UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS FÍSICAS

APLICACIÓN DE CAMPOS DE GALOIS A LA VERIFICACIÓN PROBABILISTA DE FUNCIONES BOOLEANAS Y MÉTODOS DE MULTIPLICACIÓN SOBRE CAMPOS DE EXTENSIÓN GF(2^m)

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

José Luis Imaña Pascual

Bajo la dirección del doctor

Juan Manuel Sánchez Pérez

Madrid, 2004

ISBN:978- 84-669-1742-1

Aplicación de campos de Galois a la verificación probabilista de funciones Booleanas y métodos de multiplicación sobre campos de extensión $GF(2^m)$

Memoria de Tesis Doctoral presentada por
José Luis Imaña Pascual
para optar al grado de Doctor en Ciencias Físicas

Dirigida por
Juan Manuel Sánchez Pérez

Madrid, marzo de 2003

A María
y
a mis padres

Agradecimientos

Deseo expresar mi agradecimiento al profesor Dr. D. Juan Manuel Sánchez Pérez por su dirección y asesoramiento, gracias a los cuales ha sido posible la finalización de este trabajo. También quiero agradecer su apoyo y amistad durante todo este tiempo.

Asimismo quiero expresar mi gratitud a todas aquellas personas que, de una forma u otra, han contribuido a que este trabajo se haya convertido finalmente en una realidad.

Índice general

Capítulo 1

Introducción

En este capítulo se presenta el marco en el cual se encuadra esta tesis doctoral, indicándose los objetivos que se han pretendido conseguir así como su organización en capítulos. Además, se presenta una visión general de los distintos métodos de verificación de circuitos lógicos, introduciendo la verificación probabilista y mostrando su relación con los campos de Galois. Se resalta la importancia de estos campos finitos para su utilización en numerosas aplicaciones técnicas.

En el ciclo de diseño de un circuito complejo, el paso de una especificación de alto nivel a una implementación física depende fundamentalmente de la correcta transformación de su descripción en alto nivel de abstracción a descripciones equivalentes en niveles más detallados. Cuanto más complejo sea el sistema que se diseña, más necesaria es la utilización de métodos automáticos que garanticen la corrección de dichas transformaciones. Por este motivo, el proceso de la verificación del diseño se está convirtiendo recientemente en un tema de gran interés para la industria. A nivel lógico, el proceso de verificación consiste en la comprobación de la equivalencia entre la especificación de una función Booleana y su correspondiente implementación lógica.

El método utilizado normalmente para la verificación de circuitos relativamente poco complejos ha sido la comprobación del comportamiento del diseño por medio de una *simulación* Booleana aleatoria. Este método consiste en la generación de un conjunto de patrones de entrada Booleanos que se utiliza para la comprobación (*test*) de la funcionalidad del circuito. Si el circuito satisface este test, entonces se tiene confianza sobre su correcto funcionamiento, aunque la verificación completa de la corrección del circuito no se puede realizar a menos que se puedan generar todos los posibles patrones de entrada del circuito, lo cual es imposible para circuitos grandes. Por lo tanto, este tipo de verificación no es satisfactorio porque para grandes circuitos sólo es posible comprobar

una parte de su funcionalidad y porque la generación de vectores de test de alta calidad (capaces de conmutar los nodos críticos del circuito) es difícil. Por este motivo se han desarrollado otros métodos de verificación que proporcionan una mayor confianza en el diseño. Uno de los objetivos principales de estos métodos consiste en la obtención de un nivel razonable de eficiencia, tanto en el tiempo necesario para la realización de la verificación como en la cantidad de memoria utilizada para la representación de los diseños a verificar.

Los procedimientos de verificación actuales más satisfactorios utilizan unas estructuras de datos conocidas como *Diagramas de decisión binarios ordenados y reducidos* (ROBDDs) para la representación canónica tanto de las especificaciones del circuito como de los diseños lógicos. Estos tipos de estructuras de datos han evolucionado con el tiempo hasta convertirse en una técnica de representación de funciones Booleanas muy satisfactoria. Los *Diagramas de decisión binarios* (BDDs) fueron propuestos inicialmente a finales de la década de los 50 por Lee [Lee59] y popularizados a finales de la década de los 70 por Akers [Ake78]. Por la misma época, Fortune, Hopcroft y Schmidt [FHS78] demostraron que si las variables están ordenadas, dos funciones Booleanas representadas por sendos grafos (ROBDDs) se pueden comparar para la determinación de su equivalencia en tiempo polinómico con el tamaño del grafo. Sin embargo, también demostraron que para ciertas funciones no existe un grafo ordenado cuyo tamaño sea polinómico con el número de entradas. Un año antes, Payne [Pay77] demostró que se podían obtener diagramas de decisión más simples y reducidos si se combinaban subgrafos ordenados idénticos. Todas estas investigaciones volvieron a ser retomadas por Bryant [Bry86] a mediados de los 80, quien no sólo demostró que los ROBDDs se podían manipular de manera eficiente por medio de algoritmos (*apply* y *compose*), sino que también mostró sus aplicaciones en el análisis de las propiedades de las funciones Booleanas. Desde entonces, los ROBDDs han sido objeto de un gran número de estudios.

Los métodos basados en ROBDDs producen resultados muy satisfactorios en muchos problemas de verificación de circuitos combinacionales [FFK88] [MWBSV88] y secuenciales [BCMD90], pero también presentan importantes desventajas. Para muchos circuitos, los sistemas de verificación que representan las funciones por medio de ROBDDs requieren una gran cantidad de tiempo y memoria. Para algunos circuitos, como los multiplicadores, las necesidades de recursos son prohibitivamente elevadas [Bry86][Bry91]. Más aún, los tamaños de los ROBDDs son muy sensibles a la ordenación de sus variables Booleanas [FFK88][MWBSV88]. Por estas razones, se deben buscar otras alternativas de verificación que traten de evitar este tipo de inconvenientes.

La *verificación probabilista*, basada en la teoría de Blum, Chandra y Wegman [BCW80], se fundamenta en la utilización de *transformadas algebraicas* de las funciones Booleanas, como las presentadas por Jain, Abraham, Bitner y Fussell [JABF92][Jai95][Jai96]. El objetivo consiste en la asociación de una

función con un código o *signatura* por medio de la creación de una representación algebraica de la función Booleana, la *asignación* de valores *enteros* a las variables de entrada y la *evaluación* posterior del resultado. Esta *signatura* se puede calcular de una forma más eficiente que una representación basada en grafos, consume menos tiempo y espacio y distingue cualquier par de funciones Booleanas con una probabilidad de éxito muy elevada. Si se realizan varias *ejecuciones* (asignación y evaluación) utilizando distintas asignaciones aleatorias de las variables de entrada, la *simulación algebraica* resultante presenta una probabilidad de error en la verificación que decrece exponencialmente desde un valor inicialmente pequeño.

La aproximación probabilista presenta importantes ventajas sobre los métodos deterministas que utilizan ROBDDs. En la verificación *determinista*, los ROBDDs constituyen representaciones canónicas de las funciones a verificar y proporcionan estructuras eficientes de cálculo. En la verificación *probabilista* la representación de la función no viene dada por una estructura de datos más o menos grande, sino por un código o *signatura*. Las estructuras de datos en forma de grafo se utilizan únicamente en las etapas intermedias de evaluación. Se pueden usar, por tanto, estructuras de datos basadas en BDDs como representaciones intermedias eficientes, pero cuando sea útil se pueden liberar las propiedades de canonicidad o de ordenación global de variables a estas estructuras, ya que lo que finalmente importa es la *signatura* asociada.

Como se ha mencionado, el cálculo de la *signatura* que representa a una función Booleana se realiza creando una representación algebraica de la función, asignando valores *enteros* a las variables de dicha expresión algebraica y evaluando finalmente el resultado. Los valores que se asignan a las variables de entrada de la función se seleccionan aleatoriamente de cualquier *campo*, que se representa por $\mathcal{F}$. Computacionalmente, interesa la utilización de *campos finitos*, como por ejemplo el campo de enteros *módulo* p con p primo, $\mathcal{Z}_p$ (por este motivo hemos indicado que se realiza una asignación de valores *enteros* a las variables). Este campo, $\mathcal{Z}_p$, será el que se utilizará en la presentación de los conceptos básicos dados en el capítulo 2, y es el campo utilizado por Jain et al. para el desarrollo de su teoría de verificación probabilista. Sin embargo, nosotros utilizaremos para el cálculo de signaturas los *campos de extensión* de $GF(q)$, representados como $GF(q^m)$ y conocidos como *campos de Galois* [Ste89][LN83] en honor del trabajo fundamental realizado por Evariste Galois en este tema. Estos tipos de campos finitos se introducirán en el capítulo 7.

Para aplicaciones técnicas, los campos de extensión de $GF(2)$, representados por $GF(2^m)$, son de un interés fundamental. Seleccionando una cierta *base*, se tiene que los elementos del campo $GF(2^m)$ se representan por medio de una serie de polinomios, que a su vez se representan como vectores binarios. Otra propiedad importante consiste en que la aritmética en los campos de *característica* 2 es esencialmente una aritmética *modular*, con lo que la suma

(y resta) se convierten en operaciones relativamente sencillas. Sin embargo, la multiplicación es la operación aritmética más importante y una de la más complejas, tanto computacionalmente como de implementación física. La complejidad de la multiplicación depende de varios factores, como son la selección del *polinomio irreducible* generador del campo o de la *base* utilizada para la representación de los elementos del campo.

La idea de la aplicación de los campos de Galois $GF(2^m)$ a la verificación probabilista surge de las ventajas computacionales que se obtienen debido a sus propiedades, ya que se pueden efectuar determinadas simplificaciones sobre las expresiones algebraicas obtenidas previas a su evaluación sobre un conjunto de elementos del campo $GF(2^m)$.

Las aplicaciones de los campos de Galois $GF(2^m)$ en otras áreas como la *criptografía* [Sch93][MOV97], los *códigos algebraicos* [Bla83][ML85], el *procesamiento digital de señal* [Bla85] o la *generación aleatoria de números* [WP90] se han ido incrementando ampliamente en los últimos años. Por ejemplo, los sistemas de comunicación modernos como las redes de computadores, los enlaces de satélites o los *compact disks*, utilizan aritmética sobre campos finitos $GF(2^m)$ para la corrección de errores o para algoritmos criptográficos. Estas y otras aplicaciones deben ser rápidas, por lo que en muchos casos se utilizarán implementaciones sobre circuitos VLSI (*Very Large Scale Integration*) de las operaciones aritméticas con el propósito de satisfacer estas necesidades de funcionamiento a velocidades muy elevadas [YRT84][WTS$^+$85][FBT96]. Recientemente se están comenzado a utilizar plataformas *reconfigurables* [Kli95][PR97] para la implementación, como las FPGAs (*Field Programmable Gate Array*) o CPLDs (*Complex Programmable Logic Device*) que tienen como características una gran flexibilidad y rapidez de programación.

La necesidad de funcionamiento a frecuencias elevadas de este tipo de aplicaciones y el hecho de que la operación aritmética más costosa sea la multiplicación, hacen que la búsqueda de métodos de multiplicación y de arquitecturas eficientes de módulos *multiplicadores* operando sobre $GF(2^m)$ sea de especial importancia. Además, la mayor parte de las operaciones aritméticas avanzadas, como la *exponenciación* o la *inversión*, se basan en la multiplicación. A su vez, la complejidad de un módulo multiplicador depende de factores como la selección de la *base* de representación o del *polinomio irreducible* generador del campo, como se ha mencionado anteriormente. Con respecto al tipo de arquitectura, los módulos aritméticos sobre campos de Galois se pueden clasificar en módulos de arquitectura *serie* y *paralela*. La aritmética *paralela* se implementa utilizando únicamente lógica combinacional, mientras que la aritmética *serie* requiere una implementación secuencial. Normalmente existe una contraposición de *espacio-tiempo* entre ambos tipos de aritmética. Mientras que las implementaciones *paralelas* tienden a ser más rápidas, las implementaciones *serie* necesitan normalmente un área menor que sus homólogas *paralelas*.

1.1. Objetivos

A la vista de lo expuesto anteriormente, en esta tesis se han establecido los siguientes objetivos generales:

- Estudio de los conceptos básicos de la verificación *probabilista* así como de la utilización de los campos de Galois $GF(2^m)$ en este tipo de verificación como método alternativo a la comprobación de equivalencia determinista a través de los ROBDDs.

- Desarrollo de una nueva formulación para el cálculo de operaciones sobre funciones Booleanas representadas en formato de *dos niveles* que proporcione los resultados como una *suma de productos disjuntos*. Las funciones Booleanas representadas en forma de *dos niveles* como *suma de productos* (o *cubos*) presentan una serie de ventajas, como son su generalidad y su facilidad de implementación computacional, lo que las hacen ser ampliamente estudiadas y utilizadas en numerosas aplicaciones. Por otra parte, la representación de funciones Booleanas en forma de *cubos disjuntos* es muy importante en la verificación probabilista sobre $GF(2^m)$ de circuitos de dos niveles, ya que este tipo de verificación se simplifica mucho cuando las funciones Booleanas con las que se opera son *disjuntas*.

- Revisión de la utilización de ROBDDs *clásicos* en la verificación *determinista* de funciones Booleanas y desarrollo de un algoritmo *heurístico* de ordenación de variables que permita la obtención de tamaños reducidos para este tipo de grafos. Este es uno de los problemas principales que presentan estas estructuras de datos, ya que la ordenación de sus variables de entrada influye enormemente en su tamaño y ésto afecta directamente a su eficiencia para la verificación.

- Desarrollo de una aproximación *híbrida* de verificación de circuitos combinacionales de dos niveles que combine la verificación *determinista* de los ROBDDs con la *probabilista* aplicada sobre campos $GF(2^m)$, utilizando para ello la *formulación cúbica disjunta* desarrollada previamente. Mientras no existan limitaciones de espacio o tiempo en verificación, el método híbrido se debe comportar de forma *determinista*, pero si existe alguna restricción entonces la aproximación a utilizar debe ser la *probabilista*.

- Aplicación de la verificación *probabilista* sobre $GF(2^m)$ a circuitos representados en formato *multinivel*. Para ello se utilizan las estructuras de datos en forma de grafo conocidas como $\oplus$-OBDDs. La importancia de las representaciones de circuitos *multinivel* frente a las de *dos niveles* es debida a la creciente complejidad de los circuitos a implementar y al desarrollo de nuevas tecnologías, lo que hace necesaria la utilización de formatos más flexibles y adaptables a diferentes tipos de implementación.

- Profundización en el conocimiento de los campos de Galois, principalmente los campos de extensión de $GF(2)$; estudio de los tres tipos de multiplicación (*canónica*, *normal* y *dual*) sobre $GF(2^m)$ más utilizados y comparación de las complejidades *teóricas* de los módulos multiplicadores *paralelos* correspondientes, así como de las *experimentales* obtenidas de la implementación de dichos multiplicadores sobre *hardware reconfigurable* para distintos polinomios generadores del campo. Los campos $GF(2^m)$ presentan grandes ventajas en la verificación probabilista y tienen muchas aplicaciones en áreas tan importantes como la criptografía, el procesamiento digital de señal o la generación aleatoria de números. Asimismo, la multiplicación sobre $GF(2^m)$ es la operación más costosa y sobre la que se basan la mayor parte de las operaciones aritméticas avanzadas, como la *exponenciación* o la *inversión*, de ahí la importancia que presenta su estudio.

- Desarrollo de un nuevo método de multiplicación en base *canónica* que trate de obtener reducciones de complejidades tanto *teóricas* como *experimentales* (sobre hardware reconfigurable) de los módulos multiplicadores paralelos correspondientes, aplicación del nuevo método de multiplicación sobre distintos polinomios irreducibles generadores de $GF(2^m)$ y estudio de las complejidades *teóricas* y *experimentales* correspondientes.

1.2. Organización

Tomando como eje central los campos de Galois $GF(2^m)$, la presente tesis se ha dividido en dos partes diferenciadas y relacionadas a través de dichos campos: una primera parte dedicada a la *Verificación* y que comprende los capítulos del 2 al 6, y una segunda parte centrada en los *Campos de Galois* que se desarrolla en los capítulos 7, 8 y 9. A continuación presentamos brevemente los contenidos de los distintos capítulos.

La primera parte de la tesis dedicada a la *Verificación* se inicia con el capítulo 2, en el que se introducen los conceptos teóricos básicos de la verificación *probabilista* por medio de la utilización de transformaciones algebraicas de funciones Booleanas. Algunas de las propiedades y teoremas presentados son de gran utilidad en el desarrollo de los capítulos posteriores y en la justificación de la utilización de los campos de Galois como campos finitos. En este capítulo también se pone de manifiesto la importancia de la propiedad de *disjunción* en la verificación probabilista para la simplificación del cálculo de las signaturas de las funciones Booleanas transformadas.

En el capítulo 3 se presenta una nueva formulación para el cálculo de operaciones realizadas sobre funciones Booleanas representadas en forma *cúbica* (sumas de productos) y cuyos resultados se obtienen también como conjuntos

de cubos que verifican la propiedad de *disjunción*. Este método de cálculo se ilustra a través de un ejemplo y se presentan los resultados experimentales obtenidos de su implementación.

En el capítulo 4 se describen los *diagramas de decisión binarios ordenados y reducidos* (ROBDDs) empleados en la verificación *determinista*, así como otros tipos de estructuras de datos en forma de grafos también usados para la representación de funciones Booleanas. Se introduce el problema de la ordenación de variables de entrada con respecto del tamaño del grafo, haciéndose una revisión de las distintas alternativas existentes para su resolución y se presenta un nuevo algoritmo de ordenación *heurístico* de variables junto con los resultados experimentales obtenidos.

Una aproximación *híbrida* de verificación que combina los métodos determinista y probabilista se describe en el capítulo 5. Esta nueva aproximación se aplica a la verificación de circuitos combinacionales representados en forma de *dos niveles*, por lo que se hace uso de la formulación vista en el capítulo 3 para la obtención de una representación en forma de cubos disjuntos. En esta aproximación *híbrida*, los ROBDDs se utilizan como estructuras de datos intermedias, por lo que se presentan resultados experimentales para los casos de *utilización* y de *no utilización* de estrategias de *reordenación dinámica de variables* para la reducción de sus tamaños. En este método híbrido las signaturas y operaciones aritméticas sobre las mismas se realizan utilizando un campo de Galois $GF(2^m)$.

La verificación probabilista aplicada a la lógica *multinivel* se presenta en el capítulo 6, en el que se utilizan los grafos conocidos como $\oplus$-OBDDs como estructuras de datos de representación de funciones Booleanas que permiten la manipulación y almacenamiento de las signaturas pertenecientes a un campo $GF(2^m)$. Se presenta un nuevo método de construcción de $\oplus$-OBDDs que trata de obtener tamaños reducidos para los mismos y se dan resultados experimentales, realizándose una comparación de estos resultados con los obtenidos por otros métodos similares encontrados en la literatura.

La segunda parte de la tesis dedicada a los *Campos de Galois* se inicia con el capítulo 7, en el que se presentan los conceptos teóricos básicos de los campos finitos y de los campos de extensión, así como las diferentes bases de representación de los elementos del campo finito. Se estudian los importantes campos de extensión $GF(2^m)$ y los tipos de multiplicación más importantes sobre estos campos (*canónica*, *normal* y *dual*), centrándonos en la arquitectura paralela de los módulos multiplicadores correspondientes y determinando sus complejidades teóricas *espaciales* y *temporales*. También se presentan resultados experimentales correspondientes a la implementación *reconfigurable* paralela sobre FPGAs de los tres tipos de multiplicadores estudiados y para distintos polinomios generadores del campo, realizándose la comparación de los resultados obtenidos.

En el capítulo 8 se presenta un nuevo método de multiplicación en base canónica sobre campos de Galois $GF(2^m)$ al que se le ha denominado método *transposicional*. Este método se extrae a partir de una nueva formulación para la multiplicación en base polinómica, basada en la utilización de una base *triangular* de representación y que depende del polinomio irreducible generador del campo de Galois seleccionado. La introducción del nuevo método *transposicional* se ha realizado considerando campos de Galois generados por un tipo especial de polinomios conocidos como AOPs (*all-one-polynomials*), en cuyo caso, el método también es aplicable a la multiplicación en base normal. Se ha comprobado que la descripción del multiplicador proporcionada por el método *transposicional* conlleva una reducción de complejidad *experimental* cuando se utiliza hardware reconfigurable para la implementación, mientras que la complejidad *teórica* obtenida iguala, para este tipo de polinomios, los mejores resultados encontrados en la literatura. Para ello, se ha realizado el análisis de complejidad teórico de los multiplicadores construidos utilizando nuestro método, comparándose las complejidades obtenidas con las proporcionadas por otras aproximaciones similares y se han realizado implementaciones reconfigurables sobre FPGAs y CPLDs de los multiplicadores paralelos.

En el capítulo 9, el nuevo método *transposicional* se aplica a la multiplicación en base canónica sobre campos $GF(2^m)$ generados por determinados tipos de *trinomios* irreducibles. En este caso, la utilización de nuestro método para la construcción de los multiplicadores produce no sólo una reducción de complejidad *experimental* cuando se utilizan dispositivos reconfigurables para su implementación, sino también *teórica*, en comparación con los mejores resultados encontrados en la literatura. Esta reducción de la complejidad teórica de los multiplicadores producida por nuestro método se demuestra realizando, para todos los trinomios considerados, el análisis de complejidad teórico tanto *temporal* como *espacial* (mostrado este último en el apéndice A).

Por último, en el capítulo 10 se presentan las principales conclusiones y aportaciones de esta tesis, así como las futuras líneas de investigación.

Parte I

Verificación

Capítulo 2

Verificación probabilista

En este capítulo se introducen los conceptos teóricos básicos de la verificación probabilista por medio de la utilización de transformaciones algebraicas de funciones Booleanas.

El análisis de *funciones Booleanas* o *de conmutación*[1] es muy importante en numerosos campos como la combinatoria, la inteligencia artificial, el diseño CAD (*Computer Aided Design*) de circuitos digitales o en la matemática aplicada. Por ejemplo, la resolución eficiente de cuestiones como la *tautología, equivalencia*, etc. de las funciones Booleanas tiene un gran número de aplicaciones en varios problemas de diseño automático de sistemas como la síntesis, verificación, *test*, etc.

La funciones Booleanas se suelen representar por medio de expresiones o fórmulas en las que las variables toman valores binarios **0** (*falso*) o **1** (*cierto*) y en las que las operaciones permitidas entre dos variables son las operaciones binarias AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), etc. En muchas ocasiones, el análisis de una expresión Booleana se puede mejorar enormemente si se *transforma* dicha expresión en otra en la cual se puedan utilizar las operaciones aritméticas habituales como la suma, la resta o la multiplicación. Lo que se obtendría sería una *expresión aritmética* que se podría evaluar asignando, por ejemplo, valores enteros (en lugar de valores binarios) a sus variables. Esta evaluación de funciones Booleanas *transformadas* puede ser muy útil para muchos de los problemas mencionados anteriormente.

[1] No se hará distinción entre *funciones Booleanas* y *funciones de conmutación*, y se utilizará el término habitual de *funciones Booleanas*. Toda función de conmutación de n variables definida por la correspondencia $\{0,1\}^n \rightarrow \{0,1\}$ también es una función Booleana. El álgebra Booleana de funciones de conmutación se llama álgebra de conmutación. Sin embargo, en un álgebra Booleana formada por la quíntupla $\{\mathbf{B}, \mathbf{0}, \mathbf{1}, +, \cdot\}$, el conjunto *portador* $\mathbf{B}$ puede tener más de dos elementos. En [Bro90] se puede encontrar una clara diferenciación entre las funciones Booleanas y las funciones de conmutación.

Considerando el problema de la verificación, supóngase que se desea comprobar si la función Booleana $f_1 = (x_1 \wedge \overline{x}_3) \vee (x_1 \wedge x_2)$ es equivalente a la función $f_2 = x_2 \wedge (x_1 \vee (x_2 \wedge \overline{x}_3))$. Estas dos funciones no son equivalentes, lo cual se puede comprobar realizando una simulación aleatoria o construyendo sus representaciones canónicas como los ROBDDs[2].

En la simulación Booleana aleatoria, si el primer vector que se genera es $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, tendríamos que ambas funciones f_1 y f_2 se evaluarían al valor 1. Si a continuación se utilizan $x_1 = 0$, $x_2 = 0$, $x_3 = 1$ y $x_1 = 0$, $x_2 = 1$, $x_3 = 1$ como vectores de test, tendríamos que las dos funciones se evaluarían también al mismo valor 0. Podría ser necesaria la generación de un número exponencial de vectores antes de poder demostrar la diferencia entre las dos funciones (para dos funciones equivalentes de n variables, se tendrían que generar y probar 2^n vectores para poder determinar su equivalencia). Esta alternativa sería claramente ineficiente para un valor de n grande. En general, la simulación Booleana aleatoria constituye un método adecuado cuando las diferencias entre las funciones a verificar son muchas. Sin embargo, para funciones equivalentes o para funciones con pocas diferencias, es inadecuado.

De igual forma, la verificación por medio de la utilización de ROBDDs es muy costosa y puede ser necesaria la utilización de grafos de tamaño exponencial para la representación de muchas funciones [Bry86][Bry91]. Al ser representaciones canónicas, cuando los ROBDDs requieren un tiempo exponencial para su construcción, también requieren un espacio de memoria exponencial.

En cambio, considérese la siguiente aproximación. Como veremos posteriormente, se pueden crear unas transformaciones aritméticas de las funciones f_1 y f_2, que representamos como $\mathcal{A}[f_1]$ y $\mathcal{A}[f_2]$, y que vienen dadas por las expresiones $\mathcal{A}[f_1] = x_1 - x_1 \cdot x_3 + x_1 \cdot x_2 \cdot x_3$ y $\mathcal{A}[f_2] = x_2 - x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3$. La ventaja que presentan estas transformaciones aritméticas es que $\mathcal{A}[f_1]$ y $\mathcal{A}[f_2]$ se pueden evaluar también sobre valores no Booleanos. Por ejemplo, si se selecciona de manera aleatoria la asignación $x_1 = 1$, $x_2 = 3$, $x_3 = 2$, entonces las transformaciones $\mathcal{A}[f_1](2,3,1)$ y $\mathcal{A}[f_2](2,3,1)$ se evalúan a los valores 5 y 3, respectivamente. Como estos dos valores numéricos (*signaturas*) son distintos, se puede concluir que $f_1 \not\equiv f_2$. Estas simulaciones numéricas pueden distinguir cualquier par de funciones Booleanas con una alta probabilidad de éxito. Cuanto mayor sea el conjunto del que se seleccionan aleatoriamente las asignaciones numéricas, menor será la probabilidad de error de que se evalúen a la misma signatura dos funciones diferentes. Si se efectúan varias de estas *ejecuciones* con distintas asignaciones aleatorias de las variables de entrada, la *simulación algebraica* resultante presenta una *probabilidad de error* en verificación que decrece exponencialmente desde un valor inicialmente pequeño. A esta aproximación se le conoce como *verificación probabilista* y fue enunciada inicialmente por Blum, Chandra y Wegman [BCW80].

[2]Los *Diagramas de decisión binarios ordenados reducidos* (ROBDDs) se verán en el capítulo 4.

Volvamos a considerar la función anterior $f_1 = g_1 \vee g_2$, donde $g_1 = (x_1 \wedge \overline{x}_3)$ y $g_2 = (x_1 \wedge x_2)$, y supongamos que utilizamos las expresiones de probabilidad[3] $\mathrm{AND}(x, y) = x \cdot y$, $\mathrm{NOT}(x) = 1 - x$ y $\mathrm{OR}(x, y) = x + y - x \cdot y$. Si evaluamos f_1 sobre la asignación aleatoria $x_1 = 1$, $x_2 = 3$, $x_3 = 2$, y propagamos simplemente los valores *numéricos* a cada puerta, tendríamos que los valores obtenidos (utilizando las expresiones de probabilidad anteriores) para g_1, g_2 y f_1 serían $g_1 = 1 \cdot (1 - 2) = -1$, $g_2 = 1 \cdot 2 = 2$ y $f_1 = (-1) + 2 - (-1) \cdot 2 = 3$, que es un valor incorrecto. El motivo, como se verá en las siguientes secciones, es que entre las entradas g_1 y g_2 a la función f_1, no se ha tenido en cuenta la ley de *idempotencia de la multiplicación* en la variable x_1, es decir, $x_1 \wedge x_1 = x_1$. Si no se tuviera en cuenta esta importante ley de idempotencia, se podrían realizar simulaciones numéricas que producirían signaturas *distintas* para funciones que fueran *equivalentes*, con lo que se obtendrían unos resultados erróneos en el proceso de verificación.

A continuación vemos los conceptos fundamentales de la verificación probabilista de funciones Booleanas, propuesta por Jain et al. [JABF92][Jai95] [Jai96] y que se basa en la utilización de *transformadas* algebraicas de las funciones Booleanas. Los teoremas que se presentan se dan sin demostración, pudiéndose encontrar en las referencias anteriores.

Transformaciones similares han sido estudiadas por otros autores, como por ejemplo Parker y McCluskey [PM75] y Kumar y Breuer [KB81]. En este último trabajo, se utilizan técnicas *espectrales* para la verificación probabilista de funciones Booleanas, en las que para comparar dos funciones se puede seleccionar, por ejemplo, un subconjunto de su *espectro*. Las técnicas espectrales se encuentran fuera del estudio de esta tesis, pero se pueden consultar, por ejemplo, en el libro de Hurst, Miller y Muzio [HMM85].

2.1. Transformación algebraica de funciones Booleanas

La *signatura* de una función Booleana $\beta(x_1, \ldots, x_n)$ de variables de entrada Booleanas x_1 a x_n se genera seleccionando un valor entero aleatorio para cada x_i y evaluando una versión transformada de β utilizando estos valores. De manera más general, los valores se pueden seleccionar de cualquier *campo*, que se representará por $\mathcal{F}$. La utilización de *campos finitos*[4] es deseable desde un punto de vista computacional, por lo que se utilizará $\mathcal{Z}_p$, el campo de enteros *módulo* p para algún primo p, como campo de selección de los valores numéricos aleatorios de las variables.

[3] Estas expresiones fueron analizadas inicialmente por Boole [Boo54].

[4] En esta tesis utilizamos como campos finitos los *campos de Galois* $GF(2^m)$, que son *campos de extensión* del campo binario $GF(2)$ y que se estudiarán en el capítulo 7. En el desarrollo teórico del presente capítulo se utiliza $\mathcal{Z}_p$ como campo finito, lo que ayuda a la claridad y compresión de los conceptos de la transformación algebraica.

Para poder transformar la función Booleana β en una *función de campo* equivalente definida sobre $\mathcal{F}$, se define una *transformación funcional*, $\mathcal{A}[\beta]$, que transforma β en un polinomio de campo (se escribirá $\mathcal{A}[\beta](x_1, \ldots, x_n)$ para incluir explícitamente las n variables). La evaluación de $\mathcal{A}[\beta]$ con los valores enteros aleatoriamente seleccionados de $\mathcal{Z}_p$ proporciona la *signatura* correspondiente a la función β. La transformación $\mathcal{A}[\beta]$ constituye una transformación *canónica*, es decir, si dos funciones Booleanas β_1 y β_2 son equivalentes, entonces sus polinomios de campo correspondientes $\mathcal{A}[\beta_1]$ y $\mathcal{A}[\beta_2]$ son idénticos.

2.1.1. *Transformada* $\mathcal{A}$ y propiedades fundamentales

Para poder definir la *transformada* $\mathcal{A}$, se necesita asociar en primer lugar un *polinomio clave* con cada una de las 2^n asignaciones de entrada a una función Booleana $\beta(x_1, \ldots, x_n)$. Se deben sumar posteriormente los polinomios asociados con aquellas asignaciones que producen el valor de salida *cierto* e interpretar el resultado como una función entera para obtener la transformación algebraica.

El *polinomio clave* $\mathcal{W}_n$ para una entrada (fila) de la tabla de verdad es un producto de términos $\prod_{i=1}^{n} w(b_i, x_i)$, donde cada término $w(b_i, x_i)$ está asociado con una variable de entrada particular x_i. Si b_i representa el valor asignado a x_i en una fila dada de la tabla de verdad, entonces $w(b_i, x_i)$ se define como $w(b_i, x_i) = b_i x_i + (1 - b_i)(1 - x_i)$, por lo que $w(b_i, x_i)$ es x_i si $b_i = 1$, y $(1 - x_i)$ si $b_i = 0$. Por tanto, b_i actua como un selector.

Definición 1 *Para cualquier $n \geq 0$, se define $\mathcal{W}_n : \mathcal{F}^{2n} \longrightarrow \mathcal{F}$ como*

$$\mathcal{W}_n(b_1, \ldots, b_n, x_1, \ldots, x_n) \;=\; \prod_{i=1}^{n} w(b_i, x_i) \tag{2.1}$$

Por ejemplo, $\mathcal{W}_4(0, 1, 1, 0, x_1, x_2, x_3, x_4) = (1 - x_1)x_2x_3(1 - x_4)$. Por comodidad, se escribirá $\mathcal{W}_n(b_1, \ldots, b_n, x_1, \ldots, x_n)$ como $\mathcal{W}_n(\underline{b}, \underline{x})$, donde $\underline{b}$ y $\underline{x}$ son vectores de longitud n y se utilizará $f(\underline{b})$ en lugar de $f(b_1, \ldots, b_n)$.

Se puede definir ahora la *transformada* $\mathcal{A}$ de la función f, $\mathcal{A}[f]$, como la suma de estos polinomios clave, cada uno multiplicado por el resultado Booleano de su correspondiente asignación. Por tanto, únicamente se sumarán aquellos polinomios que se correspondan con asignaciones que producen el valor de salida *cierto*.

Definición 2 *Dada una función $f : \{0,1\}^n \longrightarrow \mathcal{F}$, se define el polinomio $\mathcal{A}[f] : \mathcal{F}^n \longrightarrow \mathcal{F}$ como*

$$\mathcal{A}[f](\underline{x}) \;=\; \sum_{\forall \underline{b} \in \{0,1\}^n} f(\underline{b}) \cdot \mathcal{W}(\underline{b}, \underline{x}) \tag{2.2}$$

Por ejemplo, si la función β de dos variables x_1 y x_2 es de la forma $\beta(x_1, x_2) = x_1 \oplus x_2$, según lo visto anteriormente se tendrán los siguientes polinomios clave:

x_1	x_2	β	$\mathcal{W}_2$
0	0	0	$(1-x_1)\cdot(1-x_2)$
0	1	1	$(1-x_1)\cdot x_2$
1	0	1	$x_1\cdot(1-x_2)$
1	1	0	$x_1\cdot x_2$

Tabla 2.1: Polinomios clave de la función $\beta = x_1 \oplus x_2$

Como la función β toma valor 1 para la segunda y tercera filas de su tabla de verdad, se deben sumar sus correspondientes polinomios clave para obtener finalmente $\mathcal{A}[\beta] = (1-x_1)x_2 + x_1(1-x_2) = x_1 + x_2 - 2x_1x_2$.

Se puede observar que la ecuación 2.2 se puede aplicar a cualquier función de campo y no sólo a funciones Booleanas, por lo que se puede realizar la transformada algebraica de una función de campo de n variables.

2.1.2. Equivalencias entre *transformadas* $\mathcal{A}$

Una función de campo f se define para entradas $x_i \notin \{0,1\}$, pero en la definición de la *transformada* $\mathcal{A}$ se han considerado únicamente los valores que toma f para las asignaciones de $x_i \in \{0,1\}$. Por lo tanto, si dos funciones de campo f_1 y f_2 son iguales para todo vector Booleano, entonces sus polinomios $\mathcal{A}[f_1]$ y $\mathcal{A}[f_2]$ son idénticos.

Definición 3 *La función f_1 es 0,1-equivalente a la función f_2, representado como $f_1 \overset{0,1}{\equiv} f_2$, si y sólo si f_1 y f_2 son dos funciones tales que $f_1(\underline{b}) = f_2(\underline{b})$ para cualquier vector Booleano $\underline{b}$.*

Por tanto, dadas dos funciones de campo tales que $f_1 \overset{0,1}{\equiv} f_2$, entonces se cumple que $\mathcal{A}[f_1] = \mathcal{A}[f_2]$. Pero si $f_1 \overset{0,1}{\not\equiv} f_2$, ¿se cumple que $\mathcal{A}[f_1] = \mathcal{A}[f_2]$? Esta pregunta se responde utilizando el hecho de que cuando una *transformada* $\mathcal{A}$ se evalúa sobre un vector Booleano, únicamente se considera el polinomio clave correspondiente a ese vector particular.

Teniendo en cuenta estas consideraciones, se puede demostrar [JABF92] el siguiente teorema.

Teorema 1 *Para cualquier vector Booleano $\underline{b_0} \in \{0,1\}^n$, $\mathcal{A}[f](\underline{b_0}) = f(\underline{b_0})$, es decir, $\mathcal{A}[f] \overset{0,1}{\equiv} f$.*

Las conclusiones importantes que se pueden obtener de lo expuesto hasta el momento serían las siguientes:

- Aunque la aplicación de la *transformada* $\mathcal{A}$ a una función Booleana β incrementa su dominio de $\{0,1\}^n$ a $\mathcal{F}^n$, $\mathcal{A}[\beta]$ proporciona los mismos valores que β cuando se evalúa sobre un vector Booleano.

- Las representaciones algebraicas de dos funciones Booleanas distintas difieren en todos los vectores Booleanos que distinguen las dos funciones. Por lo tanto, las representaciones serán diferentes, es decir, si $\beta_1 \neq \beta_2$, entonces $\mathcal{A}[\beta_1] \neq \mathcal{A}[\beta_2]$.

2.1.3. Cálculo de la *transformada* $\mathcal{A}$ de una expresión Booleana

El cálculo de la *transformada* $\mathcal{A}$ de una expresión Booleana utilizando directamente los polinomios clave no es un método conveniente porque normalmente se necesitaría un número exponencial de términos. Es preferible calcular la *transformada* $\mathcal{A}$ de una expresión compleja de una manera *incremental*, calculando y posteriormente combinando las *transformadas* $\mathcal{A}$ de cada uno de sus términos componentes.

El primer paso en el cálculo de $\mathcal{A}[\beta]$ consiste en el reemplazamiento de la función Booleana β por una función de campo f tal que $\mathcal{A}[\beta] = \mathcal{A}[f]$. Para ello, es necesario reemplazar la función β por una función de campo f que sea 0,1-*equivalente*, porque si $\beta \overset{0,1}{\equiv} f$, entonces $\mathcal{A}[\beta] = \mathcal{A}[f]$. Este es un caso especial de la regla que se da a continuación [JABF92] para la manipulación de las *transformadas* $\mathcal{A}$.

Regla 1 (Sustitución) *Si ψ_1 es una subexpresión de ψ y $\psi_2 \overset{0,1}{\equiv} \psi_1$, entonces se puede reemplazar ψ_1 por ψ_2 en $\mathcal{A}[\psi]$.*

Para encontrar una función f que sea 0,1-*equivalente* a una expresión Booleana dada, se pueden utilizar las 0,1-*equivalencias* mostradas en la tabla 2.2. La primera columna de esta tabla indica las operaciones Booleanas $\odot$, donde β, β_1 y β_2 son funciones Booleanas. En la segunda columna, para cada una de las operaciones Booleanas $\odot$ se da una función de campo 0,1-*equivalente*, y en la tercera columna se definen las operaciones Booleanas extendidas $\odot_{\mathcal{F}}$ correspondientes.

Op. Booleana $\odot$	0,1-equivalencia	Extensión $\odot_{\mathcal{F}}$
$\neg\beta$	$\neg\beta \overset{0,1}{\equiv} 1 - \beta$	$\neg_{\mathcal{F}}(\beta) = 1 - \beta$
$\beta_1 \wedge \beta_2$	$\beta_1 \wedge \beta_2 \overset{0,1}{\equiv} \beta_1 \cdot \beta_2$	$\beta_1 \wedge_{\mathcal{F}} \beta_2 = \beta_1 \cdot \beta_2$
$\beta_1 \vee \beta_2$	$\beta_1 \vee \beta_2 \overset{0,1}{\equiv} \beta_1 + \beta_2 - \beta_1 \cdot \beta_2$	$\beta_1 \vee_{\mathcal{F}} \beta_2 = \beta_1 + \beta_2 - \beta_1 \cdot \beta_2$
$\beta_1 \oplus \beta_2$	$\beta_1 \oplus \beta_2 \overset{0,1}{\equiv} \beta_1 + \beta_2 - 2\beta_1 \cdot \beta_2$	$\beta_1 \oplus_{\mathcal{F}} \beta_2 = \beta_1 + \beta_2 - 2\beta_1 \cdot \beta_2$

Tabla 2.2: Operaciones Booleanas extendidas

Aunque la regla anterior permite una primera manipulación, no es suficiente para la realización de un cálculo incremental de la *transformada* $\mathcal{A}$. Por ejemplo, para el cálculo de $\mathcal{A}[\beta_1 \wedge \beta_2]$, no se puede calcular simplemente $\mathcal{A}[\beta_1]\cdot\mathcal{A}[\beta_2]$, ya que para $\beta_1 = \beta_2 = x$ esto resultaría en $\mathcal{A}[x]\cdot\mathcal{A}[x] = x\cdot x = x^2$, mientras que la respuesta correcta sería $\mathcal{A}[x \wedge x] = x$, usando la ecuación 2.2.

El cálculo correcto se puede realizar utilizando la equivalencia Booleana $x \wedge x = x$, de modo que $\mathcal{A}[x \wedge x] = \mathcal{A}[x] = x$. ¿Cómo se tendría que realizar este cálculo cuando se utilizan funciones de campo? En primer lugar se reemplazaría la operación Booleana $\wedge$ por la operación extendida $\wedge_{\mathcal{F}}$, con lo que se tendría que $\mathcal{A}[x \wedge x] = \mathcal{A}[x \cdot x] = \mathcal{A}[x^2]$. A continuación se utilizaría la 0,1-*equivalencia* $x^2 \overset{0,1}{\equiv} x$ para reemplazar x^2 por x *dentro* de la *transformada* $\mathcal{A}$, lo que resultaría en $\mathcal{A}[x]$ y, por tanto, en el resultado correcto x. Este reemplazamiento hace que se cumpla la propiedad de *idempotencia* de la multiplicación, es decir, la eliminación del exponente de x para la obtención del resultado correcto. Por tanto, la *transformada* $\mathcal{A}$ es *insensible al exponente* ya que los exponentes se pueden eliminar *dentro* de una transformada utilizando la 0,1-*equivalencia* $x^i \overset{0,1}{\equiv} x^j$, para $i, j > 0$.

Se puede establecer, por tanto, la siguiente importante regla.

Regla 2 (Reducción del exponente) *En* $\mathcal{A}[\psi]$*, cualquier aparición de* x_i^k $(k > 1)$ *se puede reemplazar por* x_i*.*

Por lo tanto, se puede hacer cumplir la propiedad de idempotencia aplicando la *transformada* $\mathcal{A}$ después de combinar las *transformadas* $\mathcal{A}$ de las subfunciones, es decir, se puede calcular $\mathcal{A}[\beta_1 \wedge \beta_2]$ como $\mathcal{A}[\mathcal{A}[\beta_1]\cdot\mathcal{A}[\beta_2]]$. Este último paso se puede realizar para cualquier operación Booleana extendida, con lo que para el cálculo de $\mathcal{A}[x \odot_{\mathcal{F}} y]$ se pueden sustituir x e y por $\mathcal{A}[x]$ y $\mathcal{A}[y]$, respectivamente, obteniendo $\mathcal{A}[\mathcal{A}[x] \odot_{\mathcal{F}} \mathcal{A}[y]]$.

Se puede establecer, por tanto, el siguiente teorema.

Teorema 2 *Para cualquier operación Booleana extendida* $\odot_{\mathcal{F}}$*,*

$$\mathcal{A}[f_1 \odot_{\mathcal{F}} f_2] = \mathcal{A}[\mathcal{A}[f_1] \odot_{\mathcal{F}} \mathcal{A}[f_2]] \tag{2.3}$$

Es importante observar que la necesidad de cumplimiento de la propiedad de idempotencia se da únicamente cuando las funciones f_1 y f_2 tienen una variable en común. Considérese el caso en que f_1 y f_2 presentan *soporte disjunto*[5]. Si f_1 y f_2 tienen soporte disjunto, sus *transformadas* $\mathcal{A}$ se pueden combinar directamente $(\mathcal{A}[f_1] \odot_{\mathcal{F}} \mathcal{A}[f_2])$, sin necesidad de aplicación de la *transformada* $\mathcal{A}$ al resultado, como en el teorema 2.

Se puede enunciar el siguiente teorema [JABF92].

[5]El *soporte* de una función es el conjunto de variables de las que depende la función. Por tanto, dos funciones f_1 y f_2 tienen *soporte disjunto* si no tienen variables comunes.

Teorema 3 (Teorema de soporte disjunto) *Si f_1 y f_2 tienen soporte disjunto, entonces*

$$\mathcal{A}[f_1 \odot_{\mathcal{F}} f_2] = \mathcal{A}[f_1] \odot_{\mathcal{F}} \mathcal{A}[f_2] \tag{2.4}$$

donde $\odot_{\mathcal{F}}$ representa cualquier operación Booleana extendida.

Los teoremas 2 y 3 permiten la evaluación incremental de la transformada algebraica de una función Booleana. Para ello, se deben evaluar las transformadas de sus subfunciones, combinarlas utilizando la operación Booleana extendida apropiada y aplicar posteriormente la *transformada* $\mathcal{A}$.

Por ejemplo, sean $\beta_1 = x_1 \wedge \overline{x}_2$ y $\beta_2 = x_1 \vee (x_2 \wedge x_3)$, cuyas transformadas son $\mathcal{A}[\beta_1] = x_1(1 - x_2)$ y $\mathcal{A}[\beta_2] = x_1 + x_2x_3 - x_1x_2x_3$, respectivamente. Si $\beta_3 = \beta_1 \wedge \beta_2$, entonces se tiene que $\mathcal{A}[\beta_3] = \mathcal{A}[\mathcal{A}[\beta_1] \wedge_{\mathcal{F}} \mathcal{A}[\beta_2]] = \mathcal{A}[\mathcal{A}[\beta_1] \cdot \mathcal{A}[\beta_2]]$ $= \mathcal{A}[x_1(1-x_2)(x_1+x_2x_3-x_1x_2x_3)] = \mathcal{A}[x_1^2+x_1x_2x_3-x_1^2x_2x_3-x_1^2x_2-x_1x_2^2x_3+$ $x_1^2x_2^2x_3] = \mathcal{A}[x_1 + x_1x_2x_3 - x_1x_2x_3 - x_1x_2 - x_1x_2x_3 + x_1x_2x_3] = \mathcal{A}[x_1 - x_1x_2] =$ $\mathcal{A}[x_1(1 - x_2)]$.

En la siguiente subsección, vemos cómo se puede obtener el polinomio una vez que se ha calculado la transformada $\mathcal{A}[\beta]$ de la función Booleana β.

2.1.4. Cálculo del polinomio simbólico de una *transformada* $\mathcal{A}$

El cálculo del polinomio simbólico $\mathcal{A}[f]$ utilizando la definición dada de la transformada $\mathcal{A}$ es muy difícil, por eso se necesitan otros métodos que permitan el cálculo del polinomio $\mathcal{A}[\psi]$ de una expresión de campo dada ψ.

Un método para el cálculo del polinomio simbólico consiste en proceder variable por variable, utilizando un método análogo a la *expansión de Boole*[6] [Bro90] de una función Booleana. La ecuación 2.2 que define la *transformada* $\mathcal{A}$, se puede expandir de forma parecida a la expansión de Boole de una función Booleana para una variable particular x, agrupando por una parte todos los términos que contengan el factor "x", y por otra todos los términos que contengan el factor "$1 - x$".

Por ejemplo, sea la función $\beta(x_1, x_2)$ y sea su *transformada* $\mathcal{A}$ dada por $\mathcal{A}[\beta] = \beta(0,0)(1-x_1)(1-x_2)+\beta(0,1)(1-x_1)x_2+\beta(1,0)x_1(1-x_2)+\beta(1,1)x_1x_2$. Entonces, operando, esta expresión se puede factorizar de la siguiente forma $\mathcal{A}[\beta] = (1 - x_1) \cdot [\beta(0,0)(1 - x_2) + \beta(0,1)x_2] + x_1 \cdot [\beta(1,0)(1 - x_2) + \beta(1,1)x_2]$. Las expresiones entre corchetes son las transformadas algebraicas de dos *restricciones* de β, donde x_1 se ha reemplazado por 0 y 1, respectivamente. Se suele utilizar la notación $f_{x=R}$ para denotar la restricción de f en la que x se reemplaza por R ($R \in \mathcal{F}$).

[6] El *teorema fundamental del álgebra Booleana*, introducido por Boole en [Boo54] y frecuentemente atribuido a Shannon [Sha49], establece que si f es una función Booleana de n variables $f : \mathbf{B}^n \longrightarrow \mathbf{B}$, entonces se tiene que $f(x_1, x_2, \ldots, x_n) = \overline{x}_1 \cdot f(0, x_2, \ldots, x_n) + x_1 \cdot f(1, x_2, \ldots, x_n)$, para todo $(x_1, x_2, \ldots, x_n)$ en $\mathbf{B}^n$.

Se puede establecer el siguiente teorema para el cálculo del polinomio simbólico.

Teorema 4 (Teorema de expansión lineal) *Sea $\mathcal{A}[f]$ dependiente de la variable x, de modo que $\mathcal{A}[f](\ldots, x, \ldots)$. Entonces,*

$$\mathcal{A}[f] = (1 - x) \cdot \mathcal{A}[f_{x=0}] + x \cdot \mathcal{A}[f_{x=1}] \tag{2.5}$$

donde $f_{x=0(1)}$ representa la restricción de f a $x = 0(1)$.

También es posible desarrollar un conjunto de reglas suficientes que permitan el cálculo del polinomio $\mathcal{A}[\psi]$. Este conjunto de reglas viene dado por el siguiente teorema [JABF92].

Teorema 5 *Para cualesquiera funciones de campo f_1 y f_2, y cualquier constante $c \in \mathcal{F}$, se tiene que*

1. $\mathcal{A}[c \cdot f_1] = c \cdot \mathcal{A}[f_1]$
2. $\mathcal{A}[f_1 + f_2] = \mathcal{A}[f_1] + \mathcal{A}[f_2]$
3. $\mathcal{A}[f_1 - f_2] = \mathcal{A}[f_1] - \mathcal{A}[f_2]$
4. $\mathcal{A}[x_i] = x_i$
5. $\mathcal{A}[c] = c$
6. $\mathcal{A}[f_1 \cdot f_2] = \mathcal{A}[f_1] \cdot \mathcal{A}[f_2]$, *si f_1 y f_2 tienen soporte disjunto*

Con el conjunto de reglas dadas en el teorema 5 y con la regla 2 de reducción del exponente, se puede calcular el polinomio simbólico de cualquier expresión de campo ψ.

2.2. Introducción de valores numéricos

En la sección anterior se ha visto cómo se puede determinar el polinomio simbólico de una expresión de campo dada, pero el objetivo no es obtener un polinomio sino un *valor numérico* o *signatura* que represente a la función, y que será el valor que toma el polinomio para la asignación aleatoriamente seleccionada de sus variables.

La introducción de un valor numérico en una representación algebraica consiste simplemente en la sustitución del valor numérico dado por el símbolo equivalente y en la realización de las simplificaciones correspondientes, obteniéndose de esta forma un polinomio *seminumérico* que se puede definir utilizando el concepto de *restricción funcional*. Como cada variable tiene asociado un valor numérico aleatorio, se puede especificar un *conjunto de restricción*

ρ que indica el conjunto de variables que serán sustituidas por sus valores correspondientes. De esta forma, el *polinomio seminumérico* de una función f se representa como $\mathcal{A}_{\{\rho\}}[f]$. Por ejemplo, si $\beta = x_1 \wedge \overline{x}_2$ con asignación aleatoria $x_1 = 2$ y $x_2 = 4$, se tiene que $\mathcal{A}[\beta] = x_1(1 - x_2)$, y $\mathcal{A}_{\{x_1\}}[\beta] = 2(1 - x_2)$, $\mathcal{A}_{\{x_2\}}[\beta] = -3x_1$ y $\mathcal{A}_{\{x_1,x_2\}}[\beta] = -6$, que es la *signatura* de β para la asignación aleatoria dada.

Dos polinomios seminuméricos no se pueden combinar si se restringen a una variable común. Por ejemplo, sea $\beta = \beta_1 \wedge \beta_2$ con $\beta_1 = x_1 \wedge x_2$ y $\beta_2 = x_1 \wedge \overline{x}_3$, y con asignación aleatoria $x_1 = 2$, $x_2 = 4$ y $x_3 = 3$. Se puede comprobar que $\mathcal{A}_{\{x_1,x_2,x_3\}}[\beta_1] = 8$, $\mathcal{A}_{\{x_1,x_2,x_3\}}[\beta_2] = -4$ y que $\mathcal{A}_{\{x_1,x_2,x_3\}}[\beta] = -16$ (signatura de β), sin embargo, si se combinan los valores numéricos de las subfunciones obtendríamos $\mathcal{A}[\mathcal{A}_{\{x_1,x_2,x_3\}}[\beta_1] \cdot \mathcal{A}_{\{x_1,x_2,x_3\}}[\beta_2]] = \mathcal{A}[8 \cdot (-4)] = -32$, que es un valor erróneo. Por lo tanto, únicamente se debe restringir uno de los polinomios a una variable x cuando el otro polinomio *no dependa* de x. Una función f *no depende* de una variable x [JABF92] si y sólo si para todo $S_1, S_2 \in \mathcal{F}$, se tiene que $f_{x=S_1} = f_{x=S_2}$, y se representa como $f \perp x$.

Se puede establecer ahora la generalización del teorema 2 para polinomios seminuméricos.

Teorema 6 (Teorema de combinación seminumérica) *Sean f_1 y f_2 dos funciones y sea ρ un conjunto de restricción. Sean $\rho_1 = \{x \in \rho \mid f_2 \perp x\}$ y $\rho_2 = \{x \in \rho \mid f_1 \perp x\}$. Entonces*

$$\mathcal{A}_{\{\rho\}}[\mathcal{A}[f_1] \odot_{\mathcal{F}} \mathcal{A}[f_2]] = \mathcal{A}_{\{\rho - \rho_1 - \rho_2\}}[\mathcal{A}_{\{\rho_1\}}[f_1] \odot_{\mathcal{F}} \mathcal{A}_{\{\rho_2\}}[f_2]] \tag{2.6}$$

donde $-$ representa diferencia de conjuntos y $\odot_{\mathcal{F}}$ representa cualquier operación Booleana extendida.

Por lo tanto, un método para el cálculo de la signatura de una función que se obtuviera de la combinación de dos funciones f_1 y f_2 consistiría en la sustitución de los valores numéricos aleatoriamente seleccionados de aquellas variables que aparecieran en f_1 pero no en f_2 (y viceversa), en la combinación de los polinomios seminuméricos obtenidos $\mathcal{A}_{\{\rho_1\}}[f_1]$ y $\mathcal{A}_{\{\rho_2\}}[f_2]$ utilizando las leyes de las *transformadas* $\mathcal{A}$ y, por último, en la sustitución de los valores aleatorios numéricos de las variables comunes a las dos funciones f_1 y f_2 en el polinomio seminumérico resultante, previa aplicación de la propiedad de idempotencia de la multiplicación.

El caso ideal sería aquel en el que se pudieran evitar todas las comprobaciones de la propiedad de idempotencia, de forma que la signatura de una función se calculara simplemente realizando la combinación apropiada de los valores numéricos.

Vemos a continuación los casos en los que es posible obtener la signatura de una función f calculando en primer lugar las signaturas de sus subfunciones y combinándolas posteriormente.

Si dos funciones f_1 y f_2 tienen *soporte disjunto*[7], entonces se cumple que $\mathcal{A}[f_1 \odot_{\mathcal{F}} f_2] = \mathcal{A}[f_1] \odot_{\mathcal{F}} \mathcal{A}[f_2]$, pudiéndose aplicar cualquier conjunto de restricción a ambos lados de la ecuación. Se puede establecer por tanto el siguiente teorema [JABF92], que sería el análogo numérico del teorema 3 y que permite la combinación de las signaturas obtenidas para dos subfunciones $\mathcal{A}[f_1]$ y $\mathcal{A}[f_2]$ cuando f_1 y f_2 no tienen variables en común.

Teorema 7 (Teorema de combinación de soporte disjunto) *Si f_1 y f_2 tienen soporte disjunto, entonces se tiene que*

$$\mathcal{A}_{\{\rho\}}[f_1 \odot_{\mathcal{F}} f_2] = \mathcal{A}_{\{\rho\}}[f_1] \odot_{\mathcal{F}} \mathcal{A}_{\{\rho\}}[f_2] \tag{2.7}$$

para cualquier conjunto de restricción ρ.

Si f_1 y f_2 no tienen soporte disjunto, únicamente se puede poner que $\mathcal{A}[f_1 \odot_{\mathcal{F}} f_2] = c_1 + c_2 \cdot \mathcal{A}[f_1] + c_3 \cdot \mathcal{A}[f_2] + c_4 \cdot \mathcal{A}[f_1 \cdot f_2]$, con $c_1, \ldots, c_4 \in \mathcal{F}$, donde $\odot_{\mathcal{F}}$ representa cualquier operación Booleana extendida en *forma bilineal*, es decir, $x \odot_{\mathcal{F}} y = c_1 + c_2 \cdot x + c_3 \cdot y + c_4 \cdot xy$. Se puede observar que el término $\mathcal{A}[f_1 \cdot f_2]$ es el que impide la combinación directa de las signaturas de dos funciones cualesquiera f_1 y f_2.

Por lo tanto, se pueden considerar las condiciones que deben cumplir las dos funciones f_1 y f_2 para que se pueda eliminar este término. La más simple sería que $\mathcal{A}[f_1 \cdot f_2] = 0$. Se puede comprobar que esta condición corresponde a la necesidad de que las dos funciones Booleanas β_1 y β_2 sean *ortogonales* (o *disjuntas*[8]), es decir, $\beta_1 \wedge \beta_2 = 0$. Cuando se cumple esta condición de ortogonalidad de las funciones, entonces se puede comprobar que $\mathcal{A}[\beta_1 \vee \beta_2] = \mathcal{A}[\beta_1] + \mathcal{A}[\beta_2]$, que se puede extender por inducción para establecer el siguiente teorema [JABF92].

Teorema 8 (Teorema de ortogonalidad) *Sean $\beta_1, \ldots, \beta_k$ funciones Booleanas. Se tiene entonces que*

$$\mathcal{A}[\beta_1 \vee \cdots \vee \beta_k] = \mathcal{A}[\beta_1] + \cdots + \mathcal{A}[\beta_k] \tag{2.8}$$

si las k funciones $\beta_1, \ldots, \beta_k$ son mutuamente ortogonales.

Por lo tanto, $\mathcal{A}_{\{\rho\}}[\beta_1 \vee \cdots \vee \beta_k] = \mathcal{A}_{\{\rho\}}[\beta_1] + \cdots + \mathcal{A}_{\{\rho\}}[\beta_k]$ para cualquier conjunto de restricción ρ.

[7] Utilizando la notación de *dependencia* de una función con respecto a una variable vista anteriormente, se tendría que dos funciones f_1 y f_2 tienen *soporte disjunto* si y sólo si para cualquier variable x, $f_1 \perp x$ o $f_2 \perp x$.

[8] Dos funciones Booleanas β_1 y β_2 son *disjuntas* cuando no tienen *mintérminos* comunes, por lo que se puede comprobar que en este caso también se cumple que $\beta_1 \wedge \beta_2 = 0$.

2.3. Aplicación a la verificación probabilista

Las secciones anteriores permiten el cálculo de la *transformada* $\mathcal{A}$ de una función Booleana y su evaluación sobre un conjunto de valores numéricos aleatoriamente seleccionados. La comprobación probabilista de la equivalencia de dos funciones β_1 y β_2 requiere la comparación de las signaturas $\mathcal{A}[\beta_1](\underline{v})$ y $\mathcal{A}[\beta_2](\underline{v})$ obtenidas bajo la asignación aleatoria $\underline{v}$ (también se podría utilizar la notación vista anteriormente $\mathcal{A}_{\{\rho\}}[\beta_1]$ y $\mathcal{A}_{\{\rho\}}[\beta_2]$ si el conjunto de restricción ρ está formado por todas las variables de las que dependen β_1 y β_2, es decir, si ρ es igual al *soporte* de β_1 y β_2). La equivalencia (o no equivalencia) de las dos funciones vendrá determinada por la igualdad (o desigualdad) de sus signaturas.

Sean β_1 y β_2 dos funciones y sea $\underline{v}$ una asignación numérica aleatoriamente seleccionada de un campo finito genérico $\mathcal{F}$. El método *probabilista* de verificación de la equivalencia de β_1 y β_2 consta de las siguientes fases:

1. Cálculo de las *transformadas* $\mathcal{A}$ de las funciones β_1 y β_2.
2. Evaluación de $\mathcal{A}[\beta_1]$ y $\mathcal{A}[\beta_2]$ sobre la asignación aleatoria $\underline{v}$, obteniendo las signaturas correspondientes $S_1 = \mathcal{A}[\beta_1](\underline{v})$ y $S_2 = \mathcal{A}[\beta_2](\underline{v})$.
3. Comparación de S_1 y S_2, existiendo dos posibilidades:
   - Si $S_1 \neq S_2$, entonces se garantiza que $\beta_1 \not\equiv \beta_2$.
   - Si $S_1 = S_2$, entonces $\beta_1 \equiv \beta_2$ (de manera probabilista).

Se puede calcular la *probabilidad de error* en la verificación [Jai95], es decir, la probabilidad de que para $S_1 = S_2$, $\beta_1 \not\equiv \beta_2$. Para ello, sea $\mathcal{F}$ un campo de cardinalidad p.

Teorema 9 *Sean f_1 y f_2 dos funciones de campo de n variables tales que $\mathcal{A}[f_1] \neq \mathcal{A}[f_2]$, y sea $\underline{v}$ un vector de longitud n de elementos aleatoriamente seleccionados de $\mathcal{F}$. Entonces*

$$\epsilon = Prob(\mathcal{A}[f_1](\underline{v}) = \mathcal{A}[f_2](\underline{v})) \leq 1 - \left(\frac{p-1}{p}\right)^n \tag{2.9}$$

Si ϵ representa el límite superior en la probabilidad del error, entonces si $p \gg n$, que es una suposición razonable, se tiene que $\epsilon \approx n/p$. En caso contrario, $\epsilon \approx 1 - e^{-n/p}$.

La reducción del error ϵ hasta un valor aceptable se puede conseguir utilizando varias técnicas. Una de ellas consiste en el incremento de la cardinalidad p del campo $\mathcal{F}$ utilizado. Otra técnica consiste en la realización de varias *ejecuciones*, donde en cada *ejecución* se selecciona aleatoriamente un conjunto independiente de asignaciones de variables de entrada y se calculan las signaturas de las funciones. Si los valores son diferentes, las funciones no son

equivalentes, y si los valores son iguales, entonces se selecciona un nuevo conjunto de asignaciones de entrada y se vuelven a evaluar las funciones. De esta forma, la probabilidad de decidir de forma errónea que las funciones son iguales decrece exponencialmente con el número de ejecuciones, de modo que después de k ejecuciones, la probabilidad del error es ϵ^k.

2.4. Comentarios

En este capítulo se han introducido los fundamentos teóricos de la verificación probabilista por medio de la utilización de transformaciones algebraicas de las funciones Booleanas. La evaluación posterior de las transformaciones obtenidas sobre un conjunto de elementos aleatoriamente seleccionados de un campo finito, proporcionará unas *signaturas* que serán utilizadas para decidir de una manera *probabilista* sobre la equivalencia de las funciones Booleanas originales. Algunos de los teoremas vistos, como el *teorema de combinación de soporte disjunto* o el *teorema de ortogonalidad*, serán muy importantes para el desarrollo de los capítulos de verificación posteriores, donde se utilizarán los campos de Galois como campos finitos y se verá la utilidad de estos campos en el cálculo y simplificación de las operaciones aritméticas a realizar para la obtención de las signaturas.

En el *teorema de ortogonalidad* se ha visto que la propiedad de *ortogonalidad* (o *disjunción*) de las funciones permite la combinación directa de sus signaturas, con lo que se simplifica en gran medida el cálculo de la signatura total. Por este motivo, en el capítulo siguiente se introduce una nueva formulación para el cálculo de operaciones Booleanas en la que se cumple la propiedad de *disjunción*. Esta formulación será utilizada a su vez en el capítulo 5 para la creación de un método *híbrido* de verificación.

Capítulo 3

Formulación para el cálculo de operaciones Booleanas

En este capítulo se describe una nueva formulación para el cálculo de operaciones Booleanas realizadas sobre funciones representadas en forma cúbica. Como resultado se obtiene un conjunto de cubos disjuntos. La propiedad de disjunción es muy importante en la verificación probabilista para la simplificación del cálculo de la signatura de las funciones Booleanas transformadas. Se presenta un ejemplo de cálculo utilizando nuestra aproximación y se dan los resultados experimentales obtenidos.

Las ecuaciones Booleanas o de *conmutación* [Boo54] son unas de las herramientas más utilizadas para el diseño y test de circuitos digitales [SBS95] [SBSV96]. Muchos de los problemas existentes en este tipo de aplicaciones, como la generación de patrones de test [BBP91] y la estimación de probabilidades de señal para circuitos combinacionales [KT89], se podrían resolver de forma eficiente si existiese un procedimiento rápido de resolución de ecuaciones Booleanas. Se han propuesto muchas alternativas para dicha resolución, pudiéndose clasificar según el método utilizado para la obtención de la solución en métodos *algebraicos*, *tabulares* y *basados en diagramas de decisión*.

Los métodos *algebraicos* [Rud74a][Rud74b] se basan principalmente en manipulaciones algebraicas de expresiones Booleanas. Sus principales ventajas son su generalidad y su compacidad, pero no son métodos adecuados para su procesamiento por computador. Los métodos *tabulares* [Svo63][Pic74][TLROL93] [Ung94] se basan en la definición de una correspondencia de los 2^n valores de una función de n variables. Su principal ventaja consiste en su facilidad de implementación computacional, aunque su necesidad de almacenamiento y evaluación de al menos 2^n valores puede convertir a este método en impracticable. Los métodos *basados en ROBDDs* [WM98] representan las expresiones

Booleanas en forma de grafos acíclicos dirigidos [Bry86]. Como se verá en el capítulo 4, sus principales ventajas son su canonicidad y su facilidad de manipulación, pero su eficiencia depende mucho de la ordenación de variables utilizada [MWBSV88][BRKM91][Rud93] y muchas de las funciones Booleanas con utilidad práctica tienen ROBDDs de tamaño exponencial para cualquier ordenación de las variables de entrada [Weg94].

En [TLROL93] se desarrolló una técnica tabular para ecuaciones Booleanas de la forma $f(x) = 1$, donde $f(x)$ está en forma factorizada. El método se mejoró posteriormente [Ung94] generalizándolo para la resolución de las ecuaciones $f(x) = 0$ y $f(x) = g(x)$. La técnica propuesta en [TLROL93] convierte formas factorizadas en expresiones de dos niveles en forma de *sumas de productos* o SOP (*sum-of-products*), representadas en forma tabular. Las expresiones Booleanas en forma SOP se representan por tablas rectangulares de 0s, 1s e *indeterminaciones* (*don't cares*), donde cada fila describe un producto de literales en la representación *cúbica* tradicional de un término producto [BHMSV84] y representa un conjunto de mintérminos. Esta técnica requiere que los términos producto no tengan mintérminos comunes, es decir, los términos producto deben ser *disjuntos* entre sí, por lo que puede ser necesaria la realización de una fase de *separación* posterior para la obtención del resultado *disjunto* final.

A continuación se presenta una nueva formulación [IB99][IS01] para el cálculo de operaciones Booleanas realizadas sobre funciones Booleanas representadas en forma *cúbica disjunta*, donde los resultados obtenidos también vienen dados en forma de conjuntos de cubos que satisfacen la propiedad de *disjunción*. En primer lugar se dan los conceptos básicos necesarios que se usarán a lo largo del capítulo. A continuación se introducen unas definiciones que se utilizarán posteriormente en los teoremas y corolarios que constituyen nuestro método de cálculo de operaciones Booleanas. Se da un ejemplo y se muestran los resultados experimentales obtenidos en la implementación del método. Finalmente se presentan algunas conclusiones.

3.1. Preliminares

Un *álgebra Booleana* [Bro90] es una estructura matemática formada por la quíntupla $\{\mathbf{B}, \mathbf{0}, \mathbf{1}, +, \cdot\}$, donde el conjunto *portador* $\mathbf{B}$ contiene al menos dos elementos distintos $\mathbf{0}$ y $\mathbf{1}$, sobre el que se definen las operaciones binarias *producto Booleano* o *conjunción* (representado por $\cdot$ o $\wedge$), *suma Booleana* o *disyunción* (representado por $+$ o $\vee$) y la operación unitaria *complemento* (representada por $\neg$ o por una línea horizontal sobre el elemento a complementar). El conjunto $\mathbf{B}$ representa cualquier conjunto de elementos que forman un álgebra Booleana, siendo el más simple el formado únicamente por los dos elementos $\mathbf{B} = \{0,1\}$.

Una función de *conmutación* de n variables es una correspondencia de la forma $f : \mathbf{B}^n \longrightarrow \mathbf{B}$, donde $\mathbf{B}$ es el álgebra Booleana de dos elementos {0,1}. El *dominio* de esta función (también llamado *espacio Booleano*) tiene 2^n elementos y el *codominio* tiene dos elementos, por lo que existen 2^{2^n} funciones de conmutación de n variables. También se puede demostrar que toda función de conmutación es una función Booleana. Una *variable Booleana* v es un símbolo que representa una *señal* Booleana o una *coordenada* del espacio Booleano [Bro90]. Un *literal* (por ejemplo, v o $\overline{v}$) es un símbolo que representa una variable o su complemento.

Se representa con X el conjunto ordenado de variables de entrada de una función Booleana, $X = \{x_{n-1}, \ldots, x_1, x_0\}$, y el *soporte* de una función Booleana denota el conjunto de variables de las que depende la función en realidad. Por ejemplo, si $X = \{x_2, x_1, x_0\}$ y la función $f = x_1 \cdot x_0$, entonces se tiene que el $soporte(f) = \{x_1, x_0\}$.

La funciones Booleanas se pueden expresar por medio de un conjunto de cubos, donde un *cubo* se puede considerar como una n-upla de valores de $\{0, 1, d\}$ (siendo d una *indeterminación*) o como un conjunto de literales. Por lo tanto, un *cubo* q es una n-upla (o vector) de literales, $q = (q_{n-1}, \ldots, q_1, q_0)$, con $q_i \in \{0, 1, d\}$.

Realizando una asignación de variables a cada posición de la tupla, un cubo representa un *término producto*, es decir, la conjunción Booleana de los literales. El símbolo 0 (1) representa la aparición en el término producto del literal de la variable correspondiente en forma complementada (sin complementar). El símbolo d (*indeterminación* o *don´t care*) indica que el literal de la variable correspondiente no está presente en el término producto. Por ejemplo, el cubo $q = (d1d0)$ con el conjunto ordenado de variables $\{x_3, x_2, x_1, x_0\}$ representa el término producto $x_2\overline{x}_0$.

Un *mintérmino* es un término producto en el que aparecen todos los literales, por lo que se representa por medio de una n-upla de valores pertenecientes a {0,1}. Un cubo que contiene el símbolo d entre sus componentes representa el conjunto de todos los cubos formados asignando 0 o 1 a d en todas las combinaciones posibles, por lo que el vector $(d, d, \ldots, d)$ de tamaño n representa el espacio Booleano completo n-dimensional $\mathbf{B}^n$. Por ejemplo, el cubo $q = (0dd1) = \{0001, 0011, 0101, 0111\}$, y este conjunto constituye el espacio Booleano representado por q.

Una función Booleana f se puede expresar por medio de un conjunto de cubos, de forma que $\{f\} = \{f^0, f^1, f^2, \ldots\}$. Este conjunto de cubos representa una expresión de dos niveles en forma de *suma de productos* (SOP) o *cobertura* (*cover*), que es una suma (o disyunción) de productos de literales (o cubos). Por ejemplo, si se tiene el conjunto de cubos $\{dd1d, 01d1\}$ con la ordenación de variables $\{x_3, x_2, x_1, x_0\}$, entonces representa la expresión en forma de suma de productos $x_1 + \overline{x}_3 x_2 x_0$.

La definición de cubo implica la no existencia de cubos idénticos, y en el desarrollo de nuestro método que se presenta a continuación se asume que todos los cubos pertenecientes a un conjunto de cubos son *disjuntos* (es decir, sin mintérminos comunes) y que están definidos sobre el mismo conjunto ordenado de variables. Este conjunto de cubos disjuntos constituye una *suma de productos disjuntos* o DSOP (*Disjoint sum-of-products*).

3.2. Definiciones básicas

Se presenta a continuación un conjunto de definiciones básicas específicas para nuestro método de cálculo de operaciones Booleanas y que se pueden encontrar en [IS01].

Definición 4 (Producto) *La operación binaria* $\odot$ *se llama* producto *y se define de la siguiente forma:* $0 \odot 0 = 0$, $0 \odot 1 = 1 \odot 0 = \emptyset$, $1 \odot 1 = 1$, $d \odot 0 = 0 \odot d = 0$, $d \odot 1 = 1 \odot d = 1$ *y* $d \odot d = d$, *como se muestra en la tabla 3.1 y donde* $\emptyset$ *representa el conjunto vacío (producto nulo).*
Para los cubos $q = (q_{n-1}, \ldots, q_1, q_0)$ *y* $r = (r_{n-1}, \ldots, r_1, r_0)$, *donde sus componentes toman los valores* 0, 1 *o* d, *se define el producto* $q \odot r$ *como*

$$q \odot r \quad = \quad (q_{n-1} \odot r_{n-1}, \ldots, q_1 \odot r_1, q_0 \odot r_0) \tag{3.1}$$

donde el resultado de $q \odot r$ *puede ser otro cubo o el conjunto vacío* $\emptyset$ *si uno de los productos de componentes es* $\emptyset$.
Para los conjuntos de cubos $\{q\}$ *y* $\{r\}$, *el producto* $\{q\} \odot \{r\}$ *es un conjunto de cubos definido como*

$$\{q\} \odot \{r\} \quad = \quad \{q^i \odot r^j \mid i \in [0, k-1],\ j \in [0, l-1],\ (q^i \odot r^j) \neq \emptyset\} \tag{3.2}$$

donde $\{q\} = \{q^0, q^1, \ldots, q^{k-1}\}$ *y* $\{r\} = \{r^0, r^1, \ldots, r^{l-1}\}$.

Según esta definición, el *producto*[1] de dos cubos produce su intersección, es decir, sus *subcubos* comunes. Si los cubos son disjuntos, entonces el resultado es un conjunto vacío.

Ejemplo 1 *La figura 3.1(a) muestra el resultado de una operación producto de cubos sobre un mapa de Karnaugh de cuatro variables* x_3, x_2, x_1, x_0. *Los tres cubos representados son* $q = (1d1d)$, $r = (11d1)$ *y* $s = (00d1)$. *El producto de q y s resulta en un conjunto vacío porque no tienen subcubos comunes (mintérminos), pero el producto de q y r produce el mintérmino común (1111) a ambos cubos (área rallada).* □

[1] No se debe confundir la operación *producto* dada en la definición 4 con la operación *producto Booleano* o *conjunción*. En este capítulo se utilizará el término *conjunción* en este último caso para evitar equívocos.

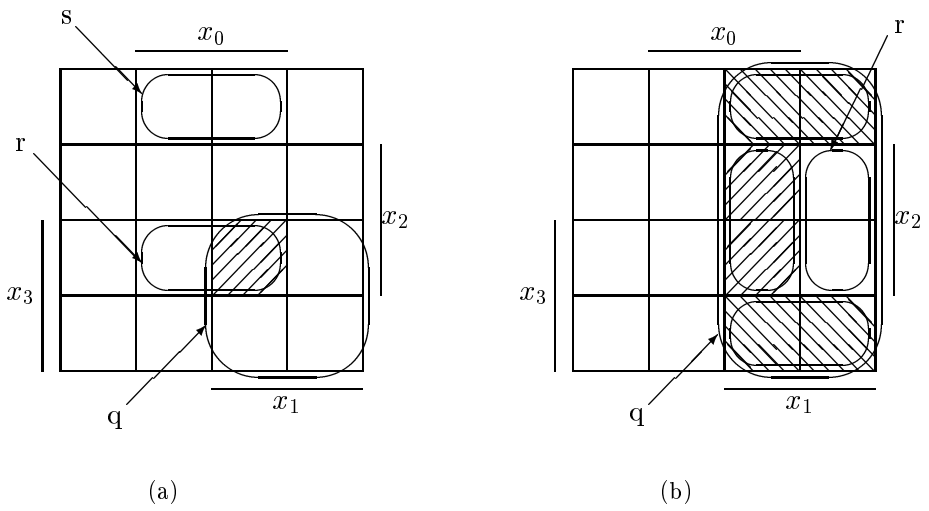


Figura 3.1: (a) Producto de cubos. (b) Restricción del cubo r por q.

Definición 5 (Diferencia) *La operación binaria* $\ominus$ *se llama* diferencia *y se define de la siguiente forma:* $0 \ominus 0 = 0$, $1 \ominus 1 = 1$, $0 \ominus 1 = 1 \ominus 0 = \emptyset$, $d \ominus 0 = 0 \ominus d = 1$, $d \ominus 1 = 1 \ominus d = 0$ *y* $d \ominus d = d$, *como se muestra en la tabla 3.1, y donde* $\emptyset$ *representa el conjunto vacío (diferencia nula).*

$\odot$	0	1	d
0	0	$\emptyset$	0
1	$\emptyset$	1	1
d	0	1	d

$\ominus$	0	1	d
0	0	$\emptyset$	1
1	$\emptyset$	1	0
d	1	0	d

Tabla 3.1: Operaciones *producto* y *diferencia*.

Definición 6 (Restricción) *Sean* q *y* r *dos cubos con* n *componentes (cardinalidad del conjunto de variables de entrada) de la forma* $q = (q_{n-1}, \ldots, q_1, q_0)$ *y* $r = (r_{n-1}, \ldots, r_1, r_0)$, *y sea* $r \subset q$, *es decir, el espacio Booleano representado por* r *está incluido en el espacio Booleano representado por* q.
La restricción de la j-ésima componente de r por q, *representado como* $\langle q|r\rangle_j$, *es un cubo definido como*

$$\langle q|r\rangle_j = (r_{n-1}, \ldots, r_{j+1}, q_j \ominus r_j, q_{j-1}, \ldots, q_0),\ j \in [n-1, 0] \tag{3.3}$$

si y sólo si $(q_j \ominus r_j) \neq \emptyset$, $\forall j \in [n-1, 0]$. *En caso contrario,* $\langle q|r\rangle_j = \emptyset$ *(conjunto vacío).*
La restricción de r por q, $\langle q|r\rangle$, *es un conjunto de cubos definido como*

$$\langle q|r\rangle = \{\langle q|r\rangle_j \mid j \in [n-1, 0],\ (q_j \ominus r_j) = 0(1) \text{ con } q_j, r_j \neq 0(1)\} \tag{3.4}$$

si y sólo si $\langle q|r\rangle_j \neq \emptyset, \forall j \in [n-1, 0]$. *En caso contrario,* $\langle q|r\rangle = \emptyset$.

Según esta definición, la *restricción de la j-ésima componente de r por q* produce el cubo complementado de r con respecto a la j-ésima componente que está incluido en q (o que pertenece a q). Por otra parte y según la misma definición, la *restricción de r por q* produce el conjunto de cubos incluidos en q que son distintos de r, es decir, el complemento de r con respecto (o restringido) al cubo q.

Ejemplo 2 *La restricción del cubo $r = (d110)$ por $q = (dd1d)$ se muestra en la figura 3.1(b). Esta operación produce los dos cubos $(d01d)$ y $(d111)$, representados por dos áreas ralladas diferentes, que son los (sub)cubos pertenecientes a q pero no a r.* □

Definición 7 (Restricción-producto) *Sean $\{q\}$ y $\{r\}$ dos conjuntos de cubos, $\{q\} = \{q^0, q^1, \ldots, q^{k-1}\}$ y $\{r\} = \{r^0, r^1, \ldots, r^{l-1}\}$, y sea $\{r\} \subset \{q\}$, es decir, el espacio Booleano representado por todos los cubos de $\{r\}$ está incluido en el espacio Booleano representado por todos los cubos de $\{q\}$.*
La restricción-producto de $\{r\}$ por el cubo q^i, *representado por $\langle q^i \odot \{r\}\rangle$, es un conjunto de cubos definido como*

$$\langle q^i \odot \{r\}\rangle = \odot_{j=0}^{l-1} \langle q^i | r^j\rangle = \langle q^i | r^0\rangle \odot \ldots \odot \langle q^i | r^{l-1}\rangle \tag{3.5}$$

Si algún $\langle q^i | r^j\rangle = \emptyset$, entonces éste no se incluye en el producto dado en la ecuación anterior. Si las restricciones $\langle q^i | q^j\rangle = \emptyset$ para todo j, entonces la restricción-producto será el cubo q^i.
La restricción-producto de $\{r\}$ por $\{q\}$, *representado por $\langle \{q\} \odot \{r\}\rangle$, es un conjunto de cubos definido como*

$$\langle \{q\} \odot \{r\}\rangle = \bigcup_{i=0}^{k-1} \langle q^i \odot \{r\}\rangle \tag{3.6}$$

Según la definición 7, la *restricción-producto* de un conjunto de cubos $\{r\}$ por el cubo q^i produce los cubos incluidos en q^i distintos de $\{r\}$, es decir, el complemento de $\{r\}$ con respecto a q^i. Si el conjunto de cubos $\{r\}$ no está incluido en q^i, entonces la restricción-producto es q^i. La *restricción-producto* de $\{r\}$ por $\{q\}$ produce, por tanto, los cubos complementados de $\{r\}$ incluidos en el conjunto de cubos $\{q\}$.

Ejemplo 3 *La figura 3.2 muestra una representación de la restricción-producto del conjunto de cubos $\{r\} = \{r^0, r^1, r^2, r^3\}$ por $\{q\} = \{q^0, q^1, q^2\}$, donde cada rectángulo representa un cubo. Las áreas ralladas son los resultados de la restricción-producto.* □

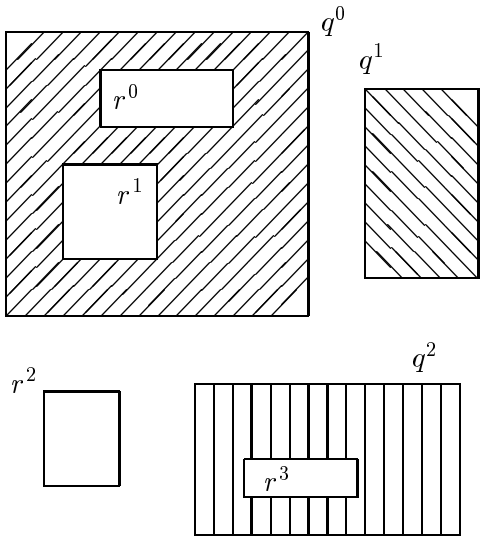


Figura 3.2: Restricción-producto de $\{r\}$ por $\{q\}$.

3.3. Nueva formulación para el cálculo de operaciones Booleanas

Utilizando las definiciones dadas en la sección anterior, a continuación se introducen una serie de teoremas [IS01] que constituyen la base de nuestro método de cálculo de operaciones Booleanas.

Teorema 10 (Conjunción) *Sean f y g dos funciones Booleanas dadas por sus conjuntos de cubos $\{f\}$ y $\{g\}$. Entonces, el conjunto de cubos de la conjunción de f y g, representado como $\{f \wedge g\}$, es*

$$\{f \wedge g\} \quad = \quad \{f\} \odot \{g\} \tag{3.7}$$

donde $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$ y $\{g\} = \{g^0, g^1, \ldots, g^{m-1}\}$.

Demostración 1 *Sea n la cardinalidad del conjunto X de variables de entrada. Los cubos f^i y g^j se pueden expresar por medio de sus componentes $f^i = (f^i_{n-1}, \ldots, f^i_1, f^i_0)$ y $g^j = (g^j_{n-1}, \ldots, g^j_1, g^j_0)$, por lo que el producto será $f^i \odot g^j = (f^i_{n-1} \odot g^j_{n-1}, \ldots, f^i_1 \odot g^j_1, f^i_0 \odot g^j_0)$. Cada uno de estos productos componentes puede tomar el valor 0, 1 o d, dependiendo de los valores de los componentes (véase la tabla 3.1). Si dos componentes son iguales, el producto resultante es el valor de los componentes y la coordenada del espacio Booleano permanece inalterada. Si alguno de los componentes es d, el producto resultante será el valor del otro componente, por lo que la coordenada del producto toma el valor de la componente que es distinta de d, es decir, se obtiene la*

'intersección' o el elemento común a ambos componentes. Si el producto es $\emptyset$, esto significa que los componentes tienen valores distintos (no comunes) para la misma coordenada del espacio Booleano y ésto implica que los cubos no tienen un espacio Booleano común. Considerando un cubo como un conjunto de estos componentes, el producto de dos cubos produce su cubo común (si existe), es decir, su 'intersección'. Por lo tanto, el producto de los conjuntos de cubos de dos funciones da el conjunto de cubos comunes, que es el conjunto de cubos de la conjunción de las funciones. □

Teorema 11 (Complemento) *Sea f una función Booleana expresada con el conjunto de cubos $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$. Entonces, el conjunto de cubos del complemento de f, representado como $\{\overline{f}\}$, es*

$$\{\overline{f}\} = \langle D \odot \{f\} \rangle \tag{3.8}$$

donde $D = (d, d, \ldots, d)$ es el cubo con todos sus componentes iguales a d, siendo n la cardinalidad del conjunto de variables de entrada.

Demostración 2 *El cubo D representa el espacio Booleano completo, y el conjunto de cubos $\{f\}$ está incluido en D, como requiere la definición 7. Por lo tanto, la restricción-producto de $\{f\}$ por D produce el conjunto de cubos incluidos en D distintos de $\{f\}$, que es el conjunto de cubos del complemento de la función f.* □

Teorema 12 (Disyunción) *Sean f y g dos funciones Booleanas dadas por sus conjuntos de cubos $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$ y $\{g\} = \{g^0, g^1, \ldots, g^{m-1}\}$, respectivamente. Entonces, el conjunto de cubos de la disyunción de f y g, representado como $\{f \vee g\}$, es*

$$\{f \vee g\} = (\langle D \odot \{f\} \rangle \odot \{g\}) \cup \{f\} \tag{3.9}$$

donde el operando derecho $\{f\}$ que aparece en la operación de restricción-producto se selecciona arbitrariamente, es decir, se podría haber seleccionado la función g para realizar la operación de restricción-producto, en cuyo caso, la expresión anterior sería equivalente a aquella en la que se intercambiaran las funciones f y g.

Demostración 3 *La restricción-producto de $\{f\}$ por D es el conjunto de cubos de $\overline{f}$ por el teorema 11, y el producto del conjunto de cubos $\langle D \odot \{f\} \rangle$ con el conjunto de cubos $\{g\}$ da los cubos comunes a $\langle D \odot \{f\} \rangle$ y $\{g\}$, es decir, el conjunto de cubos $\{g\}$ menos el conjunto de cubos de su intersección con $\{f\}$. Finalmente, la unión de este conjunto de cubos obtenido con $\{f\}$, nos da la unión de $\{f\}$ y $\{g\}$ menos los cubos del producto (intersección), que es el conjunto de cubos de la disyunción de f y g (descartando los cubos repetidos).*

También se tiene que utilizando expresiones Booleanas, la ecuación anterior se puede expresar como $f + g = \overline{f} \cdot g + f$ *(donde* $+$*,* $\cdot$ *y la línea horizontal superior representan la disyunción, conjunción y el complemento Booleano, respectivamente), que es la identidad de simplificación del algebra Booleana.*
Todas estas consideraciones también serían válidas si se intercambiaran las funciones f *y* g*.* □

En la figura 3.3 se observa una representación de la disyunción de $\{f\}$ y $\{g\}$, donde los cubos se representan mediante rectángulos. El espacio Booleano completo está representado por el cubo D, que incluye todos los cubos. El complemento de f ($\langle D \odot \{f\}\rangle$) es el espacio Booleano distinto de los cubos de f y se representa como un área rallada inclinada. La operación $\langle D \odot \{f\}\rangle \odot \{g\}$ se representa por el área rallada horizontal resultante de la intersección del complemento de f y del conjunto de cubos $\{g\}$. Finalmente, la disyunción se obtiene por la unión de los cubos resultantes de esta última operación con el conjunto de cubos $\{f\}$.

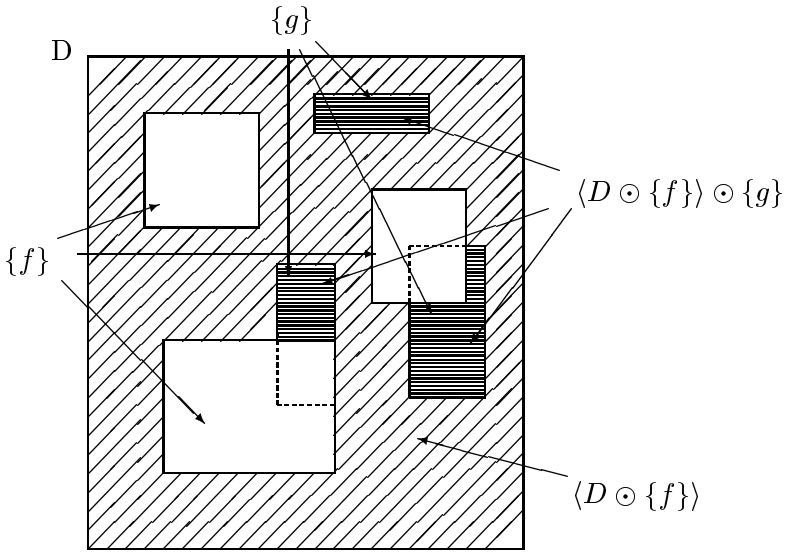


Figura 3.3: Representación de la disyunción de f y g

En el siguiente corolario se da otro enunciado del teorema 12.

Corolario 1 (Disyunción) *Sean* f *y* g *dos funciones Booleanas dadas por sus conjuntos de cubos* $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$ *y* $\{g\} = \{g^0, g^1, \ldots, g^{m-1}\}$*, respectivamente. Entonces, el conjunto de cubos de la disyunción de* f *y* g*, representado como* $\{f \vee g\}$*, es*

$$\{f \vee g\} = \langle \{f\} \odot \{f \wedge g\}\rangle \cup \{g\} \tag{3.10}$$

donde el operando izquierdo $\{f\}$ en la operación restricción-producto se selecciona de forma arbitraria, es decir, se podría haber seleccionado la función g para la realización de la operación restricción-producto, en cuyo caso la fórmula anterior sería equivalente a aquella en la que se hubieran intercambiado las funciones f y g.

Demostración 4 *$\{f \wedge g\}$ es el conjunto de cubos de la intersección de f y g, por lo que este conjunto se encuentra incluido en $\{f\}$, como requiere la definición 7. También de la definición 7 se tiene que la restricción-producto de $\{f \wedge g\}$ por $\{f\}$ da los cubos de $\{f\}$ que no están incluidos en la intersección $\{f \wedge g\}$. Finalmente, la unión con $\{g\}$ da los cubos de $\{f\}$ y $\{g\}$ menos los cubos de la intersección, es decir, el conjunto de cubos de la disyunción de f y g (donde se descartan los cubos repetidos).*
Si se intercambiaran f y g, se tendrían consideraciones similares. □

Generalmente, la operación disyunción realizada utilizando el corolario 1 será más sencilla que utilizando el teorema 12 porque las operaciones que aparecen en el teorema 12 involucran al espacio Booleano completo (representado como D), mientras que las operaciones que aparecen en el corolario 1 se restringen a $\{f\}$ y su intersección con $\{g\}$.

Teorema 13 (Disyunción exclusiva) *Sean f y g dos funciones Booleanas expresadas por sus conjuntos de cubos $\{f\}$ y $\{g\}$. El conjunto de cubos de la disyunción exclusiva de f y g, representado como $\{f \oplus g\}$, es*

$$\{f \oplus g\} = (\langle D \odot \{f\}\rangle \odot \{g\}) \cup (\langle D \odot \{g\}\rangle \odot \{f\}) \tag{3.11}$$

donde $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$ y $\{g\} = \{g^0, g^1, \ldots, g^{m-1}\}$.

Demostración 5 *En la ecuación 3.11, el término izquierdo de la unión representa el conjunto de cubos $\{g\}$ menos el conjunto de cubos de la intersección con $\{f\}$. El término derecho de la unión representa el conjunto de cubos $\{f\}$ menos el conjunto de cubos de su intersección con $\{g\}$. Por tanto, la unión de estos conjuntos de cubos proporciona los cubos incluidos en $\{f\}$ y $\{g\}$ excepto aquellos de la intersección de $\{f\}$ y $\{g\}$, que son los cubos de la disyunción exclusiva de f y g.*
Utilizando el álgebra Booleana, se tiene también que la ecuación 3.11 se puede expresar como $f \oplus g = \overline{f} \cdot g + f \cdot \overline{g}$ (donde $+$, $\cdot$ y la línea horizontal superior representan la disyunción, conjunción y el complemento Booleano, respectivamente), que es la definición de la OR exclusiva (XOR) de las funciones f y g. Todas estas consideraciones también serían válidas si se realizara el intercambio de las funciones f y g. □

En la figura 3.4 se representa gráficamente la disyunción exclusiva de los conjuntos de cubos $\{f\}$ y $\{g\}$ utilizados en la figura 3.3. Los cubos resultantes se representan por áreas ralladas horizontales y verticales sobre el espacio Booleano D, cuya unión proporciona la XOR de las funciones f y g.

