UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

Estudio e Implementación de la Transformada Wavelet para la Compresión de Imágenes y Vídeo

TESIS DOCTORAL

Presentada por:

Ricardo José Colom Palero

Dirigida por:

Dr. D. Angel Sebastiá Cortés

Dr. D. Rafael Gadea Gironés

VALENCIA, 2001

Resumen - Resum - Abstract

La Tesis Doctoral que se presenta ha sido desarrollada dentro del proyecto CICYT con la referencia TIC2000-1151-C07-05 y que lleva por título: *"Compresión de Vídeo en Tiempo Real Orientado a Aplicaciones Médicas. Estudio de la Segmentación del Algoritmo de la Transformada Wavelet e Implementación VLSI"*. En este proyecto se pretende realizar un sistema hardware para la compresión de vídeo en tiempo real, con la finalidad de poder ser transmitido por intranet o por internet. El proyecto se particulariza para aplicaciones médicas, en las que se requiere utilizar sistemas de compresión sin pérdidas. El uso de los estándares de compresión JPEG y MPEG, en aplicaciones médicas, presenta inconvenientes ya que éstos se basan en la transformada discreta del coseno (DCT).

En la presente Tesis Doctoral se realiza un diseño hardware de la transformada wavelet, con el fin de ser utilizado en un sistema de compresión de imágenes y vídeo digital. El diseño pretende ser abierto, modular y escalable, de modo que los parámetros definitivos de la transformada wavelet puedan ser elegidos en función del tipo de imágenes y de los índices de compresión. Así, una vez fijados los parámetros, el diseño puede ser rápidamente sintetizado e implementado en una FPGA. Las características de reconfigurabilidad de las FPGA permiten la modificación del funcionamiento de la transformada.

Previamente a la realización del diseño ha sido necesario realizar un estudio de la transformada wavelet de dos dimensiones, analizando los tipos de filtros que se pueden utilizar, así como cuáles son los que proporcionan una mejor relación señal-ruido. Este estudio ha permitido realizar un análisis del número de bits que hacen falta en cada etapa de la transformada wavelet, para hacer el cálculo utilizando datos del tipo entero.

La Tesi Doctoral que es presenta ha sigut desenrotllada dins del projecte CICYT amb la referència TIC2000-1151-C07-05 i que porta per títol: *"Compressió de Vídeo en Temps Real Orientat a Aplicacions Mèdiques. Estudi de la Segmentació de l'Algoritme de la Transformada Wavelet i Implementació VLSI"*. En este projecte es pretén realitzar un sistema hardware per a la compressió de vídeo en temps real, amb la finalitat de poder ser transmés per intranet o per internet. El projecte es particularitza per a aplicacions mèdiques, en les que es requerix utilitzar sistemes de compressió sense pèrdues. L'ús dels estàndards de compressió JPEG i MPEG, en aplicacions mèdiques, presenta inconvenients ja que es basen en la transformada discreta del cosinus (DCT).

En la present Tesi Doctoral es realitza un disseny hardware de la transformada wavelet, a fi de ser utilitzat en un sistema de compressió d'imatges i vídeo digital. El disseny pretén ser obert, modular i escalable, de manera que els paràmetres definitius de la transformada wavelet puguen ser elegits en funció del tipus d'imatges i dels índexs de compressió. Així, una vegada fixats els paràmetres el disseny pot ser ràpidament sintetitzat i implementat en una FPGA. Les característiques de reconfigurabilitat de les FPGA permeten la modificació del funcionament de la transformada.

Prèviament a la elaboració del disseny ha sigut necessari realitzar un estudi de la transformada wavelet de dos dimensions, analitzant els tipus de filtres que es poden utilitzar, així com quins són els que proporcionen una millor relació senyal-soroll. Este estudi ha permés realitzar un anàlisi del nombre de bits que fan falta en cada etapa de la transformada wavelet, per a fer el càlcul utilitzant dades del tipus sencer.

This Ph D. Thesis, has been developed with funds coming from CICYT organism within research project referrenced TIC2000-1151-C07-05 and titled *"Compresión de Vídeo en Tiempo Real Orientado a Aplicaciones Médicas. Estudio de la Segmentación del Algoritmo de la Transformada Wavelet e Implementación VLSI"*. The main target of this project is making a real time compression video hardware system with the possibility of internet or intranet transmission. It is focused in medical applications in which it is required using lossless compression. The JPEG y MPEG compression standards are not suitable for medical application because they are Discrete Cosine Transform based.

Within this Thesis, a Hardware design of the wavelet transform is performed to use it in a compression system of image and video. We intend a open, modular and scalable design to have the facility to choose the final wavelet transform parameters by the image features and the compression index. That makes the design could be fast synthesised and implemented once the parameters are fixed in a FPGA device. The reconfigurable features of this devices allows the modifications of the working features of the transform.

It has been necessary to study the 2-D wavelet Transform before design realisation. We have analyse the useful set of filters and these filters that optimise the S/N ratio. This study has led us to fix each wavelet transform step number of bits using integer data class.

Agradecimientos

Nací, crecí y vivo en un pueblo de la costa valenciana, cerca de la capital. Como en la mayoría de los pueblos valencianos, me eduqué en un ambiente familiar, con unos fuertes lazos afectivos y rodeado de gente humilde y trabajadora. Un pueblo hospitalario, turístico y donde las personas viven de cara a la calle. Una de las cosas que he aprendido es a ser agradecido con tus familiares, amigos y vecinos. En este sentido, espero no olvidarme de nadie que directa o indirectamente haya contribuido de algún modo a que ésto sea una realidad.

Quiero agradecer a mis padres mi formación como persona, que me hayan inculcado su amor por el trabajo y que gracias a su esfuerzo personal, haya podido realizar unos estudios que me han permitido formarme profesionalmente, lo cual les llena de orgullo.

Agradezco a mis directores de Tesis Doctoral, Dr. D. Angel Sebastiá y Dr. D. Rafael Gadea su orientación, consejos, y el haberme proporcionado una línea de investigación en la que he disfrutado realizando mi trabajo. Gracias, sobre todo, a Rafael por sus brillantes conocimientos de las herramientas y lenguajes de diseño hardware que tan certeramente me han iluminado en las noches de oscuridad.

Debo también agradecer y felicitar al Dr. D. Francisco Mora por la obtención del proyecto CYCIT, que ha servido de base y de apoyo financiero para la realización de esta Tesis Doctoral.

Gracias a mi compañero de despacho, Dr. D. Marcos Martínez por escucharme y aconsejarme, en diversos aspectos durante la investigación. Era el hombro más cercano donde comentar las peculiaridades del diseño.

Agradezco a mis compañeros de asignaturas, Dr. D. Francisco Ballester, Dr. D. Marcos Martinez, Dr. D. Rafael Gadea, D. Fulgencio Montilla, Dr. D. Angel Sebastiá y D. Miguel Larrea, por permitirme desatender en contadas ocasiones mis obligaciones docentes. Asimismo, también debo agradecer al resto de compañeros del departamento su apoyo y comprensión.

Gracias al Dr. D. Vicente Arnau por la colaboración con la Universidad de Valencia para la realización del proyecto de compresión y transmisión de vídeo en tiempo real del cual forma parte esta Tesis, ya que de su mano empece a dar los primeros pasos en el mundo de la transformada Wavelet. También quiero agradecerle su continuo y desinteresado apoyo.

A mi esposa Yolanda quiero agradecerle su apoyo y ayuda en la realización de esta Tesis. A mi hijo Joan, principal elemento motivador de este trabajo, en pro de su futuro, le dedico la conclusión del mismo.

Indice de Contenidos

1 INTRODUCCIÓN 1

1.1 INTRODUCCIÓN Y OBJETIVOS 1
1.2 METODOLOGÍA 3
1.3 PRINCIPALES APORTACIONES 6
1.4 ESQUEMA DE LA TESIS 7

2 COMPRESIÓN DE IMÁGENES Y VÍDEO DIGITAL 9

2.1 INTRODUCCIÓN 9
2.2 LA COMPRESIÓN DE SEÑALES 10
2.3 LOS ESTÁNDARES DE COMPRESIÓN JPEG Y MPEG 15
2.3.1 El Estándar JPEG 15
2.3.2 Los Estándares MPEG 17
2.4 CONSIDERACIONES SOBRE LAS APLICACIONES 18

3 LA TRANSFORMADA WAVELET 21

3.1 INTRODUCCIÓN 21
3.2 LA TRANSFORMADA WAVELET 24
3.3 LA TRANSFORMADA WAVELET DISCRETA 27
3.4 DISEÑO DE FILTROS 31
3.5 TRANSFORMADA WAVELET BIDIMENSIONAL 36
3.6 ESTUDIO COMPARATIVO ENTRE FILTROS 41
3.6.1 Filtro Daub-4 42
3.6.2 Filtro Daub-4 Modificado 44
3.6.3 Filtro CDF 3/1 45
3.6.4 Filtro CDF 3/1 Modificado 47
3.6.5 Filtro CDF 2/2 48
3.6.6 Filtro CDF 9/7 50

3.6.7 Resumen de Resultados 52
3.7 CONCLUSIONES 54

4 ARQUITECTURAS E IMPLEMENTACIONES DE LA WAVELET 55

4.1 INTRODUCCIÓN 55
4.2 ARQUITECTURAS WAVELET UNIDIMENSIONALES 56
4.2.1 Algoritmos Basados en Técnicas de Filtrado FIR Rápido 56
4.2.2 Arquitectura "Bit-Parallel" 58
4.2.2.1 Arquitectura con filtro FIR en forma directa y memoria única 59
4.2.2.2 Arquitectura con filtro FIR en forma directa y memoria distribuida61
4.2.2.3 Arquitectura con filtro FIR en forma "lattice" 62
4.2.3 Arquitectura "Digit-Serial" 63
4.3 ARQUITECTURAS WAVELET BIDIMENSIONALES 64
4.3.1 Arquitecturas Basadas en Filtros Unidimensionales 64
4.3.2 Arquitecturas Basadas en Filtros Bidimensionales 70
4.4 CONCLUSIONES 73

5 DISEÑO FÍSICO 75

5.1 INTRODUCCIÓN 75
5.2 IMPLEMENTACIÓN DIRECTA 76
5.3 IMPLEMENTACIÓN PAR-IMPAR 80
5.4 IMPLEMENTACIÓN PAR-IMPAR CON MINIMIZACIÓN DE MEMORIA 82
5.5 IMPLEMENTACIÓN CON REDUCCIÓN DE CICLOS 87
5.6 IMPLEMENTACIÓN DE TRES OCTAVAS 92
5.7 IMPLEMENTACIÓN RECURRENTE 96
5.8 IMPLEMENTACIÓN PARTICULARIZADA 103
5.9 RESUMEN DE RESULTADOS 105
5.10 CONCLUSIONES 108

6 CONCLUSIONES Y FUTURAS LÍNEAS 113

6.1 PRINCIPALES APORTACIONES 113
6.2 LÍNEAS DE INVESTIGACIÓN FUTURAS 115

7 REFERENCIAS 117

A LISTADOS FICHEROS VHDL 127

Indice de Figuras

Figura 1.1. Metodología de diseño. 4
Figura 1.2. Simulación en el proceso de diseño. 5
Figura 1.3. Herramientas utilizadas. 6
Figura 2.1. Diagrama de bloques de un sistema de compresión: (a) codificador, (b) decodificador. 11
Figura 2.2. Diagrama de bloques del (a) codificador JPEG, (b) decodificador JPEG. 16
Figura 2.3. Sistema general de compresión de vídeo. 17
Figura 3.1. Diagrama de bloques que representa la transformada de Fourier de una ventana. 22
Figura 3.2. Diagrama de bloques que representa la transformada de Fourier enventanada. 23
Figura 3.3. Transformada de Fourier enventanada y con diezmado. 24
Figura 3.4. Diagrama de bloques de la transformada wavelet. 25
Figura 3.5. Representación en frecuencia de $H_k(z)$. 28
Figura 3.6. Representación en frecuencia de $H_k(z)$ con G(z). 28
Figura 3.7. Diagrama de bloques de la DWT. 29
Figura 3.8. Diagrama de bloques de una estructura en árbol de la DWT. 29
Figura 3.9. Diagrama de bloques del banco en árbol de filtros de análisis de la DWT. 30
Figura 3.10. Diagrama de bloques de la estructura en árbol del banco de filtros de síntesis de la IDWT. 31
Figura 3.11. Banco de filtros de un sólo nivel. 31
Figura 3.12. Diagrama de bloques de un banco de análisis para el cálculo de la DWT 2-D. 37
Figura 3.13. Diagrama de bloques para la DWT 2-D. 37
Figura 3.14. a) Imagen Original. b) Subimagenes obtenidas al aplicar un nivel de DWT. c) Subimagenes obtenidas al aplicar dos niveles de la DWT. 38
Figura 3.15. a) Imagen original. b) Resultado tras aplicar tres niveles de transformada wavelet. 38

Figura 3.16. a) Imagen original. b) Efecto de bordes después de aplicar la DWT y restaurar con IDWT. 41
Figura 3.17. Respuesta en frecuencia del filtro de análisis Daub-4. 43
Figura 3.18. Imagen procesada con filtros tipo Daub-4. 43
Figura 3.19. Número de bits necesarios, a la salida de cada una de las etapas de filtrado. 44
Figura 3.20. Número de bits necesarios, a la salida de cada una de las etapas de filtrado. 44
Figura 3.21. Imagen procesada con filtros tipo Daub-4 modificado. 45
Figura 3.22. Respuesta en frecuencia del filtro de análisis CDF 3/1. 46
Figura 3.23. Imagen procesada con filtro tipo CDF 3/1. 47
Figura 3.24. Número de bits necesarios, a la salida de cada una de las etapas de filtrado. 47
Figura 3.25. Imagen procesada con filtro tipo CDF 3/1 modificado. 48
Figura 3.26. Número de bits necesarios, a la salida de cada una de las etapas de filtrado. 48
Figura 3.27. Respuesta en frecuencia del filtro de análisis CDF 2/2. 49
Figura 3.28. Imagen procesada con filtro tipo CDF 2/2. 49
Figura 3.29. Número de bits necesarios, a la salida de cada una de las etapas de filtrado. 50
Figura 3.30. Respuesta en frecuencia del filtro de análisis CDF 9/7. 51
Figura 3.31. Imagen procesada con filtro tipo CDF 9/7. 51
Figura 3.32. Número de bits necesarios, a la salida de cada una de las etapas de filtrado. 52
Figura 4.1. Diagrama de bloques del banco en árbol de filtros de análisis de la DWT. 56
Figura 4.2. (a) Bloque básico del banco de filtros de análisis de la DWT. (b) Reorganización del bloque básico, mediante descomposición polifase. 57
Figura 4.3. Implementación de un filtro FIR usando técnicas de filtrado FIR rápido. 58
Figura 4.4. Diagrama de bloques de la arquitectura "bit-parallel" de la DWT. 59
Figura 4.5. Diagrama de bloques de la arquitectura "bit-parallel" con memoria única. 60
Figura 4.6. Implementación en forma directa de un filtro FIR. 60
Figura 4.7. Arquitectura de la DWT de tres niveles con filtros de cuarto orden. La señal MC se encarga del control adecuado de los multiplexores. 61
Figura 4.8. Estructura "lattice" que implementa tres niveles de la DWT, con filtros de análisis G(z) y H(z) de cuarto orden. 62
Figura 4.9. Diagrama de bloques de una arquitectura "digit-serial" para la DWT de tres niveles. 64
Figura 4.10. Diagrama de bloques de un nivel de la DWT 2-D con filtros separables. 65
Figura 4.11. Arquitectura "systolic-parallel". 65
Figura 4.12. Arquitectura sistólica. (a) Filtro de seis etapas. (b) Célula de filtrado. 67
Figura 4.13. Arquitectura para la DWT 2-D de 3 niveles. 68
Figura 4.14. Arquitectura para la DWT 2-D "Folded" modificada. 69
Figura 4.15. Diagrama de bloques para la DWT 2-D con filtros no separables. 70
Figura 4.16. Diagrama de bloques de una DWT 2-D de tres niveles con filtros no separables. 71

Figura 5.1. Diagrama de bloques del sistema utilizado en la Fase 03 para la simulación de la transformada wavelet. 76
Figura 5.2. Diagrama de bloques de la implementación directa de una octava de la transformada wavelet. Fase 03. 77
Figura 5.3. Diagrama de bloques funcional de Data_Path_G01. 78
Figura 5.4. Diagrama de bloques de la realización de cuatro octavas. 79
Figura 5.5. Diagrama de bloques del sistema utilizado en la Fase 04 para la simulación de la transformada wavelet. 81
Figura 5.6. Diagrama de bloques de la implementación Par-Impar de una octava de la Transformada Wavelet. 82
Figura 5.7. Diagrama de bloques de la realización de cuatro octavas. 83
Figura 5.8. Diagrama de bloques del sistema utilizado en la Fase 07 para la simulación de la transformada wavelet. 85
Figura 5.9. Diagrama de bloques de la implementación Wavelet_G22 de dos octavas de la transformada wavelet. 85
Figura 5.10. Diagrama de bloques de la implementación Octava_G51 del cálculo de una octava de la transformada wavelet. 86
Figura 5.11. Diagrama de bloques del sistema utilizado en la Fase 08 para la simulación de la transformada wavelet. 89
Figura 5.12. Diagrama de bloques de la implementación Wavelet_G31. 90
Figura 5.13. Diagrama de bloques de la implementación Octava_G51, en Fase 08. 91
Figura 5.14. Diagrama de bloques funcional de Data_Path_G11. 92
Figura 5.15. Diagrama de bloques utilizado para la simulación de la Fase12. 93
Figura 5.16. Diagrama de bloques de la implementación Wavelet_G41. 95
Figura 5.17. Diagrama de bloques de la implementación Octava_G51 de la Fase12. 96
Figura 5.18. Diagrama de bloques del algoritmo piramidal recurrente. 97
Figura 5.19. Diagrama de bloques del algoritmo piramidal recurrente que se desea implementar. 97
Figura 5.20. Diagrama de bloques del sistema utilizado en la Fase 13 para la simulación. 98
Figura 5.21. Diagrama de bloques de la implementación recurrente realizada. 99
Figura 5.22. Diagrama de bloques de la implementación Octava_G51 de la Fase 13. 100
Figura 5.23. Diagrama de bloques de la implementación Octava_G61 de la Fase 13. 101
Figura 5.24. Diagrama temporal de realización de cada una de las octavas. 102
Figura 5.25. Porción del código VHDL de Data_Path_G11. 103
Figura 5.26. Porción del código VHDL de Data_Path_G32. 104

Indice de Tablas

Tabla 3.1. Coeficientes Daubechies para los filtros de la transformada wavelet. 33
Tabla 3.2. Coeficientes filtros biortogonales. 35
Tabla 3.3. Resumen de resultados I. 53
Tabla 3.4. Resumen de resultados II. 53
Tabla 4.1. Resumen de resultados de las arquitecturas. 73
Tabla 5.1. Resumen de las implementaciones realizadas I. 106
Tabla 5.2. Resumen de las implementaciones realizadas II. 107
Tabla 5.3. Resumen de las implementaciones realizadas III. 108
Tabla 5.4. Resumen comparativo entre las arquitecturas. 110

Capítulo 1

Introducción

1.1 Introducción y Objetivos

Las dos últimas décadas han presentado una tendencia generalizada hacia el uso del procesado digital de señal, debido fundamentalmente a su flexibilidad, robustez y precisión. Estos factores unidos con el desarrollo de la tecnología de circuitos integrados, el abaratamiento de costes en los procesos de fabricación, los avances en software y hardware aplicados al diseño VLSI, han permitido obtener sistemas de procesado de señal a bajo coste y elevada densidad de integración.

Las aplicaciones del procesado digital de señal han cubierto recientemente grandes áreas del mercado de consumo, como son lectores de CD y DVD, telefonía móvil o sistemas de procesado de vídeo. El futuro inmediato explota las posibilidades del procesado de señal en aplicaciones relativas a la transmisión de datos de banda ancha, compresión de señales y sistemas de vídeo por internet, con el abanico de aplicaciones derivadas que estos sistemas conllevan. Una de las áreas con mayor proyección surge de la combinación del procesado de señal y de la aplicación concreta a la cual ésta se dirige, diseñándose procesadores de señal a medida.

La Tesis Doctoral que se presenta ha sido desarrollada dentro del proyecto CICYT con la referencia TIC2000-1151-C07-05 y que lleva por título: *"Compresión de Vídeo en Tiempo Real Orientado a Aplicaciones Médicas. Estudio de la Segmentación del Algoritmo de la Transformada Wavelet e Implementación VLSI"*. En este proyecto se pretende realizar un sistema hardware para la compresión de vídeo en tiempo real, con la finalidad de poder ser transmitido por intranet o por internet. El proyecto se particulariza para aplicaciones médicas, en las que se requiere utilizar sistemas de compresión sin pérdidas. El uso de los estándares de compresión JPEG y MPEG, en aplicaciones médicas, presenta inconvenientes ya que éstos se basan en la transformada discreta del coseno (DCT).

En los últimos años ha tomado gran importancia, por sus características, la transformada wavelet, la cual no presenta los efectos que produce la DCT sobre las

imágenes médicas. En este sentido, el proyecto pretende utilizar la transformada wavelet para la realización del sistema de compresión.

Uno de los elementos más complejos a la hora de plantear el algoritmo de compresión es la realización hardware de la transformada wavelet de vídeo digital, ya que se trata de señales tridimensionales con gran cantidad de información. En este sentido, la presente Tesis Doctoral pretende realizar el estudio y la implementación hardware sobre FPGA de una arquitectura, que realice la transformada wavelet de una imagen bidimensional y que pueda ser extensiva al tratamiento de vídeo en tiempo real.

Con este planteamiento los objetivos de la Tesis Doctoral pueden establecerse por medio de los siguientes puntos:

1. El primero de los objetivos será establecer la estructura general de un sistema de compresión de imágenes y vídeo digital, realizando un estudio de las principales características de los estándares de compresión JPEG y MPEG así como su repercusión respecto a las aplicaciones.

2. Se establecerá la base teórica sobre la que se fundamenta la transformada wavelet discreta (DWT), obteniendo cuáles son los coeficientes de filtrado más idóneos para el tratamiento de imágenes, por medio de comparativas entre los diferentes tipos de coeficientes. En este punto deberán tenerse en cuenta aspectos sobre la implementación hardware.

3. Se realizará un estudio de las arquitecturas VLSI más actuales para la realización de la transformada wavelet de imágenes.

4. Llegado a este punto, se realizará una arquitectura para implementar la transformada wavelet que cumpla con las siguientes condiciones:

    - La arquitectura debe realizar la transformada wavelet de una imagen y en consecuencia debe ser capaz de procesar vídeo digital en tiempo real.

    - Presentará una estructura que permita rentabilizar al máximo los recursos hardware disponibles. Es decir, que no haya excesivas partes de la arquitectura que estén grandes tiempos sin actuar.

    - El diseño se implementará en una FPGA, con el fin de obtener un sistema reconfigurable, que pueda adaptarse a consideraciones especiales sobre determinadas imágenes.

    - Deberá tener en cuenta el efecto de los bordes que se produce al realizar la transformada wavelet de señales acotadas.

    - El diseño no se debe particularizar para un único tipo de coeficientes de filtrado wavelet, sino que debe permitir aprovechar las cualidades de reconfigurabilidad de las FPGA para cambiar los coeficientes.

    - Será un diseño modular y escalable que permita incrementar con facilidad el número de octavas o niveles de la transformada wavelet.

- Se debe mantener un tiempo de operación de un pixel por ciclo, lo cual representa que el tiempo de procesado de una imagen de $N \times N$ pixeles debe ser del orden de N^2 ciclos.

5. Por último, se particularizará la arquitectura para unos tipos de coeficientes determinados, con el fin de establecer una comparación con la arquitectura general anterior.

1.2 Metodología

En este apartado se describe la metodología de trabajo empleada en la realización de la Tesis. La finalidad es describir las etapas del proceso de investigación y las herramientas utilizadas. Básicamente las herramientas utilizadas se pueden clasificar en dos grupos:

1. Herramientas para el estudio y análisis de sistemas de tratamiento digital de señales.
2. Herramientas para el diseño de sistemas integrados digitales, basadas en lenguajes de descripción hardware (HDL).

La Tesis comienza con el estudio sobre la transformada wavelet, el cual se realizó utilizando herramientas del primer grupo, tales como MATLAB. En MATLAB 5.3 se realizó un primer banco de pruebas en el que se realizaba la transformada wavelet de imágenes con diferentes coeficientes de filtrado. Sin embargo, esta herramienta no permite trabajar con bits, y aunque admite datos del tipo entero, no permite realizar operaciones matemáticas con ellos. Por tanto, las operaciones se realizaban con datos en coma flotante y luego se redondeaban a enteros.

La implementación de la transformada wavelet en una FPGA debía realizarse mediante operaciones con enteros, por tanto los resultados obtenidos en MATLAB no tenían la suficiente precisión o no se ajustaban a la realidad de las implementaciones.

El estudio de la transformada wavelet utilizando operaciones en las que se puede controlar el número de bits, se completó utilizando herramientas de simulación HDL. Concretamente se utilizó el simulador HDL denominado ModelSim 5.3, para lo que se transcribieron los algoritmos realizados en MATLAB al lenguaje de descripción hardware denominado VHDL. También fue necesario crear las funciones para leer y guardar una imagen en el disco. Sin embargo, ModelSim no puede visualizar imágenes, así que para solucionar este aspecto se utilizó MATLAB.

Una vez realizado el estudio de la transformada wavelet hay que abordar el diseño de la arquitectura y su implementación, para lo cual se ha utilizado la metodología de diseño de sistemas integrados digitales basados en lenguajes HDL. El flujo de diseño de esta metodología se puede ver en la Figura 1.1.

Cualquier metodología clásica de realización de sistemas digitales encierra implícitamente estas etapas, sin embargo no suelen reflejarse en los flujos de diseño, ya que algunas de ellas se realizaban de forma manual sin ningún tipo de verificación. Actualmente, con el uso de lenguajes de descripción hardware para cada una de las

etapas existen una serie de herramientas y tareas a realizar que apoyan labores que antes eran realizadas totalmente por el diseñador.

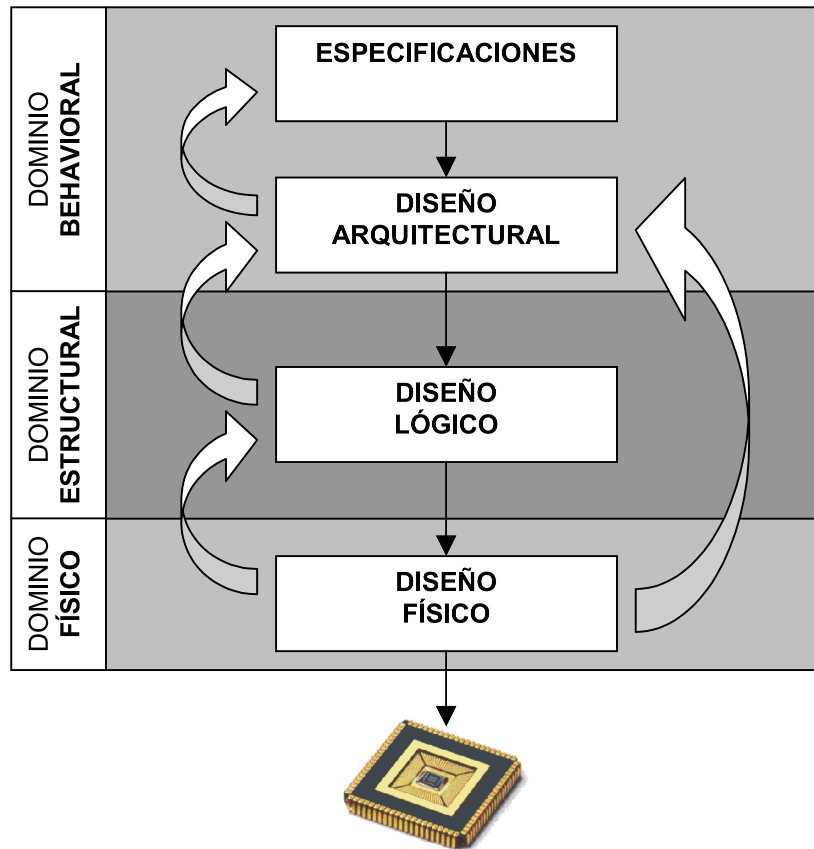


Figura 1.1. Metodología de diseño.

Un elemento común y necesario a lo largo de la metodología de diseño, que pretende garantizar el buen funcionamiento de lo que se está diseñando, son las simulaciones. La simulación es una etapa del diseño cuya función principal es verificar el correcto funcionamiento del diseño. Ello implica que después de cada una de las etapas en las que se hacen modificaciones del diseño, hay que volver a realizar la simulación para comprobar que su funcionamiento es el adecuado.

En el uso de un flujo de diseño basado en una entrada de diseño mediante HDL, las principales simulaciones que se han de realizar se corresponden con cada uno de los niveles de descripción: nivel funcional, nivel lógico y nivel físico. En la Figura 1.2 se ven las etapas de simulación en el proceso de diseño, tal como se puede apreciar, un error provoca un retroceso en el flujo de diseño.

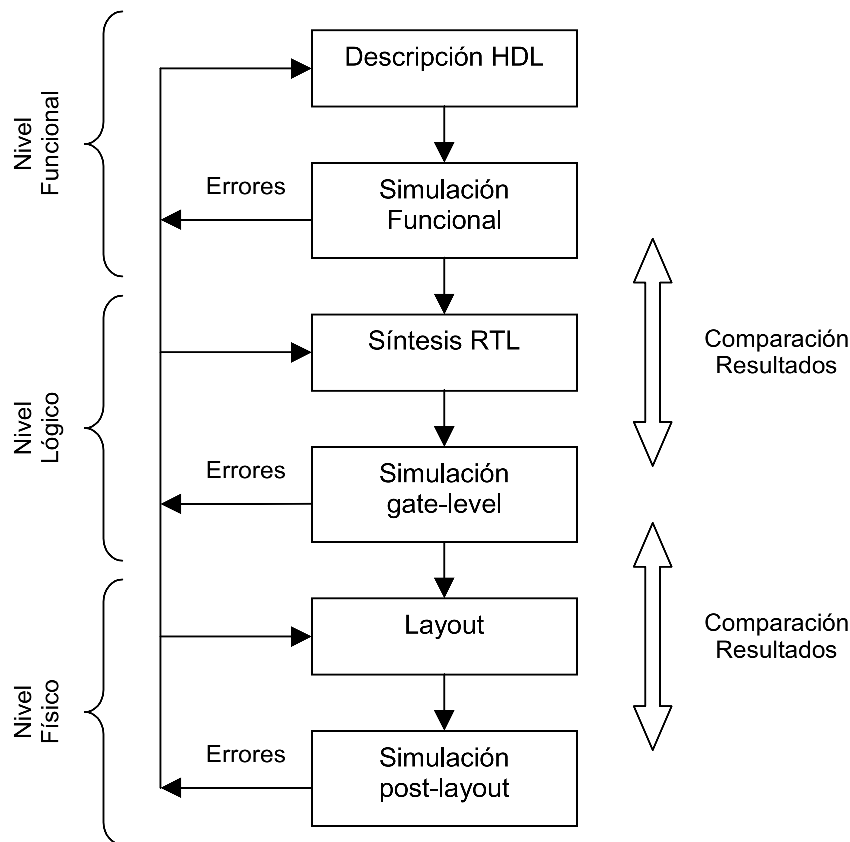


Figura 1.2. Simulación en el proceso de diseño.

En el proceso de diseño, verificación e implementación de la arquitectura de la transformada wavelet, las herramientas que se han utilizado se muestran gráficamente en la Figura 1.3. Como herramienta de simulación en cada una de las etapas se ha utilizado, tal como se hizo anteriormente, el simulador de Model Technology ModelSim en su versión 5.3. Aunque el banco de análisis y estudio de la transformada wavelet, tal como se ha comentado anteriormente, se realizó en VHDL, se trataba de una transcripción directa de MATLAB y por lo tanto no era aceptada por los sintetizadores RTL disponibles.

En cuanto a la etapa de síntesis RTL se han empleado principalmente las herramientas de Synopsys, FPGA Express y FPGA Compiler II ambas en la versión 3.5. La diferencia fundamental entre ambas herramientas se centra en la plataforma de trabajo, ya que FPGA Express funciona bajo PC mientras que FPGA Compiler II lo hace en estación de trabajo. Aunque en principio las dos herramientas son iguales, en el FPGA Compiler II se pueden utilizar las opciones de “re-timing”. La última etapa del flujo de diseño descrito en la Figura 1.3, corresponde con la etapa de emplazamiento y rutado sobre FPGA, en este caso se han utilizado las herramientas de XILINX, Foundation y Alliance en su versión 3.3i.

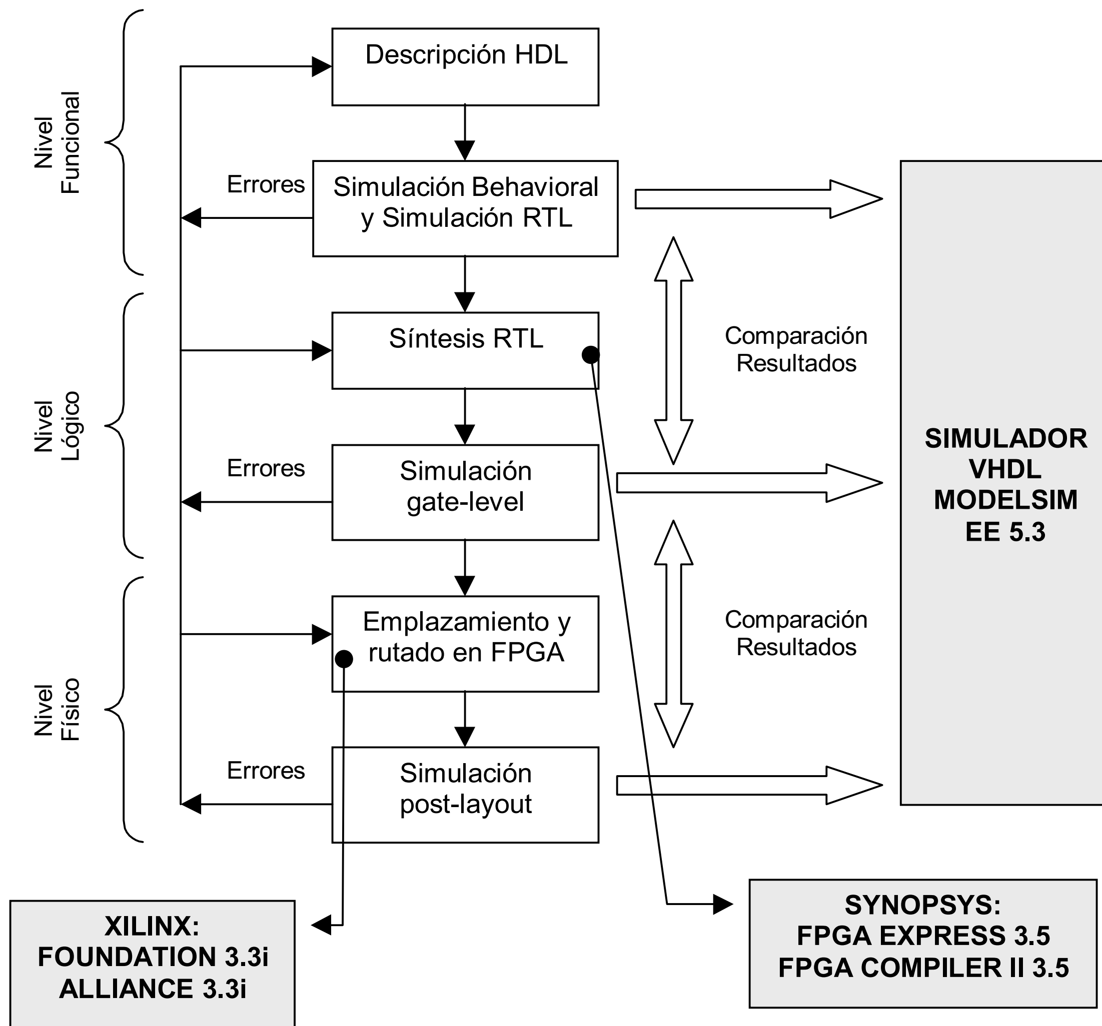


Figura 1.3. Herramientas utilizadas.

1.3 Principales Aportaciones

Este punto resume las principales aportaciones que se realizan en la Tesis presentada, las cuales también son ampliamente comentadas en el capítulo 6. En la Tesis se presenta una nueva arquitectura para la implementación de la transformada wavelet bidimensional. La arquitectura ha sido pensada para implementarse sobre un dispositivo programable, como son las FPGA, en lugar de utilizar circuitos integrados de aplicación específica como viene siendo habitual.

La arquitectura presentada se compara con las principales arquitecturas VLSI actuales, mostrando su bondad. El diseño implementado permite realizar la transformada wavelet de una imagen de $N \times N$ pixeles en N^2 ciclos, reduce el número de accesos a memoria y aprovecha al máximo los recursos del hardware ya que presenta una estructura recurrente. Esta arquitectura es modular y totalmente

escalable, ya que no dispone de una única unidad de control sino que utiliza un control distribuido.

La arquitectura propuesta puede realizar la transformada wavelet, tanto de imágenes como de vídeo en tiempo real. En este último caso puede realizar la transformada fotograma a fotograma sin necesidad de disponer previamente de todo el fotograma, sino únicamente de unas cuantas líneas.

Por otra parte, se ha realizado un estudio comparativo entre los principales coeficientes que se utilizan en procesado de imágenes para realizar la transformada wavelet. También hay que destacar, que la obtención de la arquitectura definitiva que se propone, se ha realizado por el método que denomino diseño por aproximaciones sucesivas, de este modo se puede observar la problemática y la evolución de la arquitectura.

1.4 Esquema de la Tesis

Los capítulos que presenta la Tesis muestran el proceso incremental seguido en la investigación. Inicialmente en el capítulo 2 se presentan los aspectos fundamentales de la compresión de imágenes y vídeo digital. Comenzando por la descripción a nivel de bloques de la estructura básica de un sistema de compresión, se pasará a ver los elementos que forman algunos de los estándares de compresión más conocidos como JPEG y MPEG. Se terminará el capítulo mostrando cuáles son los inconvenientes de estos algoritmos, en señales que requieren que no se produzca pérdida de información en la compresión, como ocurre con las imágenes médicas.

En el capítulo 3 se establecen las bases teóricas para el uso de la transformada wavelet. La obtención de los coeficientes de los filtros que forman la transformada, tanto ortogonales como biortogonales, dará paso al estudio comparativo entre algunos de los tipos de filtros más comúnmente utilizados en el procesado de imágenes.

Una vez planteadas las bases teóricas, se abordará la implementación física. Para ello, en el capítulo 4 se realiza una revisión de las principales arquitecturas VLSI, que habitualmente se utilizan para la implementación hardware de la transformada wavelet de imágenes o de vídeo digital.

En el capítulo 5 el autor de la Tesis presenta una nueva arquitectura para la implementación de la transformada wavelet bidimensional sobre FPGA. La presentación de la arquitectura se realiza mostrando su evolución desde la estructura planteada inicialmente hasta la arquitectura definitiva.

Capítulo 2

Compresión de Imágenes y Vídeo Digital

2.1 Introducción

Recién estrenado el nuevo milenio, apenas transcurre una semana en la que no se escuche el anuncio de un nuevo avance en el área de las tecnologías de la información y las comunicaciones. Particularmente, hay que destacar el interés y la participación del público en general en estos logros tecnológicos, como es el uso de internet, lo cual ha supuesto poder disponer de una gran cantidad de información instantánea en la mayoría de las empresas y los hogares. La mayoría de esta información está diseñada para ser empleada visualmente, en forma de textos, gráficos e imágenes o se encuentra incluida en presentaciones multimedia. Habitualmente se considera que una imagen es un cuadro, o una fotografía estática, en la que los objetos allí representados no se mueven. Sin embargo, un vídeo añade el concepto de tiempo, en el que se representan objetos o imágenes en movimiento. Las imágenes o el vídeo digital, habitualmente, se obtienen a partir de la digitalización de señales analógicas. La frecuencia con que la información es transmitida, almacenada, procesada y visualizada se incrementa rápidamente, por lo que actualmente tiene un gran interés la utilización de técnicas que permitan una transmisión eficiente manteniendo la integridad de la información.

Una característica importante de las imágenes y el vídeo digital es que son señales multidimensionales. En el estudio clásico del procesado digital de señales, habitualmente las señales son funciones unidimensionales en el tiempo. Sin embargo, las imágenes son funciones de dos o tal vez de tres dimensiones en el espacio, mientras que el vídeo incluye una tercera (o cuarta) dimensión en el tiempo. La dimensión de una señal es el número de coordenadas que se requieren para indicar la posición de un punto de la imagen. Como consecuencia, el procesado digital de

imágenes y sobre todo del vídeo, requiere un cálculo bastante intensivo, por lo que hacen falta gran cantidad de recursos computacionales y de almacenamiento.

Una imagen digital es un conjunto de elementos dispuestos habitualmente en filas y columnas, donde cada uno de ellos se denomina pixel. El número de filas y columnas suele ser una potencia de 2, ya que de este modo se simplifica el direccionamiento y el cálculo en algoritmos como la FFT (Transformada rápida de Fourier), los cuales tienen una mayor eficiencia para una cantidad de datos potencia de 2.

Los valores numéricos que representan una imagen son una medida de la intensidad de luz reflejada por el objeto que actúa de modelo. La intensidad suele ser un valor positivo. De este modo, cada uno de los elementos que forman la imagen digital puede tener uno o varios valores numéricos asociados, en función de que se trate de una imagen en blanco y negro o en color. Cuando una imagen es representada utilizando una escala de grises, cada uno de los pixeles se cuantifica con un valor entero y positivo dentro del rango $\{0, \ldots, K-1\}$, donde K es una potencia de 2 : $K = 2^B$. Así, el número de niveles de gris es K y el número de bits necesario para representar cada pixel es B. Habitualmente, se escoge $1 \leq B \leq 8$, siendo lo más común $B = 1$ para imágenes binarias y $B = 8$ para imágenes en escalas de grises. La cuantificación de imágenes en color requiere que cada pixel sea representado por tres valores, uno para cada componente de color.

La gran cantidad de datos que contienen las imágenes se incrementa proporcionalmente con su tamaño, lo cual afecta directamente al procesado, almacenamiento, transmisión y visualización de las mismas. El almacenamiento de una imagen digital monocromática, cuyo tamaño es $N \times M$ representada en una escala de grises con B bits es de NMB bits. Por ejemplo, una imagen de 512×512 pixeles, con una cuantificación de $B = 8$ bits, requiere un total de 262.144 bytes, para ser almacenada. En el caso de vídeo digital, las necesidades se incrementan considerablemente, ya que se incorpora una tercera dimensión a las imágenes, el tiempo. Por ejemplo, un vídeo a partir de las imágenes anteriores a una frecuencia de 25 imágenes por segundo, necesita 6,25 megabytes por segundo, lo cual significa alrededor de 22,5 gigabytes para almacenar una secuencia de 2 horas.

2.2 La Compresión de Señales

La finalidad de la compresión de señales consiste en representar dichas señales con el menor número posible de bits sin perder información, incrementando la velocidad de transmisión y minimizando los requerimientos de almacenamiento. Las imágenes comprimidas son representaciones que requieren menos capacidad de almacenamiento que las imágenes originales. La compresión es conocida también como codificación.

