol{d} = [1\ \ 1]^t$, Fig. 4.5,
Design #6: *NLS*, $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [1\ \ 1]^t$, $L = 2$, Fig. 4.15,
Design #7: *NLP*, $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [1\ \ 1]^t$, $L = 2$ Fig. 4.16,
Design #8: *NLS*, $\boldsymbol{s} = [0\ \ 1]$, $\boldsymbol{d} = [1\ \ 1]^t$, $L = 0.5$ Fig. 4.17.

Table 4.1: AT requirements of our different inverters

Design	#PEs	$(+,-)$	$(\times,/)$	MUX	Flip-Flop	Latency	CPD
#1	m	$3m-2$	$2m$	$(4m-2)b^1$	$(5m-1)b^1$	$2m-2$	$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$
#2,#3	m	$3m$	$3m$	$(5m)b^1$	$(6m)b^1$	$2m-2$	$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$+T_M
#4	m	$3m-2$	$2m$	$(4m-2)b^1$	$(5m-1)b^1$	$(1+a)m-1^2$	$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$
#5	m	$3m-2$	$2m$	$(4m-2)b^1$	$(5m-1)b^1$	$(2+a)m-2^2$	$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$
#6	m	$6m-4$	$4m$	$(4m-2)b^1$	$(6m-1)b^1$	$m-1$	2$\mathrm{T}_{\times,/}$+4$\mathrm{T}_{+,-}$
#7	$m/2$	$3m-2$	$2m$	$(4m-2)b^1$	$(5m-1)b^1$	$2m-2$	$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$
#8	$m/2$	$3m/2$	$m+1$	$(2m+1)b^1$	$(5m/2+1)b^1$	$4m-4$	$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$

[1] b = number of bits representing each $GF(p)$ coefficient. [2] a = number of **EEA** iterations.

TABLE 4.1 reveals that Design #6 has the largest area, the least latency of $(m-1)$ cycles and the highest CPD of (2$\mathrm{T}_{\times,/}$+2$\mathrm{T}_{+,-}$). On the contrary, Design #8 resides the least area and requires a high latency of $(4m-4)$ cycles, with the lowest CPD. Although Design #7 has the same CPD and number of PE's as Design #8, it has a relatively low latency of $(2m-2)$ cycles, it occupies larger area. It also shares the same area size and CPD with Designs #1, #4 and #5. However, Design #4 scores a higher latency of $((1+a)m-1)$ cycles, whilst the highest latency of $((2+a)m-2)$ is scored by Design #5. Since Designs #2 and #3 require additional circuitry to *rotate* the stored variables and update the broadcast variable ($\boldsymbol{q}_j$) each cycle, they occupy large areas, with similar latency to Designs #1 and #7 and a higher CPD of (T_M+$\mathrm{T}_{\times,/}$+$\mathrm{T}_{+,-}$).

4.2.2 Concurrent Polynomial Division and MAC over over GF(2^m)

EEA-based inversion over binary fields has been studied massively. Consequently, many architectures have been proposed throughout the literature including efficient work reported in [8, 27–29, 101, 117]. In [20], a novel implementation of systolic EEA-based inverter over binary fields was proposed. Considering Design #1 from previous subsection, where $\boldsymbol{s} = [0\ \ 1]$ and $\boldsymbol{d} = [1\ \ 1]^t$, the PE's in Figures 4.7, 4.8 and 4.8 can be simplified for GF(2^m) inversion as depicted in Fig. 4.18 along with CSG.

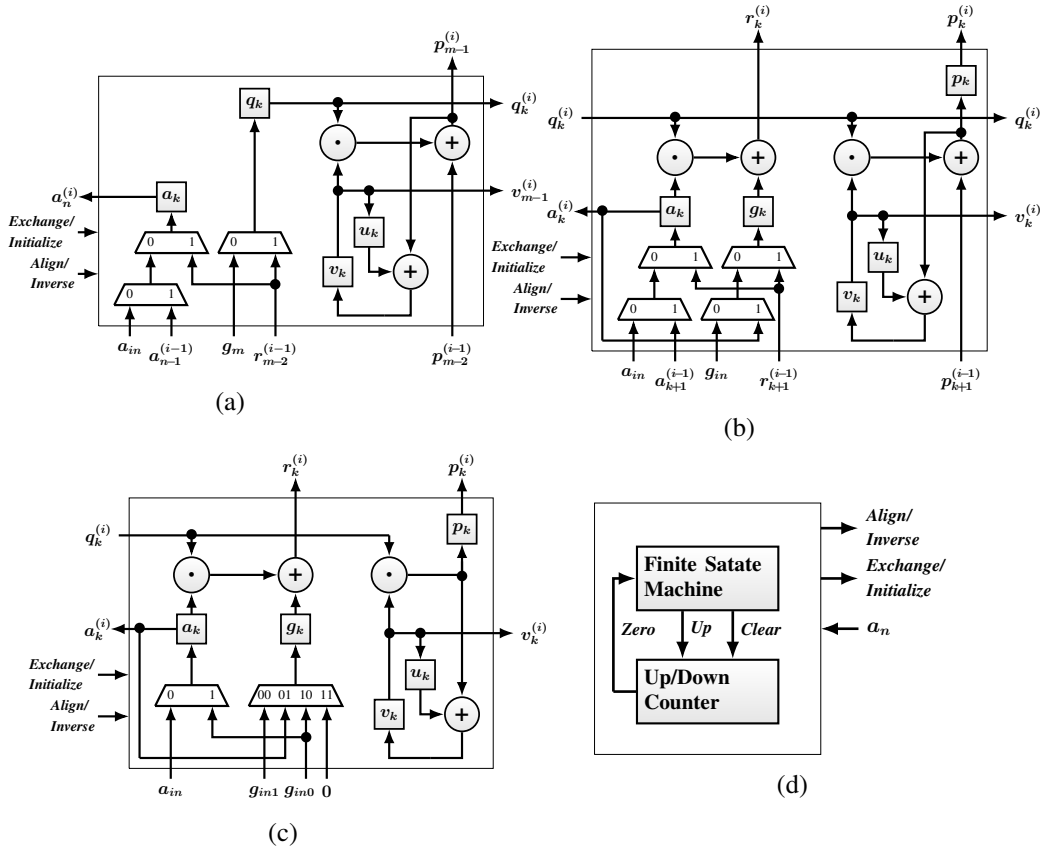


Figure 4.18: **EEA PE**'s, $\boldsymbol{s}=[1\ \ 0]^t$, $\boldsymbol{d}=[1\ \ 1]$: (a) PE_0, (b) $PE_{k>0}$, (c) PE_{m-1}, (d) CSG.

