



**Centro de Investigación y de Estudios Avanzados  
del Instituto Politécnico Nacional**

Departamento de Ingeniería Eléctrica  
Sección Computación

**CONSTRUCCIÓN DE OPERADORES BÁSICOS SOBRE  
CAMPOS FINITOS  $GF(2^m)$**

Tesis que presenta  
**Mario Alberto García Martínez**

Para obtener el grado de  
**Doctor en Ciencias**

En la especialidad de  
**Ingeniería Eléctrica**  
(Opción Computación)

Director de Tesis  
**Dr. Guillermo Morales Luna**

México D.F.

Diciembre 2004



# Dedicatoria

## **A Jesucristo:**

por haberme participado de una nueva naturaleza, y por haberme otorgado una identidad correcta, un propósito y un destino.

## **A Paty:**

compañera idónea, por su amor y su paciencia, que han permitido la culminación de este proyecto.

## **A Daniel y Jonathán mis hijos:**

bendición de Dios y motor y motivación para una constante superación como padre y profesionista.

## **A mis padres y hermanos:**

por su constante apoyo y confianza.

# Agradecimientos

**A mi asesor, el Dr. Guillermo Morales Luna:**

por la oportunidad y la confianza de que podríamos culminar este programa con éxito, y por la dirección siempre oportuna y comprometida que ha coadyuvado al cumplimiento de este propósito.

**A mis sinodales:**

por sus acertados y valiosos comentarios para darle a esta tesis una presentación cada vez más profesional.

**Al Dr. José Antonio Moreno Cadena**

por su apreciable intervención al canalizarnos en este programa y por la constante manifestación de su amistad y apoyo en el transcurso de todos mis estudios.

**Al Conacyt:**

por el apoyo recibido a través de la beca 159345.

**Al Tecnológico de Orizaba:**

que por medio de su administración y particularmente a través de la Academia de Ingeniería Electrónica, apoyaron siempre nuestro trabajo hasta la culminación de esta meta.

# Resumen

Las operaciones básicas sobre campos finitos o de Galois  $GF(2^m)$ , tales como la multiplicación, la exponenciación, la elevación al cuadrado y la división, son operaciones fundamentales para el desarrollo de algoritmos en las áreas de criptografía, códigos de corrección de errores y procesamiento digital de señales. Tales aplicaciones requieren de un alto grado de utilización de las operaciones mencionadas, por lo que estos cálculos consumen una gran cantidad de memoria y de tiempo de proceso cuando son realizados en software. Por razones de desempeño y seguridad, es preferible implementar tales algoritmos en hardware.

En esta tesis describimos la construcción de tres arquitecturas que operan sobre campos finitos  $GF(2^m)$ : un divisor, un exponenciador y un multiplicador, que se han implementado en FPGA's (*Field Programmable Gate Array*). Estos dispositivos tienen la gran ventaja de poder ser programados y reprogramados, lo que facilita el proceso de verificación de los circuitos implementados.

El uso de FPGA's se presenta como una alternativa a las soluciones de diseño tradicionales realizadas con ASIC's (*Application Specific Integrated Circuit*), y con esto se pretende evitar el inconveniente de la poca flexibilidad que implica este método de diseño. Medimos la complejidad de espacio en función del número de CLB's (*Configurable Logic Block*) requeridos en el FPGA, frente a la tradicional medida que se basa en el número de compuertas usadas en una implementación. Usamos VHDL (*VHSIC-Hardware Description Language* y, a su vez, VHSIC es *Very High Speed Integrated Circuit*) para describir cada una de tales arquitecturas, y los resultados de la síntesis y la simulación se han realizado usando el paquete computacional ISE 4.1i de la compañía Xilinx.

# Abstract

Basic operations on  $GF(2^m)$ , as multiplication, exponentiation, squaring and division, are fundamental on cryptography, error-correction codes and digital signal processing. Since those operations are greatly used on such applications, a bad implementation or management will result in great costs in time and memory. Indeed hardware implementations are more convenient than software constructions.

Our work describes the construction of three architectures for  $GF(2^m)$  operations: a divider, an exponentiator and a multiplier, that have been implemented in FPGA's (*Field Programmable Gate Arrays*), which are suited for programming and reprogramming processes, turning easier the verification of implemented circuits.

The use of FPGA's appears as an alternative to traditional solution designs that are made with ASIC's (*Application Specific Integrated Circuit*), avoiding the disadvantage of the small flexibility that implies that design method. Using the FPGA's we measure the space complexity in function of the number of required CLB's (*Configurable Logic Block*) in the FPGA, in contrast with traditional measurement based on the number of used gates in an implementation. We use VHDL (*VHSIC-Hardware Description Language*) to describe each architecture and the synthesis results. The simulation process has been realized using the tools provided by Xilinx ISE 4.1 package.

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Objetivo de la tesis . . . . .	4
1.3. Estado del arte . . . . .	5
1.4. Contribuciones propias . . . . .	6
1.5. Descripción de la tesis . . . . .	9
<b>2. Aritmética de <math>GF(2^m)</math> en hardware</b>	<b>11</b>
2.1. Preliminares matemáticos . . . . .	12
2.1.1. Grupos . . . . .	12
2.1.2. Anillos y campos . . . . .	15
2.1.3. Polinomios . . . . .	19
2.1.4. Bases de campos finitos . . . . .	22
2.2. Multiplicación en $GF(2^m)$ . . . . .	23
2.2.1. Consideraciones previas . . . . .	23
2.2.2. Multiplicación en paralelo . . . . .	28
2.2.3. Multiplicación serial . . . . .	36
2.2.4. Multiplicación por dígitos . . . . .	40
2.2.5. Comparación de multiplicadores $GF(2^m)$ . . . . .	41
2.3. Inversión y división en $GF(2^m)$ . . . . .	43
2.3.1. Teorema pequeño de Fermat . . . . .	44

2.3.2.	Algoritmo de Euclides . . . . .	46
2.3.3.	Algoritmo de Gauss-Jordan . . . . .	49
2.3.4.	Comparación de inversores $GF(2^m)$ . . . . .	50
2.4.	Exponenciación en $GF(2^m)$ . . . . .	51
2.4.1.	El método binario . . . . .	51
2.4.2.	Elevación al cuadrado en $GF(2^m)$ . . . . .	54
<b>3.</b>	<b>Dispositivos en hardware para <math>GF(2^m)</math></b>	<b>59</b>
3.1.	Consideraciones generales . . . . .	59
3.2.	Hardware reconfigurable . . . . .	61
3.2.1.	Sistemas digitales y FPGA's . . . . .	61
3.2.2.	FPGA's y ASIC's . . . . .	62
3.2.3.	Características de los FPGA's . . . . .	65
3.2.4.	Estructura interna de un FPGA . . . . .	66
3.2.5.	FPGA's y criptografía . . . . .	67
3.2.6.	Metodología de diseño con FPGA's . . . . .	71
3.3.	Arreglos sistólicos . . . . .	73
3.3.1.	Generalidades . . . . .	73
3.3.2.	Metodología de diseño . . . . .	76
3.4.	Ejemplo de diseño . . . . .	77
3.4.1.	Algoritmo de multiplicación . . . . .	77
3.4.2.	Gráfica de dependencia . . . . .	79
3.4.3.	Sistolización . . . . .	79
3.4.4.	Descripción con VHDL . . . . .	80
3.4.5.	Implementación en el FPGA y resultados de la síntesis . . . . .	80
3.5.	Resumen . . . . .	84
<b>4.</b>	<b>Implementaciones realizadas</b>	<b>85</b>
4.1.	Implementación en FPGA de un Divisor y Multiplicador sistólico y serial . . . . .	85



## ÍNDICE GENERAL

9

4.1.1. Algoritmo de división . . . . .	86
4.1.2. Arquitectura del Divisor . . . . .	86
4.1.3. Análisis de complejidad . . . . .	93
4.1.4. Resultados . . . . .	95
4.1.5. Algoritmo de multiplicación . . . . .	99
4.1.6. Arquitectura del Multiplicador . . . . .	100
4.1.7. Análisis de complejidad . . . . .	102
4.2. Implementación de un Exponenciador para $GF(2^m)$ . . . . .	103
4.2.1. Algoritmo de exponenciación . . . . .	104
4.2.2. Arquitectura del Exponenciador . . . . .	105
4.2.3. Análisis de complejidad . . . . .	107
4.2.4. Resultados . . . . .	108
4.3. Implementación eficiente de un Multiplicador Serial/Paralelo .	109
4.3.1. Algoritmo de multiplicación . . . . .	109
4.3.2. Arquitectura del Multiplicador . . . . .	110
4.3.3. Análisis de complejidad . . . . .	112
4.3.4. Resultados . . . . .	114
<b>5. Conclusiones</b>	<b>117</b>
5.1. Sumario y contribuciones . . . . .	117
5.2. Resultados y criterios de validación . . . . .	119
5.3. Perspectivas de desarrollo ulterior . . . . .	119
<b>A. Descripción en VHDL de los diseños realizados</b>	<b>121</b>



# Índice de figuras

2.1. Adición en $GF(2^m)$ . . . . .	26
2.2. Elevación al cuadrado en base normal . . . . .	55
2.3. Elevación al cuadrado en base estándar . . . . .	57
3.1. Organización de los Sistemas Digitales . . . . .	62
3.2. Estructura interna de un FPGA . . . . .	67
3.3. Estructura interna de un CLB . . . . .	68
3.4. Plataformas para implementación de algoritmos criptográficos . . . . .	68
3.5. Características entre plataformas . . . . .	70
3.6. Proceso de diseño de un FPGA . . . . .	72
3.7. Estructura de un arreglo sistólico . . . . .	74
3.8. Procesamiento de datos en un arreglo sistólico . . . . .	74
3.9. Diferentes tipos de arreglos sistólicos . . . . .	75
3.10. Gráfica de dependencia del multiplicador $GF(2^3)$ . . . . .	79
3.11. Arquitectura de la celda de la GD . . . . .	80
3.12. Arreglo sistólico del multiplicador serial $GF(2^3)$ . . . . .	83
4.1. Arquitectura del Divisor para $GF(2^m)$ . . . . .	87
4.2. Estructura del arreglo Gen-Mat . . . . .	88
4.3. Diagrama de la celda Cel101 . . . . .	89
4.4. Estructura del arreglo Solución . . . . .	90
4.5. Diagrama de Cel102 . . . . .	91

4.6. Diagrama de Cel103 . . . . .	92
4.7. Simulación de Gen-Mat . . . . .	96
4.8. Simulación de Solución . . . . .	97
4.9. Circuito del Divisor . . . . .	98
4.10. Simulación del Divisor . . . . .	99
4.11. Arquitectura del Multiplicador . . . . .	101
4.12. Estructura para la multiplicación matricial . . . . .	101
4.13. Diagrama de la celda mul . . . . .	102
4.14. Arquitectura del Exponenciador . . . . .	106
4.15. Consumo de recursos del exponenciador de 32 bits . . . . .	109
4.16. Estructuras básicas del Multiplicador . . . . .	111
4.17. Multiplicación $\alpha A(\alpha)$ en $GF(2^m)$ . . . . .	112
4.18. Celda del arreglo . . . . .	113
4.19. Multiplicador <i>LSE-first</i> $GF(2^{239})$ . . . . .	113

# Índice de cuadros

1.1. Niveles de estudio para $GF(2^m)$ . . . . .	3
2.1. Algoritmo para la adición de dos elementos en el campo de Galois.	26
2.2. Comparación de los multiplicadores paralelos $GF(2^m)$ . . . . .	36
2.3. Algoritmo para la multiplicación <i>LSE-first</i> de dos elementos en $GF(2^m)$ . . . . .	39
2.4. Algoritmo para la multiplicación <i>MSE-first</i> de dos elementos en $GF(2^m)$ . . . . .	40
2.5. Algoritmo para la multiplicación por dígitos de dos elementos en $GF(2^m)$ . . . . .	42
2.6. Comparación de los multiplicadores $GF(2^m)$ . . . . .	42
2.7. Algoritmo para la inversión de un elemento en $GF(2^m)$ . . . . .	45
2.8. Algoritmo para calcular el MCD de dos elementos en $GF(2^m)$ . . .	47
2.9. Algoritmo para calcular el inverso multiplicativo de un elemento en $GF(2^m)$ . . . . .	48
2.10. Algoritmo para calcular la división de dos elementos en $GF(2^m)$ . .	50
2.11. Comparación de inversores $GF(2^m)$ . . . . .	51
2.12. Algoritmo <i>MSB-first</i> , para elevar a una potencia, un elemento en $GF(2^m)$ . . . . .	52
2.13. Algoritmo <i>LSB-first</i> para elevar a una potencia, un elemento en $GF(2^m)$ . . . . .	52

2.14. Método binario de exponenciación $GF(2^m)$ . . . . .	53
2.15. Algoritmo binario para elevar a una potencia un elemento en $GF(2^m)$	53
3.1. Implementación de multiplicadores $GF(2^m)$ en SW/HW . . . . .	60
3.2. Algoritmo para la multiplicación sobre $GF(2^m)$ . . . . .	78
3.3. Descripción VHDL del Multiplicador . . . . .	81
3.4. Descripción VHDL de la celda del arreglo . . . . .	82
3.5. Complejidad de espacio del Multiplicador $GF(2^m)$ . . . . .	84
4.1. Descripción de la celda Ce101. . . . .	89
4.2. Descripción de las celdas Ce102 y Ce103 . . . . .	92
4.3. Complejidad de espacio del Divisor $GF(2^m)$ . . . . .	94
4.4. Requerimientos de espacio en el FPGA XCS4010 . . . . .	94
4.5. Representación de $GF(2^4)$ usando el polinomio irreducible $P(x)$ . . .	95
4.6. Consumo de recursos en el FPGA XCS4010 . . . . .	97
4.7. Comparación del divisor $GF(2^m)$ . . . . .	98
4.8. Descripción de la celda mul . . . . .	100
4.9. Complejidad de espacio del Multiplicador $GF(2^m)$ . . . . .	103
4.10. Requerimientos de espacio en el FPGA XCS4010. . . . .	103
4.11. Algoritmo Exponenciación <i>LSB-first</i> . . . . .	104
4.12. Ejemplo de exponenciación . . . . .	105
4.13. Operaciones básicas del método binario de exponenciación . . . . .	105
4.14. Complejidad de espacio del Exponenciador $GF(2^m)$ . . . . .	107
4.15. Requerimientos de espacio en el FPGA XCV300 . . . . .	107
4.16. Tabla de comparación . . . . .	108
4.17. Multiplicación <i>LSE-first</i> de dos elementos en $GF(2^m)$ . . . . .	110
4.18. Complejidad de espacio del Multiplicador . . . . .	114
4.19. Resultados de la síntesis . . . . .	115
4.20. Características del FPGA XCV300 . . . . .	115
4.21. Comparación del Multiplicador con implementaciones en SW . .	115

4.22. Comparación del Multiplicador con implementaciones en HW . . 116





# Capítulo 1

## Introducción

### 1.1. Motivación

Para muchos sistemas de comunicación modernos, tales como las redes de computadoras, enlaces satelitales, transmisión de datos digitales o la grabación de discos compactos, es de vital importancia asegurar la confiabilidad y la seguridad de la información que manejan.

La transmisión de datos digitales y su almacenamiento es acompañada a menudo con la posibilidad de corrupción de los datos, por lo que se requiere de métodos y técnicas para la solución de este problema. Los códigos de detección y corrección de errores, en particular los códigos BCH (*Bose, Chaudhuri y Hocquenghem*) y su subclase los códigos RS (*Reed-Solomon*)[45][72] son ampliamente utilizados en los sistemas digitales de comunicación para la solución del problema mencionado anteriormente. El fundamento teórico de estos códigos está basado en la *teoría de los campos finitos* ó campos de Galois, representados como  $GF(q)$ , la cual forma parte de la disciplina matemática del Algebra. Por otro lado, la transmisión de datos digitales a través de canales públicos de comunicación, como Internet, tiene implícito el riesgo de que personas no autorizadas puedan leer o alterar la información que se transmite por tales canales de comunicación. Las técnicas para el

manejo de una transmisión segura son proporcionadas por la *criptografía*, la que se ocupa del estudio de los métodos para establecer una comunicación segura en un medio que no lo es [58]. Varios esquemas criptográficos han sido desarrollados durante los últimos años basados en la dificultad de solución de ciertos problemas matemáticos, tales como RSA (*Rivest, Shamir, Adleman*)[76], desarrollado en 1978 y que se basa en el problema de factorizar números enteros grandes. Otro es el sistema ElGamal[16] basado en el problema para calcular un logaritmo discreto. En 1985, Neil Koblitz [40] y Víctor Miller [54] en forma independiente propusieron el ECC (*Elliptic Curve Cryptosystem*) cuya seguridad está basada en el problema del logaritmo discreto sobre los puntos de una curva elíptica. Estos dos últimos esquemas se realizan en última instancia en campos finitos.

Una cantidad importante de los algoritmos utilizados en las áreas de aplicación que se han mencionado requieren de operaciones aritméticas sobre campos finitos, como son la suma, la multiplicación, el inverso multiplicativo, la exponenciación, la elevación al cuadrado y la división. Por ejemplo, la multiplicación y la división son operaciones utilizadas en los codificadores y decodificadores en sistemas Reed-Solomon de códigos de corrección de errores. A su vez, la suma, la multiplicación, la elevación al cuadrado y el inverso multiplicativo son fundamentales para el diseño de criptosistemas que operan con curvas elípticas. La exponenciación, por su parte, es una operación utilizada por el algoritmo de El Gamal en el proceso de encriptamiento de mensajes en sistemas criptográficos.

Debido al alto grado de utilización de estas operaciones por los algoritmos mencionados, se requiere del consumo de una gran cantidad de memoria y tiempo de proceso cuando son realizadas por software. Así es que por razones de desempeño y seguridad, es preferible implementar tales operaciones en hardware. Este es el problema que atendemos en este trabajo, la necesidad de un mejor desempeño de las operaciones básicas sobre campos finitos  $GF(2^m)$ . Y proponemos una solución para tres de tales operaciones, la división, la exponenciación y la multiplicación, a través de la selección e implementación en *Lógica Reconfigurable* de algoritmos

<b>Aplicación</b>	Criptografía, Códigos de corrección de errores, DSP's
<b>Análisis algorítmico</b>	Suma, multiplicación, inversión, división, exponenciación
<b>Implementación</b>	VLSI, Circuitos reconfigurables

Cuadro 1.1: Niveles de estudio para  $GF(2^m)$ 

que permiten la construcción de arquitecturas eficientes.

En el estudio de los campos finitos se pueden apreciar tres niveles de abstracción en el proceso de diseño, que a su vez definen tres áreas de estudio particulares: un nivel de *implementación*, un nivel de *análisis y diseño algorítmico* y un nivel de *aplicación*. Aunque mencionamos a la criptografía como área de aplicación y estudiamos los algoritmos para las operaciones aritméticas sobre campos finitos, nuestro trabajo se ubica especialmente en el nivel de *implementación*. En el cuadro 1.1 se muestran los niveles mencionados.

En esta tesis realizamos la implementación en hardware de tres arquitecturas básicas para  $GF(2^m)$ : un divisor, un exponenciador y un multiplicador, y medimos su eficiencia en función de sus *complejidades de tiempo y de espacio*. Las características generales de cada implementación y su aportación al estado del arte se mencionan en el apartado 1.4 de este capítulo. En contraste con el uso tradicional de circuitos ASIC's (*Application Specific Integrated Circuit*), hemos usado *circuitos reconfigurables* tales como los FPGA's (*Field Programmable Gate Arrays*). Como su nombre lo indica, estos circuitos son construídos en base a arreglos de compuertas que son interconectados durante el proceso de programación. Es esta característica de programación y reprogramación la que los hace sumamente atractivos durante el proceso de diseño y construcción, ya que permiten una verificación y corrección casi inmediata, lo que no es posible con los circuitos ASIC's. Además, si la implementación tiene características de regularidad y modularidad como las sistólicas, el uso de FPGA's nos facilita la asignación de los recursos de espacio debido a su propia construcción interna, como se estudia en el capítulo 3.

## 1.2. Objetivo de la tesis

El objetivo general de esta tesis es la implementación en *Lógica Reconfigurable* de arquitecturas funcionales para las operaciones básicas de campos de Galois. Para lograr tal objetivo:

- Realizamos una selección de algoritmos que facilitaran implementaciones eficientes, considerando el *tipo de arquitectura*, que puede ser serial, paralela o por dígitos, y el *tipo de base* utilizado para representar a los elementos del campo finito, que puede ser canónica, normal o dual. Además se ha procurado que tales algoritmos permitan estructuras que sean independientes del generador del campo.
- Entre los diferentes tipos de arquitecturas, es decir seriales, paralelas y por dígitos, decidimos trabajar con las seriales debido a la menor demanda de recursos físicos para su implementación en hardware, como se estudia en el capítulo 2.
- Utilizamos FPGA's para la implementación física de los algoritmos, ya que su estructura interna y su característica de reprogramación nos han permitido diseñar arquitecturas funcionales, como se estudia en el capítulo 4.
- Hicimos descripciones *parametrizables* con VHDL (*VHSIC-Hardware Description Language*), que han permitido características de *escalabilidad* a los circuitos que hemos implementado.
- Usamos estructuras sistólicas, que debido a su modularidad presentan grandes facilidades para su implementación en hardware reconfigurable.

Como recientemente se acostumbra, medimos la complejidad de espacio por el número de CLB's (*Configurable Logic Blocks*) requeridos por la síntesis del FPGA, en contraste con el número de compuertas utilizadas, que es la medida tradicional. Una meta importante ha sido la aplicación de estas estructuras a la criptografía

con curvas elípticas, que operan con curvas sobre campos  $GF(2^m)$ , en particular  $GF(2^{193})$  y  $GF(2^{239})$ .

### 1.3. Estado del arte

En 1971, Laws y Rushforth [50] propusieron una celda básica para realizar la multiplicación sobre  $GF(2^m)$  en hardware. Posteriormente, Massey-Omura [55] y Berlekamp [5] desarrollaron diferentes arquitecturas para realizar esta operación básica. Desde entonces se han presentado en la literatura varias propuestas para realizar la multiplicación por hardware sobre  $GF(2^m)$ .

Entre los algoritmos de multiplicación más importantes propuestos para ser implementados en hardware se encuentra el de Massey-Omura [55], que opera con bases normales, y que debido a su representación regular formada por etapas de compuertas AND y XOR facilita su implementación en hardware. Se caracteriza por la forma sencilla en que puede realizar la operación de elevación al cuadrado, que consiste en un simple corrimiento de bits. Este multiplicador opera con entradas paralelas de los datos y produce una salida en forma serial del producto. Wang *et al.* en [88] realizan un buen análisis de este multiplicador. El multiplicador de Berlekamp [5] opera también con entradas en paralelo y salida en serie, pero una de las entradas usa una base dual mientras que la otra usa una base polinomial o estándar. La salida está representada en base dual. Otro multiplicador importante es el desarrollado por Mastrovito [52], que es un diseño totalmente combinatorio. Opera con entradas y salidas en paralelo, y no presenta ningún retardo de tiempo en su operación. Produce un resultado de  $m$ -bits cada ciclo de reloj y trabaja sobre una base polinomial. El multiplicador de Montgomery [57] ha sido aplicado principalmente para realizar multiplicación de números enteros, pero actualmente se han hecho algunas propuestas para utilizarlo en el campo  $GF(2^m)$  [80]. Opera con una representación redundante de los elementos del campo y su principal característica es que en el proceso de reducción modular evita las divi-

siones y las reemplaza con simples corrimientos. También hemos de mencionar el multiplicador propuesto por Hasan y Bhargava [30], el cual utiliza una base dual y opera con entrada y salida serial de datos. Su estructura sistólica facilita su implementación en hardware. Tiene un retardo inicial de  $2m$  ciclos de reloj y operando en “*pipeline*” produce un producto de  $m$ -bits cada ciclo de reloj. Paar y Rosner en [70] y Paar y Soria-Rodríguez en [71] proponen estructuras para multiplicación que trabajan sobre una representación de campos compuestos de la forma  $GF(2^k)$ , donde  $k = mn$ . Esta composición permite derivar una estructura combinatoria y sin retardos con entrada y salida paralela que permite mejorar la eficiencia de la complejidad de espacio en el dispositivo físico. En [35] Yeong y Burleson comparan los algoritmos que pueden ser implementados en circuitos VLSI. En [69], Paar y Lange comparan diversos multiplicadores considerando sus complejidades de espacio y tiempo. También, en [2], Ahlquist *et al.* hacen una importante comparación de multiplicadores implementados en FPGA’s. Las tres arquitecturas que hemos desarrollado en este trabajo operan sobre una base polinomial. La arquitectura del divisor es *serial/serial* y calcula una división en  $5m - 1$  ciclos de reloj. El exponenciador es también una arquitectura *serial/serial*, y realiza una exponenciación en  $sn^2$  ciclos de reloj, donde  $s$  es el tiempo requerido para el cálculo de la multiplicación dentro del algoritmo de exponenciación. En cuanto al multiplicador que hemos implementado, su estructura es *serial/paralela* y calcula una multiplicación en  $m$  ciclos de reloj.

## 1.4. Contribuciones propias

Describimos las principales contribuciones que aportamos con nuestro trabajo:

1. Hemos realizado la implementación en un FPGA de una arquitectura sistólica y serial para el cálculo de la división/multiplicación sobre  $GF(2^m)$ , como lo proponen Hasan y Bhargava en [30]. Aportamos las siguientes contribuciones:

- La propia implementación en hardware de un algoritmo novedoso. A diferencia del método tradicional para calcular la división, que consiste en una multiplicación del dividendo por el inverso del multiplicador, nuestro divisor calcula la división por medio de la solución de un sistema de ecuaciones lineales usando el método de Gauss-Jordan. Según nuestra información, no hay en la literatura reciente ningún reporte acerca de una implementación de este tipo. En [33], [87], [28] y [26] se reportan algunas estructuras para el cálculo del inverso multiplicativo, pero en ningún caso se reporta alguna implementación como la nuestra
  - Aunque el algoritmo ha sido propuesto para ser implementado con circuitos VLSI, hemos mostrado que su implementación en FPGA aprovecha la estructura sistólica de la arquitectura para realizar descripciones VHDL *parametrizables*, que proporcionan una característica de *escalabilidad* al circuito.
  - Aunque los autores del algoritmo consideran la posibilidad de integrar en un solo circuito integrado las dos operaciones mencionadas, nuestras pruebas han mostrado la dificultad para lograr tal propósito, debido al orden cuadrático de la *complejidad de espacio* del circuito. Con nuestros resultados mostramos que el orden mayor del campo finito que puede ser implementado en un FPGA XCS4010 es 12, es decir  $GF(2^{12})$ . Este divisor y el multiplicador tienen un aplicación directa en la construcción de decodificadores Reed-Solomon, los cuales operan con elementos de campos finitos  $GF(2^8)$ .
2. Hemos introducido una nueva arquitectura para calcular la exponenciación en  $GF(2^m)$ , basada en el método binario en su versión *LSB-first*.
- Partiendo de la observación de que en cada ciclo del algoritmo se realiza una elevación al cuadrado y una posible multiplicación, proponemos una arquitectura que sólo requiere de dos multiplicadores operando en

paralelo, junto con un multiplexor y dos registros. La importancia de esta arquitectura radica en el ahorro de hardware requerido para su implementación.

- Presentamos el algoritmo, la arquitectura, la descripción VHDL y la medida de *complejidad de espacio* en el FPGA. Por medio del proceso de síntesis del circuito, hemos verificado tal ahorro mencionado. Además, hemos comparado nuestros resultados con otros trabajos presentados en la literatura.
  - Este exponenciador tiene una aplicación importante en el algoritmo de El Gamal para encriptamiento y firmas digitales operando sobre  $GF(2^m)$ .
3. Hemos implementado en un FPGA Virtex-XCV300, un multiplicador serial/paralelo en su versión *LSE-first* que opera en los campos  $GF(2^{193})$  y  $GF(2^{239})$ , que son campos recomendados por el NIST (*National Institute of Standards and Technology*) de los E.U.A. para su aplicación en criptosistemas con Curvas Elípticas.
- Basados en la observación de las características comunes entre una arquitectura sistólica y la estructura interna de un FPGA, hemos realizado una descripción parametrizable con VHDL, logrando una implementación eficiente del multiplicador. Además, hemos hecho una estimación de la *complejidad de espacio* del circuito, basados en el hecho más particular de que las celdas idénticas que forman una de las estructuras del multiplicador se pueden hacer corresponder eficientemente con los recursos del FPGA por medio de una descripción VHDL adecuada. Este análisis es parte importante de nuestra aportación en esta tesis.
  - Por otro lado hemos comprobado la precisión de nuestro análisis, al realizar la implementación física del circuito en el FPGA, y por medio



de los reportes del sintetizador. El multiplicador usa el 11% de los recursos de área del FPGA. Además realiza la multiplicación en  $m$ -ciclos de reloj y, operando a 75 Mhz, calcula una multiplicación en 3.1 microsegundos. Comparado con otros multiplicadores reportados en la literatura reciente, presenta un mejor desempeño tanto en tiempo como en espacio.

## 1.5. Descripción de la tesis

Después de considerar los objetivos y características de nuestro trabajo en esta sección, en el capítulo 2 presentamos una introducción al estudio de los campos finitos a través de teoremas y definiciones que son básicos para la comprensión de nuestro trabajo. Además, nos ocupamos del estudio de las operaciones fundamentales sobre  $GF(2^m)$  tales como la adición, la multiplicación, la inversión, la exponenciación y la elevación al cuadrado; y revisamos los algoritmos más utilizados para su implementación en hardware. En el capítulo 3 consideramos las características generales de los dispositivos lógicos programables y de los FPGA's. También introducimos el estudio de las arquitecturas sistólicas y consideramos sus generalidades. Las implementaciones físicas que hemos realizado las presentamos en el capítulo 4 en el que describimos detalladamente cada uno de los diseños desarrollados. Por último, en el capítulo 5 consideramos las conclusiones de nuestro trabajo, en las que mencionamos los resultados obtenidos y las perspectivas de desarrollo futuro.



# Capítulo 2

## Aritmética de $GF(2^m)$ en hardware

En este capítulo presentamos los fundamentos y las definiciones matemáticas que consideramos imprescindibles para una mejor comprensión de este trabajo. Para la presentación de los conceptos formales hemos seguido el libro de Lidl y Niederreiter [48] cuya lectura recomendamos para un estudio más amplio de los campos finitos. Simultáneamente, presentamos los resultados que consideramos más representativos en cuanto a las complejidades de los algoritmos para implementar las operaciones básicas sobre campos finitos.

En cuanto a las implementaciones, la eficiencia de los operadores de campos se caracteriza mediante las siguientes dos medidas:

- *La complejidad de espacio*, definida por el número de compuertas lógicas requeridas para la construcción del circuito
- *La complejidad de tiempo*, que se mide por el número de retardos de las compuertas que forman la ruta crítica de datos.

Revisamos incluso algoritmos paralelos que, aunque no los implementamos, nos permitirán contrastar con ellos nuestros propios resultados.

## 2.1. Preliminares matemáticos

### 2.1.1. Grupos

**Definición 2.1.1** Una *operación binaria*  $\cdot$  sobre un conjunto  $S$  es una regla o función que asigna a cada par de elementos  $a$  y  $b \in S$ , un tercer elemento único  $c = a \cdot b \in S$ . La operación  $\cdot$  es del tipo  $S \times S \rightarrow S$  y por eso se dice que  $S$  es un conjunto *cerrado* bajo ella.

**Ejemplo 2.1.1** La suma es una operación binaria en el conjunto de los enteros.

**Definición 2.1.2** Una *estructura algebraica* es un conjunto  $S$  dotado de una o más operaciones binarias.

**Ejemplo 2.1.2** Los grupos, anillos y campos son estructuras algebraicas.

**Definición 2.1.3** Un *grupo*  $(G, \cdot)$  consiste de un conjunto  $G$  junto a una operación binaria  $\cdot$  sobre  $G$  que satisface los siguientes axiomas:

1. La operación  $\cdot$  es *asociativa*:  $a \cdot (b \cdot c) = (a \cdot b) \cdot c$  para todo  $a, b, c \in G$ .
2. La operación  $\cdot$  *posee unidad*: Existe un  $e \in G$ , llamado el *elemento unidad*, tal que  $a \cdot e = e \cdot a = a$  para todo  $a \in G$ .
3. La operación  $\cdot$  *posee inversos*: Para cada  $a \in G$  existe un elemento  $a^{-1} \in G$ , llamado el *inverso* de  $a$ , tal que  $a \cdot a^{-1} = a^{-1} \cdot a = e$ .