Las técnicas de compresión son cruciales para obtener una transmisión y un almacenamiento efectivo de la información visual. Por ejemplo, la transmisión de una imagen no comprimida monocromática de 512×512 pixeles requiere alrededor de 33 s para ser transmitida en una red RDSI (Red Digital de Servicios Integrados), la cual funciona a una velocidad de 64 kbps, tiempo excesivamente elevado para la transmisión de señales de vídeo. Como consecuencia, la compresión es esencial para

el almacenamiento y la transmisión en tiempo real de información audiovisual digital, donde grandes cantidades de datos deben ser manipulados por dispositivos de ancho de banda y capacidad limitadas.

La compresión de las señales en general es posible, ya que estas incorporan información redundante. La redundancia es proporcional a la correlación entre los pixeles de una imagen. Por ejemplo, en cualquier imagen hay un elevado índice de correlación entre pixeles vecinos. Asimismo, en las señales de vídeo existe además una correlación en el tiempo entre cuadros consecutivos.

En la codificación, la imagen decodificada debe ser idéntica a la original, tanto cuantitativamente (numéricamente) como cualitativamente (visualmente). Aunque este requerimiento preserva la precisión de la representación, limita los rangos de compresión a factores no superiores a 2 ó 3. Para conseguir rangos de compresión mayores, hay que recurrir al uso de métodos que eliminen la información redundante que no sea perceptible. Estos métodos únicamente exigirán que la señal original y la decodificada sean iguales visualmente y no numéricamente. Por tanto, en este caso se producirá una pequeña pérdida de información que no será perceptible.

Aunque se puedan alcanzar grandes niveles de compresión utilizando técnicas de codificación con pérdidas, hay aplicaciones en las que es necesario utilizar una codificación sin pérdidas, como es el caso de las imágenes médicas, facsímiles o imágenes bitonales.

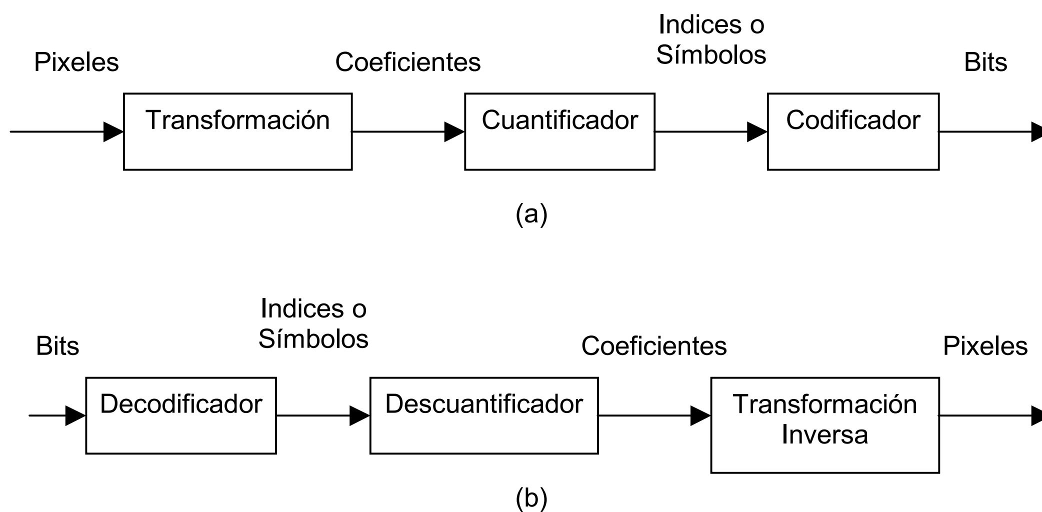


Figura 2.1. Diagrama de bloques de un sistema de compresión: (a) codificador, (b) decodificador.

El diagrama de bloques de un sistema de codificación sin pérdidas se muestra en la Figura 2.1. El codificador, Figura 2.1(a), toma una imagen de entrada y genera como salida una serie de bits comprimidos. El decodificador, Figura 2.1(b), toma como entrada los bits comprimidos y regenera la imagen original descomprimida. En general, el codificador y el decodificador están formados por tres etapas, que son: La transformación, la cuantificación o conversión de datos a símbolos y la codificación [5], [6], [12].

El bloque de transformación, genera a partir de una imagen de entrada unos coeficientes que por sus características posibilitan que la compresión sea más eficiente. La transformación ayuda en la reducción de la correlación de los datos, cambia la distribución estadística de los datos y/o concentra mayor cantidad de información en unas regiones o subbandas. Entre las transformaciones típicas que se pueden utilizar se encuentran: la predicción lineal, las transformadas lineales como la transformada discreta del coseno (DCT), la descomposición en subbandas como la transformada wavelet, las conversiones del espacio de color e incluso combinaciones de todas ellas.

El bloque cuantificador, a partir de los coeficientes genera unos elementos denominados símbolos los cuales permiten ser codificados más fácilmente por la última etapa. Métodos de cuantificación los hay de muchos tipos y formas, desde la utilización de una simple cuantificación escalar hasta el empleo de la compleja cuantificación vectorial. Los cuantificadores escalares uniformes de longitud fija son los más sencillos, ya que realiza un simple redondeo al entero más cercano utilizando siempre el mismo margen de redondeo. Sin embargo existen muchos otros métodos que se utilizan para realizar la cuantificación o la conversión en símbolos, los cuales en ocasiones dependen del tipo de transformación utilizada. Entre estos otros métodos se pueden destacar las técnicas conocidas como codificación run-length (RLC), codificación por truncamiento de bloques (BTC), codificación EZW (embedded zero-tree wavelet) o la codificación SPIHT (set partitioning in hierarchical trees).

La idea básica de la codificación run-length (RLC) consiste en convertir una secuencia de números en una secuencia de parejas de símbolos *(run, value)*, donde *value* es el valor del dato en la secuencia de entrada y *run* es el número de veces que ese valor aparece consecutivamente. En este caso cada pareja *(run, value)* es considerada como un símbolo. Por ejemplo, si la entrada es una secuencia binaria, el cuantificador indicará el número de unos y ceros consecutivos que hay en la secuencia.

La técnica de codificación BTC fracciona la imagen en bloques no solapados de tamaño regular, habitualmente de 4×4 elementos. En cada bloque se realiza el cálculo de la media de los datos y la desviación estándar. Aquel elemento que sea superior al valor de la media obtenido, será codificado con un 1 y aquel elemento que se encuentre por debajo se codificará con un 0. De este modo únicamente se utilizan dos valores para realizar la cuantificación.

La descomposición de la imagen en subbandas, tal como la realiza la transformada wavelet, implica que los coeficientes de baja frecuencia tienen una representación espacial mucho mayor que los coeficientes ubicados en bandas de alta frecuencia. Con la finalidad de aprovechar estas cualidades surgen nuevos métodos de cuantificación orientados hacia el uso de la transformada wavelet, como es la codificación EZW (embedded zero-tree wavelet). Cuando se realiza la transformada wavelet de una imagen, un elemento (padre) de la banda de baja frecuencia está directamente relacionado con otros elementos (hijos) de la banda de frecuencia inmediatamente superior, que ocupan la misma posición espacial. De este modo, se establece un árbol de jerarquía en el que cada padre tiene cuatro hijos, y a su vez los hijos tienen nuevos hijos. La codificación EZW se fundamenta en que si en una rama del árbol un padre tiene el valor 0, todos los descendientes tendrán ese mismo valor, lo cual permite sustituir todos los elementos por un solo 0.

El codificador denominado SPIHT (set partitioning in hierarchical trees), se fundamenta en la misma idea que el EZW, de hecho se considera como una versión evolucionada del primero. La principal diferencia es que el codificador SPIHT utiliza varios árboles jerárquicos, a partir de una serie de umbrales que se establecen.

El tercer bloque (representado en la Figura 2.1) convierte los símbolos en una secuencia binaria comprimida, asignando a cada uno de ellos una palabra binaria. Un nivel de compresión sin pérdidas adecuado se consigue asignado palabras binarias cortas a los símbolos que más veces se producen. Esto significa que no todas las palabras binarias serán del mismo tamaño, lo cual es conocido como codificación de longitud variable (VLC) o codificación entrópica, como por ejemplo, la codificación de Huffman y la codificación aritmética. Una alternativa a la codificación de longitud variable es la utilización de palabras binarias de longitud fija, en la que todos los símbolos se codifican con palabras de la misma longitud. Un ejemplo típico de este tipo de codificación es la denominada de Lempel-Ziv.

En general, tal como se acaba de ver, en el proceso de compresión de una señal o una imagen existen una gran cantidad de opciones para cada uno de los bloques que forman el codificador y el decodificador. Por lo tanto, las opciones que definitivamente se escojan dependerán de la aplicación, de la implementación hardware o software, del retraso permisible y del nivel de compresión que se desee. Algunos de los factores que deben ser considerados a la hora de realizar la elección se enumeran a continuación:

1. Eficiencia de la compresión: La eficiencia de la compresión se mide normalmente como:

$$C_R = \frac{Tamaño\ total\ de\ la\ imagen\ original\ en\ bits}{Tamaño\ total\ de\ la\ \mathrm{sec}uencia\ de\ bits\ comprimidos\ en\ bits} \quad (2.1)$$

   La compresión puede realizarse con pérdidas y sin pérdidas. La pérdida de información se evalúa por medio del error cuadrático medio (MSE) y de la relación señal ruido (PSNR):

$$MSE = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\left[I(i,j) - \hat{I}(i,j)\right]^2 \quad (2.2)$$

$$PSNR = 20\log_{10}\left(\frac{2^n}{MSE^{1/2}}\right) \quad (2.3)$$

   para una imagen *I* de $N \times M$ pixeles y su reconstrucción $\hat{I}$. Aunque en el análisis final, es el ojo humano el que determina la calidad de un sistema de compresión.

2. Retraso de compresión: El retraso de la compresión se puede considerar como el mínimo tiempo necesario para codificar y decodificar una señal de entrada. El retraso se incrementa con el número de operaciones aritméticas que haya que realizar. Minimizar el retraso es especialmente importante en las aplicaciones en tiempo real.

3. Complejidad de la implementación: La complejidad de la implementación se mide a través del número de operaciones aritméticas y de la cantidad de memoria que se necesite. Normalmente se pueden conseguir altos niveles de compresión a costa de incrementar la complejidad de la implementación y el retraso. El diseño óptimo se encuentra en el equilibrio entre estos parámetros.

4. Robustez: En aplicaciones en las que se transmita la secuencia binaria comprimida en un medio propenso a introducir errores, es necesario que el método de codificación sea robusto frente a la transmisión de errores.

5. Escalabilidad: Las señales de entrada deben poderse codificar con diferentes grados de resolución y diferentes tasas de compresión. En este sentido los decodificadores deben ser escalables y versátiles.

La evaluación de estos factores permitirán fijar las características del sistema de compresión que se desea desarrollar. Sin embargo, hay un aspecto muy importante a la hora de elegir los valores adecuados del compresor, se trata del tipo de implementación; es decir si el procesador de la señal va a ser software, hardware o mixto. En un procesado software habrá un programa que se ejecutará en un hardware de propósito general. En este caso, aunque los recursos sean limitados, el control de los mismos es mucho más versátil y flexible, mientras que en un procesado hardware el control de los recursos es más complejo.

Por ejemplo, la implementación de un codificador como el EZW o el SPIHT, requiere disponer de toda la transformada wavelet para poder obtener los árboles de jerarquía, ya que debe ser barrida varias veces y el árbol se obtiene partiendo de la banda de frecuencia más baja. Sin embargo, la estructura de la transformada wavelet es de tal modo que se obtienen en primer lugar la banda de frecuencia más alta, por tanto en principio parece necesario disponer de toda la transformada wavelet antes de comenzar el proceso de codificación. Si se tiene en cuenta, que en el interior de una FPGA resulta bastante complicado disponer de toda la transformada wavelet, ya que se necesitan algo más de 2 Mb para almacenar la transformada de una imagen de 512 $\times$ 512 pixeles, se puede llegar a la conclusión de que la implementación hardware de un codificador EZW o SPIHT es muy complicada, aunque no imposible.

En principio, parece más sencillo en una implementación hardware utilizar una codificación run-legth, ya que si la transformada se obtiene secuencialmente, se pueden ir contando secuencialmente unos y ceros mediante contadores. Sin embargo, la utilización de un codificador EZW o SPIHT resulta más optima con la transformada wavelet [12], por tanto, la implementación hardware de estos codificadores requerirá un estudio más detallado, lo cual no es objetivo de la presente Tesis.

La necesidad de interoperatibidad entre diversos sistemas ha llevado a la formulación de varios estándares internacionales de algoritmos de compresión. Entre las organizaciones que han participado en la realización de estos estándares se encuentra: ISO (International Standards Organization), IEC (International Electrotechnical Commission), ITU (International Telecommunication Union) y CCITT (International Consultive Committee for Telephone and Telegraph) [12]. Algunos de los estándares más conocidos son:

- JPEG (Joint Photographic Experts Group), desarrollado por el ITU en colaboración con ISO/IEC para la compresión de imágenes digitales tanto en color como en escala de grises.
- H.261 y H.263, propuestos como estándar por el ITU, se incluyen en el paquete de normas H.320 para transmisión de vídeo sobre líneas telefónicas RDSI.
- MPEG (Motion Pictures Experts Group), desarrollado por el ISO en colaboración con el IEC, es un método de compresión para señales de vídeo.

2.3 Los Estándares de Compresión JPEG y MPEG

2.3.1 El Estándar JPEG

El estándar JPEG define dos algoritmos de compresión, uno con pérdidas y otro sin pérdidas [12], [13]. La compresión con pérdidas de JPEG, representa una conversión irreversible de una imagen en una secuencia de bits, sin embargo, el estándar proporciona mecanismos para controlar la pérdida de información. El algoritmo de compresión con pérdidas genera una secuencia de bits mucho más pequeña que el algoritmo sin pérdidas. Las características del estándar JPEG con pérdidas son las siguientes:

- Permite dos modos de codificación, el secuencial y el progresivo. Estos modos hacen referencia al método en que son cuantificados los coeficientes de la transformación en la codificación. En el modo secuencial, los coeficientes son codificados por bloques realizando un barrido de izquierda a derecha y de arriba abajo. El modo progresivo codifica sólo información parcial sobre los coeficientes en la primera pasada y la información restante en sucesivos barridos.
- Baja complejidad de las implementaciones hardware y software.
- Admite todo tipo de imágenes, sin importar la fuente, el contenido, la resolución o el formato del color.
- Ofrece un buen equilibrio entre nivel de compresión y calidad de la imagen, excepto en niveles de compresión muy elevados.
- Permite la utilización de jerarquía con múltiples niveles de resolución.
- Admite resoluciones entre 8 y 12 bits.
- Utilizar ficheros de formato JFIF (JPEG File Interchange Format), los cuales permiten intercambiar secuencias de bits comprimidos JPEG entre diferentes plataformas.

El codificador y decodificador del estándar JPEG se basan en la transformada discreta del coseno (DCT). En la Figura 2.2 se muestran las unidades que los constituyen. En el proceso de compresión de una imagen en formato JPEG, en primer lugar se fracciona la imagen en bloques de 8×8 pixeles. A cada uno de estos bloques

se le aplica la DCT, de este modo, se redistribuye la información que contiene la imagen con el fin de agrupar las redundancias y poder eliminar los elementos perceptualmente irrelevantes. El coeficiente ubicado en la posición [0,0] de la matriz de 8×8, es considerado el coeficiente DC, el resto de los coeficientes se consideran como AC.

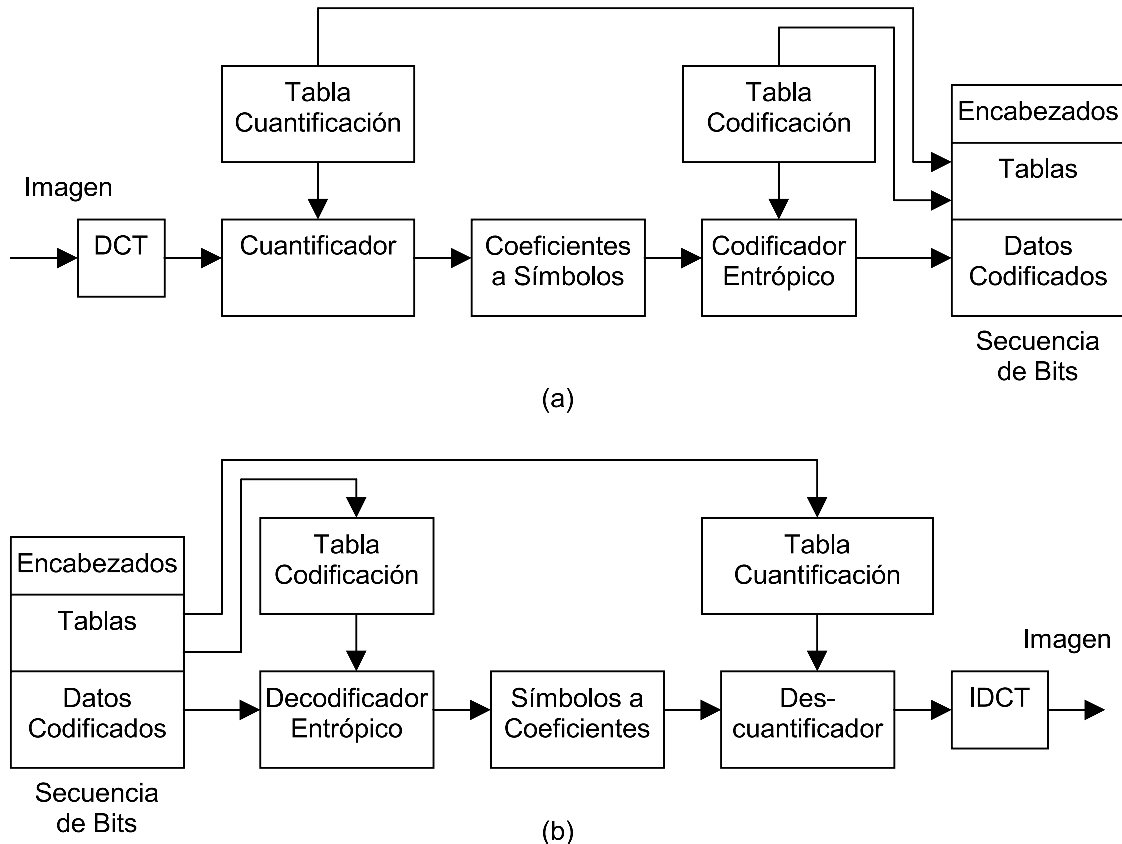


Figura 2.2. Diagrama de bloques del (a) codificador JPEG, (b) decodificador JPEG.

Si se desea recuperar la imagen original por medio de los coeficientes de la DCT, es necesario que estos se representen con una gran precisión, para lo cual se necesitan muchos bits. En la compresión con pérdidas, los coeficientes de la DCT deben ser representados con valores muy pequeños para conseguir grandes rangos de compresión. La unidad de cuantificación se encarga de compactar varios coeficientes de la DCT en uno solo, limitando de este modo las salidas.

La unidad encargada de convertir los coeficientes a símbolos, realiza la cuenta de cuántos ceros hay entre valores que no lo son, para el caso de coeficientes del tipo AC. En los coeficientes del tipo DC la conversión se basa en la diferencia de valor entre el bloque y el anterior. La unidad de codificación entrópica utiliza una codificación Huffman, aunque tiene la opción de utilizar codificación aritmética.

El algoritmo JPEG sin pérdidas (denominado JPEG-LS) también está basado en la transformada discreta del coseno, pero utiliza una modulación por codificación de pulsos diferencial (DPCM) junto con la codificación Huffman o aritmética para realizar una predicción de errores en la codificación y en la decodificación. El hecho de ser una

codificación sin pérdidas o casi sin pérdidas la hace útil para trabajar con imágenes médicas o imágenes de satélites entre otras.

Una alternativa que recientemente se está utilizando en la compresión sin pérdidas de imágenes, es el uso de la descomposición en subbandas (transformada wavelet). Esta proporciona un medio para tratar imágenes con datos no estacionarios, mediante la separación de la información en varias bandas y haciendo uso de la correlación dentro de cada una de ellas y entre ellas.

El comité JPEG actualmente está en proceso de descripción de una técnica de compresión estándar que incorpore características más modernas. El nuevo estándar, conocido como JPEG 2000, estará basado en la transformada wavelet o descomposición en subbandas, con muchas otras características, que permitirán integrar en un único marco la compresión con pérdidas y sin pérdidas.

2.3.2 Los Estándares MPEG

Las normas de compresión de vídeo H.261 y H.263 propuestas por el ITU, son la base sobre la que se fundamentan la mayoría de los sistemas de compresión de vídeo [12]. Como se ha comentado, las señales de vídeo contienen información en tres dimensiones, dos dimensiones en el dominio espacial y la tercera dimensión en el dominio temporal. En este sentido, las técnicas de compresión pretenden disminuir la redundancia de forma independiente en cada uno de los dominios.

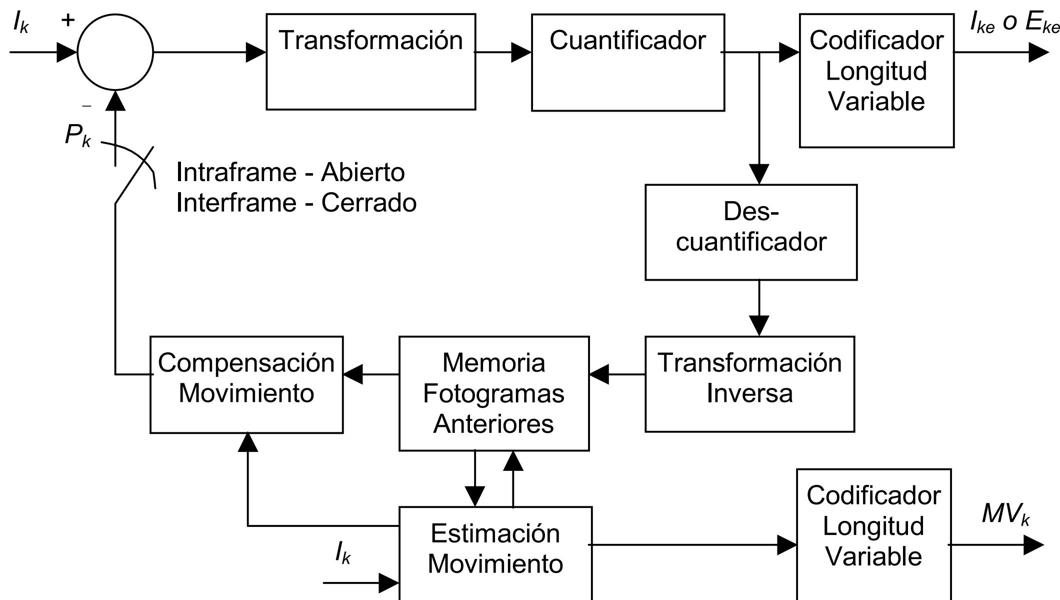


Figura 2.3. Sistema general de compresión de vídeo.

En la Figura 2.3 se muestra el diagrama de bloques de un sistema general de compresión de vídeo. El sistema tiene dos modos de funcionamiento denominados *"intraframe"* y *"interframe"*. En el modo *"intraframe"* se codifica el fotograma actual, operación que se realiza periódicamente, por ejemplo cada 15 fotogramas. En este modo no se utiliza el lazo de realimentación, ya que el fotograma I_k es codificado dando lugar a la secuencia de bits I_{ke}, que será transmitida al decodificador. En el

modo *"interframe"* se utiliza el bucle de realimentación para generar la estimación del fotograma actual P_k. La diferencia entre el fotograma actual y su predicción es lo que se denomina error de predicción E_k, el cual también es codificado E_{ke} y transmitido junto con los vectores de movimiento MV_k.

En la Figura 2.3 se describe la estructura de un codificador de vídeo general, la cual puede sufrir alteraciones en función del estándar que se utilice. Por regla general, la mayoría de los estándares utilizan la transformada discreta del coseno (DCT) aplicada a bloques de 8 $\times$ 8 pixeles en la parte de transformación.

Basándose en la recomendación H.261, utilizando como punto de partida el algoritmo de compresión JPEG y teniendo en cuenta las aplicaciones prácticas, el comité denominado MPEG (Motion Pictures Experts Group) ha desarrollado varios estándares de compresión de vídeo, los cuales se comentan a continuación:

- MPEG-1, estándar de compresión de vídeo digital en formato televisión o multimedia. Está pensado para la reproducción de vídeo con una tasa de 1,5 Mbps y una calidad similar a la de un magnetoscopio VHS. Su estructura corresponde a la Figura 2.3, pero con cuantificador JPEG. Además define un tipo de fotograma bidireccional, que depende del fotograma anterior y del siguiente.

- MPEG-2, constituye una evolución del anterior, concebido para una variedad de aplicaciones que van desde la reproducción hasta la edición profesional o la difusión de alta definición, con un ancho de banda de 4 a 15 Mbps. El modelo es similar al MPEG-1, pero contiene varias mejoras.

- MPEG-4, desarrollado con la idea de incrementar considerablemente la tasa de compresión, emplea métodos de cuantificación en el dominio transformado con compensación de movimiento sobre regiones de perfil arbitrario o agrupaciones con significado conceptual, como texturas, perfiles etc.

- MPEG-7, desarrollado sobre la base de MPEG-4 hace uso de las representaciones basadas en objetos de las herramientas de modelado e incorpora funciones complementarias. MPEG-7 se utiliza en los servicios de almacenamiento y recuperación de vídeo, ya que facilita la manipulación de datos multimedia.

2.4 Consideraciones Sobre las Aplicaciones

Un aspecto que influye considerablemente sobre las características del sistema de compresión es el tipo de imagen o aplicación en la que se vaya a utilizar. Considerar este punto permite particularizar el uso del compresor, optimizando el nivel de compresión. En este sentido, la presente Tesis se desarrolla en un proyecto cuya aplicación principal son las imágenes médicas, por lo tanto a continuación se analiza la problemática asociada a este tipo de imágenes.

En la actualidad existe un gran número de equipos de diagnostico médico que generan imágenes o vídeo digital. La información multimedia médica es diferente de cualquier otra información multimedia por sus características particulares. Existe una

normativa muy estricta y legal alrededor de la información multimedia médica, ya que la salud de los pacientes depende de la validez y exactitud de esta información. Además, la integridad, la confidencialidad y la seguridad de la información médica son cruciales para protegerla de alteraciones accidentales o maliciosas durante los procesos de almacenamiento e intercambio. Otro aspecto importante, es que la información de un paciente debe estar disponible en un espacio de tiempo pequeño, cuando y donde sea requerida y especialmente en los casos de urgencia.

La legislación dice, que toda la información obtenida sobre la salud de un paciente debe ser conservada y almacenada durante un periodo mínimo de entre 5 a 10 años. Así, los hospitales requieren de grandes elementos de almacenamiento. Por otra parte, en algunos hospitales se están popularizando las técnicas de telediagnóstico. Los médicos pueden realizar consultas con otros colegas de otros hospitales dentro del mismo país o fuera de él, para lo cual necesitan medios de transmisión de imágenes o vídeo médico en tiempo real. Debido a la gran cantidad de datos a transmitir, se necesitan redes con un gran ancho de banda para poder mantener la calidad de las imágenes y obtener un diagnóstico correcto. Finalmente, la calidad de los datos almacenados debe ser suficientemente buena para poder realizar un diagnóstico correcto cuando las imágenes sean reconstruidas. Imágenes comprimidas con poca calidad, pueden llevar a un médico a realizar un diagnóstico incorrecto, no detectar un tumor o detectarlo cuando no existe.

Actualmente, se pueden obtener rangos de compresión grandes utilizando técnicas de compresión con pérdidas como JPEG o MPEG, pero los radiólogos se resisten a utilizarlas ya que introducen elementos extraños en la compresión que pueden complicar el diagnóstico. Normalmente, los médicos prefieren utilizar técnicas de compresión sin pérdidas, ya que de este modo se mantiene la calidad y la imagen reconstruida es idéntica a la original. Sin embargo, la compresión sin pérdidas tiene menores tasas de compresión que la compresión con pérdidas. De este modo, las necesidades de almacenamiento de un hospital se incrementan considerablemente y la transmisión por una red es mucho más lenta.

Las imágenes médicas habitualmente se encuentran codificadas en escala de grises, utilizando un solo byte para cada pixel. Además, el vídeo tiene pocas variaciones entre fotogramas consecutivos.

Los estándares de compresión JPEG y MPEG presentan algunos problemas cuando se aplican en particular a la compresión de imágenes y vídeo médicos, provocan ciertos efectos no deseables sobre las imágenes que a continuación se detallan:

1. En general están orientados a la compresión de imágenes y vídeo en color, para lo cual trabajan con los componentes de luminancia y crominancia separadamente, no siendo el método más adecuado para imágenes en escala de grises.

2. Todos ellos utilizan la DCT como elemento de transformación dividiendo la imagen en bloques de 8×8 pixeles, lo que provoca que los bordes de los bloques sean visibles, afectando a la calidad del vídeo reconstruido.

3. Otro efecto que se produce es la aparición de una neblina alrededor de los objetos (especialmente en figuras humanas) y en los lugares donde hay una transición abrupta entre el objeto y el fondo.

4. Por ultimo, suelen añadir un filtro paso bajo para suavizar la imagen reconstruida y ocultar los efectos de los bloques, lo cual es perjudicial para el vídeo médico.

Como se ha comentado, el uso de JPEG o MPEG para la compresión de imágenes o vídeo médico no es adecuado, principalmente por el uso de la transformada discreta del coseno (DCT), la cual introduce efectos no deseables en el proceso de compresión. La transformada wavelet es una alternativa al uso de la DCT que evita los problemas anteriores y por lo ello ha despertado el interés de la comunidad científica. Recordar que el futuro estándar JPEG 2000 incluirá la transformada wavelet [12], [13].

Al utilizar la transformada wavelet frente a la DCT, se recomienda emplear como codificadores el EZW o el SPIHT, para mejorar los niveles de compresión. Esto añade ciertas características al sistema de compresión, como por ejemplo el ser un sistema progresivo. Esta característica permite recuperar la imagen original, aunque con menor detalle (contraste), en el caso de que se produzca en una transmisión una pérdida de parte de la información, siempre y cuando no sea la parte más baja de las bandas de la transformada wavelet.

Capítulo 3

La Transformada Wavelet

3.1 Introducción

La transformada wavelet es una herramienta que permite descomponer una señal o función en varios componentes en frecuencia, posibilitando el estudio de cada uno de los componentes con una resolución dependiente de su escala. La transformada wavelet de una señal que cambia con el tiempo depende de dos variables: la frecuencia y el tiempo, por tanto la transformada wavelet es una herramienta para el análisis tiempo-frecuencia.

En el procesado de señal una de las transformadas más utilizadas es la de Fourier, en la que a partir de una señal *x(t)* contínua en el tiempo, se obtiene una representación en frecuencia *X(ω)* de la misma, tal como se describe en la ecuación (3.1). *X(ω)*, no contiene ninguna información temporal, lo cual imposibilita el análisis correcto de señales cuya frecuencia varíe con el tiempo, sin embargo, realizando un enventanado de la señal *x(t)* se puede obtener una representación tiempo-frecuencia tal como se describe en la ecuación (3.2).

$$X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt \tag{3.1}$$

$$X(\omega, t) = \int_{-\infty}^{\infty} x(s) g(s-t) e^{-j\omega s} ds \tag{3.2}$$

En el dominio discreto, se habla de la transformada discreta de Fourier (DFT), que vendrá representada (directa e inversamente) por las ecuaciones:

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n} \tag{3.3}$$

$$x(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\omega})e^{j\omega n}d\omega \tag{3.4}$$

Si se considera una función de enventanado *v(n)* de duración finita, se puede obtener la transformada de Fourier discreta enventanada, por medio de la ecuación:

$$X(e^{j\omega}, m) = \sum_{n=-\infty}^{\infty} x(n)v(n-m)e^{-j\omega n} \tag{3.5}$$

Arreglando apropiadamente esta ecuación, se obtiene:

$$X(e^{j\omega}, m) = e^{-j\omega m}\sum_{n=-\infty}^{\infty} x(n)v(n-m)e^{j\omega(m-n)} \tag{3.6}$$

Así expresada, la transformada se convierte en un sistema lineal formado por dos partes, un filtro LTI con respuesta impulsional $v(-m)e^{j\omega_0 m}$ y un modulador $e^{-j\omega_0 m}$. Un sistema de estas características puede verse en el diagrama de bloques de la Figura 3.1.

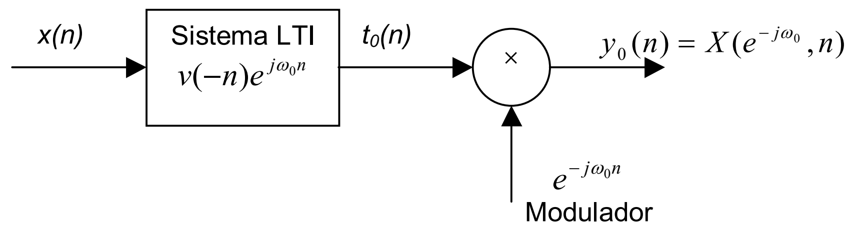


Figura 3.1. Diagrama de bloques que representa la transformada de Fourier de una ventana.

La función del modulador consiste en centrar la respuesta del sistema LTI en una frecuencia ω_0. De este modo, ω_0 representa la movilidad de la ventana a lo largo de las muestras de *x(n)* con la finalidad de realizar la transformada de Fourier enventanada. Por tanto, la transformada de Fourier enventanada puede considerarse como un banco de filtros infinito, con un filtro por cada frecuencia ω_0 que se evalúa. En la práctica, tal como puede verse en la Figura 3.2, sólo se emplean un conjunto discreto de frecuencias, en cuyo caso la transformada de Fourier enventanada se reduce a un banco de *M* filtros paso banda con una respuesta en frecuencia:

$$H_k(e^{j\omega}) = V(e^{-j(\omega-\omega_k)}) \tag{3.7}$$

Un factor clave en este tipo de transformada de Fourier, es la correcta elección de la ventana, ya que determina la resolución en frecuencia y la exactitud de la localización temporal. Ambos conceptos están íntimamente ligados, ya que si se aumenta la exactitud en la localización temporal, estrechando la respuesta impulsional

de la ventana, se está aumentando la banda de paso del filtro y por tanto la resolución en frecuencia.

Si el ancho de banda del filtro $H_k(z)$ es estrecho, la señal $y_k(n)$ variará lentamente con el tiempo (n), por tanto esta cualidad puede ser aprovechada para realizar un diezmado, eliminado un determinado número de muestras y obteniendo así una representación más económica de la señal.

Enventanando la transformada de Fourier se obtiene una representación tiempo-frecuencia de la señal, sin embargo, los filtros que se utilizan en el banco tienen todos el mismo ancho de banda y por tanto, la exactitud de la transformada es pobre a bajas frecuencias y se incrementa al aumentar la frecuencia de la señal. Este problema se puede solucionar utilizando filtros cuyo ancho de banda se incremente con su frecuencia central. Así, la anchura temporal de la ventana aumentaría para frecuencias bajas, manteniendo la resolución en frecuencia uniforme. En esta filosofía se basa la transformada wavelet.

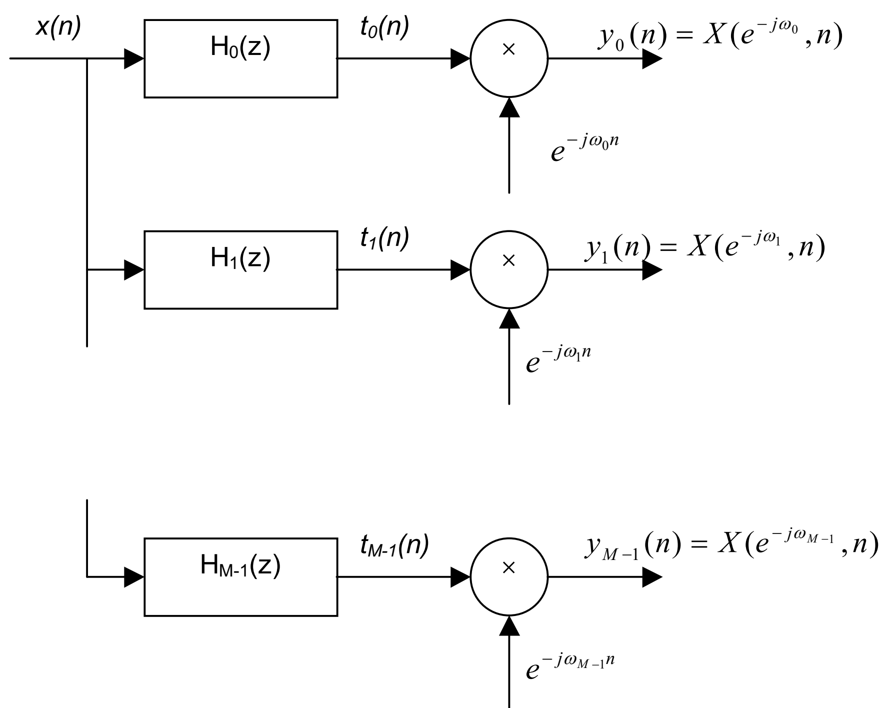


Figura 3.2. Diagrama de bloques que representa la transformada de Fourier enventanada.

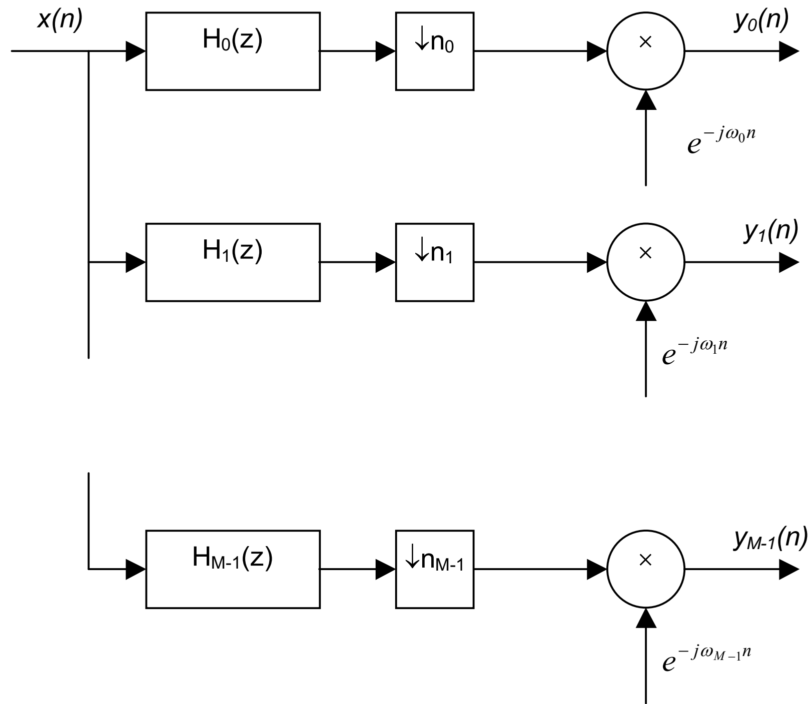


Figura 3.3. Transformada de Fourier enventanada y con diezmado.

3.2 La Transformada Wavelet

Como ya se ha comentado anteriormente la transformada wavelet es una herramienta para realizar el análisis tiempo-frecuencia de señales. En este apartado se desarrollará la transformada wavelet, a partir de los conceptos del enventanado de la transformada de Fourier vistos en el apartado anterior.

En el esquema de banco de filtros presentado en la Figura 3.3, todos los filtros tienen el mismo ancho de banda, ya que todos ellos se obtienen por modulación de un único filtro. En la transformada wavelet los filtros se obtienen por escalado en frecuencia de un prototipo de filtro, por tanto se describirán como:

$$h_k(t) = a^{-\frac{k}{2}} h\left(a^{-k} t\right) \tag{3.8}$$

donde $a > 1$ y k es un entero. El factor de escala $a^{-k/2}$ se usa para asegurar que la energía del filtro es independiente de k. En el dominio en frecuencias la expresión anterior quedará:

$$H_k(j\Omega) = a^{\frac{k}{2}} H\left(ja^k \Omega\right) \tag{3.9}$$

Como el ancho de banda de $H_k(j\Omega)$ es más pequeño, cuanto mayor sea k, se pueden muestrear las ramas inferiores del banco de filtros a intervalos mayores, obteniendo de esta manera un menor número de muestras. Desde el punto de vista

del dominio temporal la anchura de $h_k(t)$ es mayor para las ramas inferiores del banco, con lo que se puede mover la ventana a intervalos mayores, sin perder resolución. Se puede reemplazar la variable temporal contínua por na^kT, donde n es un número entero. Lo cual significa que el desplazamiento temporal de la ventana es a^kT, factor que se incrementa con k, es decir, se incrementa con la disminución de la frecuencia central del filtro de la rama correspondiente.

A partir de estas variaciones, la transformada wavelet (WT) de una seña $x(t)$, quedará definida por la ecuación:

$$X_{WT}(k,n) = a^{-\frac{k}{2}} \int_{-\infty}^{\infty} x(t) h\left(nT - a^{-k}t\right) dt \tag{3.10}$$

En la Figura 3.4(a) se muestra el diagrama de bloques de la transformada wavelet, tomando $a = 2$ en la expresión anterior.

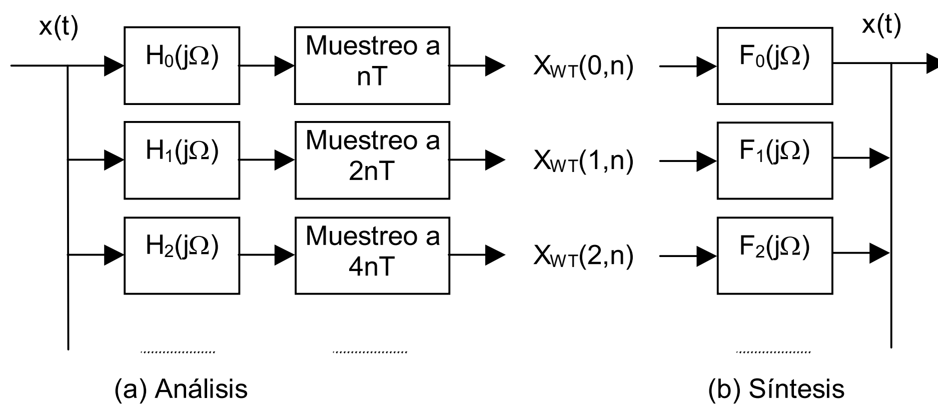


Figura 3.4. Diagrama de bloques de la transformada wavelet.

La inversa de la transformada wavelet (IWT), permite reconstruir la señal $x(t)$ a partir de los coeficientes wavelet $X_{WT}(k,n)$ obtenidos del análisis de dicha señal. La reconstrucción de $x(t)$ depende del filtro prototipo $h(t)$ y de los parámetros de discretización a y T que caracterizan la transformada. La expresión de la transformada inversa tiene la forma:

$$x(t) = \sum_k \sum_n X_{WT}(k,n) \psi_{kn}(t) \tag{3.11}$$

donde $\psi_{kn}(t)$ se denomina función base.