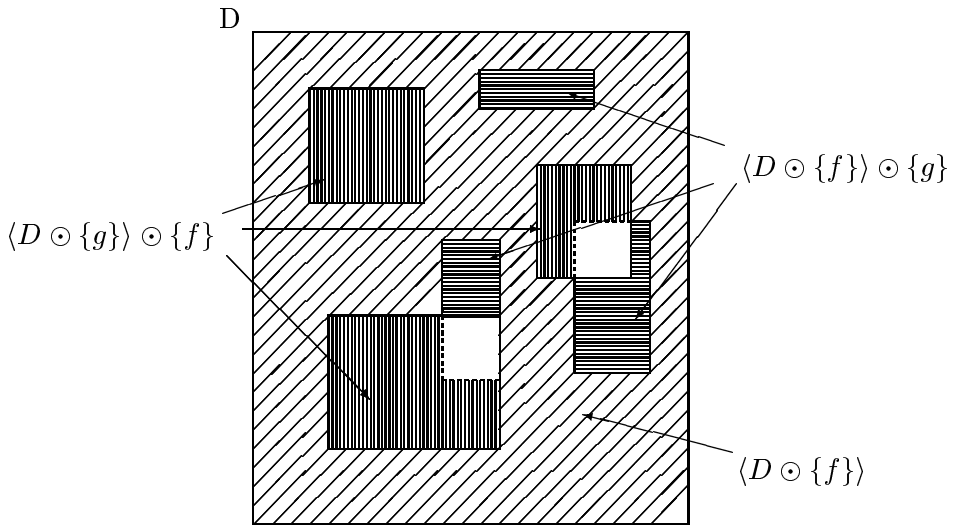


Figura 3.4: Representación de la disyunción exclusiva de f y g

En el siguiente corolario se da otro enunciado equivalente del teorema 13.

Corolario 2 (Disyunción exclusiva) *Sean f y g dos funciones Booleanas dadas por sus conjuntos de cubos $\{f\}$ y $\{g\}$. El conjunto de cubos de la disyunción exclusiva de f y g, representado como $\{f \oplus g\}$, es*

$$\{f \oplus g\} = (\langle\{f\} \odot \{f \wedge g\}\rangle \cup (\langle\{g\} \odot \{f \wedge g\}\rangle \tag{3.12}$$

donde $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$ y $\{g\} = \{g^0, g^1, \ldots, g^{m-1}\}$.

Demostración 6 *En la ecuación 3.12, el término izquierdo representa la unión de los cubos de $\{f\}$ no incluidos en la intersección $\{f \wedge g\}$, y el término derecho representa los cubos de $\{g\}$ no incluidos en $\{f \wedge g\}$. Por tanto, la unión de estos conjuntos de cubos proporciona los cubos incluidos en $\{f\}$ y $\{g\}$ excepto aquellos de la intersección de $\{f\}$ y $\{g\}$, que son los cubos de la disyunción exclusiva de f y g.* □

Utilizando los mismos argumentos que en el caso de la disyunción, se tiene que la disyunción exclusiva aplicando el corolario 2 será, generalmente, más eficiente que aplicando el teorema 13.

A continuación vemos un ejemplo que ilustra cómo opera el método utilizando los teoremas y corolarios anteriores.

3.4. Ejemplo

Sean f y g dos funciones Booleanas con conjuntos de cubos $\{f\}$ y $\{g\}$. Sea $X = \{x_4, x_3, x_2, x_1, x_0\}$. Supónganse los siguientes conjuntos de cubos para ambas funciones:

$$\begin{array}{lcll} \{f\} & = & \{f^0, f^1, f^2\} & = & \{dd0d1, d11dd, 1011d\} \\ \{g\} & = & \{g^0, g^1, g^2\} & = & \{ddd01, 0011d, d1dd0\} \end{array}$$

Las operaciones Booleanas sobre estos conjuntos de cubos se pueden realizar como se indica a continuación.

3.4.1. Conjunción

Se aplica el teorema 10 para obtener el conjunto de cubos de la conjunción de f y g, $\{f \wedge g\}$

$$\{f \wedge g\} = \{f\} \odot \{g\}$$

$$\begin{array}{lcll} (ddd01) & \odot & (dd0d1) & = & (dd001) \\ (ddd01) & \odot & (d11dd) & = & (d1101) \\ (ddd01) & \odot & (1011d) & = & (101\emptyset\ ) \quad \text{producto nulo} \\ (0011d) & \odot & (dd0d1) & = & (00\emptyset\ ) \quad \text{producto nulo} \\ (0011d) & \odot & (d11dd) & = & (0\emptyset\ ) \quad \text{producto nulo} \\ (0011d) & \odot & (1011d) & = & (\emptyset\ ) \quad \text{producto nulo} \\ (d1dd0) & \odot & (dd0d1) & = & (d10d\emptyset) \quad \text{producto nulo} \\ (d1dd0) & \odot & (d11dd) & = & (d11d0) \\ (d1dd0) & \odot & (1011d) & = & (1\emptyset\ ) \quad \text{producto nulo} \end{array}$$

Se obtienen, por lo tanto, los siguientes cubos para la conjunción de ambas funciones:

$$\{f \wedge g\} = \{dd001, d1101, d11d0\}$$

3.4.2. Disyunción

3.4.2.1. Aplicación del teorema

Aplicamos el teorema 12 y seleccionamos arbitrariamente $\{f\}$ para la realización de la operación de restricción-producto

$$\{f \vee g\} = (\langle D \odot \{f\}\rangle \odot \{g\}) \cup \{f\}$$

$$\langle D \odot \{f\}\rangle = \langle D|f^0\rangle \odot \langle D|f^1\rangle \odot \langle D|f^2\rangle$$

- $\langle D|f^0\rangle = \langle (ddddd)|(dd0d1)\rangle = \{dd1dd, dd0d0\}$

- $\langle D|f^1\rangle = \langle(ddddd)|(d11dd)\rangle = \{d0ddd, d10dd\}$
- $\langle D|f^2\rangle = \langle(ddddd)|(1011d)\rangle = \{0dddd, 11ddd, 100dd, 1010d\}$
- $\langle D|f^0\rangle \odot \langle D|f^1\rangle = \{d01dd, d00d0, d10d0\}$
- $(\langle D|f^0\rangle \odot \langle D|f^1\rangle) \odot \langle D|f^2\rangle = \{001dd, 1010d, 000d0, 100d0, 010d0, 110d0\}$

$$\langle D \odot \{f\}\rangle \odot \{g\} = \{00101, 0011d, 10101, 010d0, 110d0\}$$

Finalmente, $\{f \vee g\} = (\langle D \odot \{f\}\rangle \odot \{g\}) \cup \{f\}$, por lo que se tiene que

$$\{f \vee g\} = \{00101, 0011d, 10101, 010d0, 110d0, dd0d1, d11dd, 1011d\}$$

3.4.2.2. Aplicación del corolario

También se puede aplicar el corolario 1 y seleccionar $\{f\}$ arbitrariamente para la realización de la operación restricción-producto. En este caso se debe calcular

$$\langle\{f\} \odot \{f \wedge g\}\rangle = \langle f^0 \odot \{f \wedge g\}\rangle \cup \langle f^1 \odot \{f \wedge g\}\rangle \cup \langle f^2 \odot \{f \wedge g\}\rangle$$

- $\langle f^0 \odot \{f \wedge g\}\rangle = \langle f^0|(dd001)\rangle = \{dd011\}$
  donde las restricciones $\langle f^0|(d1101)\rangle$ y $\langle f^0|(d11d0)\rangle$ no aparecen en el producto anterior porque ambas restricciones son $\emptyset$.
- $\langle f^1 \odot \{f \wedge g\}\rangle = \langle f^1|(d1101)\rangle \odot \langle f^1|(d11d0)\rangle = \{d1111\}$
  donde la restricción $\langle f^1|(dd001)\rangle$ no aparece en el producto porque es el conjunto vacío $\emptyset$ y donde el resultado se ha calculado realizando el producto de $\langle f^1|(d1101)\rangle = \{d111d, d1100\}$ con $\langle f^1|(d11d0)\rangle = \{d11d1\}$.
- $\langle f^2 \odot \{f \wedge g\}\rangle = f^2 = \{1011d\}$
  donde todas las restricciones $\langle f^2|(dd001)\rangle$, $\langle f^2|(d1101)\rangle$ y $\langle f^2|(d11d0)\rangle$ son $\emptyset$, por lo que el resultado es el cubo f^2.

Se obtiene, por tanto,

$$\langle\{f\} \odot \{f \wedge g\}\rangle = \{dd011, d1111, 1011d\}$$

y finalmente

$$\{f \vee g\} = (\langle\{f\} \odot \{f \wedge g\}\rangle \cup \{g\} = \{dd011, d1111, 1011d, ddd01, 0011d, d1dd0\}$$

Se puede observar que este resultado es más reducido que el obtenido utilizando el teorema 12.

3.4.3. Disyunción exclusiva

3.4.3.1. Aplicación del teorema

Aplicamos el teorema 13

$$\{f \oplus g\} = (\langle D \odot \{f\}\rangle \odot \{g\}) \cup (\langle D \odot \{g\}\rangle \odot \{f\})$$

$$\langle D \odot \{g\}\rangle = \langle D|g^0\rangle \odot \langle D|g^1\rangle \odot \langle D|g^2\rangle$$

- $\langle D|g^0\rangle = \langle (ddddd)|(ddd01)\rangle = \{ddd1d, ddd00\}$
- $\langle D|g^1\rangle = \langle (ddddd)|(0011d)\rangle = \{1dddd, 01ddd, 000dd, 0010d\}$
- $\langle D|g^2\rangle = \langle (ddddd)|(d1dd0)\rangle = \{d0ddd, d1dd1\}$
- $\langle D|g^0\rangle \odot \langle D|g^1\rangle = \{1dd1d, 01d1d, 0001d, 1dd00, 01d00, 00000, 0010d\}$
- $(\langle D|g^0\rangle \odot \langle D|g^1\rangle) \odot \langle D|g^2\rangle =$
  $\{10d1d, 11d11, 01d11, 0001d, 10d00, 00000, 0010d\}$
- $\langle D \odot \{g\}\rangle \odot \{f\} = \{10011, 1011d, 11011, 11111, 01011, 01111, 00011\}$

Finalmente, utilizando el resultado obtenido anteriormente en el cálculo de la disyunción de f y g aplicando el teorema 12, se tiene que

$$\begin{aligned}\{f \oplus g\} \quad = \quad & \{00101, 0011d, 10101, 010d0, 110d0, 10011, 1011d, 11011,\\ & 11111, 01011, 01111, 00011\}\end{aligned}$$

3.4.3.2. Aplicación del corolario

Aplicamos el corolario 2 para obtener la disyunción exclusiva de f y g

$$\{f \oplus g\} = (\langle \{f\} \odot \{f \wedge g\}\rangle \cup (\langle \{g\} \odot \{f \wedge g\}\rangle$$

$$\langle \{g\} \odot \{f \wedge g\}\rangle = \langle g^0 \odot \{f \wedge g\}\rangle \cup \langle g^1 \odot \{f \wedge g\}\rangle \cup \langle g^2 \odot \{f \wedge g\}\rangle$$

- $\langle g^0 \odot \{f \wedge g\}\rangle = \langle g^0|(dd001)\rangle \odot \langle g^0|(d1101)\rangle = \{d0101\}$
  donde la restricción $\langle g^0|(d11d0)\rangle$ no aparece en el producto anterior porque es $\emptyset$, y donde el resultado se ha calculado realizando el producto de $\langle g^0|(dd001)\rangle = \{dd101\}$ con $\langle g^0|(d1101)\rangle = \{d0d01, d1001\}$
- $\langle g^1 \odot \{f \wedge g\}\rangle = g^1 = \{0011d\}$
  donde todas las restricciones $\langle g^1|(dd001)\rangle$, $\langle g^1|(d1101)\rangle$ y $\langle g^1|(d11d0)\rangle$ son $\emptyset$, por lo que el resultado es el cubo g^1.
- $\langle g^2 \odot \{f \wedge g\}\rangle = \langle g^2|(d11d0)\rangle = \{d10d0\}$
  donde las restricciones $\langle g^2|(dd001)\rangle$ y $\langle g^2|(d1101)\rangle$ no aparecen en el producto anterior porque ambas son $\emptyset$.

Se tiene, por tanto,

$$\langle \{g\} \odot \{f \wedge g\} \rangle = \{d0101, 0011d, d10d0\}$$

Utilizando el resultado obtenido anteriormente en el cálculo de la disyunción de f y g aplicando el corolario 1, se tiene finalmente

$$\{f \oplus g\} = \{dd011, d1111, 1011d, d0101, 0011d, d10d0\}$$

Se puede observar que el conjunto de cubos obtenido aplicando el corolario 2 es más reducido (y las operaciones realizadas más simples) que el obtenido utilizando el teorema 13.

3.4.4. Complemento

Aplicamos el teorema 11

$$\{\overline{f}\} = \langle D \odot \{f\} \rangle$$

Esta operación ya ha sido calculada anteriormente para las funciones f y g, por lo que $\{\overline{f}\}$ y $\{\overline{g}\}$ serán

$$\begin{array}{lcl} \{\overline{f}\} & = & \{001dd, 1010d, 000d0, 100d0, 010d0, 110d0\} \\ \{\overline{g}\} & = & \{10d1d, 11d11, 01d11, 0001d, 10d00, 00000, 0010d\} \end{array}$$

3.5. Resultados experimentales

Se han implementado en código C las operaciones de disyunción dadas por el teorema 12 y el corolario 1, utilizándose como *benchmarks* (*circuitos de prueba*) los circuitos LGSynth91[2] dados en un formato de dos niveles en forma de suma de productos (SOP) y que constituye un conjunto de cubos *no disjuntos*. Los programas que implementan la disyunción utilizando el teorema 12 y el corolario 1 convierten los archivos de dos niveles de los benchmarks (en formato PLA[3]) en archivos en forma de suma de productos disjuntos (DSOP). Los programas leen las entradas de los archivos PLA secuencialmente y generan el archivo en forma DSOP correspondiente a ese circuito.

En la tabla 3.2 se muestran las características de los circuitos de benchmark LGSynth91 utilizados en los experimentos. El formato de datos original de los benchmarks viene dado en formato PLA de dos niveles, en forma de SOPs. En la tabla se indica el número de entradas y salidas de cada benchmark, así como el número de productos no disjuntos dados por el circuito (*SOPs* en la tabla).

[2] *http : //www.cbl.ncsu.edu/CBL_Docs/lgs91.html*

[3] Formato de descripción de circuitos que utiliza expresiones de dos niveles y que está orientado a implementaciones sobre estructuras regulares y compactas llamadas *Arrays Lógicos Programables* o PLAs (*Programmable Logic Arrays*).

Circuito	Entradas	Salidas	SOPs
5xp1	7	10	75
9sym	9	1	87
add6	12	7	1092
alu1	12	8	19
alu2	10	8	87
apex4	9	19	1732
apex6	135	99	657
co14	14	1	47
count	35	16	184
duke2	22	29	242
e64	65	65	65
ex1010	10	10	1471
ex5	8	63	7620
exp	8	18	297
gary	15	11	221
rd53	5	3	32
rd73	7	3	141
rd84	8	4	411
sao2	10	4	78
sym10	10	1	837
t481	16	1	481
vg2	25	8	110
z5xp1	7	10	576
z9sym	9	1	420

Tabla 3.2: Características de los circuitos LGSynth91

Se han comparado los resultados producidos por los programas correspondientes al teorema 12 y al corolario 1 con los resultados obtenidos por el programa DJ [KDS95][Cha00], que transforma también las SOPs no disjuntas inicialmente dadas para un circuito en expresiones DSOP disjuntas. Se ha comparado el tiempo de ejecución necesario para el cálculo y el número de productos disjuntos de las sumas obtenidos por los tres programas (teorema 12, corolario 1 y programa DJ) en una estación de trabajo SUN UltraSparc-II.

En la tabla 3.3 se muestra el número de términos producto disjuntos obtenidos utilizando el teorema 12 (DSOPs-th12), el corolario 1 (DSOPs-cor1 en la tabla) y el programa DJ (DSOPs-DJ), para los benchmarks LGSynth91 dados. De los resultados obtenidos en la tabla, se puede observar que el número de productos disjuntos calculados utilizando el corolario 1 es mucho menor que el obtenido utilizando el teorema 12 (a excepción de los circuitos *alu1, vg2* y *sao*2), como se indicó anteriormente en la sección 3.3 y como se puede comprobar considerando el número total de productos disjuntos medidos para los

Circuito	DSOPs-th12	DSOPs-cor1	DSOPs-DJ
5xp1	140	75	98
9sym	307	194	199
add6	1092	1092	1092
alu1	19	23	19
alu2	276	138	241
apex4	2032	1820	1749
apex6	22381	759	2503
co14	14	14	14
count	263004	184	488
duke2	823	333	266
e64	65	65	65
ex1010	1471	1471	8670
ex5	7620	7620	7620
exp	297	297	837
gary	6772	228	390
rd53	37	32	32
rd73	163	141	146
rd84	411	411	411
sao2	212	248	147
sym10	837	837	837
t481	11952	841	5168
vg2	1225	1399	283
z5xp1	576	576	576
z9sym	420	420	420
Total	322146	19218	32271

Tabla 3.3: Comparación del número de productos disjuntos obtenidos utilizando el teorema 12, el corolario 1 y el programa DJ

benchmarks utilizados en ambos casos. Con respecto a la comparación entre los resultados obtenidos por el corolario 1 y el programa DJ, se encuentra que el corolario 1 proporciona mejores resultados en general. Considerando los valores totales medidos para los benchmarks utilizados, se tiene que el número total de términos producto disjuntos obtenidos usando el corolario 1 es un 40 % menor que el obtenido usando el programa DJ. En algunos de los benchmarks, como en $t481$ o $ex1010$, el número de productos disjuntos obtenidos por el corolario 1 es significativamente menor que el obtenido por DJ (concretamente, un 83.7 % y un 83 % menor, respectivamente).

En la tabla 3.4 se muestran los tiempos de ejecución para los benchmarks dados utilizando los tres programas, donde $Tth12$, $Tcor1$ y TDJ representan el tiempo (en segundos) necesario de generación de los productos disjuntos para los programas que implementan el teorema 12, el corolario 1 y para el

Circuito	Tth12	Tcor1	TDJ
5xp1	0.01	0.00	0.05
9sym	0.28	0.05	0.08
add6	6.14	0.67	0.81
alu1	0.00	0.00	0.03
alu2	0.03	0.01	0.10
apex4	3.41	0.46	1.07
apex6	50.36	14.47	2.07
co14	0.01	0.00	0.04
count	55.69	0.30	0.14
duke2	0.24	0.06	0.14
e64	0.03	0.02	0.07
ex1010	3.27	0.42	8.49
ex5	38.36	6.01	3.21
exp	0.11	0.04	0.19
gary	5.86	0.04	0.13
rd53	0.00	0.00	0.02
rd73	0.09	0.03	0.07
rd84	0.77	0.16	0.17
sao2	0.06	0.07	0.08
sym10	10.36	1.15	1.11
t481	150.09	2.59	3.39
vg2	0.86	1.93	0.11
z5xp1	0.37	0.08	0.12
z9sym	2.37	0.29	0.31
Total	328.77	28.85	22.0

Tabla 3.4: Comparación de tiempos de ejecución obtenidos utilizando el teorema 12, el corolario 1 y el programa DJ

programa DJ, respectivamente. De los resultados obtenidos, se observa que el teorema 12 consume mucho más tiempo que el corolario 1 en prácticamente todos los benchmarks utilizados, especialmente en el circuito $t481$. Con respecto a la comparación de los tiempos producidos por el corolario 1 y por el programa DJ, se tiene que, en general, el corolario 1 consume menos tiempo que DJ, especialmente en $ex1010$ (donde $Tcor1$ es un 95 % menor que TDJ). Sin embargo, existen dos benchmarks ($ex5$ y, especialmente, $apex6$) para los que el programa DJ invierte un tiempo mucho menor que el tiempo necesitado por el programa que implementa el corolario 1. El comportamiento en estos dos circuitos, $ex5$ y $apex6$, hace que el tiempo total de ejecución para todos los benchmarks sea menor para el programa DJ que para el corolario 1 (concretamente, un 23.7 % menor), a pesar de que en 19 de los 24 benchmarks probados, el corolario 1 es más rápido que el programa DJ.

Para tratar de decidir si existe alguna diferencia en los tiempos de ejecución utilizando el corolario 1 y el programa DJ, se ha realizado un estudio estadístico de los datos temporales medidos a través de un *diseño factorial completo de dos factores sin duplicación* (*Two-factor full factorial design without replication*) [Jai91], en el que los dos programas, corolario 1 y DJ, constituyen uno de los *factores*, mientras que los benchmarks utilizados constituyen el otro *factor*. Utilizando el *análisis de la varianza* (ANOVA) y para un *nivel de significación* $\alpha = 0.05$, se ha encontrado que el *F-ratio* para los programas es menor que el *F-ratio* obtenido de la tabla de la distribución $F(n, m)$ o distribución de *Snedecor*. Consecuentemente, se concluye que la selección del programa (corolario 1 o DJ) no tiene un impacto significativo en el rendimiento, medido en este caso por la métrica *tiempo de ejecución para los benchmarks probados*, es decir, que ambos programas tienen el mismo comportamiento con respecto de la métrica seleccionada (tiempo de ejecución) y no se puede considerar que uno de los programas sea mejor que el otro. Para una mejor comprensión de este método estadístico, remitimos al libro de Jain referenciado [Jai91].

Por lo tanto, considerando como métricas de comparación el número de términos producto disjuntos obtenido y el tiempo consumido de ejecución, se puede concluir que, en general, el programa que implementa el corolario 1 presenta un mejor rendimiento que el programa DJ, ya que genera un menor número de productos disjuntos y presenta un comportamiento temporal similar al del programa DJ.

3.6. Conclusiones

En este capítulo se ha presentado un nuevo método para la operación de funciones Booleanas cuando estas funciones están representadas en un formato de dos niveles como sumas de productos disjuntos, obteniéndose como resultado de las operaciones unas representaciones que vienen también dadas en forma DSOP. Los resultados experimentales han demostrado que cuando se tienen representaciones en formato SOP que no cumplen la propiedad de disjunción, entonces se puede aplicar el método para la obtención de una representación en formato DSOP. La comparación de nuestro método con otra técnica de creación de representaciones DSOP ha probado el mejor rendimiento presentado por nuestra aproximación. Estos resultados podrían incluso mejorarse si se utilizara algún algoritmo de selección de los cubos a operar, ya que como se vio en los teoremas 12 y 13 y en los corolarios 1 y 2, es posible el intercambio de las funciones a operar que aparecen involucradas en las ecuaciones correspondientes. La misma consideración se puede hacer con respecto del orden de utilización de los cubos de cada una de las funciones en cada una de las operaciones vistas. Este tipo de estudios algorítmicos constituye uno de nuestros futuros trabajos de investigación.

Como se vio en el capítulo 2, la propiedad de *disjunción* es muy importante cuando se pretende calcular la signatura correspondiente a una función Booleana por medio de la evaluación de su transformada algebraica. La propiedad de disjunción permite la simplificación del cálculo de dicha signatura. Por este motivo, utilizaremos la nueva formulación para el cálculo de operaciones Booleanas vista en este capítulo para su aplicación a una aproximación *híbrida* de verificación de circuitos combinacionales representados en un formato de dos niveles y que se verá en el capítulo 5.

En la aproximación *híbrida* de verificación se pretenden combinar los métodos de verificación *probabilista* y *determinista* (que utiliza ROBDDs clásicos). Por este motivo, en el siguiente capítulo se describen los *diagramas de decisión binarios ordenados y reducidos* ROBDDs como estructuras de representación canónica de funciones Booleanas, así como el problema de la ordenación de variables y su influencia en el tamaño de los ROBDDs.

Capítulo 4

Representación de funciones Booleanas por medio de grafos

En este capítulo se describen las estructuras de datos en forma de grafos más utilizadas para la representación de funciones Booleanas, los Diagramas de decisión binarios ordenados y reducidos (ROBDDs), así como otros tipos de grafos también usados para representación. Se trata el problema de la ordenación de variables de entrada para los diagramas de decisión y se presenta un algoritmo de ordenación heurístico.

La manipulación de funciones discretas es una cuestión fundamental para el diseño y test de circuitos digitales ya que muchos de sus problemas se pueden expresar como una secuencia de operaciones sobre funciones discretas. Con los recientes avances de la tecnología VLSI y la cada vez mayor capacidad de integración, el diseño *manual* de circuitos es ya muy poco utilizado salvo para aplicaciones muy concretas, y los sistemas CAD para diseño VLSI son ampliamente utilizados. El rendimiento de estos sistemas automáticos depende en gran medida de la eficiencia de las estructuras de datos que se utilicen para la representación de las funciones discretas. En general, incluso las tareas más sencillas como la comprobación de si una expresión Booleana es una *tautología* ($f = 1$) o la comprobación de si dos expresiones Booleanas representan la misma función, requieren la solución de problemas NP-*completo*[1]. En el peor de los casos, cualquiera de estos problemas puede necesitar soluciones de coste exponencial para cualquier representación lógica. Sin embargo, en muchas aplicaciones, una buena selección de la representación y de los algoritmos puede evitar estas complejidades de tipo exponencial.

[1] Estos tipos de problemas son aquellos en los que los algoritmos necesarios para su resolución presentan una complejidad no polinómica.

Se han desarrollado varios métodos para la representación y manipulación de funciones Booleanas, entre los que se encuentran las tablas de verdad, mapas de Karnaugh [Kar53], sumas de productos (SOP) canónicas, sumas de productos mínimas [McC56][SB90], sumas de productos basadas en OR-exclusiva (ESOP) [SEF93][KDS95], métodos tabulares [Rud74b][TLROL93][Ung94], representaciones en forma de *cubos* [BHMSV84][IB99][IS01], formas factorizadas, Diagramas de Decisión Binarios (BDDs) [Lee59][Ake78] y Diagramas de Decisión Binarios Ordenados y Reducidos (ROBDDs) [Bry86], entre otros.

Las tablas de verdad, los mapas de Karnaugh y las sumas de productos canónicas producen representaciones de tamaño exponencial, es decir, su tamaño es proporcional a 2^n, siendo n el número de variables de entrada. Las sumas de productos mínimas y las formas factorizadas son de las representaciones más utilizadas, dando buenos resultados para muchas aplicaciones, aunque presentan inconvenientes como son la obtención de tamaños exponenciales para ciertas funciones o que no constituyen representaciones *canónicas*, es decir, que dos expresiones Booleanas diferentes pueden representar a la misma función. Las ESOPs son representaciones que están cobrando recientemente interés debido a que sus implementaciones VLSI presentan tamaños más reducidos en ciertos casos, aunque tienen el inconveniente de que su minimización es más difícil que en el caso de las SOPs. Los métodos tabulares y las representaciones *cúbicas* son fácilmente implementables para su procesamiento en computadores debido a su compacidad y simplicidad en la aplicación de sus reglas de operación, aunque presentan problemas de evaluación y almacenamiento.

En la actualidad, los ROBDDs tienen una gran importancia debido a que proporcionan buenas soluciones a varios de los problemas mencionados. Constituyen una forma *canónica* de representación cuyo tamaño se mantiene entre límites razonables para muchas funciones de interés, por lo que permiten el desarrollo de algoritmos de manipulación Booleana rápidos y eficientes. Como inconveniente tienen que pueden ser muy dependientes de la ordenación dada a las variables de entrada, obteniéndose diferentes ROBDDs para cada caso. El efecto de la ordenación de variables depende de la naturaleza de las funciones a representar y en algunos casos el tamaño de los ROBDDs puede variar enormemente. Además, el encuentro de la mejor ordenación resulta ser un problema de tipo NP-*completo*.

4.1. Diagramas de Decisión Binarios (BDDs)

En esta sección se introducen los conceptos básicos de los BDDs así como sus técnicas de manipulación. En primer lugar, se comienza considerando la descripción de una función Booleana por medio de estructuras basadas en *grafos* [Gou88], como árboles y *grafos acíclicos dirigidos* o DAGs (*Directed Acyclic Graph*).

Definición 8 *Un Árbol de Decisión Binario o BDT (Binary Decision Tree) es un grafo dirigido, acíclico y conectado, en donde cada nodo tiene como mucho dos sucesores (apuntados por dos aristas salientes de dicho nodo) y un sólo antecesor. Existen dos tipos de nodos: no terminales (o internos) y terminales (o nodos hoja). Los nodos terminales no tienen sucesores y representan funciones Booleanas constantes. Cada nodo no terminal n_i está asociado con una variable de decisión x_i. Si $low(n_i)$ y $high(n_i)$ representan los sucesores de n_i (apuntados por la 0-arista y la 1-arista salientes de n_i, respectivamente), entonces la función Booleana representada por n_i, f^{n_i}, se puede describir por la siguiente expresión Booleana:*

$$f^{n_i} = \overline{x}_i \cdot f^{low(n_i)} \vee x_i \cdot f^{high(n_i)} \tag{4.1}$$

donde $f^{low(n_i)}$ y $f^{high(n_i)}$ son las funciones representadas por los sucesores $low(n_i)$ y $high(n_i)$, respectivamente.

Para obtener el valor de la función Booleana en un *punto* o *vértice* del espacio Booleano, se debe recorrer el árbol desde el nodo raíz hasta un nodo terminal. El camino está definido por los valores de las variables de entrada. Si la variable asociada con un nodo se evalúa a **1**, entonces se añade la rama *high* al camino, y si la variable se evalúa a **0**, entonces se toma la rama *low*.

Akers [Ake78] propuso unas reglas de simplificación para la reducción de la complejidad del BDT, transformándolo en un grafo acíclico dirigido.

Definición 9 *Dos nodos de un BDT son equivalentes si son el mismo terminal o si están asociados con la misma variable de entrada y sus sucesores "low" son equivalentes y sus sucesores "high" son equivalentes.*

Definición 10 *Un nodo n es redundante en un BDT si low(n) es equivalente a high(n).*

Definición 11 *Un Diagrama de Decisión Binario (BDD) es un grafo dirigido acíclico derivado de un BDT donde se eliminan los nodos redundantes y donde se combinan los nodos equivalentes.*

La característica principal de un BDD es que su tamaño es mucho menor que el de un BDT para muchas funciones. Por ejemplo, mientras que un BDT tiene 2^n nodos terminales, un BDD necesita únicamente dos nodos terminales para representar los valores lógicos **0** y **1**. En la figura 4.1(a) se muestra un BDT, mientras que su correspondiente BDD aparece en la figura 4.1(b), donde "*e*" indica los nodos que se han eliminado y "*c*" indica los nodos que se han combinado. En el interior de cada nodo se muestra su variable de decisión.

Akers [Ake78] presentó varios ejemplos de aplicaciones de BDDs en el análisis, simulación y test de circuitos digitales. A pesar de la eficiencia de estas estructuras de datos, permanecían problemas como la combinación de dos BDDs

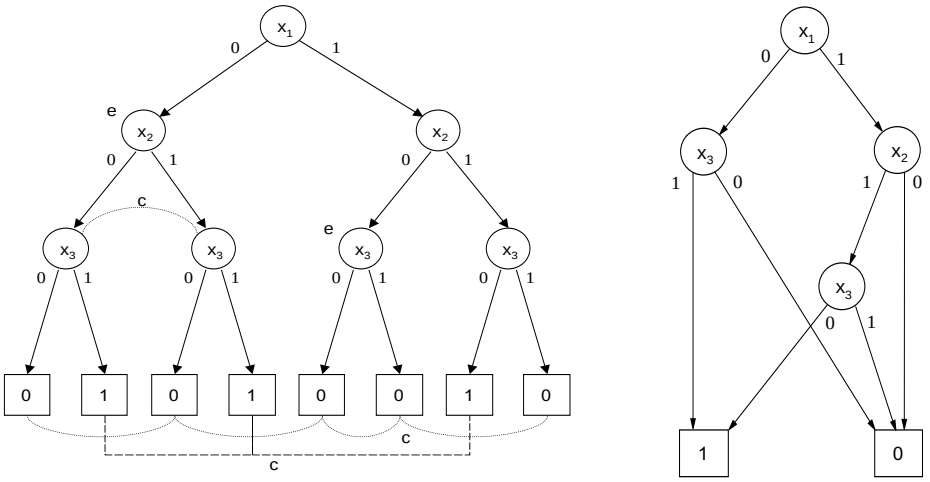


Figura 4.1: (a) Arbol de decisión binario. (b) Diagrama de decisión binario.

con los operadores lógicos, la complejidad de la equivalencia lógica o la comprobación de tautología por medio de la utilización de BDDs. Bryant [Bry86] solucionó estos problemas introduciendo la restricción de la ordenación de las variables de entrada, creando la nueva estructura llamada ROBDD.

Definición 12 *Un Diagrama de Decisión Binario Ordenado y Reducido, abreviadamente ROBDD, es un BDD en el que las variables de entrada están ordenadas de tal manera que, en todo camino del ROBDD, cada variable aparece sólo una vez y en el mismo orden relativo.*

Toda variable de entrada está asociada con un índice entero que indica su posición en la ordenación. De igual forma, cada nodo n del ROBDD tiene un atributo $index(n)$ que indica la variable con la que está asociado. El valor de $index(n)$ crece desde la raíz del ROBDD hacia las hojas del grafo, por lo que en cada camino del ROBDD los índices de los nodos deben aparecer una sola vez y en orden creciente.

Se puede calcular la función asociada con un nodo interno de un ROBDD. Sea un nodo n con $index(n) = i$, es decir, que el nodo n está asociado con la variable x_i, por lo que representamos dicho nodo como n_i. Sean también $low(n_i)$ el *0-sucesor* de n_i (es decir, el nodo al que se llega desde n_i a través de su *0-arista*) y $high(n_i)$ el *1-sucesor* de n_i (el nodo al que se llega desde n_i a través de su *1-arista*). Considerando el cálculo de la función representada por n_i, f^{n_i}, para una cierta entrada, si en la entrada el valor de x_i es 0, entonces se obtiene f^{n_i} evaluando $f^{low(n_i)}$ y si el valor de x_i es 1, entonces se obtiene f^{n_i} evaluando $f^{high(n_i)}$, es decir, $f^{low(n_i)}$ y $f^{high(n_i)}$ son las funciones que se

obtienen cuando f^{n_i} se evalúa para $x_i = 0$ y $x_i = 1$, respectivamente. Esto se representa por la siguiente expresión, que coincide con la ecuación 4.1

$$f^{n_i} = \overline{x}_i \cdot f^{low(n_i)} \vee x_i \cdot f^{high(n_i)} \tag{4.2}$$

Usando esta ecuación se pueden calcular las funciones representadas en los nodos de un ROBDD. Si un nodo n_i con $index(n) = i$ representa la función f^{n_i}, entonces el *0-sucesor* de n_i representa la subfunción (a menudo llamada *cofactor*[2]) $f^{n_i}_{\overline{x}_i}$ y el *1-sucesor* de n_i representa la subfunción $f^{n_i}_{x_i}$. En otras palabras, en n_i la función f^{n_i} se descompone por medio de la *regla de descomposición de Shannon*[3]:

$$f^{n_i} = \overline{x}_i \cdot f^{n_i}_{\overline{x}_i} \vee x_i \cdot f^{n_i}_{x_i} \tag{4.3}$$

Una de las propiedades fundamentales de un ROBDD es que constituye una representación *canónica*. Esto significa que si dos funciones Booleanas son equivalentes, entonces sus ROBDDs son *isomorfos*. La equivalencia lógica se reduce, por tanto, a la comprobación de si dos DAGs son isomorfos, que es $O(s)$, donde s es el tamaño del DAG. Otra consecuencia importante de la canonicidad de los ROBDDs es que la comprobación de tautología pasa a ser trivial: el grafo debe ser el nodo terminal **1**. El cálculo del complemento de un ROBDD consiste simplemente en el intercambio de los valores de sus terminales, que también es $O(s)$. En general, las operaciones entre dos ROBDDs son $O(s_0 \cdot s_1)$, es decir, proporcionales al producto de sus tamaños. En la figura 4.2 se muestran los grafos de algunas funciones lógicas simples.

4.1.1. Algoritmos sobre ROBDDs

Existen muchos algoritmos para la manipulación eficiente de funciones Booleanas representadas por medio de ROBDDs. Entre las operaciones básicas se pueden citar la *evaluación, reducción, equivalencia, tratabilidad, síntesis, sustitución* y las *cuantificaciones universal* y *existencial* [SF96][DS01], siendo la *síntesis* una de las operaciones más importante ya que es la utilizada para la construcción de los ROBDDs.

Los primeros algoritmos para la construcción (*síntesis*) y manipulación de ROBDDs fueron dados por Bryant en [Bry86] y se basan en dos funciones principales, conocidas como *apply* y *reduce*. En primer lugar, la función *apply* combina dos grafos utilizando un operador binario y la función *reduce*, posteriormente, lleva al ROBDD a su forma canónica eliminando todos los nodos redundantes.

[2]El *cofactor* de una función f con respecto a una variable x_i ($\overline{x}_i$) es la subfunción obtenida cuando en la función f se restringe $x_i = 1(0)$, es decir, $f(x_1, \ldots, x_i = 1(0), \ldots, x_n)$, y se representa como f_{x_i} ($f_{\overline{x}_i}$) o como $f_{x_i=1}$ ($f_{x_i=0}$).

[3]Ya hemos mencionado que esta expansión fue enunciada previamente por Boole. De todas formas, utilizaremos la terminología *expansión de Shannon* ya que es así como se conoce normalmente.

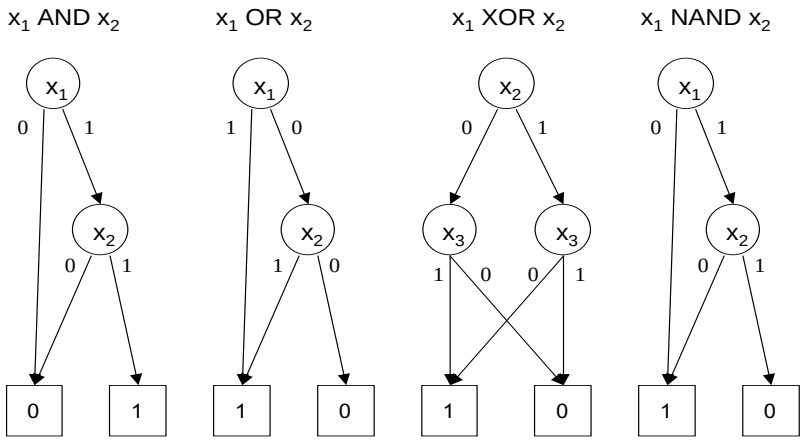


Figura 4.2: ROBDDs de algunas funciones lógicas sencillas.

Brace, Rudell y Bryant desarrollaron en [BRB90] un método de *síntesis* computacionalmente más eficiente que el de la técnica *apply/reduce*. En este método, las operaciones lógicas se basan en el operador ternario denotado como *ite* (*if-then-else*) que combina tres ROBDDs sin necesidad de la operación *reduce*. Este operador se define de la siguiente forma:

$$ite(f, g, h) = (f \cdot g) \vee (\overline{f} \cdot h) \tag{4.4}$$

Esta es la misma operación que realiza un nodo de un ROBDD. La diferencia consiste en que *ite* tiene tres funciones como parámetros, cada una de ellas descrita por un ROBDD, mientras que un nodo de un ROBDD implementa la misma función pero con el parámetro f reemplazado por una sola variable. También se puede ver que todas las operaciones Booleanas se pueden simular por el operador *ite*, como por ejemplo $f \vee g = ite(f, 1, g)$, $f \wedge g = ite(f, g, 0)$ o $f \oplus g = ite(f, \overline{g}, g)$.

La operación que realiza un nodo de un ROBDD se puede representar como $(x_i, f_{x_i}, f_{\overline{x}_i})$, donde x_i es la variable asociada con el nodo n_i, y donde f_{x_i} y $f_{\overline{x}_i}$ son los *cofactores* de la función f (es decir, la función f evaluada en $x_i = 1$ y $x_i = 0$, respectivamente). Si $z = ite(f, g, h)$ y v es la variable *superior* de f, g y h (es decir, v es la variable con menor índice de las tres funciones, o la variable que ocupa una posición más elevada en la ordenación de variables), entonces el ROBDD de z se puede construir por medio de la utilización de la siguiente expresión recursiva:

$$z = (v, ite(f_v, g_v, h_v), ite(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \tag{4.5}$$

donde los casos terminales para esta recursión son $ite(1, f, g) = ite(0, g, f) = ite(f, 1, 0) = f$. En la figura 4.3 se muestra el pseudocódigo correspondiente al algoritmo *ite*.

```
ite(f,g,h) {
   if (terminal)
     return resultado;
   else {
     sea v la variable superior de (f,g,h);
     T = ite(f_v,g_v,h_v);
     E = ite(f_v̄,g_v̄,h_v̄);
     if (T = E) return T;
     R = nuevonodo(v,T,E);
     return R;
   }
}
```

Figura 4.3: Algoritmo *ite*.

Un conjunto de ROBDDs que representen varias funciones se pueden unificar en un sólo grafo que conste de ROBDDs que compartan los subgrafos que sean comunes a todos ellos. Esta idea permite el ahorro de tiempo y espacio, ya que evita la duplicación de subgrafos. Cuando los subgrafos isomorfos se comparten completamente, nunca coexisten dos nodos equivalentes. Este tipo de estructura se conoce como *ROBDDs compartidos* [MIY90] o *ROBDDs con múltiples nodos raíz*. Cuando se utiliza un entorno software de ROBDDs compartidos, la comprobación de equivalencia se puede realizar de forma inmediata simplemente considerando los nodos raíz.

En la figura 4.4 se muestra un ROBDD compartido que representa las funciones $f_1 = x_2 \wedge \overline{x}_1$, $f_2 = x_2 \oplus x_1$, $f_3 = \overline{x}_1$ y $f_4 = x_2 \vee \overline{x}_1$.

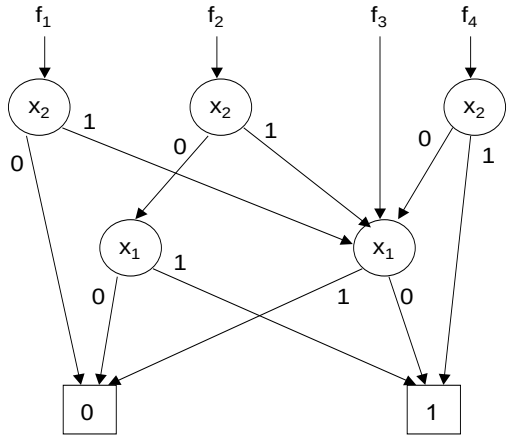


Figura 4.4: ROBDD compartido de las funciones $f_1 = x_2 \wedge \overline{x}_1$, $f_2 = x_2 \oplus x_1$, $f_3 = \overline{x}_1$ y $f_4 = x_2 \vee \overline{x}_1$.

Los algoritmos de construcción y manipulación dados por Bryant en [Bry86] consideran ROBDDs no compartidos (separados). En la actualidad, los entornos de representación y manipulación de funciones por medio de ROBDDs utilizan ROBDDs compartidos. A lo largo de la tesis, cuando hablemos de ROBDDs supondremos que se trata de ROBDDs compartidos.

4.1.2. Aspectos de implementación de ROBDDs

En una implementación típica de un entorno de manipulación de ROBDDs, la construcción se realiza por medio de la aplicación del operador *ite* y los nodos se almacenan en una única tabla (llamada la *tabla única*) en la memoria principal del computador.

En la tabla 4.1 se puede observar un ejemplo de la *tabla única* correspondiente al ROBDD mostrado en la figura 4.4 y que representa las cuatro funciones $f_1 = x_2 \wedge \overline{x}_1, f_2 = x_2 \oplus x_1, f_3 = \overline{x}_1$ y $f_4 = x_2 \vee \overline{x}_1$. Cada nodo no terminal tiene tres atributos básicos: un *índice* de la variable de entrada y dos punteros a sus sucesores, representados en la tabla como *low()* y *high()*. Cada nodo presenta, además, una serie de punteros y contadores adicionales que son necesarios para el mantenimiento de la tabla. Los nodos terminales **0** y **1** se colocan al comienzo de la tabla como nodos especiales, mientras que el resto de nodos no terminales se van generando paulatinamente como resultado de las operaciones lógicas.

Dirección	Índice	low()	high()	Función
D_0	-	-	-	0
D_1	-	-	-	1
D_2	x_1	D_0	D_1	
D_3	x_1	D_1	D_0	f_3
D_4	x_2	D_0	D_3	f_1
D_5	x_2	D_2	D_3	f_2
D_6	x_2	D_3	D_1	f_4

Tabla 4.1: Representación de un ROBDD utilizando una tabla.

Antes de la creación de un nuevo nodo, se comprueban las reglas de reducción. Si los sucesores *low()* y *high()* tienen el mismo destino o si ya existe un nodo equivalente, entonces no se crea un nuevo nodo sino que se copia simplemente el puntero al nodo ya existente. Para encontrar un nodo equivalente, se comprueba la tabla que contiene todos los nodos existentes con sus índices. Como técnica efectiva de aceleración de este proceso de comprobación se pueden utilizar las tablas *hash* [Gon84]. De esta forma se mantiene la unicidad de los nodos y toda función Booleana se puede identificar por medio de la dirección del nodo raíz.

Cuando se generan ROBDDs de expresiones Booleanas, se generan también temporalmente muchos ROBDDs intermedios. Con respecto a la eficiencia de memoria, es importante eliminar estos ROBDDs intermedios no necesarios. Para poder determinar la necesidad de los nodos, cada nodo lleva asociado un *contador de referencias* que indica el número de punteros que direccionan ese nodo (el número de referencias a ese nodo). Cuando el número de referencias a un nodo es cero entonces ese nodo se puede eliminar, liberando por tanto memoria.

Otra técnica usada para la reducción del tiempo de cálculo y de las necesidades de memoria de los ROBDDs consiste en la utilización de *aristas atribuidas*, es decir, aristas con un atributo que representa una cierta operación. Se han propuesto varios tipos de aristas atribuidas [MIY90], aunque la más efectiva y la más utilizada en la implementación de entornos software de manipulación de ROBDDs es la *arista negativa*.

La *arista negativa* es un atributo que indica que la función del subgrafo apuntado por la arista está complementada. Esta idea fue llamada *inversor* por Akers [Ake78] y *arista caracterizada* (*typed edge*) por Madre y Billon [MB88]. Entre las ventajas que presenta la utilización de aristas negativas se encuentran la reducción del tamaño del ROBDD, la realización del complemento en tiempo constante y la aceleración del cálculo de operaciones lógicas por medio de la utilización de ciertas reglas como $f \wedge \overline{f} = 0$, $f \vee \overline{f} = \mathbf{1}$, $f \oplus \overline{f} = \mathbf{1}$, etc.

En esta tesis se ha utilizado el paquete CUDD[4] de la Universidad de Colorado, que es uno de los entornos más eficientes de manipulación de ROBDDs. Este entorno utiliza también el operador *ite* para la construcción e implementa las técnicas comentadas anteriormente para la reducción de espacio y tiempo de cálculo de ROBDDs.

En el paquete CUDD, todos los nodos usados en los ROBDDs se almacenan en tablas hash especiales llamadas *tablas únicas* cuyo principal objetivo es garantizar que cada nodo sea único, es decir, que no exista otro nodo etiquetado por la misma variable y con los mismos descendientes (hijos). Esta propiedad de unicidad es la que proporciona la canonicidad a los ROBDDs. Las tablas únicas y otras estructuras de datos auxiliares constituyen el *Manager* (administrador) del entorno CUDD. La tabla única está formada a su vez por tantas tablas hash como variables se estén utilizando. Estas tablas hash se conocen como *subtablas únicas*.

Para la manipulación eficiente de los diagramas de decisión, el entorno CUDD utiliza una tabla de almacenamiento de resultados ya calculados conocida como la *Cache*. También se utilizan en el CUDD las técnicas de contador de referencias (*Cuentas de referencia*) y de aristas atribuidas negativas (*Arcos complemento*) ya mencionadas, entre otras.

[4] *CU Decision Diagram Package.* ftp://vlsi.colorado.edu/pub/

4.2. Ordenación de las variables de entrada

Los ROBDDs constituyen representaciones canónicas de funciones Booleanas bajo una ordenación fija de las variables de entrada, sin embargo, la permutación del orden de las variables puede conducir a diferentes ROBDDs para la misma función. El efecto de la ordenación de variables depende de la naturaleza de las funciones a manejar y en muchas ocasiones el tamaño de los ROBDDs varía enormemente. Como la complejidad computacional de un ROBDD está muy relacionada con sus necesidades de espacio, es importante encontrar una ordenación que intente minimizar la memoria utilizada en todas las fases de manipulación de un ROBDD. La ordenación de variables es por tanto un problema importante para la manipulación y generación de ROBDDs.

En la figura 4.5 se observa la diferencia de tamaño existente entre dos ROBDDs que representan la misma función Booleana $f = x_1x_2 \vee x_3x_4 \vee x_5x_6$ para dos ordenaciones de variables diferentes. En la figura 4.5(a) se muestra el ROBDD correspondiente a una buena ordenación de variables, mientras que en la figura 4.5(b) se tiene el ROBDD para la misma función, pero con una mala ordenación.

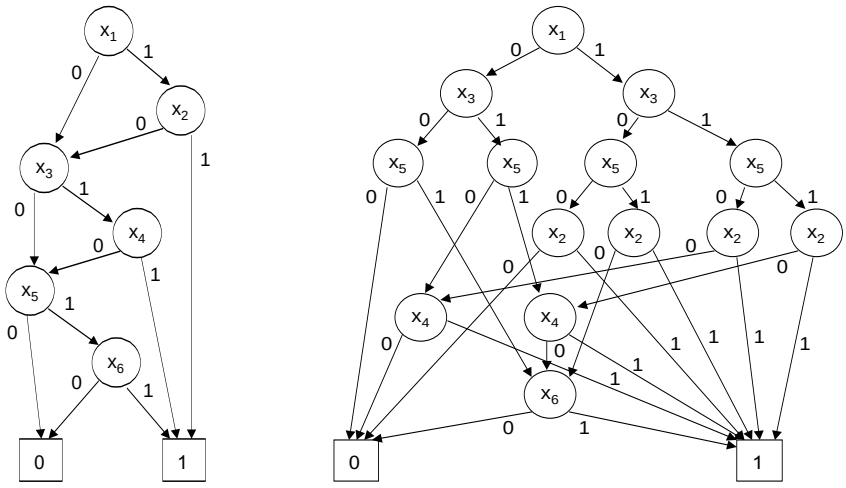


Figura 4.5: (a) Mejor ordenación. (b) Peor ordenación.

El hallazgo de la mejor ordenación posible es un problema NP-completo [THY93]. Los algoritmos existentes se limitan a problemas con un número pequeño de variables de entrada, y es difícil encontrar la mejor ordenación para problemas más complejos en un tiempo limitado. Friedman y Supowit [FS90] utilizaron programación dinámica en la construcción de un algoritmo para el cálculo de una ordenación de variables que producía tamaños mínimos de ROBDDs, pero dicho algoritmo requería tiempo y espacio exponenciales

y sólo podía ser aplicable a funciones con un pequeño número de variables. El algoritmo fue mejorado por Ishiura et al. [ISY91] y por Drechsler et al. [DDG98], presentando los mismos problemas para circuitos con un número de variables superior a 30. Para evitar estos problemas se han estudiado distintos métodos de ordenación, pudiéndose distinguir dos tipos de aproximaciones.

Con respecto a la ordenación de variables, existen propiedades empíricas conocidas, como son que los grupos de variables con una relación próxima deben estar también próximas en la ordenación, y que las variables que ejerzan un gran *control* sobre la función deben situarse en las posiciones más altas de la ordenación (por ejemplo, en el caso de los selectores de datos). Basándose en estas reglas empíricas, se han enunciado algoritmos heurísticos para la ordenación de variables, en los que principalmente se realizan análisis topológicos de los circuitos y se utilizan variaciones de las búsquedas *primero en profundidad* (*depth-first*) y *primero en anchura* (*breadth-first*). En este tipo de aproximaciones, la función a ser representada viene dada por una descripción circuital. Fujita et al. [FFK88] y Malik et al. [MWBSV88] presentaron métodos que recorren el circuito lógico desde las salidas hacia las entradas de forma primero en profundidad y que realizan la ordenacion según el algoritmo va proporcionando las variables de entrada (de mayor a menor posición en la ordenación). Minato [MIY90] ideó otro método heurístico basado en la asignación de pesos a los circuitos lógicos. Butler et al. [BRKM91] utilizaron medidas de testabilidad para la realización de la heurística, en la que se refleja la información tanto topológica como lógica del circuito. Estos y otros métodos [FMK91][JPHS91][FFM93] pueden encontrar relativamente buenas ordenaciones de variables antes de realizar la construcción de un ROBDD, pero debido a su naturaleza heurística, no hay un método que produzca ordenaciones adecuadas para todos los casos.

Existe un segundo tipo de aproximaciones que tratan de mejorar las ordenaciones calculadas por los métodos heurísticos una vez que se tiene ya construido el ROBDD, y se basan principalmente en el intercambio de variables del ROBDD. En estos métodos se selecciona, por tanto, una ordenación inicial de variables (por medio de una heurística o usando la ordenación inicial del circuito), se construye el ROBDD del circuito y finalmente se minimiza el ROBDD tratando de mejorar la ordenación de variables. Fujita et al. [FMK91] propusieron un algoritmo de búsqueda local y Minato [Min92] presentó otro método basado en intercambio de variables que mide la *anchura* de los ROBDDs como función de coste. En el método de *ordenación dinámica de variables* de Rudell [Rud93], el propio paquete ROBDD determina y mantiene el orden de las variables, donde se ejecuta automáticamente el proceso de reordenación cada vez que el ROBDD alcanza un tamaño determinado. También se están utilizando aproximaciones basadas en algoritmos de *enfriamiento simulado* (*Simulated annealing*) [BLW95][GS97a] y en algoritmos genéticos [DBG96][GSM96][GS97b][GSMR98] para el cálculo de buenas ordenaciones.

Los algoritmos *heurísticos* típicos utilizan información topológica para calcular la ordenación de las variables de entrada del circuito, reflejando por tanto su estructura. No necesitan mucho espacio de almacenamiento y el tiempo de cálculo de la ordenación es normalmente mucho más eficiente que la traslación del circuito a un ROBDD utilizando esa ordenación. Las aproximaciones basadas en el *intercambio de variables* que utilizan los ROBDDs para el cálculo de una ordenación óptima o para la mejora de una ordenación dada, presentan características diferentes. En estos métodos se comprueban muchas ordenaciones intermedias, por lo que no pueden competir en tiempo de cálculo con las aproximaciones heurísticas. Por ejemplo, el enfriamiento simulado se debería aplicar únicamente si las otras heurísticas no han proporcionado una buena ordenación o si interesan ordenaciones óptimas, ya que debido a su estrategia de búsqueda aleatoria puede encontrar buenas ordenaciones que no se pueden calcular directamente de la estructura del circuito.

A continuación se presenta un algoritmo heurístico basado en la estructura del circuito para el cálculo de una ordenación fija de variables que sea utilizada para la construcción de un ROBDD de tamaño reducido del circuito.

4.2.1. Una aproximación heurística

Nuestro metodo heurístico [IS95][ISL95] utiliza la información topológica del circuito para realizar una asignación de *pesos* a cada una de las puertas que constituyen el circuito. A partir de dichos pesos se obtendrá la ordenación en función también de pesos asignados a las variables de entrada del circuito. Esta ordenación fija de variables será utilizada para la construcción del ROBDD de tamaño reducido del circuito.

La idea básica del algoritmo consiste en encontrar *no* una nueva ordenación de variables diferente de la ordenación inicial dada por la descripción circuital, sino en seleccionar *una única variable* de entrada que será colocada en primer lugar en la ordenación, manteniendo el resto de variables en la ordenación original (desplazadas una posición). El tamaño de los ROBDDs normalmente es muy sensible a la ordenación de variables seleccionada para su construcción, y en muchos casos la ordenación original constituye una buena ordenación para la construcción del ROBDD inicial [FFM93]. Esto puede ser cierto debido a que los datos de la descripción original del circuito se pueden ordenar utilizando información procedente del diagrama del circuito real. Sobre este ROBDD inicialmente construido se pueden aplicar posteriormente otros algoritmos basados en el intercambio de variables como los mencionados anteriormente. Nuestro método pretende realizar una modificación *incremental* de la ordenación original seleccionando una variable *óptima*, cuya "influencia" sobre el circuito se obtenga de la información topológica y que se coloque en primer lugar de la nueva ordenación obtenida, desplazando consecuentemente el resto de variables de la ordenación original en una posición.

El método realiza dos *recorridos* del circuito original realizando asignaciones de pesos numéricos a las puertas del circuito en cada uno de los recorridos. Esta asignación numérica tiene en cuenta el *fanin* (número de entradas) y el *fanout* (número de salidas) de cada puerta. En el primer recorrido de asignación el objetivo es la obtención de los pesos de las puertas de salida, que serán utilizados en el segundo recorrido para obtener finalmente los pesos de las entradas que se usarán para seleccionar la *variable óptima* a situar en la primera posición de la nueva ordenación final. El *primer recorrido* consta de los siguientes pasos:

- Se asigna un peso de 1 a cada una de las entradas primarias del circuito.
- Se asigna un peso a cada puerta sumando los pesos de sus entradas.
- Para cada puerta del circuito, el peso de su salida es el peso de la puerta dividido por su fanout.

De esta forma se obtienen los pesos de las salidas haciendo un recorrido del circuito desde las entradas hacia las salidas y partiendo de una asignación inicial de pesos a las entradas. El peso de las salidas reflejaría en cierto sentido la *influencia* que ejerce la estructura del circuito sobre las mismas. Esta información es la que se utilizará en el segundo recorrido para calcular la variable óptima de entrada.

En la figura 4.6 se muestra el pseudocódigo para el cálculo de los pesos de los *nodos* del circuito para el *primer recorrido*, donde un *nodo* puede ser tanto una puerta como una entrada primaria y donde *nodofanin* representa una de las entradas de *nodo*.

```
asignapeso1(nodo) {
  if (nodo = entrada_primaria)
    peso(nodo) = 1;
  else
    while (∃ nodofanin)
      peso(nodo) = peso(nodo) + asignapeso1(nodofanin)/fanout(nodofanin);
}
```

Figura 4.6: Pseudocódigo para el cálculo del peso de un nodo para el primer recorrido del circuito.

Una vez que se han asignado unos pesos iniciales a las salidas, se realiza el *segundo recorrido* del circuito en el que, comenzando con los pesos de las salidas, se van calculando los pesos del resto de nodos del circuito hacia las entradas del mismo hasta calcular, finalmente, los nuevos pesos para las entradas primarias. Únicamente se consideran los pesos de las salidas del circuito obtenidos en el primer recorrido, realizándose una nueva asignación de pesos a partir de los mismos.

En la figura 4.7 se muestra el pseudocódigo para el cálculo de los nuevos pesos de los *nodos* para el *segundo recorrido* del circuito, donde *nodofanout* representa una de las salidas de *nodo*.

```
asignapeso2(nodo) {
  if (fanout(nodo) >1) {
    while (∃ nodofanout) {
      nfanin = fanin(nodofanout);
      nfanout = fanout(nodofanout);
      if (nfanin = nfanout)
        peso(nodo) = peso(nodo) + asignapeso2(nodofanout)·nfanin;
      else
        peso(nodo) = peso(nodo) + asignapeso2(nodofanout)/nfanout;
    }
  }
  else     /* fanout(nodo) = 1 */
    peso(nodo) = asignapeso2(nodofanout)/fanout(nodofanout);
  if (nodo = entrada_primaria)
    peso(nodo) = peso(nodo)/fanout(nodo);
  if (nodo = salida)
    peso(nodo) = peso(salida);
}
```

Figura 4.7: Pseudocódigo para el cálculo del nuevo peso de un nodo para el segundo recorrido del circuito.

Con la nueva asignación de pesos de las entradas producidas por el segundo recorrido del algoritmo, se obtiene la *variable óptima* seleccionando aquella entrada primaria con mayor peso. La nueva ordenación se obtendrá, por lo tanto, situando la variable óptima en la primera posición de la ordenación y desplazando en una posición la ordenación original de variables (de la que se habrá extraído la variable óptima). En la figura 4.8 se muestra un circuito ejemplo donde *Rec1* y *Rec2* representan los pesos asignados a cada nodo por el primer y segundo recorridos del algoritmo, respectivamente.

En el ejemplo de la figura 4.8, la ordenación original del circuito sería $\{x_1, x_2, x_3, x_4, x_5\}$. Después del primer recorrido del circuito, se obtendrían los pesos 2.625 y 2.375 para las salidas $out0$ y $out1$, respectivamente. A partir de esta primera asignación de las salidas, el segundo recorrido del algoritmo obtendría finalmente los pesos de las entradas primarias. El mayor peso obtenido para las variables de entrada es 13.812, correspondiente a la variable x_3, por lo que esta sería la *variable óptima* obtenida por la heurística. La nueva ordenación de variables sería, por tanto, $\{x_3, x_1, x_2, x_4, x_5\}$, siendo esta la ordenación inicial fija que se utilizaría para la construcción del ROBDD del circuito.

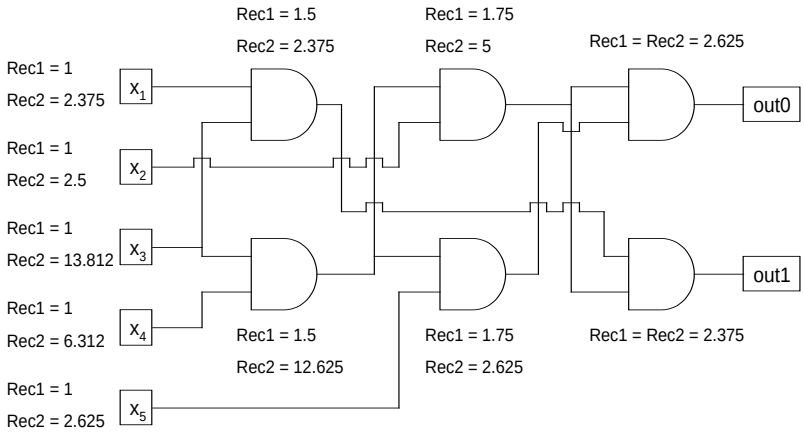


Figura 4.8: Circuito con los pesos asignados por los dos recorridos del algoritmo.

4.2.1.1. Resultados experimentales

Se ha realizado la implementación en código C del algoritmo heurístico de ordenación y se ha comparado con otras heurísticas dadas en la literatura, utilizando como benchmarks algunos de los circuitos ISCAS85 para generación de patrones de test [BF85] cuyas característica se muestran en la tabla 4.2. Se ha calculado la ordenación de variables utilizando nuestra heurística y se han construido los ROBDDs de los benchmarks utilizando dicha ordenación, comparándose posteriormente con los ROBDDs obtenidos para los mismos circuitos usando la ordenación original y la ordenación obtenida con otras heurísticas. Los experimentos se han realizado en una estación de trabajo SUN UltraSparc-II y se ha utilizado para la construcción de los ROBDDs el paquete CUDD de la universidad de Colorado.

Se han comparado los ROBDDs de los benchmarks obtenidos utilizando nuestra heurística de ordenación con los obtenidos usando la ordenación original que aparece en el formato de datos ISCAS de cada circuito. Como métricas para la comparación se han utilizado el tamaño final del ROBDD (número de nodos), el tiempo (en segundos) necesario para su construcción y el tamaño máximo del ROBDD (número máximo de nodos) obtenido durante la construcción. En la tabla 4.3 se muestran los resultados obtenidos para este primer experimento, donde *Máximo* representa el tamaño máximo del ROBDD.

De los resultados mostrados en la tabla 4.3, se observa que la ordenación obtenida con nuestra heurística es más eficiente para la construcción de los ROBDDs que la ordenación original suministrada con los circuitos. Considerando los valores totales obtenidos para los benchmarks, se tiene que el tamaño final de los ROBDDs es un 28.65 % menor utilizando nuestra heurística, el tiempo necesario para la construcción de los ROBDDs es un 17.78 % menor

Circuito	Entradas	Salidas	Niveles	Puertas
C432	36	7	7	203
C499	41	32	11	275
C880	60	26	24	469
C1355	41	32	24	619
C1908	33	25	24	619
C3540	50	22	47	1741

Tabla 4.2: Características de los circuitos de benchmark ISCAS.

y el tamaño máximo obtenido es un 26.67 % menor que el necesitado por la ordenación original. Los mejores resultados se obtienen para el circuito C880, para el cual las mejoras de nuestra heurística con respecto de la ordenación original son de un 84.65 % para el tamaño final, de un 86.34 % para el tiempo de construcción y de un 86.21 % para el tamaño máximo del ROBDD.

	Ordenación original			Nuestra heurística		
Circuito	Tamaño	Tiempo	Máximo	Tamaño	Tiempo	Máximo
C432	1733	0.12	12264	1713	0.12	11242
C499	45922	0.92	62342	45922	0.92	62342
C880	346660	18.52	1370502	53236	2.53	189070
C1355	45922	2.73	187026	45922	2.73	187026
C1908	36007	3.92	164542	19673	1.70	91980
C3540	604559	69.15	2907590	604790	70.41	2910656
Total	1080803	95.36	4704266	771256	78.41	3452316

Tabla 4.3: Comparación de los ROBDDs obtenidos usando la ordenación original y usando la ordenación obtenida con nuestra heurística.

En la tabla 4.4 se comparan los tamaños de los ROBDDs obtenidos para los benchmarks ISCAS85 seleccionados utilizando la ordenación original, la ordenación obtenida con nuestra heurística (*Ntra. heur.* en la tabla) y las obtenidas con varias de las heurísticas más conocidas. Los métodos heurísticos *level* y *fanin* fueron propuestos por Malik et al. [MWBSV88], la heurística *fanout* se debe a Fujita et al. [FFK88] y la heurística *fault simulation* fue propuesta por Butler et al. [BRKM91]. El resto de heurísticas que aparecen en la tabla corresponden a los datos dados en [BRKM91] para las heurísticas primero en profundidad (B), primero en anchura (D), simulación de fallos (SIM), heurística que utiliza medidas de testabilidad (SC), heurística propuesta por Malik que mejora los resultados dados en [MWBSV88] (ucb) y heurística propuesta por Brace et al. en [BBR90] (cmu). De los resultados obtenidos, se observa que nuestra heurística proporciona el menor tamaño total de todas las ordenaciones indicadas, siendo especialmente significativas las reducciones de tamaño obtenidas en los circuitos C432 y C880.

Ordenación	Circuitos						Total
	C432	C499	C880	C1355	C1908	C3540	
Original	1733	45922	346660	45922	36007	604559	1080803
level	29801	48431	500731	46044	36012	502184	1163203
fanin	26680	39858	346762	58513	18435	611088	1101336
fanout	26680	57434	346762	54130	9356	611088	1105450
fault sim	31073	38314	70071	49980	18764	590215	798417
B	42k	n.p.	n.p.	-	143k	n.p.	-
D	n.p.	238k	35k	483k	180k	-	-
SIM	n.p.	n.p.	n.p.	-	228k	-	-
SC	80k	n.p.	n.p.	-	342k	n.p.	-
cmu	n.p.	235k	29k	477k	147k	-	-
ucb	n.p.	217k	33k	438k	147k	n.p.	-
Ntra. heur.	1713	45922	53236	45922	19673	604790	771256

Tabla 4.4: Comparación de tamaños de los ROBDDs construidos usando la ordenación original, la obtenida por distintas heurísticas y la obtenida por nuestra heurística (*n.p.* = no posible).

Como se ha mencionado anteriormente, una vez construido el ROBDD utilizando una ordenación inicial fija (obtenida heurísticamente o usando la ordenación original) se pueden aplicar otros métodos de reducción del ROBDD basados principalmente en el intercambio de variables.

En la tabla 4.5 se muestran los resultados obtenidos de una segunda reducción de los ROBDDs anteriormente construidos usando la ordenación original y nuestra heurística, donde se ha utilizado el algoritmo de *desplazamiento* (*reordering sifting*) de Rudell [Rud93] para realizar la reducción y donde las métricas de comparación empleadas han sido el tamaño final del ROBDD (número de nodos) y el tiempo (en segundos) necesario para su construcción. De los resultados obtenidos se observa que nuestra heurística es, de nuevo, más eficiente que la ordenación original cuando se realiza la reducción de sus respectivos ROBDDs usando el algoritmo de desplazamiento de Rudell. Las mejoras obtenidas en este caso, considerando los valores totales medidos para los distintos benchmarks, son de un 2.45 % para el tamaño final y de un 11.6 % para el tiempo de construcción. De nuevo los mejores resultados se obtienen para el circuito C880, donde las mejoras en el tamaño final y en el tiempo de construcción son del 27.8 % y del 75.5 %, respectivamente. Las mejoras totales de espacio obtenidas por nuestra heurística no son, en este caso, tan significativas como las mostradas anteriormente en la tabla 4.3, aunque con respecto al tiempo de construcción sigue siendo apreciable. El motivo sería que el algoritmo de desplazamiento de Rudell permite obtener resultados bastante optimizados en tamaño para los ROBDDs, estando éstos ya próximos a los mejores tamaños obtenidos en la literatura [BLW95].

	Ordenación original		Nuestra heurística	
Circuito	Tamaño	Tiempo	Tamaño	Tiempo
C432	1210	0.32	1211	0.31
C499	30775	17.99	30775	17.99
C880	7064	36.61	5098	8.97
C1355	30775	20.13	30775	20.13
C1908	7153	13.71	6522	5.09
C3540	27908	195.54	27933	198.89
Total	104885	284.3	102314	251.38

Tabla 4.5: Comparación de los ROBDDs obtenidos tras reducir los ROBDDs previamente creados usando la ordenación original y nuestra heurística, utilizando el algoritmo de *desplazamiento* de Rudell.

En la tabla 4.6 se muestran los resultados obtenidos de reducir los ROBDDs previamente construidos usando la ordenación original y nuestra heurística, utilizando el algoritmo de *enfriamiento simulado*. Las métricas de comparación utilizadas son el tamaño final del ROBDD y el tiempo en segundos necesario para su construcción. Los datos obtenidos en este caso reflejan diferencias poco significativas, siendo prácticamente inexistentes para los tamaños totales y de apenas un 5 % de mejora en el tiempo total de construcción. Para el benchmark C880 la mejora es temporal, siendo de un 56.3 %. Los motivos serían los mismos que los señalados anteriormente para el algoritmo de Rudell. Los resultados de tamaño obtenidos con el enfriamiento simulado están ya muy optimizados y próximos a los mejores obtenidos en la literatura [BLW95], sin embargo, se puede observar que los tiempos necesarios para alcanzar estos tamaños óptimos son excesivamente grandes comparados con los métodos heurísticos (e incluso con el algoritmo de Rudell). Las mismas consideraciones se podrían hacer con los métodos que utilizan algoritmos genéticos.

	Ordenación original		Nuestra heurística	
Circuito	Tamaño	Tiempo	Tamaño	Tiempo
C432	1209	82.0	1211	88.33
C499	25866	11675.06	25866	11724.2
C880	4054	2617.04	4054	1142.96
C1355	25866	10920.5	25866	10920.5
C1908	5652	1111.38	5526	974.51
C3540	23831	10389.67	23836	10291.89
Total	86478	36795.65	86359	35142.44

Tabla 4.6: Comparación de los ROBDDs obtenidos tras reducir los ROBDDs previamente creados usando la ordenación original y nuestra heurística, utilizando el algoritmo de *enfriamiento simulado*.

4.3. Otros tipos de diagramas de decisión

Existen algoritmos eficientes de manipulación de funciones representadas por ROBDDs, sin embargo, hay muchas funciones para las que los ROBDDs son excesivamente grandes como para poder ser almacenados en la memoria del computador. Por ejemplo, Bryant [Bry86][Bry91] demostró que los ROBDDs para la multiplicación de dos números de n bits tienen al menos $2^{\frac{n}{8}}$ nodos, y hay experimentos que muestran que los ROBDDs de los multiplicadores son muy grandes incluso para pequeños valores de n. Por esta razón, han aparecido un gran número de extensiones del concepto básico de ROBDD que permiten distintos tipos de representaciones de funciones. El inconveniente que presentan estos tipos de diagramas de decisión es que normalmente no se conocen algoritmos eficientes para su manipulación.

Entre el gran número de extensiones del concepto de ROBDD existentes, se pueden citar las extensiones obtenidas del reemplazamiento de la regla de descomposición de Shannon por otra regla de descomposición diferente. Ejemplos de estas extensiones serían los *diagramas de decisión funcional ordenados* (OFDDs) [KSR92][KR93]que utilizan la *descomposición positiva de Davio* $f = f|_{x_i=0} \oplus x_i(f|_{x_i=0} \oplus f|_{x_i=1})$, los *diagramas de decisión funcional de Kronecker ordenados* (OKFDDs) [DST+94] en los que en distintos nodos se puede seleccionar entre la descomposición positiva y la *descomposición negativa* ($f = f|_{x_i=1} \oplus \overline{x_i}(f|_{x_i=0} \oplus f|_{x_i=1})$) de Davio, o los *OBDDs de paridad*, tambien llamados *Mod2*-OBDDs o $\oplus$-OBDDs [GM96][Waa97], que son representaciones que combinan los OBDDs, los OFDDs y los OKFDDs y que se verán más ampliamente en el capítulo 6.

También existen extensiones obtenidas por medio de la flexibilización de la condición de ordenación de variables del ROBDD vista en la definición 12. De esta forma se obtienen los *BDDs libres* (FBDDs) [FHS78] y los FBDDs con *ordenación de grafo* (G-FBDDs) [SW95], en los que cada variable puede aparecer en cada camino dirigido sólo una vez, pero en los que pueden existir distintas ordenaciones de variables para distintos caminos. También se puede relajar la condición dada en la definición 12 de que en cada camino dirigido del ROBDD aparezca cada variable como mucho una sola vez. De esta forma se pueden encontrar los *BDDs indexados* (IBDDs) [JBA+97] en los que el conjunto de nodos se puede particionar en k *capas* de nodos de forma que para cada capa hay una ordenación de variables, mientras que los kOBDDs [BSSW98] tienen la condición adicional de que las ordenaciones de variables de las k capas son iguales. También se pueden mencionar los *BDDs transformados* (TBDDs) [BMS95] y los *BDDs con cero suprimido* [Min93], entre otros.

Las técnicas de los ROBDDs también se pueden extender para representar funciones *multivaluadas* $f : \{0,1\}^n \longrightarrow \mathcal{Z}$ en lugar de funciones Booleanas, cuyas aplicaciones se encuentran principalmente en el área de la verificación.

Para poder representar dichas funciones, es necesario extender los ROBDDs de forma que se puedan introducir valores numéricos (enteros, si se utiliza el campo de enteros $\mathcal{Z}$). Como ejemplos de estas extensiones, se encuentran los *BDDs multi-terminales* (MTBDDs) [CFM+93] que son una extensión simple de los ROBDDs en los que se permiten más de dos terminales, o los *diagramas algebraicos de decisión* (ADDs) [BFG+93] en los que se realiza una descomposición Shannon entera en cada nodo. También Jain et al. [JABF92] introdujeron los *diagramas semi-numéricos de decisión* (snDDs) utilizando esta aproximación.

Para tratar de incrementar la cantidad de subgrafos compartidos cuando se utilizan nodos terminales con valores numéricos, Lai y Sastry [LS92] presentaron los *diagramas de decisión binarios con aristas valuadas* (EVBDDs), que son MTBDDs con aristas atribuidas con ciertos *pesos*, cuyos valores son *sumados* a la función a representar. Si, además, se permiten pesos *multiplicativos* para las aristas, entonces se tienen los *EVBDDs factorizados* (FEVBDDs) [TP97]. Shen, Devadas y Ghosh introdujeron los *Diagramas Booleanos libres* (FBDs) [SDG95] que presentan nodos de decisión junto con nodos *funcionales* para la *disyunción* y la *conjunción*, además de incluir valores numéricos en ambos tipos de nodos.

Bryant y Chen [BC95] introdujeron los *diagramas de momento binario* (BMDs) y los *diagramas de momento binario multiplicativo* (*BMDs). Los BMDs utilizan la descomposición positiva de Davio (con valores enteros) y permiten nodos terminales con valores enteros (de forma análoga a los MTBDDs), es decir, son una generalización con valores enteros de los OFDDs. Los *BMDs son una generalización de los BMDs, ya que permiten pesos multiplicativos de aristas (los valores de las aristas se multiplican por las funciones que representan). Los *diagramas de momento binario de Kronecker* (KBMDs) [DBR96] pueden utilizar distintos tipos de descomposición entera por variable (de forma análoga a los OKFDDs a nivel binario). Los K*BMDs [DBR96] se diferencian de los KBMDs en que pueden utilizar pesos enteros aditivos y multiplicativos en paralelo (como en los FEVBDDs). Recientemente, Chen y Bryant [CB97] propusieron los *diagramas de decisión híbridos de potencia multiplicativa* (*PHDDs), que tienen una estructura similar a los K*BMDs pero donde los valores de las aristas se interpretan como potencias de una base dada.

4.4. Conclusiones

En este capítulo se han presentado los *diagramas de decisión binarios ordenados y reducidos* (ROBDDs), que son las estructuras de datos en forma de grafo más utilizadas en la actualidad para representación de funciones Booleanas. Se ha dado su evolución histórica y se han descrito sus características principales, considerándose también los aspectos de implementación de un entorno que utilice ROBDDs.

Una de las características de los ROBDDs es que constituyen representaciones canónicas de funciones Booleanas bajo una ordenación fija de las variables de entrada, pero distintas ordenaciones de las variables conducen a diferentes ROBDDs con distintos tamaños. Por lo tanto, la ordenación de variables para un ROBDD constituye un problema muy importante, existiendo distintos algoritmos de ordenación que se pueden clasificar en métodos *heurísticos* y métodos *dinámicos* de intercambio de variables. Los métodos heurísticos operan sobre el formato de datos original del circuito y calculan una ordenación de variables que se utilizará para la construcción del ROBDD inicial del circuito. Los métodos dinámicos de ordenación operan sobre el ROBDD ya construido del circuito y tratan de reducirlo probando distintas ordenaciones, utilizando como función de coste el tamaño del ROBDD obtenido.

Se ha presentado un método *heurístico* de ordenación de variables basado en la información topológica del circuito. Este algoritmo realiza dos recorridos del circuito asignando pesos numéricos a las puertas en cada uno de ellos. El primer recorrido calcula los pesos de las salidas, que serán utilizados en el segundo recorrido del algoritmo para el cálculo final de los pesos de las entradas. La variable de entrada con mayor peso (*variable óptima*) será seleccionada para ocupar la primera posición en la nueva ordenación, mientras que el resto de variables mantienen la ordenación original, desplazadas en una posición. Se han realizado experimentos construyendo los ROBDDs de circuitos de benchmark ISCAS85 utilizando nuestra ordenación, la ordenación original y distintas heurísticas clásicas, comprobando la eficiencia de nuestra heurística en tamaño y tiempo de construcción de los ROBDDs. Posteriormente se han utilizado métodos dinámicos sobre los ROBDDs ya construidos heurísticamente con nuestra aproximación y con la ordenación original, comprobándose también el mejor comportamiento de nuestro método.

Finalmente, se ha dado una visión de las distintas extensiones del concepto de ROBDD para la representación de funciones Booleanas, dando sus principales características. También se han revisado los distintos tipos de grafos en forma de diagramas de decisión existentes en la literatura para la representación de funciones con valores numéricos. Este tipo de estructuras son las que tienen interés para nosotros debido a que la verificación probabilista utiliza transformaciones algebraicas de funciones Booleanas, con lo que es necesario introducir valores numéricos en este tipo de grafos para el cálculo posterior de la signatura que represente a dichas funciones. En los capítulos 5 y 6 se ven las aproximaciones que hemos utilizado para el cálculo de signaturas empleando diagramas de decisión modificados para la introducción de valores *numéricos*.

Capítulo 5

Aproximación híbrida de verificación

En este capítulo se describe una aproximación híbrida de verificación que utiliza la verificación determinista que emplea ROBDDs y la verificación probabilista. La combinación de ambos métodos requiere la introducción de valores numéricos en los ROBDDs, que se seleccionarán de un campo finito. Nosotros utilizamos el campo de Galois $GF(2^m)$, ya que permite la simplificación del cálculo de las signaturas. La aproximación híbrida se aplica a circuitos combinacionales de dos niveles, por lo que previamente se utiliza el método visto en el capítulo 3 para la obtención de una representación en forma de cubos disjuntos de dichos circuitos.

En los capítulos anteriores se han visto dos aproximaciones diferentes para la verificación de funciones Booleanas: el método probabilista y la utilización de ROBDDs como representaciones canónicas de funciones Booleanas. La *verificación probabilista* se basa en transformaciones algebraicas de las funciones Booleanas en términos de polinomios definidos sobre un cierto campo, mientras que la verificación por medio de ROBDDs realiza la *comparación* de dos funciones representadas por sus ROBDDs, de modo que si éstos son iguales (*isomorfos*) entonces ambas funciones son *equivalentes*. Este tipo de verificación que utiliza ROBDDs a veces se refiere como *verificación determinista*.

En este capítulo se utiliza una aproximación *híbrida* para la verificación de funciones Booleanas que combina los dos métodos, de forma que mientras no existan limitaciones de espacio o tiempo se utiliza la verificación determinista (ROBDDs), pero si existe alguna, entonces se utiliza la probabilista. Esta aproximación híbrida se aplica a funciones Booleanas representadas en forma cúbica de dos niveles, para lo cual se utiliza el método de cálculo de operaciones Booleanas visto en capítulo 3 que genera conjuntos de cubos *disjuntos*.

En la verificación *probabilista* se calcula la signatura de una función Booleana seleccionando un valor *numérico* aleatorio para cada variable de entrada de la función, y evaluando una versión transformada de esa función utilizando esos valores. En general, la selección aleatoria de esos valores se puede realizar utilizando cualquier campo, pero computacionalmente interesa la utilización de *campos finitos*. En el capítulo 2 se ha utilizado el campo de enteros *módulo* p para algún primo p, $\mathcal{Z}_p$, sin embargo, para nuestra aproximación híbrida vamos a considerar el campo de *Galois* $GF(2^m)$, que es un campo finito de característica 2 con 2^m elementos. En este capítulo únicamente presentamos las características básicas de los campos $GF(2^m)$ que nos permitan ver su aplicación a la verificación probabilista. En la segunda parte de esta tesis (capítulos 7 al 9) se estudian más en profundidad los campos de Galois y, al ser la multiplicación una de las operaciones más *costosas* e importantes, se presentan distintos métodos de multiplicación de elementos pertenecientes a dicho campo, que dependerán de la selección del tipo de *base* de representación de los elementos y del polinomio *irreducible* generador del campo.

Para poder combinar los dos tipos de verificación en una aproximación *híbrida*, también es necesaria la introducción de valores numéricos en la estructura de datos del ROBDD. Esto se podrá realizar interpretando el ROBDD como una representación algebraica (para lo cual se utilizan las propiedades y teoremas vistos en el capítulo 2) y evaluando posteriormente dicha representación sobre los valores *numéricos* seleccionados del campo $GF(2^m)$.

La aproximación híbrida de verificación se aplica a funciones Booleanas que representan circuitos lógicos combinacionales de dos niveles expresados en forma *cúbica* (como un conjunto de *cubos* o SOPs). De las propiedades dadas en el capítulo 2, se observa que el cálculo de la transformada $\mathcal{A}$ (y por tanto de la signatura) se simplifica en gran medida cuando se cumple la propiedad de *disjunción* u *ortogonalidad* para las funciones a operar (teorema 8). Por este motivo utilizamos el método de cálculo de operaciones Booleanas dado en el capítulo 3 para realizar la transformación de los benchmarks dados en formato de sumas de productos no disjuntos (SOP) a un formato de *sumas de productos disjuntos* (DSOP), y realizamos la verificación sobre representaciones distintas en forma DSOP de dichos benchmarks para comprobar su equivalencia.

Se realizan experimentos de verificación *probabilista*, *determinista* e *híbrida*, comparándose la memoria y el tiempo de ejecución necesarios para la verificación. El método *probabilista* implica el cálculo y manipulación de signaturas exclusivamente, mientras que los métodos *determinista* e *híbrido* implican la construcción de ROBDDs con valores *numéricos*. Por este motivo, para los experimentos realizados con estos dos modos de verificación se tienen en cuenta dos aproximaciones relativas a la ordenación de variables: usando la ordenación *original fija* dada por los circuitos de benchmark y utilizando la aproximación *dinámica* de reordenación de variables por desplazamiento de Rudell.

5.1. Aplicación de campos de Galois a la verificación probabilista

Como se vio en el capítulo 2, la verificación probabilista consiste en la generación de un código numérico o *signatura* para una función Booleana, de forma que la verificación de equivalencia de dos funciones se realice comparando sus signaturas. La signatura de una función Booleana se obtiene evaluando la representación de dicha función sobre unos valores seleccionados aleatoriamente de un *campo finito*. Si las signaturas de dos funciones son diferentes, entonces *no son equivalentes* con seguridad, pero si las signaturas son iguales, entonces las funciones son *equivalentes* con una probabilidad pequeña de error.

Para poder generar la *signatura* de una función Booleana con variables de entrada Booleanas $x_{n-1}, ..., x_1, x_0$, se debe seleccionar un valor *numérico* aleatorio (generado independientemente y uniformemente distribuido) para cada variable x_i. Estos valores se pueden seleccionar de cualquier *campo*, pero nosotros utilizaremos el campo finito $GF(p^m)$, con p *primo* y $m \in \mathcal{N}$, que representa un *campo de Galois*[1] con p^m elementos.

La evaluación de la función Booleana (con los valores aleatorios seleccionados asignados a sus variables de entrada) se puede realizar por medio del reemplazo de la función Booleana por una función *aritmética* equivalente definida sobre el campo finito. Este reemplazo está determinado por una *transformación algebraica* (la transformada $\mathcal{A}$) de la función Booleana en términos de polinomios sobre el campo finito. Si $\mathcal{A}[\beta_1]$ y $\mathcal{A}[\beta_2]$ representan los polinomios asociados a las funciones Booleanas β_1 y β_2 por la transformada $\mathcal{A}$, se vio en el capítulo 2 que se pueden establecer las siguientes transformaciones de funciones Booleanas a expresiones aritméticas (utilizando las $0, 1 - equivalencias$ y las operaciones Booleanas extendidas dadas en la tabla 2.2)

$$\begin{aligned}
\neg\beta &\xrightarrow{\mathcal{A}} 1 - \mathcal{A}[\beta] \\
\beta_1 \wedge \beta_2 &\xrightarrow{\mathcal{A}} \mathcal{A}[\beta_1] \cdot \mathcal{A}[\beta_2] \\
\beta_1 \vee \beta_2 &\xrightarrow{\mathcal{A}} \mathcal{A}[\beta_1] + \mathcal{A}[\beta_2] - \mathcal{A}[\beta_1] \cdot \mathcal{A}[\beta_2] \\
\beta_1 \oplus \beta_2 &\xrightarrow{\mathcal{A}} \mathcal{A}[\beta_1] + \mathcal{A}[\beta_2] - 2 \cdot \mathcal{A}[\beta_1] \cdot \mathcal{A}[\beta_2]
\end{aligned}$$

donde las operaciones aritméticas de suma, resta y multiplicación se realizan sobre el campo finito. En realidad, como se vio en el teorema 2, el cálculo correcto requiere el cumplimiento de la propiedad de idempotencia de la multiplicación, por lo que habría que aplicar de nuevo la transformada $\mathcal{A}$ a las combinaciones anteriores de transformadas $\mathcal{A}$ de las funciones β_1 y β_2.

Nosotros consideramos el campo de *Galois* $GF(2^m)$, que es un campo finito de característica 2 con 2^m elementos y donde cada elemento del campo se puede representar como un vector de m bits. $GF(2^m)$ es un *campo de*

[1] Los campos de Galois se estudiarán en profundidad en la segunda parte de esta tesis.

extensión del campo base $GF(2)$ de 2 elementos, es decir, $GF(2) = \{0, 1\}$. Los elementos distintos de cero del campo $GF(2^m)$ están generados por un elemento *primitivo* α, donde α es una raíz de un polinomio irreducible primitivo $g(x) = g_{m-1}x^{m-1} + \cdots + g_1x + g_0$ sobre $GF(2)$. Los elementos distintos de cero de $GF(2^m)$ se pueden representar como potencias del elemento primitivo α, es decir, $GF(2^m) = \{0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}, \alpha^{2^m-1} = 1\}$. Como α es una raíz del polinomio irreducible primitivo, se tiene que $g(\alpha) = 0$ y $\alpha^m = g_{m-1}\alpha^{m-1} + \cdots + g_1\alpha + g_0$. Por tanto, un elemento del campo $GF(2^m)$ se puede expresar también como un polinomio de α con grado menor que m. Es decir, $GF(2^m) = \{a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0 \mid a_i \in GF(2) \text{ para } 0 \leq i \leq m-1\}$. Estos polinomios se pueden representar como números decimales que varían desde 0 hasta 2^{m-1} y donde $m-1$ es el grado máximo que puede tomar cualquiera de los elementos. Por ejemplo, el polinomio $\alpha^5 + \alpha^4 + 1$ se representa por el número decimal 49, que a su vez se puede representar (con m=16) como el número (o vector) binario de 16 bits 0000000000110001 si se selecciona la *base polinómica* como base de representación de los elementos del campo $GF(2^{16})$.

La aritmética en un campo finito de característica 2 es esencialmente una aritmética *modular*. Por tanto, la *adición* (suma) de dos polinomios (elementos del campo) se convierte en la XOR (OR-exclusiva o disyunción exclusiva) bit a bit de sus representaciones binarias correspondientes, mientras que la *sustracción* (resta) es exactamente igual que la adición en aritmética *módulo* 2. Por ejemplo, para un campo $GF(2^{16})$ ($m = 16$) y utilizando una base polinómica de representación, la adición de los polinomios $\alpha^5 + \alpha^4 + \alpha + 1$ y $\alpha^4 + \alpha^2 + \alpha$ se calcula representándolos como los números binarios 0000000000110011 y 0000000000010110, respectivamente, y realizando la XOR bit a bit de ambas representaciones. El resultado obtenido sería 0000000000100101, que corresponde al polinomio $\alpha^5 + \alpha^2 + 1$. En este ejemplo, se tiene que el elemento unidad se representaría como el vector binario 0000000000000001, por lo que la suma del elemento unidad con cualquier otro elemento del campo se realizaría simplemente complementando el último bit (es decir, el bit menos significativo o el de menor peso) de la representación binaria de este elemento[2]. Esta propiedad será importante debido a que permite simplificar el cálculo del complemento de un elemento del campo en la verificación probabilista, como vemos posteriormente.

Por otra parte, la *multiplicación* de dos polinomios es la operación aritmética más importante y una de las más complejas y de mayor consumo de tiempo para su realización. Su complejidad puede depender de varios factores, como son la selección del polinomio irreducible o la selección de una *base* para la representación de los elementos del campo (generalmente, base *polinómica, dual* o *normal*).

[2]Esta propiedad se cumple cuando se utiliza la *base polinómica* como base de representación, pero no se cumple en otras bases, como se verá en el capítulo 7.

En esta primera parte de la tesis no profundizamos en la multiplicación sobre campos de *Galois* $GF(2^m)$, siendo el algoritmo de multiplicación en base *canónica* de Yeh et al. [YRT84] el utilizado para su aplicación en la verificación probabilista. En la segunda parte de la tesis se estudian en profundidad los campos de *Galois* y el algoritmo de multiplicación *canónica* aquí empleado (subsección 7.2.1), así como otros tipos distintos de multiplicación.

A partir de estas propiedades y teniendo en cuenta las transformaciones de funciones Booleanas a expresiones aritméticas vistas anteriormente, se puede justificar la utilización de $GF(2^m)$ como campo finito para la verificación probabilista:

- La *sustracción* se convierte en *adición*, por lo que para calcular el *complemento* de una función Booleana transformada, se tendrá que sumar el elemento unidad al polinomio que representa a la función Booleana original (su transformada $\mathcal{A}$).

- En la transformación aritmética de la *disyunción exclusiva* (XOR) de las funciones Booleanas β_1 y β_2, aparece el término $2 \cdot \mathcal{A}[\beta_1] \cdot \mathcal{A}[\beta_2]$. Como el campo $GF(2^m)$ tiene característica 2, este producto será igual a cero, por lo que el polinomio de la XOR de β_1 y β_2 se reduce simplemente a la adición de $\mathcal{A}[\beta_1]$ y $\mathcal{A}[\beta_2]$.

Aplicando estas simplificaciones y las transformaciones vistas, se pueden calcular las *signaturas* de las funciones Booleanas para las que se desea comprobar su equivalencia evaluando las expresiones obtenidas sobre el conjunto de elementos aleatoriamente seleccionados del campo de Galois $GF(2^m)$.

Como se vio en el capítulo 2, si S_1 y S_2 son las signaturas obtenidas para dos funciones Booleanas, estas signaturas se pueden comparar para verificación, existiendo dos posibilidades:

- Si $S_1 \neq S_2$, entonces las dos funciones *no son equivalentes* con certeza.

- Si $S_1 = S_2$, entonces las funciones son *equivalentes* con una probabilidad de error muy pequeña, que se puede reducir incrementando el tamaño del campo o realizando varias ejecuciones, utilizando en cada ejecución un conjunto independiente de asignaciones de variables aleatoriamente seleccionadas y calculando las signaturas de las funciones a verificar.

5.2. Interpretación algebraica de un ROBDD

Para un nodo *no terminal* v perteneciente a un ROBDD, sean $low(v)$ y $high(v)$ su *0-sucesor* y su *1-sucesor*, respectivamente, y sea x_v la variable simbólica asociada con el nodo v. Se ha visto en el capítulo 4 que la función Booleana representada por un ROBDD se puede definir recursivamente en

términos de f^v, la función Booleana representada por el subgrafo rutado en el nodo no terminal v, de la siguiente forma

$$f^v = [\overline{x}_v \wedge f^{low(v)}] \vee [x_v \wedge f^{high(v)}] \tag{5.1}$$

teniéndose que si v es un nodo *terminal*, entonces f^v simplemente toma el valor *cierto* (**1**) o el valor *falso* (**0**).

Ahora se puede aplicar la *transformada* $\mathcal{A}$ a la expresión anterior. Para ello, se observa que los dos términos de la disyunción son claramente ortogonales, por lo que se puede aplicar el teorema de *ortogonalidad* visto en el capítulo 2, obteniéndose

$$\mathcal{A}[f^v] = \mathcal{A}[(1 - x_v) \cdot f^{low(v)}] + \mathcal{A}[x_v \cdot f^{high(v)}] \tag{5.2}$$

donde se han utilizado las 0,1-*equivalencias* y las operaciones Booleanas extendidas dadas en la tabla 2.2 del capítulo 2. Además, en esta expresión ya aparecen operaciones aritméticas definidas sobre un campo finito (*suma*, *multiplicación* y *resta*) en vez de operaciones Booleanas (*disyunción*, *conjunción* y *complemento*).