Si el grupo además satisface que para todos  $a, b \in G$ :  $a \cdot b = b \cdot a$ , entonces se dice ser *abeliano* o *conmutativo*.

**Definición 2.1.4** El conjunto  $\mathbb{Z}_n$  de los enteros reducidos módulo el entero positivo  $n$ , forma un grupo  $(\mathbb{Z}_n, +)$  con la suma usual  $+$ , pues cumple con las tres propiedades que definen a un grupo: esta operación es asociativa, el número 0 es la unidad, y el inverso de cada elemento  $a$  es  $-a$ .

Por otro lado, para un entero  $p$  primo,  $(\mathbb{Z}_p^*, \cdot)$  es un grupo con la multiplicación módulo  $p$ : su unidad es el número 1, y el inverso de cada  $a \in \mathbb{Z}_p^*$  siempre existe dado que  $p$  es primo.

**Definición 2.1.5** Un subconjunto no vacío  $H$  de un grupo  $G$  es un *subgrupo* de  $G$  si  $H$  es en sí mismo un grupo con respecto a la operación de  $G$ . Si  $H$  es un subgrupo de  $G$  y  $H \neq G$ , entonces  $H$  es llamado un subgrupo *propio* de  $G$ .

**Ejemplo 2.1.3** Si consideramos el grupo  $\mathbb{Z}_{19}^*$ , el conjunto  $H = \{1, 7, 11\}$  es un subgrupo de  $\mathbb{Z}_{19}^*$ , ya que todas las combinaciones de cálculos módulo 19 con elementos de  $H$  dan resultados en  $H$ :

$\cdot$	1	7	11
1	1	7	11
7	7	11	1
11	11	1	7

**Definición 2.1.6** Un grupo  $(G, \cdot)$  es *cíclico* si existe un elemento  $\alpha \in G$  tal que para cada  $b \in G$  existe un número entero  $i$  tal que  $b = \alpha^i$ . Tal elemento  $\alpha$  es llamado un *generador* del grupo  $(G, \cdot)$ .

**Teorema 2.1.1** Si  $G$  es un grupo y  $a \in G$ , entonces el conjunto de todas las potencias de  $a$  forma un subgrupo cíclico de  $G$ , llamado el subgrupo generado por  $a$ , y se denota por  $\langle a \rangle$ .

**Ejemplo 2.1.4** El conjunto  $H = \{1, 7, 11\}$  es un subgrupo cíclico de  $\mathbb{Z}_{19}^*$  y de hecho posee dos generadores: 7 y 11,  $H = \langle 7 \rangle = \langle 11 \rangle$  ( $1 = 7^0$ ,  $7 = 7^1$ ,  $11 = 7^2$ ;  $1 = 11^0$ ,  $7 = 11^2$ ,  $11 = 11^1$ ).

**Definición 2.1.7** Para enteros arbitrarios  $a$  y  $b$  y un entero positivo  $n$ , se dice que  $a$  es *congruente* con  $b$  módulo  $n$ , si la diferencia  $a - b$  es un múltiplo de  $n$ , es decir, si  $a = b + kn$  para algún entero  $k$ . Se escribe  $a \equiv b \pmod{n}$ .

**Definición 2.1.8** Sea  $G$  un grupo y  $a$  un elemento en  $G$ . El *orden* de  $a$  es el menor entero positivo  $t$  tal que  $a^t = 1$ , si es que tal entero existe, en otro caso el orden de  $a$  es definido como  $\infty$ .

**Ejemplo 2.1.5** En el conjunto  $\mathbb{Z}_{16}^*$ , el orden del número 3 es  $t = 4$ , ya que, módulo 16 sus potencias son, respectivamente,  $3^1 = 3$ ,  $3^2 = 9$ ,  $3^3 = 11$  y  $3^4 = 1$ .

**Ejemplo 2.1.6** Se tiene que, de hecho,  $\mathbb{Z}_p^*$  con  $p$  primo, es un grupo cíclico. Cada generador se llama también una *raíz primitiva* de  $p$ .

En efecto, de acuerdo con [84], sea  $N = \{n \in \mathbb{N} \mid \exists x \in \mathbb{Z}_p^* : x^n = 1\}$  el conjunto de órdenes de elementos en  $\mathbb{Z}_p^*$ . Cada elemento en  $N$  divide a  $p-1$ . Sea  $m = \text{m.c.m.} N$  el mínimo común múltiplo de los elementos en  $N$ . Si, de acuerdo con el Teorema Fundamental de la Aritmética, se descompone a  $m$  como producto de potencias de primos, digamos  $m = \prod_{i \leq k} q_i^{e_i}$  donde cada  $q_i$  es primo y  $e_i$  es un entero positivo, entonces necesariamente para cada  $i \leq k$  existe  $n_i \in N$  tal que  $q_i^{e_i} \mid n_i$ , es decir,  $n_i = d_i q_i^{e_i}$ . Sea  $x_i \in \mathbb{Z}_p^*$  tal que su orden sea  $n_i$ . Entonces el orden de  $x_i^{d_i}$  es precisamente  $q_i^{e_i}$  y, en consecuencia, el elemento  $z = \prod_{i \leq k} x_i^{d_i}$  tiene orden  $m$ . Puede verse con facilidad que necesariamente  $m = p-1$  y por tanto  $z$  es un generador de  $\mathbb{Z}_p^*$ .

La localización eficiente de raíces primitivas módulo un primo es un problema que ha llamado la atención desde siempre [73].

**Definición 2.1.9** Un grupo  $G$  es *finito* si contiene un número finito de elementos, el cual se conoce como el *orden* del grupo, y se denota  $|G|$ .

**Ejemplo 2.1.7**  $\mathbb{Z}_4$  es un grupo finito de orden 4. Su tabla de operación, conocida como *tabla de Cayley*, es

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

### 2.1.2. Anillos y campos

**Definición 2.1.10** Un *anillo*  $(R, +, \cdot)$  consiste de un conjunto  $R$  con dos operaciones binarias denotadas  $+$  (adición) y  $\cdot$  (multiplicación) sobre  $R$ , tal que:

1.  $(R, +)$  es un grupo conmutativo, con unidad denotada 0.
2.  $(R, \cdot)$  es un semigrupo, es decir, la operación  $\cdot$  es asociativa:  $a \cdot (b \cdot c) = (a \cdot b) \cdot c$  para todos  $a, b, c \in R$ .
3. Existe una unidad multiplicativa, denotada 1:  $1 \cdot a = a \cdot 1$ , para todo  $a \in R$ , y  $1 \neq 0$ .
4. Las dos operaciones están relacionadas mediante las leyes *distributivas*:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \text{ y}$$

$$(b + c) \cdot a = (b \cdot a) + (c \cdot a), \text{ para todos } a, b, c \in R$$

El anillo es *conmutativo* si  $a \cdot b = b \cdot a$  para todos  $a, b \in R$ .

$(R, +)$  se dice ser el *grupo aditivo* del anillo en tanto que  $(R, \cdot)$  es su *semigrupo multiplicativo*.

**Ejemplo 2.1.8**  $\mathbb{Z}_n$  con la suma y multiplicación módulo  $n$  es un anillo conmutativo.

**Definición 2.1.11** Un subconjunto  $S$  de un anillo  $R$  es llamado un *subanillo* de  $R$  si  $S$  es cerrado bajo  $+$  y  $\cdot$ , y forma un anillo con ambas operaciones.

**Definición 2.1.12** Un subconjunto  $J$  de un anillo  $R$  es llamado un *ideal* si  $J$  es un subanillo de  $R$  y para toda  $a \in J$  y  $r \in R$  se tiene que  $ar \in J$  y  $ra \in J$ .

Un ideal es entonces un conjunto cerrado bajo la suma y que absorbe productos de un elemento en él por cualquier otro elemento en el anillo.

**Ejemplo 2.1.9** Sea  $R$  el campo  $\mathbb{Q}$  de los números racionales.

1. Entonces el conjunto  $\mathbb{Z}$  de los enteros es un subanillo de  $\mathbb{Q}$ , pero no es un ideal, ya que, por ejemplo,  $1 \in \mathbb{Z}$ ,  $\frac{1}{2} \in \mathbb{Q}$ , pero  $\frac{1}{2} \cdot 1 = \frac{1}{2} \notin \mathbb{Z}$ .
2. Sea  $R$  un anillo conmutativo,  $a \in R$ , y sea  $J = \{ra : r \in R\}$ , entonces  $J$  es un ideal.
3. Sea  $R$  un anillo conmutativo. Entonces el ideal más pequeño que contiene a un elemento dado  $a \in R$  es  $(a) = \{ra : r \in R\}$ .

**Definición 2.1.13** Sea  $R$  un anillo conmutativo. Un ideal  $J$  de  $R$  se llama *principal* si existe un elemento  $a \in R$  tal que  $J = (a)$ . En este caso,  $J$  es llamado el *ideal principal generado por  $a$* .

**Definición 2.1.14** Un elemento  $a$  de un anillo  $R$  es llamado *elemento invertible* si existe un  $b \in R$  tal que  $a \cdot b = 1$ .

**Ejemplo 2.1.10** En  $\mathbb{Z}_n$  existe el inverso multiplicativo de  $a \neq 0$  siempre que  $\text{m.c.d.}(a, n) = 1$ .

**Definición 2.1.15** Si  $R$  es un anillo arbitrario y existe un entero estrictamente positivo  $n$  tal que  $nr = 0$  para cualquier  $r \in R$ , entonces  $n$  es llamado la *característica (positiva)* del anillo  $R$ . Si tal entero positivo  $n$  no existiese, se dice que  $R$  tiene característica 0.

**Definición 2.1.16** Un *campo* es un anillo conmutativo en el cual todos los elementos no nulos tienen inversos multiplicativos.



**Ejemplo 2.1.11** Si  $p$  es un primo,  $\mathbb{Z}_p$  es un campo, según el ejemplo 2.1.10.

**Teorema 2.1.2**  $\mathbb{Z}/(p)$ , el anillo de las clases de residuos de los enteros módulo el ideal principal generado por un primo  $p$ , es un campo.

**Ejemplo 2.1.12** Sea  $p = 3$ . Entonces  $\mathbb{Z}/(p)$  está formado por los elementos  $[0],[1],[2]$ . Las operaciones en este campo se describen en las siguientes tablas de operación:

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

·	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Operaciones en el campo  $\mathbb{Z}/(p)$

**Definición 2.1.17** Un subconjunto  $F$  de un campo  $E$  es un *subcampo* de  $E$  si  $F$  es en sí mismo un campo con respecto a las operaciones de  $E$ . Si éste es el caso, se dice que  $E$  es una *extensión* de  $F$ .

**Definición 2.1.18** Un *campo finito* es un campo  $F$  el cual contiene un número finito de elementos. El *orden* de  $F$  es el número de elementos de  $F$ .

**Ejemplo 2.1.13** Si  $p$  es primo,  $\mathbb{Z}_p$  es un campo finito de orden  $p$ .

**Teorema 2.1.3 (Existencia y unicidad de los campos finitos)**

1. Si  $F$  es un campo finito, entonces  $F$  contiene  $p^m$  elementos para algún número primo  $p$  y un entero  $m \geq 1$ .
2. Para todo número de la forma  $p^m$ , donde  $p$  es primo y  $m \geq 1$ , existe un único campo finito de orden  $p^m$ . Este campo es denotado por  $\mathbb{F}_{p^m}$ , o bien  $\text{GF}(p^m)$  (las siglas  $\text{GF}$  están por la forma inglesa Galois field).

Por esta última condición, dos campos del mismo orden son *isomorfos*, es decir, poseen la misma estructura, aunque la representación de sus elementos puede ser diferente.

Note que si  $p$  es un número primo, entonces  $\mathbb{Z}_p$  es un campo, y por lo tanto todo campo de orden  $p$  es isomorfo a  $\mathbb{Z}_p$ . Naturalmente, el campo finito  $\mathbb{F}_p$  será identificado con  $\mathbb{Z}_p$ .

$\mathbb{F}_{p^m}$  contiene una copia de  $\mathbb{Z}_p$  como un subcampo y  $\mathbb{F}_{p^m}$  es una extensión de  $\mathbb{Z}_p$  de grado  $m$ .

**Definición 2.1.19** Los elementos no nulos de  $\mathbb{F}_q$  forman un grupo bajo la multiplicación llamado el *grupo multiplicativo* de  $\mathbb{F}_q$ , denotado por  $\mathbb{F}_q^*$ .

**Teorema 2.1.4** *Todo subgrupo del grupo multiplicativo  $\mathbb{F}_q^*$  es un grupo cíclico. por tanto  $\mathbb{F}_q^*$  mismo es cíclico de orden  $q - 1$ . Luego,  $a^q = a$  para todo  $a \in \mathbb{F}_q^*$ .*

Si  $q = p^m$ , para alguna  $m$ , entonces  $\mathbb{F}_q$  es de característica  $p$ :  $p \cdot 1 = 0$  en  $\mathbb{F}_q$  y, en consecuencia, para todo  $x \in \mathbb{F}_q$ ,  $p \cdot x = 0$ . Así, se tiene que la *transformación de Frobenius*  $x \mapsto x^p$  es una función inyectiva que preserva las operaciones de campo:  $(x \cdot y)^p = x^p \cdot y^p$ ,  $(x + y)^p = x^p + y^p$ ; la cual, por el Teorema Pequeño de Fermat, deja fijos a los elementos del subcampo primo  $\mathbb{F}_p$ . En otras palabras, se tiene que la transformación de Frobenius es un automorfismo en  $\mathbb{F}_q$  que deja fijo al subcampo primo. Se puede ver que cualquier otro automorfismo que deje fijo al campo primo es una iteración de la transformación de Frobenius. Por esto se tiene que el llamado *Grupo de Galois* de  $\mathbb{F}_q$  sobre  $\mathbb{F}_p$  es cíclico, posee orden  $m$ , y está generado por la transformación de Frobenius. El Teorema Fundamental de Galois implica entonces que cada subcampo de  $\mathbb{F}_q$  corresponde a un subgrupo de ese grupo de Galois.

**Teorema 2.1.5 (Subcampos de un campo finito)** *Sea  $\mathbb{F}_q$  un campo finito de orden  $q = p^m$ . Entonces todo subcampo de  $\mathbb{F}_q$  tiene orden  $p^n$ , para algún  $n \leq m$  que sea divisor de  $m$ . Viceversa, si  $n$  divide a  $m$ , entonces existe un único subcampo de  $\mathbb{F}_q$  de orden  $p^n$ .*

### 2.1.3. Polinomios

**Definición 2.1.20** Si  $R$  es un anillo conmutativo, el *anillo de polinomios*  $R[x]$  consta de los polinomios en la variable  $x$  con coeficientes en  $R$ . Las dos operaciones en él son la suma y la multiplicación usuales de polinomios, con coeficientes operando según las reglas aritméticas del anillo  $R$ .

**Ejemplo 2.1.14**  $f(x) = x^3 + x + 1$  y  $g(x) = x^2 + x$  son elementos del anillo de polinomios  $\mathbb{Z}_2[x]$ , y las operaciones son las de los enteros módulo 2.  $f(x) + g(x) = x^3 + x^2 + 1$  y  $f(x)g(x) = x^5 + x^4 + x^3 + x$ .

**Definición 2.1.21** Sea  $f(x) = \sum_{i=0}^n a_i x^i$  un polinomio no-nulo sobre  $R$ , así que se puede suponer  $a_n \neq 0$ . El elemento  $a_n$  es llamado el *coeficiente principal* o *más significativo* de  $f(x)$ , y  $a_0$  el *término constante*, en tanto que  $n$  es llamado el *grado* de  $f(x)$ . Escribimos  $n = \text{grad}(f(x)) = \text{grad}(f)$ . Se conviene en definir  $\text{grad}(0) = -\infty$ . Los polinomios de grado 0 son *constantes*. Si el coeficiente principal de  $f(x)$  es 1, entonces  $f(x)$  se dice ser un *polinomio mónico*.

En lo sucesivo,  $F$  será un campo.

**Ejemplo 2.1.15** El grado de  $f(x) = 2x^4 + x^2 + 2x + 1$  es 4.

**Ejemplo 2.1.16**  $f(x) = x^3 + x + 1$  es un polinomio mónico.

**Definición 2.1.22** Sea  $f(x) \in F[x]$  un polinomio de grado al menos 1. Entonces se dice que  $f(x)$  es *irreducible* sobre  $F$  si éste no puede ser representado como el producto de dos polinomios en  $F[x]$ , cada uno de grado positivo.

**Ejemplo 2.1.17**  $f(x) = x^2 + x + 1$  es irreducible sobre  $\mathbb{Z}_2$ . Supongamos que se tuviese  $f(x) = (a_1x + a_0)(b_1x + b_0)$ , entonces  $a_1b_1 = 1$ ,  $a_1b_0 + a_0b_1 = 1$ ,  $a_0b_0 = 1$ . Las ecuaciones primera y tercera implican que  $a_1 = b_1 = 1$  y  $a_0 = b_0 = 1$ , y esto contradice a la segunda. Así pues el ejemplo es verdadero.

**Teorema 2.1.6** *Un elemento  $b \in F$  es una raíz del polinomio  $f \in F[x]$ , es decir,  $f(b) = 0$ , si y sólo si  $x - b$  divide a  $f(x)$ .*

Por tanto, los polinomios irreducibles no pueden tener raíces en el campo.

**Definición 2.1.23** Dos polinomios  $h(x), g(x)$  son *congruentes* módulo un polinomio  $f(x)$  si la diferencia  $h(x) - g(x)$  es divisible entre  $f(x)$ , es decir, si para algún  $k(x) \in F[x]$ ,  $h(x) = g(x) + k(x)f(x)$ , y se escribe  $h(x) \equiv g(x) \pmod{f(x)}$ . Esta noción determina una relación de equivalencia en el anillo de polinomios. Su espacio cociente se denota como  $F[x]/(f(x))$ .

**Teorema 2.1.7**  *$F[x]/(f(x))$  se identifica con el conjunto de polinomios en  $F[x]$  de grado menor que  $n = \text{grad } f(x)$ . La suma y la multiplicación en  $F[x]/(f(x))$  se realizan módulo  $f(x)$ .*

**Teorema 2.1.8**  *$F[x]/(f(x))$  es un anillo conmutativo.*

**Teorema 2.1.9** *Si  $f(x)$  es irreducible sobre  $\mathbb{F}_p$ , entonces  $F[x]/(f(x))$  es un campo.*

Esta es una consecuencia directa del algoritmo de la división de Euclides: Si  $g(x) \in F[x]/(f(x))$  no es nulo, entonces  $g(x)$  y  $f(x)$  no tienen divisores comunes, luego existen  $h(x), i(x)$  tales que  $1 = h(x) \cdot g(x) + f(x) \cdot i(x)$ .  $h(x)$  es pues el inverso multiplicativo de  $g(x)$ . Así pues todo elemento no nulo en  $F[x]/(f(x))$  posee un inverso.

**Ejemplo 2.1.18**  $f(x) = x^2 + x + 1$  es irreducible sobre  $\mathbb{Z}_2$ . Luego  $\mathbb{Z}_2[x]/(f(x))$  es un campo, donde las operaciones son módulo  $f(x)$ .

Una representación muy utilizada para los elementos de un campo finito  $\mathbb{F}_q$ , con  $q = p^m$  y  $p$  un número primo, es mediante *una base polinomial*.

Primero, si  $m = 1$ , entonces  $\mathbb{F}_q$  es simplemente  $\mathbb{Z}_p$  y la aritmética es realizada módulo  $p$ . Luego, se tiene el siguiente:

**Teorema 2.1.10** Para cada  $m \geq 1$ , existe un polinomio mónico irreducible de grado  $m$  sobre  $\mathbb{F}_p$ , tal que  $\mathbb{F}_q$  es isomorfo a  $\mathbb{F}_p[x]/(f(x))$ . Por lo tanto, todo campo finito tiene una representación de base polinomial.

Así, los elementos de un campo finito  $\mathbb{F}_{p^m}$  serán representados por polinomios en  $\mathbb{Z}_p[x]$  de grado  $< m$ , y con coeficientes en  $\mathbb{Z}_p$ .

**Ejemplo 2.1.19** El campo finito  $\mathbb{F}_{2^4}$  se puede representar como el conjunto de polinomios sobre  $\mathbb{F}_2$  de grado  $< 4$ . Esto es,

$$\mathbb{F}_{2^4} = \{a_3x^3 + a_2x^2 + a_1x + a_0 \mid a_i \in \{0, 1\}\}.$$

Por conveniencia, cada polinomio  $a_3x^3 + a_2x^2 + a_1x + a_0$  se representa por el correspondiente vector de coeficientes  $(a_3, a_2, a_1, a_0)$  de dimensión 4, y así

$$\mathbb{F}_{2^4} = \{(a_3, a_2, a_1, a_0) \mid a_i \in \{0, 1\}\},$$

lo cual es muy adecuado para para propósitos computacionales. Ya que el polinomio  $f(x) = x^4 + x + 1$  es irreducible sobre  $\mathbb{Z}_2$ , las operaciones en el campo se hacen módulo  $f(x)$ .

**Definición 2.1.24** Un polinomio irreducible  $f(x) \in \mathbb{Z}_p[x]$  de grado  $m$  es llamado un *polinomio primitivo* si  $x$  es un generador de  $\mathbb{F}_{p^m}^*$ .

**Ejemplo 2.1.20** Mediante el polinomio irreducible  $f(x) = x^4 + x + 1$ , se tiene que el polinomio  $p(x) = x$  es un polinomio primitivo, es decir, el elemento  $\alpha = (0, 0, 1, 0)$  del campo es un generador de  $\mathbb{F}_{2^4}^*$ . Así pues

$$\mathbb{F}_{2^4}^* = \{x^i \bmod f(x) \mid 0 \leq i < 2^4 - 1\}.$$

**Definición 2.1.25** El campo de polinomios sobre  $GF(p)$  módulo un polinomio irreducible de grado  $m$ , es llamado el *Campo de Galois* de  $p^m$  elementos, o  $GF(p^m)$ . Para algún número  $q = p^m$ , que es una potencia de un primo, hay un campo  $GF(p)$  con  $q$  elementos.

**Teorema 2.1.11** *En  $GF(p)$  hay un elemento primitivo  $\alpha$ , esto es, un elemento de orden  $q-1$ . Todo elemento de  $GF(p)$  que es diferente de cero, puede ser expresado como una potencia de  $\alpha$ , es decir, el grupo multiplicativo de  $GF(q)$  es cíclico.*

### 2.1.4. Bases de campos finitos

Existen diferentes tipos de bases para representar a los elementos de un campo finito o de Galois. En esta sección presentamos tres tipos de bases que son ampliamente utilizadas para el diseño de operadores aritméticos sobre campos finitos, a saber bases polinomiales o estándares, bases normales y bases duales.

#### Bases polinomiales

**Definición 2.1.26** Sea  $f(x)$  un polinomio irreducible sobre  $\mathbb{F}_p$  de grado  $m > 1$ . Entonces  $\mathbb{F}_q$  es isomorfo a  $\mathbb{F}_p[x]/(f(x))$ , con  $q = p^m$ .

Sea  $\alpha$  un generador del grupo cíclico  $\mathbb{F}_q^*$ . Entonces  $\{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$  es una base algebraica y se dice ser la *base estándar o polinomial o canónica*.

Una tal base se relaciona directamente con la representación de los elementos del campo como polinomios. En este caso, un elemento  $A$  del campo se representa mediante el polinomio  $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{m-1}x^{m-1}$ . Así cada polinomio representa de hecho una clase de residuos módulo  $f(x)$ . Ya que  $\alpha$  es una raíz de  $f(x)$  en  $\mathbb{F}_q$ , la representación polinomial de  $A(x)$  es equivalente a  $A(\alpha) = a_0 + a_1\alpha + a_2\alpha^2 + \dots + a_{m-1}\alpha^{m-1}$ .

#### Bases normales

**Definición 2.1.27** El conjunto  $\{\alpha, \alpha^q, \alpha^{q^2}, \dots, \alpha^{q^{m-1}}\}$ , donde  $\alpha$  es un generador de  $\mathbb{F}_q^*$ , es llamado *base normal* siempre que los  $m$  elementos sean linealmente independientes.

Es bien sabido que existen bases normales para todos los campos de Galois. La representación en una tal base es especialmente atractiva para ciertas aplicaciones tales como la elevación al cuadrado, pues ella consiste de un simple corrimiento cíclico de los coeficientes del polinomio que representa al elemento cuyo cuadrado se busca calcular.

### Bases duales

**Definición 2.1.28** Para  $\alpha \in F = \mathbb{F}_{q^m}$  y  $K = \mathbb{F}_q$  la *traza*  $\text{Tr}_{F/K}(\alpha)$  de  $\alpha$  sobre  $K$  está definida como:

$$\text{Tr}_{F/K}(\alpha) = \alpha + \alpha^q + \cdots + \alpha^{q^{m-1}}.$$

En otras palabras, la traza de  $\alpha$  sobre  $K$  es la suma de las conjugadas de  $\alpha$  con respecto a  $K$ .

**Definición 2.1.29** Sea  $B = \{\beta_0, \beta_1, \dots, \beta_{m-1}\}$  una base de  $\mathbb{F}_q$  o  $\mathbb{F}_{p^m}$ . Una *base dual* de  $B$  es una base  $C = \{\gamma_0, \gamma_1, \dots, \gamma_{m-1}\}$  que satisfaga:

$$\text{Tr}(\beta_i \gamma_j) = \begin{cases} 1, & \text{si } i = j, \\ 0, & \text{si } i \neq j. \end{cases}$$

Es bien sabido que se puede construir una base dual para cada base dada.

## 2.2. Multiplicación en $GF(2^m)$

### 2.2.1. Consideraciones previas

Con base en el contenido de la sección anterior, hacemos las siguientes consideraciones:

Sea  $p$  un número primo. Es bien sabido que el conjunto de residuos módulo  $p$ ,  $GF(p) = \{0, 1, \dots, p-1\}$  es un campo, llamado *primo*. Todo campo finito posee un elemento cero y un elemento unidad y es una extensión de grado finito de un

campo primo. Cualesquiera dos campos finitos de una misma cardinalidad son isomorfos, y el grupo multiplicativo de todo campo finito es cíclico. Los campos finitos o de Galois son de la forma  $GF(p^m)$ , donde  $p$  es primo,  $m$  es un entero positivo.  $GF(p^m)$  contiene  $p^m$  elementos y puede representarse como el cociente del anillo de polinomios con coeficientes en  $GF(p)$ , reducido módulo un polinomio irreducible  $P(x) = x^m + Q(x) = x^m + \sum_{i=0}^{m-1} q_i x^i$  de grado  $m$  sobre  $GF(p)$ , por lo cual las operaciones del campo  $GF(p^m)$  pueden realizarse como operaciones entre polinomios módulo la división por el polinomio irreducible. Se tiene que al hacer la reducción modular, el polinomio irreducible se está identificando con el polinomio cero.

En particular, para  $p = 2$ ,  $GF(2^m)$  contiene  $2^m$  elementos y su campo primo es  $GF(2) = \{0, 1\}$ . Su grupo multiplicativo posee  $2^m - 1$  elementos, a saber todos aquellos excepto el cero. Al ser ése un grupo cíclico, existe un elemento  $\alpha$ , llamado *primitivo*, tal que todos los elementos de  $GF(2^m) - \{0\}$  pueden ser representados como potencias de él. Se tendrá

$$GF(2^m) = \{0\} \cup \{\alpha^1, \alpha^2, \dots, \alpha^{2^m-2}, \alpha^{2^m-1} = 1\} \quad (2.1)$$

Ya que  $P(\alpha) = 0$ , se tiene que:

$$\alpha^m = Q(\alpha) = q_{m-1}\alpha^{m-1} + q_{m-2}\alpha^{m-2} + \dots + q_1\alpha + q_0 \quad (2.2)$$

La ecuación 2.2 se utiliza también para representar a cualquier elemento del campo como una combinación lineal única de las potencias  $1, \alpha, \alpha^2, \dots, \alpha^{m-1}$ , es decir, cualquier elemento de  $GF(2^m)$  se representa mediante una forma polinomial en  $\alpha$  de grado menor que  $m$  y coeficientes en  $GF(2)$ . En otras palabras, cualquier elemento  $A \in GF(2^m)$  se representa de la forma:

$$A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i \quad (2.3)$$

donde  $a_i \in GF(2)$  para cada  $0 \leq i \leq m - 1$ . La colección  $\{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$  se dice ser la *base estándar*, llamada también *base polinomial* o *canónica* correspon-



diente al elemento primitivo  $\alpha$ . Diremos que la representación de la forma 2.3 es la representación *polinomial* del elemento representado.

Al implementar las operaciones en hardware, la representación polinomial de cada elemento puede a su vez ser representada como una  $m$ -tupla binaria, mediante su arreglo de coeficientes  $(a_{m-1}, a_{m-2}, \dots, a_0)$ , y decimos que esta última representación es *vectorial*. Como  $p = 2$ , los dos elementos del campo pueden ser representados convenientemente por las señales lógicas “0” y “1”. La operación de adición sobre  $GF(2)$  equivale a una operación lógica XOR, y el producto a una operación lógica AND.

Llamamos la atención del lector en la notación que estamos utilizando: un polinomio  $A(x)$  con coeficientes en  $GF(2)$ , visto como tal lo expresaremos utilizando la variable  $x$ , y al evaluarlo en  $\alpha$  obtenemos un elemento  $A(\alpha)$ . Así pues, al escribir que  $A(\alpha)$  es un elemento, nos referimos a que se está representando en la base estándar mediante el polinomio  $A(x)$ .

La operación más simple sobre campos finitos  $GF(2^m)$  es la adición de dos elementos del campo, y consiste en una operación XOR bit a bit entre ellos. La resta, es decir, la adición del minuendo con el inverso aditivo del sustraendo, puede ser realizada de la misma manera ya que cada elemento del campo es su propio inverso aditivo. Así pues, la adición de dos elementos se expresa de la manera siguiente:

$$C(\alpha) = A(\alpha) + B(\alpha) = \sum_{i=0}^{m-1} (a_i + b_i) \alpha^i = \sum_{i=0}^{m-1} c_i \alpha^i \quad (2.4)$$

En el cuadro 2.1 mostramos un pseudocódigo correspondiente a esta operación. Siendo una operación bit a bit en paralelo, su complejidad en tiempo es de  $O(1)$  ciclos de reloj. En términos de hardware, se necesitan  $m$  unidades XOR en paralelo (representadas por  $\oplus$ ), lo que determina la complejidad de espacio en  $O(m)$  compuertas (recordemos que  $m$  es el orden del campo de Galois). Cada unidad realiza el cálculo de  $(a_i \oplus b_i)$  como se muestra en la figura 2.1.

La multiplicación de dos elementos  $A(\alpha)$  y  $B(\alpha)$  en el campo finito  $GF(2^m)$  es

**Algoritmo Adición**

Input:  $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$ ,  $B(\alpha) = \sum_{i=0}^{m-1} b_i \alpha^i \in GF(2^m)$

Output:  $A(\alpha) + B(\alpha) = C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i \in GF(2^m)$

1. For  $j = 0$  to  $m - 1$

1.1  $c_i = (a_i + b_i) \bmod 2$

Cuadro 2.1: Algoritmo para la adición de dos elementos en el campo de Galois.

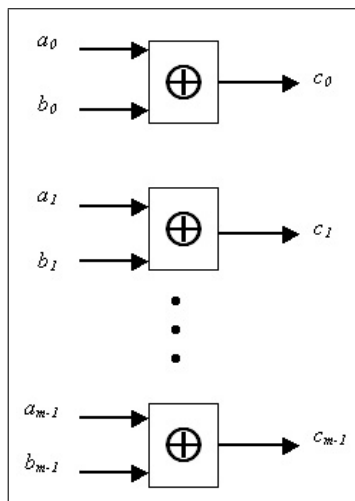


Figura 2.1: Adición en  $GF(2^m)$

una operación más compleja que la adición y requiere de dos pasos para su cálculo: una multiplicación polinomial, cuyo resultado será un polinomio de orden  $2m - 2$ , y una operación de reducción módulo el polinomio irreducible, que resultará en un polinomio de orden  $m - 1$ . Al multiplicar los polinomios  $A(x)$  y  $B(x)$  se tiene que

$$C(x) = A(x)B(x) = \left( \sum_{i=0}^{m-1} a_i x^i \right) \left( \sum_{i=0}^{m-1} b_i x^i \right) \quad (2.5)$$

y al tomar el residuo módulo el polinomio irreducible  $P(x)$ , es decir

$$C'(x) = C(x) \bmod P(x) \quad (2.6)$$

se obtendrá el resultado de la multiplicación como  $C'(\alpha)$ .

