UNIVERSIDAD D CÓRDOBA

INSTITUTO DE ESTUDIOS DE POSTGRADO
PROGRAMA DE DOCTORADO EN INGENIERÍA Y TECNOLOGÍA

Uso eficiente de aritmética redundante en FPGAs

MANUEL AGUSTÍN ORTIZ LÓPEZ

MEMORIA DE TESIS PARA OPTAR AL GRADO DE DOCTOR

LÍNEA DE INVESTIGACIÓN: INSTRUMENTACIÓN Y ELECTRÓNICA INDUSTRIAL

DIRECTORES:

DR. D. FRANCISCO JOSÉ BELLIDO OUTEIRIÑO

DR. D. FRANCISCO JAVIER HORMIGO AGUILAR

DR. D. JULIO VILLALBA MORENO

CÓRDOBA, SEPTIEMBRE DE 2013

TITULO: *Uso eficiente de aritmética redundante en FPGAs*

AUTOR: *Manuel Agustín Ortiz López*

Campus de Rabanales
Ctra. Nacional IV, Km. 396 A
14071 Córdoba

www.uco.es/publicaciones
publicaciones@uco.es

TÍTULO DE LA TESIS:

Uso eficiente de aritmética redundante en FPGAs

DOCTORANDO/A:

Manuel Agustín Ortiz López

INFORME RAZONADO DE LOS DIRECTOR/ES DE LA TESIS

(se hará mención a la evolución y desarrollo de la tesis, así como a trabajos y publicaciones derivados de la misma).

Los doctores D. Francisco José Bellido Outeiriño, profesor del Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica de la Universidad de Córdoba, y D. Francisco Javier Hormigo Aguilar y D. Julio Villalba Moreno, profesores del Departamento de Arquitectura de Computadores de la Universidad de Málaga informan que el trabajo titulado "Uso eficiente de aritmética redundante en FPGAs" que presenta D. Manuel Agustín Ortiz López, Licenciado en Ciencias Físicas, para optar al grado de Doctor por la Universidad de Córdoba ha sido realizado bajo nuestra dirección.

La presente Tesis Doctoral está bien estructurada y organizada, cubriendo de manera correcta antecedentes, objetivos, desarrollo, conclusiones y bibliografía. La metodología planteada y puesta en práctica demuestra ser la adecuada para resolver los objetivos propuestos.

Establece con claridad la situación actual de las operaciones utilizadas en numerosas áreas tecnológicas, especialmente en bioinformática, tratamiento digital de imagen, computación financiera, criptografía, etc., donde se está extendiendo cada vez más el uso de FPGA (Field Programmable Gate Array) para la implementación de unidades aritméticas adaptadas a cada aplicación concreta. Además de las representaciones en punto fijo o en punto flotante, es habitual encontrar hoy día implementaciones en FPGAs que utilizan representaciones en punto flotante con doble precisión o representaciones en punto flotante decimal. Estas nuevas aplicaciones requieren cada vez más precisión numérica lo que conlleva un número de bits elevado de los operandos; lo que justifica la necesidad de optimizar el rendimiento de estas operaciones en las FPGAs.

La innovación de esta Tesis Doctoral está basada en la necesidad de obtener una implementación eficiente de aritmética "Carry-save" en las estructuras de FPGAs actuales. Una vez desarrollada la implementación eficiente en FPGAs de bajo coste, se ha aplicado este resultado en algunas operaciones como multiplicación de operandos de ancho de palabra elevado u operaciones MAC, consiguiendo un elevado rendimiento. En especial un aumento de la velocidad de operación en sumas y productos imposible de conseguir utilizando aritmética convencional. Paralelamente, la evolución de las FPGAs ha propiciado la aparición de FPGAs más potentes que además de mejorar en velocidad y consumo de potencia, ofrecen al usuario más

recursos. Y en el presente trabajo se ha verificado si los buenos resultados obtenidos de la utilización de la aritmética “Carry-save”, son ampliables a estas FPGAs nuevas. Para lo que ha adaptado de nuevo los sumadores carry-save a estas estructuras, consiguiendo mejores resultados incluso que en las anteriores FPGAs y ha definido un nuevo formato de aritmética “Carry-save” que se ha denominado doble “carry-save” que permite aumentar el rendimiento ofrecido por la aritmética “Carry-save” clásica. Se ha demostrado con este trabajo que la aritmética “carry-save” es la mejor alternativa para la realización de multiplicadores de ancho de palabra elevado. Por tanto los resultados del trabajo son directamente aplicables, y contribuyen no sólo al conocimiento sino al desarrollo tecnológico y a la innovación.

En cuanto a la producción científica generada por la Tesis Doctoral destacar el número y calidad de las publicaciones derivadas de ésta, así como la distribución temporal de las mismas.

- **M. Ortiz**, F. Quiles, J. Hormigo, F. Jaime, J. Villalba, and E. Zapata, “Efficient implementation of carry-save adders in FPGAs”, in Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, 7-9 2009, pp. 207 –210.
- Moreno, C. D.; Quiles, F.J.; **Ortiz, M. A.**; Brox, M.; Hormigo, J.; Villalba, J.; Zapata, E.L., "Efficient mapping on FPGA of convolution computation based on combined CSA-CPA accumulator," Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on , vol., no., pp.419,422, 13-16 Dec. 2009, doi: 10.1109/ICECS.2009.5410903.
- Quiles, F.J.; **Ortiz, M.**; Brox, M.; Moreno, C.D.; Hormigo, J.; Villalba, J., "UCORE: Reconfigurable Platform for Educational Purposes," Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on , vol., no., pp.109,114, 13-15 Dec. 2010, doi: 10.1109/ReConFig.2010.60.
- Moreno-Moreno, C., Martinez-Jiménez, M., Bellido-Outeiriño, F., Hormigo-Aguilar, F., **Ortiz-Lopez, M.**, Quiles-Latorre, F., "Convolution computation in FPGA based on carry-save adders and circular buffers", International ICST Conference on IT Revolutions, Cordoba-Spain, 2011.doi:10.1007/978-3-642-32304-1_20.
- Quiles-Latorre, F., **Ortiz-Lopez, M.**, Montijano-Vizcaino, M., Moreno-Moreno, C., Brox-Jiménez, M., Hormigo-Aguilar, F., Villalba-Moreno,Julio, "Acelerador Hardware de bajo coste para bus PCI Convencional", Seminario Anual de Automática, Electrónica Industrial e Instrumentación 2012, Guimarães, Portugal.
- **Ortiz-López, M.**, Quiles-Latorre, F., Moreno-Moreno, C., Brox-Jiménez, M., "CAN2PCI: Placa con interfaz al bus CAN y PCI con finalidad docente", Revista de Formación universitaria, ISSN: 0718-5006, pp.31-38, Chile 2009, doi: 10.4067/S0718-50062009000300006.
- **Ortiz-López, M.**, Quiles-Latorre, F., Moreno-Moreno, C., Brox-Jiménez, M.," ADQPCI: Placa de adquisición de datos con fines docentes", Revista de Formación universitaria, ISSN: 0718-5006, pp. 25-30, Chile 2009, doi: 10.4067/S0718-50062009000300005.

Por todo ello, se autoriza la presentación de la tesis doctoral.

Córdoba, 26 de septiembre de 2013

Firma del/de los director/es

Fdo.: Fco. José Bellido Outeiriño

Fdo.: Fco. Javier Hormigo Aguilar

Fdo.: Julio Villalba Moreno

A Francisco J. Quiles
Maestro, compañero y amigo

Agradecimientos

Una vez finalizado este trabajo, deseo mostrar mi agradecimiento a todas las personas que me han ayudado y animado. Esta tesis ha finalizado gracias a la ayuda, tanto técnica como personal, de todas las personas integrantes del Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica de la Universidad de Córdoba.

A Francisco Javier Quiles Latorre, por su ayuda prestada no sólo en la realización de la tesis, sin la cual no hubiera sido posible, sino por tantas horas que hemos compartido y disfrutado en casi 25 años de trabajo. Cuando por el año 1988 me sentaron en una mesa a su lado, en el laboratorio de I+D de la empresa "Fujitsu Limited", no me podía imaginar que íbamos a compartir tan buenos momentos, ni mucho menos, que iba a aprender tanto a su lado. Gracias también por tu ánimo y optimismo en todos los proyectos que hemos realizado.

A mis directores de tesis, Francisco Javier Hormigo, Julio Villalba y Francisco Bellido por haber sido compañeros además de directores. Gracias por vuestro apoyo constante y por haberme hecho sentir que en esta tesis no estaba solo.

A Edmundo Sáez, Laura Ramírez, Miguel Ángel Montijano, Carlos Diego Moreno, María Brox, Andrés Gersnoviez y Lili Tapia por vuestro apoyo y compañía en los malos momentos. Y gracias por hacerme disfrutar de cada día.

A mi hija Olimpia y a mis padres, hermanas, cuñado, y sobrinos por ser el motor de mi vida y estar siempre a mi lado.

Resumen

Hasta hace pocos años, la utilización de aritmética redundante en FPGAs había sido descartada por dos razones principalmente. En primer lugar, por el buen rendimiento que ofrecían los sumadores de acarreo propagado, gracias a la lógica de de acarreo que poseían de fábrica y al pequeño tamaño de los operandos en las aplicaciones típicas para FPGAs. En segundo lugar, el excesivo consumo de área que las herramientas de síntesis obtenían cuando mapeaban unidades que trabajan en *carry-save*.

En este trabajo, se muestra que es posible la utilización de aritmética redundante *carry-save* en FPGAs de manera eficiente, consiguiendo un aumento en la velocidad de operación con un consumo de recursos razonable. Se ha introducido un nuevo formato redundante doble *carry-save* y se ha demostrado que la manera óptima para la realización de multiplicadores de elevado ancho de palabra es la combinación de multiplicadores empotrados con sumadores *carry-save*.

Abstract

Till a few years ago, redundant arithmetic had been discarded to be use in FPGA mainly for two reasons. First, the efficient results obtained using carry-propagate adders thanks to the carry-logic embedded in FPGAs and the small sizes of operands in typical FPGA applications. Second, the high number of resources that the synthesis tools utilizes to implement carry-save circuits.

In this work, it is demonstrated that carry-save arithmetic can be efficiently used in FPGA, obtaining an important speed improvement with a reasonable area cost. A new redundant format, double carry-save, has been introduced, and the optimal implementation of large size multipliers has been shown based on embedded multipliers and carry-save adders.

Índice

Índice de figuras

Índice de tablas

Índice de Gráficos

Acrónimos empleados

ASIC Circuitos integrados para aplicaciones específicas

CLA Sumador de acarreo anticipado

CLB *Configurable Logic Blocks*

CPA Sumador de acarreo propagado

CRA Sumador de acarreo encadenado

CS *Carry-save*

CSA *Carry-save* Aritmética o sumador *carry-save*

DSP Procesamiento digital de señales

FA Sumador completo

FPCA *Field Programmable Counters Array*

FPGA *Field Programmable Gate Array*

FPCT *Field Programmable Compressor Tree*

GPC *Generalized Parallel Counters*

GPU Procesador para tratamiento de gráficos

LUT *Look-up table*

LUT4 *Look-up table* con 4 entradas

LUT6 *Look-up table* con 6 entradas

MAC Unidad de multiplicación y acumulación

PCI Bus de interconexión de periféricos

PPA *Parallel prefix adders*

SAD Operación de suma de las diferencias aritméticas

SD *Signed-digit*

VHDL Lenguaje de alto nivel de descripción hardware

1. Introducción

Breve resumen

En este capítulo se hará una breve descripción de los sumadores que aparecerán en este trabajo así como una breve introducción sobre FPGAs.

1.1. Aritmética en Computadores

La aritmética en computadores se orienta en dos direcciones. Por un lado, se intenta dotar a los procesadores principales de unidades aritméticas flexibles que implementan cualquier operación aritmética mediante programación, y por otro lado, se utilizan unidades especializadas y/o coprocesadores para determinadas operaciones que requieren un elevado tiempo de cómputo o una precisión numérica variable, o para operaciones complejas sobre un gran volumen de datos.

La realización de operaciones aritméticas sobre el procesador principal tiene la ventaja de la flexibilidad ofrecida por la programación y un consumo de recursos limitados. Las unidades aritméticas integradas en los procesadores son principalmente unidades que trabajan en punto flotante, y que no siempre permiten realizar operaciones en tiempo real, sobre todo en el tratamiento digital de señales (incluido el tratamiento de imágenes), criptografía, operaciones en punto fijo, operaciones de precisión variable, etc. Para aumentar el rendimiento de estas operaciones, se han dotado a los computadores de unidades especializadas como las unidades para procesamiento digital de señales (DSP) incluidas en algunos procesadores principales, o coprocesadores para tratamiento de gráficos (GPU). Aún así, existen operaciones muy costosas cuya realización no es posible mediante programación en procesadores especializados. Se hace necesaria la utilización de unidades dedicadas que realizan algoritmos concretos por hardware, aumentando de esta forma el rendimiento.

El desarrollo de los dispositivos programables, o más comúnmente llamados dispositivos reconfigurables, ofrecen una nueva posibilidad para la realización de operaciones complejas que requieren un elevado tiempo de cómputo. Además de la posibilidad de reconfiguración de estos dispositivos, permiten la computación con un elevado rendimiento, superando en dos órdenes de magnitud a los procesadores de propósito general en algoritmos aritméticos [1][2].

Surgen con mucha fuerza los sistemas reconfigurables y los aceleradores hardware que incluyen estos dispositivos reconfigurables. Los aceleradores hardware, especialmente los que se conectan a los buses internos del sistema computacional, permiten la programación del algoritmo que se desee implementar por hardware, desde el procesador principal, de una manera cómoda e incluso *on-line*, en los dispositivos

reconfigurables que contienen [3][4]. La Figura 1 muestra la estructura que habitualmente se utiliza para realizar aceleradores hardware conectados al bus PCI [5][6].

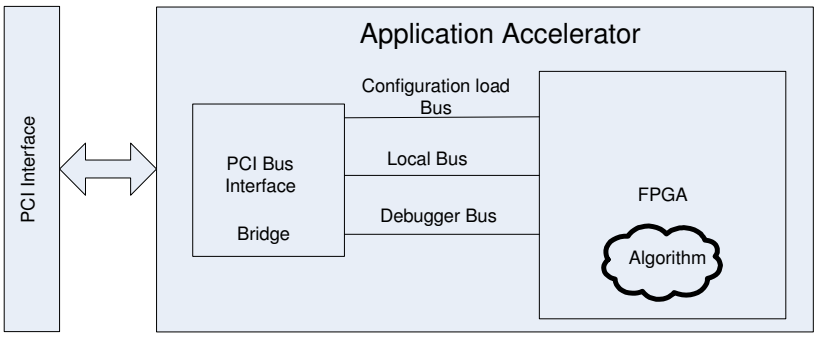

Figura 1. Acelerador hardware conectado al bus PCI

Los dispositivos reconfigurables ofrecen un elevado rendimiento ya que las operaciones se realizan por hardware, y el hardware permite aprovechar el paralelismo inherente al algoritmo. En este sentido la realización de algoritmos en *Field Programmable Gate Array* (FPGA) ofrecen una buena alternativa por la alta integración de recursos lógicos y la facilidad de programación. Estos dispositivos se utilizan ampliamente en numerosas aplicaciones, especialmente en procesamiento digital de señales [7][8][9][10], aceleradores hardware [11][12][13], criptografía [14][15], etc.

Los algoritmos complejos implementados en estas unidades dedicadas, hacen un uso intensivo de operaciones aritméticas, motivo por el cual los fabricantes de FPGAs intentan optimizar al máximo la implementación de operaciones de suma y producto para conseguir un rendimiento elevado con un mínimo coste de recursos. Este compromiso hace que los fabricantes integren sumadores rápidos con un consumo de recursos pequeño y multiplicadores muy rápidos aunque con un ancho de operandos pequeño. Sin embargo, a veces los requisitos de velocidad de las operaciones implementadas, obligan al diseñador a utilizar otros tipos de sumadores y multiplicadores más eficientes aunque con un consumo de recursos mayor, o a utilizar FPGAs rápidas y costosas. La implementación de sumadores y multiplicadores eficientes será la alternativa que pretende este trabajo.

El documento se estructura de la siguiente manera, en primer lugar, se hará una breve introducción sobre sumadores y aritmética redundante, para fijar las ideas de partida más importantes de este trabajo. Posteriormente se hará un estudio de las

incipientes líneas seguidas actualmente para la implementación de aritmética redundante en FPGAs. En los capítulos siguientes se mostrará cómo es posible una implementación eficiente de aritmética redundante en FPGAs, proponiendo soluciones que optimizan el consumo de recursos, una de las principales desventajas de la aritmética redundante, frente a la aritmética tradicional. Para finalizar se presentarán unas conclusiones y los trabajos futuros.

1.2. Sumadores

En este apartado se realizará una breve descripción de los sumadores más utilizados para realizar la operación de suma por hardware. No se pretende mostrar todos los sumadores que se han descrito hasta el momento, sino los más utilizados en FPGAs y los que se proponen como alternativa en este trabajo. Para un estudio más detallado se puede consultar [16][17].

Para realizar una operación de suma básica en aritmética digital, se utiliza un sumador completo (FA), que suma tres bits (habitualmente un bit de cada operando y un acarreo de entrada) y genera un bit de suma y un acarreo de salida. Un sumador completo se puede observar en la Figura 2.

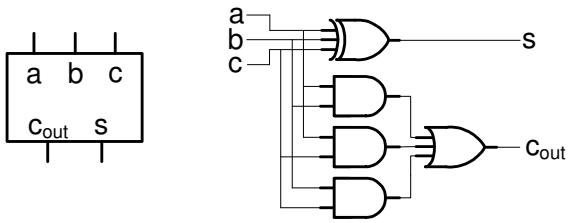

Figura 2. Sumador completo de 1 bit

Si se quiere realizar una suma de operandos de cualquier número de bits se pueden componer sumadores completos de 1 bit, que trabajan de la misma manera en como se realizaría la suma de una manera natural. Como muestra la Figura 3, se suma cada dígito junto con el acarreo generado en el dígito anterior. Estos sumadores reciben el nombre de *Basic Carry-Ripple Adder* (CRA o RCA).

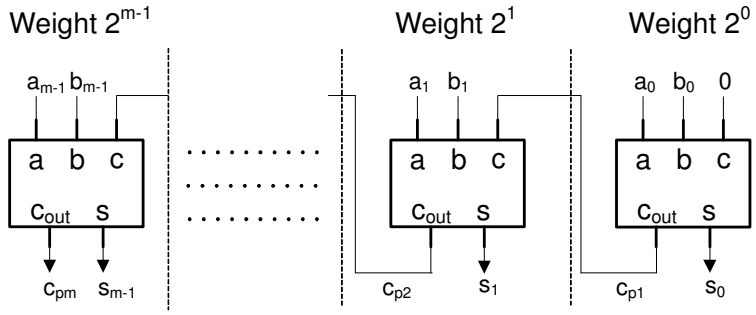

Figura 3. Sumador CRA de m bits

El sumador CRA tiene el inconveniente de que la suma en un bit concreto no finaliza hasta que no se haya generado el acarreo en el bit de peso anterior. Por tanto, la suma completa no se obtiene hasta que no se sume el último bit que no podrá tenerse hasta que no se obtengan los acarreos de todas las etapas anteriores. El retardo de un sumador CRA básico dependerá del tiempo empleado en generar el acarreo ($Delay_{Cn}$) en cada bit y el número de bits del sumador (n) (Ecu. 1).

$$S_{delay} = \sum_{1}^{n} Delay(Cn) \qquad \text{Ecu.1}$$

En este tipo de sumadores el esfuerzo para conseguir una mayor velocidad se pone en disminuir el retardo en la generación del acarreo e incluso anularlo [16][17]. En las FPGAs se intenta disminuir en la medida de lo posible el retardo del acarreo, pero hay que hacer notar que aunque sea pequeño, el retardo de un sumador CRA depende del número de bits.

Del otro lado, una solución teórica sería diseñar sumadores que realicen la suma completa en dos niveles. En este caso el retardo teórico sería igual a la generación de todos los bits de suma en paralelo e igual al retardo de los dos niveles de puertas. Esto en la realidad solamente se puede conseguir para un número de bits pequeño, ya que para un número elevado de bits, son necesarias un gran número de puertas lógicas con un número de entradas elevado.

Una solución intermedia, entre la rápida y costosa técnica de dos niveles y la lenta pero simple técnica de propagación del acarreo, para conseguir un sumador de *m* bits con un consumo de hardware razonable, sería tener un sumador completo de *m* bits que generase todos los acarreos simultáneamente. Este tipo de sumadores se les llama *Carry-Lookahead Adder* (CLA). Los CLA se nombran por el número de bits que

computan a las vez, de esta forma se tiene un CLA de módulo 1 CLA-1, que suma 1 bit de cada operando, de módulo 2, que suma 2 bits de cada operando, y en general se les llama de módulo *m* CLA-m, que suma *m* bits de cada operando. La parte del circuito que genera el acarreo no toma sus entradas directamente de las entradas primarias, sino que tiene como entrada otras señales auxiliares. En la literatura se pueden encontrar varios diseños de sumadores de acarreo anticipado [16][18]. Uno de los diseños consiste en generar dos señales auxiliares g_i y p_i, a partir de las entradas primarias, para generar el acarreo, como muestra la Figura 4 para 4 bits. Con estas funciones las ecuaciones de un sumador quedan:

$$s_i = g_i \oplus p_i \oplus c_{i-1}$$

$$c_i = g_i + p_i c_{i-1}$$

Siendo:

$$g_i = x_i y_i \text{ , y } p_i = x_i + y_i$$

Por ejemplo, para un sumador de acarreo anticipado de 4 bits las funciones necesarias para implementar el acarreo anticipado serían:

$$c_1 = g_0 + p_0 c_{in}$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_{in}$$
$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$$
$$c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$$

Por último las 4 funciones de suma quedan:

$$S_0 = p_0 \oplus g_0 \oplus C_{in}$$
$$S_1 = p_1 \oplus g_1 \oplus C_0$$
$$S_2 = p_2 \oplus g_2 \oplus C_1$$
$$S_3 = p_3 \oplus g_3 \oplus C_2$$

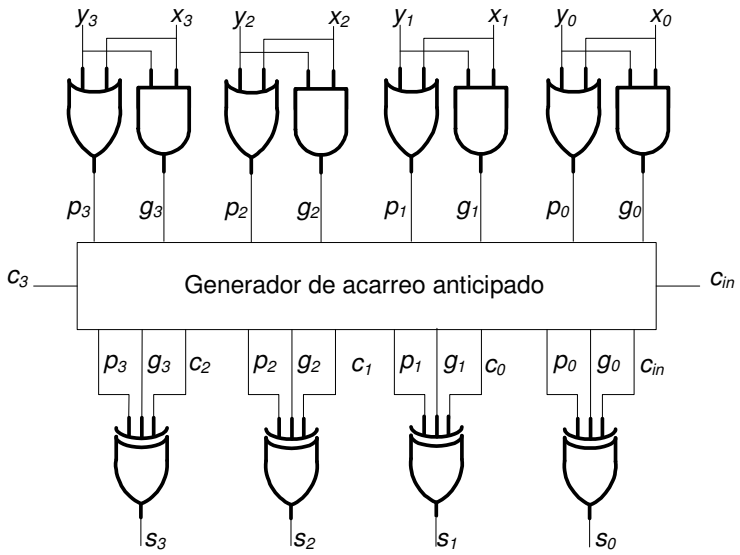

Figura 4. Sumador CLA-4.

En este caso el retardo teórico sería igual a la generación del bit de suma. Sin embargo, para un número elevado de bits es necesaria una gran cantidad de lógica, con un número de entradas también elevado, al igual que sucedía con el sumador de dos niveles. Por este motivo, este tipo de sumadores se descartan para su utilización en FPGAs. En [19] se estudia el rendimiento de este tipo de sumador y otras variantes similares obteniendo como conclusión, que utilizando las cadenas de acarreo "de fábrica" que contienen las FPGAs para formar sumadores CRA se obtienen sumadores más rápidos y que ocupan menos área.

Entre estas soluciones extremas comentadas anteriormente, la más lenta y que consume menos recursos, y la más rápida pero con un consumo elevadísimo de recursos, existen otras alternativas para implementar sumadores en FPGAs:

- Aumentar el *radix*, es decir, tener un sumador completo de más de un bit que aproveche los recursos lógicos que contienen las FPGAs.
- Optimizar la cadena de acarreo a nivel lógico, creando nuevas estructuras como se hace en los sumadores “parallel prefix adders (PPA)” que son variaciones de los sumadores CLA.
- Intentar optimizar la generación de acarreo cuidando el diseño a nivel físico y de rutado, una de las opciones por la que optan los fabricantes de FPGAs.
- Intentar realizar sumas intermedias sin necesidad de propagar el acarreo, como se hace en la aritmética redundante.

En los circuitos integrados para aplicaciones específicas (ASIC) la aritmética redundante se ha utilizado como una buena alternativa para aumentar el rendimiento frente a la aritmética tradicional, especialmente en la suma de multioperandos. La suma de multioperandos es una operación utilizada frecuentemente y que aparece en muchos algoritmos como multiplicación [20][21], filtros [22][23], SAD [24], y otros [13][25][26][27], por citar algunos.

1.3. Aritmética y sumadores carry-save

La representación redundante de números se utiliza para reducir el tiempo de suma, limitando el camino de la cadena de acarreo a varios bits. De esta manera el tiempo empleado para la realización de la suma no depende del número de bits de los operandos. Las representaciones más habituales son *carry-save* (CS) y *signed-digit* (SD). La aritmética *carry-save* (CSA) se utiliza ampliamente cuando se requieren sumar un número de operandos elevado y en las operaciones internas de los multiplicadores.

El sumador CSA básico realiza la suma de tres operandos utilizando un array de sumadores completos de un bit, pero sin conectar la cadena de acarreo, como se muestra en la Figura 5. El resultado es un número redundante en representación CS que está compuesta de una palabra de suma (S) y palabra de acarreo (C).

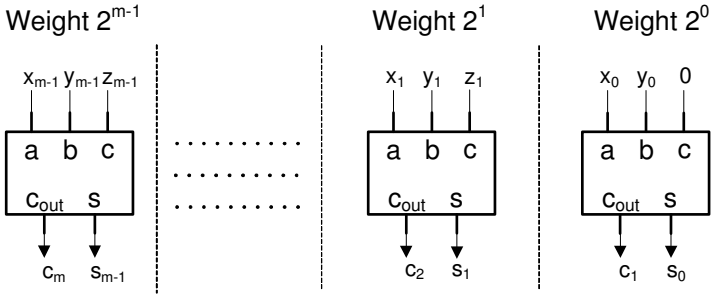

Figura 5. Sumador *carry-save* de *m* bits.

Por tanto, la suma de tres operandos X, Y, Z de n-bits se representa por dos números C y S.

$$X + Y + Z = C + S$$

Los números C y S de n-bits se obtienen sin propagación de acarreo con el retardo de un solo sumador completo. Esta representación se le llama redundante, puesto que muchas combinaciones de C y S producen el mismo número. En la Figura 6,

a modo de ejemplo, se puede observar cómo al sumar dos números distintos que dan el mismo resultado en aritmética convencional producen dos combinaciones distintas de C y S. Por tanto, hasta que no se realice la suma C y S no se tendrá un número en la representación convencional. Para ello será necesario utilizar un sumador convencional que suma dos operandos y da como resultado un único número. Mientras todas las operaciones se realicen en aritmética CSA no es necesaria realizar la conversión. Además es posible realizar la conversión "on-the-fly" [28][29], aunque incrementa el consumo de recursos ya que es necesario realizar una computación paralela.

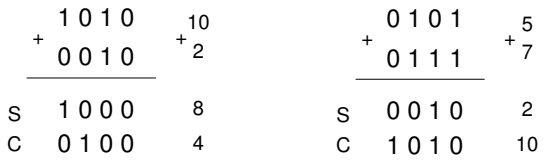

Figura 6. Ejemplo de suma en CSA

Este circuito CS desde otro punto de vista realiza la reducción de tres números binarios a dos números binarios, por lo que se le llama compresor [3:2] o contador [3:2]. Si se desea sumar dos números CS se necesita una reducción de 4 a 2. Esto se puede realizar utilizando dos compresores [3:2] como muestra la Figura 7. A este circuito se le llama compresor [4:2]. En este caso, el tiempo de computación para dos números de *n* dígitos CS es el de dos sumadores completos. Obsérvese como la propagación de acarreo se corta ya que el acarreo propagado del primer nivel se conecta a la entrada del sumador completo del segundo nivel del bit de peso siguiente.

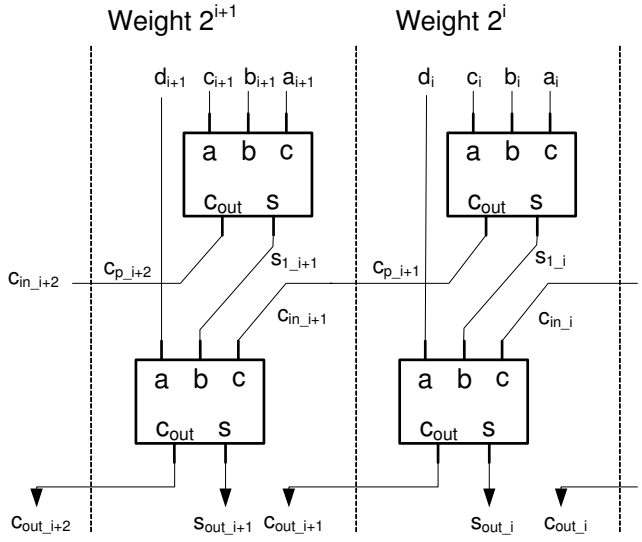

Figura 7. Compresor [4:2] construido a partir de dos compresores [3:2].

Por último, comentar que se pueden sumar *m* operandos de un bit y producir una palabra de suma S y otra de acarreo C. De este modo se tienen compresores [5:2] que realiza una reducción de cinco bits a dos, compresores [6:2] que realiza una reducción de 6 bits de entrada a dos y en general compresores [*p*:*t*] que realiza la suma de *p* bits de entrada y produce una salida de *t* bits.