La ecuación 5.2 se puede simplificar si el BDD es *libre*, es decir, si cada variable aparece solamente una vez a lo largo de cualquier camino que transcurra desde el nodo raíz hasta un nodo terminal. Pero esta propiedad se cumple en los ROBDDs, ya que constituyen un caso especial de BDDs libres. Por tanto, una variable x_v no aparecerá en los sub-ROBDDs, *low(v)* y *high(v)*, apuntados por las *0,1-aristas* del nodo v y, por tanto, las subfunciones correspondientes $f^{low(v)}$ (*cofactor negativo*) y $f^{high(v)}$ (*cofactor positivo*) no dependerán de la variable x_v. Como consecuencia, las transformadas $\mathcal{A}$ que aparecen en el segundo término de la ecuación 5.2 se calculan sobre el producto de dos funciones que no tienen ninguna variable en común $(1 - x_v) \cdot f^{low(v)}$ y $x_v \cdot f^{high(v)}$, por lo que se puede aplicar el teorema de *soporte disjunto* para obtener

$$\mathcal{A}[f^v] = (1 - x_v) \cdot \mathcal{A}[f^{low(v)}] + x_v \cdot \mathcal{A}[f^{high(v)}] \tag{5.3}$$

donde se han utilizado las reglas vistas en el teorema 5 del capítulo 2 y donde se observa que esta expresión coincide con el teorema 4 de *expansión lineal*.

5.3. Cálculo de la signatura de un ROBDD

Con la ecuación 5.3 se puede calcular el polinomio simbólico dado por la *transformada* $\mathcal{A}$ de la función f^v representada por el nodo no terminal v de un ROBDD, en función de las transformadas de sus *cofactores negativo* y *positivo*. Pero el objetivo no es la obtención de un polinomio, sino de una *signatura* que represente a la función, y que será el valor del polinomio calculado evaluado para una asignación aleatoria de sus variables.

Como se vio en el capítulo 2, la introducción de valores numéricos en una representación algebraica consiste en la sustitución del valor dado por el símbolo equivalente y en la realización de las simplificaciones correspondientes. Un *conjunto de restricción* ρ determina el conjunto de variables que son sustituidas por sus valores numéricos aleatorios correspondientes. Si el conjunto de restricción ρ está formado por todas las variables de las que depende la función representada por un ROBDD, se puede calcular el valor numérico de las representaciones algebraicas dadas por la ecuación 5.3 para todos los nodos del ROBDD aplicando la restricción al conjunto ρ, es decir

$$\mathcal{A}_{\{\rho\}}[f^v] = (1 - x_v)_{\{\rho\}} \cdot \mathcal{A}_{\{\rho\}}[f^{low(v)}] + (x_v)_{\{\rho\}} \cdot \mathcal{A}_{\{\rho\}}[f^{high(v)}] \tag{5.4}$$

Se observa que en la ecuación 5.4 se combinan los valores numéricos de los términos involucrados sin realizar ninguna consideración especial acerca de la existencia de variables comunes a dichos términos. Esto es posible debido a que la expresión 5.3 verifica el teorema 7 de *combinación de soporte disjunto* y el teorema 8 de *ortogonalidad*, vistos en el capítulo 2.

Partiendo de la ecuación 5.4, si $\mathbf{x_v}$ representa el valor numérico aleatorio asignado a la variable x_v, y $\mathbf{f^{low(v)}}$ y $\mathbf{f^{high(v)}}$ representan los valores numéricos (*signaturas*) obtenidos al realizar las restricciones $\mathcal{A}_{\{\rho\}}[f^{low(v)}]$ y $\mathcal{A}_{\{\rho\}}[f^{high(v)}]$, respectivamente, entonces la *signatura* $\mathbf{f^v}$ obtenida para la restricción $\mathcal{A}_{\{\rho\}}[f^v]$ vendrá dada por la expresión

$$\mathbf{f^v} = (1 - \mathbf{x_v}) \cdot \mathbf{f^{low(v)}} + \mathbf{x_v} \cdot \mathbf{f^{high(v)}} \tag{5.5}$$

Esta ecuación determina la signatura de un nodo v de un ROBDD en función del valor numérico asignado a la variable x_v asociada a dicho nodo, y en función de las signaturas de sus cofactores negativo $f^{low(v)}$ y positivo $f^{high(v)}$. Por tanto, se puede calcular de forma recursiva la *signatura* del ROBDD que representa a una función Booleana f, siendo dicha signatura el valor numérico obtenido para el *nodo raíz* del ROBDD.

5.4. ROBDDs con valores numéricos de $GF(2^m)$

En la representación algebraica dada por la ecuación 5.3 aparecen operaciones aritméticas sobre un campo finito, pero nosotros utilizamos el campo de *Galois* $GF(2^m)$, por lo que la operación de resta que aparece en esta ecuación se sustituye por la operación de suma (teniendo en cuenta las propiedades vistas en la sección 5.1), obteniendo la siguiente expresión

$$\mathcal{A}[f^v] = (1 + x_v) \cdot \mathcal{A}[f^{low(v)}] + x_v \cdot \mathcal{A}[f^{high(v)}] \tag{5.6}$$

donde aparecen sumas y multiplicaciones definidas sobre el campo $GF(2^m)$ seleccionado. La ecuación 5.5 también se modificará en consecuencia

$$\mathbf{f^v} = (1 + \mathbf{x_v}) \cdot \mathbf{f^{low(v)}} + \mathbf{x_v} \cdot \mathbf{f^{high(v)}} \tag{5.7}$$

Como se ha visto en la sección 5.1, en un campo finito $GF(2^m)$ los elementos se representan como vectores binarios de m bits, por lo que en la ecuación 5.7 aparecen sumas y multiplicaciones de vectores binarios. La *adición* de dos elementos del campo es simplemente la OR-exclusiva bit a bit de sus vectores binarios correspondientes. En esta primera parte de la tesis, utilizamos la *base polinómica* de representación de los elementos del campo, en la que el elemento *unidad* se representa por un vector de m bits todos cero a excepción del bit de menor peso, que toma el valor 1 (para $m = 16$, el elemento unidad es el vector 0000000000000001). Por lo tanto, la operación $(1 + \mathbf{x_v})$ que aparece en la ecuación 5.7 se reduce simplemente al complemento del bit de menor peso de la representación binaria de $\mathbf{x_v}$. Con respecto a la *multiplicación* de vectores binarios de m bits (elementos del campo $GF(2^m)$), asumimos que se utiliza *un* algoritmo de multiplicación en base canónica, sin profundizar en la estructura de dicho algoritmo. Es en la segunda parte de la tesis donde se estudian éste (subsección 7.2.1) y otros tipos de multiplicación sobre los campos de Galois.

5.4.1. Aspectos de implementación

Para la realización de nuestra aproximación híbrida de verificación, es necesaria la utilización de ROBDDs intermedios que incluyan la *signatura* de las (sub)funciones que representan en la propia estructura de datos del ROBDD. La signatura de una función será la signatura del nodo (o los nodos) *raíz* de su ROBDD. Por comodidad, vamos a llamar a estos ROBDDs con valores *numéricos* como *signatura*-ROBDDs o *s*-ROBDDs, y vamos a utilizar el paquete CUDD de la Universidad de Colorado para su implementación.

Utilizando la ecuación 5.7, se observa que la *signatura* $\mathbf{f^v}$ de la función representada por el nodo no terminal v de un ROBDD, se puede calcular conociendo el valor aleatorio $\mathbf{x_v}$ asignado a la variable x_v asociada al nodo v. El valor de $\mathbf{f^v}$ se calcula complementando el bit menos significativo de la representación binaria de $\mathbf{x_v}$, multiplicándolo por la signatura $\mathbf{f^{low(v)}}$ del cofactor negativo de v y sumando el producto de $\mathbf{x_v}$ por la signatura $\mathbf{f^{high(v)}}$ del cofactor positivo de v. En este cálculo, las operaciones aritméticas de suma y multiplicación en el campo $GF(2^m)$ se realizan sobre vectores de m bits.

Un nodo típico de un ROBDD consta de una serie de *campos*, entre los que se encuentran el campo *índice* (que hace referencia a la variable asociada con ese nodo) y los campos correspondientes al *0-sucesor* y al *1-sucesor* (punteros a los cofactores negativo y positivo, respectivamente). La inclusión de *signaturas* en un ROBDD (obteniendo un *s*-ROBDD), se realiza incluyendo un campo de *signatura* adicional de $2m$ bits en cada nodo v del ROBDD, estando este campo *signatura* dividido en dos subcampos de igual tamaño (m). Uno de los subcampos se utiliza para el almacenamiento de la signatura aleatoriamente asignada a la variable asociada con ese nodo ($\mathbf{x_v}$), mientras que el otro almacena la signatura de la función representada por dicho nodo ($\mathbf{f^v}$).

La asignación aleatoria de signaturas a las variables de entrada de la función se realiza previamente a la creación del s-ROBDD, por lo que para un nodo v, el subcampo correspondiente a la signatura asignada a la variable asociada a dicho nodo ($\mathbf{x_v}$) se almacena inicialmente cuando se crea ese nodo, mientras que el subcampo que almacena la signatura de la función representada por el nodo v ($\mathbf{f^v}$) se calcula durante la fase de *síntesis* (construcción) por la aplicación recursiva del operador *ite*, como se vio en las ecuaciones 4.4 y 4.5 y en la figura 4.3 del capítulo 4, haciendo uso de la expresión 5.7. De esta forma, el cálculo de las signaturas se realiza en la propia fase de síntesis del s-ROBDD (utilizando el operador *ite*), obteniéndose un ROBDD con la signatura de la función representada por cada nodo almacenada en dichos nodos. La signatura, por tanto, de una función representada por un s-ROBDD estará almacenada en el campo *signatura* de su nodo raíz.

5.5. Aproximación híbrida de verificación

En la aproximación *híbrida* de verificación [IDB97] combinamos los métodos probabilista y determinista de forma que mientras no existan restricciones de espacio o tiempo, se utiliza la verificación determinista por medio de la creación de s-ROBDDs, pero si existe alguna limitación entonces se aplica la verificación probabilista donde se emplean las signaturas ya calculadas por los s-ROBDDs construidos hasta ese momento para la determinación de las signaturas de las funciones a verificar. Además, el campo finito a utilizar para el cálculo de dichas signaturas es un campo *de Galois* $GF(2^m)$.

Esta aproximación híbrida se aplica a funciones Booleanas representadas como circuitos lógicos combinacionales de dos niveles expresados en forma *cúbica* (SOP). Como se vio en el capítulo 3, una representación en forma SOP es la *disyunción* Booleana de la *conjunción* de literales, también llamados términos producto o *cubos*. En el capítulo 2 también se vio que el cálculo de la transformada $\mathcal{A}$ (y por tanto de la signatura) se simplifica cuando se cumple la propiedad de *ortogonalidad*.

5.5.1. Justificación

Si se tiene una función Booleana f expresada por un conjunto de p cubos, $\{f\} = \{f^0, f^1, \ldots, f^{p-1}\}$, esta función viene representada por la expresión

$$f = f^0 \vee f^1 \vee \cdots \vee f^{p-1} \tag{5.8}$$

Consideremos que el conjunto de cubos de f cumplen la propiedad de *disjunción*. Al ser un conjunto de cubos *disjuntos*, no presentan mintérminos comunes, por lo que cumplen la propiedad de *ortogonalidad*, es decir, $f^i \wedge f^j = 0$, $\forall i, j \in \{0, \ldots, p-1\}$.

Si se aplica la transformada $\mathcal{A}$ (teniendo en cuenta lo visto en el capítulo 2) a la disyunción de cubos dada por la ecuación 5.8, se puede utilizar el teorema 8 de ortogonalidad, obteniendo

$$\mathcal{A}[f] = \mathcal{A}[f^0 \vee f^1 \vee \cdots \vee f^{p-1}] = \mathcal{A}[f^0] + \mathcal{A}[f^1] + \cdots + \mathcal{A}[f^{p-1}] \tag{5.9}$$

Como se vio en el capítulo 3, un *cubo* es una *conjunción* de literales. Si $X = \{x_{n-1}, ..., x_1, x_0\}$ es el conjunto de variables de las que depende la función f (es decir, el *soporte* de f), un cubo f^i de f se puede representar por la siguiente conjunción de literales

$$f^i = \bigwedge_{j \in D} x_j^{\nu_j} \tag{5.10}$$

donde $D \subset \{0, 1, \ldots, n-1\}$ es un subconjunto de índices que identifican el *soporte* del cubo f^i, $x_j \in X$ y $\nu_j \in \{0, 1\}$ tal que se cumple que

$$x_j^{\nu_j} = \begin{cases} x_j & si \quad \nu_j = 1 \\ \overline{x}_j & si \quad \nu_j = 0 \end{cases} \tag{5.11}$$

Por lo tanto, la ecuación 5.10 representa un cubo como una conjunción de literales pertenecientes a su soporte. Por ejemplo, si $X = \{x_3, x_2, x_1, x_0\}$ es el conjunto de variables de las que depende la función f y f^0 es un cubo de la función f, $f^0 \in \{f\}$, de forma que $f^0 = x_3 \wedge \overline{x}_1 \wedge x_0$, se tiene que $soporte(f^0) = \{x_3, x_1, x_0\} \subset X$, $D = \{3, 1, 0\}$ y $(\nu_3, \nu_1, \nu_0) = (1, 0, 1)$.

En la ecuación 5.9 se realiza la suma de las transformadas $\mathcal{A}$ de los cubos de la función f. Como se vio en la ecuación 5.10, un cubo viene dado por la conjunción de literales pertenecientes a su soporte, donde no aparece ningún literal *repetido*. Por lo tanto, es claro que un cubo se puede considerar como una conjunción de subfunciones (sus *literales*) que presentan *soporte disjunto* (no tienen variables en común), y para el cálculo de su transformada $\mathcal{A}$ se puede aplicar el teorema 3 de *soporte disjunto*, con lo que se puede escribir

$$\mathcal{A}[f^i] = \mathcal{A}[\bigwedge_{j \in D} x_j^{\nu_j}] = \prod_{j \in D} \mathcal{A}[x_j^{\nu_j}] \tag{5.12}$$

donde en el último término de esta ecuación se realizan *productos* sobre el campo de Galois $GF(2^m)$ y para la obtención de esta expresión se han utilizado las operaciones Booleanas extendidas dadas en el capítulo 2.

El cálculo de las transformadas $\mathcal{A}$ de los literales que aparecen en esta ecuación se puede realizar utilizando el teorema 5, las operaciones Booleanas extendidas y la expresión 5.11 de la siguiente forma

$$\mathcal{A}[x_j^{\nu_j}] = \begin{cases} \mathcal{A}[x_j] = x_j & si \quad \nu_j = 1 \\ \mathcal{A}[\overline{x}_j] = 1 + x_j & si \quad \nu_j = 0 \end{cases} \tag{5.13}$$

donde se ha tenido en cuenta que en un campo de Galois, la resta se sustituye por la operación de suma.

Utilizando la ecuación 5.9 para el cálculo de la transformada $\mathcal{A}$ de una función f dada por su conjunto de cubos *disjuntos* y utilizando las ecuaciones 5.12 y 5.13 para el cálculo de la transformada $\mathcal{A}$ de un cubo y de un literal, respectivamente, se puede obtener la expresión polinómica correspondiente a la función f definida sobre el campo $GF(2^m)$. Sin embargo, en lo que estamos interesados es en el cálculo de la *signatura*, no en el polinomio simbólico. Por lo tanto, se puede aplicar el teorema 7 de *combinación de soporte disjunto* y el teorema 8 de *ortogonalidad*, de forma que para cualquier conjunto de restricción ρ, se obtenga

$$\mathcal{A}_{\{\rho\}}[f] = \mathcal{A}_{\{\rho\}}[f^0 \vee \cdots \vee f^{p-1}] = \mathcal{A}_{\{\rho\}}[f^0] + \cdots + \mathcal{A}_{\{\rho\}}[f^{p-1}] \tag{5.14}$$

donde las operaciones aritméticas que aparecen en esta expresión se realizan sobre un campo de Galois $GF(2^m)$. En particular, se puede calcular la *signatura* de la función f si el conjunto de restricción ρ es el total de las variables de las que depende f, es decir, si $\rho = X$.

En la ecuación 5.14, las restricciones $\mathcal{A}_{\{\rho\}}[f^i]$, $\forall i \in \{0, \ldots, p-1\}$, se pueden calcular realizando las restricciones de las transformadas $\mathcal{A}$ de los literales que componen cada uno de los cubos, según la ecuación 5.12

$$\mathcal{A}_{\{\rho\}}[f^i] = \mathcal{A}_{\{\rho\}}[\bigwedge_{j \in D} x_j^{\nu_j}] = \prod_{j \in D} \mathcal{A}_{\{\rho\}}[x_j^{\nu_j}] \tag{5.15}$$

Si consideramos que $\rho = X$ y que $\mathbf{x_j}$ representa el valor numérico aleatorio, perteneciente a $GF(2^m)$, asignado a la variable x_j, entonces se puede calcular $\mathcal{A}_{\{\rho\}}[x_j^{\nu_j}]$ utilizando la ecuación 5.13 de la siguiente forma

$$\mathcal{A}_{\{\rho\}}[x_j^{\nu_j}] = \begin{cases} \mathcal{A}_{\{\rho\}}[x_j] = \mathbf{x_j} & si \quad \nu_j = 1 \\ \mathcal{A}_{\{\rho\}}[\overline{x}_j] = 1 + \mathbf{x_j} & si \quad \nu_j = 0 \end{cases} \tag{5.16}$$

Si $\mathbf{f}$ representa la signatura obtenida al realizar la restricción $\mathcal{A}_{\{\rho\}}[f]$ y si $\mathbf{f^i}$ representa la signatura obtenida de la restricción $\mathcal{A}_{\{\rho\}}[f^i]$, $\forall i \in \{0, \ldots, p-1\}$, la ecuación 5.14 será finalmente

$$\mathbf{f} = \mathbf{f^0} + \mathbf{f^1} + \cdots + \mathbf{f^{p-1}} \tag{5.17}$$

donde las signaturas $\mathbf{f^i}$ se calculan utilizando las ecuaciones 5.15 y 5.16 y donde las operaciones aritméticas de suma y multiplicación requeridas se realizan sobre vectores binarios de m bits que representan elementos del campo de Galois $GF(2^m)$ seleccionado.

5.5.2. Método híbrido

Para la obtención de la ecuación 5.17, se ha supuesto que el conjunto de cubos que representan a la función Booleana f cumplen la propiedad de *disjunción*, con lo que la ecuación 5.8 sería una suma de productos disjuntos o DSOP. Si esta propiedad de disjunción no se cumple (es decir, si la ecuación 5.8 representa una suma de productos no disjuntos o SOP), las consideraciones y simplificaciones vistas anteriormente no se podrían aplicar, por lo que el cálculo de la signatura **f** no sería tan sencillo como la simple aplicación de la ecuación 5.17. Por este motivo, para nuestra aproximación híbrida de verificación, se utiliza la formulación vista en el capítulo 3, con la que a partir de un circuito combinacional de dos niveles representado en forma de sumas de productos (SOPs) no disjuntos, se obtiene un formato equivalente de representación en forma de sumas de productos disjuntos (DSOPs). Esta representación DSOP obtenida aplicando la formulación vista en el capítulo 3 es *verificada híbridamente* con la representación DSOP dada por el programa DJ. Ambas proporcionan representaciones diferentes de un mismo circuito y lo que tratamos es de *verificar* su equivalencia utilizando un método híbrido en el que se puedan utilizar tanto signaturas como diagramas de decisión.

La *aproximación híbrida de verificación* [IDB97] combina las verificaciones probabilista y determinista por medio del cálculo de signaturas *al mismo tiempo* que se construyen los ROBDDs de las funciones, es decir, por medio de la creación de los s-ROBDDs. Para ello, se habrá realizado previamente la asignación aleatoria de signaturas a las variables de entrada de las funciones. El método se inicia construyendo los s-ROBDDs de las funciones a verificar, y las alternativas posibles serían las siguientes:

- Si no existe ninguna limitación de espacio de memoria o de tiempo, en realidad se realiza un verificación determinista ya que se construyen completamente los s-ROBDDs de las funciones y la comparación se realiza a través de ellos por medio de la comparación de los punteros a sus nodos raíz. En este caso las signaturas no se utilizan para la comparación.

- Si existe alguna limitación espacial, los s-ROBDDs de las funciones se construyen hasta que se alcanza ese límite. A partir de ese momento se calculan *únicamente* signaturas, utilizándose las signaturas ya calculadas (y almacenadas) en los s-ROBDDs de las subfunciones procesadas para la obtención de las signaturas totales de las funciones.

- Si existe limitación de tiempo para la verificación, también se puede seleccionar entre las dos aproximaciones, teniendo en cuenta que el cálculo exclusivo de signaturas presenta un mejor rendimiento temporal que el dado por la construcción de grafos, además de tener un consumo espacial constante, determinado por el número de salidas de los circuitos.

El formato de datos de los circuitos a verificar es un formato PLA de dos niveles, en el que cada entrada (o línea) del archivo indica los valores que toman las variables de entrada que hacen que la función tome el valor lógico *cierto*, así como las salidas que toman el valor *cierto* para esa combinación de valores de las entradas. Es decir, cada entrada del archivo especifica un *cubo* y los valores que toman las salidas para ese cubo, por lo que el número de entradas del archivo indica el número total de cubos (o términos producto) que determinan la funcionalidad del circuito.

```
.i 5
.o 3
-11-0  100
--001  100
-1101  100
--011  010
-1111  010
1011-  010
---01  010
0011-  010
-1--0  010
-0101  001
-10-0  001
.e
```

Figura 5.1: Ejemplo de archivo en formato PLA.

En la figura 5.1 se muestra el archivo en formato PLA de un circuito ejemplo. Las dos primeras entradas del archivo indican el número de variables de entrada (5) y el número de salidas (3) del circuito. El resto de las entradas del archivo están formadas por 8 columnas de datos. Los primeros 5 datos (tantos como variables de entrada tenga el circuito) especifican un *cubo*, donde el valor 0 significa que la variable correspondiente a su posición aparece complementada, el valor 1 indica que esa variable aparece sin complementar, y el valor - (*don't care* o indeterminación) indica que la variable correspondiente no aparece en el término producto. A su vez, los últimos 3 datos (tantos como salidas tenga el circuito) indican el valor lógico que toman las salidas para ese cubo. En este ejemplo, si las variables de entrada son $\{x_0, x_1, x_2, x_3, x_4\}$ y las salidas son $\{g_0, g_1, g_2\}$ y en ese orden, la primera entrada de datos que aparece corresponde a `-11-0 100`, con lo que el cubo representado sería $x_1 x_2 \overline{x}_4$ y la salida para la cual este cubo formaría parte de sus términos producto sería g_0. Recorriendo secuencialmente el archivo desde el principio, tendríamos, por ejemplo, que la salida g_0 vendría dada por la expresión en forma de suma de productos $g_0 = x_1 x_2 \overline{x}_4 + \overline{x}_2 \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 x_4$.

El método *híbrido* de verificación comienza, por tanto, recorriendo de forma secuencial los archivos en formato PLA de los circuitos a verificar y construyendo sus s-ROBDDs. El s-ROBDD para un cubo se obtiene realizando la *conjunción* (AND o producto lógico) de los s-ROBDDs de las variables que aparecen en su soporte (complementadas o sin complementar) por medio de la utilización del operador *ite*, como se ha visto en la subsección 5.4.1. El cálculo de la signatura es posible debido a que el s-ROBDD de cada variable es creado con su valor numérico aleatoriamente asignado ya incluido en su campo *signatura*. La signatura correspondiente a un cubo f^i, representada como $\mathbf{f^i}$, estará almacenada en el campo *signatura* del nodo raíz de su s-ROBDD, como se vio anteriormente en la sección 5.4. Finalmente, los s-ROBDDs correspondientes a las salidas del circuito se construyen realizando la *disyunción* (OR o suma lógica) de los s-ROBDDs de los cubos para los cuales esas salidas toman el valor lógico **1**.

Se puede observar que en el caso concreto de circuitos que estamos utilizando, dados en forma de sumas de productos disjuntos (DSOP), se cumple que la *disyunción* puede ser sustituida por la *disyunción exclusiva* sin que se modifique la función. Esto se puede demostrar teniendo en cuenta que la OR se puede representar en función de la XOR como $f^i \vee f^j = f^i \oplus f^j \oplus (f^i \wedge f^j)$, donde f^i y f^j representan dos cubos (o funciones). Por tanto, si $f^i \wedge f^j = 0$ (disjuntos), entonces se tiene que $f^i \vee f^j = f^i \oplus f^j$. Esta prueba se puede generalizar por inducción para el caso en que se tengan p cubos, obteniéndose

$$f = f^0 \vee f^1 \vee \cdots \vee f^{p-1} = f^0 \oplus f^1 \oplus \cdots \oplus f^{p-1} \tag{5.18}$$

cuando los cubos cumplan la propiedad de *disjunción*. Por lo tanto, para la construcción de los s-ROBDDs de las salidas del circuito, se puede realizar la *disyunción exclusiva* (XOR) en lugar de la *disyunción* (OR) de los s-ROBDDs de los cubos correspondientes, obteniéndose los mismos resultados.

Si no existe ninguna restricción de espacio para la verificación, entonces se recorren completamente los circuitos en formato PLA y se obtienen los s-ROBDDs completos de las funciones a verificar. En este caso, se estaría realizando una verificación determinista ya que se utilizarían los s-ROBDDs para la comprobación de la equivalencia por medio de la comparación de los punteros (direcciones de memoria) a los nodos raíz de dichos grafos. Si los punteros son iguales, entonces los circuitos son equivalentes, y si los punteros son distintos, entonces los circuitos realizan diferentes funcionalidades. Las signaturas calculadas (e incluidas en la propia estructura de datos del s-ROBDD) no se utilizarían en este caso, y su cálculo y almacenamiento llevarían consigo una penalización tanto de tiempo como de espacio con respecto a la verificación determinista *clásica* con ROBDDs *puros*. Sin embargo, la posibilidad de realizar una verificación probabilista de los circuitos (basada en signaturas) cuando el tamaño de los grafos construidos sea lo suficientemente grande, justificaría este coste adicional.

Si existe un límite de espacio (determinado por un tamaño máximo para los grafos construidos), entonces se recorre secuencialmente el circuito y se van construyendo los s-ROBDDs de las funciones hasta que se alcance dicho límite. Si una de las funciones tiene p cubos y si hasta ese momento se ha construido el s-ROBDD de la disyunción de q cubos ($q < p$), entonces el nodo raíz de este s-ROBDD tiene almacenada la signatura correspondiente a la suma de las signaturas de esos q cubos, es decir, $\mathbf{f^0} + \mathbf{f^1} + \cdots + \mathbf{f^{q-1}}$. Faltaría calcular la suma de las signaturas de los $(p - q)$ cubos restantes. Esto se realizaría continuando con el recorrido del archivo PLA y calculando las signaturas de los cubos restantes y realizando su suma, como se vio en la subsección anterior. En este caso, no se construyen los s-ROBDDs sino que únicamente se realizan productos y sumas de signaturas representadas por vectores de m bits que representan elementos del campo $GF(2^m)$. Como se vio anteriormente, la suma de estos vectores es la XOR bit a bit de los mismos, y la multiplicación se realiza usando un algoritmo de multiplicación en base polinómica sobre $GF(2^m)$. El coste de almacenamiento en este caso sería de m bits por cada salida del circuito, por lo que el espacio utilizado sería una cantidad fija y mucho menor que el necesitado para la construcción de los s-ROBDDs. La signatura final del circuito (ecuación 5.17) se obtendría realizando la suma de las signaturas obtenidas con los s-ROBDDs ($\mathbf{f^0} + \mathbf{f^1} + \cdots + \mathbf{f^{q-1}}$) y de las signaturas restantes ($\mathbf{f^q} + \mathbf{f^{q+1}} + \cdots + \mathbf{f^{p-1}}$) obtenidas realizando operaciones aritméticas directamente sobre las signaturas de las variables. La comprobación de equivalencia se realizaría en este caso comparando las signaturas de los circuitos a verificar, y actuando según los criterios vistos en la sección 5.1.

El cálculo exclusivo de signaturas (sin construcción de s-ROBDDs) presenta también un mejor rendimiento temporal, por lo que se podría seleccionar este método en el caso de existir alguna limitación de tiempo para la verificación, teniendo también en cuenta que de esta forma el consumo de espacio sería constante y estaría determinado por el número de salidas de los circuitos.

5.5.3. Resultados experimentales

Se ha realizado la implementación en código C del método híbrido de verificación, para lo cual se ha utilizado el entorno CUDD modificado de la Universidad de Colorado para la construcción de los s-ROBDDs. Para la comprobación del método, se ha utilizado la formulación vista en el capítulo 3 en la que a partir de los benchmarks LGSynth91 en formato PLA de dos niveles, se podían obtener sus expresiones en forma DSOP. En nuestros experimentos, utilizamos algunos de estos benchmarks para comprobar la equivalencia de sus representaciones en forma DSOP obtenidas utilizando el corolario 1 y las obtenidas utilizando el programa DJ. Ya se vio que ambas constituyen representaciones diferentes de los mismos circuitos, por lo que tratamos de *verificar* su equivalencia utilizando el método *híbrido*.

Para el cálculo de las signaturas se ha utilizado como campo finito el campo de Galois $GF(2^{16})$ generado por el polinomio irreducible de grado 16 $f(x) = x^{16} + x^5 + x^3 + x^2 + 1$. La base de representación de los elementos del campo utilizada ha sido la *base polinómica* y la multiplicación de elementos se ha realizado usando un algoritmo de multiplicación en dicha base. Asimismo, se ha asignado a cada variable de entrada de los circuitos a verificar un valor numérico aleatorio generado de forma independiente y uniformemente distribuido (seleccionando el valor 1 como semilla de generación).

La verificación *probabilista*, *determinista* e *híbrida* de los benchmarks se ha realizado sobre una estación de trabajo SUN UltraSparc-II. En la verificación probabilista se ha realizado la comparación utilizando exclusivamente las *signaturas* de los circuitos, mientras que en la verificación determinista la comparación se ha realizado a traves de sus s-ROBDDs. En la verificación híbrida, se han establecido ciertos límites para los tamaños de los s-ROBDDs construidos. Si se alcanza ese límite, entonces se continúa con la verificación probabilista de los circuitos.

Para los tipos de verificación en los que es necesaria la construcción de los s-ROBDDs, hemos realizado experimentos considerando dos aproximaciones relativas a la ordenación de variables: usando la ordenación original fija dada por el circuito de benchmark y utilizando una aproximación dinámica de *reordenación* de variables para la construcción. Ya se vio en el capítulo 4 que existen métodos dinámicos de búsqueda de ordenaciones de variables óptimas. En nuestros experimentos, utilizamos el algoritmo de *reordenación por desplazamiento* de Rudell [Rud93], modificándolo con el objeto de permitir la inclusión y cálculo de signaturas.

5.5.3.1. Sin reordenación de variables

En la tabla 5.1 se muestran los tiempos de ejecución (en segundos de CPU) obtenidos para la verificación probabilista y determinista, realizadas de forma independiente y donde en ambos casos se comprobó la equivalencia de las representaciones DSOP generadas por el corolario 1 y por el programa DJ de los benchmarks dados. Para el caso de la verificación probabilista se hizo una ejecución, obteniendo las mismas signaturas para los dos circuitos, con lo que se aseguraba la equivalencia con una probabilidad muy baja de error (como se vio en el capítulo 2, este error se podría reducir, por ejemplo, realizando más ejecuciones con distintas asignaciones aleatorias).

En los experimentos se observó que cualquier modificación de los circuitos producía signaturas diferentes, con lo que el método advertía de su *no equivalencia*. En la verificación determinista, la comparación se realizó construyendo los s-ROBDDs completos de los circuitos, y comprobando que los punteros de sus nodos raíz coincidían (en caso de no ser circuitos equivalentes, estas direcciones serían distintas).

De los resultados dados en la tabla 5.1, se observa que los tiempos de ejecución son mucho menores para la verificación probabilista que para la determinista. Además, no se pudieron construir los diagramas de decisión para el circuito de benchmark *dalu*, por lo que únicamente se pudo verificar de manera probabilista. Considerando los tiempos totales de ejecución de cada método para los benchmarks dados (donde no se incluye en ese total el tiempo obtenido para el circuito *dalu*), se observa que el tiempo de ejecución consumido por el método probabilista es un 85.2 % menor que el necesitado para la verificación determinista. La mayor diferencia corresponde al circuito *apex*6, para el que la verificación probabilista es un 95 % más rápida que la determinista.

Circuito	Tprob	Tdet
9sym	0.05	0.21
add6	0.28	1.75
alu2	0.03	0.17
apex4	0.42	2.78
apex6	0.62	12.36
apex7	1.78	9.83
count	0.10	0.69
dalu	2.05	n.p.
duke2	0.09	0.78
e64	0.08	1.16
ex1010	1.35	8.22
ex5	1.75	8.11
exp	0.12	0.53
gary	0.08	0.63
rd84	0.10	0.44
sao2	0.05	0.26
sym10	0.23	1.02
t481	1.09	5.71
vg2	0.26	3.12
z5xp1	0.11	0.48
z9sym	0.12	0.50
Total(†)	8.71	58.75

Tabla 5.1: Tiempos de ejecución obtenidos en las verificaciones probabilista y determinista (*n.p.* = no posible)(† = sin incluir el circuito *dalu*).

Con respecto al consumo de memoria necesario para la verificación *probabilista*, se utiliza una cantidad fija de memoria necesaria para el almacenamiento de las signaturas de las salidas de los circuitos a comparar. Para los benchmarks comprobados, se ha obtenido un consumo relativo de memoria para cada uno de ellos de 84KB, con lo que el coste total de memoria para todos los circuitos sería de 1680KB (sin la inclusión del circuito *dalu*).

El consumo de espacio necesario para la verificación *determinista* se puede observar en la tabla 5.2, donde para cada circuito se muestra el tamaño (número de nodos) final alcanzado, el número máximo de nodos obtenidos durante la construcción, así como la cantidad relativa de memoria necesitada (expresada en Kilobytes).

Circuito	Tamaño	Máximo	Memoria
9sym	25	4088	508
add6	309	7154	588
alu2	168	3066	484
apex4	928	5110	540
apex6	2760	14308	948
apex7	1660	17374	884
count	234	7154	612
dalu	n.p.	n.p.	n.p.
duke2	973	8176	628
e64	1441	22484	1004
ex1010	1067	5110	540
ex5	268	3066	484
exp	210	2044	460
gary	518	6132	564
rd84	42	3066	484
sao2	155	4088	508
sym10	31	3066	484
t481	21	6132	564
vg2	1044	12264	740
z5xp1	42	3066	484
z9sym	25	4088	508
Total(†)	11921	141036	12016

Tabla 5.2: Estadísticas de almacenamiento obtenidas en la verificación determinista (*n.p.* = no posible)(† = sin incluir el circuito *dalu*).

De los resultados mostrados en la tabla 5.2 se puede observar que la memoria total necesaria para la construcción de todos los circuitos de benchmark es de 12016KB (de nuevo sin incluir el circuito *dalu*, que no pudo ser construido). Por lo tanto, se observa que el consumo de memoria requerido si se utiliza la aproximación probabilista es un 86 % menor que si se utiliza la verificación determinista. El mejor comportamiento corresponde al benchmark *e*64, para el que el método probabilista ocupa un 91.6 % menos de memoria que la aproximación determinista. Además, no se pudo construir el *s*-ROBDD del circuito de benchmark *dalu*, no pudiendo por tanto ser verificado de forma determinista y por lo que únicamente se pudo verificar de manera probabilista a través de la comparación de signaturas.

En la verificación *híbrida* se han establecido límites para los tamaños de los grafos construidos. Si los diagramas de decisión alcanzan esos límites durante su construcción, entonces se detiene la síntesis de los circuitos y se pasan a calcular signaturas exclusivamente (utilizando las signaturas ya calculadas y almacenadas en los s-ROBDDs construidos), con lo que se realizaría la verificación probabilista únicamente. Para nuestros experimentos, hemos establecido diferentes límites en el número de nodos del s-ROBDD (aunque también se podrían haber establecido en el número máximo de nodos creados durante la síntesis o en la cantidad de memoria total ocupada) y se han determinado los tiempos y tamaños obtenidos para la verificación.

Circuito	L100		L500		L1000		L1500		L3000		L10000	
	T	V	T	V	T	V	T	V	T	V	T	V
9sym	0.18	P	0.21	D								
add6	0.92	P	1.75	D								
alu2	0.08	P	0.17	D								
apex4	1.26	P	2.24	P	2.78	D						
apex6	4.61	P	5.46	P	6.79	P	7.58	P	9.46	P	12.36	D
apex7	4.06	P	6.09	P	8.46	P	9.01	P	9.83	D		
count	0.41	P	0.69	D								
dalu	6.27	P	12.05	P	14.22	P	16.03	P	25.30	P	27.83	P
duke2	0.30	P	0.47	P	0.63	P	0.78	D				
e64	0.26	P	0.41	P	0.65	P	1.16	D				
ex1010	4.36	P	5.97	P	5.46	P	8.22	D				
ex5	4.63	P	8.11	D								
exp	0.32	P	0.53	D								
gary	0.37	P	0.55	P	0.63	D						
rd84	0.44	D										
sao2	0.14	P	0.26	D								
sym10	1.02	D										
t481	5.71	D										
vg2	0.98	P	1.85	P	1.86	P	2.13	P	3.12	D		
z5xp1	0.48	D										
z9sym	0.50	D										
Total	37.30		54.96		61.35		68.19		81.15		86.58	

Tabla 5.3: Tiempos de ejecución de la verificación híbrida para diferentes límites en el tamaño del s-ROBDD, y tipo de verificación realizado para dichos límites.

En la tabla 5.3, se muestran los tiempos de ejecución obtenidos para diferentes límites establecidos en el número de nodos de los s-ROBDDs, así como el tipo de verificación finalizado cuando se alcanzan dichos límites. En la tabla, $L100$, $L500$, $L1000$, $L1500$, $L3000$ y $L10000$ representan los límites en el número de nodos (tamaño) de los s-ROBDDs correspondientes a 100, 500,

1000, 1500, 3000 y 10000 nodos, respectivamente. Asimismo, T representa el tiempo de ejecución y V indica el tipo de verificación realizado para dicho límite (P = probabilista, D = determinista). La verificación probabilista siempre se puede realizar, independientemente del límite establecido, ya que este límite afecta únicamente al número de nodos del s-ROBDD. Por tanto, si en el tipo de verificación aparece una P, esto significa que la verificación determinista no se ha podido realizar y que únicamente se ha realizado la probabilista, indicándose el tiempo necesario para la finalización de esta verificación. Si en el tipo de verificación aparece una D, entonces se ha podido completar la verificación determinista, indicándose su tiempo de ejecución. Los tiempos de ejecución totales que se dan para cada límite incluyen los tiempos de los benchmarks ya construidos de forma determinista en algún límite anterior.

Circuito	L100	L500	L1000	L1500	L3000	L10000
9sym	468	508				
add6	468	588				
alu2	436	484				
apex4	468	516	540			
apex6	548	612	740	820	948	948
apex7	524	636	756	756	884	
count	484	612				
dalu	508	708	820	908	1580	1764
duke2	452	532	580	628		
e64	572	716	932	1004		
ex1010	468	516	540	540		
ex5	436	484				
exp	436	460				
gary	492	540	564			
rd84	484					
sao2	436	508				
sym10	484					
t481	564					
vg2	484	580	668	692	740	
z5xp1	484					
z9sym	508					
Total	10204	11524	12308	12620	13596	13780

Tabla 5.4: Consumo relativo de memoria en la verificación híbrida.

De los resultados obtenidos en la tabla 5.3, se observa que todos los benchmarks pudieron ser verificados de forma determinista en el rango de límites establecidos (incrementándose el número de verificaciones deterministas finalizadas a medida que el límite de nodos aumenta) a excepción del benchmark *dalu*, que para el límite de 10000 nodos únicamente se pudo verificar de manera

probabilista. También se observa que los tiempos de ejecución son superiores a los tiempos obtenidos para la verificación probabilista dados en la tabla 5.1, ya que en la verificación híbrida se intentan construir en primera instancia los diagramas de decisión, con lo que se consume un mayor tiempo para su construcción. Si se alcanza el límite establecido, entonces al menos se podrá realizar una comparación probabilista de los benchmarks. Se debe mencionar que el tiempo total obtenido para el límite de 10000 nodos no coincide con el total determinista dado en la tabla 5.1, ya que en la tabla 5.3 se incluyen los tiempos consumidos por el circuito *dalu* que no se pudo verificar de forma determinista.

Circuito	L100	L500	L1000	L1500	L3000	L10000
9sym	101	25				
add6	101	309				
alu2	101	168				
apex4	101	501	928			
apex6	112	522	1174	1502	3109	2760
apex7	101	501	1005	1501	1660	
count	102	234				
dalu	108	556	1401	1664	3019	10943
duke2	101	502	1001	973		
e64	142	517	1011	1441		
ex1010	102	501	1001	1067		
ex5	102	268				
exp	101	210				
gary	104	507	518			
rd84	42					
sao2	101	155				
sym10	31					
t481	21					
vg2	126	501	1002	1503	1044	
z5xp1	42					
z9sym	25					
Total	1867	6138	10571	12627	15289	22864

Tabla 5.5: Número de nodos de los s-ROBDDs en la verificación híbrida.

En la tabla 5.4 se muestran los consumos relativos de memoria (expresados en Kilobytes) obtenidos en la verificación híbrida para los distintos límites establecidos, incluyéndose en los valores totales dados para cada límite los valores obtenidos para los circuitos construidos de forma determinista en algún límite anterior. Se observa que la memoria usada aumenta conforme el límite de nodos aumenta, debido a que el grafo construido es cada vez mayor. En algunos casos, la memoria consumida para distintos limites puede ser parecida. Esto sería debido a que el entorno CUDD modificado habría realizado eliminaciones

de nodos no utilizados hasta ese momento durante la construcción del grafo. También se puede observar que la memoria total consumida para el límite de 10000 nodos no coincide con el valor determinista dado en la tabla 5.2 porque en la tabla 5.4 se incluyen los resultados obtenidos para el benchmark *dalu*.

Circuito	L100	L500	L1000	L1500	L3000	L10000
9sym	2044	4088				
add6	2044	7154				
alu2	1022	3066				
apex4	2044	4088	5110			
apex6	1022	3066	7154	10220	14308	14308
apex7	3066	7154	12264	12264	17374	
count	2044	7154				
dalu	1022	9198	12264	15330	21462	30660
duke2	1022	4088	6132	8176		
e64	4088	10220	19418	22484		
ex1010	2044	4088	5110	5110		
ex5	1022	3066				
exp	1022	2044				
gary	3066	5110	6132			
rd84	3066					
sao2	1022	4088				
sym10	3066					
t481	6132					
vg2	2044	6132	9198	10220	12264	
z5xp1	3066					
z9sym	4088					
Total	49056	103222	132860	145124	162498	171696

Tabla 5.6: Número máximo de nodos de los s-ROBDDs en la verificación híbrida.

El número de nodos obtenidos durante la construcción de los benchmarks en la verificación híbrida se muestra en la tabla 5.5 para los distintos límites establecidos, donde se incluyen también en los valores totales los resultados correspondientes al benchmark *dalu* y los correspondientes a benchmarks ya construidos de forma determinista para algún límite anterior. En esta tabla se observa que el tamaño (número de nodos) del s-ROBDD aumenta para los límites crecientes cuando aún no se ha podido construir el s-ROBDD completo de los benchmarks a verificar. En cambio, cuando se han podido construir totalmente (es decir, cuando se puede completar la verificación determinista), en algunos casos el tamaño final es menor que los tamaños intermedios anteriores. Esto es debido a que la herramienta de síntesis habría podido realizar todas las simplificaciones posibles y se habrían eliminado los nodos no utilizados o redundantes, produciendo la consiguiente reducción de tamaño.

Finalmente, en la tabla 5.6 se muestran los resultados correspondientes al número máximo de nodos obtenidos durante la construcción de los benchmarks para los distintos límites. En los valores totales que se dan para cada límite, se incluyen de nuevo los datos correspondientes al circuito *dalu* y se incluyen los valores para los circuitos ya construidos de forma determinista en alguno de los límites anteriores. De los resultados mostrados, se observa que los tamaños máximos se van incrementando a medida que aumenta el límite de nodos debido a que el s-ROBDD construido es cada vez mayor. En algunos casos, el tamaño final puede coincidir con algún resultado intermedio anterior debido a la eliminación de nodos no utilizados o redundantes realizada por el *Manager* del entorno CUDD modificado.

5.5.3.2. Con reordenación de variables

El algoritmo de *desplazamiento* de Rudell [Rud93] se basa en el encuentro de la posición óptima para una variable, asumiendo que el resto de variables permanecen fijas, por medio del intercambio de esa variable con sus adyacentes y recordando la posición de la variable en la que el tamaño del grafo sea menor. Las mejoras en los tamaños de los diagramas de decisión debido a la aplicación de este algoritmo de reordenación de variables son muy significativas, como se vio en el capítulo 4, por lo que tratamos de ver su implementación para el caso en que se utilicen signaturas.

En el capítulo 4 se vio que la función f representada por un nodo de un ROBDD asociado con una variable x se puede expresar por medio de la descomposición Shannon como $f = \overline{x} \cdot f_{\overline{x}} + x \cdot f_x$, donde + y $\cdot$ representan la *disyunción* y la *conjunción* Booleanas, respectivamente. Denotemos como g al *cofactor negativo* de f y como h al *cofactor positivo* de f, es decir, $g = f_{\overline{x}}$ y $h = f_x$. Si se efectúa la descomposición Shannon de g y h con respecto a una variable y, se obtienen las expresiones $g = \overline{y} \cdot g_{\overline{y}} + y \cdot g_y$ y $h = \overline{y} \cdot h_{\overline{y}} + y \cdot h_y$, respectivamente. Estas expresiones de g y h descompuestas se pueden sustituir en la descomposición inicial de f con respecto a x, obteniendo

$$f = \overline{x} \cdot (\overline{y} \cdot g_{\overline{y}} + y \cdot g_y) + x \cdot (\overline{y} \cdot h_{\overline{y}} + y \cdot h_y) \tag{5.19}$$

donde sacando factores comunes y operando se puede escribir

$$f = \overline{y} \cdot (\overline{x} \cdot g_{\overline{y}} + x \cdot h_{\overline{y}}) + y \cdot (\overline{x} \cdot g_y + x \cdot h_y) = \overline{y} \cdot f_{\overline{y}} + y \cdot f_y \tag{5.20}$$

Es decir, esta expresión es la descomposición Shannon de la función f con respecto a la variable y, mientras que sus cofactores $f_{\overline{y}}$ y f_y estan expresados por medio de la descomposición Shannon con respecto a la variable x. El cofactor negativo $f_{\overline{y}}$ se calcula en función de los cofactores negativos de g y h con respecto de y, y el cofactor positivo f_y se calcula en función de los cofactores positivos de g y h con respecto de y.

La ecuación 5.20 permite realizar de forma sencilla el intercambio de las variables adyacentes x e y dentro de un ROBDD. Si la función f está representada por un nodo asociado a la variable x, y si las funciones $g = f_{\overline{x}}$ y $h = f_x$ están representadas por dos nodos del ROBDD asociados a la variable y, entonces la función f se puede representar por un nodo del ROBDD asociado a la variable y de forma que:

- Su cofactor negativo ($f_{\overline{y}}$) está representado por un nodo asociado a la variable x con cofactor negativo $g_{\overline{y}}$ y con cofactor positivo $h_{\overline{y}}$.

- Su cofactor positivo (f_y) está representado por un nodo asociado a la variable x con cofactor negativo g_y y con cofactor positivo h_y.

En la figura 5.2 se muestra el intercambio de dos variables adyacentes x e y dentro de un ROBDD. En el diagrama de la izquierda, se muestra la función f y sus cofactores negativo ($g = f_{\overline{x}}$) y positivo ($h = f_x$). A su vez, $g_0 = g_{\overline{y}}$ y $g_1 = g_y$ son los cofactores de g, y $h_0 = h_{\overline{y}}$ y $h_1 = h_y$ son los cofactores de h. Para realizar el intercambio de la variable x por la variable y, se necesita un nodo asignado a la variable y que representa a una función k. Para este nodo, se tiene que su *0-sucesor* apunta a un nodo x que representa la función $m = k_{\overline{y}}$ (con su *0-sucesor* apuntando a g_0 y con su *1-sucesor* apuntando a h_0) y su *1-sucesor* apunta a un nodo x que representa la función $n = k_y$ (con su *0-sucesor* apuntando a g_1 y con su *1-sucesor* apuntando a h_1). Este intercambio se muestra en el diagrama derecho de la figura 5.2 y se observa que estos cambios se realizan simplemente creando los nodos x e y necesarios, y redirigiendo los punteros de sus *0,1-sucesores* de la forma adecuada. Las funciones f y k de la figura 5.2 son claramente iguales (ecuación 5.20) y, por lo tanto, $f = k$.

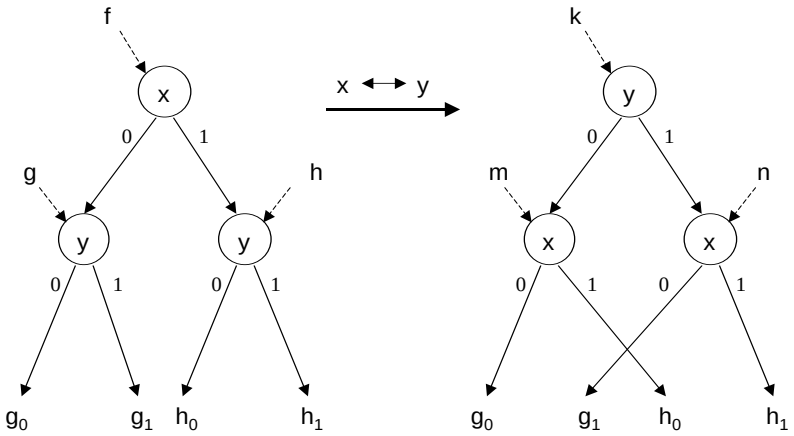


Figura 5.2: Intercambio de dos variables adyacentes en un ROBDD.

Para el cálculo de las signaturas definidas sobre un campo $GF(2^m)$ se utilizan las consideraciones vistas en la sección 5.4. Las signaturas correspondientes a las funciones representadas en la figura 5.2 serían, por tanto, $\mathbf{g} = (1+\mathbf{y})\cdot\mathbf{g_0} + \mathbf{y}\cdot\mathbf{g_1}$, $\mathbf{h} = (1+\mathbf{y})\cdot\mathbf{h_0} + \mathbf{y}\cdot\mathbf{h_1}$ y $\mathbf{f} = (1+\mathbf{x})\cdot\mathbf{g} + \mathbf{x}\cdot\mathbf{h}$ para el diagrama izquierdo, y $\mathbf{m} = (1+\mathbf{x})\cdot\mathbf{g_0} + \mathbf{x}\cdot\mathbf{h_0}$, $\mathbf{n} = (1+\mathbf{x})\cdot\mathbf{g_1} + \mathbf{x}\cdot\mathbf{h_1}$ y $\mathbf{k} = (1+\mathbf{y})\cdot\mathbf{m} + \mathbf{y}\cdot\mathbf{n}$ para el diagrama derecho de la figura, donde $\mathbf{f}$, $\mathbf{g}$, $\mathbf{h}$, $\mathbf{k}$, $\mathbf{m}$, $\mathbf{n}$, $\mathbf{g_0}$, $\mathbf{g_1}$, $\mathbf{h_0}$ y $\mathbf{h_1}$ representan las signaturas de las funciones correspondientes, $\mathbf{x}$ e $\mathbf{y}$ son las signaturas aleatorias asignadas a esas variables y donde las operaciones aritméticas están definidas sobre el campo $GF(2^m)$. Al ser las funciones f y k iguales, se tiene que

$$\mathbf{f} = (1+\mathbf{x})\cdot\mathbf{g} + \mathbf{x}\cdot\mathbf{h} = (1+\mathbf{y})\cdot\mathbf{m} + \mathbf{y}\cdot\mathbf{n} = \mathbf{k} \tag{5.21}$$

por lo que el proceso de intercambio de variables se puede acelerar asignando directamente al campo signatura del nodo y que representa a la función k, el valor $\mathbf{f}$ ya calculado.

Utilizando lo visto en los párrafos anteriores, se han realizado experimentos de verificación *determinista* e *híbrida* utilizando el algoritmo de *reordenación de variables por desplazamiento* de Rudell con el objeto de reducir el tamaño de los s-ROBDDs una vez que han sido construidos. La reordenación de variables no tiene efecto, lógicamente, en la verificación *probabilista* ya que en esta se opera únicamente con signaturas, sin realizarse la construcción de los s-ROBDDs de las funciones.

En la tabla 5.7 se muestran los resultados obtenidos para la *verificación determinista* realizando una *reordenación* final de variables. En la tabla únicamente se muestran el tiempo de ejecución y el número de nodos final alcanzado por el s-ROBDD de cada benchmark. El número máximo de nodos durante la construcción y la cantidad relativa de memoria necesaria coinciden con los valores obtenidos anteriormente sin reordenación de variables y que fueron mostrados en la tabla 5.2, debido a que la reordenación se aplica después de realizarse la construcción del s-ROBDD.

Los resultados dados en la tabla 5.7 muestran, en primer lugar, que el circuito *dalu* pudo ser construido utilizando la reordenación de variables. Para ello, se hizo que el algoritmo de reordenación se ejecutara cuando el s-ROBDD alcanzara un número de nodos determinado durante la construcción. El tamaño máximo obtenido para este benchmark fue de 23506 nodos y la memoria consumida fue de 1580KB. Para el resto de los benchmarks de la tabla, la reordenación se produjo una vez construido el grafo, bien completamente (cuando se puede finalizar la verificación determinista), bien parcialmente (cuando se alcanza algún límite). En esta tabla también se muestran los valores totales medidos del tiempo de ejecución y del tamaño de los grafos construidos, para el caso en que se incluyan los resultados obtenidos con el circuito *dalu* y para el caso en que no se incluyan.

Circuito	Tiempo	Tamaño
9sym	0.24	25
add6	1.83	69
alu2	0.24	88
apex4	3.21	909
apex6	12.15	641
apex7	11.14	304
count	1.05	81
dalu	31.47	814
duke2	1.45	387
e64	4.15	132
ex1010	9.80	1055
ex5	8.82	242
exp	0.73	166
gary	1.19	302
rd84	0.45	42
sao2	0.37	86
sym10	1.12	31
t481	5.84	21
vg2	4.14	229
z5xp1	0.50	42
z9sym	0.52	25
Total	100.41	5691
Total(†)	68.94	4877

Tabla 5.7: Tiempo de ejecución y tamaño obtenidos en la verificación determinista con reordenación de variables († = sin incluir el circuito *dalu*).

Comparando estos resultados con los dados en la tabla 5.1, se observa que el tiempo total de ejecución para los benchmarks dados (donde no se incluye el tiempo obtenido para el bechmark *dalu* por motivos de comparación) resulta ser un 17.3 % mayor que el obtenido en la verificación determinista sin reordenación. El método más rápido sigue siendo claramente el probabilista, cuyo tiempo de ejecución sería en este caso un 87.4 % menor que el determinista con reordenación. Incluyendo el circuito *dalu* en el valor total, se tiene que el tiempo de ejecución de la verificación probabilista (que en este caso sería de 10.76 segundos de CPU, según los datos proporcionados en la tabla 5.1) es un 89.3 % menor que el necesitado por la verificación determinista con reordenación.

Con respecto al consumo de memoria, el resultado de la comparación entre verificación probabilista y determinista con reordenación, sin la inclusión del circuito *dalu*, es el mismo que el indicado en la comparación sin reordenación, ya que hemos señalado anteriormente que la reordenación se realiza con posterioridad a la construcción de los s-ROBDDs, por lo que los datos de memoria

coinciden con los vistos en la tabla 5.2. Si se incluye el circuito *dalu*, entonces se tendría un consumo de memoria de 1764KB del método probabilista frente a los 13596KB de la aproximación determinista con reordenación, por lo que la memoria consumida por el método probabilista sería un 87 % menor.

En la verificación *híbrida con reordenación* de variables se han realizado experimentos estableciendo los mismos límites que en el caso de la verificación híbrida sin reordenación. En la tablas 5.8 y 5.9 se muestran los resultados obtenidos por cada benchmark para los distintos límites, junto con los valores totales medidos para cada límite, incluyendo en estas tablas los resultados obtenidos para el circuito de benchmark *dalu*. Al igual que en tablas similares vistas anteriormente para el caso de no utilización de la reordenación de variables, estos valores totales para cada límite incluyen también los valores obtenidos por aquellos benchmarks ya construidos de manera determinista en algún límite anterior.

Circuito	L100		L500		L1000		L1500		L3000		L10000	
	T	V	T	V	T	V	T	V	T	V	T	V
9sym	0.21	P	0.24	D								
add6	0.98	P	1.83	D								
alu2	0.17	P	0.24	D								
apex4	1.38	P	2.90	P	3.21	D						
apex6	4.90	P	5.93	P	7.27	P	8.35	P	11.71	P	12.15	D
apex7	4.23	P	6.48	P	9.26	P	9.93	P	11.14	D		
count	0.52	P	1.05	D								
dalu	6.44	P	12.71	P	15.48	P	17.49	P	27.71	P	31.47	D
duke2	0.43	P	0.85	P	1.38	P	1.45	D				
e64	1.21	P	2.19	P	3.10	P	4.15	D				
ex1010	4.53	P	6.61	P	6.60	P	9.80	D				
ex5	4.81	P	8.82	D								
exp	0.42	P	0.73	D								
gary	0.51	P	0.96	P	1.19	D						
rd84	0.45	D										
sao2	0.25	P	0.37	D								
sym10	1.12	D										
t481	5.84	D										
vg2	1.25	P	2.39	P	2.83	P	3.37	P	4.14	D		
z5xp1	0.50	D										
z9sym	0.52	D										
Total	40.67		62.73		72.03		80.66		96.22		100.41	

Tabla 5.8: Tiempos de ejecución de la verificación híbrida con reordenación de variables para diferentes límites en el tamaño del s-ROBDD y tipo de verificación realizado para dichos límites.

En la tabla 5.8 se muestran los tiempos de ejecución obtenidos para los circuitos de benchmark dados. La comparación de estos tiempos totales con los obtenidos en la tabla 5.3 muestra que la verificación híbrida con reordenación presenta unos mayores tiempos de ejecución para todos los límites, que varían entre el 9 % mayor para el límite 100 y el 18.6 % mayor para el límite 3000. Para el límite de 10000 nodos, se tiene que el tiempo consumido por la aproximación con reordenación es un 16 % mayor que el necesitado por la verificación híbrida sin reordenación, teniendo en cuenta que el resultado mostrado en la tabla 5.3 para este límite incluye el tiempo *probabilista* de verificación del circuito *dalu*, ya que sin reordenación no se pudo construir el grafo para este circuito.

Los datos mostrados en la tabla 5.9 indican, sin embargo, que el número de nodos de los s-ROBDDs construidos usando reordenación de variables es significativamente menor que el dado en la tabla 5.5 sin reordenación. Los tamaños varían entre un 35.9 % menor para el límite 100 y un 75.1 % menor para el límite 10000, siempre teniendo en cuenta que con reordenación el benchmark *dalu* pudo ser construido completamente (sin considerar *dalu*, los tamaños varían entre un 34 % menor para el límite 100 y un 60.8 % menor para el límite 3000).

Circuito	L100	L500	L1000	L1500	L3000	L10000
9sym	95	25				
add6	39	69				
alu2	51	88				
apex4	97	483	909			
apex6	32	71	52	96	575	641
apex7	32	94	192	258	304	
count	56	81				
dalu	35	150	191	206	275	814
duke2	61	207	401	387		
e64	69	125	151	132		
ex1010	95	497	992	1055		
ex5	87	242				
exp	68	166				
gary	82	300	302			
rd84	42					
sao2	79	86				
sym10	31					
t481	21					
vg2	57	163	218	273	229	
z5xp1	42					
z9sym	25					
Total	1196	3008	4326	4536	5086	5691

Tabla 5.9: Tamaño de los s-ROBDDs en la verificación híbrida con reordenación.

5.6. Conclusiones

En este capítulo se han combinado las verificaciones determinista y probabilista para la realización de una aproximación *híbrida* de verificación en la que se comienza construyendo los diagramas de decisión de las funciones a verificar y donde siempre se calculan las signaturas de las mismas. Si no existen restricciones de espacio o tiempo para la comprobación de la equivalencia, se finaliza la verificación *determinista* de las funciones a través de la comparación de sus ROBDDs, en cuyos nodos se incluyen campos para el almacenamiento de *signaturas* (diagramas a los que hemos denominado s-ROBDDs). Si existe un límite de espacio o de tiempo, la construcción de s-ROBDDs se detiene cuando se alcanza ese límite y se finaliza entonces la verificación *probabilista*, en la que se comparan las signaturas de las funciones para decidir su equivalencia y en donde se utilizan las signaturas calculadas por los s-ROBDDs construidos hasta el momento en que se alcanzó el límite para la obtención de las signaturas finales de las funciones.

El método híbrido de verificación se ha aplicado a circuitos combinacionales de dos niveles representados en formato DSOP, para lo cual se ha utilizado la formulación vista en el capítulo 3 para transformar las representaciones SOP originales de los benchmarks a verificar en representaciones DSOP equivalentes. El motivo de la utilización de expresiones *cúbicas disjuntas* de los circuitos es debido a que la propiedad de *disjunción* (u *ortogonalidad*) permite en gran medida la simplificación en el cálculo de signaturas cuando se utiliza la verificación probabilista, como se vio en el capítulo 2. La utilización de campos de Galois $GF(2^m)$ como campos finitos permite también la simplificación en el cálculo de las signaturas debido a sus propiedades.

La inclusión de signaturas en los ROBDDs clásicos se ha realizado añadiendo campos de *signatura* a los nodos del ROBDD y modificando los algoritmos de síntesis consecuentemente de forma que se realice el cálculo correcto de las signaturas durante la construcción. Las operaciones aritméticas involucradas en dicho cálculo son la *adición* y la *multiplicación* sobre $GF(2^m)$, habiéndose utilizado la base canónica (o polinómica) de representación. Para el cálculo del producto de signaturas sobre $GF(2^m)$ se ha utilizado un algoritmo de multiplicación en base polinómica que será estudiado en el capítulo 7.

Se han realizado experimentos de verificación *probabilista*, *determinista* e *híbrida*, comparándose el tamaño de memoria y el tiempo de ejecución necesarios para la verificación. El método probabilista implica el cálculo de signaturas exclusivamente, mientras que los métodos determinista e híbrido implican la construcción de s-ROBDDs. Por este motivo, para estos dos tipos de verificación se realizaron experimentos considerando dos aproximaciones relativas a la ordenación de variables: usando la ordenación original fija dada por los circuitos de benchmark y usando la aproximación dinámica de reordenación de

variables por desplazamiento de Rudell. De los resultados obtenidos, se ha podido comprobar que, para el tipo particular de circuitos utilizados, el método probabilista resulta ser el de mejor rendimiento tanto en tiempo de ejecución como en cantidad de memoria consumida, mientras que la verificación determinista (en la que los s-ROBDDs se construyen completamente, si es posible) es la que presenta el peor comportamiento tanto en tiempo de ejecución como en almacenamiento.

En los experimentos realizados *sin reordenación de variables* y comparando las verificaciones determinista e híbrida, se observa que el método determinista es el que produce los peores resultados, incluso no pudiéndose verificar alguno de los benchmarks debido a su tamaño. El método *híbrido* construye los s-ROBDDs (aproximación determinista) hasta que se alcanza el límite establecido, continuando a partir de ese momento con el cálculo exclusivo de signaturas (aproximación probabilista) utilizando para ello las signaturas ya calculadas y almacenadas en los s-ROBDDs construidos. Ya que la aproximación probabilista es la que consume menos tiempo y memoria, es lógico que el método híbrido tenga un rendimiento intermedio entre el mejor caso (método probabilista) y el peor caso (método determinista), dependiendo su comportamiento del límite elegido. Estas mismas consideraciones se pueden hacer para el caso de la utilización de una aproximación *dinámica* para la reordenación de variables de entrada.

En el caso de construcción *dinámica*, se obtiene una reducción significativa del número de nodos de los s-ROBDDs en comparación con los tamaños obtenidos sin reordenación de variables, incluso pudiéndose construir circuitos que no podían serlo utilizando una ordenación fija de variables. Por el contrario, el tiempo de construcción *dinámica* se ve incrementado con respecto a la síntesis de s-ROBDDs sin reordenación. Ambos resultados son lógicos debido a la eficiencia que presenta el algoritmo de desplazamiento de variables y debido al tiempo invertido por el algoritmo en la realización de los desplazamientos, respectivamente. Con respecto al consumo de memoria, los resultados obtenidos en los experimentos son menores o iguales que los obtenidos con la ordenación fija original. En cualquier caso, y para la utilización de ambos métodos de construcción, los resultados de tiempo y memoria tendrán como límite inferior los obtenidos por el método probabilista.

La ventaja que presenta la utilización de un entorno probabilista de verificación con cálculo de signaturas consiste en la posibilidad de comprobar la equivalencia de los circuitos aunque no se puedan construir sus diagramas de decisión. La verificación determinista a través de ROBDDs requiere la construcción completa de los grafos de los circuitos para poder determinar su equivalencia. Si no es posible la creación de alguno de los grafos, nada se puede decidir acerca de su equivalencia. Sin embargo, en un entorno probabilista de verificación, siempre sería posible realizar al menos una comprobación de

equivalencia a través de la comparación de signaturas (con una probabilidad de error pequeña que se podría reducir aumentando la cardinalidad del campo finito o realizando varias ejecuciones). Por lo tanto, las aproximaciones de verificación de tipo probabilista (como el método *híbrido* presentado) pueden suponer una buena alternativa a la verificación determinista clásica.

El inconveniente que presenta nuestra aproximación *híbrida* de verificación está relacionado, desde luego, con el coste adicional de memoria que supone la inclusión de campos para el almacenamiento de signaturas en cada nodo del s-ROBDD, así como con el tiempo necesario para el cálculo y manipulación de las signaturas, fundamentalmente para la *multiplicación* sobre el campo $GF(2^m)$. De este último motivo se deduce la necesidad de algoritmos *rápidos* de multiplicación sobre campos de Galois. A pesar de estos inconvenientes, la posibilidad de facilitar algún resultado con respecto de la equivalencia de las funciones a verificar, serviría para justificar este coste adicional.

Nuestra verificación *híbrida* se restringe a los circuitos de dos niveles en forma SOP, por medio de la conversión previa a formatos de tipo DSOP (usando la formulación del capítulo 3). Un trabajo futuro de investigación sería la generalización a circuitos SOP sin la restricción impuesta por la propiedad de *disjunción*. Asimismo, también sería necesaria la aplicación de aproximaciones probabilistas para la verificación de circuitos en formato multinivel. En el capítulo siguiente estudiamos esta posibilidad a través de la utilización de un tipo especial de diagramas de decisión en los que se aplican las propiedades de los campos $GF(2^m)$ para la simplificación del cálculo de signaturas.

Capítulo 6

Utilización de $\oplus$-OBDDs en la verificación probabilista

En el capítulo anterior se combinaron las verificaciones probabilista y determinista para poder obtener una aproximación híbrida de verificación de circuitos combinacionales representados en un formato de dos niveles. En este capítulo se estudian los diagramas de decisión conocidos como $\oplus$-OBDDs o $Mod2$-OBDDs y se presenta un nuevo método de construcción para su utilización en la verificación probabilista de circuitos combinacionales representados en un formato multinivel.

En el capítulo anterior se han combinado las aproximaciones probabilista y determinista para la obtención de un método híbrido de verificación de funciones Booleanas que representan circuitos lógicos combinacionales de dos niveles expresados en forma de sumas de productos.

Una representación de *dos niveles* indica que cualquier señal de entrada del circuito atraviesa como mucho dos puertas lógicas para llegar a la salida (suponiendo que se dispone directamente del complemento de las señales). Los circuitos descritos por expresiones de dos niveles se pueden implementar en estructuras regulares y compactas llamadas *Arrays Lógicos Programables* (PLAs). El desarrollo de métodos eficientes para la minimización de expresiones de dos niveles y la existencia de circuitos económicos para su implementación hacen que las representaciones de dos niveles sean ampliamente utilizadas tanto en aplicaciones industriales como de investigación. Sin embargo, la complejidad creciente de los circuitos digitales actuales a implementar y el desarrollo de nuevas tecnologías conducen a la utilización de otro tipo de representación para aquellos circuitos que sean difícilmente implementables en estructuras de dos niveles de tipo PLA.

La lógica *multinivel* es un circuito en el que las señales pueden atravesar un número arbitrario de puertas antes de alcanzar las salidas. En este caso, el circuito se describe por medio de un conjunto de funciones interconectadas, donde la *reutilización* de subfunciones es una de sus característica principales. Esta propiedad proporciona una flexibilidad importante que conducirá a una mejor adaptación a diferentes situaciones de implementación. El formato de representación de circuitos multinivel más ampliamente utilizado es el formato BLIF (*Berkeley Logic Interchange Format*) de la Universidad de Berkeley, que describe un circuito jerárquico a nivel lógico de forma textual.

En este capítulo se aborda el problema de la verificación probabilista de funciones Booleanas que representan circuitos lógicos combinacionales representados en formato BLIF multinivel, por medio de la utilización de la estructura de datos conocida como ⊕-OBDDs o $Mod2$-OBDDs. Los ⊕-OBDDs, introducidos por Gergov y Meinel en [GM96] y estudiados posteriormente en exclusiva por Meinel y Sack [MS98][MS01a][MS01b], constituyen representaciones *no canónicas* de funciones Booleanas, por lo que no parecen ser adecuados para la verificación determinista. Sin embargo, su estructura los hace adecuados para su utilización en la verificación probabilista.

Los ⊕-OBDDs son diagramas de decisión con nodos especiales que representan la operación Booleana XOR y que se conocen como ⊕-nodos. Estos nodos se pueden incluir en la estructura ROBDD clásica por medio de la utilización de unas descomposiciones de funciones diferentes de la descomposición de Shannon, conocidas como *expansiones positiva o negativa de Davio* y basadas en la aplicación de la operación XOR.

La construcción de los ⊕-OBDDs se realiza por medio de la inclusión de una *capa superior* de dos niveles de ⊕-nodos creada usando la *expansión positiva de Davio* para una variable de entrada seleccionada mientras se sintetiza el circuito en formato BLIF original. Asimismo, para la creación de los ⊕-OBDDs se utiliza el entorno CUDD modificado de forma que se puedan introducir los nodos especiales ⊕ en un ROBDD. También se utilizan las modificaciones realizadas en el capítulo anterior sobre dicho entorno CUDD para la inclusión y cálculo de signaturas en el campo de Galois $GF(2^m)$ en los nuevos diagramas ⊕-OBDDs construidos.

6.1. ⊕-OBDDs

Los ⊕-OBDDs o $Mod2$-OBDDs fueron definidos por Gergov y Meinel en [GM96] y constituyen una extensión de los ROBDDs, preservando su propiedad de manipulación eficiente y siendo en algunos casos más eficientes con respecto al tamaño que los ROBDDs. Como inconveniente se tiene que los ⊕-OBDDs no proporcionan una representación canónica de las funciones Booleanas, a diferencia de los ROBDDs.

Definición 13 *Un $\oplus$-OBDD definido sobre un conjunto $X_n = \{x_1, x_2, \ldots, x_n\}$ de variables Booleanas es un grafo conectado acíclico dirigido P en el se pueden encontrar los siguientes tipos de nodos:*

- *Existe un nodo no terminal, conocido como raíz, que no tiene antecesor. Para funciones Booleanas $f : \mathbf{B}^n \longrightarrow \mathbf{B}^m$, se consideran $\oplus$-OBDDs compartidos con múltiples nodos raíz, donde cada raíz representa una subfunción Booleana $f_i : \mathbf{B}^n \longrightarrow \mathbf{B}$, con $f = (f_1, f_2, \ldots, f_m)$.*

- *Existen dos nodos terminales sin sucesores, el nodo terminal* **1** *y el nodo terminal* **0**, *que se etiquetan con las constantes Booleanas* **1** *y* **0**, *respectivamente.*

- *Existen nodos no terminales etiquetados con variables Booleanas $x_i \in X_n$ y denominados nodos de decisión*

- *Existen nodos no terminales etiquetados con la operación Booleana XOR y conocidos como $\oplus$-nodos.*

En un $\oplus$-OBDD, se representa como $l(v)$ a la *etiqueta* del nodo v, pudiendo ser esta etiqueta una variable Booleana perteneciente a X_n o la operación Booleana $\oplus$. Se puede observar que si v es un nodo de *decisión* y la *etiqueta* del nodo es una variable Booleana x_i, con lo que $l(v) = x_i$, entonces la *etiqueta* sería equivalente al *índice* de un nodo v de un ROBDD que estuviera asociado a la variable Booleana x_i, con lo que $index(v) = i$.

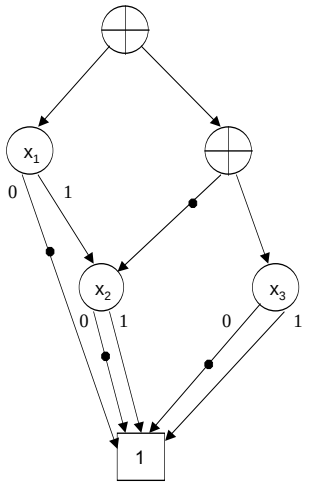
El *0-sucesor* y el *1-sucesor* de un nodo v se representan por v_0 y v_1, respectivamente. Se tiene que si v_2 es un sucesor de v_1 en P y $l(v_1), l(v_2) \in X_n$, entonces $l(v_1) < l(v_2)$ según una ordenación dada en el conjunto de variables de entrada X_n. También se tiene que al igual que en los ROBDDs, en cada camino del $\oplus$-OBDD desde el nodo raíz a los nodos terminales, cada variable debe aparecer como mucho una sola vez.

La función f_P asociada con un $\oplus$-OBDD P se puede determinar de la siguiente forma. Dada una asignación de entrada $a = (a_1, a_2, \ldots, a_n) \in \{0,1\}^n$, los valores Booleanos asignados a los nodos *hoja* (nodos terminales) se extienden a los valores Booleanos asociados con todos los nodos del $\oplus$-OBDD P de la siguiente forma:

- Si los nodos sucesores v_0 y v_1 de un nodo v de P portan (o llevan asociados) los valores Booleanos b_0 y b_1, respectivamente, y si $l(v) = x_i$, entonces se asocia con v el valor b_0 o b_1 según sea $x_i = 0$ o $x_i = 1$, respectivamente.

- Si la etiqueta del nodo v es $l(v) = \oplus$, entonces se asocia con v el valor $\oplus(b_0, b_1) = (b_0 + b_1)\ mod\ 2$.

El valor $f_P(a)$ de la función Booleana f_P representada por un ⊕-OBDD P será el valor 1 o 0 asociado con la raíz de P bajo la asignación a. Como en los ROBDDs, se puede obtener una representación aún más compacta utilizando *aristas complementadas*, es decir, *aristas caracterizadas negativas* [MB88]. En la figura 6.1(a) se muestra el ⊕-OBDD con aristas complementadas de la función $f_P = x_1x_2 \oplus (\overline{x}_2 \oplus x_3)$ y en la figura 6.1(b) se da su tabla de verdad.



(a)

x_1	x_2	x_3	f_P
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(b)

Figura 6.1: (a) ⊕-OBDD P de la función f_P. (b) Tabla de verdad de f_P.

Según la definición dada anteriormente, se observa que los ⊕-OBDDs son estructuras de datos muy adecuadas para la descripción del comportamiento de circuitos que contengan muchas puertas XOR, siendo unas representaciones más *próximas* a dichos circuitos que los ROBDDs. También se puede observar que los ⊕-OBDDs son representaciones que combinan los OBDDs, los OFDDs [KSR92][KR93] y los OKFDDs [DST+94].

6.2. Verificación probabilista aplicada a los ⊕-OBDDs

Como se ha mencionado anteriormente, los ⊕-OBDDs no constituyen representaciones canónicas de las funciones Booleanas, es decir, que para una misma función Booleana pueden existir diferentes ⊕-OBDDs. Para representaciones canónicas como los ROBDDs, la comprobación de la equivalencia de dos ROBDDs se reduce a una simple comparación de punteros en el computador, correspondientes a las direcciones de memoria de sus nodos raíz. Sin embargo, para las representaciones no canónicas la comprobación de la equivalencia se convierte en una tarea mucho más difícil. Waack [Waa97] propuso un método de comprobación de equivalencia determinista para ⊕-OBDDs ba-

sado en un algoritmo de minimización que requería un tiempo cúbico en el número de nodos, por lo que no constituye un método adecuado para casos prácticos. Por este motivo, es necesaria la utilización de otras aproximaciones más rápidas que permitan la verificación de circuitos que estén representados por medio de ⊕-OBDDs.

La verificación probabilista aplicada a los ⊕-OBDDs permite comprobar la equivalencia de dos funciones representadas por ⊕-OBDDs de forma rápida y con una baja probabilidad de error (como se vio en el capítulo 2). La equivalencia de dos ⊕-OBDDs se determina por medio de una transformación algebraica de los ⊕-OBDDs en términos de polinomios definidos sobre un campo finito. Como se vio en el capítulo 2 y en la sección 5.1 del capítulo 5, dadas las funciones Booleanas f, f_1 y f_2 representadas por los ⊕-OBDDs P, P_1 y P_2, respectivamente, se puede realizar la transformación de las operaciones Booleanas sobre estas funciones en expresiones aritméticas por medio de la aplicación de la transformada $\mathcal{A}$ y la utilización de las 0,1-*equivalencias* y las operaciones Booleanas extendidas. Si p_f, p_{f_1} y p_{f_2} representan los polinomios asociados a las funciones f, f_1 y f_2, respectivamente, obtenidos de aplicar la transformada $\mathcal{A}$ a dichas funciones (es decir, $p_f = \mathcal{A}[f]$, $p_{f_1} = \mathcal{A}[f_1]$ y $p_{f_2} = \mathcal{A}[f_2]$) y si se utiliza como campo finito el campo de Galois $GF(2^m)$, entonces

$$\begin{array}{lcll}
\neg f & \xrightarrow{\mathcal{A}} & 1 - p_f & \xrightarrow{GF} \quad 1 + p_f \\
f_1 \wedge f_2 & \xrightarrow{\mathcal{A}} & p_{f_1} \cdot p_{f_2} & \xrightarrow{GF} \quad p_{f_1} \cdot p_{f_2} \\
f_1 \vee f_2 & \xrightarrow{\mathcal{A}} & p_{f_1} + p_{f_2} - p_{f_1} \cdot p_{f_2} & \xrightarrow{GF} \quad p_{f_1} + p_{f_2} + p_{f_1} \cdot p_{f_2} \\
f_1 \oplus f_2 & \xrightarrow{\mathcal{A}} & p_{f_1} + p_{f_2} - 2 \cdot p_{f_1} \cdot p_{f_2} & \xrightarrow{GF} \quad p_{f_1} + p_{f_2}
\end{array}$$

donde las operaciones aritméticas de suma y multiplicación que aparecen en la última columna se realizan sobre $GF(2^m)$, y donde se han utilizado las simplificaciones vistas en la sección 5.1 para el campo de Galois.

Utilizando las transformaciones anteriores y al igual que se vio en el capítulo 5, se puede realizar una interpretación algebraica de un ⊕-OBDD y calcular el polinomio simbólico asociado a un nodo del ⊕-OBDD. A cada nodo v de un ⊕-OBDD P definido sobre un conjunto $X_n = \{x_1, x_2, \ldots, x_n\}$ de variables Booleanas se le puede asociar un polinomio p_v de la siguiente forma [GM96]

$$p_v = \begin{cases} 0(1) & si \quad v = terminal \;\; 0(1) \\ (1+x) \cdot p_{v_0} + x \cdot p_{v_1} & si \quad l(v) = x \in X_n \\ p_{v_0} + p_{v_1} & si \quad l(v) = \oplus \end{cases} \tag{6.1}$$

donde p_{v_0} y p_{v_1} representan los polinomios asociados a los cofactores negativo (v_0) y positivo (v_1) de v, respectivamente. Para la obtención del polinomio correspondiente a un ⊕-nodo, se ha asumido que se utiliza el campo de Galois y se ha utilizado la simplificación para la XOR vista en las transformaciones anteriores. También se puede comprobar que el polinomio p_v es el polinomio de la función Booleana f^v representada por el ⊕-OBDD rutado en el nodo v.

Con la ecuación 6.1 se puede calcular el polinomio simbólico asociado a un nodo v perteneciente a un ⊕-OBDD definido sobre un conjunto de variables Booleanas $X_n = \{x_1, x_2, \ldots, x_n\}$ cuando se utiliza un campo de Galois $GF(2^m)$ como campo finito. Sin embargo, el objetivo no es conocer el polinomio simbólico, sino una *signatura* que represente la función y que será el valor del polinomio calculado para una asignación aleatoria de sus variables seleccionada del campo finito.

Sean $a_1, a_2, \ldots, a_n \in GF(2^m)$ valores aleatorios generados independientemente y uniformemente distribuidos. Entonces la signatura de un ⊕-OBDD P se calcula evaluando el polinomio p_P asociado a P (es decir, el polinomio asociado al nodo *raíz* de P) en los valores aleatorios $a_1, a_2, \ldots, a_n$ por medio de la sustitución de cada variable $x_i \in X_n$ por el valor correspondiente $a_i \in GF(2^m)$ en el polinomio p_P. Si $\rho = \{x_1, x_2, \ldots, x_n\}$ es el *conjunto de restricción*, entonces se puede calcular la signatura para todos los nodos de un ⊕-OBDD aplicando la restricción ρ a la ecuación 6.1, para lo cual se utilizan las mismas consideraciones que las vistas en el capítulo 5, obteniendo

$$p_{v_{\{\rho\}}} = \begin{cases} 0(1) & si \quad v = terminal \;\; 0(1) \\ (1+x)_{\{\rho\}} \cdot p_{v_{0_{\{\rho\}}}} + x_{\{\rho\}} \cdot p_{v_{1_{\{\rho\}}}} & si \quad l(v) = x \in X_n \\ p_{v_{0_{\{\rho\}}}} + p_{v_{1_{\{\rho\}}}} & si \quad l(v) = \oplus \end{cases} \tag{6.2}$$

En la ecuación 6.2 se combinan valores numéricos representados por vectores de m bits y operados en el campo de Galois $GF(2^m)$ sin realizar ninguna consideración especial acerca de la existencia de variables comunes, para lo cual se han utilizado los teoremas vistos en el capítulo 2 y consideraciones similares a las realizadas en la sección 5.3 del capítulo 5.

De esta forma se puede obtener la signatura del ⊕-OBDD P, que será la signatura de su nodo raíz y que se habrá obtenido evaluando el polinomio asociado al nodo raíz p_P en el conjunto de restricción $\rho = \{x_1, x_2, \ldots, x_n\}$. Es decir, $p_{P_{\{\rho\}}} = p_P(a_1, a_2, \ldots, a_n) = \mathbf{f_P}$, que es la signatura asociada a la función f_P representada por el ⊕-OBDD P cuando se realiza la asignación aleatoria $a_1, a_2, \ldots, a_n$ a las variables de entrada $\{x_1, x_2, \ldots, x_n\}$.

Por lo tanto, sean P y Q dos ⊕-OBDDs que representan a dos funciones Booleanas f_P y f_Q, respectivamente, y sean $a_1, a_2, \ldots, a_n \in GF(2^m)$ valores aleatorios generados independientemente y uniformemente distribuidos. La equivalencia de las dos funciones f_P y f_Q se puede realizar comparando sus respectivas *signaturas* $\mathbf{f_P} = p_P(a_1, a_2, \ldots, a_n)$ y $\mathbf{f_Q} = p_Q(a_1, a_2, \ldots, a_n)$ de forma que si son diferentes, las dos funciones *no son equivalentes* con certeza, mientras que si son iguales, entonces las funciones *son equivalentes* con una pequeña probabilidad de error que se puede reducir incrementando la cardinalidad del campo finito o realizando varias ejecuciones con distintas asignaciones aleatorias de variables de entrada.

6.3. Introducción de ⊕-nodos en un ROBDD

La síntesis de ⊕-OBDDs se puede realizar utilizando el algoritmo *ite* visto en el capítulo 4 de la misma forma que se utiliza para la síntesis de ROBDDs clásicos, excepto para el cálculo de la XOR de dos funciones. En este caso, los ⊕-OBDDs se crean introduciendo un ⊕-nodo con sus *sucesores* apuntando a cada una de las dos funciones. Sin embargo, si el circuito bajo consideración no contiene ninguna puerta XOR, el algoritmo *ite* construiría un ROBDD en lugar de un ⊕-OBDD. Por lo tanto, para la creación de un ⊕-OBDD se tienen que incluir ⊕-nodos en la estructura de datos aunque no aparezcan puertas XOR de forma explícita en la descripción del circuito.

Esta inclusión de ⊕-nodos en un ROBDD se puede realizar utilizando descomposiciones de funciones alternativas a la descomposición de Shannon utilizada por *ite* y que incluyan la generación de ⊕-nodos. Para la creación de los ⊕-OBDDs se puede utilizar la *expansión positiva (pDE) o negativa (nDE) de Davio* [DDT78], también conocidas como expansión *Reed-Muller* [Ree54] [Mul54], dadas por las siguientes expresiones

$$pDE: \quad f = f_{\overline{x}_i} \oplus x_i \cdot (f_{x_i} \oplus f_{\overline{x}_i}) \tag{6.3}$$

$$nDE: \quad f = f_{x_i} \oplus \overline{x}_i \cdot (f_{x_i} \oplus f_{\overline{x}_i}) \tag{6.4}$$

donde $f_{\overline{x}_i}$ y f_{x_i} representan los cofactores negativo y positivo, respectivamente, de la función f con respecto de la variable x_i para la cual se realiza la expansión. Por tanto, aplicando cualquiera de estas dos expansiones, se puede hacer corresponder el operador XOR directamente con un ⊕-nodo.

La creación ilimitada de ⊕-nodos por medio de la aplicación de las expansiones dadas por las ecuaciones 6.3 y 6.4 puede incrementar el número total de nodos del ⊕-OBDD más allá del tamaño del ROBDD convencional para la misma función. Este efecto se puede evitar aplicando ciertas reglas de reducción de ⊕-nodos [MS98] o utilizando las descomposiciones de funciones pDE y nDE de forma alternativa, en lugar de usar siempre la misma descomposición [MS98][MS01b]. Sin embargo, el problema puede persistir si los ⊕-nodos creados no se pueden posicionar en lugares adecuados del ⊕-OBDD, por lo que el tamaño total del mismo puede verse incrementado.

En [MS98] y [MS01b], Meinel y Sack presentaron varias estrategias de ubicación de ⊕-nodos dentro del ⊕-OBDD para diferentes combinaciones de expansiones (*ite*, nDE, pDE) y utilizando para la construcción las ordenaciones de variables originales dadas por las descripciones de los distintos circuitos. En la siguiente sección se presenta una aproximación para la construcción de ⊕-OBDDs basada en la utilización de la expansión positiva de Davio pDE y que consiste en la creación de una *capa superior* de dos niveles de ⊕-nodos junto con una *capa inferior* formada por ROBDDs.

6.4. Una aproximación para la creación de ⊕-OBDDs

La ventaja principal de una estructura de datos formada por ROBDDs y ⊕-nodos para su utilización en la verificación probabilista es que la signatura de un ⊕-nodo se puede calcular realizando simplemente la XOR bit a bit de las signaturas asociadas a sus *0,1-sucesores* (si se utiliza un campo de Galois).

El objetivo de nuestra aproximación [ISV95] consiste en la construcción de un ⊕-OBDD que presente una *capa superior* de dos niveles de ⊕-nodos. La introducción de los ⊕-nodos en un ROBDD se realiza en el proceso de síntesis utilizando la *expansión positiva de Davio pDE* con respecto a una única variable de entrada seleccionada. Este criterio se establece debido a que el número de ⊕-nodos parece influir en el tamaño de los ⊕-OBDDs [MS98] construidos, por lo que un número no muy grande de ⊕-nodos podría conducir a tamaños pequeños de ⊕-OBDDs (aunque esto no siempre es cierto). La *pDE* (ecuación 6.3) con respecto de una variable x_i viene dada como

$$f(\ldots, x_i, \ldots) = f_{\overline{x}_i} \oplus x_i \cdot (f_{x_i} \oplus f_{\overline{x}_i}) = f_{\overline{x}_i} \oplus (x_i \cdot f_{x_i} \oplus x_i \cdot f_{\overline{x}_i}) \qquad (6.5)$$

Utilizando esta expresión se puede construir el ⊕-OBDD de la función f insertando dos ⊕-nodos en los dos niveles superiores del grafo, como se muestra en la figura 6.2(a). Se observa que la estructura que aparece en el lado derecho de la figura 6.2(a) se obtiene directamente de la estructura que aparece en el lado izquierdo, y ambas se deducen inmediatamente de la ecuación 6.5. En la representación derecha obtenida en la figura 6.2(a), el ⊕-nodo *superior* de la capa tiene su *0-arista* apuntando a la función $f_{x_i=0}$ (cofactor negativo de f), mientras que su *1-arista* apunta al ⊕-nodo $inferior$. A su vez, la *0-arista* de este ⊕-nodo inferior apunta a la función $x_i \cdot f_{x_i=0}$ y su *1-arista* apunta a la función $x_i \cdot f_{x_i=1}$ (donde $f_{x_i=1}$ es el cofactor positivo de f). Las funciones $x_i \cdot f_{x_i=0}$ y $x_i \cdot f_{x_i=1}$ se pueden representar por ROBDDs, por lo que la estructura general final que se obtiene para múltiples salidas es la que se muestra en la figura 6.2(b), donde aparece una *capa superior* formada por dos niveles de ⊕-nodos y una *capa inferior* formada por ROBDDs.

Esta expansión se podría haber utilizado para la construcción completa del circuito en forma ⊕-OBDD, pero nosotros hemos restringido la inclusión de ⊕-nodos a una capa de dos niveles de ⊕-nodos y hemos seleccionado una sola variable para realizar la expansión *pDE*. El objetivo es introducir un número reducido de ⊕-nodos y de esta forma tratar de mantener tamaños reducidos para los ⊕-OBDDs [MS98] sobre los que se aplicará la verificación probabilista.

La construcción de los ⊕-OBDDs con dos niveles de ⊕-nodos se realiza para circuitos combinacionales multinivel representados en formato BLIF, que describe un circuito jerárquico a nivel lógico de forma textual. En general, un circuito es una red arbitraria combinacional o secuencial de funciones lógicas, y se puede ver como un grafo dirigido de nodos lógicos combinacionales y elementos lógicos secuenciales. En esta representación, cada nodo tiene asociada una

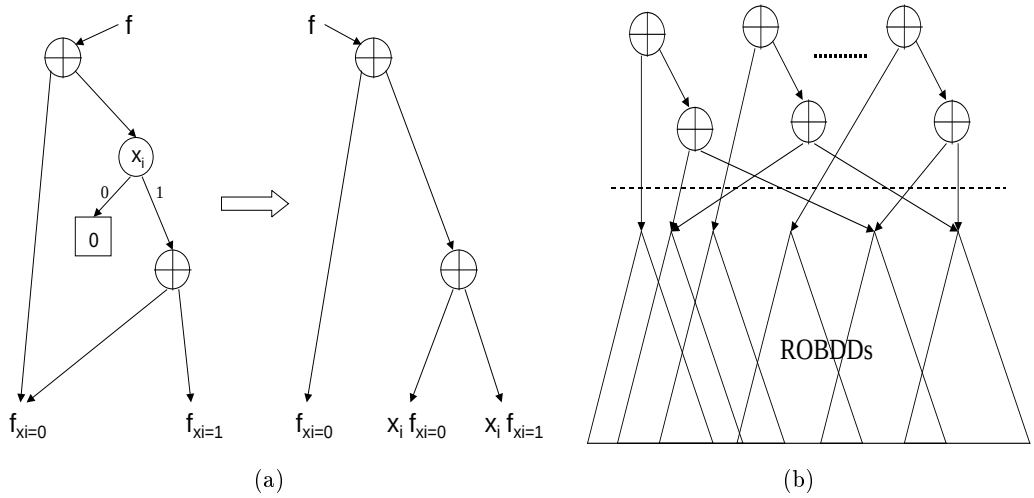


Figura 6.2: (a) Expansión pDE en la variable x_i. (b) Estructura del ⊕-OBDD.

función lógica de dos niveles con una sola salida, y cada lazo de realimentación debe contener al menos un elemento de almacenamiento (*latch*).

El formato BLIF tiene *entradas primarias*, *salidas* y *puertas internas*. Una puerta puede tener como entradas ($fanin$) únicamente entradas primarias o puede tener entradas primarias y/o salidas de puertas internas. En este formato textual, una puerta se representa con varias *líneas*, donde cada línea representa un *cubo* y donde la *disyunción* de esos cubos nos da la funcionalidad de esa puerta. Se tiene que, en general, esos cubos no son disjuntos. Cada *entrada* de una línea indica el valor (0, 1 o -) que toma la entrada correspondiente a esa puerta, de manera similar al formato PLA visto en el capítulo 5.

En la figura 6.3 se muestra un ejemplo de archivo en formato BLIF, donde las variables a, b y c son *entradas primarias*, las variables d, e, f y g son las *salidas* y las variables n y p representan *puertas internas*. Cada puerta del circuito se expresa por una línea encabezada por *.names* junto con el nombre de sus entradas y el nombre de su salida y a continuación una serie de líneas que representan los *cubos*, cuya disyunción describe la función de esa puerta.

6.4.1. Método de construcción

Para la construcción del ⊕-OBDD con una capa de dos niveles de ⊕-nodos, se selecciona en primer lugar una entrada primaria x_i con respecto de la cual se realiza la expansión positiva de Davio. Nosotros utilizamos la misma variable para la expansión pDE en todas las puertas del circuito. Si se realiza esta expansión, para cada puerta del circuito BLIF se obtienen dos funciones $F_{\overline{x}_i}$ y F_{x_i} que representan los cofactores negativo y positivo, respectivamente, con respecto a la variable x_i de la función realizada por dicha puerta.

```
.model b1
.inputs a b c
.outputs d e f g
.names n e
0 1
.names p f
0 1
.names c g
0 1
.names a b n
11 1
00 1
.names a b c p
01- 1
10- 1
1-1 1
-11 1
0-0 1
-00 1
.names c d
1 1
.end
```

Figura 6.3: Ejemplo de archivo en formato BLIF.

