Felipe Sánchez Corrionero
fcorrionero@gmail.com

Director: José Luis Imaña Pascual
jluimana@dacya.ucm.es

IMPLEMENTACIONES HARDWARE DE CIRCUITOS ARITMÉTICOS SOBRE CUERPOS FINITOS
/
HARDWARE IMPLEMENTATIONS OF ARITHMETIC CIRCUITS OVER FINITE FIELD

Key words: finite fields, Galois field, mastrovito, digit-serial, arithmetic, fpga-based, polynomial basis, normal basis.
Palabras clave: cuerpos finitos, cuerpos de Galois, mastrovito, dígito serie, aritmética, fpga, base polinómica, base normal.

ÍNDICE

1. Introducción 3
2. Conceptos previos 7
    2.1 Cuerpos finitos...7
    2.2 Dispositivos FPGAs..10
3. Diseño e implementación de operaciones aritméticas sobre GF(2^m) 12
    3.1 Multiplicación ....
    12
    3.2 Cuadrado..23
    3.3 Inversión...24
4. Resultados Experimentales 26
    4.1 Complejidades espaciales de los multiplicadores para GF(2^8) 27
    4.2 Complejidades temporales de los multiplicadores para GF(2^8) 30
    4.3 Complejidades espaciales y temporales de los multiplicadores para GF(2^m).........33
    4.4 Complejidades espaciales y temporales para la operación cuadrado.....................35
    4.5 Complejidades espaciales y temporales para la inversión.....................................36
5. Conclusiones y trabajo futuro..37
6. Bibliografía...39

Resumen: La aritmética sobre cuerpos finitos ha recivido mucho interés debido a su importancia en criptografía, control de errores de codificación y procesado de señales digitales. Una gran parte del tiempo de las rutinas criptográficas se dedica al cálculo de operaciones aritméticas sobre cuerpos finitos. Los sistemas que usan esta aritmética deben ser rápidos debido a los rendimientos requeridos en los sistemas de comunicación actuales. La suma en $GF(2^m)$ es una operación XOR binaria independiente, puede ser realizada de forma rápida y sin retardo. Sin embargo otras operaciones son mucho más complejas y con mayor retardo. La eficiencia de las implementaciones hardware se mide en términos del número de puertas (XOR y AND) y del retardo total debido a esas puertas del circuito. El objetivo de este documento es hacer un estudio comparativo de diferentes circuitos aritméticos sobre $GF(2^m)$, se utilizarán los cuerpos recomendados por el NIST y el SECG. Por su importancia, se han estudiado diferentes implementaciones para los algoritmos de multiplicación, tanto multiplicación serie como paralela junto con multiplicación dígito serie. Para el estudio de toras operaciones aritméticas, también se estudian algoritmos para obtener el cuadrado y el inverso de elementos pertencientes a $GF(2^m)$. Para realizar este trabajo se implentarán los algoritmos mencionados en VHDL para FPGAs estudiando el consumo de área y tiempo de las operaciones comparando los resultados entre sí y con los obtenidos por otros autores.

Summary: Finite field arithmetic has received much attention due to its importance in cryptography, error control coding and digital signal processing. A large portion of time from the routines of the cryptographies algorithms is used in the calculation of arithmetic operations on finite fields. Systems using this arithmetic must be faster because of performance required in current communication systems. Addition in $GF(2^m)$ is bit independent XOR operation, it can be implemented in fast and inexpensive ways. Nevertheless other operations are much more complex and expensive. The efficiency of the hardware implementations is measured in terms of the numbers of gates (XOR and AND) and of the total gate delay of the circuit. The aim of this document is to make a comparative study of different arithmetic circuits over $GF(2^m)$, NIST and SECG recommended fields will be used. Due to multiplication is one of the most complex and important operation in finite field arithmetic, different implementations will be treated, parallel and serial along with digit-serial algorithms. To perform other operations, also inversion and square algorithms over $GF(2^m)$ have been discussed. VHDL implementations of these algorithms for FPGAs have been realized to study time and area consumption and to compare the result each other and with other authors' results.

1. Introducción y objetivos

1.1 Motivación del trabajo

La aritmética sobre cuerpos finitos ha recibido mucha atención debido a su importancia en aplicaciones prácticas en criptografía, teoría de códigos, teoría de comunicación y procesamiento digital de señales. Por ejemplo, una de las mayores aplicaciones de los cuerpos finitos está en la teoría de codificación algebraica – que consiste en la corrección y detección de errores.

Los sistemas que utilizan aritmética sobre cuerpos finitos deben ser rápidos debido al continuo incremento de prestaciones necesario en los sistemas de comunicación actuales. Por este motivo, es muy importante tanto la obtención de nuevos algoritmos y métodos de cálculo de las operaciones aritméticas sobre dichos cuerpos, como la realización de implementaciones hardware y software eficientes de dichas operaciones. En cuanto a las implementaciones hardware, estas se realizan normalmente sobre circuitos integrados VLSI [15], aunque cada vez es más frecuente la utilización de plataformas reconfigurables (FPGAs)[16]. Asimismo, también se pueden implementar coprocesadores específicos para una determinada aplicación (por ejemplo, coprocesadores criptográficos).

Los cuerpos finitos [24] con p elementos se representan como GF(p). Los cuerpos finitos de gran importancia por sus aplicaciones técnicas son los cuerpos de extensión del cuerpo binario GF(2), representados como $GF(2^m)$, generados por un polinomio irreducible de grado m. El cuerpo finito GF(2) tiene dos elementos, 0 y 1. La suma y la multiplicación se realizan módulo 2. $GF(2^m)$ es la extensión del cuerpo GF(2) y tiene 2^m elementos. Cada uno de estos elementos se representa como un polinomio de grado menor o igual a m -1 con coeficientes pertenecientes al cuerpo base GF(2). Con esta representación, la adición y substracción – que es igual que la suma – se realizan de forma sencilla bit a bit. Sin embargo, la multiplicación y la división implican la multiplicación y división módulo un polinomio irreducible f(x), que es mucho más compleja.

La representación de los elementos del cuerpo tiene un papel fundamental en la eficiencia de las arquitecturas que implementen las operaciones aritméticas sobre dichos cuerpos. Existen distintas bases de representación [16] para los elementos de $GF(2^m)$. Entre ellas, las más importantes son las bases polinómica, normal y dual, aunque existen otras bases como la triangular, etc. Asimismo, la complejidad de las operaciones aritméticas (multiplicación, inversión, división, exponenciación, etc.) sobre dichos cuerpos depende fundamentalmente tanto de la base de representación seleccionada como del polinomio irreducible generador del

cuerpo elegido.

La base de representación utilizada habitualmente es la polinómica, por lo que las operaciones aritméticas basadas en la representación de base normal o dual requieren etapas adicionales de conversión de base, cuya complejidad es muy dependiente del polinomio irreducible f(x) seleccionado. Además, la representación en base polinómica es más eficiente en el sentido de que proporciona al diseñador más libertad en la elección del polinomio irreducible y en la optimización de hardware.

La multiplicación es una de las operaciones más costosas e importantes en la aritmética sobre cuerpos finitos. Es una operación en la que se basan otras operaciones tales como la exponenciación, la inversión, etc. Además es la operación básica que más recursos va a consumir – área y tiempo de ejecución – por lo que es muy importante un diseño adecuado y una optimización del rendimiento de los algoritmos multiplicativos.

Existen numerosas arquitecturas de multiplicadores en base polinómica, pudiendo realizar la multiplicación en serie – de forma iterativa en varios ciclos de reloj – o paralelo – ejecutando todo el proceso en un sólo ciclo mediante lógica combinacional –. A su vez existen arquitecturas sistólicas [9], que tienen la ventaja de la modularidad y la comunicación local, pero tienen el inconveniente de una mayor latencia y una mayor cantidad de elementos de almacenamiento. Los multiplicadores paralelos poseen un menor camino crítico y un mayor rendimiento, siendo por ello los más adecuados para aplicaciones que requieren alta velocidad, una tendencia que ha dominado el diseño de sistemas VLSI durante las últimas dos décadas. Las arquitecturas serie, que procesan 1 bit de entrada por cada ciclo de reloj, utilizan el área de manera eficiente y son adecuadas para aplicaciones de baja velocidad[18].

En sistemas con requerimientos moderados ambas arquitecturas pueden no ser efectivas; unas por ser demasiado lentas y otras por ocupar demasiada área. Con este fin, los sistemas digit-serial han empezado a ser muy atractivos para los diseñadores [23]. Estos sistemas procesan múltiples bits de la palabra de entrada, conocidos como tamaño del dígito, por ciclo de reloj. Este tamaño puede variar desde 1 bit a la longitud entera de la palabra de entrada para lograr un compromiso entre velocidad, área y las limitaciones de las entradas.

La importancia del estudio de arquitecturas para cuerpos finitos se basa en el hecho de que una gran porción de tiempo de las rutinas de los algoritmos criptográficos se emplea en el cálculo de operaciones aritméticas sobre cuerpos finitos. Por ejemplo, en el caso de sistemas criptográficos asimétricos como el RSA, DSA o ECC, la mayoría del tiempo se utiliza para computación de multiplicaciones modulares. Por tanto el rendimiento de estas rutinas afecta

directamente al rendimiento de todo el sistema criptográfico. Además, aunque existen diversos algoritmos eficientes sobre cuerpos finitos para aplicaciones de procesamiento de señales, las implementaciones prácticas para criptografía varían debido al relativamente gran tamaño de los operandos en estas aplicaciones. Una sola operación de cifrado de clave pública puede implicar miles de multiplicaciones modulares de 1.024 bits o mayores. Además, hay estrictas restricciones de hardware que deben cumplirse para una plataforma de destino determinada como tarjetas inteligentes[4] y etiquetas RFID [3], donde deben cumplir unos criterios muy definidos de consumo de energía y área.

Recientemente el diseño de sistemas VLSI de bajo consumo de potencia se ha vuelto un área activa de investigación debido a la demanda de aplicaciones portátiles de arquitecturas DSP (Procesamiento Digital de Señales). En dispositivos wireless el consumo de potencia determina el tiempo entre dos cargas sucesivas así como la vida del dispositivo y de la batería, por lo que la reducción del consumo de potencia es vital en estos sistemas. La principal fuente de disipación de potencia en un circuito CMOS es la actividad de las puertas lógicas, que puede contribuir con más del 90% del total de la energía consumida. Por tanto, la reducción de la complejidad de los circuitos lógicos es fundamental para un mayor rendimiento y un menor consumo de potencia.

1.2 Objetivos y plan de trabajo

El objetivo de este trabajo es hacer un estudio comparativo de diferentes circuitos aritméticos sobre GF(2m), entre otros se utilizan los cuerpos recomendados por el NIST y el SECG [22,26,8]. En especial, debido a su importancia, se pretenden estudiar diferentes implementaciones para los algoritmos de multiplicación, tanto multiplicación serie como paralela junto con multiplicación dígito serie. Para el estudio de otras operaciones aritméticas, también se estudian algoritmos para obtener el cuadrado y el inverso de elementos pertenecientes a $GF(2^m)$.

Para llevar a cabo este trabajo se realizarán implementaciones en VHDL para FPGAs de distintos algoritmos [21,15,16,9] para arquitecturas serie y paralelo, así como para la multiplicación en base polinómica y normal, estudiando el consumo de área y tiempo de las diferentes implementaciones en la FPGA. A su vez, se realizará una comparación experimental del consumo de área y tiempo de las diferentes operaciones entre sí y con los resultados obtenidos por otros autores[20,21], de donde se extraerán conclusiones a cerca del rendimiento de las mismas.

Esta memoria se estructura de la siguiente forma: en la primera parte de este trabajo se muestra una breve introducción teórica a la aritmética sobre los cuerpos finitos, ya que es necesario para una posterior comprensión de la notación y los algoritmos utilizados, además también se incluye una descripción de la lógica de funcionamiento de las FPGAs, pues sobre ellas se implementan las operaciones aritméticas de este estudio. La segunda parte de este trabajo consiste en la descripción de las operaciones aritméticas sobre $GF(2^m)$ que se han seguido para la implementación en VHDL de los multiplicadores [19,25,1,14,21,17]. En esta misma sección, también pueden verse los algoritmos de cada operación y la lógica cicuital empleada para su realización. En la tercera parte se encuentran los resultados experimentales obtenidos con cada una de las implementaciones realizadas así como las distintas tablas y gráficas comparativas, junto con los resultados obtenidos por otros autores. A continuación se reflejan las conclusiones que se desprenden de los datos obtenidos en los anteriores apartados, además de mostrarse las posibles vías de trabajo que se seguirán y desarrollarán a partir de este estudio.