AT-complexity of our *eight* architectures can be estimated binary fields as presented in TABLE 4.2. It important to note that PE_0 of our inverter over binary fields does not include modular division ($/$), since $\boldsymbol{a_n}$ is always 1. The delay of 2-input AND gate, 2-input XOR gate, and 2-input Multiplexer are denoted by T_A, T_X, and T_{MUX} respectively. Our CSG is estimated to cost (1) NOT $+(3\log(m)-2)$ NANDs $+(\log(m)+2)$ Flip-Flops.

Table 4.2: AT requirements of our different inverters

Design	#PEs	XOR	AND	MUX	Flip-Flop	Latency	CPD
#1	m	$3m-2$	$2m-1$	$(4m-2)$	$(5m-1)$	$2m-2$	$T_{\times,/}$+2$T_{+,-}$
#2,#3	m	$3m$	$3m$	$(5m)$	$(6m)$	$2m-2$	$T_{\times,/}$+2$T_{+,-}$+T_M
#4	m	$3m-2$	$2m-1$	$(4m-2)$	$(5m-1)$	$(1+a^1)m-1$	$T_{\times,/}$+2$T_{+,-}$
#5	m	$3m-2$	$2m-1$	$(4m-2)$	$(5m-1)$	$(2+a^1)m-2$	$T_{\times,/}$+2$T_{+,-}$
#6	m	$6m-4$	$4m-2$	$(4m-2)$	$(6m-1)$	$m-1$	2$T_{\times,/}$+4$T_{+,-}$
#7	$m/2$	$3m-2$	$2m-1$	$(4m-2)$	$(5m-1)$	$2m-2$	$T_{\times,/}$+2$T_{+,-}$
#8	$m/2$	$3m/2$	m	$(2m+1)$	$(5m/2+1)$	$4m-4$	$T_{\times,/}$+2$T_{+,-}$

[1] a = number of **EEA** iterations.

However, the reported efficient GF(2^m) EEA-based inverters are considered for area-time complexity comparison. The total gate count is estimated in terms of 2-input NAND gate based on NanGate (15nm, 0.8V) Open Cell Library, which is founded on the basis of FreePDK15 design process kit. Hence, TABLE 4.3 declares the area and delay values for standard gates in respect of 2-input NAND gate [1].

Table 4.3: Gates to NAND: Gate Count and Delay

Gate	Area (2-input NAND)	Delay (ps)
INV	0.6	5.8
NAND	1	10
AND/OR	1.2	11.3
XOR	2.5	12.7
MUX	2.5	12.4
Flip-Flop	4.3	32.8

TABLE 4.4 presents a comparison between the reported inverters and our inverter with regard to latency, and area (total gate count including flip-flops, which are modeled based on TABLE 4.3).

Table 4.4: Complexity Comparison between different inverters over GF(2^m)

Design	Latency	Flip-Flops	Total Gate Count (Area)
Guo et al. [27]	$8m-1$	$> 146.2m^2$	$> 212m^2 + 1.2m$
Yan et al. [29]	$5m-2$	$86m^2$	$115.8m^2 + 0.6$
Daneshbeh and Hasan [117]	$5m-4$	$129m$	$198.6m$
Wu et al. [28]	$5m-2$	$73.1m + 12.9H$	$99.2m + 22.1H + 0.1$
Antao [101]	$2m$	$34.4m + 4.3$	$94.4m + 17.4$
Ibrahim et al. [8]	$2m-1$	$21.5m$	$43.9m - 6.4$
This work	$2m-2$	$21.5m + 4.3H + 4.3$	$41.4m + 7.3H - 9.1$

It can be observed from TABLE 4.4 that the proposed design outperforms the compared systolic designs in terms of total gate count (Area) and latency.

Meanwhile, TABLE 4.5 compares the inverters concerning their CPD's and describes the calculations of clock period, which is defined as the CPD, considering setup delay (T_{setup}) and propagation delay ($T_{Clk \to Q}$) of the D-FF.

Table 4.5: Clock Period calculations of different inverters over GF(2^m) (ps)

Design	**CPD**	**AND/OR**	**XOR**	**MUX**	**Clock Period**
Guo et al. [27]	$T_A + T_X + 2T_{MUX}$	11.3	12.7	24.8	48.8
Yan et al. [29]	$T_A + T_{MUX}$	11.3	0	12.4	23.7
Daneshbeh and Hasan [117]	$2T_A + T_X + T_{MUX}$	22.6	12.7	12.4	47.7
Wu et al. [28]	$T_A + 2T_X + T_{MUX}$	11.3	25.4	12.4	49.1
Antao [101]	$T_X + 3T_{MUX}$	0	12.7	37.2	49.9
Ibrahim et al. [8]	$T_A + 2T_X$	11.3	25.4	0	36.7
This work	$T_A + 2T_X$	11.3	25.4	0	36.7

TABLE 4.5 shows that the CPD of our inverter is lower than the CPD of reported work in [27, 28, 101, 117], and similar to the one in [8]. The results obtained in TABLE 4.4 and TABLE 4.5 can be combined and quantified to measure AT-complexity of the compared designs. TABLE 4.6 estimates latency, clock period, total computation time (Time), total gate count (Area), and AT-complexity, i.e, area (A) $\times$ time (T), when $m = 233$.

Table 4.6: Estimated area and delay results of different systolic inverters over GF(2^{233})

Design	**Latency** (# cycles)	**Clock Period** (ps)	**Total Computation Time (T)** (ns)	**Area (A)** (Kgates)	**AT**
Guo et al. [27]	> 1863	> 48.8	> 90.91	> 11509.55	> 1046384
Yan et al. [29]	1163	23.7	27.56	6286.67	173280
Daneshbeh and Hasan [117]	1161	47.7	55.38	46.27	2563
Wu et al. [28]	1163	49.1	57.1	18.25	1330
Antao [101]	466	49.9	23.25	22.01	512
Ibrahim et al. [8]	465	36.7	17.07	10.22	174
This work	464	36.7	17.03	9.70	165

TABLE 4.6, illustrates that our EEA-based inverter outperforms the reported inverters in terms of area (9.7 Kgates) and speed ($17.03ns$) producing the lowest AT-complexity among the reported designs (165).