Así pues, hay dos formas de calcular la multiplicación: la primera, digamos el *método clásico*, consiste en realizar primero la multiplicación polinomial clásica, y después hacer la operación de reducción a la suma de los productos parciales. En una implementación en hardware esto implica la construcción de dos estructuras diferentes para cada una de estas dos operaciones, restando modularidad y simplicidad al circuito. La segunda, que realiza la reducción a cada uno de los productos parciales, es conocida como el *método de "shift-add"* o *entrelazado*, y es, entre los dos métodos, el más utilizado para implementaciones en hardware debido a que permite el desarrollo de estructuras más simples y regulares. Una discusión sobre ambos métodos se puede consultar en [12].

Existen en la literatura diversas propuestas de algoritmos para realizar la multiplicación  $GF(2^m)$  por hardware. Son dos las características principales que definen a los multiplicadores para campos finitos:

- *La base* en que son representados los elementos del campo finito, que puede ser polinomial, normal o dual, y
- *El tipo de arquitectura*, que puede ser paralela, serial o por dígitos.

Considerando estas características hemos ordenado la revisión de los multiplicadores como sigue.

### 2.2.2. Multiplicación en paralelo

Los multiplicadores paralelos requieren sólo de un pulso de reloj para realizar la multiplicación, y en general, de un número cuadrático de compuertas para su implementación, es decir, tienen complejidades  $O(1)$  y  $O(m^2)$  en tiempo y espacio respectivamente, donde  $m$  es el orden del campo. En 1962, Karatsuba y Ofman [37], introdujeron un algoritmo para la multiplicación en paralelo que mejoraba la complejidad de espacio mencionada. Este multiplicador presenta una complejidad de espacio de  $O(n^{\log_2 3})$  y ha sido estudiado por Knuth en [39] y aplicado por Paar [62] en la construcción de un multiplicador paralelo sobre el campo  $GF((2^n)^m)$ . Así pues, éste ha sido de gran relevancia para implementar las operaciones en campos llamados “compuestos”. Veamos con detalle algunos algoritmos representativos de este tipo de arquitectura, los cuales han utilizado bases diversas en campos finitos.

#### Multiplicador de Mastrovito de base polinomial

En este apartado, consideramos una arquitectura para la multiplicación de dos elementos en el campo  $GF(2^m)$  introducida por Mastrovito en [51] y [52], que tiene una de las mejores medidas de complejidad de espacio en su tipo, por lo que la hemos considerado como representativa de los multiplicadores paralelos en una base polinomial.

Supongamos que se ha de calcular  $C(x) = A(x)B(x) \bmod P(x)$ , donde  $A(x)$  y  $B(x)$  son polinomios de grado menor que  $m$ , con coeficientes binarios, y  $P(x)$  es un polinomio irreducible de grado  $m$ . Este producto, representa propiamente la multiplicación de dos elementos en el campo  $GF(2^m)$ , expresados en la base polinomial. Escribamos:

$$c_{m-1}x^{m-1} + \cdots + c_0 = (a_{m-1}x^{m-1} + \cdots + a_0)(b_{m-1}x^{m-1} + \cdots + b_0) \bmod P(x)$$

Los polinomios  $A(x)$ ,  $B(x)$  y  $C(x)$  pueden ser representados mediante los vectores columna cuyas entradas son sus coeficientes. Utilizando propiedades algebraicas

elementales (por ejemplo, que el tomar módulo es congruente con la suma y el producto de coeficientes) se puede escribir

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \mathcal{C} = \mathcal{FB} = \begin{pmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,m-1} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ f_{m-1,0} & f_{m-1,1} & \cdots & f_{m-2,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix} \quad (2.7)$$

donde la matriz  $\mathcal{F} = f(A(x), P(x))$  es una matriz que sólo depende de  $A(x)$  y de  $P(x)$  la cual es llamada “matriz del producto”. Sea  $\mathcal{Q} = (q_{ij})$  la matriz tal que

$$\begin{pmatrix} x^m \\ x^{m+1} \\ \vdots \\ x^{2m-2} \end{pmatrix} \equiv \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,m-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ q_{m-2,0} & q_{m-2,1} & \cdots & q_{m-2,m-1} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ \vdots \\ x_{m-1} \end{pmatrix} \pmod{P(x)} \quad (2.8)$$

es decir,  $\mathcal{Q}$  representa a los polinomios  $x^m, x^{m+1}, \dots, x^{2m-2} \pmod{P(x)}$ , en la base polinomial, es decir, luego de que se ha realizado la operación de reducción módulo  $P(x)$ . Entonces las entradas  $f_{ij} \in GF(2)$ , con  $0 \leq i, j \leq m-1$ , de la matriz  $\mathcal{F}$  quedan determinados como sigue:

$$f_{ij} = \begin{cases} a_i & , \text{ si } j = 0 \\ u(i-j)a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i} a_{m-1-t} & , \text{ si } j \geq 1 \end{cases}$$

donde la función  $u$  está definida como

$$u(\mu) = \begin{cases} 1 & \mu \geq 0 \\ 0 & \mu < 0. \end{cases}$$

La complejidad computacional del producto matriz-vector de la ecuación (2.7) depende sustancialmente del polinomio primitivo  $P(x)$ . Los polinomios son óptimos con respecto al número de compuertas requeridas para calcular la multiplicación en el campo finito.

Para campos en los que existen polinomios primitivos de la forma

$$P(x) = x^m + x + 1,$$

la complejidad de espacio está dada por:

$$\#AND=m^2$$

$$\#XOR=m^2 - 1$$

donde  $\#AND$  y  $\#XOR$  representan los números de compuertas AND y XOR respectivamente, utilizadas en la implementación. Tales trinomios primitivos existen para  $m = 2, 3, 4, 6, 7, 9, 10, 11, 15$ . Para otros valores de  $m$  en los que no haya trinomios primitivos, la complejidad de espacio será mayor.

La complejidad de tiempo de este multiplicador es:

$$T = T_{AND} + T_{XOR} \leq 1 + 2\lceil \log_2 n \rceil$$

Un estudio mas amplio de este multiplicador se puede consultar en [62] y [75].

### Multiplicador de Massey-Omura de base normal

En [55], Massey y Omura desarrollaron un método para el cálculo de la multiplicación en campos finitos utilizando una base normal de la forma

$$N = \{\alpha, \alpha^2, \alpha^{2^2}, \dots, \alpha^{2^{m-1}}\}.$$

Aunque la descripción original se orienta a la implementación de un multiplicador serial, se puede derivar una estructura paralela directamente, tal como se ha reportado en [92].

Consideremos dos elementos  $A, B$  representados en una base normal:

$$A = a_0\alpha + a_1\alpha^2 + a_2\alpha^{2^2} + \dots + a_{m-1}\alpha^{2^{m-1}}$$

$$B = b_0\alpha + b_1\alpha^2 + b_2\alpha^{2^2} + \dots + b_{m-1}\alpha^{2^{m-1}}$$

Una propiedad de la representación con bases normales es que la elevación al cuadrado de un elemento del campo consiste sólo de un corrimiento cíclico de sus coeficientes, como se indica enseguida:

$$\begin{aligned} A^2 &= a_{m-1}\alpha + a_0\alpha^2 + a_1\alpha^{2^2} + \dots + a_{m-2}\alpha^{2^{m-1}} \\ B^2 &= b_{m-1}\alpha + b_0\alpha^2 + b_1\alpha^{2^2} + \dots + b_{m-2}\alpha^{2^{m-1}} \end{aligned}$$

Expresemos a la multiplicación  $C = A \cdot B$  de esos dos elementos como:

$$C = c_0\alpha + c_1\alpha^2 + c_2\alpha^{2^2} + \dots + c_{m-1}\alpha^{2^{m-1}} \quad (2.9)$$

Al realizar todo el cálculo simbólico involucrado, se tiene que el coeficiente más alto  $c_{m-1}$ , ha de ser una función que depende de todos los coeficientes de entrada  $a_i, b_i$ :

$$c_{m-1} = f(a_0, a_1, \dots, a_{m-1}; b_0, b_1, \dots, b_{m-1}) \quad (2.10)$$

$f$  es una función  $\{0, 1\}^m \times \{0, 1\}^m \rightarrow \{0, 1\}$ .

Al calcular el cuadrado  $C^2 = A^2 \cdot B^2$  se obtiene, por tenerse una base normal,

$$C^2 = c_{m-1}\alpha + c_0\alpha^2 + c_1\alpha^{2^2} + \dots + c_{m-2}\alpha^{2^{m-1}}. \quad (2.11)$$

Así resulta una expresión similar a 2.10 para el coeficiente  $c_{m-2}$ :

$$c_{m-2} = f(a_{m-1}, a_0, \dots, a_{m-2}; b_{m-1}, b_0, \dots, b_{m-2}) \quad (2.12)$$

La función en 2.12 es la misma que la de 2.10, sólo que evaluada en los coeficientes de entrada  $(a_0, a_1, \dots, a_{m-1})$  y  $(b_0, b_1, \dots, b_{m-1})$  recorridos cíclicamente. Los otros coeficientes  $(c_{m-3}, c_{m-4}, \dots, c_0)$  pueden también ser obtenidos de la misma forma, es decir, por medio de un corrimiento cíclico de los valores de entrada.

Se podrá observar que la multiplicación en una base normal para un cierto orden del campo utilizado, está entonces determinada por la función  $f$ . Así pues, la complejidad de  $f$  determina a su vez la complejidad del multiplicador.

Tomemos como una medida de complejidad de  $f$  al número  $C_m$  de productos realizados en  $GF(2)$  para calcularla. Mullin ha mostrado en [59] que esa complejidad está acotada inferiormente por  $2m - 1$ , es decir  $C_m \geq 2m - 1$ .

Una base normal respecto a la cual se tiene que  $C_m = 2m - 1$  es llamada una *base normal óptima*. La complejidad espacial para una implementación en hardware de  $f$  es la de la suma de  $C_m$  compuertas AND y de  $C_m - 1$  compuertas XOR. La cuenta total de compuertas para una implementación en hardware de un multiplicador en una base normal, con  $m$  elementos, está entonces minorizada como sigue:

$$\begin{aligned}\#\text{AND} &= mC_m \geq 2m^2 - m, \\ \#\text{XOR} &= (m - 1)C_m \geq 2m^2 - 3m + 1.\end{aligned}$$

Aunque esta complejidad es aproximadamente el doble de la del multiplicador de Mastrovito para una base polinomial, las arquitecturas para bases normales son regularmente utilizadas en aplicaciones que requieren de repetidas elevaciones al cuadrado, tal como la exponenciación, ya que el corrimiento cíclico es de bajo costo en requerimientos de hardware.

### Multiplicador de Berlekamp de base dual

En [5] Berlekamp describe la implementación de un codificador de Reed-Solomon, y propone un algoritmo para realizar la multiplicación en  $GF(2^m)$ . Este multiplicador usa una representación de base dual para uno de los operandos, mientras que el otro operando es representado en una base polinomial.

Recordemos la definición de una base dual: sea  $B_e = \{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$  una base estándar o polinomial de  $GF(2^m)$ . Una *base dual* de  $B_e$  es una base  $B_d = \{\gamma_0, \gamma_1, \dots, \gamma_{m-1}\}$  que ha de satisfacer:

$$\text{Tr}(\alpha_i \gamma_j) = \begin{cases} 1, & \text{si } i = j, \\ 0, & \text{si } i \neq j. \end{cases} \quad (2.13)$$



donde  $Tr$  es la función *Traza* como se ha definido en la sección 2.1.4. Para este multiplicador, el primer operando  $A$  es representado en una base estándar:

$$A = a_0 + a_1\alpha + a_2\alpha^2 + \dots + a_{m-1}\alpha^{m-1},$$

y el segundo operando  $B$  en su correspondiente base dual:

$$B = b_0\gamma_0 + b_1\gamma_1 + b_2\gamma_2 + \dots + b_{m-1}\gamma_{m-1}.$$

Enseguida se estudia una fórmula para la multiplicación de  $B$  (representado en base dual) con un elemento  $\alpha$  de base polinomial. Considérese, para cada  $j = 0, 1, \dots, m-1$ :

$$Tr(\alpha^j B) = Tr(\alpha^j b_0\gamma_0 + \alpha^j b_1\gamma_1 + \alpha^j b_2\gamma_2 + \dots + \alpha^j b_{m-1}\gamma_{m-1}) = b_j \quad (2.14)$$

donde la definición de 2.13 ha sido usada. Si el elemento  $j$ -ésimo del producto  $\alpha B$  es denotado por  $(\alpha B)_j$ , se tiene que:

$$(\alpha B)_j = Tr(\alpha^j(\alpha B)) = Tr(\alpha^{j+1}B) = \begin{cases} b_{j+1} & , j = 0, 1, \dots, m-2 \\ Tr(\alpha^m B) & , j = m-1, \end{cases} \quad (2.15)$$

Aparentemente, todos los elementos  $(\alpha B)_j$  excepto el de mayor orden son obtenidos con un simple corrimiento de los elementos de  $B$ . El coeficiente  $(\alpha B)_{m-1}$  puede ser obtenido como sigue: Sea  $P(x) = 1 + p_1x + \dots + p_{m-1}x^{m-1} + x^m$  el polinomio primitivo tal que  $P(\alpha) = 0$ . Entonces

$$\alpha^m = 1 + p_1\alpha + \dots + p_{m-1}\alpha^{m-1}, \quad (2.16)$$

y esto puede ser usado para calcular

$$\begin{aligned} (\alpha B)_{m-1} &= Tr(\alpha^m B) = Tr((p_0 + p_1\alpha + \dots + p_{m-1}\alpha^{m-1})B) \\ &= b_0 + p_1b_1 + \dots + p_{m-1}b_{m-1} = P \cdot B \end{aligned} \quad (2.17)$$

El último término en la ecuación 2.17 representa el producto “punto” del elemento  $B$  y los coeficientes del polinomio primitivo.

Para estimar la complejidad de espacio de las implementaciones, vemos que el número de compuertas XOR es

$$C_1 = (h_w(P) - 2) \geq 1, \quad (2.18)$$

donde  $h_w(P)$  representa el peso del polinomio primitivo, es decir, el número de sus coeficientes con valor de “uno”.

Consideremos ahora el producto  $C = A \cdot B$ . El operando  $B$  y el producto  $C$  se representan en una base dual, mientras que el operando  $A$  es representado en una base polinomial. Iniciando desde la ecuación 2.14 se obtiene:

$$c_j = \text{Tr}(\alpha^j C) = \text{Tr}(\alpha^j AB) = \text{Tr}((\alpha^j B)A). \quad (2.19)$$

El primer coeficiente es entonces

$$\begin{aligned} c_0 &= \text{Tr}(BA) = \text{Tr}(a_0 B) + \text{Tr}(a_1 \alpha B) + \cdots + \text{Tr}(a_{m-1} \alpha^{m-1} B) \\ &= a_0 \text{Tr}(B) + a_1 \text{Tr}(\alpha B) + \cdots + a_{m-1} \text{Tr}(\alpha^{m-1} B) \\ &= a_0 b_0 + a_1 b_1 + \cdots + a_{m-1} b_{m-1} \\ &= A \cdot B \end{aligned} \quad (2.20)$$

el cual es el producto “punto” de los dos operandos. Similarmente, el coeficiente  $c_1$  es:

$$c_1 = \text{Tr}((\alpha B)A) = A \cdot (\alpha B).$$

El término  $(\alpha B)$  puede ser fácilmente calculado usando la ecuación 2.15 por medio de un corrimiento a la izquierda de los coeficientes y haciendo después el cálculo de 2.17. El mismo procedimiento se puede aplicar iterativamente para calcular los otros coeficientes:

$$\begin{aligned} c_2 &= \text{Tr}((\alpha^2 B)A) = A \cdot (\alpha(\alpha B)) \\ c_3 &= \text{Tr}((\alpha^3 B)A) = A \cdot (\alpha(\alpha(\alpha B))) \\ &\vdots \end{aligned}$$

Las fórmulas desarrolladas se pueden aplicar para una descripción matricial de un multiplicador paralelo. Para esto, cada elemento puede ser representado como un vector que contiene  $m$  elementos:

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & \dots & b_{m-2} & b_{m-1} \\ b_1 & b_2 & \dots & b_{m-1} & B \cdot P \\ b_2 & b_3 & \dots & B \cdot P & (\alpha B) \cdot P \\ \vdots & \vdots & & \vdots & \vdots \\ b_{m-1} & B \cdot P & \dots & (\alpha^{m-3} B) \cdot P & (\alpha^{m-2} B) \cdot P \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{pmatrix} \quad (2.21)$$

La complejidad de una implementación en hardware de un multiplicador de base dual se compone de la complejidad para calcular la multiplicación matriz-vector de 2.21 y de la complejidad para el cálculo de los productos  $(\alpha^j B)$ ,  $j = 0, 1, \dots, m-2$ . La complejidad de la ecuación 2.21 es

$$C_2 = m^2 \text{AND} + m(m-1) \text{XOR} = m^2 \text{AND} + (m^2 - m) \text{XOR}.$$

La complejidad  $C_1$  del cálculo de un producto "punto" ha sido indicada en la ecuación 2.18. Así pues, la complejidad total  $C = (m-1)C_1 + C_2$  es:

$$\# \text{AND} = m^2$$

$$\# \text{XOR} = (m^2 - m) + (m-1)(h_w(P) - 2) \geq m^2 - 1 \quad (2.22)$$

El número de compuertas AND es  $m^2$  independientemente del polinomio irreducible. El número de compuertas XOR es igualmente conocido si un *trinomio* irreducible es utilizado, ya que  $h_w(P) = 3$ . Como puede observarse, la complejidad de espacio de este multiplicador es la misma que el multiplicador de Mastrovito cuando es usado un *trinomio* como polinomio irreducible.

Otro aspecto que ha de considerarse, es la diferente representación de las entradas y la salida del multiplicador, es decir  $A$  en base estándar y  $B$  y  $C$  en base dual. Esto requiere de un proceso adicional para la transformación de bases, y ha de tomarse en cuenta para determinar la medida de complejidad. En [19] se ha

Multiplicador	Número de compuertas
Mastrovito	$2m^2 - 1$
Massey-Omura	$\geq 4m^2 - 4m + 1$
Belekamp	$\geq 2m^2 - 1$
Karatsuba	$\geq 7m^{\log_2 3} - 8m + 2$

Cuadro 2.2: Comparación de los multiplicadores paralelos  $GF(2^m)$ 

demostrado que si el polinomio irreducible es un *trinomio*, sólo se requiere de una serie de permutaciones de los coeficientes para realizar tal transformación.

En [75] se estudia un algoritmo para la multiplicación en el que tanto los operandos como el producto son representados en una base dual.

### Comparación de los multiplicadores paralelos

En la tabla 2.2 se presenta una comparación de los multiplicadores paralelos que se han revisado en este apartado. Se incluye el multiplicador de Karatsuba-Offman que se ha mencionado anteriormente.

Los multiplicadores paralelos tienen un alto desempeño y son la mejor opción para aplicaciones que requieran alta velocidad y el uso de órdenes pequeños del campo finito, tales como 4 y 8. Sin embargo, debido a que su complejidad de espacio es de orden cuadrático, resultan muy caros en requerimientos cuando son implementados en hardware, por lo que son poco utilizados en aplicaciones criptográficas, en las que los órdenes de los campos son 163, 512 o 1024.

### 2.2.3. Multiplicación serial

Los multiplicadores seriales procesan todos los coeficientes del multiplicando en paralelo en el primer paso, mientras los coeficientes del multiplicador son procesados serialmente. Para calcular una operación de multiplicación requieren de  $O(m)$

pulsos de reloj y utilizan  $O(m)$  compuertas para su construcción, lo que define sus complejidades de tiempo y espacio respectivamente. Como puede verse, no son tan rápidos como los paralelos, pero requieren de poco espacio físico para su implementación en hardware. Estos multiplicadores pueden ser diseñados en dos versiones diferentes, dependiendo del orden en que son tratados los coeficientes del multiplicador: *LSE-first* (*Least Significant Element-first*) y *MSE-first* (*Most Significant Element-first*).

Ya que la multiplicación por un elemento primitivo es una operación requerida para calcular serialmente el producto de dos elementos en el campo  $GF(2^m)$ , nos ocupamos enseguida de su estudio.

### Multiplicación por un elemento primitivo

Sea  $P(x) = p_0 + p_1x + p_2x^2 + \dots + p_{m-1}x^{m-1} + x^m$  un polinomio mónico irreducible sobre  $GF(2)$  de grado  $m$ , que se utilizará para representar a los elementos del campo  $GF(2^m)$ . Denotemos por  $\alpha$  a una raíz primitiva,  $P(\alpha) = 0$ . Sea  $A(\alpha)$  un elemento arbitrario de  $GF(2^m)$ . El producto  $D(\alpha) = \alpha A(\alpha)$  se expresa como:

$$D(\alpha) = a_0\alpha + a_1\alpha^2 + a_2\alpha^3 + \dots + a_{m-1}\alpha^m \quad (2.23)$$

Considerando el polinomio irreducible, se tiene

$$\alpha^m = [p_0 + p_1\alpha + p_2\alpha^2 + \dots + p_{m-1}\alpha^{m-1}] \text{ mod } P(\alpha) \quad (2.24)$$

sustituyendo la ecuación 2.24 en la ecuación 2.23 resulta

$$D(\alpha) = d_0 + d_1\alpha + \dots + d_{m-1}\alpha^{m-1}$$

donde

$$\begin{aligned} d_0 &= a_{m-1}p_0 && \text{y} \\ d_i &= a_{i-1} + a_{m-1}p_i, && \text{para } i = 1, 2, \dots, m-1 \end{aligned}$$

La complejidad de espacio para esta operación es

$$\#\text{AND} = m$$

$$\#\text{XOR} = m - 1 \quad (2.25)$$

El cálculo de esta operación se implementa mediante un LFSR (*Linear Feedback Shift Register*) como se estudia en la sección 4.3.2.

### Multiplicador *LSE-first*

Sean  $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$  y  $B(\alpha) = \sum_{i=0}^{m-1} b_i \alpha^i$  dos elementos del campo  $GF(2^m)$ , y sea  $C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i$  su producto. En este tipo de multiplicador, la multiplicación puede ser determinada de manera polinomial:

$$\begin{aligned} C(x) &= A(x)B(x) \bmod P(x) \\ &= A(x) \left( \sum_{i=0}^{m-1} b_i x^i \right) \bmod P(x) \\ &= \sum_{i=0}^{m-1} b_i (x^i A(x) \bmod P(x)) \\ &= b_0 A + b_1 (xA(x) \bmod P(x)) + b_2 (x^2 A(x) \bmod P(x)) + \dots + \\ &\quad b_{m-1} (x^{m-1} A(x) \bmod P(x)) \end{aligned} \quad (2.26)$$

En el cuadro 2.3 se muestra el algoritmo para esta multiplicación. Se podrá observar que se requieren  $m$  ciclos de reloj para obtener el cálculo de una multiplicación. Por otro lado, en cada iteración, se realizan las siguientes operaciones:

- 1 multiplicación de un elemento  $GF(p)$  por un elemento  $GF(p^m)$ , es decir, se requiere de  $m$  multiplicadores  $GF(p)$  en paralelo.
- 1 adición  $GF(p^m)$ . Se requieren  $m$  sumadores  $GF(p)$ .
- 1 multiplicación por  $x$ .
- 1 operación de reducción módulo  $P(x)$ .

<p><b>Algoritmo <i>LSE-first</i></b></p> <p>Input: <math>A(\alpha), B(\alpha) \in GF(2^m)</math></p> <p>Output: <math>C(\alpha) = A(\alpha) \cdot B(\alpha) \in GF(2^m)</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>C \leftarrow 0</math></li> <li>2. For <math>i = 0</math> to <math>m - 1</math> <ol style="list-style-type: none"> <li>2.1 <math>C \leftarrow b_i A + C</math></li> <li>2.2 <math>A \leftarrow Ax^i \bmod P(x)</math></li> </ol> </li> <li>3. return <math>(C(\alpha))</math></li> </ol> <p>=====</p>
--

Cuadro 2.3: Algoritmo para la multiplicación *LSE-first* de dos elementos en  $GF(2^m)$

La complejidad de las dos últimas operaciones es la que se ha indicado en la ecuación 2.25, así pues, la complejidad de espacio para este multiplicador es:

$$\#AND = 2m$$

$$\#XOR = 2m - 1$$

Este algoritmo lo utilizamos en nuestra implementación de un multiplicador serial/paralelo como se estudia en la sección 4.3. Otras aplicaciones que usan este algoritmo se pueden consultar en [79] y [6].

### Multiplicador *MSE-first*

Sean  $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$  y  $B(\alpha) = \sum_{i=0}^{m-1} b_i \alpha^i$  dos elementos del campo  $GF(2^m)$ , y sea  $C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i$  su producto. Recordando el esquema de Horner, la multiplicación polinomial  $C(x) = A(x)B(x) \bmod P(x)$ , se realiza como sigue:

$$\begin{aligned} c_0 &= b_{m-1}A \bmod P(x) \\ c_i &= (xc_{i-1} + b_{m-i-1}A) \bmod P(x) \end{aligned} \tag{2.27}$$

<p><b>Algoritmo <i>MSE-first</i></b></p> <p>Input: <math>A(\alpha), B(\alpha) \in GF(2^m)</math></p> <p>Output: <math>C(\alpha) = A(\alpha) \cdot B(\alpha) \in GF(2^m)</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>C \leftarrow 0</math></li> <li>2. For <math>i = 0</math> to <math>m - 1</math> <ol style="list-style-type: none"> <li>2.1 <math>C \leftarrow (b_{m-1-i}A + Cx) \bmod P(x)</math></li> </ol> </li> <li>3. return <math>(C(\alpha))</math></li> </ol> <p>=====</p>
--

Cuadro 2.4: Algoritmo para la multiplicación *MSE-first* de dos elementos en  $GF(2^m)$

y entonces  $C(\alpha) = c_{m-1}(\alpha)$ . Este procedimiento se muestra en el cuadro 2.4. Para calcular esta multiplicación se requieren las siguientes operaciones:

$$\#AND = 2m$$

$$\#XOR = m$$

En este multiplicador, el primer bit que se procesa del multiplicador es  $b_{m-1}$ , continuando en forma descendente hasta llegar a  $b_0$ . En [81][86] se pueden consultar diseños que utilizan este método.

#### 2.2.4. Multiplicación por dígitos

El multiplicador por dígitos, introducido en [79] presenta un equilibrio entre velocidad, área y consumo de potencia.

Su operación se basa en que es capaz de procesar varios coeficientes del elemento multiplicador  $B$  de manera simultánea. El número de coeficientes procesados en paralelo se define como el *tamaño de los dígitos* y se representa con la letra



$D$ . Así pues, requiere de un menor espacio en área que una arquitectura paralela, y tiene un mejor desempeño en velocidad que una arquitectura serial. Sus complejidades en espacio y tiempo son  $O(mD)$  y  $O(m/D)$  respectivamente.

Denotemos a  $d = \lceil m/D \rceil$ , el cual equivale al número total de “dígitos” en un polinomio de grado  $m - 1$ . Se puede representar al elemento multiplicador  $B$  como  $B = \sum_{i=0}^{d-1} B_i \alpha^{Di}$ , donde

$$B_i = \sum_{j=0}^{D-1} b_{(Di+j)} \alpha^j, \quad 0 \leq i \leq d - 1$$

y, si hubiera sido necesario,  $B$  se ha rellenado con ceros, es decir:

$$b_i = 0 \text{ para } m - 1 < i < d \cdot D.$$

Sea  $A(\alpha) = \sum_{j=0}^{m-1} a_j \alpha^j$  un elemento del campo. Entonces el producto polinomial, reducido por el polinomio irreducible, de  $A$  por  $B$  queda:

$$C(x) \equiv A(x)B(x) \bmod P(x) = \sum_{i=0}^{d-1} B_i A(x) x^{Di} \bmod P(x) \quad (2.28)$$

Como en el caso del multiplicador serial, hay dos versiones para este multiplicador: *LSDE-first* y *MSDE-first*. Se describe sólo el primer caso. En el cuadro 2.5 mostramos este procedimiento. Existen además dos métodos tradicionales para el diseño de circuitos que operan por dígitos. Uno de ellos consiste en usar como base una arquitectura serial que opera por bits y aplicar técnicas de *desdoblamiento* [66], mientras la segunda usa una arquitectura paralela por bits y le aplica técnicas de *doblamiento* [32]. Sin embargo, debido a los lazos de retroalimentación propios de estas arquitecturas, es muy difícil aplicar técnicas de “*pipeline*”.

### 2.2.5. Comparación de multiplicadores $GF(2^m)$

En la tabla 2.6 se muestran las características de complejidad de las diferentes arquitecturas de multiplicadores para  $GF(2^m)$ .

<b>Algoritmo por dígitos</b>		
Input: $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$ , $a_i \in GF(p)$ , $B = \sum_{i=0}^{\lceil m/D \rceil - 1} B_i \alpha^{Di}$		
Output: $C(\alpha) = A(\alpha) \cdot B(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i$ , con $c_i \in GF(p)$		
=====		
1. $C \leftarrow 0$		
2. For $i = 0$ to $\lceil m/D \rceil - 1$		
2.1 $C \leftarrow B_i A + C$		
2.2 $A \leftarrow Ax^D \bmod P(x)$		
3. $C := [C \bmod P(x)]$		
4. return $(C(\alpha))$		
=====		

Cuadro 2.5: Algoritmo para la multiplicación por dígitos de dos elementos en  $GF(2^m)$

Arquitectura	Complejidad de tiempo	Complejidad de espacio
Paralela	$O(1)$	$O(m^2)$
Por dígitos	$O(m/D)$	$O(mD)$
Serial	$O(m)$	$O(m)$

Cuadro 2.6: Comparación de los multiplicadores  $GF(2^m)$

## 2.3. Inversión y división en $GF(2^m)$

La inversión y la división sobre un campo finito son operaciones más complejas que la multiplicación. El cálculo de la inversión es necesario en criptosistemas que operan con curvas elípticas, y la división es a su vez requerida en aplicaciones de códigos de corrección de errores. La técnica más simple para calcular la inversión de un elemento no-nulo en  $GF(2^m)$  utiliza tablas de consulta. Para campos pequeños, por ejemplo para  $m < 8$ , una tabla de consulta ofrece un mejor desempeño en términos de espacio y de tiempo comparada con los métodos algorítmicos. Sin embargo, para campos mayores, el crecimiento exponencial de los requerimientos de memoria hacen poco atractivo este método para implementaciones en hardware.

La división sobre  $GF(2^m)$  consiste en dividir un polinomio  $A(x)$  por uno  $B(x)$  (ambos de grado menor que  $m$ ) módulo el polinomio irreducible  $P(x)$ . Esta operación se realiza usualmente en dos pasos:

1. Se encuentra el elemento inverso multiplicativo de  $B(x)$  en el grupo multiplicativo del anillo de polinomios  $GF(p)[x]$ .
2. Se calcula  $A(x)/B(x) = A(x) \cdot B^{-1}(x)$

En otras palabras, la división es realizada como una multiplicación de  $A$  por el inverso multiplicativo de  $B$ .

Los algoritmos para el cálculo de la inversión y la división en un campo finito se encuentran en una de las siguientes tres categorías:

- la basada en el *teorema pequeño de Fermat*,
- la que se fundamenta en el *algoritmo extendido de Euclides* y
- la que usa el *método de Gauss-Jordan* para la solución de un sistema de ecuaciones lineales.

Tomando como base las categorías mencionadas, realizamos una revisión de algunas propuestas representativas para el cálculo de estas dos operaciones.

### 2.3.1. Teorema pequeño de Fermat

Según el *Teorema Pequeño de Fermat* se tiene que  $A = A^{2^m} \forall A \in GF(2^m)$ , y por lo tanto  $A^{-1} = A^{2^m-2}$ . Considerando la representación binaria

$$2^m - 2 = \sum_{i=1}^{m-1} 2^i = 2 + 2^2 + 2^3 + \dots + 2^{m-1}$$

se tiene que  $A^{-1}$  puede ser calculada por medio de elevaciones al cuadrado y multiplicaciones sucesivas:

$$A^{-1} = A^{2^m-2} = \prod_{i=1}^{m-1} A^{2^i} = A^2 \cdot A^{2^2} \cdot A^{2^3} \dots A^{2^{m-1}} \quad (2.29)$$

La ecuación 2.29 requiere de  $m - 2$  multiplicaciones y de  $m - 1$  elevaciones al cuadrado. Note que debido a que se trabaja sobre un campo finito de característica dos, la elevación al cuadrado es una operación lineal. Además, si se usa una base normal para representar a los elementos del campo, se puede calcular  $A^2$  para cualquier  $A \in GF(2^m)$  con solo un corrimiento cíclico, como se explica en el apartado 2.4.2.