Suponiendo que la función $X_{WT}(k,n)$ representa los coeficientes de la wavelet obtenidos usando el banco de filtros de análisis de la Figura 3.4(a), la reconstrucción de la señal $x(t)$ básicamente consiste en obtener la expresión de los filtros $F_k(j\Omega)$, correspondientes al banco de síntesis (ver Figura 3.4(b)). Se debe tener cuidado con la interpretación del banco de síntesis de la Figura 3.4, ya que $X_{WT}(k,n)$ es una secuencia de muestras, mientras que la señal de entrada al filtro contínuo $F_k(j\Omega)$ debe ser un tren de pulsos de la forma:

$$v_k(t) = \sum_n X_{WT}(k,n)\delta_a\left(t - a^k nT\right) \tag{3.12}$$

Por tanto la salida del banco de filtros de síntesis será:

$$\hat{x}(t) = \sum_k \sum_n X_{WT}(k,n) f_k\left(t - a^k nT\right) \tag{3.13}$$

Como los filtros del banco de síntesis $F_k(j\Omega)$ trabajan con señales cuyo ancho de banda viene fijado por el filtro de análisis precedente, es razonable que se diseñen a partir de un prototipo similar al establecido para los filtros de análisis, esto es:

$$f_k(t) = a^{-\frac{k}{2}} f\left(a^{-k}t\right) \tag{3.14}$$

Sustituyendo esta respuesta impulsional en la ecuación de salida del banco de síntesis:

$$\hat{x}(t) = \sum_k \sum_n X_{WT}(k,n) a^{-\frac{k}{2}} f_k\left(a^{-k}t - nT\right) \tag{3.15}$$

Comparando con la ecuación (3.11), la función base se obtiene como:

$$\psi_{kn}(t) = a^{-\frac{k}{2}} \psi\left(a^{-k}t - nT\right) = a^{-\frac{k}{2}} \psi\left[a^{-k}\left(t - na^k T\right)\right] \tag{3.16}$$

donde $\psi(t) \equiv f(t)$.

Con lo expuesto, la ecuación de reconstrucción de $x(t)$ expresa esta señal como una combinación lineal de un conjunto de funciones base $\psi_{kn}(t)$ que son obtenidas por escalado temporal y desplazamiento de una función base $\psi(t)$ denominada wavelet madre.

Las funciones base más importantes son las ortonormales, pues permiten asegurar que cualquier $x(t)$ tiene una única transformada X_{WT}, lo que asegura que se puede invertir la operación de transformación. Estas funciones cumplen con la condición:

$$\int_{-\infty}^{\infty} \psi_{kn}^*(t)\psi_{lm}(t)dt = \delta(k-l)\delta(n-m) \tag{3.17}$$

Usando el teorema de Parseval, la condición anterior se convierte en:

$$\frac{1}{2\pi}\int_{-\infty}^{\infty} \psi_{kn}^*(j\Omega)\psi_{lm}(j\Omega)dj\Omega = \delta(k-l)\delta(n-m) \tag{3.18}$$

Con la expresión de la ortonormalidad aplicada a la expresión de la transformada se obtiene:

$$X_{WT}(k,n)=\int_{-\infty}^{\infty}x(t)\psi_{kn}^{*}(t)dt \tag{3.19}$$

Comparando esta expresión con la salida del banco de filtros de análisis se puede apreciar que las funciones base son:

$$\psi_{kn}(t)=a^{-\frac{k}{2}}h^{*}\left(nT-a^{-k}t\right)=a^{-\frac{k}{2}}h^{*}\left[a^{-k}\left(na^{k}T-t\right)\right]=h_{k}^{*}\left(a^{k}nT-t\right) \tag{3.20}$$

En particular para $\psi_{00}(t)$ = $\psi(t)$ = $h^{*}(-t)$, y como antes se ha visto $f(t)$ = $\psi(t)$. Entonces para el caso de funciones ortonormales $f(t)$ = $h^{*}(-t)$, se puede obtener la respuesta impulsional de los filtros de reconstrucción como:

$$f_{k}(t)=h_{k}^{*}(-t) \tag{3.21}$$

Esta relación es idéntica a la empleada en los bancos de filtros espejo en cuadratura (QMF) paraunitarios de reconstrucción perfecta [3], con lo que toda la teoría desarrollada para este tipo de filtros será aplicable para la transformada wavelet.

3.3 La Transformada Wavelet Discreta

El paso de la transformada wavelet al dominio discreto no es tan inmediato como inicialmente se pueda pensar, ya que si se intenta sustituir la variable t de la ecuación (3.8) por el índice n de tiempo discreto, puede ocurrir que el factor na^{-k} no siempre sea un valor entero. Así que se deberá recurrir al dominio frecuencial para pasar a tiempo discreto. Haciendo a = 2 en la ecuación (3.9) se obtendrá:

$$H_{k}\left(e^{j\omega}\right)=H\left(e^{j2^{k}\omega}\right) \tag{3.22}$$

Lo que en el dominio de la transformada Z quedaría $H_{k}(z)=H(z^{2^{k}})$ donde k es un entero no negativo. Para $H(e^{j\omega})$ paso alto las respuestas en frecuencia de $H_1(z)$ y $H_2(z)$ se representan en la Figura 3.5. Así se demuestra que en general $H_k(z)$ es un filtro multibanda y no paso banda, por lo que se necesitarán modificaciones para obtener las respuestas paso banda requeridas. Las modificaciones consistirán, en introducir en cascada junto a $H_k(z)$ un filtro $G(z)$ paso bajo con una frecuencia de corte a $\pi/2$ que proporcionará una respuesta en frecuencia como la que se muestra en la Figura 3.6.

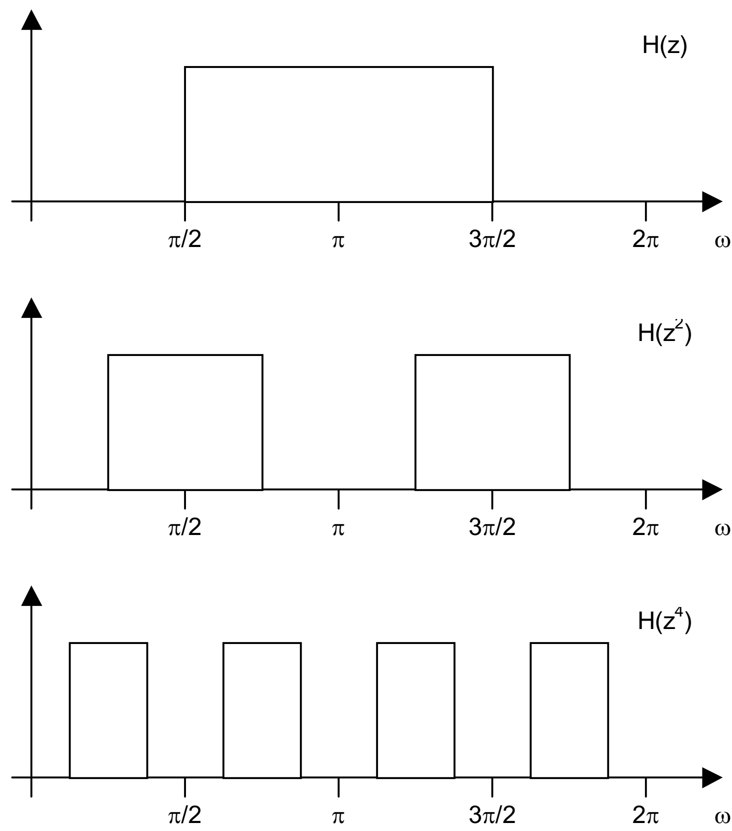


Figura 3.5. Representación en frecuencia de $H_k(z)$.

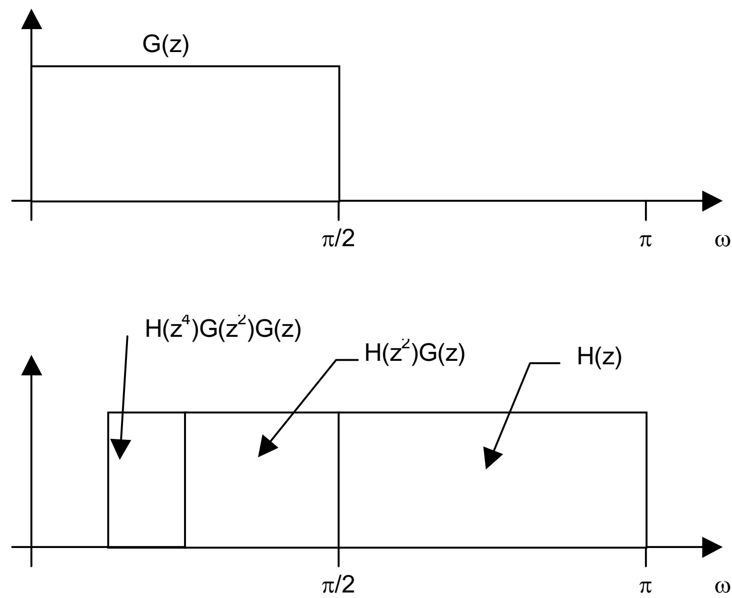


Figura 3.6. Representación en frecuencia de $H_k(z)$ con $G(z)$.

Las respuestas representadas en la Figura 3.6 son filtros paso banda que tienen sus frecuencias centrales en:

$$\omega_k = c2^{-k}\pi; \qquad 0 \le k \le M-1 \quad y \quad c = cte \tag{3.23}$$

El ancho de banda de dichos filtros es $BW_k = 2^{-k}\pi/2$, por tanto el ratio $BW_k = \omega_k$ permanece constante e independiente del índice k.

Este tipo de bancos de filtros según [15] se pueden implementar mediante una estructura como la que se ve en la Figura 3.7 para el caso de un banco de cuatro filtros. En la Figura 3.8 se puede ver el mismo banco de filtros representado mediante una estructura en árbol, lo que se conoce habitualmente como algoritmo de Mallat, en este caso se dice que el banco de filtros es de tres niveles u octavas.

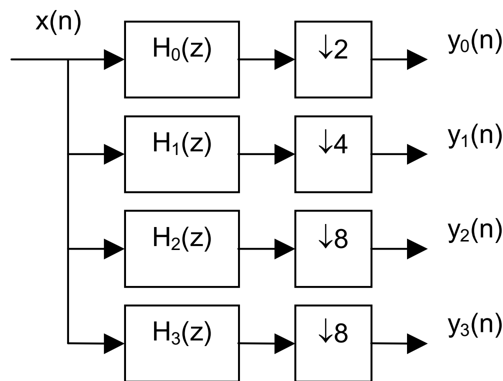


Figura 3.7. Diagrama de bloques de la DWT.

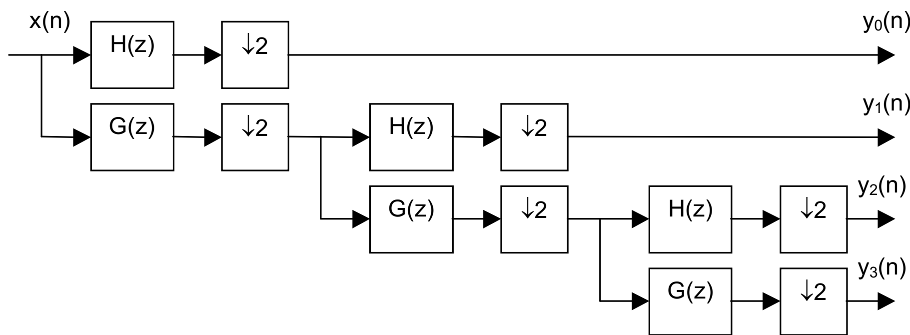


Figura 3.8. Diagrama de bloques de una estructura en árbol de la DWT.

Puede verse cómo las respuestas en frecuencia de la Figura 3.6 corresponden a la función de transferencia de cada rama del árbol. La señal $y_0(n)$ proviene únicamente del filtro $H(z)$, mientras que la señal $y_1(n)$ surge a partir de $G(z)$ seguido de un diezmador que modifica la respuesta de $H(z)$ para convertirla en $H(z^2)$ quedando así $H(z^2)G(z)$.

De modo general, se obtiene que $M = L+1$, donde M es el número de canales del filtro y L el número de niveles del árbol. A cada canal de salida $y_k(n)$, se le suele denominar octava, pues realmente corresponde a una banda frecuencial de anchura

una octava, por tanto y_0 corresponde con la octava de frecuencia más alta y $y_{M-1}(n)$ a la octava de frecuencia más baja.

Según las consideraciones anteriores y atendiendo a la estructura descrita en la Figura 3.7, las expresiones que permiten obtener la transformada wavelet discreta (DWT) de una señal *x(n)* son las siguientes:

$$y_k(n) = \sum_{m=-\infty}^{\infty} x(m) h_k\left(2^{k+1} n - m\right) \qquad para \quad 0 \le k \le M-2 \tag{3.24}$$

$$y_{M-1}(n) = \sum_{m=-\infty}^{\infty} x(m) h_{M-1}\left(2^{M-1} n - m\right) \tag{3.25}$$

Utilizando la representación de la Figura 3.8 con un pequeño cambio de nomenclatura, tal como puede apreciarse en la Figura 3.9, se puede expresar la transformada wavelet discreta por medio de las siguientes ecuaciones:

$$S_{k+1}(n) = \sum_m g(m) S_k(2n - m) \tag{3.26}$$

$$W_{k+1}(n) = \sum_m h(m) S_k(2n - m) \tag{3.27}$$

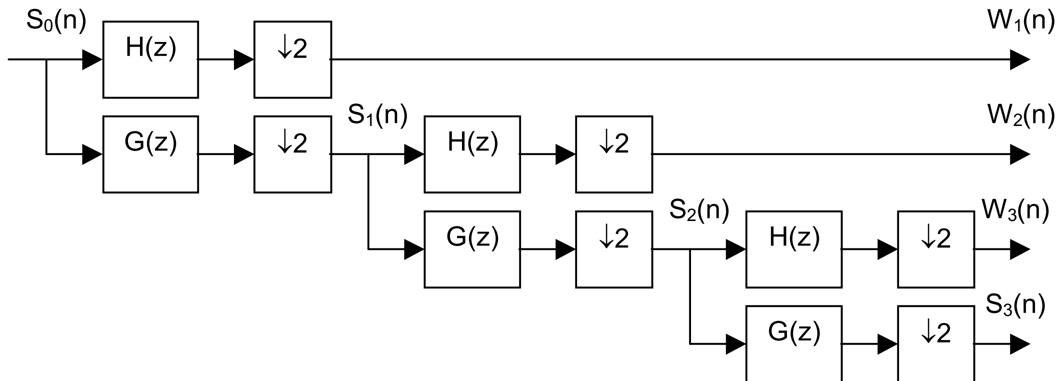


Figura 3.9. Diagrama de bloques del banco en árbol de filtros de análisis de la DWT.

La inversa de la transformada discreta wavelet (IDWT) se puede obtener mediante un banco de filtros de síntesis como el mostrado en la Figura 3.10. La ecuación que describe la inversa de la transformada discreta wavelet es:

$$x(n) = \sum_{k=0}^{M-2} \sum_{m=-\infty}^{\infty} y_k(m) f_k(n - 2^{k+1} m) + \sum_{m=-\infty}^{\infty} y_{M-.1}(m) f_{M-1}(n - 2^{M-1} m) \tag{3.28}$$

Las funciones f_k son funciones base wavelet que pueden expresarse como η_{km} y con ello simplificar la transformada inversa como:

$$x(n) = \sum_{k=0}^{M-1} \sum_{m=-\infty}^{\infty} y_k(m)\eta_{km}(n) \tag{3.29}$$

La función wavelet para *m = 0* es la respuesta impulsional del filtro f_k. En general η_{km} es la respuesta impulsional del filtro desplazada una determinada cantidad.

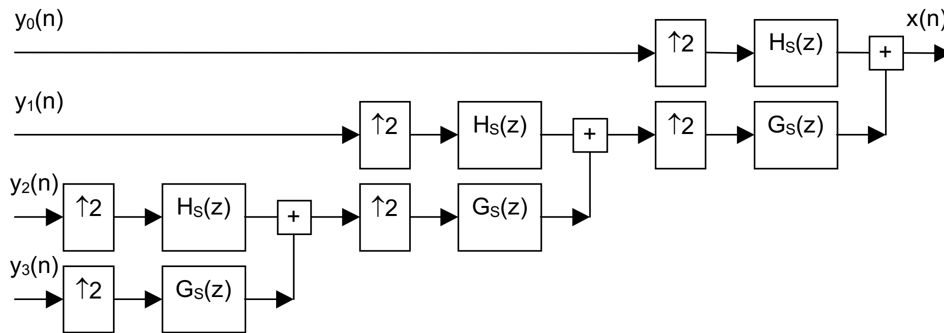


Figura 3.10. Diagrama de bloques de la estructura en árbol del banco de filtros de síntesis de la IDWT.

3.4 Diseño de Filtros

La estructura representada en la Figura 3.11 es la unidad fundamental de los bancos de análisis y síntesis mostrados en la Figura 3.8 y la Figura 3.10, las expresiones de las señales en esta estructura son:

$$y_k(n) = \sum_{m=-\infty}^{\infty} x(m)h_k(2n-m) \qquad para \quad k = 0,1 \tag{3.30}$$

$$x(n) = \sum_{m=-\infty}^{\infty} y_0(m)\underbrace{f_0(n-2m)}_{\eta_{0m}(n)} + \sum_{m=-\infty}^{\infty} y_1(m)\underbrace{f_1(n-2m)}_{\eta_{1m}(n)} \tag{3.31}$$

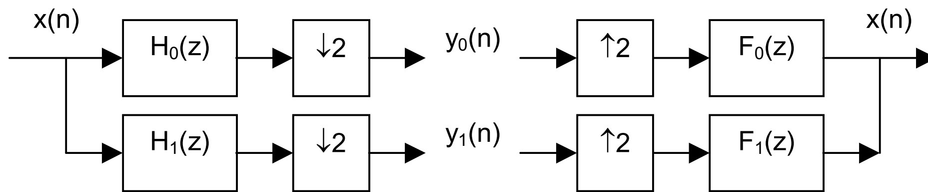


Figura 3.11. Banco de filtros de un sólo nivel.

La condición de ortonormalidad de las funciones base wavelets obtenida a partir de la ecuación (3.17), se puede reescribir como:

$$\sum_{n=-\infty}^{\infty} \eta_{km}(n)\eta_{li}^*(n) = \delta(k-l)\delta(m-i) \tag{3.32}$$

Sustituyendo a partir de la ecuación (3.31):

$$\sum_{n=-\infty}^{\infty} f_k(n-2m) f_l^*(n-2i) = \delta(k-l)\delta(m-i) \tag{3.33}$$

Haciendo un adecuado cambio de variables se obtiene:

$$\sum_{n=-\infty}^{\infty} f_k(n) f_l^*(n-2m) = \delta(k-l)\delta(m) \tag{3.34}$$

Lo cual significa que $f_k(n)$ es ortogonal a las versiones de $f_l(n)$ desplazadas un número par de muestras. La parte izquierda de la ecuación (3.34) es la expresión de la correlación cruzada entre $f_k(n)$ y $f_l(n)$ evaluada en intervalos de *2m* muestras. La expresión de la correlación cruzada en el dominio Z se escribe:

$$F_k(z)\widetilde{F}_l(z)\Big|_{\downarrow 2} = \delta(k-l) \tag{3.35}$$

donde $\widetilde{F}(z) = F^*(z^{-1})$ y el término $A(z)\big|_{\downarrow M}$ es la forma abreviada de expresar un diezmado por *M*, por ejemplo:

$$A(z)\big|_{\downarrow 2} = \frac{1}{2}\left[A(z^{1/2}) + A(-z^{1/2})\right] \tag{3.36}$$

Así la ecuación (3.35) se puede escribir:

$$\widetilde{F}_k(z)F_l(z) + \widetilde{F}_k(-z)F_l(-z) = 2\delta(k-l), \qquad 0 \le k, \quad l \le 1 \tag{3.37}$$

Representándolo en forma matricial:

$$\widetilde{\mathbf{F}}(z)\mathbf{F}(z) = 2\mathbf{I} \tag{3.38}$$

donde:

$$\mathbf{F}(z) = \begin{bmatrix} F_0(z) & F_1(z) \\ F_0(-z) & F_1(-z) \end{bmatrix} \tag{3.39}$$

Esta es precisamente la condición necesaria para que un filtro sea paraunitario, por tanto, la ortonormalidad de las funciones base wavelets equivale a que los filtros F_k cumplan la condición de filtros QMF paraunitarios [3]. Para asegurar la reconstrucción perfecta, los bancos de filtros se deben diseñar teniendo en cuenta la propiedad de simetría en potencia, por la cual se debe cumplir que:

$$\widetilde{H}_0(z)H_0(z) + \widetilde{H}_0(-z)H_0(-z) = 2 \tag{3.40}$$

Por tanto, las expresiones de los filtros serán:

$$H_1(z) = -z^{-N}\widetilde{H}_0(-z), \quad donde \quad N = orden\ de\ H_0(z), \quad y$$
$$F_0(z) = \widetilde{H}_0(z) \quad y \quad F_1(z) = \widetilde{H}_1(z) \tag{3.41}$$

Con estos filtros se asegura la reconstrucción perfecta y la condición de filtros paraunitarios que equivale a la ortonormalidad de las funciones base wavelets.

El resultado obtenido para un banco de filtros de dos canales, es ampliable a un número genérico de canales conservando las características que garantizan la ortonormalidad de las funciones base wavelet. Los coeficientes de los filtros FIR originales, para asegurar la reconstrucción perfecta, deben guardar la siguiente condición:

$$f_k(n) = h_k^*(-n) \tag{3.42}$$

Se puede definir el producto de los filtros paso bajo de la Figura 3.11 como $D(z) = H_0(z)\ F_0(z)$, teniendo en cuenta las ecuaciones (3.40) y (3.41), la condición de reconstrucción perfecta se puede expresar como:

$$D(z) + D(-z) = 2 \tag{3.43}$$

De este modo, el diseño de los filtros wavelet se reduce al diseño del producto de los filtros $H_0(z)$ y $F_0(z)$ y su factorización. La mayoría de los filtros encontrados en la literatura [14], [15], [16], [56], [63], [64], [66] se diseñan partiendo de la factorización de los polinomios de Lagrange por medio de la expresión:

$$D(z) = z^K \left(\frac{1+z^{-1}}{2}\right)^{2K} R_K(z) \tag{3.44}$$

donde

$$R_K(z) = \sum_{n=0}^{K-1} \binom{K+n-1}{n} \left(\frac{2-(z+z^{-1})}{4}\right)^n \tag{3.45}$$

De este modo, la ecuación (3.44) permite que los filtros tengan el máximo número de ceros en $z = -1$ y una respuesta en frecuencia lo más plana posible. El factor $R_K(z)$ es del menor orden posible que permite que se cumpla la ecuación (3.43).

Daubechies [14] desarrolló funciones ortonormales wavelet, que además de cumplir con las condiciones expuestas hasta este momento, presentaban la propiedad de que los filtros obtenidos tenían una fase mínima. La Tabla 3.1 muestra los coeficientes de los filtros paso bajo $H_0(z)$ (ver Figura 3.11), para distintos valores de K.

Tabla 3.1. Coeficientes Daubechies para los filtros de la transformada wavelet.

K	**n**	**$h_0(n)$**
1	0	0,70710678118655

K	n	$h_0(n)$
	1	0,70710678118655
2	0	0,48296291314453
	1	0,83651630373781
	2	0,22414386804201
	3	-0,12940952255126
3	0	0,33267055295008
	1	0,80689150931109
	2	0,45987750211849
	3	–0,13501102001025
	4	–0,08544127388203
	5	0,03522629188571
4	0	0,23037781330890
	1	0,71484657055292
	2	0,63088076792986
	3	–0,02798376941686
	4	–0,18703481171909
	5	0,03084138183556
	6	0,03288301166689
	7	–0,01059740178507

En la Tabla 3.1 puede observarse que los filtros no son simétricos, ésto significa que la respuesta en fase del filtro no es lineal, aunque en el caso de los presentados su cambio de fase sea mínimo. Los filtros simétricos presentan una respuesta en fase lineal y además permiten simplificar las implementaciones físicas, al aprovechar su simetría.

Se pueden obtener filtros simétricos para la transformada wavelet por medio de lo que se denomina funciones wavelet biortogonales. En las wavelet biortogonales, la condición de que el filtro de análisis y el de síntesis deban tener los mismos coeficientes, impuesta en los ortonormales, se hace menos estricta permitiendo que sean filtros de distintos coeficientes. De este modo los filtros paso bajo $H_0(z)$ y $F_0(z)$ de la Figura 3.11, serán distintos y por tanto las condiciones reflejadas en las ecuaciones (3.41) se convierten en:

$$\begin{aligned} H_1(z) &= -z^{-N_F} F_0(-z), \quad donde \quad N_F = orden\ de\ F_0(z), \quad y \\ F_1(z) &= -z^{-N_H} H_0(-z), \quad donde \quad N_H = orden\ de\ H_0(z) \end{aligned} \tag{3.46}$$

Utilizando las ecuaciones (3.44) y (3.45) vistas anteriormente, se pueden diseñar los filtros biortogonales para la transformada wavelet. La Tabla 3.2 muestra los coeficientes para filtros biortogonales obtenidos en [14]. Tal como se aprecia, son filtros simétricos donde $F_0(z)$ depende de K_F y $H_0(z)$ depende de K_H y K_F, por tanto, aunque los filtros de análisis y síntesis sean distintos no son independientes.

Tabla 3.2. Coeficientes filtros biortogonales.

K_F	$F_0(z)$	K_H	$H_0(z)$
1	$\frac{\sqrt{2}}{2}(1+z)$	1	$\frac{\sqrt{2}}{2}(1+z)$
		3	$\frac{\sqrt{2}}{16}(-z^{-2}+z^{-1}+8+8z+z^2-z^3)$
		5	$\frac{\sqrt{2}}{256}(3z^{-4}-3z^{-3}-22z^{-2}+22z^{-1}+128+128z+22z^2-22z^3-3z^4+3z^5)$
2	$\frac{\sqrt{2}}{4}(z^{-1}+2+z)$	2	$\frac{\sqrt{2}}{8}(-z^{-2}+2z^{-1}+6+2z-z^2)$
		4	$\frac{\sqrt{2}}{128}(3z^{-4}-6z^{-3}-16z^{-2}+38z^{-1}+90+38z-16z^2-6z^3+3z^4)$
		6	$\frac{\sqrt{2}}{1024}(-5z^{-6}+10z^{-5}+34z^{-4}-78z^{-3}-123z^{-2}+324z^{-1}+700+324z-123z^2\cdots)$
		8	$\frac{\sqrt{2}}{2^{15}}(35z^{-8}-70z^{-7}-300z^{-6}+670z^{-5}+1228z^{-4}-3126z^{-3}-3796z^{-2}+10718z^{-1}+22050+10718z-3796z^2\cdots)$
3	$\frac{\sqrt{2}}{8}(z^{-1}+3+3z+z^2)$	1	$\frac{\sqrt{2}}{4}(-z^{-1}+3+3z-z^2)$

K_F	$F_0(z)$	K_H	$H_0(z)$
		3	$\frac{\sqrt{2}}{64}\left(3z^{-3}-9z^{-2}-7z^{-1}+45+45z -7z^{2}-9z^{3}+3z^{4}\right)$
		5	$\frac{\sqrt{2}}{512}\left(-5z^{-5}+15z^{-4}+19z^{-3}-97z^{-2}-26z^{-1} +350+350z-26z^{2}\cdots\right)$
		7	$\frac{\sqrt{2}}{2^{14}}\left(35z^{-7}-105z^{-6}-195z^{-5}+865z^{-4}+336z^{-3} -3489z^{-2}-307z^{-1}+11025+11025z\cdots\right)$
		9	$\frac{\sqrt{2}}{2^{17}}\left(-63z^{-9}+189z^{-8}+469z^{-7}-1911z^{-6} -1308z^{-5}+9188z^{-4}+1140z^{-3}-29676z^{-2} +190z^{-1}+87318+87318z\cdots\right)$

3.5 Transformada Wavelet Bidimensional

La transformada wavelet bidimensional se aplica a señales 2-D, como son las imágenes. Mientras que para el cálculo de la transformada wavelet unidimensional, se utilizan filtros unidimensionales, la transformada wavelet discreta 2-D requiere el uso de filtros bidimensionales. Los filtros bidimensionales pueden ser separables o no separables, un filtro 2-D es separable si su función de transferencia se puede expresar como: $f(n_1,n_2) = f_1(n_1)f_2(n_2)$.

Para el caso de filtros separables, a partir de las ecuaciones (3.26) y (3.27) el cálculo de la octava $k+1$ de la transformada wavelet bidimensional de una imagen se puede expresar como:

$$S_{k+1}(n_1,n_2)=\sum_{m_1}\sum_{m_2} g(m_1)g(m_2)S_k(2n_1-m_1,2n_2-m_2) \tag{3.47}$$

$$W_{k+1}^{1}(n_1,n_2)=\sum_{m_1}\sum_{m_2} g(m_1)h(m_2)S_k(2n_1-m_1,2n_2-m_2) \tag{3.48}$$

$$W_{k+1}^{2}(n_1,n_2)=\sum_{m_1}\sum_{m_2} h(m_1)g(m_2)S_k(2n_1-m_1,2n_2-m_2) \tag{3.49}$$

$$W_{k+1}^{3}(n_1,n_2)=\sum_{m_1}\sum_{m_2} h(m_1)h(m_2)S_k(2n_1-m_1,2n_2-m_2) \tag{3.50}$$

donde *H(z)* y *G(z)* son filtros unidimensionales. A partir de estas expresiones se puede obtener el diagrama de bloques de la Figura 3.12, en él se representa el cálculo de una octava de la transformada wavelet discreta para una señal bidimensional. Tal como puede apreciarse, el cálculo se realiza filtrando primero por filas la imagen y luego por columnas. La señal $W_{k+1}^{1}(n_1,n_2)$ contiene las altas frecuencias verticales, ya que se obtiene filtrando la señal $S_k(n_1,n_2)$ con un filtro paso bajo en la dirección horizontal y un filtro paso alto en la dirección vertical. De igual modo, la señal $W_{k+1}^{2}(n_1,n_2)$ contiene las altas frecuencias horizontales y la señal $W_{k+1}^{3}(n_1,n_2)$ las altas frecuencias en ambas direcciones.

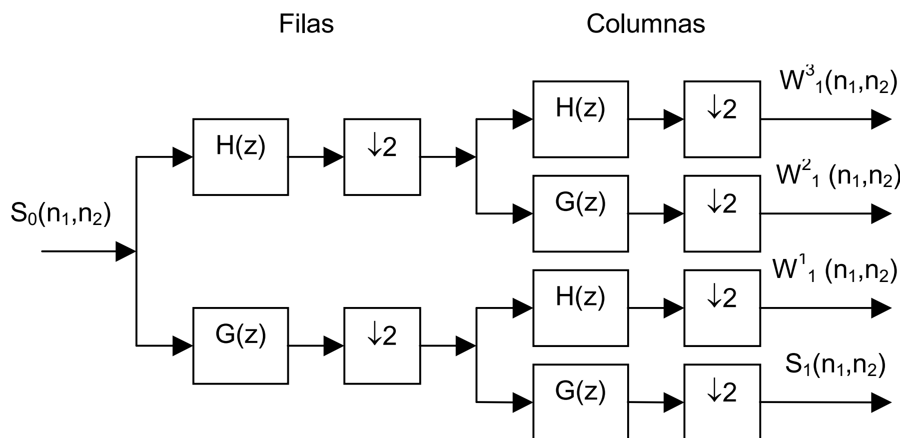


Figura 3.12. Diagrama de bloques de un banco de análisis para el cálculo de la DWT 2-D.

Tal como se observa en la Figura 3.12, para aplicar la DWT a una imagen, primero debe realizarse un filtrado por filas y posteriormente por columnas, ya que se trata de señales bidimensionales filtradas con filtros unidimensionales. Sin embargo, se pueden utilizar directamente filtros bidimensionales, de este modo, se obtienen en paralelo las cuatro salidas como se ve en la Figura 3.13. Los coeficientes de los filtros bidimensionales pueden obtenerse bien multiplicando entre sí los vectores de los coeficientes de los filtros unidimensionales, o bien directamente como coeficientes de filtros bidimensionales no separables. En ambos casos, el resultando será una matriz de coeficientes.

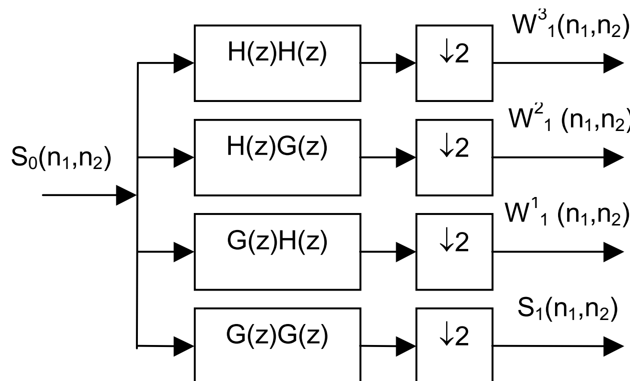


Figura 3.13. Diagrama de bloques para la DWT 2-D.

Por tanto, a partir de una imagen de 512 x 512 pixeles, tras el procesado se obtienen cuatro imágenes de 256 x 256 pixeles. De estas cuatro imágenes resultantes, se tomará la que se ha obtenido del filtro *G(z)G(z)* para volver aplicar la transformada wavelet, así sucesivamente tantas veces como octavas (o niveles) se deseen obtener. En la Figura 3.14 se observa la evolución del tamaño de las imágenes al aplicar la transformada wavelet. Mientras que en la Figura 3.15 se ve el ejemplo de aplicar la transformada wavelet con tres octavas a una imagen concreta.

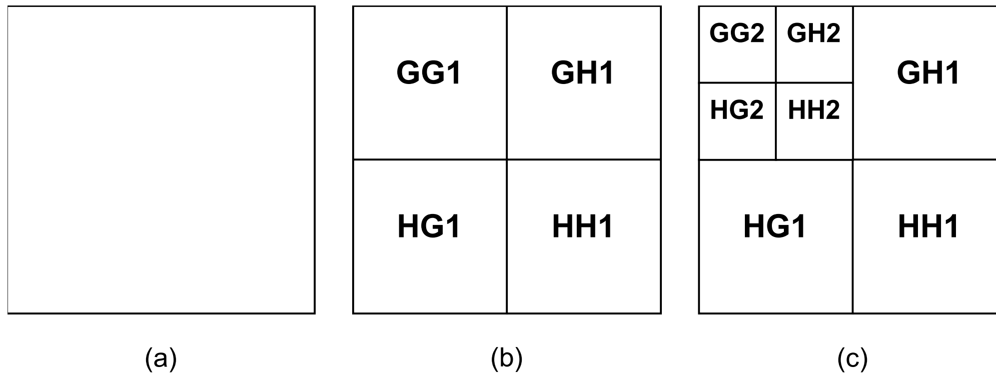


Figura 3.14. a) Imagen Original. b) Subimagenes obtenidas al aplicar un nivel de DWT. c) Subimagenes obtenidas al aplicar dos niveles de la DWT.

Figura 3.15. a) Imagen original. b) Resultado tras aplicar tres niveles de transformada wavelet.

El filtrado de imágenes presenta la problemática del tratamiento de los bordes. Una imagen es una señal finita, ya que estará formada por *N x M* pixeles. El filtro, también se corresponderá con una matriz de *n x m* coeficientes, normalmente mucho más pequeña que la imagen. La operación de filtrado, matemáticamente, consiste en realizar los productos y sumas entre la matriz de coeficientes y una porción del mismo tamaño de la imagen. La matriz de filtrado debe recorrer toda la imagen original. El

principal problema se centra en cómo proceder cuando la matriz de filtrado se acerca a los bordes de la imagen, para ello hay diversas opciones:

- La matriz de filtrado siempre debe estar enteramente contenida en la imagen. Como resultado se produce una reducción del tamaño de la imagen después del filtrado.
- El origen de la matriz de filtrado (normalmente *[n/2, m/2]*) debe estar enteramente contenido en la imagen. El resultado es una imagen del mismo tamaño.
- Algún pixel de la matriz de filtrado toca la imagen. Se produce un aumento de la imagen resultante.

En cualquiera de los casos, debe considerarse que existe un transitorio en los bordes hasta que la matriz de filtrado esté enteramente sobre la imagen. Por supuesto, si no se mantiene el tamaño original de la imagen, no puede compararse la original y la filtrada. En las dos últimas opciones de tratamiento de borde, es necesario dar un valor a los pixeles de los bordes que no se tienen, rellenando con negros, grises o replicando filas y columnas periféricas.

No obstante, el tratamiento de los bordes en la transformada wavelet se ha solucionado en la propia etapa de filtrado. Cuando se aplica la transformada wavelet con coeficientes Daub-4 a un vector finito de 8 elementos $S_0(n)$ con $n=0,...,7$; matemáticamente se puede expresar del siguiente modo:

$$\begin{vmatrix} S_1(0) \\ W_1(0) \\ S_1(1) \\ W_1(1) \\ S_1(2) \\ W_1(2) \\ S_1(3) \\ W_1(3) \end{vmatrix} = \begin{vmatrix} g(1) & g(2) & g(3) & g(4) & & & & \\ h(1) & h(2) & h(3) & h(4) & & & & \\ & & g(1) & g(2) & g(3) & g(4) & & \\ & & h(1) & h(2) & h(3) & h(4) & & \\ & & & & g(1) & g(2) & g(3) & g(4) \\ & & & & h(1) & h(2) & h(3) & h(4) \\ g(3) & g(4) & & & & & g(1) & g(2) \\ h(3) & h(4) & & & & & h(1) & h(2) \end{vmatrix} * \begin{vmatrix} S_0(0) \\ S_0(1) \\ S_0(2) \\ S_0(3) \\ S_0(4) \\ S_0(5) \\ S_0(6) \\ S_0(7) \end{vmatrix} \qquad (3.51)$$

donde $S_1(n)$ es el vector resultante de filtrar la señal paso bajo con el filtro *G(z)* y aplicar el diezmado correspondiente. Análogamente, $W_1(n)$ se obtiene al aplicar en filtro paso alto *H(z)* y diezmar por 2 la señal $S_0(n)$. Si ahora se tiene en cuenta la relación existente entre los coeficientes de *G(z)* y *H(z)*, la matriz queda del siguiente modo:

$$\begin{vmatrix} S_1(0) \\ W_1(0) \\ S_1(1) \\ W_1(1) \\ S_1(2) \\ W_1(2) \\ S_1(3) \\ W_1(3) \end{vmatrix} = \begin{vmatrix} g(1) & g(2) & g(3) & g(4) & & & & \\ g(4) & -g(3) & g(2) & -g(1) & & & & \\ & & g(1) & g(2) & g(3) & g(4) & & \\ & & g(4) & -g(3) & g(2) & -g(1) & & \\ & & & & g(1) & g(2) & g(3) & g(4) \\ & & & & g(4) & -g(3) & g(2) & -g(1) \\ g(3) & g(4) & & & & & g(1) & g(2) \\ g(2) & -g(1) & & & & & g(4) & -g(3) \end{vmatrix} * \begin{vmatrix} S_0(0) \\ S_0(1) \\ S_0(2) \\ S_0(3) \\ S_0(4) \\ S_0(5) \\ S_0(6) \\ S_0(7) \end{vmatrix} \quad (3.52)$$

El proceso de síntesis, consistente en recuperar el vector original $S_0(n)$, se puede aplicar mediante el uso una matriz cuadrada análoga a la anterior:

$$\begin{vmatrix} S_0(0) \\ S_0(1) \\ S_0(2) \\ S_0(3) \\ S_0(4) \\ S_0(5) \\ S_0(6) \\ S_0(7) \end{vmatrix} = \begin{vmatrix} g(1) & g(4) & & & & & g(3) & g(2) \\ g(2) & -g(3) & & & & & g(4) & -g(1) \\ g(3) & g(2) & g(1) & g(4) & & & & \\ g(4) & -g(1) & g(2) & -g(3) & & & & \\ & & g(3) & g(2) & g(1) & g(4) & & \\ & & g(4) & -g(1) & g(2) & -g(3) & & \\ & & & & g(3) & g(2) & g(1) & g(4) \\ & & & & g(4) & -g(1) & g(2) & -g(3) \end{vmatrix} * \begin{vmatrix} S_1(0) \\ W_1(0) \\ S_1(1) \\ W_1(1) \\ S_1(2) \\ W_1(2) \\ S_1(3) \\ W_1(3) \end{vmatrix} \quad (3.53)$$

En esta operación de síntesis destacan dos aspectos importantes: el primero de ellos es el tratamiento de los bordes y el segundo es el proceso de interpolación, el cual difiere ligeramente de lo representado en el diagrama de la Figura 3.10. Las muestras de las señales $S_1(n)$ y $W_1(n)$ se aplican de modo alternativo a ambos filtros ($G_S(z)$ y $H_S(z)$). De esta manera, cada filtro calcula una muestra de la señal original $S_0(n)$, donde $G_S(z)$ dará las muestras pares y $H_S(z)$ las muestras impares.

Con respecto al tratamiento de los bordes, la solución adoptada consiste en que para recuperar la señal $S_0(0)$ hacen falta las señales $S_1(0)$, $W_1(0)$, $S_1(3)$ y $W_1(3)$, las cuales no son consecutivas, sino que son las primeras y las últimas generadas. En el ámbito de las imágenes, se traduce en que para realizar la transformada wavelet es necesario ampliar la imagen por la derecha y por abajo con los pixeles de la izquierda y de arriba. Esta situación complica el hardware, ya que si los pixeles se suministran de izquierda a derecha y de arriba abajo de modo secuencial, será necesario almacenar las primeras filas y columnas para ser utilizadas al final. Sin embargo, se puede plantear la solución de rellenar con negros o con los pixeles vecinos en lugar de utilizar las primeras filas y columnas. El resultado obtenido en este caso, consiste en que el efecto de bordes es apreciable a la vista, tal como aparece en la Figura 3.16.

(a) (b)

Figura 3.16. a) Imagen original. b) Efecto de bordes después de aplicar la DWT y restaurar con IDWT.

3.6 Estudio Comparativo entre Filtros

Como se ha visto anteriormente, el cálculo de la transformada wavelet discreta depende en gran medida de los filtros que se escojan. En la literatura existe una gran variedad de coeficientes, en el apartado 3.4 se han presentado algunos de estos coeficientes. En el presente apartado se pretende escoger un número determinado de distintos filtros y establecer comparaciones evaluando la calidad de las imágenes, el número de operaciones necesarias y el número de bits requeridos.