Chapter 5

Conclusions

Finite field arithmetic is the building block of public-key cryptographic schemes, which are used widely for security and privacy of many hardware and software systems. Inversion is the most expensive finite field arithmetic operation, and EEA is reported to be the most efficient inversion algorithm in terms of performance and power consumption. The main objective of this work was to explore the design space of EEA-based inversion over prime extension fields, performing concurrent polynomial operations. In the process, the design space of polynomial division and multiplication over prime extension fields was explored, utilizing their dependencies in order to allow for concurrency. A new reformulation of the traditional EEA algorithm was proposed, allowing for concurrency and resource sharing. Considering the iterative equations we derived, and using a systematic methodology, the DG, DAG and Projected DAG ($\overline{\text{DAG}}$) were developed to build our dividers, multipliers and inverters. Modular arithmetic over GF(p) of small primes was introduced, observing that division can be treated evenly with multiplication in GF(2) and GF(3). As p getting larger, symmetrical figures become more observable in all modular operations, including multiplication and division operations. Therefore, only a quarter of the elements is required to be stored in the Lookup-Table, while the rest can be generated easily. Whereas, polynomial division and multiplication GF(p^m) were discussed intensively, deriving their iterative equations, which are suitable for systolic implementation. Our multiplication was proven to be a non-modular process, which eliminates the reduction requirement and simplifies the design. Consequently, several designs were proposed for each individual process. Subsequently, a concurrent divider/multiplier-accumulator was developed. With this concurrent polynomial divider/multiplier-accumulator, one iteration of the EEA algorithm could be performed without the necessity for aligning variables within the division or multiplication processes. This concurrent systolic architecture was utilized to build the EEA-based inverter.

It has been presented that scheduling and projection parameters affect the area and time complexity of the systolic designs. Furthermore, nonlinear scheduling and projection strategists has been presented to provide more flexibility in the trade-off between performance and area. It is important to note that for practical inverter, polynomial division and multiplication circuits should contain the maximum number of PE's in order to process all possible data sizes, i.e., $n_{max} + 1 = m$. The processing elements of our systolic inverters were created according to the derived three iterative equations, and enhanced to accommodate data movement throughout our EEA algorithm. While, the shift-inverse control mechanism was demonstrated subsequently. As a result of our design methodology and approach, novel, fast, and compact inverters over binary and ternary fields were proposed while the binary inverter was implemented on FPGA. Meanwhile, accurate models for the complexity analysis of the proposed dividers, multipliers and inverters were developed. The proposed binary inverter outperforms the reported inverters in terms of area and speed, showing the lowest AT-complexity. Furthermore, the ternary inverter presented significant improvement in terms of CPD, Time, Area, and AT-complexity, with the lowest Latency. Finally, the contributions of the research work are summarized as follows:

1. An exploratory study of finite field based processors such as, ECC acceletors,
2. Derivation of iterative equations for polynomial division and multiplication over extension fields that are suitable for array architectures.
3. A new reformulation of the traditional EEA, using concurrent polynomial processes.
4. Developing systolic architectures for polynomial division and multiplication, building concurrent EEA hardware architectures by applying a former methodology.
5. Presenting different architecture types, including bit-parallel and digit-serial, based on linear and nonlinear scheduling ad projection techniques.
6. Developing accurate models for complexity analysis of our systolic architectures, providing a complexity comparison with a selection of efficient field inverters.

Future Work

The proposed EEA-based inverters are considered for optimized FPGA implementation, examining data size limits of different FPGAs including Input/Output constraints with regard to executing our algorithm. In addition, two-dimensional systolic arrays are suggested to be developed using the same methodology, comparing the results with our one-dimensional array architectures in this work. Finally, fully-generalized polynomial division is targeted to be revisited, exploring its design space with an "inverse only" control mechanism.

Appendix A

Hardware Implementation of a Systolic EEA-based Inverter over Binary Fields

The objective of our FPGA implementation is to confirm the workability of our architectures in a hardware environment. However, it is considered for future work, where different implementations are targeted and primitive components of the FPGA are utilized.

Having our PE's enhanced to accommodate data exchange and alignment each iteration, the concurrent EEA inverter requires a basic unit in order to adhere to the control mechanism stated earlier. The CSG circuit consists of a Finite State Machine (FSM) and an up-down counter, prompting multiplexers and registers with two main signals, i.e., *Align/Inverse* and *Exchange/Initialize*. However for simplicity, we generate the selection and enable signals directly from the controller to the PE's in our implementation. The $\overline{\text{DAG}}$ and the enhanced PE's in Chapter 4 are considered as the data-path of our EEA-based inverter. The simple FSM controller, which performs the EEA completely with the help of a counter, is shown in Fig. A.1.

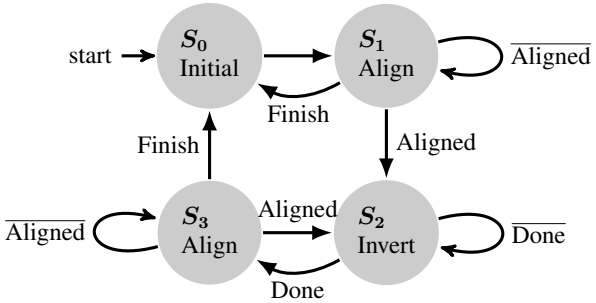


Figure A.1: FSM Controller for our GF(2^m) **EEA**-based inverter

At state S_0, the system inputs are inserted, while initializing all registers.

At state S_1, the second cycle of division is set and variables are aligned, while counting-up number of cycles.

At state S_2, concurrent inversion is performed, and the count-down assures the required number of cycles.

At state S_3, variables are exchanged and aligned, while counting-up and testing if the algorithm is completed.