En [33], Itoh y Tsujii propusieron un algoritmo para el cálculo del inverso multiplicativo de un elemento en un campo finito  $GF(2^m)$ , y que opera sobre una base normal. Este algoritmo se basa en el teorema pequeño de Fermat y presenta una complejidad de  $2[\log_2(m-1)]$  multiplicaciones en  $GF(2^m)$  y  $m-1$  corrimientos cíclicos. Como se muestra en [27], este método también es aplicable a campos finitos con una representación estándar o polinomial.

Los autores proponen en [33] tres algoritmos. Los dos primeros describen cadenas de sumas (*addition chains*) para calcular la inversión basada en la exponenciación en el campo  $GF(2^m)$ , mientras que el tercero describe un método basado en la inversión en un subcampo. Revisamos el primer algoritmo que es solo aplicable para órdenes del campo  $m$  tales que  $m = 2^r + 1$ , para algún positivo  $r$ , y que se basa en la observación que el exponente  $2^m - 2$  puede ser reescrito como

<p><b>Algoritmo Inversión con <math>m = 2^r + 1</math></b></p> <p>Input: <math>A \in GF(2^m), A \neq 0, m = 2^r + 1</math></p> <p>Output: <math>C = A^{-1}</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>C \leftarrow A</math></li> <li>2. for <math>i = 0</math> to <math>r - 1</math> do             <ol style="list-style-type: none"> <li>2.1 <math>D \leftarrow C^{2^{2^i}}</math> {Observar <math>2^i</math> corrimientos cíclicos}</li> <li>2.2 <math>C \leftarrow C \cdot D</math></li> <li>2.3 end for</li> </ol> </li> <li>3. <math>C \leftarrow C^2</math></li> <li>4. return (<math>C</math>)</li> </ol> <p>=====</p>
---

Cuadro 2.7: Algoritmo para la inversión de un elemento en  $GF(2^m)$ 

$(2^{m-1} - 1) \cdot 2$ . Así, si  $m = 2^r + 1$ , se puede calcular  $A^{-1} \equiv (A^{2^{2^r-1}})^2$ . De esta manera se puede reescribir a  $2^{2^r} - 1$  como

$$2^{2^r} - 1 = (2^{2^{r-1}} - 1)2^{2^{r-1}} + (2^{2^{r-1}} - 1) \quad (2.30)$$

De esta ecuación se deriva el algoritmo que se presenta en el cuadro 2.7. Observe que este algoritmo realiza  $r = \log_2(m-1)$  iteraciones. En cada iteración se realizan una multiplicación e  $i$ -corrimientos cíclicos, para  $0 \leq i < r$ , lo que determina una complejidad total de  $\log_2(m-1)$  multiplicaciones y  $m-1$  corrimientos cíclicos.

Como se estudia en [27], el algoritmo del cuadro 2.7 puede ser generalizado para cualquier valor de  $m$ . Primero, se puede escribir  $m-1$  como

$$m-1 = \sum_{i=1}^t 2^{k_i}, \quad \text{donde : } k_1 > k_2 > \dots > k_t \quad (2.31)$$

Usando esta ecuación y el hecho de que  $A^{-1} \equiv (A^{2^{m-1}-1})^2$ , el inverso de  $A$  puede

escribirse como:

$$(A^{2^m-1})^2 = \left[ (A^{2^{2^k t}-1})((A^{2^{2^k t-1}-1}) \dots [(A^{2^{2^k 2}-1})(A^{2^{2^k 1}-1})^{2^{2^k 3}} \dots]^{2^{2^k t}} \right]^2 \quad (2.32)$$

La complejidad de la ecuación 2.32 es:

$$\# \text{ MUL} = \lceil \log_2(m-1) \rceil + \text{HW}(m-1) - 1$$

$$\# \text{ CSH} = n - 1$$

donde  $\text{HW}(\cdot)$  representa el peso de Hamming del operando, es decir, el número de “unos” en la representación binaria del operando. MUL representa multiplicaciones en  $GF(2^m)$ , y CSH son corrimientos cíclicos sobre  $GF(2)$  cuando se usa una base normal.

En [88], [34] y [31], los autores proponen implementaciones en hardware de inversores que usan como base al teorema pequeño de Fermat, y donde se aprecia que las complejidades en tiempo y espacio son  $O(m \log m)$  y  $O(m^2)$  respectivamente.

### 2.3.2. Algoritmo de Euclides

Una propuesta diferente utiliza el algoritmo de Euclides para encontrar el máximo común divisor (MCD) de dos polinomios, y para calcular el inverso multiplicativo de dos elementos en un campo finito  $GF(2^m)$ .

En el algoritmo descrito en el cuadro 2.8,  $A(x)$  y  $B(x)$  son dos polinomios cuyo MCD ha de ser calculado, mientras  $P(x)$  es un polinomio irreducible sobre  $GF(2^m)$ .

Si  $B(x)$  es un polinomio tal que  $B(\alpha) \neq 0$  en  $GF(2^m)$ , entonces existen dos polinomios  $W(x)$  y  $U(x)$  con coeficientes en  $GF(2)$ , tales que

$$\text{MCD}(P(x) B(x)) = W(x) \cdot P(x) + U(x) \cdot B(x),$$

<b>Algoritmo MCD</b>
Input: $A(x)$ y $B(x)$
Output: $\text{MCD}(A(x), B(x))$
=====
1. $A_0 \leftarrow A(x); B_0 \leftarrow B(x); i \leftarrow 0$
2. repeat $i := i + 1;$
2.1 $Q_i \leftarrow A_{i-1} \text{DIV } B_{i-1};$
2.2 $R_i \leftarrow A_{i-1} - Q_i \cdot B_{i-1}; \quad \{R_i \leftarrow A_{i-1} \bmod B_{i-1}\}$
2.3 $A_i \leftarrow B_{i-1}; B_i \leftarrow R_i$
3. until $R_i = 0,$
4. return $\text{MCD}(A(x), B(x)) = A_i$
=====

Cuadro 2.8: Algoritmo para calcular el MCD de dos elementos en  $GF(2^m)$ 

donde  $P(x)$  es el polinomio irreducible sobre  $GF(p)$  utilizado para representar a los elementos de  $GF(2^m)$ . Como  $P(x)$  no puede tener divisores no-triviales,  $W(x) \cdot P(x) + U(x) \cdot B(x) = 1$ . Ya que toda la aritmética en  $GF(2^m)$  es realizada módulo  $P(x)$ , se tiene  $U(x) \cdot B(x) = 1 \bmod P(x)$ , y  $U(\alpha) = B(\alpha)^{-1}$ , es decir  $U(\alpha)$  es la esperada inversa multiplicativa de  $B(\alpha)$ .

Haciendo uso del algoritmo del cuadro 2.8 y considerando a los siguientes polinomios:  $R = R_i, S = R_{i-1}, U = U_i$  y  $V = U_{i-1}$ , Brunner *et al.* proponen en [4] el algoritmo mostrado en el cuadro 2.9 para calcular el inverso multiplicativo de un elemento en el campo finito  $GF(2^m)$ . El número de compuertas y registros para implementar este inversor es proporcional a  $m$ . Los autores proponen una arquitectura basada en celdas básicas reproducibles que permite el cálculo de una inversión cada  $m$  ciclos de reloj. Así pues, las complejidades en espacio y tiempo de este inversor son  $O(m)$  y  $O(m)$  respectivamente. Propuestas de arquitecturas que usan este algoritmo se pueden consultar en [8], [3], [4] y [28].

**Algoritmo de Inversión**Input:  $B(x)$ ,  $B(\alpha) \neq 0$ Output:  $B(\alpha)^{-1}$ 

```

=====
1.  $S \leftarrow P(x); V \leftarrow 0; R \leftarrow B(x); U \leftarrow 1;$ 
2. repeat   {grado de  $S \geq$  grado de  $R$ }
2.1  $Q \leftarrow SDIVR;$ 
2.2  $tmp \leftarrow S - Q \cdot R; S \leftarrow R; R \leftarrow tmp;$ 
2.3  $tmp \leftarrow V - Q \cdot U; V \leftarrow U; U \leftarrow tmp;$ 
3. until  $R = 0;$ 
4. return  $B^{-1}(x) = V$ 
=====

```

Cuadro 2.9: Algoritmo para calcular el inverso multiplicativo de un elemento en  $GF(2^m)$



### 2.3.3. Algoritmo de Gauss-Jordan

Otro método que se emplea para el cálculo directo de la división sobre un campo finito, es el algoritmo de Gauss-Jordan para la solución de un sistema de ecuaciones lineales. Hasan y Bhargava en [29] utilizan este método para calcular la división de dos elementos en el campo finito  $GF(2^m)$  basando su propuesta en el siguiente teorema:

**Teorema 2.3.1 (TeoHB)** *Sea  $G(x)$  un polinomio irreducible sobre  $GF(q)$ , y sean  $A(\alpha), B(\alpha)$  y  $C(\alpha)$  tres elementos del campo  $GF(2^m)$ . Estos elementos son representados por una base de la forma  $\{\alpha^{k_0}, \alpha^{k_1}, \dots, \alpha^{k_{m-1}}\}$ . Entonces la división*

$$B(\alpha) = C(\alpha)/A(\alpha) \text{ mod } G(\alpha), \quad A(\alpha) \neq 0,$$

*en el campo finito  $GF(2^m)$ , puede ser realizada por medio de la solución de la siguiente ecuación sobre el campo  $GF(p)$ :*

$$\begin{pmatrix} ap_{m-1}^{k_{m-1}} & ap_{m-1}^{k_{m-2}} & \dots & ap_{m-1}^{k_0} \\ ap_{m-2}^{k_{m-1}} & ap_{m-2}^{k_{m-2}} & \dots & ap_{m-2}^{k_0} \\ \vdots & \vdots & \vdots & \vdots \\ ap_0^{k_{m-1}} & ap_0^{k_{m-2}} & \dots & ap_0^{k_0} \end{pmatrix} \begin{pmatrix} b_{m-1} \\ b_{m-2} \\ \vdots \\ b_0 \end{pmatrix} \equiv \begin{pmatrix} c_{m-1} \\ c_{m-2} \\ \vdots \\ c_0 \end{pmatrix} \quad (2.33)$$

De la ecuación 2.33 se puede observar que cuando las coordenadas de  $A(\alpha), C(\alpha)$  y los elementos de soporte son conocidos, entonces  $B(\alpha) = C(\alpha)/A(\alpha) \text{ mod } G(\alpha)$  puede ser calculado por medio de la solución del sistema de  $m$  ecuaciones lineales con  $m$  incógnitas sobre  $GF(p)$ .

La ecuación 2.33 se puede representar como

$$\mathcal{Q} \cdot B(\alpha) = C(\alpha),$$

donde  $\mathcal{Q} \equiv [q_{i,j}]_{i,j=0}^{m-1} = [ap_{m-1-i}^{k_{m-1-j}}]_{i,j=0}^{m-1}$  y donde  $A, B$  y  $C$  están escritos como vectores columnas, es decir,  $GF(2^m) \approx [GF(2)]^m$  y  $\mathcal{Q} \in [GF(2)]^{m \times m}$ .

<p><b>Algoritmo División</b></p> <p>Input: <math>A(\alpha), B(\alpha)</math> y <math>G(\alpha)</math></p> <p>Output: <math>C(\alpha) = (A(\alpha)/B(\alpha) \text{MOD} G(\alpha))</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. Se determinan las coordenadas de los elementos de soporte.</li> <li>2. Se construye la ecuación 2.31.</li> <li>3. Se resuelve la ec. 2.31 para <math>B</math> y obtener el resultado esperado.</li> </ol> <p>=====</p>
--

Cuadro 2.10: Algoritmo para calcular la división de dos elementos en  $GF(2^m)$ 

De las consideraciones mencionadas, los autores proponen el algoritmo que se muestra en el cuadro 2.10. La complejidad en tiempo y espacio para este proceso de división es  $O(m)$  y  $O(m^2)$  respectivamente.

En [30] Hasan y Bhargava proponen una arquitectura sistólica para calcular la división usando este algoritmo operando sobre una base estándar. Por su parte, Fenn *et al.* en [17] proponen un algoritmo semejante, pero operando sobre una base dual. El algoritmo propuesto en [30] lo usamos en esta tesis para implementar en FPGA un divisor sistólico y serial como se estudiará en el capítulo 4.

#### 2.3.4. Comparación de inversores $GF(2^m)$

En el cuadro 2.11 presentamos un resumen de las complejidades de los operadores para la inversión y la división que hemos mencionado en este apartado.

Categoría	Compl. de tiempo	Compl. de espacio
Teorema de Fermat (Inversión)	$O(m \log m)$	$O(m^2)$
Alg. de Euclides (Inversión)	$O(m)$	$O(m)$
Alg. de Gauss-Jordan (División)	$O(m)$	$O(m^2)$

Cuadro 2.11: Comparación de inversores  $GF(2^m)$ 

## 2.4. Exponenciación en $GF(2^m)$

Existen diversos métodos para calcular operación de exponenciación sobre campos finitos. Entre ellos se encuentran el método de elevación al cuadrado y multiplicación, conocido como *método binario* [39], el denominado método de ventanas deslizantes (*sliding-windows*) [41],[67], la técnica de cadena de sumas (*addition-chains*) [47],[56] y el método de Montgomery [9]. Entre los mencionados, revisamos el método binario ya que es el que utilizamos en el capítulo 4 como base para nuestra implementación en hardware de un exponenciador. Un resumen de estos métodos se puede consultar en [25].

### 2.4.1. El método binario

Sea  $A = \sum_{i=0}^{m-1} a_i \alpha^i$  un elemento arbitrario en  $GF(2^m) - \{0\}$  y sea  $e$ , ( $1 \leq e \leq 2^{m-1}$ ) un entero, que hará las veces de potencia, cuya representación en binario es  $e = \sum_{j=0}^{n-1} e_j 2^j = (e_{n-1}, e_{n-2}, \dots, e_1, e_0)$ ,  $e_j = \{0, 1\}$ . Entonces la potencia  $R = A^e$  está también en  $GF(2^m)$  y se puede calcular utilizando el método binario [39]. Existen dos versiones de este método que dependen del primer bit del exponente que es examinado en el algoritmo: la versión *MSB-first* (*Most Significant Bit-first*) y la versión *LSB-first* (*Least Significant Bit-first*). En los cuadros 2.12 y 2.13 se muestran estos algoritmos.

El método binario requiere de las siguientes operaciones:

- Elevaciones al cuadrado:  $n$  en su versión *LSB-first* y  $n - 1$  en su versión

<p><b>Algoritmo Exponenciación <i>MSB-first</i></b></p> <p>Input: <math>A, e, P</math></p> <p>Output: <math>R = A^e \bmod P</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. If <math>e_{n-1} = 1</math> then <math>R := A</math> else <math>R = 1</math></li> <li>2. for <math>i := n - 2</math> downto 0 do <ol style="list-style-type: none"> <li>2.1 <math>R := R \cdot R \bmod P</math></li> <li>2.2 If <math>e_i = 1</math> then <math>R := R \cdot A \bmod P</math></li> </ol> </li> <li>3. return <math>R</math></li> </ol> <p>=====</p>
---

Cuadro 2.12: Algoritmo *MSB-first*, para elevar a una potencia, un elemento en  $GF(2^m)$

<p><b>Algoritmo Exponenciación <i>LSB-first</i></b></p> <p>Input: <math>A, e, P</math></p> <p>Output: <math>R = A^e \bmod P</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>C := A ; R = 1</math></li> <li>2. for <math>i := 0</math> to <math>n - 1</math> do <ol style="list-style-type: none"> <li>2.1 If <math>e_i = 1</math> then <math>C := C \cdot A \bmod P</math></li> <li>2.2 <math>C := C \cdot C \bmod P</math></li> </ol> </li> <li>3. return <math>R</math></li> </ol> <p>=====</p>
---

Cuadro 2.13: Algoritmo *LSB-first* para elevar a una potencia, un elemento en  $GF(2^m)$

No. total de multiplicaciones	<i>MSB-first</i>	<i>LSB-first</i>
Máximo	$2(n - 1)$	$2n$
Mínimo	$n - 1$	$n$
Promedio	$1.5(n - 1)$	$1.5n$

Cuadro 2.14: Método binario de exponenciación  $GF(2^m)$ 

<b>Algoritmo Binario para Exponenciación</b>
Input: $A, e, P, k$
Output: $R = A^e \bmod P$
=====
1. $l = 1$
2. for $i := 0$ to $2^k - 1$ do
2.1 $c[i] := l; l = l \cdot A; R := 1$
3. for $i_1 = m/k - 1$ downto 0 do
3.1 $R := R^{2^k}; R := R \cdot e[i_1];$
4. return $R$
=====

Cuadro 2.15: Algoritmo binario para elevar a una potencia un elemento en  $GF(2^m)$ 

*MSB-first.*

- Multiplicaciones: el número de “unos” en la representación binaria del exponente  $e$ ; excluyendo el MSB en la versión *MSB-first*.

En el cuadro 2.14 se resumen las características de este exponenciador.

Una generalización del método binario es el método *m-ario* cuyo algoritmo se muestra en el cuadro 2.15.

Se puede apreciar que las complejidades en tiempo y espacio para una implementación en hardware de un exponenciador que usa el método binario está fuerte-

mente determinada por el tipo de multiplicador que se use para esta construcción. Por otro lado, también es importante considerar que si es una base normal la utilizada para representar a los elementos del campo, la operación de elevación al cuadrado se limitará a simples corrimientos cíclicos de los coeficientes del elemento considerado, lo que permitiría una mejor medida de la complejidad de espacio del exponenciador.

Otros métodos para hacer la exponenciación en un campo finito se pueden estudiar en [36] y [85].

### 2.4.2. Elevación al cuadrado en $GF(2^m)$

Consideremos a la elevación al cuadrado como un caso de la exponenciación. Se sabe que la elevación al cuadrado es una operación lineal en el campo  $GF(2^m)$ , es decir, para dos elementos  $A(\alpha)$  y  $B(\alpha) \in GF(2^m)$  se tiene que  $(A(\alpha) + B(\alpha))^2 = A(\alpha)^2 + B(\alpha)^2$ . Así que si  $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$  entonces  $A(\alpha)^2 = \sum_{i=0}^{m-1} a_i \alpha^{2i}$  (recordamos que en  $GF(2)$ ,  $a^2 = a$  para cualquier  $a$ ).

#### Usando una base normal

Si se usa una base normal para la representación del campo finito, se obtiene la forma más simple de realizar la elevación al cuadrado en  $GF(2^m)$ . Una *base normal* es una base de la forma

$$N = \{\alpha, \alpha^2, \alpha^4, \dots, \alpha^{2^{m-1}}\}$$

para un elemento apropiado  $\alpha \in GF(2^m)$ , y  $m$  es el *orden* del campo de Galois [48]. Se sabe que para cualquier entero positivo  $m$  siempre existe una base normal en el campo finito  $GF(2^m)$ .

Cualquier elemento  $B \in GF(2^m)$  puede entonces ser representado como

$$B = \sum_{j=0}^{m-1} b_j \alpha^{2^j} = b_0 \alpha + b_1 \alpha^2 + b_2 \alpha^4 + \dots + b_{m-1} \alpha^{2^{m-1}}$$

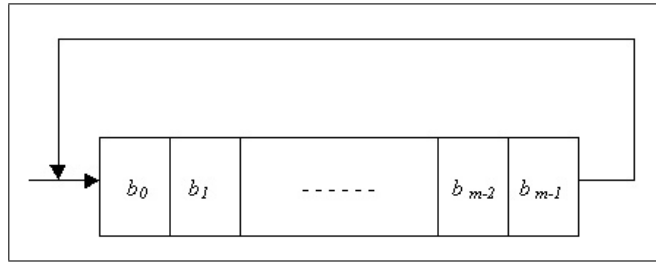


Figura 2.2: Elevación al cuadrado en base normal

donde  $b_j \in GF(2)$  y la suma es realizada en  $GF(2^m)$ .

Ya que  $\alpha \in GF(2^m)$  se tiene, por el Teorema de Fermat, que  $\alpha^{2^m} = \alpha$ . Así

$$\begin{aligned} B^2 &= b_0\alpha^2 + b_1\alpha^4 + b_2\alpha^8 + \dots + b_{m-2}\alpha^{2^{m-1}} + b_{m-1}\alpha^{2^m} \\ &= b_{m-1}\alpha + b_0\alpha^2 + b_1\alpha^4 + \dots + b_{m-2}\alpha^{2^{m-1}} \end{aligned}$$

Esto es, si  $B$  es representado como un vector cuyos componentes son los elementos en una base normal de  $GF(2^m)$  en la forma  $B = [b_0, b_1, b_2, \dots, b_{m-1}]$ , entonces  $B^2 = [b_{m-1}, b_0, b_1, \dots, b_{m-2}]$ . Así en una representación de base normal,  $B^2$  es un corrimiento cíclico de  $B$ . Por lo tanto, la elevación al cuadrado en  $GF(2^m)$  puede ser realizada físicamente por un registro binario y circuitería lógica que controle el corrimiento como se ilustra en la figura 2.2.

### Usando una base estándar

Sea  $G(x) = \sum_{i=0}^m g_i x^i$  un polinomio mónico irreducible de grado  $m$  sobre  $GF(2)$ , y sea  $\alpha$  una raíz de  $G(x)$ . Cualquier elemento  $A$  de  $GF(2^m)$  puede ser expresado como  $A = a_0\alpha^0 + a_1\alpha^1 + a_2\alpha^2 + \dots + a_{m-1}\alpha^{m-1}$ , donde las coordenadas  $a_i \in GF(2)$  para  $0 \leq i \leq m-1$ , y  $\{\alpha^0, \alpha^1, \dots, \alpha^{m-1}\}$  es una base estándar o canónica de  $GF(2^m)$ .

Según [31],  $\alpha^k$  se puede representar como

$$\alpha^k = \sum_{i=0}^{m-1} p_i^{[k]} \alpha^i, \quad (2.34)$$

o equivalentemente

$$\alpha^k = \begin{bmatrix} p_0^{[k]} \\ p_1^{[k]} \\ \vdots \\ p_{m-1}^{[k]} \end{bmatrix} \begin{bmatrix} 1 \\ \alpha \\ \alpha^2 \\ \vdots \\ \alpha^{m-1} \end{bmatrix}$$

Es decir,  $p_i^{[k]}$  puede ser considerada como la  $i$ -ésima coordenada del elemento  $\alpha^k$  con respecto a la base canónica  $\{\alpha^0, \alpha^1, \dots, \alpha^{m-1}\}$ . Cuando  $0 \leq k \leq m-1$ , se tiene que

$$p_i^{[k]} = \delta_{i,k} \quad 0 \leq i \leq m-1 \quad (2.35)$$

donde  $\delta_{i,k}$  es la delta de Kronecker, la cual es igual a 1 cuando  $i = k$ , y es cero de cualquier otra manera.

Para un polinomio irreducible del tipo AOP (*All One Polynomial*)  $G(x) = \sum_{i=0}^m x^i$  sobre  $GF(2^m)$ , una raíz  $\alpha$  de  $G(x)$  satisface que  $\alpha^m = \sum_{i=0}^{m-1} \alpha^i$  y  $\alpha^{m+1} = 1$ .

Así, para cualquier entero  $j$ , se tiene que

$$p_i^{[j]} = p_i^{[j \bmod (m+1)]} \quad (i = 0, 1, \dots, m-1). \quad (2.36)$$

Se puede calcular así  $C = A^2, C \in GF(2^m)$  como

$$C = \sum_{k=0}^{m-1} c_k \alpha^k = \sum_{i=0}^{m-1} a_i \alpha^{2i} = \sum_{i=0}^{m-1} a_i \sum_{k=0}^{m-1} p_k^{[2i]} \alpha^k.$$

donde

$$c_k = \sum_{i=0}^{m-1} a_i p_k^{[2i]}.$$

Si  $G(x)$  es un polinomio irreducible del tipo AOP, 2.35 y 2.36 resultan en

$$p_k^{[2i]} = \begin{cases} \delta_{2i,m} + \delta_{2i,k} & k - \text{par} \\ \delta_{2i,m} + \delta_{2i,k+m+1} & k - \text{impar} \end{cases} \quad (2.37)$$



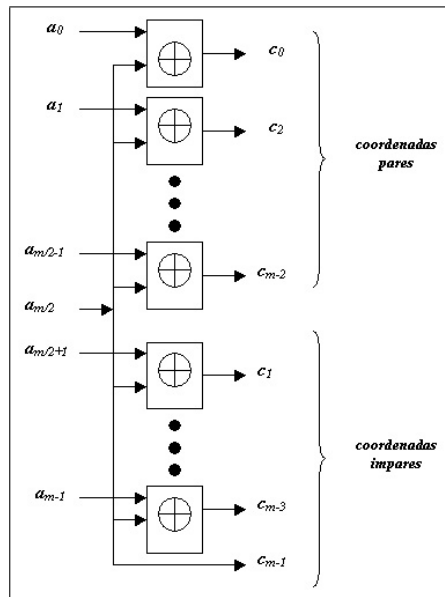


Figura 2.3: Elevación al cuadrado en base estándar

Consecuentemente,  $c_{m-1} = a_{m/2}$  y para  $0 \leq k \leq m-2$ ,

$$c_k = \begin{cases} a_{m/2} + a_{k/2} & (\text{mod } 2), k - \text{par} \\ a_{m/2} + a_{(k+m+1)/2} & (\text{mod } 2), k - \text{impar} \end{cases} \quad (2.38)$$

La figura 2.3 muestra la configuración para una operación en paralelo de elevación al cuadrado sobre una base estándar. Se requiere de  $m-1$  compuertas XOR y se tiene un tiempo de retardo de  $D_x$ . Un estudio complementario de elevadores al cuadrado sobre una base polinomial se puede consultar en [36] y en [89].



# Capítulo 3

## Dispositivos en hardware para $GF(2^m)$

En este capítulo hacemos un estudio introductorio de los dispositivos reconfigurables, es decir, de los circuitos programables, y consideramos las características particulares de los FPGA's, los cuales hemos utilizado para realizar las implementaciones en este trabajo. Un estudio más amplio acerca de este tema se puede consultar en [10] y en [53]. Además, revisamos los fundamentos de los arreglos sistólicos y sugerimos algunas referencias para un mejor estudio.

### 3.1. Consideraciones generales

Como ya se ha mencionado en la introducción, las implementaciones en hardware de los operadores aritméticos sobre campos finitos  $GF(2^m)$ , se presentan como una alternativa a las implementaciones en software debido a la mayor seguridad y desempeño que se obtiene con ellas. Tales implementaciones en hardware usan como elemento básico a los circuitos integrados (*CI's o chips*), cuyas frecuencias de operación permiten cumplir con los requerimientos de alta velocidad que se desean. Como un ejemplo de estas implementaciones y las velocidades en que

Implementación	SW/HW	$m$	Mult.	Plataforma
López <i>et al</i> (1999)	SW	162	10.5 mS	Ultra-Sparc a 300 Mhz
Savas <i>et al</i> (2000)	SW	160	18.3 $\mu S$	Micro ARM a 80 Mhz
Rodríguez (2000)	SW	163	5.4 $\mu S$	Pentium II a 450 Mhz
Rosner (1998)	HW	168	4.5 mS	FPGA XC4062 a 16 Mhz
Orlando (1999)	HW	167	0.21 mS	FPGA XCV400E a 76 Mhz
Lee <i>et al</i> (2000)	HW	192	2.88 $\mu S$	(No implementado)

Cuadro 3.1: Implementación de multiplicadores  $GF(2^m)$  en SW/HW

operan, mostramos en el cuadro 3.1 implementaciones de multiplicadores  $GF(2^m)$  que han sido realizadas recientemente en software y en hardware.

Existen básicamente dos tipos de circuitos integrados que pueden ser utilizados para realizar las implementaciones en hardware: circuitos VLSI (*Very Large Scale of Integration*), y circuitos programables.

Hay diversos aspectos que se deben de evaluar en una arquitectura para CI's. Los más importantes son los siguientes:

- *Complejidad de espacio.* Se define por el número de compuertas lógicas AND, OR y XOR requeridas para la construcción del circuito.
- *Complejidad de tiempo.* Se define por el número de retardos de las compuertas que forman la ruta crítica de datos.
- *Jerarquía.* Se refiere a la división consecutiva de un módulo en submódulos. Esto resulta eventualmente en submódulos más simples y comprensibles.
- *Regularidad.* Tiene que ver con arquitecturas que están compuestas de módulos o submódulos *similares*. Un ejemplo claro son los circuitos basados en arreglos de compuertas como los FPGA's.

- *Modularidad.* Es una propiedad en una arquitectura en la que la función de cada submódulo está definida de una manera clara y simple.

Los dos primeros aspectos son ampliamente considerados en esta tesis, y son prácticamente las medidas utilizadas en nuestro trabajo para definir la eficiencia de nuestras implementaciones. Los tres últimos se refieren a propiedades estructurales de los circuitos [90], y contar con ellas permite implementaciones más eficientes en hardware. Las arquitecturas para  $GF(2^m)$  desarrolladas en esta tesis, cuentan con las propiedades estructurales que se han mencionado.

## 3.2. Hardware reconfigurable

### 3.2.1. Sistemas digitales y FPGA's

En el desarrollo tecnológico de los circuitos integrados se aprecian claramente dos extremos: en uno, digamos inferior, se encuentra la tecnología SSI/MSI (*Small/Medium Scale of Integration*), conocida habitualmente como lógica discreta, y en el otro, superior, la tecnología VLSI, cada vez más identificada con el término ASIC (*Application Specific Integrated Circuit*). Entre ambos extremos se han desarrollado diversos dispositivos que se encuentran en el mercado de circuitos integrados. En los últimos años un nuevo grupo de dispositivos es ampliamente utilizado en el diseño de circuitos digitales, son los denominados *Dispositivos Lógicos Programables*, cuya principal característica es la de poder ser programados y reprogramados por el usuario final. Así pues, los diseñadores en el momento de desarrollar un circuito lógico disponen de estas tres alternativas mencionadas, y dependerá sobre todo, del tipo de aplicación que se requiera. En la figura 3.1 se muestra una organización de los sistemas digitales, donde se puede apreciar el lugar de los FPGA's.

Desde una perspectiva funcional, la lógica discreta resulta claramente insuficiente para la mayoría de las aplicaciones digitales, ya que la tendencia está orientada a una mayor integración con el mínimo costo posible, lo cual se consigue

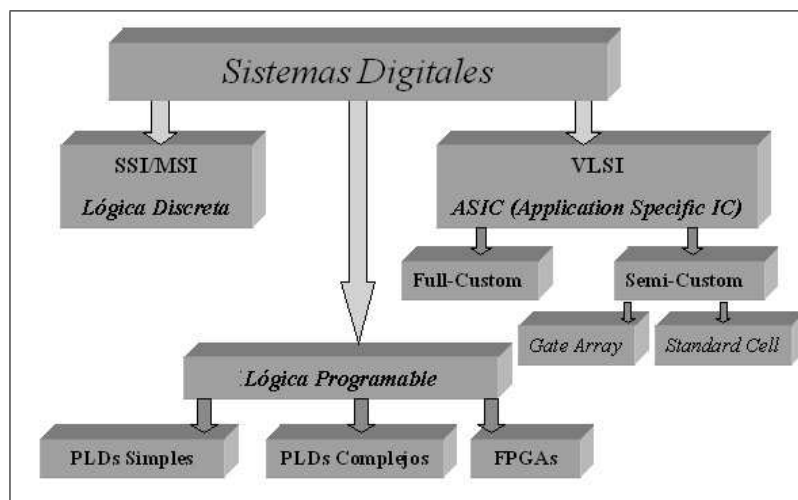


Figura 3.1: Organización de los Sistemas Digitales

integrando en un solo chip el diseño completo y adaptado al usuario. Esta integración sólo es posible mediante dos técnicas: diseñando en el propio silicio, es decir, construyendo un ASIC total o parcialmente adaptado (*full/semi custom*) a las necesidades del usuario, o bien utilizando dispositivos programables. Entre tales dispositivos programables, los más representativos son los FPGA's (*Field Programmable Gate Arrays*) y los EPLD's (*Erasable Programmable Logic Devices*), y en el área de sistemas digitales y computación se les refiere como la base del *Hardware Reconfigurable*.