Los filtros escogidos para realizar el estudio han sido los que presentan una mayor utilidad en el procesado de imágenes [58], [59], [61], [63], [64], [65], [73] y que a continuación se detallan:

- Daub-4, es un filtro ortonormal cuyos coeficientes se obtienen de la Tabla 3.1 haciendo $K = 2$.
- CDF 2/2, son filtros de carácter biortogonal cuyos coeficientes se obtienen de la Tabla 3.2 haciendo $K_F = 2$ y $K_H = 2$.
- CDF 3/1, son filtros de carácter biortogonal cuyos coeficientes se obtienen de la Tabla 3.2 haciendo $K_F = 3$ y $K_H = 1$.
- CDF 9/7, son los filtros biortogonales propuestos en la futura norma JPEG 2000 [64] y que corresponden con las ecuaciones (utilizando la nomenclatura de la Figura 3.9 y de la Figura 3.10):

$$G_S(z) = \frac{\sqrt{2}}{32}\left(-z^{-3} + 9z^{-1} + 16 + 9z - z^{3}\right) \qquad (3.54)$$

$$G(z) = \frac{\sqrt{2}}{64}\left(z^{-4} - 8z^{-2} + 16z^{-1} + 46 + 16z - 8z^{2} + z^{4}\right)$$

El método analítico llevado a cabo para realizar la comparación entre los diferentes filtros, ha consistido en la programación en VHDL de una función que calcula un solo nivel de la DWT de una imagen, utilizando las herramientas que se comentaron en el apartado 1.2. Se toma una imagen de 512 x 512 pixeles con 256 niveles de grises (por ejemplo, Figura 3.15(a)), a la cual se le aplica una descomposición en 4 octavas mediante el uso recurrente de la función de cálculo de la DWT realizada. Una vez obtenidas las 4 octavas, se vuelve a reconstruir la imagen original mediante otra función (realizada también en VHDL) que calcula la DWT inversa. Para establecer una comparación entre la imagen original y la reconstruida, además de la apreciación visual (subjetividad), se ha medido la relación señal ruido por medio de la ecuación (2.3).

Las funciones realizadas en VHDL para el cálculo de la DWT y su inversa, operan con datos tipo bit. Esto significa que los coeficientes de los filtros deben ser escalados y que en las operaciones de cálculo se deberá realizar el redondeo y/o truncamiento oportuno. Las funciones desarrolladas utilizan filtros bidimensionales, por tanto, los filtros unidimensionales que pretenden utilizarse deber ser convertidos a matrices. A continuación se presentan los detalles particulares para cada uno de los filtros escogidos y los resultados obtenidos. La nomenclatura utilizada para la representación de los filtros en el estudio corresponde con la Figura 3.9 y con la Figura 3.10.

3.6.1 Filtro Daub-4

En esta prueba se han utilizado los coeficientes Daub-4 comentados anteriormente. Como tales coeficientes no son enteros, se han multiplicado por un valor entero 2^8 y se han redondeado, obteniendo los siguientes valores:

$$g(0) = 123; \quad g(1) = 213; \quad g(2) = 57; \quad g(3) = -33 \tag{3.55}$$

Las respuestas en frecuencia de los filtros *G(z)* y *H(z)* pueden verse en la Figura 3.17(a) y (b) respectivamente. Tal como se aprecia, se trata de filtros complementarios con una frecuencia de corte en $\omega = \pi/2$ y con una ganancia de 3 dB.

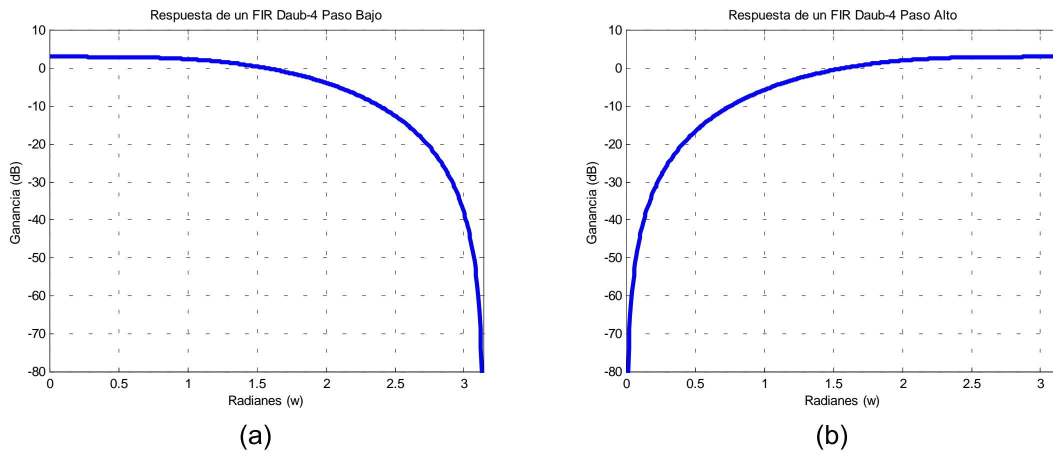


Figura 3.17. Respuesta en frecuencia del filtro de análisis Daub-4.

A partir de los coeficientes mostrados en (3.55) se obtienen las cuatro matrices de los filtros bidimensionales para el análisis y las cuatro de la síntesis. Una vez realizadas las operaciones de filtrado en cada una de las octavas, las señales resultantes se dividen por el factor de escalado 2^{16}, ya que se trata de matrices.

En la Figura 3.18 puede observarse la imagen obtenida después de la recuperación. La relación PSNR es de 26,95 dB, subjetivamente se puede apreciar una reducción del brillo con respecto a la imagen original. Los algoritmos de cálculo de la DWT, para cada octava y cada uno de los filtros, requieren un total de $4N^2$ multiplicaciones y las mismas sumas, siendo N^2 el número de pixeles de la imagen. En el caso de la DWT inversa se requiere el mismo número de operaciones, pero teniendo en cuenta que N^2 es el tamaño de la imagen a recuperar.

Figura 3.18. Imagen procesada con filtros tipo Daub-4.

Un aspecto negativo es que el número de bits necesarios para realizar la DWT se va incrementando tal cómo aumenta el número de octavas o subbandas. En la Figura 3.19, se puede ver cómo evoluciona el número de bits necesarios en la salida de cada uno de los cuatro filtros al incrementar el número de octavas. Para el filtro GG

(el que más información contiene), si inicialmente la imagen se codifica en 9 bits (en CA2), tras realizar la DWT con 4 octavas, se necesitan 13 bits para poder representar la información de la última de las subbandas obtenidas, así se obtiene un incremento de 1 bit por cada octava. Teniendo en cuenta los factores de escalado, los algoritmos de cálculo de la DWT internamente deben trabajar con un máximo de 29 bits, valor que se obtiene experimentalmente tras realizar un ajuste de los bits utilizados en el código VHDL al analizar distintas imágenes.

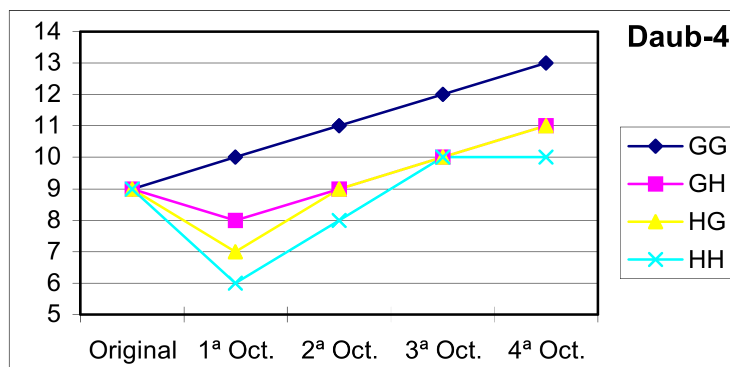


Figura 3.19. Número de bits necesarios, a la salida de cada una de las etapas de filtrado.

3.6.2 Filtro Daub-4 Modificado

Con el fin de que el número de bits no se incremente con el número de octavas, se tomó el mismo algoritmo que en el caso anterior pero intentando acotar la salida de cada DWT. Básicamente, la operación realizada consistió en dividir la salida de la DWT por 2^{17} en lugar de 2^{16} como se había hecho anteriormente. De este modo, la salida del filtro GG de la DWT siempre se limita a 9 bits, como puede verse en la Figura 3.20. Para compensar esta división extra en la transformada inversa, en lugar de dividir por 2^{16} en la salida, se hace por 2^{15}.

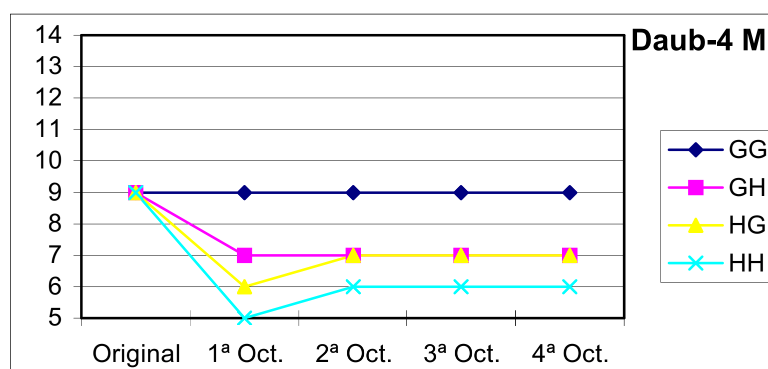


Figura 3.20. Número de bits necesarios, a la salida de cada una de las etapas de filtrado.

En este caso, el número de operaciones son las mismas que en el caso anterior, obteniendo una relación PSNR de 26,91 dB. Como se puede ver en la Figura 3.21 las diferencias entre las imágenes son inapreciables. Por tanto, con esta ligera

modificación se mantiene la calidad de la imagen a la vez que se mantiene constante el número de bits a lo largo de las octavas.

Figura 3.21. Imagen procesada con filtros tipo Daub-4 modificado.

3.6.3 Filtro CDF 3/1

Los filtros CDF 3/1 son filtros biortogonales, con el mismo número de coeficientes que los filtros Daub-4. Estos filtros vienen determinados por las ecuaciones:

$$G_S(z) = \frac{\sqrt{2}}{8}\left(z^{-1} + 3 + 3z + z^2\right)$$

$$G(z) = \frac{\sqrt{2}}{4}\left(-z^{-1} + 3 + 3z - z^2\right) \qquad (3.56)$$

Las respuestas en frecuencia de los filtros *G(z)* y *H(z)* se muestran en la Figura 3.22(a) y (b) respectivamente. Se puede ver que son filtros complementarios, aunque sus respuestas no sean simétricas, tal como ocurre con los filtros Daub-4.

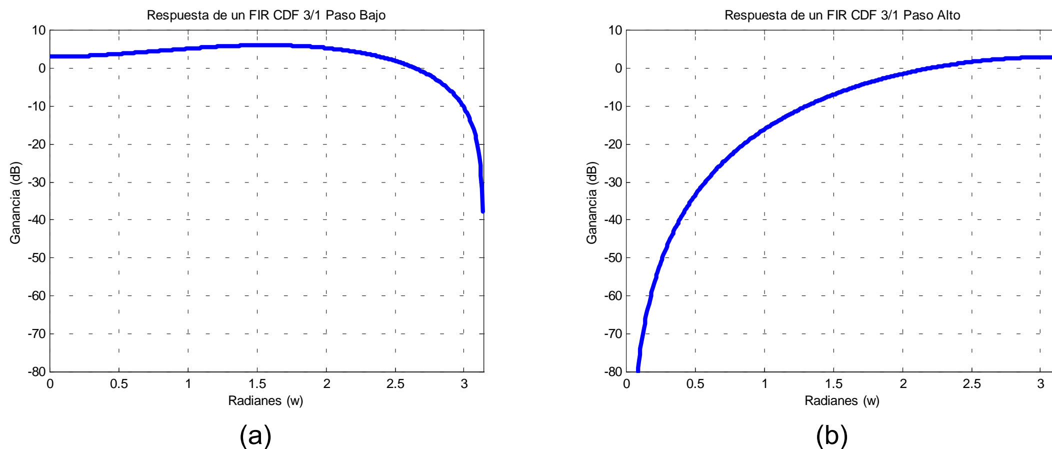


Figura 3.22. Respuesta en frecuencia del filtro de análisis CDF 3/1.

Observando las ecuaciones (3.56) que describen los filtros, se decide escalar los coeficientes por el factor 2^3, lo cual permitirá reducir el número de bits en las operaciones con respecto al filtro Daub-4. Otro aspecto a tener en cuenta en las ecuaciones (3.56), es el factor raíz de 2. Al multiplicar entre sí los coeficientes para construir las cuatro matrices, este factor se convierte en un simple 2, factor común de todas las matrices. Por tanto, una vez aplicado el filtrado, las salidas se deben dividir por 2^6, sin embargo se dividirán por 2^5 debido a que el 2 factor común de las matrices no se ha utilizado en la operación de filtrado.

Los algoritmos de cálculo de la DWT para cada octava y cada uno de los filtros, requieren un total de $4N^2$ multiplicaciones y las mismas sumas, siendo N^2 el número de pixeles de la imagen. En el caso de la DWT inversa se requiere el mismo número de operaciones pero teniendo en cuenta que N^2 es el tamaño de la imagen a recuperar.

En la Figura 3.23 se observa la imagen obtenida después de la recuperación. La relación PSNR es de 52,96 dB. Subjetivamente, no se aprecian apenas diferencias respecto a la original.

Figura 3.23. Imagen procesada con filtro tipo CDF 3/1.

En la Figura 3.19, puede verse cómo evoluciona el número de bits necesarios en la salida de cada uno de los cuatro filtros al incrementar el número de octavas. Para el filtro GG (el que más información contiene), si inicialmente la imagen se codifica en 9 bits (en CA2), tras realizar la DWT con 4 octavas, se necesitan 14 bits para poder representar la información de la última de las subbandas obtenidas, un bit más que en el filtro Daub-4. Teniendo en cuenta los factores de escalado, los algoritmos de cálculo de la DWT internamente deben trabajar con un máximo de 19 bits, valor que se obtiene experimentalmente tras realizar un ajuste de los bits utilizados en el código VHDL al analizar distintas imágenes.

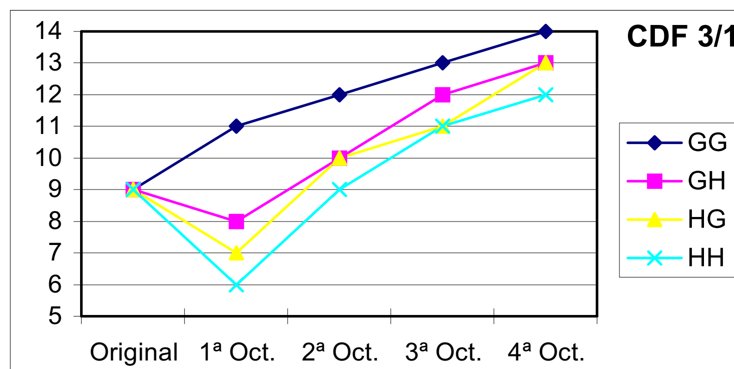


Figura 3.24. Número de bits necesarios, a la salida de cada una de las etapas de filtrado.

3.6.4 Filtro CDF 3/1 Modificado

La misma técnica utilizada con el filtro Daub-4 modificado se aplicó al filtro CDF 3/1, dando lugar a la variante modificada que aquí se presenta. En la Figura 3.25 se puede observar la imagen obtenida después de la recuperación. La relación PSNR es de 42,52 dB, lo cual representa una reducción importante respecto al filtro CDF 3/1 original.

Figura 3.25. Imagen procesada con filtro tipo CDF 3/1 modificado.

Con respecto al número de bits necesarios para implementar el algoritmo, en este caso la DWT necesita hasta 10 bits para cuatro octavas (Figura 3.26), por tanto el número de bits necesarios para realizar los cálculos internos es como máximo de 15 bits.

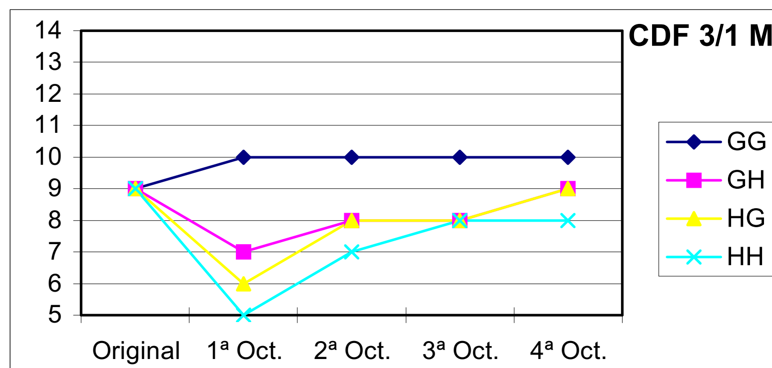


Figura 3.26. Número de bits necesarios, a la salida de cada una de las etapas de filtrado.

3.6.5 Filtro CDF 2/2

Los filtros CDF 2/2 son filtros biortogonales, cuyos coeficientes vienen determinados por las ecuaciones:

$$G_S(z) = \frac{\sqrt{2}}{4}\left(z^{-1} + 2 + z\right)$$

$$G(z) = \frac{\sqrt{2}}{8}\left(-z^{-2} + 2z^{-1} + 6 + 2z - z^{2}\right) \qquad (3.57)$$

Las respuestas en frecuencia de los filtros *G(z)* y *H(z)* se muestran en la Figura 3.27(a) y (b) respectivamente. Se puede ver que son filtros complementarios, aunque sus respuestas no sean completamente simétricas, tal como ocurre con los filtros Daub-4.

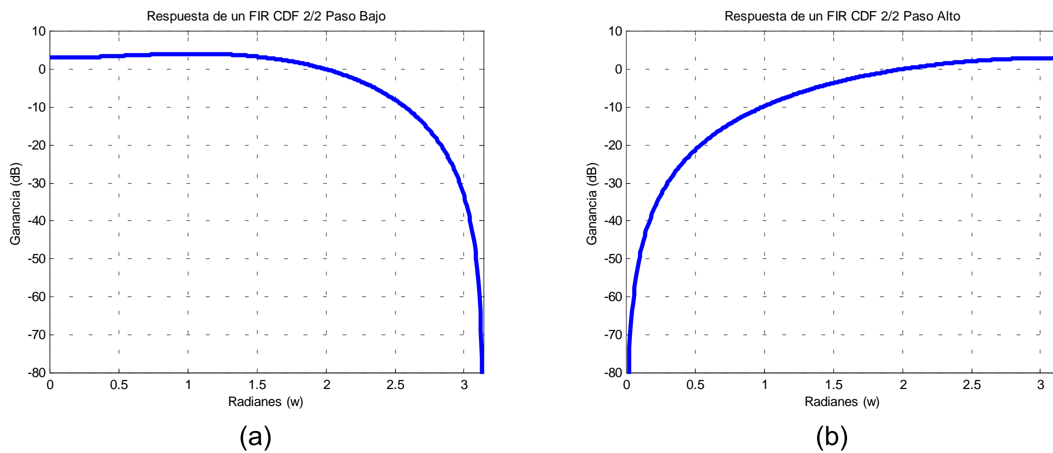


Figura 3.27. Respuesta en frecuencia del filtro de análisis CDF 2/2.

Observando las ecuaciones (3.57) que describen los filtros, se decide utilizar los mismos factores de escalado que en el caso de los filtros CDF 3/1. La única diferencia es que las cuatro matrices resultantes, serán de tamaños distintos y no necesariamente cuadradas.

Los algoritmos de cálculo de la DWT, para cada octava y cada uno de los filtros, requieren un total de $25N^2/4$ multiplicaciones y las mismas sumas, siendo N^2 el número de pixeles de la imagen. En el caso de la DWT inversa, para cada octava y cada uno de los filtros, se requieren un total de $9N^2$ multiplicaciones y las mismas sumas, pero teniendo en cuenta que N^2 es el tamaño de la imagen a recuperar.

Figura 3.28. Imagen procesada con filtro tipo CDF 2/2.

En la Figura 3.28 se puede observar la imagen obtenida después de la recuperación. La relación PSNR es de 52,06 dB. Subjetivamente, no se aprecian apenas diferencias respecto a la original.

En la Figura 3.29, puede verse cómo evoluciona el número de bits necesarios en la salida de cada uno de los cuatro filtros al incrementar el número de octavas. Para el filtro GG (el que más información contiene), si inicialmente la imagen se codifica en 9 bits (en CA2), tras realizar la DWT con 4 octavas, se necesitan 13 bits para poder representar la información de la última de las subbandas obtenidas. Teniendo en cuenta los factores de escalado, los algoritmos de cálculo de la DWT internamente deben trabajar con un máximo de 18 bits, valor que se obtiene experimentalmente tras realizar un ajuste de los bits utilizados en el código VHDL al analizar distintas imágenes.

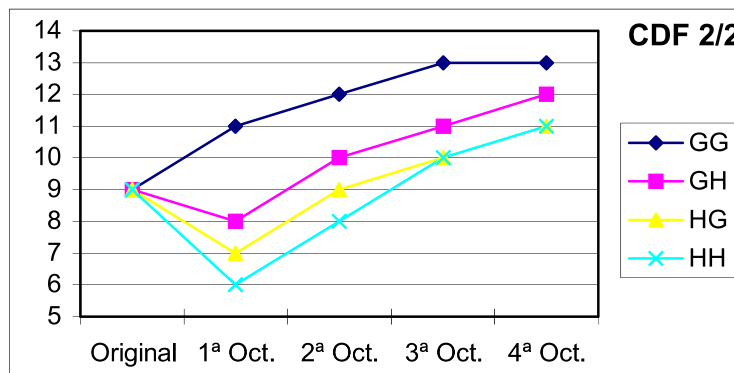


Figura 3.29. Número de bits necesarios, a la salida de cada una de las etapas de filtrado.

3.6.6 Filtro CDF 9/7

Los filtros CDF 9/7 son filtros biortogonales, adoptados por el estándar JPEG2000, cuyos coeficientes vienen determinados por las ecuaciones:

$$G_S(z) = \frac{\sqrt{2}}{32}\left(-z^{-3} + 9z^{-1} + 16 + 9z - z^{3}\right)$$

$$G(z) = \frac{\sqrt{2}}{64}\left(z^{-4} - 8z^{-2} + 16z^{-1} + 46 + 16z - 8z^{2} + z^{4}\right) \quad (3.58)$$

Las respuestas en frecuencia de los filtros *G(z)* y *H(z)* se muestran en la Figura 3.30(a) y (b) respectivamente. Puede verse que son filtros complementarios, aunque sus respuestas no sean completamente simétricas, tal como ocurre con los filtros Daub-4.

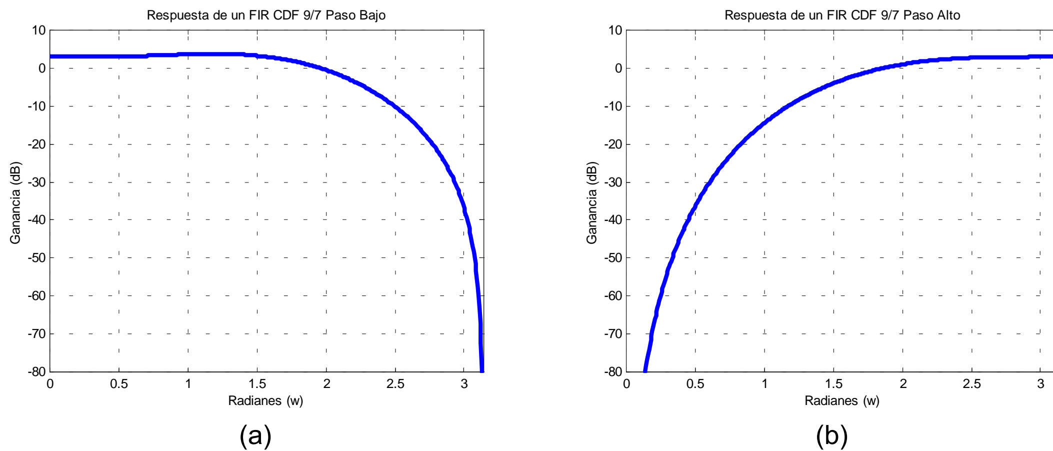


Figura 3.30. Respuesta en frecuencia del filtro de análisis CDF 9/7.

Observando las ecuaciones (3.58) que describen los filtros, se decide escalar los coeficientes por el factor 2^6. Al igual que se hizo con los filtros biortogonales anteriores, el término raíz de 2 hace que las salidas de la DWT deban ser divididas por 2^{11} en lugar de 2^{12}. Al igual que ocurre con los filtros CDF 2/2, las cuatro matrices resultantes no son del mismo tamaño ni tampoco son en todas las ocasiones cuadradas.

Los algoritmos de cálculo de la DWT, para cada octava y cada uno de los filtros, requieren un total de $81N^2/4$ multiplicaciones y las mismas sumas, siendo N^2 el número de pixeles de la imagen. En el caso de la DWT inversa, para cada octava y cada uno de los filtros, se requieren un total de $25N^2$ multiplicaciones y las mismas sumas, pero teniendo en cuenta que N^2 es el tamaño de la imagen a recuperar.

En la Figura 3.31 se observa la imagen obtenida después de la recuperación. La relación PSNR es de 56 dB. Subjetivamente, no se aprecian apenas diferencias respecto a la original.

Figura 3.31. Imagen procesada con filtro tipo CDF 9/7.

En la Figura 3.32, se puede ver cómo evoluciona el número de bits necesarios en la salida de cada uno de los cuatro filtros al incrementar el número de octavas. Para el filtro GG (el que más información contiene), si inicialmente la imagen se codifica en 9 bits (en CA2), tras realizar la DWT con 4 octavas, se necesitan 13 bits para poder representar la información de la última de las subbandas obtenidas. Teniendo en cuenta los factores de escalado, los algoritmos de cálculo de la DWT internamente deben trabajar con un máximo de 24 bits, valor que se obtiene experimentalmente tras realizar un ajuste de los bits utilizados en el código VHDL al analizar distintas imágenes.

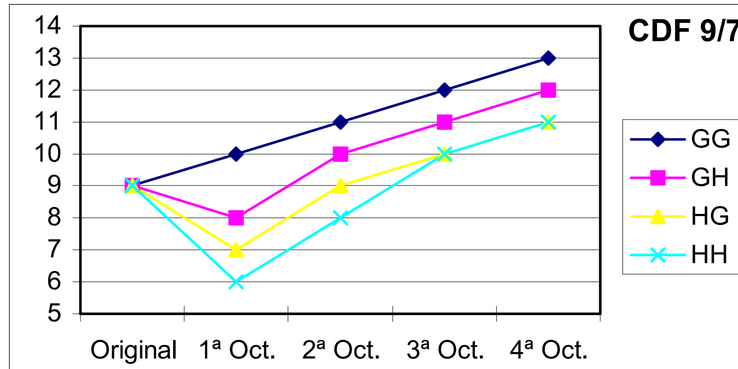


Figura 3.32. Número de bits necesarios, a la salida de cada una de las etapas de filtrado.

3.6.7 Resumen de Resultados

En este apartado se presenta mediante tablas el resumen de los resultados obtenidos en el estudio de los filtros realizado en los apartados anteriores. Estos mismos resultados han sido presentados en [116].

En la Tabla 3.3 se muestran los siguientes resultados por columnas:

1. Denominación del filtro en estudio realizado.
2. Tipo de filtro. Ortonormal o Biortogonal.
3. El número de coeficientes que utiliza el filtro en cuestión. Se indica el número de coeficientes del filtro paso bajo y del paso alto en el análisis. Se consideran filtros de una dimensión, posteriormente hay que convertirlos en las matrices de filtrado.
4. La relación señal-ruido obtenida.
5. El número de multiplicaciones que se realizan en cada filtro (bidimensional) de cada octava en el cálculo de la DWT de una imagen de $N \times N$ pixeles.
6. El número de multiplicaciones que se realizan en cada filtro (bidimensional) de cada octava en el cálculo de la IDWT para reconstruir una imagen de $N \times N$ pixeles.

Tabla 3.3. Resumen de resultados I.

Denominación	Tipo	Número Coeficientes	S/N (dB)	Multiplicaciones en DWT	Multiplicaciones en IDWT
Daub-4	Ortonormal	4/4	26,95	$4N^2$	$4N^2$
Daub-4 M	Ortonormal	4/4	26,91	$4N^2$	$4N^2$
CDF 3/1	Biortogonal	4/4	52,96	$4N^2$	$4N^2$
CDF 3/1 M	Biortogonal	4/4	42,52	$4N^2$	$4N^2$
CDF 2/2	Biortogonal	5/3	52,02	$25N^2/4$	$9N^2$
CDF 9/7	Biortogonal	9/7	56	$81N^2/4$	$25N^2$

En la Tabla 3.4 se muestran los siguientes resultados por columnas:

1. Denominación del filtro.
2. El número de bits que se han empleado para la cuantificación de los coeficientes.
3. El número máximo de bits necesario para la representación de la matriz GG4; es decir, la salida del filtro GG después de la cuarta octava.
4. El número máximo de bits teórico necesario para la realización de las operaciones aritméticas internas de la DWT. Para obtener este valor se han tenido en cuenta los bits iniciales de la imagen y los bits utilizados en la cuantificación de los coeficientes.
5. El número máximo de bits que han sido necesarios en la práctica para la realización de las operaciones aritméticas internas de la DWT sin pérdida de información. Este valor se ha obtenido experimentalmente.
6. En esta columna se pretende reflejar si los coeficientes de los filtros pueden expresarse sencillamente como combinación de sumas y desplazamientos y por tanto eliminar las multiplicaciones.

Tabla 3.4. Resumen de resultados II.

Denominación	Bits Coeficientes	Max. Bits en 4 Octavas	Max. Bits para las Operaciones Teórico	Max. Bits para las Operaciones Práctico	Implementación sin multiplicadores
Daub-4	9	13	30	29	No
Daub-4 M	9	9	26	25	No
CDF 3/1	4	14	21	19	Si

Denominación	Bits Coeficientes	Max. Bits en 4 Octavas	Max. Bits para las Operaciones Teórico	Max. Bits para las Operaciones Práctico	Implementación sin multiplicadores
CDF 3/1 M	4	10	17	15	Si
CDF 2/2	4	13	20	18	Si
CDF 9/7	7	13	26	24	Si

3.7 Conclusiones

La transformada wavelet discreta (DWT) presenta un comportamiento que se asemeja más a la respuesta del ojo humano que la transformada discreta del coseno (DCT), por tanto, parece el método ideal para la compresión de imágenes y vídeo digital. No obstánte, el buen resultado de la DWT depende en gran medida de los filtros que se utilicen.

Para la elección del filtro no sólo hay que evaluar parámetros objetivos como la relación señal-ruido, sino que también hay que tener en cuenta aspectos subjetivos. En esta línea, el estudio realizado demuestra que los filtros biortogonales en general presentan una mejor respuesta que los filtros ortonormales.

Los filtros biortogonales presentaban una mayor relación señal-ruido y una mejor respuesta subjetiva. También permiten disminuir el número de bits necesarios en las operaciones y además sustituir las multiplicaciones por sumas y desplazamientos. No obstante, no todos los filtros biortogonales tienen el mismo número de coeficientes en el paso alto y el paso bajo, lo cual provoca que las matrices de filtrado no sean siempre cuadradas, como ocurre con los filtros CDF 2/2 y CDF 9/7, los cuales además incrementan considerablemente el número de operaciones.

El hecho de que las matrices de filtrado no sean cuadradas y además sean entre sí de dimensiones distintas, no provoca apenas conflictos en una implementación software de la DWT. Sin embargo, cuando se trata de una implementación hardware, se producen retrasos y descompensaciones del flujo de datos que obligan al almacenamiento de los mismos para sincronizar el intercambio de datos entre etapas del diseño.

En cuanto a las variantes de los filtros, modificadas con el fin de acotar la salida, el filtro Daub-4 mantiene su relación señal-ruido, sin embargo, el filtro CDF 3/3 presenta una reducción de 10 dB, lo cual no es admisible. Por tanto, la acotación de la salida no será una técnica a aplicar en los filtros biortogonales.

Capítulo 4

Arquitecturas e Implementaciones de la Wavelet

4.1 Introducción

En este capítulo se presentan las arquitecturas más comunes que se encuentran en la literatura para la implementación de la transformada wavelet. Previamente hay que destacar dos aspectos importantes respecto a las mismas: el primero es que la mayoría de las arquitecturas que se proponen son VLSI, con implementaciones sobre ASIC, no encontrándose apenas información sobre implementaciones en dispositivos programables, como son las FPGA. El otro aspecto a destacar, es que una gran parte de las arquitecturas propuestas para la implementación de la transformada wavelet bidimensional, son extensiones de arquitecturas para la wavelet unidimensional, con lo cual no son lo suficientemente eficientes para el tratamiento de imágenes y vídeo digital.

Teniendo en cuenta lo comentado en el párrafo anterior, el presente capítulo tiene la siguiente estructura: en primer lugar se presentarán algunas de las arquitecturas más comunes para la realización de la transformada wavelet unidimensional, lo cual dará paso al estudio de la extensión de dichas arquitecturas al caso de la wavelet bidimensional. Posteriormente se comentarán las arquitecturas que han sido desarrolladas directamente para el tratamiento de imágenes, por tanto se tratará de arquitecturas pensadas para realizar la wavelet bidimensional. Se concluirá el capítulo haciendo un resumen de resultados de las arquitecturas e implementaciones presentadas.

4.2 Arquitecturas Wavelet Unidimensionales

Tal como se vió en el capítulo 3 la transformada wavelet discreta de una dimensión se describe por medio de las ecuaciones (4.1) y (4.2) y de la Figura 4.1. La señal $S_0(n)$, es la entrada los filtros paso bajo y paso alto, los cuales tienen a la salida un diezmador, provocando así una reducción del número de muestras con respecto a la entrada. La señal $S_1(n)$ tiene la mitad de muestras que la señal $S_0(n)$, lo cual implica que el segundo nivel de la transformada wavelet necesitará a su vez la mitad de operaciones de cálculo que el nivel anterior. Extendiendo este razonamiento el número de operaciones por cada muestra de entrada será aproximadamente 1 + 1/2 + 1/4 + … = 2. Así, para un filtro de longitud *L* el cálculo de la DWT requiere un máximo de *2L* multiplicaciones y *2(L–1)* sumas por cada muestra de entrada.

$$S_{k+1}(n) = \sum_m g(m) S_k(2n - m) \tag{4.1}$$

$$W_{k+1}(n) = \sum_m h(m) S_k(2n - m) \tag{4.2}$$

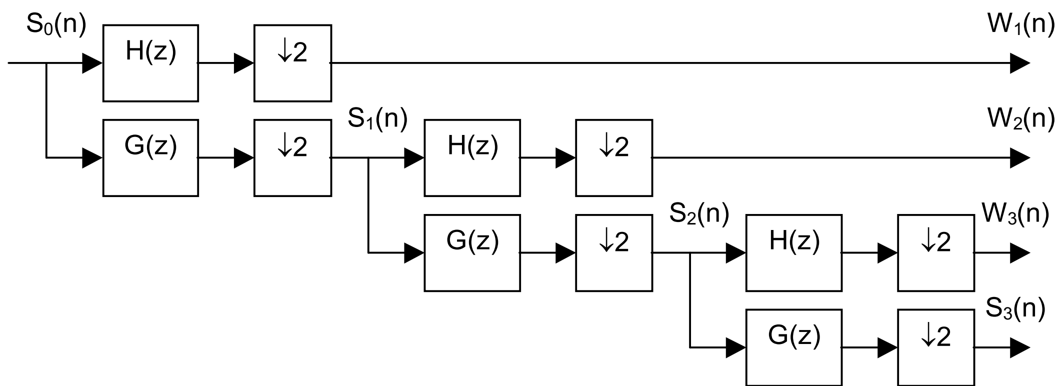


Figura 4.1. Diagrama de bloques del banco en árbol de filtros de análisis de la DWT.

4.2.1 Algoritmos Basados en Técnicas de Filtrado FIR Rápido

El hecho de que el cálculo de la DWT requiera *2L* multiplicaciones y *2(L–1)* sumas por cada muestra de entrada, hace pensar que es suficientemente eficiente. Sin embargo, se puede obtener una reducción adicional del número de operaciones si se utilizan técnicas de filtrado rápido FIR.

El bloque principal del banco de filtros de análisis de la Figura 4.1, se muestra en la Figura 4.2(a). Este bloque consiste en dos filtros de orden *L* y dos diezmadores. Considerando que una de cada dos muestras de entrada es descartada, una implementación sencilla de esta estructura necesitaría *L* multiplicaciones y *L–1* suma por cada muestra de entrada. El filtro paso bajo que viene dado por la expresión:

$$G(z) = \sum_{k=0}^{L-1} g(k) z^{-k} \tag{4.3}$$

se puede descomponer de la forma:

$$G(z) = G_0(z^2) + z^{-1}G_1(z^2) \quad (4.4)$$

donde:

$$G_0(z) = \sum_{k=0}^{L/2-1} g(2k)z^{-k} \quad (4.5)$$

$$G_1(z) = \sum_{k=0}^{L/2-1} g(2k+1)z^{-k} \quad (4.6)$$

De igual modo, el filtro paso alto y la señal de entrada se pueden expresar como:

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2) \quad (4.7)$$

$$X(z) = X_0(z^2) + z^{-1}X_1(z^2) \quad (4.8)$$

Con estas definiciones el bloque de la Figura 4.2(a), se puede reorganizar tal como se muestra en la Figura 4.2(b). Esta nueva estructura consiste en cuatro filtros de orden *L/2* y dos sumadores que necesitan *L* multiplicaciones y *L–1* suma por cada muestra de entrada. El número de operaciones necesarias se puede reducir utilizando las técnicas de filtrado FIR rápido en los filtros $G_0(z)$, $G_1(z)$, $H_0(z)$ y $H_1(z)$.

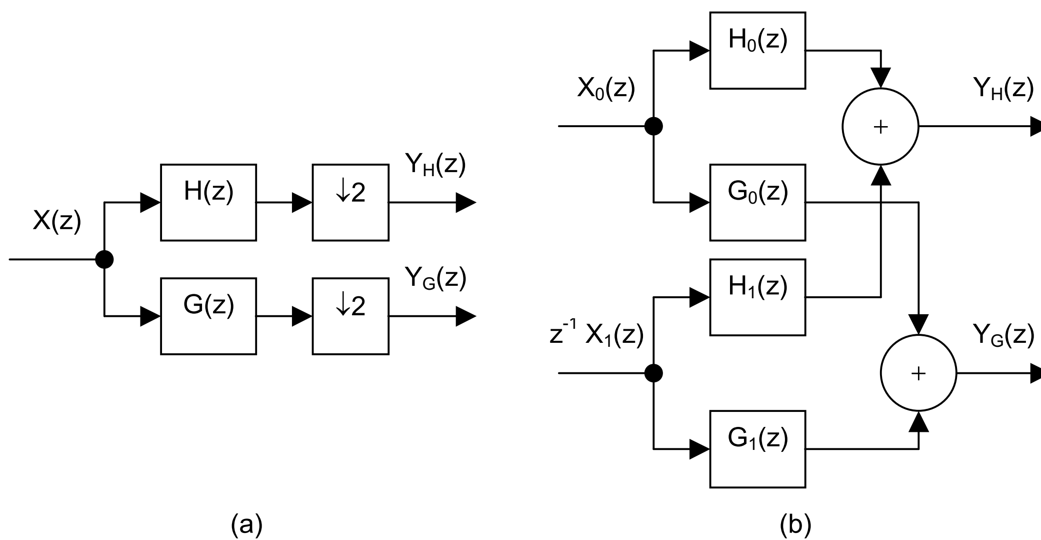


Figura 4.2. (a) Bloque básico del banco de filtros de análisis de la DWT. (b) Reorganización del bloque básico, mediante descomposición polifase.

Considerando un filtro *F(z)* de orden *N* cuya señal de entrada se denomina *U(z)* y la señal de salida *V(z)*, se puede poner que:

$$U(z) = U_0(z^2) + z^{-1}U_1(z^2) \quad (4.9)$$

$$V(z) = V_0(z^2) + z^{-1}V_1(z^2) \quad (4.10)$$

$$F(z) = F_0(z^2) + z^{-1}F_1(z^2) \quad (4.11)$$

Una implementación eficiente de este filtro se puede ver en la Figura 4.3. Mientras que una implementación directa del filtro requeriría *N* multiplicaciones y *N–1* sumas por cada muestra de entrada, la implementación de la Figura 4.3, necesita únicamente *3(N/2*) multiplicaciones y *3(N/2–1)+4* sumas por cada par de entradas; es decir, una media de *3N/4* multiplicaciones y *(3/4)N+1/2* sumas por cada muestra de entrada.

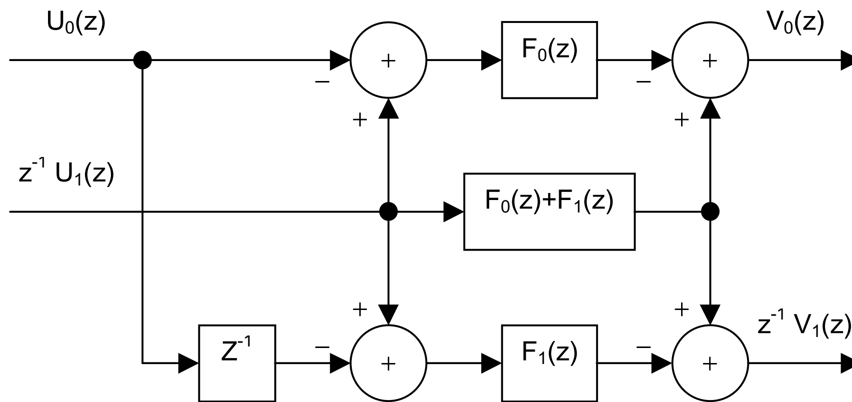


Figura 4.3. Implementación de un filtro FIR usando técnicas de filtrado FIR rápido.

El bloque básico de la Figura 4.2(a), puede implementarse sustituyendo la estructura de la Figura 4.3 en cada uno de los cuatro filtros de orden *L/2* de la Figura 4.2(b). Utilizando esta técnica, según [7] se puede conseguir una reducción del número de operaciones de un 13 % para *L = 8*. Sin embargo, únicamente se utiliza en implementaciones software de la DWT. Las implementaciones VLSI utilizando esta técnica resultan desastrosas, ya que para realizar la DWT con *J* octavas se requieren *J* estructuras como las comentadas y ello significa la utilización de *J* relojes diferentes.

4.2.2 Arquitectura "Bit-Parallel"

En la arquitectura denominada "bit-parallel" todos los bits de la muestra de entrada son procesados en un ciclo de reloj. Este tipo de arquitectura se basa en el concepto de multiplexación temporal. Tal como se ve en las ecuaciones (4.1) y (4.2), en realidad únicamente hacen falta dos filtros para realizar el cálculo de la DWT, el filtro paso bajo *G(z)* y el paso alto *H(z)*. Estos filtros se replican, tal como se observa en la Figura 4.1, para incrementar el número de niveles o de octavas de la transformada wavelet. Con la finalidad de optimizar la eficiencia del hardware, se plantea la utilización de una estructura recurrente para el cálculo de la DWT como la que se muestra en la Figura 4.4. En dicha estructura, únicamente se utilizan un par de filtros *G(z)* y *H(z)* hardware, consiguiendo así una mayor eficiencia del hardware. El par de filtros realizan el cálculo del primer nivel en ciclos de reloj alternos y usan el resto de ciclos para realizar el cálculo de los demás niveles.

La arquitectura "bit-parallel" se basa en el concepto denominado algoritmo piramidal recurrente (RPA), desarrollado por [93]. En otras bibliografías se puede encontrar también bajo la denominación de arquitectura plegada.