2. Conceptos previos

2.1 Cuerpos finitos

El cuerpo finito [24] GF(2^m) contiene 2^m elementos. Es una extensión del cuerpo GF(2), que posee el elemento cero y el uno. Todos los cuerpos finitos contienen un elemento cero, un elemento unidad, un elemento primitivo y tienen por lo menos un polinomio primitivo irreducible $f(x)=x^m+p_{m-1}x^{m-1}+...+p_1x+p_0$ sobre GF(2) asociado al él. El elemento primitivo α genera todos los elementos distintos de cero de GF(2^m) y es una raíz del polinomio primitivo *f(x)*. Los elementos distintos de cero de GF(2^m) pueden representarse en dos formas, exponencial y polinómica. De forma *exponencial* se representan como potencias del elemento primitivo α, esto es

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

que también se denomina representación en *potencias*. Esta representación es eficiente para la multiplicación, división y exponenciación en cuerpos finitos.

Escribiendo el polinomio irreducible *f(x)* como *f(x)* = x^m + *F(x)* donde $F(x)=f_{m-1}x^{m-1}+...+f_1x+f_0$. Si α es una raíz del polinomio irreducible *f(x)*, se tiene

$$\alpha^m=f_{m-1}\alpha^{m-1}+f_{m-2}\alpha^{m-2}+...+f_1\alpha+f_0 \quad (2)$$

que es equivalente a α^m = *F(x)*. Por tanto, los elementos de GF(2^m) pueden también expresarse como polinomios de α con un grado menor que *m* por medio de la operación *mod f(x)* para $\alpha^k, 0\leq k\leq 2^m-2$ de acuerdo con (2). A esta representación se la conoce como *polinómica*. Por lo tanto, se tiene

$$GF(2^m) = \{ A=a_{m-1}\alpha^{m-1}+a_{m-2}\alpha^{m-2}+...+a_1\alpha+a_0 \quad donde \quad a_i\in GF(2),\ 0\leq i\leq m-1 \} \quad (3)$$

La base $\{1,\alpha,\alpha^2,...,\alpha^{m-1}\}$ de la representación polinómica, se conoce como *base estándar* o *polinómica.*

De entre todas las posibles bases sobre GF(2^m) existen dos tipos de bases de mayor importancia. Una es la *base polinómica* ya mencionada, construida con las potencias del elemento primitivo α. La otra base más importante es la *base normal*. Una base $\{\beta_0,\beta_1,...,\beta_{m-1}\}$ será una base normal para GF(2^m) si $\beta_i=\alpha G_i$ definiendo G_i como el sigueinte automorfismo

$$G_i: GF(2) \rightarrow GF(2)$$
$$\alpha \rightarrow \alpha^{2^i}=\alpha G_i,\ \ \alpha\in GF(2) \quad (4)$$

siendo $G_i=G_I^i$ y $G_1^m=G_1^0=G_0=I$ el automorfismo identidad. Por tanto, el conjunto

$\{\alpha, \alpha^2, \alpha^{2^2}, \ldots, \alpha^{2^{m-1}}\}$ será una *base normal* si los *m* elementos son linealmente independientes y α será el elemento generador de la base normal .

Se define la multiplicación en base polinómica de la siguiente manera. Sean A y B dos elementos pertenecientes al cuerpo GF(2^m) representados como

$$A = a_{m-1}\alpha^{m-1} + \ldots + a_1\alpha + a_0$$
$$B = b_{m-1}\alpha^{m-1} + \ldots + b_1\alpha + b_0 \quad (5)$$

La suma o la resta de dos elementos en GF(2^m) se define como la suma o la resta polinómica sobre GF(2) respectivamente. Por tanto, estas dos operaciones se ejecutan mediante el empleo de puertas XOR para cada coeficiente.

Si definimos *C* como el producto de A por B, $C = A \cdot B = c_{m-1}\alpha_{m-1} + \ldots + c_1\alpha + c_0$, se obtendrán coeficientes de grado mayor que *m,* por lo que es necesario hacer una reducción módulo el polinomio irreducible *f(x)* generador del cuerpo. Para ello basta emplear la expresión (2) y sustituir sucesivamente todas las potencias de α mayores o iguales a *m*.

Para la multiplicación en *base normal* sean $T = (t_{0,} t_{1,\ldots}, t_{m-1})$ y $K = (k_{0,} k_{1,\ldots}, k_{m-1})$ dos elementos de GF(2^m) en una representación en base normal. El producto *C* se puede representar como $C = T \cdot K = (c_{0,} c_{1,\ldots}, c_{m-1})$ y el último término c_{m-1} del producto será alguna función binaria de los componentes de *T* y *K*

$$c_{m-1} = v((t_{0,} t_{1,\ldots}, t_{m-1}), (k_{0,} k_{1,\ldots}, k_{m-1})) \quad (6)$$

Como la potencia cuadrada significa el desplazamiento cíclico de un elemento en la representación en base normal [7,2], para cualesquiera elementos A y B de GF(2^m) , se verifica que $(A+B)^2 = A^2 + B^2 + 2AB = A^2 + B^2$, ya que 2AB = 0. Del teorema de Fermat se obtiene que $A^{2^m - 1} = 1$ por lo que $A^{2^m} = A$ es decir, el cuadrado se realiza simplemente con un desplazamiento cíclico a la izquierda, su implementación es muy poco costosa, entonces

$$C^2 = T^2 \cdot K^2 = (t_{m-1}, t_{0,} t_{1,\ldots}, t_{m-2}) \cdot (k_{m-1}, k_{0,} k_{1,\ldots}, k_{m-2}) = (c_{m-1}, c_{0,} c_{1,\ldots}, c_{m-2}) \quad (7)$$

por tanto, el último componente c_{m-2} de C^2 se obtiene de la función *v* dada en (6), es decir

$$c_{m-2} = v((t_{m-1}, t_{0,} t_{1,\ldots}, t_{m-2}), (k_{m-1}, k_{0,} k_{1,\ldots}, k_{m-2})) \quad (8)$$

Elevando *C* al cuadrado repetidamente se obtiene

$$\begin{gathered} c_{m-1} = v((t_{0,} t_{1,\ldots}, t_{m-1}), (k_{0,} k_{1,\ldots}, k_{m-1})) \\ c_{m-2} = v((t_{m-1}, t_{0,} t_{1,\ldots}, t_{m-2}), (k_{m-1}, k_{0,} k_{1,\ldots}, k_{m-2})) \\ \ldots \\ c_0 = v((t_{1,} t_{2,\ldots}, t_{m-1}, t_0), (k_{1,} k_{2,\ldots}, k_{m-1}, k_0)) \end{gathered} \quad (9)$$

Estas ecuaciones definen la multiplicación en la base normal. Se requiere únicamente una función lógica *v* de los *2m* componentes de *T* y *K* para el cálculo secuencial de los *m*

componentes del producto.

2.2 Dispositivos FPGAs

Una FPGA – *Field Programmable Gate Array* – es un circuito integrado diseñado para ser configurado por el usuario después de su fabricación, para ello dispone de una matriz de bloques lógicos configurables (CLBs). En este trabajo se han utilizado FPGAs fabricadas por Xilinx, en ellas, cada CLB está compuesto de dos *slices,* dicho *slice* tiene dos registros y dos generadores de funciones, que son dos LUTs – *Look Up Table* – con cuatro entradas cada una.

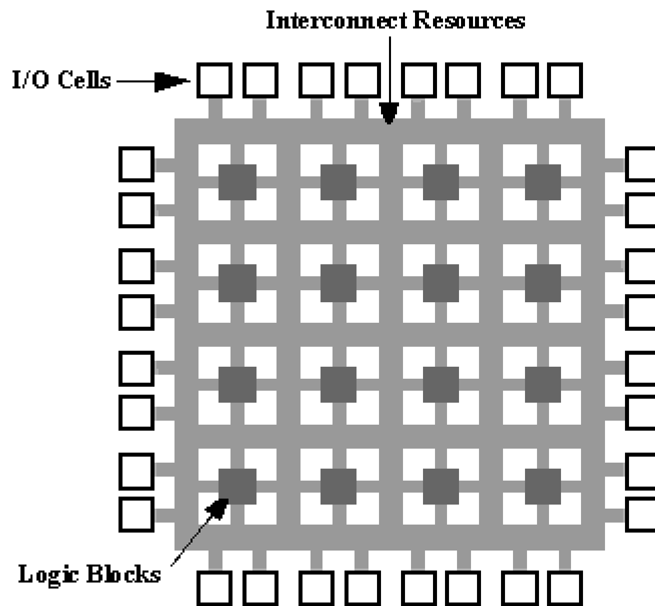


Figura 1: Arquitectura interna de una FPGA

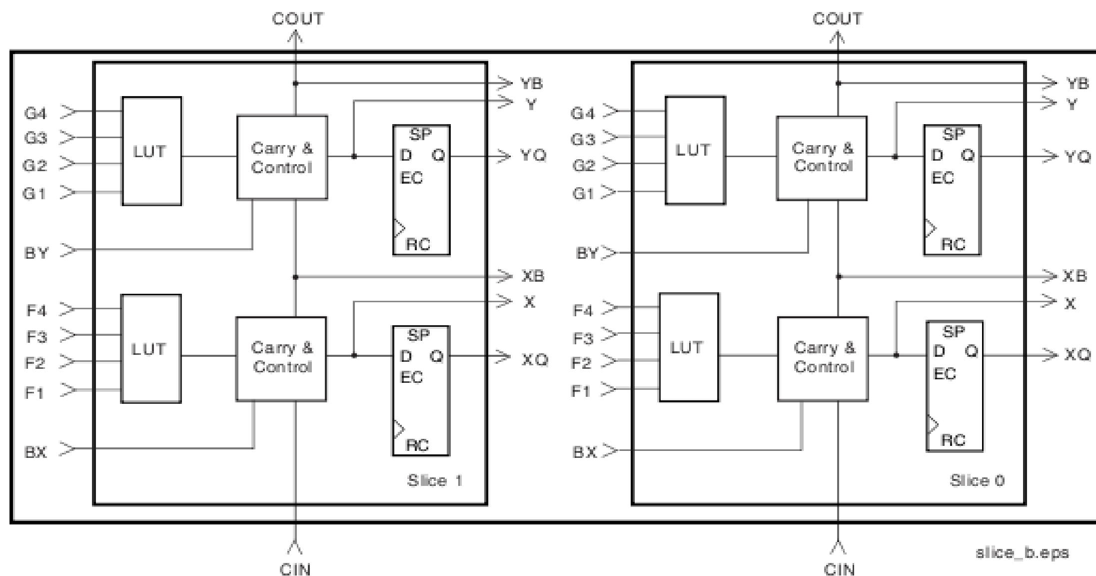


Figura 2: CLB de dos slices de una FPGA Virtex de Xilinx

Otros bloques importantes en la FPGA son el bloque de control de los puertos de entradasalida y el bloque de memoria. Las memorias pueden ser de tipo ROM, RAM,

EPROM, etc. Las FPGAs Virtex poseen una gran cantidad de bloques de memoria RAM que se organizan en dos columnas a lo largo de todo el chip. Por ejemplo la FPGA Virtex XCV300 posee 65,536 bloques de memora RAM [27]. El número de bloques de entrada salida es también variable en función de la FPGA escogida, la mencionada anteriormente posee 316 bloques I/O [28].

Para programar una FPGA en primer lugar hay traducir las funciones deseadas a un lenguaje de descripción de hardware, en este trabajo se ha usado el lenguaje VHDL – Very Hardware Description Languaje – que permite describir circuitos síncronos y asíncronos. La estructura básica de este lenguaje consiste en describir entidades que son la abstracción de un circuito, que puede ser desde un sistema muy complejo a una simple puerta lógica, en las que se describen las entradas y salidas del circuito. Asociadas a estas entidades deben diseñarse las arquitecturas; las arquitecturas describen el funcionamiento de la entidad a la que hace referencia. Se pueden utilizar principalmente dos descripciones para las arquitecturas, estructural y de comportamiento. En la primera se describe las interconexiones entre entidades definidas previamente, por tanto es una descripción de alto nivel ya que equivaldría al diseño de un circuito utilizando componentes sin conocer cuál es la estructura interna de cada componente, sino sólo su función. La descripción por comportamiento consta de una serie de instrucciones, que ejecutadas modelan el funcionamiento del circuito, esto es, para una puerta lógica, por ejemplo, se escribiría su tabla de verdad traducida a VHDL.

Posteriormente se realiza la síntesis de los diseños en VHDL para traducirlos a CLBs en la FPGA, para ello se emplean software de diseño como el proporcionado por Xilinx [30]. En este proceso además de compilar el código VHDL, se realiza una simulación de donde se extraen resultados teóricos del diseño en la FPGA. En el proceso siguiente de implementación ya se obtienen resultados reales, pues el software se encarga de trazar y unir los bloques de entrada / salida con los CLBs, memorias, etc; y también se obtiene el mapa de interconexiones y de los pines de entrada / salida. Para ello realiza tres procesos, translate, map y place & route, que efectúan los procesos mencionados anteriormente.