Para cualquier puerta del circuito, se pueden distinguir dos casos:

a) La puerta tiene únicamente *entradas primarias* como $fanin$. En este caso, para cada línea (*cubo*) de la puerta se realiza la *conjunción* de todas las entradas primarias distintas de x_i. Al resultado de esta conjunción le llamamos *producto*. A continuación se comprueba el valor que toma la entrada x_i para ese cubo y se calculan los *cofactores negativo* y *positivo* del cubo, representados como $C_{\overline{x}_i}$ y C_{x_i}, respectivamente. Los tres casos posibles que se pueden dar son los siguientes

x_i	$C_{\overline{x}_i}$	C_{x_i}
0	*producto*	0
1	0	*producto*
-	*producto*	*producto*

donde estos cálculos se realizan para cada línea (*cubo*) de la puerta. Finalmente se realiza la *disyunción* de todos los cofactores $C_{\overline{x}_i}$ de los cubos (obteniendo $F_{\overline{x}_i}$) y la *disyunción* de todos los cofactores C_{x_i} de los cubos (obteniendo F_{x_i}). De esta forma se calculan los dos cofactores $F_{\overline{x}_i}$ y F_{x_i} que representan la función realizada por esa puerta.

b) La puerta tiene *entradas primarias* y/o salidas de *puertas internas* como $fanin$. En este caso, se tiene que algunas entradas a la puerta (las correspondientes a *puertas internas*) ya han sido descompuestas con respecto a la variable x_i. Si representamos a la *conjunción* de los cofactores $F_{\overline{x}_i}$ de esas puertas internas como $N_{\overline{x}_i}$ y a la *conjunción* de los cofactores F_{x_i} de las puertas internas como N_{x_i}, entonces los cofactores $C_{\overline{x}_i}$ y C_{x_i} de cada cubo se obtienen de la siguiente forma según el valor de x_i

x_i	$C_{\overline{x}_i}$	C_{x_i}
0	$producto \cdot N_{\overline{x}_i}$	0
1	0	$producto \cdot N_{x_i}$
-	$producto \cdot N_{\overline{x}_i}$	$producto \cdot N_{x_i}$

Como en el caso anterior, estas operaciones se realizan para cada cubo de la puerta y finalmente la *disyunción* de todos los cofactores $C_{\overline{x}_i}$ de los cubos proporciona $F_{\overline{x}_i}$ y la *disyunción* de todos los cofactores C_{x_i} obtenidos para todas las líneas nos da F_{x_i}. Por tanto, la funcionalidad de la puerta vendrá representada por $F_{\overline{x}_i}$ y F_{x_i}.

Tanto las variables como las funciones mencionadas en los dos casos anteriores (cofactores y *producto*) se representan por ROBDDs (en realidad, por s-ROBDDs, ya que estamos interesados en el cálculo de signaturas), por lo que las operaciones realizadas con ellas también producen s-ROBDDs. Su construcción se realiza utilizando el algoritmo *ite* modificado para el cálculo de signaturas mientras se recorre el circuito multinivel dado originalmente en formato BLIF. Para la realización de esta síntesis, será necesario trasladar previamente el formato textual BLIF a un formato intermedio de listas enlazadas que refleje la topología de la red Booleana dada por el circuito multinivel. Es este formato intermedio el que es leido y sobre el que se aplica el algoritmo *ite* para la construcción.

Una vez obtenidos los dos cofactores $F_{\overline{x}_i}$ y F_{x_i} que representan las puertas del circuito, es inmediata la construcción del ⊕-OBDD con los dos niveles de ⊕-nodos de las salidas del circuito aplicando la ecuación 6.5, obteniéndose una estructura como la mostrada en la figura 6.2(a). El grafo creado para *puertas internas* presenta una estructura similar a la de la figura 6.2(a) excepto en que los cofactores del ⊕-nodo inferior no aparecen *multiplicados* por la variable x_i. Esto es debido a que la descomposición de todo el circuito se realiza con respecto a esta misma variable x_i y la construcción de las puertas internas se utiliza únicamente como paso intermedio para la construcción de las salidas del circuito, siendo únicamente necesario, por lo tanto, conocer los *cofactores* totales de las puertas internas. Para el caso de las *salidas* del circuito, sí que se realiza esta *multiplicación* de la variable x_i por los cofactores totales de las salidas $F_{\overline{x}_i}$ y F_{x_i}, obteniéndose la estructura dada en la figura 6.2(a).

Como se ha mencionado anteriormente, los ⊕-OBDDs obtenidos con este método no constituyen representaciones canónicas de las salidas del circuito, por lo que el tipo de verificación a utilizar sería la probabilista. Como lo que interesa es el cálculo de la signatura del circuito utilizando esta estructura de datos, se puede simplificar el grafo no incluyendo *explícitamente* la variable x_i en el mismo. La *contribución* de la variable se considera multiplicando la signatura del ⊕-nodo inferior de la capa de dos niveles, en la figura 6.2(a), por la signatura asignada inicialmente a la variable x_i. La estructura del ⊕-OBDD obtenida finalmente se muestra en la figura 6.2(b), con una *capa superior de dos niveles* de ⊕-nodos y una parte inferior formada por ROBDDs clásicos (en realidad, s-ROBDDs).

6.4.2. Resultados experimentales

Se han realizado experimentos para la verificación probabilista de los circuitos de benchmark LGSynth91 representados en formato BLIF multinivel por medio de la construcción de sus ⊕-OBDDs. Se ha utilizado para ello el paquete CUDD, modificándolo para permitir la inclusión de ⊕-nodos según el método visto anteriormente y usando las modificaciones vistas en el capítulo 5 para la inclusión y cálculo de signaturas sobre el campo de Galois $GF(2^m)$. La introducción de ⊕-nodos en la estructura de datos del s-ROBDD (obteniendo, por tanto, un ⊕-OBDD) se ha realizado creando una nueva *subtabla* para los ⊕-nodos diferente de las *subtablas* existentes para los distintos *nodos de decisión*.

Para el cálculo de las signaturas se ha utilizado el campo de Galois $GF(2^{16})$ generado por el polinomio irreducible $f(x) = x^{16} + x^5 + x^3 + x^2 + 1$ como campo finito. La base de representación de los elementos del campo utilizada ha sido la *base polinómica* y la multiplicación de elementos se ha realizado usando un algoritmo de multiplicación en dicha base (este algoritmo se estudiará en la subsección 7.2.1 perteneciente a la segunda parte de esta tesis). Asimismo, se ha asignado a cada variable de entrada de los circuitos a verificar un valor numérico aleatorio generado de forma independiente y uniformemente distribuido perteneciente a dicho campo finito.

Los experimentos se han realizado sobre una estación de trabajo SUN UltraSparc-II, utilizándose para la construcción de los ⊕-OBDDs las ordenaciones iniciales de variables dadas por las descripciones de los circuitos. Por motivos de comparación con los resultados dados por Meinel y Sack [MS01b], no se ha utilizado reordenación dinámica de variables para los s-ROBDDs existentes en la capa inferior de los ⊕-OBDDs.

Se ha visto anteriormente que para la construcción de los ⊕-OBDDs con una capa de dos niveles de ⊕-nodos se debe seleccionar una variable x_i con respecto de la cual realizar la descomposición positiva de Davio, que llamaremos *variable de descomposición*. Como se ha utilizado la ordenación original de variables

proporcionada por el circuito, en una primera aproximación se ha seleccionado como *variable de descomposición* la primera variable (*variable inicial*) de esa ordenación original y se han recogido los resultados. Posteriormente se ha seleccionado cada una de las variables restantes del circuito como *variable de descomposición*, comprobándose que existe una variable para la cual se obtienen los mejores resultados. A esta variable se le ha denominado *variable óptima* y se han recogido sus resultados correspondientes.

6.4.2.1. Comparación de tiempos

Se han construido en primer lugar los ROBDDs y los s-ROBDDs para algunos de los circuitos de benchmark LGSynth91 y se han comparado los tiempos necesarios para su construcción. Ya se vio que la diferencia entre estos dos tipos de grafos radica en que los s-ROBDDs incluyen y calculan signaturas para una asignación numérica aleatoria dada, teniendo ambos diagramas el mismo número de nodos.

En la tabla 6.1 se muestran los *ratios* de tiempos necesarios para la construcción de los benchmarks dados, donde $T_{s-ROBDD}/T_{ROBDD}$ representa el cociente entre el tiempo de construcción del s-ROBDD y el tiempo de construcción del ROBDD clásico.

Circuito	$T_{s-ROBDD}/T_{ROBDD}$
apex6	2.1
apex7	2.4
c1355	2.4
c1908	2.3
cm151a	2.5
cordic	1.5
des	2.9
frg2	2.3
i2	2.3
k2	2.9
mux	3.1
pcler8	2.0
term1	2.5
too_large	3.0
ttt2	2.0
vda	2.9
x3	2.3
x4	1.8
Total(†)	2.4

Tabla 6.1: *Ratios* de tiempo de construcción entre s-ROBDDs y ROBDDs para distintos benchmarks († = promedio).

De los resultados mostrados en la tabla 6.1 se puede observar que el tiempo de construcción de los s-ROBDDs es 2.4 veces mayor (en promedio) que el tiempo de construcción de los ROBDDs clásicos, lo que es lógico debido a las operaciones aritméticas que se deben realizar para el cálculo de signaturas. El peor caso de construcción del s-ROBDD corresponde al benchmark $mux.blif$, cuyo s-ROBDD se construye 3.1 veces más despacio que su ROBDD. Este tiempo de construcción mayor de un s-ROBDD frente a un ROBDD no parece ser, sin embargo, excesivo si se tiene en cuenta que con los s-ROBDDs se podrían realizar aún verificaciones de tipo probabilista en el caso de que los tamaños de los grafos fueran excesivamente grandes y no pudieran ser completados. En este caso, no se podría finalizar la verificación determinista pero las signaturas de los grafos construidos hasta ese momento podrían ser utilizadas para completar al menos una comprobación de equivalencia probabilista.

A continuación se han construido los ⊕-OBDDs con una capa superior de dos niveles de ⊕-nodos utilizando el método dado en la subsección 6.4.1. Como se ha utilizado la ordenación original de variables, en una primera aproximación se ha seleccionado como *variable de descomposición* la primera variable de esa ordenación original y se han medido sus tiempos de construcción. Posteriormente se han ido seleccionando cada una de las variables restantes como *variables de descomposición* y se ha comprobado que existe una variable para la cual se obtienen los mejores resultados. A esta variable se le ha denominado *variable óptima* y se han medido sus tiempos correspondientes de construcción.

Como estamos interesados en su aplicación a la verificación probabilista, en la tabla 6.2 se comparan los tiempos de construcción de los ⊕-OBDDs con los tiempos de los s-ROBDDs, mostrándose los *ratios* correspondientes para los benchmarks dados así como el promedio total. Los resultados mostrados en la tabla 6.2 corresponden a los ⊕-OBDDs construidos considerando como *variables de descomposición* la variable *inicial* de la ordenación original y la variable *óptima*, donde $T_{\oplus-OBDD}/T_{s-ROBDD}$ representa el cociente entre el tiempo de construcción del ⊕-OBDD y el tiempo de construcción del s-ROBDD.

En la tabla 6.2 se observa que cuando se selecciona la variable *inicial* como variable de descomposición, los tiempos de construcción de los ⊕-OBDDs son prácticamente idénticos a los tiempos de construcción de los s-ROBDDs. Por otra parte, si se selecciona como variable de descomposición la variable *óptima*, se observa que en prácticamente todos los benchmarks la construcción del ⊕-OBDD resulta ser más rápida que la del s-ROBDD, incluso para grandes circuitos como $c1908.blif$ (*ratio* de 0.45). El caso más favorable corresponde al benchmark $cm151a.blif$ para el que el tiempo de construcción del ⊕-OBDD es un 90 % menor que el tiempo de construcción de su s-ROBDD correspondiente. Considerando el promedio para todos los benchmarks, se obtiene que el tiempo de construcción de los ⊕-OBDDs cuando se utiliza la variable *óptima* como variable de descomposición es un 35 % menor que el de los s-ROBDDs.

	$T_{\oplus-OBDD}/T_{s-ROBDD}$	
Circuito	Inicial	Optima
apex6	0.98	0.64
apex7	0.99	0.24
c1355	1.00	1.00
c1908	1.01	0.45
cm151a	1.00	0.10
cordic	1.00	1.00
des	1.02	1.02
frg2	1.01	0.99
i2	0.95	0.71
k2	1.00	0.18
mux	0.99	0.99
pcler8	1.00	0.50
term1	1.00	0.75
too_large	0.99	0.65
ttt2	1.00	0.87
vda	0.99	0.24
x3	1.01	0.56
x4	1.01	0.93
Total(†)	0.99	0.65

Tabla 6.2: *Ratios* de tiempo de construcción entre ⊕-OBDDs y s-ROBDDs para las variables de descomposición *inicial* y *óptima* († = promedio).

En el siguiente apartado se presentan los resultados obtenidos en los experimentos para el número de nodos de los grafos y se dan algunas conclusiones acerca de estas mejoras en los tiempos de construcción, relacionándolas con las medidas de tamaño obtenidas.

6.4.2.2. Comparación de tamaños

En la tabla 6.3 se muestran los tamaños obtenidos para los s-ROBDDs y para los ⊕-OBDDs utilizando como *variables de descomposición* la variable *inicial* y la variable *óptima*. Los resultados muestran que no existen diferencias apreciables de tamaño entre los s-ROBDDs y los ⊕-OBDDs construidos con la variable inicial. Esto sería debido a que el número de ⊕-nodos introducidos en la capa superior no es lo suficientemente grande (comparado con el número de nodos de decisión existentes en la capa inferior) como para que puedan influir positivamente en el tamaño total del ⊕-OBDD. Esta misma consideración sería aplicable para los resultados temporales vistos en la tabla 6.2, donde los tiempos necesarios de construcción de los ⊕-OBDDs y de los s-ROBDDs eran prácticamente idénticos.

		$\oplus - OBDD$	
Circuito	s-ROBDD	Inicial	Optima
apex6	2760	2757	1520
apex7	1660	1658	526
c1355	45922	45953	45953
c1908	36007	36028	19697
cm151a	511	513	49
cordic	45	46	46
des	73919	73920	73496
frg2	6471	6605	5905
i2	335	335	268
k2	28336	28371	7333
mux	131071	131071	131071
pcler8	139	154	137
term1	580	577	447
too_large	7096	7097	5129
ttt2	223	207	198
vda	4345	4381	1919
x3	2760	2757	1520
x4	891	889	831
Total	343071	343319	296045

Tabla 6.3: Tamaños de los s-ROBDDs y de los ⊕-OBDDs (para las variables de descomposición *inicial* y *óptima*).

Los tamaños obtenidos, sin embargo, para los ⊕-OBDDs construidos usando la variable *óptima* como variable de descomposición son significativamente menores que los obtenidos con los s-ROBDDs. Especialmente apreciables son los resultados obtenidos para los benchmarks $cm151a$, $k2$, $apex7$, vda o $c1908$, que presentan reducciones de tamaños de sus ⊕-OBDDs con respecto de sus s-ROBDDs del 90.4 %, 74.1 %, 68.3 %, 55.8 % y 45.3 %, respectivamente. Para el total de los benchmarks construidos de esta forma, se obtiene que el tamaño de los ⊕-OBDDs es un 13.7 % menor que el de los s-ROBDDs.

Se observa que la selección de la variable *óptima* como variable de descomposición produce mejoras significativas en el tamaño de los ⊕-OBDDs, y como se ha visto en la tabla 6.2, también en el tiempo necesario para su construcción. Como el número de ⊕-nodos introducidos utilizando nuestro método de construcción no parece influir excesivamente sobre el tamaño total de los ⊕-OBDDs (comparándolo con el de los s-ROBDDs), esta mejora en el tamaño podría estar más relacionada con el problema de ordenación de variables que con la inclusión de los ⊕-nodos. Este hecho demostraría la fuerte influencia de una buena ordenación de variables sobre el tamaño de los diagramas de decisión en general, tanto (s)-ROBDDs clásicos como de otros tipos.

En nuestro método de síntesis se selecciona una *variable de descomposición*, pero se mantiene la ordenación original de las variables del circuito. Por lo tanto, se podría utilizar el algoritmo de ordenación heurístico desarrollado en el capítulo 4 en el que se obtenía una *variable óptima* que era colocada en primera posición de la nueva ordenación, desplazando en una posición la ordenación original de variables de la que se habría extraído dicha variable óptima. Aplicando esta heurística de ordenación a nuestro método de construcción de ⊕-OBDDs, la variable óptima obtenida por la heurística sería la variable de descomposición que se seleccionaría para la síntesis, con lo que la combinación de ambas aproximaciones podría producir resultados óptimos para la construcción de los ⊕-OBDDs.

Se han comparado también nuestros resultados con los más recientemente obtenidos por Meinel y Sack en [MS01b]. A nuestro conocimiento, estos son los únicos autores que han presentado resultados acerca de la utilización de los ⊕-OBDDs. Los datos mostrados en [MS01b] han sido obtenidos con la ordenación de variables original suministrada por los circuitos y sin la utilización de algoritmos de reordenación dinámica de variables. Por este motivo, nosotros tampoco hemos utilizado la reordenación de variables en nuestros experimentos. En la tabla 6.4 se muestran de nuevo los tamaños obtenidos con nuestro método de construcción (para las variables de descomposición *inicial* y *óptima*) junto con los mejores tamaños obtenidos por Meinel y Sack para algunos de los benchmarks LGSynth91 en común.

Circuito	Inicial	Optima	Meinel&Sack
apex7	1658	526	1570
c1355	45953	45953	45921
c1908	36028	19697	35869
frg2	6605	5905	6348
i2	335	268	334
k2	28371	7333	26361
mux	131071	131071	131072
term1	577	447	584
too_large	7097	5129	7091
vda	4381	1919	4214
x3	2757	1520	2429
Total	264833	219768	261793

Tabla 6.4: Comparación de tamaños de ⊕-OBDDs obtenidos utilizando nuestra aproximación (variables *inicial* y *óptima*) y los obtenidos por Meinel y Sack.

Los resultados dados en la tabla 6.4 muestran que para el total de los benchmarks verificados, el tamaño de los ⊕-OBDDs construidos usando nuestra aproximación y seleccionando la variable *inicial* como variable de descomposición resulta ser únicamente un 1.16 % mayor que el tamaño de los ⊕-OBDDs

construidos por Meinel y Sack. Sin embargo, si se selecciona la variable *óptima* para la construcción utilizando nuestro método, entonces se obtienen tamaños de ⊕-OBDDs que son un 16.1 % menores que los obtenidos por Meinel y Sack para el total de los benchmarks comprobados. Los mejores resultados se obtienen para los circuitos $k2$, $apex7$ y $c1908$ con una reducción del número de nodos del 72.1 %, 66.5 % y 45.1 %, respectivamente, con respecto a los ⊕-OBDDs construidos por Meinel y Sack.

En [MS01b] no se dan resultados acerca de los tiempos de ejecución necesarios para la construcción de los ⊕-OBDDs, sin embargo, el método de síntesis que emplean sus autores utiliza una aproximación dinámica en la que en cada paso de la construcción se comparan los tamaños de los grafos obtenidos usando distintas alternativas. Esta forma de construcción de los ⊕-OBDDs explicaría los mejores resultados de tamaño obtenidos por Meinel y Sack y permite deducir unos tiempos de construcción de ⊕-OBDDs muy elevados en comparación con los tiempos utilizados por nuestro método de síntesis.

6.5. Conclusiones

En este capítulo se han utilizado las estructuras de datos conocidas como ⊕-OBDDs para su aplicación a la verificación probabilista de circuitos combinacionales representados en un formato BLIF multinivel. La complejidad creciente de los circuitos utilizados en las aplicaciones actuales hace necesario el empleo de representaciones en formato multinivel de dichos circuitos, por lo que la comprobación de la equivalencia de circuitos multinivel pasa a ser un problema de gran importancia.

La utilización de ⊕-OBDDs en la verificación probabilista se debe a que este tipo de grafos incluyen, junto a estructuras ROBDD clásicas, unos nodos especiales (⊕-nodos) que realizan la XOR de las funciones representadas por sus subgrafos cofactores. La operación XOR (transformada) aplicada a los elementos de un campo de Galois $GF(2^m)$ se reduce simplemente a la XOR bit a bit de las representaciones binarias correspondientes a dichos elementos del campo, como se vio en el capítulo 5. De ahí que la utilización de ⊕-OBDDs como estructuras de representación en la verificación probabilista constituya una buena elección debido a que permiten simplificar en gran medida el proceso de cálculo de las signaturas.

El método de síntesis que se ha presentado en este capítulo realiza la construcción de ⊕-OBDDs con una *capa superior* de dos niveles de ⊕-nodos, utilizando para ello la *expansión positiva de Davio pDE* con respecto a una única variable de entrada seleccionada, llamada *variable de descomposición*. Este criterio se estableció debido a que el número de ⊕-nodos parece influir en el tamaño total de los ⊕-OBDDs construidos, por lo que un número no muy grande de ⊕-nodos podría conducir a tamaños pequeños de ⊕-OBDDs.

En los experimentos realizados se han construido $\oplus$-OBDDs para los circuitos de benchmark LGSynth91 representados originalmente en formato BLIF multinivel, utilizando para ello el método de síntesis propuesto y para una *variable de descomposición* seleccionada. De los resultados obtenidos, se ha podido observar la existencia de una variable de descomposición *óptima* para la cual tanto los tiempos de construcción como los tamaños de los $\oplus$-OBDDs obtenidos resultan ser significativamente menores que los obtenidos utilizando otras variables. Por ejemplo, la selección de la variable *inicial* dada por la ordenación original como variable de descomposición produce tiempos y tamaños de $\oplus$-OBDDs similares a los obtenidos por los *s*-ROBDDs "*clásicos*". Asimismo, la comparación de los resultados obtenidos por nuestro método de síntesis usando la variable *óptima* de descomposición con los resultados obtenidos por Meinel y Sack han demostrado el buen comportamiento de nuestra forma de construcción de $\oplus$-OBDDs, tanto en tamaño como en tiempo de construcción.

Estos diferentes comportamientos de los $\oplus$-OBDDs en función de la variable de descomposición seleccionada, parecen indicar que las mejoras obtenidas en tiempo y tamaño pueden estar más relacionadas con el problema de ordenación de variables que con el método de construcción utilizado para la inclusión de $\oplus$-nodos. De esta forma se demostraría la fuerte dependencia de los diagramas de decisión, tanto *clásicos* como de otro tipo, con la ordenación de sus variables de entrada. Por lo tanto, la combinación de nuestro método de síntesis y de nuestro algoritmo de ordenación heurístico (visto en el capítulo 4) para la determinación de la variable de *descomposición óptima* podría conducir a la mejora tanto de los tiempos de construcción como de los tamaños obtenidos para los $\oplus$-OBDDs. La combinación de ambos métodos para su aplicación a la verificación probabilista multinivel constituye uno de nuestros trabajos futuros de desarrollo.

Parte II

Campos de Galois

Capítulo 7

Campos de Galois

> *"Tu prieras publiquement Jacobi ou à Gauss de donner leur avis, non sur la vérité, mais sur l'importance de ces théorèmes ... Après cela il se trouvera, j'espère, des gens qui trouveront leur profit à déchiffrer tout ce gâchis."*[1]

La disciplina matemática del *álgebra* incluye la teoría de los campos finitos, cuyo desarrollo se remonta a principios del siglo XIX cuando Carl Friedrich Gauβ y Evariste Galois trabajaron en la *teoría general de campos finitos*. El trabajo previo fue desarrollado por Pierre de Fermat, Leonhard Euler, Joseph-Louis Lagrange y Adrien-Marie Legendre. Los campos finitos también se conocen como *campos de Galois* en honor a Evariste Galois [Ste89] y su trabajo fundamental en este tema. Los campos finitos con q elementos se representan como $GF(q)$, siendo los *campos de extensión* de $GF(2)$, representados como $GF(2^m)$, los que tienen gran importancia por sus aplicaciones técnicas.

Durante los últimos años los campos de Galois se han utilizado en un gran número de aplicaciones. Además de su aplicación a la *verificación probabilista* estudiada en la primera parte de la tesis, existen distintas áreas en las que se utilizan los campos finitos. Entre ellas se pueden mencionar los *códigos algebraicos* [Bla83], los *esquemas criptográficos* [Sch93], los *generadores de números aleatorios* [WP90], el *procesamiento digital de señal* [Bla85] y el *test VLSI* [GSB91]. Las dos primeras áreas juegan un papel importante en las comunicaciones digitales modernas, debido al número creciente de aplicaciones de los sistemas de comunicación en todos los campos de nuestra sociedad.

[1] Últimas palabras escritas por Evariste Galois en una carta dirigida a su amigo Chevalier el 29 de Mayo de 1832, un día antes del duelo por una mujer que acabó con su vida. En esta cita, Galois pide a su amigo que solicite públicamente a Jacobi o a Gauss que den su opinión sobre la importancia de sus teoremas, confiando que en el futuro hubiera alguien que encontrara de utilidad organizar sus ideas.

La transmisión y el almacenamiento de datos digitales van a menudo acompañados de la posibilidad de corrupción de los datos. Para evitar este problema, se puede introducir la redundancia de los datos antes de su transmisión o almacenamiento. Este es el concepto de la *codificación de canal*. Debido a la información adicional añadida a los datos, los códigos de canal pueden determinar si han ocurrido errores y dónde han ocurrido. Por ejemplo, los códigos de corrección de errores de Reed-Solomon son muy útiles en los sistemas de comunicación actuales, donde se utilizan en aplicaciones como la comunicación por satélite, las redes de computadores y los sistemas de almacenamiento óptico y magnético. Los códigos de Reed-Solomon se basan en la aritmética sobre campos de Galois. La utilización de campos con *característica* 2 permite una representación directa de datos binarios como elementos del campo, por este motivo los códigos de Reed-Solomon tienden a realizar la aritmética sobre el campo finito específico $GF(2^8)$. Este campo de Galois $GF(2^8)$ es muy utilizado en aplicaciones técnicas (como en los *compact disks*) y ha sido estandarizado para la comunicación vía satélite por la ESA (*European Space Agency*) y la NASA (*National Aeronautics and Space Administration*) [Kum83].

Otro de los riesgos que presentan los sistemas de comunicación consiste en la posibilidad de lectura no autorizada y la falsificación de los datos digitales. Existen áreas como el comercio bancario electrónico en que los aspectos de *seguridad* tienen un interés crucial obvio. Ejemplos de *esquemas criptográficos* ampliamente utilizados son el protocolo de intercambio de claves de Diffie-Hellman [DH76] y el esquema de ElGamal [ElG85].

Los sistemas que utilizan aritmética sobre campos finitos deben ser *rápidos* debido al continuo incremento de prestaciones necesario en los sistemas de comunicación actuales. Para poder alcanzar estos requerimientos, puede ser necesaria la implementación en circuitos integrados de los módulos que realizan la aritmética sobre campos de Galois. Las implementaciones hardware de la aritmética en campos de Galois se realizan normalmente sobre circuitos integrados VLSI [YRT84][WTS$^+$85][FBT96], aunque recientemente se están comenzando a utilizar plataformas *reconfigurables* [Kli95][PR97][EP99][OP00][EP00] [EYCP00][Ima02] para la implementación.

Los módulos aritméticos sobre campos de Galois se pueden clasificar en arquitecturas *serie* y *paralelo*. La aritmética *paralela* [Paa94] tiende a ser más rápida que la *serie* y se implementa utilizando únicamente lógica combinacional, mientras que la aritmética *serie* requiere una implementación secuencial y necesita normalmente un área menor que su homóloga *paralela*. Asimismo, las métricas del rendimiento utilizadas habitualmente para la comparación de estas arquitecturas aritméticas son las complejidades *espacial* (necesidades de área) y *temporal* (retardo del circuito). La necesidad de que los módulos aritméticos que operan sobre campos de Galois funcionen a frecuencias elevadas y ocupen el menor área posible, junto con el hecho de que la ope-

ración aritmética más costosa e importante sea la multiplicación, hacen que la búsqueda de algoritmos de multiplicación y de arquitecturas eficientes de módulos *multiplicadores* operando sobre $GF(2^m)$ sea de especial importancia. Además, la mayor parte de las operaciones aritméticas avanzadas, como la *exponenciación* o la *inversión*, se basan en la multiplicación. La complejidad de un módulo multiplicador depende de factores como la selección de la *base* de representación o del *polinomio irreducible* generador del campo finito. De entre las distintas bases de representación de los elementos del campo $GF(2^m)$ existentes, las más utilizadas son las bases *canónica*, *normal* y *dual*, aunque existen otras bases posibles como la base *triangular*.

En este capítulo se presentan los conceptos teóricos básicos de los campos finitos y de los campos de extensión, así como las diferentes bases de representación de los elementos del campo finito. Se estudian los importantes campos de extensión $GF(2^m)$ y los tipos de multiplicación más importantes sobre estos campos, centrándonos en su arquitectura paralela. Para cada uno de los multiplicadores vistos, se determinan también sus complejidades *espacial* y *temporal* teóricas. También se introducen las diferentes aproximaciones de implementación hardware y las metodologías de diseño existentes. Finalmente, se presentan resultados experimentales correspondientes a la implementación *reconfigurable* paralela sobre FPGAs de los tres tipos de multiplicadores estudiados y para distintos polinomios generadores del campo, realizándose la comparación de los resultados obtenidos. La mayor parte de las definiciones, teoremas y corolarios (con sus demostraciones) que se presentan en este capítulo se pueden encontrar en el excelente libro de Lidl y Niederreiter [LN83].

7.1. Campos finitos

El campo de enteros módulo un número primo es el ejemplo más familiar de campo finito, pero muchas de sus propiedades se extienden a campos finitos arbitrarios. La caracterización de los campos finitos muestra que todo campo finito es de orden la potencia de un número primo y que para toda potencia de un primo existe un campo finito cuyo número de elementos es exactamente esa potencia. Además, los campos finitos con el mismo número de elementos son isomorfos y se pueden identificar.

Para todo *primo* p, el anillo de clases residuales $\mathcal{Z}/(p)$ de los enteros módulo el ideal principal generado por un primo p, forma un campo finito con p elementos y se puede identificar con el *campo de Galois* $GF(p)$ de orden p. Comenzando con los campos primos $GF(p) = \mathcal{F}_p$, se pueden construir otros campos por el proceso de *adjunción de raíces*. A continuación se revisan conceptos como el *campo de extensión* y las *raíces de polinomios irreducibles* que llevan a la interpretación de los campos finitos como campos de descomposición de polinomios irreducibles.

7.1.1. Campos de extensión

Sea $\mathcal{F}$ un campo. Un subconjunto $\mathcal{K}$ de $\mathcal{F}$, que es asimismo un campo bajo las operaciones de $\mathcal{F}$, se llama un *subcampo* de $\mathcal{F}$ y en este contexto, $\mathcal{F}$ se llama un *campo de extensión* de $\mathcal{K}$. Si $\mathcal{K} \neq \mathcal{F}$, entonces se dice que $\mathcal{K}$ es un *subcampo propio* de $\mathcal{F}$.

Definición 14 *Sea $\mathcal{K}$ un subcampo del campo $\mathcal{F}$ y sea $\mathcal{M}$ cualquier subconjunto de $\mathcal{F}$. Entonces, el campo $\mathcal{K}(\mathcal{M})$ se define como la intersección de todos los subcampos de $\mathcal{F}$ que contienen a $\mathcal{K}$ y a $\mathcal{M}$ y se llama el campo de extensión de $\mathcal{K}$ obtenido adjuntando los elementos de $\mathcal{M}$. Para $\mathcal{M}$ finito, $\mathcal{M} = \{\theta_1, \ldots, \theta_n\}$, se escribe $\mathcal{K}(\mathcal{M}) = K(\theta_1, \ldots, \theta_n)$. Si $\mathcal{M}$ consta de un solo elemento $\theta \in \mathcal{F}$, entonces $\mathcal{L} = \mathcal{K}(\theta)$ se dice que es una extensión simple de $\mathcal{K}$ y θ se llama un elemento definitorio de $\mathcal{L}$ sobre $\mathcal{K}$.*

Definición 15 *Sea $\mathcal{K}$ un subcampo de $\mathcal{F}$ y sea $\theta \in \mathcal{F}$. Si θ satisface una ecuación polinómica no trivial con coeficientes en $\mathcal{K}$, es decir, si se cumple que $a_n \cdot \theta^n + \cdots + a_1 \cdot \theta + a_0 = 0$ con $a_i \in \mathcal{K}$ no todos cero, entonces θ se dice que es algebraico sobre $\mathcal{K}$. Una extensión $\mathcal{L}$ de $\mathcal{K}$ se llama algebraica sobre $\mathcal{K}$ (o una extensión algebraica de $\mathcal{K}$) si todo elemento de $\mathcal{L}$ es algebraico sobre $\mathcal{K}$. En caso contrario se llama extensión trascendente.*

Definición 16 *Si $\theta \in \mathcal{F}$ es algebraico sobre $\mathcal{K}$, entonces el único polinomio mónico $g \in \mathcal{K}[x]$ que genera el ideal $\mathcal{J} = \{f \in \mathcal{K}[x] : f(\theta) = 0\}$ de $\mathcal{K}[x]$ (el anillo de polinomios sobre $\mathcal{K}$) se llama el polinomio mínimo (o irreducible o definitorio) de θ sobre $\mathcal{K}$. Por el grado de θ sobre $\mathcal{K}$ se refiere al grado del polinomio g.*

El campo de extensión $\mathcal{L}$ de $\mathcal{K}$ se puede considerar como un espacio vectorial sobre $\mathcal{K}$.

Definición 17 *Sea $\mathcal{L}$ un campo de extensión de $\mathcal{K}$. Si $\mathcal{L}$, considerado como un espacio vectorial sobre $\mathcal{K}$, es de dimensión finita, entonces $\mathcal{L}$ se llama una extensión finita de $\mathcal{K}$. A la dimensión del espacio vectorial $\mathcal{L}$ sobre $\mathcal{K}$ se le llama entonces el grado de $\mathcal{L}$ sobre $\mathcal{K}$ y se representa como $[\mathcal{L} : \mathcal{K}]$.*

Dada una extensión simple $\mathcal{K}(\theta)$ de $\mathcal{K}$ obtenida adjuntando un elemento algebraico, se observa que si $\mathcal{F}$ es una extensión de $\mathcal{K}$ y $\theta \in \mathcal{F}$ es algebraico sobre $\mathcal{K}$, entonces $\mathcal{K}(\theta)$ es una extensión finita y algebraica de $\mathcal{K}$. Además, $\mathcal{K}(\theta)$ será isomorfo a $\mathcal{K}[x]/(g)$ si $\theta \in \mathcal{F}$ es algebraico de grado n sobre $\mathcal{K}$ y si g es el polinomio mínimo de θ sobre $\mathcal{K}$. También se puede demostrar que los elementos de la extensión algebraica simple $\mathcal{K}(\theta)$ de $\mathcal{K}$ son expresiones polinómicas en θ y que cualquier elemento de $\mathcal{K}(\theta)$ se puede representar de forma única como $a_0 + a_1 \cdot \theta + \cdots + a_{n-1} \cdot \theta^{n-1}$ con $a_i \in \mathcal{K}$ para $0 \leq i \leq n-1$, donde $n = [\mathcal{K}(\theta) : \mathcal{K}]$ y donde $\{1, \theta, \ldots, \theta^{n-1}\}$ es una base de $\mathcal{K}(\theta)$ sobre $\mathcal{K}$.

Teorema 14 *Sea $f \in \mathcal{K}[x]$ irreducible sobre el campo $\mathcal{K}$. Entonces existe una extensión algebraica simple de $\mathcal{K}$ con una raíz de f como elemento definitorio.*

Se puede encontrar un campo de extensión al que pertenezcan todas las raíces de un polinomio dado.

Definición 18 *Sea $f \in \mathcal{K}[x]$ de grado positivo y sea $\mathcal{F}$ un campo de extensión de $\mathcal{K}$. Entonces se dice que f se descompone en $\mathcal{F}$ si f se puede escribir como un producto de factores lineales en $\mathcal{F}[x]$, es decir, si existen elementos $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathcal{F}$ tales que*

$$f(x) = a(x - \alpha_1)(x - \alpha_2) \cdots (x - \alpha_n) \tag{7.1}$$

donde a es el coeficiente principal de f. El campo $\mathcal{F}$ es un campo de descomposición de f sobre $\mathcal{K}$ si f se descompone en $\mathcal{F}$ y si $\mathcal{F} = \mathcal{K}(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

El campo de descomposición de f sobre $\mathcal{K}$ se obtiene adjuntando una cantidad finita de elementos algebraicos a $\mathcal{K}$. Por tanto, se puede demostrar que el campo de descomposición de f sobre $\mathcal{K}$ es una extensión finita de $\mathcal{K}$.

7.1.2. Caracterización de campos finitos

Con respecto al número de elementos de un campo finito, se puede demostrar que si $\mathcal{F}$ es un campo finito que contiene un subcampo $\mathcal{K}$ con q elementos, entonces $\mathcal{F}$ tiene q^m elementos, donde $m = [\mathcal{F} : \mathcal{K}]$. También se puede establecer el siguiente teorema.

Teorema 15 *Un campo finito $\mathcal{F}$ tiene p^n elementos, donde el primo p es la característica de $\mathcal{F}$ y donde n es el grado de $\mathcal{F}$ sobre su subcampo primo.*

Los campos $\mathcal{F}_p$ juegan un papel muy importante en la teoría general de campos. Como se ha visto anteriormente, comenzando con los campos primos $\mathcal{F}_p$ se pueden construir otros campos finitos a través del proceso de adjunción de raíces. Si $f \in \mathcal{F}_p[x]$ es un polinomio irreducible sobre $\mathcal{F}_p$ de grado n, entonces adjuntando una raíz de f a $\mathcal{F}_p$ se obtiene un campo finito con p^n elementos (campo de extensión de $\mathcal{F}_p$). Otra propiedad importante es que para todo primo p y para todo $n \in \mathcal{N}$ hay un campo finito con p^n elementos, por lo que se puede hablar de *el* campo finito (o *el* campo de Galois) con $q = p^n$ elementos, o de *el* campo finito (o *el* campo de Galois) de orden q. Este campo se denota como $\mathcal{F}_q = GF(p^n)$, donde q es una potencia de la característica prima p de $\mathcal{F}_q$. Los p^n elementos del campo $GF(p^n)$ se pueden representar como polinomios con grado máximo $n - 1$ sobre $GF(p)$, donde estos p^n polinomios son las clases residuales módulo $f(x)$ de todos los polinomios sobre $GF(p)$.

Para un campo finito $\mathcal{F}_q$ se denota por $\mathcal{F}_q^*$ al grupo multiplicativo de elementos diferentes de cero de $\mathcal{F}_q$, que tiene la propiedad de ser *cíclico*.

Definición 19 *Un generador del grupo cíclico $\mathcal{F}_q^*$ se llama elemento primitivo de $\mathcal{F}_q$.*

De la existencia de elementos primitivos se tiene que todo campo finito se puede considerar como una extensión algebraica simple de su subcampo primo.

Teorema 16 *Sea $\mathcal{F}_q$ un campo finito y sea $\mathcal{F}_r$ un campo de extensión finito. Entonces, $\mathcal{F}_r$ es una extensión algebraica simple de $\mathcal{F}_q$ y todo elemento primitivo de $\mathcal{F}_r$ puede servir como elemento definitorio de $\mathcal{F}_r$ sobre $\mathcal{F}_q$.*

7.1.3. Raíces de polinomios irreducibles

Se revisan a continuación algunos conceptos importantes de las raíces de un polinomio irreducible sobre un campo finito [LN83].

Teorema 17 *Si f es un polinomio irreducible en $\mathcal{F}_q[x]$ de grado m, entonces f tiene una raíz α en $\mathcal{F}_{q^m}$. Además, todas las raíces de f son simples y están dadas por los m elementos distintos $\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{m-1}}$ de $\mathcal{F}_{q^m}$.*

Corolario 3 *Sea f un polinomio irreducible en $\mathcal{F}_q[x]$ de grado m. Entonces el campo de descomposición de f sobre $\mathcal{F}_q$ está dado por $\mathcal{F}_{q^m}$.*

Se puede dar una terminología conveniente para los elementos mencionados en el teorema 17, independientemente de si $\alpha \in \mathcal{F}_{q^m}$ es una raíz de un polinomio irreducible en $\mathcal{F}_q[x]$ de grado m o no.

Definición 20 *Sea $\mathcal{F}_{q^m}$ una extensión de $\mathcal{F}_q$ y sea $\alpha \in \mathcal{F}_{q^m}$. Entonces, los elementos $\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{m-1}}$ se llaman conjugados de α con respecto a $\mathcal{F}_q$.*

Los conjugados de $\alpha \in \mathcal{F}_{q^m}$ con respecto a $\mathcal{F}_q$ son distintos si y sólo si el polinomio mínimo de α sobre $\mathcal{F}_q$ tiene grado m. También se cumple que si α es un elemento primitivo de $\mathcal{F}_q$, entonces también lo son sus conjugados con respecto a cualquier subcampo de $\mathcal{F}_q$.

7.1.4. Bases de campos finitos

Si se considera de nuevo una extensión finita $\mathcal{F} = \mathcal{F}_{q^m}$ del campo finito $\mathcal{K} = \mathcal{F}_q$ como un espacio vectorial sobre $\mathcal{K}$, entonces $\mathcal{F}$ tiene dimensión m sobre $\mathcal{K}$ y si $\{\alpha_1, \ldots, \alpha_m\}$ es una base de $\mathcal{F}$ sobre $\mathcal{K}$, entonces cada elemento $\alpha \in \mathcal{F}$ se puede representar de forma única como

$$\alpha = c_1\alpha_1 + \cdots + c_m\alpha_m, \quad c_j \in \mathcal{K}, \quad 1 \leq j \leq m. \tag{7.2}$$

Se puede establecer la siguiente correspondencia lineal de $\mathcal{F}$ en $\mathcal{K}$.

Definición 21 *La traza* $Tr_{\mathcal{F}/\mathcal{K}}(\alpha)$ *de* α *sobre* $\mathcal{K}$ *se define como*

$$Tr_{\mathcal{F}/\mathcal{K}}(\alpha) = \alpha + \alpha^q + \cdots + \alpha^{q^{m-1}} \tag{7.3}$$

donde $\alpha \in \mathcal{F} = \mathcal{F}_{q^m}$ *y* $\mathcal{K} = \mathcal{F}_q$.

Si $\mathcal{K}$ es el subcampo primo de $\mathcal{F}$, entonces la $\mathrm{Tr}_{\mathcal{F}/\mathcal{K}}(\alpha)$ se llama la *traza absoluta de* α y se representa por $\mathrm{Tr}_{\mathcal{F}}(\alpha)$. Se observa que la traza de α sobre $\mathcal{K}$ es la suma de los conjugados de α con respecto a $\mathcal{K}$ y es un elemento de $\mathcal{K}$.

Definición 22 *Sea* $\mathcal{K}$ *un campo finito y sea* $\mathcal{F}$ *una extensión finita de* $\mathcal{K}$. *Entonces, dos bases* $\{\alpha_1, \ldots, \alpha_m\}$ *y* $\{\beta_1, \ldots, \beta_m\}$ *de* $\mathcal{F}$ *sobre* $\mathcal{K}$ *se dice que son bases duales (o complementarias) si*

$$Tr_{\mathcal{F}/\mathcal{K}}(\alpha_i\beta_j) = \begin{cases} 0 & si\ i \neq j, \\ 1 & si\ i = j. \end{cases} \tag{7.4}$$

para $1 \leq i, j \leq m$.

Se puede comprobar que para cualquier base $\{\alpha_1, \ldots, \alpha_m\}$ de $\mathcal{F}$ sobre $\mathcal{K}$ existe una base dual $\{\beta_1, \ldots, \beta_m\}$ que está determinada de forma única.

Aunque el número de bases diferentes de $\mathcal{F}$ sobre $\mathcal{K}$ es bastante grande, hay dos tipos de bases de especial importancia. La primera de ellas es la *base polinómica* $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, construida con las potencias de un elemento definitorio α de $\mathcal{F}$ sobre $\mathcal{K}$, donde el elemento α se elige a menudo para que sea un elemento primitivo de $\mathcal{F}$. Esta base está directamente relacionada con la representación de los elementos del campo como polinomios, como se vio anteriormente. En este caso, un elemento del campo A se representa por el polinomio $A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{m-1}x^{m-1}$ y cada elemento representa una clase residual módulo el polinomio irreducible $f(x)$ de grado m sobre $\mathcal{K} = \mathcal{F}_q = GF(q)$. Como α es una raíz de $f(x)$, la representación polinómica $A(x)$ es equivalente a $A(\alpha) = a_0 + a_1\alpha + a_2\alpha^2 + \cdots + a_{m-1}\alpha^{m-1}$.

El otro tipo de base de especial interés es una *base normal* definida por un elemento apropiado de $\mathcal{F}$. Un $\mathcal{K}$-*automorfismo* de $\mathcal{F}$ es un automorfismo de $\mathcal{F} = \mathcal{F}_{q^m} = GF(q^m)$ que deja fijo todo elemento de $\mathcal{K} = \mathcal{F}_q = GF(q)$. El conjunto de los $\mathcal{K}$-*automorfismos* de $\mathcal{F}$ es un grupo, llamado el *grupo de Galois* de $\mathcal{F}$ sobre $\mathcal{K}$, generado por el automorfismo *Frobenius* $\sigma(\alpha) = \alpha^q$ para $\alpha \in \mathcal{F}$ y formado por los m elementos distintos $G_0, G_1, \ldots, G_{m-1}$ definidos de la siguiente forma

$$\begin{array}{rcl} G_i : \mathcal{F} & \longrightarrow & \mathcal{F} \\ \alpha & \longrightarrow & \alpha^{q^i} = \alpha G_i, \quad \alpha \in \mathcal{F} \end{array}$$

donde $G_i = G_1^i$ y $G_1^m = G_1^0 = G_0 = I$ (*automorfismo identidad*). Entonces, una base $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ será una *base normal* para $\mathcal{F}$ sobre $\mathcal{K}$ si $\beta_i = \alpha G_i$

para algún α de $\mathcal{F}$. Por tanto, el conjunto $\{\alpha, \alpha^q, \alpha^{q^2} \ldots, \alpha^{q^{m-1}}\}$ donde α es un elemento adecuado de $\mathcal{F}$ será una *base normal* si los m elementos son linealmente independientes y α será el elemento *generador* de la base normal, conocido como *elemento normal*.

Definición 23 *Sean $\mathcal{K} = \mathcal{F}_q$ y $\mathcal{F} = \mathcal{F}_{q^m}$. Entonces, una base de $\mathcal{F}$ sobre $\mathcal{K}$ de la forma $\{\alpha, \alpha^q, \ldots, \alpha^{q^{m-1}}\}$ que consta de un elemento apropiado $\alpha \in \mathcal{F}$ y sus conjugados con respecto a $\mathcal{K}$, se llama base normal de $\mathcal{F}$ sobre $\mathcal{K}$ y α es un elemento normal.*

Las *bases normales* y los polinomios sobre $\mathcal{K}$ se relacionan a través de la siguiente propiedad. Si $f \in \mathcal{K}[x]$ es un polinomio irreducible de grado m con raíces linealmente independientes sobre $\mathcal{K}$, entonces cualquier raíz $\alpha \in \mathcal{F}$ es un *elemento normal* de $\mathcal{F}$ sobre $\mathcal{K}$.

Existen distintos métodos para comprobar la *normalidad* de un elemento $\alpha \in \mathcal{F}$ [Men93]. Uno de estos métodos, que utilizaremos posteriormente, establece que si $m = p^t$, donde p es la *característica* de $\mathcal{K}$, entonces un elemento $\alpha \in \mathcal{F}$ es un *elemento normal* sobre $\mathcal{K}$ si y sólo si $\mathrm{Tr}_{\mathcal{F}/\mathcal{K}}(\alpha) \neq 0$.

Por último, se puede demostrar que para cualquier campo finito $\mathcal{K}$ y para cualquier extensión finita $\mathcal{F}$ de $\mathcal{K}$ existe una base normal de $\mathcal{F}$ sobre $\mathcal{K}$.

7.2. Campos de extensión de $GF(2)$

Utilizando los conceptos dados en la sección anterior, se puede considerar el campo finito $GF(2^m) = \mathcal{F}_{2^m}$ que recientemente se está utilizando en un gran número de aplicaciones. Se puede observar que este campo $GF(2^m)$ es una extensión algebraica simple de $GF(2) = \mathcal{F}_2$ por el teorema 16. De hecho, si f es un polinomio *irreducible* en $GF(2)[x]$ de grado m, entonces f tiene una raíz α en $GF(2^m)$ según el teorema 17 y, por lo tanto, $GF(2^m) = \mathcal{F}_{2^m} = \mathcal{F}_2(\alpha)$. Entonces, todo elemento de $GF(2^m)$ se puede expresar de forma única como un polinomio en α sobre $GF(2)$ de grado menor que m. También se puede ver $\mathcal{F}_{2^m}$ como el anillo de clase residual $\mathcal{F}_2[x]/(f)$.

La aritmética en un campo finito de característica 2 es fundamentalmente una aritmética *modular*. Los elementos del campo $GF(2^m)$ son los polinomios $\{0, 1, \alpha, \alpha+1, \alpha^2, \alpha^2+1, \ldots, \alpha^{m-1}+\alpha^{m-2}+\ldots+\alpha+1\}$, que se pueden representar como números decimales que varían entre 0 y $2^m - 1$. La *adición* de estos polinomios se realiza bajo la aritmética *módulo* 2, por lo que la suma de dos polinomios se convierte simplemente en la XOR bit a bit de sus representaciones binarias. A su vez, la *sustracción* es exactamente igual que la adición en la aritmética *módulo* 2, por lo que $1 - \alpha$ es igual que $1 + \alpha$.

La *multiplicación* de dos polinomios es la operación aritmética sobre campos finitos más importante y una de las más complejas y con mayor consumo de

tiempo para su realización. Su complejidad puede depender de varios factores, como la selección del polinomio irreducible o la base elegida para representar los elementos del campo. Como se ha visto en la sección anterior, existen diferentes bases de representación, siendo las más habitualmente utilizadas las bases *polinómica*, *normal* y *dual*. Por lo tanto, los tres tipos diferentes de multiplicación en campos finitos que se pueden considerar son la multiplicación en base *polinómica* (o *canónica* o *estándar*), *normal* y *dual*, que además son muy adecuadas para implementaciones VLSI. Las multiplicaciones *polinómica* [YRT84] y *normal* [WTS+85] utilizan las bases *canónica* y *normal*, respectivamente, para la representación de los elementos del campo, mientras que la multiplicación en base *dual* [FBT96] utiliza la base canónica para representar un operando y la base dual para representar el otro operando y el producto resultante. En este caso, la complejidad de la conversión de base depende en gran medida de la selección del polinomio irreducible primitivo que genera el campo. Si el polinomio se selecciona de una forma adecuada, la conversión de base resulta en una operación muy simple.

A continuación vemos estas tres aproximaciones de multiplicación mencionadas, centrándonos en sus arquitecturas paralelas. Para cada una de ellas, se presentan sus complejidades teóricas *espaciales* (por medio del número de puertas AND y XOR utilizadas en la implementación paralela) y *temporales* (especificando los retardos de puertas AND y XOR).

7.2.1. Multiplicación en base polinómica

La multiplicación en base *polinómica* opera sobre la base *polinómica* para todos los elementos del campo. Los elementos no nulos de $GF(2^m)$ son generados por un elemento primitivo α, donde α es una raíz de un polinomio irreducible primitivo $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1 x + f_0$ sobre $GF(2)$. Los elementos no nulos de $GF(2^m)$ se pueden representar como potencias de α, es decir, $GF(2^m) = \{0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}, \alpha^{2^m-1} = 1\}$. Como $f(\alpha) = 0$, se tiene que $\alpha^m = f_{m-1}\alpha^{m-1} + \cdots + f_1\alpha + f_0$. Por tanto, un elemento de $GF(2^m)$ se puede expresar también como un polinomio de α con grado menor que m, es decir, $GF(2^m) = \{a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0 | a_i \in GF(2) \text{ para } 0 \leq i \leq m-1\}$.

Sean $A = a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0$ y $B = b_{m-1}\alpha^{m-1} + \cdots + b_1\alpha + b_0$ dos elementos pertenecientes al campo $GF(2^m)$. Supóngase que P es el producto de A y B, es decir, $P = A \cdot B = p_{m-1}\alpha^{m-1} + \cdots + p_1\alpha + p_0$. Entonces, P se puede escribir como [Ber68][YRT84]

$$P = \sum_{k=0}^{m-1}(A\alpha^k)b_k = \sum_{k=0}^{m-1}(\sum_{n=0}^{m-1} a_n^{(k)}\alpha^n)b_k = \sum_{n=0}^{m-1}(\sum_{k=0}^{m-1} a_n^{(k)}b_k)\alpha^n \qquad (7.5)$$

donde $a_n^{(k)}$ es el coeficiente de α^n en $A\alpha^k$, es decir, $A\alpha^k = a_{m-1}^{(k)}\alpha^{m-1} + \cdots + a_1^{(k)}\alpha + a_0^{(k)}$ para $0 \leq k \leq m-1$. De la ecuación 7.5 se obtiene que los coeficientes

p_n (con $0 \leq n \leq m-1$) del producto P vienen dados por la expresión

$$p_n = a_n^{(0)} b_0 + a_n^{(1)} b_1 + \cdots + a_n^{(m-2)} b_{m-2} + a_n^{(m-1)} b_{m-1} \tag{7.6}$$

El cálculo de $A\alpha^k$ se puede realizar recursivamente en k para $0 \leq k \leq m-1$. Inicialmente, para $k = 0$, se tiene que $A\alpha^0 = A$, es decir, $a_n^{(0)} = a_n$ para $0 \leq n \leq m-1$. Para $1 \leq k \leq m-1$, se tendrá que

$$A\alpha^k = (A\alpha^{k-1})\alpha = \sum_{n=0}^{m-1} a_n^{(k-1)} \alpha^{n+1} = a_{m-1}^{(k-1)} \alpha^m + \sum_{n=1}^{m-1} a_{n-1}^{(k-1)} \alpha^n \tag{7.7}$$

Sustituyendo $\alpha^m = f_{m-1}\alpha^{m-1} + \cdots + f_1\alpha + f_0$ en esta expresión se obtiene

$$A\alpha^k = \sum_{n=1}^{m-1} (a_{n-1}^{(k-1)} + a_{m-1}^{(k-1)} f_n)\alpha^n + a_{m-1}^{(k-1)} f_0 \tag{7.8}$$

De esta última expresión, se obtiene finalmente

$$\begin{aligned} a_n^{(k)} &= a_{n-1}^{(k-1)} + a_{m-1}^{(k-1)} f_n \quad para \quad 1 \leq n \leq m-1 \\ a_0^{(k)} &= a_{m-1}^{(k-1)} f_0 \end{aligned} \tag{7.9}$$

A partir de las ecuaciones recursivas 7.9 se pueden obtener los coeficientes del producto de los elementos A y B pertenecientes al campo finito $GF(2^m)$, donde la multiplicación se ha realizado utilizando la base canónica de representación. Este algoritmo de multiplicación en base polinómica es el que fue utilizado en la primera parte de la tesis dedicada a la *verificación*, donde los productos se realizaron sobre el campo $GF(2^{16})$. A continuación analizamos su complejidad teórica.

7.2.1.1. Análisis de complejidad

La complejidad teórica de la implementación paralela del módulo multiplicador en base canónica sobre $GF(2^m)$ utilizando este algoritmo de multiplicación depende del polinomio irreducible $f(x)$ seleccionado y se puede obtener a partir de las ecuaciones 7.5 y 7.9. La complejidad *espacial* se mide por el número de puertas AND y puertas XOR utilizadas en la implementación, mientras que para la complejidad *temporal* se tiene en cuenta el retardo del circuito medido en retardos de puertas AND y XOR, representado por T_{AND} y T_{XOR}, respectivamente.

Con respecto de la complejidad *espacial* y para valores $0 \leq i \leq m-1$, se observa que los términos $a_i^{(0)}$ no presentan consumo de puertas ya que coinciden con las señales de entrada a_i. A partir de la ecuación 7.9 se tiene que los términos $a_0^{(j)}$ tampoco presentan un consumo adicional de puertas ya que se

calculan a partir de términos $a_{m-1}^{(k-1)}$ ya construidos, mientras que los términos $a_1^{(j)}, a_2^{(j)}, \ldots, a_{m-1}^{(j)}$, $1 \leq j \leq m-1$, requieren una puerta XOR adicional cada uno. Por otra parte, se observa que los términos $a_i^{(j)}$, $0 \leq i, j \leq m-1$, son distintos de cero, independientemente del polinomio irreducible seleccionado, por lo que el cálculo de los productos $a_i^{(j)} b_j$, $0 \leq i, j \leq m-1$, dados en la ecuación 7.5 requiere la utilización de m^2 puertas AND. Las sumas de estos términos producto (ecuación 7.6, con $0 \leq n \leq m-1$) requieren $m(m-1)$ puertas XOR, que junto con las $(m-1)(m-1)$ puertas XOR utilizadas para los términos $a_i^{(j)} b_j$, $1 \leq i, j \leq m-1$, harían un total de $(m-1)(m-1)+m(m-1) = 2m^2-3m+1$ puertas XOR, en el caso peor de utilizar un polinomio irreducible con ningún coeficiente nulo.

Se puede establecer una expresión general para el número de puertas XOR necesarias en función del número de coeficientes no nulos del polinomio $f(x)$ seleccionado. Para ello se utiliza el *peso de Hamming* de $f(x)$, es decir, el número de coeficientes distintos de cero y que representamos como $h_w(f)$. Al estar utilizando polinomios mónicos irreducibles, se tiene que los coeficientes f_0 y f_m son la unidad, por lo que el número de coeficientes f_i $(i = 1, \ldots, m-1)$ no nulos que influyen en los términos $a_1^{(j)}, a_2^{(j)}, \ldots, a_{m-1}^{(j)}$ para un valor de j dado y con $1 \leq j \leq m-1$, será $(h_w(f)-2)$ y este valor nos determina el número de puertas XOR adicionales necesarias para ese valor de j. Por lo tanto, el número de puertas XOR proporcionadas por los términos producto $a_i^{(j)} b_j$, $1 \leq i, j \leq m-1$, será $(m-1)(h_w(f)-2)$, que junto con las $m(m-1)$ puertas XOR necesarias para realizar las sumas de estos términos producto (ecuación 7.6, con $0 \leq n \leq m-1$) hacen un total de $(m-1)(h_w(f)-2)+m(m-1)$ puertas XOR. Por lo tanto, la complejidad *espacial* en número de puertas AND y XOR necesarias para el multiplicador será

$$\#AND = m^2 \tag{7.10}$$

$$\#XOR = (m-1)(h_w(f)-2)+m(m-1) \tag{7.11}$$

donde la ecuación 7.11 toma su valor mínimo cuando el polinomio generador del campo es un trinomio irreducible primitivo, en cuyo caso $h_w(f) = 3$ y el número de puertas XOR es m^2-1. También se observa que la expresión anterior de complejidad en número de puertas XOR se ha obtenido permitiendo la *compartición* de subexpresiones calculadas previamente en forma de sumas exclusivas de coeficientes de uno de los operandos de entrada.

Con respecto de la complejidad *temporal* del multiplicador, ésta dependerá del polinomio irreducible primitivo seleccionado. Consideramos a continuación el peor de los casos, en el que se utilizara un polinomio con todos sus coeficientes no nulos. La implementación de los términos $a_0^{(j)}$ $(1 \leq j \leq m-1)$ requiere que las señales de entrada atraviesen $j-1$ niveles de puertas XOR, mientras que los términos $a_i^{(j)}$ $(1 \leq i, j \leq m-1)$ requieren que las entradas atra-

viesen j niveles de puertas XOR. Los términos $a_0^{(j)}b_j$ y $a_i^{(j)}b_j$ $(1 \leq i, j \leq m-1)$ introducen un retardo adicional de puerta AND debido al producto, por lo que sus retardos serán $T_{AND} + (j-1)T_{XOR}$ y $T_{AND} + jT_{XOR}$, respectivamente. Estos resultados indican que la salida (coeficiente) p_0 del multiplicador proporciona el resultado un retardo de puerta XOR antes que el resto de salidas p_n $(1 \leq n \leq m-1)$, por lo que la complejidad temporal total del multiplicador vendrá determinado por el retardo de éstas. Debido a que el número de niveles de puertas XOR que debe atravesar una señal de entrada en cada término $a_i^{(j)}b_j$ $(1 \leq i, j \leq m-1)$ se incrementa con el valor de j (en una cantidad dada por j), una buena opción de implementación de los coeficientes p_n dados en 7.6 consiste en la construcción de un *árbol lineal* de puertas XOR, en lugar de un *árbol binario* (esta opción dependerá del polinomio seleccionado). De esta forma, el retardo de los p_n $(0 \leq n \leq m-1)$ sería el retardo de los términos $a_i^{(m-1)}b_{m-1}$ $(0 \leq i \leq m-1)$ al que habría que añadir el retardo de un puerta XOR, es decir, $T_{AND} + (m-1)T_{XOR}$ para p_0 y $T_{AND} + mT_{XOR}$ para los $p_1, p_2, \ldots, p_{m-1}$. Por lo tanto, el retardo total del multiplicador (en el peor caso) tendrá el límite superior

$$T \leq T_{AND} + mT_{XOR} \tag{7.12}$$

que dependerá en gran medida del polinomio irreducible primitivo generador del campo empleado.

La multiplicación en base polinómica y, por tanto, las expresiones anteriores 7.5 a 7.9, se calculan realizando en primer lugar la *multiplicación de polinomios* y posteriormente una *reducción módulo* el polinomio irreducible primitivo seleccionado. Mastrovito [Mas89][Mas91] propuso un esquema para la realización de un multiplicador paralelo eficiente en el que se introduce una *matriz producto* $\mathbf{Z}$ que combina los dos pasos anteriores de *multiplicación* y *reducción modular*. A continuación vemos este multiplicador de Mastrovito, ya que es uno de los más ampliamente utilizados y su arquitectura es una de las que presenta un menor número de puertas.

7.2.1.2. Multiplicador de Mastrovito

En primer lugar se puede dar una notación matricial para la multiplicación $A(x)B(x) = P(x) \bmod f(x)$ en el campo de Galois $GF(2^m)$, donde el polinomio $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1x + 1$, $f_i \in GF(2)$. Todos los elementos son polinomios binarios de grado menor que m:

$$p_{m-1}x^{m-1} + \cdots + p_0 = (a_{m-1}x^{m-1} + \cdots + a_0)(b_{m-1}x^{m-1} + \cdots + b_0) \ \ mod \ \ f(x)$$

Los elementos $B(x)$ y $C(x)$ también se pueden representar como vectores columna que contengan los coeficientes del polinomio. Utilizando la matriz

$\mathbf{Z} = h(A(x), f(x))$ se puede describir la multiplicación como

$$P = \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{m-1} \end{pmatrix} = \mathbf{Z}B = \begin{pmatrix} h_{0,0} & \cdots & h_{0,m-1} \\ \vdots & \ddots & \vdots \\ h_{m-1,0} & \cdots & h_{m-2,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix} \quad (7.13)$$

La matriz $\mathbf{Z}$ se llama *matriz producto*. Sus coeficientes $h_{i,j} \in GF(2)$ dependen recursivamente de los coeficientes a_i y de los coeficientes $f_{i,j}$ de la matriz $\mathbf{F}$ (que se introduce posteriormente en la ecuación 7.16) de la siguiente forma

$$h_{i,j} = \begin{cases} a_i & ; j = 0; \;\; i = 0, ..., m-1 \\ u(i-j)a_{i-j} + \sum_{t=0}^{j-1} f_{j-1-t,i} a_{m-1-t} & ; j, i = 0, 1, ..., m-1; j \neq 0 \end{cases} \quad (7.14)$$

donde la función *escalón* u se define como

$$u(\mu) = \begin{cases} 1 & \mu \geq 0 \\ 0 & \mu < 0 \end{cases} \quad (7.15)$$

El producto dado en la ecuación 7.13 describe completamente la multiplicación en el campo finito. La matriz $\mathbf{F}$ necesaria para la construcción de $\mathbf{Z}$ es una función del polinomio irreducible primitivo binario $f(x)$ de grado m que genera $GF(2^m)$. Sus entradas binarias $f_{i,j}$ se definen de forma que

$$\begin{pmatrix} x^m \\ x^{m+1} \\ \vdots \\ x^{2m-2} \end{pmatrix} \equiv \begin{pmatrix} f_{0,0} & \cdots & f_{0,m-1} \\ f_{1,0} & \cdots & f_{1,m-1} \\ \vdots & \ddots & \vdots \\ f_{m-2,0} & \cdots & f_{m-2,m-1} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} mod\, f(x) \quad (7.16)$$

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

La complejidad de implementación del producto matriz-vector dado por la ecuación 7.13 depende únicamente del polinomio $f(x)$. En [Mas89] se dan polinomios irreducibles primitivos generadores de los campos $GF(2^m)$ para valores de $m = 2, 3, \ldots, 16$, que son óptimos con respecto al número de puertas necesarias para realizar la multiplicación en el campo. El número de puertas necesarias para la implementación de este multiplicador se puede deducir teniendo en cuenta que las entradas binarias $f_{i,j}$ de la matriz $\mathbf{F}$ dadas en 7.16 se pueden calcular recursivamente una vez que la primera fila de la matriz se ha formado con los coeficientes de $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1 x + 1$, es decir, $f_{0,j} = f_j$, según la expresión [Paa94]

$$f_{i,j} = \begin{cases} f_{i-1,m-1} & ; i = 1, \ldots, m-2 \;\; ; j = 0 \\ f_{i-1,j-1} + f_{i-1,m-1} f_{0,j} & ; i = 1, \ldots, m-2 \;\; ; j = 1, \ldots, m-1 \end{cases} \quad (7.17)$$

Como el producto matriz-vector dado en la ecuación 7.13 requiere m^2 multiplicaciones módulo 2 , entonces la complejidad *espacial* en número de puertas AND y XOR viene dada por

$$\begin{aligned} \#AND &= m^2 && (7.18) \\ \#XOR &\geq m^2 - 1 && (7.19) \end{aligned}$$

donde la expresión 7.19 se convierte en igualdad cuando el generador del campo es un trinomio irreducible primitivo de la forma $f(x) = x^m + x + 1$ [Paa94].

La complejidad *temporal* se puede especificar teniendo en cuenta que cada camino a través del multiplicador contiene únicamente un multiplicador módulo 2 [Paa94], por lo que el retardo total tiene el límite superior

$$T \leq T_{AND} + 2T_{XOR}\lceil log_2 m \rceil \tag{7.20}$$

7.2.2. Multiplicación en base normal

A partir de los conceptos vistos en la sección 7.1, se puede comprobar que siempre existe una base normal en el campo finito $GF(2^m)$ para todos los enteros positivos m. Es decir, se puede encontrar un elemento del campo α (*elemento normal*) de forma que $N = \{\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{m-1}}\}$ es una base del campo finito $GF(2^m)$. Por lo tanto, cualquier elemento del campo $\beta \in GF(2^m)$ se puede expresar de forma única como

$$\beta = a_0\alpha + a_1\alpha^2 + a_2\alpha^4 + \cdots + a_{m-1}\alpha^{2^{m-1}} \tag{7.21}$$

donde los coeficientes $a_0, a_1, a_2, \ldots, a_{m-1}$ son dígitos binarios y donde la suma se realiza en $GF(2^m)$.

Se pueden enunciar [WTS+85] las siguientes propiedades importantes de un campo finito $GF(2^m)$:

1. La potencia cuadrada en $GF(2^m)$ es una operación lineal. Es decir, dados cualesquiera dos elementos α y β en $GF(2^m)$, se tiene que

$$(\alpha + \beta)^2 = \alpha^2 + \beta^2 \tag{7.22}$$

2. Para cualquier elemento α perteneciente a $GF(2^m)$, se cumple que

$$\alpha^{2^m} = \alpha \tag{7.23}$$

3. Si α es una raíz de un polinomio irreducible $f(x)$ de grado m sobre $GF(2)$, entonces las potencias $\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{m-1}}$ están en $GF(2^m)$ y constituyen un conjunto completo de raíces de $f(x)$.

Con respecto de la propiedad 3, en general es difícil verificar la independencia lineal del conjunto de raíces $\{\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}\}$. Una forma directa de hacerlo es representando los α^{2^i}, $i = 0, 1, \ldots, m-1$, como vectores m-dimensionales en la base canónica $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ y comprobando si la matriz $m \times m$ compuesta por estos m vectores es *no singular* [Wan86]. Para valores grandes de m este método requiere una gran cantidad de cálculos, sin embargo, existen ciertos casos en los que la determinación de la independencia lineal es sencilla. Uno de estos casos establece que si $m = 2^n$, entonces una condición necesaria y suficiente de que el conjunto $\{\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}\}$ sea una base normal de $GF(2^m)$ (y, por tanto, que los α^{2^i}, $i = 0, 1, \ldots, m-1$, sean linealmente independientes) es que la *traza* de α sea la unidad ($\mathrm{Tr}(\alpha) = 1$) o, de forma equivalente, que el coeficiente f_{m-1} del polinomio $f(x) = x^m + f_{m-1}x^{m-1} + \cdots + f_1 x + f_0$ sea la unidad [Men93].

Si esta independencia lineal se cumple en la propiedad 3 anterior, es decir, si un polinomio $f(x)$ de grado m sobre $GF(2)$ es irreducible y si sus raíces son linealmente independientes sobre $GF(2)$, entonces se dice que $f(x)$ es un *N-polinomio* [Men93]. Por lo tanto, un *N-polinomio* constituye otra forma de describir una base normal.

Otro problema importante consiste en determinar si un elemento del campo constituye un *elemento normal*, que se puede realizar de la siguiente forma [Men93]. Si $\alpha \in GF(2^m)$ y $\alpha_i = \alpha^{2^i}$, $i = 0, 1, \ldots, m-1$, entonces α es un elemento normal y genera, por tanto, una base normal $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ de $GF(2^m)$ sobre $GF(2)$ si y sólo si la matriz

$$\begin{pmatrix} \mathrm{Tr}(\alpha_0\alpha_0) & \mathrm{Tr}(\alpha_0\alpha_1) & \cdots & \mathrm{Tr}(\alpha_0\alpha_{m-1}) \\ \mathrm{Tr}(\alpha_1\alpha_0) & \mathrm{Tr}(\alpha_1\alpha_1) & \cdots & \mathrm{Tr}(\alpha_1\alpha_{m-1}) \\ \cdots & \cdots & \ddots & \cdots \\ \mathrm{Tr}(\alpha_{m-1}\alpha_0) & \mathrm{Tr}(\alpha_{m-1}\alpha_1) & \cdots & \mathrm{Tr}(\alpha_{m-1}\alpha_{m-1}) \end{pmatrix} \tag{7.24}$$

es *no singular*.

Ahora supóngase que $\{\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{m-1}}\}$ es una base normal de $GF(2^m)$. Si β se expresa como se vio en la ecuación 7.21, entonces utilizando las propiedades anteriores se tiene que el cuadrado de β es

$$\begin{aligned} \beta^2 &= a_0\alpha^2 + a_1\alpha^4 + a_2\alpha^8 + \cdots + a_{m-2}\alpha^{2^{m-1}} + a_{m-1}\alpha^{2^m} = \\ & a_{m-1}\alpha + a_0\alpha^2 + a_1\alpha^4 + \cdots + a_{m-2}\alpha^{2^{m-1}} \end{aligned} \tag{7.25}$$

Por lo tanto, si β se representa como un vector de componentes de los elementos de la base normal de $GF(2^m)$ en la forma $\beta = (a_0, a_1, a_2, \ldots, a_{m-1})$, entonces $\beta^2 = (a_{m-1}, a_0, a_1, \ldots, a_{m-2})$, es decir, en la representación en base normal, β^2 es un *desplazamiento cíclico* de β. También se puede comprobar que $1 = \alpha + \alpha^2 + \alpha^4 + \cdots + \alpha^{2^{m-1}}$ para cualquier elemento α de $GF(2^m)$, por lo que la representación en una base normal del elemento unidad viene dada como (1,1,1,...,1).

Sean $\beta = (a_0, a_1, \ldots, a_{m-1})$ y $\gamma = (b_0, b_1, \ldots, b_{m-1})$ dos elementos de $GF(2^m)$ en una representación en base normal. Entonces, el producto δ se puede representar como

$$\delta = \beta \cdot \gamma = (p_0, p_1, \ldots, p_{m-1}) \tag{7.26}$$

y el último término p_{m-1} de este producto será alguna función binaria de los componentes de β y de γ, es decir,

$$p_{m-1} = h(a_0, a_1, \ldots, a_{m-1}; b_0, b_1, \ldots, b_{m-1}) \tag{7.27}$$

Como la potencia cuadrada significa el desplazamiento cíclico de un elemento en la representación en base normal, se tiene

$$\begin{aligned} \delta^2 &= \beta^2\gamma^2 = (a_{m-1}, a_0, a_1, \ldots, a_{m-2}) \cdot (b_{m-1}, b_0, b_1, \ldots, b_{m-2}) \\ &= (p_{m-1}, p_0, p_1, \ldots, p_{m-2}) \end{aligned} \tag{7.28}$$

Por consiguiente, el último componente p_{m-2} de δ^2 se obtiene de la función h (mencionada anteriormente en la ecuación 7.27) operando en los componentes de β^2 y γ^2. Es decir,

$$p_{m-2} = h(a_{m-1}, a_0, a_1, \ldots, a_{m-2}; b_{m-1}, b_0, b_1, \ldots, b_{m-2}) \tag{7.29}$$

Elevando al cuadrado δ de manera repetida, resulta evidente que

$$\begin{aligned} p_{m-1} &= h(a_0, a_1, \ldots, a_{m-1}; b_0, b_1, \ldots, b_{m-1}) \\ p_{m-2} &= h(a_{m-1}, a_0, a_1, \ldots, a_{m-2}; b_{m-1}, b_0, b_1, \ldots, b_{m-2}) \\ \ldots &\quad \ldots \\ p_0 &= h(a_1, a_2, \ldots, a_{m-1}, a_0; b_1, b_2, \ldots, b_{m-1}, b_0) \end{aligned} \tag{7.30}$$

Estas ecuaciones definen la multiplicación en la base normal. Como se puede observar en las ecuaciones 7.30, en la representación en base normal la multiplicación tiene la propiedad de que la misma función lógica h utilizada para calcular el último componente p_{m-1} del producto δ se puede utilizar para encontrar secuencialmente los componentes restantes del producto $p_{m-2}, p_{m-3}, \ldots, p_0$. Esta característica de la operación producto requiere únicamente una función lógica h de los $2m$ componentes de β y γ para el cálculo secuencial de los m componentes del producto. A continuación se muestra un ejemplo del cálculo de la función h para un polinomio $f(x)$ dado [Paa94].

Ejemplo 4 *Se considera el campo $GF(2^4)$ con el polinomio $f(x) = x^4 + x^3 + 1$. La base normal en este caso es $\{\alpha, \alpha^2, \alpha^4, \alpha^8\}$, con $f(\alpha) = \alpha^4 + \alpha^3 + 1 = 0$. La multiplicación de dos elementos del campo $\delta = \beta \cdot \gamma$ en la base normal sería:*

$$\begin{aligned} \delta &= p_0\alpha + p_1\alpha^2 + p_2\alpha^4 + p_3\alpha^8 \\ &= \beta \cdot \gamma = (a_0\alpha + a_1\alpha^2 + a_2\alpha^4 + a_3\alpha^8)(b_0\alpha + b_1\alpha^2 + b_2\alpha^4 + b_3\alpha^8) \\ &= \alpha^{12}(a_2b_3 + a_3b_2) + \alpha^{10}(a_1b_3 + a_3b_1) + \alpha^9(a_3b_0 + a_0b_3) \\ &\quad + \alpha^8(a_2b_2) + \alpha^6(a_2b_1 + a_1b_2) + \alpha^5(a_2b_0 + a_0b_2) \\ &\quad + \alpha^4(a_1b_1) + \alpha^3(a_0b_1 + a_1b_0) + \alpha^2(a_0b_0) + \alpha(a_3b_3) \end{aligned}$$

Se observa que en la multiplicación se han creado los elementos $\alpha^{12}, \alpha^{10}, \alpha^9, \alpha^6, \alpha^5$ *y* α^3*, que tendrán que expresarse en términos de la base normal. Para ello se utilizan las propiedades vistas anteriormente, obteniéndose:*

$$\begin{aligned}
\alpha^{12} &= \alpha^8 + \alpha^4 + \alpha^2 \\
\alpha^{10} &= \alpha^8 + \alpha^2 \\
\alpha^9 &= \alpha^8 + \alpha^4 + \alpha \\
\alpha^6 &= \alpha^4 + \alpha^2 + \alpha \\
\alpha^5 &= \alpha^4 + \alpha \\
\alpha^3 &= \alpha^8 + \alpha^2 + \alpha
\end{aligned}$$

Por lo tanto, el coeficiente p_3 *vendrá dado, operando, por la siguiente expresión*

$$\begin{aligned}
p_3 &= h(a_0, a_1, a_2, a_3; b_0, b_1, b_2, b_3) \\
&= a_2b_3 + a_3b_2 + a_1b_3 + a_3b_1 + a_3b_0 + a_0b_3 + a_2b_2 + a_0b_1 + a_1b_0
\end{aligned}$$

La suma de productos de esta ecuación es la función h *que se utilizará para el cálculo del resto de los coeficientes del producto* δ*.* □

La primera descripción del diseño de un circuito para la multiplicación de dos elementos de un campo finito representados en una base normal fue realizada por Massey y Omura [MO84][OM86]. Por este motivo, los multiplicadores en base *normal* a menudo se refieren como *multiplicadores de Massey-Omura.*

De la descripción de la multiplicación dada anteriormente es obvio que la complejidad de la función h determina la complejidad total del multiplicador, que debe ser lo menor posible. Mullin, Onyszchuck, Vanstone y Wilson [MOVW89] comprobaron que existen determinadas bases normales para las cuales se puede conseguir una baja complejidad en la implementación del circuito multiplicador, y que se comentan a continuación.

7.2.2.1. Bases normales óptimas

En general, la multiplicación de dos elementos $A, B \in GF(2^m)$ representados en una base genérica $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ por medio de sus coeficientes $A = (a_0, a_1, \ldots, a_{m-1})$ y $B = (b_0, b_1, \ldots, b_{m-1})$, con $a_i, b_i \in GF(2)$, produce otro elemento del campo $P = (p_0, p_1, \ldots, p_{m-1})$ cuyos coeficientes $p_i s \in GF(2)$ se deben expresar en función de los coeficientes a_is y b_is de la forma más sencilla posible. El producto de los elementos de la base se puede expresar de la siguiente forma [Men93]

$$\alpha_i \alpha_j = \sum_{k=0}^{m-1} t_{ij}^{(k)} \alpha_k, \quad t_{ij}^{(k)} \in GF(2) \tag{7.31}$$

donde el array $m \times m \times m$ de todos los $t_{ij}^{(k)}$ se conoce como *tensor de multiplicación* de la base $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$. Entonces, se puede comprobar que

$$p_k = \sum_{i,j} a_i b_j t_{ij}^{(k)} = A\mathbf{T_k}B^t, \quad 0 \leq k \leq m-1 \tag{7.32}$$

donde $\mathbf{T_k} = (t_{ij}^{(k)})$ es una matriz $m \times m$ sobre $GF(2)$ y donde B^t es la traspuesta de B. El conjunto de matrices $\{\mathbf{T_k}\}$ son independientes de los elementos A y B y se conocen como la *tabla de multiplicación* de $GF(2^m)$ sobre $GF(2)$.

Una implementación inmediata de la multiplicación sobre $GF(2^m)$ consistiría en la construcción de m circuitos correspondientes a las $\mathbf{T_k}$s de forma que cada uno de estos circuitos proporcionara un coeficiente del producto P en función de los coeficientes de A y B. Si m es grande, esta solución sería imposible de realizar. Sin embargo, existen determinadas bases para las que sus tablas de multiplicación $\{\mathbf{T_k}\}$ son más simples que las de otras bases por tener un menor número de entradas distintas de cero o por presentar un mayor número de regularidades.

Una base normal $N = \{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ de $GF(2^m)$ sobre $GF(2)$ donde $\alpha_i = \alpha^{2^i}$, verifica que $\alpha_i^{2^k} = \alpha_{i+k}$ para cualquier entero k, donde los índices de α se reducen módulo m. Se observa que N es la base normal considerada anteriormente $N = \{\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{m-1}}\}$, donde el cuadrado de un elemento del campo es el *desplazamiento cíclico* de su vector de coordenadas y donde cada término del producto P de dos elementos del campo A y B se calcula desplazando sucesivamente los vectores de coordenadas de A y B. Utilizando la notación matricial dada en las ecuaciones 7.31 y 7.32, se observa que la función h utilizada para el cálculo del componente p_{m-1} del producto se puede expresar como

$$p_{m-1} = h(a_0, \ldots, a_{m-1}; b_0, \ldots, b_{m-1}) = \sum_{i,j} a_i b_j t_{ij}^{(m-1)} = A\mathbf{T_{m-1}}B^t \tag{7.33}$$

por lo que el número de puertas necesario para la implementación del cálculo de p_{m-1} será igual al número de entradas distintas de cero de la matriz $\mathbf{T_{m-1}}$. La complejidad del multiplicador viene determinada por la complejidad de la función h, por lo que para la obtención de una buena implementación se debe seleccionar una base normal cuyo número de entradas distintas de cero en $\mathbf{T_{m-1}}$ sea lo menor posible.

Sea el producto [Men93]

$$\alpha\alpha_i = \sum_{j=0}^{m-1} t_{ij}\alpha_j, \quad 0 \leq i \leq m-1, \;\; t_{ij} \in GF(2) \tag{7.34}$$

y sea la matriz $m \times m$ (t_{ij}) representada como $\mathbf{T}$. Se puede comprobar que

$$t_{ij}^{(k)} = t_{i-j,k-j}, \quad \forall\, i, j, k \tag{7.35}$$

donde los subíndices se reducen módulo m. Por lo tanto, el número de entradas distintas de cero en $\mathbf{T_{m-1}}$ es igual al número de entradas distintas de cero en $\mathbf{T}$, que se conoce como la *complejidad* de la base normal N [MOVW89] y se representa por C_N. Como las matrices $\{\mathbf{T_k}\}$ estan determinadas de forma única por $\mathbf{T}$, a $\mathbf{T}$ se le conoce como la *matriz de multiplicación* de la base normal N. Asimismo, como la función h y, por tanto, la matriz $\mathbf{T}$, están determinadas por el polinomio de campo seleccionado $f(x)$, se tiene que la complejidad del multiplicador en base normal depende únicamente de dicho polinomio $f(x)$ para un campo dado $GF(2^m)$.

Mullin demostró en [MOVW89] que la complejidad C_N está limitada inferiormente por $C_N \geq 2m-1$. Además, las bases normales con $C_N = 2m-1$ se dice que son *bases normales óptimas*. En [Paa94] se muestra que el número total de puertas de la realización paralela de un multiplicador en base normal está limitado inferiormente por

$$\#AND = mC_N \geq 2m^2 - m \tag{7.36}$$

$$\#XOR = (m-1)C_N \geq 2m^2 - 3m + 1 \tag{7.37}$$

El límite inferior dado en la ecuación 7.36 para el número de puertas AND de este multiplicador paralelo es aproximadamente el doble que las complejidades 7.10 y 7.18 obtenidas para el primer multiplicador en base canónica visto en la sección 7.2.1 y para el multiplicador de Mastrovito, respectivamente, mientras que el límite inferior dado en la ecuación 7.37 para el número de puertas XOR de este multiplicador coincide con el peor valor obtenido para la complejidad 7.11 del primer multiplicador visto (utilizando un polinomio con todos sus coeficientes no nulos) y es aproximadamente el doble que la complejidad 7.19 del multiplicador de Mastrovito. A pesar de estos valores, las arquitecturas basadas en la utilización de bases normales siguen siendo interesantes para los esquemas criptográficos basados en el problema del logaritmo discreto [Men93], en los que la operación básica a realizar es la exponenciación en campos de gran cardinalidad. La mayoría de los algoritmos de exponenciación utilizan cuadrados y multiplicaciones, por lo que la base normal puede ser muy útil si se encuentra un equilibrio entre la eficiencia del cuadrado (desplazamiento cíclico) y la complejidad de la multiplicación.

7.2.3. Multiplicación en base dual

La multiplicación en base *dual* opera sobre las bases *polinómica* y *dual*, y el primer multiplicador de este tipo fue propuesto por Berlekamp en [Ber82]. Conceptos fundamentales para la operación de multiplicación en la base dual son la función *traza* y el concepto de *dualidad* vistos en las definiciones 21 y 22, respectivamente. La multiplicación en base dual se puede establecer en los siguientes teoremas [HTDR88][FBT96].

Teorema 18 *Sea $\{\mu_j\}$ la base de $GF(p^m)$ y sea $\{\lambda_k\}$ su base dual. Entonces, un elemento del campo z se puede expresar en la base dual $\{\lambda_k\}$ por la expansión*

$$z = \sum_{k=0}^{m-1} z_k \lambda_k = \sum_{k=0}^{m-1} Tr(z\mu_k)\lambda_k \tag{7.38}$$

donde $z_k = Tr(z\mu_k)$ es el k-ésimo coeficiente del elemento z en la base dual.

Teorema 19 *Sea $\{\mu_j\}$ una base de $GF(p^m)$ y sea $\{\lambda_k\}$ su base dual. El producto $w = z \cdot g$ de dos elementos de $GF(p^m)$ se puede expresar en la base dual por medio de la expansión*

$$w = \sum_{k=0}^{m-1} Tr(w\mu_k)\lambda_k = \sum_{k=0}^{m-1} Tr(zg\mu_k)\lambda_k \tag{7.39}$$

donde $Tr(w\mu_k)$ es el k-ésimo coeficiente en la base dual del producto de los dos elementos del campo y donde

$$z = \sum_{k=0}^{m-1} z_k \lambda_k, \qquad g = \sum_{k=0}^{m-1} g_k \mu_k \tag{7.40}$$

En este teorema se observa que el elemento g está representado en la base $\{\mu_j\}$, mientras que el elemento z y el producto de ambos w se encuentran representados en la base dual $\{\lambda_k\}$.

Ya se ha mencionado la importancia de la función traza en la multiplicación dual. La traza es una función lineal de $GF(p^m)$ en $GF(p)$, sin embargo, la idea de la función traza se puede ampliar de forma que se considere cualquier función lineal general. Utilizando una función lineal y otra definición de dualidad, se puede dar la siguiente aproximación a la multiplicación en base dual [FBT96].

Sea $\mathcal{L}_{p^m}$ el conjunto de todas las funciones lineales $l : GF(p^m) \to GF(p)$. Se puede comprobar que la traza es un ejemplo de función lineal definida de esta forma, sin embargo, existen otras funciones lineales disponibles cuya utilización frecuentemente es más conveniente que el empleo de la función traza.

Teorema 20 *Sea $\{\lambda_i\}$ una base de $GF(p^m)$ y sea $z \in GF(p^m)$ dado por*

$$z = \sum_{i=0}^{m-1} z_i \lambda_i \tag{7.41}$$

con $z_i \in GF(p)$. Entonces, existen p^m funciones lineales $l \in \mathcal{L}_{p^m}$ y, además, estas funciones son de la forma

$$l(z) = \sum_{i=0}^{m-1} x_i z_i, \quad \forall z \in GF(p^m) \tag{7.42}$$

donde $x_i \in GF(p)$ y donde la suma se realiza módulo p.

A menudo conviene que la base $\{\lambda_i\}$ sea la base polinómica $\{1, \alpha, \ldots, \alpha^{m-1}\}$ donde α es una raíz del polinomio definitorio irreducible de $GF(p^m)$. En este caso, las funciones de la forma

$$l(z) = z_i, \quad (i \in 0, 1, ..., m-1) \tag{7.43}$$

son particularmente útiles porque los valores de $l(z), \forall z \in GF(p^m)$, se pueden obtener directamente de la representación en base polinómica del campo sin la necesidad de cálculos adicionales. Por tanto, se puede dar una definición más general de *dualidad* [FBT96].

Definición 24 *Sean* $\{\lambda_i\}$ *y* $\{\mu_i\}$ *bases de* $GF(p^m)$, $l \in \mathcal{L}_{p^m}$ *y* $\beta \in GF(p^m)$, $\beta \neq 0$. *Entonces, se dice que las bases son duales con respecto a* l *y* β *si*

$$l(\beta\lambda_i\mu_j) = \begin{cases} 1 & si\ i = j \\ 0 & si\ i \neq j. \end{cases} \tag{7.44}$$

En este caso, $\{\lambda_i\}$ *es la base estándar y* $\{\mu_i\}$ *es la base dual.*

Teorema 21 *Sean* $\{\lambda_i\}$ *y* $\{\mu_i\}$ *bases duales de* $GF(p^m)$ *con respecto a* l *y* β. *Entonces, cualquier* $z \in GF(p^m)$ *se puede representar en la base dual como*

$$z = \sum_{i=0}^{m-1} l(z\beta\lambda_i)\mu_i \tag{7.45}$$

Variando l y β en la definición anterior de dualidad (definición 24), se tienen ahora $(p^m - 1)$ bases duales (en lugar de una sola) para cualquier base dada, pudiéndose seleccionar la base dual más conveniente de entre ellas. Esta idea de una base dual conveniente concierne a la complejidad de la conversión de base dual a base polinómica. En [MKW89] se demostró que con cierto campo finito $GF(2^m)$, se puede seleccionar β de forma que la base dual sea simplemente una reordenación de la base polinómica. Esta propiedad se cumple para cualquier función l general, no sólo para la función traza.

En el siguiente teorema [Fen93] se da una expresión generalizada para la multiplicación en base dual.

Teorema 22 *Sean* $A, B, P \in GF(p^m)$ *tales que* $P = A \cdot B$. *Sea* α *una raíz del polinomio irreducible definitorio del campo, sea* $l \in \mathcal{L}_{p^m}$, $\beta \in GF(p^m)$, P *y* A *representados en la base dual y sea la representación de* B *en la base polinómica dada por*

$$B = \sum_{i=0}^{m-1} b_i\alpha^i \tag{7.46}$$

Entonces se cumple la siguiente relación

$$\begin{pmatrix} l(A\beta) & l(A\beta\alpha) & \cdots & l(A\beta\alpha^{m-1}) \\ l(A\beta\alpha) & l(A\beta\alpha^2) & \cdots & l(A\beta\alpha^m) \\ \cdots & \cdots & \ddots & \cdots \\ l(A\beta\alpha^{m-1}) & l(A\beta\alpha^m) & \cdots & l(A\beta\alpha^{2m-2}) \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \cdots \\ b_{m-1} \end{pmatrix} = \begin{pmatrix} l(P\beta) \\ l(P\beta\alpha) \\ \cdots \\ l(P\beta\alpha^{m-1}) \end{pmatrix} \tag{7.47}$$

A partir de esta expresión general para la multiplicación, se puede deducir el multiplicador de Berlekamp restringiendo la operación a $GF(2^m)$ y haciendo que $a_k = l(A\beta\alpha^k)$ con $k = 0, 1, \ldots, 2m-2$, y $p_k = l(P\beta\alpha^k)$ con $k = 0, 1, \ldots, m-1$. La expresión 7.47 se convierte entonces en

$$\begin{pmatrix} a_0 & a_1 & \cdots & a_{m-1} \\ a_1 & a_2 & \cdots & a_m \\ \cdots & \cdots & \ddots & \cdots \\ a_{m-1} & a_m & \cdots & a_{2m-2} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \cdots \\ b_{m-1} \end{pmatrix} = \begin{pmatrix} p_0 \\ p_1 \\ \cdots \\ p_{m-1} \end{pmatrix} \tag{7.48}$$

Sin embargo, tomando l y β como en la definición 24, entonces se tiene que p_k y a_k (para $k = 0, 1, \ldots, m-1$) son los coeficientes en la base dual de P y A, respectivamente. Por lo tanto, si se pueden generar los valores de a_k para $k = m, m+1, \ldots, 2m-2$, entonces la ecuación 7.48 representa un algoritmo de multiplicación en base dual. Estos valores son calculados en [FBT96] por medio de la expresión

$$a_{m+k} = \sum_{j=0}^{m-1} f_j a_{j+k} \tag{7.49}$$

donde a_k ($k = 0, 1, \ldots, m-1$) son los coeficientes en la base dual de A y donde f_j son los coeficientes del polinomio irreducible que define el campo.

La implementación paralela del multiplicador en base dual se puede realizar, por tanto, utilizando las ecuaciones 7.48 y 7.49. La ecuación 7.48 requiere m módulos que realizan el producto interno de vectores de m bits sobre $GF(2)$. La estructura de estos módulos es independiente del polinomio irreducible $f(x)$ y depende únicamente del valor m, por lo que cada módulo estará formado por m puertas AND y por $(m-1)$ puertas XOR. Por otra parte, la ecuación 7.49 se puede implementar con un módulo que genere los a_k para $k = m, m+1, \ldots, 2m-2$, y que dependerá tanto de m como del polinomio $f(x)$ seleccionado. Si $h_w(f)$ es el *peso de Hamming* de $f(x)$, entonces cada término a_{m+k} en 7.49 se puede construir con $(h_w(f)-2)$ puertas XOR.

En total, se obtiene que la complejidad en número de puertas AND y XOR del multiplicador paralelo en base dual vendrá dada por

$$\#AND = m^2 \tag{7.50}$$
$$\#XOR = (m-1)(h_w(f)-2+m) \tag{7.51}$$

Con respecto de la complejidad *temporal* del multiplicador dual, se observa que el retardo a través de los módulos de producto interno dados por la ecuación 7.48 es $T_{AND} + T_{XOR} \cdot \lceil log_2 m \rceil$, mientras que el retardo a través del módulo generador de los coeficientes a_k dado por 7.49 viene dado por $T \cdot T_{XOR} \cdot \lceil log_2(h_w(f)-1) \rceil$ donde T se calcula con la expresión [FBT96]

$$T = \begin{cases} 1 & k = 1 \\ 1 + \lfloor \frac{m-2}{m-k} \rfloor & k \geq 2 \end{cases} \tag{7.52}$$

Por lo tanto, el retardo total del multiplicador en base dual será

$$T_{AND} + T_{XOR} \cdot (T \cdot \lceil log_2(h_w(f)-1) \rceil + \lceil log_2 m \rceil) \tag{7.53}$$

De estas expresiones, se puede comprobar que la complejidad espacial es mínima cuando el polinomio $f(x)$ es un *trinomio* y que la complejidad temporal es mínima cuando $f(x)$ es un *trinomio* de la forma $f(x) = x^m + x + 1$. También se puede demostrar [MKW89] que cuando el polinomio irreducible definitorio del campo $GF(2^m)$ es un *trinomio* de la forma $f(x) = x^m + x^k + 1$ $(m > k)$ o un *pentanomio* de la forma $f(x) = x^m + x^{k+2} + x^{k+1} + x^k + 1$ $(m > k+2)$, entonces se pueden encontrar bases duales convenientes. Además, en [Fen93] se comprobó que estos resultados se cumplen en el caso general en que l sea cualquier función lineal.