TABLE A.1 exhibits the state table of our state machine, where *one* input is inserted, i.e., $\boldsymbol{a_n}$ and another input is internal, i.e. ***Zero***. Whereas, *eleven* outputs are connected directly to the data-path and *two* are internally connected to the counter.

Table A.1: 4-State FSM controller: Table of States and Control Signals

State	**Next State**	**Input**		**Output**											
		a_n	CLZ[1]	Sel_{g1}	Sel_{g2}	Sel_q	Sel_{a1}	Sel_{a2}	En_q	En_g	En_a	En_u	En_v	En_p	**up**[1]	**clear**[1]
S0	**S1**	x	x	0	0	0	0	0	1	1	1	0	1	0	0	1
S1	**S1**	0	x	0	1	0	0	1	1	0	1	0	0	0	1	0
	S2	1	x	1	0	1	0	1	1	1	0	0	0	1	0	0
S2	**S2**	x	0	1	1	1	1	1	1	1	0	0	0	1	0	0
	S3	x	1	0	1	1	1	1	1	1	1	1	1	1	1	0
S3	**S2**	1	x	1	1	1	0	1	1	1	0	0	0	1	0	0
	S3	0	x	0	1	0	0	1	1	0	1	0	0	0	1	0

[1] Internal signals between the state machine and the counter

Finally, FIg. A.2 depicts samples of simulation results, where two polynomials are the multiplicative inverse of each other.

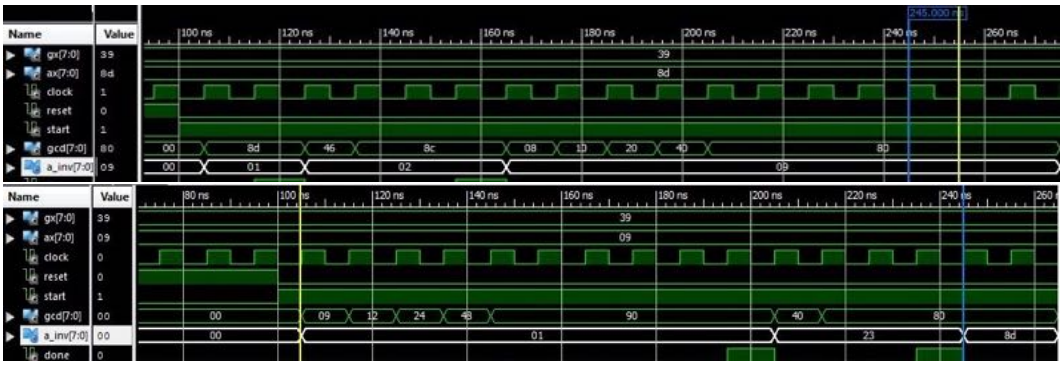


Figure A.2: VHDL Simulation $m = 8$, $\boldsymbol{A} = 0x8d$ and $A^{-1} = 0x09$, $\boldsymbol{G} = 0x139$.

Acronyms

AES Advanced Encryption Standard.
AIA Almost Inverse Algorithm.
AT Area-Time.
BCH Bose–Chaudhuri–Hocquenghem.
CED Concurrent Error Detection.
CPD Critical Path Delay.
CRC Cyclic Redundancy Check.
CSG Control Signals Generator.
DAG Directed Acyclic Graph.
DG Dependency Graph.
ECC Elliptic Curve Cryptography.
EEA Extended Euclidean Algorithm.
ESA Extended Stein's Algorithm.
FLT Fermat's Little Theorem.
FPGA Field Programmable Gate Array.
FSM Finite State Machine.
GCD Greatest Common Divisor.
GNB Gaussian Normal Basis.
IBE Identity Based Encryption.
IOT Internet Of Things.
ITA Itoh-Tsujii Algorithm.
LSB Least Significant Bit.
MAC Multiplier–accumulator.
MSB Most Significant Bit.
ONB Optimal Normal Basis.
PE Processing Element.
PQC Post-Quantum Cryptography.
SIDH Supersingular Isogeny Diffie-Hellman.
VLSI Very Large Scale Integrated Circuit.

Glossary

Throughput The rate at which data can be processed, which measures productivity of the system. It can viewed as the time difference between two consecutive outputs. It also refers to the rate at which input data can be applied to system.

Irreducible polynomial A non-constant polynomial, which cannot be factored into the product of non-constant polynomials over the same field.

Trinomial An irreducible polynomial with only three non-zero coefficients

Pentanomial An irreducible polynomial with only five non-zero coefficients

Equally-spaced In irreducible equally-spaced polynomials, similar space separates each two consecutive non-zero coefficients.

Redundant assignment One element in GF(3) has two different representations, but the other two elements are uniquely assigned.

Latency The time required to produce output after input is applied, which can be expressed as a number of clock cycles. It measures the delay response of a design. Higher the latency value, slower is the system.

Critical path The longest path in the circuit, which limits the clock speed.

Parallelism The maximum number of nodes that can be processed in parallel.

Concurrency The decomposability property of an algorithm into order-independent components, allowing for parallel execution and improving the overall speed of multi-core systems.

Bibliography

[1] Atef Ibrahim, Hamed Elsimary, and Fayez Gebali. New systolic array architecture for finite field division. *IEICE Electronics Express*, 15(11):20180255–20180255, 2018.

[2] I. H. Hazmi and F. Gebali. Systolic design space exploration of polynomial division over GF(2^m). In *IEEE 30th CCECE*, 2017.

[3] J-P Deschamps et al. *Guide to FPGA implementation of arithmetic functions*, volume 149. Springer Science & Business Media, 2012.

[4] I. Hazmi, F. Zhou, F. Gebali, and T. Al-Somani. Review of elliptic curve processor architectures. In *IEEE PACRIM'15*, pages 192–200, 2015.

[5] F. Rodrıguez-Henrıquez et al. Cryptographic algorithms on reconfigurable hardware, 2006.

[6] H. Modares. *A scalar multiplication in elliptic curve cryptography with binary polynomial operations in Galois Field*. PhD thesis, University of Malaya, 2009.

[7] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography*. Springer Science & B.M., 2004.

[8] A. Ibrahim, T. Al-Somani, and F. Gebali. New systolic array architecture for finite field inversion. *Canadian Journal of Electrical and Computer Engineering*, 40:23–30, 2017.