Las FPGAs están tomando relevancia en la computación de altas prestaciones, puesto que pueden acelerar segmentos críticos en las aplicaciones de alto rendimiento. Ello conlleva a que el uso de FPGAs en la implementación sobre cuerpos finitos GF(2 m) haya aumentado recientemente [21,20], aunque es un campo en el que aún no se ha profundizado mucho.

3. Diseño e implementación de operaciones aritméticas sobre GF(2m)

En esta sección se presentan los algoritmos, métodos y arquitecturas que se han utilizado para la implementación de las distintas operaciones aritméticas sobre $GF(2^m)$.

Se han desarrollado arquitecturas para la multiplicación en paralelo siguiendo el método de Mastrovito[25,1] para polinomios generales y para AOPs [6] en base polinómica, y para la base normal tipo I con AOPs [7]. A su vez, se han implementado arquitecturas para la multiplicación serie mediante los algoritmos de Least Significant Bit y Most Significant Bit [14]. Por otro lado se ha realizado un multiplicador con arquitectura mixta Dígito – Serie, es decir, que opera con grupos de bits siguiendo un algoritmo de multiplicación en serie. También se ha implementado la inversión[17,11,12,29,14] y el cuadrado [13].

Para poder implementar estas arquitecturas en lógica combinacional y secuencial, en primer lugar se estudió la aritmética de los distintos algoritmos matemáticos de cada método. Se analizaron varias arquitecturas para cada algoritmo de manera que se pudiese elegir la que mejor se adaptase a nuestras expectativas de diseño, es decir, que presentase una estructura descomponible en lógica combinacional y secuencial. Por último se procedió al desarrollo de una arquitectura programable a partir de los algoritmos obtenidos, para poder realizar la implementación en FPGAs y obtener los resultados experimentales pertinentes.

Todos los algoritmos se han implementado en VHDL para FPGAs mediante el software de diseño de hardware ISE Xilinx 10.1 WebPACK. A su vez, estos diseños fueron verificados con el programa Maple 12.01.

3.1 Multiplicación

3.1.1 Multiplicador de Mastrovito para polinomios genéricos irreducibles

En primer lugar debemos dar una notación matricial para la multiplicación en el cuerpo de Galois $GF(2^m)$

$$C(x)=A(x)B(x) \bmod f(x) \tag{10}$$

donde el polinomio irreducible es de la forma

$$f(x)=x^m+f_{m-1}x^{m-1}+\cdots+f_1x+1, \quad f_i \in GF(2) \tag{11}$$

Todos los elementos del cuerpo son polinomios de grado menor que m cuyos coeficientes pertenecen al cuerpo GF(2):

$$c_{m-1}x^{m-1}+\cdots+c_0=(a_{m-1}x^{m-1}+\cdots+a_0)(b_{m-1}x^{m-1}+\cdots+b_0) \bmod f(x) \tag{12}$$

Los elementos *B(x)* y C*(x)* también se pueden expresar como vectores columna que contengan los coeficientes del polinomio. Utilizando la matriz **Z**=*h(A(x),f(x))* se puede

describir la multiplicación como:

$$C=\begin{pmatrix} c_o \\ c_1 \\ \dots \\ c_{m-1} \end{pmatrix}=\mathbf{Z}\,B=\begin{pmatrix} h_{oo} & \dots h_{0,m-1} \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ h_{m-1,0} & \dots h_{m-1,0} \end{pmatrix}\begin{pmatrix} b_0 \\ b_1 \\ \dots \\ b_{m-1} \end{pmatrix} \tag{13}$$

La matriz **Z** se denomina *matriz producto* o de *Mastrovito*. Sus coeficientes $h_{i,j}$ Є $GF(2)$ dependen recursivamente de los coeficientes a_i y de los coeficientes $f_{i,j}$ de la matriz **F** (8) de la siguiente forma

$$h_{i,j}=\begin{cases} a_i & j=0\,;i=0,1\dots,m-1 \\ u(i-j)a_{i-j}+\Sigma_{t=0}^{j-1} f_{j-1-t,i}\,a_{m-1-t} & j,i=0,1\dots,m-1\,;j\neq 0 \end{cases} \tag{14}$$

donde la función *escalón u* se define como

$$u(\mu)=\begin{cases} 1 & para \quad \mu\geq 0 \\ 0 & para \quad \mu<0 \end{cases} \tag{15}$$

El producto dado en la ecuación (13) describe completamente la multiplicación en el cuerpo finito. La matriz **F** necesaria para la construcción de **Z** es una función del polinomio irreducible binario $f(x)$ de grado m que genera $GF(2^m)$. Sus entradas binarias $f_{i,j}$ se definen de forma que

$$\begin{pmatrix} x^m \\ x^{m+1} \\ \vdots \\ x^{2m-2} \end{pmatrix}=\begin{pmatrix} f_{00} & \dots & f_{0,m-1} \\ f_{1,0} & \cdots & f_{1,m-1} \\ \vdots & \ddots & \vdots \\ f_{m-2,0} & \cdots & f_{m-2,m-1} \end{pmatrix}\begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} mod \quad f(x) \tag{16}$$

La matriz **F** describe la representación de los polinomios x^m, x^{m+1},..., x^{2m-2} en las clases de equivalencia módulo $f(x)$, es decir, después de la reducción módulo $f(x)$.

Las entradas binarias $f_{i\,j}$ de la matriz **F** se pueden calcular recursivamente una vez que la primera fila de la matriz se ha formado con los coeficientes del polinomio $f(x)$, es decir, $f_{0\,j} = f_j$ según la siguiente expresión

$$f_{i,j}=\begin{cases} f_{i-1,m-1} & i=1,\dots,m-2 \quad ;j=0 \\ f_{i-1,j-1}+f_{i-1,m-1}f_{0,j} & i=1,\dots,m-2 \quad ;j=1,\dots,m-1 \end{cases} \tag{17}$$

Para la implementación en VHDL de este multiplicador se ha escogido una implementación combinacional en la que se calculan en paralelo las matrices necesarias para la multiplicación y así efectuar la operación (13). Todo se ha realizado en un único módulo multiplicador que contiene una función para el cálculo de los coeficientes de **F**, otra para los coeficientes de **Z** y otra para realizar el producto.

3.1.2 Multiplicador de Mastrovito para AOP irreducibles

Un AOP (*all one polynomial*) de grado m es un polinomio con todos sus coeficientes distintos de cero de la forma $f(x) = x^m + x^{m-1} + \cdots + x + 1$. Este polinomio es irreducible y genera el cuerpo $GF(2^m)$, si y sólo si $m+1$ es primo y 2 es primitivo módulo $m+1$, siendo muy útil en numerosas aplicaciones. Para $m \leq 100$, se tiene que el AOP es irreducible para los valores de m : 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82 y 100.

El esquema de Mastrovito utiliza dos matrices en el proceso de diseño del multiplicador para el cálculo del producto de dos elementos A y B de $GF(2^m)$ representados en base polinómica: la matriz ($m-1$) x m de *reducción* de base **F** y la matriz m x m *producto* **Z**=h(A(x),f(x)).

La multiplicación en base polinómica sobre el cuerpo finito $GF(2^m)$ generado por un AOP irreducible se puede realizar *descomponiendo* la matriz **Z** en las matrices $\mathbf{Z_1}$ y $\mathbf{Z_2}$. Para construir estas matrices, se debe escribir en primer lugar la matriz **F** para un AOP, utilizando para ello la identidad $x^{m-1} = 1$ como

$$\begin{pmatrix} x^m \\ x^{m+1} \\ \vdots \\ x^{2m-2} \end{pmatrix} = \begin{pmatrix} 1\,1\,1 \ldots 1\,1\,1 \\ 1\,0\,0 \ldots 0\,0\,0 \\ 0\,1\,0 \ldots 0\,0\,0 \\ \vdots\,\vdots\,\vdots \ddots \vdots\,\vdots\,\vdots \\ 0\,0\,0 \ldots 1\,0\,0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} \quad (18)$$

Usando la expresión (18) de **F** y la expresión (14) de **Z** se construye la matriz producto **Z** para el cuerpo $GF(2^m)$ con un AOP como la suma de las matrices $\mathbf{Z_1}$ y $\mathbf{Z_2}$ dadas de la siguiente forma:

$$\mathbf{Z_1} = \begin{pmatrix} a_0 & 0 & a_{m-1} & a_{m-2} & \ldots & a_2 \\ a_1 & a_0 & 0 & a_{m-1} & \ldots & a_3 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & a_{m-5} & \cdots & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0 \end{pmatrix} \quad (19)$$

$$\mathbf{Z_2} = \begin{pmatrix} 0 & a_{m-1} & a_{m-2} & a_{m-3} & \ldots & a_1 \\ 0 & a_{m-1} & a_{m-2} & a_{m-3} & \ldots & a_1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 \\ 0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 \end{pmatrix} \quad (20)$$

Para calcular el producto $C = \mathbf{Z}B = (\mathbf{Z_1} + \mathbf{Z_2})B$, primero se calculan los vectores $D = \mathbf{Z_1} B$ y $E = \mathbf{Z_2} B$ en paralelo, y después se calcula el resultado $C = D + E$. Por tanto para la implementación en VHDL de este multiplicador se empleó un único módulo en el que se

emplean dos funciones para calcular $\boldsymbol{Z}_1$ y $\boldsymbol{Z}_2$, otras dos para D y E , y por último una más para obtener el producto C.

3.1.3 Multiplicador serie con arquitectura Least Significant Bit.

La multiplicación de dos elementos $A, B \in GF(2^m)$ con $A=\Sigma_{i=0}^{m-1} a_i x^i$ y $B=\Sigma_{i=0}^{m-1} b_i x^i$ esta dada por

$$C(x)=\Sigma_{i=0}^{m-1} c_i x^i = A(x)\cdot B(x) \, mod \; f(x) \tag{21}$$

que es equivalente a

$$C(x)=(b_0 a(x)+b_1 a(x)x+b_2 a(x)x^2+...+b_{m-1} a(x)x^{m-1}) \, mod f(x) \tag{22}$$

Se obtendrán coeficientes x^t con $t \geq m$ que deben ser reducidos *módulo f(x),* donde el polinomio irreducible viene dado por $f(x)=x^m+G(x)=x^m+\Sigma_{i=0}^{m-1} g_i x^i$ que pertenece a GF(2^m) y es de grado m. Si asumimos que α es una raíz de $f(x)$, se cumple que $f(\alpha) = 0$. Por lo que

$$x^m=-G(x)=\Sigma_{i=0}^{m-1} -g_i x^i \tag{23}$$

proporcionando un método sencillo para realizar la reducción módulo *f(x)* cuando se encuentran potencias de x mayores *que m-1*. Puesto que estamos en un cuerpo con característica 2, la ecuación (23) queda como

$$x^m=G(x)=\Sigma_{i=0}^{m-1} g_i x^i \tag{24}$$

El algoritmo más simple de multiplicación es el método de desplazamiento y suma con un paso de reducción intercalado [14], tal y como se muestra en el siguiente algoritmo

Input: $A=\Sigma_{i=0}^{m-1} a_i x^i$ $B=\Sigma_{i=0}^{m-1} b_i x^i$ con a_i, b_i Є GF(2^m)
Output: $C=A\cdot B \, mod \; f(x)=\Sigma_{i=0}^{m-1} c_i x^i$ donde c_i Є GF(2^m)

1: C:= 0
2: **for** i=0 **to** m-1 **do**
3: C:= b_i·A + C
4: A:= A·x mod f(x)
5: **end for**
6: Return(C)

En este algoritmo se puede observar que, en efecto, se procesa un bit en cada iteración, realizándose este proceso un número de veces igual al tamaño de las entradas *m.* Esto se realiza desde el bit menos significativo hasta el más significativo del operando B. En el paso 4 hay que realizar una reducción módulo *f(x)*, para ello consideramos que al multiplicar A por x se obtiene

$$A\cdot x=\Sigma_{i=0}^{m-1} a_i x^{i+1} = a_{m-1} x^m + \Sigma_{i=0}^{m-2} a_i x^{i+1} \tag{25}$$

Usando la propiedad de reducción polinómica como en (20), se puede sustituir x^m y la ecuación (25) se reescribe como

$$A \cdot x \bmod f(x) = \Sigma_{i=0}^{m-1}(g_i a_{m-1}) x^i + \Sigma_{i=0}^{m-1} a_i x^i = (g_0 a_{m-1}) + \Sigma_{i=1}^{m-1}(a_{i-1} + g_i a_{m-1}) x^i \quad (26)$$

Para la implementación en VHDL del algoritmo anterior se ha realizado un módulo para la ruta de datos, que contiene el algoritmo en sí, y un módulo para la unidad de control que no es más que una máquina de estados.