7.2.3.1. Trinomios irreducibles $f(x) = x^m + x^k + 1$

Cuando el polinomio irreducible definitorio de $GF(2^m)$ es un *trinomio* de la forma $f(x) = x^m + x^k + 1$, seleccionando β en la definición 24 tal que

$$l(\beta\alpha^i) = \begin{cases} 1 & i = k-1 \\ 0 & i = 0, 1, ..., m-1 \quad (i \neq k-1) \end{cases} \tag{7.54}$$

entonces se puede demostrar que la base dual de la base polinómica es

$$\{\alpha^{k-1}, \alpha^{k-2}, \ldots, \alpha, 1, \alpha^{m-1}, \alpha^{m-2}, \ldots, \alpha^k\}. \tag{7.55}$$

Por lo tanto, en este caso se observa que la base dual es simplemente una permutación de la base polinómica.

7.2.3.2. Pentanomios irreducibles $f(x) = x^m + x^{k+2} + x^{k+1} + x^k + 1$

Cuando el polinomio irreducible definitorio del campo finito $GF(2^m)$ es un *pentanomio* de la forma $f(x) = x^m + x^{k+2} + x^{k+1} + x^k + 1$ (con $m > k + 2$), seleccionando β en la definición 24 tal que

$$l(\beta\alpha^i) = \begin{cases} 1 & i = 0, k \\ 0 & i = 0, 1, ..., m-1 \quad (i \neq 0, k) \end{cases} \tag{7.56}$$

entonces se puede demostrar que la base dual de la base polinómica es

$$\{\alpha^k, \alpha^{k-1}, \alpha^{k-2}, \ldots, \alpha, 1 + \alpha^k, \alpha^{k+1} + \alpha^{m-1}, \\ \alpha^{m-2}, \alpha^{m-3}, \ldots, \alpha^{k+2}, \alpha^{k+1}\} \tag{7.57}$$

Por lo tanto, en este caso la base dual se puede obtener de la base polinómica con dos sumas y una reordenación de los coeficientes de la base.

7.3. Otros multiplicadores

La mayor parte de los multiplicadores paralelos propuestos en la literatura pertenece a alguno de los tres tipos vistos anteriormente, sin embargo, existen otras aproximaciones recientes de arquitecturas paralelas que están comenzando a ser utilizadas y de entre las cuales se pueden citar las siguientes.

Un método propuesto por Afanasyev [Afa90] permite la aplicación del algoritmo de *Karatsuba-Ofman* [KO63] a la multiplicación de elementos del campo finito $GF(2^m)$, estando dichos elementos representados en la base canónica. El método de Afanasyev optimiza la multiplicación de polinomios, que es el primer (y más importante) paso a realizar en la multiplicación en base polinómica sobre campos de Galois. El algoritmo de Karatsuba-Ofman se basa en el principio de *divide y vencerás*, de forma que permite realizar la multiplicación de polinomios con un número reducido de multiplicaciones a costa, a menudo, de incluir sumas adicionales. Por lo tanto, la multiplicación debe ser más *cara* que la suma para que la aplicación del algoritmo constituya una mejora. La aplicación directa del algoritmo de Karatsuba-Ofman requiere $log_2 m$ iteraciones para polinomios de grado m. En el campo $GF(2)$, la multiplicación y la suma se pueden considerar aproximadamente igual de costosas, por lo que el algoritmo de Karatsuba-Ofman no se puede aplicar directamente a la multiplicación de elementos de $GF(2^m)$ ya que estos son polinomios con coeficientes pertenecientes a $GF(2)$. El método de Afanasyev aplica únicamente $\delta < log_2 m$ iteraciones del algoritmo de Karatsuba-Ofman a los elementos del campo, reduciendo consecuentemente la complejidad [Afa90]. Para la reducción *módulo* el polinomio de campo, esta arquitectura utiliza los polinomios indicados por Mastrovito en [Mas91]. Como consecuencia, este método reduce de manera considerable la complejidad del multiplicador en base canónica sobre $GF(2^m)$.

Otra arquitectura propuesta por Afanasyev en [Afa91] se basa en extensiones de campo de grado 2. El método se fundamenta en la descomposición del campo $GF(2^m)$ en subcampos con múltiples extensiones de grado 2, de manera que si $m = n2^\gamma$, entonces el campo $GF(2^m)$ se descompone en γ subcampos de la forma $GF(2^m) \cong GF((\cdots(((2^n)^2)^2)\cdots)^2)$. La complejidad espacial de esta arquitectura es muy baja, aunque requiere módulos aritméticos pertenecientes a γ subcampos *diferentes* (incrementándose, por tanto, el número de módulos distintos a utilizar) y pierde la modularidad inherente a los métodos que utilizan el algoritmo de Karatsuba-Ofman.

Entre otras arquitecturas propuestas basadas en la utilización de subcampos de campos de Galois, se pueden citar el multiplicador paralelo en base normal de Pincin [Pin89] cuya arquitectura es adecuada para la descomposición en múltiples subcampos (cadena descendente de campos) y el algoritmo de Hsu et al. [HTRG88] que utiliza un subcampo $GF(2^{m/2})$ para realizar una *tabla de búsqueda* (*table lookup*) en el campo $GF(2^m)$ y que emplea esta tabla para todas las operaciones del subcampo.

Paar [Paa94][Paa96] ha estudiado ampliamente las arquitecturas sobre campos compuestos en general del tipo $GF((2^n)^k)$, donde $m = n \cdot k$, obteniendo mejoras en la complejidad sobre el método propuesto en [Afa90] a través del uso de campos con cierta composición $GF((2^n)^2)$ e investigando clases especiales de polinomios de campo [PFR97]. El multiplicador paralelo sobre campos compuestos propuesto por Paar realiza la multiplicación de los elementos del campo (representados en base canónica) utilizando el algoritmo de Karatsuba-Ofman para realizar la multiplicación de los polinomios y aplicando el multiplicador de Mastrovito [Mas89] a la multiplicación de los coeficientes del polinomio.

También es importante mencionar las multiplicaciones sobre *curvas elípticas* [BP01] e *hiperelípticas* [Wol01], que son unas clases especiales de *curvas algebraicas* y que están comenzando a ser muy utilizadas en aplicaciones criptográficas. La multiplicación sobre curvas elípticas definidas sobre un campo finito se basa en la realización de la multiplicación por medio de la suma de puntos pertenecientes a una curva elíptica y cuyo resultado es también un punto perteneciente a dicha curva. Entre las numerosas ventajas que presentan los criptosistemas basados en curvas elípticas se tiene que permiten longitudes de clave más cortas (en comparación con otros métodos convencionales de encriptación de *clave pública* como RSA y sistemas basados en el problema del *logaritmo discreto*) sin que se vea afectada la seguridad del sistema.

7.4. Implementaciones hardware

Las diferentes aproximaciones de implementación de circuitos integrados digitales se pueden clasificar en aproximaciones *Custom* y *Semicustom*. A su vez, la aproximación *Semicustom* se puede dividir en *Cell-Based* (comprendiendo

las *Standard Cells*, *Compiled Cells* y *Macro Cells*) y *Array-Based* (*Pre-diffused* y *Pre-wired*) [Rab96]. Las implementaciones hardware (en circuitos integrados semiconductores) de la aritmética en campos de Galois se realizan normalmente sobre circuitos integrados VLSI utilizando aproximaciones como *Standard Cells* o *Gate Arrays* (un tipo de implementación *Array-Based Pre-diffused*). Sin embargo, recientemente se está comenzando a utilizar *Hardware reconfigurable* como plataforma para la implementación de la aritmética sobre campos de Galois, existiendo pocos estudios con respecto a su utilización y, especialmente, con respecto a la comparación de implementaciones realizadas utilizando las distintas bases de representación y distintos polinomios generadores [Kli95][PR97][EP99][OP99][EP00][OP00][EYCP00][Ima02].

Con respecto a las *arquitecturas de implementación*, se pueden clasificar en arquitecturas de tipo *serie* y *paralelo*. La aritmética paralela [Paa94][PFSR99] se implementa utilizando únicamente lógica combinacional, mientras que la aritmética serie utiliza lógica secuencial (lógica combinacional y registros). Las métricas del rendimiento utilizadas habitualmente para la comparación de estas arquitecturas aritméticas son las complejidades de *espacio* (necesidades de área) y de *tiempo* (retardo del circuito). Normalmente se observa un equilibrio *espacio-tiempo* entre ambos tipos de aritmética, en el sentido de que las arquitecturas paralelas tienden a ser más rápidas pero ocupan un mayor área, mientras que las arquitecturas serie requieren menos área pero tienden a ser más lentas que sus homólogas paralelas con la misma funcionalidad.

En las aplicaciones comerciales, por *hardware reconfigurable* se refiere principalmente a FPGAs (*Field Programmable Gate Arrays*) y CPLDs (*Complex Programmable Logic Devices*) [Jen94], que se pueden incluir en la clasificación anterior dentro de la aproximación *Semicustom Array-Based Pre-wired*. Nosotros nos centraremos en las FPGAs de *Xilinx* [Xil92], que constan de una matriz de celdas lógicas programables llamadas *bloques lógicos configurables* o CLBs (*configurable logic blocks*). Un CLB consta de una sección *combinacional* con cinco entradas y dos salidas que se puede programar para que realice cualquier función de cinco variables o dos funciones de cuatro variables y una sección de *almacenamiento* que consta de dos *flip-flops*. Los CLBs están rodeados por recursos de interconexión, que se pueden clasificar en *interconexiones directas* entre CLBs, *líneas largas* y *canales de conexión* conectados por *matrices de conmutación*. A su vez existen bloques de entrada-salida programables o IOBs (*input-output blocks*) situados en el perímetro del dispositivo, que proporcionan interconexiones externas al chip. La ventaja principal del hardware reconfigurable como las FPGAs es que la funcionalidad del dispositivo no está determinada y se puede programar muchas veces, por lo que muchos circuitos se pueden implementar utilizando el mismo chip. Además, es el propio usuario quien realiza la programación del dispositivo en su lugar de trabajo en lugar de en la fábrica de semiconductores (*semiconductor foundry*).

7.4.1. Metodología de diseño

En la actualidad existen dos metodologías básicas de diseño hardware disponibles, que son el diseño basado en la utilización de un *lenguaje* de programación (*diseño de alto nivel*) y el diseño basado en la utilización de *esquemáticos* (*diseño de bajo nivel*) [Rab96]. El diseño basado en *lenguaje* requiere la utilización de herramientas de *síntesis* para la implementación del hardware deseado. En este contexto, la *síntesis* se puede definir como la transformación entre dos *visiones* diferentes del diseño y normalmente representa una traducción de una especificación del *comportamiento* de una entidad del diseño a una descripción *estructural*. Las herramientas de síntesis raramente consiguen las implementaciones más optimizadas en términos de *área* y *velocidad* cuando se comparan con las implementaciones obtenidas de la utilización de esquemáticos (captura de la conectividad del circuito), pero las metodologías de diseño basadas en la utilización de esquemáticos no pueden soportar el enorme incremento de complejidad de las arquitecturas modernas. Como resultado, se deben utilizar metodologías de diseño de alto nivel basadas en *lenguajes* para la implementación de grandes circuitos. Además, el tiempo de diseño necesario es mucho menor utilizando estas metodologías basadas en lenguajes que usando esquemáticos.

De entre los diferentes *lenguajes de descripción hardware* disponibles en la actualidad, el VHDL (VHSIC[2] *Hardware Description Language*) [Coe89] [VHD00] es uno de los más utilizados por las herramientas comerciales, siendo un lenguaje especialmente diseñado para la descripción de diseños hardware. El lenguaje VHDL dispone de diferentes *modos* de descripción [LSU89], como son el *estructural* [Ima93][IS93] y el modo de *comportamiento*. En el modo *estructural*, VHDL describe un diseño como una conexión de módulos funcionales (*netlist*), mientras que en el modo de *comportamiento* del lenguaje la funcionalidad se describe como un conjunto de relaciones entrada-salida independientemente del tipo de implementación seleccionada.

7.5. Resultados experimentales

Ya se ha mencionado que existen pocos estudios sobre la utilización de hardware reconfigurable para implementaciones aritméticas sobre campos de Galois, siendo la mayor parte ellos debidos a Paar [PR97][EP99][OP00][EYCP00]. Con respecto de la comparación de distintos tipos de multiplicadores sobre campos de Galois tampoco existen muchos trabajos en la literatura, aunque se pueden citar las comparaciones realizadas por Hsu et al. en [HTDR88] y por Paar en [Paa94][PL95][PR97]. En [HTDR88] y [Paa94][PL95] se realizan implementaciones VLSI de distintos multiplicadores y se compara su consumo

[2] *Very High Speed Integrated Circuit.*

de área (Hsu, Paar) y sus retardos asociados (Paar), mientras que en [PR97] se realizan implementaciones reconfigurables del multiplicador en base canónica sobre $GF(2^8)$ utilizando las aproximaciones de Mastrovito y de la *composición de campos* de la forma $GF(2^k)$, con $k = n \cdot m$ ($m = 4$, $n = 2$), comparándose las complejidades teóricas con las experimentales obtenidas.

Debido a esta ausencia de comparaciones, nosotros hemos realizado la implementación paralela sobre hardware reconfigurable de multiplicadores sobre $GF(2^8)$ en base *polinómica*, *normal* y *dual* para distintos polinomios generadores de dicho campo [Ima02], con el objeto de realizar la comparación de sus complejidades espaciales y temporales. Para ello hemos utilizado el software *Xilinx Foundation F2.1i* y hemos seleccionado como dispositivos de implementación las FPGAs 4013XLPQ160 pertenecientes a la familia XC4000XL. Las descripciones de los multiplicadores paralelos se han realizado en VHDL utilizando modelos de descripción de comportamiento.

Para poder realizar la comparación de las arquitecturas implementadas, se deben seleccionar las métricas del rendimiento a utilizar, que normalmente están relacionadas con el *espacio* y *tiempo* (*velocidad*). En dispositivos reconfigurables como las FPGAs, la métrica para la complejidad espacial es el *número de bloques lógicos* (CLBs) utilizados para la implementación, mientras que la métrica seleccionada para la determinación de la velocidad es el *retardo máximo de camino combinacional* (*Maximum combinational path delay*).

Hemos realizado distintos experimentos para cada multiplicador correspondientes al establecimiento de distintas opciones de *síntesis*. Éstas han sido la optimización para velocidad (*Speed*) y para área (*Area*), independientemente, con niveles de esfuerzo alto (*High*) y bajo (*Low*). Cada experimento se ha realizado, a su vez, para la más alta y más baja condiciones de esfuerzo de *place & route* (*Place & Route Effort Level: Fastest Runtime* y *High Effort*) y para una frecuencia de reloj de 50 nanosegundos (*Target Clock Frecuency*). Además del número de CLBs ocupado por el diseño y el retardo máximo de camino combinacional, se ha medido el *número total de puertas equivalentes del diseño* (*Total equivalent gate count for design*), donde la métrica *número de puertas equivalentes* se puede definir como el mayor número de puertas NAND de dos entradas (o cualquier otra primitiva simple) necesarias para la implementación de cualquier configuración circuital de un dispositivo [OD95] y que es una métrica del rendimiento relativa al espacio también utilizada habitualmente en el diseño microelectrónico.

En las tablas presentadas a continuación [Ima02], *Esfuerzo P&R* representa el esfuerzo de *place & route* que toma los valores *Min* (*Fastest Runtime*) y *Max* (*High Effort*), mientras que *Ret. Max (ns)* representa el retardo máximo de camino combinacional (en nanosegundos). En las tablas también se indican el número de CLBs y de puertas equivalentes (*Número PEs*) ocupados por el diseño implementado.

7.5.1. Multiplicación en base polinómica

Se ha realizado la implementación de la multiplicación en base polinómica sobre el campo $GF(2^8)$ utilizando el algoritmo propuesto por Yeh et al. [YRT84] dado en la subsección 7.2.1 por medio de las ecuaciones 7.9. Los polinomios irreducibles primitivos que hemos utilizado para la generación del campo han sido $f(x) = x^8 + x^5 + x^3 + x^2 + 1$ y $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. También se ha implementado el multiplicador de Mastrovito utilizando estos dos polinomios generadores.

En la tabla 7.1 se muestran los resultados experimentales obtenidos para el multiplicador canónico sobre $GF(2^8)$ utilizando las ecuaciones 7.9 para el polinomio irreducible $f(x) = x^8 + x^5 + x^3 + x^2 + 1$, mientras que en la tabla 7.2 se dan los resultados del multiplicador utilizando las mismas ecuaciones para el polinomio $f(x) = x^8 + x^4 + x^3 + x^2 + 1$.

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	32	32	32	32	36	36	35	35
Número PEs	372	372	372	372	402	402	396	396
Ret. Max (ns)	27.3	23.1	27.0	23.0	25.8	24.0	23.8	21.8

Tabla 7.1: Resultados del multiplicador canónico para $f(x) = x^8 + x^5 + x^3 + x^2 + 1$.

Con respecto de las complejidades espaciales obtenidas en la implementación se observa que el menor número de CLBs dado en la tabla 7.1 es de 32, correspondiente a la optimización por área e independientemente de los esfuerzos escogidos, mientras que el número de puertas equivalentes correspondientes es de 372. Para la complejidad temporal, el menor retardo es de 21.8 nanosegundos correspondientes a la optimización de velocidad con esfuerzo alto y con esfuerzo de place & route máximo.

En la tabla 7.2 se muestran los resultados obtenidos para la implementación utilizando el polinomio generador $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. En este caso, el menor número de CLBs es de 34 (con 394 puertas equivalentes) para una optimización de área y el menor retardo es de 22.5 nanosegundos obtenidos en la optimización para velocidad.

La primera observación que se puede realizar al comparar estos dos multiplicadores es que los resultados obtenidos en ambos casos (menor número de CLBs y menor retardo en optimizaciones de área y velocidad, respectivamente) se corresponden con lo esperado en función del tipo de síntesis seleccionada, observándose que las diferencias existentes entre ambas optimizaciones, tanto en número de CLBs como en retardo, varían de una implementación a otra. Por ejemplo, en la tabla 7.1, la diferencia entre valores mínimos de CLBs obtenidos en la optimización de área y de velocidad es de 3 CLBs, mientras que

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	34	34	34	34	35	35	35	35
Número PEs	394	394	394	394	426	426	426	426
Ret. Max (ns)	29.6	25.2	27.6	26.7	24.2	22.5	22.8	25.1

Tabla 7.2: Resultados del multiplicador canónico para $f(x) = x^8 + x^4 + x^3 + x^2 + 1$.

la diferencia en retardo mínimo obtenido en ambos casos es de 1.2 ns. En la tabla 7.2, estas diferencias son de 1 CLB y de 2.7 ns., respectivamente.

Otra observación corresponde al número de puertas equivalentes obtenidas en las implementaciones. Por ejemplo, en la tabla 7.1 se obtienen 396 puertas equivalentes para 35 CLBs (optimización de velocidad con esfuerzo alto), mientras que en la tabla 7.2 se obtienen 426 puertas equivalentes para el mismo número de CLBs (optimización de velocidad). Por lo tanto, la relación entre el número de CLBs y el número de puertas equivalentes no es siempre constante, con lo que el *número de puertas equivalentes* no parece ser una buena métrica espacial del rendimiento cuando se utilizan dispositivos reconfigurables (como las FPGAs) para la implementación de un diseño.

La observación más importante que se puede realizar al comparar las tablas 7.1 y 7.2, consiste en que los datos obtenidos corresponden a las implementaciones de dos multiplicadores sobre $GF(2^8)$ para dos pentanomios irreducibles primitivos distintos. Este hecho justificaría las diferencias en los datos obtenidas, sin embargo, al utilizar dos polinomios primitivos con el mismo *peso de Hamming* ($h_w = 5$), la ecuación 7.11 nos indica que la complejidad *teórica* en número de puertas XOR de ambos multiplicadores es la misma ($\#XOR = 77$), al igual que el número de puertas AND ($\#AND = 64$). Por lo tanto, se puede deducir que la complejidad *teórica* en el número de puertas no parece predecir de forma exacta el consumo de CLBs [PR97]. Lo mismo se puede decir para la complejidad *temporal*, ya que la complejidad teórica obtenida para ambos multiplicadores es la misma y viene dada por $T_{AND} + 7T_{XOR}$, mientras que los retardos obtenidos en la implementación difieren en todos los casos.

Observaciones similares se pueden extraer de las implementaciones sobre FPGAs del multiplicador de Mastrovito para los dos polinomios seleccionados. En las tablas 7.3 y 7.4 se muestran los resultados obtenidos para los polinomios $f(x) = x^8 + x^5 + x^3 + x^2 + 1$ y $f(x) = x^8 + x^4 + x^3 + x^2 + 1$, respectivamente.

Con respecto de las complejidades obtenidas para los multiplicadores Mastrovito, se tiene que el menor número de CLBs obtenido en las tablas 7.3 y 7.4 es de 33 en ambos casos (optimización por área con esfuerzo alto), mientras que los menores retardos obtenidos son de 21.7 ns. (tabla 7.3 con optimización de velocidad con esfuerzos máximos) y de 23.7 ns. (tabla 7.4 para optimiza-

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	34	34	33	33	37	37	41	41
Número PEs	372	372	373	373	511	511	465	465
Ret. Max (ns)	25.8	22.9	25.1	23.8	23.1	23.2	23.4	21.7

Tabla 7.3: Resultados del multiplicador Mastrovito para $f(x) = x^8+x^5+x^3+x^2+1$.

ción de velocidad). Sin embargo, esta igualdad en número de CLBs contrasta con la diferente complejidad espacial teórica (64 AND y 84 XOR [PR97] para $f(x) = x^8+x^5+x^3+x^2+1$ y 64 AND y 82 XOR para $f(x) = x^8+x^4+x^3+x^2+1$), al igual que la diferencia de retardos en implementación contrasta con la igualdad temporal teórica ($T_{AND} + 5T_{XOR}$ para ambos polinomios).

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	34	34	33	33	39	39	40	40
Número PEs	405	405	394	394	558	558	510	510
Ret. Max (ns)	25.5	23.7	26.6	27.3	26.0	24.7	23.7	25.6

Tabla 7.4: Resultados del multiplicador Mastrovito para $f(x) = x^8+x^4+x^3+x^2+1$.

A continuación comparamos los dos tipos de multiplicadores implementados y los resultados obtenidos, extrayendo una serie de conclusiones.

7.5.1.1. Comparación de los multiplicadores

Ya se ha mencionado anteriormente que el multiplicador de Mastrovito es uno de los más eficientes en cuanto al número de puertas necesarias para su implementación paralela. Sin embargo, la implementación de multiplicadores canónicos utilizando las ecuaciones 7.9 por medio de la *compartición de subexpresiones* hace que se obtengan complejidades teóricas eficientes en cuanto al número de puertas XOR utilizadas.

En la tabla 7.5 se comparan las complejidades espaciales teóricas de los multiplicadores de Mastrovito [Paa94] con las de los multiplicadores construidos utilizando el algoritmo de Yeh et al. dado por las ecuaciones 7.9 (calculadas en las ecuaciones 7.10 y 7.11), para algunos polinomios irreducibles primitivos con grados $m = 2, 3, \ldots, 16$. En esta tabla, las entradas numéricas ubicadas en la columna encabezada por $f(x)$ representan los coeficientes del polinomio irreducible que son distintos de cero. Por ejemplo, la entrada $(8, 5, 3, 2, 0)$ representa el polinomio irreducible $f(x) = x^8 + x^5 + x^3 + x^2 + x^0$.

		Mastrovito		Yeh et al.	
m	f(x)	#AND	#XOR	#AND	#XOR
2	2,1,0	4	3	4	3
3	3,1,0	9	8	9	8
4	4,1,0	16	15	16	15
5	5,2,0	25	24	25	24
6	6,1,0	36	35	36	35
7	7,1,0	49	48	49	48
8	8,5,3,2,0	64	84	64	77
9	9,4,0	81	80	81	80
10	10,3,0	100	99	100	99
11	11,2,0	121	120	121	120
12	12,8,5,1,0	144	207	144	165
13	13,7,6,1,0	169	202	169	192
14	14,9,7,2,0	196	282	196	221
15	15,1,0	225	224	225	224
16	16,11,6,5,0	256	281	256	285
Total		1495	1712	1495	1596

Tabla 7.5: Comparación de complejidades espaciales teóricas de los multiplicadores de Mastrovito y Yeh para distintos polinomios irreducibles primitivos.

De los resultados totales obtenidos, se observa que el número de puertas XOR teóricas necesarias para la construcción de multiplicadores canónicos paralelos utilizando el algoritmo dado por las ecuaciones 7.9 es un 6.8 % menor que las necesitadas utilizando el esquema de Mastrovito, mientras que el número de puertas AND es el mismo en ambos casos. Ejemplos significativos son los multiplicadores para los polinomios $(14, 9, 7, 2, 0)$ y $(12, 8, 5, 1, 0)$ para los que el número de puertas XOR usando el algoritmo de Yeh es un 21.6 % y un 20.3 % menores, respectivamente, que utilizando el algoritmo de Mastrovito. Por lo tanto, la aproximación paralela de construcción que hemos realizado utilizando el algoritmo de Yeh et al. (ecuaciones 7.9) con la *compartición de subexpresiones* parece ser una buena alternativa para la reducción de área en determinados casos que dependerán del polinomio seleccionado.

Con respecto de la complejidad teórica temporal de ambos multiplicadores, se tiene que las ecuaciones 7.12 y 7.20 proporcionan los límites superiores de sus retardos, a partir de los cuales se puede deducir un mayor retardo del multiplicador de Yeh que el de Mastrovito, aunque esta complejidad dependerá del polinomio seleccionado. Por ejemplo, de la implementación realizada de ambos multiplicadores para el polinomio irreducible $f(x) = x^8 + x^5 + x^3 + x^2 + 1$ se observa que el multiplicador de Yeh presenta una complejidad temporal dada por $T_{AND} + 7T_{XOR}$ mientras que la de Mastrovito es $T_{AND} + 5T_{XOR}$, por lo que en este caso se cumplen las previsiones teóricas de retardos.

Sin embargo, como se ha mencionado anteriormente, estas complejidades teóricas no siempre se corresponden con las complejidades obtenidas en las implementaciones sobre hardware reconfigurable como las FPGAs. Por ejemplo, para el polinomio $f(x) = x^8 + x^5 + x^3 + x^2 + 1$ se tiene que el multiplicador de Yeh presenta una complejidad espacial teórica de 64 AND y 77 XOR y un retardo teórico de $T_{AND} + 7T_{XOR}$, mientras que el multiplicador de Mastrovito tiene un número de puertas teórico de 64 AND y 84 XOR [PR97] y un retardo teórico de $T_{AND} + 5T_{XOR}$. En este caso, las previsiones espaciales teóricas se cumplen en la implementación, ya que el número de CLBs respectivos obtenidos es de 32 y 33. En cambio, el menor retardo teórico del multiplicador de Mastrovito no se refleja en los resultados experimentales, ya que los retardos obtenidos son prácticamente idénticos (21.8 y 21.7 nanosegundos).

En el caso de los multiplicadores implementados utilizando el polinomio $f(x) = x^8 + x^4 + x^3 + x^2 + 1$, no se cumple ninguno de los resultados teóricos. Las complejidades teóricas espaciales y temporales del multiplicador de Yeh son las mismas que las del polinomio anterior (64 AND, 77 XOR y $T_{AND} + 7T_{XOR}$), mientras que las del multiplicador de Mastrovito son en este caso de 64 AND y 82 XOR y de $T_{AND} + 5T_{XOR}$, respectivamente. Los resultados obtenidos en la implementación son de 34 CLBs y 22.5 ns. de retardo para el multiplicador de Yeh y 33 CLBs y un retardo de 23.7 ns. para el multiplicador de Mastrovito. Por lo tanto, experimentalmente no se cumplen las previsiones teóricas de un menor área del multiplicador de Yeh y de un menor retardo del multiplicador de Mastrovito.

7.5.2. Multiplicación en base normal

Utilizando los conceptos vistos en la subsección 7.2.2, se ha realizado la implementación de la multiplicación en base normal sobre el campo $GF(2^8)$ usando los polinomios irreducibles (pentanomios) $f(x) = x^8 + x^7 + x^6 + x + 1$ y $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. Para ello, se debe determinar en primer lugar el elemento normal generador de la base normal a utilizar. Posteriormente, se podrá calcular la expresión del coeficiente p_{m-1} (función h) dado por la ecuación 7.33 por medio del cálculo de la matriz $\mathbf{T_{m-1}}$ a través de la ecuación 7.32. Las salidas p_i, para $0 \leq i \leq m-2$, se calcularán a partir de la función h por medio del desplazamiento cíclico de los operandos de entrada.

7.5.2.1. Pentanomio $f(x) = x^8 + x^7 + x^6 + x + 1$

Este polinomio constituye un *N-polinomio*, ya que siendo α una raíz de $f(x)$ y expresando cada α^{2^i}, $0 \leq i \leq m-1$, en la base polinómica de la forma $\alpha^{2^i} = \sum_{j=0}^{m-1} b_{ij}\alpha^j$, $b_{ij} \in GF(2)$, se tiene que la matriz $m \times m$ de la forma (b_{ij}) es *no singular*, por lo que los elementos $\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}$ son linealmente independientes. También se podría haber concluido que $f(x)$ es un *N-polinomio*

teniendo en cuenta que el grado $m = 8$ del polinomio es una potencia de 2 y que el coeficiente $f_{m-1} = f_7$ es la unidad. Por lo tanto, la raíz α constituye un *elemento normal* y junto con sus conjugados forma una *base normal*.

Utilizando las ecuaciones 7.31 a 7.35 se calcula la matriz de multiplicación $\mathbf{T}$ de la base normal $\{\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}\}$ y la matriz $\mathbf{T_7}$, que proporciona la siguiente expresión de la función h

$$\begin{aligned} p_7 &= h(a_0, a_1, \ldots, a_7; b_0, b_1, \ldots, b_7) \\ &= a_0b_1 + a_0b_4 + a_0b_5 + a_0b_6 + a_1b_0 + a_1b_2 + a_1b_3 + a_1b_7 \\ &\quad + a_2b_1 + a_2b_3 + a_2b_4 + a_2b_5 + a_3b_1 + a_3b_2 + a_4b_0 + a_4b_2 \\ &\quad + a_4b_5 + a_4b_7 + a_5b_0 + a_5b_2 + a_5b_4 + a_5b_7 + a_6b_0 + a_6b_6 \\ &\quad + a_7b_1 + a_7b_4 + a_7b_5 \end{aligned} \tag{7.58}$$

y que será la expresión a utilizar para el cálculo del resto de los coeficientes de la multiplicación sin más que realizar el desplazamiento cíclico de los operandos.

Para determinar la complejidad *espacial* teórica de la multiplicación en base normal, utilizamos el término C_N que es en este caso $C_N = 27$ (el número de 1's presentes en la matriz de multiplicación $\mathbf{T}$). Como se vio en la subsección 7.2.2, una base normal con una complejidad $C_N = 2m - 1$ se dice que es una *base normal óptima*, que en este caso de $m = 8$ sería $C_N = 15$. Sin embargo, no existen bases normales óptimas para $m = 8$ [Men93], por lo que las complejidades que obtengamos siempre serán superiores a 15. La complejidad *temporal* teórica se puede deducir de la expresión 7.58, siendo en este caso $T_{AND} + 5T_{XOR}$.

Los resultados experimentales obtenidos de la implementación reconfigurable de este multiplicador en base normal para el *elemento normal* α se muestran en la tabla 7.6.

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	49	49	48	48	49	49	49	49
Número PEs	525	525	526	526	696	696	696	696
Ret. Max (ns)	26.8	24.8	29.5	25.8	27.0	27.0	27.0	27.0

Tabla 7.6: Resultados del multiplicador normal con elemento normal α.

En este caso se ha seleccionado la raíz α de $f(x) = x^8 + x^7 + x^6 + x + 1$ como *elemento normal* generador de la base normal, sin embargo, existen varios elementos normales [Men93] pertenecientes al campo $GF(2^m)$ que pueden generar bases normales de diferentes complejidades C_N. Por ejemplo, se puede observar que el elemento del campo $\tilde{\alpha} = 1 + \alpha$ verifica las condiciones de *normalidad* vistas y genera una base normal $\{\tilde{\alpha}, \tilde{\alpha}^2, \ldots, \tilde{\alpha}^{2^{m-1}}\}$. La matriz de multiplicación $\mathbf{T}$ presenta en este caso una complejidad espacial teórica $C_N = 21$ y a

partir de $\mathbf{T}$ se obtiene la matriz $\mathbf{T_7}$ que nos permite calcular la función h, que vendrá dada por la siguiente expresión

$$\begin{aligned} p_7 &= h(a_0, a_1, \ldots, a_7; b_0, b_1, \ldots, b_7) \\ &= a_0b_5 + a_0b_6 + a_1b_3 + a_1b_5 + a_2b_4 + a_2b_5 + a_2b_6 \\ &\quad + a_2b_7 + a_3b_1 + a_3b_4 + a_4b_2 + a_4b_3 + a_5b_0 + a_5b_1 \\ &\quad + a_5b_2 + a_5b_6 + a_6b_0 + a_6b_2 + a_6b_5 + a_6b_6 + a_7b_2 \end{aligned} \tag{7.59}$$

y que es más simple que la obtenida anteriormente para el elemento normal α. A partir de esta expresión 7.59, también se puede deducir la complejidad teórica *temporal*, que viene dada por $T_{AND} + 5T_{XOR}$.

La implementación de la multiplicación para este elemento normal $\tilde{\alpha} = 1+\alpha$ produce los resultados dados en la tabla 7.7.

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	44	44	45	45	45	45	53	53
Número PEs	472	472	486	486	660	660	624	624
Ret. Max (ns)	24.5	23.7	24.3	26.3	25.2	23.2	28.4	26.7

Tabla 7.7: Resultados del multiplicador normal con elemento normal $\tilde{\alpha} = 1 + \alpha$.

Los resultados experimentales coinciden en este caso con las complejidades teóricas obtenidas, ya que el multiplicador implementado con el elemento normal α (complejidad $C_N = 27$) necesita un mínimo de 48 CLBs (tabla 7.6) y tiene un retardo mínimo de 24.8 nanosegundos, mientras que el multiplicador obtenido para el elemento normal $\tilde{\alpha} = 1 + \alpha$ ($C_N = 21$) requiere un mínimo de 44 CLBs con retardo mínimo de 23.2 ns. (tabla 7.7).

7.5.2.2. Pentanomio $f(x) = x^8 + x^4 + x^3 + x^2 + 1$

Si α es una raíz de $f(x)$, se observa que la matriz (b_{ij}) de representación de sus conjugados α^{2^i}, $0 \leq i \leq m-1$, en la base polinómica es *singular*, por lo que α y sus conjugados no son linealmente independientes y no forman una base. Por lo tanto, el polinomio irreducible $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ no constituye un *N-polinomio* y α no es un *elemento normal*. Esta conclusión también se habría obtenido teniendo en cuenta que en este caso de $m = 2^3$, la traza de α es igual a cero o viendo que el coeficiente del polinomio $f_7 = 0$.

A pesar de que α no constituya un elemento normal, se pueden buscar otros elementos del campo $GF(2^8)$ que verifiquen las condiciones de *normalidad*. Por ejemplo, elementos como $1+\alpha$, α^3, $1+\alpha^3$, $\alpha+\alpha^3$ o α^{253} tampoco son elementos normales, sin embargo, el elemento del campo $\tilde{\alpha} = \alpha^5$ cumple las condiciones de normalidad y genera una base normal $\{\tilde{\alpha}, \tilde{\alpha}^2, \ldots, \tilde{\alpha}^{2^{m-1}}\}$, al igual que los elementos $1 + \alpha^5$ y $\alpha + \alpha^5$.

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	50	50	49	49	47	47	52	52
Número PEs	544	544	562	562	616	616	579	579
Ret. Max (ns)	27.5	29.6	26.3	25.7	24.9	23.0	26.3	26.2

Tabla 7.8: Resultados del multiplicador normal con elemento normal $\tilde{\alpha} = \alpha^5$.

El cálculo de las matrices de multiplicación para estos tres elementos normales α^5, $1 + \alpha^5$ y $\alpha + \alpha^5$ proporciona las complejidades C_N teóricas 31, 29 y 27, respectivamente, obteniéndose unas complejidades temporales teóricas respectivas de $T_{AND} + 6T_{XOR}$, $T_{AND} + 6T_{XOR}$ y $T_{AND} + 5T_{XOR}$. En las tablas 7.8, 7.9 y 7.10 se muestran los resultados obtenidos en las implementaciones reconfigurables de los multiplicadores para los elementos normales α^5, $1 + \alpha^5$ y $\alpha + \alpha^5$, respectivamente.

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	49	49	49	49	58	58	56	56
Número PEs	555	555	559	559	624	624	600	600
Ret. Max (ns)	25.4	27.0	25.6	25.8	31.9	27.3	26.0	30.3

Tabla 7.9: Resultados del multiplicador normal con elemento normal $\tilde{\alpha} = 1 + \alpha^5$.

Se puede observar de los resultados que el multiplicador construido utilizando el elemento normal $\tilde{\alpha} = \alpha^5$ necesita al menos 47 CLBs con un retardo mínimo de 23 nanosegundos (tabla 7.8 con optimización para velocidad). Para el elemento normal $\tilde{\alpha} = 1 + \alpha^5$ se obtiene un mínimo de 49 CLBs y un retardo mínimo de 25.4 ns. (tabla 7.9, optimización para área), mientras que para $\tilde{\alpha} = \alpha + \alpha^5$ el multiplicador correspondiente tiene un consumo y un retardo mínimos de 48 CLBs y 24.8 ns., respectivamente (optimización para área en la tabla 7.10).

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	49	49	48	48	49	49	49	49
Número PEs	525	525	526	526	696	696	696	696
Ret. Max (ns)	26.8	24.8	29.5	25.7	27.0	26.9	27.0	26.9

Tabla 7.10: Resultados del multiplicador normal con elemento normal $\tilde{\alpha} = \alpha + \alpha^5$.

De estos resultados, se observa que las complejidades teóricas no coinciden con las complejidades experimentales obtenidas. El multiplicador construido a partir del elemento normal $\tilde{\alpha} = \alpha^5$ es el de mayor C_N (31), sin embargo, es el que presenta el menor consumo de CLBs (47) de los tres multiplicadores implementados para $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. Los retardos teóricos y experimentales tampoco se corresponden, ya que la menor complejidad teórica es $T_{AND} + 5T_{XOR}$ del multiplicador de elemento normal $\tilde{\alpha} = \alpha + \alpha^5$, mientras que experimentalmente el multiplicador con menor retardo es el de elemento normal $\tilde{\alpha} = \alpha^5$, con 23.0 ns.

La comparación de los resultados de este multiplicador de mejor caso para $\tilde{\alpha} = \alpha^5$ ($C_N = 31$, $T_{AND} + 6T_{XOR}$, 47 CLBs, 23.0 ns.) con el multiplicador de mejor caso obtenido para el pentanomio anterior $f(x) = x^8 + x^7 + x^6 + x + 1$ con elemento normal $\tilde{\alpha} = 1 + \alpha$ ($C_N = 21$, $T_{AND} + 5T_{XOR}$, 44 CLBs, 23.2 ns.), se correspondería con lo esperado. Sin embargo, mientras que la complejidad teórica $C_N = 21$ es un 32.3 % menor que la complejidad $C_N = 31$, esta proporción no se corresponde con la obtenida en el número de CLBs (6.4 % menor). Tampoco los retardos se corresponden ya que experimentalmente son prácticamente idénticos, mientras que teóricamente es más rápido el multiplicador con $C_N = 21$. Por lo tanto, en el caso de la multiplicación en base normal tampoco queda clara la correspondencia entre complejidades teóricas y experimentales cuando se utiliza hardware reconfigurable para la implementación.

7.5.3. Multiplicación en base dual

Se ha realizado la implementación reconfigurable paralela de la multiplicación en base dual sobre el campo $GF(2^8)$ utilizando el pentanomio irreducible $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. Ya se vio en la subsección 7.2.3 que para polinomios de la forma $f(x) = x^m + x^{k+2} + x^{k+1} + x^k + 1$ (con $m > k + 2$) se pueden encontrar bases duales convenientes que simplifican el cálculo del producto. En nuestro caso, se tiene que $k = 2$ y de la ecuación 7.57 se obtiene que la base dual de la base polinómica $\{1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7\}$ es de la forma

$$\{\alpha^2, \alpha, 1 + \alpha^2, \alpha^3 + \alpha^7, \alpha^6, \alpha^5, \alpha^4, \alpha^3\} \tag{7.60}$$

con lo que la base dual se obtiene de la polinómica con dos sumas y una reordenación de los coeficientes de la base.

El multiplicador en base dual opera sobre dos operandos representados uno en la base *dual* y el otro en la base *polinómica*, obteniéndose el resultado también representado en la base *dual* a partir de las ecuaciones 7.48 y 7.49. El multiplicador que hemos implementado calcula el producto de los dos operandos de entrada, estando todos ellos representados en la base polinómica. Tenemos que realizar, por tanto, una transformación de base polinómica a base dual de uno de los operandos de entrada y una transformación de base dual

a polinómica del resultado de la multiplicación. Estas transformaciones de base añadirán una complejidad adicional de 4 puertas XOR al multiplicador, para la base dual que utilizamos y que viene dada en la ecuación 7.60.

La complejidad *espacial* teórica de este multiplicador en base dual implementado se puede calcular a partir de las ecuaciones 7.50 y 7.51, añadiendo las 4 puertas XOR necesarias para los cambios de base. De esta forma obtenemos que el número de puertas necesarias es de 64 AND y 81 XOR. Con respecto de la complejidad *temporal* teórica, se puede calcular de las ecuaciones 7.52 y 7.53, obteniéndose ($m = 8$, $k = 2$) un retardo de $T_{AND} + 7T_{XOR}$. Sin embargo, a este tiempo habría que añadirle el retardo introducido por los cambios de base, que sería de 1 retardo de puerta XOR por cada uno de ellos. Por lo tanto, el retardo teórico total del multiplicador dual implementado será $T_{AND} + 9T_{XOR}$.

Optimización	Área				Velocidad			
Esfuerzo	Bajo		Alto		Bajo		Alto	
Esfuerzo P&R	Min	Max	Min	Max	Min	Max	Min	Max
Número CLBs	28	28	28	28	30	30	32	32
Número PEs	330	330	330	330	372	372	367	367
Ret. Max (ns)	26.3	31.3	27.9	25.8	27.1	23.6	27.4	25.3

Tabla 7.11: Resultados del multiplicador dual para $f(x) = x^8 + x^4 + x^3 + x^2 + 1$.

En la tabla 7.11 se muestran los resultados experimentales obtenidos en la implementación reconfigurable, donde se observa que el menor número de CLBs obtenidos es de 28 en la optimización para área (independientemente del esfuerzo establecido). El menor retardo es de 23.6 nanosegundos, correspondiente a una optimización para velocidad con esfuerzo bajo y con esfuerzo de place & route máximo.

A continuación realizamos la comparación de los tres tipos de multiplicadores implementados para el polinomio irreducible $f(x) = x^8 + x^4 + x^3 + x^2 + 1$.

7.5.4. Comparación de los multiplicadores

Podemos realizar la comparación de los resultados teóricos y experimentales de los tres tipos de multiplicadores implementados para el polinomio irreducible $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ común a todos ellos. Consideramos todas las opciones de implementación realizadas, es decir, los multiplicadores polinómicos usando el algoritmo de Yeh et al. y el de Mastrovito, los multiplicadores normales para los tres elementos normales considerados y el multiplicador dual.

En la tabla 7.12 se muestra un resumen de las complejidades teóricas y experimentales obtenidas de los multiplicadores implementados utilizando el polinomio irreducible $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ para las tres bases consideradas: *polinómica* (utilizando el algoritmo de Yeh et al. y Mastrovito), *normal*

(para los elementos normales α^5, $1+\alpha^5$ y $\alpha+\alpha^5$) y *dual*. Las complejidades *teóricas* vienen dadas por el número de puertas AND (#AND), número de puertas XOR (#XOR) y por el retardo expresado en retardos de puertas AND y XOR (T_{AND}, T_{XOR}). En el caso de los multiplicadores normales, la complejidad espacial teórica venía representada por el término C_N, que para los elementos normales α^5, $1+\alpha^5$ y $\alpha+\alpha^5$ toma los valores 31, 29 y 27, respectivamente. Sin embargo, se pueden obtener a partir de C_N las complejidades en número de puertas AND y XOR aplicando las ecuaciones 7.36 y 7.37. Con respecto de las complejidades *experimentales*, éstas se representan por el número mínimo de CLBs necesarios para la implementación y por el retardo mínimo (expresado en nanosegundos).

	Polinómica		Normal			Dual
	Yeh	Mastrovito	α^5	$1+\alpha^5$	$\alpha+\alpha^5$	
#AND	64	64	248	232	216	64
#XOR	77	82	217	203	189	81
(T_{AND}, T_{XOR})	(1,7)	(1,5)	(1,6)	(1,6)	(1,5)	(1,9)
Número CLBs	34	33	47	49	48	28
Retardo (ns)	22.5	23.7	23.0	25.4	24.8	23.6

Tabla 7.12: Resumen de complejidades teóricas y experimentales de los tres tipos de multiplicadores implementados para $f(x) = x^8 + x^4 + x^3 + x^2 + 1$.

En la tabla 7.12 se observa que el multiplicador con menor complejidad espacial teórica corresponde al multiplicador en base polinómica construido utilizando el algoritmo de Yeh et al. (64 AND y 77 XOR), mientras que los multiplicadores con menor complejidad temporal teórica serían el de Mastrovito y el de elemento normal $\alpha+\alpha^5$, ambos con retardo $T_{AND}+5T_{XOR}$. También se puede comprobar que los multiplicadores normales son, con diferencia, los de más costosa implementación, llegando casi a cuadruplicar (el multiplicador normal α^5) el número de puertas teóricas del multiplicador canónico de Yeh. Por otra parte, el multiplicador dual es el de mayor retardo teórico debido a los 9 niveles de puertas XOR que deben atravesar las señales.

Estas complejidades teóricas, sin embargo, no se corresponden con los resultados experimentales obtenidos de las implementaciones en FPGAs. Estos datos muestran que el multiplicador dual es el que consume menor área (28 CLBs), mientras que el multiplicador polinómico de Yeh ocupa 34 CLBs. El multiplicador dual podría ocupar incluso un área menor, ya que la implementación realizada incluye dos transformaciones de base que añaden una ligera complejidad adicional. También se observa que los multiplicadores normales apenas ocupan un 75 % más área (49 CLBs para $1+\alpha^5$) que el multiplicador dual, a pesar de que sus complejidades teóricas son muy superiores. Los retardos experimentales tampoco se corresponden con sus equivalentes teóri-

cos, siendo el multiplicador de Yeh el de menor retardo (22.5 nanosegundos). El multiplicador dual, de mayor complejidad temporal teórica, presenta un retardo apenas un 5 % mayor que el de Yeh, siendo incluso más rápido que multiplicadores teóricamente más veloces, como los normales para $1 + \alpha^5$ y $\alpha + \alpha^5$ y prácticamente igual de rápido que el de Mastrovito. Por lo tanto, se vuelve a poner de manifiesto el hecho de que las complejidades teóricas (dadas en número y en retardo de puertas) no parecen predecir el consumo de CLBs, ni los retardos *reales* obtenidos, cuando se utilizan dispositivos reconfigurables como las FPGAs para la implementación.

7.6. Conclusiones

En este capítulo se han presentado los conceptos básicos de los campos finitos (de Galois) y de los campos de extensión, así como las distintas bases de representación de elementos del campo finito y su obtención a partir de polinomios generadores del campo. También se han estudiado los campos de Galois $GF(2^m)$, campos de extensión del campo binario $GF(2)$ que se utilizan actualmente en un gran número de aplicaciones técnicas.

La multiplicación sobre $GF(2^m)$ es considerada como la operación aritmética más importante y una de las más complejas, siendo esta complejidad dependiente de factores como la selección de la base de representación o del polinomio irreducible generador del campo. Las bases más habitualmente utilizadas son la *polinómica*, la *normal* y la *dual* (aunque existen otras como la base *triangular*, que se verá en el siguiente capítulo), que dan lugar a tres tipos de multiplicación diferentes y cuya implementación da lugar, por tanto, a tres tipos de multiplicadores. Para la multiplicación en base *canónica* o *polinómica*, hemos realizado el análisis de complejidad teórico (dado en función del número y retardo de puertas AND y XOR necesarias para la implementación) del multiplicador paralelo construido a partir del algoritmo descrito por Yeh et al.

Las implementaciones hardware de la aritmética sobre campos de Galois se realizan normalmente en circuitos integrados VLSI, aunque recientemente se están comenzando a utilizar plataformas reconfigurables (como FPGAs) para su implementación. No existen en la actualidad muchos estudios sobre la utilización de hardware reconfigurable para implementaciones aritméticas sobre campos de Galois, así como tampoco existen muchos trabajos en la literatura que realicen comparaciones de los distintos tipos de multiplicadores mencionados. Por estos motivos, nosotros hemos realizado la implementación paralela sobre FPGAs (a partir de descripciones VHDL y utilizando *Xilinx Foundation F2.1i*) de multiplicadores sobre el campo finito $GF(2^8)$ en las bases *polinómica*, *normal* y *dual* para distintos polinomios generadores y hemos realizado la comparación de sus complejidades espaciales y temporales, tanto *teóricas* como *experimentales*.

De la comparación de las complejidades teóricas de los multiplicadores, se ha observado la importancia que tiene la selección tanto de la base de representación como del polinomio generador del campo en la complejidad final teórica del multiplicador. Por ejemplo, los multiplicadores que operan en base normal son, con diferencia, los que presentan una complejidad teórica mayor, sin embargo, propiedades como la sencillez de cálculo del cuadrado de un elemento del campo (desplazamiento cíclico) hacen que su utilización sea muy adecuada en determinados casos. Por otra parte, la selección de trinomios irreducibles, por ejemplo, en los multiplicadores sobre base dual, hace que la complejidad se reduzca debido a que la base dual resulta ser simplemente una permutación de la base canónica, comprobándose la influencia que tiene el polinomio irreducible generador sobre la complejidad total del multiplicador.

Otra conclusión importante que se ha obtenido de los experimentos realizados es que las complejidades *teóricas*, tanto *espacial* (en número de puertas AND y XOR) como *temporal* (en retardos de puertas AND y XOR), no predicen de forma exacta el consumo de CLBs ni el retardo real cuando se utilizan FPGAs como plataformas para la implementación. Este hecho lo hemos comprobado en casi todas las comparaciones realizadas entre complejidades teóricas y experimentales. Además, en aquellos casos en los que las *previsiones* teóricas se verifican experimentalmente, se observa que las *proporciones* teóricas de complejidad no se cumplen experimentalmente. Por ejemplo, el multiplicador normal sobre $GF(2^8)$ para $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ (con elemento normal α^5) ocuparía, *teóricamente*, un 221 % más de área (total de puertas AND y XOR) que el multiplicador dual, mientras que, *experimentalmente*, ocupa únicamente un 68 % más de CLBs.

Volviendo a la multiplicación en base canónica, se han estudiado los algoritmos propuestos por Yeh et al. y por Mastrovito. El análisis de complejidad que hemos realizado para el primero de ellos, nos ha llevado a observar que la propiedad de *compartición de subexpresiones* extraída de las expresiones del multiplicador dadas por este algoritmo, produce un consumo teórico menor de puertas XOR que el eficiente multiplicador de Mastrovito (experimentalmente, se han obtenido resultados contrapuestos para dos polinomios con igual peso de Hamming). Únicamente para un determinado tipo de *trinomios* irreducibles se alcanzan las mismas complejidades teóricas de puertas XOR para ambos multiplicadores. Los trinomios irreducibles también se mencionaron en la base dual como un tipo especial de polinomios para el cual se simplificaban los cambios de base dual a canónica, y viceversa.

La extracción de la propiedad de *compartición de subexpresiones* a partir de las expresiones dadas por el algoritmo de multiplicación (descripción del multiplicador), y el hecho de que, por tanto, la forma de describir el multiplicador pueda influir tanto en su complejidad teórica como en la experimental, nos ha llevado a desarrollar un nuevo método de multiplicación en base canónica,

aplicable también en determinados casos a la multiplicación en base normal, en el que se proporciona una nueva forma de descripción del multiplicador. Este método está basado en la *agrupación* y *compartición* de subexpresiones, y conlleva una reducción de complejidad *experimental* y, en determinados casos, también *teórica*, cuando se utiliza hardware reconfigurable para la implementación. El nuevo método de multiplicación sobre campos de Galois se introduce en el siguiente capítulo, y su desarrollo se ha realizado, inicialmente, a partir de la utilización de un tipo especial de polinomios irreducibles conocidos como AOPs. Posteriormente en el capítulo 9, el nuevo método se aplicará a campos de Galois generados por determinados tipos de trinomios irreducibles.

Capítulo 8

Nuevo método de multiplicación sobre $GF(2^m)$

En este capítulo se presenta un nuevo método de multiplicación en base canónica sobre campos de Galois $GF(2^m)$ al que se le ha denominado método transposicional. Esta nueva aproximación conlleva una reducción de complejidad experimental cuando se utiliza hardware reconfigurable para la implementación de los multiplicadores. El nuevo método se introduce considerando campos de Galois generados por un tipo especial de polinomios conocidos como AOPs, en cuyo caso, el método también es aplicable a la multiplicación en base normal.

En el capítulo 7 se observó que de las expresiones dadas por un algoritmo de multiplicación se podían extraer *subexpresiones comunes*, de forma que su *compartición* en las expresiones finales que describen el multiplicador pueden influir tanto en su complejidad teórica como en la experimental. También se vio que para determinados tipos de polinomios irreducibles generadores del campo finito, las complejidades de los módulos multiplicadores que operan sobre dichos campos son más reducidas que si se emplean otros tipos de polinomios para la generación del campo.

Teniendo en cuenta estas consideraciones, en este capítulo se ha desarrollado una nueva aproximación para la multiplicación en base *canónica*, a la que se le ha denominado método *transposicional* [ISF02], basada en la *agrupación* y *compartición* de subexpresiones comunes. Este método se extrae a partir de una nueva formulación para la multiplicación en base polinómica (basada en la utilización de una base *triangular* de representación) que depende del polinomio irreducible generador del campo de Galois seleccionado. Asimismo, el método transposicional también es aplicable, en determinados casos, a la multiplicación en base *normal*.

Se ha comprobado que la descripción del multiplicador proporcionada por el método transposicional iguala las mejores complejidades teóricas encontradas en la literatura y que conlleva una reducción de complejidad *experimental* cuando se utiliza hardware reconfigurable para la implementación. Este nuevo método se introduce considerando campos de Galois generados por un tipo especial de polinomios conocidos como AOPs (*all-one-polynomials*).

8.1. Base triangular

Como se vio en el capítulo 7, una base *canónica* o *polinómica* Ω está formada por un conjunto de elementos $\Omega = \{1, \omega, \omega^2, \ldots, \omega^{m-1}\}$, donde ω es una raíz en $GF(2^m)$ de un polinomio irreducible generador $f(x) = \sum_{i=0}^{m} f_i x^i$ de grado m sobre $GF(2)$. Utilizando esta base, los elementos del campo $GF(2^m)$ son polinomios de grado máximo $m-1$ sobre $GF(2)$ y la aritmética se realiza módulo el polinomio irreducible $f(x)$.

El conjunto de elementos $\Lambda = \{\lambda_0, \lambda_1, \ldots, \lambda_{m-1}\}$ se llama base *triangular* [HB95][Has98] de Ω si $\lambda_i = \sum_{j=0}^{m-1-i} f_{i+j+1}\omega^j$, $0 \leq i \leq m-1$, donde los términos f_i son los coeficientes del polinomio irreducible $f(x)$. Un elemento $\alpha \in GF(2^m)$ se puede representar con respecto a la base canónica como

$$\alpha = \sum_{i=0}^{m-1} a_{\Omega_i}\omega^i = (1, \omega, \ldots, \omega^{m-1}) \cdot \begin{pmatrix} a_{\Omega_0} \\ a_{\Omega_1} \\ \cdots \\ a_{\Omega_{m-1}} \end{pmatrix} \tag{8.1}$$

donde los términos a_{Ω_i} son las coordenadas de α con respecto de Ω. Se puede representar como $\underline{\alpha}_\Omega$ al vector de coordenadas de α con respecto de Ω, es decir, $\underline{\alpha}_\Omega = (a_{\Omega_0}, a_{\Omega_1}, \ldots, a_{\Omega_{m-1}})^t$. El vector de coordenadas de α con respecto de la base triangular Λ se puede calcular en función de $\underline{\alpha}_\Omega$ por medio de la expresión

$$\underline{\alpha}_\Lambda = \mathbf{T} \cdot \underline{\alpha}_\Omega \tag{8.2}$$

donde [Has98]

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & t_1 \\ 0 & 0 & 0 & \cdots & 1 & t_1 & t_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 1 & t_1 & \cdots & t_{m-4} & t_{m-3} & t_{m-2} \\ 1 & t_1 & t_2 & \cdots & t_{m-3} & t_{m-2} & t_{m-1} \end{pmatrix} \tag{8.3}$$

siendo $t_j = \sum_{i=0}^{j-1} f_{m-j+i}t_i$, para $0 < j \leq m-1$ y con $t_0 = 1$. También se tiene que la matriz $m \times m$ definida como $\mathbf{H}(\underline{\alpha}_\Lambda) = (\underline{\alpha}_\Lambda, \underline{\alpha\omega}_\Lambda, \ldots, \underline{\alpha\omega}_\Lambda^{m-1})$ es una matriz de *Hankel*[1] [Woa01] y que sus vectores columna se pueden calcular por

[1] Matriz cuadrada cuyas entradas son constantes a lo largo de sus contradiagonales.

medio de la expresión [Has98]

$$\underline{\alpha\omega}^{i+1}_{\Lambda_j} = \begin{cases} \underline{\alpha\omega}^{i}_{\Lambda_{j+1}} & 0 \leq j \leq m-1 \\ \sum_{l=0}^{m-1} \underline{\alpha\omega}^{i}_{\Lambda_l} f_l & j = m-1 \end{cases} \tag{8.4}$$

Utilizando estos conceptos, se da a continuación un nueva formulación general para la multiplicación en base canónica sobre el campo finito $GF(2^m)$ basada en la utilización de la base triangular de representación y a partir de la cual se deducirá, posteriormente, un nuevo método de multiplicación llamado *transposicional*.

8.2. Algoritmo de multiplicación en base canónica

Sean tres elementos $\alpha, \delta, \chi \in GF(2^m)$ y sean $\underline{\alpha}_\Omega, \underline{\delta}_\Omega, \underline{\chi}_\Omega$ sus vectores de coordenadas, respectivamente, en la base canónica Ω. Entonces, el producto $\delta = \alpha \cdot \chi$ se puede realizar de la siguiente forma:

1. Se representa χ en la base triangular Λ utilizando la ecuación 8.2, obteniéndose $\underline{\chi}_\Lambda$, es decir, $\underline{\chi}_\Lambda = \mathbf{T} \cdot \underline{\chi}_\Omega$.

2. Se construye la matriz de Hankel para $\underline{\chi}_\Lambda$, $\mathbf{H}(\underline{\chi}_\Lambda) = (\underline{\chi}_\Lambda, \underline{\chi\omega}_\Lambda, \ldots, \underline{\chi\omega}^{m-1}_\Lambda)$.

3. Se construye una nueva matriz $m \times m$, $\mathbf{K}(\underline{\chi}_\Lambda)$, definida como

$$\mathbf{K}(\underline{\chi}_\Lambda) = \mathbf{H}(\underline{\chi}_\Lambda) \cdot \mathbf{F} \tag{8.5}$$

donde la también nueva matriz $\mathbf{F}$[2] se define como

$$\mathbf{F} = \begin{pmatrix} f_m & f_{m-1} & f_{m-2} & \cdots & f_2 & f_1 \\ 0 & f_m & f_{m-1} & \cdots & f_3 & f_2 \\ 0 & 0 & f_m & \cdots & f_4 & f_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & f_m & f_{m-1} \\ 0 & 0 & 0 & \cdots & 0 & f_m \end{pmatrix} \tag{8.6}$$

y donde los términos f_i, $1 \leq i \leq m$, son los coeficientes del polinomio irreducible $f(x)$ generador del campo seleccionado.

4. Si $\underline{\delta}^r_\Omega$ representa las coordenadas invertidas de $\underline{\delta}_\Omega$, se tiene entonces que el producto $\delta = \alpha \cdot \chi$ en la base canónica Ω se puede calcular como

$$\underline{\delta}^r_\Omega = (d_{\Omega_{m-1}}, d_{\Omega_{m-2}}, \ldots, d_{\Omega_1}, d_{\Omega_0}) = \underline{\alpha}^t_\Omega \cdot \mathbf{K}(\underline{\chi}_\Lambda) \tag{8.7}$$

[2]No confundir con la matriz $\mathbf{F}$ definida en la ecuación 7.16 para el multiplicador de Mastrovito.

Se puede observar que las expresiones de las coordenadas del producto obtenidas utilizando la ecuación 8.7 vienen dadas en forma de *sumas de productos*, donde dichas *sumas* y *productos* se refieren a operaciones sobre el campo binario $GF(2)$. También se observa que este nuevo algoritmo constituye un método general de multiplicación en base canónica, basado en la utilización de la base triangular de representación, que depende del polinomio irreducible $f(x)$ generador del campo seleccionado.

La diferencia existente entre este nuevo algoritmo de multiplicación y otros métodos de multiplicación canónica existentes en la literatura (como los vistos en la sección 7.2.1) radica en que a partir de la ecuación 8.7 se deduce otro método de multiplicación canónica, denominado *transposicional*. Para ello, se considera que la matriz **K** definida en 8.5 se puede descomponer en la suma de una serie de matrices de forma que a partir de las expresiones de las coordenadas del producto obtenidas se puedan realizar *agrupaciones* de subexpresiones cuya *compartición* conlleva una reducción de complejidad *experimental* y, en determinados casos, también *teórica*, cuando se utiliza hardware reconfigurable para la implementación.

El número y la forma de las matrices obtenidas de la descomposición de la matriz **K** depende del polinomio irreducible seleccionado. Esta idea de la descomposición de una matriz en una suma de matrices ha sido ya utilizada en otras aproximaciones similares [HWB93][KS98] de multiplicación sobre campos finitos, obteniéndose buenos resultados en cuanto a la complejidad final alcanzada para el módulo multiplicador.

Debido a que el método *transposicional* se obtiene a partir del nuevo algoritmo de multiplicación dado por la ecuación 8.7 que depende, a su vez, del polinomio generador del campo seleccionado, a continuación se realiza el estudio y deducción de este método para los AOPs irreducibles y en el capítulo 9 se amplía el estudio a ciertos tipos de *trinomios* irreducibles.

8.3. AOPs irreducibles $f(x) = x^m + x^{m-1} + \cdots + x + 1$

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, por tanto, el campo $GF(2^m)$, si y sólo si $m + 1$ es primo y 2 es primitivo módulo $m + 1$ [Men93], 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. En un AOP también se cumple que para la matriz **T** definida por la ecuación 8.3, el coeficiente $t_1 = 1$, mientras que $t_2 = t_3 = \cdots = t_{m-1} = 0$.

A continuación se vuelve a revisar el nuevo algoritmo de multiplicación en base polinómica dado de forma general en la sección 8.2, particularizado para un AOP irreducible de grado m.

8.3.1. Multiplicación en base canónica

Cuando se utiliza un AOP irreducible como polinomio generador del campo $GF(2^m)$, se observa que la matriz $\mathbf{F}$ definida en 8.6 viene dada como

$$\mathbf{F} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & 1 & \cdots & 1 & 1 \\ 0 & 0 & 1 & \cdots & 1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \tag{8.8}$$

por lo que la matriz $\mathbf{K}$ definida en 8.5 tiene la forma [ISF02]

$$\mathbf{K}(\underline{\chi}_\Lambda) = (\underline{\chi}_\Lambda, \underline{\chi}_\Lambda + \underline{\chi\omega}_\Lambda, \ldots, \sum_{i=0}^{m-1} \underline{\chi\omega^i}_\Lambda) \tag{8.9}$$

Esta matriz $\mathbf{K}$ se puede descomponer en la suma de dos matrices $m \times m$, $\mathbf{K_1}$ y $\mathbf{K_2}$, $\mathbf{K} = \mathbf{K_1} + \mathbf{K_2}$, dadas de la siguiente forma

$$\mathbf{K_1} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ c_{\Omega_{m-1}} & c_{\Omega_{m-1}} & \cdots & c_{\Omega_{m-1}} & c_{\Omega_{m-1}} \\ c_{\Omega_{m-2}} & c_{\Omega_{m-2}} & \cdots & c_{\Omega_{m-2}} & c_{\Omega_{m-2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{\Omega_2} & c_{\Omega_2} & \cdots & c_{\Omega_2} & c_{\Omega_2} \\ c_{\Omega_1} & c_{\Omega_1} & \cdots & c_{\Omega_1} & c_{\Omega_1} \end{pmatrix} \tag{8.10}$$

$$\mathbf{K_2} = \begin{pmatrix} c_{\Omega_{m-1}} & c_{\Omega_{m-2}} & \cdots & c_{\Omega_1} & c_{\Omega_0} \\ c_{\Omega_{m-2}} & c_{\Omega_{m-3}} & \cdots & c_{\Omega_0} & 0 \\ c_{\Omega_{m-3}} & c_{\Omega_{m-4}} & \cdots & 0 & c_{\Omega_{m-1}} \\ c_{\Omega_{m-4}} & c_{\Omega_{m-5}} & \cdots & c_{\Omega_{m-1}} & c_{\Omega_{m-2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{\Omega_0} & 0 & \cdots & c_{\Omega_3} & c_{\Omega_2} \end{pmatrix} \tag{8.11}$$

donde los términos c_{Ω_i} son las coordenadas de χ con respecto de Ω. De la estructura dada para las matrices $\mathbf{K_1}$ y $\mathbf{K_2}$, se observa que $\mathbf{K_1}$ es una matriz con sus m columnas idénticas y que $\mathbf{K_2}$ es una matriz de Hankel.

Si la matriz $\mathbf{K}$ se descompone en la suma de estas dos matrices, se tiene entonces que la ecuación 8.7 del producto $\delta = \alpha \cdot \chi$ en la base canónica Ω generada por un AOP toma la siguiente forma

$$\underline{\delta}^r_\Omega = \underline{\alpha}^t_\Omega \cdot \mathbf{K}(\underline{\chi}_\Lambda) = \underline{\alpha}^t_\Omega \cdot (\mathbf{K_1} + \mathbf{K_2}) = \underline{\alpha}^t_\Omega \cdot \mathbf{K_1} + \underline{\alpha}^t_\Omega \cdot \mathbf{K_2} \tag{8.12}$$

Se puede observar que las expresiones de las coordenadas del producto obtenidas en la ecuación 8.12 vienen dadas en forma de *sumas de productos*. También se puede comprobar que estas expresiones son iguales a las obtenidas utilizando otros algoritmos similares de multiplicación en base canónica [KS98][HK00][ZP01]. La diferencia radica en que a partir de la ecuación 8.12 se puede deducir otra forma (el método *transposicional*) de calcular las coordenadas del producto, de modo que las expresiones de estas coordenadas se pueden *reescribir* en una forma *diferente* basada en la *agrupación* y *compartición* de subexpresiones.

Como se comprobará posteriormente en los resultados experimentales, la descripción del multiplicador utilizando este nuevo método produce complejidades menores cuando se utilizan plataformas reconfigurables para la implementación.

8.3.1.1. Método transposicional de multiplicación

A partir de la estructura de la matriz $\mathbf{K_1}$ dada en 8.10, se puede observar que el producto $\underline{\alpha}^t_\Omega \mathbf{K_1}$ es un vector con todos sus componentes idénticos e iguales a la suma de productos $a_{\Omega_1} c_{\Omega_{m-1}} + a_{\Omega_2} c_{\Omega_{m-2}} + \cdots + a_{\Omega_{m-1}} c_{\Omega_1}$, que es el *producto interno* del vector $\underline{\alpha}^t_\Omega$ y cualquier vector columna de la matriz $\mathbf{K_1}$. Llamamos a este producto interno $\mathbf{P_{m-1}}$, ya que es la suma de $m-1$ términos producto (de hecho, se puede considerar como el producto interno de dos vectores de longitud $m-1$). Por lo tanto, la i-ésima coordenada del producto $\delta = \alpha \cdot \chi$ vendrá dada por

$$d_{\Omega_i} = \mathbf{P_{m-1}} + (\underline{\alpha}^t_\Omega \mathbf{K_2})_{m-1-i} \tag{8.13}$$

con $i = 0, 1, \ldots, m-1$, y donde $(\underline{\alpha}^t_\Omega \mathbf{K_2})_j$ es la j-ésima coordenada de $\underline{\alpha}^t_\Omega \mathbf{K_2}$.

Para poder dar una expresión para el cálculo de $\underline{\alpha}^t_\Omega \mathbf{K_2}$ y de esta forma establecer un nuevo método de multiplicación en base canónica basado en el algoritmo de multiplicación introducido en la sección 8.2 y particularizado para un AOP en la subsección 8.3.1, se utiliza la notación usada en teoría de grupos para las *permutaciones* y que se recuerda en el siguiente ejemplo.

Ejemplo 5 *Se desea calcular la 8^a coordenada de $\underline{\alpha}^t_\Omega \mathbf{K_2}$ sobre el campo finito $GF(2^{10})$. Se puede comprobar que esta octava coordenada viene dada por*

$$(\underline{\alpha}^t_\Omega \mathbf{K_2})_8 = a_0c_1 + a_1c_0 + a_3c_9 + a_4c_8 + a_5c_7 + a_6c_6 + a_7c_5 + a_8c_4 + a_9c_3$$

que es, a su vez, el producto interno de los vectores de coordenadas de α y χ representados en la base canónica Ω por $(a_0, a_1, a_3, a_4, a_5, a_6, a_7, a_8, a_9)$ y $(c_1, c_0, c_9, c_8, c_7, c_6, c_5, c_4, c_3)$, donde se omiten los subíndices Ω por claridad. Los términos producto a_ic_j se pueden representar entonces por la permutación

$$\begin{pmatrix} 0 & 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 0 & 9 & 8 & 7 & 6 & 5 & 4 & 3 \end{pmatrix}$$

donde la fila superior contiene los subíndices de las coordenadas de α *que se multiplicarán por las coordenadas de* χ *cuyos subíndices se encuentran en la fila inferior de la permutación.*

Comenzando por el símbolo 0 *de la fila superior, se observa que la permutación lleva el* 0 *al* 1*, que representa el producto* a_0c_1*. A continuación nos fijamos en el siguiente símbolo de la fila superior, que es el* 1*, y se observa que la permutación lleva el* 1 *al* 0 *(producto* a_1c_0*), con lo que se cierra un ciclo y que lo escribimos como* $(0,1)$*. Este ciclo representará la suma* $(a_0c_1 + a_1c_0)$ *en nuestra notación. A continuación se selecciona algún otro símbolo de la fila superior, por ejemplo el* 6*. La permutación lleva el* 6 *al* 6*, obteniéndose el ciclo* (6) *que representa el producto* a_6c_6*.*

Continuando de esta forma, se encuentran los ciclos $(3,9)$*,* $(4,8)$ *y* $(5,7)$*. Se puede escribir finalmente la permutación del ejemplo por medio de los ciclos* $(0,1)(3,9)(4,8)(5,7)(6)$*. La octava coordenada del producto* $\underline{\alpha}_\Omega^t\mathbf{K_2}$ *para este ejemplo será la suma de los términos* $x_{ij} = (a_ic_j + a_jc_i)$ *y* $x_k = (a_kc_k)$ *representados por los* 2*-ciclos* (i,j) *y por los* 1*-ciclos* (k)*, respectivamente.*

Los 2*-ciclos* (i,j) *se conocen en teoría de grupos como transposiciones.* □

Utilizando la notación dada en el ejemplo 5, se introducen las funciones $\mathbf{EC_i^m}$ (para valores de i pares) y $\mathbf{OC_i^m}$ (para valores impares de i) que proporcionan los ciclos de la permutación correspondiente a los valores de i y m, y que vienen dadas por las siguientes expresiones [ISF02]

$$\mathbf{EC_i^m} = \begin{cases} (k) & k = \frac{m}{2} + \frac{i}{2} \\ (p,r) & \begin{cases} j = 1,2,\ldots,(\frac{m}{2} + \frac{i}{2}) - (i+1) \\ p = (i+j); \quad r = (m-j) \end{cases} \\ (h,l) & h = 0,1,\ldots,\frac{i}{2} - 1; \quad l = (i-h-1); \quad h \geq 0 \end{cases} \tag{8.14}$$

$$\mathbf{OC_i^m} = \begin{cases} (k) & k = \frac{i-1}{2} \\ (p,r) & \begin{cases} j = 1,2,\ldots,(\frac{m}{2} + \frac{i+1}{2}) - (i+1) \\ p = (i+j); \quad r = (m-j) \end{cases} \\ (h,l) & h = 0,1,\ldots,\frac{i-1}{2} - 1; \quad l = (i-h-1); \quad h \geq 0 \end{cases} \tag{8.15}$$

Definiendo las funciones $\mathbf{E_i}$ y $\mathbf{O_i}$ como la suma de los términos x_{ij} y x_k representados por los ciclos dados por las funciones $\mathbf{EC_i^m}$ (para valores pares de i) y $\mathbf{OC_i^m}$ (para valores de i impares), respectivamente, se observa que el producto interno $\mathbf{P_{m-1}}$ definido anteriormente es $\mathbf{P_{m-1}} = \mathbf{E_0}$.

A partir de la estructura de la matriz $\mathbf{K_2}$ dada en 8.11, se observa que la primera columna de $\mathbf{K_2}$ coincide con el vector de coordenadas invertidas $\underline{\chi}_\Omega^r$. Por lo tanto, uno de los componentes del vector producto $\underline{\alpha}_\Omega^t\mathbf{K_2}$ será el producto interno de los vectores $\underline{\alpha}_\Omega^t$ y $\underline{\chi}_\Omega^r$, que vendrá dado por la suma de productos $\underline{\alpha}_\Omega^t \cdot \underline{\chi}_\Omega^r = a_{\Omega_0}c_{\Omega_{m-1}} + a_{\Omega_1}c_{\Omega_{m-2}} + \cdots + a_{\Omega_{m-1}}c_{\Omega_0}$. Llamamos a este

producto interno $\mathbf{P_m}$ que, a su vez, también se puede representar por medio de los ciclos de una permutación. Para ello, se define una nueva función $\mathbf{MC^m}$ que proporciona los ciclos de la permutación correspondiente al valor de m de la siguiente forma [ISF02]

$$\mathbf{MC^m} = (j, n) \quad \begin{cases} j = 0, 1, 2, \ldots, (\frac{m}{2} - 1) \\ n = (m - j - 1) \end{cases} \tag{8.16}$$

y se define la función $\mathbf{M}$ como la suma de los términos x_{ij} representados por los ciclos dados por la función $\mathbf{MC^m}$, obteniéndose finalmente que el producto interno $\mathbf{P_m} = \mathbf{M}$.

De la definición de las funciones $\mathbf{EC_i^m}$ y $\mathbf{OC_i^m}$, se puede observar que el número total de ciclos dados por cada una de ellas, para un valor determinado de i, es igual a $\frac{m}{2}$ (se debe recordar que se están considerando valores de m *pares*, ya que estos son los valores para los que un AOP es irreducible, como se vio al comienzo de la sección 8.3). Esta observación también es válida para la función $\mathbf{MC^m}$. Se puede comprobar esta propiedad en el ejemplo 5 en el que se daba la octava coordenada del producto para el campo $GF(2^{10})$ y donde se obtuvieron los *cinco* ciclos (0,1)(3,9)(4,8)(5,7)(6).

Se pueden establecer finalmente las nuevas expresiones generales para el cálculo de las coordenadas del producto $\delta = \alpha \cdot \chi$ en la base canónica Ω dadas por la ecuación 8.13, utilizando las funciones $\mathbf{E_i}$, $\mathbf{O_i}$ y $\mathbf{M}$ y asumiendo que estas funciones operan con las coordenadas de α y χ en Ω, como [ISF02]

$$d_{\Omega_i} = \mathbf{E_0} + \begin{cases} \mathbf{O_{i+1}} & i \ par \\ \mathbf{E_{i+1}} & i \ impar, \ \ i \neq m-1 \\ \mathbf{M} & i = m-1 \end{cases} \tag{8.17}$$

Al método dado por la ecuación 8.17 se le denomina método *transposicional* porque se basa en el cálculo de los 1-ciclos y de los 2-ciclos (*transposiciones*).

Las expresiones de las coordenadas del producto obtenidas con el método *transposicional* (ecuación 8.17) están dadas en forma de *sumas de productos* y coinciden con las obtenidas utilizando el nuevo algoritmo de multiplicación (ecuación 8.12), pero con sus términos producto colocados en *posiciones diferentes* dentro de las sumas de productos. Esta ordenación diferente es debida a las funciones $\mathbf{E_i}$, $\mathbf{O_i}$ y $\mathbf{M}$, ya que tienden a *agrupar* las coordenadas de entrada por medio de los términos $x_{ij} = (a_i c_j + a_j c_i)$ y $x_k = (a_k c_k)$ representados por los 2-ciclos (i, j) y por los 1-ciclos (k), respectivamente. Esta *agrupación* de términos producto es la que hace que la herramienta de síntesis reconfigurable realice mejores optimizaciones de área, como se comprobará posteriormente en los resultados experimentales. También se observa que la ecuación 8.17 es más simple y más general que la ecuación 8.12, porque las coordenadas del producto se pueden obtener calculando únicamente los ciclos dados por las ecuaciones 8.14, 8.15 y 8.16 para los valores de m e i. Ilustramos el nuevo algoritmo de multiplicación y el método transposicional a través del siguiente ejemplo.

8.3.1.2. Ejemplo de multiplicación sobre $GF(2^4)$

Sea la base canónica $\Omega = \{1, \omega, \omega^2, \omega^3\}$ generada por un AOP irreducible de grado 4, sean $\alpha, \delta, \chi \in GF(2^4)$ y sean $\underline{\alpha}_\Omega, \underline{\delta}_\Omega, \underline{\chi}_\Omega$ sus vectores de coordenadas $(a_0, a_1, a_2, a_3)^t$, $(d_0, d_1, d_2, d_3)^t$ y $(c_0, c_1, c_2, c_3)^t$, respectivamente, con respecto de Ω. Entonces, el producto $\delta = \alpha \cdot \chi$ se puede calcular utilizando el nuevo algoritmo de multiplicación dado en la sección 8.2 (y particularizado para un AOP en la subsección 8.3.1) como sigue:

1. Se representa χ en la base triangular Λ, obteniéndose

$$\underline{\chi}_\Lambda = (c_3, c_3 + c_2, c_2 + c_1, c_1 + c_0)^t$$

2. Se construye la matriz de Hankel de $\underline{\chi}_\Lambda$, obteniéndose

$$\mathbf{H}(\underline{\chi}_\Lambda) = \begin{pmatrix} c_3 & c_3 + c_2 & c_2 + c_1 & c_1 + c_0 \\ c_3 + c_2 & c_2 + c_1 & c_1 + c_0 & c_0 \\ c_2 + c_1 & c_1 + c_0 & c_0 & c_3 \\ c_1 + c_0 & c_0 & c_3 & c_3 + c_2 \end{pmatrix}$$

3. Se construye la matriz $\mathbf{K}(\underline{\chi}_\Lambda)$, obteniéndose

$$\mathbf{K}(\underline{\chi}_\Lambda) = \begin{pmatrix} c_3 & c_2 & c_1 & c_0 \\ c_3 + c_2 & c_3 + c_1 & c_3 + c_0 & c_3 \\ c_2 + c_1 & c_2 + c_0 & c_2 & c_3 + c_2 \\ c_1 + c_0 & c_1 & c_3 + c_1 & c_2 + c_1 \end{pmatrix} =$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ c_3 & c_3 & c_3 & c_3 \\ c_2 & c_2 & c_2 & c_2 \\ c_1 & c_1 & c_1 & c_1 \end{pmatrix} + \begin{pmatrix} c_3 & c_2 & c_1 & c_0 \\ c_2 & c_1 & c_0 & 0 \\ c_1 & c_0 & 0 & c_3 \\ c_0 & 0 & c_3 & c_2 \end{pmatrix} = \mathbf{K_1} + \mathbf{K_2}$$

4. Las coordenadas (invertidas) de $\underline{\delta}_\Omega$ se calculan realizando el producto de $\underline{\alpha}^t_\Omega$ por $(\mathbf{K_1} + \mathbf{K_2})$, obteniéndose finalmente

$$\begin{aligned} d_0 &= a_1c_3 + a_2c_2 + a_3c_1 + a_0c_0 + a_2c_3 + a_3c_2 \\ d_1 &= a_1c_3 + a_2c_2 + a_3c_1 + a_0c_1 + a_1c_0 + a_3c_3 \\ d_2 &= a_1c_3 + a_2c_2 + a_3c_1 + a_0c_2 + a_1c_1 + a_2c_0 \\ d_3 &= a_1c_3 + a_2c_2 + a_3c_1 + a_0c_3 + a_1c_2 + a_2c_1 + a_3c_0 \end{aligned} \tag{8.18}$$

Se puede comprobar que las expresiones de las coordenadas del producto dadas en la ecuación 8.18 coinciden exactamente (con los términos producto escritos en el *mismo orden*) con las expresiones obtenidas por otras aproximaciones similares encontradas en la literatura [KS98][HK00][ZP01].

A continuación utilizamos el método *transposicional* dado por la ecuación 8.17 para la obtención de las coordenadas del producto $\delta = \alpha \cdot \chi$. En este ejemplo el campo de Galois es $GF(2^4)$, con lo que el valor de m será 4. Se tendrán que calcular, por lo tanto, los ciclos dados por las funciones $\mathbf{EC_0^4}$, $\mathbf{OC_1^4}$, $\mathbf{EC_2^4}$, $\mathbf{OC_3^4}$ y $\mathbf{MC^4}$ utilizando las ecuaciones 8.14, 8.15 y 8.16, obteniéndose que $\mathbf{EC_0^4} = (2)(1,3)$, $\mathbf{OC_1^4} = (0)(2,3)$, $\mathbf{EC_2^4} = (3)(0,1)$, $\mathbf{OC_3^4} = (1)(0,2)$ y $\mathbf{MC^4} = (0,3)(1,2)$. Ahora se pueden calcular las funciones $\mathbf{E_0}$, $\mathbf{O_1}$, $\mathbf{E_2}$, $\mathbf{O_3}$ y $\mathbf{M}$, respectivamente, como la suma de los términos x_{ij} y x_k representados por estos ciclos:

$$\begin{array}{lcll} \mathbf{E_0} & = & x_2 + x_{13} & = a_2c_2 + (a_1c_3 + a_3c_1) \\ \mathbf{O_1} & = & x_0 + x_{23} & = a_0c_0 + (a_2c_3 + a_3c_2) \\ \mathbf{E_2} & = & x_3 + x_{01} & = a_3c_3 + (a_0c_1 + a_1c_0) \\ \mathbf{O_3} & = & x_1 + x_{02} & = a_1c_1 + (a_0c_2 + a_2c_0) \\ \mathbf{M} & = & x_{03} + x_{12} & = (a_0c_3 + a_3c_0) + (a_1c_2 + a_2c_1) \end{array} \tag{8.19}$$

Utilizando la ecuación 8.17 se obtiene finalmente:

$$\begin{array}{lcll} d_0 & = & \mathbf{E_0} + \mathbf{O_1} & = a_2c_2 + (a_1c_3 + a_3c_1) + a_0c_0 + (a_2c_3 + a_3c_2) \\ d_1 & = & \mathbf{E_0} + \mathbf{E_2} & = a_2c_2 + (a_1c_3 + a_3c_1) + a_3c_3 + (a_0c_1 + a_1c_0) \\ d_2 & = & \mathbf{E_0} + \mathbf{O_3} & = a_2c_2 + (a_1c_3 + a_3c_1) + a_1c_1 + (a_0c_2 + a_2c_0) \\ d_3 & = & \mathbf{E_0} + \mathbf{M} & = a_2c_2 + (a_1c_3 + a_3c_1) + (a_0c_3 + a_3c_0) + (a_1c_2 + a_2c_1) \end{array} \tag{8.20}$$

Las expresiones de las coordenadas dadas en la ecuación 8.20 son iguales que las expresiones calculadas en 8.18, pero *reescritas* de forma diferente: ha habido una *reordenación* dada por la *agrupación* de los términos producto en las sumas de productos que determinan cada coordenada d_i. También en la ecuación 8.20 se puede observar claramente la *compartición* entre todas las coordenadas de la subexpresión $a_2c_2 + (a_1c_3 + a_3c_1)$ determinada por el término $\mathbf{E_0}$. Por otra parte, se tiene que para el cálculo de las ecuaciones obtenidas en 8.20 únicamente ha sido necesario conocer el tamaño del campo finito (dado por m) y aplicar la ecuación 8.17.

8.3.1.3. Arquitectura paralela

Se puede extraer fácilmente la arquitectura paralela del multiplicador en base canónica obtenido utilizando el nuevo algoritmo de multiplicación o el método *transposicional* aplicado a un AOP irreducible, examinando las expresiones dadas por sus ecuaciones correspondientes. Se pueden distinguir de esta forma los siguientes módulos:

- Un módulo $\mathbf{PI_m}$ que realice el producto interno de dos vectores de longitud m usado para el cálculo del término $(\underline{\alpha}_\Omega^t \mathbf{K_2})_0$. Este término es, en realidad, el producto interno $\mathbf{P_m}$ definido anteriormente y que corresponde a la función $\mathbf{M}$ en el método transposicional.

- m módulos $\mathbf{PI_{m-1}}$ que realicen el producto interno de dos vectores de longitud $m-1$:
  - $m-1$ módulos utilizados para el cálculo de los términos $(\underline{\alpha}_\Omega^t \mathbf{K_2})_1$, $(\underline{\alpha}_\Omega^t \mathbf{K_2})_2$, ..., $(\underline{\alpha}_\Omega^t \mathbf{K_2})_{m-1}$. En el método *transposicional*, se tiene que $(\underline{\alpha}_\Omega^t \mathbf{K_2})_1 = \mathbf{O_{m-1}}$, $(\underline{\alpha}_\Omega^t \mathbf{K_2})_2 = \mathbf{E_{m-2}}$, ..., $(\underline{\alpha}_\Omega^t \mathbf{K_2})_{m-1} = \mathbf{O_1}$.
  - 1 módulo usado para el cálculo del término $\mathbf{P_{m-1}}$ definido anteriormente y que en el método *transposicional* corresponde a $\mathbf{P_{m-1}} = \mathbf{E_0}$.
- Un módulo sumador **Add** utilizado para realizar la *suma* de $\mathbf{P_{m-1}}$ con los m términos $(\underline{\alpha}_\Omega^t \mathbf{K_2})_0$, $(\underline{\alpha}_\Omega^t \mathbf{K_2})_1$, ..., $(\underline{\alpha}_\Omega^t \mathbf{K_2})_{m-1}$, obteniéndose las salidas del multiplicador $d_{\Omega_0}, d_{\Omega_1}, \ldots, d_{\Omega_{m-1}}$.
- También existirán unos módulos de *reordenación* $\mathbf{R}$ (deducidos fácilmente de la estructura de las matrices $\mathbf{K_1}$ y $\mathbf{K_2}$) que tienen como entradas las coordenadas de los operandos $a_{\Omega_0}, a_{\Omega_1}, \ldots, a_{\Omega_{m-1}}$ y $c_{\Omega_0}, c_{\Omega_1}, \ldots, c_{\Omega_{m-1}}$ y cuyas salidas se llevan a las entradas de los módulos que realizan el producto interno ($\mathbf{PI_{m-1}}$ y $\mathbf{PI_m}$).

En la figura 8.1 se muestra el diagrama de bloques de la arquitectura paralela correspondiente al multiplicador en base canónica propuesto, donde aparecen los módulos de reordenación ($\mathbf{R}$), los módulos que realizan el producto interno de vectores de longitud $m-1$ ($\mathbf{PI_{m-1}}$) y de longitud m ($\mathbf{PI_m}$), y el módulo sumador (**Add**) cuyas salidas constituyen el resultado de la multiplicación (coordenadas del producto).

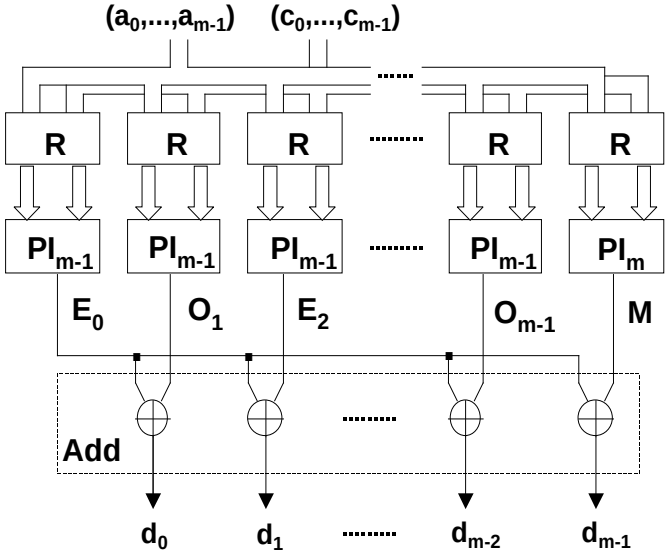


Figura 8.1: Arquitectura paralela del multiplicador en base *canónica*.

8.3.1.4. Análisis teórico de complejidad

Ahora consideramos la complejidad teórica del multiplicador en términos del número de puertas AND y XOR necesarias (consumo de área) y en términos de retardos de puerta (retardo total del multiplicador). Para ello, consideramos las complejidades de cada uno de los módulos constituyentes del multiplicador canónico paralelo de la siguiente forma [ISF02]:

- Un módulo $\mathbf{PI_m}$ que realiza el producto interno de dos vectores de longitud m, genera los términos producto en paralelo utilizando m puertas AND y, posteriormente, sumando los productos parciales utilizando un árbol binario de $m-1$ puertas XOR. La profundidad del árbol binario XOR será $\lceil log_2(m-1) \rceil$, por lo que el retardo de este módulo será $T_{AND} + \lceil log_2(m-1) \rceil T_{XOR}$, donde T_{AND} y T_{XOR} representan los retardos de las puertas AND y XOR, respectivamente.

- Un módulo $\mathbf{PI_{m-1}}$ que realiza el producto interno de dos vectores de longitud $m-1$ utiliza $m-1$ puertas AND y un árbol binario de $m-2$ puertas XOR. El retardo de este módulo será $T_{AND} + \lceil log_2(m-1) \rceil T_{XOR}$.

- El módulo sumador **Add** utilizará m puertas XOR en paralelo y, por tanto, su retardo será T_{XOR}.

- Los módulos de reordenación **R** no tienen coste de puertas ni de retardo.

Por lo tanto, el coste teórico total en número de puertas del multiplicador paralelo en base canónica es de $m + m(m-1) = m^2$ puertas AND y de $(m-1)+m(m-2)+m = m^2-1$ puertas XOR, mientras que el retardo teórico total es $T_{AND} + \lceil log_2(m-1) \rceil T_{XOR} + T_{XOR} = T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$.

En la tabla 8.1 se muestran las complejidades teóricas obtenidas utilizando nuestros algoritmos y las obtenidas por otras aproximaciones similares encontradas en la literatura, para un multiplicador paralelo en base canónica generado por un AOP (existen otros métodos de multiplicación sobre AOPs [Leo01] que reducen el número de puertas, pero a costa de un incremento en el retardo de puertas XOR).

	#XOR	#AND	Retardo
[IT89]	$m^2 + 2m$	$m^2 + 2m + 1$	$T_{AND} + \lceil log_2 m + log_2(m+2) \rceil T_{XOR}$
[HWB92]	$m^2 + m - 2$	m^2	$T_{AND} + (m + \lceil log_2(m-1) \rceil) T_{XOR}$
[KS98]	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$
[HK00]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$
[ZP01]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$
Ntra.Ap.	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$

Tabla 8.1: Comparación de complejidades teóricas de multiplicadores canónicos.

De los resultados mostrados en la tabla 8.1, se observa que las complejidades teóricas obtenidas usando nuestras aproximaciones (*Ntra.Ap.* en la tabla) igualan los mejores resultados obtenidos con otros métodos similares, sin embargo, la ventaja del método *transposicional* se encuentra cuando se eligen plataformas reconfigurables para la implementación del multiplicador, como se observa en los resultados experimentales que se muestran a continuación.

8.3.1.5. Resultados experimentales

Como se concluyó en el capítulo 7, la complejidad teórica de los multiplicadores vista anteriormente y dada en la tabla 8.1 no proporciona una predicción exacta del consumo de área ni del retardo cuando se realizan implementaciones *reconfigurables*. Para comprobar el mejor comportamiento del método *transposicional* con respecto de otras aproximaciones similares dadas en la literatura cuando se utiliza hardware reconfigurable, se han realizado implementaciones de varios multiplicadores en base canónica generados por AOPs sobre dispositivos FPGAs y CPLDs. Para ello, se han utilizado las expresiones 8.17 para describir los multiplicadores por medio del método transposicional y las ecuaciones 8.12 para describir los multiplicadores por otros métodos. Como ya se mencionó en la subsección 8.3.1, el nuevo algoritmo de multiplicación propuesto en la misma y descrito por la ecuación 8.12 proporciona expresiones iguales a las obtenidas utilizando otros algoritmos similares de multiplicación canónica [KS98][HK00][ZP01]. En estas expresiones, las coordenadas del producto aparecen en forma de sumas de productos de los operandos de entrada sin que estos métodos determinen ninguna *agrupación* de los términos producto ni ningún tipo de *compartición* de subexpresiones. El método transposicional sí que proporciona expresiones en las que aparecen términos producto *agrupados* (obtenidos de los 1-ciclos y de los 2-ciclos) y donde se indican las subexpresiones que se pueden *compartir* (si existen) entre las distintas coordenadas. Para el caso de los AOPs se da tanto la agrupación de términos producto como la compartición de la subexpresión $\mathbf{E_0}$. En el capítulo 9 se verán ejemplos de trinomios irreducibles para los que no existe *compartición* de subexpresiones.

Se ha empleado *Xilinx Foundation F2.1i* y se han realizado descripciones en VHDL de los multiplicadores paralelos utilizando las ecuaciones anteriormente mencionadas para los dos métodos usados. Para la implementación de los multiplicadores sobre FPGAs, se han utilizado los campos finitos $GF(2^m)$ generados por AOPs con $m = 4$, 10, 12, 18, 28 y 36, y se han utilizado los dispositivos 4013XLPQ160 pertenecientes a la familia XC4000XL. Asimismo, las métricas del rendimiento utilizadas para la realización de las comparaciones han sido las mismas que las utilizadas en el capítulo 7, es decir, el *número de CLBs* (área) y el *retardo máximo combinacional* (tiempo) expresado en nanosegundos. Como estamos interesados en la comparación de *área*, la síntesis se realizó usando optimización para *Area* con un nivel de esfuerzo *High*.