[9] G. De Dormale and J. Quisquater. Novel iterative digit-serial modular division over GF(2^m). *CRASH*, 2005.

[10] Abdulah Abdulah Zadeh. Division and inversion over finite fields. In *Cryptography and Security in Computing*, pages 117–130. InTech, 2012.

[11] M. Masoumi and H. Mahdizadeh. Efficient hardware implementation of an elliptic curve cryptographic processor over GF(2^163). *IJCEACIE*, 6(5):190 – 197, 2012. ISSN 1307-6892.

[12] S. Gashkov et al. Complexity of computation in finite fields. *Journal of Mathematical Sciences*, 191(5):661–685, 2013.

[13] B. Koziel, R. Azarderakhsh, M. M. Kermani, and D. Jao. Post-quantum cryptography on FPGA based on isogenies on elliptic curves. *IEEE Transactions on Circuits and Systems I*, 64(1):86–99, 2017.

[14] Kimmo Järvinen et al. *Studies on high-speed hardware implementation of cryptographic algorithms*. Teknillinen korkeakoulu, 2008.

[15] Chang Shu. *Hardware architectures of elliptic curve based cryptosystems over binary fields*. PhD thesis, George Mason University, 2007.

[16] E. Wenger and M. Hutter. Exploring the design space of prime field vs. binary field ECC-hardware implementations. In *NordSec'11*, pages 256–271. Springer, 2011.

[17] Ilker Yavuz, Siddika Berna Örs Yalçin, and Cetin Kaya Koç. FPGA implementation of an elliptic curve cryptosystem over GF(3^m). In *ReConFig'08*, pages 397–402. IEEE, 2008.

[18] Fayez Gebali. *Algorithms and parallel computing*. John Wiley & Sons, 2011.

[19] Ibrahim Hazmi, Fayez Gebali, and Atef Ibrahim. Systolic design space exploration of polynomial division over GF (3^m). In *FTC Vancouver, Accepted on May 1st*, 2018.

[20] Ibrahim Hazmi, Fayez Gebali, and Atef Ibrahim. Systolic design space exploration of EEA-based inversion over binary fields. *IEEE Transactions on Consumer Electronics, Received in June 14th*, 2018.

[21] Ibrahim Hazmi, Fayez Gebali, and Atef Ibrahim. Systolic design space exploration of EEA-based inversion over ternary fields. *IET Information Security, Received in August 8th*, 2018.

[22] I. H. Hazmi. Project: EEA-based polynomial inversion over GF(2^m): FPGA design and implementation. Technical report, ECE, University of Victoria, 2015. URL `https://www.researchgate.net/profile/Ibrahim_Hazmi/contributions`.

[23] I. H. Hazmi. Project: Design and implementation of the euclidean algorithm for computing the GCD using Spartan6 FPGA. Technical report, ECE, University of Victoria, 2014. URL `https://www.researchgate.net/profile/Ibrahim_Hazmi/contributions`.

[24] Fan Junfeng. *Efficient arithmetic for embedded cryptography and cryptanalysis*. PhD thesis, Katholieke Universiteit Leuven, 2012.

[25] R. Brent and HT Kung. Systolic VLSI arrays for polynomial GCD-computation. *IEEE Transactions on Computers*, 100(8):731–736, 1984.

[26] H. Brunner et al. On computing multiplicative inverses in GF(2^m). *IEEE Transactions on Computers*, 42(8):1010–1015, 1993.

[27] J-H. Guo et al. Systolic array implementation of euclid's algorithm for inversion and division in GF(2^m). *IEEE Transactions on Computers*, 47(10):1161–1167, 1998.

[28] C-H. Wu, C-M. Wu, M-D. Shieh, and Y-T. Hwang. High-speed, low-complexity systolic designs of novel iterative division algorithms in GF(2^m). *IEEE Transactions on Computers*, 53(3):375–380, 2004.

[29] Z. Yan, D. Sarwate, and Z. Liu. High-speed systolic architectures for finite field inversion. *Integration, the VLSI journal*, 38(3):383–398, 2005.

[30] A. Ibrahim and F. Gebali. Scalable and unified digit-serial processor array architecture for multiplication & inversion over GF(2^m). *IEEE Transactions on Circuits and Systems I*, 64:2894–2906, 2017.

[31] Omran Ahmadi, Darrel Hankerson, and Alfred Menezes. Software implementation of arithmetic in GF(3^m). In *WAIFI*, pages 85–102. Springer, 2007.

[32] T. Kerins, E. Popovici, and W. Marnane. Algorithms and architectures for use in FPGA implementations of identity based encryption schemes. In *FPL*, pages 74–83. Springer, 2004.

[33] T Kerins, WP Marnane, and EM Popovici. Hardware architectures for arithmetic in GF(p^m) for use in public key cryptography. *Preprint*, 2004.

[34] T. Kerins et al. Efficient hardware for the tate pairing calculation in characteristic three. In *CHES*, pages 412–426. Springer, 2005.

[35] Robert Ronan, O Colm, Colin Murphy, Tim Kerins, Paulo SLM Barretto, et al. A reconfigurable processor for the cryptographic nt pairing in characteristic 3. In *4th ITNG'07*, pages 11–16. IEEE, 2007.

[36] J-P Deschamps et al. Hardware implementation of finite-field division. *Acta Applicandae Mathematica*, 93(1):119–147, 2006. URL `https://doi.org/10.1007/s10440-006-9049-y`.

[37] C-H. Wu, C-M. Wu, M-D. Shieh, and Y-T. Hwang. Novel algorithms and VLSI design for division over GF(2^m). *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 85(5):1129–1139, 2002.

[38] Guerric Meurice De Dormale, Philippe Bulens, and Jean-Jacques Quisquater. Efficient modular division implementation: ECC over GF(p) affine coordinates application. In *FLP*, pages 231–240. Springer, 2004.

[39] Naofumi Takagi. A VLSI algorithm for modular division based on the binary GCD algorithm. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 81(5):724–728, 1998.

[40] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery. Trading inversions for multiplications in elliptic curve cryptography. *Designs, codes and cryptography*, 39(2): 189–206, 2006.