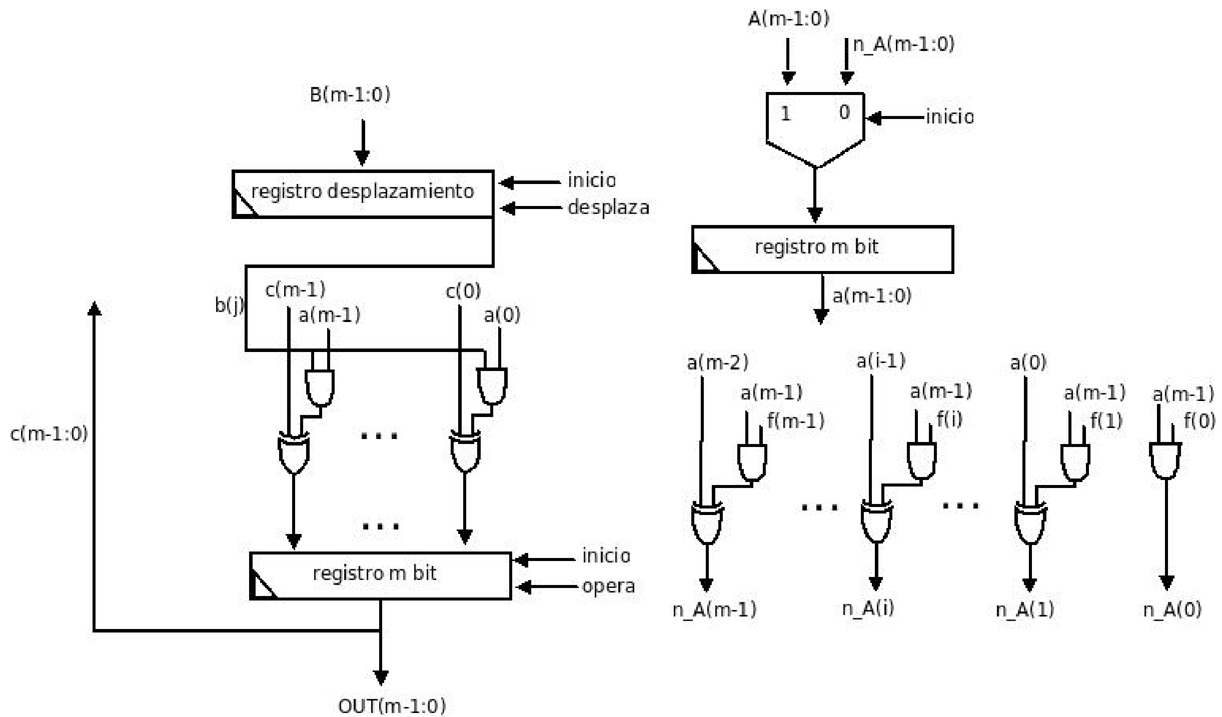


Figura 3: Ruta de datos del multiplicador serie LSB

En la Figura 3 se puede ver el diagrama de la ruta de datos del multiplicador. En este diagrama se diferencian claramente dos partes, la parte izquierda que realiza la operación 3 y la parte derecha que realiza la operación 4 del algoritmo. La operación termina cuando se alcanza el último bit de la entrada B.

Esta ruta de datos consta de un registro de desplazamiento que se encarga de ir obteniendo el bit de B correspondiente a cada iteración, y de dos registros de m bits que cargan los nuevos valores para A y C en función de sus valores anteriores y de las operaciones lógicas correspondientes.

3.1.4 Multiplicador serie con arquitectura Most Significant Bit.

El algoritmo de este multiplicador es muy similar que el del multiplicador LSB de la

sección anterior, salvo que en este caso los bits de *B* se procesan desde el bit más significativo hacía el bit menos significativo. El algoritmo del multiplicador serie MSB es el siguiente

Input: $A=\Sigma_{i=0}^{m-1} a_i x^i$ $B=\Sigma_{i=0}^{m-1} b_i x^i$ con a_i, b_i Є GF(2^m)
Output: $C=A \cdot B mod\ f(x)=\Sigma_{i=0}^{m-1} c_i x^i$ donde c_i Є GF(2^m)

1: C := 0
2: **for** i=m-1 **downto** 0 **do**
3: C := C·x mod f(x) + b_i·A
4: **end for**
5: Return(C)

Se observa que en el paso 3 de este algoritmo se pueden realizar en paralelo las operaciones *C·x mod f(x)* y $b_i \cdot A$ al ser independientes. No obstante, al depender el valor de *C* en cada iteración de su valor en la iteración previa y del valor de $b_i \cdot A$ hace que el camino

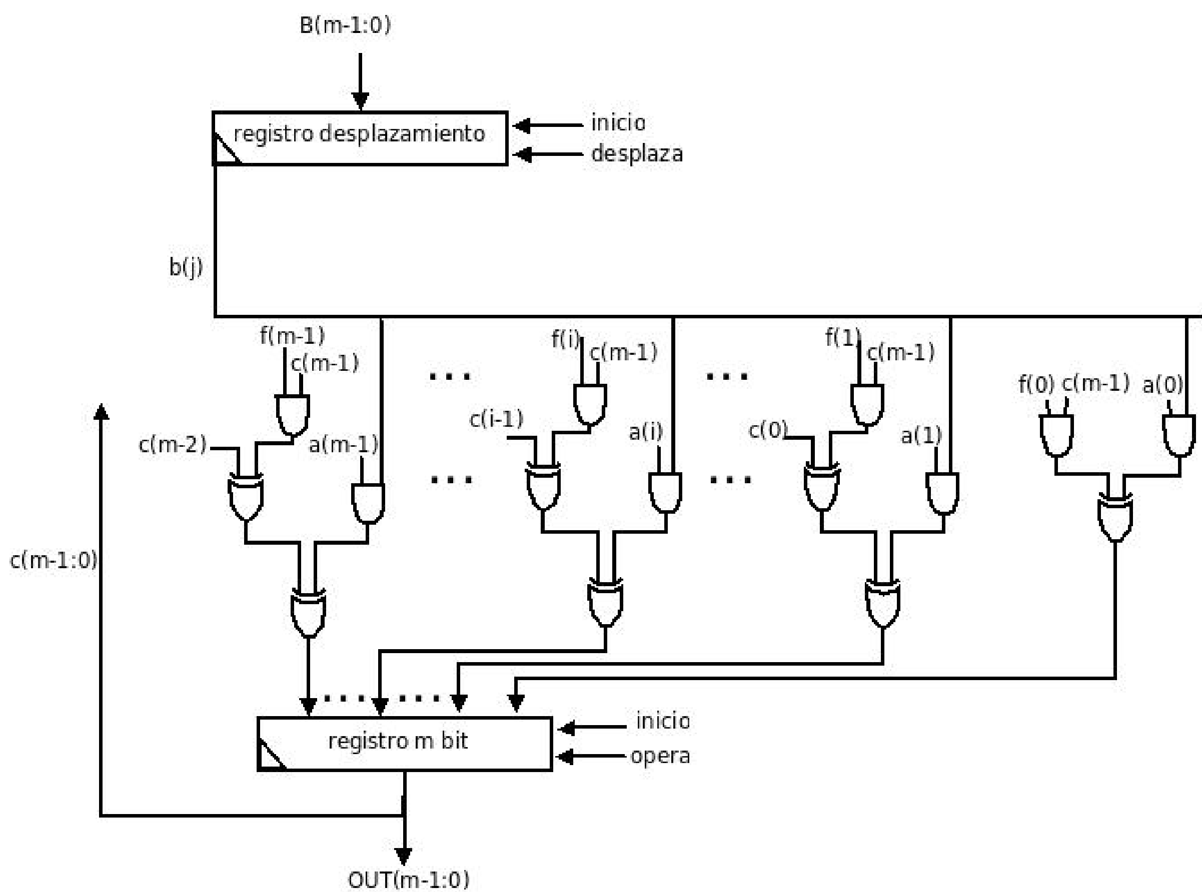


Figura 4: Ruta de datos del multiplicador serie MSB

crítico sea mayor que el del multiplicador serie LSB.

Para la implementación en VHDL, al igual que en el multiplicador serie LSB se ha empleado un módulo para la ruta de datos y otro para la unidad de control. La ruta de datos puede verse en la Figura 4, consiste en un registro de desplazamiento para ir obteniendo los

bits de *B* desde el más significativo al menos significativo, una parte de lógica combinacional en la que se obtienen los valores de *C* correspondientes al paso 3 del algoritmo y un registro de *m* bits para ir almacenando los valores obtenidos de *C* en cada ciclo para utilizarlos en la siguiente iteración. El módulo de control es de nuevo una máquina de estados que se encarga entre otras cosas, de detectar cuándo se ha alcanzado el primer bit de *B* para finalizar la multiplicación.

3.1.5 Multiplicador dígito – serie.

Como se ha comentado anteriormente las arquitecturas dígito – serie son las más adecuadas para sistemas que requieren un rendimiento moderado y donde el área y el consumo de potencia son críticos [19].

Para la realización de este algoritmo asumamos un tamaño de dígitos de *D* bits. Sea *d* el número de dígitos en la palabra $d=[m/D]$ el mayor entero del cociente m/D. Las entradas serán $A=\Sigma_{j=0}^{m-1} a_j x^j$ y $B=\Sigma_{i=0}^{d-1} B_i x^{D_i}$ donde

$$B_i=\begin{cases} \Sigma_{j=0}^{D-1} b_{D_{i+j}} x^j & 0\leq i\leq d-2 \\ \Sigma_{j=0}^{m-1-D(d-1)} b_{D_{i+j}} x^j & i=d-1 \end{cases} \quad (27)$$

Entonces

$$C=A\cdot B \bmod f(x)=A\cdot\Sigma_{i=0}^{d-1} B_i x^{D_i} \bmod f(x) \quad (28)$$

De acuerdo con esto tenemos la siguiente ecuación para el multiplicador

$$C=(B_0 A+B_1(A\cdot x^D \bmod f(x))+B_2(Ax^D\cdot x^D \bmod f(x)))+...$$
$$...+B_{d-1}(Ax^{D(d-2)}\cdot x^D \bmod f(x)) \bmod f(x) \quad (29)$$

para el esquema *Least Significant Digit* que es el que se ha usado en este trabajo.

Por tanto se obtiene el siguiente algoritmo para este multiplicador

Input: $A=\Sigma_{i=0}^{m-1} a_i x^i \quad B=\Sigma_{i=0}^{d-1} B_i x^{D_i}$
Output: $C=A\cdot B \bmod f(x)=\Sigma_{i=0}^{m-1} c_i x^i$

1: C := 0
2: **for** i=0 **to** d-1 **do**
3: C := B_i·A +C
4: A := A·x^D mod f(x)
5: **end for**
6: Return(C mod f(x))

Para la operación de reducción se consideran los siguientes teoremas [19]

Teorema 1: *Asumiendo que* $f(x)=x^m+f_k x^k+\Sigma_{j=0}^{k-1} f_j x^j$ *. Para t ≤ m – 1 – k, las coordenadas de* x^{m+t} *pueden obtenerse a partir de la siguiente ecuación*

$$x^{m+t} \bmod f(x)=(x^m \bmod f(x))\cdot x^t=f_{kx}^{k+t}+\Sigma_{j=0}^{k-1} f_j x^{j+t} \quad (30)$$

esto es, este grado puede reducirse a uno menor que m en un sólo paso.