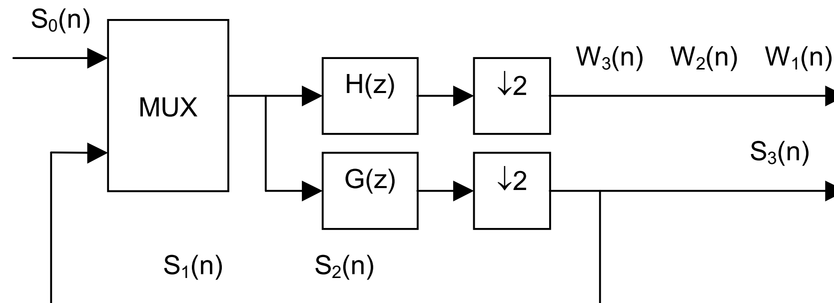


Figura 4.4. Diagrama de bloques de la arquitectura "bit-parallel" de la DWT.

En la implementación de la transformada wavelet discreta, es deseable que la arquitectura tenga las siguientes características:

- Sea una arquitectura escalable, en cuanto a la longitud de los filtros y el número de octavas.
- Sea modular.
- Realice un uso eficiente de la memoria.
- Tenga una alta utilización del hardware.
- Tenga un rutado y control sencillo.
- El área ocupada por el diseño sea independiente del tamaño de la señal de entrada.

Las dos primeras características permiten obtener un diseño fiable y seguro, mientras que las cuatro últimas reducirán el área empleada por el diseño. A continuación se verán algunas arquitecturas que utilizando la estructura "bit-parallel", presentan variantes como pueden ser el implementar los filtros FIR en forma directa o en forma "lattice" o utilizar como elemento de almacenamiento memoria única o memoria distribuida.

4.2.2.1 Arquitectura con filtro FIR en forma directa y memoria única

Cuando se utiliza una estructura recurrente para la realización de la transformada wavelet, como es el caso de la arquitectura "bit-parallel", es necesario utilizar elementos de memoria para el almacenamiento de resultados intermedios. En la Figura 4.5 se muestra cómo queda la arquitectura en el caso de utilizar una única unidad de memoria. En este caso, la señal de entrada $S_0(n)$ entra directamente al bloque de memoria. El bloque de memoria se encarga de suministrar la señal $S_0(n)$ al filtro paso alto $H(z)$ y paso bajo $G(z)$ para realizar el cálculo de $S_1(n)$ y $W_1(n)$. La señal $W_1(n)$ sale al exterior mientras que la señal $S_1(n)$ es realimentada hacia la memoria. De igual modo, la señal $S_1(n)$ permitirá por medio de los filtros obtener las señales de salida $W_2(n)$ y $S_2(n)$, la cual a su vez se vuelve a realimentar hacia la memoria. Así,

sucesivamente hasta concluir con todas las octavas. La salida de los filtros se puede desmultiplexar para obtener las señales de salidas en paralelo.

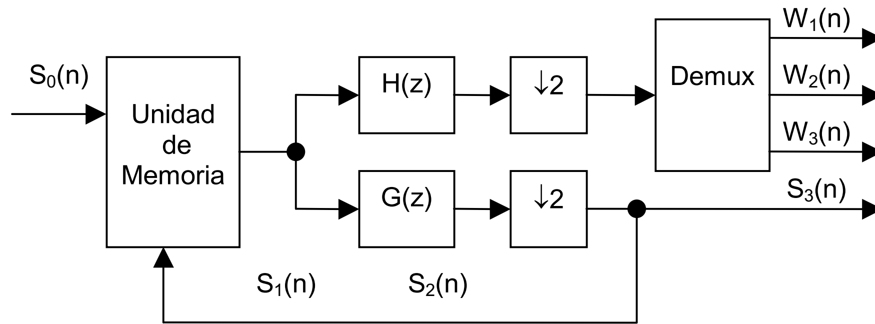


Figura 4.5. Diagrama de bloques de la arquitectura "bit-parallel" con memoria única.

Los filtros *G(z)* y *H(z)* pueden implementarse mediante una estructura en forma directa, como la que se muestra en la Figura 4.6.

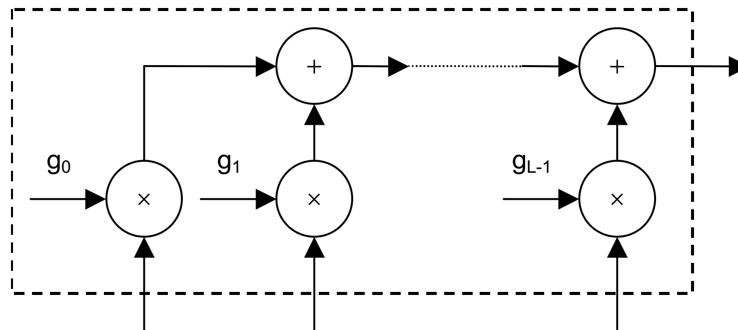


Figura 4.6. Implementación en forma directa de un filtro FIR.

La unidad de memoria de la Figura 4.5, puede implementarse mediante unidades FIFO de longitud *L*, una por cada señal $S_i(n)$. Las memorias FIFO necesitan una señal que actúe desplazando los datos, esta señal será distinta para cada una de las FIFO. La señal de desplazamiento de $S_i(n)$ tendrá una frecuencia doble que la señal de desplazamiento de $S_{i+1}(n)$. Además, a las FIFO hay que añadirles *L* multiplexores para seleccionar la entrada adecuada a los filtros. Esta arquitectura es escalar, modular y tiene una buena utilización de la memoria y del hardware. Su mayor desventaja es la complejidad del rutado.

Sin embargo, esta no es la única forma de implementar la unidad de memoria, existen otras posibilidades que a continuación se comentan.

Con la finalidad de reducir el número de registros necesarios en la unidad de memoria, se puede realizar un estudio del tiempo de vida de las variables, de este modo, se puede utilizar una técnica de ubicación de los registros que permita reducir el número de los mismos. En este caso, se implementan los filtros mediante una estructura "pipeline" de dos etapas. La principal ventaja de esta arquitectura es el uso eficiente de la memoria y del hardware, sin embargo, se trata de una arquitectura que no es escalable ni modular.

Existe la posibilidad de implementar la memoria utilizando una simple memoria RAM, ésto reducirá la actividad de conmutación, ya que los resultados intermedios se almacenan siempre en la misma posición. La principal desventaja es que se necesita memoria RAM con varios puertos de lectura.

Se pueden utilizar memorias con una estructura sistólica o semisistólica. De este modo, se obtiene una unidad de memoria modular pero con elementos de memoria, multiplicadores y sumadores infrautilizados. Además de un sistema de control en absoluto nada trivial.

4.2.2.2 Arquitectura con filtro FIR en forma directa y memoria distribuida

En el apartado anterior se ha utilizado un único bloque de memoria para almacenar los resultados intermedios de la transformada wavelet. Este bloque de memoria se puede fraccionar y distribuir dentro de la arquitectura, lo cual se muestra en la arquitectura presentada en la Figura 4.7. Esta arquitectura es análoga a la mostrada en la Figura 4.5, está formada por un filtro paso bajo *G(z)*, un filtro paso alto *H(z)* y tres registros de desplazamiento para las señales $S_i(n)$ para $i = 0, 1$ y 2. La principal diferencia es que la arquitectura de la Figura 4.7, presenta un rutado más simple que la arquitectura de la Figura 4.5, ya que la memoria se encuentra repartida por la estructura.

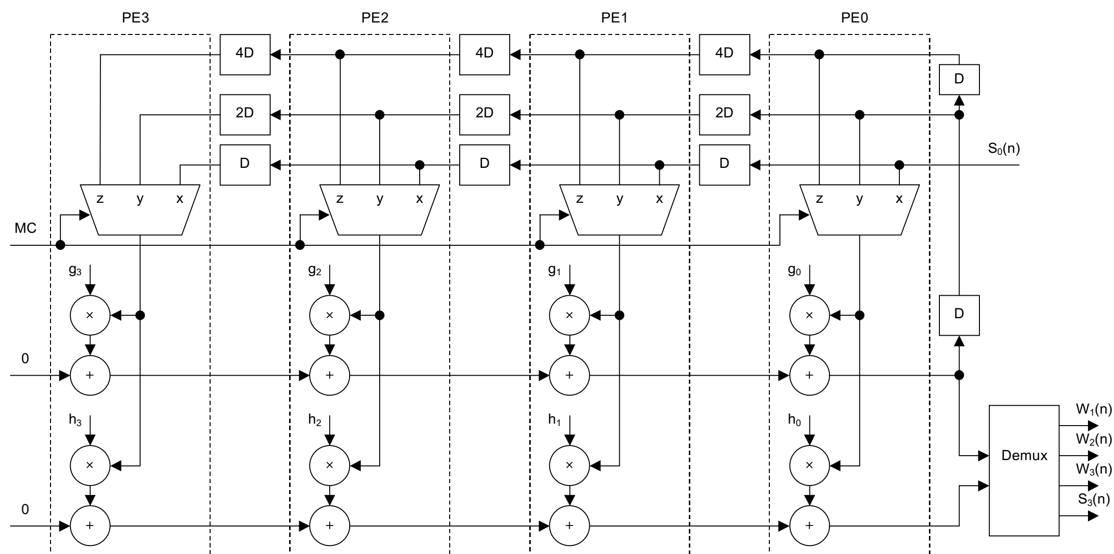


Figura 4.7. Arquitectura de la DWT de tres niveles con filtros de cuarto orden. La señal MC se encarga del control adecuado de los multiplexores.

La arquitectura de la Figura 4.7 es escalable, modular, presenta una buena utilización del hardware y su rutado y control son sencillos. Sin embargo, presenta "path" combinacionales que pasan a través de un multiplicador y cuatro sumadores, lo cual limita la frecuencia máxima de funcionamiento del circuito, e indirectamente limita el orden de los filtros. Un modo de aumentar la frecuencia máxima de trabajo consiste en modificar el circuito utilizando la técnica conocida como "retiming". El "retiming" cambia la ubicación de los registros sin que ello afecte a las características de entrada y salida del circuito.

4.2.2.3 Arquitectura con filtro FIR en forma "lattice"

Los filtros FIR de la transformada wavelet se pueden implementar utilizando una estructura "lattice" o estructura en celosía. La estructura "lattice" presenta una mayor eficiencia en área que la estructura en forma directa, ya que utiliza menos multiplicadores y sumadores. La estructura que aquí se presenta está basada en filtros QMF (Quadrature Mirror Filter). El principal inconveniente de esta arquitectura es que no puede ser utilizada en todos los casos, sino únicamente en los filtros ortonormales de la DWT, entre los cuales se encuentran los popularmente conocidos como de Daubechies (ver Tabla 3.1).

La estructura "lattice" QMF para la DWT de tres niveles, utilizando filtros *G(z)* y *H(z)* de cuarto orden se muestra en la Figura 4.8. Los parámetros α_0, α_1 y *S* son calculados para los filtros *G(z)* y *H(z)*. Se puede observar que mientras que en una aplicación en forma directa se necesitan 8 multiplicadores y 6 sumadores, en la estructura lattice únicamente hacen falta 6 multiplicadores y 4 sumadores por octava.

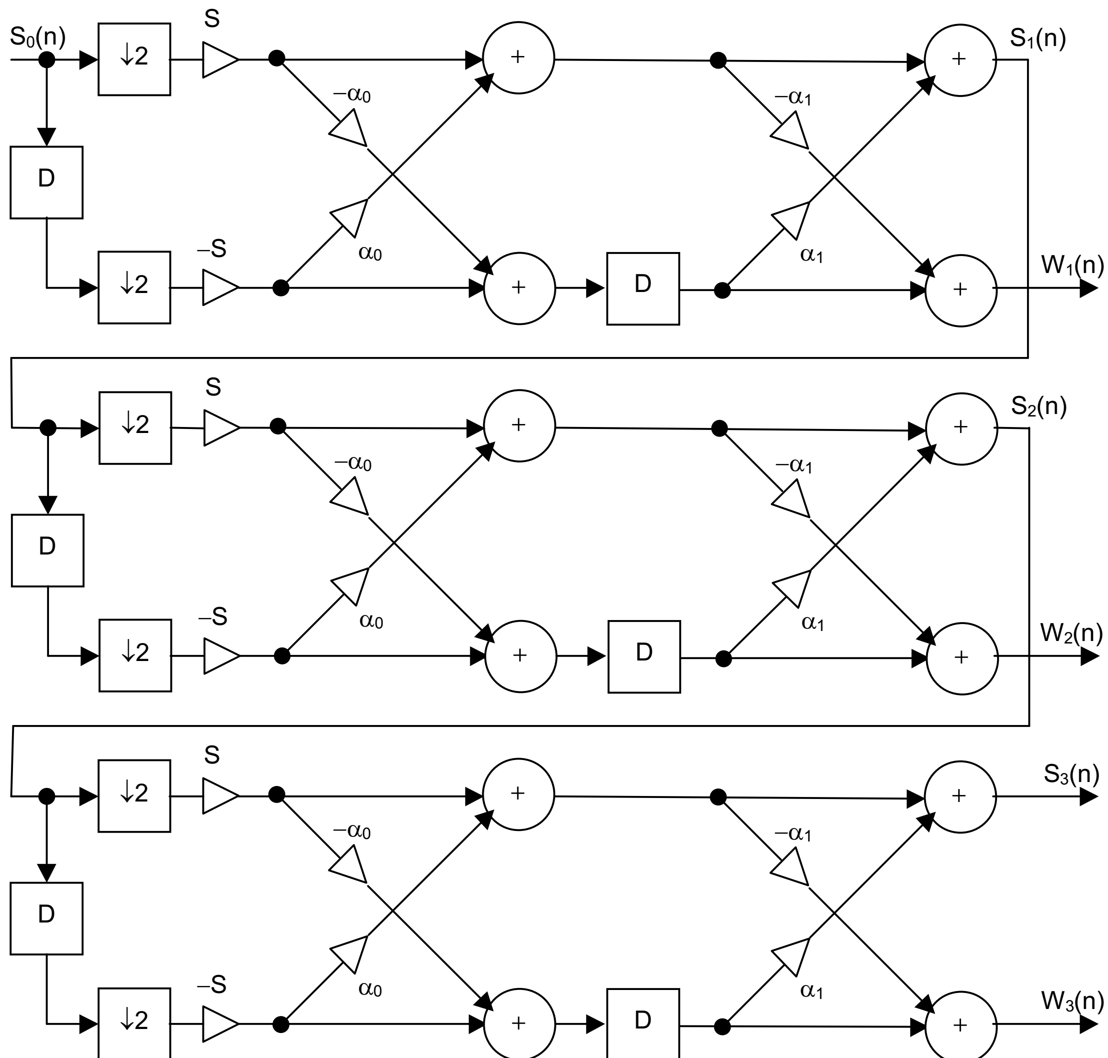


Figura 4.8. Estructura "lattice" que implementa tres niveles de la DWT, con filtros de análisis G(z) y H(z) de cuarto orden.

El ahorro conseguido en multiplicadores y sumadores puede incrementarse si el lugar de realizar una estructura desplegada como la de la Figura 4.8, se utiliza una estructura recurrente como la planteada en la Figura 4.4. De este modo la arquitectura estaría formada por un único filtro "lattice" (en lugar de tres), algunos elementos de memoria y una pequeña unidad de control.

Las ventajas que presenta una estructura "lattice" como la planteada, son su modularidad y su fácil escalabilidad, el alto índice de utilización del hardware y que el área ocupada es independiente de la longitud de la señal de entrada. Además, con pocos multiplicadores y sumadores, tiene la misma funcionalidad que la estructura en forma directa. La principal desventaja de la estructura "lattice", radica en que las señales $S_1(n)$ y $S_2(n)$ deben recorrer un largo camino para ser realimentadas desde la salida a la entrada del filtro.

4.2.3 Arquitectura "Digit-Serial"

Con la finalidad de mejorar la eficiencia de utilización del hardware y reducir la interconexión en las arquitecturas "bit-parallel" vistas anteriormente, puede utilizarse la arquitectura "digit-serial" para la implementación de la transformada wavelet. Un procesador "digit-serial" procesa más de un bit de una muestra, pero no todos los bits de la misma. El número de bits procesados por ciclo se denomina tamaño del dígito (digit-size). Si el tamaño del dígito es 1, la arquitectura se denomina "bit-serial", y si el tamaño del dígito es igual al número de bits que forman la palabra de la muestra, la arquitectura se conoce como "bit-parallel".

La arquitectura "digit-serial" presenta un gran parecido con el algoritmo de cálculo de la transformada wavelet. En una transformada wavelet de tres niveles, las cuatro señales de salida denominadas $W_1(n)$, $W_2(n)$, $W_3(n)$ y $S_3(n)$, generan respectivamente 4, 2, 1 y 1 valores de salida por cada ocho ciclos de reloj. Teniendo en cuenta estas consideraciones, parece natural utilizar, en el procesado de estas cuatro señales, dígitos de tamaño 1/2, 1/4, 1/8 y 1/8 de la palabra de la muestra original. Este tipo de procesado siempre necesitará ocho ciclos de reloj para realizar la transmisión de las salidas. Como mejora destaca la reducción del coste de las interconexiones y la posibilidad de alcanzar la utilización completa del hardware. El inconveniente de esta arquitectura es la variedad de los múltiples tamaños de palabra que se necesitan, así como el incremento de la cadencia de salida de los datos. Sin embargo, la utilización de diferentes tamaños de dígito en los niveles de la wavelet, hace que únicamente sea necesario la utilización de una señal de reloj.

El nivel i de la wavelet se implementará utilizando un tamaño de dígito de $w/2^i$, donde w es el tamaño de la palabra. La Figura 4.9 muestra la estructura a nivel de bloques de la transformada wavelet de tres niveles. Los filtros utilizados en la implementación "digit-serial" pueden ser con una estructura en forma directa (Figura 4.6) o en celosía (Figura 4.8).

El bloque básico en la arquitectura "digit-serial" de la Figura 4.9, es el módulo de conversión entre los bloques de filtrado. La conversión entre el nivel i e $i+1$, convierte datos con una longitud de palabra $w/2^i$ en dos palabras en paralelo cuyo tamaño es la mitad del de entrada. Esta conversión se ha denominado (1, $w/2^i$) → (2, $w/2^{i+1}$) en la Figura 4.9.

La ventaja de utilizar una arquitectura "digit-serial" en el cálculo de la DWT es la utilización completa del hardware y la poca interconectividad, lo cual permite obtener una gran eficiencia en términos de área de silicio. La principal desventaja es que cada nivel de la wavelet es diferente y requiere un mayor tiempo de diseño.

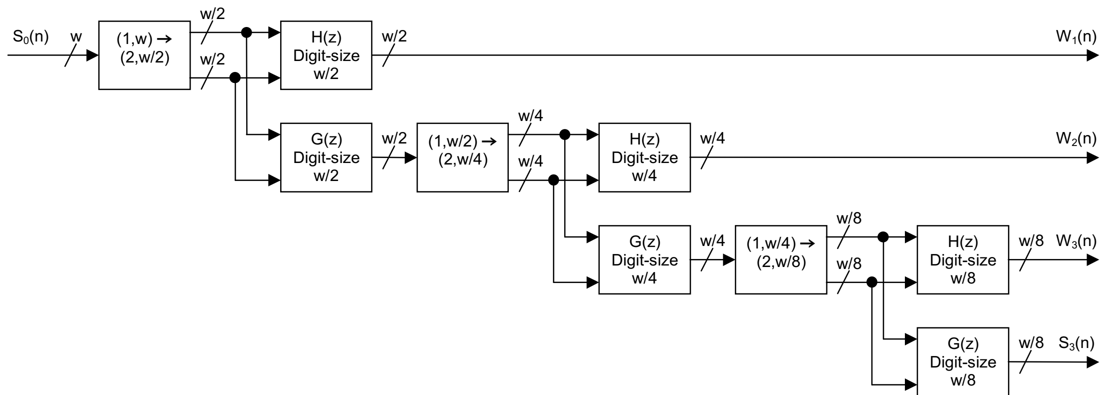


Figura 4.9. Diagrama de bloques de una arquitectura "digit-serial" para la DWT de tres niveles.

4.3 Arquitecturas Wavelet Bidimensionales

Tal como se ha visto en el apartado 3.5, la transformada wavelet bidimensional requiere el uso de filtros bidimensionales. Los filtros 2-D pueden ser separables o no separables, lo cual a su vez da lugar a dos tipos de arquitecturas.

Los filtros separables se pueden descomponer en dos filtros unidimensionales, en cuyo caso el cálculo de la wavelet se descompone en dos etapas de filtrado unidimensionales. La primera etapa consistirá en realizar un filtrado por filas mientras que en la segunda se realizará un filtrado por columnas (ver Figura 4.10). Según ésto se puede plantear una arquitectura para el cálculo de la DWT 2-D, como una mera extensión de la wavelet unidimensional. Es decir, utilizando dos estructuras de wavelet 1-D encadenadas en serie, de tal modo que una realiza el filtrado por filas y la otra por columnas.

Los filtros no separables no se pueden descomponer en filtros unidimensionales, por lo tanto hay que realizar estructuras para la wavelet que realicen directamente el filtrado en dos dimensiones (ver Figura 3.13). La ventaja que presentan las arquitecturas wavelet basadas en filtros de dos dimensiones, es que pueden utilizar tanto filtros separables como filtros no separables, lo cual no es posible en las arquitecturas que se fundamentan en filtros de una dimensión.

En este apartado se presentan dos tipos de arquitecturas wavelet bidimensionales, las que se basan en filtros unidimensionales y las que utilizan filtros de dos dimensiones.

4.3.1 Arquitecturas Basadas en Filtros Unidimensionales

La transformada wavelet en dos dimensiones usando filtros de una sola dimensión se describe gráficamente en la Figura 4.10. En ella se ha representado únicamente un

nivel de la transformada. A continuación, se presentan cinco arquitecturas de diferentes autores con el denominador común de realizar la implementación de la DWT 2-D utilizando filtros de una dimensión o filtros separables.

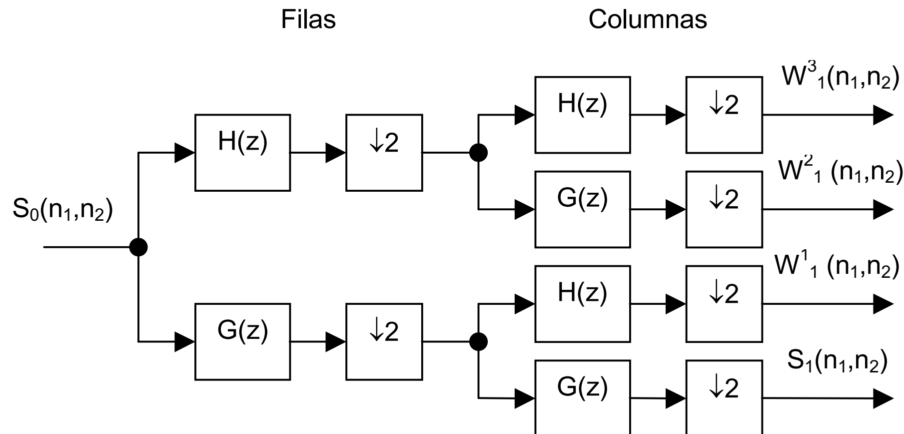


Figura 4.10. Diagrama de bloques de un nivel de la DWT 2-D con filtros separables.

La **primera** aproximación para implementar la DWT 2-D presentada consiste en utilizar dos filtros paso bajo y dos filtros pasa alto. Los filtros se pueden implementar utilizando una arquitectura "bit-parallel" con entrada de datos en serie, los coeficientes de los filtros se incluirán en los multiplicadores utilizando una representación canónica para reducir la complejidad del hardware. Una propuesta de arquitectura basada en esta aproximación es la denominada "systolic-parallel" representada en [31]. La estructura de la arquitectura "systolic-parallel" se muestra en la Figura 4.11.

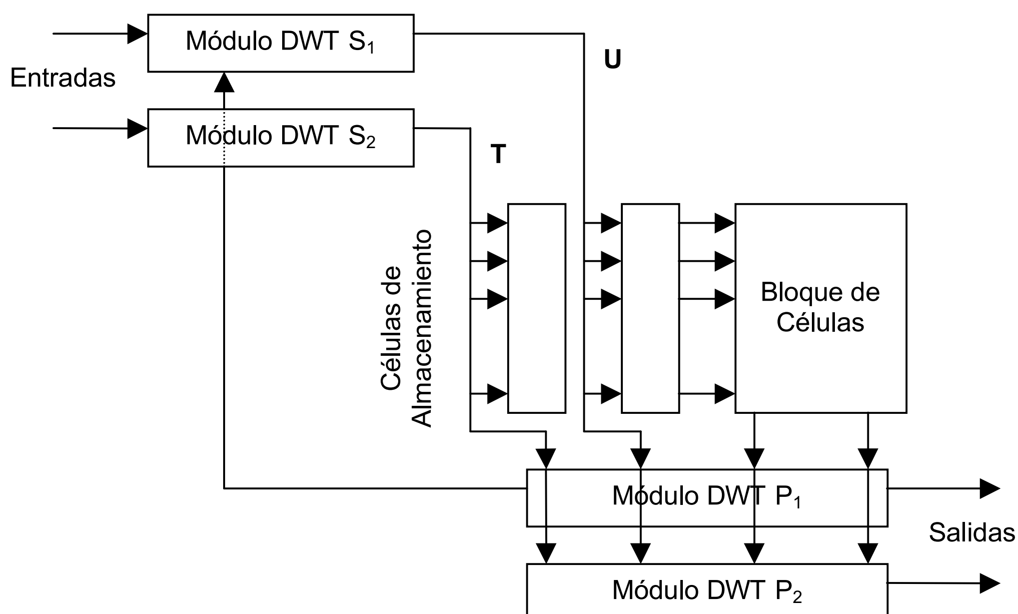


Figura 4.11. Arquitectura "systolic-parallel".

Tal como se ve en la Figura 4.11, la arquitectura "systolic-parallel" presenta una estructura recurrente, está formada por dos filtros paso bajo (S_1 y P_1) y dos filtros paso alto (S_2 y P_2). En primer lugar se realiza un filtrado por filas utilizando los filtros sistólicos denominados S_1 y S_2, a continuación se realiza un filtrado por columnas utilizando filtros que requieren todas las entradas en paralelo (P_1 y P_2), por este motivo es denominada arquitectura "systollic-parallel". La recurrencia del diseño se observa en que la salida del filtro P_1 es realimentada hacia la entrada del filtro S_1. Es necesario utilizar células de almacenamiento entre el filtrado de las filas y el filtrado de las columnas, ya que hasta que no se dispone de la suficiente información no se puede comenzar el cálculo de las columnas.

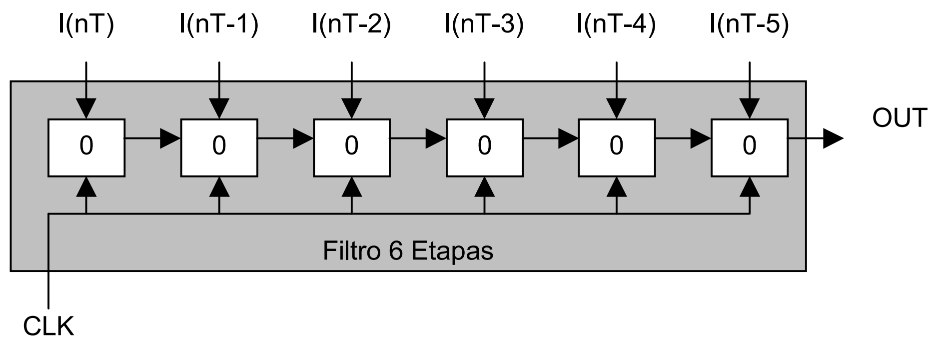
La arquitectura "systolic-parallel" presenta en resumen las siguientes características:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: $2N(L+2)$, siendo L la longitud de los filtros.
- Número de MAC: $2L$.
- Número de Multiplicadores: $4L$.
- Número de Sumadores: $4L$.
- Cadencia de salida de los datos: 1.

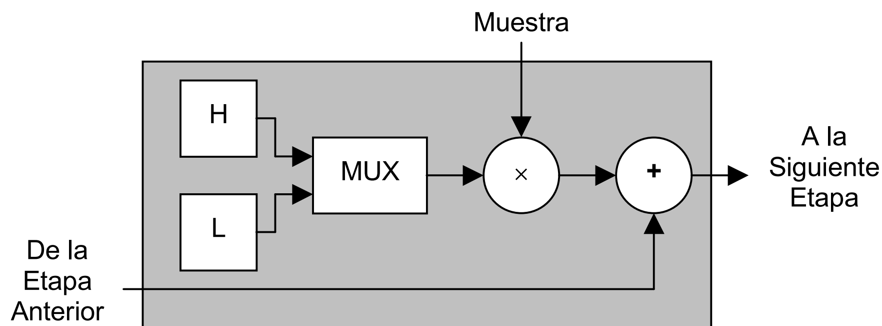
La **segunda** aproximación consiste en utilizar una combinación entre estructuras "bit-parallel" y "digit-serial" como las que se muestran en [38]. La primera arquitectura que se presenta utiliza un filtro "bit-parallel" y otro "bit-serial" de tamaño la mitad de la palabra, para implementar tanto el filtro paso bajo como el paso alto. El primer nivel de cálculo de la wavelet está constituido por dos filtros "bit-parallel" mientras que los niveles restantes utilizan dos filtros "digit-serial" de tamaño la mitad de la palabra. Esta arquitectura utiliza un total de tres filtros y todos los multiplicadores utilizan coeficientes fijos. El otro método presentado en [38], utiliza filtros "bit-parallel" en el primer nivel de calculo de la DWT, mientras que en los demás niveles utiliza filtros "digit-serial" como los de la Figura 4.9. Este método tiene la ventaja de utilizar el 100 % del hardware. Sin embargo, ninguno de los dos métodos presentados en [38], utiliza estructuras recurrentes.

La **tercera** aproximación presentada, consiste en utilizar únicamente estructuras "digit-serial" para la realización de la transformada wavelet. Un ejemplo de este tipo de aproximación se presenta en [27]. La arquitectura mostrada en [27] consiste en filtros sistólicos que utilizan una versión mejorada de la estructura "digit-serial" presentada en la Figura 4.9. La mejora según los autores, consiste en que utiliza una única célula de filtrado para implementar tanto el filtro paso bajo como el filtro paso alto de una sóla dimensión, en lugar de utilizar una estructura de dos filtros en paralelo como ocurre en la Figura 4.9. En la Figura 4.12(a) se muestra la estructura del filtro sistólico de seis etapas utilizado en [27]. Mientras que en la Figura 4.12(b) se puede ver la estructura de cada célula de filtrado. Tal como puede apreciarse hacen falta tantas células de filtrado como etapas tenga el filtro. Los autores indican que los multiplicadores utilizados en este tipo de estructura deben ser muy rápidos. Otra

mejora que presenta, es que no utiliza elementos de memoria ni interna ni externa, sino que utiliza como células de almacenamiento, un banco de registros. La utilización del hardware tiene una efectividad del 85 %.



(a)



(b)

Figura 4.12. Arquitectura sistólica. (a) Filtro de seis etapas. (b) Célula de filtrado.

Para la implementación de la DWT 2-D, los autores de [27], proponen utilizar una arquitectura recurrente, en la que se empleen dos filtros sistólicos como los que se acaban de mostrar. Un filtro realizará el filtrado por filas y el otro realizará el filtrado por columnas. El enlace entre el filtrado por filas y por columnas se realizaría por medio de una célula de almacenamiento, que además se encargaría de hacer una transposición rápida de la matriz resultante.

Los autores de [27], no dan información en detalle de los requerimientos de su propuesta, la única información consiste en indicar que su arquitectura ha sido implementada con tecnología CMOS de 1,2 μm, siendo capaz de realizar la DWT de una imagen monocroma de 512 × 512 pixeles en 13 ms, utilizando un reloj de 20 MHz. A partir de aquí se pueden deducir las siguientes características:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: desconocido.
- Número de Multiplicadores: $2L$, siendo L la longitud de los filtros.

- Número de Sumadores: *2L*.
- Cadencia de salida de los datos: 1.

En la **cuarta** aproximación se implementa una arquitectura en la que se utilizan cuatro filtros paso bajo y otros cuatro paso alto para realizar el cálculo de la DWT 2-D. Los filtros se pueden implementar utilizando una arquitectura "bit-parallel" con entrada de datos en serie. Un ejemplo de este tipo de aproximación es la que se presenta en [40]. La arquitectura propuesta en [40], realiza una modificación del algoritmo piramidal recurrente (RPA). La Figura 4.13 muestra el diagrama de bloques de la DWT 2-D de tres niveles propuesta en [40]. Tal como se observa, se dispone de un bloque H1, que se encarga de realizar el filtrado paso bajo y paso alto por filas del primer nivel únicamente. Mientras que el bloque H2 realiza el filtrado por filas del resto de los niveles, en este caso, del segundo y tercer nivel. Los bloques V1 y V2 realizan los filtrados paso bajo y paso alto por columnas (sentido vertical) de todos los niveles.

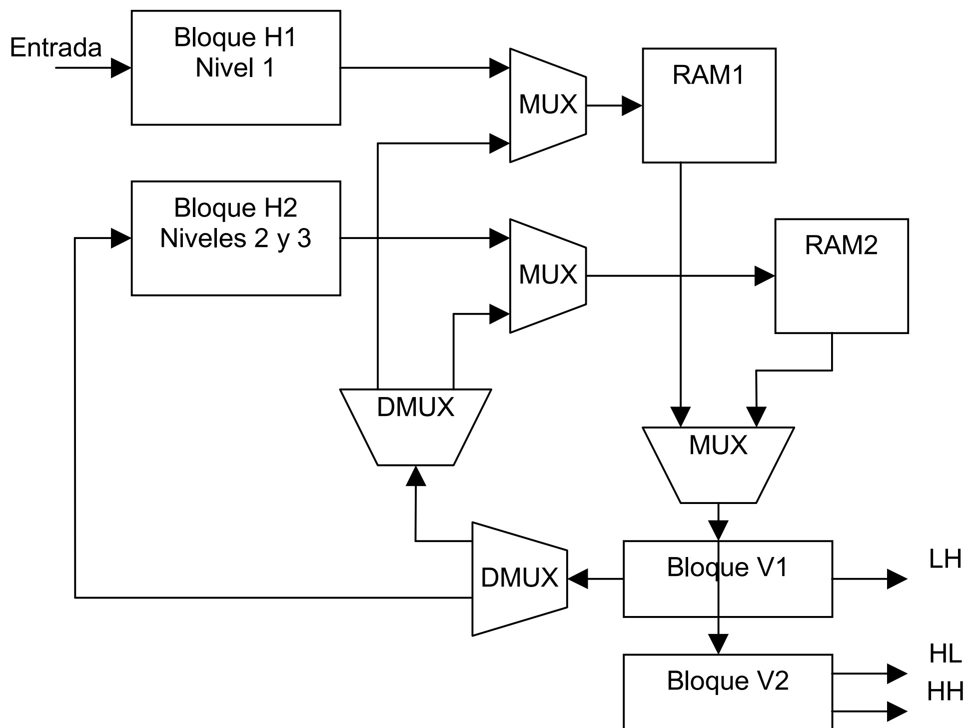


Figura 4.13. Arquitectura para la DWT 2-D de 3 niveles.

Entre las ventajas que presenta esta arquitectura se encuentran la disminución del tamaño de la memoria y la alta efectividad del hardware, alrededor de un 100 %. A continuación se presentan de forma resumida las principales características de la arquitectura mostrada:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: *7N/4*.

- Número de Multiplicadores: *5L/2*, siendo *L* la longitud de los filtros.
- Número de Sumadores: *5L/2*.
- Cadencia de salida de los datos: 1.

Este mismo tipo de arquitectura también se presenta en [96], la cual es denominada por los autores como "Folded". Los resultados obtenidos en este caso son ligeramente distintos, como se muestra a continuación:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: $N/2+2LN(1-(1/2)^J)+N(1-(1/2)^{J-1})$, siendo *L* la longitud de los filtros y *J* el número de niveles de la DWT.
- Número de MAC: *4L*.
- Cadencia de salida de los datos: *N*.

La **quinta** y última aproximación presenta una versión modificada y mejorada de la arquitectura presentada en la aproximación anterior. Esta aproximación se presenta en [96] bajo la denominación de arquitectura "Folded" modificada. En ella se incrementa el número de elementos de filtrado a seis, de los cuales tres son filtros paso bajo y los otros tres filtros paso alto, los cuales se implementan utilizando una estructura "bit-parallel". En el diagrama de bloques de la Figura 4.14 se puede observar la disposición de los filtros.

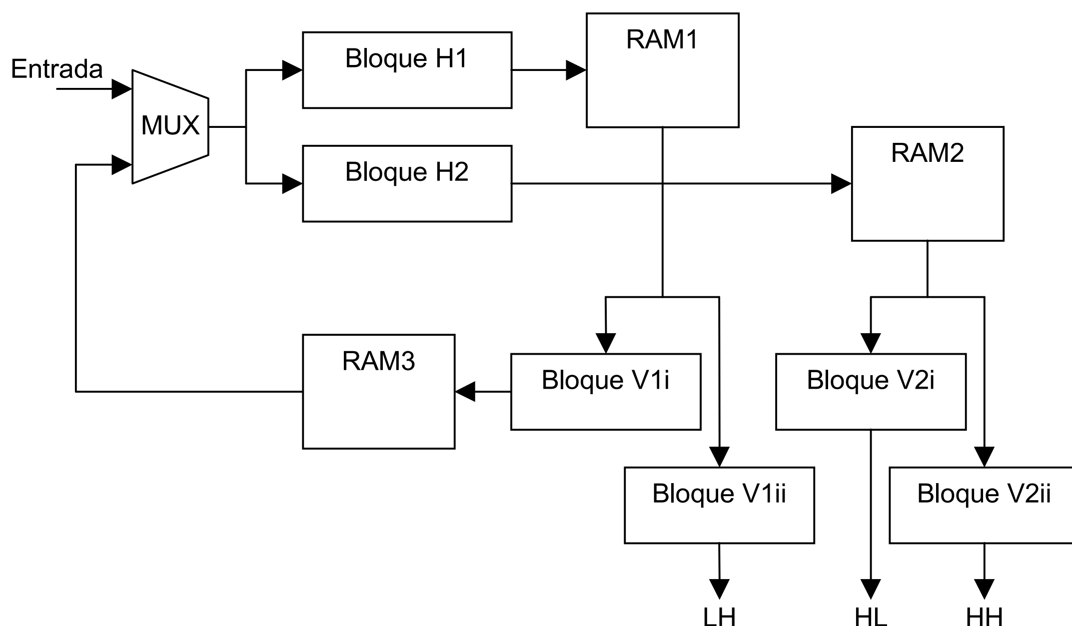


Figura 4.14. Arquitectura para la DWT 2-D "Folded" modificada.

La estructura de la Figura 4.14, se trata de una estructura recurrente, en la que los bloques H1 y H2 realizan el filtrado por filas, mientras que los bloques V1i, V1ii, V2i

y V2ii realizan el filtrado por columnas. En este caso particular los bloques H1, V1i y V2i son filtros paso bajo, mientras que los bloques H2, V1ii y V2ii corresponden con filtros paso alto.

Un aspecto importante a destacar es que en la referencia [96], se hace mención por única vez del efecto de bordes que se produce en el cálculo de la DWT, del cual se habló en el apartado 3.5 (páginas 36 y siguientes). En esta aproximación los autores realizan un rellenado con ceros de los bordes, en lugar de almacenar las primeras filas y columnas, la consecuencia directa es que se reduce el almacenamiento y se simplifica la complejidad del cálculo.

Los resultados obtenidos con la arquitectura "Folded" modificada son los siguientes:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: $2LN(1-(1/2)^J)+N(1-(1/2)^{J-1})$, siendo L la longitud de los filtros y J el número de niveles de la DWT.
- Número de MAC: $6L$.
- Cadencia de salida de los datos: 1.

4.3.2 Arquitecturas Basadas en Filtros Bidimensionales

Los filtros utilizados para la realización de la DWT 2-D pueden ser del tipo no separable, en este caso el diagrama de bloques representado en la Figura 4.10, se convierte en el diagrama de bloques mostrado en la Figura 4.15. En dicha figura se observa que el cálculo de un nivel de la trasformada en dos dimensiones, se realiza utilizando cuatro filtros bidimensionales. Los filtros de dos dimensiones están formados, básicamente, por un conjunto de coeficientes distribuidos en forma de matriz. Por tanto, el filtrado en dos dimensiones de una imagen se reduce al realizar el producto de dos matrices.

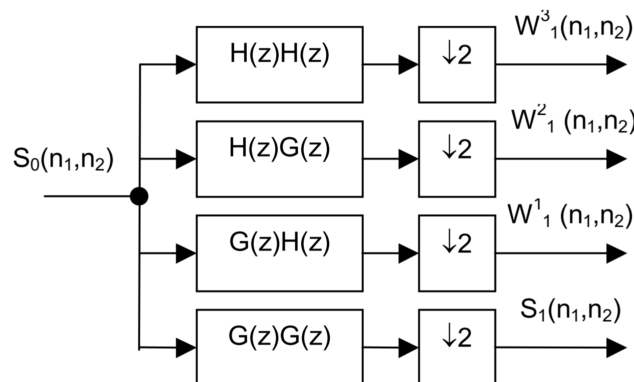


Figura 4.15. Diagrama de bloques para la DWT 2-D con filtros no separables.

El uso de filtros no separables, requiere utilizar arquitecturas específicas para la implementación de la DWT. Las arquitecturas vistas en el apartado anterior no sirven para la implementación de la DWT con filtros no separables. Sin embargo, las arquitecturas basadas en filtros no separables, pueden ser utilizadas en la implementación de la DWT con filtros separables, ya que los coeficientes de los filtros separables pueden convertirse en matrices como las de los filtros no separables.

En este apartado se presentan dos aproximaciones de arquitecturas que realizan la DWT con filtros no separables.

La **primera** aproximación para implementar la transformada wavelet en dos dimensiones utilizando filtros no separables, consiste en utilizar cuatro filtros de dos dimensiones con una estructura recurrente. Los filtros se pueden implementar utilizando una arquitectura "bit-parallel", con entrada de datos en serie. Una propuesta de arquitectura basada en esta aproximación es la presentada [23].

En la Figura 4.16 se muestra el diagrama de bloques de la arquitectura presentada en [23]. En ella puede observarse que se trata de una arquitectura recurrente de tres niveles, en la que se utiliza un único bloque que implementa los filtros de dos dimensiones. También, se utilizan células de almacenamiento para el primer, segundo y tercer nivel de la transformada.

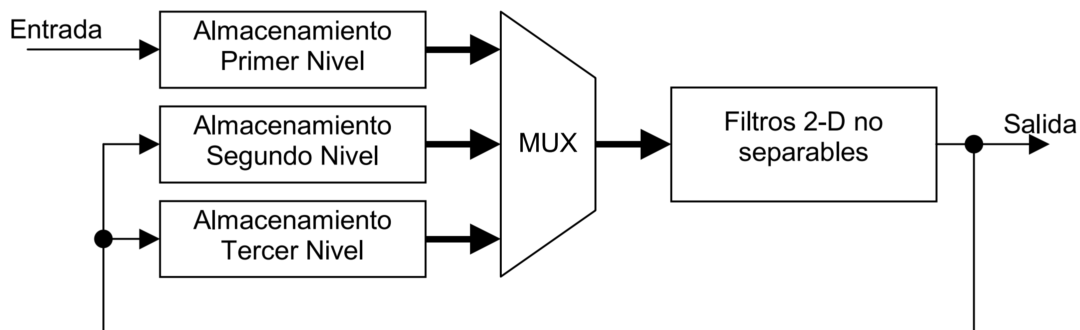


Figura 4.16. Diagrama de bloques de una DWT 2-D de tres niveles con filtros no separables.