Una de las desventajas de la representación CS es que el número de bits involucrados en la suma es el doble. Para reducir el hardware una alternativa es la utilización de un *radix* superior, que será una de las opciones propuestas en la literatura y se propondrá también como una de las alternativas de implementación que se mostrarán en este trabajo, para optimización en FPGAs.

La utilización de la representación CS es especialmente útil en algoritmos que requieran muchas sumas intermedias puesto que todas las sumas se pueden llevar a cabo en representación CS y donde la conversión a representación convencional no consuma el tiempo ahorrado en la representación CS. Este es el caso de operaciones de acumulación, suma de multioperandos, multiplicación, división, raíz cuadrada, etc.

La aritmética CS se utiliza habitualmente para la suma de multioperandos especialmente en multiplicadores, debido a que se debe realizar la suma de los productos parciales [30][31]. Para llevarla a cabo se crean árboles de compresores (no se deben confundir con los compresores como circuito) que generan la salida en aritmética redundante CS. La suma S y el acarreo C se suman finalmente para producir una salida convencional.

El resto del trabajo está centrado en los sumadores *carry-save*, puesto que la extensión a la representación *signed-digit* se puede conseguir fácilmente invirtiendo determinadas entradas y salidas en los compresores, como se demostró en [32].

1.4. Operaciones de suma y producto en FPGAs

Los principales recursos disponibles en una FPGA para la implementación de circuitos combinacionales y secuenciales son los *Configurable Logic Blocks* (CLB). Los CLBs están distribuidos formando un array bidimensional. Cada CLB (Figura 8) está conectado a una matriz de conmutación que se conecta a su vez a una matriz general de rutado, de forma que todos los CLBs se pueden interconectar programando estas matrices. Los CLBs están divididos a su vez en *slice*. Los s*lices* no están

conectados entre sí, sino que van conectados a una matriz de conmutación. Entre *slice* existe una conexión directa llamada C_{in} vista desde la entrada del *slice* y C_{out} vista desde la salida del *slice* y se utilizan principalmente para la propagación de acarreo cuando se sintetizan sumadores. Estas conexiones entre *slices* adyacentes no pasan por la matriz de rutado lo que reduce significativamente el retardo.

Los CLBs varían entre fabricantes y por cada serie de un mismo fabricante [33][34][35][36]. En las modernas FPGAs los fabricantes han integrado además multiplicadores dedicados, bloques DSP para procesamiento digital de señales, bloques RAM, etc. En [37] Kuon y Rose demuestran que la utilización eficiente de todos estos recursos reduce la distancia entre las implementaciones en FPGAs y ASICs.

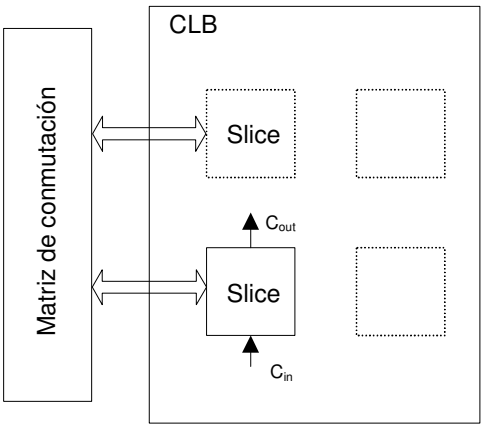

Figura 8. ***Configurable Logic Block.***

En la síntesis de las operaciones de suma y producto, a partir de una descripción en lenguajes de alto nivel, los sintetizadores mapean estos operadores teniendo en cuenta los recursos lógicos que contienen las FPGAs. Así, el operador de suma se sintetiza aprovechando la lógica de acarreo creada "de fábrica" que contienen las FPGAs, se crean así sumadores CRA, que en adelante llamaremos sumadores de acarreo propagado (CPA). En el caso del operador producto, se sintetiza preferiblemente utilizando los multiplicadores empotrados si están disponibles.

Durante el desarrollo de este trabajo se ha ido produciendo un cambio en los elementos lógicos que contienen los s*lices*, de forma que se está pasando de unas series de FPGAs que contienen unos determinados recursos, especialmente los generadores de funciones o *look-up tables* (LUT) con 4 entradas y una salida, que en adelante llamaremos LUT4, a otras mucho más potentes y con más recursos lógicos. Estas

últimas, que en adelante llamaremos LUT6, se pueden configurar como una *look-up table* con 6 entradas y 1 salida o dos *look-up tables* de 5 entradas y 1 salida por LUT.

Aunque en la descripción de los sistemas reconfigurables se utilizan lenguajes universales de alto nivel como VHDL [38], Verilog [39], o ABEL [40], cada fabricante de FPGAs incluye elementos lógicos distintos, especialmente para las operaciones aritméticas básicas. Por otro lado, un mismo fabricante ha comercializado distintas series de FPGAs, y podemos encontrar FPGAs con distintos elementos lógicos. Teniendo en cuenta esta realidad, no es posible realizar una implementación eficiente de un algoritmo para todas las FPGAs existentes en el mercado. Razón por la que se han elegido en este trabajo las FPGAs de la compañía Xilinx [35], porque sin lugar a dudas es uno de los líderes en FPGAs en el momento actual, y se han elegido los dispositivos que contienen LUT4 por su precio y bajo consumo de potencia y los que contienen LUT6 por ser los nuevos dispositivos. Dentro de las series que llevan estos componentes básicos se han elegido las series de bajo coste Spartan 3 [41] y Spartan 6 [42] para mostrar los datos y las implementaciones realizadas. Algunos de los resultados que se presentan se hacen también extensivos a otros fabricantes.

1.5. Motivación del trabajo

La utilización de FPGAs se ha extendido con mucha fuerza a nuevos campos de aplicación, como computación de alto rendimiento, computación financiera, criptografía, etc. Estas nuevas aplicaciones requieren una precisión mucho más elevada que las implementadas en los tradicionales bloques DSP. Además de las representaciones en punto fijo o en punto flotante, es habitual encontrar hoy día implementaciones en FPGAs que utilizan representaciones en punto flotante con doble precisión o representaciones en punto flotante decimal. En estos casos se utilizan tamaños de operandos de 54 bits y mayores.

La implementación de operaciones aritméticas con operandos de tamaño elevado, aumenta el camino crítico de las señales en los sumadores cuando se utiliza la cadena de acarreo que implementan los sumadores CPA en las FPGAs, lo que reduce el rendimiento. En la implementación de ASIC, la aritmética redundante, especialmente *carry-save*, se ha utilizado para aumentar el rendimiento para operandos de tamaño elevado, o un número elevado de operandos. La aritmética *carry-save* permite que el

camino critico que recorren las señales en los sumadores sea prácticamente independiente del tamaño de operando.

La utilización de la representación *carry-save* en FPGAs se descartó hasta hace pocos años debido a varias razones. En primer lugar por el tamaño pequeño de operandos que se utilizaban en las aplicaciones típicas de FPGAs. En segundo lugar, los sumadores de acarreo propagado (CPA) que poseían de fábrica ofrecían un buen rendimiento debido a que la lógica de la cadena de acarreo se había optimizado. Por último, el excesivo consumo de área por las herramientas de síntesis cuando mapeaban unidades que trabajan en *carry-save*. Sin embargo, en los últimos años, varios trabajos han demostrado los beneficios en el rendimiento, de la utilización de sumadores *carry-save*, cuando el tamaño de los operandos crece y además es posible mapear de manera eficiente los sumadores *carry-save* en FPGAs reales con un aumento de área muy pequeño[44][45][46].

Por tanto, merece la pena estudiar la implementación de sumadores y multiplicadores en FPGAs utilizando aritmética redundante en especial aritmética *carry-save*, ya que es una técnica muy utilizada para mejorar el rendimiento de los sumadores en las implementaciones en circuitos dedicados ASIC.

Por otro lado, operaciones como la multiplicación en punto fijo se implementan utilizando los multiplicadores empotrados en FPGA altamente optimizados. Estos multiplicadores tienen un tamaño fijo, y se deben combinar para obtener operaciones con operadores de tamaño elevado. En este caso, son necesarios sumadores para los productos parciales que limitan la velocidad del multiplicador. Se intuye que la aritmética CS puede ser la mejor alternativa para la realización de multiplicadores de ancho de palabra elevado.

1.6. Comentarios sobre los datos y las herramientas de desarrollo utilizadas

En primer lugar hay que hacer notar que se han utilizado dos herramientas de análisis temporal para las FPGAs, una de ellas forma parte del entorno ISE [47] y la otra es ModelSim [48]. La herramienta ISE es el entorno de desarrollo de FPGAs de Xilinx y determina el retardo analizando el camino más crítico que es la manera más rápida de determinar el retardo. Sin embargo, en determinadas implementaciones puede existir un

camino largo pero que realmente ninguna señal lo recorre porque se corta realmente por algún elemento lógico (puerta, multiplexor, etc.), al no tenerse nunca la combinación que permite que la información se propague por este camino. En este caso el retardo proporcionado al analizar los caminos no es correcto y es necesaria una herramienta de simulación temporal y contemplar todas las posibles combinaciones en las entradas y ver el resultado en las salidas. Para estos casos el retardo se ha medido utilizado ModelSim.

A lo largo del desarrollo del trabajo han aparecido distintas versiones de la herramienta ISE que incluye el sintetizador "Xilinx Synthesis Technology" (XST) y que puede influir a la hora de sintetizar una descripción VHDL. Por este motivo los datos ofrecidos se han recalculado con la última versión que se tenía disponible en el momento de comenzar la redacción de este trabajo (ISE versión 12.1), a excepción de las síntesis manuales con primitivas, ya que estas síntesis no dependen de la versión.

Por último, se quiere enfatizar que los datos calculados se han obtenido registrando las entradas y salidas de los circuitos sintetizados, para dar una idea real de la velocidad del circuito. Este hecho permite también convertir los datos de retardo a datos de frecuencia de una manera directa e inequívoca.

Todos los diseños realizados y descritos en este trabajo están altamente optimizados en velocidad si nos circunscribimos a los *slices*, con retardos mucho menores que los que se muestran en las tablas. Gran parte de los retardos de los circuitos están asociados a la red de rutado. No es verosímil aportar los datos para circuitos que se implementan completamente dentro de un *slice*, donde los retardos son muy pequeños, ya que en algún momento esas señales tendrán que conectarse a otras partes que están fuera del *slice*. Aún hecha esta aclaración, en algunos circuitos se describirá exactamente como se han realizado las medidas.

Debido a que se han registrado las entradas y salidas, en algunos casos el número de LUT o *slices* dados para los circuitos puede variar ligeramente del cálculo teórico, bien porque sea necesario para registrar alguna señal concreta o bien, porque por razones que se escapan a nuestro conocimiento de la herramienta ISE, ésta desperdicia algunos recursos. Por ejemplo, en lugar de utilizar el *flip-flop* que tiene próximo dentro de un *slice*, utiliza otro fuera del *slice*. No se ha querido filtrar estos

casos porque la influencia en los datos obtenidos es mínima, y corresponde con los datos aportados por la herramienta.

Se pueden obtener síntesis de los circuitos en las FPGAs con ligeras variaciones dependiendo de los parámetros que se configuren en la herramienta de síntesis XST de Xilinx. Por ejemplo, se puede pedir el mayor esfuerzo en velocidad o en área, esto da lugar a distintas versiones sintetizadas y para no ocupar el trabajo que se presenta con numerosos datos innecesarios, se ha elegido la síntesis más adecuada dependiendo del estudio que se esté realizando en ese momento.

2. Aritmética redundante en FPGAs: Situación actual

Breve resumen

En este capítulo se hará una breve descripción de los trabajos publicados relacionados con la implementación de aritmética redundante y de multiplicadores en FPGAs. Hay que hacer notar, que son muy pocos los trabajos publicados relacionados con la implementación a bajo nivel de sumadores *carry*-save, ya que la utilización de sumadores *carry-save* en FPGAs se había descartado hasta hace algunos años. De ahí que el trabajo que presentamos pretenda estudiar la parte que tiene que ver con la implementación a bajo nivel.

2.1. Introducción

Aunque la utilización de aritmética redundante es suficientemente conocida y empleada en ASIC, sobre todo para suma de multioperandos, no ha sido utilizada hasta hace unos años en FPGAs y aún actualmente sus detractores la descartan por el deficiente mapeo que se consigue debido a que las herramientas de alto nivel no utilizan este tipo de aritmética. Sin embargo, en los últimos años se propone la utilización de la aritmética redundante como una alternativa para aumentar la velocidad de computación aunque suponga un aumento en el consumo de recursos. Además varios trabajos como [44], [45], [49], [50], [51], etc., demostraron que era posible un mapeo eficiente en FPGAs utilizando la descripción adecuada.

La implementación de aritmética redundante en FPGAs, sobre todo en cuanto a la suma multioperando, tiene dos aspectos. El primero de ellos es cómo conseguir de una manera automatizada los árboles de compresores, sintetizables a partir de una descripción en alto nivel y utilizando unas librerías de compresores básicos. Cuando se utiliza el operador suma en un lenguaje de alto nivel, por ejemplo VHDL, la herramienta de síntesis implementa la suma de los operandos utilizando los sumadores CPA, y se hace necesario automatizar la generación de un árbol de compresores óptimo para la suma redundante. El otro aspecto es si esa librería de compresores básicos se ha implementado de manera eficiente. Ambos aspectos influyen enormemente en el rendimiento de los árboles de compresores utilizados en cada aplicación. Por tanto en algunos trabajos, como en [45] Parandeh y otros, proponen una síntesis de compresores mixta *top-down* y *bottom-up* para conseguir el mayor rendimiento en el menor área posible.

Además de los trabajos dedicados a la implementación de aritmética CSA en FPGA, otros trabajos proponen incluso la modificación de la arquitectura de las FPGAs para que incluyan bloques diseñados con aritmética CSA para aumentar de esta manera el rendimiento. Por otro lado, numerosos trabajos de aplicaciones concretas, demuestran la ventaja de utilización de aritmética CSA en algoritmos recursivos, modificando algoritmos anteriores, y que la proponen como la mejor alternativa. A continuación se realizará una retrospectiva de los trabajos publicados hasta el momento clasificándolos en:

- Aplicaciones específicas y multiplicadores
- Mapeo eficiente de compresores basados en la estructura interna de las FPGAs.
- Diseño de árboles de compresores para suma de multioperandos
- Propuestas de modificación del hardware de las FPGAs

2.2. Aplicaciones que han demostrado los beneficios de la utilización de aritmética redundante en FPGAs

En este apartado se quiere citar trabajos que detallan los beneficios de la utilización de aritmética redundante, bien por aplicar algoritmos ya conocidos en los que se ha introducido sumadores CS, o algoritmos novedosos que sacan partido a la aritmética CS. No se van a incluir en este apartado los trabajos que combinan aritmética redundante con multiplicadores empotrados para realizar multiplicaciones de un tamaño de operandos elevado, que se estudiaran en el capítulo 4 de este trabajo.

Muchos trabajos están dedicados a la multiplicación modular Montgomery [52]. Se van a destacar aquellos que describen una mejora en el rendimiento por la utilización de aritmética *carry-save*. En [53] Manochehri y Pourmozafari implementan un nuevo algoritmo que utiliza sumadores *carry-save* que obtiene una notable mejora en el rendimiento tanto para ASIC como FPGAs.

En [54] Shieh y otros modifican algoritmos previos para utilizar solamente compresores [3:2]. Implementan sus algoritmos en librerías estándar CMOS y en la FPGA Virtex II de Xilinx. El interés en utilizar solamente compresores [3:2] en los algoritmos, se debe a que los autores suponen que un compresor [4:2] tiene el doble de retardo que un compresor [3:2], pero no es cierto en FPGAs, como se podrá comprobar por los datos de velocidad que se mostrarán en este trabajo.

En [55] Sutter y otros han modificado los algoritmos previos de multiplicación Montgomery para poder utilizar aritmética CSA. Utilizan compresores [3:2] y [4:2], desgraciadamente no ofrecen detalles ni rendimientos de estos compresores que han utilizado. En [56] Wu y otros presenta las mejoras en velocidad conseguidas en el multiplicador escalable Montgomery, debido entre otras razones, a la utilización de la aritmética CSA *radix-2* y *radix-4*, en una FPGA Virtex II de Xilinx.

En la multiplicación con constantes Gutiérrez y otros en [46] proponen la utilización de aritmética *carry-save* en multiplicaciones con constantes multiplexadas en

el tiempo implementadas en FPGAs. Definen unas celdas básicas para las operaciones a bajo nivel. En una de ellas implementan un sumador/restador de tres entradas, con la posibilidad de bypass de una o dos entradas o poner las salidas cero. La celda está basada en un compresor [3:2] con alguna pequeña modificación y se implementa en una única LUT6. La desventaja de utilizar este tipo de celda no estándar es que para obtener compresores mayores, los tienen que realizar componiéndolos con esta celda básica, por lo que se multiplican los recursos lógicos de control necesarios. Sin embargo, destacan la mejora del rendimiento de hasta un 50% y la reducción de área en las aplicaciones de prueba, frente a la utilización de los sumadores CPA.

En [57] Verma y otros presentan un nuevo método para sintetizar clusters que realizan operaciones de suma en punto flotante, utilizando árboles de compresores. Su método reduce el tiempo de los caminos críticos en un 20 % y el consumo de área en un 29% al sintetizarlos en una FPGA Stratix III de Altera [36].

2.3. Mapeo eficiente de sumadores carry-save

Varios trabajos han demostrado que se puede realizar una implementación eficiente de compresores aislados de distinto tipo. En [50] Beuchat y Muller proponen la utilización de un *radix* elevado para la representación *carry-save*, convirtiendo las sumas, en sumas de unos cuantos dígitos, consiguen así acomodar estas sumas para que utilicen completamente el *slice* para LUT4, consiguiendo de esta forma un mapeo eficiente. Se estudia el mapeo de compresores aislados [3:2] a bajo nivel proponiendo una implementación eficiente. Sin embargo, esta representación no convencional con un *radix* superior tiene importantes limitaciones, por ejemplo, en la representación que utilizan no esta permitido el desplazamiento y por otro lado la división de las sumas se tiene que acomodar a cada FPGA en particular, variando entre FPGAs con LUT4 y LUT6.

En [58] se estudia la implementación de un sumador *signed-digit* de *radix-4* basado en FPGAs con LUT6, pero debido a que no utilizan las cadenas de acarreo el consumo de área es muy elevado, del orden de un 88% de consumo extra de recursos en LUT6 frente a la representación en complemento a dos y sumadores CPA. El consumo de área aumenta aún más en LUT4, siendo 2,58 veces el consumo del mismo sumador en representación de complemento a dos sintetizado con sumadores CPA.

En [44] y [59] se realiza el diseño a bajo nivel utilizando primitivas para FPGAs con LUT4 de sumadores redundantes que utilizan todos los recursos del slice, incluidos la lógica de propagación de acarreo. En ambos estudios se crean compresores [3:2] y [4:2] que pueden ser utilizados como una librería básica para la creación de árboles de compresores, con un alto rendimiento y sin que se produzca ningún desperdicio de recursos. Parte de las aportaciones de este trabajo se encuentran resumidas y publicadas en [44].

Parandeh y otros en [49] y [45] proponen un diseño eficiente de árboles de compresores en FPGAs con LUT6. Estos trabajos se comentarán con detalle en el apartado 2.4. En [49] comentan que este trabajo demuestra que la creencia hasta ese momento de que no era posible implementar eficientemente árboles de compresores en FPGAs con LUT6 era errónea.

En [60] los autores han comprobado experimentalmente en el caso de suma de multioperandos, que los compresores CSA especialmente los compresores [6:3] aún en anchura de operandos pequeña (16 bits), ofrecen una mejora de velocidad de 7,9 % aunque a costa de un aumento de área de 28 %, para un total de 6 operandos. Según sus resultados es el número óptimo de operandos al realizar las pruebas en FPGAs con LUT6.

2.4. Descripción y síntesis desde lenguajes de alto nivel de aritmética CS: suma multioperando

Independientemente de la síntesis de los componentes básicos CSA, está la problemática de cómo conseguir que los sintetizadores a partir de la descripción a alto nivel generen sumadores CSA, en lugar de sumadores CPA. Por ejemplo, si desde el lenguaje de alto nivel VHDL se quieren sumar cuatro operandos $A+B+C+D$, el sintetizador lo implementará como sumas CPA, ¿Cómo conseguir que la suma se realice mediante un árbol de compresores?

En ASIC la suma de multioperandos se ha realizado tradicionalmente utilizando aritmética CSA mediante la creación de árboles de compresores siendo los compresores [3:2] y [4:2] los más utilizados [61]. Por ejemplo, un compresor [4:2] al que se le añade una etapa final de suma convencional, puede sumar 4 operandos de n bits con un retardo que será la suma del retardo del compresor [4:2] (independiente del número de bits) más el retardo del sumador convencional. La otra opción sería utilizar dos

sumadores binarios que producen una salida cada uno y sumarlas en un tercer sumador. El retardo en este caso será dos veces el retardo del sumador convencional que en este caso depende del número de bits. En FPGAs se considera que es mejor utilizar las cadenas de propagación de acarreo o los bloques DSP antes que construir árboles de compresores utilizando las LUT. Esto es cierto siempre que el único criterio, a la hora de la síntesis, sea ocupar la menor área posible.

Como ya se ha comentado, la utilización de árboles compresores ha demostrado los beneficios conseguidos en numerosas aplicaciones, especialmente las que se han comentado en el punto 2.2. Las técnicas para construir los árboles de compresores fueron propuestas por Wallace [30] y Dadda [31] en los años sesenta. Posteriormente en el contexto de los multiplicadores paralelos, suponiendo que los bits a la entrada de los compresores no se tenían en el mismo instante, Stelling y otros [62] describieron un algoritmo óptimo llamado *3-greedy* (3GD) para la construcción de árboles de compresores. Um y Kim en [63] describen un método para intentar minimizar la distancia entre CSAs teniendo en cuenta el layout producido en la síntesis. Tanto [62] como [63] utilizaron bloques básicos [2:2] y [3:2] para la construcción de los árboles. Estas técnicas descritas son adecuadas para diseño de circuitos ASICs pero no para FPGAs.

Dos líneas de trabajo se han seguido para la síntesis de árboles de compresores en FPGAs. Una línea es la que describe Hormigo y otros en [64], muy próxima a la línea de este trabajo que se presenta, y está basada en la eficiente implementación de compresores *carry-save* genéricos en FPGAs. La otra línea de construcción automatizada de árboles de compresores, se basa en la utilización de los llamados "Generalized Parallel Counters (GPC)" [65][66]. La suma multioperando basada en GPC ha sido estudiada para LUT6 por Parandeh y otros en [45], [49], [67], [68], [69] y por Matsunaga y otros en [70], pero ninguno de los métodos propuestos son válidos para FPGAs con LUT4.

En [64] la síntesis de suma multioperando está basada en árboles de compresores *carry-save* genéricos, independiente del número de bits de los operandos, sin un consumo en área excesivo, comparado con los árboles CPA y con una optimización de los caminos críticos. Además de la clásica estructura de árboles CSA, se presenta una novedosa estructura de array lineales que aprovecha las rápidas cadenas de acarreo. En

el caso de gran número de operandos y para tamaños de 16 bits, se alcanzan velocidades de 2,14 a 2,29 veces comparados con los árboles binarios y ternarios basados en los sumadores CPA. Y en el caso de un ancho de 64 bits se alcanzan velocidades de entre 3,11 a 3,81 veces. El método propuesto es un código parametrizable descrito en VHDL que utiliza los sumadores CPA y válido tanto para FPGAs con LUT4 como para FPGAs con LUT6.

Un contador paralelo o también llamado contador de columna, es un circuito que toma *m* bits de entrada, cuenta el número de bits que están a uno y genera un valor entero sin signo a la salida de *n* bits. El rango de salida será [0,m]. Verma e Ienne en [71], utilizando una formulación "Integer linear programming" describen el diseño de árboles de compresores que utiliza una librería de contadores *m:n* para $2 \leq m \leq 8$.

Los GPC son una extensión de los contadores paralelos que pueden sumar bits de entrada con distinto peso a diferencia de los contadores paralelos donde todas las entradas tienen el mismo peso. Por ejemplo un (2,3:3) puede sumar 2 bits de peso 1 y 3 de peso 0, por lo que como máximo a la salida se tendrá $2x2^1+3x2^0 = 7$, de ahí que sean necesarios 3 bits para representar la salida. Hay dos diferencias importantes entre los contadores y los compresores: la primera es que la salida de los compresores es redundante y los contadores no y la segunda es que los compresores tiene acarreos de entrada y salida para encadenar y los contadores no.

Hecha esta pequeña descripción de los GPC, Parandeh y otros en [49] proponen un nuevo método heurístico de síntesis de árboles de compresores basados en GPC que se sintetizan en LUT6. Obtienen un menor retardo en el camino más critico, pero debido a que no utilizan las cadenas de acarreo y al método heurístico que utilizan, los árboles de compresores consumen muchas más celdas lógicas que los árboles basados en sumadores ternarios. En [68] mejoran el problema del mapeo desarrollando una nueva solución basada en "Integer linear programming" obteniendo, según los autores, una mejora de un 32% en retardo y una mejora de un 3% en área respecto a los árboles de sumadores.

En [45] Parandeh y otros proponen utilizar la cadena de acarreo propagado en la síntesis de GPCs para mejorar el consumo de área. Este nuevo método contempla dos aspectos, por un lado la síntesis basada en su método heurístico para crear la red de GPCs, y por otro lado la utilización de un modelado atómico de los GPCs a bajo nivel,

que utilice eficientemente los recursos lógicos de las FPGAs, creando una librería de componentes. Argumentan que el modelo a bajo nivel de los GPCs es necesario para tener un buen rendimiento, ya que las herramientas de alto nivel no son capaces de utilizar los recursos lógicos eficientemente. Contemplando estos dos aspectos especialmente el diseño a bajo nivel obtienen una mejora en área de un 20% con sus técnicas anteriores y el mismo retardo. Respecto a los árboles de sumadores mejoran un 23 % el retardo del camino crítico con un consumo de área superior en 11% solamente. Este fue uno de los argumentos por los que en este trabajo se ha comenzado por estudiar la implementación de compresores aislados a bajo nivel para poder crear unas librerías de componentes.

Matsunaga y otros en [70] proponen la generación de árboles de compresores basada en la utilización de GPC. Parten de la idea de que contadores con un número de entradas similar al número de entradas de las LUTs pueden tener un coste similar en área al de un contador [3:2]. El rendimiento en FPGAs de los árboles de compresores no sólo depende del número de niveles de los compresores, sino también del número de compresores utilizados, no utilizan ningún método heurístico como los métodos de Parandeh y otros, y comentan que su método, que pretende reducir el tiempo de ejecución en casos prácticos del algoritmo, está basado en el método de Dadda poniendo limite en los pesos intermedios del proceso de compresión eliminando GPC innecesarios en la red de GPCs.

2.5. Propuestas de modificación del hardware de las FPGAs

Actualmente los bloques DSP contienen multiplicadores de un ancho de bit fijo que se puede utilizar para generar multiplicadores mayores. Los multiplicadores integrados en estos bloques utilizan árboles de compresores para la suma final de las sumas parciales. Estos compresores no están disponibles para el usuario final. En [72] y [73] se propone la creación de un módulo nuevo para FPGAs llamado "Field Programmable Counters Array" (FPCA), un acelerador para aritmética *carry-save* que permita la suma multioperando. Integra contadores *m*:*n* conectados en una red de rutado programable.

En [69] Parandeh y otros proponen mejorar estos aceleradores FPCA utilizando GPC, que permitan crear árboles de compresores configurables por el usuario, y en [74] proponen otra arquitectura llamada "Field Programmable Compressor Tree (FPCT)".

Tanto las arquitecturas FPCA como FPCT [73][75], están pensadas para suma de multioperandos exclusivamente. La arquitectura FPCA consiste en un conjunto de bloques lógicos que permiten la configuración de GPCs. Dichos bloques se conectan a través de una red local programable de rutado, mientras que en la arquitectura FPCT los bloques básicos llamados "Compressor Slices (CSlices)" están conectados a través de cadenas de acarreo implementadas "de fábrica". Cada bloque *CSlices* contiene un contador [31:5] configurable que permite implementar una amplia variedad de GPCs. Aunque la arquitectura FPCT es menos flexible que la FPCA, debido a que los bloques tienen una conexión prefijada, es mucho más eficiente que la FPCA.

En [76] Seyed y otros proponen además una herramienta para hacer una exploración del espacio ocupado por el árbol de compresores programables orientado a los fabricantes de FPGAs, llamada "Design Space exploration" (DSE), con el objetivo de adaptar sus FPGAs a grandes clientes.

3. Implementación eficiente de sumadores *carry-save* en FPGAs

Breve resumen

En este capítulo demostraremos, como a pesar de lo que se creía hasta hace pocos años, es posible una implementación eficiente de compresores en FPGAs. Esta creencia provenía del hecho de que se habían diseñado los compresores desde alto nivel, dejando la responsabilidad a la herramienta de síntesis. Desgraciadamente estas herramientas de síntesis no mapeaban estos compresores adecuadamente en los *slices* de las FPGAs de manera automática especialmente en FPGAs con LUT4.

3.1. Introducción

Como se ha comentado en el capitulo 1, la aritmética redundante fue desechada hasta hace algunos años para su utilización en FPGAs, sobre todo en FPGAs basadas en LUT4, a pesar de que se había utilizado frecuentemente en ASIC. En las nuevas FPGAs basadas en LUT6 se comienza a utilizar CSA, por la facilidad de implementación de compresores [3:2]. Sin embargo, su uso extensivo no se ha conseguido; tampoco se han hecho exploraciones para aprovechar los recursos de estas nuevas FPGAs con LUT6. Principalmente se ha debido a dos causas: en primer lugar, cuando se describen sumadores genéricos u otras operaciones como multiplicación, que deben combinar sumas multioperando, los sintetizadores utilizan como operador básico de suma, los sumadores de acarreo propagado y no los compresores CS, y en segundo lugar, los sintetizadores no mapean eficientemente los compresores en el *slice* porque no los identifican adecuadamente. No existen operadores en los lenguajes de alto nivel que representen sumas en *carry-save*.