Teorema 2: *Para la multiplicación dígito – serie con tamaño de dígito D, cuando D ≤ m – k, el grado de los resultados intermedios puede reducirse a un grado menor que m en un sólo paso y la operación de reducción de grado* mod f(x) *puede implementarse usando un árbol binario de puertas XOR tal que su tiempo de computación se reduce de ser proporcional a D a ser proporcional al logaritmo de D.*

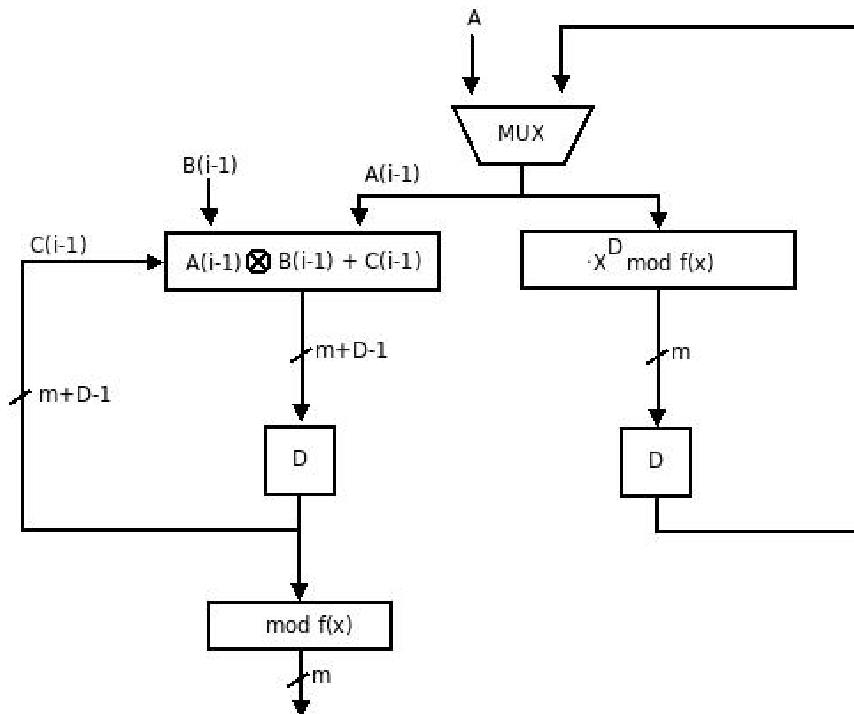
Existen dos términos que necesitan la operación de reducción módulo *f(x),* son *A* en la operación *4* del algoritmo y *C* en la operación 6. El grado máximo de *A* es *m+D-1* y el de *C* es *m+D-2*. Así pues para ambos podemos asumir el resultado intermedio $W^{(0)}$ como sigue

$$W^{(0)}=w_{m+D-1}^{(0)}x^{m+D-1}+\ldots+w_m^{(0)}x^m+w_{m-1}^{(0)}x^{m-1}+\ldots+w_0^{(0)} \tag{31}$$

Teniendo en cuenta que $f(x)=x^m+f_k x^k+\Sigma_{j=0}^{k-1}f_j x^j$. Sustituyendo $x^m=x^k+\Sigma_{j=0}^{k-1}f_j x^j$ en (30) para la reducción de grado, se tiene

$$W^{(1)}=w_{m+D-1}^{(0)}x^{D-1}\left(f_k x^k+\Sigma_{j=0}^{k-1}f_j x^j\right)+w_{m+D-2}^{(0)}x^{D-2}\left(f_k x^k+\Sigma_{j=0}^{k-1}f_j x^j\right)+\ldots$$
$$\ldots+w_m^{(0)}\left(f_k x^k+\Sigma_{j=0}^{k-1}f_j x^j\right)+w_{m-1}^{(0)}x^{m-1}+w_{m-2}^{(0)}x^{m-2}+\ldots+w_0^{(0)} \tag{32}$$

Donde *D*≤ *m-k*, *D-1* ≤ *m-1-k*, por el teorema 1, el grado de cada resultado temporal, $w_{m+t}^{(0)}x^t(f_k x^k+\Sigma_{j=0}^{k-1}f_j x^j)$ para 0≤t≤D-1*,* es menor que *m*. Por tanto, no se requieren más operaciones para la reducción de grado. Todos estos resultados temporales pueden obtenerse y acumularse simultáneamente usando un árbol binario de puertas XOR, como se muestra en la figura 6, para *m = 8* y *D = 3*..



Como en anteriores diseños se han empleado dos módulos, el módulo correspondiente a la ruta de datos y el módulo correspondiente a la unidad de control. En la Figura 5 se muestra el esquema usado para la implementación en VHDL de la ruta de datos de este algoritmo. Este esquema consta de dos bucles. El bucle derecho realiza la operación 4 y el bucle izquierdo la operación 3 del algoritmo. Los resultados parciales de estas operaciones se almacenan en los biestables tipo D hasta la siguiente iteración.

La operación 3 del algoritmo que se realiza en el bucle de la izquierda de la figura 5 [19], es una multiplicación de los vectores A de m elementos y de B_i de D elementos. Por tanto el número de puertas AND y XOR empleadas dependerá del tamaño D del dígito. A su vez las operaciones de reducción módulo $f(x)$ dependerán del polinomio empleado.

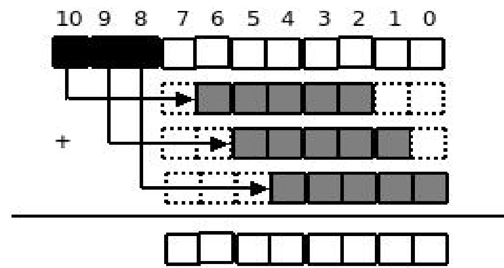


Figura 6: Esquema de reducción de grado, con D=3 y m=8.

Para la realización de este multiplicador se han implementado varias arquitecturas con distinto tamaño de dígito D para un mismo tamaño de palabra. De esta manera se puede comprobar cómo influye este parámetro en el rendimiento y en el área usada por el dispositivo. Los límites al tamaño de D vienen determinados por el Teorema 2.

3.1.6 Multiplicador de Mastrovito en base normal óptima de Tipo I

Sea $N=\{\zeta, \zeta^2, \zeta^{2^2}, \ldots, \zeta^{2^{m-1}}\}$ una base normal de $GF(2^m)$, donde $\zeta \in GF(2^m)$ y m es el grado del polinomio generador. Una base normal óptima cumple que $\forall\, o \leq i_1 \neq i_2 \leq m-1$ existen j_1, j_2 tales que $\zeta^{2^{i_1}+2^{i_2}} = \zeta^{2^{j_1}} + \zeta^{2^{j_2}}$ siendo ζ el generador de la base.

Sea $m+1$ un primo p y sea 2 un primitivo módulo p (es decir, el orden multiplicativo de 2 módulo p es m), y supóngase también que ζ es una raíz primitiva (m+1)-ésima de la unidad. Entonces, los m conjugados distintos de ζ son también raíces primitivas (m+1)-ésimas de la unidad y constituyen una *base normal óptima de Tipo I*, con elemento *generador* ζ, de forma que

$$N=\{\zeta, \zeta^2, \zeta^{2^2}, \ldots, \zeta^{2^{m-1}}\} = \{\zeta, \zeta^{2,} \zeta^{3,\ldots}, \zeta^m\} \quad (33)$$

y donde se observa que

$$\zeta\zeta^{i}=\zeta^{i+1}\in N,\quad 1\leq i\leq m \tag{34}$$

$$\zeta\zeta^{m}=1 \tag{35}$$

También se observa que ζ es una raíz del polinomio $x^m + x^{m-1} + \cdots + x + 1$.

Tal y como se vio en 2.2, un AOP es irreducible si y sólo si *m+1* es primo y 2 es primitivomódulo *m+1*. Por tanto, una raíz ζ de un AOP irreducible verifica la propiedad $\zeta^{m+1}=1$, es decir, ζ es una raíz primitiva (m+1)-ésima de la unidad.

Por lo tanto, se concluye que un *N – polinomio* generador de una *base normal óptima de Tipo I* es también un AOP y que la raíz de un AOP irreducible es un elemento *generador* de dicha base.

La base dada en (33) se puede considerar una versión *desplazada* de la base polinómica, por lo que se pueden utilizar la arquitectura vista en 3.1.2 de multiplicación en base polinómica para AOP irreducibles para su aplicación a la multiplicación en base *normal* cuando se utiliza un AOP irreducible como polinomio generador del cuerpo finito.

Un elemento $A \in GF(2^m)$ representado en la *base normal óptima de Tipo I* se puede convertir a su representación en la *base polinómica desplazada* utilizando sencillamente una permutación de las coordenadas de *A* con respecto de la base normal *N*.

Llamando Γ a la base canónica desplaza y a_{Γ_i} un elemento perteneciente a dicha base. La conversión a $a_{N_i}\in N$

$$A=\Sigma_{i=0}^{m-1}a_{N_i}\zeta^{2^i}=\Sigma_{i=0}^{m-1}a_{\Gamma_i}\zeta^{i+1} \tag{36}$$

se puede realizar utilizando la permutación definida por

$$a_{\Gamma_{(2^i-1)mod(m+1)}}=a_{N_i}\qquad i=0,1,\ldots,m-1 \tag{37}$$

Por lo tanto, utilizando las relaciones anteriores, se puede realizar la multiplicación en *base normal óptima de Tipo I* sobre el cuerpo finito GF(2^m) generado por un AOP irreducible con el siguiente algoritmo:

I. Tomar los operandos de entrada *A* y *B* representados en base normal
II. Realizar la permutación (37) para convertir los operandos a la base polinómica desplazada
III. Ejecutar la multiplicación en base polinómica para AOP irreducibles
IV. Realizar la permutación inversa a (37) para obtener el resultado en la *base normal óptima de Tipo I*.

La multiplicación en base polinómica desplazada es muy parecida a la multiplicación en

base polinómica, pero no idéntica, pues existen diferencias a la hora de calcular el resultado antes de realizar la permutación inversa [7].

Al igual que en el multiplicador anterior se ha escogido una implementación combinacional para el desarrollo del algoritmo en VHDL. Se utilizó el multiplicador anterior añadiéndole las dos operaciones de cambio de base mencionadas. Éstas permutaciones se realizan simplemente por medio de cableado. Por lo tanto, la complejidad de este multiplicador – tanto temporal como espacial – es la misma que el multiplicador en base polinómica para AOP, es decir, requiere el mismo número de puertas AND y XOR que el multiplicador mencionado.

3.2 Cuadrado

El cuadrado en $GF(2^m)$, siendo $A=\Sigma_{i=0}^{m-1} a_i x^i$ la representación de un elemento arbitrario de ese cuerpo, es

$$C=\Sigma_{i=0}^{m-1} c_i x^i = A^2 \, mod \; f(x) = a_0 + a_1 x^2 + a_2 x^4 + \ldots$$
$$\ldots a_{[\frac{m}{2}]} x^{2[\frac{m}{2}]} + \ldots + a_{m-1} x^{2m-2} \, mod \; f(x) \quad (38)$$

Cuando $f(x)$ es un trinomio irreducible se puede reducir la longitud crítica y el número de puertas [13]. Considerando el siguiente trinomio $f(x)=x^m+x^k+1, \;\; 1 \le k \le m/2$. Entonces

$$\Sigma_{i=0}^{m-1} c_i x^i = \Sigma_{i=0}^{m-1} a_i x^{2i} = \Sigma_{i=0}^{2m-2} a'_i x^i \quad (39)$$

donde a'_i viene dada por

$$a'_i = \left\{ \begin{array}{l} a_{\frac{i}{2}} \;\; si \; i \; es \; par \\ 0 \;\; en \; otro \; caso \end{array} \right\} \quad (40)$$

Según sea k y m tendremos distintos casos. Para la realización de este algoritmo se ha escogido $k = 3$ y $m = 10$. De acuerdo con [13] tendremos las siguientes ecuaciones para el cuadrado en $GF(2^m)$ y $f(x) = x^m + x^k + 1$.

$c_i = a'_i + a'_{m+i}$	i=0, 2,..., k-2
$c_i = a'_{2m-k+i}$	i=1, 3,..., k-1
$c_i = a'_i + a'_{m+i} + a'_{2m-2k+i}$	i=k+1, k+3,..., 2k-2
$c_i = a'_{m-k+i}$	i=k, k+2,..., m-2
$c_i = a'_i + a'_{m+i}$	i=2k, 2k+2,..., m-1

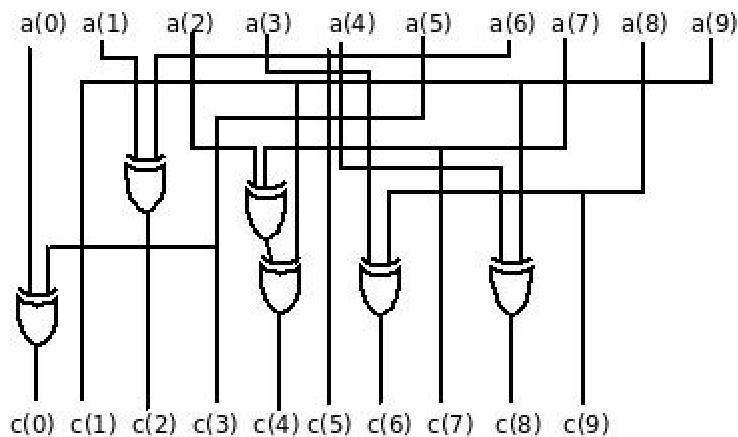


Figura 7: Arquitectura para calcular el cuadrado de un número cuando el polinomio irreducible es $f(x) = x^{10} + x^3 + 1$.

Para el trinomio escogido se obtiene la figura 7. El cuadrado de la entrada A se obtiene en un sólo paso mediante el empleo de cuatro puertas XOR.