En la tabla 8.2 se muestran los resultados experimentales obtenidos [ISF02] para los campos seleccionados utilizando las expresiones dadas por el nuevo algoritmo de multiplicación (y, por tanto, las expresiones dadas por otros métodos similares) y las obtenidas con el método *transposicional*. También se muestra el número total de CLBs y el retardo total obtenido para todos los multiplicadores implementados por cada uno de los dos métodos.

	Otros métodos		Transposicional	
	CLBs	Ret. Max (ns)	CLBs	Ret. Max (ns)
$GF(2^4)$	6	14.7	5	14.8
$GF(2^{10})$	40	25.0	37	25.8
$GF(2^{12})$	54	27.3	53	28.6
$GF(2^{18})$	116	33.0	106	36.8
$GF(2^{28})$	281	45.3	259	43.0
$GF(2^{36})$	454	51.7	428	55.1
Total	951	197.0	888	204.1

Tabla 8.2: Resultados experimentales de multiplicadores canónicos sobre campos generados por AOPs utilizando FPGAs.

De los resultados mostrados en la tabla 8.2, se observa que la implementación de los multiplicadores utilizando la aproximación *transposicional* consume un menor número de CLBs que si se utiliza el nuevo algoritmo de multiplicación y, por tanto, que si se utilizan otros métodos de multiplicación similares [KS98][HK00][ZP01]. Considerando el número total de CLBs consumidos por cada método para todas las implementaciones, se tiene que el método *transposicional* consume un 6.6 % menos de CLBs que las otras aproximaciones. Los mejores resultados se obtienen para los campos $GF(2^4)$, $GF(2^{18})$ y $GF(2^{28})$ en los que la reducción del número de CLBs utilizando el método *transposicional* es de un 16.6 %, 8.6 % y 7.8 %, respectivamente. Con respecto al máximo retardo combinacional se obtiene que, considerando los retardos totales de ambos métodos, la aproximación transposicional produce implementaciones un 3.6 % más lentas que los otros métodos, pero esto sería debido a que en los experimentos realizados se han establecido optimizaciones para *área*, no para tiempo.

La explicación de la reducción del número de CLBs obtenida con el método *transposicional* sería la siguiente. Uno de los tipos de FPGAs disponible comercialmente es la FPGA basada en LUT (*look-up table*), en la que los recursos programables son controlados por celdas RAM [Xil92]. En este tipo de FPGAs, cada CLB contiene al menos una de estas tablas de búsqueda (LUT), donde una LUT puede implementar cualquier función Booleana cuyo número de variables no exceda la restricción hardware en el número de entradas dada por un valor k. Las herramientas automáticas típicas de diseño con FPGAs realizan, en primer lugar, una reducción de la complejidad de la red Booleana del circuito que se desea implementar. Para ello, la herramienta de *optimi-*

zación lógica encuentra equivalentes algebraicos de las expresiones Booleanas de forma que se minimice el número de literales de la red. A continuación, la herramienta de *mapping* transforma la red en una red equivalente en la que el número de entradas de cada puerta sea *igual* o *menor* que k, porque una puerta lógica con más de k entradas no se puede realizar con una sola k-LUT. Por lo tanto, las puertas con un menor número de entradas se pueden incluir fácilmente en una LUT y pueden incrementar la posibilidad de que la herramienta de *mapping* obtenga una mejor solución [KL01].

En este sentido, la descripción del multiplicador utilizando el método *transposicional* ayuda a las herramientas automáticas debido a que proporciona ya en la descripción de alto nivel *agrupaciones* de términos producto por medio de los 2-ciclos (i, j) y de los 1-ciclos (k) que determinan los términos $x_{ij} = (a_i c_j + a_j c_i)$ y $x_k = (a_k c_k)$, respectivamente. Es decir, permite la formación de *grupos* de dos puertas AND de dos entradas y una puerta XOR de dos entradas que se pueden realizar fácilmente con una LUT, por lo que la herramienta de *mapping* reconfigurable puede realizar una mejor optimización para área [KL01] incluyendo dentro de un mismo CLB términos x_{ij} y/o x_k que estén relacionados. Por lo tanto, esta técnica de *agrupación* parece constituir un buen método cuando se utilizan plataformas reconfigurables como las FPGAs basadas en LUT (como la familia XC4000XL utilizada en los experimentos) para la implementación de los multiplicadores.

Para comprobar si el método *transposicional* sigue presentando un mejor comportamiento que las otras aproximaciones cuando se utiliza otra plataforma reconfigurable distinta, se ha realizado la implementación de los multiplicadores sobre CPLDs para los campos de Galois $GF(2^m)$ generados por AOPs con $m = 4$, 10, 12 y 18. Los CPLDs seleccionados han sido los dispositivos XC95288XV, pertenecientes a la familia XC9500XV, y las descripciones de los multiplicadores se han realizado en VHDL. La métrica del rendimiento utilizada para la complejidad *espacial* ha sido el *número de macroceldas* (MCs) empleadas, mientras que la métrica de complejidad *temporal* utilizada ha sido el *retardo de propagación* (*pad to pad propagation delay*) T_{PD}. El estilo de optimización seleccionada en este caso ha sido una optimización de área *balanced*. En la tabla 8.3 se muestran los resultados obtenidos utilizando el nuevo algoritmo de multiplicación (otros métodos similares) y el método *transposicional*.

De los resultados mostrados en la tabla 8.3, se observa que la implementación sobre CPLDs de los multiplicadores utilizando el método *transposicional* consume un menor número de macroceldas (MCs) que si se utilizan otros métodos de multiplicación similares [KS98][HK00][ZP01]. Considerando el número total de MCs consumidos por cada método para todas las implementaciones, se tiene que el método *transposicional* consume un 12.8 % menos de MCs que las otras aproximaciones. El mejor resultado obtenido corresponde al campo $GF(2^{18})$, en el que la reducción del número de MCs es de un 17.8 %. Unicamen-

	Otros métodos		Transposicional	
	MCs	Ret. T_{PD} (ns)	MCs	Ret. T_{PD} (ns)
$GF(2^4)$	10	9.8	9	9.1
$GF(2^{10})$	55	17.3	57	15.9
$GF(2^{12})$	89	24.1	79	20.7
$GF(2^{18})$	213	38.4	175	17.3
Total	367	89.6	320	63.0

Tabla 8.3: Resultados experimentales de multiplicadores canónicos sobre campos generados por AOPs utilizando CPLDs.

te para el campo $GF(2^{10})$ se obtiene un mayor número de MCs utilizando el método *transposicional* (3.6 % mayor). Con respecto al retardo de propagación, también se obtienen mejores resultados utilizando la aproximación *transposicional*. Para el retardo total de las implementaciones, nuestro método es un 29.7 % más rápido, obteniéndose incluso un mejor caso para $GF(2^{18})$ en el que la reducción del retardo es de un 55 % con respecto de los otros métodos.

La arquitectura de los CPLDs es distinta de la arquitectura de las FPGAs, pero al igual que en estas, el método *transposicional* produce una reducción del número de macroceldas utilizadas en los CPLDs. Esta reducción sería debida a que la agrupación de términos producto generada por los 1-ciclos y 2-ciclos, ayuda a las herramientas de síntesis a la ubicación de los términos producto en las macroceldas de los *bloques funcionales*, de tal forma que el número total de macroceldas utilizadas se vea reducido. Por lo tanto, esta técnica de *agrupación* parece constituir un buen método de construcción de multiplicadores canónicos sobre AOPs para *cualquier* plataforma reconfigurable que se utilice.

8.3.2. Multiplicación en base normal óptima de Tipo I

Como se vio en el capítulo 7, una base $N = \{\zeta, \zeta^2, \zeta^{2^2}, \ldots, \zeta^{2^{m-1}}\}$ es una base *normal* de $GF(2^m)$ sobre $GF(2)$, donde $\zeta \in GF(2^m)$ y m es el grado del polinomio generador. También en el capítulo 7 se introdujeron las bases *normales óptimas*, en las que se cumple que para todo $0 \leq i_1 \neq i_2 \leq m-1$, 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 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* [Men93], con elemento *generador* ζ, de forma que

$$N = \{\zeta, \zeta^2, \zeta^{2^2}, \ldots, \zeta^{2^{m-1}}\} = \{\zeta, \zeta^2, \zeta^3, \ldots, \zeta^m\} \tag{8.21}$$

y en donde se observa que $\zeta\zeta^i = \zeta^{i+1} \in N$, $1 \leq i \leq m$, y que $\zeta\zeta^m = 1$. También se observa que ζ es una raíz del polinomio $x^m + x^{m-1} + \cdots + x + 1$.

Se vio anteriormente que un AOP es irreducible si y sólo si $m+1$ es primo y 2 es primitivo módulo $m+1$. 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 [Men93] y que la raíz de un AOP irreducible es un elemento *generador* de dicha base.

La base dada en la ecuación 8.21, $\{\zeta, \zeta^2, \zeta^3, \ldots, \zeta^m\}$, se puede considerar como una versión *desplazada* de la base canónica, por lo que se pueden utilizar los métodos vistos en la subsección anterior de multiplicación en base *canónica* para su aplicación a la multiplicación en base *normal* cuando se utiliza un AOP irreducible como polinomio generador del campo.

Sea $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_{m-1}\} = \{\zeta, \zeta^2, \ldots, \zeta^m\}$. Un elemento $\alpha \in GF(2^m)$ representado en la base *base normal óptima de Tipo I* se puede convertir a su representación en la *base canónica desplazada* utilizando simplemente una permutación de las coordenadas de α con respecto de la base normal N. Si a_{Γ_i} y a_{N_i} representan las coordenadas de α con respecto de Γ y N, respectivamente, entonces la conversión $\alpha = \sum_{i=0}^{m-1} a_{N_i}\zeta^{2^i} = \sum_{i=0}^{m-1} a_{\Gamma_i}\zeta^{i+1}$ se puede realizar utilizando la permutación definida por

$$a_{\Gamma_{(2^i-1) mod (m+1)}} = a_{N_i}, \quad i = 0, 1, \ldots, m-1 \tag{8.22}$$

Utilizando los hechos anteriores, se puede realizar la multiplicación en *base normal óptima de Tipo I* sobre el campo finito $GF(2^m)$ generado por un AOP irreducible de la siguiente forma.

8.3.2.1. Algoritmo de multiplicación en base normal

Sean $\alpha, \delta, \chi \in GF(2^m)$ y sean $\underline{\alpha}_N, \underline{\delta}_N, \underline{\chi}_N$ sus vectores de coordenadas, respectivamente, en la base normal N. Entonces, el producto $\delta = \alpha \cdot \chi$ se puede realizar utilizando el siguiente algoritmo [KS98]:

1. Se convierten los operandos α y χ representados en la base normal óptima de Tipo I como $\underline{\alpha}_N$ y $\underline{\chi}_N$, respectivamente, a la base *canónica desplazada* Γ utilizando la permutación dada en la ecuación 8.22, obteniéndose los vectores de coordenadas $\underline{\alpha}_\Gamma$ y $\underline{\chi}_\Gamma$, respectivamente.

2. Se realiza la multiplicación en base *canónica desplazada* de $\underline{\alpha}_\Gamma$ y $\underline{\chi}_\Gamma$ utilizando los algoritmos dados en la subsección 8.3.1 con algunas *modificaciones*, obteniéndose el vector de coordenadas del producto δ representado en la base *canónica desplazada* ($\underline{\delta}_\Gamma$).

3. Se convierte el resultado de la multiplicación, $\underline{\delta}_\Gamma$, a la base normal óptima de Tipo I realizando la inversa de la permutación dada en 8.22, obteniéndose finalmente $\underline{\delta}_N$.

La multiplicación en base *canónica desplazada* se puede realizar utilizando los mismos algoritmos vistos en la subsección 8.3.1 (el *nuevo algoritmo de multiplicación* y el *método transposicional*), pero con algunas *modificaciones* deducidas de las diferencias existentes entre la base *canónica* Ω y la base *canónica desplazada* Γ. Los operandos de entrada a dichos algoritmos serán $\underline{\alpha}_\Gamma$ y $\underline{\chi}_\Gamma$ (paso 2 del algoritmo de multiplicación en base normal óptima de Tipo I dado anteriormente), y el resultado obtenido será $\underline{\delta}_\Gamma$, por lo que las *modificaciones* a realizar en los algoritmos introducidos en la sección 8.2 y en la subsección 8.3.1 serán las siguientes:

- El cálculo del vector de coordenadas de χ en la base triangular Λ utilizando el vector de coordenadas $\underline{\chi}_\Gamma$ (paso 1 del *nuevo algoritmo de multiplicación en base canónica* dado en la sección 8.2), se realiza utilizando la expresión $\underline{\chi}_\Lambda = \mathbf{T}' \cdot \underline{\chi}_\Gamma$, donde la matriz $\mathbf{T}'$ es de la forma

$$\mathbf{T}' = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & 1 & t_1 \\ 0 & 0 & 0 & \cdots & 1 & t_1 & t_2 \\ 0 & 0 & 0 & \cdots & t_1 & t_2 & t_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 1 & t_1 & \cdots & t_{m-4} & t_{m-3} & t_{m-2} \\ 1 & t_1 & t_2 & \cdots & t_{m-3} & t_{m-2} & t_{m-1} \\ t_1 & t_2 & t_3 & \cdots & t_{m-2} & t_{m-1} & t_m \end{pmatrix} \tag{8.23}$$

  siendo $t_j = \sum_{i=0}^{j-1} f_{m-j+i} t_i$, para $0 < j \leq m$ y con $t_0 = 1$. Para un AOP, se vio en la sección 8.3 que el coeficiente $t_1 = 1$, mientras que $t_2 = t_3 = \cdots = t_m = 0$.

- Las matrices $m \times m$ $\mathbf{K_1}$ y $\mathbf{K_2}$ definidas en la subsección 8.3.1 (ecuaciones 8.10 y 8.11, respectivamente) vendrán dadas en la multiplicación en base *canónica desplazada* como

$$\mathbf{K_1} = \begin{pmatrix} c_{\Gamma_{m-1}} & c_{\Gamma_{m-1}} & \cdots & c_{\Gamma_{m-1}} & c_{\Gamma_{m-1}} \\ c_{\Gamma_{m-2}} & c_{\Gamma_{m-2}} & \cdots & c_{\Omega_{m-2}} & c_{\Gamma_{m-2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{\Gamma_2} & c_{\Gamma_2} & \cdots & c_{\Gamma_2} & c_{\Gamma_2} \\ c_{\Gamma_1} & c_{\Gamma_1} & \cdots & c_{\Gamma_1} & c_{\Gamma_1} \\ c_{\Gamma_0} & c_{\Gamma_0} & \cdots & c_{\Gamma_0} & c_{\Gamma_0} \end{pmatrix} \tag{8.24}$$

$$\mathbf{K_2} = \begin{pmatrix} c_{\Gamma_{m-2}} & c_{\Gamma_{m-3}} & \cdots & c_{\Gamma_0} & 0 \\ c_{\Gamma_{m-3}} & c_{\Gamma_{m-4}} & \cdots & 0 & c_{\Gamma_{m-1}} \\ c_{\Gamma_{m-4}} & c_{\Gamma_{m-5}} & \cdots & c_{\Gamma_{m-1}} & c_{\Gamma_{m-2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{\Gamma_0} & 0 & \cdots & c_{\Gamma_3} & c_{\Gamma_2} \\ 0 & c_{\Gamma_{m-1}} & \cdots & c_{\Gamma_2} & c_{\Gamma_1} \end{pmatrix} \tag{8.25}$$

donde los términos c_{Γ_i} son las coordenadas de χ con respecto de Γ. Al igual que ocurría en las ecuaciones 8.10 y 8.11, $\mathbf{K_1}$ es una matriz con sus m columnas idénticas y $\mathbf{K_2}$ es una matriz de Hankel.

- Todas las consideraciones realizadas en el método *transposicional* siguen siendo válidas para la multiplicación en base *canónica desplazada* utilizándose, en este caso, las matrices $\mathbf{K_1}$ y $\mathbf{K_2}$ definidas anteriormente en las ecuaciones 8.24 y 8.25. Además, empleando las funciones $\mathbf{EC_i^m}$, $\mathbf{OC_i^m}$ y $\mathbf{MC^m}$ definidas en las ecuaciones 8.14, 8.15 y 8.16, respectivamente, y utilizando las funciones $\mathbf{E_i}$, $\mathbf{O_i}$ y $\mathbf{M}$ también definidas en el método *transposicional*, se tiene que las expresiones generales para el cálculo de las coordenadas del producto $\delta = \alpha \cdot \chi$ en la base *canónica desplazada* Γ utilizando el método *transposicional* vienen dadas por

$$d_{\Gamma_i} = \mathbf{M} + \begin{cases} \mathbf{E_i} & i \ par \\ \mathbf{O_i} & i \ impar \end{cases} \tag{8.26}$$

donde los términos d_{Γ_i} son las coordenadas de δ con respecto de Γ.

8.3.2.2. Arquitectura paralela y análisis de complejidad

La arquitectura paralela del multiplicador en *base normal óptima de Tipo I* construido utilizando el algoritmo dado anteriormente (con las modificaciones reseñadas) se extrae realizando las mismas consideraciones que en el apartado 8.3.1.3 para el multiplicador canónico y se utilizan, por lo tanto, los mismos módulos que en dicho multiplicador. Las únicas diferencias con respecto del multiplicador en base canónica son:

- Inclusión de módulos de *reordenación*, utilizados para la conversión de representación de base *normal* a base *canónica desplazada* de los operandos de entrada (paso 1 del algoritmo de multiplicación en base normal) y para la conversión de representación del producto de base *canónica desplazada* a base *normal* (paso 3 del algoritmo de multiplicación en base normal). Estos módulos realizan la *permutación* dada en la ecuación 8.22 y su *permutación inversa*, respectivamente.

- Modificación de las interconexiones de los módulos en el multiplicador de base *canónica desplazada* (paso 2 del algoritmo de multiplicación en base normal óptima de Tipo I), que se deducen fácilmente a partir de los cambios introducidos anteriormente en 8.24, 8.25 y 8.26.

En la figura 8.2 se muestra el diagrama de bloques de la arquitectura paralela correspondiente al multiplicador de base *canónica desplazada*, donde se pueden observar las modificaciones realizadas en las conexiones de los distintos módulos con respecto del multiplicador en base canónica de la figura 8.1.

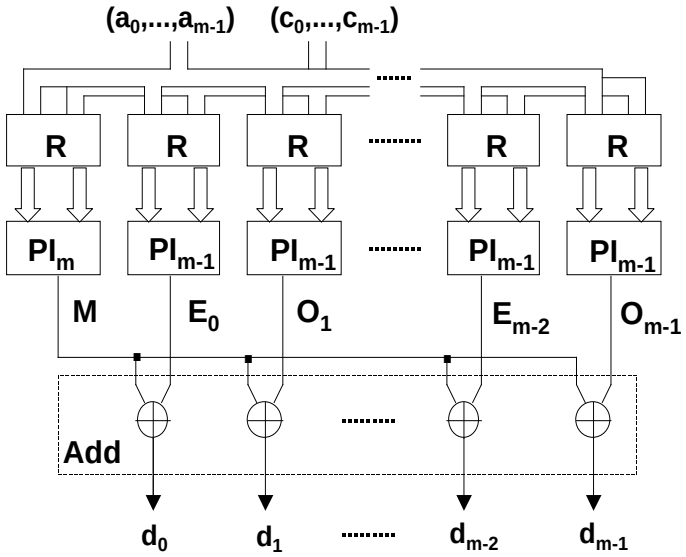


Figura 8.2: Arquitectura paralela del multiplicador en base *canónica desplazada*.

Con respecto de las complejidades teóricas *espacial* (número de puertas AND y XOR) y *temporal* (retardo expresado en retardos de puertas AND y XOR) del multiplicador en *base normal óptima de Tipo I*, se observa que ambas complejidades son idénticas a las complejidades respectivas obtenidas para el multiplicador en base canónica dadas en el apartado 8.3.1.4, ya que se utilizan los mismos módulos en ambos multiplicadores. Como se ha mencionado anteriormente, las únicas diferencias existentes se encuentran en las interconexiones de los distintos módulos y en la inclusión de nuevos módulos de *reordenación* que realizan las conversiones de base *normal* a base *canónica desplazada* para los operandos de entrada y de base *canónica desplazada* a base *normal* para la conversión del producto. Sin embargo, ninguna de estas modificaciones introduce complejidad adicional al multiplicador, ni en número de puertas ni en retardo. Por lo tanto, las complejidades teóricas *espacial* y *temporal* del multiplicador en *base normal óptima de Tipo I* son las mismas que las calculadas para el multiplicador en base canónica, es decir, m^2 puertas AND, $m^2 - 1$ puertas XOR y $T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$.

En la tabla 8.4 se muestran las complejidades teóricas obtenidas utilizando nuestra aproximación (*Ntra.Ap.*) y las obtenidas por otros métodos similares [OM86][HWB93][KS98] encontrados en la literatura, para un multiplicador paralelo en base normal generada por un AOP irreducible. Al igual que en el caso de la multiplicación en base canónica, existen otros métodos de multiplicación normal sobre AOPs irreducibles [Leo01] que reducen el número de puertas, pero a costa de un incremento en el retardo de puertas XOR.

	#XOR	#AND	Retardo
[OM86]	$2m^2 - 2m$	m^2	$T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$
[HWB93]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$
[KS98]	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$
Ntra.Ap.	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2(m-1) \rceil) T_{XOR}$

Tabla 8.4: Comparación de complejidades teóricas de multiplicadores en base normal.

Se observa en la tabla 8.4 que las complejidades teóricas obtenidas usando nuestra aproximación igualan los mejores resultados obtenidos con otros métodos similares, sin embargo, la ventaja del método *transposicional* se presenta cuando se realizan implementaciones sobre plataformas reconfigurables, como se muestra a continuación.

8.3.2.3. Resultados experimentales

Se ha realizado la implementación sobre dispositivos FPGA de multiplicadores en base normal utilizando el algoritmo de multiplicación en *base normal óptima de Tipo I* dado en el apartado 8.3.2.1 (en el que la multiplicación en base *canónica desplazada* se ha realizado usando el método *transposicional*) y utilizando el método dado por Koç y Sunar en [KS98]. Para ello, se ha utilizado *Xilinx Foundation F2.1i* y se han realizado descripciones VHDL de los multiplicadores paralelos utilizando ambos métodos para los campos finitos $GF(2^m)$ generados por AOPs irreducibles con $m = 4$, 10, 12, 18, 28 y 36. Los dispositivos utilizados han sido los 4013XLPQ160 pertenecientes a la familia XC4000XL y la síntesis se realizó utilizando optimización para *Area* con nivel de esfuerzo *High*. Asimismo, las métricas del rendimiento utilizadas han sido las mismas que en experimentos anteriores, es decir, el *número de CLBs* y el *retardo máximo combinacional*. Los resultados experimentales obtenidos se muestran en la tabla 8.5.

	Koç & Sunar		Transposicional	
	CLBs	Ret. Max (ns)	CLBs	Ret. Max (ns)
$GF(2^4)$	6	14.5	6	15.4
$GF(2^{10})$	39	28.9	37	27.2
$GF(2^{12})$	54	28.6	53	29.4
$GF(2^{18})$	103	37.6	107	35.0
$GF(2^{28})$	271	45.1	260	44.0
$GF(2^{36})$	458	46.9	429	50.6
Total	931	201.6	892	201.6

Tabla 8.5: Resultados experimentales de multiplicadores normales sobre campos generados por AOPs utilizando FPGAs.

De los resultados mostrados en la tabla 8.5, se observa que la implementación de los multiplicadores en base normal sobre campos $GF(2^m)$ generados por AOPs utilizando la aproximación *transposicional* consume un menor número de CLBs que si se utiliza un método similar dado por Koç y Sunar en [KS98]. Considerando el número total de CLBs consumidos por cada método para todas las implementaciones, se observa que el método *transposicional* consume un 4.2 % menos de CLBs, obteniéndose los mejores resultados para los campos $GF(2^{36})$ y $GF(2^{10})$ con unas reducciones en el número de CLBs del 6.3 % y del 5.1 %, respectivamente. También se observa que ambos métodos proporcionan los mismos retardos totales, a pesar de que la síntesis se realizó estableciendo optimización para *área*, no para tiempo.

8.3.3. Comparación de los multiplicadores

La conclusión más importante que se extrae de la comparación de los dos tipos de multiplicación considerados (*canónica* y *normal óptima de Tipo I*) es la igualdad de complejidad teórica (tanto *espacial* como *temporal*) que se obtiene en ambos tipos de multiplicación. Se vio en el capítulo 7 que los multiplicadores en base normal eran los de mayor complejidad de entre los tres tipos de multiplicadores estudiados (*canónicos*, *normales* y *duales*). De ahí la importancia que el algoritmo de multiplicación en base normal dado en el apartado 8.3.2.1 tiene para la obtención de multiplicadores normales de complejidad reducida. Además, este algoritmo permite transferir las mejoras obtenidas en la multiplicación canónica a la multiplicación en base normal cuando se utilizan AOPs irreducibles como generadores del campo finito.

Con respecto de la utilización del método *transposicional* frente a otras aproximaciones similares encontradas en la literatura, se observa que las complejidades teóricas obtenidas por el método transposicional *igualan* las mejores complejidades *espaciales* y *temporales* obtenidas por esos otros métodos. Esta igualdad se obtiene tanto en la multiplicación canónica como en la multiplicación en base normal.

Sin embargo, esta igualdad teórica de complejidades no se refleja en las implementaciones realizadas sobre plataformas reconfigurables como FPGAs o CPLDs. Los resultados experimentales obtenidos con FPGAs para los multiplicadores en base *canónica* han mostrado una reducción promedio del 6.6 % (con mejoras máximas del 16.6 %) en cuanto al número de CLBs utilizados frente a los multiplicadores implementados utilizando otros métodos, mientras que usando CPLDs la mejora promedio ha sido de un 12.8 % (con una mejora máxima del 17.8 %) en el consumo de MCs. Con respecto a los retardos, nuestro método ha sido un 3.6 % más lento que los otros métodos cuando se han utilizado FPGAs, mientras que con CPLDs nuestro método fue un 29.7 % más rápido (con un valor máximo del 55.0 %) que las otras aproximaciones, a pesar de que las optimizaciones se realizaron para área, no para tiempo.

En cuanto a los resultados experimentales obtenidos en las implementaciones sobre FPGAs de los multiplicadores en base *normal*, la mejora promedio obtenida por el método *transposicional* ha sido algo más reducida que la obtenida en los multiplicadores canónicos, situándose en un 4.2 % con una mejora máxima del 6.3 %, mientras que nuestro método obtuvo retardos promedio idénticos a los obtenidos con otras aproximaciones.

8.4. Conclusiones

En este capítulo se ha presentado un nuevo método de multiplicación en base canónica sobre campos de Galois $GF(2^m)$ al que se le ha denominado método *transposicional*. Este nuevo método se basa en el cálculo de 1-ciclos y 2-ciclos (*transposiciones*) que permiten realizar la *agrupación* y *compartición* de subexpresiones comunes existentes entre las coordenadas del producto.

El método *transposicional* se extrae a partir de un nuevo algoritmo de multiplicación en base polinómica, basado en la utilización de una base *triangular* de representación, que depende del polinomio irreducible generador del campo finito seleccionado. Además, todo el desarrollo y formulación del método *transposicional* se ha realizado a partir de la aplicación del nuevo algoritmo de multiplicación en base canónica a los AOPs irreducibles, obteniéndose expresiones que determinan las coordenadas del producto de dos elementos de cualquier campo de Galois generado por dichos polinomios.

Se ha realizado el análisis de complejidad teórico (espacial y temporal) de los multiplicadores construidos usando el método *transposicional*, obteniéndose complejidades iguales a las mejores encontradas en la literatura. Asimismo, la utilización de AOPs irreducibles como polinomios generadores del campo finito ha permitido aplicar el método *transposicional* a la multiplicación en base *normal óptima de Tipo I*, pudiéndose obtener de esta forma multiplicadores normales cuyas complejidades teóricas espaciales y temporales igualan a las complejidades de los multiplicadores canónicos. Este hecho es muy importante, dada la elevada complejidad teórica (vista en el capítulo 7) de los multiplicadores normales frente a los canónicos o duales.

Se ha comprobado que la descripción de un multiplicador proporcionada por el método transposicional conlleva una reducción de complejidad *experimental* cuando se utiliza hardware reconfigurable para la implementación. Este hecho se ha constatado realizando implementaciones de numerosos multiplicadores utilizando las descripciones proporcionadas tanto por el método *transposicional* como por otros métodos dados en la literatura. En todos los experimentos realizados, nuestro método ha obtenido reducciones de *área* (número de CLBs o MCs) y, en algunos casos también de *tiempo*, para los multiplicadores implementados, independientemente del tipo de multiplicación realizada (*canónica* o *normal*) y del tipo de plataforma reconfigurable utilizada (FPGAs o CPLDs).

La reducción de área obtenida con nuestro método *transposicional* en la implementación de multiplicadores sobre FPGAs basadas en LUT se debe a que las *agrupaciones* de términos producto producidas por los 2-ciclos y los 1-ciclos permiten la formación de *grupos* de dos puertas AND de dos entradas y una puerta XOR de dos entradas que pueden ser fácilmente realizables con una LUT, por lo que la herramienta de *mapping* reconfigurable puede realizar una mejor optimización para área. Esto es debido a que las puertas con un menor número de entradas se pueden incluir fácilmente en una LUT y esto incrementa la posibilidad de que la herramienta de *mapping* obtenga mejores soluciones [KL01]. En este sentido, la descripción del multiplicador utilizando el método *transposicional* ayuda a las herramientas automáticas debido a que proporciona, ya en la descripción de alto nivel, dichas agrupaciones. Aunque los CPLDs tienen una arquitectura diferente de las FPGAs, se pueden hacer consideraciones similares con respecto del método *transposicional* y su ayuda a las herramientas automáticas en la ubicación de los términos producto entre las diferentes macroceldas cuando se utiliza este tipo de dispositivos reconfigurables para la implementación. Por lo tanto, esta técnica de *agrupación* dada por el método *transposicional* parece constituir un buen método cuando se utiliza *cualquier tipo* de plataformas reconfigurables para la implementación de los multiplicadores sobre $GF(2^m)$.

En el siguiente capítulo se amplía el estudio a la utilización del método *transposicional* para la multiplicación en base canónica sobre campos de Galois generados por distintos tipos de *trinomios* irreducibles.

Capítulo 9

Método transposicional aplicado a trinomios irreducibles

En este capítulo se aplica el método transposicional a la multiplicación en base canónica de elementos pertenecientes a campos de Galois $GF(2^m)$ generados por determinados trinomios irreducibles. La utilización de este método produce para ciertos trinomios una reducción teórica de la complejidad de los multiplicadores, mientras que para todos los trinomios estudiados se obtienen reducciones experimentales de complejidad cuando se utilizan dispositivos reconfigurables para su implementación.

En el capítulo anterior se introdujo el método *transposicional* de multiplicación, basado en el cálculo de 1-ciclos y 2-ciclos que determinan la *agrupación* y *compartición* de subexpresiones de las coordenadas de los elementos del campo de Galois $GF(2^m)$ a multiplicar. Este método se definió utilizando AOPs como polinomios irreducibles generadores del campo finito, obteniéndose tanto *agrupaciones* de términos producto como la *compartición* de *una* subexpresión entre las distintas coordenadas. Para los AOPs, el método transposicional no obtuvo reducciones de complejidad teóricas, aunque sí experimentales cuando se emplearon plataformas reconfigurables para la implementación.

En este capítulo, el método *transposicional* se aplica a la multiplicación en base canónica sobre campos $GF(2^m)$ generados por ciertos tipos de *trinomios* irreducibles. Estos polinomios fueron mencionados en el capítulo 7 por ser generadores de campos de Galois para los que se obtenía una reducción en la complejidad de los multiplicadores que operan sobre dichos campos.

La utilización del método transposicional con trinomios irreducibles produce no sólo la *agrupación* de términos pertenecientes a las coordenadas del producto, sino que también permite (para varios de los trinomios estudiados) la *compartición* de un cierto número de subexpresiones comunes a dichas coor-

denadas que produce una reducción *teórica* de la complejidad de los multiplicadores, en comparación con los mejores casos encontrados en la literatura. Asimismo, la implementación reconfigurable paralela de estos multiplicadores produce también una reducción *experimental* de las complejidades para *todos* los trinomios irreducibles considerados.

9.1. Trinomios irreducibles

En el capítulo 7 se mencionaron los *trinomios irreducibles* $f(x) = x^m + x^n + 1$ como un tipo especial de polinomios para los que se obtenían simplificaciones en determinadas multiplicaciones sobre campos de Galois. Por este motivo, se ha realizado un estudio para ciertos tipos de trinomios irreducibles.

El cálculo del producto δ en una base canónica Ω de dos elementos α y χ, $\delta = \alpha \cdot \chi$, pertenecientes a un campo finito generado por un *trinomio irreducible* se puede realizar utilizando el algoritmo de multiplicación introducido en la sección 8.2. En este algoritmo, es necesaria la construcción de las matrices $\mathbf{T}$ (definida en la ecuación 8.3), $\mathbf{H}$, $\mathbf{F}$ y $\mathbf{K}$ para la obtención de las coordenadas del producto en forma de *sumas de productos*. Se puede observar que cuando se utilizan *trinomios irreducibles* $f(x) = x^m + x^n + 1$ como polinomios generadores del campo, la matriz $\mathbf{F}$ dada por la ecuación 8.6 presenta únicamente dos términos f_i no nulos, correspondientes a los coeficientes f_m y f_n del polinomio distintos de cero. Asimismo y como ya se mencionó en la sección 8.2, se observa que la matriz $\mathbf{K}$ definida en la ecuación 8.5 se puede descomponer en la suma de una serie de matrices cuyo número dependerá de los valores de m y n correspondientes a los coeficientes distintos de cero del trinomio irreducible utilizado. Esta descomposición de la matriz $\mathbf{K}$ en una suma de matrices $m \times m$ se realiza de la siguiente forma

$$\mathbf{K} = \mathbf{K_0} + \sum_{i=1}^{\tau} \mathbf{K_i} \tag{9.1}$$

donde $\tau = \lceil \frac{m-1}{m-n} \rceil$ y donde la matriz $\mathbf{K_0}$ es común a cualquier trinomio irreducible seleccionado. La forma de $\mathbf{K_0}$ es la siguiente

$$\mathbf{K_0} = \begin{pmatrix} c_{\Omega_{m-1}} & c_{\Omega_{m-2}} & \cdots & c_{\Omega_1} & c_{\Omega_0} \\ c_{\Omega_{m-2}} & c_{\Omega_{m-3}} & \cdots & c_{\Omega_0} & c_{\Omega_{m-1}} \\ c_{\Omega_{m-3}} & c_{\Omega_{m-4}} & \cdots & c_{\Omega_{m-1}} & c_{\Omega_{m-2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{\Omega_1} & c_{\Omega_0} & \cdots & c_{\Omega_3} & c_{\Omega_2} \\ c_{\Omega_0} & c_{\Omega_{m-1}} & \cdots & c_{\Omega_2} & c_{\Omega_1} \end{pmatrix} \tag{9.2}$$

donde los términos c_{Ω_i} son las coordenadas de χ con respecto de Ω y donde se observa que $\mathbf{K_0}$ es una matriz de Hankel.

De la descomposición dada en la ecuación 9.1, se observa que para cualquier trinomio $f(x) = x^m + x^n + 1$ existe, al menos, una matriz $\mathbf{K_1}$ cuya forma se muestra en la ecuación 9.3 (donde se ha eliminado el subíndice Ω en los términos c_{Ω_i} por claridad) y que presenta varias particularidades. La primera de ellas es que la columna n-ésima de esta matriz (numeradas de la columna 1 a la m, de derecha a izquierda) tiene todos sus elementos nulos, observándose que coincide el índice de esta columna con el índice del coeficiente no nulo f_n (distinto de f_m) del trinomio irreducible seleccionado. Asimismo, también se puede considerar que esta n-ésima columna nula divide a la matriz $m \times m$ en dos *submatrices* formadas por las $m-n$ columnas situadas a la izquierda de la columna nula y por las $n-1$ columnas situadas a su derecha, respectivamente. Denotamos a la submatriz izquierda de dimensión $m \times (m-n)$ como $\mathbf{I_1}$ y a la submatriz derecha de dimensión $m \times (n-1)$ como $\mathbf{D_1}$, de forma que la matriz $\mathbf{K_1}$ se puede representar como $\mathbf{K_1} = (\mathbf{I_1}|\mathbf{D_1})$. Se puede observar que tanto la submatriz $\mathbf{I_1}$ como la $\mathbf{D_1}$ están formadas por el desplazamiento sucesivo de sus columnas menos significativas (numeradas de derecha a izquierda en orden creciente comenzando en el índice 1) una fila hacia abajo con inserción de cero en la fila superior. También se observa que la columna más significativa de $\mathbf{I_1}$ está formada por $m-n$ ceros en las filas superiores y por n filas no nulas que contienen coeficientes c_i, mientras que la columna menos significativa de $\mathbf{D_1}$ está formada por $m-n+1$ filas nulas y por $n-1$ filas con coeficientes c_i.

$$
\begin{array}{c}
\begin{array}{cccccccccc}
\multicolumn{4}{c}{\text{m}} & & \multicolumn{3}{c}{\cdots} & & \text{n} & & \multicolumn{3}{c}{\cdots} & & 1
\end{array} \\
\left(\begin{array}{cccccccccc}
0 & \cdots & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & \cdots & 0 & c_{m-1} & 0 & 0 & 0 & \cdots & 0 \\
0 & \cdots & c_{m-1} & c_{m-2} & 0 & 0 & 0 & \cdots & 0 \\
0 & \cdots & c_{m-2} & c_{m-3} & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \iddots & \vdots & \vdots & \vdots & \vdots & \vdots & \iddots & \vdots \\
0 & \cdots & c_{n+2} & c_{n+1} & 0 & 0 & 0 & \cdots & 0 \\
c_{m-1} & \cdots & c_{n+1} & c_n & 0 & 0 & 0 & \cdots & 0 \\
c_{m-2} & \cdots & c_n & c_{n-1} & 0 & 0 & 0 & \cdots & c_{m-1} \\
\vdots & \iddots & \vdots & \vdots & \vdots & \vdots & \vdots & \iddots & \vdots \\
c_{m-n+1} & \cdots & c_3 & c_2 & 0 & 0 & c_{m-1} & \cdots & c_{m-n+2} \\
c_{m-n} & \cdots & c_2 & c_1 & 0 & c_{m-1} & c_{m-2} & \cdots & c_{m-n+1}
\end{array}\right)
\begin{array}{l}
\text{n} \\
\\
\\
\\
\\
\\
\\
\\
\\
\\
\text{n-1}
\end{array} \\
\begin{array}{cccccccccc}
\multicolumn{4}{c}{\text{m-n}} & & \multicolumn{4}{c}{\text{n-1}}
\end{array}
\end{array}
\qquad (9.3)
$$

La existencia de matrices $\mathbf{K_2}$, $\mathbf{K_3}$, ..., en el sumatorio dado en la ecuación 9.1 dependerá de la posición de la n-ésima columna nula (es decir, del valor de la potencia no nula, n, distinta de m y de 0 del trinomio) en la matriz dada en la ecuación 9.3 como se determina a continuación.

Partiendo de la matriz $\mathbf{K_1}$ dada en la ecuación 9.3, se verifica que existirá una matriz $\mathbf{K_2} = (\mathbf{I_2}|\mathbf{D_2})$ en la descomposición de $\mathbf{K}$ dada en la ecuación 9.1 si la columna más significativa de $\mathbf{I_1}$ presenta $n > 1$ términos c_i no nulos. La construcción de las submatrices $\mathbf{I_2}$ y $\mathbf{D_2}$ se realizará a partir de $\mathbf{D_1}$ de la siguiente forma:

- Si $m < 2n - 1$, entonces existirá una matriz $\mathbf{K_3}$ (cuya construcción se realizará de manera análoga a la de $\mathbf{K_2}$ y a partir de ésta) y la submatriz $\mathbf{I_2}$ estará formada por las $m - n$ primeras columnas de $\mathbf{D_1}$ (referidas de derecha a izquierda), mientras que las $2n - m - 1$ restantes columnas de $\mathbf{D_1}$ constituirán las primeras columnas de $\mathbf{D_2}$. Las restantes $m - n$ columnas de $\mathbf{D_2}$ se completarán con columnas nulas.

- Si $m \geq 2n-1$, entonces $\mathbf{K_2}$ será la última matriz existente en el sumatorio dado en la ecuación 9.1, no existiendo, por tanto, matrices adicionales $\mathbf{K_j}$, $j > 2$. La submatriz $\mathbf{I_2}$ estará formada por las $n - 1$ columnas de $\mathbf{D_1}$, mientras que las $m - 2n + 1$ columnas restantes de $\mathbf{I_2}$ (si hubiese) se completarán con columnas nulas. En este caso, $\mathbf{D_2}$ será una submatriz con todos sus elementos nulos.

En general, se puede establecer que para un matriz $\mathbf{K_j} = (\mathbf{I_j}|\mathbf{D_j})$, con subíndice $j \geq 2$, como la mostrada en la ecuación 9.4, se verifica que si la columna $(m - n)$-ésima de $\mathbf{I_j}$ presenta un número $h > 1$ de coeficientes c_i en sus filas inferiores, entonces existirá una matriz $\mathbf{K_{j+1}} = (\mathbf{I_{j+1}}|\mathbf{D_{j+1}})$ en la que la construcción de las submatrices $\mathbf{I_{j+1}}$ y $\mathbf{D_{j+1}}$ se realiza a partir de $\mathbf{D_j}$ de la siguiente forma:

- Si $h - 1 > m - n$, entonces $\mathbf{I_{j+1}}$ estará formada por las primeras $m - n$ columnas de $\mathbf{D_j}$, mientras que las restantes $h - 1 - m + n$ columnas de $\mathbf{D_j}$ constituirán las primeras columnas de $\mathbf{D_{j+1}}$, completada con $m - h$ columnas nulas. En este caso, existirá otra matriz $\mathbf{K_{j+2}}$ construida de forma análoga a $\mathbf{K_{j+1}}$.

- Si $h - 1 \leq m - n$, entonces las primeras $h - 1$ columnas de la submatriz $\mathbf{I_{j+1}}$ serán las $h-1$ columnas no nulas de $\mathbf{D_j}$ completada con $m-n-h+1$ columnas nulas (si fuese necesario), mientras que $\mathbf{D_{j+1}}$ tendrá todas sus columnas nulas. En este caso, no existirá otra matriz $\mathbf{K_{j+2}}$ ni matrices sucesivas.

Como el número de matrices $\mathbf{K_j}$, $j \geq 2$, existentes en el sumatorio dado en la ecuación 9.1 depende del valor de la potencia n-ésima no nula del trinomio irreducible $f(x) = x^m + x^n + 1$ seleccionado, se podrán distinguir una serie de trinomios especiales que tendrán un número determinado de matrices en el sumatorio de descomposición de la matriz $\mathbf{K}$. En posteriores secciones de este

capítulo se estudiará el método *transposicional* de multiplicación aplicado a estos diferentes tipos de trinomios irreducibles.

$$
\begin{array}{c}
\begin{array}{cccccccccc}
\multicolumn{4}{c}{\text{m}} & & \multicolumn{4}{c}{\text{n}} & & \text{1}
\end{array} \\
\left(\begin{array}{cccc|c|cccc}
0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & \cdots & 0 & c_{m-1} & 0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & \cdots & c_{m-1} & c_{m-2} & 0 & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \iddots & \vdots & \vdots & \vdots & \vdots & \iddots & \vdots & \iddots & \vdots \\
0 & \cdots & c_{n+2} & c_{n+1} & 0 & 0 & \cdots & 0 & \cdots & 0 \\
c_{m-1} & \cdots & c_{n+1} & c_n & 0 & 0 & \cdots & 0 & \cdots & 0 \\
c_{m-2} & \cdots & c_n & c_{n-1} & 0 & 0 & \cdots & 0 & \cdots & c_{m-1} \\
\vdots & \iddots & \vdots & \vdots & \vdots & \vdots & \iddots & \vdots & \iddots & \vdots \\
c_{m-h} & \cdots & c_{n-h+2} & c_{n-h+1} & 0 & 0 & \cdots & c_{m-1} & \cdots & c_{m-h+1}
\end{array}\right) \\
\begin{array}{cccc}
\multicolumn{4}{c}{\text{m-n}}
\end{array}
\end{array}
\quad
\begin{array}{l}
\text{h} \\
\\
\\
\\
\\
\\
\\
\\
\\
\\
\\
\text{h-1}
\end{array}
\qquad (9.4)
$$

Una vez descompuesta la matriz $\mathbf{K}$ según la ecuación 9.1 en una suma de $\tau+1$ matrices, $\tau = \lceil \frac{m-1}{m-n} \rceil$, y utilizando el algoritmo de multiplicación dado en la sección 8.2, se tiene entonces que el producto $\delta = \alpha \cdot \chi$ en la base canónica Ω generada por un trinomio irreducible se calcula por medio de la expresión

$$\underline{\delta}^r_\Omega = \underline{\alpha}^t_\Omega \cdot \mathbf{K}(\underline{\chi}_\Lambda) = \underline{\alpha}^t_\Omega \cdot (\mathbf{K_0} + \sum_{i=1}^{\tau} \mathbf{K_i}) = \underline{\alpha}^t_\Omega \cdot \mathbf{K_0} + \cdots + \underline{\alpha}^t_\Omega \cdot \mathbf{K}_\tau \qquad (9.5)$$

donde las coordenadas del producto δ vienen dadas en forma de *sumas de productos* de las coordenadas de los operandos de entrada α y χ. A partir de la ecuación 9.5 y al igual que se hizo en la sección 8.3 para los AOPs irreducibles, se pueden establecer las expresiones generales para el cálculo del producto utilizando el método *transposicional* a través de la *agrupación* y *compartición* de subexpresiones comunes.

9.2. Método transposicional de multiplicación

La ecuación 9.5 establece que las coordenadas del producto δ se calculan realizando la adición de los vectores $\underline{\alpha}^t_\Omega \mathbf{K_i}$ $(i = 0, 1, \ldots, \tau)$, donde los vectores $\underline{\alpha}^t_\Omega \mathbf{K_0}$ y $\underline{\alpha}^t_\Omega \mathbf{K_1}$ siempre aparecen en dicha suma, independientemente del trinomio irreducible seleccionado.

La matriz $\mathbf{K_0}$ definida en la ecuación 9.2 consta de m columnas formadas por las rotaciones sucesivas del vector de coordenadas en Ω del elemento del campo χ, mientras que las matrices $\mathbf{K_1}$ y $\mathbf{K_j}$ $(j = 2, 3, \ldots, \tau)$ dadas en las

ecuaciones 9.3 y 9.4, respectivamente, estan formadas por desplazamientos sucesivos del vector de coordenadas de χ. Asimismo, se puede comprobar que los componentes de los vectores $\underline{\alpha}_\Omega^t \mathbf{K_1}$ y $\underline{\alpha}_\Omega^t \mathbf{K_j}$ $(j = 2, 3, \ldots, \tau)$ están formados por *sumas de productos* que *también aparecen* entre los componentes del vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$. Por lo tanto, es posible determinar subexpresiones comunes *agrupadas* en forma de *sumas de productos* y extraídas de los componentes del vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$, de forma que su *compartición* permita la reducción de la complejidad del multiplicador.

El método *transposicional* de multiplicación, aplicado a trinomios irreducibles, utiliza la notación introducida en la sección 8.3 para el caso de los AOPs, en la que se determinaban 1-ciclos (k) y 2-ciclos (i, j) por medio del cálculo de ciertas funciones. Estos ciclos (k) e (i, j) representaban términos producto $x_k = (a_k c_k)$ y sumas de términos producto $x_{ij} = (a_i c_j + a_j c_i)$, respectivamente, cuya adición permitía calcular las coordenadas del producto de dos elementos del campo. En el caso de los trinomios irreducibles, estos ciclos y las funciones que los determinan se deducen completamente del vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$.

A continuación se introducen las funciones que proporcionan los ciclos correspondientes a determinados valores de los parámetros i y m, y que agrupamos en función de su dependencia o no de m y en función de los valores *pares* o *impares* de i de la siguiente forma:

- Función independiente del parámetro m y válida para valores de i *pares*:

$$\mathbf{EC_{i0}} = (h, l) \quad h = 0, 1, \ldots, \frac{i}{2} - 1;\ l = (i - h - 1);\ h \geq 0 \tag{9.6}$$

- Función independiente de m y válida para valores de i *impares*:

$$\mathbf{OC_{i0}} = \begin{cases} (k) & k = \frac{i-1}{2} \\ (h, l) & h = 0, 1, \ldots, \frac{i-1}{2} - 1;\ l = (i - h - 1);\ h \geq 0 \end{cases} \tag{9.7}$$

- Funciones dependientes de m y válidas para valores de i *pares* o *impares*:

$$\mathbf{EC_{i1}} = \begin{cases} (k) & k = \lceil \frac{m}{2} \rceil + \lfloor \frac{i}{2} \rfloor \\ (p, r) & \begin{cases} j = 1, 2, \ldots, (\lceil \frac{m}{2} \rceil + \lfloor \frac{i}{2} \rfloor) - (i + 1) \\ p = (i + j);\ \ r = (m - j) \end{cases} \end{cases} \tag{9.8}$$

$$\mathbf{OC_{i1}} = (p, r) \quad \begin{cases} j = 1, 2, \ldots, (\lceil \frac{m}{2} \rceil + \lfloor \frac{i+1}{2} \rfloor) - (i + 1) \\ p = (i + j);\ \ r = (m - j) \end{cases} \tag{9.9}$$

De estas definiciones, se observa que las funciones $\mathbf{EC_{i1}}$ y $\mathbf{OC_{i1}}$ calculan los ciclos en función del parámetro m, donde m puede tomar valores *pares* o *impares*, a diferencia de las funciones $\mathbf{EC_i^m}$ y $\mathbf{OC_i^m}$ dadas en las ecuaciones

8.14 y 8.15, respectivamente, que únicamente eran válidas para valores de m *pares* (donde los AOPs son irreducibles). También se observa que las funciones $\mathbf{EC_{i0}}$ y $\mathbf{OC_{i0}}$ están incluidas en las ecuaciones 8.14 y 8.15, respectivamente. En estas ecuaciones de $\mathbf{EC_i^m}$ y $\mathbf{OC_i^m}$, también se pueden identificar las funciones $\mathbf{EC_{i1}}$ y $\mathbf{OC_{i1}}$, respectivamente, con la modificación indicada de ser válidas para valores *pares* e *impares* de m. Por último, se puede comprobar que la función $\mathbf{EC_{i0}}$, particularizada para $i = m$, coincide con la función $\mathbf{MC^m}$ dada en la ecuación 8.16 para un AOP.

Las funciones $\mathbf{EC_{i0}}$, $\mathbf{OC_{i0}}$, $\mathbf{EC_{i1}}$ y $\mathbf{OC_{i1}}$ que calculan los 1-ciclos y 2-ciclos para valores dados de i y m, son las que determinan la *agrupación* de subexpresiones característica del método *transposicional* por medio de la definición de las funciones $\mathbf{E_{i0}}$, $\mathbf{O_{i0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ que realizan la suma de los términos x_k y x_{ij} representados por los ciclos dados por dichas funciones $\mathbf{EC_{i0}}$, $\mathbf{OC_{i0}}$, $\mathbf{EC_{i1}}$ y $\mathbf{OC_{i1}}$, respectivamente. Asimismo, se cumple que las subexpresiones dadas por las funciones $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ constituyen los componentes de *todos* los vectores $\underline{\alpha}_\Omega^t\mathbf{K_i}$ $(i = 1, 2, 3, \ldots, \tau)$, por lo que su *compartición* permite la reducción de la complejidad del multiplicador, como se verá posteriormente. Es importante señalar que para multiplicaciones realizadas sobre campos de Galois generados por determinados tipos de *trinomios* irreducibles, la compartición de subexpresiones es mayor que la que se obtiene en las multiplicaciones sobre campos generados por AOPs irreducibles.

Se ha mencionado anteriormente que en el sumatorio dado en la ecuación 9.5 para el cálculo de las coordenadas del producto siempre aparecen, al menos, los vectores $\underline{\alpha}_\Omega^t\mathbf{K_0}$ y $\underline{\alpha}_\Omega^t\mathbf{K_1}$, independientemente del trinomio irreducible seleccionado. Además, tanto la composición del vector $\underline{\alpha}_\Omega^t\mathbf{K_1}$ como el número y composición de los vectores $\underline{\alpha}_\Omega^t\mathbf{K_j}$ $(j = 2, 3, \ldots, \tau$, con $\tau = \lceil \frac{m-1}{m-n} \rceil)$ que aparecen en dicho sumatorio dependerá del *tipo* de *trinomio* irreducible utilizado, como se verá posteriormente. Sin embargo, la composición del vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$ es *común* a cualquier trinomio seleccionado.

Utilizando las funciones $\mathbf{E_{i0}}$, $\mathbf{O_{i0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ anteriormente definidas, se pueden dar las siguientes expresiones que permiten el cálculo de los componentes i-ésimos $(i = 0, 1, \ldots, m-1)$ del vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$ y en las que se distinguen los casos para valores del parámetro m *pares* e *impares*.

- Valores de m *pares*:

$$(\underline{\alpha}_\Omega^t\mathbf{K_0})_i = \begin{cases} \mathbf{O_{(m-i)0}} + \mathbf{E_{(m-i-1)1}} & i \ impar \\ \mathbf{E_{(m-i)0}} + \begin{cases} \mathbf{O_{(m-i-1)1}} & i \ par, \ i \neq 0 \\ - & i = 0 \end{cases} \end{cases} \tag{9.10}$$

- Valores de m *impares*:

$$(\underline{\alpha}_\Omega^t\mathbf{K_0})_i = \begin{cases} \mathbf{E_{(m-i)0}} + \mathbf{E_{(m-i-1)1}} & i \ impar \\ \mathbf{O_{(m-i)0}} + \begin{cases} \mathbf{O_{(m-i-1)1}} & i \ par, \ i \neq 0 \\ - & i = 0 \end{cases} \end{cases} \tag{9.11}$$

Se vio anteriormente que la matriz $\mathbf{K_0}$ definida en la ecuación 9.2 estaba formada por las rotaciones sucesivas del vector de coordenadas en Ω del elemento del campo χ, mientras que las matrices $\mathbf{K_1}$ y $\mathbf{K_j}$ $(j = 2, 3, \ldots, \tau)$ dadas en las ecuaciones 9.3 y 9.4, respectivamente, eran los desplazamientos sucesivos del vector de coordenadas de χ. Por este motivo, los vectores $\underline{\alpha}^t_\Omega \mathbf{K_1}$ y $\underline{\alpha}^t_\Omega \mathbf{K_j}$ $(j = 2, 3, \ldots, \tau)$ están formados *exclusivamente* por sumas de funciones $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ que se encuentran entre los componentes del vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$.

A continuación se dan las expresiones para el cálculo de los componentes del vector $\underline{\alpha}^t_\Omega \mathbf{K_1}$, en las que también se distinguen los casos para valores *pares* e *impares* de m y donde se define el parámetro Δ como $\Delta = m - n$.

- Valores de m *pares*:

$$(\underline{\alpha}^t_\Omega \mathbf{K_1})_i = \begin{cases} \left.\begin{array}{ll} \mathbf{E}_{((\Delta+m)-(i+1))1} & i > \Delta \\ - & i = \Delta \\ \mathbf{E}_{(\Delta-(i+1))1} & i < \Delta \end{array}\right\} \begin{array}{l}(i\ par\ y\ \Delta\ impar)\ o \\ (i\ impar\ y\ \Delta\ par)\end{array} \\ \\ \left.\begin{array}{ll} \mathbf{O}_{((\Delta+m)-(i+1))1} & i > \Delta \\ - & i = \Delta \\ \mathbf{O}_{(\Delta-(i+1))1} & i < \Delta \end{array}\right\} \begin{array}{l}(i\ impar\ y\ \Delta\ impar)\ o \\ (i\ par\ y\ \Delta\ par)\end{array} \end{cases} \tag{9.12}$$

- Valores de m *impares*:

$$(\underline{\alpha}^t_\Omega \mathbf{K_1})_i = \begin{cases} \left.\begin{array}{ll} \mathbf{E}_{((\Delta+m)-(i+1))1} & i > \Delta \\ - & i = \Delta \\ \mathbf{O}_{(\Delta-(i+1))1} & i < \Delta \end{array}\right\} \begin{array}{l}(i\ par\ y\ \Delta\ impar)\ o \\ (i\ impar\ y\ \Delta\ par)\end{array} \\ \\ \left.\begin{array}{ll} \mathbf{O}_{((\Delta+m)-(i+1))1} & i > \Delta \\ - & i = \Delta \\ \mathbf{E}_{(\Delta-(i+1))1} & i < \Delta \end{array}\right\} \begin{array}{l}(i\ impar\ y\ \Delta\ impar)\ o \\ (i\ par\ y\ \Delta\ par)\end{array} \end{cases} \tag{9.13}$$

En estas ecuaciones 9.12 y 9.13 se tiene que $i = 0, 1, \ldots, m-1$. También se observa que aparecen únicamente las funciones $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$, como ya se mencionó anteriormente.

Se pueden establecer también las expresiones que permiten el cálculo recursivo de los componentes de los vectores $\underline{\alpha}^t_\Omega \mathbf{K_j}$ $(j = 2, 3, \ldots, \tau = \lceil \frac{m-1}{\Delta} \rceil)$, independientemente del valor del parámetro m, de la siguiente forma:

$$(\underline{\alpha}^t_\Omega \mathbf{K_j})_i = \begin{cases} - & i \in \{\Delta, \Delta+1, \ldots, 2\Delta\}\ mod\ \mathrm{m} \\ (\underline{\alpha}^t_\Omega \mathbf{K_{j-1}})_{(i-\Delta)\ mod\ m} & i \notin \{\Delta, \Delta+1, \ldots, 2\Delta\}\ mod\ \mathrm{m} \end{cases} \tag{9.14}$$

donde $i = 0, 1, \ldots, m-1$ y donde el cálculo de los vectores se realiza a partir del conocimiento de $\underline{\alpha}^t_\Omega \mathbf{K_1}$. También se tiene que la recursión se detendrá cuando se obtenga un vector $\underline{\alpha}^t_\Omega \mathbf{K_j}$ con todos sus componentes nulos (representado

por el símbolo − en las ecuaciones 9.10 a 9.14) el cual, lógicamente, no se considerará. Se puede comprobar que el último vector no nulo es el $\underline{\alpha}_\Omega^t \mathbf{K}_\tau$, con subíndice $\tau = \lceil \frac{m-1}{\Delta} \rceil$, como se vio en la ecuación 9.1.

A partir de las ecuaciones 9.10 a 9.14, se pueden establecer las expresiones generales de cálculo de las coordenadas del producto δ en base canónica Ω de dos elementos α y χ, $\delta = \alpha \cdot \chi$, usando el método *transposicional* para *trinomios* irreducibles generadores del campo $GF(2^m)$. Estas expresiones se determinan a partir de la ecuación 9.5 de la forma

$$d_{\Omega_i} = (\underline{\alpha}_\Omega^t \mathbf{K_0})_{m-1-i} + (\underline{\alpha}_\Omega^t \mathbf{K_1})_{m-1-i} + \sum_{j=2}^{\tau} (\underline{\alpha}_\Omega^t \mathbf{K_j})_{m-1-i} \tag{9.15}$$

donde $i = 0, 1, \ldots, m-1$ y donde los términos d_{Ω_i} son las coordenadas del producto δ con respecto de la base canónica Ω.

El número y la composición de los vectores que aparecen en el sumatorio dado en la ecuación 9.15 dependerá del *tipo* de *trinomio* utilizado. Por este motivo, a continuación se realiza el estudio del método *transposicional* de multiplicación en base canónica para diferentes trinomios irreducibles, estableciéndose las ecuaciones particulares que el método determina para cada uno de esos trinomios. También se realiza el análisis de complejidad teórico de los multiplicadores construidos con dicho método y se muestran los resultados experimentales obtenidos de las implementaciones reconfigurables paralelas sobre dispositivos FPGA.

9.3. Trinomios irreducibles $f(x) = x^m + x^{m-1} + 1$

Cuando el trinomio irreducible generador del campo finito es de la forma $f(x) = x^m + x^{m-1} + 1$ se tiene que el parámetro $\tau = \lceil \frac{m-1}{\Delta} \rceil = \lceil \frac{m-1}{m-n} \rceil = m - 1$ ($\Delta = 1$), por lo que la descomposición de la matriz $\mathbf{K}$ dada en la ecuación 9.1 está formada por la suma $\mathbf{K_0} + \mathbf{K_1} + \cdots + \mathbf{K_{m-1}}$.

Este hecho también se puede deducir de la estructura de la matriz $\mathbf{K_1}$ dada en la ecuación 9.3 (según se vio al comienzo de la sección 9.1) de la siguiente forma. Para este tipo de trinomios, la columna nula se encuentra en la posición $m-1$ de $\mathbf{K_1}$, por lo que $\mathbf{I_1}$ consta de *una* columna con $m-1$ coeficientes no nulos mientras que $\mathbf{D_1}$ tiene $m-2$ columnas no nulas. Por este motivo, existirá una matriz $\mathbf{K_2} = (\mathbf{I_2}|\mathbf{D_2})$, donde $\mathbf{I_2}$ estará formada por la primera columna de $\mathbf{D_1}$ y donde las primeras columnas de $\mathbf{D_2}$ son las $m-3$ restantes columnas de $\mathbf{D_1}$, junto con una última columna nula. Asimismo, si se cumple que $m > 3$, existirá una tercera matriz $\mathbf{K_3}$ que se construirá de forma similar a $\mathbf{K_2}$ (se observa, por tanto, que la descomposición de $\mathbf{K}$ para los trinomios irreducibles $f(x) = x^2 + x + 1$ y $f(x) = x^3 + x^2 + 1$ está formada por 2 y 3 matrices, respectivamente). Continuando de esta forma y utilizando las consideraciones

hechas para las matrices $\mathbf{K_j}$ dadas en la ecuación 9.4, se obtiene finalmente que el número total de matrices adicionales a $\mathbf{K_0}$ en el sumatorio dado en la ecuación 9.1 es de $m-1$. Por lo tanto, el trinomio $f(x) = x^m + x^{m-1} + 1$ es el que presenta la mayor descomposición de la matriz $\mathbf{K}$ de entre todos los trinomios irreducibles.

A partir de las expresiones generales de las coordenadas d_{Ω_i} del producto δ dadas en la ecuación 9.15 y usando las ecuaciones 9.10 a 9.14 que calculan las coordenadas de los vectores $\underline{\alpha}_\Omega^t\mathbf{K_0}$, $\underline{\alpha}_\Omega^t\mathbf{K_1}$ y $\underline{\alpha}_\Omega^t\mathbf{K_j}$ $(j = 2, 3, \ldots, \tau)$ utilizando exclusivamente las funciones $\mathbf{E_{i0}}$, $\mathbf{O_{i0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$, se dan a continuación las ecuaciones que permiten el cálculo de las coordenadas d_{Ω_i} del producto a partir de dichas funciones utilizando el método *transposicional* para trinomios irreducibles y donde, al igual que en casos anteriores, se hace distinción entre valores del parámetro m *pares* e *impares*.

Las coordenadas del producto δ para valores de m *pares* vienen dadas por las expresiones

$$d_{\Omega_i} = \begin{cases} (\mathbf{E_{(i+1)0}} + \mathbf{O_{i1}}) + \displaystyle\sum_{\substack{j=i+\Delta \\ (par)}}^{m-2} \mathbf{E_{j1}} + \sum_{\substack{j=i+\Delta \\ (impar)}}^{m-2} \mathbf{O_{j1}} & i \ impar \\ (\mathbf{O_{(i+1)0}} + \mathbf{E_{i1}}) + \begin{cases} \displaystyle\sum_{\substack{j=i+\Delta \\ (par)}}^{m-2} \mathbf{E_{j1}} + \sum_{\substack{j=i+\Delta \\ (impar)}}^{m-2} \mathbf{O_{j1}} & i \, par \\ - & i = m-2 \end{cases} & \\ (\mathbf{E_{m0}}) + \displaystyle\sum_{\substack{j=0 \\ (par)}}^{m-2} \mathbf{E_{j1}} + \sum_{\substack{j=0 \\ (impar)}}^{m-2} \mathbf{O_{j1}} & i = m-1 \end{cases} \tag{9.16}$$

mientras que las coordenadas para valores *impares* de m son

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) + \displaystyle\sum_{\substack{j=i+\Delta \\ (par)}}^{m-2} \mathbf{O_{j1}} + \sum_{\substack{j=i+\Delta \\ (impar)}}^{m-2} \mathbf{E_{j1}} & i \ par \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \begin{cases} \displaystyle\sum_{\substack{j=i+\Delta \\ (par)}}^{m-2} \mathbf{O_{j1}} + \sum_{\substack{j=i+\Delta \\ (impar)}}^{m-2} \mathbf{E_{j1}} & i \, impar \\ - & i = m-2 \end{cases} & \\ (\mathbf{O_{m0}}) + \displaystyle\sum_{\substack{j=0 \\ (par)}}^{m-2} \mathbf{O_{j1}} + \sum_{\substack{j=0 \\ (impar)}}^{m-2} \mathbf{E_{j1}} & i = m-1 \end{cases} \tag{9.17}$$

donde $i = 0, 1, \ldots, m-1$ y donde los términos entre paréntesis son los proporcionados por el vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$ (en los que aparecen las funciones $\mathbf{E_{i0}}$, $\mathbf{O_{i0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$), mientras que los sumatorios corresponden a términos suministrados por los vectores $\underline{\alpha}^t_\Omega \mathbf{K_1}$ y $\underline{\alpha}^t_\Omega \mathbf{K_j}$ $(j = 2, 3, \ldots, m-1)$ en los que únicamente aparecen las funciones $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$. La *agrupación* y *compartición* de estos términos permitirá (como se verá en un análisis posterior) reducir la complejidad del multiplicador construido utilizando este método transposicional.

A continuación se presenta un ejemplo de multiplicación sobre el campo finito $GF(2^6)$ en el que se muestran las agrupaciones y comparticiones a realizar para la reducción de la complejidad final del multiplicador.

9.3.1. Ejemplo de multiplicación sobre $GF(2^6)$

El producto δ de dos elementos α y χ pertenecientes al campo $GF(2^6)$ generado por el trinomio irreducible $f(x) = x^6 + x^5 + 1$ se puede calcular utilizando las expresiones anteriormente definidas. Para este trinomio, se tiene que el parámetro $\Delta = 6 - 5 = 1$, por lo que la descomposición de la matriz $\mathbf{K}$ se convierte en este caso en la suma de $1 + \tau = 1 + \lceil \frac{6-1}{6-5} \rceil = 6$ matrices dadas de la siguiente forma:

$$\begin{aligned}
\mathbf{K} = \mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2} + \mathbf{K_3} + \mathbf{K_4} + \mathbf{K_5} = \\
\begin{pmatrix} c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ c_4 & c_3 & c_2 & c_1 & c_0 & c_5 \\ c_3 & c_2 & c_1 & c_0 & c_5 & c_4 \\ c_2 & c_1 & c_0 & c_5 & c_4 & c_3 \\ c_1 & c_0 & c_5 & c_4 & c_3 & c_2 \\ c_0 & c_5 & c_4 & c_3 & c_2 & c_1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ c_5 & 0 & 0 & 0 & 0 & 0 \\ c_4 & 0 & 0 & 0 & 0 & c_5 \\ c_3 & 0 & 0 & 0 & c_5 & c_4 \\ c_2 & 0 & 0 & c_5 & c_4 & c_3 \\ c_1 & 0 & c_5 & c_4 & c_3 & c_2 \end{pmatrix} + \\
\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ c_5 & 0 & 0 & 0 & 0 & 0 \\ c_4 & 0 & 0 & 0 & 0 & c_5 \\ c_3 & 0 & 0 & 0 & c_5 & c_4 \\ c_2 & 0 & 0 & c_5 & c_4 & c_3 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ c_5 & 0 & 0 & 0 & 0 & 0 \\ c_4 & 0 & 0 & 0 & 0 & c_5 \\ c_3 & 0 & 0 & 0 & c_5 & c_4 \end{pmatrix} + \\
\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ c_5 & 0 & 0 & 0 & 0 & 0 \\ c_4 & 0 & 0 & 0 & 0 & c_5 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ c_5 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}
\end{aligned} \tag{9.18}$$

donde los términos c_i son las coordenadas de χ en la base canónica y donde las matrices $\mathbf{K_0}$, $\mathbf{K_1}$ y $\mathbf{K_j}$ $(j = 2, \ldots, 5)$ se construyen a partir de las ecuaciones 9.2, 9.3 y 9.4, respectivamente.

Las ecuaciones 9.5 y 9.18 permiten calcular las coordenadas d_i del producto δ en la base canónica utilizando el algoritmo de multiplicación dado en la sección 8.2. Sin embargo, de las sumas de productos obtenidas con este algoritmo no se deduce ni la *agrupación* ni la *compartición* de términos que produce el método *transposicional* y que permiten la reducción de la complejidad del multiplicador. Los 1-ciclos y 2-ciclos obtenidos con este método se calculan a partir de las ecuaciones 9.6 a 9.9 y están determinados, para nuestro ejemplo de $GF(2^6)$, por las funciones $\mathbf{OC_{10}} = (0)$, $\mathbf{EC_{20}} = (0,1)$, $\mathbf{OC_{30}} = (1)(0,2)$, $\mathbf{EC_{40}} = (0,3)(1,2)$, $\mathbf{OC_{50}} = (2)(0,4)(1,3)$ y $\mathbf{EC_{60}} = (0,5)(1,4)(2,3)$, y por las funciones $\mathbf{EC_{01}} = (3)(1,5)(2,4)$, $\mathbf{OC_{11}} = (2,5)(3,4)$, $\mathbf{EC_{21}} = (4)(3,5)$, $\mathbf{OC_{31}} = (4,5)$ y $\mathbf{EC_{41}} = (5)$.

Las sumas de los términos representados por estos ciclos vienen dadas por las funciones $\mathbf{O_{10}}$, $\mathbf{E_{20}}$, $\mathbf{O_{30}}$, $\mathbf{E_{40}}$, $\mathbf{O_{50}}$, $\mathbf{E_{60}}$ y $\mathbf{E_{01}}$, $\mathbf{O_{11}}$, $\mathbf{E_{21}}$, $\mathbf{O_{31}}$, $\mathbf{E_{41}}$, respectivamente, obteniéndose

$$\begin{array}{lcll}
\mathbf{O_{10}} & = & x_0 & = a_0c_0 \\
\mathbf{E_{20}} & = & x_{01} & = (a_0c_1 + a_1c_0) \\
\mathbf{O_{30}} & = & x_1 + x_{02} & = a_1c_1 + (a_0c_2 + a_2c_0) \\
\mathbf{E_{40}} & = & x_{03} + x_{12} & = (a_0c_3 + a_3c_0) + (a_1c_2 + a_2c_1) \\
\mathbf{O_{50}} & = & x_2 + x_{04} + x_{13} & = a_2c_2 + (a_0c_4 + a_4c_0) + (a_1c_3 + a_3c_1) \\
\mathbf{E_{60}} & = & x_{05} + x_{14} + x_{23} & = (a_0c_5 + a_5c_0) + (a_1c_4 + a_4c_1) + (a_2c_3 + a_3c_2)
\end{array} \tag{9.19}$$

$$\begin{array}{lcll}
\mathbf{E_{01}} & = & x_3 + x_{15} + x_{24} & = a_3c_3 + (a_1c_5 + a_5c_1) + (a_2c_4 + a_4c_2) \\
\mathbf{O_{11}} & = & x_{25} + x_{34} & = (a_2c_5 + a_5c_2) + (a_3c_4 + a_4c_3) \\
\mathbf{E_{21}} & = & x_4 + x_{35} & = a_4c_4 + (a_3c_5 + a_5c_3) \\
\mathbf{O_{31}} & = & x_{45} & = (a_4c_5 + a_5c_4) \\
\mathbf{E_{41}} & = & x_5 & = a_5c_5
\end{array} \tag{9.20}$$

donde los términos a_i y c_i son las coordenadas en la base canónica de los elementos α y χ de $GF(2^6)$, respectivamente.

Se pueden también determinar los componentes de los vectores $\underline{\alpha}^t_\Omega\mathbf{K_0}$, $\underline{\alpha}^t_\Omega\mathbf{K_1}$ y $\underline{\alpha}^t_\Omega\mathbf{K_j}$ $(j = 2,3,4,5)$ utilizando las ecuaciones generales 9.10, 9.12 y 9.14, respectivamente, para valores *pares* de m, obteniéndose:

$$\begin{array}{l}
\underline{\alpha}^t_\Omega\mathbf{K_0} = (\mathbf{E_{60}}, \mathbf{O_{50}} + \mathbf{E_{41}}, \mathbf{E_{40}} + \mathbf{O_{31}}, \mathbf{O_{30}} + \mathbf{E_{21}}, \mathbf{E_{20}} + \mathbf{O_{11}}, \mathbf{O_{10}} + \mathbf{E_{01}}) \\
\underline{\alpha}^t_\Omega\mathbf{K_1} = (\mathbf{E_{01}}, -, \mathbf{E_{41}}, \mathbf{O_{31}}, \mathbf{E_{21}}, \mathbf{O_{11}}) \\
\underline{\alpha}^t_\Omega\mathbf{K_2} = (\mathbf{O_{11}}, -, -, \mathbf{E_{41}}, \mathbf{O_{31}}, \mathbf{E_{21}}) \\
\underline{\alpha}^t_\Omega\mathbf{K_3} = (\mathbf{E_{21}}, -, -, -, \mathbf{E_{41}}, \mathbf{O_{31}}) \\
\underline{\alpha}^t_\Omega\mathbf{K_4} = (\mathbf{O_{31}}, -, -, -, -, \mathbf{E_{41}}) \\
\underline{\alpha}^t_\Omega\mathbf{K_5} = (\mathbf{E_{41}}, -, -, -, -, -)
\end{array} \tag{9.21}$$

donde el símbolo $-$ representa el componente nulo. Con los vectores dados en 9.21 y utilizando la ecuación general 9.15, se obtienen los componentes del producto como sumas de términos agrupados por las funciones $\mathbf{E_{i(0,1)}}$ y

$\mathbf{O_{i(0,1)}}$ dadas anteriormente. Estos componentes se pueden calcular directamente utilizando la ecuación 9.16 que constituye el método *transposicional* de multiplicación para trinomios irreducibles del tipo $f(x) = x^m + x^{m-1} + 1$.

En la tabla 9.1 se muestran las coordenadas del producto δ para el trinomio $f(x) = x^6 + x^5 + 1$, en donde se especifican los componentes de los vectores $\underline{\alpha}^t_\Omega \mathbf{K_0}$, $\underline{\alpha}^t_\Omega \mathbf{K_1}$ y $\underline{\alpha}^t_\Omega \mathbf{K_j}$ $(j = 2, 3, 4, 5)$, y donde la coordenada d_i estará formada por la suma de las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ existentes en la fila i-ésima.

En la tabla 9.1 también se muestra la propiedad de *compartición* que presenta la multiplicación con este tipo de trinomio y que permite la reducción de la complejidad del multiplicador. Se observa que comenzando con la coordenada $d_{m-2} = d_4$, el término $\mathbf{E_{41}}$ perteneciente al vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$ aparece en la suma de términos de d_3, cuya subexpresión $(\mathbf{O_{31}} + \mathbf{E_{41}})$ también aparece en la suma correspondiente a d_2. Lo mismo sucede con la subexpresión $(\mathbf{E_{21}} + \mathbf{O_{31}} + \mathbf{E_{41}})$ que aparece en d_2 y en d_1, así como con $(\mathbf{O_{11}} + \mathbf{E_{21}} + \mathbf{O_{31}} + \mathbf{E_{41}})$ que pertenece tanto a d_1 como a d_0. Por último, la subexpresión $(\mathbf{E_{01}} + \mathbf{O_{11}} + \mathbf{E_{21}} + \mathbf{O_{31}} + \mathbf{E_{41}})$ perteneciente a d_0 se repite también en la coordenada d_5. Todo esto implica que los términos mencionados se pueden construir *una sola vez* y pueden ser *reutilizados* en los distintos sumatorios que determinan las coordenadas del producto, sin necesidad de volver a ser construidos. Esta *compartición* de subexpresiones es la que permite reducir la complejidad *espacial* del multiplicador, como se verá en un análisis posterior. Teniendo esto en cuenta, se obtiene finalmente que el número de puertas necesarias para la construcción del multiplicador sobre $GF(2^6)$ es de 36 puertas AND y de 35 puertas XOR.

	$\underline{\alpha}^t_\Omega \mathbf{K_0}$		$\underline{\alpha}^t_\Omega \mathbf{K_1}$	$\underline{\alpha}^t_\Omega \mathbf{K_2}$	$\underline{\alpha}^t_\Omega \mathbf{K_3}$	$\underline{\alpha}^t_\Omega \mathbf{K_4}$	$\underline{\alpha}^t_\Omega \mathbf{K_5}$
d_0	$\mathbf{O_{10}}$	$\mathbf{E_{01}}$	$\mathbf{O_{11}}$	$\mathbf{E_{21}}$	$\mathbf{O_{31}}$	$\mathbf{E_{41}}$	-
d_1	$\mathbf{E_{20}}$	$\mathbf{O_{11}}$	$\mathbf{E_{21}}$	$\mathbf{O_{31}}$	$\mathbf{E_{41}}$	-	-
d_2	$\mathbf{O_{30}}$	$\mathbf{E_{21}}$	$\mathbf{O_{31}}$	$\mathbf{E_{41}}$	-	-	-
d_3	$\mathbf{E_{40}}$	$\mathbf{O_{31}}$	$\mathbf{E_{41}}$	-	-	-	-
d_4	$\mathbf{O_{50}}$	$\mathbf{E_{41}}$	-	-	-	-	-
d_5	$\mathbf{E_{60}}$	-	$\mathbf{E_{01}}$	$\mathbf{O_{11}}$	$\mathbf{E_{21}}$	$\mathbf{O_{31}}$	$\mathbf{E_{41}}$

Tabla 9.1: Coordenadas d_i del producto para la multiplicación sobre $GF(2^6)$.

Esta forma de construcción de las coordenadas del producto también permite la reducción de la complejidad *temporal* del multiplicador por medio de la utilización de un árbol *híbrido* de puertas XOR en el que los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ se construyan usando un árbol *binario* de puertas XOR, mientras que la suma de dichos términos se realice con un árbol *lineal* de puertas XOR de la forma indicada en la tabla 9.1.

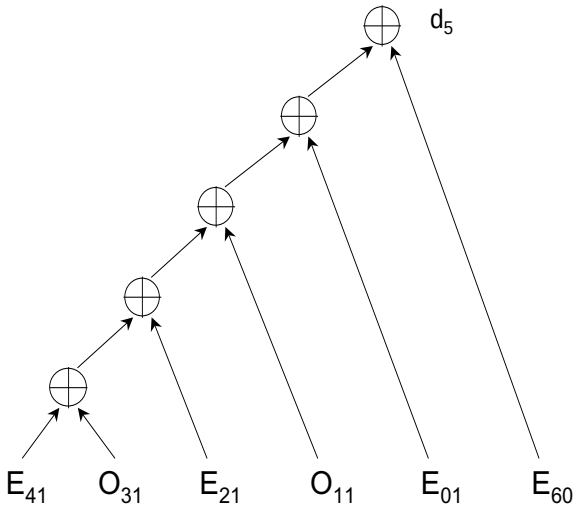


Figura 9.1: Árbol lineal de puertas XOR que realiza la suma de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ para la construcción de la coordenada d_5 del ejemplo.

En la figura 9.1 se muestra este tipo de construcción para la coordenada d_5 del ejemplo, que a su vez es la que presenta una mayor complejidad tanto *espacial* como *temporal*. El mayor coste en número de puertas AND y XOR utilizadas para d_5 (en comparación con las empleadas por la coordenada d_0) es debido a que el término $\mathbf{E_{60}}$ está formado por la suma de 6 términos producto, mientras que el término $\mathbf{O_{10}}$ está formado únicamente por un término producto (véase la ecuación 9.19). Asimismo, esta coordenada d_5 es, junto con d_0, la que tiene un mayor retardo debido a que presenta la mayor suma de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ de todo el multiplicador. El retardo viene dado por la suma de los cinco términos ($\mathbf{E_{01} + O_{11} + E_{21} + O_{31} + E_{41}}$) comunes a d_5 y d_0, y su suma con el término $\mathbf{E_{60}}$. Teniendo en cuenta que $\mathbf{O_{31}}$ es la suma de dos términos producto (ecuación 9.20), se obtiene que el retardo total del multiplicador es de 6 retardos de puertas XOR y de 1 retardo de puerta AND ($T_{AND} + 6T_{XOR}$). También se observa que en este ejemplo existen dos *caminos críticos* dados por la conexión de los términos $\mathbf{O_{31}}$ y $\mathbf{E_{21}}$ con la salida de d_5 y d_0, por lo que el retardo no viene determinado por la *diferencia* entre d_5 y d_0 ($\mathbf{E_{60}}$ y $\mathbf{O_{10}}$, respectivamente) sino por el término común ($\mathbf{E_{01} + O_{11} + E_{21} + O_{31} + E_{41}}$).

9.3.2. Análisis teórico de complejidad

Las ecuaciones 9.16 (para m *par*) y 9.17 (para m *impar*) proporcionan las coordenadas del producto $\delta = \alpha \cdot \chi$ en forma de sumatorios de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$. Para cualquier valor de m, se verifica que dichos sumatorios son de la forma dada en la tabla 9.1 del ejemplo anterior, por lo que se puede utilizar

el criterio de construcción visto en dicho ejemplo. Esta forma de construcción determina que, comenzando con el término $\mathbf{E_{(m-2)1}}$ perteneciente al sumatorio de la coordenada d_{m-2}, $\mathbf{E_{(m-2)1}}$ se *agrupe* con el término $\mathbf{O_{(m-3)1}}$ perteneciente a d_{m-3}. Si a este grupo se le denomina $\mathbf{G_1} = \mathbf{E_{(m-2)1}} + \mathbf{O_{(m-3)1}}$, se tiene que la coordenada d_{m-4} incluirá en su sumatorio al grupo $\mathbf{G_2} = \mathbf{G_1} + \mathbf{E_{(m-4)1}}$, la coordenada d_{m-5} incluirá al grupo $\mathbf{G_3} = \mathbf{G_2} + \mathbf{O_{(m-5)1}}$ y así sucesivamente. El último grupo formado será el $\mathbf{G_{m-2}} = \mathbf{G_{m-3}} + \mathbf{E_{01}}$ (para valores de m *pares*) o el $\mathbf{G_{m-2}} = \mathbf{G_{m-3}} + \mathbf{O_{01}}$ (si m *impar*), que pertenecerá al sumatorio de las coordenadas d_0 y d_{m-1}. Por lo tanto, se observa que los $m-2$ grupos $\mathbf{G_i}$ están formados exclusivamente por las sumas sucesivas de términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ determinados por el vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$.

En la figura 9.2 se muestran los $m-2$ grupos $\mathbf{G_i}$. Se observa que la estructura obtenida es la indicada en el ejemplo anterior de un árbol *híbrido* en donde los términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ se construyen usando un árbol *binario* de puertas XOR, mientras que la suma de dichos términos se realiza siguiendo una estructura de árbol *lineal* de puertas XOR. Este árbol lineal es necesario para poder realizar la construcción de los grupos $\mathbf{G_i}$ a partir de los $\mathbf{G_{i-1}}$, como se ha descrito anteriormente, y de esta forma poder realizar la *compartición* de expresiones que permite la reducción de la complejidad.

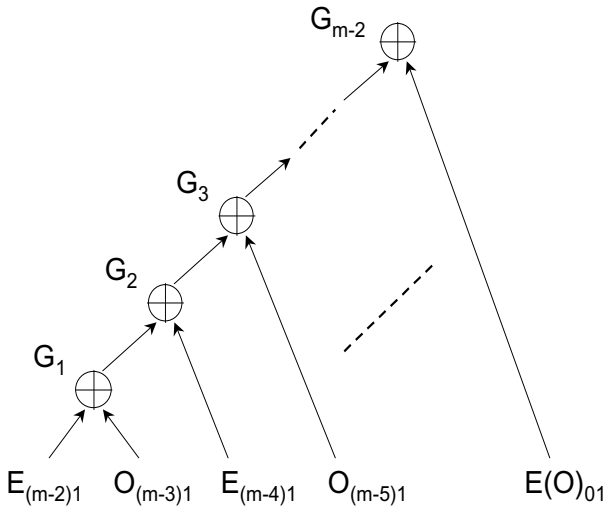


Figura 9.2: Estructura en árbol de puertas XOR para la construcción de grupos $\mathbf{G_i}$.

Una vez formados los grupos $\mathbf{G_i}$, las coordenadas del producto se obtendrán realizando la suma del grupo perteneciente a cada coordenada con su término $\mathbf{O_{i0}}$ o $\mathbf{E_{i0}}$ correspondiente (según lo establecido por las ecuaciones 9.16 o 9.17). La única coordenada que no presenta un grupo $\mathbf{G_i}$ asociado es la coordenada d_{m-2}, que incluye el término origen $\mathbf{E_{(m-2)1}}$ (véase la tabla 9.1). Por lo tanto, para valores *pares* de m, se obtienen las siguientes nuevas expresiones de las

coordenadas del producto d_i $(i = 0, 1, \ldots, m-1)$

$$d_i = \begin{cases} \mathbf{E_{m0}} + \mathbf{G_{m-2}} & i = m-1 \\ \mathbf{O_{(m-1)0}} + \mathbf{E_{(m-2)1}} & i = m-2 \\ \mathbf{G_{m-2-i}} + \begin{cases} \mathbf{O_{(i+1)0}} & i \ par \\ \mathbf{E_{(i+1)0}} & i \ impar \end{cases} \end{cases} \tag{9.22}$$

mientras que para valores *impares* de m se tiene

$$d_i = \begin{cases} \mathbf{O_{m0}} + \mathbf{G_{m-2}} & i = m-1 \\ \mathbf{E_{(m-1)0}} + \mathbf{E_{(m-2)1}} & i = m-2 \\ \mathbf{G_{m-2-i}} + \begin{cases} \mathbf{O_{(i+1)0}} & i \ par \\ \mathbf{E_{(i+1)0}} & i \ impar \end{cases} \end{cases} \tag{9.23}$$

De estas ecuaciones 9.22 y 9.23 y de la estructura de construcción mostrada en la figura 9.2 se deduce fácilmente la arquitectura paralela del multiplicador.

Para poder determinar la complejidad teórica del multiplicador propuesto, comenzamos viendo las complejidades de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$. La construcción de estas funciones se realiza siguiendo la estructura de un árbol *binario* de puertas XOR con un último nivel de puertas AND correspondiente a los productos $a_i c_j$ de coordenadas de los elementos α y χ. Más concretamente, se tienen las siguientes complejidades *teóricas* para dichas funciones con los valores de subíndices dados por las ecuaciones 9.16 (o 9.22) y 9.17 (o 9.23):

- Las funciones $\mathbf{O_{i0}}$ están formadas por la suma de $\lceil \frac{i}{2} \rceil$ términos $x_k = a_k c_k$ y $x_{ij} = (a_i c_j + a_j c_i)$, es decir, representan $\lceil \frac{i}{2} \rceil$ 1-ciclos (k) y 2-ciclos (i, j). Específicamente, se tiene un término x_k y $(\lceil \frac{i}{2} \rceil - 1)$ términos x_{ij}. Por lo tanto, la implementación de las funciones $\mathbf{O_{i0}}$ requieren $2\lceil \frac{i}{2} \rceil - 1$ puertas AND de dos entradas y un árbol binario de $2(\lceil \frac{i}{2} \rceil - 1)$ puertas XOR de dos entradas. La profundidad del árbol binario XOR será $\lceil log_2(2\lceil \frac{i}{2} \rceil - 1) \rceil$, por lo que el retardo de los términos $\mathbf{O_{i0}}$ será $T_{AND} + \lceil log_2(2\lceil \frac{i}{2} \rceil - 1) \rceil T_{XOR}$.

- Las funciones $\mathbf{E_{i0}}$ están formadas por la suma de $\lceil \frac{i}{2} \rceil$ términos x_{ij}, por lo que su implementación requiere $2\lceil \frac{i}{2} \rceil$ puertas AND y un árbol binario de $2\lceil \frac{i}{2} \rceil - 1$ puertas XOR con una profundidad dada por $\lceil log_2(2\lceil \frac{i}{2} \rceil) \rceil$. El retardo de los términos $\mathbf{E_{i0}}$ será, por tanto, $T_{AND} + \lceil log_2(2\lceil \frac{i}{2} \rceil) \rceil T_{XOR}$.

- Las funciones $\mathbf{E_{i1}}$ constan de un término x_k y de $(\frac{m-i}{2} - 1)$ términos x_{ij}. Por lo tanto, su implementación requiere $m-i-1$ puertas AND y un árbol binario de $m-i-2$ puertas XOR de profundidad $\lceil log_2(m-i-1) \rceil$. Por lo tanto, el retardo de los términos $\mathbf{E_{i1}}$ será $T_{AND} + \lceil log_2(m-i-1) \rceil T_{XOR}$.

- Las funciones $\mathbf{O_{i1}}$ constan de $\frac{m-i-1}{2}$ términos x_{ij}, por lo que su implementación requiere $m-i-1$ puertas AND y un árbol binario de $m-i-2$ puertas XOR con una profundidad dada por $\lceil log_2(m-i-1) \rceil$. El retardo de los términos $\mathbf{E_{i1}}$ será, por tanto, $T_{AND} + \lceil log_2(m-i-1) \rceil T_{XOR}$.

Una vez establecidas las complejidades de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, cuyo resumen se muestra en la tabla 9.2, se puede determinar la complejidad *teórica* total del multiplicador a partir de las ecuaciones 9.22 y 9.23.

	#AND	#XOR	Retardo
$\mathbf{O_{i0}}$	$2\lceil \frac{i}{2} \rceil - 1$	$2(\lceil \frac{i}{2} \rceil - 1)$	$T_{AND} + \lceil log_2(2\lceil \frac{i}{2} \rceil - 1) \rceil) T_{XOR}$
$\mathbf{E_{i0}}$	$2\lceil \frac{i}{2} \rceil$	$2\lceil \frac{i}{2} \rceil - 1$	$T_{AND} + \lceil log_2(2\lceil \frac{i}{2} \rceil) \rceil) T_{XOR}$
$\mathbf{O_{i1}}$	$m - i - 1$	$m - i - 2$	$T_{AND} + \lceil log_2(m - i - 1) \rceil) T_{XOR}$
$\mathbf{E_{i1}}$	$m - i - 1$	$m - i - 2$	$T_{AND} + \lceil log_2(m - i - 1) \rceil) T_{XOR}$

Tabla 9.2: Complejidades teóricas de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$.

En estas ecuaciones (y en la figura 9.2) se observa que la construcción de un grupo $\mathbf{G_i}$ empleado para la determinación de una coordenada d_j, es *reutilizado* para la construcción del grupo $\mathbf{G_{i+1}}$ usado en la coordenada d_{j-1}. Teniendo en cuenta la tabla 9.1 del ejemplo, las definiciones dadas para los grupos $\mathbf{G_i}$ y la figura 9.2, se tiene que el coste de construcción de un grupo $\mathbf{G_i}$ es de una puerta XOR únicamente, ya que es la suma del grupo $\mathbf{G_{i-1}}$, ya construido, y del término $\mathbf{E_{(m-2-i)1}}$ o $\mathbf{O_{(m-2-i)1}}$ (dependiendo del valor de m) que también ha sido construido previamente. Además, se observa en las ecuaciones 9.22 y 9.23 que las coordenadas están formadas por la suma de un término $\mathbf{O_{i0}}$ o $\mathbf{E_{i0}}$ con un grupo $\mathbf{G_i}$ o con el término $\mathbf{E_{(m-2)1}}$ cuando se trata de la coordenada d_{m-2}, por lo que será necesaria una puerta XOR adicional por cada coordenada. Por lo tanto, la complejidad *espacial* del multiplicador vendrá dada por las complejidades individuales de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ necesarios, por las complejidades de los grupos $\mathbf{G_i}$ y por las m puertas XOR adicionales. En el apéndice A se demuestra que la complejidad *espacial* del multiplicador es de m^2 puertas AND y de $m^2 - 1$ puertas XOR.

Para poder determinar la complejidad *temporal* teórica del multiplicador, denotemos por $Prof(\mathbf{T})$ la profundidad del árbol de puertas XOR de un término $\mathbf{T}$. A partir de la definición de los grupos $\mathbf{G_i}$ y teniendo en cuenta la figura 9.2, se observa que

$$Prof(\mathbf{G_i}) = max(Prof(\mathbf{G_{i-1}}), Prof(\mathbf{E(O)_{(m-2-i)1}})) + 1 \qquad (9.24)$$

Se definió anteriormente el grupo $\mathbf{G_1} = \mathbf{E_{(m-2)1}} + \mathbf{O_{(m-3)1}}$, donde $\mathbf{E_{(m-2)1}}$ es un término producto y donde $\mathbf{O_{(m-3)1}}$ es la suma de dos términos producto, por lo que $Prof(\mathbf{G_1}) = 2$. También se vio que las profundidades de los árboles de puertas XOR de los términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ eran iguales y de valor $Prof(\mathbf{E(O)_{i1}}) = \lceil log_2(m - i - 1) \rceil$. A partir de estos valores y utilizando la expresión recursiva 9.24, se observa que las profundidades de los grupos $\mathbf{G_i}$ obtenidas son $Prof(\mathbf{G_2}) = 3$, $Prof(\mathbf{G_3}) = 4$, $Prof(\mathbf{G_4}) = 5$, ..., comprobándose que $Prof(\mathbf{G_i}) = i + 1$. Aplicando la ecuación 9.24 se obtiene que

la mayor profundidad corresponderá al grupo $\mathbf{G_{m-2}}$, que vendrá dada por

$$\begin{aligned} Prof(\mathbf{G_{m-2}}) &= max(Prof(\mathbf{G_{m-3}}), Prof(\mathbf{E(O)_{01}})) + 1 \\ &= max(m-2, \lceil log_2(m-1) \rceil) + 1 = m-1 \end{aligned} \tag{9.25}$$

debido al mayor crecimiento de $m-2$ en comparación con $\lceil log_2(m-1) \rceil$.