El bloque de filtrado incluye los cuatro filtros de dos dimensiones, los cuales han sido implementados con una estructura que los autores denominan "parallel-systolic", es decir, se trata de una arquitectura sistólica con entrada en paralelo. Esto impone que haya que utilizar una célula de almacenamiento previa al primer nivel de DWT, ya que se debe convertir la entrada de datos serie a paralelo. Las otras células de almacenamiento, también son de entrada serie y salida paralelo, aunque se utilizan para almacenar los cálculos intermedios de la transformada.

Suponiendo que los filtros que se desean implementar son matrices de tamaño $L \times L$, la entrada a la etapa de filtrado será de L datos en paralelo. La etapa de filtrado debe incluir los cuatro filtros de dos dimensiones, sin embargo, los autores de [23], utilizan un único elemento de filtrado para implementar los cuatro filtros, para lo cual recurren al uso de un reloj interno cuatro veces más rápido que el reloj externo. Así, se produce un ahorro de área en la implementación, ya que con un solo bloque se implementan los cuatro filtros.

Cada célula de almacenamiento está formada por *L* bloques de memoria RAM interconectados con una estructura en forma de FIFO, de este modo, se producen los desplazamientos en serie de los datos de entrada para poderlos leer en paralelo.

La arquitectura presentada en [23] tiene de forma resumida las siguientes características:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: $N(2L-1)$, siendo *L* la longitud de los filtros.
- Número de MAC: $L^2/2$.
- Número de Sumadores: $L-1$.

La arquitectura presentada en [23], fue implementada con tecnología CMOS de 0,6 μm. Esta implementación es capaz de calcular la DWT 2-D de una imagen de 128 × 128 pixeles con una frecuencia máxima de 55 MHz. Tiene la desventaja de que no es capaz de procesar vídeo en tiempo real, ya que los autores reconocen que para ello necesitan N^2 células de almacenamiento.

La **segunda** aproximación para implementar la transformada wavelet en dos dimensiones utilizando filtros no separables, consiste en utilizar filtros paralelos de dos dimensiones con una estructura recurrente. Los filtros pueden implementarse utilizando una arquitectura "bit-serial". Una propuesta de arquitectura basada en esta aproximación es la presentada en [80].

La arquitectura presentada en [80], fue implementada con tecnología CMOS de 0,35 μm. Esta implementación es capaz de calcular la DWT 2-D de una secuencia de vídeo de 30 cuadros/s con imágenes de 256 × 256 pixeles. La frecuencia máxima de la implementación es de 50 MHz. La arquitectura es programable, puede elegirse el tamaño de los filtros y el número de octavas. Debido a la limitación de la memoria interna, el producto entre el número de niveles y el número de coeficientes del filtro es una constante (*LJ = cte.*).

La arquitectura presentada en [80] tiene de forma resumida las siguientes características:

- Entrada de datos: Serie.
- Número de ciclos de reloj para procesar una imagen de $N \times N$ pixeles: N^2+N.
- Células de almacenamiento: $N(L+1)J$, siendo *L* la longitud de los filtros y *J* el número de niveles.
- Número de Sumadores: $4N$ de un sólo bit.

4.4 Conclusiones

En la Tabla 4.1 se muestran de forma resumida las principales características de las arquitecturas vistas en los apartados anteriores. En la tabla se ha considerado una imagen de $N \times N$ pixeles, con filtros de longitud L ($L \times L$, para no separables) y J niveles de la wavelet.

Tabla 4.1. Resumen de resultados de las arquitecturas.

Implementación	**[31]**	**[27]**	**[40]**	**[96]**	**[23]**	**[80]**
Arquitectura Filtros	"systolic-parallel" "bit-parallel"	"bit-parallel" "digit-serial"	"bit-parallel"	"bit-parallel"	"bit-parallel"	"bit-serial"
Tipo Filtros	Separable	Separable	Separable	Separable	No Separable	No Separable
Cadencia	1	1	1	1	1	
MAC	2L			6L	$L^2/2$	
Multiplicadores	4L	2L	5L/2			
Sumadores	4L	2L	5L/2		L-1	4N de un sólo bit
Células Almacenamiento	2N(L+2)		7N/4	$2LN(1-(1/2)^J) + N(1-(1/2)^{J-1})$	N(2L–1)	N(L+1)J
Frecuencia (MHz)			25		55	50
Tiempo de Cálculo (ciclos)	N^2+N	N^2+N	N^2+N	N^2+N	N^2+N	N^2+N

De la Tabla 4.1, puede extraerse las siguientes conclusiones:

- Todas las arquitecturas estudiadas necesitan del orden de $N^2 + N$ ciclos de reloj para realizar la transformada wavelet en dos dimensiones de una imagen de $N \times N$ pixeles.
- Las arquitecturas basadas en la DWT de una dimensión son las denominadas de filtros separables. Mientras que los filtros no separables son directamente de dos dimensiones. Los filtros no separables son más flexibles, ya que pueden utilizar coeficientes de filtros separables.

- El número de multiplicadores, sumadores o MAC depende directamente del número de coeficientes del filtro. Entre las arquitecturas se puede ver que algunas utilizan MAC en lugar de multiplicadores y sumadores, destacando [31] que utiliza los tres elementos.

- Las células de almacenamiento dependen del tamaño de la imagen ($N \times N$), el número de coeficientes de los filtros (L) y del número de octavas (J). Haciendo $N= 512$, $L = 4$ y $J = 3$, pueden observarse tamaños desde 896 bytes para [40] hasta 7680 bytes en [80].

- El tamaño de las células de almacenamiento condiciona en gran medida, el que una implementación pueda procesar vídeo en tiempo real. La implementación [80], es la única que puede procesar vídeo en tiempo real. La implementación [23], puede procesar vídeo en tiempo real si se incrementa la capacidad de su memoria a N^2 (262.144 bytes).

- Importante destacar que la mayoría de los filtros utilizan una estructura "bit-parallel".

Capítulo 5

Diseño Físico

5.1 Introducción

En este capítulo se describe la arquitectura implementada para el cálculo de la transformada wavelet de dos dimensiones. El capítulo se ha estructurado de tal modo que, previamente a la presentación de la arquitectura definitiva, se muestran diferentes opciones que han sido evaluadas e implementadas, las cuales son un reflejo de la evolución del pensamiento hasta llegar al diseño final. Se parte de un planteamiento muy sencillo, como es la implementación directa del árbol de filtros de la transformada, para progresivamente incrementar la complejidad del diseño hasta alcanzar los requerimientos deseados.

Todos los diseños realizados que aquí se muestran, se fundamentan o intentan cumplir las siguientes premisas:

- Orientado al procesado de imágenes y vídeo digital. El diseño debe permitir realizar el procesado de secuencias de vídeo, por tanto trabajará con señales bidimensionales.
- Procesado de vídeo en tiempo real. El diseño debe ser capaz de procesar una imagen en el tiempo que transcurre entre esa imagen y la siguiente.
- Los bordes de las imágenes deben ser tratados como tales. No es posible rellenar los bordes con ceros u otras técnicas que alteren la imagen original.
- Una imagen es un elemento de dos dimensiones, por tanto, lo más lógico es que la transformada wavelet emplee filtros de dos dimensiones. El diseño realizado se fundamentará en el concepto de filtros no separables vistos en los capítulos anteriores.
- Implementación en FPGA.

- Diseño genérico y modular. El diseño debe ser capaz de poder adaptarse con cierta facilidad a los coeficientes que se requieran en cada aplicación. Debe ser independiente del tamaño de la imagen y modular para poder ampliar el número de octavas.

Teniendo en cuenta estos puntos y utilizando las herramientas de diseño electrónico comentadas en el apartado 1.2 de metodología, se han realizado las implementaciones que se muestran en los siguientes apartados.

5.2 Implementación Directa

La implementación directa consiste en una transcripción literal de los diagramas de las Figuras 4.1 y 4.10, al hardware. Esto significa que se ha realizado una implementación octava a octava, lo cual requiere una gran cantidad de hardware, con una baja utilización en muchas partes del mismo.

En la Figura 5.1 puede verse el diagrama de bloques utilizado para la simulación del sistema que se desea implementar. Tal como se aprecia, se utilizan dos memorias de doble puerto mas un tercer bloque. La memoria Dualmem_img contiene la imagen de partida sobre la que se va a realizar la transformada wavelet. El resultado de realizar la transformada será almacenado en la memoria Dualmem_out. El bloque central denominado Octava_G02 realiza una octava de la transformada wavelet, su comportamiento se describe en el diagrama de bloques de la Figura 5.2.

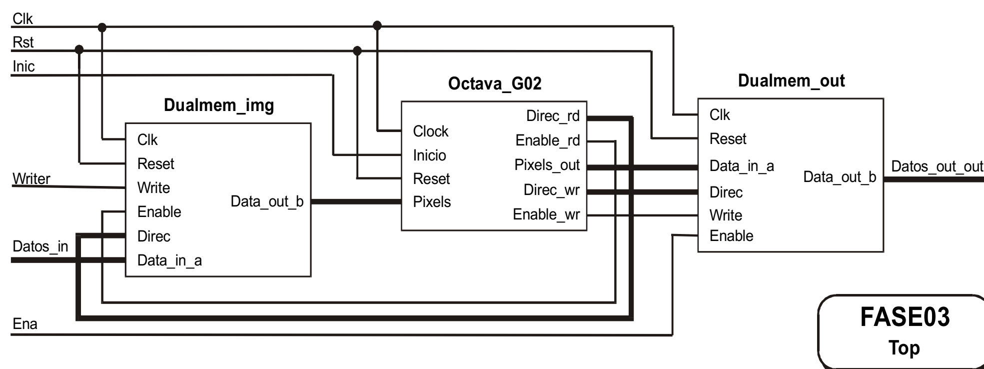


Figura 5.1. Diagrama de bloques del sistema utilizado en la Fase 03 para la simulación de la transformada wavelet.

Octava_G02 lee la imagen de la memoria, pixel a pixel y calcula la transformada wavelet. La unidad de filtrado, proporcionará los resultados de cada elemento de la matriz en serie, por medio de cuatro salidas, cada una de ellas correspondiente a una de las matrices resultantes. Las salidas de la unidad de filtrado serán registras y multiplexadas con la finalidad de almacenar el resultado en la correspondiente memoria.

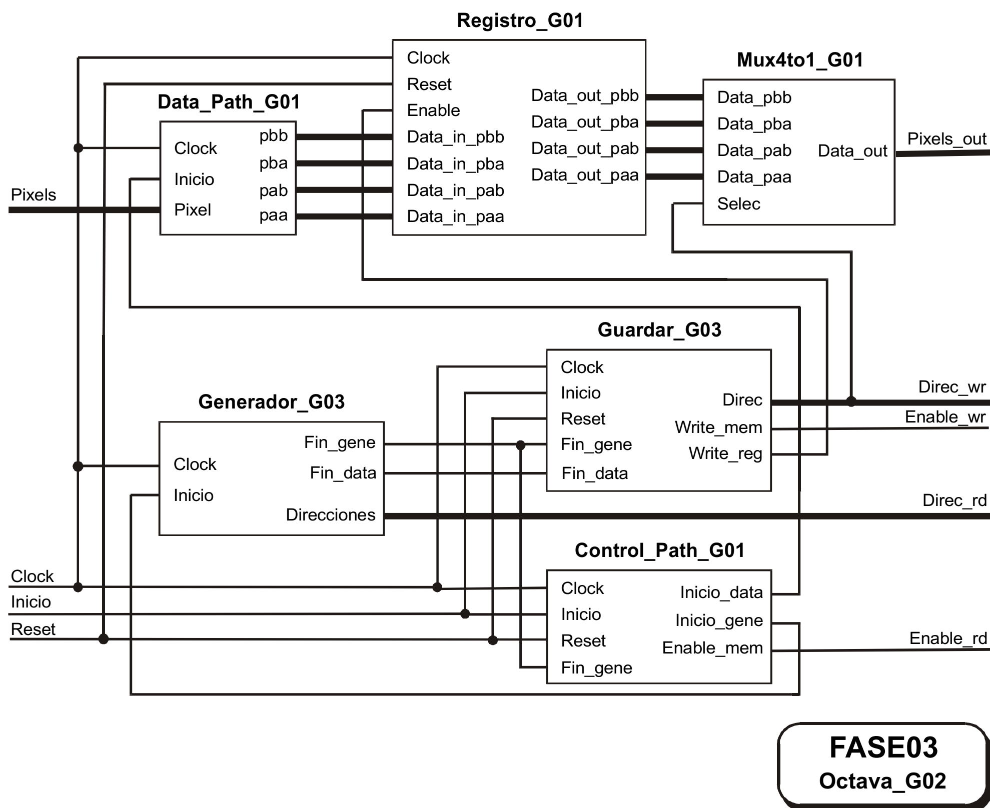


Figura 5.2. Diagrama de bloques de la implementación directa de una octava de la transformada wavelet. Fase 03.

La funcionalidad de cada uno de los bloques de la Figura 5.2, diseñados en VHDL, se describe en los siguientes puntos:

- **Generador_G03**, se encarga de generar las direcciones para leer la memoria que contiene la imagen, indica cuándo se está direccionando el último pixel de la imagen necesario para obtener un dato de salida (Fin_data) y cuándo termina de direccionarse toda la imagen completa (Fin_gene). Hay que tener presente que este bloque se encarga de tener en cuenta aspectos como, el diezmado y la ampliación de la imagen para eliminar el efecto de bordes.

- **Control_Path_G01**, este bloque a partir de la señal de “Inicio”, pone en marcha el generador de direcciones para leer la imagen (Generador_G03) y activar el bloque encargado de realizar las operaciones de filtrado (Data_Path_G01). También detiene el proceso cuando se han completado las operaciones.

- **Data_Path_G01**, es el encargado de realizar las operaciones matemáticas, que realizan el filtrado. A partir de un pixel de entrada, proporciona en paralelo los cuatro pixeles de salida de los filtros. En la Figura 5.3 se muestra el diagrama de bloques funcional de la implementación realizada. Las salidas son registradas (bloque Registro_G01) en el momento adecuado y multiplexadas (bloque Mux4to1_G01) para ser almacenadas en una memoria común.

- **Guardar_G03**, se encarga de generar las direcciones para guardar el resultado final de la transformada wavelet en memoria.

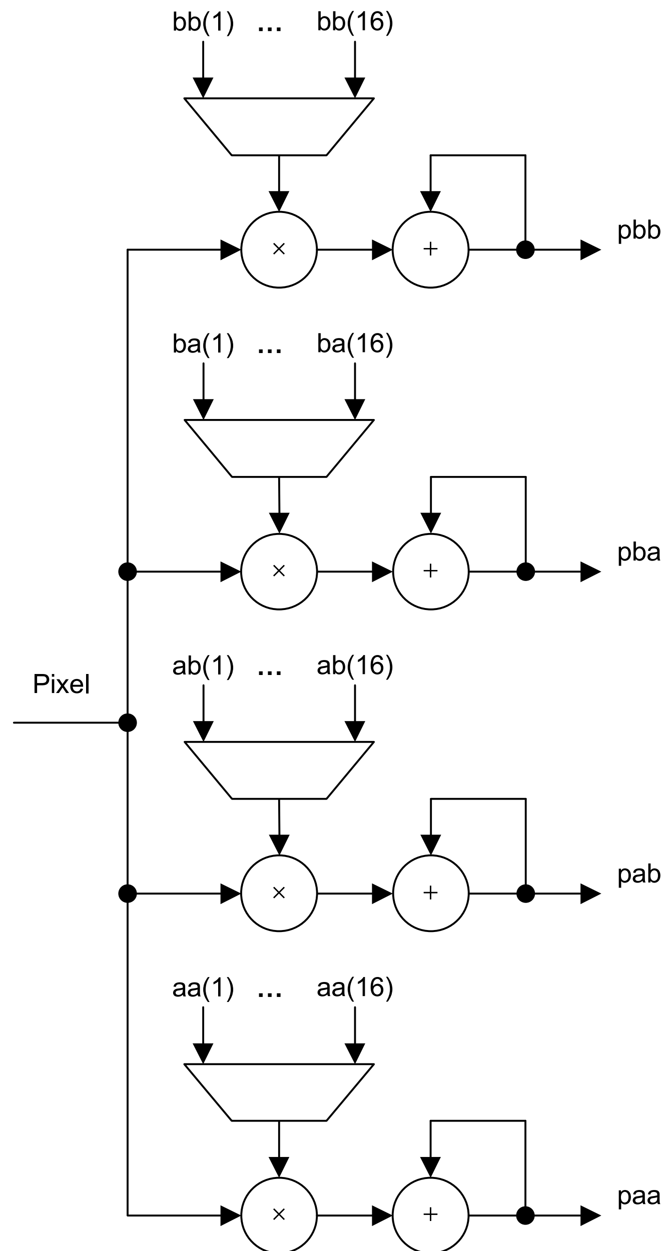


Figura 5.3. Diagrama de bloques funcional de Data_Path_G01.

Tras comprobar su correcto funcionamiento, el bloque Octava_G02 se emplazó y rutó en la FPGA Virtex 200 de Xilinx, obteniendo los siguientes resultados:

- Número de ciclos de reloj: $4N^2$, para una imagen de $N \times N$ pixeles.
- Ocupación en Slices: 431 (18% de una Virtex 200).
- Estimación de la frecuencia máxima de trabajo: 50 MHz.

Según estos resultados, el cálculo de una octava de la trasformada wavelet de una imagen de 512 × 512 pixeles con un reloj de 20 MHz, tardaría aproximadamente alrededor de 52,4 ms.

Para realizar el cálculo de cuatro octavas, sería necesaria la utilización de memorias entre cada una de las octavas, con el fin de almacenar la imagen intermedia, tal como se plantea en el esquema de la Figura 5.4. En este caso una octava puede empezar a calcularse antes de que termine la octava anterior. Sincronizando adecuadamente las octavas (para ello hay que tener en cuenta el tiempo que tarda en generar una octava una fila de la imagen de salida) y teniendo en cuenta los parámetros anteriores, el cálculo de las cuatro octavas de una imagen de 512 x 512 con un reloj de 20 MHz tardaría aproximadamente alrededor de 52,6 ms. Por tanto, si se desea procesar vídeo con un sistema como éste, únicamente se podrían procesar 12 imágenes por segundo (83,3 ms/imagen), aumentando la frecuencia del reloj a 40 MHz se alcanzarían las 24 imágenes por segundo (41,6 ms/imagen).

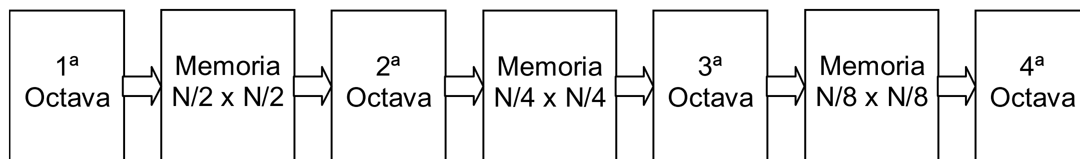


Figura 5.4. Diagrama de bloques de la realización de cuatro octavas.

Esta implementación presenta los siguientes inconvenientes:

- Elevado número de ciclos ($4N^2$), debido a que cada pixel se lee de la memoria cuatro veces, lo cual significa que la limitación radica en el número de accesos a memoria para leer la imagen.
- Para implementar varias octavas se requieren del orden de $21N^2/64$ células de almacenamiento. Por tanto, para una imagen de 512 × 512 se necesitan un total de 86.016 células de memoria (alrededor de 1 Mb), lo cual es demasiada memoria para implementar dentro de una FPGA.
- Utilizando filtros de tamaño $L \times L$, se obtiene una muestra de salida cada L^2 ciclos.
- La baja utilización del hardware. El hardware del primer nivel está ocupado prácticamente el 100% del tiempo, sin embargo, los otros niveles apenas se utilizan.

Por último destacar la posibilidad de utilizar un único bloque de cálculo de la octava para realizar las cuatro octavas, empleando una estructura recurrente, de este modo, se consigue reducir el hardware necesario, sobre todo, el número de células de almacenamiento que pasaría a ser de $N^2/4$ (65.536 células). Sin embargo, los requerimientos temporales empeorarían, ya que el cálculo de las cuatro octavas tardaría $85N^2$ ciclos de reloj. Con un reloj de 20 MHz una imagen de 512 × 512 tardaría 1,11 s en ser procesada.

5.3 Implementación Par-Impar

La implementación Par-Impar pretende reducir el número de ciclos en el cálculo de la wavelet. Como el número de ciclos está directamente relacionado con los accesos a memoria para leer los pixeles de la imagen, se tratará de reducir el número de accesos a memoria. Esta implementación se basa en la idea de procesar en paralelo los pixeles pares e impares, entendiendo por pixeles pares e impares los pixeles que ocupan dichas posiciones en la matriz de salida de la transformada wavelet. Es decir, en lugar de generar la salida pixel a pixel, se obtendrán los pixeles de dos en dos.

Para poner en práctica esta idea, será necesario incrementar el hardware, con lo cual se necesitarán dos unidades de cálculo en paralelo (el denominado Data_Path en el diseño anterior), además hay que cambiar la filosofía de lectura de la memoria (imagen).

Si se consideran filtros de tamaño 4 × 4 y que $P_{in}(i,j)$ es el pixel de la imagen que ocupa la fila *i* y la columna *j*, la implementación directa sigue la secuencia de lectura de la memoria siguiente:

$$P_{in}(1,1) - P_{in}(1,2) - P_{in}(1,3) - P_{in}(1,4) - P_{in}(2,1) - P_{in}(2,2) - P_{in}(2,3) - P_{in}(2,4) - P_{in}(3,1) - P_{in}(3,2) - P_{in}(3,3) - P_{in}(3,4) - P_{in}(4,1) - P_{in}(4,2) - P_{in}(4,3) - P_{in}(4,4)$$

A partir de estos dieciséis valores se obtiene el primero de los pixeles de salida $Pbb_{out}(1,1)$, $Pba_{out}(1,1)$, $Pab_{out}(1,1)$ y $Paa_{out}(1,1)$. Para obtener los siguientes pixeles ($Pbb_{out}(1,2)$, $Pba_{out}(1,2)$, $Pab_{out}(1,2)$ y $Paa_{out}(1,2)$), hay que tomar dieciséis nuevos valores de los cuales ocho coinciden con los anteriores, según se ve en la siguiente secuencia:

$$P_{in}(1,3) - P_{in}(1,4) - P_{in}(1,5) - P_{in}(1,6) - P_{in}(2,3) - P_{in}(2,4) - P_{in}(2,5) - P_{in}(2,6) - P_{in}(3,3) - P_{in}(3,4) - P_{in}(3,5) - P_{in}(3,6) - P_{in}(4,3) - P_{in}(4,4) - P_{in}(4,5) - P_{in}(4,6)$$

En la implementación Par-Impar la secuencia de lectura será tal como se describe a continuación:

$$P_{in}(1,1) - P_{in}(2,1) - P_{in}(3,1) - P_{in}(4,1) - P_{in}(1,2) - P_{in}(2,2) - P_{in}(3,2) - P_{in}(4,2) - P_{in}(1,3) - P_{in}(2,3) - P_{in}(3,3) - P_{in}(4,3) - P_{in}(1,4) - P_{in}(2,4) - P_{in}(3,4) - P_{in}(4,4) - P_{in}(1,5) - P_{in}(2,5) - P_{in}(3,5) - P_{in}(4,5) - P_{in}(1,6) - P_{in}(2,6) - P_{in}(3,6) - P_{in}(4,6)$$

De este modo, después de leer el pixel $P_{in}(4,4)$ (16 ciclos después de comenzar el procesado) se obtienen los pixeles de salida $Pbb_{out}(1,1)$, $Pba_{out}(1,1)$, $Pab_{out}(1,1)$ y $Paa_{out}(1,1)$. Tras la lectura del pixel $P_{in}(4,6)$ (8 ciclos más tarde), se obtienen los datos $Pbb_{out}(1,2)$, $Pba_{out}(1,2)$, $Pab_{out}(1,2)$ y $Paa_{out}(1,2)$. A partir de este momento se obtendrán datos de salida cada ocho ciclos de reloj. Para que el sistema funcione adecuadamente, será necesario disponer de dos unidades de cálculo, la primera de ellas comenzará a operar con el primer pixel y calculará los pixeles de salida pares (por ejemplo, $Pbb_{out}(1,1)$, $Pba_{out}(1,1)$, $Pab_{out}(1,1)$ y $Paa_{out}(1,1)$), la segunda unidad comenzará el cálculo ocho pixeles más tarde, es decir, al leer $P_{in}(1,3)$, calculando así los pixeles de salida impares (por ejemplo, $Pbb_{out}(1,2)$, $Pba_{out}(1,2)$, $Pab_{out}(1,2)$ y $Paa_{out}(1,2)$).

La Figura 5.5 muestra el esquema de la simulación de la implementación Par-Impar. Tal como se ve, se ha utilizado la misma estructura que en el apartado anterior. La Figura 5.6 muestra el diagrama de bloques de la implementación Par-Impar,

denominada Octava_G22. La estructura es idéntica a la implementación directa, salvo por la existencia de dos Data_Path_G01 o unidades de cálculo.

La unidad de generación de las direcciones de lectura de la memoria (Generador_G05), ha sido convenientemente modificada para realizar la nueva secuencia de lectura, además se han añadido las señales de sincronismo necesarias, no sólo por la duplicidad de unidades de cálculo, sino porque también hace falta conocer cuando termina de calcularse una fila de pixeles.

La unidad de control (Control_Path_G02) también ha tenido que ser convenientemente modificada para atender la nueva estructura del diseño.

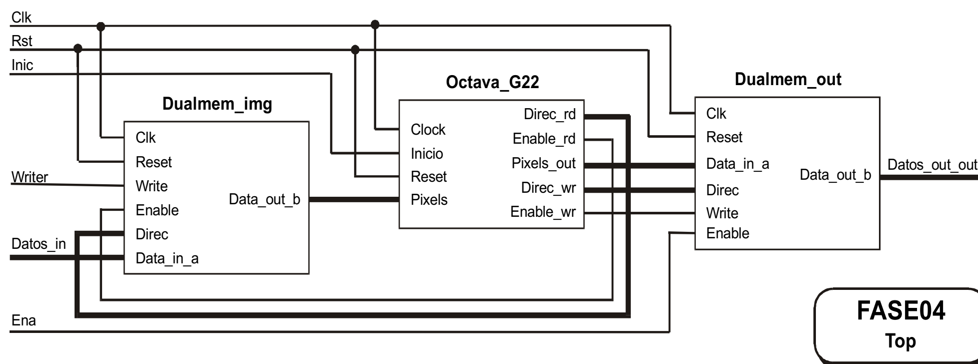


Figura 5.5. Diagrama de bloques del sistema utilizado en la Fase 04 para la simulación de la transformada wavelet.

Tras comprobar su correcto funcionamiento, el bloque Octava_G22 se emplazó y rutó en la FPGA Virtex 200 de Xilinx, obteniendo los siguientes resultados:

- Número de ciclos de reloj: $2N^2+4N$, para una imagen de $N \times N$ pixeles.
- Ocupación en Slices: 751 (32% de una Virtex 200).
- Estimación de la frecuencia máxima de trabajo: 51 MHz.

Según estos resultados, el cálculo de una octava de la trasformada wavelet de una imagen de 512×512 pixeles con un reloj de 20 MHz, tardaría aproximadamente alrededor de 26,3 ms, lo cual significa una reducción del 50% del tiempo de cálculo con respecto a la implementación directa. No obstante, se sigue manteniendo el problema del uso excesivo de memoria para incrementar el número de octavas y la baja utilización del hardware a partir del segundo nivel.

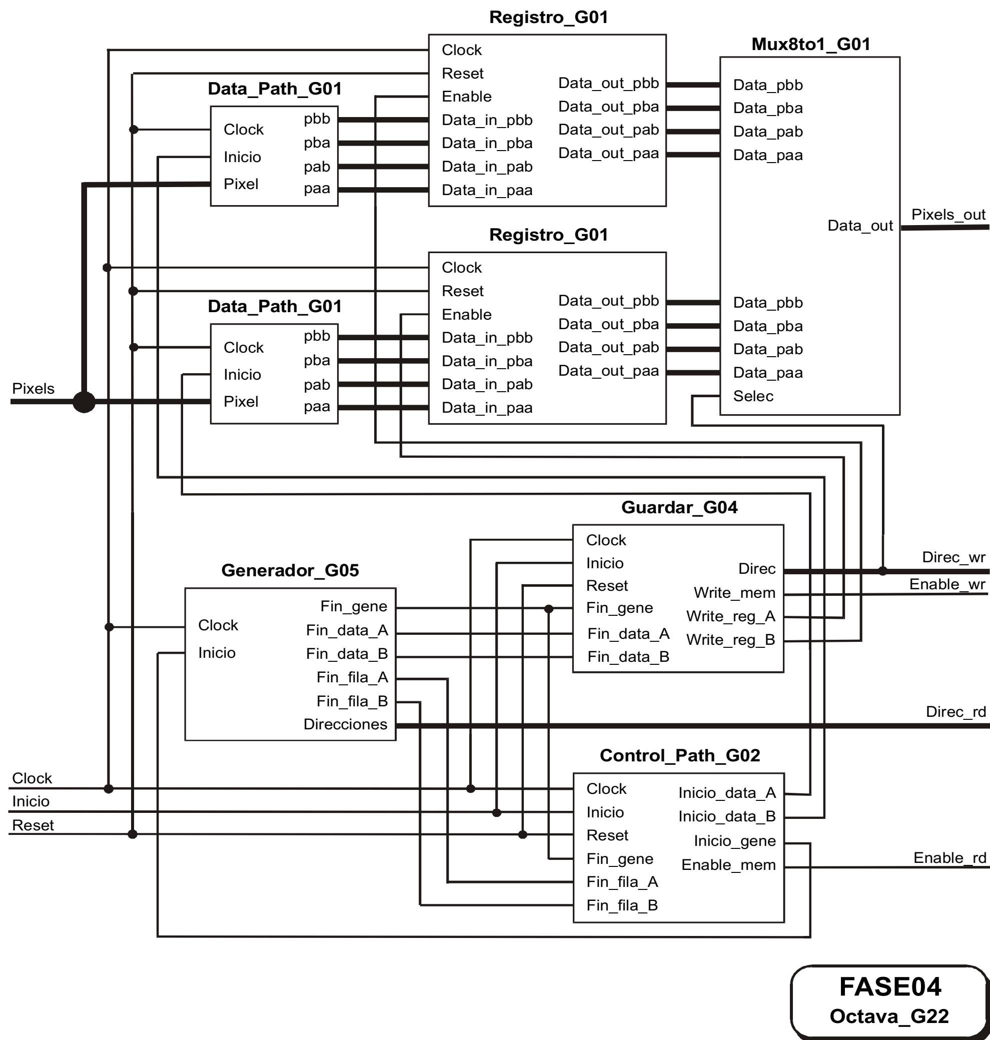


Figura 5.6. Diagrama de bloques de la implementación Par-Impar de una octava de la Transformada Wavelet.

5.4 Implementación Par-Impar con Minimización de Memoria

La implementación anterior reduce a la mitad el número de ciclos con respecto a la implementación directa; sin embargo, se sigue necesitando una cantidad excesiva de memoria para poder realizar más de una octava. La pregunta fundamental que se plantea para poder resolver esta cuestión es: ¿realmente se necesita tener almacenada toda la imagen para realizar la siguiente octava? Debido a que una imagen es un elemento finito, acotado, para generar la última columna de la transformada wavelet es necesario disponer de las dos primeras columnas de la imagen de entrada, del mismo modo para generar la última fila es necesario disponer de las dos primeras filas de la imagen de entrada, por estos motivos se cree que es necesario almacenar toda la imagen de la transformada wavelet para realizar la siguiente octava.

En realidad lo que se necesita para realizar la wavelet, tal como se ha planteado en la implementación anterior, es disponer de cuatro filas de la imagen. Si a esto se añade la necesidad, como se ha comentado, de mantener las dos primeras filas para el final, en realidad sólo hacen falta seis filas. Ahora bien, hay que tener presente que se deben concatenar dos octavas, por tanto, se necesita un pequeño margen de variación entre la octava que genera la imagen y aquella que la lee. En definitiva, únicamente almacenando ocho filas es suficiente para poder realizar la transformada wavelet de varias octavas en tiempo real. De este modo, el diagrama de bloques de la Figura 5.4, se convierte en el que puede verse en la Figura 5.7, en el cual sólo son necesarias 7N células de almacenamiento.

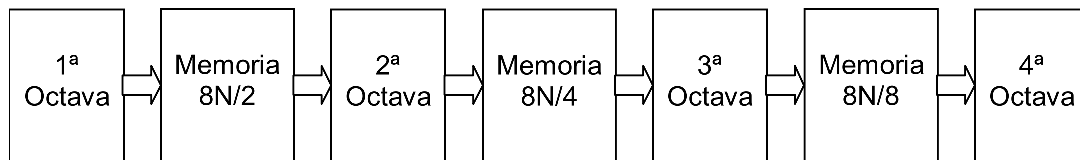


Figura 5.7. Diagrama de bloques de la realización de cuatro octavas.

Esta cantidad de memoria sí es posible tenerla dentro de una FPGA, ya que la familia Virtex de Xilinx dispone de pequeños bloques de memoria RAM (memoria embebida) que el diseñador puede utilizar. Utilizando la herramienta CoreGen de Xilinx, se puede generar una memoria de doble puerto en la que uno de los puertos se utiliza sólo de escritura y el otro únicamente de lectura.

Teniendo en cuenta estas consideraciones, se ha realizado un nuevo diseño consistente en la realización de una transformada wavelet de dos octavas utilizando la implementación Par-Impar y concatenando las dos octavas por medio de una memoria de doble puerto en la que se almacenan ocho filas de la primera octava.

La Figura 5.8 muestra el esquema de la simulación de la implementación Par-Impar con minimización de memoria. Tal como se ve, se ha utilizado la misma estructura que en los apartados anteriores, con la excepción del cambio de nombre del bloque principal, que ha pasado a denominarse Wavelet_G22. Este bloque realiza básicamente la misma función que el bloque denominado anteriormente Octava_G22, la única diferencia funcional es que Wavelet_G22 realiza la transformada wavelet de una imagen en dos niveles, mientras que Octava_G22 sólo hacía un nivel de la wavelet. La estructura interna de Wavelet_G22 se muestra en el diagrama de bloques de la Figura 5.9. La funcionalidad de cada uno de sus bloques se describe a continuación:

- **Generador_G41**, se encarga de generar las direcciones para leer la memoria que contiene la imagen, indica cuando se está direccionando el último pixel de la imagen necesario para obtener un dato de salida (Fin_data_A y Fin_data_B), indica cuando se está direccionando el último pixel de la fila (Fin_fila_A y Fin_fila_B) y cuándo termina de direccionarse toda la imagen completa (Fin_gene). Hay que tener presente que este bloque se encarga de tener en cuenta aspectos como, el diezmado y la ampliación de la imagen para eliminar el efecto de bordes.

- **Guardar_G21**, se encarga de generar las direcciones para guardar el resultado de la primera octava en la memoria de doble puerto de paso. Controla el multiplexor (bloque Mux2to1_G01) de salida a la memoria.
- **Dualmem_pas01**, memoria donde se almacenan las ocho filas necesarias entre la primera y segunda octava. Esta memoria es de doble puerto, siendo uno sólo de lectura mientras que el otro es únicamente de escritura. La memoria ha sido generada con la herramienta CoreGen de Xilinx, en la que se obtiene un bloque emplazable en las memorias embebidas de la FPGA de la familia Virtex.
- **Generador_G31**, se encarga de generar las direcciones para leer la memoria de doble puerto que contiene la imagen generada por la octava anterior. Básicamente funciona igual que el bloque Generador_G41, con la diferencia de que lee la memoria cíclicamente; es decir, cuando llega al final vuelve al principio, teniendo en cuenta que las dos primeras filas almacenadas deben conservarse hasta el final.
- **Control_General_G11**, este bloque a partir de la señal de "Inicio", pone en marcha el cálculo de cada una de las octavas en el momento adecuado, encargándose del sincronismo entre ambas para que la transformada wavelet se realice correctamente. También detiene el proceso cuando se han completado las operaciones. Este control se ha diseñado inicialmente para que el primer nivel de la transformada no se detenga nunca. El segundo nivel, será iniciado y parado en función de la disponibilidad de filas en la memoria entre niveles (Dualmem_pas01).
- **Guardar_G24**, se encarga de generar las direcciones para guardar el resultado final de la transformada wavelet en memoria. Controla el multiplexor (bloque Mux16to1_G01) de salida a la memoria.
- **Octava_G51**, este bloque se encarga del cálculo de una octava. Su estructura interna se puede ver en la Figura 5.10, basada en el bloque Octava_G22 de la implementación Par-Impar. Los elementos que lo constituyen son:
    - **Control_Path_G31**, este bloque a partir de la señal de “Inicio”, pone en marcha el generador de direcciones para leer la imagen (Generador_G41 o Generador_G31) y activa el bloque encargado de realizar las operaciones de filtrado (Data_Path_G01). También detiene el proceso cuando se le indica por medio de la señal "Stop". Las señales "Inicio" y "Stop" provienen del módulo de control general (Control_General_G11) de la jerarquía superior.
    - **Data_Path_G01**, es el encargado de realizar las operaciones matemáticas que realizan el filtrado. A partir de un pixel de entrada, proporciona en paralelo los cuatro pixeles de salida de los filtros. En la Figura 5.3 se muestra el diagrama de bloques funcional. Las salidas son registradas (bloque Registro_G01) en el momento adecuado para ser almacenadas en una memoria.
    - **Registrar_G01**, se encarga de generar las señales que permiten registrar el resultado del cálculo en el momento adecuado.

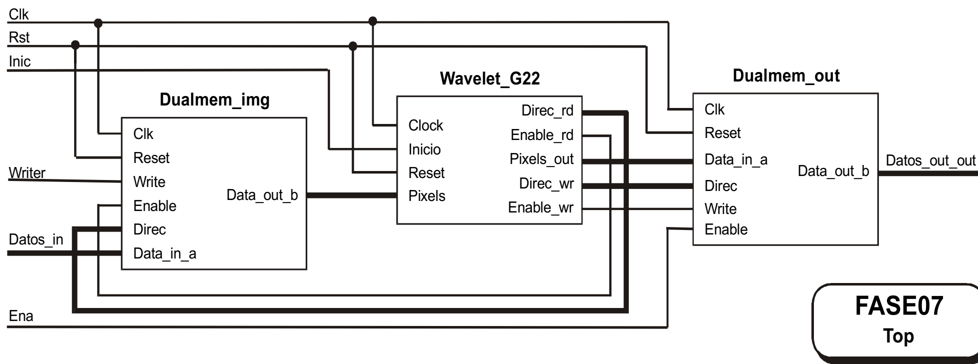


Figura 5.8. Diagrama de bloques del sistema utilizado en la Fase 07 para la simulación de la transformada wavelet.

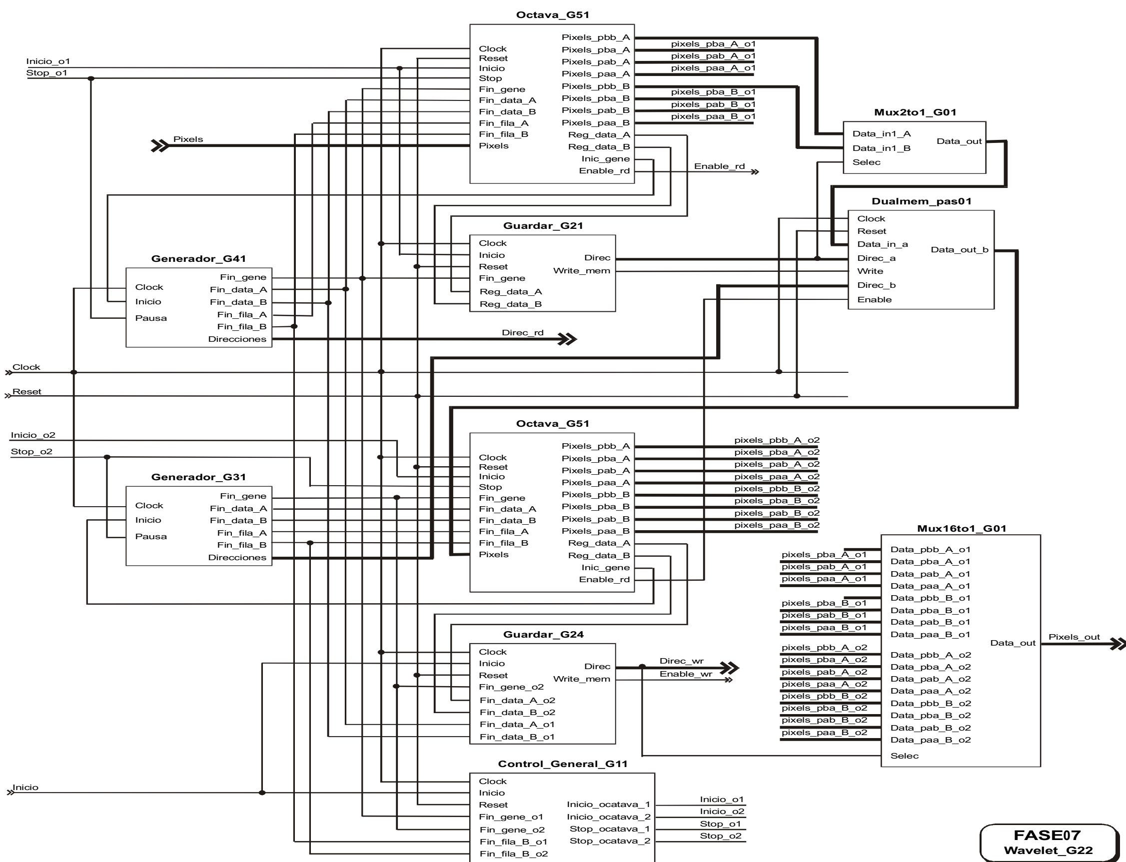


Figura 5.9. Diagrama de bloques de la implementación Wavelet_G22 de dos octavas de la transformada wavelet.

Tras comprobar su correcto funcionamiento, el bloque Wavelet_G22 se emplazó y rutó en la FPGA Virtex 200 de Xilinx, obteniendo los siguientes resultados:

- Número de ciclos de reloj: $2N^2+8N+24$, para una imagen de $N \times N$ pixeles.

- Ocupación en Slices: 1620 (68% de una Virtex 200).
- Ocupación de BlockRAMs: 7 (50% de una Virtex 200).
- Estimación de la frecuencia máxima de trabajo: 49 MHz.

Según estos resultados, el cálculo de dos octavas de la trasformada wavelet de una imagen de 512 x 512 pixeles con un reloj de 20 MHz, tardaría aproximadamente alrededor de 26,4 ms.