[41] Nigel P Smart and EJ Westwood. Point multiplication on ordinary elliptic curves over fields of characteristic three. *Applicable Algebra in Engineering, Communication and Computing*, 13(6):485–497, 2003.

[42] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. Technical report, Cryptology ePrint Archive, Report 2016/1017, 2016. http://eprint. iacr. org/2016/1017, 2016.

[43] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th IEEE SFCS '94*, pages 124–134. IEEE, 1994.

[44] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. A post-quantum digital signature scheme based on supersingular isogenies. In *Financial Crypto*, volume 2017, 2017.

[45] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani. NEON-SIDH: efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In *15th CANS'16*, pages 88–103. Springer, 2016.

[46] L Chen, S Jordan, Y Liu, D Moody, R Peralta, R Perlner, and D Smith-Tone. Report on post-quantum cryptography (NISTIR 8105), 2016.

[47] Vladimir Soukharev. *Post-Quantum Elliptic Curve Cryptography*. PhD thesis, University of Waterloo, 2016.

[48] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. In *Annual Cryptology Conference*, pages 572–601. Springer, 2016.

[49] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange-a new hope. *IACR Cryptology ePrint Archive*, 2015:1092, 2015.

[50] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Adv. in Math. of Comm.*, 4(2): 215–235, 2010.

[51] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8 (3):209–247, 2014.

[52] Jeremy Wohlwend. Elliptic curve cryptography: Pre and post quantum. Technical report, MIT, 2016.

[53] C. Wilkerson, A. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S. Lu. Reducing cache power with low-cost, multi-bit error-correcting codes. *SIGARCH Comput. Archit. News*, 38(3):83–93, 2010.

[54] Yongjin Jeong and Wayne Burleson. VLSI array synthesis for polynomial GCD computation. In *IEEE ASAP '93*, pages 536–547. IEEE, 1993.

[55] Danilo Silva. *Error control for network coding*. PhD thesis, University of Toronto, 2009.

[56] Gilles Millérioux and Jamal Daafouz. On persistent excitations for the identification of switched linear dynamical systems over finite fields. *Automatica*, 50(12):3246–3252, 2014.

[57] Harald Niederreiter. A survey of some applications of finite fields. *Designs, Codes and Cryptography*, 78(1):129–139, 2016.

[58] Phillip Kaye and Christof Zalka. Optimized quantum implementation of elliptic curve arithmetic over binary fields. *arXiv preprint quant-ph/0407095*, 2004.

[59] John Proos and Christof Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *arXiv preprint quant-ph/0301141*, 2003.

[60] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge university press, 1994.

[61] Anne Canteaut. Error-correcting codes and applications to cryptography: Finite fields, 2014-2018. URL `https://www.paris.inria.fr/secret/Anne.Canteaut/MPRI/ff.pdf`.

[62] W. Stallings and M. Tahiliani. *Cryptography and network security: principles and practice*. Pearson Education, Upper Saddle River, NJ, USA, 6th edition, 2013. ISBN 0133354695, 9780133354690.

[63] M.Morales-Sandoval. *A reconfigurable and interoperable hardware architecture for elliptic curve cryptography*. PhD thesis, Tesis de Doctorado, Instituto Nacional de Astrofísíca, Óptica y Electrónica, México, 2008.

[64] E. Wenger and M. Hutter. Exploring the design space of prime field vs. binary field ECC-hardware implementations. In *Information Security Technology for Applications*, pages 256–271. Springer, 2012.

[65] M.A.Fayed. *A security coprocessor for next generation IP telephony: architecture, abstraction, and strategies*. University of Victoria, 2007.

[66] Çetin Kaya Koç. *Cryptographic Engineering*. Springer, 2009.

[67] V. Tujillo-Olaya and J. Velasco-Medina. Hardware architectures for elliptic curve cryptoprocessors using polynomial and gaussian normal basis over GF(2^{233}). In *Transactions on computational science XI*, pages 79–103. Springer, 2010.

[68] M. Morales-Sandoval et al. Bit-serial and digit-serial GF(2^m) montgomery multipliers using linear feedback shift registers. *IET Computers & Digital Techniques*, 5(2): 86–94, 2011.

[69] Avinash Kak. Lecture 6: Finite fields (part 3): Polynomial arithmetic, 2 2016.

[70] R. Afreen and S.C. Mehrotra. A review on elliptic curve cryptography for embedded systems. *arXiv preprint: 1107.3631*, 2011.

[71] N.Boston and M.Darnall. Elliptic and hyperelliptic curve cryptography. In *Cryptographic Engineering*, pages 171–189. Springer, 2009.

[72] M. Cenk, C. Koç, and F. Ozbudak. Polynomial multiplication over finite fields using field extensions and interpolation. In *19th IEEE ARITH'09*, pages 84–91. IEEE, 2009.

[73] C. Paar and J. Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[74] Sedat Akleylek, Ferruh Özbudak, and C Özel. On the arithmetic operations over finite fields of characteristic three with low complexity. *Journal of Computational and Applied Mathematics*, 259:546–554, 2014.

[75] Y. Kawahara, K. Aoki, and T. Takagi. Faster implementation of ηT pairing over GF (3^m) using minimum number of logical instructions for gf (3)-addition. In *ICPBC*, pages 282–296. Springer, 2008.

[76] Steven D. Galbraith, K. Harrison, and D. Soldera. Implementing the Tate pairing. In *ANTS*, pages 324–337. Springer, 2002.

[77] K. Harrison, D. Page, and N. P. Smart. Software implementation of finite fields of characteristic three, for use in pairing-based cryptosystems. *LMS Journal of Computation and Mathematics*, 5:181–193, 2002. doi: 10.1112/s1461157000000747.

[78] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[79] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[80] Sun Yuan Kung. *VLSI array processors*, volume 1. Prentice Hall, NJ, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy, 1988.

[81] Hsiang-Tsung Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[82] Zhiyuan Yan. Digit-serial systolic architectures for inversions over GF(2^m). In *IEEE Workshop on Signal Processing Systems Design and Implementation. SIPS'06*, pages 77–82. IEEE, 2006.

[83] Y. Sun and M. S. Kim. A table-based algorithm for pipelined crc calculation. In *IEEE ICC'10*, pages 1–5. IEEE, 2010.