3.3 Inversión

Sea $f(x)=x^m+f_{m-1}x^{m-1}+...+f_1x+1$ el polinomio irreducible sobre GF(2). Como se ha visto anteriormente, se puede representar un elemento de GF(2^m) definido por *f(x)* como $A(x)=a_{m-1}x^{m-1}+...+a_1x+a_0$ donde cada $a_i \in GF(2)$.

Emplearemos para este inversor la multiplicación en GF(2^m) polinómica módulo *f(x)*. En el cuerpo *GF(2^m)* se define el elemento inverso de *A* como A^{-1} si cumple que $A \cdot A^{-1} = 1$. Donde el · denota multiplicación en GF(2^m). El algoritmo que se ha usado se basa en el algoritmo de Euclides extendido [17].

El algoritmo de Euclides para polinomios calcula el polinomio máximo común divisor (MCD) de dos polinomios. Este algoritmo puede extenderse para calcular dos polinomios, $U(x),\ W(x)$ que cumplen

$$MCD(A(x),B(x))=U(x)\times A(x)+W(x)\times B(x) \tag{41}$$

El polinomio MCD de *A(x)* y *f(x)* es 1, por lo que se puede obtener el elemento inverso $A^{-1}(x)$ como *U(x) mod f(x)* reemplazando *f(x)* por *B(x)* en (41), tal y como se muestra a continuación

$$\begin{gathered} MCD(A(x),f(x))=U(x)\times A(x)+W(x)\times f(x) \\ 1\equiv U(x)\times A(x)\ \ (mod\ \ f(x)) \\ A^{-1}(x)\equiv U(x)\ \ (mod\ \ f(x)) \end{gathered} \tag{42}$$

A continuación está explicada la arquitectura basada en el algoritmo de Euclides extendido. Esta arquitectura comprueba los *m* coeficientes de dos polinomios para obtener el máximo común denominador. Para ello se ha empleado una estructura que emplea *2m* iteraciones para obtener el resultado.

En este algoritmo la notación *{Q1,Q2}* donde *Q1* y *Q2* son dos operaciones, significa que dichas operaciones se realizan en paralelo. La multiplicación por *x* significa un desplazamiento a la izquierda del polinomio, así como la división por *x* quiere decir un desplazamiento a la derecha. El algoritmo empleado es el siguiente:

```
S(x) := f(x); V(x) := 0;
R(x) := A(x); U(x) := 1;
δ := 0;
for i = 1 to 2m do
        if R(m) = 0 then
          R(x) := x·R(x);
          U(x) := x·U(x);
          δ =  δ + 1;
        else
          if S(m) = 1 then
            S(x) := S(x) – R(x);
            V(x) := V(x) – U(x);
          end if
          S(x) := x·S(x);
          if δ = 0 then
            {R(x) := S(x), S(x) := R(x)}
            {U(x) :=x·V(x), V(x) := U(x)}
            δ := 1;
          else
            U(x) := U/x;
            δ := δ – 1;
          end if
        end if
end for
```

La salida es $U(x)=A^{-1}$. El símbolo δ mantiene la diferencia entre el grado máximo de *R(x)* y el de *S(x)*.

4. Resultados experimentales

En esta sección se muestran los resultados obtenidos a partir de la descripción de las operaciones aritméticas del apartado anterior. Estos datos se han obtenido en un PC portátil ACER Aspire 1360 con procesador AMD Sempron 3000, para ello se han implementado las arquitecturas de las operaciones aritméticas mencionadas en VHDL, utilizando el software de diseño de hardware ISE Xilinx 10.1 WebPACK.

En las arquitecturas diseñadas se ha empleado el cuerpo $GF(2^m)$ con m = 8 generado por el polinomio irreducible $f(x)=x^8+x^4+x^3+x+1$, pues constituye un estándar del AES (Advanced Encryption Standard), también se emplea en los códigos Reed-Solomon para la codificación y es un estándar de la ESA (European Space Agency) y de la NASA (National Aeronautics and Space Administration).

También se han empleado para el diseño el cuerpo $GF(2^m)$ con m = 163 generado por el trinomio irreducible $f(x)=x^{163}+x^7+1$, el cuerpo $GF(2^m)$ con m = 193 generado por el polinomio irreducible $f(x)=x^{193}+x^{15}+1$, el cuerpo $GF(2^m)$ con m = 239 generado por el polinomio irreducible $f(x)=x^{239}+x^{36}+1$ y el cuerpo $GF(2^m)$ con m = 233 generado por el polinomio irreducible $f(x)=x^{233}+x^{74}+1$ ya que son cuerpos recomendados por el NIST (National Institute of Standards an Technology) [22,26] y el SECG [8] para aplicaciones en criptosistemas de curva elíptica. Los cuerpos $GF(2^m)$ con m = 16, 32, 64, 128 también se han usado para el diseño de multiplicadores.

En los multiplicadores, los resultados para el cuerpo $GF(2^8)$ se han simulado en la FPGA Spartan2 XCS-2S15. A su vez, los resultados para los otros cuerpos se han simulado en la FPGA Virtex XCS-300, ambas de Xilinx. En el cuadrado y en el inversor se ha utilizado la FPGA XCS-2S15. Se eligieron FPGAs diferentes debido a los distintos requerimientos de área. Todos los resultados se han verificado mediante el programa de cálculo matemático Maple 12.01.

En este estudio se han considerado los requerimientos de tiempo y área de cada diseño, esto se ha conseguido tomado medidas del retardo, la frecuencia máxima de operación y de área. Para las medidas de área ocupada en la FPGA se ha considerado el número de slices, LUTs, registros y puertas XOR empleados en cada implementación. La frecuencia máxima de operación se obtiene directamente de la herramienta de síntesis del software empleado. En los diseños estrictamente combinacionales – como son el multiplicador de Mastrovito, el multiplicador de Mastrovito para AOPs y el operador cuadrado – el retardo total lo proporciona también la herramienta de síntesis. Para los diseños con lógica secuencial – tales

como los multiplicadores serie LSB y MSB y el multiplicador dígito-serie – para calcular el retardo total del circuito, se ha utilizado el retardo proporcionado por la herramienta de síntesis, que corresponde a la parte combinacional, y se ha extrapolado dicho retardo al número total de ciclos que se ejecutan. El retardo así obtenido no se corresponde con el real, pero se considera una buena aproximación [20].

4.1 Complejidades espaciales de los multiplicadores para GF(28)

En este apartado se tratarán los resultados obtenidos de consumo de área de FPGA de las implementaciones de los multiplicadores para GF(28). Se ha tenido en cuenta la utilización de slices, LUTs y registros, también se ha considerado la cantidad de puertas XOR de cada implementación. Los datos se han obtenido de la herramienta de síntesis del software de Xilinx.

4.1.1 Utilización de slices y LUTs.

Los resultados obtenidos del uso de slices, LUTs y ocupación en la FPGA se muestran en la tabla 1, el porcentaje de ocupación en la FPGA se calcula respecto al número total de slices en la misma.

Como era de esperar, las arquitecturas de multiplicadores paralelo utilizan más cantidad de slices que las arquitecturas serie, con unos porcentajes de ocupación de 16% para el multiplicador de Mastrovito y de 7% para el multiplicador MSB entre otros. También se observa que el multiplicador dígito – serie utiliza una cantidad intermedia de slices, un 13% de ocupación para el multiplicador con tamaño de dígito 3.

Multiplicador	***Área (Slices)***	***% de ocupación en FPGA***
Mastrovito	32	16 %
MSB	14	7 %
LSB	20	10 %
Mastrovito AOP	34	17 %
Dígito Serie D=2	27	14 %
Dígito Serie D=3	26	13 %
Dígito Serie D=4	31	16 %

En cuanto al consumo de LUTs se cumple lo esperado para los multiplicadores paralelo y serie. Se observa que el mayor consumo de área es por parte de los multiplicadores paralelo,

principalmente por el multiplicador de Mastrovito para AOPs – y por tanto por el multiplicador de Mastrovito para base normal óptima tipo I – seguido por el multiplicador de Mastrovito para polinomios genéricos. Los multiplicadores serie que trabajan bit a bit son los que menor número de LUTs emplean.

El que no existan diferencias apreciables en consumo de slices por parte del multiplicador dígito – serie, se debe en parte a que el cuerpo escogido tiene un tamaño de palabra relativamente pequeño y por tanto los tamaños de dígito no varían lo suficiente como para afectar al número de slices utilizados. Sin embargo esta variación si es suficiente como para que se observen diferencias significativas en cuánto a utilización de puertas XOR, tal y como se muestra en la tabla 2. En estas tablas se ha omitido el multiplicador para base normal óptima tipo I pues sus datos son los mismos que para el multiplicador de Mastrovito para AOPs, ya que tal y como se mencionó en el apartado 3.1.6 de este trabajo su arquitectura es la misma que para el multiplicador de Mastrovito para AOPs pero cableando las entradas y las salidas para hacer la conversión de base.

A pesar de las similitudes entre los algoritmos LSB y MSB, sí se aprecia una variación significativa en el uso de slices. Esto se debe principalmente a que el número de registros empleado por cada arquitectura es diferente, tal y como se deduce de los datos de la tabla 2. También se puede deducir fácilmente de las figuras 3 y 4 correspondientes a la ruta de datos de estos multiplicadores, donde se observa una mayor utilización de registros por parte del multiplicador LSB y donde también se observa un uso similar de puertas XOR, cuya comprobación experimental queda reflejada en la tabla 2, tal y como se ha mencionado anteriormente .

Multiplicador	***Área (puertas XOR)***	***Área (Registros)***
Mastrovito	77	--
MSB	11	18
LSB	11	26
Mastrovito AOP	170	--
Dígito Serie D=2	25	27
Dígito Serie D=3	39	28
Dígito Serie D=4	53	29

Tabla 2: Utilización de registros y puertas XOR de los multiplicadores para $GF(2^8)$.

4.1.2 Utilización de Registros y puertas XOR

Los datos se han obtenido de la herramienta de síntesis del software de Xilinx y se muestran en la tabla 2. Se ha obviado el multiplicador de Mastrovito para base normal óptima tipo I, pues sus datos coinciden con el multiplicador de Mastrovito para AOPs. Se comprueba que los multiplicadores paralelo utilizan un número significativamente mayor de puertas XOR que el resto de multiplicadores, lo que es congruente con lo esperado al tratarse de arquitecturas de multiplicación en paralelo.

Como se ha mencionado anteriormente los multiplicadores MSB y LSB utilizan un número igual de puertas XOR, no así de registros, donde es mayor el uso de los mismos por parte del multiplicador LSB.

Para el multiplicador dígito serie existe muy poca variación en el número de registros empleados, ello se debe principalmente a que la parte secuencial de la ruta de datos no cambia al cambiar la longitud del dígito D, cambiando principalmente la parte combinacional del multiplicador. Como se observa en la tabla 2, al aumentar el tamaño de D aumenta el número de puertas XOR empleadas por el multiplicador, este hecho también se refleja en la figura 8. A partir de esto se puede obtener un compromiso entre el área utilizada del dispositivo y la velocidad de operación del mismo.

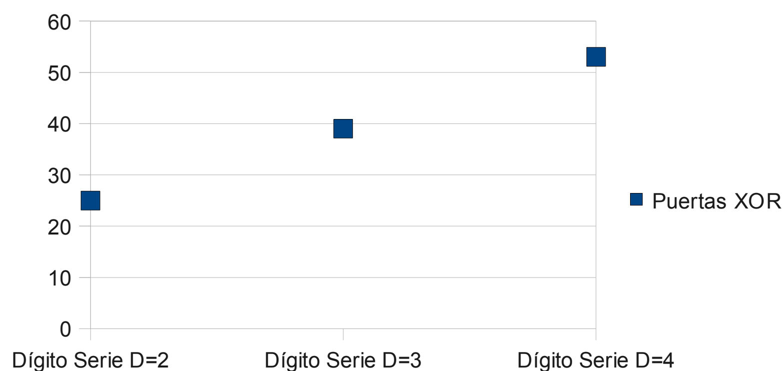
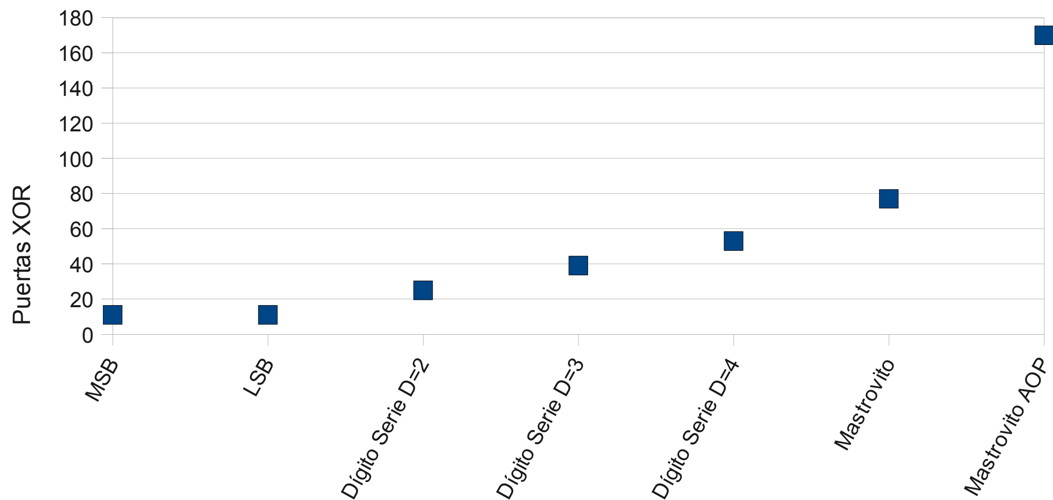


Figura 8: Variación del número de puertas XOR empleadas por el multiplicador Dígito - Serie en función del tamaño del dígito D

En la figura 9 se muestra una gráfica de tamaños comparativos de los diferentes multiplicadores en función del número de puertas XOR que emplea cada uno.