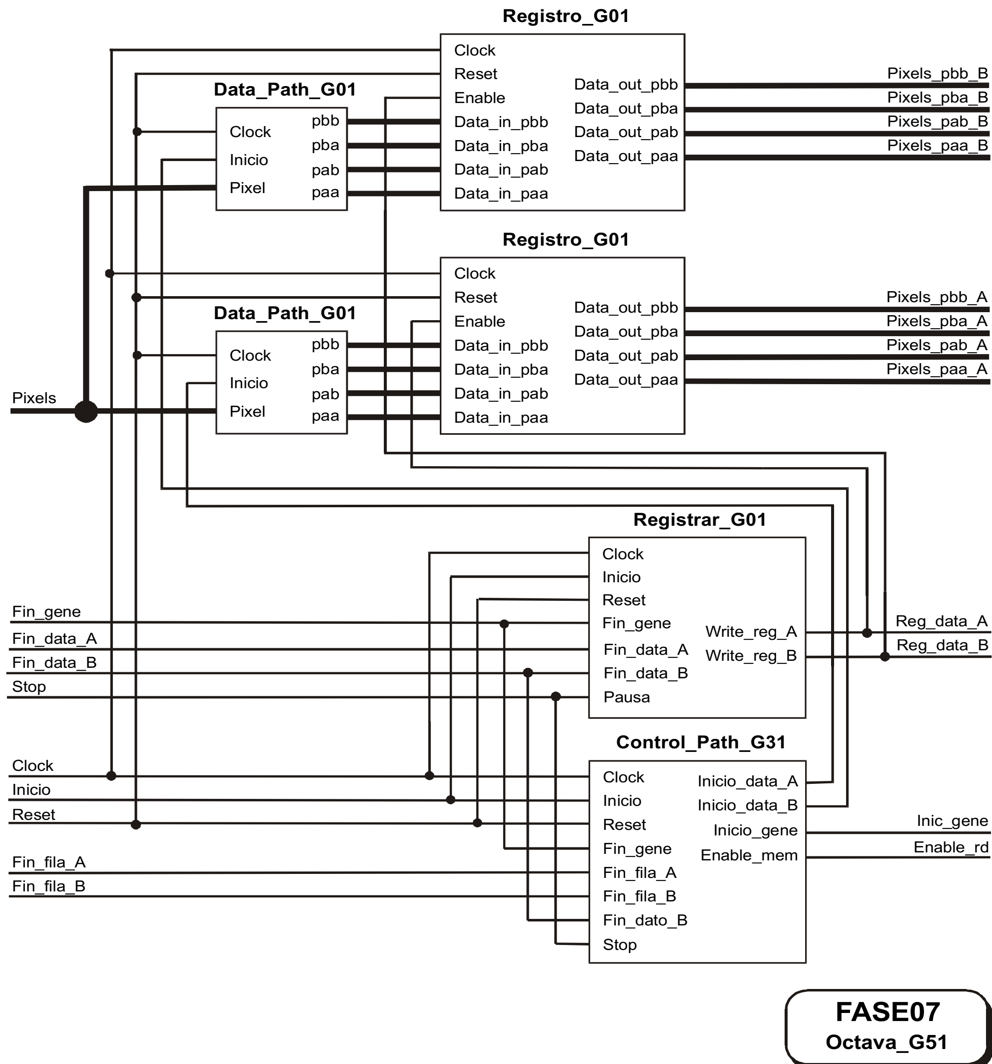


Figura 5.10. Diagrama de bloques de la implementación Octava_G51 del cálculo de una octava de la transformada wavelet.

A partir de esta implementación se pueden extraer las siguientes conclusiones:

- Es posible realizar una wavelet de cuatro octavas mediante una FPGA utilizando pocas células de almacenamiento internas. Por supuesto, se debería de emplear una FPGA superior a la utilizada como por ejemplo una Virtex 300 o 400.

- Del mismo modo que se utiliza una pequeña memoria para enlazar dos octavas, se puede utilizar una memoria de las mismas características para la imagen original. De este modo se puede procesar vídeo, ya que tal como llegan las filas se procesan, sin necesidad de tener toda la imagen completa en la memoria.
- Se puede utilizar en procesado de vídeo, ya que la transformada wavelet puede realizarse en tiempo real entre dos cuadros, incluso a 30 cuadros por segundo, y no es necesario disponer de cada cuadro completo para iniciar la transformada.

5.5 Implementación con Reducción de Ciclos

Con la implementación Par-Impar se obtiene un tiempo de procesado de la DWT de una imagen de $N \times N$ pixeles, del orden de $2N^2$ ciclos de reloj. Sin embargo, es deseable que el tiempo de procesado sea del orden de N^2 ciclos de reloj. Por tanto, el objetivo de la siguiente implementación, consiste en reducir a la mitad el número de ciclos de reloj necesarios para procesar la DWT de una imagen.

Ya que el número de ciclos se debe reducir a la mitad, una técnica sencilla sería leer dos pixeles de la memoria y procesarlos en paralelo. Para ello, se debe tener en cuenta el dispositivo sobre el que va a realizarse la implementación. En la implementación anterior se ha utilizado una memoria de doble puerto para enlazar un nivel de la transformada con el siguiente. Esta memoria es un bloque de memoria embebida que forma parte de los dispositivos de la familia Virtex de Xilinx. Utilizando la herramienta CoreGen de Xilinx se pueden definir o generar las memorias con las características que se deseen, siempre que no se superen las limitaciones del dispositivo. Analizando la herramienta CoreGen, se observa que pueden generarse memorias de doble puerto donde uno de los puertos es del doble de tamaño que el otro. Por tanto, si se define un puerto únicamente de escritura con un tamaño de 8 bits, se puede definir el otro puerto como sólo de lectura de 16 bits de tamaño, de este modo, al realizar una operación de lectura en la memoria se obtiene el contenido de dos células de memoria consecutivas. Así, se pueden leer en un sólo ciclo dos pixeles.

Teniendo en cuenta las consideraciones sobre las memorias entre octavas, se puede extender el concepto a la memoria de entrada que contiene la imagen. En este sentido se realizará un nuevo rediseño de la implementación anterior, en el que se leerán dos pixeles de la imagen a la vez. Los pixeles leídos corresponderán a los de la posición par e impar de la imagen, por tanto, habrá que diferenciarlos de los pixeles par e impar que se generan de salida.

En la Figura 5.11 puede verse el esquema de la simulación de esta nueva implementación. Tal como se ve se ha utilizado la misma estructura que en las implementaciones anteriores, pero con pequeñas diferencias adaptadas a las nuevas condiciones. La memoria que contiene la imagen (Dualmem_img_2b), ha sido modificada para que el puerto de lectura, permita obtener dos pixeles de la imagen a la vez. Los pixeles leídos cada vez, serán los de posiciones consecutivas. Esto significa que habrá un cambio en el bus de direcciones de lectura, ya que su tamaño será de una línea menos que el de escritura. El bloque principal (Wavelet_G31) realiza el cálculo de dos niveles de la transformada wavelet, leyendo dos pixeles en paralelo de

la imagen. La estructura interna de Wavelet_G31 se muestra en el diagrama de bloques de la Figura 5.12. La funcionalidad de cada uno de los bloques se describe a continuación:

- **Generador_G41**, se encarga de generar las direcciones para leer la memoria que contiene la imagen, indica cuándo se está direccionando el último pixel de la imagen necesario para obtener un dato de salida (Fin_data_A y Fin_data_B), cuándo se está direccionando el último pixel de la fila (Fin_fila_A y Fin_fila_B) y cuándo termina de direccionarse toda la imagen completa (Fin_gene). La generación de las direcciones ha sido modificada con respecto a la implementación anterior, ya que al leer dos pixeles a la vez se utiliza una línea menos de direccionamiento. Este bloque se encarga de tener en cuenta aspectos como el diezmado y la ampliación de la imagen para eliminar el efecto de bordes.

- **Guardar_G21**, se encarga de generar las direcciones para guardar el resultado de la primera octava en la memoria de doble puerto de paso. Controla el multiplexor (bloque Mux2to1_G01) de salida a la memoria.

- **Dualmem_pas21**, memoria donde se almacenan las ocho filas necesarias entre la primera y segunda octava. Esta memoria es de doble puerto, siendo uno sólo de lectura mientras que el otro es únicamente de escritura. El puerto de lectura es del doble de tamaño que el de escritura, lo cual permite leer dos pixeles a la vez. La memoria se ha generado con la herramienta CoreGen de Xilinx, en la que se obtiene un bloque emplazable en las memorias embebidas de la FPGA de la familia Virtex.

- **Generador_G31**, se encarga de generar las direcciones para leer la memoria de doble puerto que contiene la imagen generada por la octava anterior. Básicamente funciona igual que el bloque Generador_G41, con la diferencia de que lee la memoria cíclicamente; es decir, cuando llega al final vuelve al principio, teniendo en cuenta que las dos primeras filas almacenadas deben conservarse hasta el final. También ha sido convenientemente modificado para atender los nuevos requerimientos de la memoria Dualmem_pas21.

- **Control_General_G11**, este bloque a partir de la señal de "Inicio", pone en marcha el cálculo de cada una de las octavas en el momento adecuado, encargándose del sincronismo entre ambas para que la transformada wavelet se realice correctamente. También detiene el proceso cuando se han completado las operaciones. Este control se ha diseñado inicialmente, para que el primer nivel de la transformada no se detenga nunca. El segundo nivel, será iniciado y parado en función de la disponibilidad de filas en la memoria entre niveles (Dualmem_pas21).

- **Guardar_G24**, se encarga de generar las direcciones para guardar el resultado final de la transformada wavelet en memoria. Controla el multiplexor (bloque Mux16to1_G01) de salida a la memoria.

- **Octava_G51**, este bloque se encarga del cálculo de una octava. Su estructura interna puede verse en la Figura 5.13, basada en el mismo bloque de la implementación anterior ha sido modificada para poder procesar dos pixeles en paralelo. Los elementos que lo constituyen son:

- **Control_Path_G31**, este bloque a partir de la señal de "Inicio", pone en marcha el generador de direcciones para leer la imagen (Generador_G41 o Generador_G31) y activa el bloque encargado de realizar las operaciones de filtrado (Data_Path_G11). También detiene el proceso cuando se le indica por medio de la señal "Stop". Las señales "Inicio" y "Stop" provienen del módulo de control general (Control_General_G11) de la jerarquía superior.

- **Data_Path_G11**, es el encargado de realizar las operaciones matemáticas que realizan el filtrado. A partir de dos pixeles de entrada, proporciona en paralelo los cuatro pixeles de salida de los filtros. En la Figura 5.14 se muestra el diagrama de bloques funcional de la implementación realizada. Las salidas son registradas (bloque "Registro_G01") en el momento adecuado para ser almacenadas en una memoria. Tarda la mitad de ciclos en sacar un resultado, que el Data_Path_G01 anterior.

- **Registrar_G01**, se encarga de generar las señales que permiten registrar el resultado del cálculo en el momento adecuado.

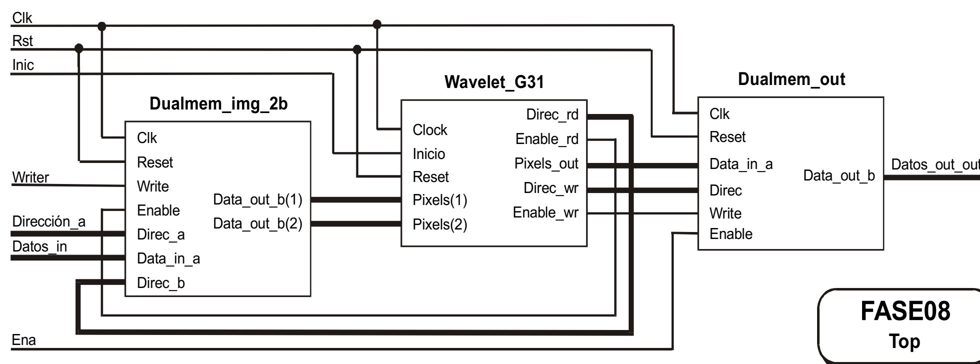


Figura 5.11. Diagrama de bloques del sistema utilizado en la Fase 08 para la simulación de la transformada wavelet.

Tras comprobar su correcto funcionamiento, el bloque Wavelet_G31 se emplazó y rutó en la FPGA Virtex 300 de Xilinx, obteniendo los siguientes resultados:

- Número de ciclos de reloj: $N^2+4N+12$, para una imagen de $N \times N$ pixeles.
- Ocupación en Slices: 2619 (85% de una Virtex 300).
- Ocupación de BlockRAMs: 7 (43% de una Virtex 300).
- Estimación de la frecuencia máxima de trabajo: 39 MHz.

Según estos resultados, el cálculo de dos octavas de la trasformada wavelet de una imagen de 512 x 512 pixeles con un reloj de 20 MHz, tardaría aproximadamente alrededor de 13,2 ms.

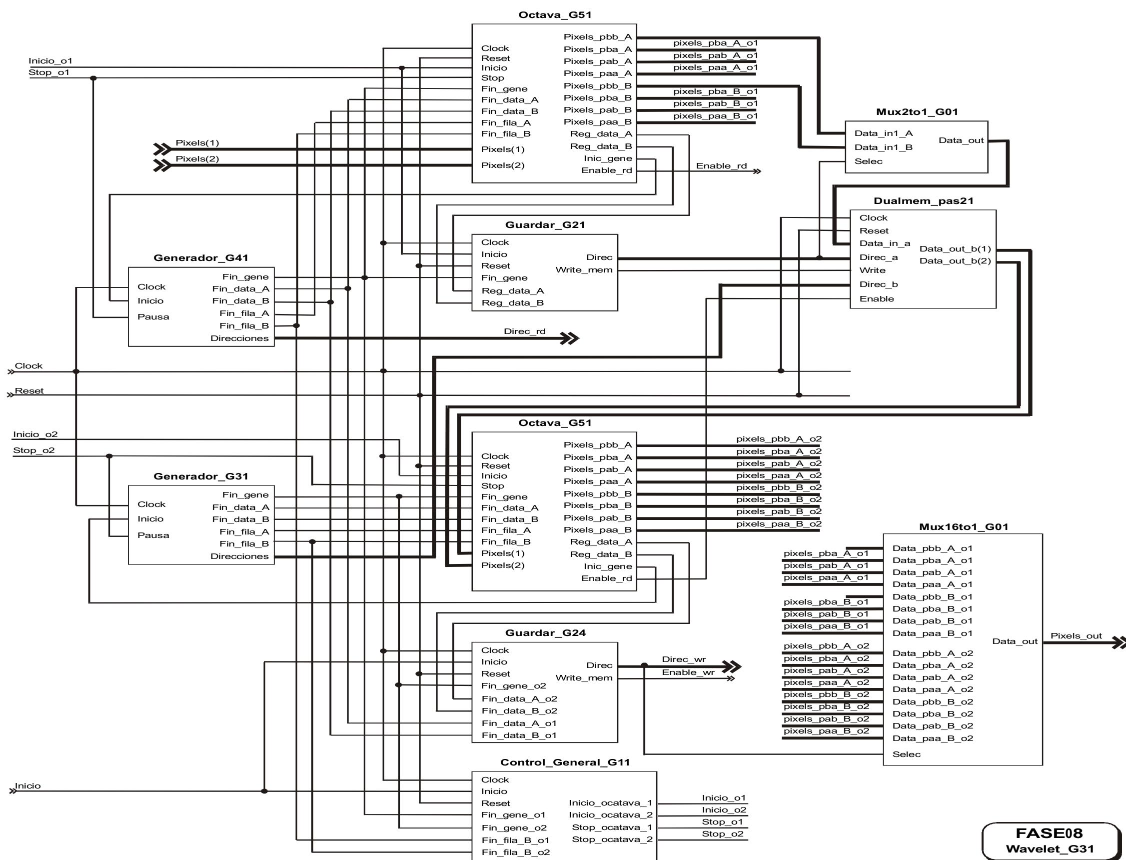


Figura 5.12. Diagrama de bloques de la implementación Wavelet_G31.

A partir de esta implementación pueden extraerse las siguientes conclusiones:

- Se ha conseguido reducir el número de ciclos a la mitad, con lo cual se mejoran las prestaciones del sistema. La consecuencia directa es que el tamaño del diseño se ha incrementado, por tanto, la utilización de una FPGA Virtex 300 no es suficiente.
- Al reducir el número ciclos que tarda en realizarse la transformada, también se reduce el tiempo entre dos muestras de salida consecutivas. En esta implementación, para filtros de tamaño $L \times L$, se obtiene una muestra de salida cada L ciclos.
- Otro aspecto negativo es que la complejidad del diseño ha reducido también la frecuencia máxima de trabajo, aunque los 39 MHz siguen siendo suficientes para procesar vídeo en tiempo real a 30 cuadros por segundo.

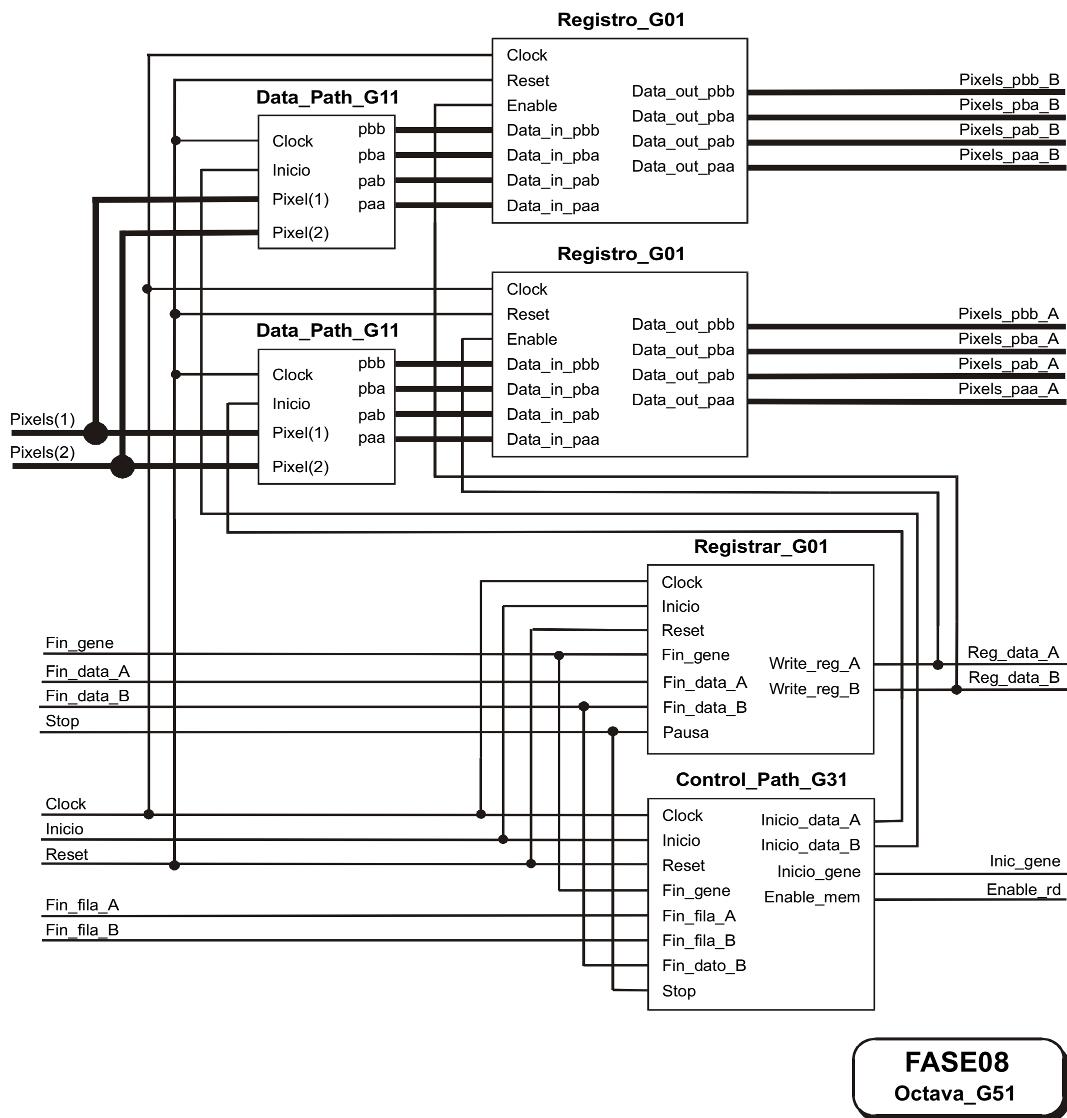


Figura 5.13. Diagrama de bloques de la implementación Octava_G51, en Fase 08.

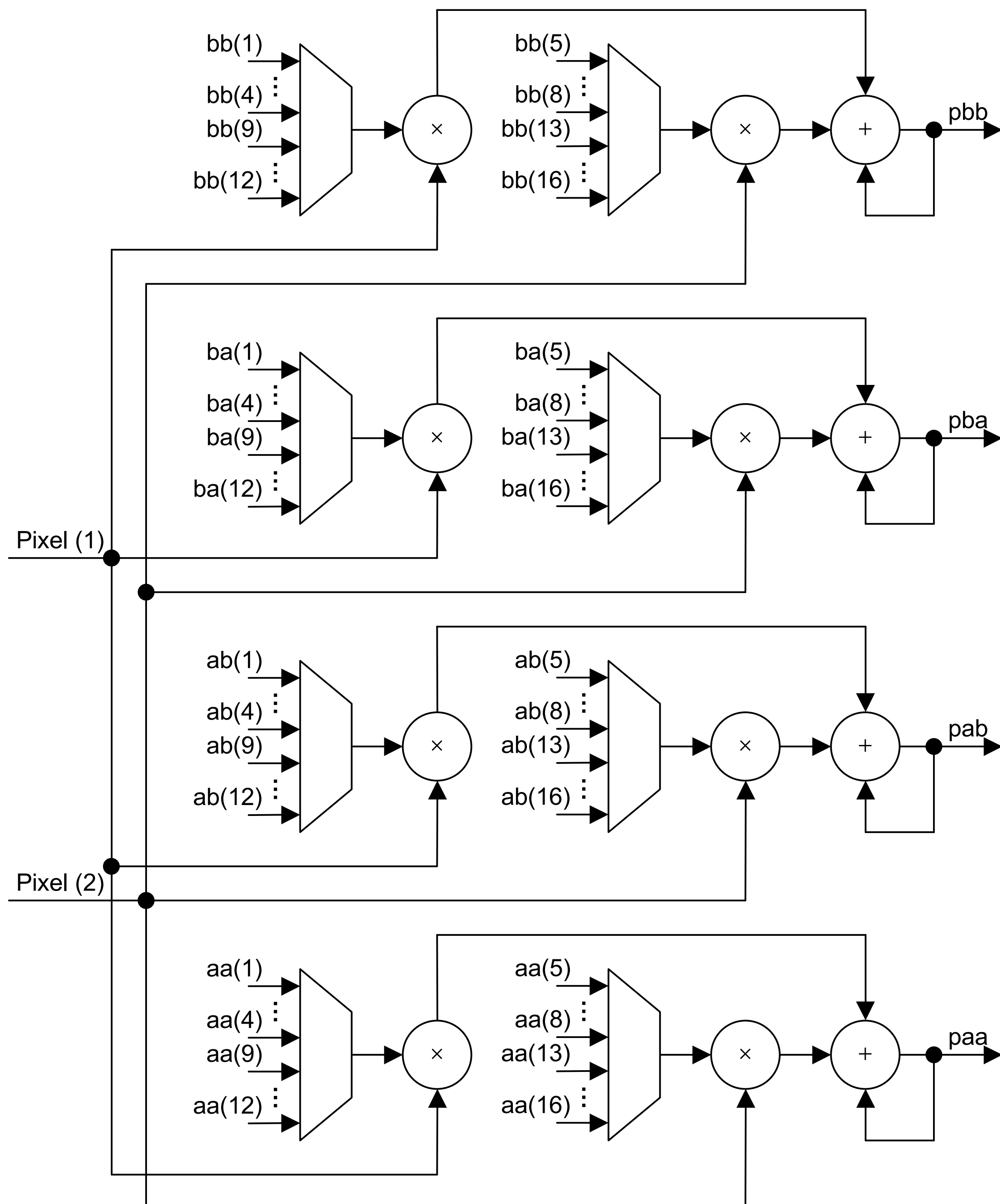


Figura 5.14. Diagrama de bloques funcional de Data_Path_G11.

5.6 Implementación de Tres Octavas

Entre las características que debe tener el diseño, se encuentran la modularidad y escalabilidad del mismo. En este sentido, el diseño debe poder ampliar con facilidad el número de octavas de la DWT. La implementación anterior únicamente realizaba la transformada wavelet de una imagen con dos octavas. Por tanto, en este apartado se

realiza una nueva implementación en la que el número de niveles u octavas de la DWT es de tres. La finalidad de esta implementación consistirá en analizar los aspectos de modularidad y escalabilidad del diseño, además de realizar una evaluación de la ocupación.

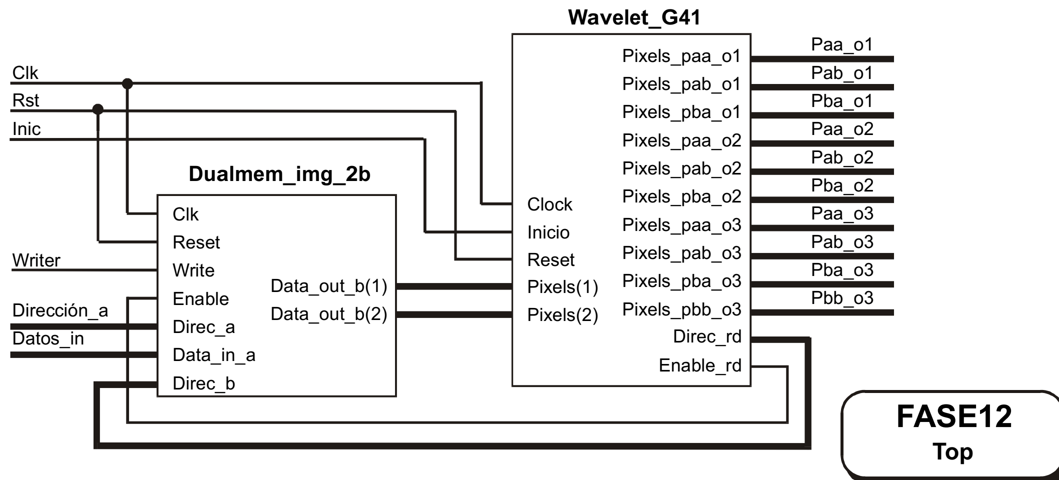


Figura 5.15. Diagrama de bloques utilizado para la simulación de la Fase12.

Para esta nueva implementación se parte de la implementación anterior, realizando las modificaciones oportunas, las cuales consisten en:

- El nivel más alto de la jerarquía del diseño de la DWT se denomina Wavelet_G41, el cual difiere ligeramente de los anteriores en dos aspectos (ver Figura 5.15). En primer lugar, por supuesto, realiza la DWT con tres niveles. El otro aspecto se observa en las señales de salida, en este caso no se han multiplexado para almacenar el resultado en una memoria. Esto se debe a que en realidad el diseño va a formar parte de un sistema de compresión de imágenes, por tanto, después de la DWT van las etapas de cuantificación y codificación, con lo cual no tiene sentido almacenar el resultado en una memoria.

- Con la finalidad de incrementar el número de octavas de dos a tres, algunos bloques se han replicado. En la Figura 5.16 se muestra el diagrama de bloques de la nueva implementación Wavelet_G41. En este caso, se ha añadido una nueva memoria de doble puerto Dualmem_pas21, entre la salida del segundo nivel y la entrada del tercer nivel. Asimismo, también se han replicado los bloques conocidos como Octava_G51, Guardar_G21 y Generador_G31, con respecto a la Figura 5.12, ya que son los necesarios para realizar una nueva octava.

- Los bloques conocidos como Mux16to1_G01 y Guardar_G24 de la Figura 5.12, se han eliminado, ya que las salidas de la DWT no se multiplexan en una memoria.

- El bloque denominado Mux2to1_G01 de la Figura 5.12 se ha eliminado, ya que las salidas del bloque Octava_G51 han sido modificadas. Ahora las salidas A y B provenientes de los dos Data_Path_G11, en primer lugar se multiplexan y

después se registran, al contrario que se hacia anteriormente. En la Figura 5.17 puede verse el nuevo diagrama de bloques de Octava_G51.

- El bloque denominado Control_General_G11, se encarga del control en el esquema de la Figura 5.16. Al incrementar el número de niveles de la DWT, lo normal sería modificar el diseño del bloque Control_General_G11 para poder gestionar tres niveles. Sin embargo, analizándolo con detalle se observa que en realidad se encarga de gestionar el flujo de datos entre un nivel y el siguiente. Por tanto, no hay que modificarlo sino simplemente añadir otro que se encargue de la gestión entre el segundo y el tercer nivel.

Tras comprobar su correcto funcionamiento, el bloque Wavelet_G41 se emplazó y rutó en la FPGA Virtex 600E de Xilinx. La familia Virtex-E, presenta las ventajas de que es más rápida y tiene más recursos que la familia Virtex. Los resultados obtenidos son:

- Número de ciclos de reloj del orden de N^2+N, para una imagen de $N \times N$ pixeles.
- Ocupación en Slices: 3580 (51% de una Virtex 600E).
- Ocupación de BlockRAMs: 14 (19% de una Virtex 600E).
- Estimación de la frecuencia máxima de trabajo: 48 MHz.

A partir de esta implementación se pueden extraer las siguientes conclusiones:

- En primer lugar se ha cambiado de familia de dispositivo, utilizando uno más rápido y de mayores recursos. Por tanto, el incremento de velocidad se ve justificado.
- Se puede incrementar el número de octavas con facilidad, no es necesario realizar un diseño nuevo, sino que simplemente hay que replicar bloques. Por tanto, el diseño es escalar y modular.
- La unidad de control del diseño se ha realizado de modo que no es necesario su rediseño, al incrementar el número de octavas, únicamente hay que replicarla. A esta propiedad se denominará control distribuido.
- La principal desventaja de esta implementación, está en el tiempo que el hardware realmente está haciendo operaciones. Del total de ciclos de reloj que se tarda en realizar el cálculo de la DWT de una imagen, la primera octava está realizando operaciones durante el 99,8 % del tiempo, la segunda octava únicamente realiza cálculos durante el 25 % del tiempo y la tercera octava sólo emplea un 6 % del tiempo. Por tanto, el hardware de la segunda y tercera octava está muy poco utilizado, la mayor parte del tiempo se encuentra detenido (en Stop).

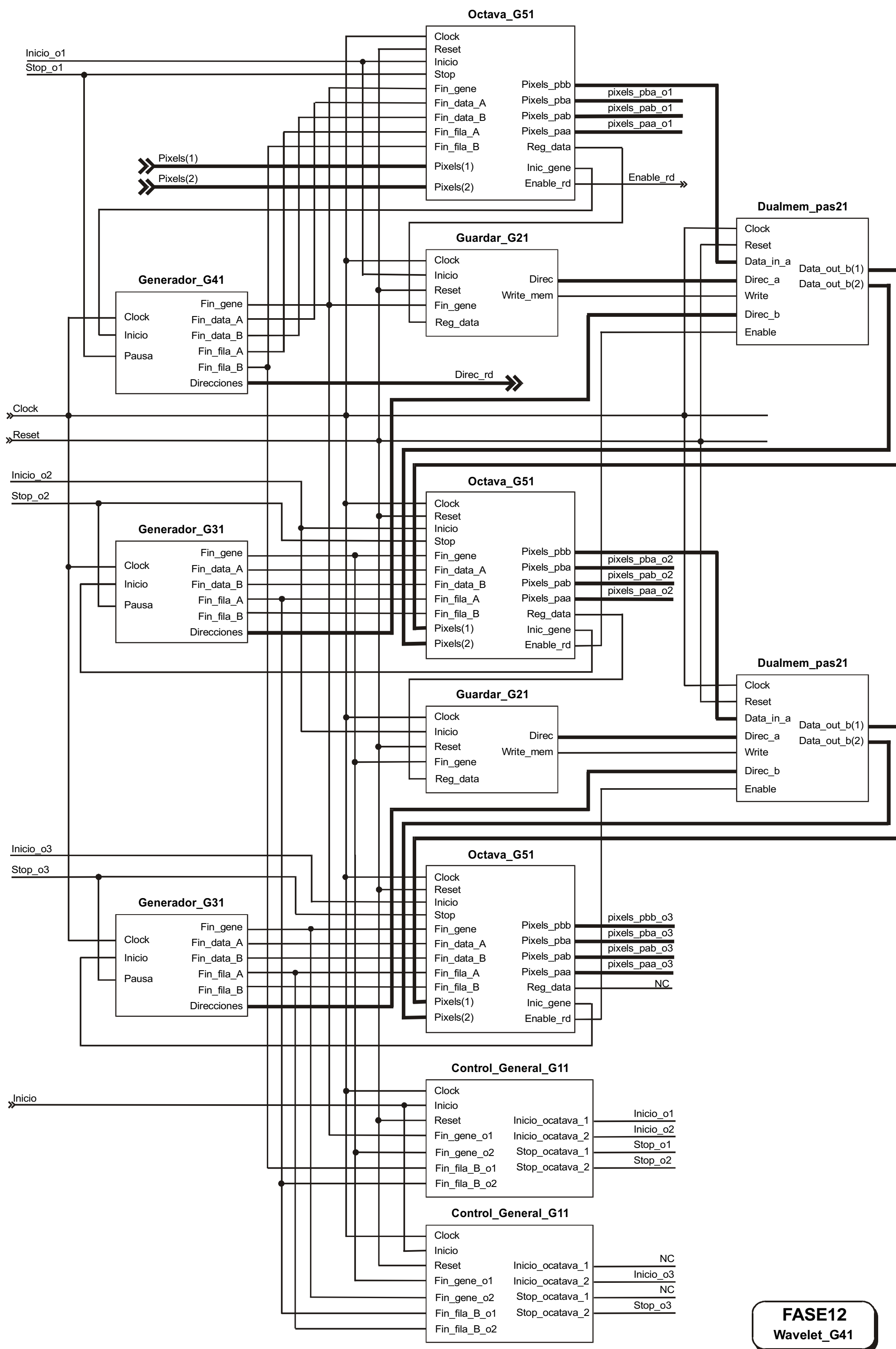


Figura 5.16. Diagrama de bloques de la implementación Wavelet_G41.

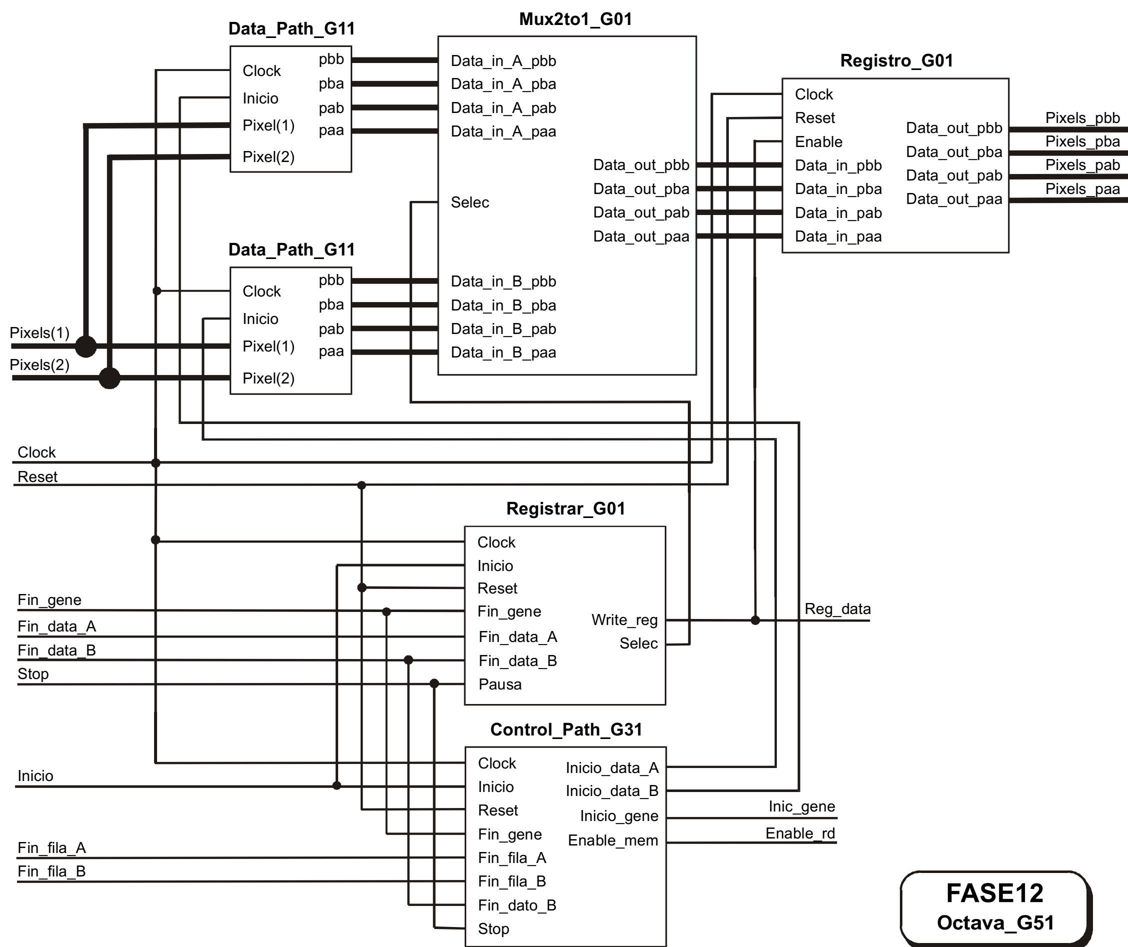


Figura 5.17. Diagrama de bloques de la implementación Octava_G51 de la Fase12.

5.7 Implementación Recurrente

El hardware, en las implementaciones anteriores, se encuentra muy poco utilizado y en global muy descompensado entre los porcentajes de utilización de cada uno de los bloques del diseño. Con la idea de mejorar el uso del hardware y a la vez reducir su tamaño se puede utilizar una arquitectura recurrente tal como se vió en el apartado 4.2.2.

El algoritmo piramidal recurrente (RPA) presenta la estructura del diagrama de bloques que se ve en la Figura 5.18. Tal como se aprecia, este algoritmo utiliza una unidad de cálculo (H(z) y G(z)) para realizar el cálculo de todos los niveles u octavas de la DWT. Sin embargo, se ha visto anteriormente que el primer nivel de la wavelet, emplea el 99,8 % del tiempo total de cálculo; es decir, prácticamente no se detiene nunca.

Si en las implementaciones anteriores se intentase realizar el cálculo de la DWT, utilizando un sólo bloque de filtrado, el hardware se reduciría y además tendría una utilización del 100 %, pero sin embargo, el tiempo total de cálculo de la wavelet se vería incrementado en un 31 %.

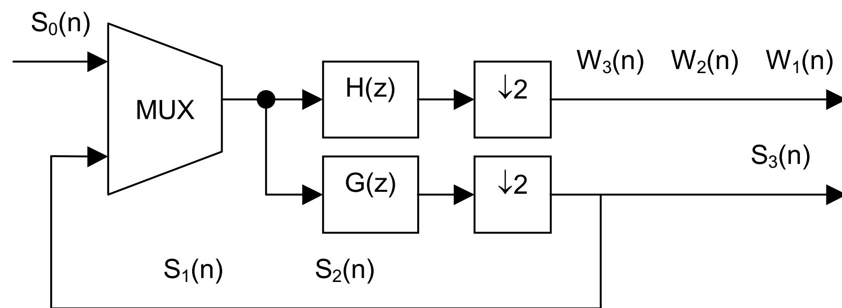


Figura 5.18. Diagrama de bloques del algoritmo piramidal recurrente.

En una de las arquitecturas presentadas en el capítulo 4, se planteaba un algoritmo recurrente en el que se utilizaba un bloque de filtrado para el primer nivel y otro bloque de filtrado para los demás niveles [40]. Esta opción es la que se ha elegido para la implementación del algoritmo recurrente. En la Figura 5.19 se muestra el diagrama de bloques de la estructura recurrente que se va a realizar en este apartado. Consiste en dos unidades de filtrado, la primera de ellas a partir de la imagen original ($S_0(n)$) obtiene en paralelo el resultado de la primera octava de la wavelet ($S_1(n)$ y $W_1(n)$). La segunda unidad de filtrado, tendrá características idénticas a la primera y proporcionará el resultado de las octavas segunda y tercera. De este modo, se consigue reducir el hardware, ya que se elimina una unidad de filtrado sin incrementar el tiempo de procesado global y mejorando el rendimiento del hardware. Con esta estructura, el hardware de la primera octava sigue utilizándose un 99,8 %, mientras que el hardware que realiza el cálculo del resto de las octavas se utiliza durante un 31% del tiempo total.

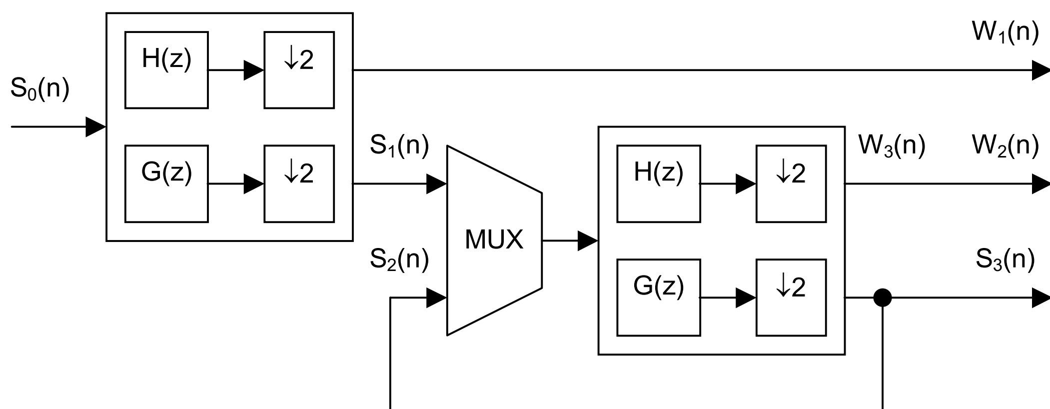


Figura 5.19. Diagrama de bloques del algoritmo piramidal recurrente que se desea implementar.

Para esta nueva implementación se parte de la implementación anterior, se ha intentado aprovechar al máximo los bloques ya realizados con el fin de afianzar las características de modularidad y escalabilidad evaluadas en el apartado anterior. Las modificaciones que presenta esta realización, con respecto a la implementación anterior, se comentan en los siguientes puntos:

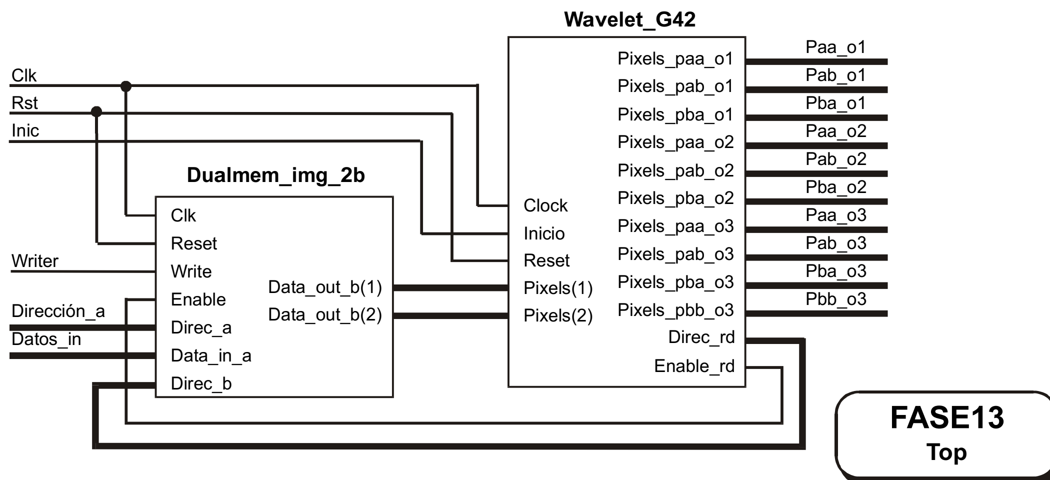


Figura 5.20. Diagrama de bloques del sistema utilizado en la Fase 13 para la simulación.