[84] Y-T. Horng and S-W. Wei. Fast inverters and dividers for finite field GF(2^m). In *IEEE APCCAS*, pages 206–211, 1994.

[85] J-H. Guo and C-L. Wang. Hardware-efficient systolic architecture for inversion and division in GF(2^m). In *IEE Proceedings-Computers and Digital Techniques*, volume 145/4, pages 272–278. IET, 1998.

[86] S. H. Zak and K. Hwang. Polynomial division on systolic arrays. *IEEE Transactions on Computers*, 100(6):577–578, 1985.

[87] HT Kung. Use of VLSI in algebraic computation: Some suggestions. In *4th ACM ISSAC'16*, pages 218–222. ACM, 1981.

[88] Selçuk Baktır and Berk Sunar. Finite field polynomial multiplication in the frequency domain with application to elliptic curve cryptography. In *International Symposium on Computer and Information Sciences*, pages 991–1001. Springer, 2006.

[89] Utsav Banerjee, Chiraag Juvekar, Andrew Wright, Anantha P Chandrakasan, et al. An energy-efficient reconfigurable dtls cryptographic engine for end-to-end security in iot applications. In *IEEE ISSCC'18*, pages 42–44. IEEE, 2018.

[90] Larry Hardesty. Energy-efficient encryption for the internet of things, 2018. MIT news.

[91] R. Brent and HT Kung. A systolic algorithm for integer GCD computation. In *IEEE 7th ARITH'85*, pages 118–125. IEEE, 1985.

[92] Jean-Pierre Deschamps. *Hardware implementation of finite-field arithmetic*. McGraw-Hill, Inc., 2009.

[93] K. Araki et al. Fast inverters over finite field based on Euclid's algorithm. *IEICE*, 72 (11):1230–1234, 1989.

[94] R. Doyle et al. An improved systolic extended euclidean algorithm for reed-solomon decoding. In *ICASAP*, pages 448–456. IEEE, 1990.

[95] J-H. Guo and C-L. Wang. Systolic array implementation of euclid's algorithm for inversion and division in GF (2^m). *IEEE*, 1996.

[96] J-H Guo and C-L. Wang. Bit-serial systolic array implementation of euclid's algorithm for inversion and division in GF(2^m). In *VLSI-TSA'97*. Institute of Electrical and Electronics Engineers, Taiwan, 1997.

[97] J-H. Guo and C-L. Wang. Novel digit-serial systolic array implementation of euclid's algorithm for division in GF(2^m). In *IEEE ISCAS'98.*, volume 2, pages 478–481. IEEE, 1998.

[98] Katsuki Kobayashi. Studies on hardware-assisted implementation of arithmetic operations in galois field GF(2^m). Master's thesis, VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Rohm, Inc., 2009.

[99] S. Moon, J. Park, and Y. Lee. Fast VLSI arithmetic algorithms for high-security ECC applications. *IEEE Transactions on Consumer Electronics*, 47(3):700–708, 2001.

[100] M. Schmalisch and D. Timmermann. A reconfigurable arithmetic logic unit for ECC over GF(2^m). In *46th IEEE MWSCAS'03*, pages 831–834, 2003.

[101] Samuel Antao. Portable embedded systems: Efficient units for data processing and cryptography. Master's thesis, Instituto Superior Técnico, TU Lisbon, Portugal, 2008.

[102] Y. Jeong and W. Burleson. VLSI array synthesis for polynomial-GCD computation and application to finite field division. *IEEE Transactions on Circuits and Systems I*, 41(12):891–897, 1994.

[103] W. Drescher, K. Bachmann, and G. Fettweis. VLSI architecture for non-sequential inversion over GF(2^m) using the euclidean algorithm. In *ICSPAT*, volume 2, pages 1815–1819, 1997.

[104] W. Wilhelm. A new scalable VLSI architecture for reed-solomon decoders. *IEEE JSSC*, 34(3):388–396, 1999.

[105] Chih-Tsun Huang and Cheng-Wen Wu. High-speed C-Testable systolic array design for galois-field inversion. In *IEEE EDTC'97*, page 342. IEEE Computer Society, 1997.

[106] Yu-Chun Chuang and Cheng-Wen Wu. On-line error detection schemes for a systolic finite-field inverter. In *7th IEEE ATS'98*, pages 301–305. IEEE, 1998.

[107] C-H. Hu, C-H. Wu, M-D. Shieh, and Y-T. Hwang. Novel iterative division algorithm over GF(2^m) and its semi-systolic VLSI realization. In *43rd IEEE MWSCAS'00*, volume 1, pages 280–283. IEEE, 2000.

[108] 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. In *IEEE ISCAS'01*, volume 4, pages 33–36, 2001.

[109] 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. In *IEEE ISCAS'02*, volume 5, pages V–733. IEEE, 2002.

[110] W-C. Lin, M-D. Shieh, and C-M. Wu. Design of high-speed bit-serial divider in GF(2^m). In *IEEE ISCAS'10*, pages 713–716. IEEE, 2010.

[111] M-D. Shieh et al. Design of high-speed iterative dividers in GF(2^m). *JISE*, 27: 953–967, 2011.

[112] M. Hasan. Double-basis multiplicative inversion over GF(2^m). *IEEE Transactions on Computers*, 47(9):960–970, 1998.

[113] M. Hasan. Efficient computation of multiplicative inverses for cryptographic applications. In *15th IEEE ARITH'01*, pages 66–72. IEEE, 2001.

[114] Amir Daneshbeh. Bit serial systolic architectures for multiplicative inversion and division over GF(2^m). Master's thesis, University of Waterloo, 2005.

[115] A. Daneshbeh and M. Hasan. *A Class of Scalable Unidirectional Bit Serial Systolic Architectures for Multiplicative Inversion and Division over GF(2^m).* University of Waterloo, 2002.

[116] A. Daneshbeh and M. Hasan. A unidirectional bit serial systolic architecture for double-basis division over GF(2^m). In *16th IEEE ARITH'03*, pages 174–180, 2003.

[117] A. Daneshbeh and M. Hasan. A class of unidirectional bit serial systolic architectures for inversion over GF(2^m). *IEEE Transactions on Computers*, 54(3):370–380, 2005.