En las FPGAs basadas en LUT6 los sintetizadores realizan un mapeo mucho más eficiente de los compresores [3:2] si se definen específicamente como un circuito con dos salidas, pero aún así, cuando se utiliza aritmética redundante, mostraremos que es posible una mejora utilizando sumadores doble *carry-save*, frente a los sumadores clásicos *carry-save*.

Por otro lado, las FPGAs basadas en LUT4 siguen teniendo interés por dos razones principalmente. La primera es por su menor precio y consumo de potencia y la segunda, es porque existen numerosas placas en el mercado con distintos interfaces de elevado precio, cuya sustitución no se justifica solamente por el hecho de utilizar una FPGA de última generación. Por lo tanto, la implementación eficiente de compresores *carry-save* en FPGA LUT4 sigue teniendo interés.

3.2. Compresores en FPGAs con LUT4

3.2.1. Introducción

Las FPGAs con LUT4 de Xilinx incluyen lógica dedicada a la propagación de acarreo, lo cual permite la implementación de sumadores con acarreo propagado (CPA) eficientes. Más específicamente, el camino para la propagación del acarreo ha sido especialmente optimizado y junto con lógica de acarreo dedicada, suma y propaga el valor del acarreo rápidamente. Por esta razón, cuando el número de bits es pequeño, se prefieren los sumadores de acarreo propagado (CPA) frente a los sumadores *carry-save* (CSA). Sin embargo, cuando el número de bits es elevado el retardo de los sumadores CPA aumenta como se vio en el capitulo 1, y se hace conveniente una alternativa como los sumadores CS.

Respecto a la implementación de sumadores no redundantes, en [77] se hace un estudio comparativo en una FPGA Spartan 3E, entre los sumadores CPA y otros sumadores *carry-tree*, entre ellos el de acarreo anticipado. El estudio demuestra que hasta una anchura de 128 bits, el sumador CPA implementado usando la cadena de acarreo, tiene una mayor velocidad y ocupa menos área que el resto, y solamente a partir de 256 bits los sumadores *carry-tree* tienen una mayor velocidad.

En el resto del apartado 3.2 demostraremos que a partir de una determinada anchura de operando, los sumadores *carry-save* son más rápidos que los sumadores CPA en FPGAs con LUT4. También demostraremos cómo es posible implementar compresores [3:2] en una sola LUT4 y compresores [4:2] utilizando un único *slice* y sin desperdicio de recursos. Para ello realizaremos un estudio de los tiempos de propagación y de consumo de recursos de los sumadores CPA frente a los sumadores *carry-save* para determinar en qué caso se saca partido al uso de sumadores CS. Finalmente se mostrará una mejora en los tiempos de propagación de estos sumadores *carry-save* clásicos, sacando provecho al menor retardo de la entrada de acarreo del *slice*.

3.2.2. Estudio de los tiempos de propagación de sumadores CSA frente a CPA en FPGAs de bajo coste basadas en LUT4.

En primer lugar se realizará un estudio de los tiempos de propagación de sumadores de acarreo propagado en FPGAs basadas en LUT4, para ver en qué medida influye el número de bits en el tiempo de propagación del sumador. La Tabla 1 muestra los tiempos de propagación en nanosegundos de sumadores de acarreo propagado de distinto número de bits para la FPGA de bajo coste XC3S200-4ft256 de Xilinx [78] obtenidos con la herramienta ModelSim. Los tiempos que se muestran se han tomado a la entrada y salida del slice y registrando las salidas y entradas. Como cabía esperar, un sumador CPA de 32 bits tiene un retardo superior a un sumador CPA de 2 bits, del orden de tres veces. La Tabla 1 muestra también el esfuerzo del fabricante en optimizar la cadena de acarreo puesto que el tiempo de un sumador de 4 bits no es el doble de uno de 2 bits, y el retardo de un sumador de 32 bits es poco más del triple de uno de 4 bits.

Sumador	2 bits	4 bits	8 bits	16 bits	32 bits	64 bits	128 bits	256 bits	512 bits
CPA (ns)	2,375	2,665	3,180	3,874	6,186	6,738	14,288	25,670	53,204
CSA [3:2] (ns)	2	2,23	2,23	2,31	2,5	2,68	2,92	3,1	3,4

Tabla 1. Tiempos de propagación para sumadores CPA y CSA para la FPGA XC3S200-4ft256.

En el caso de sumadores CSA desde 2 bits hasta 512 bits, el tiempo de propagación máximo obtenido, varía desde 2 a 3,4 ns., dependiendo del mapeo concreto en la FPGA. Los sumadores CSA tienen la ventaja de mantener prácticamente constante los tiempos de propagación independientemente del número de bits de los operandos, ya que no es necesaria la propagación del acarreo. Para tamaños de operandos pequeños de hasta 8 bits, la utilización de sumadores CSA no está justificada puesto que los retardos son similares a los de los CPA, siendo una de las razones por lo que no se han utilizado los sumadores CSA para anchos de palabra pequeños. Además, hay que tener en cuenta que para obtener el resultado final, cuando se utilizan sumadores CSA, es necesario realizar una suma convencional del acarreo (C) y suma (S). Aunque se trabaje con un tamaño de bits de operando elevados, los sumadores CSA resultarán útiles siempre que se realicen sumas intermedias en *carry-save*, y este menor retardo no se consuma en la última suma convencional.

En conclusión, los sumadores CSA ofrecen una mayor velocidad en aquellas aplicaciones en las que se realicen sumas intermedias con un número de bits elevado, y

en aquellos en los que el número de sumas que se puedan realizar con aritmética redundante sea elevado, o también en el caso de suma de multioperandos. Una vez estudiado los sumadores CSA versus CPA teniendo en cuenta la velocidad, en el que claramente resulta ventajoso los sumadores CSA, cabe plantearse el consumo de recursos en esta FPGA de bajo coste de Xilinx.

3.2.3. Estudio de utilización de recursos de sumadores CSA frente a CPA en FPGAs de bajo coste basadas en LUT4

Como se ha comentado en la introducción, la aritmética redundante utiliza doble número de bits que la aritmética convencional para representar un número, y además para tener el número en representación convencional se debe realizar una suma final. Por tanto, no se puede esperar un menor consumo de recursos, pero si se consiguen utilizar los mismos recursos para un sumador CPA de 1 bit que para uno en CSA, en el caso de radix-2, se habrá conseguido una implementación eficiente y por tanto este será el objetivo de este apartado. Se ha de tener en cuenta que en las FPGAs debe tenerse disponible la suma y el acarreo como dos salidas accesibles para ser rutadas a donde se desee.

En trabajos anteriores como [79], los autores han descartado la utilización de la aritmética redundante en FPGAs con LUT4 porque se producía un desperdicio de recursos elevados al sintetizar compresores [3:2] y/o [4:2], como demuestra el hecho de que son muy pocos los trabajos de aritmética redundante en LUT4. Sin embargo, en [44] demostramos que esta creencia era errónea.

Esta creencia se debía a que cuando se describe un compresor [3:2] desde alto nivel, se sintetiza en dos LUT4s porque al definir dos salidas, una para la suma y otra para el acarreo, y al tener las LUT4 una única salida, el sintetizador utiliza dos LUT4. Además, aunque algunos autores estudian que es posible una implementación de un compresor [3:3] utilizando una LUT4 y la entrada de acarreo, queda otra LUT4 en el *slice* que no pueden utilizar para sus aplicaciones. Es decir, por un lado para la implementación de un compresor [3:2] se utilizan más recursos de los que son necesarios al describirlo desde alto nivel, y por otro lado quedan recursos en el *slice* que no son utilizados por los sintetizadores en las aplicaciones que presentan los autores en [50]. Veamos cómo se pueden sintetizar sumadores de distinto tipo en un *slice* de una

FPGA con LUT4 y demostraremos cómo podemos implementar eficientemente compresores.

La Figura 9 describe la arquitectura simplificada de un *slice* para una FPGA Spartan 3 de Xilinx [41]. Cada *slice* contiene dos *look-up table* de cuatro entradas (G-LUT y F-LUT), dos *flip-flops* (FFY y FFX), lógica de acarreo (CYSELG, CYMUXG, CYSELF, CYMUXF y CYINIT), puertas lógicas (GAND, FAND, XORG y XORF), y multiplexores con varias funciones.

Recuérdese que un sumador completo (FA) es un circuito con tres bits de entrada (los bits a sumar x_i e y_i, y el acarreo de entrada c_{in}) y dos de salida (el bit de suma s_i y el acarreo de salida c_{out}). Tenemos que $s_i = x_i \oplus y_i \oplus c_{in}$, y el acarreo de salida $c_{out} = x_i$, si $x_i = y_i$, y $c_{out} = c_{in}$ en cualquier otro caso. Supongamos que F-LUT calcula $x_i \oplus y_i$, entonces la puerta XORF obtiene el bit de suma s_i, mientras que el cálculo del acarreo de salida c_{out} involucra a tres multiplexores (CYOF, CYSELF y CYMUX). El bit de suma s_i se puede propagar a otro *slice* (salida X) o se puede almacenar en el *flip-flop* FFX. El bit de acarreo c_{out} se puede propagar o tenerlo disponible en la salida XB.

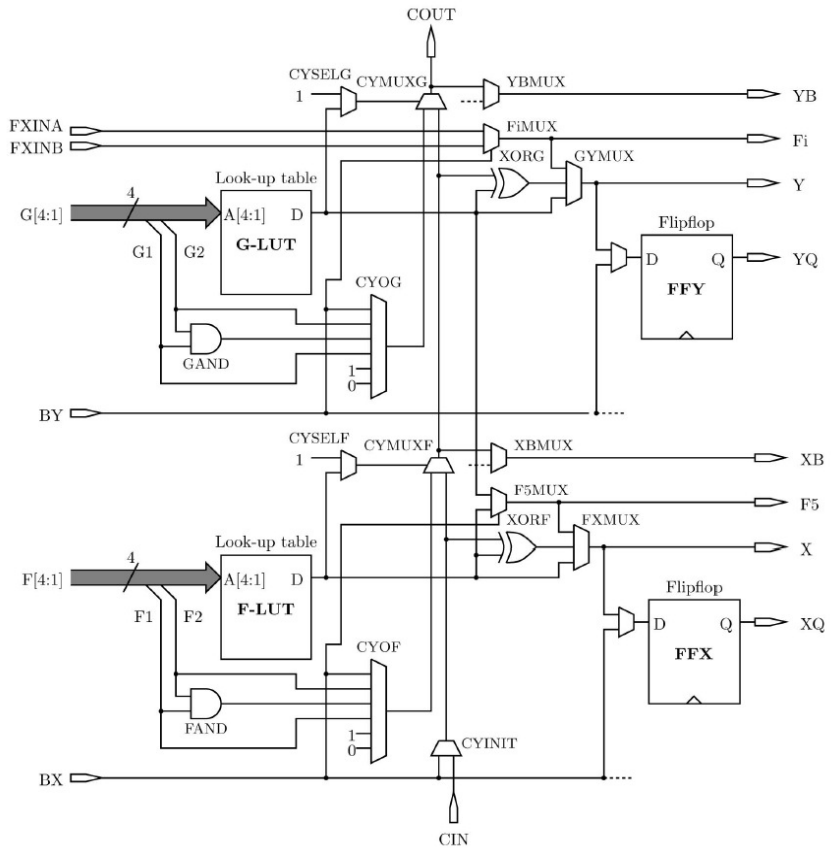

Figura 9. Diagrama simplificado de una FPGA Spartan-3.

Si consideramos también la G-LUT se podría implementar un segundo sumador, integrando así a un sumador CPA de 2 bits en el mismo *slice*. Sin embargo, como se estudia en trabajos de algunos autores como [50] o deducir de la Figura 9, es imposible implementar dos sumadores completos con acarreos de entrada independientes en el mismo *slice*, ya que cada *slice* dispone solamente de una sola entrada de acarreo, con lo que se requeriría el doble de s*lices*. Por este motivo los autores de [50] proponen utilizar sumadores CSA de un *radix* superior para aprovechar completamente el *slice* cuando se utilice una descripción VHDL.

La Figura 10 describe la arquitectura simplificada de un *slice* que implementa un sumador con acarreo propagado (CPA) de dos bits o un CSA de *radix-4* sintetizado a partir de una descripción VHDL.

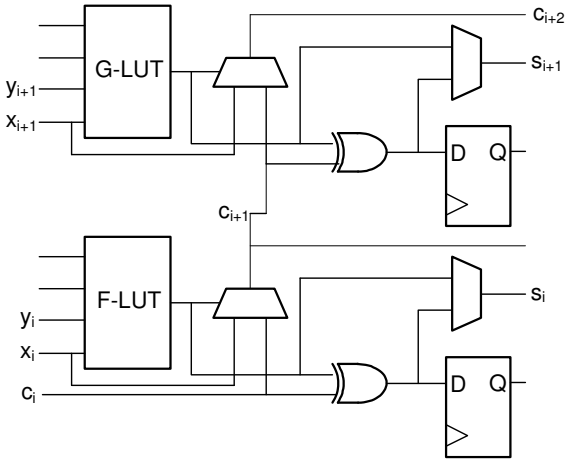

Figura 10. Implementación de un sumador CPA de dos bits o CSA de *radix-4*.

Sin embargo, es cierto que desde una descripción VHDL de un sumador CSA se consumen dos LUT4s (*slice* completo), es decir, el sintetizador utiliza una LUT4 para C y otra LUT4 para S. Observando detenidamente el slice se puede utilizar la LUT4 inferior (F-LUT) y la entrada de acarreo para implementar un CSA de 1 bit en una sola LUT4. En [44] mostramos como utilizando primitivas se puede construir un sumador CSA de 1 bit utilizando la F-LUT como se puede observar en la Figura 11.

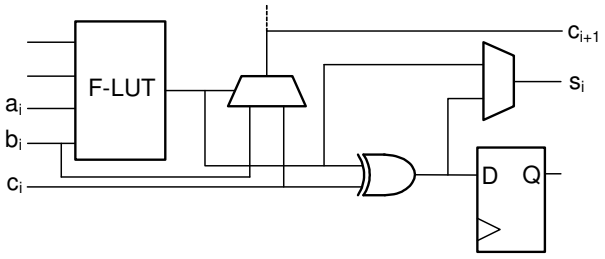

Figura 11. Implementación eficiente de un sumador CSA de 1 bit

Por tanto, con una descripción VDHL adecuada utilizando primitivas se puede realizar un sumador CSA de 1 bit o un compresor [3:2] de 1 bit mapeado en una sola LUT4. La G-LUT queda libre para ser utilizada en la síntesis de otras funciones lógicas como ha quedado claro en nuestros trabajos como [44] y [51]. Y aunque autores como [50] indicaban, sin argumentar, que en sus aplicaciones no se aprovechaba la G-LUT, siendo ésta unas de las razones por las que proponían utilizar *radix* superiores, en general esto no es cierto. Lo que si es cierto, es que no es posible utilizar la G-LUT para implementar otro sumador CSA de 1 bit, porque no están disponibles las tres entradas necesarias.

Por tanto, queda comprobado que un CSA de 1 bit se puede implementar utilizando una única LUT4. A modo de ejemplo, la Tabla 2 muestra los resultados obtenidos con la herramienta ISE de Xilinx para la FPGA XC3S200-4ft256 para un sumador *carry-save* de 8 bits.

8 bits CSA	LUT	*SLICE*
Descripción VHDL sin primitivas	16	8
Descripción VHDL y primitivas	8	8

Tabla 2. Ocupación de área de un sumador CSA de 8 bits.

En el mismo trabajo [44] indicamos cómo es posible realizar un compresor [4:2] optimizado que aprovecha completamente todos los recursos de un *slice* como muestra la Figura 12 y la Figura 13. En la Figura 12 se muestra en gris los dos compresores [3:2] que se mapean en el mismo *slice*. Para facilitar la comprobación, téngase en cuenta que todas las sumas y acarreos distintos de $S_{out\_}$ y $C_{out\_}$ son sumas y acarreos intermedios que corresponden a los nombres utilizados en la Figura 7 o en la Figura 12.

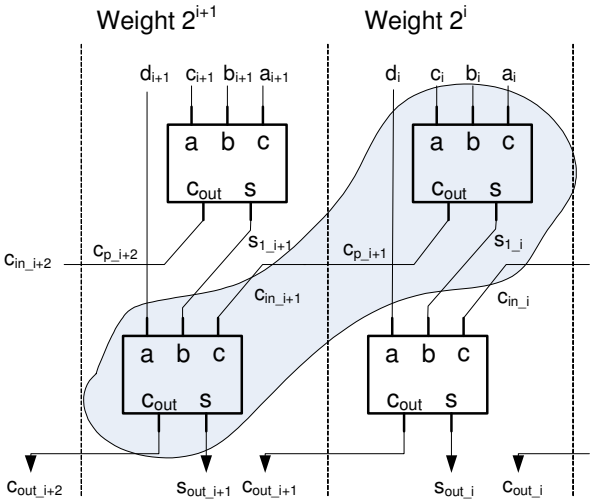

Figura 12. Mapeo de compresores [3:2] en un *slice* para tener un compresor [4:2].

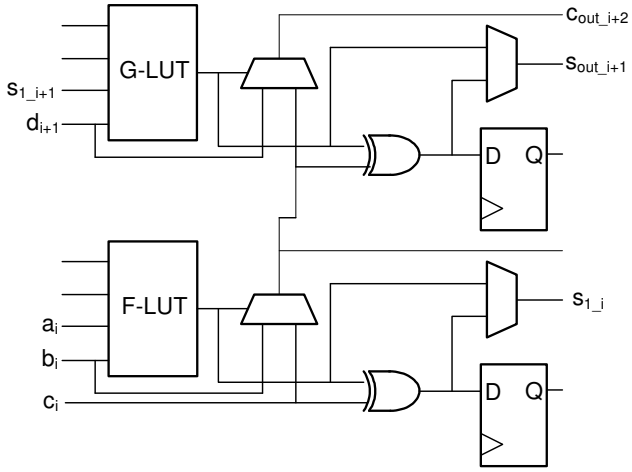

Figura 13. Implementación de un compresor [4:2] en un *slice*.

Otro aspecto relativo a la aritmética CSA es que para tener un número en su representación convencional es necesaria una suma final de C y S. Aparentemente pudiera parecer que siempre se tendrá un consumo de recursos extra para este sumador, pero en muchos casos, como el que mostramos en [51], es posible obtener una implementación optimizada de un sumador combinado CSA-CPA para algoritmos que utilizan suma CSA y que finalmente ofrecen un resultado final convencional. En nuestro trabajo [51] se puede ver cómo el sintetizador aprovecha el *slice* completamente, y en la F-LUT se tiene una implementación que unas veces trabaja como sumador de 1 bit CSA y en otro caso como sumador CPA de 1 bit para la suma final.

Por último, cabe decir que a partir de este compresor [3:2] y/o [4:2] es posible construir compresores de orden mayor [5:2], [7:2], etc. En el apartado siguiente mostraremos cómo es posible una optimización de estos compresores de orden mayor

construidos a partir de un compresor [3:2], teniendo en cuenta que la entrada de acarreo del *slice* proporciona un camino más rápido que las otras dos entradas que llegan a las LUT. Este resultado se ha aprovechado en otros trabajos como en [64], que muestran cómo es posible generar de manera automática desde VHDL compresores de alto orden optimizados en velocidad.

3.2.4. Optimización de compresores CSA aprovechando la entrada de acarreo propagado

En el apartado anterior se ha mostrado que utilizando primitivas es posible implementar compresores CSA en FPGA con LUT4 sin que se produzca un consumo excesivo de LUT. A partir de este compresor [3:2] o [4:2] es posible optimizar en velocidad compresores mayores aprovechando que la lógica de acarreo es más rápida que el resto.

La Figura 14 y Figura 16 muestran la implementación habitual de compresores según [16], en los que todas las entradas se consideran por igual, sin tener en cuenta que algunas de ellas serían entradas de acarreo. En el caso del compresor [5:2] (Figura 14), el sumador completo de salida suma dos entradas de acarreo junto con la suma de los dos sumadores de los niveles superiores. Esta implementación clásica produce retardos de propagación mayores que la versión optimizada, que se mostrará a continuación, cuando se mapea el compresor en una FPGA, porque no considera que dos de las entradas del sumador completo pasan por caminos más lentos que la entrada que discurre por la línea de acarreo del *slice*. La solución reside en aprovechar los recursos de propagación de acarreo de la FPGA para los caminos más críticos. Modificando las conexiones del compresor clásico se pueden optimizar los compresores.

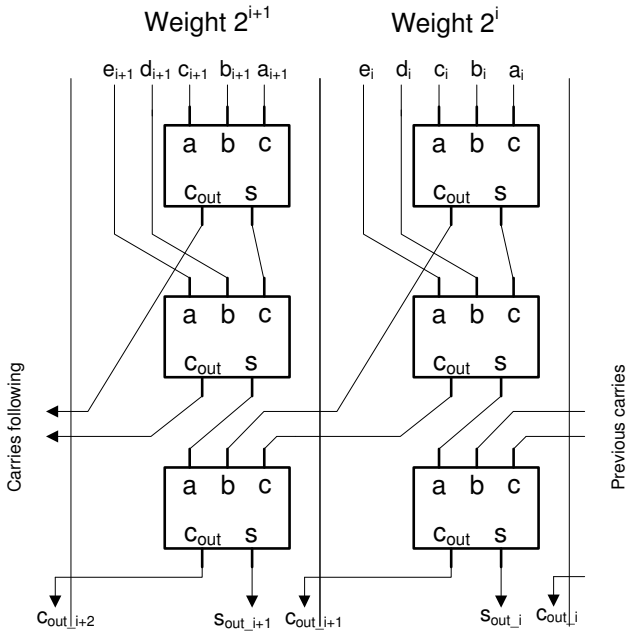

Figura 14. Implementación clásica de compresores [5:2].

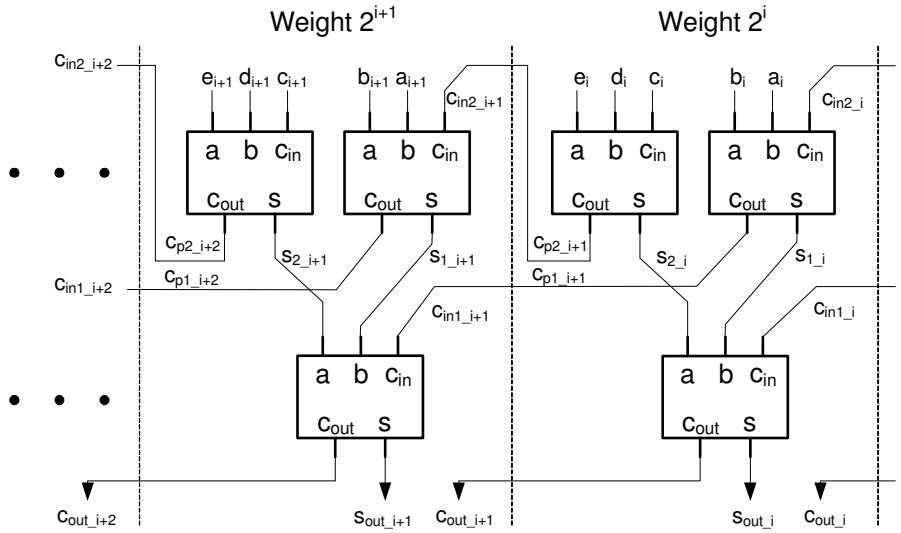

Figura 15. Compresor [5:2] optimizado para FPGA con LUT4.

La Figura 15 muestra una implementación optimizada para la FPGA Spartan 3, donde se ha tenido en cuenta los distintos tiempos de retardo en el *slice*. En el bloque que representa al sumador completo se ha distinguido la entrada de acarreo de las otras dos entradas. Dicha entrada corresponde con la entrada de acarreo de cada *slice*, y por tanto el camino más rápido. La clave está en conseguir que los acarreos se propaguen a través de las líneas de propagación de acarreo de los *slices* y que el resto de entradas del sumador completo estén ya estables desde el momento inicial. Esto es lo habitual, ya que los bits de cada número se propagan en paralelo. Es por ello, por lo que los acarreos propagados siempre se conectan a las entradas de acarreo del sumador completo.

Obsérvese en la Figura 15 el acarreo propagado C_{p2_i+1} se conecta a la entrada C_{in2_i+1}. Este es el camino más largo, ya que se deben propagar los resultados por dos sumadores completos para obtener la suma y acarreo final. Por tanto, este acarreo se ha conducido por la línea de acarreo propagado de los *slices* usados. El acarreo propagado C_{p1_i+1} solamente atraviesa un sumador completo y se conecta a la línea de entrada de acarreo del *slice*. Se consigue así que la propagación de acarreo se realice por las líneas más rápidas.

La Figura 17 muestra la implementación optimizada de un compresor [7:2] teniendo en cuenta las mismas consideraciones en el diseño que para el compresor de [5:2] descrito en el párrafo anterior. Se han modificado las conexiones del compresor clásico pero se utilizan el mismo número de sumadores completos y los mismos niveles por lo que el beneficio se obtiene solamente por el aprovechamiento de la línea de acarreo para los caminos más largos.

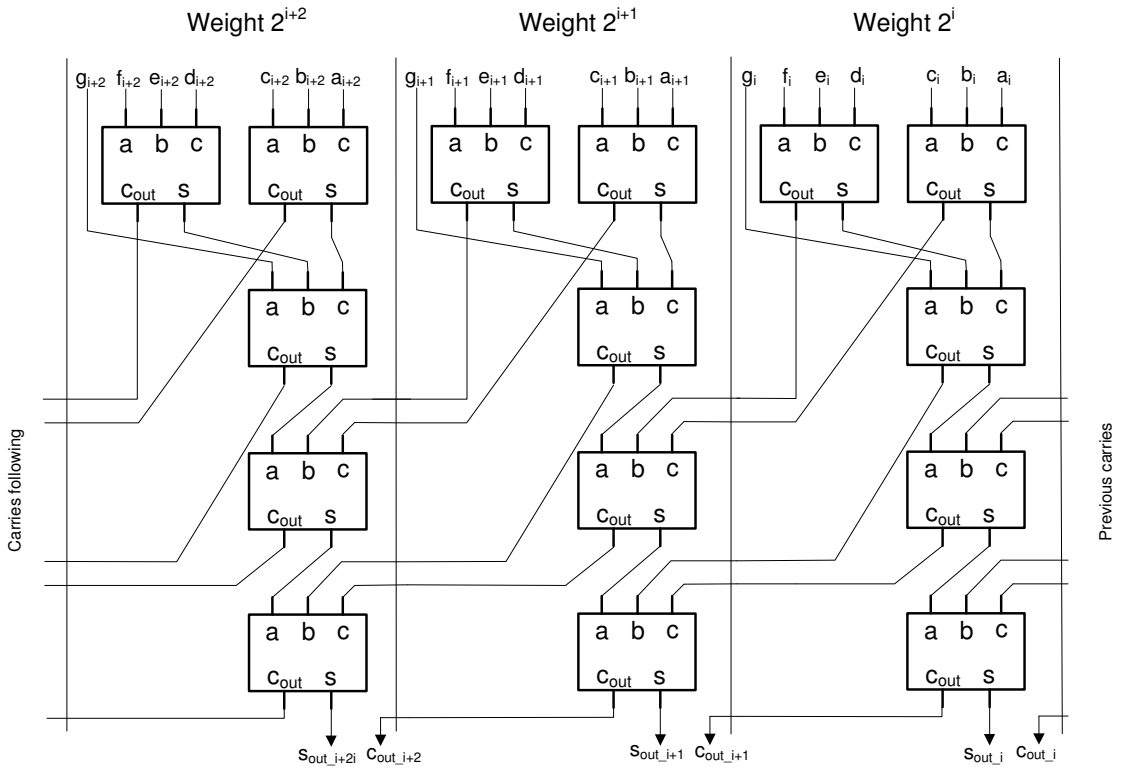

Figura 16. Implementación clásica de compresores [7:2].

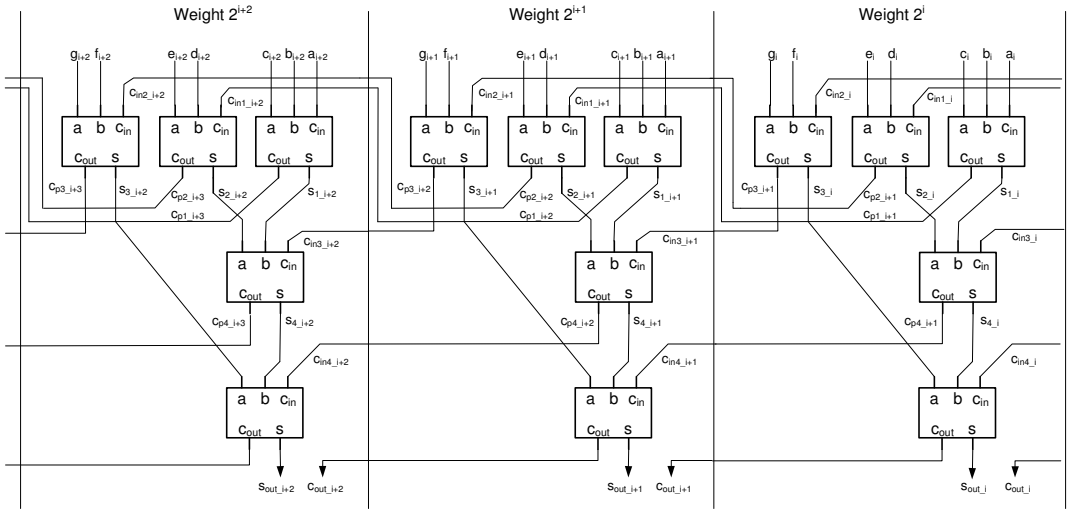

Figura 17. Compresor [7:2] optimizado para FPGA con LUT4.

La Tabla 3 muestra los resultados de la implementación de los compresores con un diseño clásico y los resultados de la implementación optimizada teniendo en cuenta los retardos dentro del *slice* para una FPGA de Xilinx Spartan 3 XC3S-1600E y velocidad grado-5. Para conseguir una medida real de la frecuencia, en ambos casos, se han registrado las entradas y salidas, por lo que el número de *slices* y LUT algo es mayor que los estrictamente necesarios para implementar los sumadores CSA. Los sumadores CSA se han diseñado para palabras de 32 bits. Como se puede observar se obtienen mejoras del orden del 23% con el mismo consumo de LUT.