4.2 Complejidades temporales de los multiplicadores para GF(28)

En esta subsección se exponen los resultados correspondientes al rendimiento en velocidad de los dispositivos en la FPGA. Para ello se han tomado medidas de la frecuencia y el retardo de los multiplicadores.

4.2.1 Frecuencia de operación de los multiplicadores.

Los tiempos de la tabla 3 se han medido usando los resultados de síntesis de la FPGA proporcionados por el software de simulación de Xilinx. Para obtener el periodo y la frecuencia máxima de operación de los multiplicadores paralelo se ha variado su arquitectura incluyendo un registro de carga de datos al principio de los mismos. La frecuencia máxima se obtiene a partir del periodo de cada multiplicador.

Las mejores frecuencias de operación se obtienen para los multiplicadores LSB y MSB, y las peores para los multiplicadores de Mastrovito. Esto concuerda con el hecho de que la frecuencia de operación está relacionada con el retardo de la parte combinacional del multiplicador, a mayor retardo menor frecuencia de operación. Por supuesto, el tiempo de ejecución de cada algoritmo dependerá también del retardo total de cada multiplicador.

Multiplicador	*Área (LUTs)*	*% de ocupación en FPGA*
Mastrovito	62	16 %
MSB	27	7 %
LSB	38	9 %
Mastrovito AOP	68	17 %
Dígito Serie D=2	47	12 %
Dígito Serie D=3	50	13 %
Dígito Serie D=4	59	15 %

Los periodo de la tabla 3 están relacionados con el tiempo que se emplea en cada iteración en los algoritmos multiplicativos, así, el periodo de los que realizan la multiplicación en paralelo corresponde a una multiplicación completa, mientras que en los que realizan la multiplicación en serie el periodo corresponde a cada ciclo del algoritmo.

4.2.2 Retardo temporal de los multiplicadores

Los datos de retardo de los multiplicadores se muestran en la tabla 4. El retardo combinacional corresponde a la parte combinacional de los multiplicadores y se ha obtenido directamente de la herramienta de síntesis del software de Xilinx. El retardo total se ha obtenido de multiplicar el retardo combinacional por el número de ciclos necesarios para efectuar la multiplicación por cada multiplicador. El número de ciclos viene determinado por la arquitectura de cada diseño.

Para los multiplicadores que efectúan la multiplicación en paralelo, el número de ciclos necesario es uno, es decir, la multiplicación se ejecuta en un único paso. En los multiplicadores serie, puesto que la multiplicación se realiza bit a bit, el número de ciclos necesario es igual a m. En el multiplicador dígito serie, el número de ciclos necesario para obtener la multiplicación depende del tamaño de dígito D, de manera que se necesitan $d = \lceil m/D \rceil$ ciclos, siendo d el entero resultante, redondeando hacía arriba, de la división de m entre D.

Multiplicador	*Retado combinacional (ns)*	*Nº de ciclos*	*Retardo total (ns)*
Mastrovito	14,94	1	14,94
MSB	6,74	8	53,91
LSB	6,56	8	52,47
Mastrovito AOP	17,05	1	17,05
Dígito Serie D=2	7,29	4	29,16
Dígito Serie D=3	7,29	3	21,87
Dígito Serie D=4	7,38	2	14,76

Tabla 4: Retardos de los multiplicadores para GF(2^8)

El retardo combinacional de los multiplicadores paralelo es superior al de los multiplicadores serie, incluido el dígito – serie, ya que efectúan la multiplicación con todos los bits m de la palabra al mismo tiempo con lo que necesitan mayor cantidad de puertas lógicas, haciendo que el recorrido crítico – distancia entre las entradas y las salidas correspondientes en número de puertas – sea mayor. Por otra parte, el retardo total del multiplicador es bastante superior en los multiplicadores serie, especialmente en los que realizan la multiplicación bit a bit, como son el LSB y el MSB, pues aunque su recorrido crítico sea menor, tienen que efectuar una mayor cantidad de iteraciones para obtener el resultado. En el multiplicador dígito – serie, se puede observar que el retardo combinacional apenas cambia al variar el tamaño del dígito D, sin embargo, el número de ciclos determina el retardo total de dicho multiplicador. Por medio del multiplicador dígito – serie se puede lograr un mayor compromiso entre área y retardo que con los otros multiplicadores; por ejemplo, una buena relación entre número de slices empleados y retardo se conseguiría para D = 3 en este multiplicador, así dependiendo de los requerimientos de área y tiempo de nuestro sistema podríamos elegir el diseño adecuado.

Los retardos combinacionales se deben en parte a la lógica empleada y a la ruta de datos que implementa la herramienta de síntesis en la FPGA [31]. Los datos referentes a este hecho se pueden comprobar en la tabla 5. En los multiplicadores paralelo, el retardo se debe a la lógica y a la ruta prácticamente por igual, mientras que en los multiplicadores serie, la mayor parte del retardo se debe a la ruta. Esto puede explicarse al tener en cuenta la cantidad de puertas lógicas empleada por cada multiplicador, sin embargo hay que tener en cuenta que a mayor cantidad de puertas lógicas, mayor ha de ser la ruta empleada por la herramienta de síntesis en la FPGA, no obstante, la cantidad de puertas lógicas ha de ser sustancialmente

mayor como para apreciar este efecto, ya que las funciones se implementan en los slices, donde las rutas son relativamente pequeñas. Por esto, en el multiplicador dígito – serie no se aprecian apenas diferencias entre el retardo originado por la lógica y el retardo debido a la ruta. Para los multiplicadores de Mastrovito sí cambian significativamente los orígenes de los retardo respecto a los multiplicadores serie, debido a que la cantidad de puertas lógicas empleadas es bastante superior.

Multiplicador	***% debido a lógica***	***% debido a ruta***
Mastrovito	53	47
MSB	32,2	64,8
LSB	36,2	63,8
Mastrovito AOP	53,3	46,7
Dígito Serie D=2	34,6	65,4
Dígito Serie D=3	34,6	65,4
Dígito Serie D=4	34,1	65,9

Tabla 5: Origen de los retardos combinacionales para los multiplicadores en GF(2^8)

4.3 Complejidades espaciales y temporales de los multiplicadores para GF(2m)

A continuación se exponen las medidas de ocupación y tiempo de los multiplicadores para GF(2^m) con m = 16, 32,64,128,163, 193, 233 y 239. Los datos se han tomados de los multiplicadores serie LSB y MSB para todos los cuerpos, y del multiplicador de Mastrovito para polinomios generales para los cuerpos GF(2^{16}) y GF(2^{32}) .

Los resultados de este apartado se han obtenido, al igual que los resultados anteriores, de la herramienta de síntesis del software de Xilinx, y se muestran en la tabla 6. Se han tomado medidas de número de slices, LUTs, registros Flip-Flops, de frecuencia y de retardo. El número de ciclos es m para los multiplicadores LSB y MSB, y uno para el multiplicador de Mastrovito. El retardo se ha obtenido de multiplicar el número de ciclos por el retardo combinacional – que se ha omitido de la tabla para facilitar su lectura –.

Como era de esperar al aumentar el tamaño del cuerpo aumenta el área usada por los multiplicadores y el tiempo necesario para efectuar las operaciones. El retardo del multiplicador de Mastrovito aumenta al aumentar m, pues como se mencionó anteriormente, aumenta el recorrido crítico al incrementarse la complejidad de la multiplicación, sin embargo es mucho menor que para los multiplicadores serie. Los retardos de los multiplicadores LSB y MSB están directamente relacionados con el tamaño de m, variando desde 100 ns hasta

prácticamente 2 µs.

Multiplicador	***Cuerpo***	***Slices***	***LUTs***	***Flip-Flops***	***Frecuencia (MHz)***	***ciclos***	***Retardo (ns)***
Mastrovito	$GF(2^{16})$	118	234	-	-	1	16,700
LSB		40	67	64	168,66	16	94,86
MSB		29	47	44	168,66	16	94,86
Mastrovito	$GF(2^{32})$	454	905	-	-	1	20,92
LSB		65	113	117	146,43	32	218,53
MSB		45	77	79	144,43	32	220,53
LSB	$GF(2^{64})$	122	212	225	168,66	64	448,00
MSB		83	144	145	168,66	64	448,00
LSB	$GF(2^{128})$	235	409	433	138,24	128	925,95
MSB		159	227	295	138,24	128	925,95
LSB	$GF(2^{163})$	316	512	549	133,42	163	1221,68
MSB		200	349	370	133,42	163	1221,68
LSB	$GF(2^{193})$	343	607	647	133,26	193	1448,27
MSB		350	408	435	133,26	193	1448,27
LSB	$GF(2^{233})$	448	725	778	125,72	233	1746,33
MSB		300	522	490	125,72	233	1746,33
LSB	$GF(2^{239})$	422	749	796	130,91	239	1825,72
MSB		283	286	535	130,91	239	1825,72

El área usada por el multiplicador de Mastrovito es proporcional al tamaño de m, así para m = 16 es incluso mayor que el área usada por los otros multiplicadores para m = 239. Aunque las áreas utilizadas de los multiplicadores LSB y MSB son comparables, el multiplicador MSB utiliza menor cantidad de recursos, pues como se explicó en 4.1.1, la cantidad de registros utilizada por el multiplicador LSB es mayor que la utilizada por el MSB.

4.3.1 Comparación con otros autores.

En esta sección se exponen los resultados obtenidos por otros autores para la multiplicación para los cuerpos GF(2m) con m = 163, 193, 233 y 239 [20,10,5]. En la tabla 7 pueden verse los resultados obtenidos por otros autores junto con los resultados de este

estudio. Se ha escogido el multiplicador MSB para la comparación pues es el que menor número de slices utiliza.

Multiplicador	***Cuerpo***	***FPGA***	***Área (Slices)***	***Frecuencia máxima de operación (MHz)***
Referencia [F2]	$GF(2^{163})$	Virtex xcv-300	246	no aparece
Referencia [F2]	$GF(2^{193})$	Virtex xcv-300	290	no aparece
Referencia [F2]	$GF(2^{239})$	Virtex xcv-300	359	75
Referencia [R5]	$GF(2^{233})$	Virtex xc2v-6000	415	no aparece
Referencia [R5]	$GF(2^{239})$	Virtex xcv-300	385	no aparece
Referencia [R6]	$GF(2^{233})$	Virtex xc2v-6000	22403	93,20

Tabla 7: Resultados de área y frecuencia obtenidos por otros autores para los cuerpos $GF(2^m)$ con m = 163, 193, 233, 239

Para los cuerpos $GF(2^{163})$, $GF(2^{239})$ y $GF(2^{233})$ se obtienen un multiplicador MSB con menor consumo de slices que los obtenidos por los otros autores, hay que destacar que el multiplicador de C. Grabbe [5] opera en paralelo, por lo que es lógica la diferencia de consumo de área respecto al multiplicador MSB. El multiplicador de M.A.G-Martínez [20] opera de manera algo diferente al multiplicador MSB, pues aún siendo un multiplicador serie utiliza registros LFSR – Linear Feedback Shift Register –.También se comprueba que las frecuencias de operación del multiplicador de este estudio son mayores que las obtenidas por M.A.-Martínez y C.Grabbe, por lo que para arquitecturas serie el multiplicador MSB de este estudio tiene un menor retardo total.

4.4 Complejidades espaciales y temporales para la operación cuadrado

En esta sección se muestran los resultados obtenidos para la operación cuadrado de la sección 3.2. Al igual que para los multiplicadores, los resultados se han obtenido de la herramienta de síntesis del software de simulación de Xilinx. El algoritmo se ha implementado sobre una FPGA Spartan2 XCS-2S15.