### 3.2.2. FPGA's y ASIC's

Como se ha mencionado anteriormente, los diseñadores de sistemas digitales deben decidir usar generalmente FPGA's o ASIC's en el momento de implementar sus circuitos diseñados. Revisamos enseguida las diferencias más importantes entre ambos tipos de dispositivos, lo que nos permitirá tener los elementos para la toma de una correcta decisión a la hora de implementar.

Un dispositivo de lógica programable (PLD), es un dispositivo de propósito general capaz de implementar la lógica de decenas o cientos de paquetes de circuitos SSI. Es programado por el usuario usando hardware de programación económico. Sin embargo, se ven limitados por su consumo de energía y retardos. Por otro lado los ASIC's consisten de una base de transistores prediseñados con sus cableados correspondientes. El cableado es construido durante el proceso de fabricación, por esto cada diseño necesita de *máscaras* adecuadas para realizar el cableado. El desarrollo de las máscaras para estos circuitos eleva el costo para volúmenes pequeños de producción. Los tiempos típicos de desarrollo de estos dispositivos son de 4 a 6 semanas.

De manera similar a los circuitos ASIC's, las nuevas generaciones de FPGA's implementan miles de compuertas lógicas en un solo circuito integrado y cada vez son más populares para el diseño de sistemas digitales de regular tamaño. El uso de FPGA's en el campo de la computación, determina actualmente el camino para una clase general de organización de computadoras conocida como *Cómputo Reconfigurable* [11]. Esto permite que una máquina pueda ser reconfigurada de acuerdo a las necesidades que se presenten en una aplicación, mientras que permite la característica de ser reutilizada en cualquier momento.

Consideremos las siguientes características:

- *Costo de fabricación.* Cada diseño que ha de ser implementado en un ASIC requiere desarrollar una máscara de acuerdo a los patrones de cableado. El costo de cada máscara es de alrededor de miles de dólares, costo que debe ser amortizado sobre el número total de unidades fabricadas. Entre más unidades sean construidas se reduce el impacto del costo de desarrollo. Los FPGA's no requieren de este proceso, por lo que no tienen asociado este costo, además son excelentes para volúmenes relativamente pequeños (1000 a 10000 unidades).
  
- *Tiempo de desarrollo.* El proceso de manufactura de un ASIC toma varias

semanas desde el término del diseño hasta la liberación de las partes desarrolladas. Un FPGA puede ser programado en minutos por el usuario. En un FPGA, una modificación para corregir el diseño puede realizarse rápidamente y con un bajo costo. El rápido desarrollo produce a su vez que la aparición de nuevos productos en el mercado sea más rápida.

- *Verificación del diseño.* Debido a los costos de la ingeniería y a los retardos de manufactura, los usuarios de ASIC's deben verificar sus diseños por medio de una intensa simulación antes de la fabricación. Para verificar la funcionalidad del diseño en un sistema, debe simularse durante grandes períodos de tiempo. Los FPGA's reducen estos problemas. En vez de simular por grandes períodos de tiempo, los usuarios de FPGA's pueden elegir realizar una verificación en el circuito. Los diseñadores pueden implementar el diseño y usar partes funcionales como un prototipo. Este prototipo puede ser verificado en tiempo real y con una buena precisión del tiempo.
- *Costo de pruebas.* Todos los circuitos integrados deben de ser probados para verificar la fabricación correcta y el empaquetamiento. La prueba es diferente para cada diseño. Los diseños en un ASIC incurren en costos asociados, además que el desarrollo de pruebas efectivas es difícil, por lo que puede ocurrir que algunos chips defectuosos pasen las pruebas y solo hasta que fallan en un sistema es como se detectan los errores generando así mas costos. En contraste, el programa de prueba es el mismo para todos los diseños FPGA's. Los programas de prueba de los fabricantes verifican que cada FPGA pueda funcionar para cualquier posible diseño o que pueda ser implementado en él. Los usuarios solo necesitan pruebas específicas para sus implementaciones particulares.

Hemos de mencionar también algunas desventajas de los FPGA's.

- Estos circuitos tienen un área dentro del chip que no puede ser usada por los diseñadores, Los interruptores programables incrementan el retardo de



la señal agregando resistencia y capacitancia a las interconexiones. Como resultado, los FPGA's son muy grandes y más lentos que sus equivalentes ASIC's [82].

- Algunos FPGA's son alrededor de 10 veces más grandes para la misma capacidad de compuertas que sus contrapartes ASIC's, y son correspondientemente más caros. Para diseños mayores, los diseñadores deben particionar el diseño en varios FPGA's o mudar el diseño a un ASIC.
- Las conexiones en un FPGA se retardan por la circuitería de programación. Los puntos de interconexión programables a lo largo de una ruta de cableado agregan resistencia a la ruta. Todos los puntos de programación en las interconexiones agregan capacitancia a las rutas internas. Finalmente, debido a que es necesaria más área para la misma cantidad de lógica, las líneas de interconexión entre elementos lógicos puede ser muy grande. Los FPGA's actuales son dos o tres veces más lentos que los ASIC's.

### **3.2.3. Características de los FPGA's**

Entre los circuitos programables, los FPGA's están siendo ampliamente utilizados en los sistemas digitales modernos debido a las mejoras tecnológicas que los fabricantes están introduciendo en su construcción, tales como mayores velocidades de operación, que alcanzan los 200 mhz y mayor densidad de integración que puede ser de hasta un millón de compuertas lógicas. Entre los fabricantes más importantes se pueden mencionar a Xilinx, Altera, Actel y Cypress, entre otros.

Un FPGA es un dispositivo que tiene las siguientes características:

- Se puede implementar en él una gran variedad de circuitos.
- Se puede programar y reprogramar en campo.
- Se compone de elementos booleanos y de almacenamiento.

- Se implementan grandes circuitos de cientos de miles de compuertas.

### 3.2.4. Estructura interna de un FPGA

La estructura interna de un FPGA está compuesta por celdas o bloques semejantes e independientes que son programables y que se pueden interconectar entre sí a través de canales también programables, como se muestra en la figura 3.2. Estos generadores tienen la capacidad de operar con funciones de salida que pueden ser complejas. El nombre de estos bloques varía según el fabricante. Nos referiremos en este trabajo a los FPGA's fabricados por Xilinx, quien da a estos bloques el nombre de CLB's (*Configurable Logic Blocks*). En la serie Virtex de FPGA's de Xilinx cada CLB está dividido en dos partes semejantes conocidas como *slices*, y cada uno de éstos se compone de dos *generadores de función* de cuatro entradas, un *generador de función* de dos entradas, dos registros y multiplexores programables, como se aprecia en la figura 3.3.

Tales *generadores de función*, representados por las letras  $F$ ,  $G$  y  $H$ , son LUT's (*Look Up Tables*) que pueden manipular cualquier función hasta de ocho entradas con dos salidas o dos funciones de cuatro entradas y una salida cada una. Los multiplexores pueden *enrutar* las salidas de las LUT's directamente a las salidas de los registros, es decir que éstos pueden ser utilizados para almacenar las entradas directamente, lo cual es importante para algún diseño que requiere de una gran cantidad de almacenamiento. El mismo *slice* puede ser usado para almacenar dos bits y calcular dos funciones lógicas independientes.

Ya que dentro del FPGA se cuenta con cientos o miles de *slices* según el modelo del FPGA, se puede apreciar la gran cantidad de recursos que ofrecen estos circuitos para el diseño de sistemas digitales de considerable importancia.

Por otro lado, los recursos de *enrutamiento* interno conectan entre sí a los CLB's a través de una red de conexiones, arreglada en forma de matriz. Este *enrutamiento* se realiza dentro del FPGA en forma jerárquica. Cada renglón o

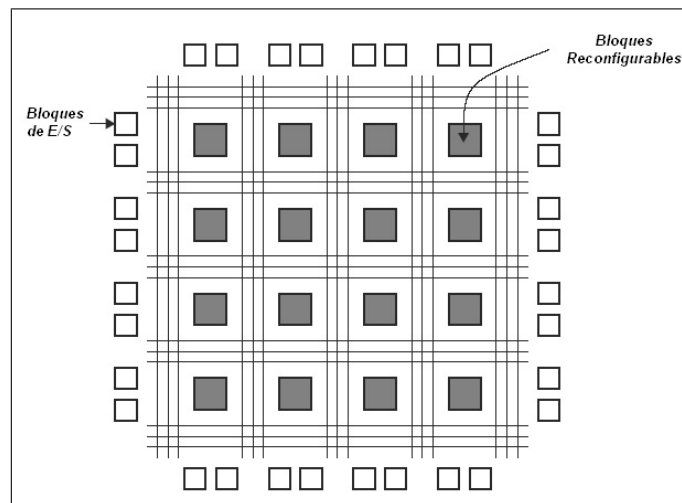


Figura 3.2: Estructura interna de un FPGA

columna de *enrutamiento* entre CLB's tiene un número de diferentes tipos de líneas. Estas pueden ser líneas simples, que se usan para conectar a CLB's adyacentes, o líneas dobles o cuádruples que interconectan a dos CLB's cualesquiera dentro del FPGA.

### 3.2.5. FPGA's y criptografía

Ya que una de las principales aplicaciones de los operadores aritméticos sobre campos finitos que desarrollamos en este trabajo es la criptografía, estudiamos algunos aspectos importantes de la relación entre los FPGA's y la criptografía basados en [65].

#### Plataformas para algoritmos criptográficos

En la figura 3.4 se muestran las plataformas posibles para la implementación de un algoritmo criptográfico.

La selección de la plataforma más adecuada depende de los siguientes criterios:

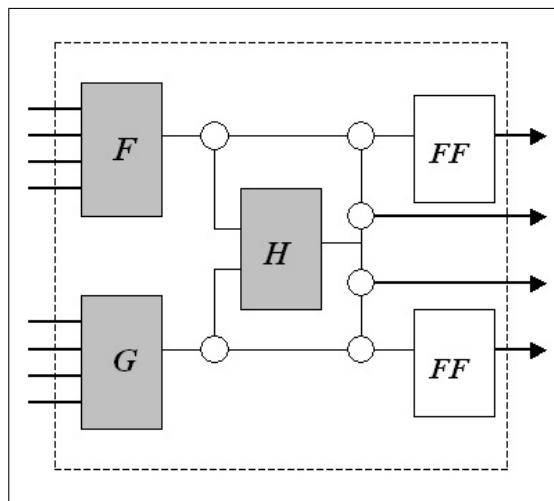


Figura 3.3: Estructura interna de un CLB

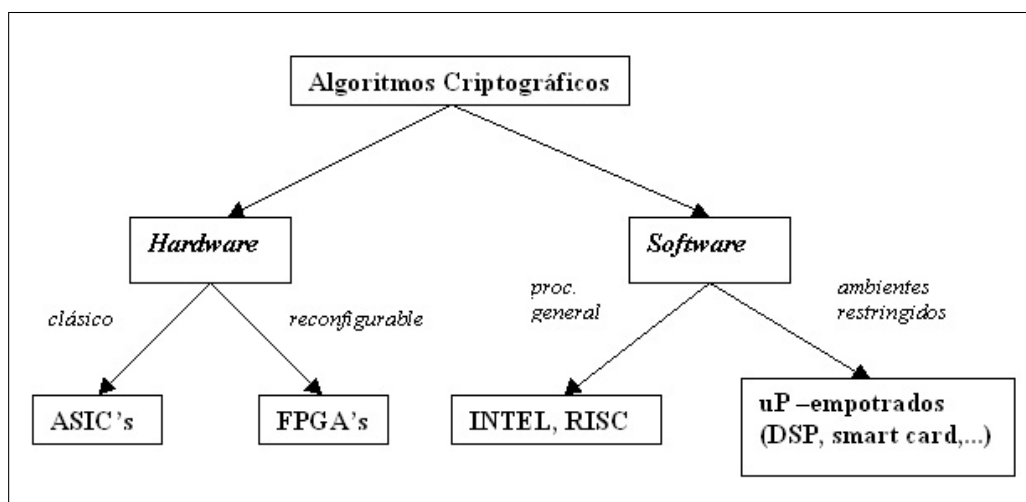


Figura 3.4: Plataformas para implementación de algoritmos criptográficos

- Desempeño algorítmico. Es de importancia primordial asegurar que en la plataforma seleccionada se tendrá el mejor desempeño del algoritmo implementado.
- Costo. Han de verificarse los costos asociados a la plataforma seleccionada que se derivarán de la fabricación de una sola unidad y del desarrollo de una fabricación en serie del producto.
- Consumo de potencia. En una implementación en hardware es importante considerar el consumo de potencia debido al alambrado interno del circuito, el cual es diferente dependiendo de la tecnología utilizada en el proceso de fabricación.
- Flexibilidad. Se ha de evaluar que la plataforma seleccionada permita las siguientes posibilidades que son importantes en un sistema criptográfico:
  - Cambio de parámetros
  - Agilidad en el manejo de llaves
  - Agilidad en el manejo del algoritmo
- Seguridad física. Al momento de seleccionar la plataforma correspondiente se deberá considerar la posibilidad de ataques a la integridad del sistema.

Tal selección dependerá de los requerimientos de la aplicación en particular. Podemos comparar gráficamente las características de las plataformas como se muestra en la figura 3.5. Donde se puede apreciar que los FPGA's combinan las ventajas del software y el hardware.

Existen dos razones fundamentales para realizar implementaciones de algoritmos criptográficos en hardware:

- Las implementaciones en hardware son más rápidas que las implementaciones en software.

	Bajo	Alto
Desempeño	<i>SW</i>	<i>FPGA ASIC</i> ⊗
Costo de desarrollo	<i>SW FPGA</i> ⊗	<i>ASIC</i>
Costo por unidad	<i>ASIC</i> ⊗	<i>SW FPGA</i>
Flexibilidad	<i>ASIC</i>	<i>FPGA SW</i> ⊗

( ⊗ ): Plataforma Ideal

Figura 3.5: Características entre plataformas

- Las implementaciones en hardware son intrínsecamente más seguras que las realizadas en software. El acceso a las llaves y la modificación de los algoritmos es considerablemente más difícil de realizar.

Por otro lado, el uso de lógica reconfigurable para la implementación de algoritmos criptográficos presenta las siguientes ventajas:

- Agilidad algorítmica. Existen diversos algoritmos criptográficos para sistemas de llave pública y privada; y la posibilidad de reprogramación de la lógica reconfigurable ofrece la oportunidad de implementar diversos algoritmos en el mismo dispositivo sin los costos que esto implicaría en el caso de un ASIC.
- Actualización y modificación de los algoritmos. Varios de los algoritmos criptográficos han sido “rotos” por atacantes de los criptosistemas. En otros casos los estándares corrientes han expirado. Estos dos casos implicarían una actualización o modificación de los algoritmos implementados en hardware. En un ASIC esta actualización sería prácticamente imposible si una gran cantidad de dispositivos fueran afectados.
- Eficiencia de las arquitecturas. El cómputo reconfigurable puede permitir

arquitecturas optimizadas para algoritmos específicos.

- Costos moderados. Como se ya ha mencionado, existe un gran ahorro en los costos de fabricación y de actualización en comparación con los ASIC's.
- Alto desempeño. Para cada aplicación específica, el uso de dispositivos reconfigurables permite la posibilidad de construir arquitecturas optimizadas y eficientes.
- Niveles flexibles de seguridad. La posibilidad de modificar el número de bits de una llave o el cambio de un polinomio irreducible permite la flexibilidad para determinar diversos niveles de seguridad.

### 3.2.6. Metodología de diseño con FPGA's

En la figura 3.6 se presenta el proceso de diseño con un FPGA. Este inicia con una *descripción* del circuito con un lenguaje de descripción por hardware como VHDL. Posteriormente usando alguna herramienta computacional como *Synopsys*, se realiza el proceso de *síntesis* cuyo resultado es un *netlist* que representa al circuito lógico que se desea diseñar. En este momento se pueden realizar pruebas al circuito haciendo uso de la *simulación* con otra herramienta como *ModelSim*. Después se realiza la acción de *enrutado* y *mapeo* del circuitpo dentro del modelo del dispositivo en particular. El resultado es un archivo del tipo *bitstream* que se usará para hacer la *implantación* física sobre el FPGA usando una tarjeta prototipo de desarrollo.

Cada vez es más atractivo el empleo de los FPGA's, ya que permiten un gran ahorro de tiempo en el proceso de diseño, fabricación y verificación, lo que en el caso de los ASIC's llega a tomar varios meses para su desarrollo. Estas son algunas razones por las que en este trabajo hemos decidido usar a los FPGA's como dispositivos lógicos para nuestros diseños.

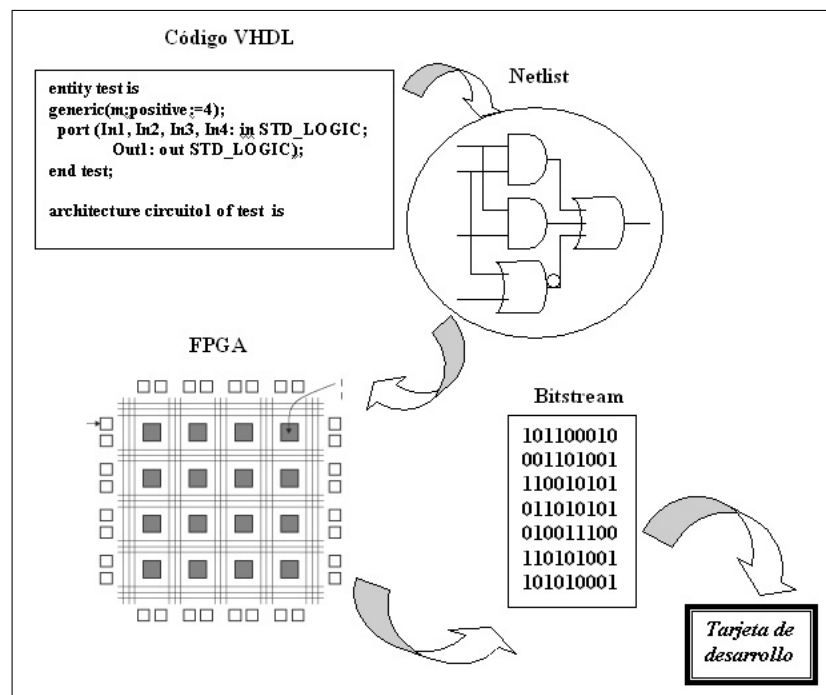


Figura 3.6: Proceso de diseño de un FPGA



## 3.3. Arreglos sistólicos

### 3.3.1. Generalidades

El término *arreglo sistólico* fué introducido en las ciencias computacionales por H.T. Kung en 1982 [43]. Un arreglo sistólico se caracteriza por las siguientes propiedades:

- Modularidad. El arreglo está integrado por elementos de proceso cuya función está definida en una forma clara y simple.
- Regularidad. El arreglo está compuesto por módulos o submódulos similares.
- Interconexión local. Cada elemento de proceso tiene conexión solo con sus vecinos.
- Alto grado de procesamiento en línea. Para arreglos que operan en forma serial, la característica de “pipeline” opera de una manera natural debido a la propia arquitectura.
- Multiprocesamiento sincronizado. Ya que todo el arreglo está bien sincronizado por una señal de reloj, cada elemento de proceso ejecuta su correspondiente proceso en forma rítmica.

Un arreglo sistólico típicamente consiste en un gran número de elementos de proceso (EP's) simples e idénticos, los cuales están interconectados únicamente entre procesadores adyacentes (*comunicación local*) y con los cuales interactúan como se ilustra en la figura 3.7. Los datos se mueven a una velocidad constante a través del arreglo pasando de un elemento de proceso al siguiente, y cada uno de los elementos realiza una cierta parte del proceso contribuyendo de manera conjunta a la culminación del proceso. Los datos que se reciben en el sistema desde una memoria externa, son pulsados rítmicamente a través de los procesadores antes de regresar a la memoria en una forma similar al flujo sanguíneo pulsado por el

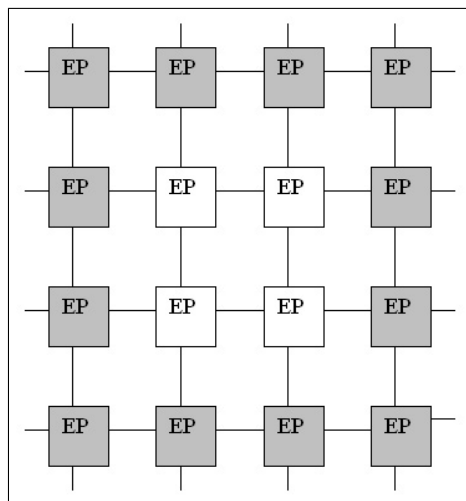


Figura 3.7: Estructura de un arreglo sistólico

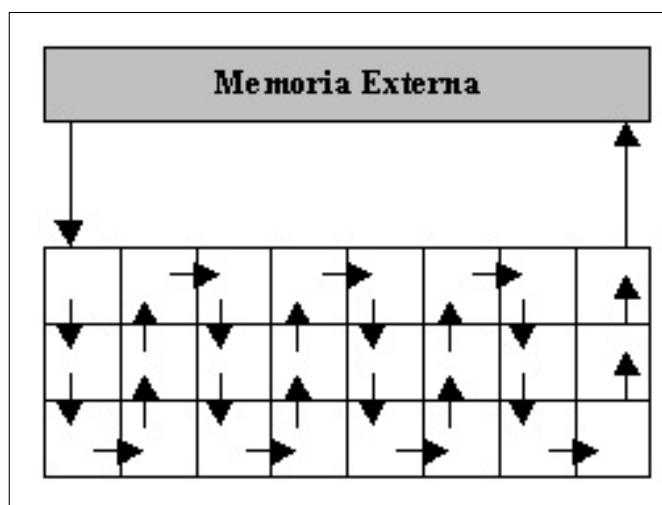


Figura 3.8: Procesamiento de datos en un arreglo sistólico

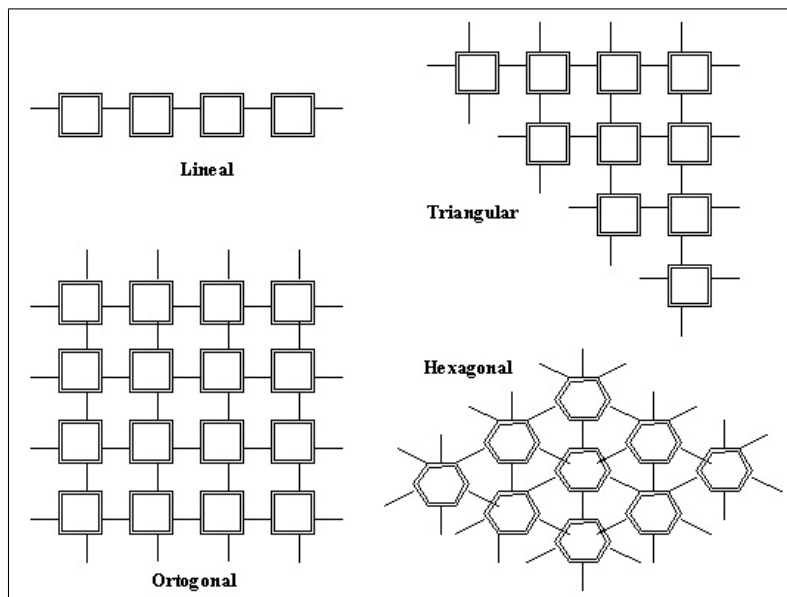


Figura 3.9: Diferentes tipos de arreglos sistólicos

corazón a través del cuerpo humano, como se muestra en la figura 3.8. La palabra *sístole*, anatómicamente, se refiere a un latido y un flujo sanguíneo pulsado por el corazón, lo cual trasladado al dominio de sistemas digitales como *sistólico*, significa un flujo de información en línea controlado por un pulso de reloj.

Se pueden formar diferentes tipos de redes sistólicas, que pueden ser en forma lineal, ortogonal o hexagonal, como se muestra en la figura 3.9. Además, en estos sistemas no son permitidos los buses de información, por lo que las únicas señales globales son las líneas de alimentación y tierra ( $V_{dd}$  y  $G_{nd}$ ), y la señal de sincronización y control que es el reloj ( $Clk$ ). Mediante las arquitecturas sistólicas se optimizan las operaciones sobre los datos ya que éstos al entrar al arreglo, son transferidos acompasadamente a través de muchos procesadores que los requieren para realizar su función, sin ningún requerimiento de memoria local. Los puertos de entrada y salida son los procesadores que se encuentran localizados en la periferia del arreglo y son los únicos que se comunican con la memoria externa, como

se ha mostrado en la figura 3.7.

Así pues, podemos resumir las ventajas de los sistemas diseñados con esta filosofía en los siguientes puntos:

- Altos rendimientos operacionales con moderadas capacidades de memoria
- Control de información sencillo y regular
- Uso de un número reducido de celdas simples y uniformes
- Comunicación local
- Alta velocidad
- Facilidad de construcción
- Expandibilidad y reconfigurabilidad

### 3.3.2. Metodología de diseño

La metodología de diseño de sistemas sistólicos se basa en una representación de *espacio-tiempo* de las estructuras operacionales y consta de tres etapas principales:

1. Selección del algoritmo. Se selecciona un algoritmo que pueda ser representado en forma *recursiva* y localmente dependiente. Esta condición es necesaria para la aplicación del método. Se hace la adaptación del algoritmo a una forma localizada y de asignamiento sencillo, y se construye su correspondiente *gráfica de dependencia*.
2. Transformación del algoritmo. Se deriva de la construcción abstracta de una *gráfica de flujo* de señales mediante la proyección de la gráfica de dependencia y se especifica un ordenamiento lineal.

3. Implantación del arreglo sistólico. Se realiza una apropiada *retemporización* a la gráfica de flujo de señales y se aplican técnicas de *cut-set* dando como resultado el arreglo esperado.

Un ejemplo clásico de arreglos sistólicos es la multiplicación de dos matrices, y en [43], [68] y [46] se explica ampliamente esta operación y la metodología de diseño.

### 3.4. Ejemplo de diseño

Con el propósito de ilustrar la metodología de diseño usando hardware reconfigurable y arreglos sistólicos, revisamos la implementación en hardware reconfigurable de un multiplicador sistólico para  $GF(2^m)$  que ha sido presentado en [38].

#### 3.4.1. Algoritmo de multiplicación

Sean  $A(x)$  y  $B(x)$  dos elementos en  $GF(2^m)$  y sea  $P(x)$  un polinomio mónico irreducible, los cuales pueden ser representados como

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \dots + a_1x + a_0 \quad (3.1)$$

$$B(x) = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \dots + b_1x + b_0 \quad (3.2)$$

$$P(x) = p^m + p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \dots + p_1x + p_0 \quad (3.3)$$

Donde  $a_j, b_j$  y  $p_j \in GF(2)$ . El producto  $A(x)B(x) \bmod P(x)$  puede ser representado como

$$\begin{aligned} C(x) &= A(x)B(x) \bmod P(x) = \left[ \sum_{j=0}^{m-1} A(x)b_jx^j \right] \bmod P(x) \\ &= \left[ \sum_{j=1}^{m-1} A(x)b_{m-i}x^{m-i} \right] \bmod P(x) + A(x)b_0 \\ &= c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + \dots + c_1x + c_0 \end{aligned} \quad (3.4)$$

Y la multiplicación puede ser realizada usando el algoritmo recursivo del cuadro 3.2, el cual se estudia en [81].

<p><b>Algoritmo para la multiplicación sobre</b>  <math>GF(2^m)</math>  Input: <math>A(x), B(x), P(x)</math>  Output: <math>C(x) = A(x)B(x) \bmod P(x)</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>c_k^{(0)} = 0, \quad 0 \leq k \leq m - 1</math></li> <li>2. <math>c_{-1}^{(i)} = 0, \quad 1 \leq i \leq m</math></li> <li>3. for <math>i := 1</math> to <math>m</math> do</li> <li>4.   for <math>k = m - 1</math> downto <math>0</math> do</li> <li>5.     <math>c_k^{(i)} = c_{m-1}^{(i-1)} p_k + b_{m-i} a_k + p_{k-1}^{(i-1)}</math></li> <li>6.   end</li> <li>7. end</li> <li>8. <math>C(x) = c^{(m)}(x)</math></li> </ol> <p>=====</p>
--

Cuadro 3.2: Algoritmo para la multiplicación sobre  $GF(2^m)$

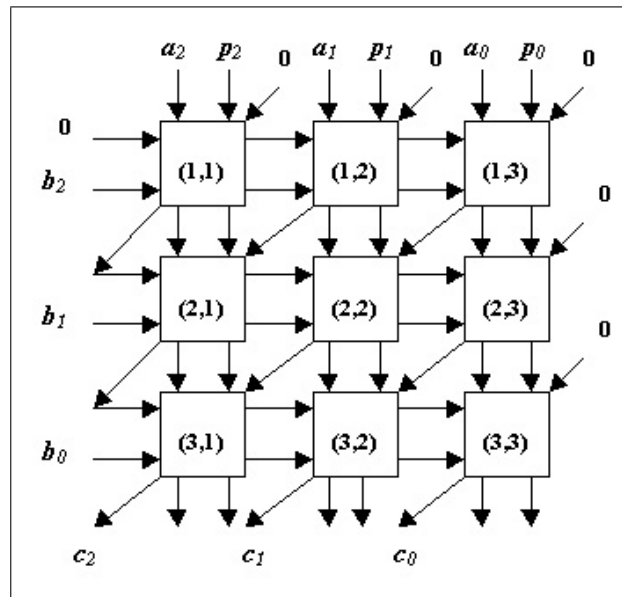


Figura 3.10: Gráfica de dependencia del multiplicador  $GF(2^3)$

### 3.4.2. Gráfica de dependencia

Tomando como base este algoritmo recursivo, se puede derivar una *Gráfica de Dependencia* (GD), como se muestra en la figura 3.10. Esta GD consiste en un arreglo de  $m \times m$  celdas básicas. En el ejemplo mostrado  $m = 3$ , y en la figura 3.11 se observa la arquitectura interna de la celda básica que realiza la operación  $c_k^{(i)} = c_{m-1}^{(i-1)}p_k + b_{m-i}a_k + p_{k-1}^{(i-1)}$ . En esta GD,  $a_j$  y  $p_j$  son introducidos en la  $(m-j)$ -ésima columna por la parte superior,  $b_j$  entra por el  $(m-j)$ -ésimo renglón por el lado izquierdo, y  $c_j$  tiene su salida por la parte inferior de la  $(m-j)$ -ésima columna después de  $m$  iteraciones.

### 3.4.3. Sistolización

La GD de la figura 3.10 puede ser retemporizada y se le pueden aplicar técnicas de *cut-set* como se estudia en [44]. Así entonces, se puede construir un arreglo

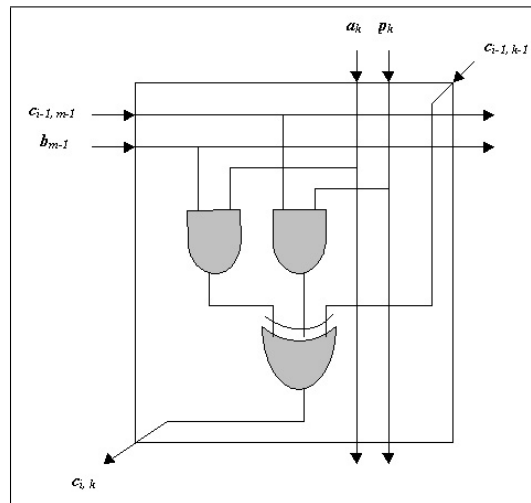


Figura 3.11: Arquitectura de la celda de la GD

sistólico y serial para calcular la multiplicación sobre  $GF(2^m)$ , como se muestra en la figura 3.12. En la figura mencionada, se agregan elementos de retardo (indicados por “•”), a la entrada y a la salida de cada celda del arreglo.

#### 3.4.4. Descripción con VHDL

En los cuadros 3.3 y 3.4 presentamos la descripción VHDL del multiplicador sistólico  $GF(2^3)$  y de la celda básica del arreglo.