Compresores CSA	Implementación clásica			Implementación optimizada		
	Slice	LUT	Frec.(MHz)	*Slice*	LUT	Frec.(MHz)
[5:2]	66	96	153,67	66	96	187,52
[7:2]	147	160	113,89	147	160	139,81

Tabla 3. Implementación clásica de compresores versus implementación optimizada.

3.2.5. Análisis de características de compresores en LUT4

La Tabla 4 muestra un resumen de los resultados de implementación de distintos compresores en una Spartan 3. Para la comparativa se ha elegido las opciones optimizadas. El primer resultado es que no se desperdician recursos del *slice* y el segundo resultado, si se observa además como el Gráfico 1, es que al contrario que se creía por extrapolación a lo que ocurre en ASIC, la velocidad del compresor no depende del número de niveles de compresor [3:2] que sea necesario para construirlo, es decir, un compresor [4:2] no tiene doble retardo que un [3:2], ni un compresor [5:2] tiene triple retardo, ni tampoco el compresor [7:2] tiene un retardo cuatro veces mayor.

Por tanto, hay que construir los árboles utilizando los compresores de mayor tamaño posible.

Compresores CSA de 32 bits	Implementación optimizada para LUT4				
	Slice	LUT	*Delay* (ns)	*Delay* [*m*:2]/[3:2]	Nº de niveles teóricos de [3:2]
[3:2]	32	32	2,0	1	1
[4:2]	32	64	2,4	1,2	2
[5:2]	66	96	5,33	2,22	3
[7:2]	147	160	7,16	3,58	4

Tabla 4. Características de compresores de 32 bits *carry-save* para LUT4.

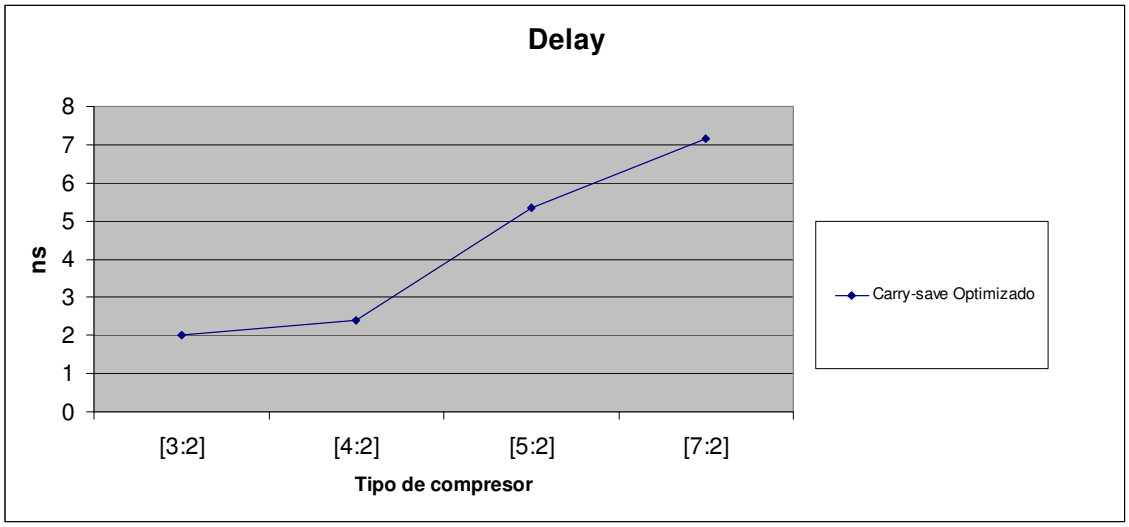

Gráfico 1. Retardo de compresores de 32 bits *carry-save* para LUT4.

3.3. Compresores en FPGAs con LUT6

3.3.1. Introducción

Las FPGAs Spartan 6 [43] y Virtex 6 [80] de la compañía Xilinx contienen LUT que pueden ser configuradas como LUT con seis entradas y una salida, o bien, como dos LUT con 5 entradas y una salida por LUT. La Figura 18 muestra una estructura simplificada de las LUT6 y parte del *slice*. Cada *slice* contiene cuatro LUT6 además de otros recursos.

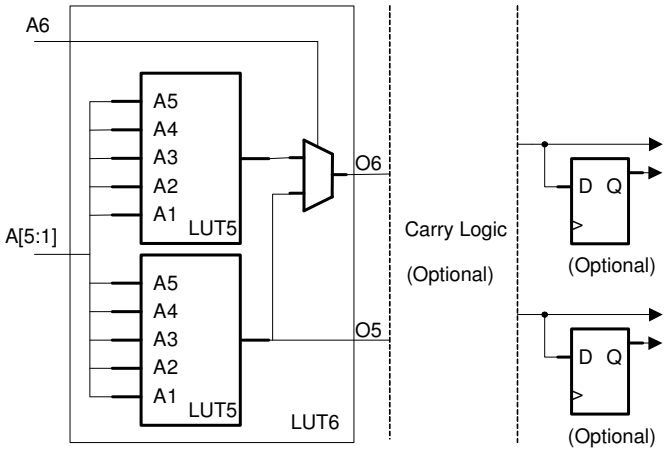

Figura 18. *Slice* simplificado de FPGAs con LUT6.

En el caso de una Spartan 6 existen tres tipos de *slices* que contienen diferentes recursos llamados *SliceX*, *SliceL* y *SliceM*. Los tres tipos de *slices* contienen como elementos comunes 4 LUT6 y *flip-flops* para almacenamiento. La Tabla 5 resume los recursos lógicos que contiene cada tipo de *slice*.

Características	SLICEX	SLICEL	SLICEM
LUTs de 6 entradas	√	√	√
8 *flips-flops*	√	√	√
Multiplexores (grandes)		√	√
Lógica de acarreo		√	√
RAM distribuida			√
Registros de desplazamiento			√

Tabla 5. Recursos contenidos en los *slices* según el tipo.

La FPGA Virtex 6 sólo contiene *slices* del tipo L y M. Como muestra la Tabla 5, los SliceX no contienen lógica de acarreo, sin embargo son el 50% de los recursos de una Spartan 6, y dado que el interés principal de este trabajo se centra en aprovechar al

máximo los recursos de las FPGAs de bajo coste se deben considerar estos *slices* en las implementaciones. A diferencia de las Spartan 3 con LUT4, se preferirán implementaciones que no utilicen la lógica de acarreo para las LUT6. Además como se mostrará más adelante, además de compresores *carry-save* con salida C y S presentados hasta ahora, se puede trabajar con compresores doble *carry-save* [*m*:3] altamente optimizados. Dada la gran cantidad de datos que se van a presentar en este capitulo se refundirán al final del mismo para poder obtener unas conclusiones finales.

3.3.2. Implementando un compresor [3:2] en LUT6

Las LUT6 disponen de dos salidas por lo que a diferencia de las LUT4 que solamente tienen una, el sintetizador mapea directamente un compresor [3:2] en una única LUT6 (Figura 19). No es necesario definirlo con primitivas como en el caso de LUT4, tampoco es necesario utilizar la lógica de acarreo, por lo que se puede implementar en cualquier tipo de *slices*, incluidos los SliceX. La frecuencia de trabajo para un compresor [3:2] es de 595,50 Mhz (retardo 1,68ns) para una FPGA Spartan 6 (referencia XC6SLX100-3FGG676).

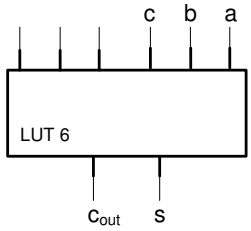

Figura 19. Compresor [3:2] sintetizado en una LUT6.

3.3.3. Implementando un compresor [4:2] en LUT6

Es difícil demostrar de una manera sencilla que no se puede implementar un compresor [4:2] en una sola LUT6 buscando el mapeo adecuado como se hizo con los compresores [3:2] en las LUT4. Sin embargo, se puede intuir observando detenidamente los *slices* que contienen a las LUT6 y el compresor [4:2] de la Figura 7, en especial la salida de propagación de acarreo al siguiente compresor, que se necesitan tres salidas, una para la suma S otra para el acarreo C y otra para el acarreo propagado. La salida para S y C es la misma que para el compresor [3:2], pero la salida para el acarreo propagado no se puede implementar al no disponer las LUT6 de tres salidas. Otra posibilidad sería utilizar la lógica de acarreo para implementar este acarreo que se

propaga, pero tampoco es posible porque se necesita una función lógica de las entradas que no se puede sintetizar en la cadena de acarreo.

A partir del compresor [3:2] se podría implementar cualquier otro compresor mayor como se ha hecho con las LUT4. Sin embargo, como se mostrará a continuación, es posible obtener distintas implementaciones en LUT6 de un compresor [4:2] y de orden mayor a partir de descripciones VHDL distintas. Además, es posible conseguir una versión con una velocidad mayor utilizando la lógica de acarreo de los *slices* L o M y utilizando primitivas.

Como ya se ha comentado, los compresores [3:2], se generan directamente utilizando la descripción VHDL. En este caso el sintetizador utiliza una LUT6 y genera dos salidas. A partir del compresor [3:2] se pueden generar compresores de orden superior. El compresor [4:2] se genera con dos sumadores completos como se mostró en la introducción. A la hora de realizar la descripción en VHDL del compresor [4:2] de la Figura 7, se utilizaron descripciones distintas y se obtuvieron dos síntesis diferentes. Además de estas dos síntesis, se puede realizar utilizando primitivas, otra síntesis que utilice la lógica de acarreo, para conseguir una mejora en velocidad. Por tanto, se tienen tres síntesis distintas del compresor [4:2] dependiendo de la descripción utilizada:

- Utilizando dos LUT6s por bit. Un compresor [3:2] en cada LUT6.
- Utilizando tres LUT6s por cada dos bits, es decir 1,5 LUT6s por bit.
- Utilizando dos LUT6s y la lógica de acarreo, si se disponen de *slices* L y M.

En la síntesis que consume dos LUT6s por bit, el sintetizador mapea un compresor [3:2] por LUT6 como muestra la Figura 20. Obsérvese cómo el sintetizador ha llevado las cuatro entradas a las dos LUT6, de esta manera la conexión S_{1_i} (Figura 7) no es necesaria, y por tanto se suprime el retardo asociado a este camino. El camino más crítico es el asociado a la señal C_{in} y atraviesa dos LUT6, la conexión se realiza dentro del *slice* y los retardos asociados son pequeños.

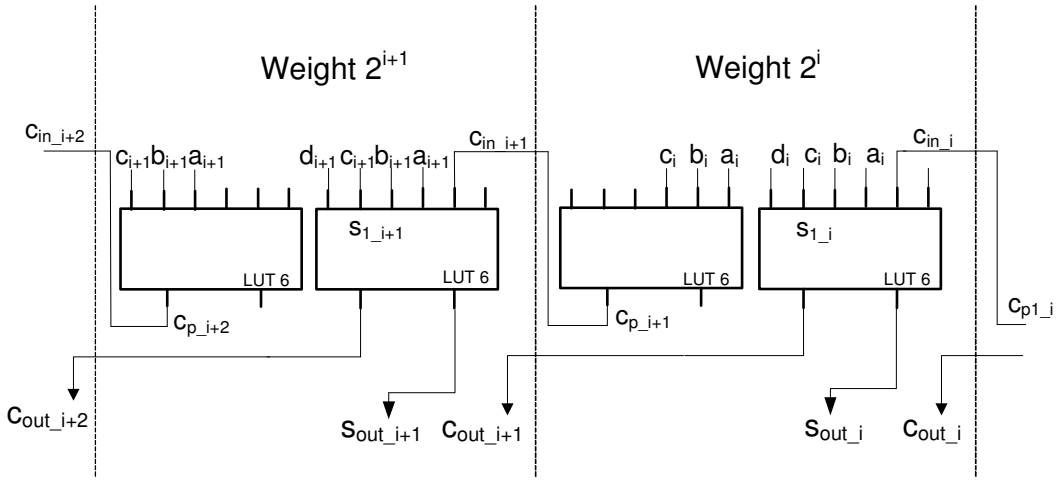

Figura 20. Compresor [4:2] sintetizado en 2 LUT6.

Los recursos utilizados para este compresor son 2 LUT6 por cada bit. En el caso de una Spartan 6 (referencia XC6SLX100-3FGG676) el retardo es de 2,585 ns. en la síntesis de un compresor de 32 bits. Téngase en cuenta que las entradas y salidas se han registrado para una mejor precisión en los datos de frecuencia.

Las otras dos síntesis que se van a mostrar suponen una optimización en área realizada por el propio sintetizador y una optimización en velocidad propuesta en este trabajo.

Optimización en área de compresores [4:2]

La Figura 21 muestra una optimización en área para compresores [4:2] de más de 1 bit de ancho de palabra. Esta optimización se puede conseguir a partir de la descripción VHDL y poniendo esfuerzo en el sintetizador en la optimización de área. La optimización realizada por el sintetizador se ha conseguido empaquetando los compresores de dos en dos bits, consiguiéndose mapear dos bits en 3 LUT6s en lugar de 4 LUT6s como en el caso anterior. Para una mejor comprensión de la síntesis véase también la Figura 7.

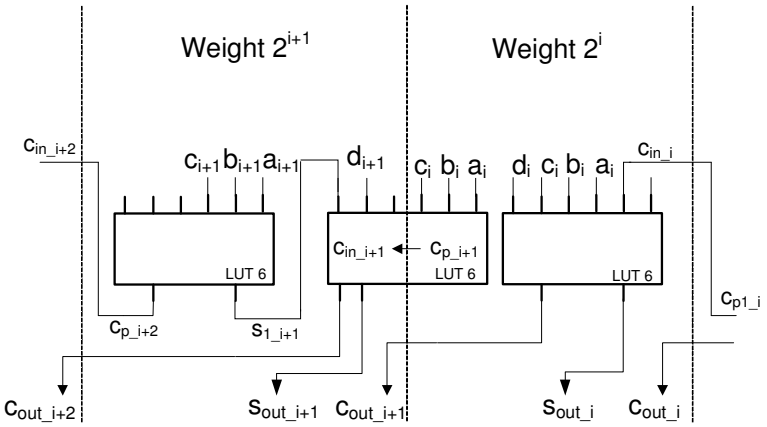

Figura 21. Compresor [4:2] optimizado en área.

Optimización en velocidad de compresores [4:2]

Si se observa la Figura 7 o la Figura 21, el camino más crítico (C_{p_i} a C_{out_i}) atraviesa dos FA y por tanto el retardo es al menos del doble del retardo asociado a una sola LUT. Si se disponen de *slices* con lógica de acarreo, se podría obtener un [4:2] utilizando dos LUT6 y aprovechando la lógica de acarreo, pero con un camino critico que solamente pase por una única LUT6 y la lógica de acarreo. En este caso la síntesis se debe realizar utilizando primitivas y generando las señales adecuadas. La Figura 22 muestra la síntesis propuesta, y las funciones lógicas que se han generado en las LUT6s. Para ello se ha estudiado las funciones lógicas que se deben generar para implementar el compresor [4:2] de acuerdo a los recursos disponibles en la lógica de acarreo. La Tabla 6 muestra las funciones lógicas generadas de acuerdo a los recursos y a las siguientes reglas:

- $MuxselC_p$ siempre vale 0.
- Acarreo almacenado temporal (C_{s_tmp}) vale 1 si las 4 entradas son 1. Se genera en este caso el acarreo propagado (C_p) y almacenado (C_s).
- Acarreo propagado (C_p) vale 1 si dos o más entradas están a 1. Este acarreo se genera exclusivamente a partir de los bits de los operandos, y como es el que se propaga se consigue cortar la cadena de propagación de acarreo.
- Acarreo almacenado (C_s), siempre tiene el valor de C_{in}, puesto que es la OR exclusiva de $MuxselC_p$ (siempre vale 0) con C_{in}.
- S_{tmp} es la OR exclusiva de los bits de los operandos.
- Suma final (S_{out}) es la OR exclusiva de S_{tmp} junto con el acarreo de entrada al compresor C_{in}.

d	c	b	a	C_p	C_{s_tmp}	S_{tmp}	Comentario
0	0	0	0	0	0	0	Se suma C_{in} a S_{tmp}
0	0	0	1	0	0	1	Se almacena C_{in}
0	0	1	0	0	0	1	Se almacena C_{in}
0	0	1	1	1	0	0	Se suma C_{in} a S_{tmp}
0	1	0	0	0	0	1	Se almacena C_{in}
0	1	0	1	1	0	0	Se suma C_{in} a S_{tmp}
0	1	1	0	1	0	0	Se suma C_{in} a S_{tmp}
0	1	1	1	1	0	1	Se almacena C_{in}
1	0	0	0	0	0	1	Se almacena C_{in}
1	0	0	1	1	0	0	Se suma C_{in} a S_{tmp}
1	0	1	0	1	0	0	Se suma C_{in} a S_{tmp}
1	0	1	1	1	0	1	Se almacena C_{in}
1	1	0	0	1	0	0	Se suma C_{in} a S_{tmp}
1	1	0	1	1	0	1	Se almacena C_{in}
1	1	1	0	1	0	1	Se almacena C_{in}
1	1	1	1	1	1	0	Se generan los dos acarreos

Tabla 6. Funciones lógicas utilizadas para la síntesis del compresor [4:2] optimizado en velocidad.

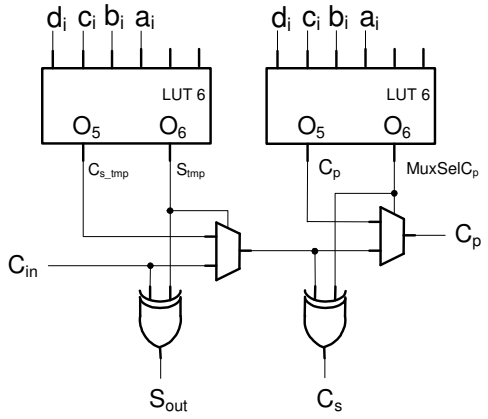

Figura 22. Compresor [4:2] optimizado en velocidad.

El circuito trabaja de la siguiente manera. Lo primero que se debe conseguir es cortar la cadena de acarreo por tanto el multiplexor de la derecha siempre selecciona la entrada Cp. Esto obliga a generar cuidadosamente el acarreo propagado como puede observarse en la Tabla 6. Al depender el acarreo propagado Cp únicamente de las entradas, la cadena de acarreo se corta. Con la LUT6 de la izquierda junto con el MUX y la XOR hay que almacenar la suma y el acarreo almacenado. La suma S_{out} se genera con S_{tmp}, que es la suma de las cuatro entradas, y el acarreo de entrada. Ahora sólo queda generar adecuadamente el acarreo almacenado teniendo en cuenta que el

multiplexor está controlado también por la señal S_{tmp}. El acarreo almacenado es la salida del multiplexor de la izquierda. Se han generado dos señales en la LUT6 de la izquierda. Una de ellas es la suma para controlar el multiplexor.

En este caso los recursos utilizados se muestran en la propia Figura 22 y el retardo es de 1,8 ns. El mayor inconveniente es que la lógica de acarreo del *slice* es indivisible ya que la primitiva opera sobre toda la lógica de acarreo del *slice*, además de consumir dos LUT6s en lugar de 1,5 como en el caso del compresor [4:2] optimizado en área. Si se crea un único compresor se consume completamente la cadena de acarreo, pero se pueden utilizar las restantes LUT6 del *slice* para otras funciones. Si se necesitan más de un compresor la clave está en empaquetarlos dentro del mismo *slice* para aprovechar completamente la cadena de acarreo. Por ejemplo, un compresor [6:2] optimizado en velocidad compuesto al unir dos compresores [4:2] optimizados en velocidad, como se verá más adelante en este mismo apartado, puede aprovechar el Slice completo (Figura 26). Dos LUT6s para un compresor [4:2], dos LUT6s para el otro y la lógica de acarreo completa compartida por los dos compresores [4:2], resultando este compresor [6:2] el más eficiente en velocidad.

En el diseño presentado se ha visto cómo se utiliza la lógica de acarreo como parte de la lógica para generar funciones cualesquiera y no sólo como acarreo. La utilización de la lógica de acarreo, en la medida de lo posible, permite sintetizar circuitos más rápidos ya que utiliza estas conexiones "de fábrica" y permite implementar funciones (junto con las LUTs ya disponibles) de forma que se tiene una entrada más disponible en las CLBs. Esta idea no es nueva y se han propuesto trabajos que intentan aprovechar la lógica de acarreo junto con las LUTs para cualquier circuito general. Una utilización de la cadena de acarreo es muy útil para crear comparadores muy eficientes. En [81] y [82] y [83] hemos utilizado este tipo de comparadores en circuitos periféricos a los que se accede a través de un bus.

También se han propuestos métodos para mapear de manera automática funciones en las LUTs y la lógica de acarreo. En trabajos como [84] y [85] Frederick y Somani intentan un mapeo automático de funciones lógicas generales en estas cadenas de acarreo junto con las LUTs. Sin embargo suponen algunas funcionalidades extra de la cadena de acarreo que no están disponibles en las principales familias de FPGAs. En [86] y [87] se describe un mapeo similar, pero con el único requisito de que las cadenas

soporten suma binaria. Sin embargo, aún en este caso las cadenas de acarreo tienen ciertas restricciones, como por ejemplo en las FPGAs Spartan 6, donde la cadena de acarreo se trata como un elemento indivisible, por lo que no basta con la condición de suma binaria.

3.3.4. Implementando compresores de orden superior en LUT6

Como se ha mostrado en el apartado anterior, son posibles dos opciones para la implementación de compresores de orden mayor al [4:2], por un lado, describiendo en VHDL compresores mayores componiendo compresores [3:2] para conseguir una optimización en área, y por otro lado utilizando compresores [4:2] como elemento básico como el que se describió en el apartado anterior con primitivas. Esta segunda opción es sólo posible en *slices* L y M que son los que contienen lógica de acarreo. A continuación se muestra como se construyen estos compresores de orden superior al [4:2].

Compresores de orden superior [5:2], [6:2] y [7:2] optimizados en área

La Figura 23 muestra, a modo de ejemplo, la síntesis de un compresor [5:2] a partir de compresores [3:2] como muestra la Figura 14. Como se puede observar se consumen dos LUT6. Obsérvese cómo uno de los acarreos se genera en la propia LUT6. Téngase en cuenta que las entradas a_i, b_i y c_i se utilizan en una LUT6 para generar el c_{p1_i} o s_{2_i} por lo que no se debe pensar que en esas LUT6 se suman cinco bits y se van a necesitar 3 salidas.

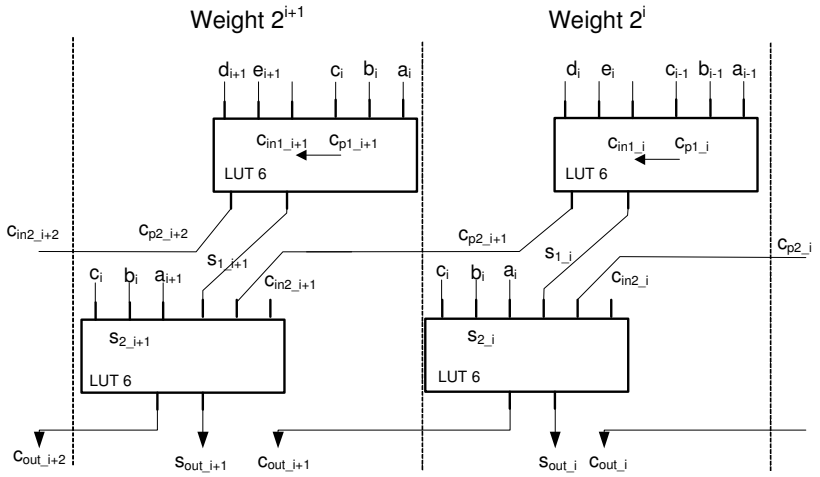

Figura 23. Síntesis del compresor [5:2] optimizado en área.

El compresor [6:2] se construye con compresores [3:2] como muestra la Figura 24 y el compresor [7:2] se construye también conectando los compresores [3:2] como muestra la Figura 16 (no se muestra la figura del compresor [6:2] ni [7:2], resultado de la síntesis de la descripción VHDL por la complejidad de las conexiones).

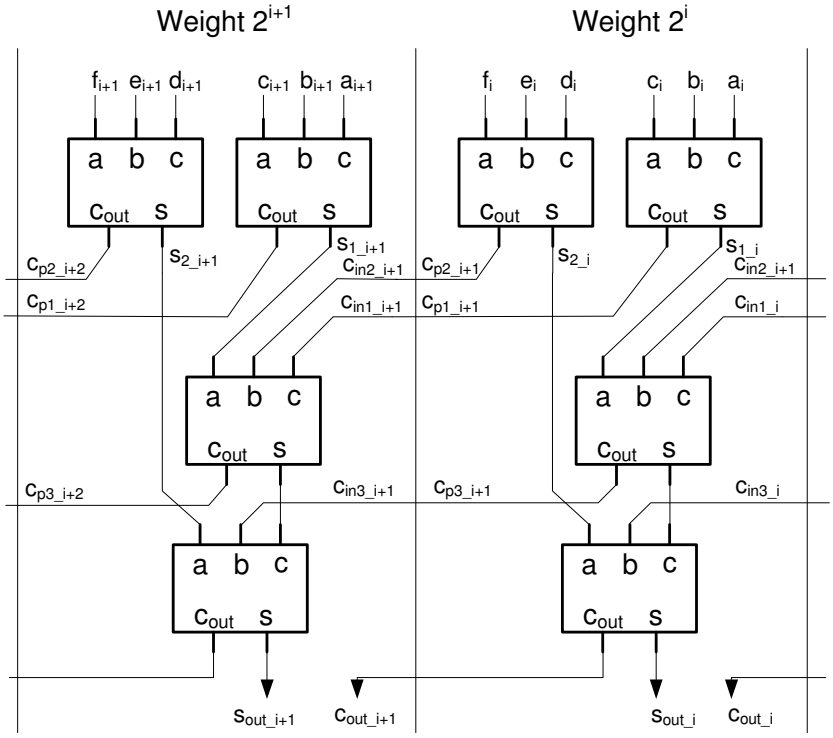

Figura 24. Implementación clásica de un compresor [6:2].

La Tabla 7 muestra el consumo en LUT6 y retardo para un compresor [*m*:2] de ancho de 32 bits en cada uno de los casos. Se ha utilizado un ancho de palabra de 32 bits para tener una medida más precisa, y no quede todo implementado en un solo *slice* que ofrecería el mejor resultado.

Compresor [*m*:2] de 32 bits Optimizado en área	Spartan 6 xc6slx100-3fgg484		
	LUT6	Nº LUT6 *path Delay*	*Delay* (ns)
[4:2]	48	2	2,59
[5:2]	66	2	2,99
[6:2]	97	3	4,14
[7:2]	132	3	4,21

Tabla 7. Compresores [*m*:2] de 32 bits optimizados en área.

Hay que hacer notar que hay una diferencia de velocidad entre el compresor [4:2] y [5:2], cosa que en principio parece extraña, ya que utilizan los mismos niveles de

LUT6. Estudiado el caso, se ha comprobado que es correcto, y se debe al retardo de la red de conexión de uno de los acarreos propagados. El compresor [4:2] sólo tiene como entrada un acarreo propagado del bit anterior, que se genera en la propia LUT6, mientras que el compresor [5:2] tiene dos. Uno de ellos se genera en la propia LUT6 como en el caso de la implementación del [4:2], mientras que el restante acarreo del [5:2], proveniente del bit anterior, pasa por una red más lenta. Motivo por el cual el compresor [5:2] siempre es un poco más lento.

Compresores de orden superior [5:2], [6:2] y [7:2] optimizados en velocidad

En el caso de compresores [5:2] y [7:2] se sintetizan componiendo compresores [4:2] optimizados en velocidad con compresores [3:2], mientras que en el caso de compresores [6:2] se sintetizan únicamente con compresores [4:2] optimizados en velocidad. La Figura 25 muestra como se sintetiza el compresor [5:2].

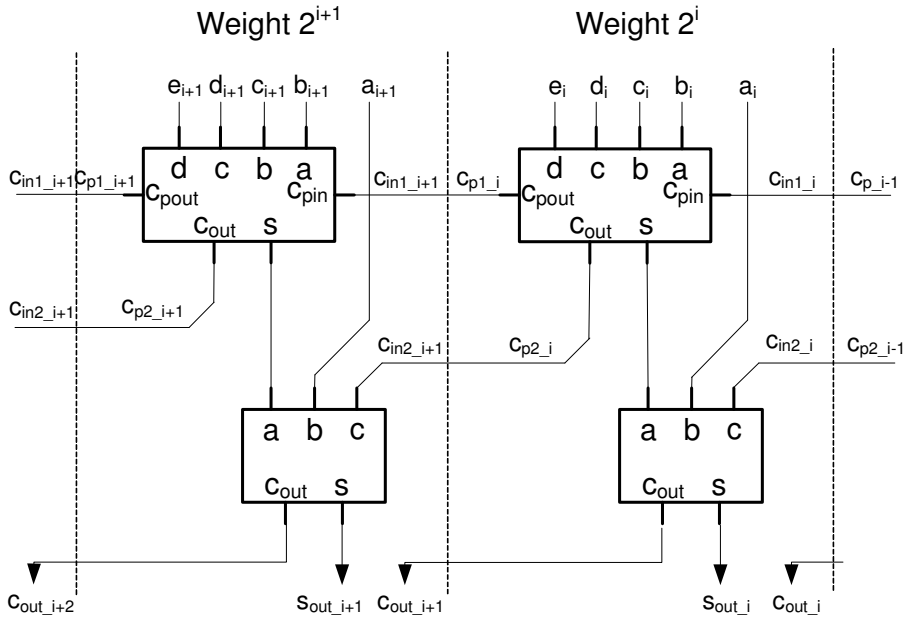

Figura 25. Compresor [5:2] optimizado en velocidad.

La Figura 26 muestra un compresor [6:2] componiendo dos compresores [4:2]. Téngase en cuenta observando la figura del compresor [4:2] anterior que la entrada c_{pin} no se propaga. Por tanto, la salida c_{pout} debe llevarse a la entrada c_{pin} del siguiente compresor para cortar el acarreo. Un compresor [6:2] ocupa un *slice* completo. Dado que los retardos dentro del *slice* son más pequeños que en la red de rutado este compresor está altamente optimizado. Finalmente, la Figura 27 muestra cómo se construye un compresor [7:2].