[118] Z. Yan and D. V. Sarwate. High-speed systolic architectures for finite field inversion and division. In *14th ACM GLSVLSI'04*, pages 462–465. ACM, 2004.

[119] Z. Yan and D. Sarwate. New systolic architectures for inversion and division in GF(2^m). *IEEE Transactions on Computers*, 52(11):1514–1519, 2003.

[120] Z Yan, DV Sarwate, and Z Liu. Hardware-efficient systolic architectures for inversion in GF(2^m). *IEE Proceedings-Information Security*, 152(1):31–45, 2005.

[121] Z. Yan and D. V. Sarwate. Reduced-complexity pipelined architectures for finite field inversions. In *IEEE Workshop on Signal Processing Systems Design and Implementation, SIPS'06*, pages 56–61. IEEE, 2006.

[122] D. V. Sarwate and Z. Yan. Modified euclidean algorithms for decoding reed-solomon codes. In *IEEE ISIT'09*, pages 1398–1402. IEEE, 2009.

[123] K. Kobayashi, N. Takagi, and K. Takagi. An algorithm for inversion in GF(2^m) suitable for implementation using a polynomial multiply instruction on GF(2). In *18th IEEE ARITH'07*, pages 105–112, 2007.

[124] K. Kobayashi and N. Takagi. Fast hardware algorithm for division in GF(2^m) based on EEA with parallelization of modular reductions. *IEEE Transactions on Circuits and Systems II*, 56(8):644–648, 2009.

[125] Ashwini Shende, Ujwala Ghodeswar, and GG Sarate. A systolic algorithm and architecture for inversion in GF(2^8). *International Journal on Mechanical Engineering and Robotics (IJMER)*, 2013.

[126] Chang Hoon Kim and Chun Pyo Hong. High-speed division architecture for GF(2^m). *Electronics Letters*, 38(15):835–836, 2002.

[127] Chang Hoon Kim, Soonhak Kwon, Chun Pyo Hong, and In Gil Nam. Efficient bit-serial systolic array for division over GF(2^m)[elliptic curve cryptosystem applications]. In *IEEE ISCAS'03*, volume 2, pages II–252. IEEE, 2003.

[128] Chang Hoon Kim, Soonhak Kwon, Jong Jin Kim, and Chun Pyo Hong. A compact and fast division architecture for a finite field GF(2^m). In *Computational Science and Its Applications—ICCSA 2003*, pages 855–864. Springer, 2003.

[129] Sosun Kim, Nam Su Chang, Chang Han Kim, Young-Ho Park, and Jongin Lim. A fast inversion algorithm and low-complexity architecture over GF(2^m). In *Computational Intelligence and Security*, pages 1–8. Springer, 2005.

[130] Guerric Meurice de Dormale and Jean-Jacques Quisquater. Iterative modular division over GF(2^m): novel algorithm and implementations on FPGA. In *Reconfigurable Computing: Architectures and Applications*, pages 370–382. Springer, 2006.

[131] G Meurice De Dormale and J-J Quisquater. Area and time trade-offs for iterative modular division over GF(2^m): novel algorithm and implementations on FPGA. *International journal of electronics*, 94(5):515–529, 2007.

[132] A. Cohen and K. Parhi. A new reconfigurable bit-serial systolic divider for GF(2^m) and GF(p). In *IEEE ICASSP'05*, pages 105–108, 2005.

[133] W. Chelton and M. Benaissa. Design space exploration of division over GF(2^m) on FPGA. In *IEEE 13th ICECS'06*, pages 172–175, 2006.

[134] Abdulah Abdulah Zadeh. High speed modular divider based on gcd algorithm. In *ICICS'07*, pages 189–200. Springer, 2007.

[135] M-S. Kang. Design of 163-bit modular divider based on e-GCD algorithm. *JSE*, 10 (2):233–242, 2013.

[136] J-P Deschamps and G Sutler. Finite field division implementation. In *IEEE FPL'05*, pages 670–674. IEEE, 2005.

[137] MM. Kermani et al. Reliable concurrent error detection architectures for EEA-based division over GF(2^m). *IEEE Transactions on VLSI Systems*, 22(5):995–1003, 2014.

[138] Alexandre F Tenca and LA Tawalbeh. Algorithm for unified modular division in GF(p) and GF(2^n) suitable for cryptographic hardware. *Electronics Letters*, 40(5): 304–306, 2004.

[139] E Savas and CK Koc. Architectures for unified field inversion with applications in elliptic curve cryptography. In *9th IEEE ICECS'02*, volume 3, pages 1155–1158. IEEE, 2002.

[140] Junfeng Fan and Ingrid Verbauwhede. A digit-serial architecture for inversion and multiplication in GF(2^m). In *2008 IEEE Workshop onSignal Processing Systems, SiPS*, pages 7–12. IEEE, 2008.

[141] J. Fan, L. Batina, and I. Verbauwhede. Design and design methods for unified multiplier and inverter and its application for HECC. *Integration, the VLSI journal*, 44(4):280–289, 2011.

[142] Apostolos P Fournaris and O Koufopavlou. A systolic inversion architecture based on modified extended euclidean algorithm for GF(2^k) fields. In *13th IEEE ICECS'06n*, pages 1081–1084. IEEE, 2006.

[143] A. P. Fournaris and O Koufopavlou. Applying systolic multiplication-inversion architectures based on modified extended euclidean algorithm for GF(2^k) in elliptic curve cryptography. *Computers & Electrical Engineering*, 33(5):333–348, 2007.

[144] A. P. Fournaris and O. Koufopavlou. One dimensional systolic inversion architecture based on modified GF(2^k) extended euclidean algorithm. In *12th IEEE DSD'09*, pages 736–741. IEEE, 2009.

[145] Wen-Ching Lin, Jun-Hong Chen, Ming-Der Shieh, and Chien-Ming Wu. A combined multiplication/division algorithm for efficient design of ECC over GF(2^m). In *IEEE Region 10 TENCON'07*, pages 1–4. IEEE, 2007.

[146] Lo'Ai Ali Tawalbeh. *A novel unified algorithm and hardware architecture for integrated modular division and multiplication in GF(p) and GF(2^n) suitable for public-key cryptography*. PhD thesis, Oregon State University, 2004.