#### 3.4.5. Implementación en el FPGA y resultados de la síntesis

Se hizo una estimación de la cantidad de CLB's que se ocuparían para la construcción del Multiplicador bajo las siguientes consideraciones:

- Se requiere de  $m$  celdas del tipo *celda\_mul*.



```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Multi_array is
  generic(m:positive:=3);
  Port ( Bin : in std_logic_vector(0 to m-1);
        CLK, CLR, Cin, Pin, Ain, Zin : in std_logic;
        Cout : out std_logic);
end Multi_array;
architecture Behavioral of Multy_array is
component celda.mul
  port(Ci, Pi, Ai, Bi, Zi: in std_logic;
        CLK, CLR: in std_logic;
        Co, Po, Ao, Zo: out std_logic)
end component;
signal Ctemp, Ptemp, Atemp, Ztemp: std_logic_vector(1 to m-1);
begin Arreglo:
  for I in m-1 downto 0 generate
    celm: if (I=m-1) generate
      celmm: celda.mul port map (CLK=>CLK, CLR=>CLR, Ci=>'0',
        Pi=>Pin, Ai=>Ain, Bi=>Bin(I), Zi=>Zin, Co=>Ctemp(I));
    end generate celm;
    celx: if (I<=m-1 and I>0) generate
      celxx: celda.mul port map (CLK=>CLK, CLR=>CLR,
        Ci=>Ctemp(I+1), Pi=>Ptemp(I+1), Ai=>Atemp(I+1),
        Bi=>Bin(I), Zi=>Ztemp(I+1), Co=>Ctemp(I),
        Po=>Ptemp(I), Ao=>Atemp(I), Zo=>Ztemp(I));
    end generate celx;
    cel0: if (I=0) generate
      cel00: celda.mul port map (CLK=>CLK, CLR=>CLR,
        Ci=>Ctemp(I+1), Pi=>Ptemp(I+1), Ai=>Atemp(I+1),
        Bi=>Bin(I), Zi=>Ztemp(I+1), Co=>Cout);
    end generate cel0;
  end generate Arreglo;
end Behavioral;

```

Cuadro 3.3: Descripción VHDL del Multiplicador

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity celda_mul
  Port ( CLK, CLR: in STD_LOGIC;
        Ci, Pi, Ai, Bi, Zi : in STD_LOGIC;
        Co, Po, Ao, Zo: out STD_LOGIC);
end celda_mul;

architecture celda_mul_arch of celda_mul is

begin
process(CLK, CLR)
variable Mi,Bm;
begin
  if (CLKévent and CLK ='1') then
    if(CLR='0') then
      Co<='0'; Po<='0'; Ao<='0'; Zo<='0'; Co<='0';
    elsif Zi='1'then
      Co <= (Ci and Zi) xor (Pi and Mi) xor (Ai and Bm);
    end if;
    Po<=Pi; Ao<=Ai; Zo<=Zi; Bo<=Bi;
  end if;
end process;
end Behavioral;

```

Cuadro 3.4: Descripción VHDL de la celda del arreglo

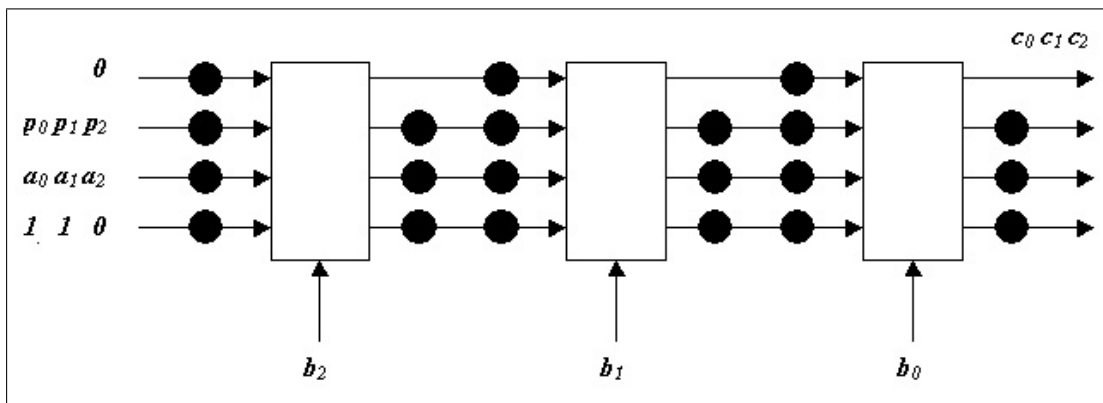


Figura 3.12: Arreglo sistólico del multiplicador serial  $GF(2^3)$

- Cada celda del arreglo sistólico requiere de 4 compuertas y 7 Flip-Flops (FF). Básicamente la cantidad de FF determina la cantidad de *slices* que se necesitan para la construcción de cada celda.
- Cada *slice* contiene 2 FF y 2 generadores de función. Estimamos que se ocuparían 4 *slices* por cada celda.
- En el peor de los casos, se desperdiciarían 1 FF y 4 generadores de función de cada 4 *slices*. Así pues, el número total de *slices* requeridos por el multiplicador sería de  $4m$ .

Una vez realizada la simulación funcional del circuito, el proceso de síntesis para diferentes valores de  $m$  (orden del campo) ha producido los resultados mostrados en el cuadro 3.5. Se ha usado el sintetizador del paquete computacional ISE4.1i de la compañía Xilinx y se ha considerado el uso de un FPGA Virtex XCV300 que contiene 6144 *slices*.

<b>m</b>	<b>No. de slices</b>	<b>% consumo</b>
4	16	0.06
16	64	0.2
64	256	4
193	772	12

Cuadro 3.5: Complejidad de espacio del Multiplicador  $GF(2^m)$ 

### 3.5. Resumen

Son tres los elementos fundamentales que hemos utilizado en el procedimiento de diseño: arreglos sistólicos, VHDL y FPGA's.

A manera de resumen enumeramos los pasos que hemos seguido en este proceso:

1. Definición de la operación sobre  $GF(2^m)$  que se pretende implementar.
2. Análisis y selección de un algoritmo que se puede expresar en forma recursiva.
3. Definición del arreglo sistólico que se espera construir.
4. Descripción *parametrizable* con VHDL del arreglo sistólico. Se describe en forma sencilla la función de cada elemento de proceso y se explota el uso del comando *generate* de VHDL para generar todo el arreglo.
5. Simulación y síntesis del diseño.
6. Optimización del diseño usando las herramientas del paquete computacional.
7. Implementación en el FPGA.

En los años recientes, se han publicado en la literatura diversas propuestas de arquitecturas para operadores básicos sobre campos finitos que utilizan arreglos sistólicos, tales como [28], [91], [13], [93] y [20].

# Capítulo 4

## Implementaciones realizadas

En este apartado presentamos las implementaciones que hemos realizado. Es decir, un divisor y multiplicador sistólico y serial, un exponenciador y un multiplicador del tipo serial/paralelo para  $GF(2^m)$ . En cada caso, se presentan los algoritmos que se han utilizado, las arquitecturas que se derivan de tales algoritmos, y los resultados que se han obtenido de los procesos de síntesis e implementación en el FPGA.

### 4.1. Implementación en FPGA de un Divisor y Multiplicador sistólico y serial

Según Hasan y Bhargava [30], la división sobre  $GF(2^m)$  se puede reducir a la solución de un sistema de ecuaciones lineales. En comparación con otros algoritmos, este método requiere de menos tiempo de cálculo y usa estructuras muy simples de cómputo. Además, esta arquitectura no depende del polinomio irreducible asociado al campo, pues éste es suministrado como un argumento de entrada al circuito.

Ya que el divisor se ha construido por medio de dos estructuras diferentes, hemos aprovechado una de ellas como base para implementar un multiplicador

serial, con lo que hemos pretendido contar con dos de las operaciones básicas sobre campos finitos.

#### 4.1.1. Algoritmo de división

Sean  $A(\alpha), B(\alpha)$  dos elementos sobre  $GF(2^m)$ , con  $A(\alpha) \neq 0$ . El cociente  $B(\alpha) = C(\alpha)/A(\alpha)$  puede ser calculado como sigue:  $C(\alpha) = A(\alpha)B(\alpha)$ , con  $A(\alpha) = \mathbf{a}^T \mathcal{A}^{(m)}$ ,  $B(\alpha) = \mathbf{b}^T \mathcal{A}^{(m)}$ ,  $C(\alpha) = \mathbf{c}^T \mathcal{A}^{(m)}$ , y donde  $\mathcal{A}^{(m)} = (\alpha^i)_{i=0}^{m-1}$  es la matriz cuyas columnas representan a las potencias de un generador  $\alpha$  de  $GF(2^m)$ , respecto a la base polinomial correspondiente a  $\alpha$  (de hecho  $\mathcal{A}^{(m)}$  coincide con la matriz identidad), y  $\mathbf{a}, \mathbf{b}$  y  $\mathbf{c}$  son los vectores-coordenadas de  $A(\alpha), B(\alpha)$  y  $C(\alpha)$  relativos a la base  $\mathcal{A}^{(m)}$ . Entonces

$$\mathbf{c}^T \mathcal{A}^{(m)} = (\mathbf{a} * \mathbf{b})^T \mathcal{A}^{(2m-1)} = (\mathbf{a} * \mathbf{b})^T \mathcal{P}^T \mathcal{A}^{(m)}$$

donde  $\mathbf{a} * \mathbf{b}$  es la convolución de  $\mathbf{a}$  con  $\mathbf{b}$  y  $\mathcal{P} = (p_{ij})_{i=0, \dots, m-1}^{j=0, \dots, 2m-2}$  es la matriz de coordenadas de las potencias de  $\alpha$  en términos de la base seleccionada  $\mathcal{A}^{(m)}$ . Además  $\mathcal{A}^{(2m-1)} = \mathcal{P}^T \mathcal{A}^{(m)}$ . Así,  $\mathbf{c} = \mathcal{P}(\mathbf{a} * \mathbf{b})$ , donde para cada  $i = 0, \dots, m-1$ ,  $c_i = \sum_{k=0}^{2m-2} p_{ik} \sum_{\ell+j=k} a_\ell b_j$ , entonces

$$\mathbf{c} = \mathcal{Q}\mathbf{b} \tag{4.1}$$

donde  $\mathcal{Q} = (q_{ij})_{i=0, \dots, m-1}^{j=0, \dots, m-1}$  tiene entradas  $q_{ij} = \sum_{k=0}^{m-1} p_{i, k+j} a_k$ . De esta forma, la división puede ser realizada por medio de la solución de un sistema de  $m$  ecuaciones lineales sobre  $GF(2^m)$ [30].

#### 4.1.2. Arquitectura del Divisor

El diseño general del divisor serial  $GF(2^m)$  es mostrado en la figura 4.1. Las señales de entrada son las siguientes:

- $(g_i)_{i=0}^{m-1}$  es el vector de coeficientes del polinomio irreducible,

#### 4.1. IMPLEMENTACIÓN EN FPGA DE UN DIVISOR Y MULTIPLICADOR SISTÓLICO Y SERIAL87

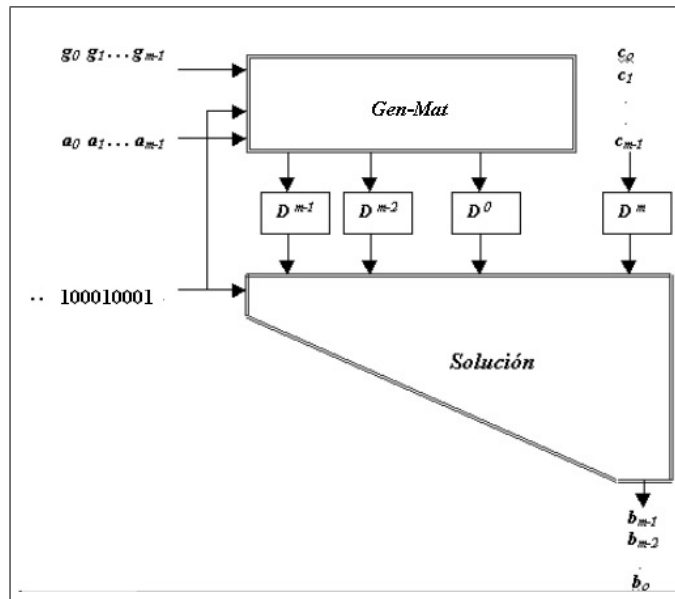


Figura 4.1: Arquitectura del Divisor para  $GF(2^m)$

- $(q_i)_{i \geq 0}$  es una secuencia de control (indicando la longitud de  $m$ ),
- $(a_i)_{i=0}^{m-1}$  son los coeficientes del divisor  $GF(2^m)$  y
- $(c_i)_{i=0}^{m-1}$  son los coeficientes del dividendo.

La señal de salida proporciona los coeficientes del cociente  $(b_i)_{i=0}^{m-1}$ . El primer trabajo del divisor es la generación de los coeficientes de la matriz  $Q$  en la ecuación (4.1), lo que hace a lo largo de  $m$ -ciclos o  $m$ -pulsos de reloj. Posteriormente, el sistema de ecuaciones lineales es resuelto en un proceso con “*pipeline*” requiriendo, como se puede ver,  $4m - 1$  ciclos de reloj. El tiempo total requerido para el cálculo de una división es así de  $5m - 1$  ciclos de reloj. El grupo de celdas intermedias  $D_i$ 's son *flip-flops* utilizados para sincronización entre ambas estructuras principales Gen-Mat y Solución. Se describe cada componente enseguida:

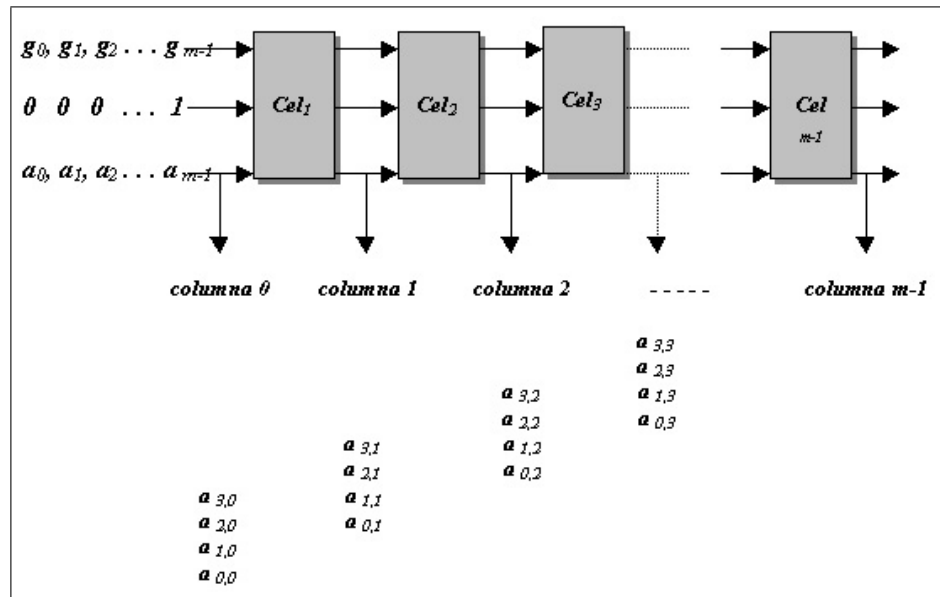


Figura 4.2: Estructura del arreglo Gen-Mat

### Arreglo generador de la matriz

El arreglo **Gen-Mat** que genera los coeficientes de la matriz se muestra en la figura 4.2. Este consiste en  $m-1$  celdas secuencialmente arregladas y denominadas  $Cel_1, Cel_2, Cel_3, \dots, Cel_{m-1}$ . Cada  $Cel_i$  es del tipo **Cel01**, y se muestra en la figura 4.3, en la que los bloques FF representan *flip-flops*. La operación que realiza esta celda se describe en el pseudocódigo mostrado en el cuadro 4.1.

Las coordenadas de  $\mathbf{a}$  se introducen serialmente en  $Cel_1$  iniciando con el “*dígito más significativo*”  $a_{m-1}$ . En la salida, la columna 0 es exactamente una copia de  $\mathbf{a}$ . Las columnas de la 1 a la  $m-1$  son las salidas respectivas de las celdas  $Cel_1, Cel_2, Cel_3, \dots, Cel_{m-1}$ . Cada celda requiere de dos ciclos de reloj adicionales para desplegar el primer elemento de cada columna en la salida. La salida de la celda  $Cel_{j-1}$  es introducida dentro del siguiente procesador  $Cel_j$ . El código VHDL del circuito **Gen-Mat** se muestra en el apéndice A.



4.1. IMPLEMENTACIÓN EN FPGA DE UN DIVISOR Y MULTIPLICADOR SISTÓLICO Y SERIAL89

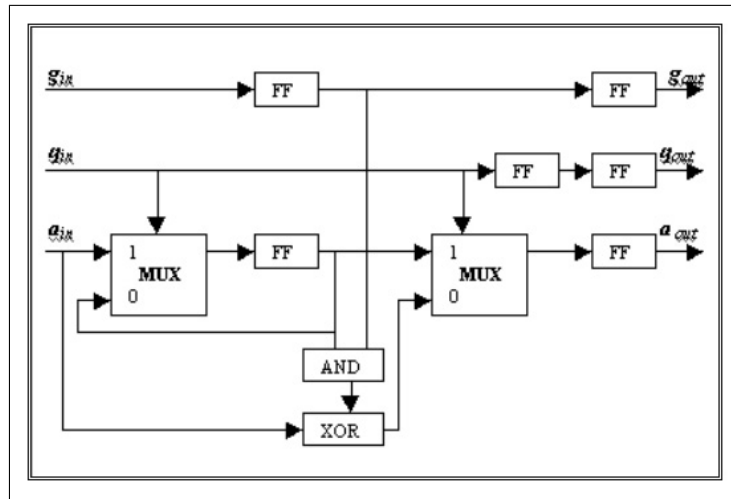


Figura 4.3: Diagrama de la celda Ce101

```

if  $q_{in} := 1$  then
  {  $a_{out} := r; r := a_{in}$  };
else
  {  $a_{out} := g_{temp}r + a_{in}$  };
   $g_{out} := g_{temp}; g_{temp} := g_{in}$ ;
   $q_{out} := q_{temp}; q_{temp} := q_{in}$ ;

```

Cuadro 4.1: Descripción de la celda Ce101.

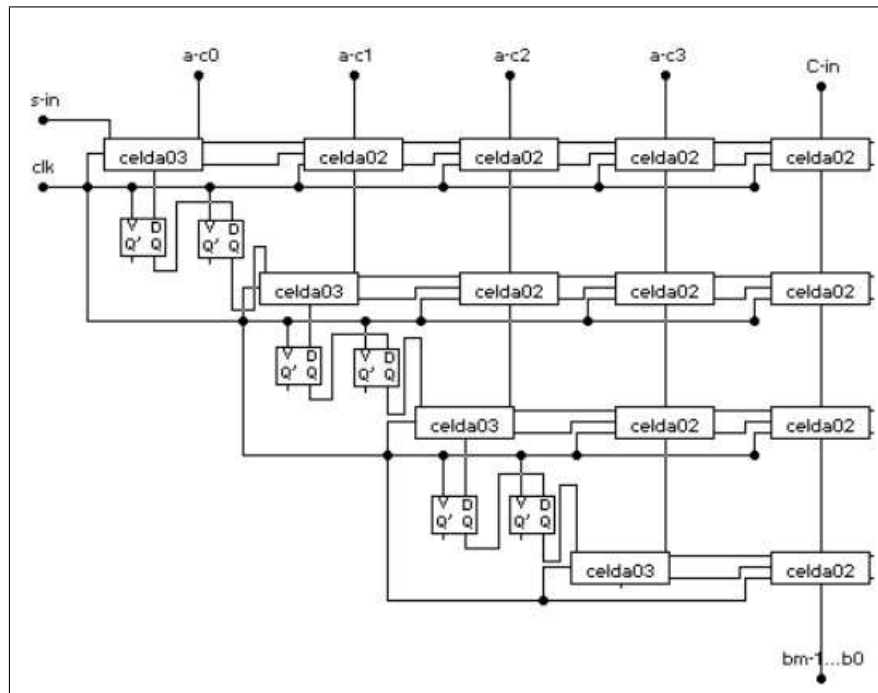


Figura 4.4: Estructura del arreglo Solución

### Arreglo que resuelve el sistema de ecuaciones lineales

La estructura propuesta que usa el método de triangulación de Gauss-Jordan para resolver el sistema de ecuaciones (4.1) se muestra en la figura 4.4.

Las señales indicadas en la figura son las siguientes:

- $ac0...ac.[m - 1]$  : Columnas de la matriz
- $C_{in}$  : Dividendo
- $b.[m - 1]...b0$  : Solución de la ecuación (4.1)
- $S_{in}$  : Señal de control
- $Clk$  : Reloj

#### 4.1. IMPLEMENTACIÓN EN FPGA DE UN DIVISOR Y MULTIPLICADOR SISTÓLICO Y SERIAL91

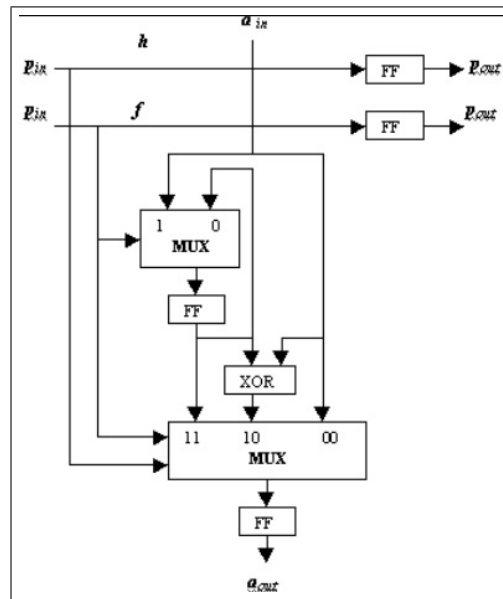


Figura 4.5: Diagrama de Cel102

Se usan aquí dos tipos de celdas, Cel102 y Cel103. Los diagramas correspondientes de Cel102 y Cel103 se muestran en las figuras 4.5 y 4.6.

La operación de cada una se describe en pseudocódigo en el cuadro 4.2. Los *flip-flops* son usados para propósitos de sincronización. El arribo inicial de las entradas a la matriz es determinado por la señal de control  $S_{in}$ . Las entradas del vector solución  $\mathbf{b}$  son salidas de la  $(m - 1)$ -ésima celda de tipo Cel102. La  $m$ -ésima coordenada aparece en el ciclo  $3m - 1$ . Posteriormente, una nueva coordenada aparece cada ciclo de reloj. Así, en este arreglo que soluciona el sistema de ecuaciones, se obtiene una salida total en  $4m - 1$  ciclos de reloj. Es decir, que junto con los  $m$  ciclos usados por el primer arreglo Gen-Mat se tienen  $5m - 1$  ciclos de reloj para realizar una división completa. El código VHDL de la estructura Solución que resuelve el sistema de ecuaciones (4.1), se muestra en el apéndice A.

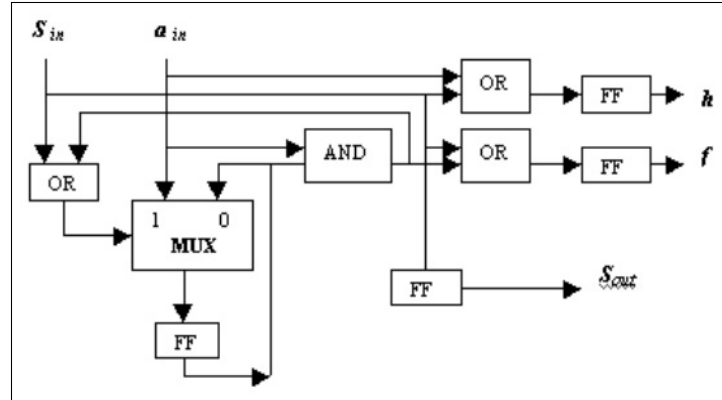


Figura 4.6: Diagrama de Cel103

```

if  $s_{in} := 1$  then
  {  $P_{out} := (1, 1)$ ;  $r := a_{in}$  }
else
  case( $r, a_{in}$ ) of
    (0,0): $P_{out} := (0, 0)$ ;
    (0,1): $P_{out} := (1, 1)$ ;
    (1,0): $P_{out} := (0, 0)$ ;
    (1,1): $P_{out} := (1, 0)$ ;
  endcase;
 $s_{out} := s_{in}$ ;

```

(a) Operación de Cel102.