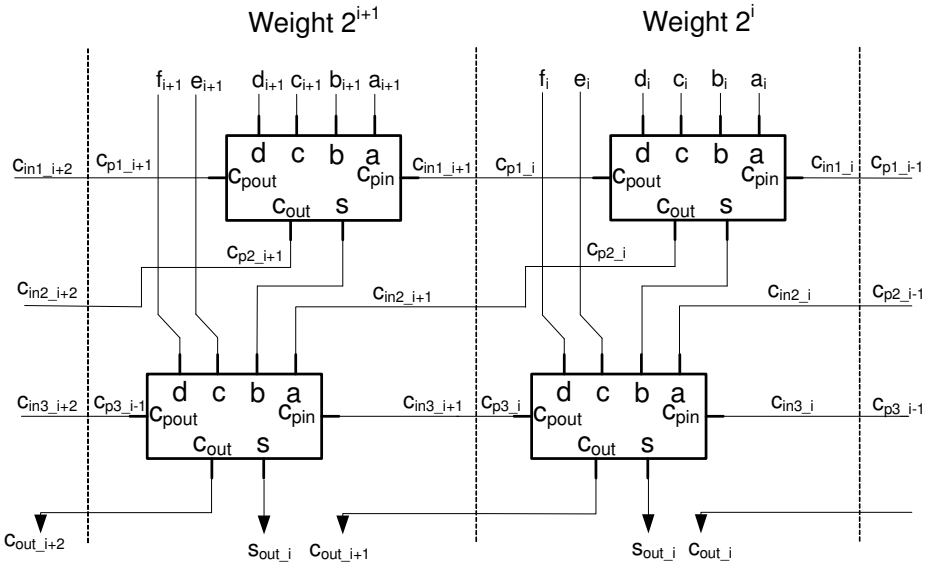

Figura 26. Compresor [6:2] optimizado en velocidad.

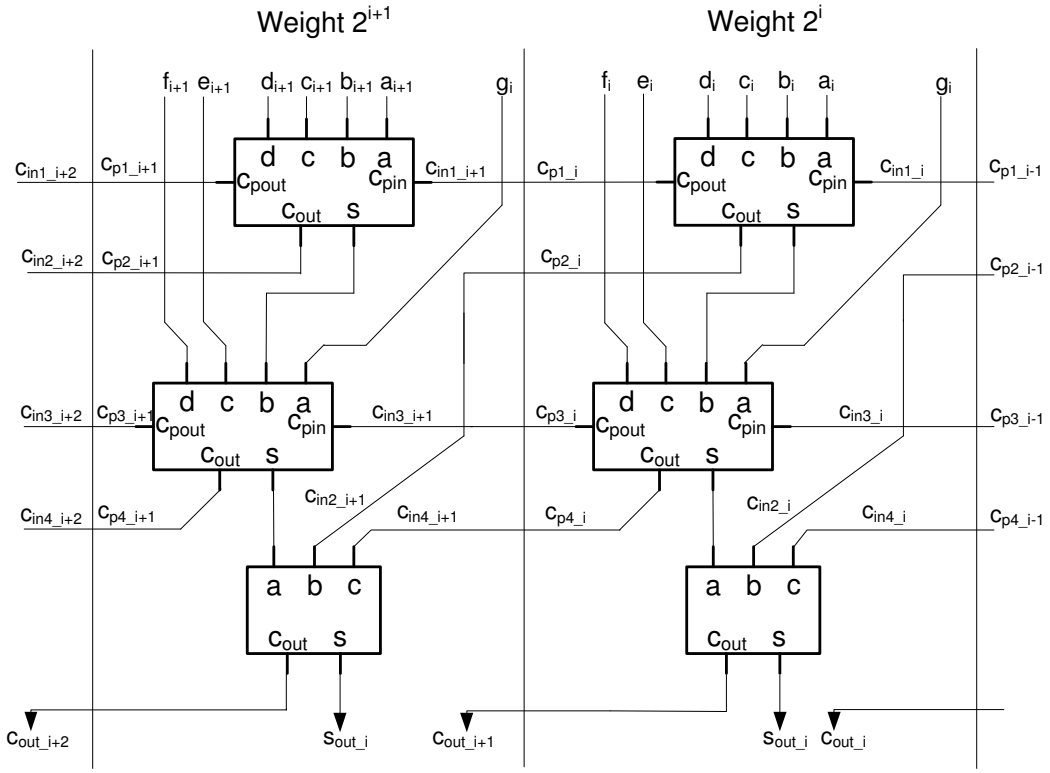

Figura 27. Compresor [7:2] optimizado en velocidad.

Por último, para tener una comparativa de los compresores optimizados en velocidad, la Tabla 8 muestra el consumo en LUT6 y retardo para un compresor [*m*:2] de ancho de 32 bits en cada uno de los casos. Se ha utilizado un ancho de palabra de 32 bits para tener una medida más precisa como en el caso de los compresores optimizados en área y no quede todo implementado en un solo *slice* donde los retardos son mínimos. En esta tabla además del número de LUT6s que atraviesa el camino más largo hay que

tener en cuenta el retardo de la lógica de acarreo (a diferencia de la Tabla 7) ya que en esta implementación se mezclan en el camino más largo, las LUT6s y la lógica de acarreo.

Compresor [m:2] de 32 bits Optimizado en velocidad	Spartan 6 xc6slx100-3fgg484		
	LUT6	Nº LUT6 *path Delay*	*Delay* (ns)
[4:2]	64	1+*Carry*	1,82
[5:2]	96	2+*Carry*	2,7
[6:2]	128	2+2x*Carry*	2,9
[7:2]	160	3+*Carry*	3,4

Tabla 8. Compresores [*m*:2] de 32 bits optimizados en velocidad.

3.3.5. Sumador ternario en LUT6

La facilidad de implementación de sumadores ternarios en una sola LUT6, permiten la suma final de tres operandos con un consumo similar en área a un sumador de dos. En la Tabla 9 se muestra una comparativa entre un sumador de 2 y de 3 operandos en cuanto a consumo de espacio y velocidad para suma de operandos de ancho en número de bits elevado. En cuanto a velocidad, los sumadores ternarios son más lentos, pero como se ha comentado ampliamente en los estudios de los capítulos anteriores, la suma convencional sólo se realiza en la conversión final cuando se utiliza aritmética CS, y sin embargo consumen el mismo área que un sumador de dos operandos. La posibilidad de implementar sumadores ternarios en una sola LUT6 fueron introducidos en las FPGAs de Xilinx por primera vez en la familia Virtex-5 [88] y mantenida esta posibilidad en el resto de FPGAs con LUT6 introducidas posteriormente [89].

Spartan 6 Anchura del sumador	Sumador (A+B)-CPA			Sumador (A+B+C)-*Ternary*		
	Slice	LUT6	*Delay* (ns)	*Slice*	LUT6	*Delay* (ns)
32 bits	8	32	2,05	9	32	3,03
64 bits	16	64	2,8	17	64	3,77
128 bits	32	128	4,29	33	128	5,26
256 bits	64	256	7,26	65	256	8,24

Tabla 9. Sumador de 2 operandos versus 3 operandos.

3.3.6. Sumador doble *carry-save* [4:3], [5:3], [6:3] y [7:3]

Como se ha visto en el apartado 3.3.5., los sumadores ternarios en FPGAs con LUT6 consumen el mismo espacio que un sumador de dos operandos aunque con un retardo mayor. En todo caso se pueden sumar 3 bits con el mismo coste que 2 bits. Proponemos un nuevo formato redundante donde cada dígito se representa por 3 bits.

En formato *carry-save* clásico cada dígito está representado por dos bits, uno de suma y otro de acarreo con valores posibles {0, 1, 2}. En el nuevo formato doble *carry-save* cada dígito está representado por tres bits, uno de suma y dos de acarreo con valores posibles {0, 1, 2, 3}.

Los sumadores doble *carry-save* [*m*:3] computan *m* bits de igual peso y generan una salida de tres bits del mismo peso. Un inconveniente que puede plantear la utilización del formato doble *carry-save* es que para almacenar un número en este formato se necesita un 50% más de *flip-flops*; sin embargo, éste no es ningún inconveniente para sistemas *pipeline* o serie puesto que a las salidas de los *slices* siempre se tienen disponibles *flip-flops*.

En lo que sigue mostraremos la estructura de los sumadores doble *carry-save* que proponemos y los resultados de implementación para estos sumadores en FPGAs con LUT6.

Sumador doble *carry-save* [4:3]

El sumador básico para el formato doble *carry-save* es el sumador [4:3] que se muestra en la Figura 28. Como se puede ver tiene el mismo coste y velocidad que un sumador [4:2]. Se puede utilizar, por ejemplo, para sumar un número convencional con uno redundante (por ejemplo en operaciones MAC) y aunque el retardo introducido es el mismo que un sumador [3:2] como parte del acumulador, el retardo práctico puede eliminarse, si el operando convencional se conecta a la entrada que va directa a la salida, la suma se puede realizar en paralelo con la computación de la entrada convencional (multiplicador).

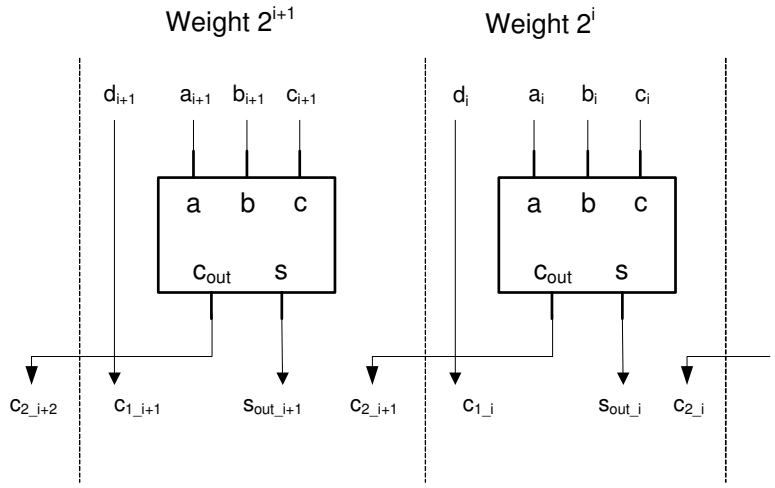

Figura 28. Sumador básico doble *carry-save* [4:3].

Sumadores doble *carry-save* [5:3], [6:3] y [7:3]

Los sumadores doble *carry-save* [5:3] (Figura 29), [6:3] (Figura 30) y [7:3] (Figura 31), se construyen a partir de contadores (5:3], (6:3] y (7:3] respectivamente. Cada contador genera dos bits de acarreo con pesos 2 y 4 que se redireccionan a sus correspondientes salidas. Estos contadores se describen fácilmente y solamente tienen un nivel de lógica. En el caso del sumador [5:3] su implementación requiere tres LUT5 para generar cada salida, por tanto se consumen sólo dos LUT6. En el caso del sumador [6:3] se necesitan tres LUT6, una para cada salida, y por último en el caso del sumador [7:3] se requieren dos LUT6 por salida, en total seis LUT6 de *slices* M y L en el caso más óptimo mostrado en la Figura 31.

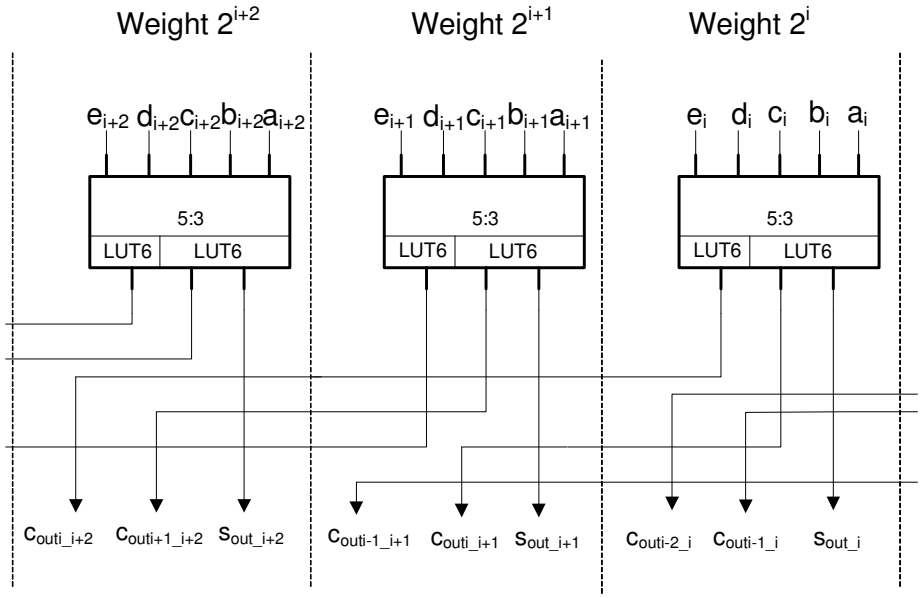

Figura 29. Sumador doble *carry-save* [5:3].

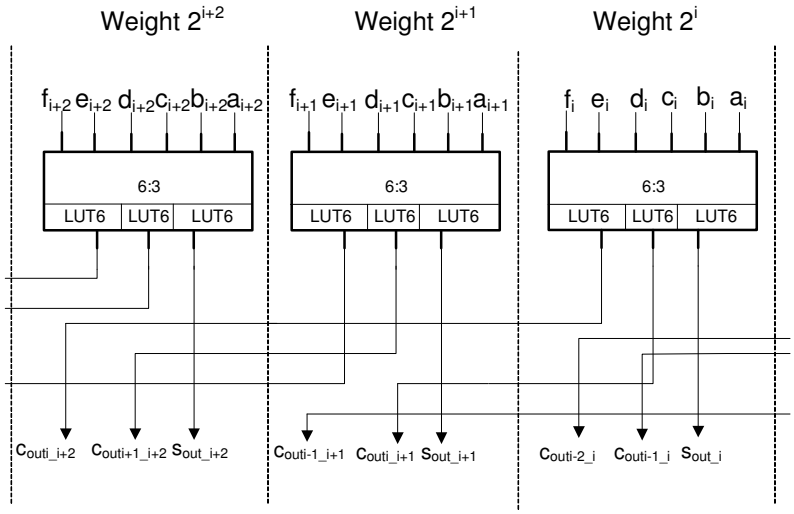

Figura 30. Sumador doble *carry-save* [6:3].

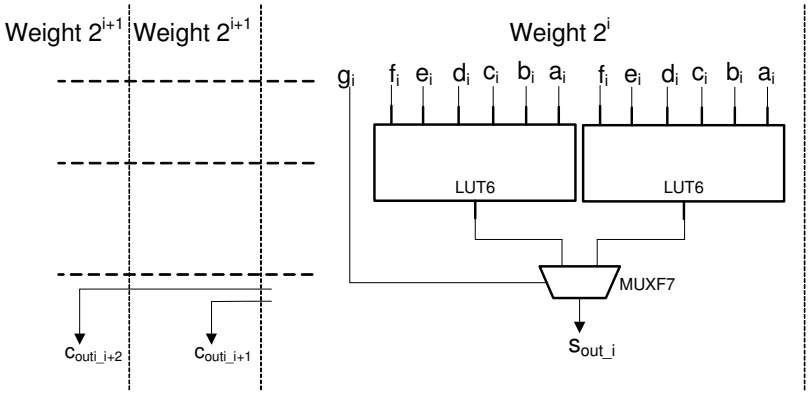

Figura 31. Sumador doble *carry-save* [7:3].

El sumador [5:3] reduce el número de operandos de entrada en dos, de la misma forma que el convencional compresor [4:2] por lo que podría sumar un número CS a un número doble *carry-save*. De igual manera el sumador [6:3] reduce el número de operandos en 3 de manera similar al clásico [5:2], por lo que se puede utilizar para sumar dos números en formato doble *carry-save* o tres en formato clásico *carry-save*.

La Tabla 10 muestra el retardo y consumo de espacio de los sumadores doble *carry-save*. Aunque los sumadores doble *carry-save* tienen un consumo mayor en área, respecto a los sumadores *carry-save* clásicos, destacan por un retardo menor. Esto también es cierto para el compresor [4:3], ya que como se comentó cuando se describió anteriormente, el retardo se puede eliminar en la mayoría de las aplicaciones reales. Destaca también que la variación del retardo con respecto al número de entradas es muy pequeña. Entre el sumador [4:3] al [7:3] hay una variación en el retardo de tan sólo 0,52 ns.

Compresor doble *carry-save* [*m*:3] de 32 bits	Spartan 6 xc6slx100-3fgg484		
	LUT6	Nº LUT6 *path Delay*	*Delay* (ns)
[4:3]	32	1	1,68 ns
[5:3]	64	1	1,93 ns
[6:3]	96	1	1,96 ns
[7:3]	192	1+ Mux	2,2 ns

Tabla 10. Sumadores doble *carry-save* [*m*:3] de 32 bits.

3.3.7. Análisis de características de compresores en LUT6

Hasta aquí se ha mostrado el diseño, consumo de recursos y velocidad de las alternativas para implementar compresores en LUT6. Debido al gran número de compresores y/o sumadores, se ha dejado para el apartado final una comparación entre ellos.

Los resultados se muestran para un sumador de 32 bits y la FPGA utilizada para la síntesis es la Spartan 6, referencia xc6slx100-3fgg484, utilizada en todo el trabajo para síntesis con LUT6. No se dan los resultados para un único bit, porque no hay una relación entera entre el número de bits y número de LUT6s. Por ejemplo, un compresor CSA [4:2] de 2 bits ocupa 3 LUT6s. Es importante señalar que los datos que se dan corresponden con las entradas y salidas registradas y por tanto se ha contabilizado el retardo de la red de rutado entre los registros y las LUT6s. Creemos que es más correcto. Por ejemplo, en el CSA [3:2] la implementación se hace en una sola LUT6, no sería correcto poner como único retardo el de la LUT6, porque ese CSA habrá que conectarlo a algún lugar. El retardo de una LUT6 es del orden de 0,2 ns., decir que se puede trabajar a una frecuencia de 5GHz no sería real.

Por otro lado, una de las ventajas de la utilización de compresores es que el retardo es prácticamente independiente del número de bits, por lo que dar resultados para distintos tamaños no tiene mucho sentido. Es cierto que aparecen pequeñas variaciones, pero están asociados a los retardos de la red de rutado de la FPGA.

La Tabla 11 muestra una comparativa en consumo y retardo de los distintos tipos de sumadores. Se han incluido los sumadores CPA a modo de referencia puesto que su retardo depende del número de bits a sumar. Al igual que ocurre con en las FPGAs con

LUT4, el retardo no es proporcional al número de niveles teóricos de compresor [3:2], como se suponía por extrapolación a lo que ocurre en ASIC.

En cuanto a consumo de LUT6 se observa que los compresores *carry-save* clásicos optimizados en área son la mejor alternativa. Estos compresores se consiguen con una descripción adecuada a partir de compresores [3:2] y una parametrización adecuada de la herramienta de síntesis XST, integrada en el entorno ISE. Los compresores *carry-save* clásicos optimizados en área son los que ocupan menos área, aproximadamente ocupa el 75% del área del sumador *carry-save* clásico optimizado en velocidad, y entre un 75% a 50% del área de un sumador doble *carry-save*.

Se puede observar que la mejor alternativa en el caso de necesitar velocidad, sería la utilización de los sumadores doble *carry-save* que consumen el mismo número de LUT6 hasta los sumadores [6:3] y un retardo en general menor que los sumadores *carry-save* clásicos. La única excepción es el compresor [4:2] optimizado en velocidad que aprovecha al máximo la lógica de acarreo. Destaca también que la variación del retardo respecto al número de entradas en el sumador doble *carry-save* es muy pequeña si la comparamos con el sumador *carry-save* clásico. La diferencia de retardo entre el sumador *carry-save* clásico [6:2] y [4:2] optimizados en área es de 1,55 ns., y en el caso de sumador *carry-save* optimizado en velocidad esta diferencia es de 1,08 ns., mientras que en el caso de los sumadores de doble *carry-save* la diferencia de retardo entre [7:3] y [5:3] es de tan sólo 0,27 ns.

Carry-save Optimizado en área			*Carry-save* Optimizado en velocidad			Doble *Carry-save*			Acarreo propagado (CPA)		
Tipo	LUT6	*Delay*	Tipo	LUT6	*Delay*	Tipo	LUT6	*Delay*	Tipo	LUT6	*Delay*
[3:2]	32	1,68 ns	-	-	-	[4:3]	32	1,68 ns	[2:1]	34	2,71 ns
[4:2]	48	2,59 ns	[4:2]	64	1,82 ns	[5:3]	64	1,93 ns	[3:1]	34	3,63 ns
[5:2]	66	2,99 ns	[5:2]	96	2,7 ns	[6:3]	96	1,96 ns	[4:1]	68	4,78 ns
[6:2]	97	4,14 ns	[6:2]	128	2,9 ns	[7:3]	192	2,2 ns	[5:1]	70	5,73 ns

Tabla 11. Características de sumadores de 32 bits *carry-save* para LUT6.

En el Gráfico 2 se representa el retardo de los distintos tipos de sumadores en función de la reducción de las entradas. Si se observa la pendiente de las gráficas, se puede comprobar la variación del retardo en función de la compresión que realizan. Como ya se ha comentado, destaca la variación del retardo tan pequeña en el caso de los sumadores doble *carry-save*.

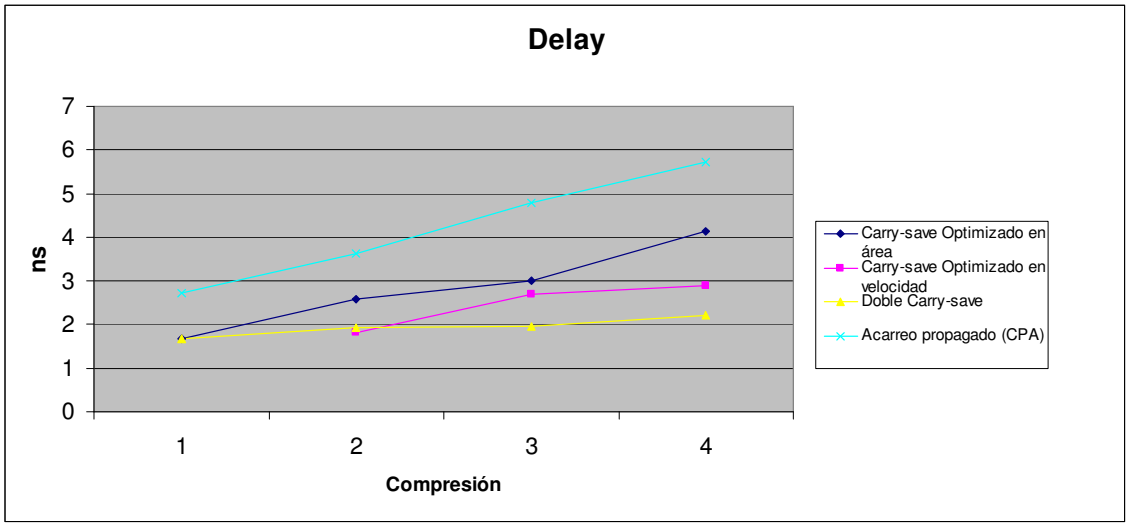

Gráfico 2. Comparativa de velocidad en sumadores de 32 bits *carry-save* para LUT6.

En el Gráfico 3 se representa el área ocupada de los distintos tipos de sumadores en función de la reducción de las entradas. El área aumenta de manera similar en todos los tipos de sumadores, excepto en el sumador doble *carry-save* donde se produce un salto importante en consumo de área al pasar del sumador [6:3] a [7:3], esto es debido a que las LUT6s tienen 6 entradas y cada salida se puede obtener consumiendo una sola LUT6 mientras que el compresor [7:3] tiene 7 por lo que no se puede obtener una salida de una sola LUT6.

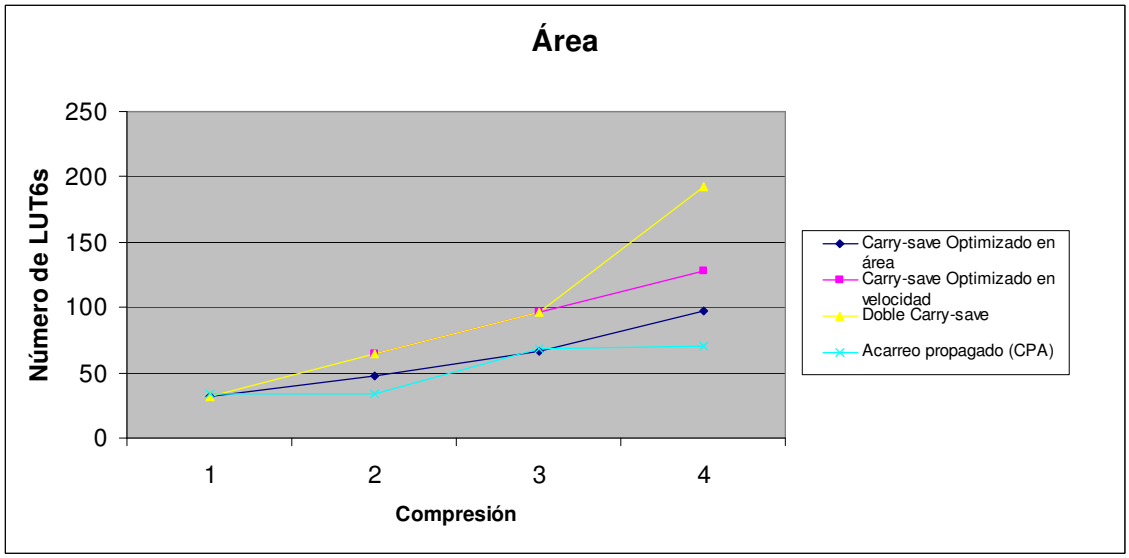

Gráfico 3. Comparativa de área consumida en sumadores de 32 bits *carry-save* para LUT6.

Por tanto, la suma redundante mejora el rendimiento de las FPGAs con LUT6, puesto que producen la misma reducción que el sumador CPA pero con menor retardo.

Por otro lado, los sumadores doble *carry-save* son los más rápidos a costa de un consumo de área mayor. Se concluye que no hay solución única, sino que dependiendo de cada aplicación, se debe elegir la mejor representación en función de los requisitos de área y velocidad.

4. Implementación de multiplicadores de ancho de palabra elevado

Breve resumen

En la realización de multiplicadores de ancho de palabra elevado en FPGAs se ha demostrado que la utilización de los multiplicadores empotrados combinándolos adecuadamente para crear los productos parciales es la mejor opción. Sorprendentemente en los trabajos actuales no se ha utilizado aritmética redundante para las sumas de los productos parciales, a pesar de ser la técnica que mejores resultados ha dado en ASIC. En este capítulo mostraremos como la combinación de compresores *carry-save* con los multiplicadores empotrados son la opción que ofrece el mejor rendimiento.

4.1. Introducción

La construcción eficiente de multiplicadores de ancho de palabra elevado a partir de los multiplicadores empotrados son objeto de interés por el elevado retardo y consumo de área. El diseño de unidades de multiplicación en punto fijo envuelve la generación de productos parciales. Para la suma de estos productos parciales se utiliza habitualmente aritmética redundante y se realiza la suma final con un sumador de acarreo propagado que concentra la mayoría del retardo. En muchas aplicaciones la suma final se puede posponer o mezclarse con las siguientes operaciones por los que el resultado de la multiplicación puede ser dado en aritmética redundante usualmente *carry-save* y de esta manera mejorar enormemente la velocidad del multiplicador.

Por estas razones, los multiplicadores con salida *carry-save* se utilizan ampliamente en los multiplicadores de unidades en punto flotante [90], unidades de multiplicación y suma [91][92], MAC [93] [94], procesamiento de imágenes [95][96], criptografía [98][99]. Los bloques DSP empotrados en las modernas FPGAs están implementados utilizando sumadores *carry-save* [100].

Una eficiente implementación de la multiplicación tiene un significativo impacto en FPGAs en términos de velocidad, consumo de potencia y área. Por estos motivos, los fabricantes de FPGAs incluyen multiplicadores empotrados, muy eficientes, que pueden ser utilizados por los diseñadores. La casi totalidad de FPGAs, incluyendo las de bajo coste, disponen de estos multiplicadores empotrados o bloques DSP que integran multiplicadores, cuyo uso es muy recomendado. En [101] Parandeh e Ienne realizan medidas de rendimiento entre multiplicadores "soft" y empotrados, proponiendo mejoras en el algoritmo de multiplicación adaptadas a las arquitecturas de las FPGAs que minimicen la distancia entre ambos tipos de multiplicadores.

La utilización de multiplicadores empotrados solamente se desaconseja para la realización de multiplicadores de ancho de palabra pequeños en torno a 9x9 bits o inferiores. En tales casos, podría tener mejor rendimiento una síntesis optimizada que utilice las LUTs. La razón es que el multiplicador quedaría cerca del diseño general y no se sumaría el retardo de la red de rutado, que es importante en el caso de que se utilicen los bloques empotrados debido a que se encuentran en posiciones fijas.

Cuando el tamaño de operandos del multiplicador es del orden de los multiplicadores empotrados (sin superarlo) la mejor opción es la utilización de estos multiplicadores empotrados, mientras que en el caso de que se necesiten multiplicadores de un ancho de palabra elevado, superior al de los multiplicadores empotrados, la mejor opción es la utilización de estos multiplicadores empotrados para realizar los productos parciales, y sumarlos posteriormente para tener el resultado final. Lo que difieren los trabajos actuales es en cómo realizar la suma de los productos parciales. Mientras que algunos autores se limitan a realizar la suma desde VHDL, y por tanto con sumadores CPA, otros proponen que la mejor manera de realizar las sumas es con la utilización de árboles de compresores.

Aunque en este capítulo se van a tratar multiplicadores paralelos de ancho de palabra elevados, los multiplicadores empotrados se han utilizado también en otras aplicaciones con un alto coste computacional, donde los autores han destacado la mejora del rendimiento [102] [103] [104].