El grupo $\mathbf{G_{m-2}}$ participa en el cálculo de las coordenadas d_0 y d_{m-1} del producto, por lo que estas serán las que determinen el retardo total del multiplicador. Utilizando las ecuaciones 9.22 y 9.23 y teniendo en cuenta que los términos $\mathbf{E(O)_{m0}}$ son los de mayor complejidad, se observa que la coordenada d_{m-1} es la que determinará el mayor retardo, cuya profundidad será

$$\begin{aligned} Prof(d_{m-1}) &= max(Prof(\mathbf{G_{m-2}}), Prof(\mathbf{E(O)_{m0}})) + 1 \\ &= max(m-1, \lceil log_2(2\lceil \tfrac{m}{2} \rceil) \rceil) + 1 = m \end{aligned} \tag{9.26}$$

y donde se ha utilizado la mayor profundidad del término $\mathbf{E_{m0}}$ frente a $\mathbf{O_{m0}}$.

Finalmente, la complejidad teórica *temporal* vendrá dada por este número de niveles de puertas XOR, a los que habrá que añadir un nivel de puertas AND que realizan los productos $a_i c_j$ de las coordenadas de los elementos α y χ de $GF(2^m)$. Por lo tanto, el *retardo* total del multiplicador será $T_{AND} + mT_{XOR}$.

En la tabla 9.3 se comparan las complejidades teóricas obtenidas por nuestro método (*Ntra.Ap.* en la tabla) con los mejores resultados conocidos en la literatura dados por Halbutogullari y Koç [HK00] y por Zhang y Parhi [ZP01] que utilizan otros métodos similares para la construcción de un multiplicador paralelo en base canónica generado por trinomios irreducibles del tipo $f(x) = x^m + x^{m-1} + 1$.

	#XOR	#AND	Retardo
[HK00]	m^2-1	m^2	$T_{AND} + (m-1+\lceil log_2 m \rceil)T_{XOR}$
[ZP01]	m^2-1	m^2	$T_{AND} + (m-1+\lceil log_2 m \rceil)T_{XOR}$
Ntra.Ap.	m^2-1	m^2	$T_{AND} + mT_{XOR}$

Tabla 9.3: Complejidades teóricas de multiplicadores canónicos usando trinomios irreducibles $f(x) = x^m + x^{m-1} + 1$.

Los resultados mostrados en la tabla 9.3 corresponden a complejidades en las que se mantiene un equilibrio entre el número de puertas XOR y el retardo del multiplicador. En [ZP01] existe otro resultado teórico en el que se reduce el número de puertas XOR a costa de incrementar el retardo.

En la tabla 9.3 se observa que el retardo obtenido usando nuestro método *mejora* en una cantidad $(\lceil log_2 m \rceil - 1)$ el menor retardo encontrado en la literatura para este tipo de multiplicadores. Además, la complejidad espacial (número de puertas AND y XOR) obtenida usando nuestro método iguala la menor complejidad obtenida con estas otras aproximaciones.

En la tabla 9.4 se comparan los retardos de puertas XOR (T_{XOR}) obtenidos por nuestra aproximación *transposicional* con los obtenidos por el método propuesto por Halbutogullari y Koç en [HK00] para todos los trinomios irreducibles $f(x) = x^m + x^{m-1} + 1$ existentes hasta el grado $m = 100$. En esta tabla, las entradas numéricas ubicadas en la columna encabezada por $f(x)$ representan los coeficientes del polinomio irreducible que son distintos de cero. Por ejemplo, la entrada $(9, 8, 0)$ representa el trinomio $f(x) = x^9 + x^8 + x^0$. También se muestra el porcentaje de mejora obtenida por nuestro método, así como los valores totales.

		Ntra.Ap.	Halbut.-Koç	Mejora
m	f(x)	T_{XOR}	T_{XOR}	%
2	2,1,0	2	2	0.0
3	3,2,0	3	4	25.0
4	4,3,0	4	5	20.0
6	6,5,0	6	8	25.0
7	7,6,0	7	9	22.2
9	9,8,0	9	12	25.0
15	15,14,0	15	18	16.7
22	22,21,0	22	26	15.4
28	28,27,0	28	32	12.5
30	30,29,0	30	34	11.8
46	46,45,0	46	51	9.8
60	60,59,0	60	65	7.7
63	63,62,0	63	68	7.4
Total		295	334	11.7

Tabla 9.4: Comparación de retardos T_{XOR} de los multiplicadores obtenidos con nuestra aproximación y por Halbutogullari y Koç para distintos trinomios irreducibles.

De los resultados mostrados en la tabla 9.4 se observa que los porcentajes de mejora de nuestro método se van reduciendo según se incrementa el grado del polinomio, obteniéndose en promedio una mejora del 11.7 %. Para polinomios de grado hasta $m = 9$, las reducciones en T_{XOR} son iguales o superiores al 20 %. Este hecho es muy importante cuando se consideran *campos compuestos*[1] (especialmente cuando se utiliza el importante campo $GF(2^4)$ como campo *base*) para la construcción del multiplicador, ya que la reducción de la complejidad de la multiplicación en el campo *base* (que con nuestro método es de un 20 %) influye directamente en la complejidad total del multiplicador.

Al igual que en casos anteriores, la complejidad teórica no se corresponde con la complejidad obtenida cuando se realizan implementaciones reconfigurables. La igualdad *espacial* y la mejora *temporal* teórica obtenida por el método

[1] Los *campos compuestos* $GF((2^n)^k)$ son extensiones del campo base $GF(2^n)$ y son isomorfos a los campos de Galois $GF(2^m)$ con $m = n \cdot k$.

transposicional en comparación con los métodos dados en [HK00] y [ZP01] no se corresponden exactamente con los resultados experimentales obtenidos, como se muestra a continuación.

9.3.3. Resultados experimentales

Se ha realizado la implementación paralela sobre dispositivos FPGA de multiplicadores en base canónica utilizando el método *transposicional* dado por las ecuaciones 9.22 y 9.23 y utilizando el método propuesto por Halbutogullari y Koç en [HK00]. Para ello, se ha utilizado *Xilinx Foundation F2.1i* y se han realizado descripciones VHDL de los multiplicadores paralelos utilizando ambos métodos para los campos finitos $GF(2^m)$ generados por trinomios irreducibles del tipo $f(x) = x^m + x^{m-1} + 1$ de grados m = 4, 6, 7, 9, 15 y 22. Los dispositivos utilizados han sido los 4013XLPQ160 pertenecientes a la familia XC4000XL y la síntesis se realizó utilizando optimización para *Area* con nivel de esfuerzo *High*. Asimismo, las métricas del rendimiento utilizadas han sido las mismas que en experimentos anteriores, es decir, el *número de CLBs* y el *retardo máximo combinacional*. Los resultados experimentales obtenidos se muestran en la tabla 9.5.

	Halbut.-Koç		Transposicional	
	CLBs	Ret. Max (ns)	CLBs	Ret. Max (ns)
$GF(2^4)$	7	15.5	6	13.9
$GF(2^6)$	16	19.7	12	21.1
$GF(2^7)$	24	26.8	18	24.7
$GF(2^9)$	40	25.4	30	29.8
$GF(2^{15})$	108	30.5	79	49.6
$GF(2^{22})$	220	49.6	168	94.5
Total	415	167.5	313	233.6

Tabla 9.5: Resultados experimentales de multiplicadores canónicos sobre campos generados por trinomios $f(x) = x^m + x^{m-1} + 1$ irreducibles utilizando FPGAs.

De los resultados mostrados en la tabla 9.5, se observa que la implementación de los multiplicadores en base canónica utilizando la aproximación *transposicional* consume un menor número de CLBs que si se utiliza el método dado por Halbutogullari y Koç. Considerando el número total de CLBs consumidos por cada método para todas las implementaciones, se observa que nuestro método utiliza un 24.6 % menos de CLBs, siendo el mejor resultado el correspondiente al campo $GF(2^{15})$ con una reducción del 26.9 %. Sin embargo, se observa en la tabla que los retardos obtenidos con nuestra aproximación son un 39.5 % peores que los conseguidos por el otro método, hecho que no se corresponde con el resultado teórico obtenido anteriormente de un mejor comportamiento temporal del método transposicional.

La mejora de la complejidad espacial *experimental* obtenida por nuestro método no se corresponde tampoco con la igualdad obtenida *teóricamente* con las otras aproximaciones. Además, la mejora experimental conseguida por nuestro método es significativamente mayor que la obtenida en la sección 8.3 para los AOPs irreducibles. Esto se debe a que para el tipo de trinomios estudiados en esta sección, el método transposicional proporciona un gran número de *agrupaciones* $\mathbf{G_i}$ (formadas por sumas de términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$) que son *compartidas* entre las distintas coordenadas del producto, mientras que en el caso de los multiplicadores sobre AOPs irreducibles, la compartición entre coordenadas se reducía únicamente a un sólo término ($\mathbf{E_0}$ para la multiplicación canónica y $\mathbf{M}$ para la multiplicación normal). Esta mayor *agrupación* y *compartición* suministradas en la descripción de alto nivel del multiplicador ayudaría en mayor medida a las herramientas de *mapping* reconfigurable en la realización de una mejor optimización para área, como se concluyó en el capítulo 8.

9.4. Trinomios irreducibles $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m impar)

Cuando el trinomio irreducible generador del campo finito es de la forma $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m impar) se tiene que el parámetro $\tau = \lceil \frac{m-1}{\Delta} \rceil = 2$ ($\Delta = m - n = \frac{m-1}{2}$), por lo que la descomposición de la matriz $\mathbf{K}$ dada en la ecuación 9.1 está formada por la suma de las tres matrices $\mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2}$. Al igual que se hizo en la sección 9.3, esta descomposición también se puede deducir a partir de la estructura de la matriz $\mathbf{K_1}$ dada en la ecuación 9.3, teniendo en cuenta que para este tipo de trinomios la columna nula se encuentra en la posición $\frac{m+1}{2}$ de $\mathbf{K_1}$.

A partir de las expresiones generales de las coordenadas d_{Ω_i} del producto δ dadas en la ecuación 9.15 y usando las ecuaciones 9.10 a 9.14, se dan a continuación las ecuaciones que permiten el cálculo de las coordenadas d_{Ω_i} del producto utilizando el método *transposicional*, en las que se hace distinción entre valores del parámetro Δ *pares* e *impares*. Asimismo, se define el nuevo parámetro $\Delta_m = (m-1) - \Delta$, que para el tipo de trinomios que estamos considerando, coincide con el parámetro Δ.

Se tiene, entonces, que las coordenadas del producto δ para valores *pares* del parámetro Δ vienen dadas por la expresiones

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) + \begin{cases} \mathbf{O_{(\Delta+i)1}} & i\ par,\ i < \Delta_m \\ - & i = \Delta_m \\ \mathbf{E_{(i-(\Delta_m+1))1}} + \mathbf{E_{(i-1)1}} & i\ par,\ i > \Delta_m \end{cases} \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \begin{cases} \mathbf{E_{(\Delta+i)1}} & i\ impar,\ i < \Delta_m \\ \mathbf{O_{(i-(\Delta_m+1))1}} + \mathbf{O_{(i-1)1}} & i\ impar,\ i > \Delta_m \end{cases} \\ (\mathbf{O_{m0}}) + \mathbf{E_{(i-(\Delta_m+1))1}} + \mathbf{E_{(i-1)1}} \quad i = m-1 \end{cases} \tag{9.27}$$

mientras que las coordenadas para valores *impares* de Δ son

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) + \begin{cases} \mathbf{E_{(\Delta+i)1}} & i\ par,\ i < \Delta_m \\ \mathbf{O_{(i-(\Delta_m+1))1}} + \mathbf{E_{(i-1)1}} & i\ par,\ i > \Delta_m \end{cases} \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \begin{cases} \mathbf{O_{(\Delta+i)1}} & i\ impar,\ i < \Delta_m \\ - & i = \Delta_m \\ \mathbf{E_{(i-(\Delta_m+1))1}} + \mathbf{O_{(i-1)1}} & i\ impar,\ i > \Delta_m \end{cases} \\ (\mathbf{O_{m0}}) + \mathbf{O_{(i-(\Delta_m+1))1}} + \mathbf{E_{(i-1)1}} \quad i = m-1 \end{cases} \tag{9.28}$$

donde $i = 0, 1, \ldots, m-1$ y donde los términos entre paréntesis son proporcionados por el vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$, mientras que el tercer y cuarto sumandos (segundo y tercero, si $i = m-1$) de ambas ecuaciones pertenecen a los vectores $\underline{\alpha}^t_\Omega \mathbf{K_1}$ y $\underline{\alpha}^t_\Omega \mathbf{K_2}$, respectivamente.

A continuación se presenta un ejemplo de multiplicación sobre el campo de Galois $GF(2^7)$ utilizando el método transposicional, en el que se muestran las agrupaciones y comparticiones a realizar para poder reducir finalmente la complejidad total del multiplicador.

9.4.1. Ejemplo de multiplicación sobre $GF(2^7)$

El producto δ de dos elementos α y χ pertenecientes al campo $GF(2^7)$ generado por el trinomio $f(x) = x^7 + x^4 + 1$ se calcula utilizando las expresiones anteriormente definidas. En este caso, el parámetro $\Delta = \frac{m-1}{2} = 3 = \Delta_m$ y la matriz $\mathbf{K}$ se descompone en la suma de las tres matrices siguientes:

$$\mathbf{K} = \mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2} = \begin{pmatrix} c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & c_6 \\ c_4 & c_3 & c_2 & c_1 & c_0 & c_6 & c_5 \\ c_3 & c_2 & c_1 & c_0 & c_6 & c_5 & c_4 \\ c_2 & c_1 & c_0 & c_6 & c_5 & c_4 & c_3 \\ c_1 & c_0 & c_6 & c_5 & c_4 & c_3 & c_2 \\ c_0 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_6 & 0 & 0 & 0 & 0 \\ 0 & c_6 & c_5 & 0 & 0 & 0 & 0 \\ c_6 & c_5 & c_4 & 0 & 0 & 0 & 0 \\ c_5 & c_4 & c_3 & 0 & 0 & 0 & c_6 \\ c_4 & c_3 & c_2 & 0 & 0 & c_6 & c_5 \\ c_3 & c_2 & c_1 & 0 & c_6 & c_5 & c_4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_6 & 0 & 0 & 0 & 0 \\ 0 & c_6 & c_5 & 0 & 0 & 0 & 0 \\ c_6 & c_5 & c_4 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{9.29}$$

donde los términos c_i son las coordenadas de χ en base canónica y donde las matrices $\mathbf{K_0}$, $\mathbf{K_1}$ y $\mathbf{K_2}$ se construyen a partir de las ecuaciones 9.2, 9.3 y 9.4, respectivamente.

El método *transposicional* permite el cálculo de los 1-ciclos y 2-ciclos utilizando las ecuaciones 9.6 a 9.9, obteniéndose las funciones $\mathbf{OC_{10}} = (0)$, $\mathbf{EC_{20}} = (0,1)$, $\mathbf{OC_{30}} = (1)(0,2)$, $\mathbf{EC_{40}} = (0,3)(1,2)$, $\mathbf{OC_{50}} = (2)(0,4)(1,3)$, $\mathbf{EC_{60}} = (0,5)(1,4)(2,3)$, $\mathbf{OC_{70}} = (3)(0,6)(1,5)(2,4)$ así como las funciones $\mathbf{OC_{01}} = (1,6)(2,5)(3,4)$, $\mathbf{EC_{11}} = (4)(2,6)(3,5)$, $\mathbf{OC_{21}} = (3,6)(4,5)$, $\mathbf{EC_{31}} = (5)(4,6)$ $\mathbf{OC_{41}} = (5,6)$, $\mathbf{EC_{51}} = (6)$. Las sumas de los términos representados por estos ciclos vienen dadas por las siguientes funciones

$$\begin{aligned}
\mathbf{O_{10}} &= a_0c_0 \\
\mathbf{E_{20}} &= (a_0c_1 + a_1c_0) \\
\mathbf{O_{30}} &= a_1c_1 + (a_0c_2 + a_2c_0) \\
\mathbf{E_{40}} &= (a_0c_3 + a_3c_0) + (a_1c_2 + a_2c_1) \\
\mathbf{O_{50}} &= a_2c_2 + (a_0c_4 + a_4c_0) + (a_1c_3 + a_3c_1) \\
\mathbf{E_{60}} &= (a_0c_5 + a_5c_0) + (a_1c_4 + a_4c_1) + (a_2c_3 + a_3c_2) \\
\mathbf{O_{70}} &= a_3c_3 + (a_0c_6 + a_6c_0) + (a_1c_5 + a_5c_1) + (a_2c_4 + a_4c_2)
\end{aligned} \tag{9.30}$$

$$\begin{aligned}
\mathbf{O_{01}} &= (a_1c_6 + a_6c_1) + (a_2c_5 + a_5c_2) + (a_3c_4 + a_4c_3) \\
\mathbf{E_{11}} &= a_4c_4 + (a_2c_6 + a_6c_2) + (a_3c_5 + a_5c_3) \\
\mathbf{O_{21}} &= (a_3c_6 + a_6c_3) + (a_4c_5 + a_5c_4) \\
\mathbf{E_{31}} &= a_5c_5 + (a_4c_6 + a_6c_4) \\
\mathbf{O_{41}} &= (a_5c_6 + a_6c_5) \\
\mathbf{E_{51}} &= a_6c_6
\end{aligned} \tag{9.31}$$

donde los términos a_i y c_i son las coordenadas en base canónica de los elementos α y χ de $GF(2^7)$, respectivamente.

La aplicación de la ecuación 9.28 (Δ impar) permite determinar las coordenadas del producto δ como sumas de términos agrupados por las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ dadas en las ecuaciones 9.30 y 9.31. En la tabla 9.6 se muestran estas coordenadas para el trinomio $f(x) = x^7 + x^4 + 1$, en donde se especifican los componentes de los vectores $\underline{\alpha}^t_\Omega\mathbf{K_0}$, $\underline{\alpha}^t_\Omega\mathbf{K_1}$ y $\underline{\alpha}^t_\Omega\mathbf{K_2}$, y donde la coordenada d_i es la suma de las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ existentes en la fila i-ésima.

En la tabla 9.6 también se muestra la *compartición* de términos que permite la reducción de la complejidad del multiplicador construido con este tipo de trinomios. Se observa que la subexpresión $(\mathbf{O_{21}} + \mathbf{E_{51}})$ perteneciente a la coordenada d_2, también aparece en la coordenada d_6. Lo mismo sucede con la subexpresión $(\mathbf{E_{11}} + \mathbf{O_{41}})$, perteneciente a d_1 y d_5, y con $(\mathbf{O_{01}} + \mathbf{E_{31}})$ que aparece en las coordenadas d_0 y d_6. Estos términos, por lo tanto, se pueden construir *una sola vez* y pueden ser *reutilizados* por las coordenadas del producto sin necesidad de volver a ser construidos, reduciéndose de esta forma su complejidad. El número de puertas obtenido finalmente para la construcción del multiplicador sobre $GF(2^7)$ es de 49 puertas AND y de 48 puertas XOR.

Esta forma de construcción de las coordenadas del producto también permite la reducción de la complejidad *temporal* del multiplicador por medio de la utilización de árboles *binarios* de puertas XOR para la construcción de los

	$\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$		$\underline{\alpha}_{\Omega}^{t}\mathbf{K_1}$	$\underline{\alpha}_{\Omega}^{t}\mathbf{K_2}$
d_0	$\mathbf{O_{10}}$	$\mathbf{O_{01}}$	$\mathbf{E_{31}}$	-
d_1	$\mathbf{E_{20}}$	$\mathbf{E_{11}}$	$\mathbf{O_{41}}$	-
d_2	$\mathbf{O_{30}}$	$\mathbf{O_{21}}$	$\mathbf{E_{51}}$	-
d_3	$\mathbf{E_{40}}$	$\mathbf{E_{31}}$	-	-
d_4	$\mathbf{O_{50}}$	$\mathbf{O_{41}}$	$\mathbf{O_{01}}$	$\mathbf{E_{31}}$
d_5	$\mathbf{E_{60}}$	$\mathbf{E_{51}}$	$\mathbf{E_{11}}$	$\mathbf{O_{41}}$
d_6	$\mathbf{O_{70}}$	-	$\mathbf{O_{21}}$	$\mathbf{E_{51}}$

Tabla 9.6: Coordenadas d_i del producto para la multiplicación sobre $GF(2^7)$.

términos individuales $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, así como para la construcción de las subexpresiones compartidas mencionadas anteriormente y para la suma final de dichos términos obtenida de la tabla 9.6. Se obtiene finalmente que la complejidad temporal total del multiplicador es de 5 retardos de puertas XOR y de 1 retardo de puerta AND, es decir, $T_{AND} + 5T_{XOR}$.

9.4.2. Análisis teórico de complejidad

Las ecuaciones 9.27 (para Δ *par*) y 9.28 (para Δ *impar*) proporcionan las coordenadas del producto $\delta = \alpha \cdot \chi$ en función de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, y su aplicación a cualquier trinomio irreducible de la forma $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m impar) determina agrupaciones de términos con la misma estructura que la mostrada en el ejemplo visto para $GF(2^7)$.

Al igual que se hizo en la sección anterior con los trinomios irreducibles $f(x) = x^m + x^{m-1} + 1$, se pueden determinar unos grupos formados por la suma de dos términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$. A estos grupos se les denota como $\mathbf{G_i}$, donde el subíndice i coincide con el subíndice de la coordenada del producto a la cual pertenece dicho grupo. Según esto, se tiene que los grupos $\mathbf{G_i}$, para valores del parámetro Δ *pares*, vienen dados por las expresiones

$$\mathbf{G_i} = \begin{cases} \left.\begin{array}{ll} \mathbf{O_{i1}} + \mathbf{O_{(\Delta+i)1}} & i\ par \\ \mathbf{E_{i1}} + \mathbf{E_{(\Delta+i)1}} & i\ impar \end{array}\right\} & i < \Delta_m \\ \mathbf{G_{i-(\Delta_m+1)}} & i > \Delta_m \end{cases} \tag{9.32}$$

mientras que para valores *impares* de Δ vienen dados como

$$\mathbf{G_i} = \begin{cases} \left.\begin{array}{ll} \mathbf{O_{i1}} + \mathbf{E_{(\Delta+i)1}} & i\ par \\ \mathbf{E_{i1}} + \mathbf{O_{(\Delta+i)1}} & i\ impar \end{array}\right\} & i < \Delta_m \\ \mathbf{G_{i-(\Delta_m+1)}} & i > \Delta_m \end{cases} \tag{9.33}$$

para $i = 0, 1, \ldots, m-1$ con $i \neq \Delta_m$, ya que se observa que para $i = \Delta_m$, no existe grupo $\mathbf{G_i}$ asociado.

Utilizando estas ecuaciones 9.32 y 9.33 junto con las ecuaciones 9.27 y 9.28, se pueden dar unas nuevas expresiones de las coordenadas del producto δ en función de los grupos $\mathbf{G_i}$ definidos anteriormente. Para Δ *par* se tiene que las coordenadas del producto vienen dadas por

$$d_{\Omega_i} = \begin{cases} \mathbf{G_i} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} & i\ par \\ \mathbf{E_{(i+1)0}} & i\ impar \end{array} \right\} & i < \Delta_m \\ \mathbf{O_{(i+1)0}} + \mathbf{O_{i1}} & i = \Delta_m \\ \mathbf{G_{i-(\Delta_m+1)}} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} + \mathbf{O_{i1}} & i\ par \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} & i\ impar \end{array} \right\} & i > \Delta_m \\ \mathbf{O_{m0}} + \mathbf{G_{i-(\Delta_m+1)}} & i = m-1 \end{cases} \tag{9.34}$$

mientras que para Δ *impar* se tiene

$$d_{\Omega_i} = \begin{cases} \mathbf{G_i} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} & i\ par \\ \mathbf{E_{(i+1)0}} & i\ impar \end{array} \right\} & i < \Delta_m \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} & i = \Delta_m \\ \mathbf{G_{i-(\Delta_m+1)}} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} + \mathbf{O_{i1}} & i\ par \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} & i\ impar \end{array} \right\} & i > \Delta_m \\ \mathbf{O_{m0}} + \mathbf{G_{i-(\Delta_m+1)}} & i = m-1 \end{cases} \tag{9.35}$$

para $i = 0, 1, \ldots, m-1$. La arquitectura paralela del multiplicador se extrae fácilmente de estas expresiones, teniendo en cuenta la utilización de árboles *binarios* de puertas XOR tanto para la construcción de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, como para la construcción de los grupos $\mathbf{G_i}$ y para la suma final dada por las ecuaciones 9.34 y 9.35.

La complejidad *espacial* teórica del multiplicador se calcula utilizando las complejidades de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ dadas en la tabla 9.2 y teniendo en cuenta las ecuaciones 9.32 a 9.35. Es decir, la complejidad del multiplicador vendrá dada por la suma de complejidades de los términos $\mathbf{E_{i(0,1)}}$, $\mathbf{O_{i(0,1)}}$ y $\mathbf{G_i}$ necesarios, así como de la complejidad necesaria para la suma de los mismos, según las ecuaciones 9.34 y 9.35.

De las ecuaciones 9.32 y 9.33 se observa que únicamente son necesarios los grupos $\mathbf{G_i}$ ($i < \Delta_m$), cuya complejidad es de 1 puerta XOR cada uno, ya que los grupos para $i > \Delta_m$ son iguales a estos y no es necesaria su construcción. Además, no existe grupo $\mathbf{G_i}$ para la coordenada $i = \Delta_m$, por lo que se requieren Δ_m puertas XOR para la construcción de todos los grupos $\mathbf{G_i}$. Por otra parte, utilizando las ecuaciones 9.34 y 9.35 se observa que para la suma de estos términos $\mathbf{E_{i(0,1)}}$, $\mathbf{O_{i(0,1)}}$ y $\mathbf{G_i}$ son necesarias Δ_m puertas XOR (para las coordenadas de d_0 a d_{Δ_m-1}), una puerta XOR (para la coordenada d_{Δ_m}),

$2(\Delta_m - 1)$ puertas XOR (para las coordenadas de d_{Δ_m+1} a d_{m-2}) y una puerta XOR (para la coordenada d_{m-1}). Finalmente, en el apéndice A se demuestra que la complejidad *espacial* del multiplicador es de m^2 puertas AND y de $m^2 - 1$ puertas XOR.

Para poder determinar la complejidad *temporal* teórica del multiplicador, calculamos en primer lugar el retardo teórico de los grupos $\mathbf{G_i}$ ($i < \Delta_m$). A partir de las definiciones dadas en las ecuaciones 9.32 y 9.33, se obtiene que la profundidad del árbol de puertas XOR para un grupo $\mathbf{G_i}$ viene dada por

$$\begin{aligned} Prof(\mathbf{G_i}) &= max(Prof(\mathbf{E(O)_{i1}}), Prof(\mathbf{E(O)_{(\Delta+i)1}})) + 1 \\ &= Prof(\mathbf{E(O)_{i1}}) + 1 = \lceil log_2(m - i - 1) \rceil + 1 \end{aligned} \tag{9.36}$$

donde se ha utilizado el hecho de que los términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ tienen el mismo retardo (según se muestra en la tabla 9.2) y el que dicha complejidad temporal sea mayor cuanto menor sea el índice i, con lo que se obtiene que $Prof(\mathbf{E(O)_{i1}}) > Prof(\mathbf{E(O)_{(\Delta+i)1}})$. Asimismo, la ecuación 9.36 determina que el número de niveles de puertas XOR se incrementa con el decremento del índice i, por lo que los grupos $\mathbf{G_i}$ con menor subíndice son los que presentan un mayor retardo.

Utilizando la ecuación 9.36 junto con las ecuaciones 9.34 y 9.35, se pueden hacer las siguientes distinciones, en función del parámetro Δ_m, con respecto del retardo de las coordenadas del producto:

- Cuando $i < \Delta_m$, se verifica que la profundidad del árbol de puertas XOR de la coordenada d_i viene dada por

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{E(O)_{(i+1)0}}), Prof(\mathbf{G_i})) + 1 \\ &= Prof(\mathbf{G_i}) + 1 = \lceil log_2(m - i - 1) \rceil + 2 \end{aligned} \tag{9.37}$$

  Para Δ *impar*, se verifica que las coordenadas d_i con $i < \Delta_m - 1$ presentan la misma complejidad (siendo esta la máxima de las d_i para $i < \Delta_m$), mientras que la complejidad de $d_{\Delta_m - 1}$ es menor. Por otra parte, para Δ *par*, se tiene que todas las d_i con $i < \Delta_m$ presentan la misma complejidad.

- Para $i = \Delta_m$, la coordenada d_i no tiene grupo $\mathbf{G_i}$ asociado, por lo que el retardo para Δ *par* viene dado por

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{O_{(i+1)0}}), Prof(\mathbf{O_{i1}})) + 1 \\ &= Prof(\mathbf{O_{(i+1)0}}) + 1 = \lceil log_2(2\lceil \tfrac{i+1}{2} \rceil - 1) \rceil + 1 \end{aligned} \tag{9.38}$$

  mientras que para Δ *impar* es de la forma

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{E_{(i+1)0}}), Prof(\mathbf{E_{i1}})) + 1 \\ &= Prof(\mathbf{E_{(i+1)0}}) + 1 = \lceil log_2(2\lceil \tfrac{i+1}{2} \rceil) \rceil + 1 \end{aligned} \tag{9.39}$$

  donde se han utilizado los resultados mostrados en la tabla 9.2.

- Para $i > \Delta_m$ $(i \neq m-1)$, se observa que las coordenadas d_i se calculan por medio de la suma de dos términos $\mathbf{E(O)_{i0}}$ y $\mathbf{E(O)_{i1}}$, y del grupo $\mathbf{G_i}$. En este caso, es más óptimo agrupar la suma de los términos $\mathbf{E(O)_{i(0,1)}}$ (cuyo retardo iguala, en el peor caso, el retardo del grupo asociado) y posteriormente sumarle el grupo $\mathbf{G_i}$. De esta forma se tiene que, independientemente de que Δ sea *par* o *impar*, la profundidad del árbol XOR viene dada por

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{E(O)_{(i+1)0}} + \mathbf{E(O)_{i1}}), Prof(\mathbf{G_i})) + 1 \\ &= Prof(\mathbf{G_i}) + 1 = \lceil log_2(m - i + \Delta_m) \rceil + 2 \end{aligned} \tag{9.40}$$

  donde se ha utilizado el hecho de que para $i > \Delta_m$, $\mathbf{G_i} = \mathbf{G_{i-(\Delta_m+1)}}$, como se vio en las ecuaciones 9.32 y 9.33.

- Para $i = m-1$, se tiene de las ecuaciones 9.34 y 9.35 que la coordenada d_i es la suma de $\mathbf{O_{m0}}$ y $\mathbf{G_i} = \mathbf{G_{i-(\Delta_m+1)}}$, por lo que

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{O_{m0}}), Prof(\mathbf{G_{m-\Delta_m-2}})) + 1 \\ &= max(\lceil log_2 m \rceil, 1 + \lceil log_2(\tfrac{m+1}{2}) \rceil) + 1 = \lceil log_2 m \rceil + 1 \end{aligned} \tag{9.41}$$

  donde se ha utilizado el valor de $\Delta_m = \frac{m-1}{2}$ para este tipo de trinomios.

La complejidad temporal del multiplicador vendrá determinada por la máxima complejidad temporal de entre las establecidas en las ecuaciones anteriores 9.37 a 9.41 para las distintas coordenadas, obteniéndose como retardo máximo $\lceil log_2(m-i-1) \rceil + 2$ para las coordenadas d_i con $i < \Delta_m$ (se puede comprobar que para los valores de $i > \Delta_m$, el retardo $\lceil log_2(m - i + \Delta_m) \rceil + 2$ coincide con este retardo máximo dado). Esta complejidad temporal máxima se puede escoger para la coordenada d_0 debido a que, como se vio anteriormente, los grupos $\mathbf{G_i}$ con menor subíndice son los que presentan un mayor retardo, estableciéndose $2 + \lceil log_2(m-1) \rceil$ como retardo máximo. Finalmente, la complejidad teórica *temporal* vendrá dada por este número de niveles de puertas XOR, a los que habrá que añadir un nivel de puertas AND que realizan los productos $a_i c_j$ de las coordenadas de los elementos α y χ de $GF(2^m)$. Por lo tanto, el *retardo* total del multiplicador construido utilizando el método *transposicional* será $T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$.

	#XOR	#AND	Retardo
[HK00]	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil log_2 m \rceil) T_{XOR}$
[ZP01]	$m^2 - 1$	m^2	$T_{AND} + (\lfloor \frac{2(m-2)}{m-1} \rfloor + 1 + \lceil log_2 m \rceil) T_{XOR}$
Ntra.Ap.	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$

Tabla 9.7: Complejidades teóricas de multiplicadores canónicos usando trinomios irreducibles $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ $(m$ *impar*$)$.

En la tabla 9.7 se comparan las complejidades teóricas obtenidas por nuestro método (*Ntra.Ap.*) con los mejores resultados dados en la literatura por Halbutogullari y Koç [HK00] y por Zhang y Parhi [ZP01] para la construcción de un multiplicador paralelo en base canónica generado por trinomios irreducibles del tipo $f(x) = x^m + x^{\frac{m+1}{2}} + 1$, para m *impar*. Asimismo, los resultados teóricos dados en esta tabla corresponden a complejidades en las que se mantiene un equilibrio entre el número de puertas XOR y el retardo del multiplicador. En [ZP01] existe otro resultado en el que se reduce el número de puertas XOR a costa de un incremento en el retardo.

En la tabla 9.8 se comparan los retardos de puertas XOR (T_{XOR}) obtenidos por nuestra aproximación *transposicional* con los obtenidos en [HK00] para todos los trinomios irreducibles $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m *impar*) existentes hasta el grado $m = 100$. También se muestra el porcentaje de mejora obtenida por nuestro método, así como los valores totales. Se puede comprobar que los resultados teóricos dados en [ZP01] coinciden con los dados en [HK00].

		Ntra.Ap.	Halbut.-Koç	Mejora
m	f(x)	T_{XOR}	T_{XOR}	%
3	3,2,0	3	4	25.0
5	5,3,0	4	5	20.0
7	7,4,0	5	5	0.0
9	9,5,0	5	6	16.7
15	15,8,0	6	6	0.0
41	41,21,0	8	8	0.0
63	63,32,0	8	8	0.0
65	65,33,0	8	9	11.1
71	71,36,0	9	9	0.0
Total		56	60	6.7

Tabla 9.8: Comparación de retardos T_{XOR} de los multiplicadores obtenidos con nuestra aproximación y por Halbutogullari y Koç para distintos trinomios irreducibles.

Los resultados dados en la tabla 9.8 muestran que, en este caso, nuestro método obtiene una menor reducción en el retardo que en los trinomios vistos en la sección 9.3. Esta mejora más reducida obtenida por nuestro método se debe a que se comparte un menor numero de grupos $\mathbf{G_i}$ que en la sección anterior y a que los resultados dados en esta tabla están muy optimizados con respecto a T_{XOR}, lo que hace más difícil la obtención de reducciones mayores del retardo. A pesar de esto y para el total de polinomios mostrados, se obtiene una mejora con respecto al método dado en [HK00] del 6.7 %. Además, si se consideran únicamente los multiplicadores para los que se reduce el retardo (para $m = 3, 5, 9$ y 65), se observa entonces que nuestro método obtiene una mejora en la complejidad temporal del 16.7 %.

De los resultados obtenidos en la tabla 9.8, también se puede señalar la coincidencia de resultados para el trinomio $f(x) = x^3 + x^2 + 1$ con los obtenidos en la tabla 9.4, ya que este polinomio pertenece a estos dos tipos de trinomios estudiados ($f(x) = x^m + x^{m-1} + 1$ y $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ con *m impar*). Asimismo, la comparación de los resultados mostrados en las tablas 9.4 y 9.8 permite observar la menor complejidad temporal de los multiplicadores construidos utilizando el trinomio $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (*m impar*) como generador del campo, demostrando la importancia de la selección del polinomio generador en la complejidad final del multiplicador.

9.4.3. Resultados experimentales

Se ha realizado la implementación paralela sobre FPGAs de multiplicadores canónicos utilizando el método *transposicional* dado por las ecuaciones 9.34 y 9.35 y utilizando el método propuesto por Halbutogullari y Koç en [HK00]. Se ha empleado *Xilinx Foundation F2.1i* y se han realizado descripciones VHDL de los multiplicadores utilizando ambos métodos para los campos generados por trinomios irreducibles $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ con grados *impares* $m = 5$, 7, 9 y 15. Los dispositivos utilizados han sido los 4013XLPQ160 pertenecientes a la familia XC4000XL y se ha realizado la síntesis utilizando optimización para *Area* con un nivel de esfuerzo *High*. Asimismo, las métricas del rendimiento utilizadas han sido el *número de CLBs* y el *retardo máximo combinacional*, obteniéndose los resultados experimentales mostrados en la tabla 9.9.

	Halbut.-Koç		Transposicional	
	CLBs	Ret. Max (ns)	CLBs	Ret. Max (ns)
$GF(2^5)$	11	15.5	9	15.8
$GF(2^7)$	23	20.4	20	18.6
$GF(2^9)$	35	24.5	32	23.7
$GF(2^{15})$	92	33.3	81	31.5
Total	161	93.7	142	89.6

Tabla 9.9: Resultados experimentales de multiplicadores canónicos sobre campos generados por trinomios irreducibles $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (*m impar*).

De los resultados mostrados en la tabla 9.9, se observa que la implementación de los multiplicadores en base canónica utilizando nuestro método *transposicional* consume un menor número de CLBs que si se utiliza la aproximación dada por Halbutogullari y Koç. Considerando el número total de CLBs para todas las implementaciones, se observa que nuestro método utiliza un 11.8 % menos de CLBs, siendo el mejor resultado el correspondiente al campo $GF(2^5)$ con una reducción del 18.2 %. Nuestro método también obtiene una mejora del 4.4 % en el retardo del multiplicador, con un mejor caso del 8.8 %

de reducción del retardo correspondiente al campo $GF(2^7)$. Esta complejidad *temporal* experimental sí que se corresponde con la complejidad teórica obtenida anteriormente, aunque el mejor resultado experimental obtenido por nuestro método en cuanto al número de CLBs no coincide con la igualdad *espacial* teórica mostrada en la tabla 9.7.

La mejora de la complejidad espacial *experimental* obtenida por nuestro método para este tipo de trinomios es menor que la reducción de área obtenida para los trinomios vistos en la sección 9.3. Esto se debe a que con los trinomios $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m *impar*), la *compartición* de términos *agrupados* es menor que la obtenida para $f(x) = x^m + x^{m-1} + 1$ y, según lo visto en el capítulo 8, las herramientas automáticas realizan una menor compartición de *grupos* de dos puertas AND de dos entradas y una XOR de dos entradas implementables con una LUT, por lo que la herramienta de *mapping* realiza una menor optimización para área.

9.5. Trinomios irreducibles $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m impar)

Cuando el trinomio irreducible generador del campo finito es de la forma $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m impar) se tiene que el parámetro $\Delta = m - n = \frac{m+1}{2}$, con lo que $\tau = \lceil \frac{m-1}{\Delta} \rceil = \lceil \frac{2(m-1)}{(m+1)} \rceil$. Se puede comprobar que para valores *impares* de m con $m > 3$, el parámetro $\tau = 2$, por lo que la descomposición de la matriz $\mathbf{K}$ dada en la ecuación 9.1 está formada por la suma de las tres matrices $\mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2}$. Sin embargo, para $m = 3$, el parámetro $\tau = 1$ por lo que la matriz $\mathbf{K}$ se descompone en la suma de dos matrices $\mathbf{K_0}+\mathbf{K_1}$ únicamente (de hecho, el trinomio $f(x) = x^3 + x + 1$ pertenece al grupo de trinomios de la forma $f(x) = x^m + x + 1$ que se estudiará en la sección 9.7).

Utilizando la ecuación 9.15 y las ecuaciones 9.10 a 9.14, se dan a continuación las expresiones que permiten el cálculo de las coordenadas d_{Ω_i} del producto δ por el método *transposicional*, en función de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$. Al igual que en trinomios anteriores, se distingue entre valores del parámetro Δ *pares* e *impares* y se tiene en cuenta que para el tipo de trinomios que se consideran, $\Delta_m = \frac{m-3}{2} \neq \Delta$.

Las coordenadas del producto δ para valores *pares* de Δ vienen dadas por

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) + \begin{cases} \mathbf{O_{(\Delta+i)1}} & i\ par,\ i < \Delta_m \\ - & i = \Delta_m \\ \mathbf{E_{(i-(\Delta_m+1))1}} + \mathbf{E_{(i+1)1}} & i\ par,\ i > \Delta_m \end{cases} \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \begin{cases} \mathbf{E_{(\Delta+i)1}} & i\ impar,\ i < \Delta_m \\ \mathbf{O_{(i-(\Delta_m+1))1}} + \mathbf{O_{(i+1)1}} & i\ impar,\ i > \Delta_m \end{cases} \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \mathbf{O_{(i-(\Delta_m+1))1}} & i = m-2 \\ (\mathbf{O_{m0}}) + \mathbf{E_{(i-(\Delta_m+1))1}} & i = m-1 \end{cases} \quad (9.42)$$

mientras que las coordenadas para valores *impares* de Δ son

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) + \begin{cases} \mathbf{E_{(\Delta+i)1}} & i\ par,\ i < \Delta_m \\ \mathbf{O_{(i-(\Delta_m+1))1}} + \mathbf{E_{(i+1)1}} & i\ par,\ i > \Delta_m \end{cases} \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \begin{cases} \mathbf{O_{(\Delta+i)1}} & i\ impar,\ i < \Delta_m \\ - & i = \Delta_m \\ \mathbf{E_{(i-(\Delta_m+1))1}} + \mathbf{O_{(i+1)1}} & i\ impar,\ i > \Delta_m \end{cases} \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \mathbf{E_{(i-(\Delta_m+1))1}} & i = m-2 \\ (\mathbf{O_{m0}}) + \mathbf{O_{(i-(\Delta_m+1))1}} & i = m-1 \end{cases} \tag{9.43}$$

donde $i = 0, 1, \ldots, m-1$ y donde los términos entre paréntesis son proporcionados por el vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$, mientras que el tercer sumando (segundo, si $i = m-1$) de ambas ecuaciones pertenece al vector $\underline{\alpha}_\Omega^t \mathbf{K_1}$ y el cuarto sumando de ambas ecuaciones pertenece a $\underline{\alpha}_\Omega^t \mathbf{K_2}$.

A continuación se presenta un ejemplo de multiplicación sobre el campo de Galois $GF(2^7)$ utilizando el método transposicional, en el que se muestran las agrupaciones y comparticiones a realizar para poder reducir finalmente la complejidad total del multiplicador.

9.5.1. Ejemplo de multiplicación sobre $GF(2^7)$

El producto δ de dos elementos α y χ pertenecientes al campo $GF(2^7)$ generado por el trinomio $f(x) = x^7 + x^3 + 1$ se calcula utilizando las ecuaciones anteriores. Para este trinomio, el parámetro $\Delta = \frac{m+1}{2} = 4$ y $\Delta_m = \frac{m-3}{2} = 2$. La descomposición de la matriz $\mathbf{K}$ es de la forma:

$$\mathbf{K} = \mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2} = \begin{pmatrix} c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & c_6 \\ c_4 & c_3 & c_2 & c_1 & c_0 & c_6 & c_5 \\ c_3 & c_2 & c_1 & c_0 & c_6 & c_5 & c_4 \\ c_2 & c_1 & c_0 & c_6 & c_5 & c_4 & c_3 \\ c_1 & c_0 & c_6 & c_5 & c_4 & c_3 & c_2 \\ c_0 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c_6 & 0 & 0 & 0 \\ 0 & 0 & c_6 & c_5 & 0 & 0 & 0 \\ 0 & c_6 & c_5 & c_4 & 0 & 0 & 0 \\ c_6 & c_5 & c_4 & c_3 & 0 & 0 & 0 \\ c_5 & c_4 & c_3 & c_2 & 0 & 0 & c_6 \\ c_4 & c_3 & c_2 & c_1 & 0 & c_6 & c_5 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c_6 & 0 & 0 & 0 \\ 0 & 0 & c_6 & c_5 & 0 & 0 & 0 \end{pmatrix} \tag{9.44}$$

donde los términos c_i son las coordenadas de χ en base canónica y donde las matrices $\mathbf{K_0}$, $\mathbf{K_1}$ y $\mathbf{K_2}$ se construyen a partir de las ecuaciones 9.2, 9.3 y 9.4, respectivamente.

Los 1-ciclos y 2-ciclos dados por el método *transposicional* para este trinomio son los mismos que los calculados en el ejemplo visto en la subsección 9.4.1 para el trinomio $f(x) = x^7 + x^4 + 1$. Por lo tanto, las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ para nuestro trinomio ejemplo $f(x) = x^7 + x^3 + 1$ son las mismas que las dadas en las ecuaciones 9.30 y 9.31.

La aplicación de la ecuación 9.42 (Δ par) permite determinar las coordenadas del producto δ como sumas de términos agrupados por las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$. En la tabla 9.10 se muestran estas coordenadas para el trinomio $f(x) = x^7 + x^3 + 1$, en donde se especifican los componentes de los vectores $\underline{\alpha}^t_\Omega \mathbf{K_0}$, $\underline{\alpha}^t_\Omega \mathbf{K_1}$ y $\underline{\alpha}^t_\Omega \mathbf{K_2}$, y donde la coordenada d_i está formada por la suma de las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ existentes en la fila i-ésima.

	$\underline{\alpha}^t_\Omega \mathbf{K_0}$		$\underline{\alpha}^t_\Omega \mathbf{K_1}$	$\underline{\alpha}^t_\Omega \mathbf{K_2}$
d_0	$\mathbf{O_{10}}$	$\mathbf{O_{01}}$	$\mathbf{O_{41}}$	-
d_1	$\mathbf{E_{20}}$	$\mathbf{E_{11}}$	$\mathbf{E_{51}}$	-
d_2	$\mathbf{O_{30}}$	$\mathbf{O_{21}}$	-	-
d_3	$\mathbf{E_{40}}$	$\mathbf{E_{31}}$	$\mathbf{O_{01}}$	$\mathbf{O_{41}}$
d_4	$\mathbf{O_{50}}$	$\mathbf{O_{41}}$	$\mathbf{E_{11}}$	$\mathbf{E_{51}}$
d_5	$\mathbf{E_{60}}$	$\mathbf{E_{51}}$	$\mathbf{O_{21}}$	-
d_6	$\mathbf{O_{70}}$	-	$\mathbf{E_{31}}$	-

Tabla 9.10: Coordenadas d_i del producto para la multiplicación sobre $GF(2^7)$.

En la tabla 9.10 se muestra la *compartición* de términos que permite la reducción de la complejidad del multiplicador construido con este tipo de trinomios. Se observa que la subexpresión $(\mathbf{O_{01}} + \mathbf{O_{41}})$ perteneciente a la coordenada d_0, también aparece en la coordenada d_3, y lo mismo sucede con la subexpresión $(\mathbf{E_{11}} + \mathbf{E_{51}})$, perteneciente a d_1 y d_4. Por lo tanto, estos términos se pueden construir *una sola vez* y pueden ser *reutilizados* por las coordenadas del producto, reduciéndose de esta forma la complejidad. El número de puertas obtenido finalmente para la construcción de este multiplicador sobre $GF(2^7)$ es de 49 puertas AND y de 48 puertas XOR.

Esta forma de construcción de las coordenadas del producto también permite la reducción de la complejidad *temporal* del multiplicador por medio de la utilización de árboles *binarios* de puertas XOR para la construcción de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, para la construcción de las subexpresiones compartidas mencionadas y para la suma final de dichos términos dada en la tabla 9.10. Se obtiene finalmente que el retardo total del multiplicador es de 5 retardos de puertas XOR y de 1 retardo de puerta AND, es decir, $T_{AND} + 5T_{XOR}$.

9.5.2. Análisis teórico de complejidad

Las ecuaciones 9.42 (Δ *par*) y 9.43 (Δ *impar*) permiten calcular las coordenadas del producto $\delta = \alpha \cdot \chi$ en función de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, y su aplicación a cualquier trinomio irreducible del tipo $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m impar) determina agrupaciones de términos con la misma estructura que la mostrada en la subsección anterior para la multiplicación sobre $GF(2^7)$.

Al igual que se hizo en la sección anterior con los trinomios irreducibles $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ con m *impar*, se pueden determinar unos grupos formados por la suma de dos términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$. A estos grupos también se les denota como $\mathbf{G_i}$, donde el subíndice i coincide con el subíndice de la coordenada del producto a la cual pertenece dicho grupo. Asimismo, las expresiones que los definen coinciden con las dadas en las ecuaciones 9.32 (Δ *par*) y 9.33 (Δ *impar*) con la diferencia de que para los trinomios $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m impar), el rango de variación del subíndice i es $i = 0, 1, \ldots, m-3$. También en este caso se cumple que para $i = \Delta_m$, no existe grupo $\mathbf{G_i}$ asociado.

Utilizando las ecuaciones 9.42 y 9.43 junto con las ecuaciones 9.32 y 9.33, se pueden establecer unas nuevas expresiones de las coordenadas del producto δ en función de los grupos $\mathbf{G_i}$. Para Δ *par*, las coordenadas del producto vienen dadas por las expresiones

$$d_{\Omega_i} = \begin{cases} \mathbf{G_i} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} & i\ par \\ \mathbf{E_{(i+1)0}} & i\ impar \end{array} \right\} & i < \Delta_m \\ \mathbf{O_{(i+1)0}} + \mathbf{O_{i1}} & i = \Delta_m \\ \mathbf{G_{i-(\Delta_m+1)}} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} + \mathbf{O_{i1}} & i\ par \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} & i\ impar \end{array} \right\} & i > \Delta_m \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} + \mathbf{O_{(i-(\Delta_m+1))1}} & i = m-2 \\ \mathbf{O_{m0}} + \mathbf{E_{(i-(\Delta_m+1))1}} & i = m-1 \end{cases} \tag{9.45}$$

mientras que para Δ *impar* se tiene

$$d_{\Omega_i} = \begin{cases} \mathbf{G_i} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} & i\ par \\ \mathbf{E_{(i+1)0}} & i\ impar \end{array} \right\} & i < \Delta_m \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} & i = \Delta_m \\ \mathbf{G_{i-(\Delta_m+1)}} + \left\{ \begin{array}{ll} \mathbf{O_{(i+1)0}} + \mathbf{O_{i1}} & i\ par \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} & i\ impar \end{array} \right\} & i > \Delta_m \\ \mathbf{E_{(i+1)0}} + \mathbf{E_{i1}} + \mathbf{E_{(i-(\Delta_m+1))1}} & i = m-2 \\ \mathbf{O_{m0}} + \mathbf{O_{(i-(\Delta_m+1))1}} & i = m-1 \end{cases} \tag{9.46}$$

para $i = 0, 1, \ldots, m-1$. La arquitectura paralela del multiplicador se extrae fácilmente de estas expresiones, teniendo en cuenta la utilización de árboles *binarios* de puertas XOR para la construcción de los términos $\mathbf{E_{i(0,1)}}$, $\mathbf{O_{i(0,1)}}$ y $\mathbf{G_i}$ y para la suma final dada por las ecuaciones 9.45 y 9.46.

La complejidad *espacial* teórica del multiplicador se calcula utilizando las ecuaciones 9.32, 9.33, 9.45, 9.46 y las complejidades dadas en la tabla 9.2. Por lo tanto, la complejidad del multiplicador vendrá dada por la suma de complejidades de los términos $\mathbf{E_{i(0,1)}}$, $\mathbf{O_{i(0,1)}}$ y $\mathbf{G_i}$ necesarios, así como de la complejidad necesaria para su suma, según las ecuaciones 9.45 y 9.46.

Haciendo consideraciones similares a las hechas en la sección 9.4, se tiene que únicamente son necesarios los grupos $\mathbf{G_i}$ ($i < \Delta_m$), cuya complejidad es de 1 puerta XOR cada uno, por lo que se requieren Δ_m puertas XOR para la construcción de los $\mathbf{G_i}$ (no existe grupo $\mathbf{G_{\Delta_m}}$). Utilizando las ecuaciones 9.45 y 9.46, se observa que para la construcción del multiplicador se necesitan, además, Δ_m puertas XOR (para las coordenadas d_0 a d_{Δ_m-1}), una puerta XOR (para la coordenada d_{Δ_m}), $2\Delta_m$ puertas XOR (para las coordenadas d_{Δ_m+1} a d_{m-3}), dos puertas XOR (para d_{m-2}) y una puerta XOR (para la coordenada d_{m-1}). En el apéndice A se demuestra que la complejidad *espacial* del multiplicador es de m^2 puertas AND y de m^2-1 puertas XOR.

Para poder determinar la complejidad *temporal* teórica del multiplicador, se utiliza la ecuación 9.36 que proporciona el retardo teórico de los grupos $\mathbf{G_i}$ y que establece que el número de niveles de puertas XOR se incrementa con el decremento del índice i, por lo que los grupos $\mathbf{G_i}$ con menor subíndice son los que presentan un mayor retardo. Esto, junto con la utilización de las ecuaciones 9.45 y 9.46 y usando los resultados obtenidos en el análisis de retardos visto en la subsección 9.4.2, permite hacer las siguientes distinciones, en función del parámetro Δ_m, con respecto del retardo de las coordenadas del producto:

- Cuando $i < \Delta_m$, se utiliza la ecuación 9.37 y las consideraciones asociadas para establecer que la profundidad del árbol de puertas XOR de la coordenada d_i viene dada por $Prof(d_i) = \lceil log_2(m-i-1) \rceil + 2$.

- Para $i = \Delta_m$, se utilizan las ecuaciones 9.38 y 9.39 y las consideraciones asociadas para establecer que el retardo viene dado por las expresiones $Prof(d_i) = \lceil log_2(2\lceil \frac{i+1}{2} \rceil - 1) \rceil + 1$ (Δ *par*) y $Prof(d_i) = \lceil log_2(2\lceil \frac{i+1}{2} \rceil) \rceil + 1$ (Δ *impar*), respectivamente.

- Para $i > \Delta_m$, con $i \neq \{m-2, m-1\}$, se utiliza la ecuación 9.40 y las consideraciones asociadas para establecer que la profundidad del árbol XOR viene dada por $Prof(d_i) = \lceil log_2(m-i+\Delta_m) \rceil + 2$, independientemente del valor *par* o *impar* del parámetro Δ.

- Para $i = m-2$, se tiene de las ecuaciones 9.45 y 9.46 que la coordenada d_i es la suma de los términos $\mathbf{E_{(i+1)0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O(E)_{(i-(\Delta_m+1))1}}$ para Δ par(impar). Utilizando la tabla 9.2, se observa que el término $\mathbf{E_{(i+1)0}}$ es el que presenta un mayor número de niveles XOR, por lo que en este caso es más óptimo agrupar los términos $\mathbf{E_{i1}}$ y $\mathbf{O(E)_{(i-(\Delta_m+1))1}}$ (cuya suma iguala, en el peor caso, el retardo de $\mathbf{E_{(i+1)0}}$) y posteriormente sumarle

dicho término $\mathbf{E_{(i+1)0}}$. Se obtiene de esta forma que la profundidad del árbol XOR viene dada por

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{E_{(i+1)0}}), Prof(\mathbf{E_{i1}} + \mathbf{O(E)_{(i-(\Delta_m+1))1)}})) + 1 \\ &= Prof(\mathbf{E_{(i+1)0}}) + 1 = \lceil log_2(2\lceil \tfrac{i+1}{2} \rceil) \rceil + 1 \end{aligned} \quad (9.47)$$

independientemente del valor *par* o *impar* del parámetro Δ.

- Para $i = m-1$, se tiene de las ecuaciones 9.45 y 9.46 que la coordenada d_i es la suma de $\mathbf{O_{m0}}$ y $\mathbf{E(O)_{(i-(\Delta_m+1))1}}$ para Δ par(impar). De la tabla 9.2 se obtiene que el término $\mathbf{O_{m0}}$ es el de mayor complejidad *temporal*, por lo que el retardo viene dado como

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{O_{m0}}), Prof(\mathbf{E(O)_{(m-\Delta_m-2)1}})) + 1 \\ &= max(\lceil log_2 m \rceil, \lceil log_2(\tfrac{m-1}{2}) \rceil) + 1 = \lceil log_2 m \rceil + 1 \end{aligned} \quad (9.48)$$

donde se ha utilizado el valor de $\Delta_m = \frac{m-3}{2}$ para este tipo de trinomios.

La complejidad temporal del multiplicador vendrá determinada por la máxima complejidad temporal de entre las establecidas anteriormente para las distintas coordenadas, obteniéndose el retardo máximo $\lceil log_2(m-i-1) \rceil + 2$ para las coordenadas $d_i, i < \Delta_m$ (ya que $\lceil log_2(m-i+\Delta_m) \rceil + 2$ coincide, para valores de $i > \Delta_m$, con este retardo máximo). El valor máximo del retardo se puede escoger para la coordenada d_0 debido a que los grupos $\mathbf{G_i}$ con menor subíndice son los que presentan un mayor retardo, estableciéndose $2 + \lceil log_2(m-1) \rceil$ como retardo máximo. Finalmente, la complejidad teórica *temporal* vendrá dada por este número de niveles de puertas XOR, a los que se añade un nivel de puertas AND que realizan los productos $a_i c_j$ de las coordenadas de los elementos α y χ de $GF(2^m)$. Por lo tanto, el *retardo* total del multiplicador será $T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$, que coincide con el retardo obtenido para los trinomios $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ con m *impar*.

	#XOR	#AND	Retardo
[HK00]	$m^2 - 1$	m^2	$T_{AND} + (\lceil \frac{2(m-1)}{m+1} \rceil + \lceil log_2 m \rceil) T_{XOR}$
[ZP01]	$m^2 - 1$	m^2	$T_{AND} + (\lfloor \frac{2(m-2)}{m+1} \rfloor + 1 + \lceil log_2 m \rceil) T_{XOR}$
Ntra.Ap.	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$

Tabla 9.11: Complejidades teóricas de multiplicadores canónicos usando trinomios irreducibles $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m *impar*).

En la tabla 9.11 se comparan las complejidades teóricas obtenidas por nuestro método (*Ntra.Ap.*) con los mejores resultados dados en la literatura por Halbutogullari y Koç [HK00] y por Zhang y Parhi [ZP01] para la construcción de un multiplicador paralelo en base canónica generado por trinomios irreducibles del tipo $f(x) = x^m + x^{\frac{m-1}{2}} + 1$, para m *impar*.

Al igual que sucedía con los trinomios estudiados en las secciones anteriores, se tiene que los resultados teóricos mostrados en la tabla 9.11 corresponden a complejidades en las que se mantiene un equilibrio entre el número de puertas XOR y el retardo del multiplicador. En [ZP01] se da otro resultado en el que se reduce el número de puertas XOR a costa de un incremento en el retardo.

En la tabla 9.12 se comparan los retardos de puertas XOR (T_{XOR}) obtenidos por nuestra aproximación *transposicional* con los obtenidos por el método propuesto por Halbutogullari y Koç en [HK00] para todos los trinomios irreducibles $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m *impar*) existentes hasta el grado $m = 100$. También se muestra el porcentaje de mejora obtenida por nuestro método, así como los valores totales. Los resultados teóricos dados por Zhang y Parhi en [ZP01] coinciden exactamente con los obtenidos por Halbutogullari y Koç.

		Ntra.Ap.	Halbut.-Koç	Mejora
m	f(x)	T_{XOR}	T_{XOR}	%
3	3,1,0	3	3	0.0
5	5,2,0	4	5	20.0
7	7,3,0	5	5	0.0
9	9,4,0	5	6	16.7
15	15,7,0	6	6	0.0
41	41,20,0	8	8	0.0
63	63,31,0	8	8	0.0
65	65,32,0	8	9	11.1
71	71,35,0	9	9	0.0
Total		56	59	5.1

Tabla 9.12: Comparación de retardos T_{XOR} de multiplicadores obtenidos con nuestra aproximación y por Halbutogullari y Koç para distintos trinomios irreducibles.

Los resultados dados en la tabla 9.12 muestran que el método *transposicional* obtiene una menor reducción en el retardo de puertas XOR que la obtenida en la sección 9.3, debido a una menor compartición de grupos $\mathbf{G_i}$ y a la alta optimización con respecto a T_{XOR} de los resultados dados en esta tabla 9.12, lo que dificulta la obtención de mayores reducciones del retardo. A pesar de esto y para el total de polinomios mostrados, se obtiene una mejora con respecto al método dado en [HK00] del 5.1 %. Si se consideran únicamente los multiplicadores para los que se reduce el retardo (para $m = 5, 9$ y 65), entonces nuestro método obtiene una mejora en la complejidad temporal del 15.0 %.

Se puede observar que estos resultados *teóricos* son similares a los obtenidos en la sección anterior, ya que la creación y compartición de grupos $\mathbf{G_i}$ es similar a la obtenida con los trinomios $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m *impar*). También se puede observar en la tabla 9.12 que el trinomio $f(x) = x^3 + x + 1$ pertenece a los trinomios del tipo $f(x) = x^m + x + 1$ y que se estudiarán en la sección 9.7.

9.5.3. Resultados experimentales

Se ha realizado la implementación paralela sobre FPGAs de multiplicadores canónicos utilizando el método *transposicional* dado por las ecuaciones 9.45 y 9.46 y utilizando el método propuesto por Halbutogullari y Koç en [HK00]. Se ha empleado *Xilinx Foundation F2.1i* y se han realizado descripciones VHDL de los multiplicadores paralelos utilizando ambos métodos para los campos $GF(2^m)$ generados por trinomios irreducibles $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ para grados *impares* $m = 5$, 7, 9 y 15. Los dispositivos utilizados han sido los 4013XLPQ160 pertenecientes a la familia XC4000XL y se ha realizado la síntesis utilizando optimización para *Area* con un nivel de esfuerzo *High*. Asimismo, las métricas del rendimiento utilizadas han sido el *número de CLBs* y el *retardo máximo combinacional*, obteniéndose los resultados experimentales mostrados en la tabla 9.13.

	Halbut.-Koç		Transposicional	
	CLBs	Ret. Max (ns)	CLBs	Ret. Max (ns)
$GF(2^5)$	11	17.1	11	20.0
$GF(2^7)$	23	19.6	20	20.5
$GF(2^9)$	33	24.8	30	24.2
$GF(2^{15})$	90	30.4	80	29.2
Total	157	91.9	141	93.9

Tabla 9.13: Resultados experimentales de multiplicadores canónicos sobre campos generados por trinomios irreducibles $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ *(m impar)*.

De los resultados mostrados en la tabla 9.13, se observa que la implementación de los multiplicadores en base canónica utilizando nuestro método *transposicional* consume un menor número de CLBs que si se utiliza la aproximación dada por Halbutogullari y Koç. Considerando el número total de CLBs para todas las implementaciones, se observa que nuestro método utiliza un 10.2 % menos de CLBs, siendo el mejor resultado el correspondiente al campo $GF(2^7)$ con una reducción del 13.0 %. Sin embargo, se observa en la tabla que los retardos obtenidos con nuestra aproximación son un 2.2 % peores que los conseguidos por el otro método, hecho que no se corresponde con el resultado teórico obtenido anteriormente de un mejor comportamiento temporal del método transposicional. Tampoco coincide el mejor resultado experimental obtenido por nuestro método en cuanto al número de CLBs con la igualdad *espacial* teórica mostrada en la tabla 9.11.

La mejora de la complejidad espacial *experimental* obtenida por nuestro método para este tipo de trinomios es similar a la obtenida para los trinomios estudiados en la sección anterior, pudiéndose hacer consideraciones similares con respecto a esta mejora.

9.6. Trinomios irreducibles $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m par)

Los trinomios irreducibles de la forma $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m par) son un tipo especial de ESPs (*Equally-Spaced-Polynomials*) conocidos como ESTs (*Equally-Spaced-Trinomials*). Los ESPs irreducibles son polinomios de la forma $f(x) = x^{k\Psi} + x^{(k-1)\Psi} + \cdots + x^{\Psi} + 1$ donde $m = k\Psi$ y $k \geq 2$. Los ESPs se reducen a los ESTs cuando $k = 2$, y se reducen a los AOPs para $\Psi = 1$.

Para el EST $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m par) se tiene que el parámetro $\Delta = \frac{m}{2}$ y que $\Delta_m = \frac{m-2}{2} = \Delta - 1$. Por otra parte, se tiene que $\tau = \lceil \frac{2(m-1)}{m} \rceil$, pudiéndose comprobar que para valores *pares* de m con $m > 2$, el parámetro $\tau = 2$, por lo que la descomposición de la matriz $\mathbf{K}$ dada en la ecuación 9.1 está formada por la suma de las tres matrices $\mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2}$. Sin embargo, para $m = 2$, el parámetro $\tau = 1$ y la matriz $\mathbf{K}$ se descompone en la suma de dos matrices $\mathbf{K_0} + \mathbf{K_1}$ únicamente (se observa que $f(x) = x^2 + x + 1$ pertenece al grupo de trinomios de la forma $f(x) = x^m + x + 1$ que se estudiará en la sección 9.7).

Teniendo en cuenta la forma de las matrices $\mathbf{K_0}$ y $\mathbf{K_2}$ dadas en las ecuaciones 9.2 y 9.4, respectivamente, y utilizando la numeración de columnas usada en la sección 9.1, se observa que la columna $(n+1)$-ésima de $\mathbf{K_0}$ es de la forma $(c_n, c_{n-1}, \ldots, c_0, c_{m-1}, \ldots, c_{n+2}, c_{n+1})$, mientras que la columna $(n+1)$-ésima de $\mathbf{K_2}$ es de la forma $(0, 0, \ldots, 0, c_{m-1}, \ldots, c_{n+2}, c_{n+1})$. Por lo tanto, la suma de $\mathbf{K_0}$ y $\mathbf{K_2}$ hace que la columna $(n+1)$-ésima resultante presente términos nulos debido a la aparición de expresiones $c_i + c_i = 0$, obteniéndose como resultado la columna $(c_n, c_{n-1}, \ldots, c_0, 0, \ldots, 0, 0)$. Cancelaciones similares ocurren al realizarse la suma de las restantes $n-2$ columnas no nulas de $\mathbf{K_2}$ con las columnas correspondientes de $\mathbf{K_0}$, pudiéndose considerar que el resultado de la suma es la matriz $\mathbf{K_0}$ con una serie de elementos nulos correspondientes a las posiciones no nulas de la matriz $\mathbf{K_2}$. Este hecho hace que la complejidad *espacial* de los multiplicadores sobre ESTs sea menor que la del resto de trinomios estudiados, como se verá posteriormente en el análisis de complejidad.

Utilizando estas consideraciones, se obtienen las siguientes expresiones que permiten el cálculo de las coordenadas d_{Ω_i} del producto δ por el método *transposicional* en función de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$:

$$d_{\Omega_i} = \begin{cases} \mathbf{O_{(i+1)0}} + \begin{cases} \mathbf{E_{i1}} + \mathbf{O_{(\Delta+i)1}} & i \; par, \; i < \Delta_m \\ \mathbf{E_{i1}} & i = \Delta_m \\ \mathbf{O_{(i-(\Delta_m+1))1}} & i \; par, \; i > \Delta_m \end{cases} \\ \mathbf{E_{(i+1)0}} + \begin{cases} \mathbf{O_{i1}} + \mathbf{E_{(\Delta+i)1}} & i \; impar, \; i < \Delta_m \\ \mathbf{E_{(i-(\Delta_m+1))1}} & i \; impar, \; i > \Delta_m \end{cases} \end{cases} \tag{9.49}$$

para valores de $i = 0, 1, \ldots, m-1$. Estas ecuaciones son válidas para Δ *par*, condición que se cumple para todos los ESTs irreducibles.

A continuación se presenta un ejemplo de multiplicación sobre el campo de Galois $GF(2^6)$ utilizando el método transposicional.

9.6.1. Ejemplo de multiplicación sobre $GF(2^6)$

El producto δ de dos elementos α y χ pertenecientes al campo $GF(2^6)$ generado por el EST irreducible $f(x) = x^6 + x^3 + 1$ se puede calcular utilizando la ecuación 9.49. Para este trinomio, se tiene que $\Delta = 3$ y $\Delta_m = 2$. La matriz $\mathbf{K}$ se descompone de la siguiente forma:

$$\mathbf{K} = \mathbf{K_0} + \mathbf{K_1} + \mathbf{K_2} = \begin{pmatrix} c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ c_4 & c_3 & c_2 & c_1 & c_0 & c_5 \\ c_3 & c_2 & c_1 & c_0 & c_5 & c_4 \\ c_2 & c_1 & c_0 & c_5 & c_4 & c_3 \\ c_1 & c_0 & c_5 & c_4 & c_3 & c_2 \\ c_0 & c_5 & c_4 & c_3 & c_2 & c_1 \end{pmatrix} +$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_5 & 0 & 0 & 0 \\ 0 & c_5 & c_4 & 0 & 0 & 0 \\ c_5 & c_4 & c_3 & 0 & 0 & 0 \\ c_4 & c_3 & c_2 & 0 & 0 & c_5 \\ c_3 & c_2 & c_1 & 0 & c_5 & c_4 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_5 & 0 & 0 & 0 \\ 0 & c_5 & c_4 & 0 & 0 & 0 \end{pmatrix} = \tag{9.50}$$

$$\begin{pmatrix} c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ c_4 & c_3 & c_2 & c_1 & c_0 & c_5 \\ c_3 & c_2 & c_1 & c_0 & c_5 & c_4 \\ c_2 & c_1 & c_0 & c_5 & c_4 & c_3 \\ c_1 & c_0 & 0 & c_4 & c_3 & c_2 \\ c_0 & 0 & 0 & c_3 & c_2 & c_1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_5 & 0 & 0 & 0 \\ 0 & c_5 & c_4 & 0 & 0 & 0 \\ c_5 & c_4 & c_3 & 0 & 0 & 0 \\ c_4 & c_3 & c_2 & 0 & 0 & c_5 \\ c_3 & c_2 & c_1 & 0 & c_5 & c_4 \end{pmatrix}$$

donde los términos c_i son las coordenadas de χ en la base canónica y donde se observa que la suma de las matrices $\mathbf{K_0}$ y $\mathbf{K_2}$ produce una nueva matriz igual a $\mathbf{K_0}$ con términos nulos en las posiciones no nulas de $\mathbf{K_2}$.

Los 1-ciclos y 2-ciclos dados por el método *transposicional* para este trinomio son los mismos que los calculados en el ejemplo visto en la subsección 9.3.1 para el trinomio $f(x) = x^6 + x^5 + 1$. Por lo tanto, las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ para nuestro trinomio ejemplo $f(x) = x^6 + x^3 + 1$ son las mismas que las dadas en las ecuaciones 9.19 y 9.20.

En la tabla 9.14 se muestran las coordenadas del producto δ para el EST $f(x) = x^6 + x^3 + 1$. A la izquierda de la tabla se especifican los componentes de los vectores $\underline{\alpha}_\Omega^t \mathbf{K_0}$, $\underline{\alpha}_\Omega^t \mathbf{K_1}$ y $\underline{\alpha}_\Omega^t \mathbf{K_2}$, donde la suma de funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ existentes en la fila i-ésima determina la coordenada d_i. A la derecha de la tabla se muestran los componentes resultantes de la simplificación realizada al observar que los términos $\mathbf{O_{31}}$ y $\mathbf{E_{41}}$ aparecen sumados dos veces en las coordenadas d_3 y d_4, respectivamente, a la izquierda de la tabla. A la derecha de la tabla se muestran los componentes de los vectores $\underline{\alpha}_\Omega^t \mathbf{K_0'}$ (resultante de la simplificación anterior) y $\underline{\alpha}_\Omega^t \mathbf{K_1}$, cuya suma de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ para el cálculo de las coordenadas d_i es la proporcionada por la ecuación 9.49.

	$\underline{\alpha}^t_\Omega \mathbf{K_0}$		$\underline{\alpha}^t_\Omega \mathbf{K_1}$	$\underline{\alpha}^t_\Omega \mathbf{K_2}$		$\underline{\alpha}^t_\Omega \mathbf{K'_0}$		$\underline{\alpha}^t_\Omega \mathbf{K_1}$
d_0	$\mathbf{O_{10}}$	$\mathbf{E_{01}}$	$\mathbf{O_{31}}$	-		$\mathbf{O_{10}}$	$\mathbf{E_{01}}$	$\mathbf{O_{31}}$
d_1	$\mathbf{E_{20}}$	$\mathbf{O_{11}}$	$\mathbf{E_{41}}$	-		$\mathbf{E_{20}}$	$\mathbf{O_{11}}$	$\mathbf{E_{41}}$
d_2	$\mathbf{O_{30}}$	$\mathbf{E_{21}}$	-	-	=	$\mathbf{O_{30}}$	$\mathbf{E_{21}}$	-
d_3	$\mathbf{E_{40}}$	$\mathbf{O_{31}}$	$\mathbf{E_{01}}$	$\mathbf{O_{31}}$		$\mathbf{E_{40}}$	-	$\mathbf{E_{01}}$
d_4	$\mathbf{O_{50}}$	$\mathbf{E_{41}}$	$\mathbf{O_{11}}$	$\mathbf{E_{41}}$		$\mathbf{O_{50}}$	-	$\mathbf{O_{11}}$
d_5	$\mathbf{E_{60}}$	-	$\mathbf{E_{21}}$	-		$\mathbf{E_{60}}$	-	$\mathbf{E_{21}}$

Tabla 9.14: Coordenadas d_i del producto para la multiplicación sobre $GF(2^6)$.

En la parte derecha de la tabla 9.14 se observa que no se pueden formar grupos $\mathbf{G_i}$ de términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$ que puedan ser compartidos entre varias coordenadas, por lo que para este tipo de trinomios no se cumple la propiedad de *compartición*, aunque sí se da la *agrupación* proporcionada por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$.

Se puede comprobar que el número de puertas necesarias para la construcción de este multiplicador sobre $GF(2^6)$ es de 36 puertas AND y de 33 puertas XOR. Con respecto al retardo del multiplicador, se tiene que es de 4 retardos de puertas XOR y de 1 retardo de puertas AND, es decir, $T_{AND} + 4T_{XOR}$. Para la obtención de este resultado, ha sido más óptimo realizar la suma en primer lugar de los términos $\mathbf{O_{10}} + \mathbf{O_{31}}$ (en d_0) y $\mathbf{E_{20}} + \mathbf{E_{41}}$ (en d_1) y posteriormente hacer su suma con $\mathbf{E_{01}}$ y $\mathbf{O_{11}}$, respectivamente, para la obtención de las coordenadas d_0 y d_1. Esto es posible ya que para los ESTs ya no existe la restricción impuesta por los grupos $\mathbf{G_i}$ con respecto de la agrupación de términos y cuya finalidad era la *compartición* de expresiones comunes entre las distintas coordenadas.

9.6.2. Análisis teórico de complejidad

Como se ha mencionado en el ejemplo anterior, se tiene que para la multiplicación canónica utilizando ESTs no existen grupos $\mathbf{G_i}$ formados por sumas de términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$ que puedan ser *compartidos* entre varias coordenadas del producto, por lo que es directamente la ecuación 9.49 la que determina las coordenadas de dicho producto.

La complejidad *teórica* del multiplicador se calcula, por tanto, utilizando la ecuación 9.49 y las complejidades dadas en la tabla 9.2. En el apéndice A se demuestra que la complejidad *espacial* del multiplicador es de m^2 puertas AND y de $m^2 - \Delta$ puertas XOR, siendo, por tanto, menor que en el resto de trinomios estudiados hasta el momento.

Para poder determinar la complejidad *temporal* teórica del multiplicador, se tiene en cuenta la observación realizada en el ejemplo anterior con respecto del orden a utilizar en la suma de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$. Se realizan, por tanto, las siguientes distinciones en función del parámetro Δ_m, con respecto del retardo de las coordenadas del producto:

- Cuando $i < \Delta_m$, se tiene de la ecuación 9.49 que los términos $\mathbf{E_{i1}}$ (i par) y $\mathbf{O_{i1}}$ (i impar) son los que presentan un mayor número de niveles de puertas XOR. Por este motivo, es más óptimo realizar en primer lugar la suma de términos $\mathbf{O_{(i+1)0}}+\mathbf{O_{(\Delta+i)1}}$ (i par) y $\mathbf{E_{(i+1)0}}+\mathbf{E_{(\Delta+i)1}}$ (i impar) y posteriormente sumarles los términos $\mathbf{E_{i1}}$ o $\mathbf{O_{i1}}$ correspondientes. De esta forma, y como el retardo XOR de la primera suma $\mathbf{O(E)_{(i+1)0}} + \mathbf{O(E)_{(\Delta+i)1}}$ es menor o igual que el número de niveles XOR del término $\mathbf{E(O)_{i1}}$ correspondiente, se toma la complejidad de este último para determinar la profundidad del árbol de puertas XOR de la coordenada d_i como

$$Prof(d_i) = Prof(\mathbf{E(O)_{i1}}) + 1 = \lceil log_2(m - i - 1) \rceil + 1 \qquad (9.51)$$

  donde se han utilizado los resultados dados en la tabla 9.2.

- Para $i = \Delta_m$ y utilizando la ecuación 9.49, se tiene que la coordenada d_i es la suma de los términos $\mathbf{O_{(i+1)0}}$ y $\mathbf{E_{i1}}$, por lo que

$$Prof(d_i) = max(Prof(\mathbf{O_{(i+1)0}}), Prof(\mathbf{E_{i1}})) + 1 = \lceil log_2(\tfrac{m}{2}) \rceil + 1 \quad (9.52)$$

  donde se ha utilizado el hecho de que el número de niveles XOR de ambas funciones sean iguales para valores de m *pares* y que para un EST el parámetro $\Delta_m = \frac{m}{2} - 1$.

- Para $i > \Delta_m$, se observa en la ecuación 9.49 que las coordenadas d_i vienen dadas por la suma de términos $\mathbf{O_{(i+1)0}} + \mathbf{O_{(i-(\Delta_m+1))1}}$ (i par) y $\mathbf{E_{(i+1)0}}+\mathbf{E_{(i-(\Delta_m+1))1}}$ (i impar). Como los términos $\mathbf{E_{i0}}$ son más complejos que los términos $\mathbf{O_{i0}}$ (véase tabla 9.2), se tiene que

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{E_{(i+1)0}}), Prof(\mathbf{E_{(i-(\Delta_m+1))1}})) + 1 \\ &= Prof(\mathbf{E_{(i+1)0}}) + 1 = \lceil log_2(2\lceil \tfrac{i+1}{2} \rceil) \rceil + 1 \end{aligned} \qquad (9.53)$$

  donde se ha utilizado la mayor complejidad de los términos $\mathbf{E_{i0}}$ frente a los términos $\mathbf{E_{(i-(\Delta_m+1))1}}$.

La complejidad temporal del multiplicador vendrá determinada por la máxima complejidad temporal de entre las establecidas anteriormente para las distintas coordenadas, observándose que el retardo máximo corresponde a la coordenada d_{m-1} cuya profundidad del árbol XOR es $\lceil log_2 m \rceil + 1$ (obtenida de la ecuación 9.53 para $i = m - 1$ con m par).

Finalmente, la complejidad teórica *temporal* vendrá dada por este número de niveles de puertas XOR, a los que se añade un nivel de puertas AND que realizan los productos $a_i c_j$ de las coordenadas de los elementos α y χ de $GF(2^m)$. Por lo tanto, el *retardo* total del multiplicador será $T_{AND} + (1 + \lceil log_2 m \rceil) T_{XOR}$, que coincide con los retardos obtenidos por Halbutogullari y Koç [HK00] y por Zhang y Parhi [ZP01] para este tipo especial de trinomios.

En la tabla 9.15 se comparan las complejidades teóricas obtenidas por nuestro método (*Ntra.Ap.*) con los mejores resultados encontrados en la literatura para la construcción de un multiplicador paralelo en base canónica generado por un EST irreducible $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m *par*), observándose que las complejidades obtenidas por nuestro método *transposicional* son idénticas a las obtenidas en [HK00] y en [ZP01].

	#XOR	#AND	Retardo
[HK00]	$m^2 - \Delta$	m^2	$T_{AND} + (1 + \lceil log_2 m \rceil) T_{XOR}$
[ZP01]	$m^2 - \Delta$	m^2	$T_{AND} + (1 + \lceil log_2 m \rceil) T_{XOR}$
Ntra.Ap.	$m^2 - \Delta$	m^2	$T_{AND} + (1 + \lceil log_2 m \rceil) T_{XOR}$

Tabla 9.15: Complejidades teóricas de multiplicadores canónicos utilizando ESTs irreducibles.

Los ESTs son polinomios poco importantes con respecto a la generación de campos de Galois, ya que existen muy pocos ESTs irreducibles. Concretamente, para ESTs de grado hasta 1000, únicamente existen seis ESTs irreducibles (para m = 2, 6, 18, 54, 162 y 486). Por este motivo, no hemos realizado implementaciones reconfigurables de multiplicadores paralelos sobre este tipo de trinomios irreducibles.

9.7. Trinomios irreducibles $f(x) = x^m + x + 1$

Cuando el trinomio irreducible generador del campo finito es de la forma $f(x) = x^m + x + 1$, se tiene que el parámetro $\Delta = m - 1$ y, por lo tanto, $\tau = \lceil \frac{m-1}{\Delta} \rceil = 1$. La descomposición de la matriz $\mathbf{K}$ dada en la ecuación 9.1 estará formada en este caso por la suma de las dos matrices $\mathbf{K_0} + \mathbf{K_1}$. Además, se tiene que para estos trinomios el parámetro $\Delta_m = (m-1) - \Delta = 0$.

Las expresiones que permiten el cálculo de las coordenadas d_{Ω_i} del producto δ por el método *transposicional* en función de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ para valores del parámetro Δ *pares*, vienen dadas por

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) & i = 0 \\ (\mathbf{O_{(i+1)0}} + \mathbf{O_{i1}}) + \mathbf{E_{(i-1)1}} & i \; par \\ (\mathbf{E_{(i+1)0}} + \mathbf{E_{i1}}) + \mathbf{O_{(i-1)1}} & i \; impar \\ (\mathbf{O_{(i+1)0}}) + \mathbf{E_{(i-1)1}} & i = m-1 \end{cases} \tag{9.54}$$

mientras que las coordenadas para valores *impares* de Δ son

$$d_{\Omega_i} = \begin{cases} (\mathbf{O_{(i+1)0}} + \mathbf{E_{i1}}) & i = 0 \\ (\mathbf{O_{(i+1)0}} + \mathbf{E_{i1}}) + \mathbf{O_{(i-1)1}} & i\ par \\ (\mathbf{E_{(i+1)0}} + \mathbf{O_{i1}}) + \mathbf{E_{(i-1)1}} & i\ impar \\ (\mathbf{E_{(i+1)0}}) + \mathbf{E_{(i-1)1}} & i = m-1 \end{cases} \tag{9.55}$$

donde $i = 0, 1, \ldots, m-1$ y donde los términos entre paréntesis son proporcionados por el vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$, mientras que el tercer sumando (segundo, para $i = m-1$) de ambas ecuaciones pertenece al vector $\underline{\alpha}_\Omega^t \mathbf{K_1}$.

A continuación se presenta un ejemplo de multiplicación sobre el campo de Galois $GF(2^6)$ utilizando el método transposicional.

9.7.1. Ejemplo de multiplicación sobre $GF(2^6)$

El producto δ de dos elementos α y χ pertenecientes al campo $GF(2^6)$ generado por el trinomio irreducible $f(x) = x^6 + x + 1$, para el que $\Delta = 5$, se puede calcular utilizando la ecuación 9.55. La matriz $\mathbf{K}$ se descompone de la siguiente forma:

$$\mathbf{K} = \mathbf{K_0} + \mathbf{K_1} = \begin{pmatrix} c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ c_4 & c_3 & c_2 & c_1 & c_0 & c_5 \\ c_3 & c_2 & c_1 & c_0 & c_5 & c_4 \\ c_2 & c_1 & c_0 & c_5 & c_4 & c_3 \\ c_1 & c_0 & c_5 & c_4 & c_3 & c_2 \\ c_0 & c_5 & c_4 & c_3 & c_2 & c_1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_5 & 0 \\ 0 & 0 & 0 & c_5 & c_4 & 0 \\ 0 & 0 & c_5 & c_4 & c_3 & 0 \\ 0 & c_5 & c_4 & c_3 & c_2 & 0 \\ c_5 & c_4 & c_3 & c_2 & c_1 & 0 \end{pmatrix} \tag{9.56}$$

donde los términos c_i son las coordenadas de χ en la base canónica y donde las matrices $\mathbf{K_0}$ y $\mathbf{K_1}$ se construyen a partir de las ecuaciones 9.2 y 9.3, respectivamente.

Los 1-ciclos y 2-ciclos dados por el método *transposicional* para este trinomio son los mismos que los calculados en el ejemplo visto en la subsección 9.3.1 para el trinomio $f(x) = x^6 + x^5 + 1$. Por lo tanto, las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ para nuestro trinomio ejemplo $f(x) = x^6 + x + 1$ son las mismas que las dadas en las ecuaciones 9.19 y 9.20.

En la tabla 9.16 se muestran las coordenadas del producto δ resultantes de aplicar la ecuación 9.55 al trinomio irreducible $f(x) = x^6 + x + 1$, donde se especifican los componentes de los vectores $\underline{\alpha}_\Omega^t \mathbf{K_0}$ y $\underline{\alpha}_\Omega^t \mathbf{K_1}$ y donde la suma de funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ en la fila i-ésima determina la coordenada d_i.

En la tabla 9.16 se observa que no existen grupos $\mathbf{G_i}$ formados por sumas de términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$ que se puedan compartir entre varias coordenadas, por lo que para este tipo de trinomios no se cumple la propiedad de *compartición*, aunque sí se da la *agrupación* proporcionada por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$.

	$\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$		$\underline{\alpha}_{\Omega}^{t}\mathbf{K_1}$
d_0	$\mathbf{O_{10}}$	$\mathbf{E_{01}}$	-
d_1	$\mathbf{E_{20}}$	$\mathbf{O_{11}}$	$\mathbf{E_{01}}$
d_2	$\mathbf{O_{30}}$	$\mathbf{E_{21}}$	$\mathbf{O_{11}}$
d_3	$\mathbf{E_{40}}$	$\mathbf{O_{31}}$	$\mathbf{E_{21}}$
d_4	$\mathbf{O_{50}}$	$\mathbf{E_{41}}$	$\mathbf{O_{31}}$
d_5	$\mathbf{E_{60}}$	-	$\mathbf{E_{41}}$

Tabla 9.16: Coordenadas d_i del producto para la multiplicación sobre $GF(2^6)$.

Asimismo, se puede comprobar que el número de puertas necesarias para la construcción de este multiplicador paralelo sobre $GF(2^6)$ es de 36 puertas AND y de 35 puertas XOR. Con respecto al retardo del multiplicador, se tiene que es de 4 retardos de puertas XOR y de 1 retardo de puertas AND, es decir, $T_{AND} + 4T_{XOR}$.

9.7.2. Análisis teórico de complejidad

Como se ha mencionado en el ejemplo anterior, se tiene que para la multiplicación canónica utilizando los trinomios irreducibles $f(x) = x^m + x + 1$ no se pueden formar grupos $\mathbf{G_i}$ de términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$ que puedan ser *compartidos* entre varias coordenadas del producto, por lo que son directamente las ecuaciones 9.54 (Δ par) y 9.55 (Δ impar) las que determinan las coordenadas de dicho producto.

La complejidad *teórica* del multiplicador se calcula, por tanto, utilizando las ecuaciones 9.54 y 9.55, junto con las complejidades dadas en la tabla 9.2. En el apéndice A se demuestra que la complejidad *espacial* del multiplicador es de m^2 puertas AND y de $m^2 - 1$ puertas XOR.

Para determinar la complejidad *temporal* teórica del multiplicador, se realizan las siguientes distinciones con respecto de las coordenadas del producto:

- Para $i = 0$, se tiene de las ecuaciones 9.54 y 9.55 que la coordenada d_i está formada por la suma de $\mathbf{O_{(i+1)0}}$ y $\mathbf{O(E)_{i1}}$ para Δ par (impar), por lo que la profundidad del árbol de puertas XOR de la coordenada d_i es

$$\begin{aligned} Prof(d_i) &= max(Prof(\mathbf{O_{(i+1)0}}), Prof(\mathbf{O(E)_{i1}})) + 1 \\ &= Prof(\mathbf{O(E)_{i1}}) + 1 = \lceil log_2(m-1) \rceil + 1 \end{aligned} \tag{9.57}$$

  donde se ha utilizado el hecho de que $i = 0$, así como el mayor retardo de los términos $\mathbf{O(E)_{i1}}$ para dicha coordenada.

- Para $i \neq \{0, m-1\}$ con Δ *par*, se observa que (utilizando las ecuaciones 9.54 y 9.55) las coordenadas están formadas por la suma de tres términos. Se puede comprobar que el retardo XOR de las coordenadas d_i es menor o igual que el retardo de la coordenada d_1, por lo que esta será la que determine el máximo retardo. En este caso, la coordenada d_1 está formada por la suma de los términos $\mathbf{E_{20}}$, $\mathbf{E_{11}}$ y $\mathbf{O_{01}}$ (para cualquier m), donde $\mathbf{E_{11}}$ y/o $\mathbf{O_{01}}$ presentan el mayor número de niveles de puertas XOR. Si $Prof(\mathbf{E_{11}}) = Prof(\mathbf{O_{01}})$, entonces se tiene la siguiente expresión para la profundidad del árbol XOR de las coordenadas d_i

$$Prof(d_i) \leq Prof(\mathbf{O_{01}}) + 2 = \lceil log_2(m-1) \rceil + 2 \tag{9.58}$$

  donde se han utilizado las complejidades mostradas en la tabla 9.2. Si por el contrario, se tiene que $Prof(\mathbf{E_{11}}) < Prof(\mathbf{O_{01}})$, entonces resulta más óptimo realizar en primer lugar la suma de los términos $\mathbf{E_{11}}$ y $\mathbf{E_{20}}$, y posteriormente sumar el término $\mathbf{O_{01}}$. De esta forma, se obtendría que la profundidad del árbol binario de puertas XOR de las coordenadas d_i vendría dada como

$$Prof(d_i) \leq Prof(\mathbf{O_{01}}) + 1 = \lceil log_2(m-1) \rceil + 1 \tag{9.59}$$

  utilizando también las complejidades dadas en la tabla 9.2.

- Para $i \neq \{0, m-1\}$ con Δ *impar*, las ecuaciones 9.54 y 9.55 determinan que las coordenadas están formadas por la suma de tres términos. También se puede comprobar que el retardo XOR de las coordenadas d_i es menor o igual que el retardo de la coordenada d_1, por lo que esta determina el máximo retardo. En este caso, la coordenada d_1 está formada por la suma de los términos $\mathbf{E_{20}}$, $\mathbf{O_{11}}$ y $\mathbf{E_{01}}$ (para cualquier m), donde $\mathbf{O_{11}}$ y/o $\mathbf{E_{01}}$ presentan el mayor número de niveles de puertas XOR. Si $Prof(\mathbf{O_{11}}) = Prof(\mathbf{E_{01}})$, entonces se tiene la siguiente expresión para la profundidad del árbol XOR de las coordenadas d_i

$$Prof(d_i) \leq Prof(\mathbf{E_{01}}) + 2 = \lceil log_2(m-1) \rceil + 2 \tag{9.60}$$

  donde se han utilizado las complejidades dadas en la tabla 9.2. En cambio, si $Prof(\mathbf{O_{11}}) < Prof(\mathbf{E_{01}})$, entonces resulta más óptimo realizar en primer lugar la suma de $\mathbf{O_{11}}$ con $\mathbf{E_{20}}$, y posteriormente sumar el término $\mathbf{E_{01}}$. De esta forma, se obtiene que la profundidad del árbol XOR de las coordenadas d_i es

$$Prof(d_i) \leq Prof(\mathbf{E_{01}}) + 1 = \lceil log_2(m-1) \rceil + 1 \tag{9.61}$$

  donde se han utilizado las complejidades dadas en la tabla 9.2.

- Cuando $i = m-1$, se observa en las ecuaciones 9.54 y 9.55 que la coordenada d_{m-1} viene dada por la suma de los términos $\mathbf{O_{m0}} + \mathbf{E_{(m-2)1}}$ si Δ es *par*, y por la suma $\mathbf{E_{m0}} + \mathbf{E_{(m-2)1}}$ si Δ es *impar*. De ambas expresiones, los términos más complejos corresponden a los $\mathbf{O(E)_{m0}}$, por lo que estos términos serán los que determinen la complejidad temporal de d_{m-1}. Por lo tanto, el número de niveles de puertas XOR para esta coordenada cuando Δ es *par*, vendrá dado por

$$Prof(d_{m-1}) = Prof(\mathbf{O_{m0}}) + 1 = \lceil log_2(2\lceil \tfrac{m}{2} \rceil - 1) \rceil + 1 \qquad (9.62)$$

mientras que si Δ es *impar* se tiene que

$$Prof(d_{m-1}) = Prof(\mathbf{E_{m0}}) + 1 = \lceil log_2(2\lceil \tfrac{m}{2} \rceil) \rceil + 1 \qquad (9.63)$$

donde se han utilizado las complejidades dadas en la tabla 9.2.

La complejidad temporal del multiplicador vendrá determinada por la máxima complejidad temporal de entre las establecidas anteriormente para las distintas coordenadas. Se puede observar que cuando $Prof(\mathbf{E_{11}}) = Prof(\mathbf{O_{01}})$ o $Prof(\mathbf{O_{11}}) = Prof(\mathbf{E_{01}})$, entonces el retardo viene determinado por las ecuaciones 9.58 y 9.60, respectivamente, con lo que el número de niveles de puertas XOR es $\lceil log_2(m-1) \rceil + 2$. En cambio, cuando $Prof(\mathbf{E_{11}}) < Prof(\mathbf{O_{01}})$ o $Prof(\mathbf{O_{11}}) < Prof(\mathbf{E_{01}})$, entonces el retardo viene determinado por las ecuaciones 9.62 o 9.63, ya que estas presentan una mayor complejidad que la proporcionada por las ecuaciones 9.57, 9.59 o 9.61. Como la complejidad dada por las ecuaciones 9.62 y 9.63 es la misma para los valores que nos interesan, se puede utilizar la expresión $\lceil log_2(2\lceil \frac{m}{2} \rceil) \rceil + 1$ para el retardo en este caso.

Se puede observar que únicamente para los valores de $m = 3$, 4 y 6 se cumple que $Prof(\mathbf{E(O)_{11}}) < Prof(\mathbf{O(E)_{01}})$, por lo que únicamente para estos valores de m se utiliza la complejidad *temporal* (incluyendo el retardo del nivel de puertas AND) del multiplicador $T_{AND} + (1 + \lceil log_2(2\lceil \frac{m}{2} \rceil) \rceil) T_{XOR}$, mientras que $T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$ se utiliza para el resto de valores de m.