```

case ( $P_{in}$ ) of
  (0,0): $a_{out} := a_{in}$ ;
  (1,0): $a_{out} := r + a_{in}$ ;
  (1,1): $a_{out} := r$ ;  $r := a_{in}$ ;
 $P_{out} := P_{in}$ ;

```

(b) Operación de Cel103

Cuadro 4.2: Descripción de las celdas Cel102 y Cel103

### 4.1.3. Análisis de complejidad

Como se ha explicado ya en la descripción de las dos estructuras que integran el divisor, la complejidad de tiempo del circuito es de  $5m - 1$  ciclos de reloj, lo cual es la suma de los  $m$ -ciclos que requiere la primera estructura **Gen-Mat** para entregar sus resultados y los  $4m - 1$  ciclos que la segunda estructura **Solución** necesita para entregar su correspondiente resultado.

En cuanto a la complejidad de espacio se hicieron las siguientes consideraciones:

- La primera estructura **Gen-Mat** es un arreglo lineal de  $m - 1$  celdas del tipo Cel01.
- Cada celda Cel01 está integrada por 1 compuerta AND, 1 compuerta XOR, 2 multiplexores de 2 entradas y una salida, y 6 registros.
- Así pues, esta primera estructura ocupa  $m - 1$  compuertas AND,  $m - 1$  compuertas XOR,  $2(m - 1)$  multiplexores 2 a 1 y  $6(m - 1)$  registros.
- La segunda estructura **Solución** es una estructura triangular y está conformada por dos tipos de celdas:  $(m^2 + m)/2$  celdas Cel02 y  $m$  celdas Cel03.
- La celda Cel02 contiene 1 compuerta XOR, un multiplexor 2 a 1, un multiplexor 3 a 1 y 4 registros.
- La celda Cel03 contiene 1 compuerta AND, 3 compuertas OR, un multiplexor de 2 a 1 y 4 registros.
- Además se ocupan  $2m - 2$  registros adicionales.
- Entonces, la estructura **Solución** ocupa  $0.5m^2 + 0.5m$  compuertas XOR,  $m$  compuertas AND,  $3m$  compuertas OR,  $0.5m^2 + 1.5m$  multiplexores 2 a 1,  $0.5m^2 + 0.5m$  multiplexores 3 a 1 y  $2m^2 + 8m - 2$  registros.

Operadores	Cantidades
Registros	$2.5m^2 + 14m - 2$
Compuertas AND	$2m - 1$
Compuertas OR	$3m$
Compuertas XOR	$0.5m^2 + 1.5m - 1$
Multiplexores 2 a 1	$0.5m^2 + 3.5m - 2$
Multiplexores 3 a 1	$0.5m^2 + 0.5m$

Cuadro 4.3: Complejidad de espacio del Divisor  $GF(2^m)$ 

Operadores	Número de CLB's
Registros	$1.25m^2 + 7m - 1$
Operaciones Lógicas	$0.75m^2 + 5.25m - 2$

Cuadro 4.4: Requerimientos de espacio en el FPGA XCS4010

- Adicionalmente, se requiere de un arreglo de  $m^2/2$  registros para sincronizar la operación entre las dos estructuras. En el cuadro 4.3 se resume este análisis.
- Considerando el uso de un FPGA XCS4010 para la implementación y sabiendo que cada CLB contiene dos generadores de función y dos registros, se hizo un cálculo previo del consumo de recursos en el FPGA como se muestra el cuadro 4.4.

Como se puede observar, los requerimientos de espacio crecen de una forma cuadrática con respecto a  $m$ , lo cual nos ha limitado al uso de órdenes del campo finito de hasta 12 como se indica en la sección de resultados.

$\alpha^0 = 1$	$\alpha^4 = \alpha^3 + 1$	$\alpha^8 = \alpha^3 + \alpha^2 + \alpha$	$\alpha^{12} = \alpha + 1$
$\alpha^1 = \alpha$	$\alpha^5 = \alpha^3 + \alpha + 1$	$\alpha^9 = \alpha^2 + 1$	$\alpha^{13} = \alpha^2 + \alpha$
$\alpha^2 = \alpha^2$	$\alpha^6 = \alpha^3 + \alpha^2 + \alpha + 1$	$\alpha^{10} = \alpha^3 + \alpha$	$\alpha^{14} = \alpha^3 + \alpha^2$
$\alpha^3 = \alpha^3$	$\alpha^7 = \alpha^2 + \alpha + 1$	$\alpha^{11} = \alpha^3 + \alpha^2 + 1$	$\alpha^{15} = 1$

Cuadro 4.5: Representación de  $GF(2^4)$  usando el polinomio irreducible  $P(x)$ .

#### 4.1.4. Resultados

Todas las pruebas han sido hechas en el campo  $GF(2^4)$  representado por el polinomio irreducible  $P(x) = x^4 + x^3 + 1$ . Ya que  $\alpha$  es una raíz de  $P(x)$ , el grupo cíclico multiplicativo de  $GF(2^4)$  es representado como se muestra en el cuadro 4.5, y  $\alpha^4 = \alpha^3 + 1$  es un generador. Consideremos a  $g_{in} = [1\ 0\ 0\ 1]$  como el polinomio irreducible y sea  $q_{in} = [1\ 0\ 0\ 0]$  una señal de control. Como un ejemplo ilustrativo tomemos a:

$$\begin{aligned} \mathbf{a}_{in} &= \alpha^6 = \alpha^3 + \alpha^2 + \alpha + 1 = [1\ 1\ 1\ 1] \\ \mathbf{c}_{in} &= \alpha^{14} = \alpha^3 + \alpha^2 = [1\ 1\ 0\ 0] \end{aligned}$$

como dividendo y divisor respectivamente, cuyo cociente evidentemente es  $\mathbf{b} = \alpha^8 = [1\ 1\ 1\ 0]$ . Así, las matrices  $\mathcal{Q}$  y  $\mathcal{P}$  son:

$$\mathcal{P} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathcal{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

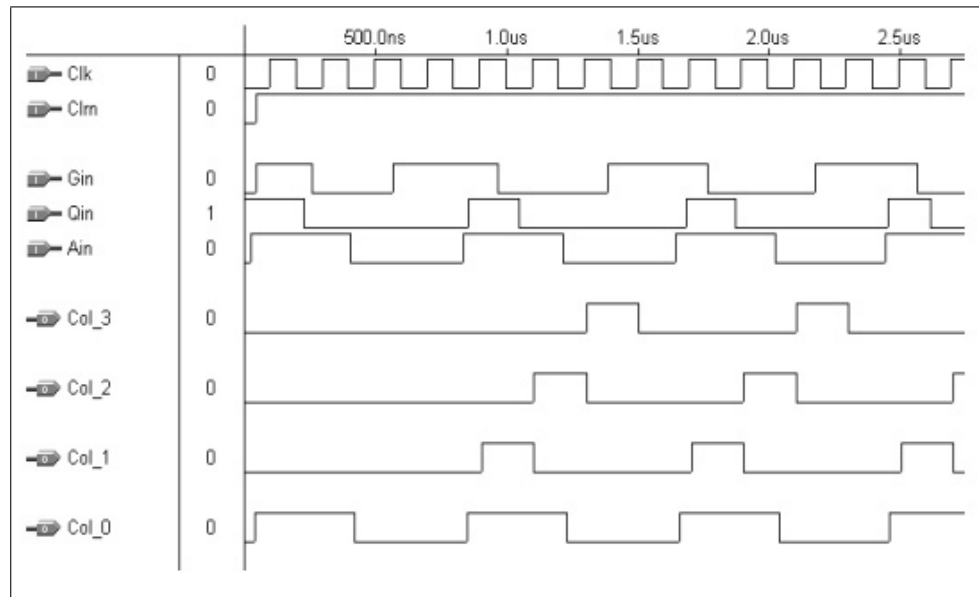


Figura 4.7: Simulación de Gen-Mat

Con este ejemplo se ilustra el comportamiento de la estructura Gen-Mat como se muestra en la figura 4.7. Es importante indicar que la primera columna de la matriz, indicada en la gráfica como *column 0*, inicia en la subida del pulso 1, la segunda columna, *column 1*, en el pulso 2, la tercera columna en el pulso 4 y la última columna en el pulso 6. El comportamiento de la estructura Solución se presenta en la figura 4.8, donde la señal B\_out = [1 1 1 0] inicia su salida después del pulso 12.

En la figura 4.9 se presenta el circuito completo del Divisor, en el que se incluye la etapa de sincronización integrada por un arreglo de *flip-flops*, y en la figura 4.10 se puede observar la simulación del divisor para los valores de entrada que se han mencionado previamente, es decir:

$$\mathbf{a}_{in} = \alpha^6 = \alpha^3 + \alpha^2 + \alpha + 1 = [1 \ 1 \ 1 \ 1]$$

$$\mathbf{c}_{in} = \alpha^{14} = \alpha^3 + \alpha^2 = [1 \ 1 \ 0 \ 0]$$

Se puede observar la salida  $\mathbf{b} = \alpha^8 = [1 \ 1 \ 1 \ 0]$  a partir del pulso 16.



#### 4.1. IMPLEMENTACIÓN EN FPGA DE UN DIVISOR Y MULTIPLICADOR SISTÓLICO Y SERIAL97

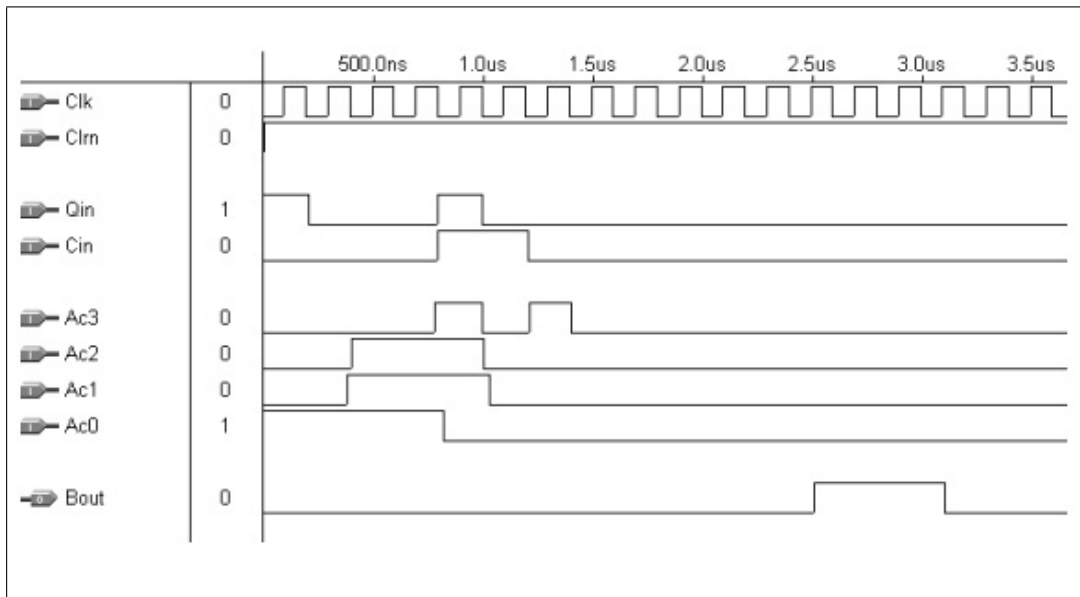


Figura 4.8: Simulación de Solución

Orden del campo	CLB's	Registros
$GF(2^4)$	97 de 400 (24%)	132 de 800 (16%)
$GF(2^8)$	228 de 400 (57%)	346 de 800 (43%)
$GF(2^{12})$	388 de 400 (97%)	622 de 800 (77%)

Cuadro 4.6: Consumo de recursos en el FPGA XCS4010

En cuanto a los resultados del proceso de síntesis en el FPGA XCS4010, el cuadro 4.6 nos muestra los recursos utilizados en el FPGA en función del valor de  $m$ , y podrá observarse que el orden más alto del campo finito que se pudo utilizar fué de 12, el que consumía prácticamente la totalidad de recursos del FPGA. En el cuadro 4.7 comparamos nuestro divisor con dos trabajos presentados previamente.

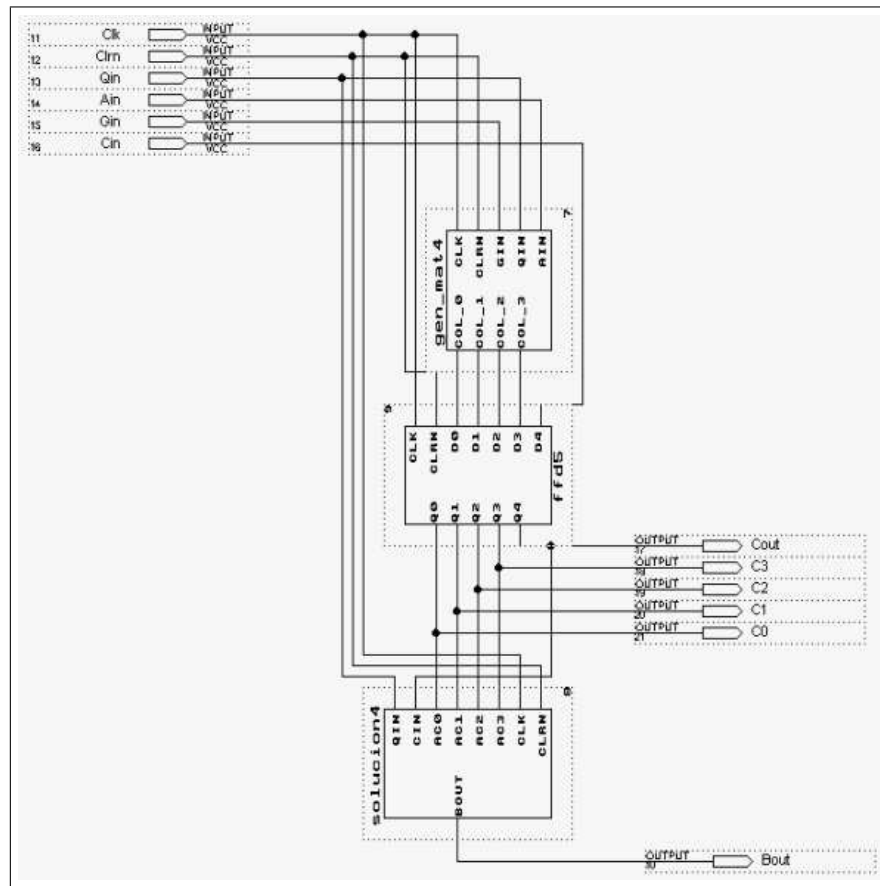


Figura 4.9: Circuito del Divisor

-	[92]	[3]	Aquí
<b>Operación</b>	Inversión	Inversión	División
<b>No. de registros</b>	$O(m)$	$O(m)$	$O(m^2)$
<b>No. de compuertas</b>	$O(m^3)$	$O(m)$	$O(m^2)$
<b>Señales de control</b>	2	4	1
<b>Independencia del pol. irreducible</b>	No	Si	Si
<b>Tiempo de retardo</b>	$O(m \log m)$	$O(m^2)$	$O(m)$

Cuadro 4.7: Comparación del divisor  $GF(2^m)$

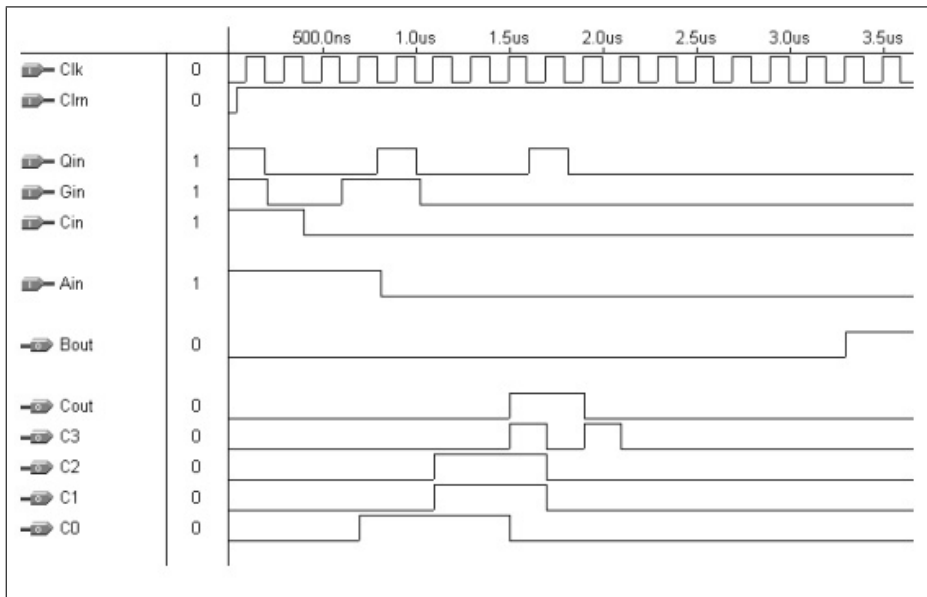


Figura 4.10: Simulación del Divisor

#### 4.1.5. Algoritmo de multiplicación

Para cualesquiera dos elementos  $A(\alpha), B(\alpha) \in GF(2^m)$ , el producto  $C(\alpha) = A(\alpha)B(\alpha)$  puede ser calculado directamente usando la ecuación (4.1) y la matriz  $Q$ . Se requiere así de los siguientes dos pasos:

1. Cálculo de los coeficientes de la matriz  $Q$ .
2. La multiplicación Matriz-vector para obtener las coordenadas del producto  $C(\alpha) = A(\alpha)B(\alpha)$ .

El primer paso coincide con el mismo del divisor. Para el segundo paso, una relación recursiva trivial es utilizada. Así, de la ecuación (4.1), para  $i = 0, \dots, m - 1$  se tiene que:

$$c_i = \sum_{j=0}^{m-1} q_{ij}b_j = \left( \sum_{j=0}^{m-2} q_{ij}b_j \right) + q_{i,m-1}b_{m-1} \quad (4.2)$$

$d_{out} := d_{in};$ $c_{out} := c_{in} + r a_{in};$ $b_{out} := b_{temp};$ $b_{temp} := b_{in};$ <b>if</b> $d_{in} := 1$ <b>then</b> $r := b_{in};$
--

Cuadro 4.8: Descripción de la celda mul

esto es:

$$\begin{aligned} c_i^{(0)} &= 0 \\ c_i^{(k)} &= c_i^{(k-1)} + q_{i,k-1} b_{k-1} \end{aligned} \quad (4.3)$$

Obviamente,  $\forall i : c_i = c_i^{(m-1)}$ .

#### 4.1.6. Arquitectura del Multiplicador

En la figura 4.11 se muestra la arquitectura del **Multiplicador**. Usamos la estructura **Gen-Mat** del Divisor para realizar el primer paso. Para el segundo paso, se propone un arreglo lineal de procesadores  $mul_0, mul_1, \dots, mul_{m-1}$ , como se muestra en la figura 4.12.

Las coordenadas de  $\mathbf{b}$  entran en  $mul_0$ , con el “bit más significativo” llegando primero. Como las coordenadas de  $\mathbf{b}$  pasan a través de los procesadores, cada bit  $b_i$  es almacenado en el registro interno de  $mul_i$ . Este procesador identifica a  $b_i$  con la ayuda de la señal de control  $d$ . Una vez que  $b_i$  es almacenado en  $mul_i$ ,  $mul_i$  simplemente hace las operaciones de multiplicación y suma sobre  $GF(2)$ . Las coordenadas del producto  $\mathbf{c}$  inician su salida de  $mul_{m-1}$  en el ciclo  $2m$  a un promedio de una coordenada por ciclo. Esto proporciona un tiempo de cómputo de  $3m-1$  ciclos. Entre ambas estructuras está el bloque denominado **Delay**, que es un arreglo de *flip-flops* para sincronizar el paso de las señales entre ambas estructuras. El diagrama de la celda mul se presenta en la figura 4.13, y su operación se describe en el pseudocódigo en el cuadro 4.8, de acuerdo con la ecuación 4.3.

4.1. IMPLEMENTACIÓN EN FPGA DE UN DIVISOR Y MULTIPLICADOR SISTÓLICO Y SERIAL101

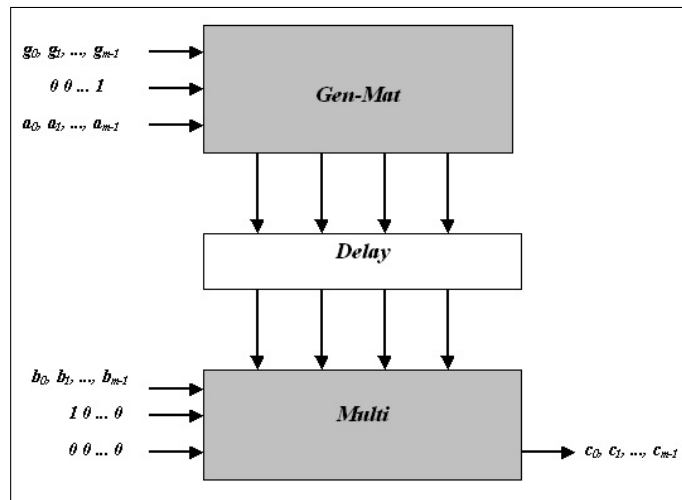


Figura 4.11: Arquitectura del Multiplicador

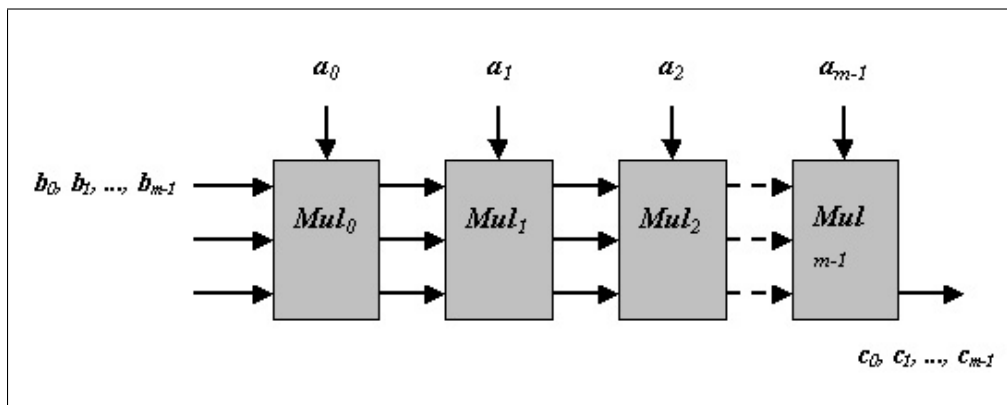


Figura 4.12: Estructura para la multiplicación matricial

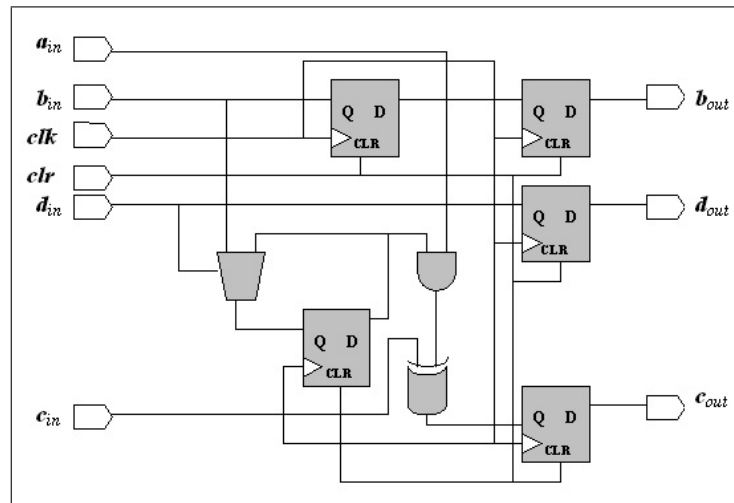


Figura 4.13: Diagrama de la celda mul

#### 4.1.7. Análisis de complejidad

La complejidad de tiempo del multiplicador es de  $3m - 1$  ciclos de tiempo como se ha explicado en el apartado anterior, y hacemos las siguientes consideraciones para calcular la complejidad de espacio:

- Conocemos ya la complejidad de espacio de las estructuras **Gen-Mat** y **Delay**.
- La segunda estructura es un arreglo lineal de  $m - 1$  celdas del tipo mul.
- Cada celda contiene 1 multiplexor 2 a 1, 1 compuerta AND, 1 compuerta XOR y 4 registros.
- Así pues esta segunda estructura está construída con  $m - 1$  multiplexores de 2 a 1,  $m - 1$  compuertas AND,  $m - 1$  compuertas XOR y  $4(m - 1)$  registros. Esto se resume en los cuadros 4.9 y 4.10.

El trabajo desarrollado en esta sección lo hemos presentado en [21] y se ha publicado en [20].

Operadores	Cantidades
Registros	$0.5m^2 + 10m - 10$
Compuertas AND	$2m - 2$
Compuertas XOR	$2m - 2$
Multiplexores 2 a 1	$3m - 3$

Cuadro 4.9: Complejidad de espacio del Multiplicador  $GF(2^m)$ 

Operadores	Número de CLB's
Registros	$0.25m^2 + 5m - 5$
Operaciones Lógicas	$3.5m - 3.5$

Cuadro 4.10: Requerimientos de espacio en el FPGA XCS4010.

## 4.2. Implementación de un Exponenciador para $GF(2^m)$

Como se ha estudiado en la sección 2.5, el método de elevación al cuadrado y multiplicación, conocido como *método binario* [39], es una técnica aceptada generalmente para el cálculo de una exponenciación en campos finitos, es decir:  $R = A^e$ . Como se ha mencionado anteriormente, existen dos versiones para este algoritmo: *MSB-first* y *LSB-first*. Cada una depende de cuál es el primer bit del exponente que se examina en el algoritmo. En este trabajo hemos utilizado la versión *LSB-first* ya que nos permite el desarrollo de una estructura que usa dos multiplicadores operando en paralelo para realizar el cálculo de la exponenciación. Tales multiplicadores han sido diseñados en [21]. Si comparamos nuestra estructura con las presentadas en [36] y [85], apreciaremos el ahorro en hardware que se obtiene.

<p><b>Algoritmo Exponenciación-<i>LSB-first</i></b></p> <p>Input: <math>A, e, P</math></p> <p>Output: <math>R = A^e(\text{mod}P)</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>C := A ; R = 1</math></li> <li>2. for <math>i := 0</math> to <math>n - 1</math> do <ol style="list-style-type: none"> <li>2.1 If <math>e_i = 1</math> then <math>C := C \cdot A(\text{mod}P)</math></li> <li>2.2 <math>C := C \cdot C(\text{mod}P)</math></li> </ol> </li> <li>3. return <math>R</math></li> </ol> <p>=====</p>
--

Cuadro 4.11: Algoritmo Exponenciación *LSB-first*

#### 4.2.1. Algoritmo de exponenciación

Sea  $A$  un elemento arbitrario en  $GF(2^m)$  expresado como:

$$A = \sum_{i=0}^{i=m-1} a_i \alpha^i \quad (4.4)$$

y sea  $e$ , ( $1 \leq e \leq 2^m - 1$ ) un entero cuya representación binaria es:

$$e = \sum_{j=0}^{n-1} e_j 2^j = (e_{n-1}, e_{n-2}, \dots, e_1, e_0), e_j = \{0, 1\} \quad (4.5)$$

Entonces la potencia  $R = A^e$  está también en  $GF(2^m)$  y se puede calcular utilizando el método binario [39] en su versión *LSB-first*, como se muestra en el cuadro 4.11.

#### Ejemplo de exponenciación:

Considere el siguiente exponente:  $e = (11111010)_2 = 250$ , en el cuadro 4.12 se muestra cómo opera el algoritmo.



e	Paso 2.a (R)	Paso 2.b (C)
0	1	$(A)^2 = A^2$
1	$1 * A^2 = A^2$	$(A^2)^2 = A^4$
0	$A^2$	$(A^4)^2 = A^8$
1	$A^2 * A^8 = A^{10}$	$(A^8)^2 = A^{16}$
1	$A^{10} * A^{16} = A^{26}$	$(A^{16})^2 = A^{32}$
1	$A^{26} * A^{32} = A^{58}$	$(A^{32})^2 = A^{64}$
1	$A^{58} * A^{64} = A^{122}$	$(A^{64})^2 = A^{128}$
1	$A^{122} * A^{128} = A^{250}$	$(A^{128})^2 = A^{256}$

Cuadro 4.12: Ejemplo de exponenciación

<b>Elevaciones al cuadrado</b>	$n-1$
<b>Multiplicaciones</b>	No. de 1's en $e$
<b>No. total de multiplicaciones:</b>	
<b>Máximo</b>	$2(n-1)$
<b>Mínimo</b>	$n-1$
<b>Promedio</b>	$1.5(n-1)$

Cuadro 4.13: Operaciones básicas del método binario de exponenciación

El método binario requiere de  $n-1$  elevaciones al cuadrado. El número de multiplicaciones es igual al número de bits con valor de "1" en la representación binaria del exponente, excluyendo al bit más significativo, como se muestra en el cuadro 4.13.

### 4.2.2. Arquitectura del Exponenciador

En la figura 4.14, proponemos una arquitectura paralela para la exponenciación en  $GF(2^m)$  basada en el *método binario*. Esta estructura requiere de  $n$  multiplica-

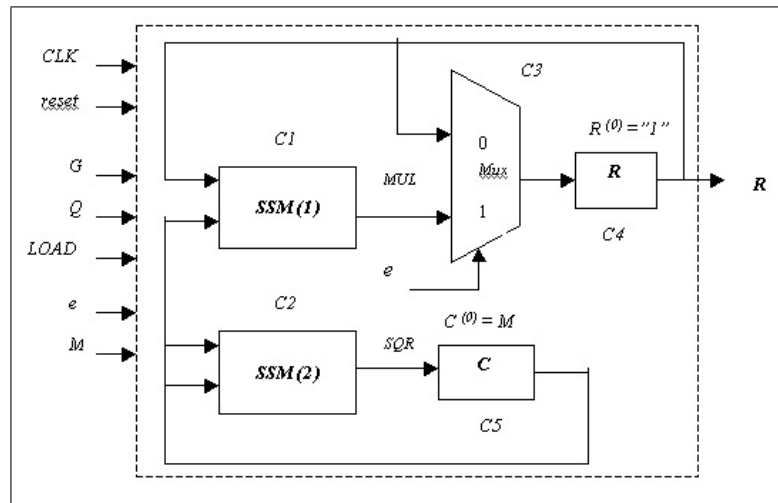


Figura 4.14: Arquitectura del Exponenciador

ciones y  $sn^2$  ciclos de reloj para el cálculo de una exponenciación modular en la que el exponente es un número de  $n$ -bits y  $s$  es el número de ciclos de reloj requeridos para calcular la multiplicación. En nuestro caso, usamos un multiplicador sistólico y serial (*SSM*) [20] cuyo tiempo de operación es de  $3m - 1$  ciclos de reloj.

La arquitectura utiliza dos multiplicadores *SSM* que trabajan en paralelo, dos registros  $R$  y  $C$  de  $m$ -bits y un multiplexor que selecciona la operación correspondiente de *SMM(1)*. Como se puede observar en el algoritmo, los registros  $R$  y  $C$  son cargados inicialmente con "1" y  $A$  respectivamente. Entonces, con cada ciclo de reloj, *SMM(2)* operará para calcular  $C * C$ ; y, dependiendo del valor de  $e_i$ , *SMM(1)* trabajará para realizar el producto de  $R * C$  cada vez que  $e_i = 1$ . El resultado final se obtendrá en el registro  $R$  cuando el bit  $n-1$  de  $e$  sea examinado. En la estructura, las señales son:

- SSM=Multiplicador sistólico y serial
- Mux=Multiplexor
- R,C=Registros con carga paralela

Operadores	Cantidades
Registros	$m^2 + 22m - 20$
Compuertas AND	$4m - 4$
Compuertas XOR	$4m - 4$
Multiplexores 2 a 1	$6m - 5$

Cuadro 4.14: Complejidad de espacio del Exponenciador  $GF(2^m)$ 

Operadores	Número de CLB's
Registros	$0.5m^2 + 11m - 10$
Operaciones Lógicas	$7m - 6.5$

Cuadro 4.15: Requerimientos de espacio en el FPGA XCV300

- G= Modulo
- Q= Señal de control

### 4.2.3. Análisis de complejidad

Como ya se ha considerado, la complejidad de tiempo del exponenciador es  $sn^2$ , donde  $s$  es a su vez la complejidad de tiempo del multiplicador  $SMM$  utilizado en la estructura del exponenciador, que en este caso corresponde a un valor de  $3m - 1$  ciclos de reloj. Con respecto a la complejidad de espacio consideremos lo siguiente:

- Se usan dos multiplicadores  $SMM$  cuya complejidad ya es conocida.
- Se requiere de 1 multiplexor de 2 a 1, y de 2 registros de  $m$  bits cada uno.

En el cuadro 4.14 se resume la complejidad de espacio del exponenciador, y en el cuadro 4.15 se indica el uso de recursos en un FPGA Virtex XCV300.

Este circuito se especifica directamente en VHDL. Su código se puede consultar en el Apéndice A.

-	[36]	[85]	Aquí
Multiplicaciones	$2(m-1)$	$2(m-1)$	$2m$
Multiplicadores	$m-1$	$2(m-1)$	2
Elevadores al cuadrado	$m-1$	—	—
Registros auxiliares	—	—	2
Multiplexores	$m$	—	—
Tiempo de retardo	$m^2 - m/2 + 1$	$2m^2 + 2$	$3m^2 - m$

Cuadro 4.16: Tabla de comparación

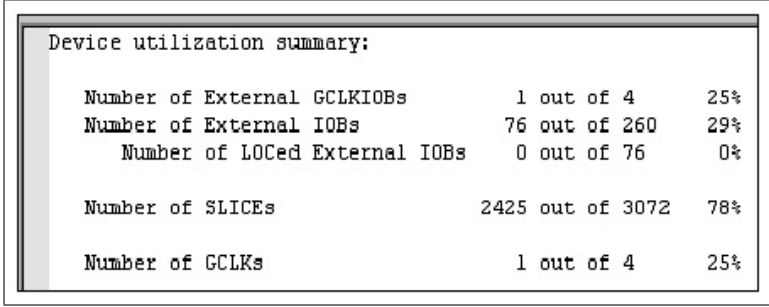
#### 4.2.4. Resultados

Hemos implementado un exponenciador para  $GF(2^{32})$  que ocupa el 75 % de los recursos de un FPGA Virtex XCV300. En el cuadro 4.16 comparamos las características de la arquitectura de nuestro exponenciador con [36] y [85]. Como se puede observar, nuestro exponenciador demanda de mucho menos requerimientos de hardware que las referencias indicadas, si bien necesita de un 30 % más de tiempo para realizar la exponenciación que sus contrapartes.

Considerando que la operación del exponenciador se puede optimizar con el uso de un multiplicador más eficiente, solamente hemos realizado la síntesis del circuito para verificar la complejidad de espacio esperada. Los resultados se presentan en la figura 4.15, que es parte de uno de los archivos de salida del proceso de síntesis (*Place and Route Report*). Una vez que terminemos el diseño de un multiplicador más eficiente, procederemos a la implementación del exponenciador en un FPGA, sin embargo, este trabajo ya no será parte de esta tesis. Hemos utilizado las herramientas del programa ISE 5.1i de Xilinx para la descripción en VHDL y para la síntesis del circuito.

El trabajo desarrollado en esta sección lo hemos presentado en [22] y [23].

### 4.3. IMPLEMENTACIÓN EFICIENTE DE UN MULTIPLICADOR SERIAL/PARALELO109



Device utilization summary:		
Number of External GCLKIOBs	1 out of 4	25%
Number of External IOBs	76 out of 260	29%
Number of LOCed External IOBs	0 out of 76	0%
Number of SLICES	2425 out of 3072	78%
Number of GCLKs	1 out of 4	25%

Figura 4.15: Consumo de recursos del exponenciador de 32 bits

## 4.3. Implementación eficiente de un Multiplicador Serial/Paralelo

Presentamos en este apartado la implementación de un multiplicador para  $GF(2^m)$  que aprovecha eficientemente los recursos de un FPGA Virtex de la serie XCV300. Es un multiplicador Serial/Paralelo del tipo *LSE-first*, que opera en los campos  $GF(2^{193})$  y  $GF(2^{239})$ , y hace el cálculo de una multiplicación en  $m$ -ciclos de reloj.

### 4.3.1. Algoritmo de multiplicación

Hemos utilizado el algoritmo de multiplicación *LSE-first* que se ha estudiado previamente en el apartado 2.2.3, y que se muestra enseguida.

Sean  $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$  y  $B(\alpha) = \sum_{i=0}^{m-1} b_i \alpha^i$  dos elementos del campo  $GF(2^m)$ , y sea  $C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i$  su producto. La multiplicación puede ser calculada de la manera siguiente:

$$\begin{aligned} C(x) &= A(x)B(x) \bmod P(x) \\ &= A(x) \left( \sum_{i=0}^{m-1} b_i x^i \right) \bmod P(x) \end{aligned}$$

<p><b>Algoritmo <i>LSE-first</i></b></p> <p>Input: <math>A(\alpha), B(\alpha) \in GF(2^m)</math></p> <p>Output: <math>C(\alpha) = A(\alpha) \cdot B(\alpha) \in GF(2^m)</math></p> <p>=====</p> <ol style="list-style-type: none"> <li>1. <math>C \leftarrow 0</math></li> <li>2. For <math>i = 0</math> to <math>m - 1</math> <ol style="list-style-type: none"> <li>2.1 <math>C \leftarrow b_i A + C</math></li> <li>2.2 <math>A \leftarrow Ax^i \bmod P(x)</math></li> </ol> </li> <li>3. return <math>(C(\alpha))</math></li> </ol> <p>=====</p>
--

Cuadro 4.17: Multiplicación *LSE-first* de dos elementos en  $GF(2^m)$ 

$$\begin{aligned}
&= \sum_{i=0}^{m-1} b_i (x^i A(x) \bmod P(x)) \\
&= b_0 A + b_1 (xA(x) \bmod P(x)) + b_2 (x^2 A(x) \bmod P(x)) + \dots + \\
&\quad b_{m-1} (x^{m-1} A(x) \bmod P(x)) \tag{4.6}
\end{aligned}$$

En el cuadro 4.17 se muestra el algoritmo para este proceso de multiplicación. Se podrá observar que se requieren  $m$  ciclos de reloj para obtener el cálculo de una multiplicación.

### 4.3.2. Arquitectura del Multiplicador

El multiplicador está integrado por dos estructuras básicas, un LFSR (*Linear Feedback Shift Register*) de  $m$ -bits, cuya operación particular se puede estudiar en [7] y [78], y un arreglo de  $m$ -celdas como se muestra en la figura 4.16. El LFSR se usa para calcular la operación  $\alpha A(\alpha)$  como se ha mencionado en la sección 2.2.3, y como se muestra en la figura 4.17. Las señales  $p_i$  representan los coeficientes del polinomio irreducible. Se podrá observar que cuando  $p_i = 1$ , la rama

### 4.3. IMPLEMENTACIÓN EFICIENTE DE UN MULTIPLICADOR SERIAL/PARALELO111

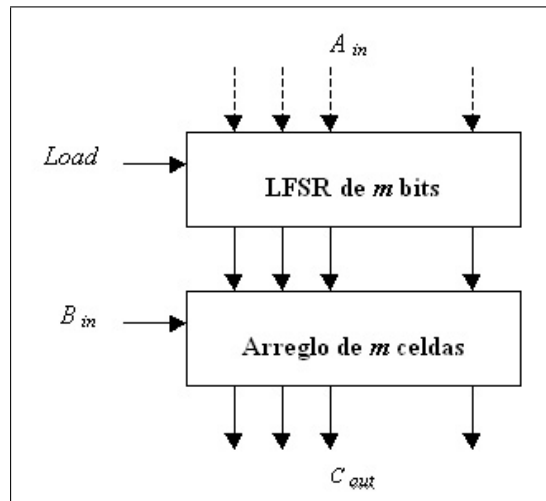
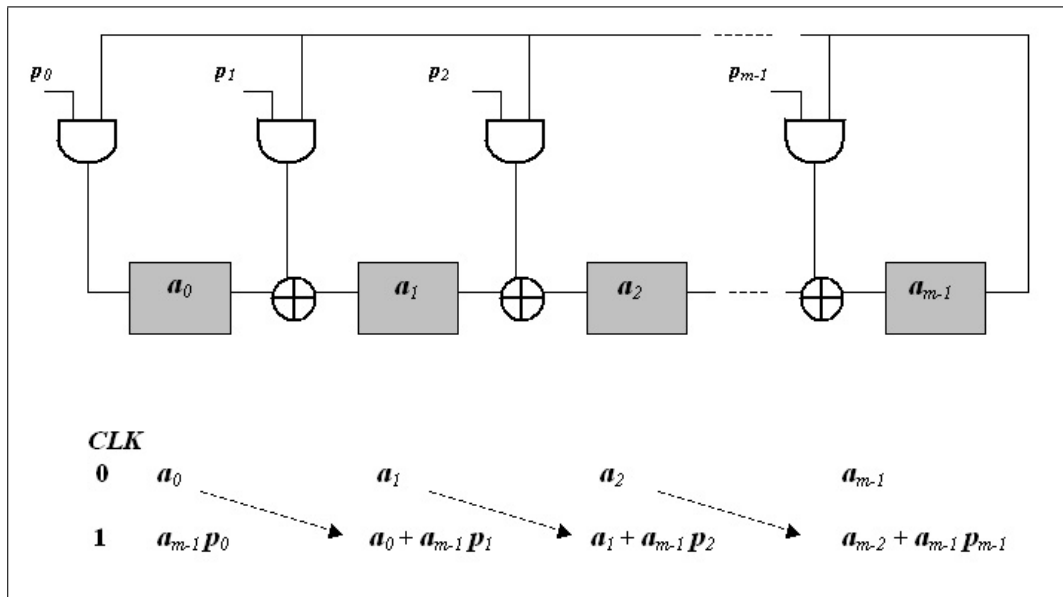


Figura 4.16: Estructuras básicas del Multiplicador

correspondiente del circuito será un “corto circuito”, y cuando  $p_i = 0$ , será un “circuito abierto”. Es decir que las líneas de retroalimentación están definidas por el polinomio irreducible, que en este caso es  $P(x) = x^{239} + x^{36} + 1$ . Inicialmente la representación  $(a_0, a_1, a_2, \dots, a_{m-1})$  de  $A$  es cargada dentro del registro. Ya que un registro requiere de una *señal de reloj* (la cual es un tren de pulsos periódico) para *pasar* datos de su entrada a su salida, se aplica tal señal de reloj al registro, y después de  $m$ -pulsos o  $m$ -ciclos de reloj, el nuevo contenido dentro del registro forma la representación vectorial de  $D(\alpha) = \alpha A(\alpha)$ , como se aprecia en la figura.

La construcción interna de cada una de las 239 celda del arreglo inferior se observa en la figura 4.18, y corresponde al cálculo de  $c_i$ . La arquitectura completa del multiplicador  $GF(2^{239})$  se muestra en la figura 4.19. Inicialmente, el registro  $C$  es puesto a cero y se carga el LFSR con el valor de  $A$ . Después, al aplicarse a los registros la señal de reloj, se obtiene  $A\alpha$ . Los coeficientes  $b_0, b_1, \dots, b_{m-1}$  son introducidos serialmente al multiplicador y los valores  $b_i A\alpha^i$ , ( $i = 0, 1, \dots, m - 1$ ) son entonces calculados, los cuales son acumulados en el registro  $C$  para formar,

Figura 4.17: Multiplicación  $\alpha A(\alpha)$  en  $GF(2^m)$ 

en  $m$ -pulsos de reloj, los bits del producto  $c_0, c_1, \dots, c_{m-1}$ .

### 4.3.3. Análisis de complejidad

Como ya se ha considerado previamente, la complejidad de tiempo de nuestro multiplicador es de  $O(m)$  ciclos de reloj. En cuanto a su complejidad de espacio, hemos realizado un precálculo de los requerimientos de hardware para la implementación del circuito, y se ha basado en las siguientes observaciones.

- Cada CLB de un FPGA Virtex XCV300 se compone de 2 *slices*. Cada *slice* cuenta con 2 generadores de función de 4-entradas y 2 registros.
- La estructura del LFSR ocupa  $m$ -registros y 1- compuerta XOR si el polinomio irreducible es un trinomio, o 3-compuertas XOR si es un pentanomio.
- Cada celda del arreglo requiere de 1-registro, 1-compuerta XOR y 1-compuerta



4.3. IMPLEMENTACIÓN EFICIENTE DE UN MULTIPLICADOR SERIAL/PARALELO113

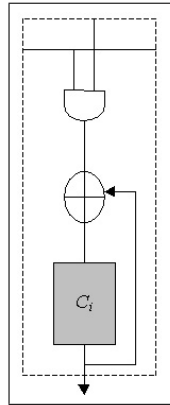


Figura 4.18: Celda del arreglo

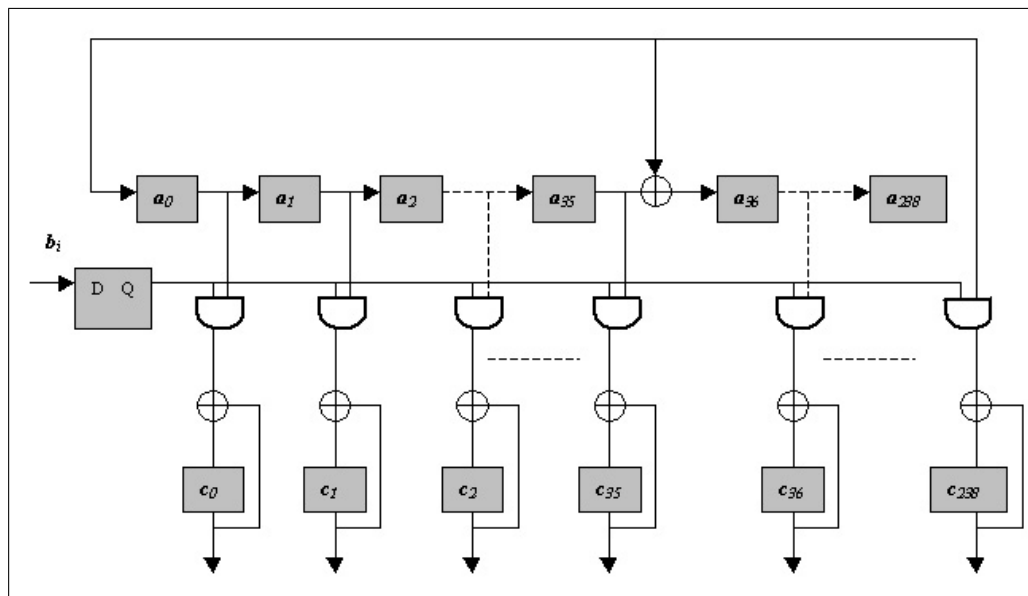


Figura 4.19: Multiplicador *LSE-first*  $GF(2^{239})$

-	Registros	Compuertas	<i>slices</i>
LFSR	$m$	1 o 3	$m/2$
Arreglo	$m$	$2m$	$m$
Total	-	-	$1.5m$

Cuadro 4.18: Complejidad de espacio del Multiplicador

AND.

- Hacemos una estimación considerando que cada registro y cada celda ocuparán medio-*slice* y un *slice* respectivamente.

En el cuadro 4.18 se resume la medida de complejidad del multiplicador.