En [102] Mcivor y otros realizaron una comparativa de síntesis en FPGAs de multiplicadores Montgomery. En los algoritmos comparados utilizaron los multiplicadores empotrados como la mejor alternativa, sin embargo, para la suma de los productos parciales utilizaban sumadores CPA.

En [103] De Dinechin y otros proponen un nuevo algoritmo para realizar la raíz cuadrada en FPGAs de "evaluación polinomial" que utilice los multiplicadores empotrados, frente a los algoritmos clásicos de "recurrencia de dígito" que no pueden utilizar este recurso. La latencia de este algoritmo es mucho menor que los anteriores a expensas de un consumo en recursos mayor.

En [104] Nadjia y otros proponen un algoritmo de multiplicación de precisión variable que utiliza los multiplicadores empotrados, distinto al de Karatsuba [105]. Este nuevo algoritmo presenta una baja complejidad en el rutado por lo que se obtiene una mejora de velocidad.

En el resto del capítulo se estudiará cómo se construyen multiplicadores con operandos de gran ancho de palabra en FPGAs utilizando los multiplicadores empotrados combinados con el eficiente mapeo de compresores *carry-save* como se mostró en el capitulo anterior. Aún con salida convencional, los multiplicadores

realizados con árboles de compresores *carry-save* obtienen una mejor velocidad a costa de aumentar el área. Si además se trabaja en formato CS, los multiplicadores *carry-save* consiguen duplicar la velocidad frente a los que utilizan sumadores CPA.

4.2. Diseño de multiplicadores de gran ancho de palabra en FPGAs

El diseño de multiplicadores de gran ancho de palabra en FPGAs se realiza a partir de los multiplicadores empotrados que contienen todas las FPGAs actuales. Para realizar multiplicadores de un número de bits mayor que el de los multiplicadores empotrados, se deben usar varios de estos multiplicadores para realizar una serie de productos parciales, y posteriormente sumarlos con la composición adecuada de los operandos intermedios.

En las FPGAs de bajo coste de Xilinx, objetivo de nuestro trabajo, los multiplicadores empotrados tienen un tamaño de operandos de 18 bits incluido el signo. Tanto las tradicionales series Spartan 3 como las más recientes como Spartan 6 contienen multiplicadores de 18x18 bits con signo, y el número varía según el modelo concreto. En las Spartan 6 los multiplicadores empotrados se encuentran dentro de otros bloques DSP empotrados más complejos [89].

Estos multiplicadores empotrados son relativamente pequeños, de 18x18 bits en las FPGAs de bajo coste o 25x18 bits en las FPGAs con alto rendimiento más recientes. Si se necesita un multiplicador dedicado de tamaño elevado es necesario combinar estos multiplicadores empotrados. Si suponemos que tenemos un multiplicador de tamaño *mxn* empotrado y un multiplicando de *p* bits de anchura y un multiplicador de *s* bits de anchura entonces tanto el multiplicando como el multiplicador deben dividirse en segmentos de un número de bits igual al ancho de los operandos de los multiplicadores empotrados. La división de los operandos en segmentos adecuados y el alineamiento de manera automática salen fuera de los objetivos de este trabajo. En [106], [107], [108], [109], [110] y [111] se estudia esta división y concatenación de manera automática. De esta manera, el multiplicando se descompone en $pd = p/m$ y el multiplicador en $sd = p/n$ [16]. En segundo lugar se computan los *pdxsd* productos parciales. Por último los productos parciales se alinean convenientemente y suman para tener el resultado final.

Los productos parciales y la suma final se pueden realizar de varias maneras. Se pueden calcular uno tras otro utilizando un único bloque multiplicador [112], que

requiere varios ciclos y se ha utilizado en computación multiprecisión. Se pueden utilizar varios multiplicadores especializados en *pipeline* para conseguir el resultado en un único ciclo con una pequeña latencia [100][111].

Se pueden realizar los cálculos en un solo ciclo (en paralelo), computándose todos los productos parciales al mismo tiempo y sumarlos [113]. Independientemente de la manera en cómo se computen, la utilización de sumadores *carry-save* para realizar la suma de los productos parciales aumenta el rendimiento. Este trabajo se centrará en conseguir multiplicadores del tamaño de los operandos iniciales y que se realicen en un solo ciclo, calculándose los productos parciales de forma paralela que es la manera más sintetizada en FPGAs.

Varios estudios han tratado la implementación paralela de multiplicadores de gran tamaño o elevar un operando al cuadrado utilizando los multiplicadores empotrados. Algunos de ellos para enteros sin signo [114], para enteros en complemento a dos tanto para multiplicadores de 18x18 bits [115] como para los nuevos multiplicadores asimétricos (25x18 bits) de las más nuevas FPGAs [116][117]. Como demostraremos en este capítulo, en todos estos trabajos se pueden sustituir los sumadores convencionales CPAs por sumadores *carry-save* aumentando el rendimiento.

Hay que hacer hincapié en que el signo debe tratarse con cuidado para un correcto diseño de los multiplicadores de gran ancho de palabra. Los multiplicadores empotrados trabajan en complemento a dos, independientemente de que las entradas sean operandos con signo o sin signo. En el caso de operandos sin signo, todos los segmentos en que se han dividido los operandos iniciales son operandos sin signo, mientras que para operandos iniciales con signo los segmentos deben tener signo o no dependiendo de la posición de los segmentos. En la representación de complemento a dos solamente los segmentos que contengan el bit de signo, es decir los segmentos que contengan los bits más significativos de los operandos, deben operarse con signo mientras que el resto deben tratarse como operandos sin signo.

Por tanto, en la generación de los productos parciales de aquellos segmentos que deben operarse sin signo, se debe introducir un cero en el bit más significativo de los multiplicadores empotrados. De esta manera el tamaño de los dígitos que deben introducirse en estos multiplicadores dedicados debe ser una unidad menor. Por otro lado, para el caso de los segmentos que contengan el bit de signo de alguno de los

operandos, en el resultado del producto parcial debe realizarse una extensión de signo. En el caso de los productos parciales sin signo, antes de realizar una extensión con ceros, se debe estudiar si se pueden combinar varios productos parciales para componer un solo resultado.

A modo de ejemplo, la Figura 32 muestra cómo se puede conseguir implementar un multiplicador con signo de 35x35 bits utilizando cuatro multiplicadores empotrados con signo y dos sumadores. Cada uno de los operandos se divide en dos grupos; por ejemplo, para el operando *A* se tendría: $A_1 = A[34..17]$ y $A_0 = A[16..0]$. El segundo grupo (el menos significativo) tiene un bit menos porque hay que añadirle un 0 por la izquierda para forzarlo positivo, antes de aplicarlo al multiplicador correspondiente *(0, A[16..0])*. De igual manera se procede con el operando *B*. Al tenerse 4 grupos de operandos, pero agrupados dos a dos, serán necesarios 4 multiplicadores empotrados de 18 bits.

En la Figura 33 se muestra la composición de las sumas parciales a realizar para obtener el multiplicador final de 35x35 bits. Tal y como se puede observar, los 17 bits menos significativos del producto se obtienen directamente al multiplicar A_0xB_0, y harán falta 2 sumadores, uno de 36 bits para realizar la suma intermedia de A_0xB_1 y A_1xB_0, y otro final de 53 bits, que suma el resultado de la suma intermedia junto con el sumando compuesto por A_1xB_1 y los 17 bits de mayor peso del producto de A_0xB_0. En este caso no se debe extender el resultado de este producto parcial (A_0xB_0) sin signo con ceros, ya que entonces no se podría combinar con los otros productos parciales. Antes de realizar la suma final debe extenderse el signo del resultado de la suma intermedia.

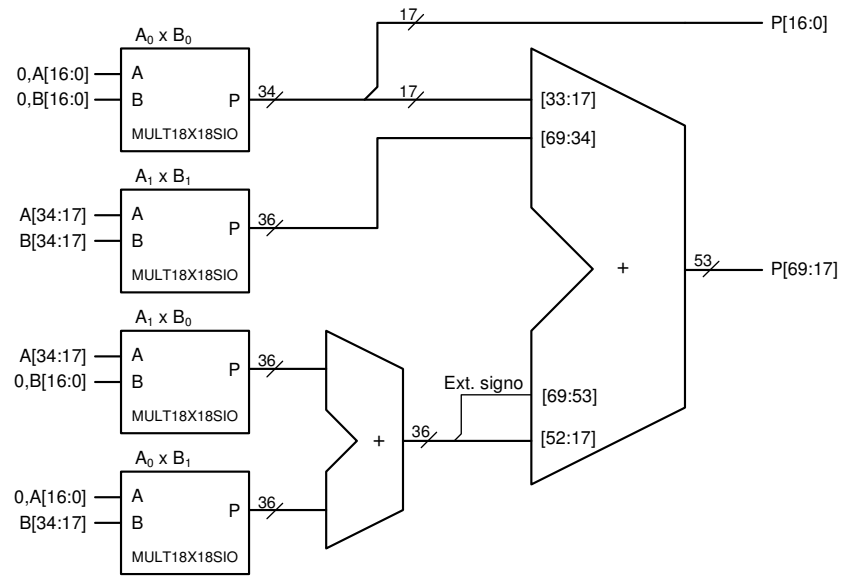

Figura 32. Multiplicador con signo de 35x35 bits.

Figura 33. Productos parciales de un multiplicador de 35x35 bits con signo.

La implementación del multiplicador mostrada en la Figura 32 tiene el inconveniente de que se necesitan dos sumadores, y si éstos son del tipo CPA tendrán un retardo elevado, por lo que el multiplicador final tendrá un retardo de propagación, que no sería admisible para la muchas de las aplicaciones. La solución sería usar un sumador CSA que permite sumar 3 operandos y al representar la salida mediante dos bits, el de acarreo y el de suma, el retardo final del sumador es independiente del número de bits. Esta mejora es mayor a medida que aumenta el número de bits del multiplicador, ya que aumenta el número de productos parciales a sumar.

Multiplicadores utilizando aritmética CSA

En el ejemplo anterior del multiplicador de 35x35 bits (Figura 32) basta sustituir los sumadores CPA por CSA para obtener a la salida un multiplicador con salida CSA. Para ello se deben utilizar compresores [3:2]. Se consigue así un multiplicador con salida CS. Para ver mejor los beneficios de la utilización de los sumadores CS, a modo de ejemplo, se mostrará cómo se obtiene un multiplicador de 52x35 bits con signo y

salida *carry-save* a partir de multiplicadores empotrados y sumadores CSA. La Figura 34 muestra los productos parciales indicando los casos en los que debe realizarse la extensión de signo.

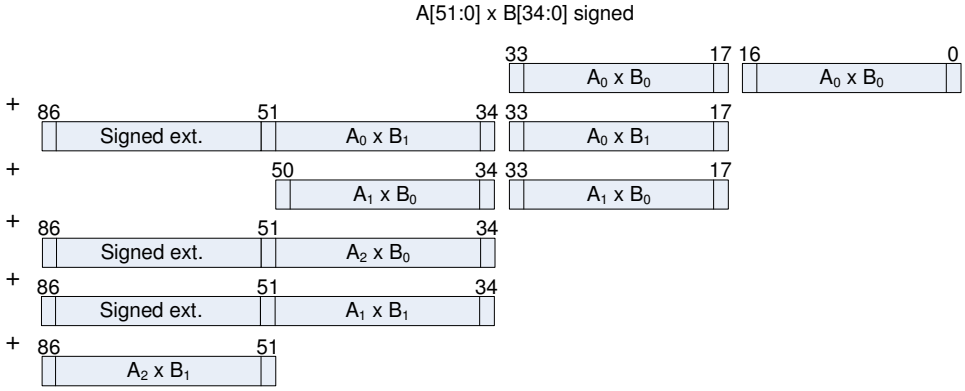

Figura 34. Productos parciales de un multiplicador de 52x35 bits con signo.

La Figura 35 muestra la implementación del multiplicador utilizando multiplicadores empotrados y sumadores CSA. Como se puede observar es necesario utilizar sumadores CSA de [3:2] y [4:2]. En [44] demostramos cómo se podían integrar de manera óptima sumadores *carry-save* [3:2] y [4:2].

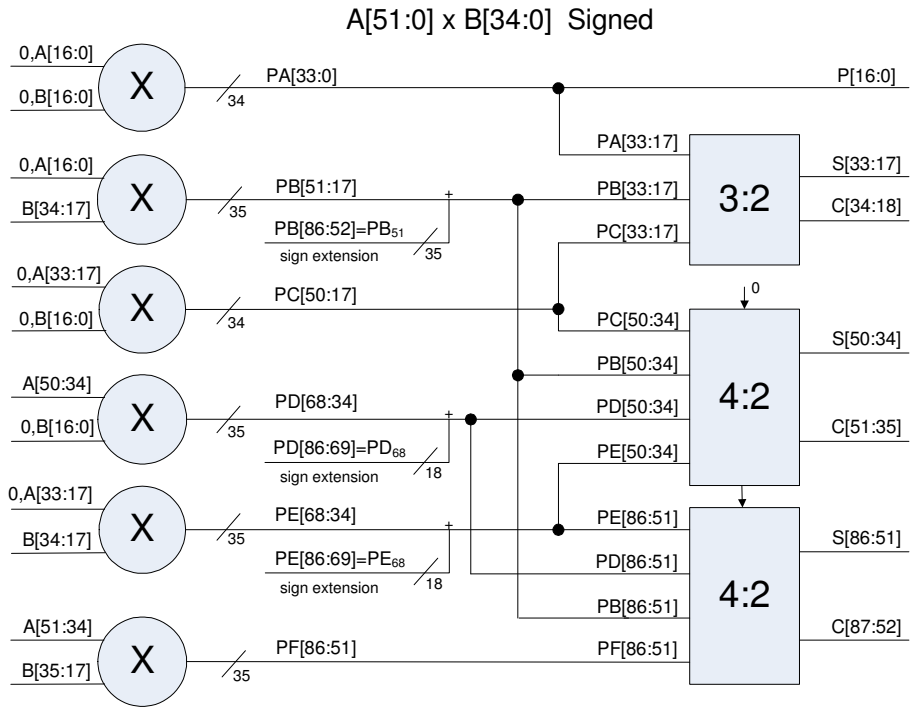

Figura 35. Multiplicador de 52x35 bits con signo y salida de suma y acarreo.

Para multiplicadores de número de bits superiores al mostrado anteriormente, por ejemplo de 69x69, se necesitan compresores superiores, en concreto de [5:2] y

[7:2]. Por este motivo, la optimización de los sumadores CS influye enormemente en el retardo del multiplicador. En el capitulo 3 ya mostramos además cómo conseguir estos compresores de manera óptima.

4.3. Resultados de la implementación de multiplicadores de ancho de palabra elevado en FPGAs con LUT4

Para ver las ventajas de utilizar multiplicadores con salida *carry-save* en FPGAs de Xilinx con LUT4, se han diseñado varios multiplicadores para ancho de operandos en el rango de 35 a 69 bits y se han sintetizado en una FPGA Xilinx Spartan3E-5 utilizando ISE 12.1 y se han comparado con los multiplicadores generados por IP core generator de Xilinx (Coregen 12.1) [118] y los generados directamente del operador de multiplicación de VHDL utilizando XST. Los datos que se muestran se han obtenido registrando las entradas y salidas de cada bloque. De esta forma se puede determinar la frecuencia máxima de funcionamiento de una forma más fiable e independiente del *place and route* del diseño en la FPGA. Por el contrario, esto implica un aumento del número de LUT y *slices*. Sobre los datos mostrados cabe indicar, que éstos se han realizado para diseños sin *pipeline*, y para un grado de velocidad de la FPGA indicada anteriormente de -5, y que los multiplicadores obtenidos con la herramienta *Coregen* de Xilinx y con la descripción VHDL tienen salida convencional.

La Tabla 12 muestra los resultados obtenidos. En primer lugar el número de multiplicadores empotrados es el mismo en las tres implementaciones. El número de LUTs construidas en el diseño con salida CS es menor, variando desde un 5% a un 19%, aunque se debe tener en cuenta que en la implementación CS no está incluido el sumador para la conversión final.

En cuanto a mejora en la velocidad, se puede comprobar que, para un tamaño de operando de 35 bits, la mejora de velocidad con multiplicadores CSA es superior en un 82% para los multiplicadores generados con *Coregen* y en un 56% a la obtenida con la descripción VHDL. A medida que aumenta el tamaño del operando la mejora de velocidad es aún mayor, consiguiendo una frecuencia de trabajo dos veces superior a los multiplicadores generados con *Coregen* y 73% superior a la obtenida con la descripción VHDL.

Otro resultado obtenido, es que el retardo de los multiplicadores CSA es prácticamente el mismo en el caso de operandos de 35x52 bits y de 52x52 bits ya que los compresores utilizados en ambos casos son de [5:2]. Este es un resultado esperado, ya que los compresores son los que determinan la velocidad del multiplicador CSA. Esta conclusión se puede extrapolar a operandos de mayor número de bits.

Multiplicador Tamaño en bits	Multiplicador salida CSA				*IP Coregen* de Xilinx				Multiplicador descrito en VHDL			
	Slice	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)
35x35	53	106	4	134,2	108	124	4	71	72	125	4	82,74
35x52	159	193	6	106,27	117	212	6	55,5	117	215	6	65,72
52x52	214	320	9	105	171	336	9	45,74	206	392	9	57,60
64x64(1)	334	566	16	89.52	358	703	16	41,65	382	737	16	52,62
69x69(2)	386	745	16	89,1	-	-	-	-	419	817	16	51,60

Tabla 12. Resultados de multiplicadores con signo en FPGA con LUT4.

Notas:
(1) Tamaño máximo con *IP Coregen*.
(2) Tamaño máximo diseñado con sumadores CSA.

La Tabla 13 muestra los resultados para multiplicadores sin signo. La mejora en velocidad alcanzada por el multiplicador sin signo con salida CS es similar al caso de multiplicadores con signo, ya que el tamaño máximo de los compresores necesarios para realizar las sumas de los productos parciales es el mismo tanto para multiplicadores con signo como sin signo. La síntesis con compresores *carry-save* mejora en un rango entre el 59% al 72% a la síntesis VHDL y en un rango de 88% a 124% a la síntesis con *Coregen*. Sin embargo, se observa una importante reducción de área cuando se utiliza el multiplicador con salida *carry-save*, siendo menor en un 40% respecto a *Coregen* y en un rango de 33% a 20% con respecto a la descripción VHDL.

Multiplicador Tamaño en bits	Multiplicador salida CSA				*IP Coregen* de Xilinx				Multiplicador descrito en VHDL			
	Slice	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)
34x34	34	68	4	144.89	104	120	4	77.22	69	102	4	84.47
34x51	86	120	6	117.9	158	208	6	60.20	94	154	6	73.96
51x51	172	206	9	110.72	231	334	9	49.46	163	256	9	65.12
63x63(1)	229	388	16	92.31	443	727	16	45.56	316	505	16	55.58
68x68(2)	448	515	16	91.90	-	-	-	-	342	545	16	54.34

Tabla 13. Resultados de multiplicadores sin signo en FPGAs con LUT4.

Notas:
(1) Tamaño máximo con IP Coregen.
(2) Tamaño máximo diseñado con sumadores CSA.

Esta reducción en área del multiplicador sin signo CSA con respecto al que trabaja con signo es debida a que no es necesaria la extensión de signo. Los compresores de mayor tamaño se reducen especialmente, sobre todo los compresores que intervienen en los bits más significativos del resultado como se puede ver en la Figura 36. En el caso del multiplicador con salida convencional los productos parciales que contribuyen en los bits más significativos del resultado se tienen que sumar realmente incluyendo la extensión de cero para propagar los posibles acarreos. Sin embargo, en *carry-save* sólo se propaga a un bit y el resto de la suma solamente se debe almacenar.

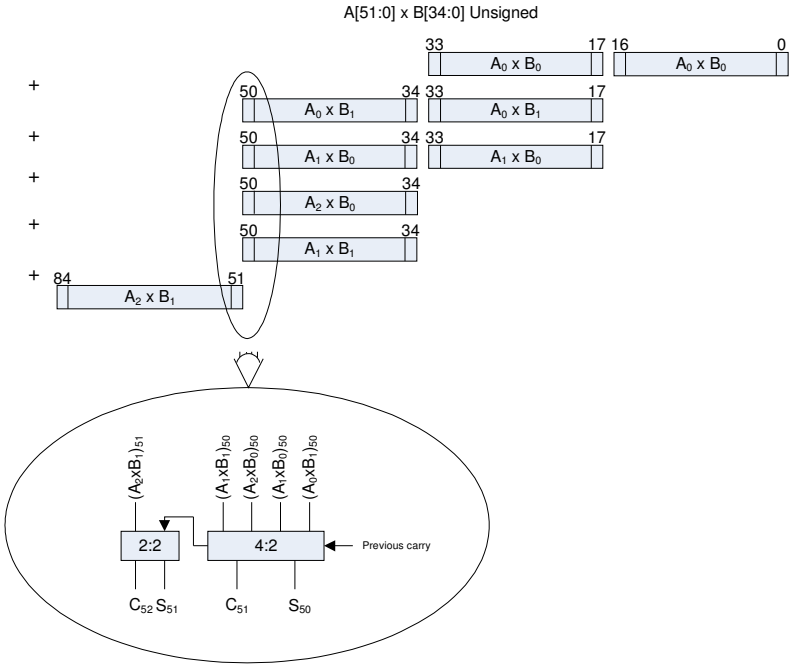

Figura 36. Multiplicador de 51x34 bits sin signo con salida CSA.

4.4. Resultados de la Implementación de multiplicadores de ancho de palabra elevado en FPGAs con LUT6

Los multiplicadores se realizan a partir de la suma de las multiplicaciones parciales realizadas con los multiplicadores empotrados. Los multiplicadores empotrados en las FPGAs con LUT6 se encuentran dentro de los bloques DSPs. En estos bloques también existen sumadores CPA que pueden combinarse con las salidas de los multiplicadores empotrados.

La implementación de multiplicadores en LUT6 sigue la misma filosofía que la expuesta para LUT4, es decir, que las sumas de los productos parciales se realicen con compresores CS, utilizando el compresor del tamaño adecuado para realizar la suma de cada bit de todos los operandos. No hemos creado árboles de compresores a partir de compresores más pequeños, sino que utilizamos el compresor con el número de entradas adecuado, ya que como se comentó en el apartado 3.3.7., no existe una relación lineal en el sentido de que dos compresores [3:2] sintetizados y unidos para componer un compresor [4:2] (retardo 2 x 1,68 ns.) no tiene el doble retardo que un compresor [4:2] descrito y sintetizado posteriormente (retardo 2,59 ns.).

La Tabla 14 muestra los resultados de síntesis para distintos tamaños de multiplicadores. Se muestran resultados para multiplicadores con salida en formato redundante *carry-save* y multiplicadores con salida en formato convencional, realizados completamente con aritmética en formato convencional como ocurre en los multiplicadores sintetizados con *Coregen* o VHDL, o realizados con aritmética CS a los que se les ha añadido un sumador CPA para realizar la conversión a formato tradicional.

Los multiplicadores con salida doble *carry-save* solamente tienen sentido compararlos a partir de un tamaño del multiplicador de 52x52 bits que es cuando se necesitan sumadores de [5:3] o superiores. Las versiones actuales de *Coregen* solamente permiten un tamaño máximo de multiplicador de 64x64 bits, aunque los datos de este tamaño se utilizan para realizar comparaciones con el tamaño de 69x69 en el resto de las síntesis, sin que esto induzca a conclusiones erróneas.

En la documentación de Xilinx [89] se propone la creación de multiplicadores y otros circuitos que requieran sumas intermedias realizándolas en cascada en lugar de árbol, aunque eso sí, siempre con sumadores CPA. La propuesta reduce enormemente el

consumo de área porque aprovecha los sumadores que contienen los bloques DSP. Por esta razón, el consumo de LUT6s de las síntesis con *Coregen* es nulo. Sin embargo, como muestra la Tabla 14, la frecuencia de trabajo disminuye enormemente con el ancho del número de bits. Por tanto, si se necesita aumentar la velocidad, la mejor alternativa es la utilización de multiplicadores realizados con CSA.

Multiplicador Tamaño en bits	Multiplicador salida CS				Multiplicador salida doble *carry-save*				*IP Coregen* de Xilinx			
	Slice	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)
35x35	24	53	4	146,71	-	-	-	-	24	0	4	53,82
35x52	37	96	6	122,56	-	-	-	-	30	0	6	39,28
52x52	71	199	9	120,17	56	157	9	135,28	36	0	9	28,61
64x64(1)	-	-	-	-	-	-	-	-	45	0	16	17,67
69x69(2)	178	456	16	100,99	282	548	16	122,97	-	-	-	-

Multiplicador Tamaño en bits	Multiplicador descrito en VHDL				Multiplicador(3) CSA con salida convencional (CPA)				Multiplicador(3) doble *carry-save* con salida convencional (CPA)			
	Slice	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)	*Slice*	LUT	Mul 18x18	Frec. Max. (MHz)
35x35	6	0	4	55,46	53	89	4	124,18	-	-	-	-
35x52	19	53	6	46,65	60	165	6	98,2	-	-	-	-
52x52	46	160	9	43,47	83	243	9	97,73	133	330	9	95,08
64x64(1)	-	-	-	-	-	-	-	-	-	-	-	-
69x69(2)	67	245	16	41,44	225	592	16	81,69	323	762	16	87,59

Tabla 14. Resultados de multiplicadores con signo en FPGAs con LUT6.

Notas:
(1) Tamaño máximo con IP Coregen.
(2) Tamaño máximo diseñado con sumadores CSA.
(3) Los datos que se muestran corresponden a una optimización por área. También se ha sintetizado con optimización por velocidad, pero la mejora es inapreciable y el consumo de área extra es elevado.

En cuanto a velocidad, los multiplicadores con salida CS triplican en el peor caso la velocidad de los multiplicadores realizados con *Coregen* o VHDL, siendo de hasta 6 veces más rápidos que el multiplicador mayor realizado con *Coregen*. En cuanto a consumo de recursos, la síntesis realizada con *Coregen* es la que menos área ocupa ya que el multiplicador se realiza completamente con los bloques DSPs construyendo sumadores en cascada. Sin embargo, en la síntesis con *Coregen* la frecuencia de trabajo es altamente dependiente del tamaño del multiplicador, siendo en unas tres veces superior en el caso del multiplicador más pequeño (35x35) con respecto al mayor

(64x64). En el caso de la síntesis desde VHDL la variación de frecuencia respecto al tamaño del multiplicador es relativamente pequeña y aún menor en el caso de los multiplicadores realizados con CSA.

El Gráfico 4 y Gráfico 5 muestran los resultados de síntesis para multiplicadores con salida convencional realizados con aritmética CSA y tradicional de la Tabla 14. En el caso de que se necesiten multiplicadores con salida convencional, los sumadores *carry-save* son los más rápidos aunque con un consumo en área mayor. Con respecto al formato *carry-save*, se puede observar que aunque el formato doble *carry-save* permite algo más de velocidad que el clásico lo hace con un consumo en área mayor. El formato doble *carry-save* en este caso no supone una mejora apreciable. Esto es debido a que en el proceso de conversión a formato tradicional, se utiliza un sumador ternario que tiene un retardo mayor que el de dos operandos y consume el tiempo que mejora el uso de sumadores con formato doble *carry-save*.

Con respecto al formato *carry-save*, no hay una única solución idónea para multiplicadores, dependerá de la aplicación concreta. En el caso de multiplicaciones que se realicen en serie, o por ejemplo en operaciones MAC, los multiplicadores con formato doble *carry-save* son los más rápidos, aumentando la diferencia a medida que aumenta el número de bits del multiplicador siendo 1,13 veces más rápido para un multiplicador de 52x52 bits y de 1,22 veces más rápido para un multiplicador de 69x69 bits. En multiplicaciones paralelas, si tenemos en cuenta que para el multiplicador de 52x52 bits se necesitan sumadores *carry-save* con 5 entradas ([5:2], [5:3]) y para el multiplicador 69x69 bits se han necesitado sumadores con 7 entradas ([7:2], [7:3]), podemos concluir que excepto que la velocidad sea el único condicionante el formato *carry-save* clásico será el más adecuado.

Por último, comentar que al igual que en el caso de LUT4, si se necesitan multiplicadores con ancho de palabra elevado y sin signo, cuando se sintetizan utilizando aritmética CS tienen una velocidad similar y el área disminuye notablemente, mientras que en el caso de multiplicadores sin signo realizados con aritmética tradicional, la velocidad y el área son la misma que para el multiplicador con signo.

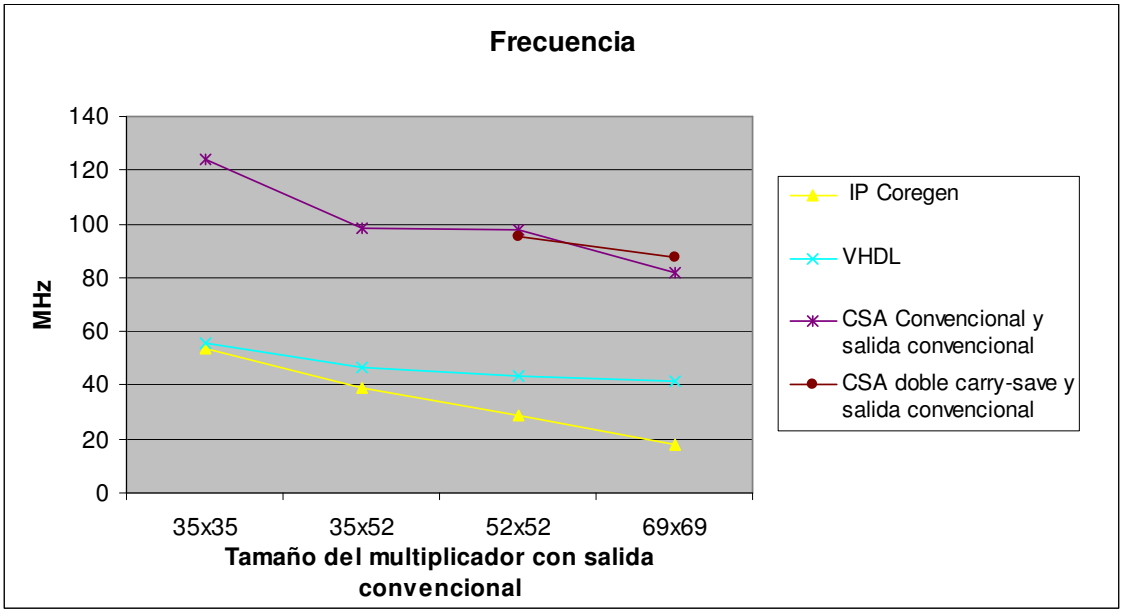

Gráfico 4. Frecuencia de trabajo de multiplicadores con salida CPA en LUT6s.