	#XOR	#AND	Retardo
[HK00]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2 m \rceil) T_{XOR}$
[ZP01]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2 m \rceil) T_{XOR}$
$Ntra.Ap^1.$	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil log_2(2\lceil \frac{m}{2} \rceil) \rceil) T_{XOR}$
$Ntra.Ap^2.$	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil log_2(m-1) \rceil) T_{XOR}$

Tabla 9.17: Complejidades teóricas de multiplicadores canónicos utilizando trinomios irreducibles $f(x) = x^m + x + 1$.

En la tabla 9.17 se comparan los mejores resultados teóricos obtenidos en la literatura por Halbutogullari y Koç [HK00] y por Zhang y Parhi [ZP01] con las complejidades obtenidas por nuestro método. En la tabla se especifica el

resultado de nuestro método para $m = 3$, 4 y 6, que se denota como $Ntra.Ap^1$. y el resultado teórico para el resto de valores de m, denotado por $Ntra.Ap^2$.

En la tabla 9.18 se comparan los retardos de puertas XOR (T_{XOR}) obtenidos por nuestra aproximación *transposicional* con los obtenidos por el método propuesto por Halbutogullari y Koç en [HK00] para todos los trinomios irreducibles $f(x) = x^m + x + 1$ existentes hasta el grado $m = 100$, observándose que nuestro método obtiene peores resultados que el método dado en [HK00].

		Ntra.Ap.	Halbut.-Koç
m	f(x)	T_{XOR}	T_{XOR}
2	2,1,0	2	2
3	3,1,0	3	3
4	4,1,0	3	3
6	6,1,0	4	4
7	7,1,0	5	4
9	9,1,0	5	5
15	15,1,0	6	5
22	22,1,0	7	6
28	28,1,0	7	6
30	30,1,0	7	6
46	46,1,0	8	7
60	60,1,0	8	7
63	63,1,0	8	7
Total		73	65

Tabla 9.18: Comparación de retardos T_{XOR} de multiplicadores obtenidos con nuestra aproximación y por Halbutogullari y Koç para distintos trinomios irreducibles.

Los resultados dados en la tabla 9.18 muestran que con el método *transposicional* se obtienen peores retardos de puertas XOR que si se utilizan los otros métodos propuestos por Halbutogullari y Koç y por Zhang y Parhi. Considerando los valores totales mostrados en la tabla, se obtiene que la complejidad T_{XOR} es un 11.0 % mayor utilizando el método *transposicional*. Estos resultados teóricos no se corresponden del todo con los resultados experimentales obtenidos cuando se utilizan plataformas reconfigurables para la implementación de los multiplicadores, como se muestra en la siguiente subsección.

Se puede mencionar que el trinomio $f(x) = x^2 + x + 1$ pertenece también a los tipos ya estudiados de trinomios $f(x) = x^m + x^{m-1} + 1$ y $f(x) = x^m + x^{\frac{m}{2}} + 1$, en los que se había obtenido el mismo retardo T_{XOR}. Sin embargo, ya que la complejidad *espacial* de los trinomios $f(x) = x^m + x^{\frac{m}{2}} + 1$ es menor, sería más óptimo considerarlo como perteneciente a este tipo de trinomios. Por último, el trinomio $f(x) = x^3 + x + 1$ también pertenece al tipo de trinomios $f(x) = x^m + x^{\frac{m-1}{2}} + 1$, en donde se obtuvo la misma complejidad temporal de retardo T_{XOR} que la mostrada en la tabla 9.18.

9.7.3. Resultados experimentales

Se ha realizado la implementación paralela sobre FPGAs de multiplicadores canónicos utilizando el método *transposicional* dado por las ecuaciones 9.54 y 9.55 y utilizando el método propuesto por Halbutogullari y Koç en [HK00]. Se ha empleado *Xilinx Foundation F2.1i* y se han realizado descripciones VHDL de los multiplicadores paralelos utilizando ambos métodos para los campos $GF(2^m)$ generados por trinomios irreducibles $f(x) = x^m + x + 1$ de grados $m = 4$, 6, 7, 9 y 15. Los dispositivos utilizados han sido los 4013XLPQ160 pertenecientes a la familia XC4000XL y se ha realizado la síntesis utilizando optimización para *Area* con un nivel de esfuerzo *High*. Asimismo, las métricas del rendimiento utilizadas han sido las mismas que en otros experimentos realizados, es decir, el *número de CLBs* y el *retardo máximo combinacional*, obteniéndose los resultados experimentales mostrados en la tabla 9.19.

	Halbut.-Koç		Transposicional	
	CLBs	Ret. Max (ns)	CLBs	Ret. Max (ns)
$GF(2^4)$	8	14.8	5	14.0
$GF(2^6)$	12	15.7	14	20.4
$GF(2^7)$	22	20.4	17	18.2
$GF(2^9)$	27	24.4	31	24.5
$GF(2^{15})$	83	26.8	81	28.0
Total	152	102.1	148	105.1

Tabla 9.19: Resultados experimentales de multiplicadores canónicos sobre campos generados por trinomios irreducibles $f(x) = x^m + x + 1$.

De los resultados mostrados en la tabla 9.19, se observa una pequeña mejora en cuanto al consumo de CLBs de nuestro método *transposicional* frente al método propuesto por Halbutogullari y Koç en [HK00]. Considerando el número total de CLBs para todas las implementaciones, se observa que nuestro método utiliza un 2.6 % menos de CLBs que la aproximación dada por Halbutogullari y Koç, siendo el mejor resultado el correspondiente al campo $GF(2^4)$ con una reducción del 37.5 % (en cambio, para $GF(2^6)$, nuestro método consume un 16.6 % más de CLBs).

Estos peores resultados en cuanto a la reducción de área obtenida por nuestro método son debidos a que para este tipo de trinomios no existe ningún *grupo* $\mathbf{G_i}$ que sea compartido entre las coordenadas del producto (las únicas agrupaciones existentes corresponden a los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$), por lo que la optimización de área realizada por la herramienta de *mapping* será menor que en otros casos. Con respecto de los retardos obtenidos, se observa en la tabla 9.19 que los resultados proporcionados por nuestro método son un 2.9 % peores que los conseguidos con la otra aproximación, hecho que se corresponde con el resultado teórico obtenido anteriormente.

9.8. Comparación de los multiplicadores

De lo visto en las secciones anteriores, se puede observar que la utilización del método *transposicional* para la realización de la multiplicación sobre los cinco tipos de trinomios considerados, produce mejoras de los retardos teóricos de puertas XOR de los multiplicadores correspondientes en comparación con los mejores resultados encontrados en la literatura. Estas reducciones en el número de niveles XOR van disminuyendo desde el mejor resultado obtenido con el primer trinomio $f(x) = x^m + x^{m-1} + 1$ hasta la obtención de un peor resultado teórico con el último trinomio visto $f(x) = x^m + x + 1$.

Concretamente y considerando todos los trinomios irreducibles de grado hasta 100, con el trinomio $f(x) = x^m + x^{m-1} + 1$ se obtuvo una mejora promedio (en complejidad *temporal* del multiplicador dada en retardos de puertas XOR) del 11.7 %, con unos valores máximos de mejora del 25.0 %; con el trinomio $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m *impar*) la mejora promedio fue del 6.7 %, con un valor máximo de mejora del 25.0 %; con el trinomio $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m *impar*) se obtuvo una mejora promedio del 5.1 %, con valor máximo del 20.0 %; para el trinomio $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m *par*) se obtuvo una igualdad teórica con los mejores métodos encontrados en la literatura y, finalmente, con el trinomio $f(x) = x^m + x + 1$ se obtuvieron resultados teóricos un 11.0 % peores en promedio que los obtenidos por dichos otros métodos.

Con respecto de la complejidad *espacial* teórica, se obtuvo en todos los casos una igualdad teórica en número de puertas AND y XOR con los mejores métodos encontrados en la literatura.

Estos resultados teóricos no se corresponden con los resultados experimentales obtenidos en las implementaciones reconfigurables sobre FPGAs. En cuanto al número de CLBs consumidos (valores promedio de los campos utilizados) por los multiplicadores implementados utilizando el método *transposicional* y utilizando otras aproximaciones dadas en la literatura, se ha obtenido con el trinomio $f(x) = x^m + x^{m-1} + 1$ una mejora del 24.6 %, con una reducción máxima del 26.9 %; para el trinomio $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m *impar*) la reducción del número de CLBs fue del 11.8 %, con un valor máximo del 18.2 %; con el trinomio $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m *impar*) se obtuvo una mejora del 10.2 % con un máximo del 13.0 %, y con el trinomio $f(x) = x^m + x + 1$ se obtuvo una reducción promedio del 2.6 %, con una mejora máxima del 37.5 %. Para el trinomio $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m *par*) no se realizaron experimentos.

Con respecto al retardo *temporal* obtenido experimentalmente (valores promedios), con el trinomio $f(x) = x^m + x^{m-1} + 1$ se han tenido retardos un 39.5 % peores; para el trinomio $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m *impar*) el retardo ha sido un 4.4 % mejor; con el trinomio $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m *impar*) el retardo fue un 2.2 % peor y con el trinomio $f(x) = x^m + x + 1$ se obtuvo un retardo un 2.9 % peor que el obtenido por otros métodos.

En la tabla 9.20 se resumen los porcentajes promedio anteriores de mejora o empeoramiento de los resultados teóricos y experimentales obtenidos de la comparación de los multiplicadores canónicos construidos utilizando el método *transposicional* y los métodos dados en [HK00] y [ZP01], para los cinco tipos de trinomios irreducibles considerados. No se incluyen las complejidades teóricas en número de puertas AND y XOR, ya que se obtuvieron los mismos resultados que en los otros métodos encontrados en la literatura. En la tabla, los símbolos $\uparrow$, $\downarrow$ y $\leftrightarrow$ denotan *mejora* (es decir, reducción del parámetro correspondiente), *empeoramiento* (incremento del parámetro) e *igualdad* en la comparación, respectivamente. Asimismo, los trinomios se representan en la notación ya utilizada en capítulos anteriores. Por ejemplo, el trinomio $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ se representa como $(m, \frac{m-1}{2}, 0)$.

	Teórico	Experimental	
Trinomios	Retardo T_{XOR}	#CLBs	Retardo
$(m, m-1, 0)$	$\uparrow$ 11.7 %	$\uparrow$ 24.6 %	$\downarrow$ 39.5 %
$(m, \frac{m+1}{2}, 0)$	$\uparrow$ 6.7 %	$\uparrow$ 11.8 %	$\uparrow$ 4.4 %
$(m, \frac{m-1}{2}, 0)$	$\uparrow$ 5.1 %	$\uparrow$ 10.2 %	$\downarrow$ 2.2 %
$(m, \frac{m}{2}, 0)$	$\leftrightarrow$ 0.0 %	-	-
$(m, 1, 0)$	$\downarrow$ 11.0 %	$\uparrow$ 2.6 %	$\downarrow$ 2.9 %

Tabla 9.20: Porcentajes promedio de mejora/empeoramiento de resultados teóricos y experimentales obtenidos por el método *transposicional* para multiplicadores sobre campos generados por distintos trinomios irreducibles.

De los resultados mostrados en la tabla 9.20 se observa que la implementación reconfigurable de los multiplicadores utilizando nuestro método *transposicional* produce *siempre* mejores resultados (en cuanto al número de CLBs consumidos) que las implementaciones realizadas utilizando el método propuesto por Halbutogullari y Koç. Su porcentaje de mejora va disminuyendo gradualmente desde el 24.6 % correspondiente al trinomio $f(x) = x^m + x^{m-1} + 1$ hasta el 2.6 % obtenido con el trinomio $f(x) = x^m + x + 1$.

Estos resultados se pueden explicar teniendo en cuenta la posibilidad de *compartición* de términos proporcionada por cada trinomio utilizado. En las secciones anteriores se vio que la mayor *compartición* de expresiones (formación de *grupos* $\mathbf{G_i}$) entre las distintas coordenadas del producto correspondía al trinomio $f(x) = x^m + x^{m-1} + 1$, que es el que produce la mayor reducción experimental de área (número de CLBs). Los trinomios $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ y $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m *impar*) eran los siguientes trinomios en cuanto al número de grupos $\mathbf{G_i}$ formados y compartidos, respectivamente, observándose que este hecho coincide con los porcentajes de mejora de área obtenida experimentalmente. Finalmente, con los trinomios $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m *par*) y $f(x) = x^m + x + 1$ no se obtenía ningún grupo, por lo que no se dan *com-*

particiones de términos entre coordenadas. Únicamente se tiene la *agrupación* proporcionada por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$. Este hecho también se refleja en los resultados experimentales obtenidos con $f(x) = x^m + x + 1$, que es el que presenta una menor reducción, de entre todos los trinomios, en cuanto al número de CLBs utilizados para los multiplicadores.

En realidad, esta última afirmación con respecto de la *no compartición* de términos entre las coordenadas del producto cuando se utilizan los trinomios $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m *par*) y $f(x) = x^m + x + 1$ no es cierta, ya que *sí* se produce la compartición de algunos *términos individuales agrupados* $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ que se repiten en algunas coordenadas. Esta *compartición* de términos individuales depende de la composición de la matriz $\mathbf{K_1}$ que determina, a su vez, los componentes del vector $\underline{\alpha}_{\Omega}^{t}\mathbf{K_1}$. Lo que *no* se cumple en la multiplicación realizada con estos dos tipos de trinomios es la existencia de *expresiones complejas agrupadas* $\mathbf{G_i}$ formadas por sumas de términos $\mathbf{E_{i1}}$ y/o $\mathbf{O_{i1}}$ que puedan ser compartidas entre las distintas coordenadas del producto. A la *existencia* y *compartición* de estas expresiones complejas $\mathbf{G_i}$ es a lo que nos hemos referido a lo largo de este capítulo cuando hemos mencionado la *compartición* o *no compartición* de términos entre las coordenadas del producto.

Como ya se comentó en el capítulo 8, las puertas con un menor número de entradas pueden ser fácilmente incluidas en las LUT de una FPGA y pueden incrementar la posibilidad de que una herramienta de *mapping* obtenga una mejor solución cuando se utiliza hardware reconfigurable para la implementación de los multiplicadores. Por este motivo, las *agrupaciones* de términos producidas por el método *transposicional* a través del cálculo de 2-ciclos y de 1-ciclos y su *compartición* entre varias coordenadas del producto, permite la formación de *grupos* de dos puertas AND de dos entradas y una puerta XOR de dos entradas que se pueden realizar fácilmente con una LUT, por lo que la herramienta de *mapping* reconfigurable puede conseguir una mejor optimización para área [KL01]. Estas consideraciones se han constatado con los resultados experimentales obtenidos.

Por último, se puede observar en la tabla 9.20 que los mejores resultados teóricos obtenidos por nuestro método en cuanto al retardo de puertas XOR no se corresponde con la peor complejidad temporal obtenida experimentalmente para los multiplicadores. Esta discrepancia sería debida a que en los experimentos realizados se establecieron optimizaciones para *área*, no para tiempo.

9.9. Conclusiones

En este capítulo se ha aplicado el método *transposicional* introducido en el capítulo 8 a la multiplicación en base canónica de elementos pertenecientes a campos de Galois $GF(2^m)$ generados por varios tipos de *trinomios* irreducibles. Para ello, se han dado expresiones generales de cálculo del producto

para cualquier tipo de trinomio y se han particularizado posteriormente dichas expresiones para cada uno de los trinomios irreducibles considerados, obteniéndose unas nuevas ecuaciones que constituyen el método *transposicional* de multiplicación aplicado a cada uno de los trinomios estudiados.

Se vio en el capítulo 8 que el método *transposicional* determina *agrupaciones sencillas* (representadas por las funciones $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$) de términos producto $x_k = (a_k c_k)$ y de sumas de términos producto $x_{ij} = (a_i c_j + a_j c_i)$ pertenecientes a las coordenadas de los elementos del campo finito a multiplicar. La aplicación del método *transposicional* a los trinomios irreducibles ha permitido, además, mostrar la existencia de *agrupaciones complejas* (denotadas como $\mathbf{G_i}$) de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$, cuya compartición entre las distintas coordenadas permite reducir la complejidad teórica del multiplicador. De esta forma, se ha visto que nuestro método de construcción de multiplicadores sobre campos $GF(2^m)$ ha conseguido reducir en tres de los cinco trinomios estudiados la menor complejidad *temporal* teórica (dada en retardos de puertas XOR) encontrada en la literatura y ha igualado la menor complejidad *espacial* (dada en número de puertas AND y XOR) también encontrada en la literatura. Para comprobar estos resultados, se han realizado los análisis de complejidad teóricos correspondientes y se han dado ejemplos de multiplicaciones para cada uno de los trinomios considerados.

Junto a estos resultados de complejidad teóricos, se han realizado implementaciones sobre hardware reconfigurable (FPGAs) de los multiplicadores descritos por nuestro método *transposicional* y por otro método similar dado en la literatura para cuatro de los cinco trinomios irreducibles estudiados, encontrándose en *todos* los casos que nuestro método también conlleva una reducción de complejidad *espacial* experimental, representada por el número de CLBs consumidos por los multiplicadores.

Esta reducción de área obtenida con nuestro método *transposicional* es debida a que tanto las *agrupaciones sencillas* producidas por los 2-ciclos y los 1-ciclos como las *agrupaciones complejas* producidas por los grupos $\mathbf{G_i}$ ayudan a las herramientas automáticas en la formación y compartición de puertas con un menor número de entradas que pueden ser fácilmente incluidas en una sola LUT y esto incrementa la posibilidad de obtención de mejores optimizaciones de área. Por lo tanto, las *agrupaciones* y *comparticiones* proporcionadas en las descripciones de alto nivel del multiplicador utilizando el método *transposicional* parecen constituir un buen método cuando se utilizan plataformas reconfigurables como las FPGAs basadas en LUT para la implementación de dichos multiplicadores.

Capítulo 10

Conclusiones y trabajo futuro

Se resume el trabajo presentado en esta memoria, enunciando las principales aportaciones efectuadas y planteando las futuras líneas de investigación.

Los campos de extensión de $GF(2)$ tienen actualmente un gran interés debido a su posibilidad de aplicación en numerosas áreas como la criptografía, los códigos algebraicos, el procesamiento digital de señal, la generación aleatoria de números o la *verificación probabilista* de circuitos. En esta tesis nos hemos centrado en esta última aplicación de los campos de Galois y en la operación aritmética de *multiplicación* sobre $GF(2^m)$ utilizada en la misma, debido a su complejidad e importancia por ser la base de otras operaciones aritméticas más complejas.

La creciente complejidad de los circuitos digitales actuales y la necesaria utilización de herramientas automáticas hacen que la *verificación* del diseño sea un tema de gran interés para la industria. La utilización de las estructuras de datos conocidas como ROBDDs constituye uno de los procedimientos de verificación más satisfactorios y más ampliamente utilizados, aunque presenta importantes inconvenientes como son el tamaño y tiempo de verificación para determinados circuitos y la gran dependencia de la ordenación de sus variables Booleanas. En esta tesis se ha estudiado la aplicación de los *campos de Galois* $GF(2^m)$ a la *verificación probabilista*, basada en la comparación de *signaturas*, como un método alternativo de comprobación de la equivalencia de circuitos lógicos combinacionales representados tanto en formato de *dos niveles* como en forma *multinivel*.

La operación aritmética sobre $GF(2^m)$ más costosa utilizada en la verificación probabilista es la *multiplicación*, que a su vez es la base de otras operaciones aritméticas avanzadas. Este hecho, unido a que la rapidez requerida a las aplicaciones que utilizan aritmética sobre $GF(2^m)$ propicia que estas operaciones sean a menudo implementadas sobre circuitos VLSI, hacen que la búsqueda

de métodos de multiplicación y de arquitecturas eficientes de módulos *multiplicadores* sobre $GF(2^m)$ tenga una especial importancia. Por este motivo, en esta tesis se han estudiado tanto métodos de multiplicación como la utilización de dispositivos reconfigurables para la implementación de los multiplicadores.

Teniendo en cuenta los objetivos enunciados en el capítulo 1, a continuación se resume el trabajo realizado, enumerándose los resultados obtenidos y las principales aportaciones efectuadas en esta tesis.

10.1. Principales aportaciones

1. Se han introducido los fundamentos teóricos básicos de la *verificación probabilista* basada en la comparación de *signaturas* por medio de la utilización de transformaciones algebraicas, encontrándose que la propiedad de *ortogonalidad* o *disjunción* de las funciones a verificar simplifica en gran medida el cálculo de dichas *signaturas*.

2. Se ha presentado un nuevo método para el cálculo de operaciones sobre funciones Booleanas representadas en forma *cúbica* de *dos niveles* que proporciona los resultados en forma *cúbica disjunta* o DSOP, propiedad muy importante para su utilización en la verificación probabilista. Se ha comparado este nuevo método con otra técnica de creación de representaciones DSOP, comprobándose el mejor rendimiento obtenido por nuestra aproximación.

3. Se han revisado las estructuras de datos en forma de grafo más utilizadas para la representación de funciones Booleanas, denominadas ROBDDs. Se han descrito sus características principales, los aspectos de implementación de un entorno de manipulación de ROBDDs y se han revisado las distintas extensiones del concepto de ROBDD existentes en la literatura, interesándonos en los grafos capaces de representar funciones con valores *numéricos*. Se ha estudiado el importante problema de la *ordenación* de variables, así como los métodos *heurísticos* y *dinámicos* de intercambio de variables existentes para tratar de solucionarlo.

4. Se ha presentado un nuevo método *heurístico* de ordenación de variables basado en la información topológica del circuito. Se han realizado experimentos construyendo los ROBDDs de circuitos de benchmark utilizando nuestra ordenación, la ordenación original y distintas heurísticas clásicas, comprobando la eficiencia de nuestra heurística tanto en el tamaño como en el tiempo de construcción de los ROBDDs. Posteriormente, se han utilizado métodos dinámicos sobre los ROBDDs ya construidos heurísticamente con nuestra aproximación y con la ordenación original, comprobándose también el mejor comportamiento de nuestro método.

5. Se ha desarrollado una aproximación *híbrida* de verificación de funciones Booleanas que combina la verificación *determinista* de ROBDDs con la verificación *probabilista* aplicada sobre $GF(2^m)$ y que comienza construyendo los s-ROBDDs (ROBDDs modificados para el almacenamiento de *signaturas*) de las funciones a verificar. Si no existen *restricciones* de espacio o tiempo, se finaliza la verificación *determinista* por medio de la comparación de los s-ROBDDs. Si existe algún límite, la construcción de s-ROBDDs se detiene cuando se alcanza ese límite y se finaliza entonces con la verificación *probabilista*, en la que se comparan las signaturas de las funciones para decidir su equivalencia, utilizándose para ello las signaturas ya calculadas por los s-ROBDDs construidos hasta el momento en que se alcanzó el límite.

6. El nuevo método *híbrido* se ha aplicado a circuitos combinacionales de *dos niveles* representados en formato DSOP. Tanto la utilización de expresiones *cúbicas disjuntas* como el empleo de campos de Galois $GF(2^m)$ simplifican el cálculo de las *signaturas*.

7. Se han realizado experimentos de verificación *probabilista*, *determinista* e *híbrida*, comparándose el tamaño de memoria y el tiempo de ejecución necesarios para la verificación. Los resultados obtenidos han demostrado que el método *probabilista* es el de mejor rendimiento tanto en tiempo de ejecución como en cantidad de memoria consumida, mientras que la verificación *determinista* (en la que, si es posible, los s-ROBDDs se construyen completamente) es la que presenta el peor comportamiento tanto en tiempo de ejecución como en recursos de memoria.

8. Los experimentos realizados también han demostrado que el método *híbrido* tiene un rendimiento intermedio entre el peor caso *determinista* y el mejor caso *probabilista*, dependiendo su comportamiento del límite elegido. Por lo tanto, las aproximaciones de verificación de tipo *probabilista*, como el método *híbrido* presentado, pueden suponer una buena alternativa a la verificación determinista clásica.

9. Se han aplicado las estructuras de datos denominadas $\oplus$-OBDDs a la verificación *probabilista* sobre $GF(2^m)$ de circuitos combinacionales representados en formato *multinivel*, ya que simplifican en gran medida el proceso de cálculo de las signaturas.

10. Se ha presentado un nuevo método de síntesis para la construcción de los $\oplus$-OBDDs con una *capa superior* de dos niveles de $\oplus$-nodos, utilizando la *expansión positiva de Davio* con respecto a una única variable de entrada seleccionada, llamada *variable de descomposición*.

11. Los resultados obtenidos muestran la existencia de una variable de descomposición *óptima* para la cual tanto los tiempos de construcción como los tamaños de los $\oplus$-OBDDs obtenidos utilizando nuestro método de síntesis resultan ser significativamente menores que los obtenidos por otros autores.

12. Se han estudiado los *campos de Galois* y los *campos de extensión* $GF(2^m)$, así como las distintas bases de representación de elementos del campo y su obtención a partir de polinomios generadores. Nuestro interés se ha centrado en la multiplicación sobre $GF(2^m)$, por lo que hemos comparado las complejidades *teóricas* de los multiplicadores sobre las bases *canónica*, *normal* y *dual*.

13. Se han implementado multiplicadores paralelos con FPGAs sobre $GF(2^8)$ en las bases *polinómica*, *normal* y *dual* para distintos polinomios generadores y se han comparado sus complejidades espaciales y temporales, tanto *teóricas* como *experimentales*. Las motivaciones han sido la escasez de estudios existentes sobre la utilización de hardware reconfigurable para implementaciones aritméticas sobre campos de Galois y la práctica ausencia de trabajos en la literatura que realicen comparaciones de los distintos tipos de multiplicadores mencionados.

14. El estudio sobre las complejidades *teóricas* de los multiplicadores ha mostrado la importancia que tienen la *base de representación* y el *polinomio generador* en la complejidad teórica del multiplicador.

15. Los experimentos realizados también han mostrado que, en general, las complejidades *teóricas*, tanto *espaciales* (número de puertas AND y XOR) como *temporales* (retardos de puertas AND y XOR), *no predicen* de forma exacta el consumo de CLBs ni el retardo real cuando se utilizan FPGAs. Además, en aquellos casos en los que las *previsiones* teóricas se verifican experimentalmente, se observa que las *proporciones* teóricas de complejidad no se cumplen experimentalmente.

16. Se ha desarrollado un nuevo método de multiplicación en base canónica sobre campos $GF(2^m)$, al que hemos denominado método *transposicional*. El desarrollo y formulación inicial de este método se ha realizado aplicando un nuevo algoritmo de multiplicación en base canónica a los AOPs irreducibles.

17. Se han analizado las complejidades *teóricas* espacial y temporal de los multiplicadores construidos usando el método *transposicional*, obteniéndose complejidades iguales a las mejores encontradas en la literatura.

18. Se ha realizado el análisis *teórico* de complejidad de los multiplicadores construidos con el método *transposicional* para distintos tipos de trinomios irreducibles y se ha comprobado que nuestro método ha conseguido *reducir* en tres de los cinco trinomios estudiados la menor complejidad *temporal* teórica (dada en retardos de puertas XOR) encontrada en la literatura. Además, nuestro método iguala la menor complejidad *espacial* (dada en número de puertas AND y XOR) también encontrada en la literatura.

19. Se ha comprobado que la descripción de un multiplicador proporcionada por el método *transposicional*, aplicado tanto a AOPs como a trinomios irreducibles, conlleva una reducción de complejidad *experimental* cuando se utiliza hardware reconfigurable para la implementación.

20. Para los AOPs, se han realizado además implementaciones de multiplicadores *canónicos* sobre CPLDs y de multiplicadores en base *normal* sobre FPGAs, obteniéndose también resultados *experimentales* que muestran la reducción de *área* y, en determinados casos también de *tiempo*, de los multiplicadores implementados con nuestro método.

21. La reducción de área obtenida con nuestro método *transposicional* en las implementaciones sobre FPGAs es debida a que la *agrupación* y *compartición* de subexpresiones proporcionadas en las descripciones de alto nivel del multiplicador ayudan a las herramientas automáticas en la formación y compartición de puertas con un menor número de entradas que pueden ser fácilmente incluidas en una sola LUT, lo cual incrementa la posibilidad de obtener mejores optimizaciones de área.

10.2. Líneas futuras de investigación

A la vista de los temas tratados en esta tesis y de los resultados obtenidos, se pueden establecer varias líneas de investigación que continuarían el trabajo aquí desarrollado y que se podrían encuadrar dentro de las áreas de la *verificación* o de la *aritmética sobre campos de Galois*.

Pertenecientes al tema de la *verificación*, se podrían mencionar las siguientes líneas de estudio:

- Nuestra verificación *híbrida* se restringe a los circuitos de *dos niveles* en forma SOP, por medio de la conversión previa a formatos de tipo DSOP utilizando la nueva formulación introducida en el capítulo 3. Un trabajo futuro de investigación sería la generalización a circuitos SOP sin la restricción impuesta por la propiedad de *disjunción*.

- Búsqueda de nuevos métodos de síntesis de $\oplus$-OBDDs aplicados a la verificación *probabilista multinivel* que mantengan tamaños de grafos reducidos. En esta tesis se ha utilizado la *expansión positiva de Davio* con respecto a una única *variable de descomposición* seleccionada para la creación de una *capa superior* de dos niveles de $\oplus$-nodos. Se podrían utilizar otros tipos de descomposición, ampliar el número de variables de descomposición e introducir $\oplus$-nodos en niveles intermedios de los $\oplus$-OBDDs con el objeto de reducir su tamaño.

- Extensión a los circuitos *secuenciales* del estudio de *verificación*, tanto *determinista* como *probabilista*, aplicando los ROBDDs a la *comprobación simbólica de modelos* [McM93] y a la comparación de *iteraciones sobre el punto fijo*. En este sentido y aplicable al *análisis de alcanzabilidad*, hemos desarrollado un algoritmo de particionamiento de la *relación de transición* que produce una *disyunción* de particiones *ortogonales*. Esta propiedad es muy importante para el cálculo de *signaturas* en la verificación *probabilista*, por lo que nuestro siguiente paso sería la aplicación del algoritmo a este tipo de verificación.

Con respecto de la *aritmética sobre campos de Galois*, se pueden mencionar las siguientes líneas de trabajo:

- Profundización en el estudio del empleo de dispositivos reconfigurables para la implementación de los distintos tipos de multiplicadores sobre $GF(2^m)$ construidos con el método *transposicional*, como por ejemplo, el estudio de la influencia de la utilización de arquitecturas de *grano fino* o de *grano grueso* en las complejidades de área y tiempo obtenidas para los multiplicadores.

- Aplicación del método *transposicional* a la multiplicación sobre *campos compuestos* $GF((2^n)^k)$, que son extensiones del campo base $GF(2^n)$ isomorfas a los campos de Galois $GF(2^m)$ con $m = n \cdot k$. Ya hemos iniciado el desarrollo de esta aplicación, y los experimentos preliminares de implementación sobre FPGAs han mostrado una reducción de área en torno al 10 % en comparación con otras aproximaciones.

- Estudio de la multiplicación sobre *curvas algebraicas* (*elípticas* e *hiperelípticas*) y determinación de si técnicas de *agrupación* y *compartición* de subexpresiones (como el método *transposicional*) son aplicables a este tipo de problemas y, en caso de serlo, si su aplicación conlleva algún tipo de reducción de complejidad.

- Por supuesto, otra línea de investigación evidente sería la aplicación de los estudios y trabajos realizados en la multiplicación sobre $GF(2^m)$ a temas tan importantes como la *criptografía*, los *códigos algebraicos* o el *procesamiento digital de señales*.

Apéndice A

Cálculo de la complejidad espacial de multiplicadores sobre $GF(2^m)$ para trinomios irreducibles

Para realizar el cálculo de la complejidad *espacial* teórica de los multiplicadores en base canónica sobre $GF(2^m)$ generados utilizando el método *transposicional* para los trinomios irreducibles estudiados, es necesario determinar la complejidad *espacial* del vector $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$. Este vector incluye *todos* los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ necesarios para la determinación de las coordenadas del producto utilizando nuestro método, ya que el resto de vectores $\underline{\alpha}_{\Omega}^{t}\mathbf{K_i}$ $(i = 1, 2, \ldots, \tau)$ están formados por términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ pertenecientes a dicho vector. Por lo tanto se debe determinar, en primer lugar, la suma de las complejidades *espaciales* (número de puertas AND y XOR de dos entradas) determinadas por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ pertenecientes a $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ y, posteriormente, se deben añadir las puertas XOR adicionales necesarias para el cálculo de las distintas coordenadas del producto en función del trinomio irreducible utilizado y según las expresiones dadas en el capítulo 9.

Para la realización de estos cálculos, vamos a utilizar el hecho (comprobable a partir de la observación de las expresiones dadas en el capítulo anterior) de que el vector $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ consta de $\lfloor \frac{m}{2} \rfloor$ términos $\mathbf{E_{i0}}$, de $\lceil \frac{m}{2} \rceil$ términos $\mathbf{O_{i0}}$, de $\lceil \frac{m-1}{2} \rceil$ términos $\mathbf{E_{i1}}$ y de $\lfloor \frac{m-1}{2} \rfloor$ términos $\mathbf{O_{i1}}$, donde m determina la cardinalidad del campo de Galois $GF(2^m)$.

Se comprobará posteriormente que el número de puertas AND necesarias para la realización de los multiplicadores es el mismo para *todos* los trinomios irreducibles. También se demostrará que lo mismo sucede con el número de puertas XOR, con la sola excepción del multiplicador correspondiente al EST $f(x) = x^m + x^{\frac{m}{2}} + 1$ con m par.

A continuación realizamos el cálculo del número de puertas AND de dos entradas necesarias para la realización de los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ incluidos en $\underline{\alpha}_\Omega^t\mathbf{K_0}$. Posteriormente se realiza un cálculo similar del número de puertas XOR de dos entradas necesarias para la realización de dichos términos. Por último, se utilizan las expresiones de las coordenadas del producto dadas en el capítulo 9 para cada trinomio con el objeto de determinar el número total de puertas XOR de los multiplicadores. En todos los cálculos, se tiene en cuenta el rango de subíndices que toman los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ y que se obtiene de las ecuaciones dadas por el método *transposicional* para cada trinomio.

A.1. Cálculo del número de puertas AND

En el capítulo 9 se vio que la matriz $\mathbf{K_0}$ era común a *cualquier* trinomio irreducible seleccionado para un campo $GF(2^m)$ determinado. Lo mismo sucede, por tanto, con el vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$. Por otra parte, también se vio que el resto de vectores $\underline{\alpha}_\Omega^t\mathbf{K_i}$ $(i = 1, 2, \ldots, \tau)$ estan formados por términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ pertenecientes a $\underline{\alpha}_\Omega^t\mathbf{K_0}$. Todo esto implica que la complejidad *espacial* en número de puertas AND viene determinada *exclusivamente* por el vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$ y que esta complejidad es *común* para cualquier multiplicador sobre un campo $GF(2^m)$ generado por cualquier trinomio irreducible de grado m.

A continuación calculamos el número de puertas AND correspondientes a cada conjunto de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ existentes en el vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$, teniendo en cuenta la existencia de $\lfloor\frac{m}{2}\rfloor$ términos $\mathbf{E_{i0}}$, de $\lceil\frac{m}{2}\rceil$ términos $\mathbf{O_{i0}}$, de $\lceil\frac{m-1}{2}\rceil$ términos $\mathbf{E_{i1}}$ y de $\lfloor\frac{m-1}{2}\rfloor$ términos $\mathbf{O_{i1}}$. También se utilizan las complejidades espaciales de dichos términos dadas en la tabla 9.2.

Términos $\mathbf{E_{i0}}$ El número de puertas AND determinado por los $\lfloor\frac{m}{2}\rfloor$ términos $\mathbf{E_{i0}}$ existentes en $\underline{\alpha}_\Omega^t\mathbf{K_0}$ se puede calcular utilizando la tabla 9.2 y los valores tomados por los subíndices i determinados por las ecuaciones vistas en el capítulo anterior. Esta complejidad viene dada por la siguiente expresión

$$\otimes(\sum \mathbf{E_{i0}}) = \sum_{i=1}^{\lfloor\frac{m}{2}\rfloor} \otimes(\mathbf{E_{(2i)0}}) = \sum_{i=1}^{\lfloor\frac{m}{2}\rfloor} 2\left\lceil\frac{2i}{2}\right\rceil = 2\sum_{i=1}^{\lfloor\frac{m}{2}\rfloor} i = \left\lfloor\frac{m}{2}\right\rfloor\left(\left\lfloor\frac{m}{2}\right\rfloor + 1\right) \quad \text{(A.1)}$$

donde $\otimes(\sum \mathbf{E_{i0}})$ representa el número de puertas AND dado por todos los términos $\mathbf{E_{i0}}$, donde $\otimes$ representa el número de puertas AND del término correspondiente y donde se ha utilizado el hecho de que la suma de la progresión aritmética $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$. Se observa que $\lfloor\frac{m}{2}\rfloor = \frac{m}{2}$ para m *par*, mientras que si m es *impar* entonces $\lfloor\frac{m}{2}\rfloor = \frac{m-1}{2}$. Por lo tanto se obtiene que

$$\otimes(\sum \mathbf{E_{i0}}) = \begin{cases} \frac{m^2+2m}{4} & m \ par \\ \frac{m^2-1}{4} & m \ impar \end{cases} \quad \text{(A.2)}$$

Términos $\mathbf{O_{i0}}$ El número de puertas AND determinado por los $\lceil \frac{m}{2} \rceil$ términos $\mathbf{O_{i0}}$ existentes en $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ también se calcula utilizando la tabla 9.2 y el rango de valores tomados por los subíndices i para este caso. La complejidad viene dada por la expresión siguiente

$$\otimes(\sum \mathbf{O_{i0}}) = \sum_{i=1}^{\lceil \frac{m}{2} \rceil} \otimes(\mathbf{O_{(2i-1)0}}) = \sum_{i=1}^{\lceil \frac{m}{2} \rceil} (2 \left\lceil \frac{2i-1}{2} \right\rceil - 1) = \sum_{i=1}^{\lceil \frac{m}{2} \rceil} (2i-1)$$
$$= 2\sum_{i=1}^{\lceil \frac{m}{2} \rceil} i - \left\lceil \frac{m}{2} \right\rceil = \left\lceil \frac{m}{2} \right\rceil \left(\left\lceil \frac{m}{2} \right\rceil + 1 \right) - \left\lceil \frac{m}{2} \right\rceil = \left\lceil \frac{m}{2} \right\rceil^2 \qquad \text{(A.3)}$$

donde $\otimes(\sum \mathbf{O_{i0}})$ representa el número de puertas AND dado por todos los términos $\mathbf{O_{i0}}$ y donde, al igual que en el cálculo anterior, se ha utilizado la suma de la progresión aritmética $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$. Se observa que $\lceil \frac{m}{2} \rceil = \frac{m}{2}$ para m *par* y que $\lceil \frac{m}{2} \rceil = \frac{m+1}{2}$ si m es *impar*, por lo que se tiene que

$$\otimes(\sum \mathbf{O_{i0}}) = \begin{cases} \frac{m^2}{4} & m \ par \\ \frac{m^2+2m+1}{4} & m \ impar \end{cases} \qquad \text{(A.4)}$$

Términos $\mathbf{E_{i1}}$ En el cálculo de los términos anteriores se han diferenciado los valores de m *pares* o *impares* una vez que se ha obtenido una expresión general del número de puertas AND. Esto ha sido posible debido a que el rango de valores que toman los subíndices i es el mismo para cualquier m (cuyo valor determina los límites superiores de los rangos). Esta igualdad en el rango de subíndices i no se cumple para los términos $\mathbf{E_{i1}}$, por lo que tendremos que distinguir entre valores *pares* o *impares* para m desde el comienzo del cálculo.

El número de términos $\mathbf{E_{i1}}$ existentes en $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ es $\lceil \frac{m-1}{2} \rceil$. Se observa que $\lceil \frac{m-1}{2} \rceil = \frac{m}{2}$ para m *par*, mientras que si m es *impar* entonces $\lceil \frac{m-1}{2} \rceil = \frac{m-1}{2}$. Por lo tanto, se tiene que para m *par* el número de puertas AND determinado por los términos $\mathbf{E_{i1}}$ viene dado por la expresión

$$\otimes(\sum \mathbf{E_{i1}}) = \sum_{i=1}^{\frac{m}{2}} \otimes(\mathbf{E_{(2i-2)1}}) = \sum_{i=1}^{\frac{m}{2}} (m - (2i-2) - 1) = \sum_{i=1}^{\frac{m}{2}} (m - 2i + 1)$$
$$= m\left(\frac{m}{2}\right) - 2\sum_{i=1}^{\frac{m}{2}} i + \frac{m}{2} = m\left(\frac{m}{2}\right) - \frac{m}{2}\left(\frac{m}{2} + 1\right) + \frac{m}{2} = \frac{m^2}{4} \qquad \text{(A.5)}$$

donde se ha tenido en cuenta el rango de subíndices para este caso y donde se han utilizado los resultados de la tabla 9.2 junto con el resultado de la suma de la progresión aritmética vista en los cálculos de términos anteriores.

Para valores *impares* de m se tiene que el número de puertas AND determinadas por los términos $\mathbf{E_{i1}}$ viene dado por la siguiente expresión

$$\otimes(\sum \mathbf{E_{i1}}) = \sum_{i=1}^{\frac{m-1}{2}} \otimes(\mathbf{E_{(2i-1)1}}) = \sum_{i=1}^{\frac{m-1}{2}} (m-(2i-1)-1) = \sum_{i=1}^{\frac{m-1}{2}} (m-2i)$$
$$= m\left(\frac{m-1}{2}\right) - \left(\frac{m-1}{2}\right)\left(\frac{m-1}{2}+1\right) = \frac{m^2-2m+1}{4} \quad \text{(A.6)}$$

donde se ha tenido en cuenta el rango de subíndices para este caso y donde se han utilizado los mismos datos adicionales que en los cálculos anteriores.

En resumen, el número de puertas AND de dos entradas determinado por todos los términos $\mathbf{E_{i1}}$, $\otimes(\sum \mathbf{E_{i1}})$, viene dado por la expresión

$$\otimes(\sum \mathbf{E_{i1}}) = \begin{cases} \frac{m^2}{4} & m \ par \\ \frac{m^2-2m+1}{4} & m \ impar \end{cases} \quad \text{(A.7)}$$

Términos $\mathbf{O_{i1}}$ Para estos términos $\mathbf{O_{i1}}$ tampoco se cumple la igualdad en el rango de subíndices i con independencia de m, por lo que también se tiene que hacer la distinción entre valores *pares* o *impares* para m desde el comienzo del cálculo. En este caso, el número de términos $\mathbf{O_{i1}}$ existentes en $\underline{\alpha}_\Omega^t \mathbf{K_0}$ es $\lfloor \frac{m-1}{2} \rfloor$, observándose que $\lfloor \frac{m-1}{2} \rfloor = \frac{m-2}{2}$ para m *par* y que $\lfloor \frac{m-1}{2} \rfloor = \frac{m-1}{2}$ para valores de m *impares*.

Utilizando las mismas consideraciones y datos adicionales que en los cálculos anteriores, se tiene que el número de puertas AND determinado por los términos $\mathbf{O_{i1}}$ para m *par* viene dado por la expresión

$$\otimes(\sum \mathbf{O_{i1}}) = \sum_{i=1}^{\frac{m-2}{2}} \otimes(\mathbf{O_{(2i-1)1}}) = \sum_{i=1}^{\frac{m-2}{2}} (m-(2i-1)-1) = \sum_{i=1}^{\frac{m-2}{2}} (m-2i)$$
$$= m\left(\frac{m-2}{2}\right) - \left(\frac{m-2}{2}\right)\left(\frac{m-2}{2}+1\right) = \frac{m^2-2m}{4} \quad \text{(A.8)}$$

mientras que para m *impares* el número de puertas AND viene dado por

$$\otimes(\sum \mathbf{O_{i1}}) = \sum_{i=1}^{\frac{m-1}{2}} \otimes(\mathbf{O_{(2i-2)1}}) = \sum_{i=1}^{\frac{m-1}{2}} (m-(2i-2)-1) = \sum_{i=1}^{\frac{m-1}{2}} (m-2i+1)$$
$$= m\left(\frac{m-1}{2}\right) - \left(\frac{m-1}{2}\right)\left(\frac{m-1}{2}+1\right) + \frac{m-1}{2} = \frac{m^2-1}{4} \quad \text{(A.9)}$$

En resumen, el número de puertas AND de dos entradas determinado por todos los términos $\mathbf{O_{i1}}$, $\otimes(\sum \mathbf{O_{i1}})$, vendrá dado por la siguiente expresión

$$\otimes(\sum \mathbf{O_{i1}}) = \begin{cases} \frac{m^2-2m}{4} & m \ par \\ \frac{m^2-1}{4} & m \ impar \end{cases} \quad \text{(A.10)}$$

Número de puertas AND El número total de puertas AND necesario para la realización de los multiplicadores sobre $GF(2^m)$ generado por cualquier trinomio irreducible de grado m utilizando el método *transposicional*, viene determinado por la suma de las complejidades dadas por todos los términos $\mathbf{E_{i0}}$, $\mathbf{O_{i0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ pertenecientes al vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$

Por lo tanto, para valores *pares* de m se tiene que el número de puertas AND de los multiplicadores viene dado por

$$\begin{aligned}\#AND &= \otimes(\sum \mathbf{E_{i0}}) + \otimes(\sum \mathbf{O_{i0}}) + \otimes(\sum \mathbf{E_{i1}}) + \otimes(\sum \mathbf{O_{i1}}) \\ &= \frac{m^2+2m}{4} + \frac{m^2}{4} + \frac{m^2}{4} + \frac{m^2-2m}{4} = m^2 \end{aligned} \tag{A.11}$$

mientras que para valores de m *impares* se tiene que

$$\begin{aligned}\#AND &= \otimes(\sum \mathbf{E_{i0}}) + \otimes(\sum \mathbf{O_{i0}}) + \otimes(\sum \mathbf{E_{i1}}) + \otimes(\sum \mathbf{O_{i1}}) \\ &= \frac{m^2-1}{4} + \frac{m^2+2m+1}{4} + \frac{m^2-2m+1}{4} + \frac{m^2-1}{4} = m^2 \end{aligned} \tag{A.12}$$

observándose, finalmente, que la complejidad en número de puertas AND es m^2 *independientemente* del valor de m y del trinomio irreducible generador del campo $GF(2^m)$ seleccionado.

A.2. Cálculo del número de puertas XOR

El número de puertas XOR de los multiplicadores viene determinado por la suma de puertas XOR dada por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ pertenecientes al vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$ a la que se añadiran, posteriormente, las puertas XOR necesarias para la adición de términos $\mathbf{E_{i1}}$, $\mathbf{O_{i1}}$ y de grupos $\mathbf{G_i}$ (si existen) pertenecientes al resto de vectores $\underline{\alpha}_\Omega^t\mathbf{K_i}$ $(i = 1, 2, \ldots, \tau)$. Estas puertas XOR adicionales dependen del tipo de trinomio irreducible seleccionado y se deducen de las expresiones dadas en el capítulo 9 por el método *transposicional*.

A continuación calculamos, de forma similar al apartado anterior, el número de puertas XOR de dos entradas correspondientes a cada conjunto de términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ existentes en el vector $\underline{\alpha}_\Omega^t\mathbf{K_0}$. Para ello se tiene en cuenta la existencia de $\lfloor \frac{m}{2} \rfloor$ términos $\mathbf{E_{i0}}$, de $\lceil \frac{m}{2} \rceil$ términos $\mathbf{O_{i0}}$, de $\lceil \frac{m-1}{2} \rceil$ términos $\mathbf{E_{i1}}$ y de $\lfloor \frac{m-1}{2} \rfloor$ términos $\mathbf{O_{i1}}$, así como los rangos de subíndices para cada término y sus complejidades espaciales dadas en la tabla 9.2.

Términos $\mathbf{E_{i0}}$ El número de puertas XOR determinado por los $\lfloor \frac{m}{2} \rfloor$ términos $\mathbf{E_{i0}}$ existentes en $\underline{\alpha}_\Omega^t\mathbf{K_0}$ se calcula en función del rango de subíndices i dado por las ecuaciones vistas en al capítulo anterior y utilizando los resultados dados en la tabla 9.2. Esta complejidad en número de puertas XOR viene dada

por la expresión siguiente

$$\oplus(\sum \mathbf{E_{i0}}) = \sum_{i=1}^{\lfloor \frac{m}{2} \rfloor} \oplus(\mathbf{E_{(2i)0}}) = \sum_{i=1}^{\lfloor \frac{m}{2} \rfloor} (2 \left\lceil \frac{2i}{2} \right\rceil - 1) = 2 \sum_{i=1}^{\lfloor \frac{m}{2} \rfloor} i - \left\lfloor \frac{m}{2} \right\rfloor$$
$$= \left\lfloor \frac{m}{2} \right\rfloor \left(\left\lfloor \frac{m}{2} \right\rfloor + 1 \right) - \left\lfloor \frac{m}{2} \right\rfloor = \left\lfloor \frac{m}{2} \right\rfloor^2 \tag{A.13}$$

donde $\oplus(\sum \mathbf{E_{i0}})$ representa el número de puertas XOR dado por todos los términos $\mathbf{E_{i0}}$, donde $\oplus$ representa el número de puertas XOR del término correspondiente y donde se ha utilizado el hecho de que la suma de la progresión aritmética $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$. Se observa que $\lfloor \frac{m}{2} \rfloor = \frac{m}{2}$ para valores *pares* de m, mientras que si m es *impar* entonces $\lfloor \frac{m}{2} \rfloor = \frac{m-1}{2}$. Se obtiene, por tanto, la siguiente expresión

$$\oplus(\sum \mathbf{E_{i0}}) = \begin{cases} \frac{m^2}{4} & m \ par \\ \frac{m^2-2m+1}{4} & m \ impar \end{cases} \tag{A.14}$$

Términos $\mathbf{O_{i0}}$ El número de puertas XOR determinado por los $\lceil \frac{m}{2} \rceil$ términos $\mathbf{O_{i0}}$ existentes en $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ viene dado en este caso por la expresión

$$\oplus(\sum \mathbf{O_{i0}}) = \sum_{i=1}^{\lceil \frac{m}{2} \rceil} \oplus(\mathbf{O_{(2i-1)0}}) = \sum_{i=1}^{\lceil \frac{m}{2} \rceil} (2 \left\lceil \frac{2i-1}{2} \right\rceil - 2) = \sum_{i=1}^{\lceil \frac{m}{2} \rceil} (2i - 2)$$
$$= \left\lceil \frac{m}{2} \right\rceil \left(\left\lceil \frac{m}{2} \right\rceil + 1 \right) - 2 \left\lceil \frac{m}{2} \right\rceil = \left\lceil \frac{m}{2} \right\rceil \left(\left\lceil \frac{m}{2} \right\rceil - 1 \right) \tag{A.15}$$

donde se ha tenido en cuenta el rango de subíndices para este caso y donde se han utilizado los resultados de la tabla 9.2 junto con el resultado de la suma de la progresión aritmética vista en los cálculos de términos anteriores. Se puede observar que $\lceil \frac{m}{2} \rceil = \frac{m}{2}$ para m *par* y que $\lceil \frac{m}{2} \rceil = \frac{m+1}{2}$ si m toma valores *impares*, obteniéndose la expresión

$$\oplus(\sum \mathbf{O_{i0}}) = \begin{cases} \frac{m^2-2m}{4} & m \ par \\ \frac{m^2-1}{4} & m \ impar \end{cases} \tag{A.16}$$

Términos $\mathbf{E_{i1}}$ En el cálculo de los términos anteriores (al igual que se hizo en la sección A.1 para el cálculo del número de puertas AND) se han diferenciado los valores de m *pares* o *impares* una vez obtenida una expresión general del número de puertas XOR. Esto ha sido posible debido a que el rango de valores de los subíndices i es el mismo para cualquier m (cuyo valor determina los límites superiores de los rangos). Esta igualdad en el rango de subíndices i no se cumple para los términos $\mathbf{E_{i1}}$, por lo que se debe distinguir entre valores *pares* o *impares* para m desde el inicio de los cálculos.

El número de términos $\mathbf{E_{i1}}$ existentes en $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ es $\lceil\frac{m-1}{2}\rceil$. Para m *par* se tiene que $\lceil\frac{m-1}{2}\rceil = \frac{m}{2}$, mientras que si m es *impar* entonces $\lceil\frac{m-1}{2}\rceil = \frac{m-1}{2}$. Por lo tanto, el número de puertas XOR determinado por los términos $\mathbf{E_{i1}}$ para valores *pares* de m viene dado por la expresión

$$\oplus(\sum \mathbf{E_{i1}}) = \sum_{i=1}^{\frac{m}{2}} \oplus(\mathbf{E_{(2i-2)1}}) = \sum_{i=1}^{\frac{m}{2}} (m-(2i-2)-2) = \sum_{i=1}^{\frac{m}{2}} (m-2i)$$

$$= m\left(\frac{m}{2}\right) - 2\sum_{i=1}^{\frac{m}{2}} i = m\left(\frac{m}{2}\right) - \frac{m}{2}\left(\frac{m}{2}+1\right) = \frac{m^2-2m}{4} \qquad \text{(A.17)}$$

donde se ha tenido en cuenta el rango de subíndices para este caso y donde se han utilizado los mismos datos adicionales que en los cálculos anteriores.

Para valores *impares* de m, se tiene que el número de puertas XOR determinado por los términos $\mathbf{E_{i1}}$ viene dado por la siguiente expresión

$$\oplus(\sum \mathbf{E_{i1}}) = \sum_{i=1}^{\frac{m-1}{2}} \oplus(\mathbf{E_{(2i-1)1}}) = \sum_{i=1}^{\frac{m-1}{2}} (m-(2i-1)-2) = \sum_{i=1}^{\frac{m-1}{2}} (m-2i-1)$$

$$= m\left(\frac{m-1}{2}\right) - \left(\frac{m-1}{2}\right)\left(\frac{m-1}{2}+1\right) - \frac{m-1}{2} = \frac{m^2-4m+3}{4} \qquad \text{(A.18)}$$

En resumen, el número de puertas XOR de dos entradas determinado por todos los términos $\mathbf{E_{i1}}$, $\oplus(\sum \mathbf{E_{i1}})$, viene dado por la expresión

$$\oplus(\sum \mathbf{E_{i1}}) = \begin{cases} \frac{m^2-2m}{4} & m \ par \\ \frac{m^2-4m+3}{4} & m \ impar \end{cases} \qquad \text{(A.19)}$$

Términos $\mathbf{O_{i1}}$ Para estos términos $\mathbf{O_{i1}}$ tampoco se cumple la igualdad en el rango de subíndices i con independencia de m, por lo que también se tiene que hacer la distinción entre valores *pares* o *impares* para m desde el comienzo del cálculo. En este caso, el número de términos $\mathbf{O_{i1}}$ existentes en $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ es $\lfloor\frac{m-1}{2}\rfloor$, observándose que $\lfloor\frac{m-1}{2}\rfloor = \frac{m-2}{2}$ para m *par* y que $\lfloor\frac{m-1}{2}\rfloor = \frac{m-1}{2}$ para valores de m *impares*.

Utilizando las mismas consideraciones y datos adicionales que en los cálculos anteriores, se tiene que el número de puertas XOR determinado por los términos $\mathbf{O_{i1}}$ para m *par* viene dado por la expresión

$$\oplus(\sum \mathbf{O_{i1}}) = \sum_{i=1}^{\frac{m-2}{2}} \oplus(\mathbf{O_{(2i-1)1}}) = \sum_{i=1}^{\frac{m-2}{2}} (m-(2i-1)-2) = \sum_{i=1}^{\frac{m-2}{2}} (m-2i-1)$$

$$= m\left(\frac{m-2}{2}\right) - \left(\frac{m-2}{2}\right)\left(\frac{m-2}{2}+1\right) - \frac{m-2}{2} = \frac{m^2-4m+4}{4} \qquad \text{(A.20)}$$

Para valores *impares* de m, se tiene que el número de puertas XOR determinado por los términos $\mathbf{O_{i1}}$ viene dado por la siguiente expresión

$$\begin{aligned}\oplus(\sum \mathbf{O_{i1}}) &= \sum_{i=1}^{\frac{m-1}{2}} \oplus(\mathbf{O_{(2i-2)1}}) = \sum_{i=1}^{\frac{m-1}{2}} (m-(2i-2)-2) = \sum_{i=1}^{\frac{m-1}{2}} (m-2i) \\ &= m\left(\frac{m-1}{2}\right) - \left(\frac{m-1}{2}\right)\left(\frac{m-1}{2}+1\right) = \frac{m^2-2m+1}{4} \end{aligned} \quad (A.21)$$

En resumen, el número de puertas XOR de dos entradas determinado por todos los términos $\mathbf{O_{i1}}$, $\oplus(\sum \mathbf{O_{i1}})$, viene dado por la siguiente expresión

$$\oplus(\sum \mathbf{O_{i1}}) = \begin{cases} \frac{m^2-4m+4}{4} & m \ par \\ \frac{m^2-2m+1}{4} & m \ impar \end{cases} \quad (A.22)$$

Número de puertas XOR El número total de puertas XOR dado por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ pertenecientes al vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$, viene determinado por la suma de las complejidades calculadas anteriormente para todos los términos $\mathbf{E_{i0}}$, $\mathbf{O_{i0}}$, $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$.

Por lo tanto, para valores *pares* de m se tiene que este número de puertas XOR viene dado por

$$\begin{aligned} \#XOR &= \oplus(\sum \mathbf{E_{i0}}) + \oplus(\sum \mathbf{O_{i0}}) + \oplus(\sum \mathbf{E_{i1}}) + \oplus(\sum \mathbf{O_{i1}}) \\ &= \frac{m^2}{4} + \frac{m^2-2m}{4} + \frac{m^2-2m}{4} + \frac{m^2-4m+4}{4} \\ &= m^2-2m+1 \end{aligned} \quad (A.23)$$

mientras que para valores de m *impares* se tiene que

$$\begin{aligned} \#XOR &= \oplus(\sum \mathbf{E_{i0}}) + \oplus(\sum \mathbf{O_{i0}}) + \oplus(\sum \mathbf{E_{i1}}) + \oplus(\sum \mathbf{O_{i1}}) \\ &= \frac{m^2-2m+1}{4} + \frac{m^2-1}{4} + \frac{m^2-4m+3}{4} + \frac{m^2-2m+1}{4} \\ &= m^2-2m+1 \end{aligned} \quad (A.24)$$

observándose, por tanto, que la complejidad en número de puertas XOR dada por los términos $\mathbf{E_{i(0,1)}}$ y $\mathbf{O_{i(0,1)}}$ pertenecientes a $\underline{\alpha}_\Omega^t \mathbf{K_0}$ es *independiente* del valor de m y del trinomio irreducible seleccionado, y que dicha complejidad toma el valor m^2-2m+1.

A este número de puertas XOR calculado, hay que añadirle una serie de puertas XOR adicionales cuyo número vendrá determinado por las expresiones dadas por el método *transposicional* para cada trinomio seleccionado. Estas puertas adicionales provienen de la nueva adición de términos $\mathbf{E_{i1}}$, $\mathbf{O_{i1}}$ (ya realizados por $\underline{\alpha}_\Omega^t \mathbf{K_0}$) y de grupos $\mathbf{G_i}$ (si existen) proporcionados por los vectores $\underline{\alpha}_\Omega^t \mathbf{K_i}$ $(i = 1, 2, \ldots, \tau)$. Este cálculo se realiza a continuación.

A.2.1. Complejidad para $f(x) = x^m + x^{m-1} + 1$

En la sección 9.3 se observó la existencia de grupos complejos $\mathbf{G_i}$ formados por la suma de términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$, y en la subsección 9.3.2 se dedujo que el coste de construcción de un grupo $\mathbf{G_i}$ es de una puerta XOR únicamente, ya que es la suma del grupo $\mathbf{G_{i-1}}$, ya construido, y del término $\mathbf{E_{(m-2-i)1}}$ o $\mathbf{O_{(m-2-i)1}}$ (dependiendo de m) que también ha sido construido previamente. De las ecuaciones 9.22 y 9.23 se deduce la existencia de $m-2$ grupos $\mathbf{G_i}$ diferentes, por lo que dichos grupos añadirán una complejidad adicional de $m-2$ puertas XOR.

En la subsección 9.3.2 también se observó que las coordenadas del producto están formadas por la suma de un término $\mathbf{O_{i0}}$ o $\mathbf{E_{i0}}$ con un grupo $\mathbf{G_i}$ o con el término $\mathbf{E_{(m-2)1}}$ cuando se trata de la coordenada d_{m-2}, por lo que cada una de las m coordenadas necesita una puerta XOR adicional.

El número total de puertas XOR de dos entradas necesarias para la realización de los multiplicadores sobre $GF(2^m)$ generado por el trinomio irreducible $f(x) = x^m + x^{m-1} + 1$ utilizando el método *transposicional*, viene determinado por la suma de las $m^2 - 2m + 1$ puertas XOR dadas por el vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$ y las puertas XOR adicionales anteriores, obteniéndose finalmente

$$\#XOR = (m^2 - 2m + 1) + (m - 2) + m = m^2 - 1 \tag{A.25}$$

A.2.2. Complejidad para $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m impar)

En la sección 9.4 se observó la existencia de grupos $\mathbf{G_i}$ y en la subsección 9.4.2 se dedujo que únicamente son necesarios los grupos $\mathbf{G_i}$ ($i < \Delta_m$), cuya complejidad es de 1 puerta XOR cada uno, ya que los grupos para $i > \Delta_m$ son iguales a estos y no es necesaria su construcción. Además se observó que no existe grupo $\mathbf{G_i}$ para la coordenada $i = \Delta_m$, requiriéndose, por tanto, Δ_m puertas XOR adicionales para la construcción de todos los grupos $\mathbf{G_i}$.

En la subsección 9.4.2 también se observó que para la obtención de las coordenadas del producto (formadas por sumas de términos $\mathbf{E_{i(0,1)}}$, $\mathbf{O_{i(0,1)}}$ y $\mathbf{G_i}$) son necesarias Δ_m puertas XOR (para las coordenadas de d_0 a $d_{\Delta_m - 1}$), una puerta XOR (para la coordenada d_{Δ_m}), $2(\Delta_m - 1)$ puertas XOR (para las coordenadas de $d_{\Delta_m + 1}$ a d_{m-2}) y una puerta XOR (para la coordenada d_{m-1}).

Finalmente, el número total de puertas XOR de dos entradas necesario para la realización de los multiplicadores sobre $GF(2^m)$ generado por el trinomio irreducible $f(x) = x^m + x^{\frac{m+1}{2}} + 1$ (m impar) utilizando el método *transposicional*, viene dado por la suma de $m^2 - 2m + 1$ puertas XOR del vector $\underline{\alpha}_\Omega^t \mathbf{K_0}$ y las puertas XOR adicionales anteriores, obteniéndose

$$\#XOR = (m^2 - 2m + 1) + \Delta_m + \Delta_m + 1 + 2(\Delta_m - 1) + 1 = m^2 - 1 \tag{A.26}$$

donde $\Delta_m = (m-1) - \Delta = (m-1) - \frac{m-1}{2} = \frac{m-1}{2}$ para este tipo de trinomios.

A.2.3. Complejidad para $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m impar)

En la sección 9.5 se observó la existencia de grupos $\mathbf{G_i}$ y en la subsección 9.5.2 se dedujo que únicamente son necesarios los grupos $\mathbf{G_i}$ ($i < \Delta_m$), cuya complejidad es de 1 puerta XOR cada uno, ya que los grupos para $i > \Delta_m$ son iguales a estos y no es necesaria su construcción. Además se observó que no existe grupo $\mathbf{G_i}$ para la coordenada $i = \Delta_m$, requiriéndose, por tanto, Δ_m puertas XOR adicionales para la construcción de todos los grupos $\mathbf{G_i}$.

En la subsección 9.5.2 también se observó que para la obtención de las coordenadas del producto son necesarias Δ_m puertas XOR (para las coordenadas de d_0 a d_{Δ_m-1}), una puerta XOR (para la coordenada d_{Δ_m}), $2\Delta_m$ puertas XOR (para las coordenadas de d_{Δ_m+1} a d_{m-3}), dos puertas XOR (para d_{m-2}) y una puerta XOR (para la coordenada d_{m-1}).

El número total de puertas XOR de dos entradas necesario para la realización de los multiplicadores sobre $GF(2^m)$ generado por el trinomio irreducible $f(x) = x^m + x^{\frac{m-1}{2}} + 1$ (m impar) utilizando el método *transposicional*, viene dado por la suma de $m^2 - 2m + 1$ puertas XOR del vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$ y las puertas XOR adicionales anteriores, obteniéndose finalmente

$$\#XOR = (m^2 - 2m + 1) + \Delta_m + \Delta_m + 1 + 2\Delta_m + 2 + 1 = m^2 - 1 \quad \text{(A.27)}$$

donde $\Delta_m = (m-1) - \Delta = (m-1) - \frac{m+1}{2} = \frac{m-3}{2}$ para este tipo de trinomios.

A.2.4. Complejidad para $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m par)

En la sección 9.6 se observó que para este tipo de trinomios no existen grupos complejos $\mathbf{G_i}$ que puedan ser compartidos entre las expresiones de las distintas coordenadas del producto. Las únicas comparticiones existentes corresponden a las de términos individuales $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$.

Observando la parte derecha de la tabla 9.14, se puede comprobar que el vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$ carece de los Δ_m términos $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$ de mayor subíndice. Sin embargo, estos términos aparecen entre los componentes del vector $\underline{\alpha}^t_\Omega \mathbf{K_1}$ (esta misma observación se puede deducir a partir de la ecuación 9.49). Por lo tanto, la complejidad en número de puertas AND y XOR calculada anteriormente para el vector $\underline{\alpha}^t_\Omega \mathbf{K_0}$, se sigue manteniendo para este trinomio.

Observando de nuevo la tabla 9.14 se tiene que para la obtención de las coordenadas del producto también son necesarias $2\Delta_m$ puertas XOR (para las coordenadas de d_0 a d_{Δ_m-1}), una puerta XOR (para la coordenada d_{Δ_m}), Δ_m puertas XOR (para las coordenadas de d_{Δ_m+1} a d_{m-2}) y una puerta XOR (para la coordenada d_{m-1}).

El número total de puertas XOR de dos entradas necesario para la realización de los multiplicadores sobre $GF(2^m)$ generado por el EST irreducible $f(x) = x^m + x^{\frac{m}{2}} + 1$ (m par) utilizando el método *transposicional* viene dado,

por tanto, por la suma de $m^2 - 2m + 1$ puertas XOR del vector $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ y las puertas XOR adicionales anteriores, obteniéndose finalmente

$$\#XOR = (m^2 - 2m + 1) + 2\Delta_m + 1 + \Delta_m + 1 = m^2 - \frac{m}{2} = m^2 - \Delta \quad \text{(A.28)}$$

donde $\Delta = \frac{m}{2}$ y donde $\Delta_m = (m-1) - \Delta = \frac{m-2}{2}$ para este tipo de trinomios.

A.2.5. Complejidad para $f(x) = x^m + x + 1$

En la sección 9.7 se observó que tampoco para este tipo de trinomios existen grupos complejos $\mathbf{G_i}$ que se puedan compartir entre las expresiones de las distintas coordenadas del producto. Por lo tanto, las únicas comparticiones existentes corresponden a las de términos individuales $\mathbf{E_{i1}}$ y $\mathbf{O_{i1}}$.

A partir de la ecuación 9.54, se puede observar que al número de puertas XOR dado por el vector $\underline{\alpha}_{\Omega}^{t}\mathbf{K_0}$ hay que añadirle 1 puerta XOR adicional (para la coordenada d_0), $2(m-2)$ puertas XOR (para el cálculo de las coordenadas d_1 a d_{m-2}) y 1 puerta XOR adicional (para la coordenada d_{m-1}).

Por lo tanto, el número total de puertas XOR de dos entradas necesario para la realización de los multiplicadores sobre $GF(2^m)$ generado por el trinomio irreducible $f(x) = x^m + x + 1$ utilizando el método *transposicional*, se calcula por medio de la siguiente suma

$$\#XOR = (m^2 - 2m + 1) + 1 + 2(m-2) + 1 = m^2 - 1 \quad \text{(A.29)}$$

Bibliografía

[Afa90] V.B. Afanasyev. Complexity of VLSI implementation of finite field arithmetic. En *II Int. Workshop on Algebraic and Combinatorial Coding Theory*, pags. 6–7, Septiembre 1990.

[Afa91] V.B. Afanasyev. On the complexity of finite field arithmetic. En *5th Joint Soviet-Swedish Int. Workshop on Information Theory*, pags. 9–12, Enero 1991.

[Ake78] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, Junio 1978.

[BBP91] P.R. Bhattacharjee, S.K. Basu y J.C. Paul. Translation of the problem of complete test set generation to pseudo-Boolean programming. *IEEE Transactions on Computers*, 40(7):864–867, 1991.

[BBR90] K.S. Brace, R.E. Bryant y R.L. Rudell. Efficient implementation of a BDD package. En *Proc. 27th ACM/IEEE DAC*, pags. 40–45, Junio 1990.

[BC95] R.E. Bryant y Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. En *Proc. 32nd ACM/IEEE DAC*, pags. 535–541, Junio 1995.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan y D.L. Dill. Sequential circuit verification using symbolic model checking. En *27th Design Automation Conference*, pags. 46–51, 1990.

[BCW80] M. Blum, A.K. Chandra y M.N. Wegman. Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10:80–82, Marzo 1980.

[Ber68] E.R. Berlekamp. *Algebraic coding theory.* McGraw-Hill, New York, 1968.

[Ber82] E.R. Berlekamp. Bit-serial Reed-Solomon encoders. *IEEE Trans. Information Theory*, 28:869–874, Noviembre 1982.

[BF85] F. Brglez y H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. En *IEEE Int. Symp. Circuits and Systems*, Junio 1985.

[BFG+93] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo y F. Somenzi. Algebraic decision diagrams and their applications. En *Proc. IEEE/ACM ICCAD'93*, pags. 188–191, Noviembre 1993.

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen y A.L. Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis.* Kluwer Academic, Boston, 1984.

[Bla83] R.E. Blahut. *Theory and practice of error control codes.* Addison-Wesley, Reading, Massachussets, 1983.

[Bla85] R.E. Blahut. *Fast algorithms for digital signal processing.* Addison-Wesley, Reading, Massachussets, 1985.

[BLW95] B. Bollig, M. Löbbing y I. Wegener. Simulated annealing to improve variable orderings for OBDDs. En *Int. Workshop Logic Synth.*, pags. 5.1–5.10, 1995.

[BMS95] J. Bern, C. Meinel y A. Slobodová. Efficient OBDD-based boolean manipulation in CAD beyond current limits. En *DAC'95*, pags. 408–413, 1995.

[Boo54] G. Boole. *An investigation of the laws of thought.* Walton, Londres, 1854 (Reimpreso por Dover Books), New-York, 1954.

[BP01] D.V. Bailey y C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[BRB90] K.S. Brace, R.L. Rudell y R.E. Bryant. Efficient implementation of a BDD package. En *27th Design Automation conference*, 1990.

[BRKM91] K.M. Butler, D.E. Ross, R. Kapur y M.R. Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. En *Proc. 28th ACM/IEEE DAC*, pags. 417–420, Junio 1991.

[Bro90] F.M. Brown. *Boolean reasoning. The logic of Boolean equations.* Kluwer Academic Publisher, 1990.

[Bry86] R.E. Bryant. Graph based algorithms for Boolean function representation. *IEEE Transactions on Computers*, C-35:677–690, Agosto 1986.

[Bry91] R.E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, C-40:206–213, Febrero 1991.

[BSSW98] B. Bollig, M. Sauerhoff, D. Sieling y I. Wegener. Hierarchy theorems for kOBDDs and kIBDDs. *Theor. Comput. Sci.*, 205:45–60, 1998.

[CB97] Y.-A. Chen y R.E. Bryant. *PHDD: an efficient graph representation for floating point circuit verification. En *Int. Conf. CAD*, pags. 2–7, 1997.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K.L. McMillan, J. Yang y X. Zhao. Multi terminal binary decision diagrams: An efficient data structure for matrix representation. En *Int. Workshop Logic Synth.*, pags. P6a:1–15, 1993.

[Cha00] L. Chai. *ESOP circuit minimization based on the function ON-set*. Master's Thesis. Mississippi State University, Mississippi, 2000.

[Coe89] D.R. Coelho. *The VHDL handbook*. Kluwer Academic Publishers, 1989.

[DBG96] R. Drechsler, B. Becker y N. Göckel. A genetic algorithm for variable ordering of OBDDs. *IEE Proceedings - Computers and Digital Techniques*, 143(6):364–368, Noviembre 1996.

[DBR96] R. Drechsler, B. Becker y S. Ruppertz. K*BMDs: a new data structure for verification. En *European Design Test Conference*, pags. 2–8, 1996.

[DDG98] R. Drechsler, N. Drechsler y W. Günther. Fast exact minimization of BDDs. En *DAC'98*, pags. 200–205, 1998.

[DDT78] M. Davio, J.P. Deschamps y A. Thayse. *Discrete and switching functions*. McGraw-Hill International, 1978.

[DH76] W. Diffie y M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[DS01] R. Drechsler y D. Sieling. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer*, 3(2):112–136, 2001.

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker y M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. En *DAC'94*, pags. 415–419, 1994.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[EP99] A.J. Elbirt y C. Paar. Towards an FPGA architecture optimized for public-key algorithms. En *The SPIE's Symposium on Voice, Video, and Data Communications*, Septiembre 1999.

[EP00] A.J. Elbirt y C. Paar. An FPGA implementation and performance evaluation of the Serpent Block Cipher. En *Eighth ACM International Symposium on Field-Programmable Gate Arrays*, Febrero 2000.

[EYCP00] A.J. Elbirt, W. Yip, B. Chetwynd y C. Paar. An FPGA implementation and performance evaluation of the AES Block Cipher candidate algorithm finalists. En *The Third Advanced Encryption Standard (AES3) Candidate Conference*, Abril 2000.

[FBT96] S.T.J. Fenn, M. Benaissa y D. Taylor. GF(2^m) multiplication and division over the dual basis. *IEEE Transactions on Computers*, 45(3):319–327, Marzo 1996.

[Fen93] S.T.J. Fenn. *Optimised algorithms and circuit architectures for performing finite field arithmetic in Reed-Solomon codecs.* PhD Thesis. University of Huddersfield, 1993.

[FFK88] M. Fujita, H. Fujisawa y N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. En *Proc. IEEE/ACM ICCAD'88*, pags. 2–5, Noviembre 1988.

[FFM93] M. Fujita, H. Fujisawa y Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Trans. CAD*, 12(1):6–12, Enero 1993.

[FHS78] S. Fortune, J. Hopcroft y E.M. Schmidt. *The complexity of equivalence and containment for free single variable program schemes*, volumen 62:227-240 de *Lecture Notes in Computer Science.* Springer-Verlag, New-York, 1978.

[FMK91] M. Fujita, Y. Matsunaga y T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. En *Proc. European Design Automation Conference*, pags. 50–54, 1991.

[FS90] S.J. Friedman y K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, C-39(5):710–713, 1990.

[GM96] J. Gergov y C. Meinel. Mod-2-OBDD's - a data structure that generalizes EXOR-sum-of-products and ordered binary decision diagrams. *Formal Methods in System Design*, 8:273–282, 1996.

[Gon84] G.H. Gonnet. *Handbook of algorithms and data structures.* Addison-Wesley Publishing Company, 1984.

[Gou88] R. Gould. *Graph theory.* The Benjamin/Cummings Publishing Company, Inc., California, 1988.

[GS97a] J.A. Gómez y J.M. Sánchez. An application of genetic algorithms to the ROBDD optimization. En *GALESIA'97*, pags. 290–295, Septiembre 1997.

[GS97b] J.A. Gómez y J.M. Sánchez. Minimización del tamaño de ROBDDs mediante el algoritmo de enfriamiento simulado. En *III Workshop de Iberchip*, pags. 194–201, Febrero 1997.

[GSB91] T.A. Gukkiver, M. Serra y V.K. Bhargava. The generation of primitive polynomials in GF(q) with independent roots and their application for power residue codes, VLSI testing and finite field multipliers using normal bases. *Int. J. Electronics*, 71(4):559–576, 1991.

[GSM96] J.A. Gómez, J.M. Sánchez y J.A. Moreno. Metodología basada en algoritmos genéticos para optimizar diagramas de decisión binarios. En *DCISs'96*, pags. 699–700, Noviembre 1996.

[GSMR98] J.A. Gómez, J.M. Sánchez, J.A. Moreno y D. Rodríguez. Distributed genetic algorithms for logical synthesis optimization. En *EIS'98*, pags. 331–337, Febrero 1998.

[Has98] M.A. Hasan. Double-basis multiplicative inversion inversion over $GF(2^m)$. *IEEE Transactions on Computers*, 47(9):960–970, Septiembre 1998.

[HB95] M.A. Hasan y V.K. Bhargava. Architecture for a low complexity rate-adaptive Reed-Solomon encoder. *IEEE Transactions on Computers*, 44(6):938–942, Junio 1995.

[HK00] A. Halbutogullari y Ç.K. Koç. Mastrovito multiplier for general irreducible polynomials. *IEEE Transactions on Computers*, 49(5):503–518, Mayo 2000.

[HMM85] S.L. Hurst, D.M. Miller y J.C. Muzio. *Spectral techniques in digital logic*. Academic Press, 1985.

[HTDR88] I.S. Hsu, T.K.Truong, L.J. Deutsch y I.S. Reed. A comparison of VLSI architecture of finite field multipliers using dual, normal, or standard bases. *IEEE Transactions on Computers*, 37(6):735–739, Junio 1988.

[HTRG88] I.S. Hsu, T.K.Truong, I.S. Reed y N. Glover. A VLSI architecture for performing finite field arithmetic with reduced table lookup. *Linear Algebra and its Applications*, 98:249–262, 1988.

[HWB92] M.A. Hasan, M.Z. Wang y V.K. Bhargava. Modular construction of low complexity parallel multipliers for a class of finite fields $GF(2^m)$. *IEEE Transactions on Computers*, 41(8):962–971, Agosto 1992.

[HWB93] M.A. Hasan, M.Z. Wang y V.K. Bhargava. A modified Massey-Omura parallel multiplier for a class of finite fields. *IEEE Transactions on Computers*, 42(10):1278–1280, Octubre 1993.

[IB99] J.L. Imaña y V.M. Barragán. Cube set method for switching operations computation. Technical Report 24, Dpto. Arquitectura de Computadores y Automática, UCM, 1999.

[IDB97] J.L. Imaña, R. Drechsler y B. Becker. A hybrid approach to combine deterministic and probabilistic verification. Technical Report 93, Albert-Ludwigs-University Freiburg, Noviembre 1997.

[Ima93] J.L. Imaña. VHDL description of a digital neural network. En *VHDL-Forum for CAD in Europe, Spring'93 Meeting*, pags. 139–150, Marzo 1993.

[Ima02] J.L. Imaña. Bit-parallel arithmetic implementations over finite fields $GF(2^m)$ with reconfigurable hardware. *Acta Applicandae Mathematicae*, 73(3):337–356, Septiembre 2002.

[IS93] J.L. Imaña y J.M. Sánchez. Specification and synthesis of an ALU in VHDL. En *IASTED International Symposium Modelling, Identification and Control*, pags. 385–386, Febrero 1993.

[IS95] J.L. Imaña y J.M. Sánchez. Una aproximación a la verificación de circuitos lógicos por medio de Diagramas de Decisión. En *XXV Reunión Bienal de la Real Sociedad Española de Física*, pags. 311–312, Septiembre 1995.

[IS01] J.L. Imaña y J.M. Sánchez. Formulation for the computation of Boolean operations. *IEE Proceedings - Computers and Digital Techniques*, 148(6):189–195, Noviembre 2001.

[ISF02] J.L. Imaña, J.M. Sánchez y M. Fernández. Método de multiplicación canónica sobre campos $GF(2^m)$ generados por AOPs orientado a hardware reconfigurable. En *II Jornadas sobre Computación Reconfigurable y Aplicaciones, JCRA-2002*, pags. 215–220, Septiembre 2002.

[ISL95] J.L. Imaña, J.M. Sánchez y J. Lanchares. Utilización de grafos para la verificación de circuitos lógicos. En *X Congreso de Diseño de Circuitos Integrados*, pags. 349–350, Noviembre 1995.

[ISV95] J.L. Imaña, J.M. Sánchez y J. Vicente. Probabilistic verification of logic functions using Decision Diagrams. En *2nd Advanced Training Course: Mixed Design of VLSI Circuits-Education of Computer Aided Design of Modern VLSI Circuits (MixVLSI'95)*, pags. 271–276, Mayo 1995.

[ISY91] N. Ishiura, H. Sawada y S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. En *Proc. IEEE/ACM ICCAD'91*, pags. 472–475, Noviembre 1991.

[IT89] T. Itoh y S. Tsujii. Structure of parallel multipliers for a class of finite fields $GF(2^m)$. *Information and Computation*, 83:21–40, 1989.

[JABF92] J. Jain, J.A. Abraham, J. Bitner y D.S. Fussell. *Probabilistic verification of Boolean functions*, volumen 1:61-115 de *Formal Methods in System Design*. Kluwer Academic Publisher, 1992.

[Jai91] R. Jain. *The art of computer systems performance analysis*. Wiley, 1991.

[Jai95] J. Jain. On an arithmetic transform of Boolean functions. En *Workshop on Applications of the Reed-Muller Expansion in Circuit design*, IFIP WG 10.5, pags. 70–79, Agosto 1995.

[Jai96] J. Jain. *Arithmetic transform of Boolean functions*, capítulo 6. Kluwer Academic Publishers, 1996.

[JBA+97] J. Jain, J. Bitner, M.S. Abadir, J.A. Abraham y D.S. Fussell. Indexed BDDs: algorithmic advances in techniques to represent and verify Boolean functions. *IEEE Transactions on Computers*, 46:1230–1245, 1997.

[Jen94] J.H. Jenkins. *Designing with FPGAs and CPLDs*. PTR Prentice Hall, 1994.

[JPHS91] S.-W. Jeong, B. Plessier, G.D. Hachtel y F. Somenzi. Variable ordering and selection for FSM traversal. En *Proc. ACM/IEEE ICCAD*, pags. 476–479, 1991.

[Kar53] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of A.I.E.E.*, 72(1):593–599, Noviembre 1953.

[KB81] S.K. Kumar y M.A. Breuer. Probabilistic aspects of Boolean switching functions via a new transform. *Journal of ACM*, 28(3):502–520, Julio 1981.

[KDS95] T. Kozlowski, E.L. Dagless y J.M. Saul. An enhanced algorithm for the minimization of Exclusive-OR sum-of-products for incompletely specified functions. En *IEEE International Conference on Computer Design, ICCD95*, pags. 244–249, Octubre 1995.

[KL01] C.C. Kao y Y.T. Lai. A routability driven technology mapping algorithm for LUT based FPGA designs. *IEICE Trans. Fundamentals*, E84-A(11):2690–2696, Noviembre 2001.

[Kli95] A. Klindworth. FPLD-Implementation of computations over finite fields $GF(2^m)$ with applications to error control coding. En *5th. Intern. Workshop on Field-Programmable Logic and Applications*, LNCS975, pags. 261–271, Septiembre 1995.

[KO63] A. Karatsuba y Y. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys.-Dokl.*, 7(7):595–596, 1963.

[KR93] U. Kebschull y W. Rosenstiel. Efficient graph-based computation and manipulation of functional decision diagrams. En *European Conf. Design Automation*, pags. 278–282, 1993.

[KS98] Ç.K. Koç y B. Sunar. Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(3):353–356, Marzo 1998.

[KSR92] U. Kebschull, E. Schubert y W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. En *European Conf. Design Automation*, pags. 43–47, 1992.

[KT89] B. Krishnamurthy y J.G. Tollis. Improved techniques for estimating signal probabilities. *IEEE Transactions on Computers*, 38(7):1041–1045, 1989.

[Kum83] H. Kummer. Recommendation for space data system standards: Telemetry channel coding: Issue-1. *Consult. Comm. Space Data Syst.*, Septiembre 1983.

[Lee59] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technology Journal*, 38:985–999, 1959.

[Leo01] M. Leone. A new low complexity parallel multiplier for a class of finite fields. En *Cryptographic Hardware and Embedded Systems, CHES'2001*, Mayo 2001.

[LN83] R. Lidl y H. Niederreiter. *Finite fields.* Addison-Wesley, Reading, Massachussets, 1983.

[LS92] Y.-T. Lai y S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. En *DAC'92*, pags. 608–613, 1992.

[LSU89] R. Lipsett, C.F. Schaefer y C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Press, 1989.

[Mas89] E.D. Mastrovito. VLSI design for multiplication over finite fields $GF(2^m)$. *Lecture Notes in Computer Science*, 357:297–309, Marzo 1989.

[Mas91] E.D. Mastrovito. *VLSI architectures for computation in Galois fields.* PhD Thesis. Dept. Electr. Eng., Linköping University, Linköping, Sweden, 1991.

[MB88] J.C. Madre y J.P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. En *Proc. 25th ACM/IEEE DAC*, pags. 205–210, Junio 1988.

[McC56] E.J. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, 35, 1956.

[McM93] K.L. McMillan. *Symbolic model checking.* Kluwer Academic, Boston-London-Dordrecht, 1993.

[Men93] A.J. Menezes, editor. *Applications of finite fields.* Kluwer Academic, Boston-London-Dordrecht, 1993.

[Min92] S. Minato. Minimum-width method of variable ordering for binary decision diagrams. *IEICE Trans. Fundamentals*, E75-A(3):392–399, Marzo 1992.

[Min93] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. En *Proc. 30th ACM/IEEE DAC*, pags. 272–277, Junio 1993.

[MIY90] S. Minato, N. Ishiura y S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. En *Proc. 27th IEEE/ACM DAC*, pags. 52–57, Junio 1990.

[MKW89] M. Morii, M. Kasahara y D.L. Whiting. Efficient bit-serial multiplication and the discrete-time Wiener-Hopft equation over finite fields. *IEEE Transactions on Information Theory*, 35:1177–1183, Noviembre 1989.

[ML85] A.M. Michelson y A.H. Levesque. *Error-control techniques for digital communication.* Willey & Sons Inc., 1985.

[MO84] J.L. Massey y J.K. Omura. Apparatus for finite field computation. *US Patent Application*, pags. 21–40, 1984.

[MOV97] A.J. Menezes, P.C.V. Oorschot y S.A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1997.

[MOVW89] R. Mullin, I. Onyszchuk, S.A. Vanstone y R. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Math.*, 22:149–161, 1988/1989.

[MS98] C. Meinel y H. Sack. $\oplus$-OBDDs - a BDD structure for probabilistic verification. En *Proceedings of the International Workshop on Logic Synthesis, IWLS98*, pags. 19–24, Junio 1998.

[MS01a] C. Meinel y H. Sack. Heuristics for $\oplus$-OBDD minimization. En *Proceedings of the IEEE/ACM International Workshop of Logic and Synthesis, IWLS2001*, pags. 304–309, Junio 2001.

[MS01b] C. Meinel y H. Sack. Improving XOR-Node placement for $\oplus$-OBDDs. En *Proceedings of 5th. Int. Workshop on Applications of the Reed-Muller Expansion in Circuit Design, Reed-Muller 2001*, pags. 51–56, Agosto 2001.

[Mul54] D.E. Muller. Application of Boolean algebra to switching circuit design and error detection. *IRE Trans. on Electronic Computing*, 3:6–12, 1954.

[MWBSV88] S. Malik, A. Wang, R.K. Brayton y A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. En *Proc. IEEE/ACM ICCAD'88*, pags. 6–9, 1988.

[OD95] J.V. Oldfield y R.C. Dorf. *Field Programmable Gate Arrays*. John Wiley & Sons, Inc., 1995.

[OM86] J.K. Omura y J.L. Massey. Computational method and apparatus for finite field arithmetic. *US Patent Number 4,587,627*, Mayo 1986.

[OP99] G. Orlando y C. Paar. A super-serial Galois fields multiplier for FPGAs and its application to public-key algorithms. En *IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'99*, Abril 1999.

[OP00] G. Orlando y C. Paar. A high-performance reconfigurable elliptic curve processor for $GF(2^m)$. En *Cryptographic Hardware and Embedded Systems, CHES'2000*, Agosto 2000.

[Paa94] C. Paar. *Efficient VLSI architectures for bit-parallel computation in Galois fields*. PhD Thesis. Institute for Experimental Mathematics, University of Essen, Essen, Germany, Junio 1994.

[Paa96] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, Julio 1996.

[Pay77] R.W. Payne. Reticulation and other methods for reducing the size of printed diagnostic keys. *Journal of General Microbiology*, 98:595–597, 1977.

[PFR97] C. Paar, P. Fleischmann y P. Roelse. Efficient multiplier architectures for Galois fields. *IEEE Transactions on Computers*, 47(2):162–170, Febrero 1997.

[PFSR99] C. Paar, P. Fleischmann y P. Soria-Rodríguez. Fast arithmetic for public-key algorithms in Galois fields with composite exponents. *IEEE Transactions on Computers*, 48(10):1025–1034, Octubre 1999.

[Pic74] W. Del Picchia. A numerical algorithm for the resolution for boolean equations. *IEEE Transactions on Computers*, 23:983–986, 1974.

[Pin89] A. Pincin. A new algorithm for multiplication in finite fields. *IEEE Transactions on Computers*, 38(7):1045–1049, Julio 1989.

[PL95] C. Paar y N. Lange. A comparative VLSI synthesis of finite field multipliers. En *3rd International Symposium on Communication Theory & Applications*, Julio 1995.

[PM75] K.P. Parker y E.J. McCluskey. Probabilistic treatment of general combinational networks. *IEEE Transactions on Computers*, pags. 668–670, Junio 1975.

[PR97] C. Paar y M. Rosner. Comparison of arithmetic architectures for Reed-Solomon decoders in reconfigurable hardware. En *Fifth Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'97*, Abril 1997.

[Rab96] J.M. Rabaey. *Digital integrated circuits. A design perspective.* Prentice-Hall International, 1996.

[Ree54] L.S. Reed. A class of multiple error-correcting codes and their decoding scheme. *IRE Trans. on Information Theory*, 3:38–42, 1954.

[Rud74a] S. Rudeanu. An algebraic approach to boolean equations. *IEEE Transactions on Computers*, 23:206–207, 1974.

[Rud74b] S. Rudeanu. *Boolean functions and equations.* Amsterdam, The Netherlands: North-Holland, 1974.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. En *Proc. IEEE/ACM ICCAD'93*, pags. 42–47, Noviembre 1993.

[SB90] T. Sasao y Ph. Besslich. On the complexity of mod-2 sum PLAs. *IEEE Transactions on Computers*, 39:262–266, 1990.

[SBS95] R.T. Stanion, A.D. Bhattachary y C. Sechen. An efficient method for generating exhaustive test sets. *IEEE Transactions on CAD*, 14(12):1516–1525, 1995.

[SBSV96] P. Stephan, R.K. Brayton y A.L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on CAD*, 15(9):1167–1176, 1996.

[Sch93] B. Schneier. *Applied cryptography.* Willey & Sons, 1993.

[SDG95] A. Shen, S. Devadas y A. Ghosh. Probabilistic manipulation of Boolean functions using Free Boolean Diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and systems*, 14(1):87–95, 1995.

[SEF93] J.M. Saul, B. Eschermann y J. Froessl. Two-level logic circuits using EXOR sums of products. *IEE Proceedings - Computers and Digital Techniques*, 140(6):348–356, 1993.

[SF96] T. Sasao y M. Fujita, editores. *Representations of discrete functions.* Kluwer Academic, Boston-London-Dordrecht, 1996.

[Sha49] C.E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[Ste89] I. Stewart. *Galois theory.* Chapman & Hall Mathematics, 2nd Edition, 1989.

[Svo63] S. Svoboda. An algorithm for solving Boolean equations. *IEEE Transactions on Electronic Computers*, 12:557–559, 1963.

[SW95] D. Sieling y I. Wegener. Graph driven BDDs - a new data structure for Boolean functions. *Theor. Comput. Sci.*, 141:283–310, 1995.

[THY93] S. Tani, K. Hamaguchi y S. Yajima. The complexity of the optimal variable ordering of a shared binary decision diagram. Technical Report 93-6, Department of Information Science, Faculty of Science, University of Tokyo, December 1993.

[TLROL93] P. Trabado, A. Lloris-Ruíz y J. Ortega-Lopera. Solution of switching equations based on a tabular algebra. *IEEE Transactions on Computers*, 42(5):591–596, Mayo 1993.

[TP97] P. Tafertshofer y M. Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2):243–270, 1997.

[Ung94] S. Unger. Some additions to 'Solution of switching equations based on a tabular algebra'. *IEEE Transactions on Computers*, 43(3):365–367, 1994.

[VHD00] *IEEE Standard VHDL Language Reference Manual.* Std. 1076-2000, 2000.

[Waa97] S. Waack. On the descriptive and algorithmic power of parity ordered binary decision diagrams. En *14th Symp. Theor. Aspects Comp. Sci., LNCS 1200*, pags. 201–212. New York: Springer-Verlag, 1997.

[Wan86] C.C. Wang. A generalized algorithm to design finite field normal basis multipliers. En *The Telecommunications and Data Acquisition Progress Report 42-87*, pags. 125–139, Julio-Septiembre 1986.

[Weg94] I. Wegener. The size of reduced OBDD's and optimal read-once branching programs for almost all Boolean functions. *IEEE Transactions on Computers*, 43(11):1262–1269, 1994.

[WM98] Y. Wang y C. McCrosky. Solving Boolean equations using RO-SOP forms. *IEEE Transactions on Computers*, 47(2):171–177, 1998.

[Woa01] W. Woan. Hankel matrices and lattice paths. *Journal of Integer Sequences*, 4(1): Artículo 01.1.2, 2001.

[Wol01] T. Wollinger. *Computer Architectures for Cryptosystems Based on Hyperelliptic Curves*. Master's Thesis. Worcester Polytechnic Institute, Worcester, Abril 2001.

[WP90] C.C. Wang y D. Pei. A VLSI design for computing exponentiation in $GF(2^m)$ and its application to generate pseudorandom number sequences. *IEEE Transactions on Computers*, C-39(2):258–262, Febrero 1990.

[WTS+85] C.C. Wang, T.K. Truong, H.M. Shao, L.J. Deutsch, J.K. Omura y I.S. Reed. VLSI architectures for computing multiplications and inverses in $GF(2^m)$. *IEEE Transactions on Computers*, C-34(8):709–717, Agosto 1985.

[Xil92] *XILINX: The programmable gate array data book*. Xilinx, Inc., San Jose, CA, 1992.

[YRT84] C.S. Yeh, I.S. Reed y T.K. Truong. Systolic multipliers for finite fields $GF(2^m)$. *IEEE Transactions on Computers*, C-33(4):357–360, Abril 1984.

[ZP01] T. Zhang y K.K. Parhi. Systematic design of original and modified Mastrovito multipliers for general irreducible polynomials. *IEEE Transactions on Computers*, 50(7):734–749, Julio 2001.

Índice de figuras

Índice de tablas