#### 4.3.4. Resultados

Se ha hecho una descripción parametrizable del multiplicador con VHDL, cuyo código se presenta en el apéndice A.

#### Síntesis del circuito

Para la prueba de síntesis hemos utilizado la herramienta computacional *Synopsys*, del paquete ISE4.1i de Xilinx.

Estas pruebas de síntesis se hicieron usando los campos  $GF(2^{193})$  y  $GF(2^{239})$ , que son campos recomendados en los estándares del NIST [60]. Los polinomios irreducibles utilizados en cada caso, también recomendados por el NIST han sido:

$$P(x) = x^{193} + x^{15} + 1 \text{ y } P(x) = x^{239} + x^{36} + 1 \text{ respectivamente.}$$

En el cuadro 4.19 enumeramos los resultados del proceso de síntesis, y añadimos un tercer valor con un campo  $GF(2^{163})$ . Como referencia, en el cuadro 4.20 se muestran las características del FPGA XCV300 que se ha utilizado en este trabajo.

### 4.3. IMPLEMENTACIÓN EFICIENTE DE UN MULTIPLICADOR SERIAL/PARALELO115

Campo	Complejidad estimada	Complejidad medida	% de uso
$GF(2^{163})$	245 <i>slices</i>	246 <i>slices</i>	8 %
$GF(2^{193})$	290 <i>slices</i>	291 <i>slices</i>	9 %
$GF(2^{239})$	359 <i>slices</i>	347 <i>slices</i>	11 %

Cuadro 4.19: Resultados de la síntesis

Característica	Cantidad
No. máximo de <i>slices</i>	3072
No. máximo de registros	6144
No. máximo de I/O	260
Frecuencia máxima	200 Mhz

Cuadro 4.20: Características del FPGA XCV300

#### Comparación del Multiplicador

En el cuadro 4.21 comparamos el tiempo de operación de nuestro multiplicador con algunas implementaciones recientes desarrolladas en software. En este cuadro  $m$  es el orden del campo. En el cuadro 4.22 comparamos los resultados de nuestro multiplicador con implementaciones en hardware reportadas recientemente. Introducimos la medida de *Frecuencia/área* para tener una mejor referencia en

-	SW/HW	$m$	Tiempo	Plataforma
López <i>et al</i> [49]	SW	162	10.5 $\mu S$	Ultra-Sparc (300 Mhz)
Savas <i>et al</i> [80]	SW	160	18.3 $\mu S$	Micro-ARM (80 Mhz)
Rodríguez [75]	SW	163	5.4 $\mu S$	Pentium II (450 Mhz)
Aquí	HW	239	3.1 $\mu S$	Virtex-300 (75 Mhz)

Cuadro 4.21: Comparación del Multiplicador con implementaciones en SW

-	Area ( <i>slices</i> )	Frec. (Mhz)	Frec/área	Tiempo ( $\mu$ S)
[42]- $GF(2^{210})$	$\tilde{4}20$	17.1	0.04	12.3
[14]- $GF(2^{240})$	1211	60	0.05	4
Aquí- $GF(2^{239})$	359	75	0.2	3.1

Cuadro 4.22: Comparación del Multiplicador con implementaciones en HW

la comparación. Se puede observar una mejor medida en el comportamiento de nuestro multiplicador y un mejor tiempo para el cálculo de la multiplicación comparado con los otros dos.

*Nota:* En la referencia [42] la medida de complejidad de área la proporcionan en número de compuertas. Hemos hecho una estimación en función de los *slices* que ocuparía.

El trabajo desarrollado en esta sección lo hemos presentado en [24].

# Capítulo 5

## Conclusiones

### 5.1. Sumario y contribuciones

El objetivo principal de este trabajo ha sido el estudio e implementación en hardware reconfigurable, de los principales operadores aritméticos sobre campos finitos  $GF(2^m)$ , tales como la multiplicación, la división y la exponenciación. Las arquitecturas que hemos construido tienen su aplicación en áreas como los códigos de corrección de errores del tipo Reed-Solomon y en criptografía de curvas elípticas.

Hemos calificado la eficiencia de nuestras implementaciones en función de sus complejidades de tiempo y de espacio. Esta última, en contraste con la medida tradicional de número de compuertas, la hemos medido por el número de CLB's o *slices* del FPGA que se ha utilizado como dispositivo físico para la implementación.

Hemos realizado una revisión de los multiplicadores sobre  $GF(2^m)$  más representativos que encontramos en el estado del arte, considerando el tipo de arquitectura, que puede ser paralela, serial o por dígitos, y la base de representación de los elementos del campo finito, que puede ser polinomial, normal o dual.

El capítulo 4 está basado en tres artículos [20],[22] y [24], que hemos presentado en diversos congresos internacionales del área.

- En el primero de ellos hemos descrito la implementación en un FPGA de una arquitectura sistólica y serial para el cálculo de la división/multiplicación sobre  $GF(2^m)$ , como lo proponen Hasan y Bhargava en [29]. La novedad de esta arquitectura se basa en el procedimiento usado para el cálculo de la división, el cual se basa en el método de Gauss-Jordan para la solución de un sistema de ecuaciones lineales. El circuito realiza una división en  $5m - 1$  ciclos de reloj y una multiplicación en  $3m - 1$  ciclos, y tiene una complejidad de espacio de  $O(m^2)$ . Además la arquitectura es de tipo sistólico, y esto nos ha permitido realizar una descripción VHDL *parametrizable* del divisor, proporcionando al circuito características de escalabilidad y facilitando el proceso de síntesis y de prueba.

Por otro lado, con nuestros resultados hemos mostrado que el orden mayor del campo finito que puede ser implementado en un FPGA XCS4010 es 12, es decir  $GF(2^{12})$ , lo que se ha debido al crecimiento de orden cuadrático de la complejidad de espacio. Así pues, este divisor tiene una aplicación directa en decodificadores Reed-Solomon que operan regularmente con campos de orden 8.

- En el segundo artículo hemos introducido una nueva arquitectura para calcular la exponenciación en  $GF(2^m)$ , basada en el método binario en su versión *LSB-first*. Hemos propuesto una arquitectura que sólo requiere de dos multiplicadores operando en paralelo, y que comparada con otras propuestas encontradas en el estado del arte, introduce una mejora en su medida de complejidad de espacio.
- En el tercer artículo hemos presentado los resultados de la implementación en un FPGA Virtex-XCV300 de un multiplicador serial/paralelo en su versión *LSE-first* que opera en los campos  $GF(2^{193})$  y  $GF(2^{239})$ , que son campos recomendados por el NIST (*National Institute of Standards and Technology*) de

los E.U.A. para su aplicación en criptosistemas con Curvas Elípticas. Hemos realizado un análisis previo para determinar la *complejidad de espacio* del circuito, y lo hemos comprobado al realizar la implementación física del circuito en el FPGA por medio de los reportes del sintetizador. El multiplicador usa el 11 % de los recursos de área del FPGA. Además realiza la multiplicación en  $m$ -ciclos de reloj y, operando a 75 Mhz, calcula una multiplicación en 3.1 microsegundos. Lo hemos comparado con otros multiplicadores reportados en la literatura reciente, y presenta un mejor desempeño tanto en tiempo como en espacio que sus contrapartes.

## 5.2. Resultados y criterios de validación

Para la validación de los resultados de nuestros diseños hemos utilizado las herramientas computacionales que provee el fabricante XILINX. Nos referimos al programa ISE4.1i que integra al manejador de proyectos, al sintetizador *Synopsys* y el simulador *ModelSim*. Con estas herramientas se ha hecho la descripción con VHDL de todos los circuitos, usando además el editor esquemático como soporte. Posteriormente se han realizado la simulación y la síntesis de los diseños, que nos han permitido obtener los resultados que se han presentado en el capítulo anterior.

## 5.3. Perspectivas de desarrollo ulterior

Usando como base el multiplicador serial que hemos construido en la apartado 4.3, se podrá implementar un exponenciador eficiente para  $GF(2^m)$ , como lo hemos propuesto en la sección 4.3. Además, también podrá utilizarse para la implementación de un circuito inversor, de un elevador al cuadrado o de un multiplicador por dígitos.

En general, los circuitos que hemos realizado, se podrán usar como operadores básicos para su integración en un procesador dedicado para operar con curvas elípticas y ser aplicado en criptografía. Sin duda, cada uno de estos circuitos puede ser objeto de un proceso de optimización aprovechando las herramientas que para este fin ofrecen los fabricantes de FPGA's. Por otro lado, ya que ésta es un área en continuo desarrollo, habrá que estar atentos a la propuesta de nuevos algoritmos que nos permitan mejorar las medidas de complejidad y tiempo de las arquitecturas que se han desarrollado hasta este momento, particularmente con el uso de polinomios irreducibles especiales tales como los *trinomios*, *pentanomios*, *AOP(All One Polynomial)* y *ESP(Equally Space Polynomial)*.

No dudamos que en los siguientes años, el avance tecnológico en el desarrollo de circuitos integrados, especialmente los programables, permitirá un importante impulso al estudio de la *Lógica Reconfigurable* y su aplicación en el área de criptografía, códigos de corrección de errores y el procesamiento digital de señales.

Por último, hemos de mencionar que el estudio de ésta área de trabajo nos permitirá fortalecer una línea de investigación poco tratada en nuestro país, específicamente en la implementación en circuitos programables FPGA's de estructuras aritméticas sobre campos finitos en sus diferentes áreas de aplicación.



# Apéndice A

## Descripción en VHDL de los diseños realizados

El contenido de este apéndice se encuentra en un disco compacto que se anexa con este documento de tesis.

En el disco mencionado se puede consultar:

- El código VHDL de los circuitos que se han implementado.
- Los artículos que hemos escrito y presentado en diversos foros y que son resultado de este trabajo de investigación.
- Incluimos también diversas presentaciones de las participaciones que hemos tenido en los congresos mencionados.
- Además, proporcionamos una serie de enlaces en la red de Internet donde se podrán consultar temas relacionados con el trabajo de investigación que hemos desarrollado.



# Bibliografía

- [1] K. Araki, I. Fujita, and M. Moriuse. Fast inverters over finite field based on Euclid's algorithm. *Trans. IEICE*, E-72, no.11: 1230-1234, 1989.
- [2] C.G. Ahlquist, B. Nelson and M. Rice. Optimal Finite Field Multipliers for FPGA's. *Irvine, P. Lysaght and R. Hartenstein, editors, 9th International Workshop on Field Programmable Logic and Applications (FPL99), Springer-Verlag: 51-61,1999.*
- [3] K. Araki, I. Fujita and M. Morisue. Fast inverter over finite field based on Euclid's algorithm. *Trans. IEICE*, vol. E-72, pp. 1230-1234, Nov. 1989.
- [4] H. Brunner, A. Curiger and M. Hofstetter. On Computing Multiplicative Inverses in  $GF(2^m)$ . *IEEE Trans. on Comp.*, 42, no.3:1010-1015,1993.
- [5] E.R. Berlekamp. Bit-serial Reed-Solomon encoders. *IEEE Trans. on Inform.*, 28(6):869-874, 1982.
- [6] G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar and T. Wollinger. Efficient  $GF(2^m)$  Arithmetic Architectures for Cryptography Applications. <http://citeseer.nj.nec.com/579064.html>, 2003.
- [7] R.E. Blahut. *Theory and Practice of Error Control Codes*. Addison Wesley, 1984.
- [8] R.P. Brent and H.T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Trans. on Comp.*, C-33, no.8:731-736, 1984.

- [9] T. Blum and C. Paar. Montgomery Modular Exponentiation on Reconfigurable Hardware. *IEEE Symp. on Computer Arithmetic (ARITH-14)*, Adelaide, Australia, 1999.
- [10] P.K. Chan and S. Mourad. *Digital Design Using Field Programmable Gate Arrays*, Prentice-Hall, 1994.
- [11] A. DeHon and J. Wawrzynek. *Reconfigurable Computing: What, Why and Implications for Design Automation*. Berkeley Reconfigurable Architectures, Software and Systems. Computer Science Division. 1998.
- [12] W. Drescher and G. Fettweis. VLSI Architectures for Multiplication in  $GF(2^m)$  for Applications Tailored Digital Signal Processors. Dep. of Elect. Eng. Dresden University of Technology, Rep. D-01062.
- [13] A. Daneshbeh and M. Hasan. A Class of Scalable Unidirectional Bit Serial Systolic Architectures for Multiplicative Inversion and Division over  $GF(2^m)$ . <http://www.cacr.math.uwaterloo.ca>. Dec. 2002. Technical report CORR 2002-35.
- [14] A. Daly and W. Marnane. Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. *FPGA '02, Monterey, Ca. USA*, 2002.
- [15] M. Elia and M. Leone. On The Inherent Space Complexity of Fast Parallel Multipliers for  $GF(2^m)$ . *IEEE Trans. on Comp.*, 51:346-351, 2002.
- [16] T. ElGamal. A public-key cryptosystem and signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31:469-472, 1985.
- [17] S.T.J. Fenn, M. Benaissa and D. Taylor.  $GF(2^m)$  Multiplication and Division over the Dual Basis. *IEEE Trans. on Comp.*, 45, no.3:319-327, 1996.
- [18] G-L. Feng. A VLSI Architecture for Fast Inversion in  $GF(2^m)$ . *IEEE Trans. on Comp.*, 38, no.10:1383-1386, 1989.

- [19] W. Geiselman and D. Gollman. Self-dual bases in  $F_{q^n}$ . *designs, Codes and Criptography*, 3:333-345, 1993.
- [20] M.A. García Martínez and G. Morales-Luna. VHDL Specification of a FPGA to Divide and Multiply in  $GF(2^m)$ . In Mullen G.L., Stichtenoth H., Tapia Recillas H. (eds). *Finite Fields with Applications to Coding Theory, Cryptography and Related Areas*, Springer-Verlag, 2002.
- [21] M.A. García-Martínez, G. Morales-Luna. Especificación en VHDL de un Multiplicador sistólico y serial para  $GF(2^m)$ . *8o. Workshop Iberchip*. Guadalajara Jal. Mex. Abr. 2002.
- [22] M.A. García-Martínez, G. Morales-Luna and F. Rodríguez-Henríquez. Hardware Implementation of the Binary Method for Exponentiation over  $GF(2^m)$ . *4o. Encuentro Internacional de Ciencias de la Computación*. Tlaxcala, Mex. Sep. 2003.
- [23] M.A. García-Martínez, G. Morales-Luna and F. Rodríguez-Henríquez. Descripción en VHDL de un exponenciador para campos finitos  $GF(2^m)$ . *X Workshop Iberchip*. La Habana, Cuba. Mar. 2003.
- [24] M.A. García-Martínez, G. Morales-Luna and F. Rodríguez-Henríquez. Implementación en FPGA de un Multiplicador Eficiente para Campos Finitos  $GF(2^m)$ . *X Workshop Iberchip*. Cartagena de Indias, Col. Mar. 2004.
- [25] D.M. Gordon. A Survey of Fast Exponentiation Methods. *Journal of Algorithms*, 27:129-146, 1998.
- [26] J. Guajardo and C. Paar. Fast inversion in composite Galois Fields  $GF((2^m)^n)$ . *WPI: Worcester Polytechnic Institute*, 1998.
- [27] J. Guajardo and C. Paar. Itoh-Tsujii Inversion Algorithm. <http://www.win.tue.nl/henkvt/ItohTsujiiEnciclopediaOfInfoSec-Submitted.pdf>, 2001.

- [28] J.H. Guo and C.L. Wang. Hardware-efficient systolic architecture for inversion and division in  $GF(2^m)$ . *IEE Proc. Comput. Digit. Tech.*, 145: 272-278, 1998.
- [29] M.A. Hasan and V.K. Bhargava. Division and bit-serial multiplication over  $GF(2^m)$ . *IEE Proceedings-E*, Vol. 139, No. 3, pp.230-235, May 1992.
- [30] M.A. Hasan and V.K. Bhargava. Bit-serial systolic divider and multiplier for finite fields  $GF(2^m)$ . *IEEE Trans. on Comp.*, 41(8):972-980, 1992.
- [31] M.A. Hasan, M. Wang and V.K. Bhargava. Modular Construction of Low Complexity Parallel Multipliers for a Class of Finite Fields  $GF(2^m)$ . *IEEE Trans. on Comp.*, 41(8):962-971, 1992.
- [32] R. Hartley and P. Corbett. Digit-Serial Processing Techniques. *IEEE Trans. on Circuits and Systems*. 37:707-719, 1990.
- [33] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in  $GF(2^m)$  Using Normal Basis. *Information and Computation J.*, 78:171-177, 1988.
- [34] T. Itoh and S. Tsujii. Structure of parallel multipliers for a class of fields  $GF(2^m)$ . *Inform. Com.*, vol. 83. pp. 21-40, 1989.
- [35] Y. Jeong and W. Burleson. Choosing VLSI Algorithms for Finite Field Arithmetic. *IEEE Symposium on Circuits and Systems, ISCAS'92*, 1992.
- [36] S.K. Jain, L. Song and K.K. Parhi. Efficient Semisystolic Architectures for Finite-Field Arithmetic. *IEEE Trans. on VLSI Systems*, 6, no.1: 101-113, 1998.
- [37] A.A. Karatsuba and Y.P. Ofman. Multiplication of Multiplace Numbers with Automata. *Dokl.Akad. nauk SSSR*, 1962, vol. 145, no. 2, pp. 293-294.

- [38] C.H. Kim, S.Kwon, J.J. Kim and C.P. Hong. A New Arithmetic Unit in  $GF(2^m)$  for Reconfigurable Hardware Implementation. *P.Y.K. Cheung et al. (Eds.): FPL 2003*, LNCS 2778, pp. 670-680, 2003.
- [39] D.E. Knuth. *The art of computer programming, vol. II: Seminumerical algorithms*. Addison-Wesley, 2a. Ed., 1982.
- [40] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203-209, 1987.
- [41] C.K. Koc. Speeding up the computations on an elliptic curve using addition-subtraction chains. <http://citeseer.nj.nec.com/Koc95analysis.html>, 1995.
- [42] P. Kitsos, G. Theorodidis and O. Koufopavlou. An efficient reconfigurable multiplier for Galois Fields  $GF(2^m)$ . *Microelectronics Journal*, 34: 975-980, 2003.
- [43] H.T. Kung. Why systolic architectures?. *IEEE Trans. on Comp.*, pp. 37-46, 1982.
- [44] S.Y. Kung. On supercomputing with systolic/wavefront array processors. *Proc. IEEE*, vol. 72, pp. 867-884, July 1984.
- [45] S. Lin and D.J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, Englewood Cliffs, N.J., 1983.
- [46] M. Linares. *Diseño de celdas de procesamiento sistólico matricial para sistemas de alta velocidad*. Tesis doctoral. Cinvestav, México, 1996.
- [47] C.H. Lim and P.J. Lee. More flexible exponentiation with precomputation. *Advances in Cryptology. Proceedings of Crypto'94*, 839:95-107, 1994.
- [48] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge, U.K., 1994.

- [49] J. López y R. Dahab. Fast Multiplication on Elliptic Curves over  $GF(2^m)$  without Precomputation. *Cryptography Hardware and Embedded Systems, CHES 1999*, C.K. Koc and C. Paar Eds. Second International Workshop, Worcester, MA, USA, pp. 316-327, Springer-Verlag, LNCS 1999.
- [50] B.A. Laws and C.K. Rushforth. A celular-array multiplier for finite fields. *IEEE Trans. on Comp.*, pp. 1573-1578, 1971.
- [51] E.D. Mastrovito. *VLSI design for multiplication over finite fields  $GF(2^m)$* . In *Lecture Notes in Computer Science 357*, pp. 297-309. Springer-Verlag, Berlín, March 1989.
- [52] E.D. Mastrovito. *VLSI Architectures for Computation in Galos Fields*. PhD thesis, Linkoping University, Dept. Electr. Eng., Linkoping, Sweden, 1991.
- [53] R. Murgai, R.K. Brayton and A. Sangiovanni-Vincentelli. *Logic Synthesis for Field-Programmable Gate Arrays*. Kluwer Academic Publishers. 1995.
- [54] V. Miller. Uses of elliptic curves in cryptography. In *H.C. Williams, editor. Advances in Cryptology-Crypto'85, Proceedings, LNCS, Springer-Verlag*, 218:417-426, 1985.
- [55] J.L Masey and J.K. Omura. Apparatus for finite field computation. *U.S. Patent Application*, pages 21-40, 1984.
- [56] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-substraction chains. *Inform. Theory Appl.*, 24:531-543, 1990.
- [57] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519-521, 1985.
- [58] A.J. Menezes, P. Van Oorshot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL., 1997.



- [59] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone and R.M. Wilson. Optimal normal bases in  $GF(p^n)$ . *Discrete Applied Mathematics, North Holland*, 22:149-161, 1988/89.
- [60] National Institute of Standards and Technology. *Recommended elliptic curves for federal government use*. 1999
- [61] S. Oh, C-H Kim, J. Lim and D-H Cheon. Efficient Normal Basis Multipliers in Composite Fields. *IEEE Trans. on Comp.*,49:1133-1138, 2000.
- [62] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD Thesis, VDI-verlag, Dusseldorf, 1994.
- [63] C. Paar. Some Remarks on Efficient Inversion in Finite Fields. <http://citeseer.nj.nec.com/paar95some.html>, 1995.
- [64] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Trans. on Computers*, 45(7):856-861, July 1996.
- [65] C. Paar. Reconfigurable Hardware in Modern Cryptography. <http://www.ece.wpi.edu/Research/crypt>. 2000.
- [66] K.K. Parhi. A Systematic Approach for Design of Digit-Serial Processing Architectures. *IEEE Trans. on Circuits and Systems*, 38:358-375, 1991.
- [67] B.J. Phillips and N. Burgess. Implementing 1024-bit RSA Exponentiation on a 32-bit Processor Core. *IEEE Inf. Conf. on Application-Specific Systems, Architectures and Processors (ASAP'00)*, pp. 127-137, 2000.
- [68] N. Petkov. *Systolic parallel processing*. North-Holland, 1993.
- [69] C. Paar and N. Lange. A Comparative VLSI Synthesis of Finite Field Multipliers. *Proceedings of the 3rd. International Symposium on Communication Theory and Applications*, 1995.

- [70] C. Paar and M. Rosner. Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware. *IEEE Symposium on Field- Programmable Custom Computing Machines, FCCM'97*, 1997.
- [71] C. Paar and P. Soria-Rodríguez. Fast Arithmetic Architectures for Public-Key Algorithms Over Galois Field  $GF((2^n)^m)$ . *Advances in Cryptography-EUROCRYPT'97, Springer-Verlag, LNCS*, 1233:363-378, 1997.
- [72] W.W. Peterson and E.J. Weldon. *Error-Correcting Codes*. MIT Press, Cambridge, Massachusetts, 1972.
- [73] M. Ram Murty: Artin's conjecture for primitive root, *The Mathematical Intelligencer*, Vol.4, Nr. 10, pp. 59-67, 1988.
- [74] A. Reyhani-Masoleh and M.A. Hasan. A New Construction of Massey-Omura Parallel Multiplier Over  $GF(2^m)$ . *IEEE Trans. on Comp.*, 51, no.5: 511-520, 2002.
- [75] F. Rodríguez-Henríquez. *New Algorithms and Architectures for Arithmetic in  $GF(2^m)$  Suitable for Elliptic Curve Cryptography*. PhD Thesis, Oregon State University, 2000.
- [76] R.L. Rivest, A. Shamir and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21, no.2: 120-126, 1978.
- [77] B. Sunar and C.K. Koc. An Efficient Optimal Normal Basis Type II Multiplier. *IEEE Trans. on Comp.*, 50:83-87, 2001.
- [78] L. Song and K.K. Parhi. Efficient Finite Field Serial/Parallel Multiplication. *Proc. of International Conf. on Application Specific Systems, Architectures and Processors*. pp. 72-82, Chicago, Aug. 1996.
- [79] L.Song and K.K. Parhi. Low energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing*, 19(2):149-166, 1998.

- [80] E. Savas, A.F. Tenca and C.K. Koc. A Scalable and Unified Multiplier Architecture for Finite Fields  $GF(p)$  and  $GF(2^m)$ . *C.K. Koc and C. Paar (Eds.): CHES 2000, LNCS, Springer-Verlag*, pp. 277-292, 2000.
- [81] P.A. Scott, S.E. Tavares and L.E. Peppard. A Fast VLSI Multiplier for  $GF(2^m)$ . *IEEE Journal on Selected Areas in Communications*, SAC-4(1): 62-65, 1986.
- [82] S.M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers. USA. 1994.
- [83] N. Takagi, J. Yoshiki and K. Takagi. A Fast Algorithm for Multiplicative Inversion in  $GF(2^m)$  Using Normal Basis. *IEEE Trans. on Comp.* 50, no.5:394-398, 2000.
- [84] I. Vinogradov, *Fundamentos de la Teoría de los Números*, Ed. MIR, Moscú, 1977.
- [85] C.L. Wang. Bit-Level Systolic Array for Fast Exponentiation in  $GF(2^m)$ . *IEEE Trans. on Comp.*, 43(7): 838-841, 1994.
- [86] C.L. Wang and J.L. Ling. Systolic Array Implementation of Multipliers for Finite Fields  $GF(2^m)$ . *IEEE Trans. on Circuits and Systems*, 38: 796-800, 1991.
- [87] C.L. Wang and J.L. Ling. A systolic architecture for computing inverses and divisions in  $GF(2^m)$ . *IEEE Trans. on Comp.*, C-42:1141-1146, 1993.
- [88] C.C. Wang, T.K. Truong, H.M. Shao, L.J. Deutsch, J.K. Omura and L.S. Reed. VLSI architectures for computing multiplications and inverses in  $GF(2^m)$ . *IEEE Trans. on Comp.*, C-34:709-717, 1985.
- [89] H. Wu. On complexity on squaring using polynomial basis in  $GF(2^m)$ . <http://citeseer.nj.net.com/301008.html>. 2000.

- [90] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective*. Addison-Wesley Publishing Company, second edition, 1992
- [91] C.L. Wang and J.H. Guo. New Systolic Arrays for  $C + AB^2$ , Inversion, and Division in  $GF(2^m)$ . *IEEE Trans. on Comp.* vol. 49, no. 10, pp.1120-1125, Oct.2000.
- [92] C.C. Wang, T.K. Truong, H.M. Shao, L.J. Deutsch, J.K. Omura and I.S. Reed. VLSI architectures for computing multiplications and inverses in  $GF(2^m)$ . *IEEE Transactions on Computers*, C-34:709-717, August 1985.
- [93] Z. Yan and D. Sarwate. Systolic Architectures for Finite Field Inversion and Division. *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'02*, V:789-792, June 2002.