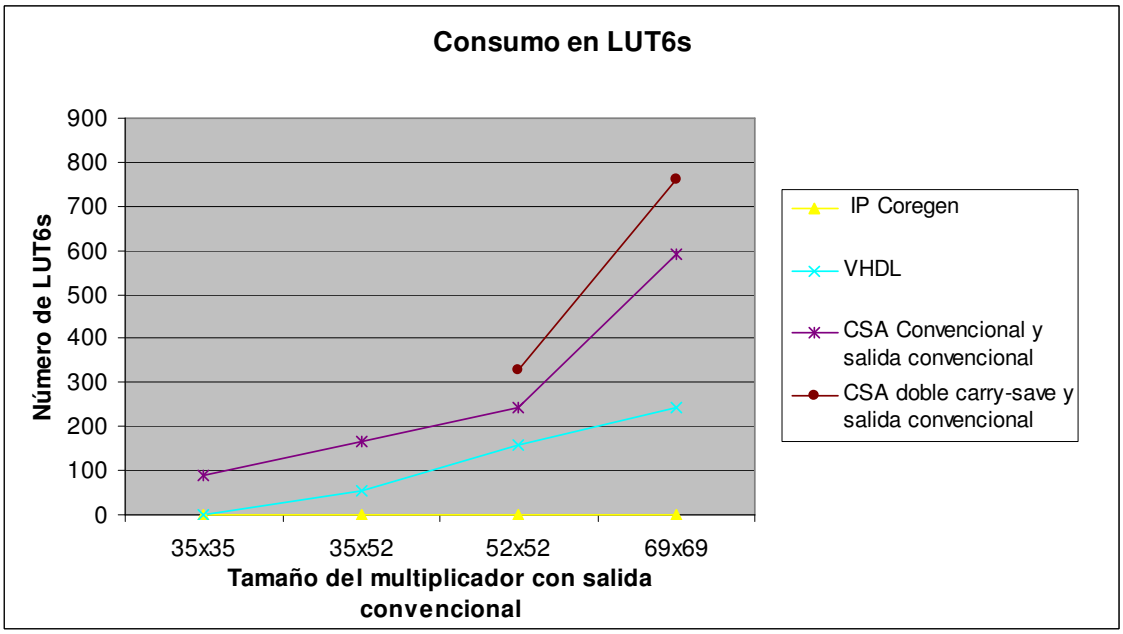

Gráfico 5. Área consumida de multiplicadores con salida CPA en LUT6s.

5. Conclusiones y líneas futuras

Actualmente las FPGAs han dejado de ser únicamente un conjunto de puertas lógicas, que se utilizaban como lógica para adaptar partes de un sistema a otro ("Glue logic"), o para implementar pequeños circuitos dedicados, para pasar a ser utilizadas como la mejor alternativa en el caso de sistemas con un elevado coste computacional.

A lo largo de la Tesis Doctoral demostramos que es posible la implementación eficiente de aritmética *carry-save* en FPGAs, en contra de lo que se pensaba hasta hace algunos años, y se corrigen algunas de las creencias anteriores, derivadas de dejar completamente la responsabilidad del mapeo de la aritmética *carry-save* a las herramientas de síntesis. Y por otro lado, hemos demostrado con los resultados de las implementaciones, que la manera más eficiente de crear multiplicadores paralelos de elevado ancho de palabra en FPGAs actualmente se consigue con la combinación de los multiplicadores empotrados con árboles de sumadores *carry-save* o doble *carry-save*.

Por tanto, es posible la utilización eficiente en FPGAs de aritmética redundante, incluso en FPGAs de bajo coste. Se consigue un aumento de velocidad importante, sin la necesidad de recurrir a FPGAs de alto rendimiento, de mayor precio y consumo de potencia, en muchas aplicaciones que hacen un uso intensivo de las operaciones de suma y multiplicación.

De manera detallada las conclusiones a las que se ha llegado con este trabajo se pueden enumerar de la siguiente manera:

1. Es posible la implementación eficiente de compresores [3:2] y [4:2] en FPGAs con LUT4 en contra de lo publicado hasta hace algunos años.

2. Hemos demostrado que en el caso de compresores de orden superior realizados componiendo compresores [3:2] y [4:2] en FPGAs con LUT4, se aumenta el rendimiento, si en las conexiones entre ellos se aprovecha para los caminos más críticos el menor retardo de propagación de la lógica de acarreo.

3. Hemos introducido los compresores con un nuevo formato doble *carry-save* para compresores en FPGAs con LUT6, con tiempos de

propagación menores que los compresores *carry-save* clásicos, y por tanto más eficientes en cuanto a velocidad.

4. Hemos diseñado un compresor [4:2] optimizado en velocidad para FPGAs con LUT6 a bajo nivel con un tiempo de propagación inferior a los diseñados anteriormente en otros trabajos, aprovechando la lógica de acarreo para generar parte de las funciones lógicas.

5. Hemos demostrado con síntesis reales, que la combinación de multiplicadores empotrados con aritmética *carry-save*, son la mejor alternativa para la realización de multiplicadores de ancho de palabra elevados en FPGAs. Esta combinación tanto para producir salida convencional como *carry-save* no ha sido propuesta hasta el momento de manera general.

Parte de los resultados obtenidos en este trabajo se presentaron en el congreso "Application-specific Systems, Architectures and Processors, 2009. ASAP 2009" con el trabajo titulado "Efficient implementation of carry-save adders in FPGAs", donde por primera vez se mostró como es posible implementar compresores [3:2] y [4:2] de una manera eficiente en FPGAs con LUT4, sin un derroche de recursos y en contra de lo publicado hasta el momento. Este trabajo respaldado por su aceptación en unos de los congresos más prestigiosos, fue el punto de partida de esta línea de investigación que está siendo utilizada en otras aplicaciones, demostrando que es posible utilizar aritmética redundante en operaciones más complejas de manera eficiente en FPGAs.

Fruto del estudio exhaustivo de los recursos lógicos de las FPGAs, se ha utilizado la lógica de propagación de acarreo para generar funciones generales, distintas a las puramente aritméticas, con una mejora en la velocidad, aunque sin ser el objetivo primario de la tesis.

En cuanto a las líneas de trabajo futuras caben destacar tres. Una de ellas se ha comenzado ya, y consiste en utilizar aritmética redundante para conseguir operaciones más complejas en FPGAs especialmente aquellas que permitan la utilización de la representación CS. Se ahorra el tiempo en el paso a representación convencional. Por ejemplo en operaciones MAC, SAD y en multiplicaciones serie para ancho de palabras muy elevado, como ocurre en criptografía.

La segunda línea tiene que ver con la utilización de la lógica de acarreo de las FPGAs para sintetizar funciones generales que requieren un mínimo retardo. Por ejemplo, las líneas de acarreo pueden verse por la rapidez, como un punto donde se "unen" las salidas de las *look- up tables* pudiéndose utilizar en aplicaciones clásicas del tipo de "and por conexión", por ejemplo para arbitrar buses o para realizar comparadores de manera eficiente. Con la utilización de la lógica de acarreo para crear comparadores hemos conseguido aumentar la velocidad y reducir el área en el desarrollo de nuestros aceleradores hardware.

La tercera línea de trabajo futura, es la continuación natural de este trabajo y tiene que ver con la automatización de árboles de compresores para suma multioperando y multiplicadores de ancho de palabra elevado. Con este trabajo se dispone de diseños optimizados a partir de los cuales se puede crear una librería básica de componentes.

6. Referencias

[1] B. Cope, P. Cheung,W. Luk, and L. Howes, “Performance comparison of graphics processors to reconfigurable logic: A case study”, IEEE Transactions on Computers, vol. 59, no. 4, pp. 433–448, Apr. 2010.

[2] Stephen Craven and Peter Athanas, "Examining the Viability of FPGA Supercomputing", EURASIP Journal on Embedded Systems 2007, doi:10.1155/2007/93652.

[3] Herbordt, M.C.; VanCourt, T.; Yongfeng Gu; Sukhwani, B.; Conti, A.; Model, J.; DiSabello, D., "Achieving High Performance with FPGA-Based Computing", Computer , vol.40, no.3, pp.50,57, March 2007.

[4] Xilinx Inc., www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf, September 10, 2010.

[5] Peripheral Component Interconnect Special Interest Group (PCI-SIG),” Conventional PCI 3.0 & 2.3: An Evolution of the Conventional PCI Local Bus Specification”. Avaliable: http://www.pcisig.com.

[6] Quiles-Latorre, F., Ortiz-Lopez, Manuel Agustin, Montijano-Vizcaino, M., Moreno-Moreno, C., Brox-Jiménez, M., hormigo-Aguilar, F., Villalba-Moreno,Julio, "Acelerador Hardware de bajo coste para bus PCI Convencional", Seminario Anual de Automática, Electrónica Industrial e Instrumentación 2012, Guimarães, Portugal.

[7] S. Dikmese, A. Kavak, K. Kucuk, S. Sahin, A. Tangel, and H. Dincer, “Digital signal processor against field programmable gate array implementations of space-code correlator beamformer for smart antennas,” IET Microwaves, Antennas Propagation, vol. 4, no. 5, pp. 593–599, May 2010.

[8] S. Roy and P. Banerjee, “An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design,” IEEE Transactions on Computers, vol. 54, no. 7, pp. 886–896, Jul. 2005.

[9] F. Schneider, A. Agarwal, Y. M. Yoo, T. Fukuoka, and Y. Kim,“A fully programmable computing architecture for medical ultrasound machines,” IEEE Transactions on Information Technology in Biomedicine, vol. 14, no. 2, pp. 538–540, Mar. 2010.

[10] J. Hill, “The soft-core discrete-time signal processor peripheral [applications corner],” IEEE Signal Processing Magazine, vol. 26, no. 2, pp. 112–115, Mar. 2009.

[11] J. S. Kim, L. Deng, P. Mangalagiri, K. Irick, K. Sobti, M. Kandemir, V. Narayanan, C. Chakrabarti, N. Pitsianis, and X. Sun, "An automated framework for accelerating numerical algorithms on reconfigurable platforms using algorithmic/architectural optimization," IEEE Transactions on Computers, vol. 58, no. 12, pp. 1654–1667, Dec. 2009.

[12] H. Lange and A. Koch, "Architectures and execution models for hardware/software compilation and their system-level realization," IEEE Transactions on Computers, vol. 59, no. 10, pp. 1363–1377, Oct. 2010.

[13] L. Zhuo and V. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," IEEE Transactions on Computers, vol. 57, no. 8, pp. 1057–1071, Aug. 2008.

[14] C. Mancillas-Lopez, D. Chakraborty, and F. Rodriguez Henriquez, "Reconfigurable hardware implementations of tweakable enciphering schemes," IEEE Transactions on Computers,, vol. 59, no. 11, pp. 1547–1561, Nov. 2010.

[15] T. Kasper, M. Novotny, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," IEEE Transactions on Computers, vol. 57, no. 11, pp. 1498–1513, Nov. 2008.

[16] M.D. Ercegovac and T. Lang, "Digital Arithmetic", Morgan Kaufmann Publishers, 2004.

[17] Parhami, B. Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, New York, NY, USA, 2000.

[18] Jhon P. Hayes, "Introducción al Diseño Lógico Digital", Addison-Wesley Iberoamericana, 1996.

[19] Vitoroulis, K.; Al-Khalili, A.J., "Performance of Parallel Prefix Adders implemented with FPGA technology," Circuits and Systems, 2007. NEWCAS 2007. IEEE Northeast Workshop on , vol., no., pp.498,501, 5-8 Aug. 2007.

[20] S. Gao, D. Al-Khalili, and N. Chabini, "Implementation of large size multipliers using ternary adders and higher order compressors," International Conference on Microelectronics (ICM), pp. 118–121, 2009.

[21] S. Gao, D. Al-Khalili, and N. Chabini, "FPGA Realization of High Performance Large Size Computational Functions: Multipliers and Applications," Analog Integrated Circuits and Signal Processing, vol. 70, no. 2, pp. 165-179, Feb. 2011.

[22] K. Macpherson and R. Stewart, "Rapid prototyping - area efficient FIR filters for high speed FPGA implementation," Vision, Image and Signal Processing, IEE Proceedings -, vol. 153, no. 6, pp. 711–720, dec. 2006.

[23] P. Meher, S. Chandrasekaran, and A. Amira, “FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic,” Signal Processing, IEEE Transactions on, vol. 56, no. 7, pp. 3009 –3017, july 2008.

[24] Z. Kincses, Z. Nagy, L. Orzo, P. Szolgay, and G. Mezo, “Implementation of a parallel SAD based wavefront sensor architecture on FPGA,” in Circuit Theory and Design, 2009. ECCTD 2009. European Conference on, aug. 2009, pp. 823 – 826.

[25] F. Bensaali, A. Amira, and A. Bouridane, “Accelerating matrix product on reconfigurable hardware for image processing applications,” Circuits, Devices and Systems, IEE Proceedings -, vol. 152, no. 3, pp. 236 – 246, june 2005.

[26] Y. F. Chan, M. Moallem, and W. Wang, “Design and implementation of modular FPGA-based PID controllers,” Industrial Electronics, IEEE Transactions on, vol. 54, no. 4, pp. 1898 –1906, aug. 2007.

[27] C. Villalpando, A. Morfopolous, L. Matthies, and S. Goldberg, “FPGA Implementation of Stereo Disparity with High Throughput for Mobility Applications,” Proc. IEEE Aerospace Conf., pp. 1-10, Mar. 2011.

[28] M. Ercegovac and T. Lang, “On-the-fly conversion of redundant into conventional representations”, Computers, IEEE Transactions on, vol. C-36, no. 7, pp. 895 –897, july 1987.

[29] V. Charoensiri and A. Surarerks, “On-the-fly conversion from signed-digit number system into complement representation,, in Communications and Information Technologies, 2006. ISCIT ’06. International Symposium on, 18 2006-sept. 20, pp. 1056 –1061.

[30] C. Wallace, “A suggestion for a fast multiplier,” IEEE Transactions on Electronic Computers, vol. EC-13, no. 2, pp. 14–17, 1964.

[31] L. Dadda, “Some schemes for parallel multipliers,” Alta Frequenza, vol. 34, no. 5, pp. 349–356, 1965.

[32] P. Kornerup, “Reviewing 4-to-2 adders for multi-operand addition,” The Journal of VLSI Signal Processing, vol. 40, pp. 143–152, 2005.

[33] Max Maxfield, "FPGA Instant Access", Ed. Newnes-Elsevier Inc.. 2008.

[34] Gina R. Smith, "FPGA 101. Everything you need to know to get started", Ed. Newnes. 2010.

[35] Xilinx Inc., www.xilinx.com.

[36] Altera Corporation, www.altera.com/devices.

[37] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203–215, 2007.

[38] Institute of Electrical and Electronics Enginers Inc., "IEEE Standard VHDL Language Reference Manual: IEEE Std 1076™-2002", Institute of Electrical and Electronics Enginers Inc, New York, May 2002

[39] IEEE Verilog Standardization Group's, "IEEE P1364-2005: IEEE Standard for Verilog® Hardware Description Language ", IEEE, New York, April 2006.

[40] Xilinx Inc., "ABEL Reference Guide", www.xilinx.com, 2008.

[41] Xilinx Inc., “Spartan-3 Generation FPGA User Guide. Extended Spartan-3A, Spartan-3E, and Spartan-3 FPGA Families UG331 (v1.8)", www.xilinx.com/support/documentation, june 13, 2011.

[42] Xilinx Inc., "Spartan-6 Family Overview. DS160 (v2.0)", october 25, 2011.

[43] Xilinx Inc., "Spartan-6 FPGA Configurable Logic Block. User Guide UG384 (v1.1)", www.xilinx.com/support/documentation, february 23, 2010.

[44] M. Ortiz, F. Quiles, J. Hormigo, F. Jaime, J. Villalba, and E. Zapata, “Efficient implementation of carry-save adders in FPGAs”, in Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, 7-9 2009, pp. 207 –210.

[45] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Exploiting fast carry-chains of FPGAs for designing compressor trees”, International Conference on Field Programmable Logic and Applications (FPL), pp. 242–249, aug. 2009.

[46] R. Gutierrez, J. Valls, and A. Perez-Pascual, “FPGA implementation of time-multiplexed multiple constant multiplication based on carry-save arithmetic”, 19th International Conference on Field Programmable Logic and Applications (FPL), pp. 609–612, 2009.

[47] Xilinx Inc., "ISE Design Suite", www.xilinx.com/products/design-tools/ise-design-suite.

[48] Mentor graphics Inc., " ModelSim ASIC and FPGA design", www.mentor.com/products/fv/modelsim.

[49] H. Parandeh-Afshar, P. Brisk, and P. Ienne, ”Efficient synthesis of compressor trees on FPGAs,” in Proceedings of the Asia-South Pacific Design Automation Conf., Seoul, Korea, January, 2008, pp. 138- 43.

[50] J.-L. Beuchat and J.-M. Muller, “Automatic generation of modular multipliers for fpga applications”, IEEE Transactions on Computers, vol. 57, no. 12, pp. 1600–1613, December 2008.

[51] Moreno, C. D.; Quiles, F.J.; Ortiz, M. A.; Brox, M.; Hormigo, J.; Villalba, J.; Zapata, E.L., "Efficient mapping on FPGA of convolution computation based on combined CSA-CPA accumulator," Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on , vol., no., pp.419,422, 13-16 Dec. 2009, doi: 10.1109/ICECS.2009.5410903.

[52] P.L. Montgomery, "Modular Multiplication without Trial Division," Mathematics of Computation, Vol. 44, No. 170, Apr. 1985, pp. 519-521.

[53] K. Manochehri and S. Pourmozafari, "Modified radix-2 montgomery modular multiplication to make it faster and simpler," in IEEE International Conference on Information Technology: Coding and Computing, ITCC 2005, April 2005.

[54] Ming-Der Shieh; Jun-Hong Chen; Wen-Ching Lin; Hao-Hsuan Wu, "A New Algorithm for High-Speed Modular Multiplication Design," Circuits and Systems I: Regular Papers, IEEE Transactions on , vol.56, no.9, pp.2009,2019, Sept. 2009, doi: 10.1109/TCSI.2008.2011585.

[55] Sutter, G.; Deschamps, J.; Iman~a, J.L., "Efficient FPGA Modular Multiplication and Exponentiation Architectures Using Digit Serial Computation," Field Programmable Logic and Applications (FPL), 2010 International Conference on , vol., no., pp.496,501, Aug. 31 2010-Sept. 2 2010, doi: 10.1109/FPL.2010.99.

[56] Tao Wu; Shuguo Li; Litian Liu, "CSA-based design of feedforward scalable montgomery modular multiplier," Signal Processing and Information Technology (ISSPIT), 2011 IEEE International Symposium on , vol., no., pp.054,059, 14-17 Dec. 2011, doi: 10.1109/ISSPIT.2011.6151535.

[57] Verma, A.; Verma, A.K.; Parandeh-Afshar, H.; Brisk, P.; Ienne, P., "Synthesis of Floating-Point Addition Clusters on FPGAs Using Carry-Save Arithmetic," Field Programmable Logic and Applications (FPL), 2010 International Conference on , vol., no., pp.19,24, Aug. 31 2010-Sept. 2 2010, doi: 10.1109/FPL.2010.15.

[58] G. Cardarilli, S. Pontarelli, M. Re, and A. Salsano, "On the use of signed digit arithmetic for the new 6-inputs LUT based FPGAs," 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp. 602–605, 2008.

[59] W. Kamp, A. Bainbridge-Smith, and M. Hayes, "Efficient implementation of fast redundant number adders for long wordlengths in FPGAs," International Conference on Field-Programmable Technology (FPT'09), pp. 239–246, 2009.

[60] Kamboh, H.M.; Khan, S.A., "FPGA implementation of fast adder," Computing and Convergence Technology (ICCCT), 2012 7th International Conference on , vol., no., pp.1324,1327, 3-5 Dec. 2012.

[61] S.-F. Hsiao, M.-R. Jiang, and J.-S. Yeh, “Design of high-speed low power 3-2 counter and 4-2 compressor for fast multipliers,” IEE Electronics Letters, vol. 34, no. 4, pp. 341–343, Feb. 1998.

[62] Stelling, P. F., Martel, C. U., Oklobdzija, V. G., and Ravi, R., “Optimal circuits for parallel multipliers”, IEEE Transactions on Computers C-47, 3 (Mar. 1998), 273–85.

[63] Um, J., and Kim, T. ,“An optimal allocation of carry-save-adders in arithmetic circuits”, IEEE Transactions on Computers C-50, 3 (Mar. 2001), 215–33.

[64] Hormigo, J.; Villalba, J.; Zapata, E.L., "Multioperand Redundant Adders on FPGAs," Computers, IEEE Transactions on , vol.62, no.10, pp.2013,2025, Oct. 2013.doi: 10.1109/TC.2012.

[65] W. J. Stenzel,W. J. Kubitz, and G. H. Garcia, “Compact high-speed parallel multiplication scheme,” IEEE Transactions on Computers, vol. C-26, no. 10, pp. 948–957, 1977.

[66] S. Dormido and M. Canto, “Synthesis of generalized parallel counters,” IEEE Transactions on Computers, vol. C-30, no. 9, pp. 699–703, Sep. 1981.

[67] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Efficient synthesis of compressor trees on FPGAs”, in Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific, 2008, pp. 138 –143.

[68] H. Parandeh-Afshar, P. Brisk, and P. Ienne, “Improving synthesis of compressor trees on FPGAs via integer linear programming,” International Conference on Design, Automation and Test in Europe (DATE’08), pp. 1256–1261, 2008.

[69] H. Parandeh-Afshar, A. Verma, P. Brisk, and P. Ienne, “Improving FPGA performance for carry-save arithmetic,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 4, pp. 578–590, apr. 2010.

[70] T. Matsunaga, S. Kimura, and Y. Matsunaga, “Multi-operand adder synthesis on FPGAs using generalized parallel counters”, in Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 2010, pp. 337–342.

[71] Verma, A. K., y Ienne, P., “Automatic synthesis of compressor trees: Reevaluating large counters”, In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (Nice, France, Apr. 2007), pp. 443–48.

[72] P. Brisk, A. K. Verma, P. Ienne, and H. Parandeh-Afshar, “Enhancing FPGA performance for arithmetic circuits,” in Proc. Des. Autom. Conf., San Diego, CA, Jun. 2007, pp. 404–409.

[73] A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. Verma, P. Brisk, F. Gurkaynak, Y. Leblebici, and P. Ienne, "Architectural improvements for field programmable counter arrays: Enabling efficient synthesis of fast compressor trees on FPGAs," ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays, pp. 181–190, 2008.

[74] Parandeh-Afshar, H.; Cevrero, A.; Athanasopoulos, P.; Brisk, P.; Leblebici, Y.; Ienne, P., "A flexible DSP block to enhance FPGA arithmetic performance," Field-Programmable Technology, 2009. FPT 2009. International Conference on , vol., no., pp.70,77, 9-11 Dec. 2009, doi: 10.1109/FPT.2009.5377638.

[75] A. Cevrero, P. Athanasopoulos, H. Parandeh-Afshar, A. K. Verma, H. S. A. Niaki, C. Nicopoulos, F. K. Gurkaynak, P. Brisk, Y. Leblebici, and P. Ienne, "Field programmable compressor trees: Acceleration of multi-input addition on FPGAs," ACM Trans. On Reconfigurable Technology Systems, vol. 2, pp. 13:1–13:36, Jun 2009.

[76] Seyed Hosein Attarzadeh Niaki, Alessandro Cevrero, Philip Brisk, Chrysostomos Nicopoulos, Frank K. Gurkaynak, Yusuf Leblebici and Paolo Ienne, "Design Space Exploration for Field Programmable Compressor Trees", CASES08, Atlanta, October 2008.

[77] Renukuntla Kiran and Sunitha Nampally, "Analyzing the performance of carry tree adders based on FPGA's", International Journal of Electronics Signals and Systems (IJESS) ISSN: 2231- 5969, Vol-2, ISS-2,3,4, 2012.

[78] Xilinx Inc., "Spartan-3 FPGA Family Data Sheet. DS099", www.xilinx.com/support/documentation, june 27, 2013.

[79] J. Detrey, F. de Dinechin, and X. Pujol, "Return of the hardware floating point elementary function," in Proceedings of the 18th IEEE Symposium on Computer Arithmetic (Montpellier, France), Kornerup and Muller, Eds. Los Alamitos, CA: IEEE Computer Society Press, June 2007, pp. 161– 168.

[80] Xilinx Inc., "Virtex-6 FPGA Configurable Logic Block. User Guide UG364 (v1.2)", www.xilinx.com/support/documentation, february 3, 2012.

[81] Ortiz-Lopez, M., Quiles-Latorre, F., Moreno-Moreno, C., Brox-Jiménez, M., "CAN2PCI: Placa con interfaz al bus CAN y PCI con finalidad docente", Revista de Formación universitaria, ISSN: 0718-5006, pp.31-38, Chile 2009, doi: 10.4067/S0718-50062009000300006.

[82] Ortiz-Lopez, M., Quiles-Latorre, F., Moreno-Moreno, C., Brox-Jiménez, M.," ADQPCI: placa de adquisición de datos con fines docentes", Revista de Formación universitaria, ISSN: 0718-5006, pp. 25-30,Chile 2009, doi: 10.4067/S0718-50062009000300005.

[83] Quiles, F.J.; Ortiz, M.; Brox, M.; Moreno, C.D.; Hormigo, J.; Villalba, J., "UCORE: Reconfigurable Platform for Educational Purposes," Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on , vol., no., pp.109,114, 13-15 Dec. 2010, doi: 10.1109/ReConFig.2010.60.

[84] M. Frederick and A. Somani, "Multi-bit carry chains for high performance reconfigurable fabrics," Int. Conf. on Field Programmable Logic and Applications, pp. 1–6, 2006.

[85] M. T. Frederick and A. K. Somani, "Beyond the arithmetic constraint: depth-optimal mapping of logic chains in LUTbased FPGAs," in 16th International ACM/SIGDA Symposium on Field-Programmable Gate Arrays (FPGA). New York, NY, USA: ACM, 2008, pp. 37–46.

[86] Preusser, T.B.; Spallek, R.G., "Enhancing FPGA Device Capabilities by the Automatic Logic Mapping to Additive Carry Chains," Field Programmable Logic and Applications (FPL), 2010 International Conference on , vol., no., pp.318,325, Aug. 31 2010-Sept. 2 2010, doi: 0.1109/FPL.2010.70.

[87] T. B. Preußer and R. G. Spallek, "Mapping basic prefix computations to fast carry-chain structures," in International Conference on Field Programmable Logic and Applications (FPL), IEEE, Ed., Aug. 2009.

[88] Xilinx Inc., "Achieving higher system performance with the virtex-5 family of FPGAs, WP245", www.xilinx.com/support/documentation, 2006.

[89] Xilinx Inc., "Ug369 virtex-6 FPGA dsp48e1 slice, user guide," www.xilinx.com/support/documentation, 2009.

[90] C. Tsen, S. Gonzalez-Navarro, M. Schulte, B. Hickmann, and K. Compton, "A combined decimal and binary floating-point multiplier", in Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors, 2009, pp. 8–15.

[91] Lang, T.; Bruguera, J.D., "Floating-point multiply-add-fused with reduced latency," Computers, IEEE Transactions on , vol.53, no.8, pp.988,1003, Aug. 2004. doi: 10.1109/TC.2004.44

[92] J. Bruguera and T. Lang, "Floating-point fused multiplyadd: reduced latency for floating-point addition", in Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on, june 2005, pp. 42 – 51.

[93] Y.-H. Seo and D.-W. Kim, "A new vlsi architecture of parallel multiplier-accumulator based on radix-2 modified booth algorithm, " Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 18, no. 2, pp. 201 –208, feb. 2010.

[94] Moreno-Moreno, C., Martinez-Jiménez, M., Bellido-Outeiriño, F., Hormigo-Aguilar, F., Ortiz-Lopez, M., Quiles-Latorre, F., "Convolution computation in FPGA based on carry-save adders and circular buffers", International ICST Conference on IT Revolutions, Cordoba-Spain, 2011.doi:10.1007/978-3-642-32304-1_20.

[95] C. Moses, D. Selvathi, and S. Rani, “Fpga implementation of an efficient partial volume interpolation for medical image registration”, in Communication Control and Computing Technologies (ICCCCT), 2010 IEEE International Conference on, oct. 2010, pp. 132 –137.

[96] Walters, E. III, Arnold, M.G., and Schulte, M.J.: “Using truncated multipliers in DCT and IDCT hardware accelerators”. Proc. SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations XIII, San Diego, California, August 2003, pp. 573–584.

[97] P. Brisk, A. Verma, P. Ienne, and H. Parandeh-Afshar, “Enhancing FPGA performance for arithmetic circuits,” 44th ACM/IEEE Design Automation Conference (DAC’07), pp. 334–337, 2007.

[98] H. Eberle, N. Gura, S. Shantz, V. Gupta, L. Rarick, and S. Sundaram, “A public-key cryptographic processor for rsa and ecc”, in Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on, sept. 2004, pp. 98 – 110.

[99] T. Wu, S. Li, and L. Liu, “Csa-based design of feedforward scalable montgomery modular multiplier”, in Signal Processing and Information Technology (ISSPIT), 2011 IEEE International Symposium on, dec. 2011, pp. 054 –059.

[100] Xilinx, “Ug193 virtex-5 xtremedsp design considerations, user guide”, www.xilinx.com/support/documentation, 2007.

[101] Parandeh-Afshar, H.; Ienne, P., "Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs," Field Programmable Logic and Applications (FPL), 2011 International Conference on , vol., no., pp.225,231, 5-7 Sept. 2011, doi: 10.1109/FPL.2011.48.