Los datos de consumo de área y tiempo pueden verse en la tabla 8. Estos datos corresponden a la implementación de la figura 7. Se utilizan 6 puertas XOR, y el camino crítico es de 2 puertas, por lo que se obtiene un retardo pequeño. De las especificaciones de la herramienta de síntesis [31] para la ruta de datos se extrae que un 70% de ese retardo se debe a la lógica empleada y un 30% a la ruta de datos que implementa la herramienta de síntesis en la FPGA.

Slices	*LUTs*	*Xor*	*Retardo (ns)*
4	5	6	8,36

Para esta operación se ha escogido una arquitectura que realiza el cuadrado en paralelo, por lo que se obtiene el resultado en un sólo paso. Esta implementación es válida para el cuerpo GF(2^{10}) generado por el trinomio irreducible $f(x)=x^{10}+x^3+1$.

De acuerdo con [13] el número de puertas XOR requeridas para esta implementación es igual a *#XOR* = $\frac{m+k-1}{2}$ siendo *m=10* y *k=3*. Se cumple que *#XOR* = 6, coincidiendo con los resultados obtenidos para esta operación.

4.5 Complejidades espaciales y temporales para la inversión

Los datos referentes a las complejidades espaciales y temporales del inversor descrito en el apartado 3.3 pueden verse en la tabla 9. Estos resultados se han obtenido de la herramienta de síntesis del software de Xilinx y se han simulado sobre una FPGA Spartan2 XCS-2S15 .

Cuerpo	*Slices*	*LUTs*	*Flip-Flops*	*XOR*	*Retardo (ns)*	*Frecuencia máxima (MHz)*
GF(2^8)	43	43	86	184	17,07	57,84
GF(2^{16})	1909	3758	-	1003	268,41	3,70

Tabla 9: Datos de consumo de área y tiempo para la operación de inversión

El retardo se ve afectado fuertemente por el recorrido crítico de la implementación. De hecho se comprueba que un 48,4 % se debe a la lógica utilizada y un 51,6% a la ruta en el cuerpo GF(28). Estos porcentajes son típicos de arquitecturas que consumen bastante área, como ya se vio para los multiplicadores de Mastrovito, en los que el origen del retardo combinacional se debía aproximadamente a un 50% por la lógica y un 50% por la ruta. Como puede verse, el número de puertas XOR empleado es bastante grande, si lo comparamos con el resto de arquitecturas de este estudio; por lo que tiene sentido esa distribución de porcentajes en el retardo. También se puede comprobar, que al tener un mayor retardo opera a una menor frecuencia, pues como se ha mencionado anteriormente, esta depende en gran medida del retardo, que viene dado por el recorrido crítico que a su vez depende de los niveles de puertas existentes.

5. Conclusiones y trabajo futuro

En este trabajo se han estudiado diferentes implementaciones para las operaciones aritméticas de multiplicación, cuadrado e inversión sobre los cuerpos finitos $GF(2^m)$. Principalmente se ha tratado la operación de multiplicación pues es una de las más importantes en la aritmética sobre cuerpos finitos, por lo que su consumo de área y tiempo de ejecución son críticos. Por ello se han implementado multiplicadores serie y paralelo, pues ambos presentan pros y contras a la hora de integrarse en un circuito. Además se ha realizado un multiplicador con arquitectura dígito – serie por sus buenas características.

Estos algoritmos se han implementado en VHDL para ser integrados en una FPGA, ya que es una buena plataforma de simulación por su versatilidad, pues al ser una estructura programable por software permite integrar circuitos en ella sin modificarla físicamente.

Con las implementaciones desarrolladas se han obtenido medidas de ocupación de área en la FPGA y de retardo de cada algoritmo de multiplicación. Dichos resultados muestran que los multiplicadores que operan en paralelo, como son los multiplicadores de Mastrovito para polinomios generales, para AOPs y para base normal óptima tipo I, consumen mayor cantidad de área – slices, LUTs, registros Flips-Flops, etc – mientras que su retardo total es menor que el de los multiplicadores serie.

De entre los multiplicadores paralelo, el multiplicador de Mastrovito para polinomios generales es el que presenta menor ocupación en la FPGA, esto se debe a que al poder elegir el polinomio generador del cuerpo f(x) se pueden optimizar los recursos, cosa que no sucede en el multiplicador de Mastrovito para AOPs y para base normal.

Por otra parte los multiplicadores serie con arquitectura LSB y MSB presentan un consumo de área significativamente menor respecto a los anteriores, sin embargo su retardo total es mayor, al depender del tamaño del cuerpo m. El porcentaje de ocupación en la FPGA es menor en el multiplicador con arquitectura MSB que con arquitectura LSB, ya que en la implementación del primero se utiliza una menor cantidad de registros. Por otra parte ambos presentan retardos similares.

El multiplicador con arquitectura mixta dígito – serie presenta unas propiedades muy buenas en cuanto a porcentaje de ocupación en la FPGA y retardo total obtenido. En general su porcentaje de ocupación es similar, aunque ligeramente mayor, al de los multiplicadores serie LSB y MSB, pero sus retardos son bastante menores, llegando a ser comparables con los de los multiplicadores que operan en paralelo. Esto se debe a que al poder variar el tamaño del dígito se pueden conseguir diferentes proporciones área – retardo en función de los

requerimientos del sistema.

También se han obtenido medidas de ocupación y área para los algoritmos de inversión y cuadrado. La implementación para el cuadrado cumple con los requisitos que se esperaban de su desarrollo teórico en cuanto a ocupación y retardo. Igualmente, la implementación del inversor ha permitido comprobar la influencia de la ruta programada por el software de simulación en la FPGA en el cálculo del retardo.

A partir de este estudio se espera poder profundizar en las operaciones estudiadas, para ello se implementarán operaciones aritméticas en otras bases para estudiar sus características, tales como la base normal, dual, weakly-dual y otras. La base normal es especialmente interesante para las operaciones de cuadrado y exponenciación, pues sus propiedades permiten una implementación con un menor consumo de recursos. Junto con estas nuevas bases se estudiarán implementaciones para otros cuerpos $GF(2^m)$ generados por polinomios diferentes a los estudiados, también se pretende estudiar operaciones aritméticas en $GF(3^m)$r. Además se espera poder desarrollar mejores implementaciones para el multiplicador dígito – serie, debido a que, tal y como se menciono anteriormente, se pueden desarrollar arquitecturas óptimas en cuanto al consumo de recursos se refiere. También se desea poder realizar nuevas arquitecturas serie y dígito – serie para la operación de inversión para poder optimizar su uso de recursos.

Otro posible desarrollo futuro importante es la aplicación a la criptografía de las operaciones aritméticas realizadas. Para ello se piensan desarrollar arquitecturas ISE – Instruction Set Extensions – que son un conjunto de instrucciones integradas en el procesador y que permiten una ejecución más eficiente de las operaciones aritméticas, especialmente las operaciones criptográficas, siendo muy útiles entre otras cosas para la fabricación de etiquetas RFID y tarjetas inteligentes.

6. Bibliografía

[1] A. Halbutogullari, Ç.K. Koç. “*Mastrovito multiplier for general irreducible polynomials*”. IEEE Transactions on computers. Vol 49(5), pp. 503-518, 2000.

[2] A.J. Menezes (ed). “*Applications for finite fields*”. Kluwer Academic Publishers, 1993.

[3] Alex K. Jones, Raymond Hoare, Swapna Dontharaju , Shenchih Tung , Ralph Sprang, Joshua Fazekas, James T. Cain a, Marlin H. Mickle. “*An automated, FPGA-based reconfigurable, low-power RFID tag*”, Microprocessors and Microsystems. Vol 31, pp. 116–134, 2007 .

[4] Benjamin Arazi. “*Architectures for Exponentiation Over GF(2^m) Adopted for Smartcard Application*”, IEEE Transactions on Computers Vol 42(4), pp. 494-497, abril 1993.

[5] C. Grabbe, M. Bednara, J. Teich. “*FPGA desings of parallel high performance GF(2^{233}) multipliers*”. 0-7803-7762-1/03/$17,00 © 2003 IEEE.

[6] Chiou-Yng Lee, Erl-Huei Lu, Jau-Yien Lee. “*Bit-Parallel Systolic Multipliers for GF(2m) fields defined by All-One and Equally Spaced Polynomials*”. IEEE Transactions on Computers. Vol. 50(5), pp. 385-393. Mayo 2001

[7] Ç.K. Koç. “*Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields*”. IEEE Transactions on Computers. Vol. 47(3), pp. 353-356, Marzo 1998.

[8] Certicom Research, SEC 2: Recommended Elliptic Curve Domain Parameters, v1.0, 2000.

[9] C. S. Yeit, I.S. Reed, T.K. Truong. “*Systolic multipliers for finite fields GF(2^m)*”. IEEE Trans. Comput. Vol. C-33, pp 357-360, 1984.

[10] E. Ferrer, D.Bollman, O. Moreno. “*A Fast Finite field multiplier*”. P.C. Diniz et al. (Eds): ARC 2007, LNCS 4419, pp. 238-246, 2007.

[11] F.Rodriguez-Henríquez, N. A. Saqib, N. Cruz-Cortés. “*A fast implementation of multiplicative inversion over GF(2^m)*”. International Conference on Information Technology: Coding and Computing. 2009.

[12] G. M. de Dormale, J.J. Quisquater. “*Iterative modular division over GF(2^m): Novel algorithm and implementations on FPGA*”. K.Beretels, J.M.P. Cardoso, and S. Vassiliadis (Eds.). pp. 370-382. 2006

[13] Huapeng Wu. “*Bit Parallel finite field multiplier and squarer using polynomial basis*”. IEEE Transactions on Computers. Vol. 51(7), pp. 750-758. Julio 2002.

[14] Jorge Guajardo, Tim Güneysu, Sandeep S. Kumar, Christof Paar, Jan Pelzl. “*Efficient*

Hardware Implementation of Finite Fields with Applications to Cryptography", Acta Appl Math, 93: pp 75-118, 2006.

[15] J.L. Imaña, J.M. Sánchez, F. Tirado. "*Bit-Parallel Finite Field Multipliers for Irreducible Trinomials*", IEEE Transactions on Computers, 55(5): 520-533, mayo 2006.

[16] J.-P. Deschamps, J.L. Imaña, G.D. Sutter. "*Hardware Implementation of Finite-Field Arithmetic*", McGraw-Hill, 2009.

[17] K.Kobayashi, N. Takagi, K. Takagi. "*An algoritm for Inversion in GF(2^m) suitable for implementation using a polynomial multiply instruction on GF(2)*". IEEE Transactions on computers. Vol. 47(10), pp. 1161-1167, 1998.

[18] K.K. Parhi. "*A systematic approach for design of digit-serial signal processing architectures*". IEEE Trans. Circuits and Systems. Vol 38, pp. 358-375, 1991.

[19] Leilei Song, Keshab K. Parhi. "*Low-Energy Digit-Serial/Parallel Finite Field Multipliers*", Journal of VLSI Signal Processing 19, 149-166, 1998.

[20] M.A García-Martínez, R. Posada-Gómez, G. Morales-Luna, F. Rodríguez-Henríquez. "*FPGA implementation of an efficient multiplier over finite fields GF(2^m)*". International Conference on Reconfigurable Computing and FPGAs. 2005.

[21] P. Kitsos, G. Theodoridis, O. Koufopavlou. "*An efficient reconfigurable multiplier for Galois Field GF(2^m)*". Microelectronics Journal. Vol 34, pp 975-980. 2003.

[22] Recommended Elliptic Curves for Federal Government Use. http://csrc.nist.gov/

[23] R.I. Hartley, K.K Parhi. Digit-Serial Computation. Kluwer Academic Publishers. 1995.

[24] R. Lidl, H. Niederreiter. "*Finite Fields*". Addison-Wesley, Reading, Massachusets, 1983.

[25] T. Zang. "*Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible Polynomials*". IEEE Transactions on Computers. Vol. 50(7), pp. 734-749. Julio 2001.

[26] U.S. Department of Commerce/National Institute of Standards and Technology (NIST), Digital Signature Standard (DSS), FIPS PUB 182-2changel, 2000.

[27] Virtex 2.5V FPGA Detailed Functional Description.

http://www.xilinx.com/support/documentation/data_sheets/ds003-2.pdf

[28] Virtex 2.5V FPGA Introduction and Ordering Information.

http://www.xilinx.com/support/documentation/data_sheets/ds003-1.pdf

[29] W.Chelton, M. Benaissa. "*Design space exploration of division over GF(2^m) on FPGA: A*

digit-serial approach”. 1-4244-0395-2/06/$20.00 © 2006 IEEE.