[102] McIvor, C.; McLoone, M.; McCanny, J.V., "FPGA Montgomery multiplier architectures - a comparison," Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on , vol., no., pp.279,282, 20-23 April 2004, doi: 10.1109/FCCM.2004.36.

[103] de Dinechin, F.; Joldes, M.; Pasca, B.; Revy, G., "Multiplicative Square Root Algorithms for FPGAs," Field Programmable Logic and Applications (FPL), 2010 International Conference on , vol., no., pp.574,577, Aug. 31 2010-Sept. 2 2010, doi: 10.1109/FPL.2010.112.

[104] Nadjia, A.; Mohamed, A.; Hamid, B.; Mohamed, I.; Khadidja, M., "Hardware algorithm for variable precision multiplication on FPGA," Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on , vol., no., pp.845,848, 10-13 May 2009, doi: 10.1109/AICCSA.2009.5069427.

[105] A. Karatsuba, Ofman Yu, "Multiplication of multiple numbers by mean of automata", Dokadly Akad. Nauk SSSR 145, no 2, pp.293-294,1962.

[106] M.Machhout, M.Zeghid, W.El hadj youssef, B.Bouallegue, A.Baganne, and R.Tourki, "Efficient Large Numbers Karatsuba- Ofman Multiplier Designs for Embedded Systems," International Journal of Electronics, Circuits and Systems, 2009, pp. 1-10.

[107] Shuli Gao; Chabini, N.; Al-Khalili, D., "Dynamic Programming Addition Optimization approach for large size multipliers in FPGAs," Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on , vol., no., pp.521,524, 1-4 Aug. 2010, doi: 10.1109/MWSCAS.2010.5548744.

[108] Shuli Gao; Al-Khalili, D.; Chabini, N., "Efficient techniques for realizing large-size signed multipliers in FPGAs," Microelectronics, 2008. ICM 2008. International Conference on , vol., no., pp.1,4, 14-17 Dec. 2008. doi: 10.1109/ICM.2008.5393833.

[109] Gao, S.; Chabini, N.; Al-Khalili, D.; Langlois, P., "Optimised realisations of large integer multipliers and squarers using embedded block," Computers & Digital Techniques, IET , vol.1, no.1, pp.9,16, January 2007. doi: 10.1049/iet-cdt:20060074.

[110] Shuli Gao; Al-Khalili, D.; Chabini, N., "Asymmetric large size multiplication using embedded blocks with efficient compression technique in FPGAs," Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on , vol., no., pp.137,140, 11-14 Dec. 2011. doi: 10.1109/ICECS.2011.6122233.

[111] J. Athow and A. Al-Khalili, “Implementation of large-integer hardware multiplier in Xilinx FPGA”, in Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on, 31 2008-sept. 3 2008, pp. 1300 – 1303.

[112] M. Schulte and E. Swartzlander Jr., “A family of variableprecision interval arithmetic processors”, IEEE Transactions on Computers, vol. 49, no. 5, pp. 387–397, 2000.

[113] Xilinx Inc., “Ug331 spartan-3 generation fpga, user guide”, www.xilinx.com/support/documentation, 2011.

[114] S. Gao, N. Chabini, D. Al-Khalili, and P. Langlois, “Optimized multipliers for large unsigned integers”, in 23rd NORCHIP Conference 2005, 2005.

[115] S. Gao, D. Al-Khalili, and N. Chabini, “Efficient realization of large size two’s complement multipliers using embedded blocks in FPGAs”, Circuits, Systems, and Signal Processing, vol. 27, no. 5, pp. 713–731, 2008.

[116] F. De Dinechin and B. Pasca, “Large multipliers with fewer dsp blocks”, in Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, 31 2009-sept. 2 2009, pp. 250–255.

[117] S. Gao, D. Al-Khalili, and N. Chabini, “Asymmetric large size signed multipliers using embedded blocks in FPGAs”, in Parallel and Distributed Processing Workshops and PhdForum (IPDPSW), 2011 IEEE International Symposium on, may 2011, pp. 271 –277.

[118] Xilinx INC. , " Xilinx CORE Generator System", www.xilinx.com/tools/coregen.htm.

Anexo de publicaciones

Trabajos presentados

Directamente relacionados con lo expuesto en este trabajo se han presentado resultados a varios congresos. Estamos preparando para en este momento dos trabajos para difundir los resultados obtenidos acerca de la síntesis de compresores en LUT6 y síntesis de multiplicadores de ancho de palabra elevado tanto en LUT4 como LUT6.

Queremos destacar el trabajo presentado al Congreso "Application-specific Systems, Architectures and Processors, 2009. ASAP 2009", que supuso el punto de partida de la línea de estudio de implementación eficiente de aritmética CSA en FPGAs. Este congreso es el que ocupa el primer lugar en los "Conference Rankings" en cuanto a síntesis en FPGAs de aritmética. Este trabajo se ha citado en otros trabajos que hablan de la situación actual de la síntesis de compresores en FPGAs con LUT4:

1- **M. Ortiz**, F. Quiles, J. Hormigo, F. Jaime, J. Villalba, and E. Zapata, "Efficient implementation of carry-save adders in FPGAs", in Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, 7-9 2009, pp. 207 –210.

Otro trabajo presentado al congreso "Seminario Anual de Automática, Electrónica Industrial e Instrumentación 2012. SAEEI 2012" de ámbito internacional, implementa también aritmética redundante en este caso en los interfaces del acelerador:

2- Quiles-Latorre, F., **Ortiz-Lopez, M.**, Montijano-Vizcaino, M., Moreno-Moreno, C., Brox-Jiménez, M., Hormigo-Aguilar, F., Villalba-Moreno, Julio, "Acelerador Hardware de bajo coste para bus PCI Convencional", Seminario Anual de Automática, Electrónica Industrial e Instrumentación 2012, Guimarães, Portugal.

Otros trabajos que se han presentado están relacionados con la utilización de compresores sintetizados eficientemente en FPGAs en algunas aplicaciones:

3- Moreno, C. D.; Quiles, F.J.; **Ortiz, M. A**.; Brox, M.; Hormigo, J.; Villalba, J.; Zapata, E.L., "Efficient mapping on FPGA of convolution computation based on combined CSA-CPA accumulator," Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on , vol., no., pp.419,422, 13-16 Dec. 2009, doi: 10.1109/ICECS.2009.5410903.

4- **Ortiz-Lopez, M.**, Quiles-Latorre, F., Moreno-Moreno, C., Brox-Jiménez, M., "CAN2PCI: Placa con interfaz al bus CAN y PCI con finalidad docente", Revista de Formación universitaria, ISSN: 0718-5006, pp.31-38, Chile 2009, doi: 10.4067/S0718-50062009000300006.

5- **Ortiz-Lopez, M**., Quiles-Latorre, F., Moreno-Moreno, C., Brox-Jiménez, M.," ADQPCI: placa de adquisición de datos con fines docentes", Revista de Formación universitaria, ISSN: 0718-5006, pp. 25-30, Chile 2009, doi: 10.4067/S0718-50062009000300005.

6- Quiles, F.J.; **Ortiz, M**.; Brox, M.; Moreno, C.D.; Hormigo, J.; Villalba, J., "UCORE: Reconfigurable Platform for Educational Purposes," Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on , vol., no., pp.109,114, 13-15 Dec. 2010, doi: 10.1109/ReConFig.2010.60.

7- Moreno-Moreno, C., Martinez-Jiménez, M., Bellido-Outeiriño, F., Hormigo-Aguilar, F., **Ortiz-Lopez, M**., Quiles-Latorre, F., "Convolution computation in FPGA based on carry-save adders and circular buffers", International ICST Conference on IT Revolutions, Cordoba-Spain, 2011.doi:10.1007/978-3-642-32304-1_20.

2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors

Efficient implementation of carry-save adders in FPGAs

Manuel Ortiz, Francisco Quiles
Dept. Computer Architecture
University of Cordoba
Cordoba, Spain
{el1orlom,el1qulaf}@uco.es

Javier Hormigo, Francisco J. Jaime, Julio Villalba, Emilio L. Zapata
Dept. Computer Architecture
University of Malaga
Malaga, Spain
{hormigo,franj,julio,ezapata}@ac.uma.es

***Abstract*—Most Field Programmable Gate Array (FPGA) devices have a special fast carry propagation logic intended to optimize addition operations. The redundant adders do not easily fit into this specialized carry-logic and, consequently, they require double hardware resources than carry propagate adders, while showing a similar delay for small size operands. Therefore, carry-save adders are not usually implemented on FPGA devices, although they are very useful in ASIC implementations.**

In this paper we study efficient implementations of carry-save adders on FPGA devices, taking advantage of the specialized carry-logic. We show that it is possible to implement redundant adders with a hardware cost close to that of a carry propagate adder. Specifically, for 16 bits and bigger wordlengths, redundant adders are clearly faster and have an area requirement similar to carry propagate adders. Among all the redundant adders studied, the 4:2 compressor is the fastest one, presents the best exploitation of the logic resources within FPGA slices and the easiest way to adapt classical algorithms to efficiently fit FPGA resources.

I. Introduction

In despite of specific purpose ASIC designs always show a better performance regarding to area and time than FPGA designs, the use of FPGA devices has extended in the hardware design community in the last years due to its use facility, flexibility, liability, low cost and short development time. An FPGA device has a special inner structure that tries to cover the most general case of designs. Basically, an FPGA is structured as a grid of small elements which are able to implement basic logic operations and store resources, together with routes for interconnecting these elements. Besides, some other hardware resources have been recently added in order to accelerate some specific operations. However, most current hardware-oriented algorithms are intended to be implemented on ASIC-based chips, thus they do not take into account FPGAs especial configuration. Because of this, a big amount of hardware resources from within FPGAs are wasted in many designs. Due to this fact, recently there is an increasing interest of the scientific community to design new algorithms which take advantage of the special FPGA inner architecture [1].

Addition is one of the most usual operations in any design. The architecture of modern FPGA devices use to have a special hardware in charge of dealing with addition, which is mainly focused on improving the performance of carry propagate adders (CPA). More specifically, the path for carry propagation has been specially optimized so that it goes from one basic element to the next one using a specific fast route, together with some specific carry-logic to add and propagate the carry value. For this reason, carry propagated adders are preferred to carry-save adders (CSA) for implementation on FPGA devices, since, for non very long wordlengths, CSAs have similar delays to CPAs, but double the number of logic resources [2]. Nevertheless, we think that this is due to the fact that the software tool does not efficiently manage the system resources when mapping the carry-save adders into an FPGA platform.

In this paper we prove that it is possible to implement carry-save adders on FPGA devices with a similar hardware cost to that of carry-propagate adders, while keeping a constant computation time (the computation time increases with the operands wordlength if we use the inner carry propagate adder within the FPGA), in such a way that considering operands with number of bits greater or equal to 16, the speed gain is notorious (similar to what happens in an ASIC-based design). Consequently, many current FPGA based applications that use additions can be improved if the techniques proposed in this paper are applied on them [3]–[5].

II. Carry save adders on FPGA

This paper focuses on the inner architecture of FPGAs with four–input LUTs and specialized carry-logic like Virtex 2, 4 and Spartan 2, 3 of Xilinx. In spite of the new generation of FPGAs having a different inner architecture, FPGAs with four-input LUTs are currently the most widely used devices for medium complexity applications due to its low price and low consumption.

Fig. 1 describes a simplified architecture of a slice implementing a CPA. Each slice includes two four-input generators (LUTs), two flip-flops, the specialized carry-logic and the necessary logic and multiplexers. These elements are combined as shown in the figure to operate like a CPA: the lower slice produces a carry bit (c_{i+1}) and a sum bit (s_i) from three input bits x_i, y_i, c_i. The carry bit c_{i+1} is then passed to the upper slice using the carry propagation logic, where it will be added with x_{i+1} and y_{i+1}, producing the next sum and carry bits, s_{i+1} and c_{i+2}. Thus, each slice allocates the full addition of two pairs of bits.

1063-6862/09 $25.00 © 2009 IEEE
DOI 10.1109/ASAP.2009.22

Acelerador Hardware de bajo coste para bus PCI Convencional

Francisco J. Quiles, Manuel Ortiz, Miguel A. Montijano, Carlos D. Moreno, Maria Brox
Dept. Arquitectura de Computadores
Universidad de Córdoba
Campus Universitario de Rabanales
14071 Córdoba. Spain
email: el1orlom@uco.es

Javier Hormigo, Julio Villalba
Dept. Arquitectura de Computadores
Universidad de Málaga
Campus de Teatinos
29080 Málaga. Spain
email: hormigo@ac.uma.es

Resumen—**En este trabajo se presenta un acelerador hardware con programación dinámica de bajo coste para el bus PCI convencional. El acelerador está formada por dos partes: un interfaz al bus PCI y una FPGA para la implementación del algoritmo del acelerador. La simplicidad de este acelerador hardware permite que el diseñador dedique su esfuerzo al algoritmo que implementará en la FPGA mientras que el esfuerzo que dedica al interfaz PCI es mínimo ya que el interfaz al bus PCI se ha reducido al máximo. El bridge con el bus PCI convencional está implementado en un core IP, que permite la programación dinámica de la FPGA e implementa facilidades para el control de este recurso por el sistema operativo.**

Palabras clave: acelerador hardware; FPGA; PCI Convencional; computación de alto rendimiento.

I. INTRODUCCION

En numerosas áreas de aplicación, especialmente en bioinformática, tratamiento digital de imagen y aplicaciones con alta precisión aritmética se requiere una alta demanda de procesamiento computacional (High Performance Computing). Esta necesidad está haciendo que emerjan nuevas tecnologías que exploten el enorme potencial de las FPGAs, gracias al paralelismo inherente en el hardware [1], a la vez que minimicen el esfuerzo requerido para llevar los algoritmos secuenciales a algoritmos paralelos e implementarlos en una FPGA.

Muchos trabajos de investigación están relacionados con la optimización de recursos en FPGAs y es en este campo donde se centra el trabajo de nuestro grupo. La optimización de recursos en FPGAs conduce a un menor coste económico y consumo de potencia del sistema. En este sentido los aceleradores hardware para HPC programables dinámicamente permiten cambiar el algoritmo programado en la FPGA, eliminando la necesidad de tenerlos previamente implementados y por tanto se pueden utilizar FPGAs de menor tamaño. Por otro lado, otro objetivo que se persigue es conseguir que nuestros alumnos desarrollen prácticas y proyectos de alta tecnología donde se simplifica al máximo el sistema.

El resto del trabajo está organizado de la siguiente manera: en el capítulo 2 se presenta brevemente la plataforma UCORE de computación reconfigurable, desarrollada para utilización docente y de investigación. En el capítulo 3 se muestra la arquitectura general del acelerador hardware y en el capítulo 4 se describe una de las placas que se ha desarrollado como componente hardware del acelerador. Finalmente en el capítulo 5, se mostrará la aplicación que maneja la configuración y programación dinámica del acelerador de forma interactiva.

II. PLATAFORMA PARA COMPUTACION RECONFIGURABLE UCORE

El acelerador hardware de bajo coste que se va a describir en este trabajo forma parte de la plataforma UCORE de computación reconfigurable que se presenta en [2]. La plataforma de computación reconfigurable UCORE se diseñó con propósitos educacionales y se está utilizando también como plataforma de prototipado rápido en trabajos de investigación. Los elementos que forman parte de la plataforma UCORE se muestran en la Figura 1, y consta de:

- La placa de prototipado rápido UCOS3E5 que contiene dos dispositivos programables, un CPLD y una FPGA, y varios periféricos e interfaces más comunes. El CPLD realiza el interfaz al bus PCI convencional [3] y está conectado con la FPGA.
- IPs cores: El usuario dispone actualmente de varios IP cores para utilización con la placa de rápido prototipado o para cualquier otro sistema. Cabe destacar tres IP cores optimizados para el CPLD que actúa de puente entre el bus PCI convencional y la FPGA.
- Un analizador lógico virtual sencillo optimizado en la utilización de recursos (en desarrollo) que se integra en el sistema reconfigurable para su depuración. El analizador lógico tiene dos partes: un IP core que se integra con el diseño a depurar y extrae los datos del sistema y una aplicación en el ordenador personal que configura la adquisición y muestra los datos.
- La aplicación, *PCI Bus Handler*, que permite el acceso al espacio de configuración de cada uno de los dispositivos PCI. En el caso de nuestro sistema

Efficient Mapping on FPGA of Convolution Computation based on Combined CSA-CPA Accumulator

C.D. Moreno, F. J. Quiles, M.A. Ortiz, M. Brox
Department of Computer Architecture
University of Cordoba, SPAIN
Email: el1orlom,el1qulaf@uco.es

J. Hormigo, J. Villalba, E.L. Zapata
Department of Computer Architecture
University of Malaga, SPAIN
Email: hormigo,julio@ac.uma.es

***Abstract*— In this paper we present some architectures to deal with fast convolution computation based on carry save adders which are intended to be specifically implemented on FPGAs. Carry–save adders are not frequent in FPGA implementations since FPGA has a fast carry propagation path. In this paper we prove that it is possible to use carry-save arithmetic in a efficient way on FPGA for convolution operation. We make use of the specific structure of the FPGA to design an optimized accumulator which is able to deal with carry-save additions as well as carry-propagate additions using the same hardware. This lead to an efficient combined CSA-CPA architecture with fast computation and optimizing the hardware cost. Experimental results for different word lengths are presented to validate our proposal.**

I. Introduction

The convolution function is the key of a lot of digital signal processing applications (filtering, fft, correlation, neuronal networks ...). It is based on multiplication and accumulation operations. Due to its fundamental role in DSP applications, many high performance FPGA devices have incorporated special cells for DSP which deal with these operations. Nevertheless, it is not found in low cost FPGA devices.

To reduce the cost of the multiplication involved in applications with constant terms, a constant multiplier or distributed arithmetic is usually implemented. Nevertheless, in other applications the multiplication can not be avoided (adaptive filters, correlation, neuronal networks, ...). This has taken to manufacturers to use embedded multipliers to perform this kind of operations efficiently.

Convolution involves a lot of multiplications and accumulations. If an FPGA device has few multipliers, it is frequent to use them in an iterative way to implement the function. To increase the throughput it is necessary to reduce the cycle time specially when there are a lot of operations. Since the multiplier size is fixed in a FPGA device and the accumulator is customized by the user, a good design of the accumulator can lead to better performance of the full operation.

Many authors have implemented different design to carry out the convolution operation. Modern FPGA devices include fast carry-logic which allows the implementation of fast carry propagate adders. Thus, these designs use carry propagate adders [1], [2], [3]. In this paper we propose the use of carry-free adders as an efficient alternative which reduces the computation time of the convolution. It is based on a low level assignment of inner resources of the slices, which gives an optimized performance in comparison with automatic assignment [4].

On the other hand, recently there is an increasing interest of the scientific community to design new algorithms which take advantage of the special FPGA inner architecture [5].

In this paper we present an iterative architecture for convolution computation which is faster than previous implementations. It is achieved by using redundant arithmetic. In spite of redundant arithmetic involves an increase of hardware, we have developed a technique which allows us to reuse the hardware resources to obtain the final result in conventional arithmetic.

II. Architecture for Convolution

The architecture that we propose for convolution is based on an iterative use of the embedded multipliers presented in most of modern FPGA devices. In Fig. 1 a typical architecture for iterative multiplication and accumulation for convolution of two vectors is presented, where the operands are n-bits wide and the accumulator has $2n + i$ bits (we use i bits to prevent overflow in the additions). One value of each vector is introduced to the multiplier and its result is accumulated with the previous partial result on each iteration. Thus, the basic cycle time is $T_{cycle} = T_{mult} + T_{acc}$ and the time required for one computation is $T_{total} = N * T_{cycle}$ where N is the number of elements of the vectors.

To reduce the total computation time (T_{total}) we have to reduce the cycle time T_{cycle}. Multipliers are embedded (prefixed by the manufacturers) and can be pipelined to reduce its effective cycle time, whereas the accumulator can not be pipelined due to true dependence problems. To reduce the effective time of the accumulator we propose the use of carry–save adders (CSA). This arithmetic is specially useful when a lot of additions are carried out, which is the case of the convolution computation. Nevertheless, it requires a final conversion to conventional representation which is normally performed by

978-1-4244-5091-6/09/$25.00 ©2009 IEEE

Formación Universitaria
Vol. 2(3), 31-38 (2009)
doi:10.1612/form.univ.4168bfu.09

CAN2PCI: Placa con Interfaz al Bus CAN y PCI con Finalidad Docente

Manuel A. Ortiz, Francisco J. Quiles, Carlos D. Moreno y María Brox
Universidad de Córdoba, Escuela Politécnica Superior, Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica, Edificio Leonardo Da Vinci, Campus Universitario de Rabanales, Ctra. Madrid-Cádiz Km. 396-A, 14071 Córdoba-España
(e-mail: el1orlom@uco.es)

Resumen

En este trabajo se presenta una placa con interfaz al bus CAN y PCI desarrollada para la utilización en prácticas de las asignaturas relacionadas con redes de control. Se trata de una placa de altas prestaciones pensada para su utilización docente gracias a su facilidad de programación. La placa dispone de dos canales CAN independientes y permite acceso directo a los registros del controlador CAN. Las prácticas tienen como objetivo conocer la red de control en los niveles físico y de enlace y desarrollar un software de conectividad (middleware) que realice la interfaz entre estas capas y la de aplicación de usuario. Se exponen también brevemente las prácticas realizadas en una de las asignaturas donde se imparte redes de control, en la que es fundamental la utilización de un hardware conocido que permita programar las funciones básicas que operan directamente con el controlador de bus CAN.

Palabras clave: bus CAN, redes de control, bus PCI, finalidad docente

CAN2PCI: Board with Interface to CAN and PCI Bus for Educational Purposes

Abstract

In this work the development of a board with interface to the CAN and PCI bus for its use in lab courses related to control networks, is presented. This board has high benefits and advantages and has been implemented for educational purposes due to its programming facility. The board has two independent CAN channels and it allows direct access to the registers of the CAN controller. The objective of the lab experiments is to study the control networks in the physical and link levels and to develop a middleware that performs the interface between these layers and the user application. The experiments done in one of the courses, which includes control networks, are briefly described. In these practical labs it is very important the use of a known hardware that allows programming the basic functions which directly operate with the CAN bus controller.

Keywords: CAN bus, networks control, PCI bus, educational objectives

Formación Universitaria
Vol. 2(3), 25-30 (2009)
doi:10.1612/form.univ.4168afu.09

ADQPCI: Placa de Adquisición de Datos con Fines Docentes

Francisco J. Quiles, Manuel A. Ortiz, Carlos D. Moreno y María Brox
Universidad de Córdoba, Escuela Politécnica Superior, Departamento de Arquitectura de Computadores, Electrónica y Tecnología Electrónica, Edificio Leonardo Da Vinci, Campus Universitario de Rabanales, Ctra. Madrid-Cádiz Km. 396-A, 14071 Córdoba-España
(e-mail: el1qulaf@uco.es)

Resumen

En este trabajo se presenta una de las placas de adquisición de datos que se ha desarrollado con fines docentes para su utilización en prácticas relacionadas con sistemas en tiempo real e informática industrial y se plantean algunas de las ventajas e inconvenientes frente a la utilización de placas comerciales. A lo largo del trabajo se detalla el diseño del hardware, en el que se ha priorizado la facilidad de programación, siendo ésta una de las ventajas frente a las placas comerciales. En estas prácticas es fundamental que el alumno tome conciencia de la importancia de la interfaz hardware-software, si se quiere conseguir un sistema fiable y que explote al máximo las características del hardware. Con el desarrollo de una placa de adquisición de datos se consigue un sistema que el alumno puede utilizar en varias asignaturas de su titulación que están relacionadas con el desarrollo y programación de sistemas empotrados.

Palabras clave: adquisición de datos, diseño hardware, sistemas empotrados, interfaz hardware-software, fines docentes

ADQPCI: Data Acquisition Board for Educational Purposes

Abstract

In this work a data acquisition board developed for educational use in subjects related to real-time systems and industrial computing, is presented. The main advantages and disadvantages of using these boards versus the use of commercial boards are discussed. The hardware design described along this work emphasizes the facility of programming the board, which is one of the main advantages versus the commercial boards. In these practices it is essential that student comprehend the importance of the hardware-software interface in order to obtain a reliable system which exploits in a maximum way the characteristics of the hardware. The development of a data acquisition board allows to obtain a system that the students can use in several course during his university career which are related to the development and programming of embedded systems.

Keywords: data acquisition, hardware design, embedded systems, hardware-software interface, educational objectives

2010 International Conference on Reconfigurable Computing

UCORE: RECONFIGURABLE PLATFORM FOR EDUCATIONAL PURPOSES

Francisco J. Quiles, Manuel Ortiz, Maria Brox, Carlos D. Moreno
Department of Computer Architecture
University of Cordoba
University Campus of Rabanales
14071 Cordoba. Spain
email: el1orlom@uco.es

Javier Hormigo, Julio Villalba
Department of Computer Architecture
University of Malaga
University Campus of Teatinos
29080 Malaga. Spain
email: hormigo@ac.uma.es

Abstract— **This paper presents an educational platform for digital system practices with a conventional PCI bus interface, based on reconfigurable hardware especially useful for the designing of hardware accelerators and systems with a PCI bus interface. The aim of the platform is to provide students with a single tool to develop rapid prototypes that covers all aspects involved in the study of digital systems. The platform consists of hardware and software components that allow to easily develop prototypes of general electronic systems, simple systems with a conventional PCI bus interface, hardware accelerators, and buses. The platform is focused on reconfigurable computing practices for university degrees in Electronics Industrial Engineering and Computer Engineering of the new framework of European Higher Education Area (EHEA).**

Keywords-Reconfigurable Systems; CPLD; FPGA; European Higher Education Area

I. INTRODUCTION

The development of Digital Electronic Systems has undergone a rapid evolution throughout the last decades. The first systems were built with low integration digital circuits and a high number of components. As the level of integration of components began to increase, the system components were grouped into functional blocks that were implemented in complex programmable logic devices (PLD) that contained mainly logic control [1].

Due to the large development and integration of programmable logic devices, current systems tend to be completely implemented in a single chip (SOC), and may even contain a large number of processors (MPSoC).

Parallel to this development, the teaching of electronic systems has had to change. The programmable logic devices and reconfigurable systems are an important part of studies of Electronics Industrial Engineering and Computer Engineering [2]. In addition, in new degrees tailored to the European Higher Education Area (EHEA) [3], embedded systems and reconfigurable systems have become particularly relevant.

The rest of the paper is organized as follows: Chapter II will discuss the teaching methodology, Chapter III will show all the components of the UCORE platform used in PLDs practices of the current studies of Electronics Industrial Engineering and supplemented for the new EHEA studies. Chapter IV will describe the different types of practices that can be carried out with this platform, and finally conclusions and future work will be shown.

II. TEACHING METHODOLOGY

In our opinion (which is also widely believed), the fact that students have a basic education in digital systems is essential for the development of digital electronic systems, and therefore, it is compulsory in degrees in Electronics Engineering and Computer Engineer (either electronic or digital) [4]. This basic education begins with the study and development of systems with logic gates and flip-flops; following their basic education with the analysis and design of small functional blocks such as decoders, multiplexers, registers, and counters. To complete the basic education, the students learn the fundamentals of semiconductor memories and small units of control [5]. This is the ideal starting point in order for students to begin the study of programmable logic devices to complete the study and development of complex reconfigurable systems.

Given the horizontal nature of knowledge in EHEA studies, the students should cover the main stages of the development of a reconfigurable system. Regardless of the physical design of the electronic system, including the PCB, the students should:

1 Design the overall system- functional blocks level.
2 Identify external interfaces that the system will have.
3 Determine the functionalities that will be integrated into programmable logic devices.
4 Design features of the programmable logic device-architecture and functional blocks level, and make the description in a high-level language, such as VHDL.
5 Simulate the design that will be integrated into the programmable logic device with a simulation tool such as ModelSim.
6 Programme programmable logic devices.
7 Debug the reconfigurable system.
8 Check the external interfaces.
9 Carry out a global testing.

The teaching methodology used in degrees at the University of Cordoba and Malaga is the one shown in Figure 1. The first part is carried out with commercial synthesis tools.

978-0-7695-4314-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ReConFig.2010.60

Convolution Computation in FPGA Based on Carry-Save Adders and Circular Buffers

Carlos D. Moreno[1,4], Pilar Martínez[2,4], Francisco J. Bellido[1], Javier Hormigo[3,5], Manuel A. Ortiz[1], and Francisco J. Quiles[1]

[1] Computer Architecture, Electronics and Electronic Technology Department, University of Córdoba, Spain
[2] Applied Physics Department, EPS, University of Córdoba, Spain
[3] Computer Architecture Department, University of Málaga, Spain
[4] Campus Universitario de Rabanales. 14071 Córdoba, Spain
[5] Campus Universitario de Teatinos. 29080 Málaga, Spain
el1momoc@uco.es

Abstract. In this article, we present some architectures to carry out the convolution computation based on carry–save adders and circular buffers implemented on FPGAs. Carry-save adders are not frequent in the implementation in FPGA devices, since these have a fast carry propagation path. We make use of the specific structure of the FPGA to design an optimized accumulator which is able to deal with carry–save additions as well as carry–propagate additions using the same hardware. On the other hand, this structure of circular buffers allows the convolution computation of two signals with two algorithms of calculation: the input side algorithm and the output side algorithm, in a more efficient way.

Keywords: Convolution, FPGA, carry-save adders, circular buffers.

1 Introduction

There are several methods to describe the relation between the input and output of the linear time invariant systems (LTI), when both are represented according to time. One of the methods to describe it is by means of differential linear equations (in continuous time) or equations in differences of constant coefficients (in discrete time). Another way of representing this relation would be by means of a block diagram which represents the system as an interconnection of three elementary operations: multiplication, addition, and displacement in the time for systems in discreet time, or integration for systems in continuous time. The third form is the description by means of variables of state, which corresponds to a series of differential equations or in differences of the first order connected that represent the behavior of the 'state' of the system and an equation that relates the state to the output.

However, the most widely used method to represent this relation is related to its response to impulse. The response to impulse is the output of the system associated with the input of the impulse. Considering the response to the impulse, we determine

M.L. Reyes et al. (Eds.): IT Revolutions 2011, LNICST 82, pp. 237–248, 2012.