- El nivel más alto de la jerarquía del diseño de la implementación recurrente se denomina ahora Wavelet_G42 (ver Figura 5.20). Este nivel de diseño no se diferencia en nada de la implementación anterior.

- Con la finalidad de realizar una estructura recurrente como la planteada en la Figura 5.19, se ha eliminado alguno de los bloques. En la Figura 5.21 se muestra el diagrama de bloques de la nueva implementación Wavelet_G42. Las diferencias básicas consisten en que los dos bloques de cálculo Octava_G51, que realizaban la segunda y tercera octava, son fusionados en un único bloque denominada Octava_G61. Otro aspecto es que las líneas de lectura de datos de las memorias de enlace entre la primera y la segunda octava y entre la segunda y tercera octava, han sido multiplexadas mediante el bloque Mux2to1_G21. El resto de elementos permanecen igual que en la implementación anterior.

- El bloque Octava_G51 tiene la misma estructura que se utilizaba en la implementación anterior. Su diagrama de bloques puede verse en la Figura 5.22. Este bloque únicamente se utiliza para realizar el cálculo de la primera octava de la transformada.

- El bloque Octava_G61 no es un nuevo diseño, sino que consiste en una ampliación del bloque Octava_G51. En la Figura 5.23 se puede ver la estructura interna del bloque Octava_G61. Este bloque realiza el cálculo de dos octavas de modo secuencial o en serie. Su función es la equivalente a dos bloques Octava_G51, pero utilizando menos hardware. Al bloque Octava_G51 se le han replicado los bloques Mux2to1_G01, Registro_G01, Registrar_G01 y Control_Path_G31, obteniendo así la estructura representada en la Figura 5.23. Los bloques Mux2to1_G01, Registro_G01 y Registrar_G01 se han replicado con la finalidad de obtener en paralelo las señales del cálculo de la segunda y tercera octava.

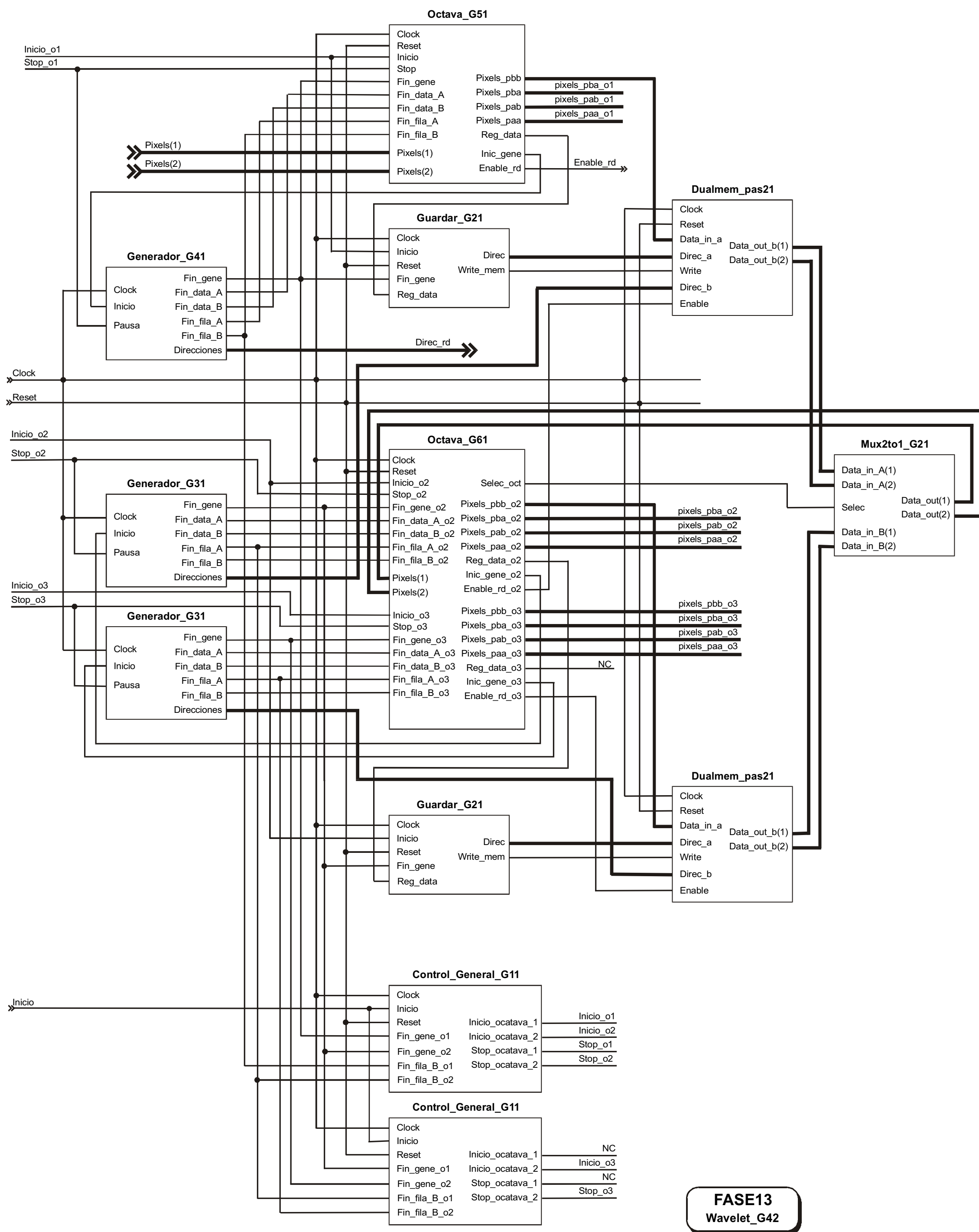


Figura 5.21. Diagrama de bloques de la implementación recurrente realizada.

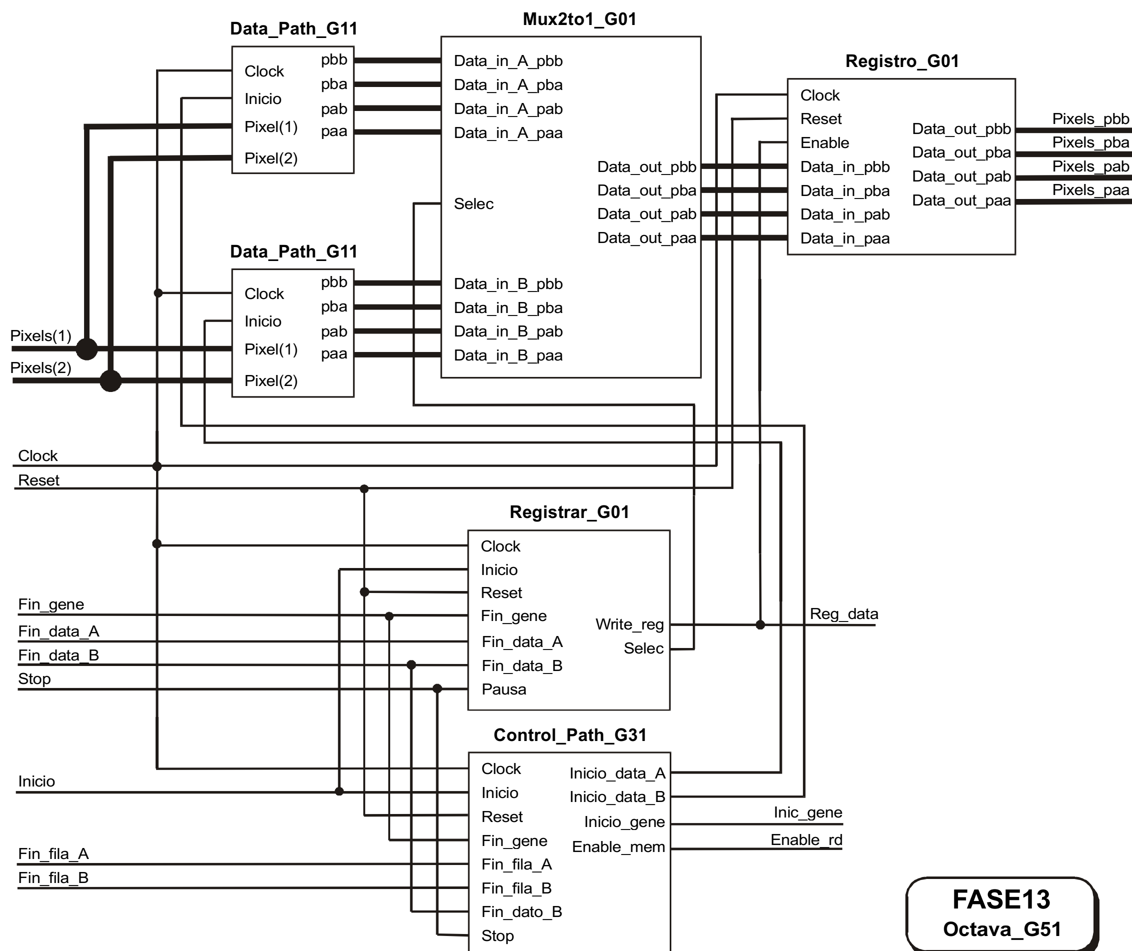


Figura 5.22. Diagrama de bloques de la implementación Octava_G51 de la Fase 13.

- En el bloque Octava_G61 se han utilizado dos unidades de control (Control_Path_G31), para gestionar el cálculo de la segunda y tercera octava por medio de un sólo bloque de operaciones (dos Data_Path_G11). Un sincronismo adecuado hace posible que no se produzcan colisiones en el cálculo. El sincronismo viene proporcionado por las unidades Control_General_G11 del nivel jerárquico superior. Cada una de estas unidades controla una octava y la siguiente. Así la segunda octava, en régimen permanente, no inicia las operaciones de una fila hasta que la primera octava no haya generado dos nuevas filas de salida. Por tanto, cuando la segunda octava haya generado dos nuevas filas, la tercera octava iniciará el cálculo de una nueva fila de salida. Según esto, en el peor de los casos, el bloque Octava_G61 debe concatenar el cálculo de una fila de la segunda octava y una fila de la tercera octava, el tiempo total de cálculo de ambas filas es 3/4 partes del tiempo de cálculo de una fila de la primera octava. Además, como se necesitan dos filas de la primera octava para iniciar la segunda, puede considerarse que el tiempo empleado es 3/8 partes del tiempo de dos filas de la primera octava. Por tanto, el cálculo de una fila de la segunda octava no puede ser nunca activado cuando se está realizando el cálculo de una fila de la tercera. En la Figura 5.24 puede verse un diagrama temporal, en el que se

reflejan cómo se realizan las operaciones de cálculo de las filas de cada una de las octavas.

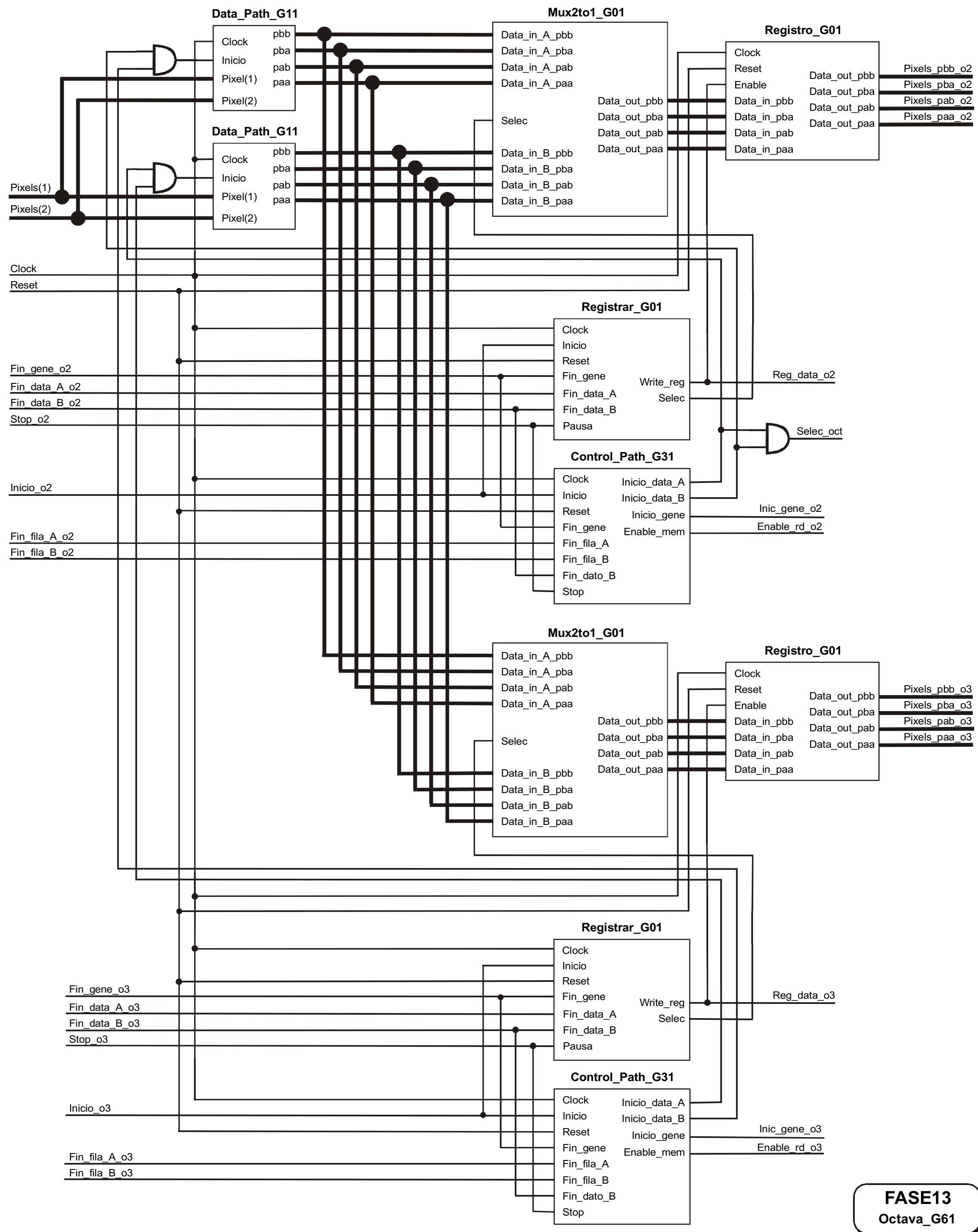
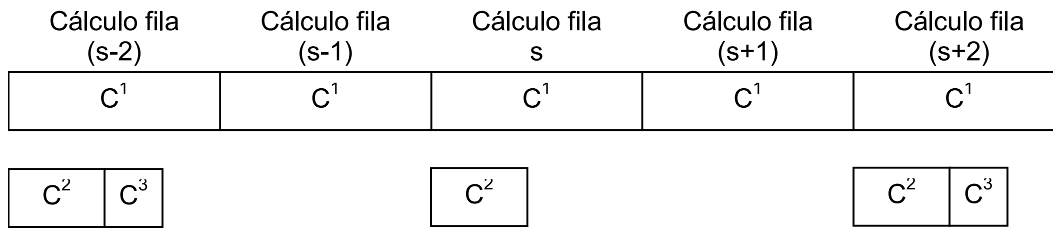


Figura 5.23. Diagrama de bloques de la implementación Octava_G61 de la Fase 13.



C^m : Cálculo de la octava m.

s : Fila de salida. Donde $1 \leq s \leq N/2$ siendo N el número de filas de la imagen de entrada.

Figura 5.24. Diagrama temporal de realización de cada una de las octavas.

Tras comprobar su correcto funcionamiento, el bloque Wavelet_G42 se emplazó y rutó en la FPGA Virtex 600E de Xilinx. Los resultados obtenidos son:

- Número de ciclos de reloj del orden de N^2+N, para una imagen de $N \times N$ pixeles.
- Ocupación en Slices: 2554 (36% de una Virtex 600E).
- Ocupación de BlockRAMs: 14 (19% de una Virtex 600E).
- Estimación de la frecuencia máxima de trabajo: 47 MHz.

A partir de esta implementación pueden extraerse las siguientes conclusiones:

- Con respecto a la estructura desplegada de la implementación anterior se produce una reducción de un 30% en la ocupación del dispositivo.
- Esta implementación utiliza 4 elementos Data_Path_G11 en lugar de 6 que se utilizaban anteriormente. Por tanto, cada uno de los Data_Path_G11 ocupa alrededor de unos 500 slices (7% de una Virtex 600E), lo cual significa que los Data_Path_G11 representan el 78% del diseño en ocupación, mientras que el resto de tareas de control únicamente utilizan un 22% del diseño.
- La utilización del hardware es más completa, las unidades de cálculo de la primera octava son utilizadas prácticamente el 100% del tiempo, mientras que las unidades de cálculo del resto de octavas tienen una utilización del 31% del tiempo, porcentaje que se incrementa al aumentar el número de octavas.
- Aunque la utilización del hardware no es completa se mantiene el tiempo de cálculo de la wavelet.
- El diseño mantiene la escalabilidad y la modularidad, características que ya fueron demostradas en el apartado anterior y que se reafirman con mayor ímpetu en esta implementación, ya que no ha sido necesario diseñar ningún elemento nuevo.
- Un aspecto importante a destacar es el control distribuido. Realizado mediante dos bloques (Control_Path_G31 y Control_General_G11) consistentes en dos

máquinas de estados que se utilizan en diferentes niveles jerárquicos y que se comunican entre sí.

- La frecuencia máxima de trabajo prácticamente se mantiene con respecto al diseño anterior.

5.8 Implementación Particularizada

Los diseños anteriores son generalistas en cuanto a los coeficientes de la transformada wavelet. El código VHDL del bloque Data_Path_G11, está realizado de tal modo que los coeficientes de los filtros son tratados como constantes. Así, un filtro de tamaño $L \times L$ está compuesto por L^2 valores o constantes.

En la Figura 5.25 se muestran dos porciones del código VHDL del bloque Data_Path_G11. En el recuadro superior se muestra la declaración de las constantes de los coeficientes. Estas constantes han sido definidas en un "package" adjunto como un vector de L^2 valores. La constante "bb" representa los coeficientes del filtro GG de la Figura 4.15, la constante "ba" tiene los coeficientes del filtro GH, la constante "ab" son los coeficientes del filtro HG y por último la constante "aa" contiene los coeficientes del filtro HH.

```
constant bb : MATRIX_BIORTO3:= MATRIZ_BB_BI3;
constant ba : MATRIX_BIORTO3:= MATRIZ_BA_BI3;
constant ab : MATRIX_BIORTO3:= MATRIZ_AB_BI3;
constant aa : MATRIX_BIORTO3:= MATRIZ_AA_BI3;
```

```
pbb2 := pixel(13 downto 0)*bb(i);
pbb3 := pixel(27 downto 14)*bb(i+4);
pbb1 := pbb3+pbb2+pbb1;
pba2 := pixel(13 downto 0)*ba(i);
pba3 := pixel(27 downto 14)*ba(i+4);
pba1 := pba3+pba2+pba1;
pab2 := pixel(13 downto 0)*ab(i);
pab3 := pixel(27 downto 14)*ab(i+4);
pab1 := pab3+pab2+pab1;
paa2 := pixel(13 downto 0)*aa(i);
paa3 := pixel(27 downto 14)*aa(i+4);
paa1 := paa3+paa2+paa1;
```

Figura 5.25. Porción del código VHDL de Data_Path_G11.

En el recuadro inferior de la Figura 5.25 se muestra la porción de código VHDL que realiza las operaciones matemáticas de filtrado. Tal como se aprecia, en cada ciclo de reloj, por cada filtro se realizan dos operaciones de multiplicación y dos sumas. Hay que recordar que se trabaja con dos pixeles en paralelo, con lo cual "pixel(13 downto 0)" corresponde con el denominado Pixel(1) en los diagramas de bloque y "pixel(27

downto 14)" es el Pixel(2). Según esto, el cálculo de las cuatro salidas en paralelo requiere realizar 8 multiplicaciones y 8 sumas por cada uno de los Data_Path_G11.

```
temp_0A(13 downto 0) :=pixel(13 downto 0);
temp_0A(21 downto 14) := (others => pixel(13));
temp_1A:=pixel(13 downto 0)*"00000011";
temp_2A:=pixel(13 downto 0)*"00001001";
Mtemp_0A := -temp_0A;
Mtemp_1A := -temp_1A;
Mtemp_2A := -temp_2A;
temp_0B(13 downto 0) :=pixel(27 downto 14);
temp_0B(21 downto 14) := (others => pixel(27));
temp_1B:=pixel(27 downto 14)*"00000011";
temp_2B:=pixel(27 downto 14)*"00001001";
Mtemp_0B := -temp_0B;
Mtemp_1B := -temp_1B;
Mtemp_2B := -temp_2B;

case i is
      when 1 =>
            pbb1 := (2=>'1', others => '0');
            pba1 := (3=>'1', others => '0');
            pab1 := (3=>'1', others => '0');
            paa1 := (4=>'1', others => '0');

            pbb2 := temp_0A;
            pbb3 := Mtemp_1B;
            pba2 := temp_1B;
            pba3 := Mtemp_0A;
            pab2 := temp_1B;
            pab3 := Mtemp_0A;
            paa2 := temp_0A;
            paa3 := Mtemp_1B;

      when 2 =>
            pbb2 := temp_2B;
            pbb3 := Mtemp_1A;
            pba2 := temp_1A;
            pba3 := Mtemp_2B;
            pab2 := temp_1A;
            pab3 := Mtemp_2B;
            paa2 := temp_2B;
            paa3 := Mtemp_1A;
            .
            .
      end case;

      pbb1 := pbb2+pbb3+pbb1;
      pba1 := pba2+pba3+pba1;
      pab1 := pab2+pab3+pab1;
      paa1 := paa2+paa3+paa1;

      if i = 8 then
            i:=1;
      else
            i:=i+1;
      end if;
```

Figura 5.26. Porción del código VHDL de Data_Path_G32.

Tomando unos coeficientes determinados, por ejemplo los denominados CDF 3/1 que fueron analizados en el apartado 3.6.3 (página 45), se puede observar que las cuatro matrices resultantes tienen muchos elementos en común. Por tanto, las operaciones de multiplicación anteriores pueden reducirse únicamente a dos.

Con este fin se ha diseñado un nuevo bloque de cálculo denominado Data_Path_G32. En la Figura 5.26 se muestra una parte del código del bloque de cálculo Data_Path_G32. En esta nueva implementación únicamente se realizan cuatro operaciones de multiplicación, consistentes en multiplicar los dos pixeles de entrada por dos constantes. El número de sumas se sigue manteniendo en ocho, pero además se añaden restas (por ejemplo: Mtemp_0A := –temp_0A;) y una sentencia "case".

Utilizando el bloque Data_Path_G32, se ha vuelto a realizar la síntesis, emplazamiento y rutado de Wavelet_G42 y se ha obtenido que la ocupación del diseño es de 2152 slices (31% de una Virtex 600E) y su frecuencia máxima de trabajo es de 35 MHz.

A la vista de los resultados el código VHDL del Data_Path_G32 ha sido modificado para eliminar completamente los multiplicadores, por medio del uso de sumas y desplazamientos. El nuevo diseño se ha denominado Data_Path_G42, lo cual ha permitido realizar una nueva síntesis, emplazado y rutado de Wavelet_G42. En este caso se ha obtenido una ocupación de 2131 slices (30 % de una Virtex 600E) y una frecuencia máxima de trabajo de 37 MHz.

Con esta particularización del diseño se consigue una reducción de un 16 % en la ocupación del mismo, aunque la cantidad de multiplicaciones se haya reducido a la mitad o se hayan sustituido por sumas y desplazamientos. La opción de sustituir las multiplicaciones por sumas y desplazamientos apenas supone un ahorro de área. Destacar que en estas particularizaciones se produce una reducción bastante importante de la frecuencia máxima de trabajo (alrededor de 10 MHz), con respecto a la implementación generalista.

5.9 Resumen de Resultados

En este apartado se presenta mediante tablas el resumen de las principales características y los resultados obtenidos en las implementaciones realizadas en el presente capítulo. Estos mismos resultados han sido presentados en [117].

En la Tabla 5.1 se muestran los siguientes resultados por columnas:

1. Denominación de la implementación realizada.
2. Número de Octavas que realiza el diseño implementado en FPGA.
3. Ciclos de Reloj. Se refiere al número aproximado de ciclos de reloj que necesita la implementación realizada para calcular la transformada wavelet de una imagen de $N \times N$ pixeles. Este parámetro depende directamente del número de octavas que se hayan implementado.
4. Estimación de Células de Memoria. Esta columna refleja la cantidad de células de memoria que hacen falta para realizar la transformada wavelet en cuatro octavas de una imagen de $N \times N$ pixeles.

5. Tipo de Memoria. Según sea el tamaño de la memoria que se necesite, ésta puede emplazarse en el interior de la FPGA o deberá de utilizarse memoria externa al dispositivo.

6. Estimación del Tiempo de Cálculo. Esta columna refleja el tiempo que necesita la implementación para realizar el cálculo de la transformada wavelet, considerando una imagen de 512 $\times$ 512 pixeles, con filtros de 4 $\times$ 4 coeficientes, cuatro octavas y utilizando un reloj de 20MHz.

Tabla 5.1. Resumen de las implementaciones realizadas I.

Implementación	Número de Octavas Implementadas	Ciclos de Reloj	Estimación de Células de Memoria	Tipo de Memoria	Estimación de Tiempo de Cálculo
Directa	1	$4N^2$	$21N^2/64$	Externa	52,6 ms
Par-Impar	1	$2N^2+2N$	$21N^2/64$	Externa	26,6 ms
Par-Impar Minimización Memoria	2	$2N^2+2N$	7N	Interna	26,6 ms
Reducción Ciclos	2	N^2+N	7N	Interna	13,3 ms
Tres Octavas	3	N^2+N	7N	Interna	13,3 ms
Recurrente	3	N^2+N	7N	Interna	13,3 ms
Particularizada con Reducción Multiplicaciones	3	N^2+N	7N	Interna	13,3 ms
Particularizada sin Multiplicaciones	3	N^2+N	7N	Interna	13,3 ms

En la Tabla 5.2 se muestran los siguientes resultados por columnas:

1. Denominación de la implementación realizada.

2. Algoritmo Recurrente. No todas las implementaciones realizadas han sido pensadas para utilizar un algoritmo recurrente. Por supuesto, las implementaciones recurrentes presentan una mejor utilización del hardware.

3. Lectura Imagen. En esta columna se pretende reflejar el modo en el que las imágenes son leídas de las memorias, tanto la imagen inicial como los enlaces entre octavas. Básicamente se han utilizado dos técnicas, la lectura pixel a pixel de modo secuencial y la lectura en paralelo de dos pixeles a la vez.

4. Filtrado. El filtrado se refiere al modo en el que han sido implementadas la unidades de cálculo que realizan el filtrado de la imagen. Básicamente se han utilizado tres técnicas, relacionadas directamente con el modo de lectura de la

imagen. La primera (serie) consiste en realizar el cálculo en serie tal como llegan los pixeles de la imagen. La segunda técnica se ha denominado Serie-Paralelo Desfasado, la cual consiste en calcular en paralelo dos pixeles de salida, en cada uno de estos pixeles las operaciones se realizan en serie y la salida de los resultados está desfasada 8 ciclos. La última técnica utilizada (Paralelo-Paralelo Desfasado), realiza el filtrado de dos pixeles de entrada en paralelo obteniendo el resultado de dos pixeles de salida desfasados 4 ciclos.

5. Salida Primer Dato. Refleja el número de ciclos necesarios para obtener el primer dato de salida.

6. Cadencia Salida Datos. En esta columna se muestran el número de ciclos que transcurren entre la salida de un dato y el siguiente. Suponiendo un cálculo continuo.

Tabla 5.2. Resumen de las implementaciones realizadas II.

Implementación	Algoritmo Recurrente	Lectura Imagen	Filtrado	Salida Primer Dato	Cadencia Salida Datos
Directa	No	Serie Pixel a Pixel	Serie	16	16
Par-Impar	No	Serie Pixel a Pixel	Serie-Paralelo Desfasado	16	8
Par-Impar Minimización Memoria	No	Serie Pixel a Pixel	Serie-Paralelo Desfasado	16	8
Reducción Ciclos	No	Serie-Paralelo Dos pixeles cada vez	Paralelo-Paralelo Desfasado	8	4
Tres Octavas	No	Serie-Paralelo Dos pixeles cada vez	Paralelo-Paralelo Desfasado	8	4
Recurrente	Si	Serie-Paralelo Dos pixeles cada vez	Paralelo-Paralelo Desfasado	8	4
Particularizada con Reducción Multiplicaciones	Si	Serie-Paralelo Dos pixeles cada vez	Paralelo-Paralelo Desfasado	8	4
Particularizada sin Multiplicaciones	Si	Serie-Paralelo Dos pixeles cada vez	Paralelo-Paralelo Desfasado	8	4

En la Tabla 5.3 se muestran los siguientes resultados por columnas:

1. Denominación de la implementación realizada.

2. Número de Octavas que realiza el diseño implementado en FPGA.
3. Dispositivo. FPGA sobre la que se ha realizado la implementación.
4. Ocupación de la FPGA en Slices. También se refleja el porcentaje del dispositivo utilizado.
5. Ocupación Interna de Memoria. La mayoría de las implementaciones utilizan memoria interna a la FPGA. Esta memoria ha sido realizada utilizando los bloques de memoria embebida que dispone el dispositivo. Por tanto, en esta columna se refleja la cantidad de bloques de memoria interna utilizada o BlockRAMs.
6. Frecuencia Máxima de Trabajo.

Tabla 5.3. Resumen de las implementaciones realizadas III.

Implementación	Número de Octavas Implementadas	Dispositivo	Ocupación de la FPGA en Slices	Ocupación Interna Memoria	Frecuencia Máxima (MHz)
Directa	1	Virtex 200	431 (18%)		50
Par-Impar	1	Virtex 200	751 (32%)		51
Par-Impar Minimización Memoria	2	Virtex 200	1620 (68%)	7 (50%)	49
Reducción Ciclos	2	Virtex 300	2619 (85%)	7 (43%)	39
Tres Octavas	3	Virtex 600E	3580 (51%)	14 (19%)	48
Recurrente	3	Virtex 600E	2554 (36%)	14 (19%)	47
Particularizada con Reducción Multiplicaciones	3	Virtex 600E	2152 (31%)	14 (19%)	35
Particularizada sin Multiplicaciones	3	Virtex 600E	2131 (30%)	14 (19%)	37

5.10 Conclusiones

En este capítulo, se ha realizado el diseño e implementación de varias estructuras para el cálculo de la transformada wavelet con el fin de ser utilizados en la compresión de imágenes y vídeo. De todas las estructuras presentadas, la que se ha denominado implementación recurrente (apartado 5.7), es la que presenta unas mejores características cumpliendo los objetivos planteados al principio de la Tesis.

La estructura presentada, ha sido diseñada íntegramente en lenguaje VHDL, ha sido sintetizada, emplazada y rutada en una FPGA de la familia Virtex de Xilinx. El dispositivo utilizado tiene una capacidad media dentro de la familia Virtex y solamente se ha empleado un 36% del mismo. Por tanto, el diseño es válido y utilizable.

La implementación diseñada utiliza el algoritmo recurrente en la variante en que emplea un elemento de filtrado para la primera octava y otro elemento de filtrado para el resto de las octavas. Las unidades de filtrado se han desarrollado para utilizar filtros no separables, lo cual las dota de más versatilidad. Su estructura está realizada de tal modo que procesa en paralelo dos datos de entrada, realizando internamente un filtrado en serie. Dos unidades de filtrado se combinan adecuadamente para obtener cada unos de los elementos de filtrado. Estas dos unidades de filtrado comparten la entrada de datos, aunque generan salidas en paralelo. El funcionamiento de las dos unidades se encuentra desfasado cuatro ciclos de reloj. De este modo, los pixeles de las imágenes únicamente se leen una vez.

Entre las principales virtudes del diseño se encuentran su modularidad y escalabilidad. Tal como se ha visto a lo largo del capítulo, a partir de unos bloques ya diseñados ha sido posible pasar de una estructura no recurrente de dos octavas a una estructura de tres octavas y posteriormente a una estructura recurrente. Por tanto, combinando adecuadamente los distintos bloques diseñados se puede adaptar el diseño al tamaño de la imagen, al número de octavas y al número de coeficientes. Uno de los éxitos de la flexibilidad del diseño se debe al sistema de control utilizado. Empleando un control distribuido en dos bloques, ha sido posible escalar el diseño sin necesidad de cambiar la unidad de control, tan sólo hay que replicar los bloques de control.

Otra peculiaridad del diseño realizado es el tratamiento de los bordes. Uno de los objetivos planteados era tratar adecuadamente los bordes de la imagen. La imagen no puede sufrir alteraciones, lo cual obliga a que las dos primeras columnas y filas de la imagen deban ser conservadas hasta el final del procesado, ésto fuerza al uso de unidades de almacenamiento y condiciona la arquitectura.

El diseño ha sido particularizado para un tipo de coeficientes, mediante la sustitución de las operaciones de multiplicación por sumas y desplazamientos, el resultado ha sido una reducción del área ocupada del 16% y una penalización en la frecuencia máxima de trabajo al disminuir en 10 MHz.

En la Tabla 5.4 se muestra la comparativa entre la arquitectura diseñada [117] y las presentadas en el capítulo 4. En esta tabla se ha considerado una imagen de 512 $\times$ 512 pixeles, con filtros de longitud 4 (4 $\times$ 4, para no separables) y 3 niveles de la wavelet.

Tabla 5.4. Resumen comparativo entre las arquitecturas.

Implementación	[117]	[31]	[27]	[40]	[96]	[23]	[80]
Arquitectura Filtros	"systolic-parallel" "bit-parallel"	"systolic-parallel" "bit-parallel"	"bit-parallel" "digit-serial"	"bit-parallel"	"bit-parallel"	"bit-parallel"	"bit-serial"
Tipo Filtros	No Separable	Separable	Separable	Separable	Separable	No Separable	No Separable
Cadencia	4	1	1	1	1	1	
MAC		8			24	8	
Multiplicadores	32	16	8	10			
Sumadores	32	16	8	10		3	2048 de un sólo bit
Células Almacenamiento	3072	6144		896	3968	3584	7680
Frecuencia (MHz)	47			25		55	50
Tiempo de Cálculo (ciclos)	N^2+N	N^2+N	N^2+N	N^2+N	N^2+N	N^2+N	N^2+N

Intentando realizar una comparación entre el diseño aquí realizado [117] y las otras arquitecturas destacan los siguientes aspectos:

- Todos los diseños utilizan una arquitectura recurrente.
- La mayoría se basan en arquitecturas 1-D y por ello utilizan filtros separables. El diseño aquí realizado utiliza filtros no separables o de dos dimensiones.
- La cadencia de salida de los datos de [117] es superior, ya que los demás consiguen una cadencia unitaria.
- El diseño [117] es el único realizado sobre una FPGA. Entre los otros diseños únicamente [23] y [80] realizan una implementación sobre ASIC, los demás plantean arquitecturas teóricas.
- En un diseño realizado sobre FPGA es difícil determinar cuántos multiplicadores y sumadores se utilizan, ya que las herramientas de síntesis realizan una compilación del código VHDL en la que se pierde en control sobre los elementos finales. Sin embargo, analizando el código VHDL realizado se

puede decir que en [117] se han utilizado 32 operaciones de multiplicación y suma. Comparando con los otros diseños se observa que utilizan muchos menos multiplicadores. En [23] se utilizan 8 MAC, sin embargo, emplea internamente un reloj 4x para reducir el hardware, en caso contrario emplearía 32 MAC. Una implementación de [117] particularizada, se ha visto que utiliza 16 operaciones de multiplicación al igual que ocurre en [31], no obstante, ésto sólo conseguía reducir el hardware un 16% en lugar de a la mitad que sería lo lógico.

- En cuanto a los elementos de almacenamiento, únicamente [40] emplea menos que [117], sin embargo, no puede realizar procesado de vídeo en tiempo real.
- El procesado de vídeo en tiempo real únicamente puede ser realizado por [23], [80] y [117], aunque [23] necesite más elementos de almacenamiento.
- Todos los diseños emplean el mismo tiempo en realizar el procesado.
- Únicamente [117] realiza tratamiento de los bordes, los demás no hacen referencia a este tema excepto [96] que utiliza un rellenado con ceros.

Capítulo 6

Conclusiones y Futuras Líneas

6.1 Principales Aportaciones

En la presente Tesis Doctoral se realiza un diseño hardware de la transformada wavelet, con el fin de ser utilizado en un sistema de compresión de imágenes y vídeo digital. El diseño pretende ser abierto, modular y escalable, de modo que los parámetros definitivos de la transformada wavelet puedan ser elegidos en función del tipo de imágenes y de los índices de compresión. Así, una vez fijados los parámetros el diseño puede ser rápidamente sintetizado e implementado en una FPGA. Las características de reconfigurabilidad de las FPGA permiten la modificación del funcionamiento de la transformada.

Previamente a la realización del diseño ha sido necesario realizar un estudio de la transformada wavelet de dos dimensiones, analizando los tipos de filtros que se pueden utilizar, así como cuáles son los que proporcionan una mejor relación señal-ruido. Este estudio ha permitido realizar un análisis del número de bits que hacen falta en cada etapa de la transformada wavelet, para realizar el cálculo utilizando datos del tipo entero.

Las principales aportaciones realizadas por el presente trabajo se detallan en los siguientes puntos:

1. Se ha establecido la estructura general de un sistema de compresión de imágenes y vídeo digital. Se han analizado las principales características de los estándares de compresión JPEG y MPEG, viendo la problemática que presentan en el caso de imágenes que son muy sensibles a pequeñas alteraciones de su contenido, como ocurre con las imágenes médicas. Estos sistemas se fundamentan en la transformada discreta del coseno (DCT), de la cual derivan sus inconvenientes.
2. Se ha hecho un estudio de los tipos de filtros que se pueden utilizar para la realización de la transformada wavelet de dos dimensiones. Dicho estudio se

ha realizado utilizando operadores enteros, obteniéndose las siguientes características:

- El estudio se ha realizado mediante la programación de los algoritmos de la transformada wavelet en lenguaje VHDL.
- Se han realizado elementos de enlace entre una herramienta de simulación VHDL como ModelSim y el programa MATLAB, con el objetivo de visualizar los resultados de la simulación VHDL como imágenes.
- Los resultados después de las simulaciones ha permitido conocer que filtros proporcionan una mejor calidad de imagen. Considerando que se está trabajando con números enteros y que no hay pérdidas que no provengan de posibles redondeos o truncamientos.
- El estudio ha permitido conocer cuántos bits son necesarios para la realización de la transformada wavelet en función de los filtros y el número de octavas.
- También se ha obtenido una estimación del número de operaciones matemáticas necesarias para el cálculo de la transformada wavelet, dependiendo del número de coeficientes de los filtros.
- En el análisis, los bordes de la imagen han sido tratados de modo que la imagen recuperada después de la transformada, sea aproximadamente igual a la imagen original. Para ello, es necesario ampliar la imagen en la operación de cálculo, replicando las primeras filas y columnas al final de las filas y columnas respectivamente.

3. Se han analizado las arquitecturas básicas para la implementación de la transformada wavelet, así como algunas de las implementaciones, lo cual ha permitido realizar un análisis comparativo.
4. Se ha realizado un diseño que implementa la transformada wavelet de dos dimensiones en una FPGA. Dicho diseño presenta sobre el resto de arquitecturas las siguientes características:

- Utiliza filtros no separables. Filtros de dos dimensiones.
- La implementación se ha realizado sobre un dispositivo programable como es la FPGA.
- El diseño realizado utiliza una variante del algoritmo recurrente (RPA).
- Minimiza el uso interno de células de almacenamiento.
- El diseño es modular y escalable en su totalidad. Está formado por una serie de bloques básicos que combinados adecuadamente permite incrementar el número de octavas o variar las características de los filtros.
- El diseño utiliza un sistema de control distribuido que incrementa la capacidad de modularidad y escalabilidad. Sólo hay que replicar las unidades para incrementar el número de octavas.

- El diseño realizado trata los bordes de la imagen mediante el uso de las primeras filas y columnas para el procesado del final de la fila y la columna respectivamente.
- La implementación recurrente mostrada en el apartado 5.7, puede realizar el procesado de vídeo en tiempo real. Por ejemplo, el codificador de vídeo SAA7113 de Philips, trabaja con una señal de reloj de 27 MHz, lo cual proporciona un cuadro de luminancia de 512×512 pixeles cada 19 ms. La implementación recurrente empleada tarda alrededor de 9,7 ms en procesar una imagen de 512×512 pixeles con un reloj de 27 MHz.
- El diseño ha sido íntegramente realizado en lenguaje VHDL.
- El diseño está preparado para enlazar con las etapas de cuantificación y codificación, que junto a la transformación constituyen un sistema de compresión. El enlace no necesita de elementos intermedios de almacenamiento.
- El diseño permite realizar, en el caso del vídeo digital, una transformada wavelet en tres dimensiones, añadiendo antes del diseño una etapa que se encargaría de realizar la transformada wavelet en el tiempo.

6.2 Líneas de Investigación Futuras

La presente Tesis Doctoral se enmarca dentro de un proyecto CICYT cuyo objetivo es realizar un sistema de compresión hardware de vídeo en tiempo real. En este trabajo únicamente se ha presentado la etapa de transformación del sistema de compresión. A partir de este momento la investigación debería centrarse en completar el proyecto, realizando las etapas de cuantificación y de codificación y una etapa previa, al diseño presentado, que realizaría la transformada wavelet en el tiempo.

Por tanto, las futuras líneas de investigación se centrarán en el estudio de métodos de cuantificación y codificación, así como su implementación hardware mediante FPGA. Así, se podría completar el sistema de compresión de imágenes y vídeo digital.

Tras la realización de la compresión, hay que hacer un sistema hardware de descompresión, con la finalidad de poder establecer transmisiones del vídeo entre un sistema y el otro. La implementación de la descompresión depende directamente de la compresión.

Las características de reconfigurabilidad dinámica de las FPGA de la familia Virtex de Xilinx, permiten ampliar las posibilidades de investigación hacia la realización de un sistema que pueda hacer una evaluación previa del tipo de imágenes a comprimir para adaptar el diseño de la compresión de vídeo.

Otro aspecto interesante en las futuras líneas de investigación sería abordar el diseño de un circuito integrado de aplicación especifica. Las arquitecturas e implementaciones para FPGA deberían ser adaptadas a la filosofía de diseño VLSI para ASIC. El objetivo es concluir con la fabricación del circuito integrado, utilizando como diseño el sistema de compresión de vídeo planteado anteriormente.

La investigación del procesado de imágenes y vídeo digital ofrece un abanico muy amplio de posibilidades, sobre todo, dentro del campo de la implementación hardware de algoritmos. Por tanto, esta línea de investigación puede continuar con la implementación de técnicas de reconocimiento de patrones, detección y estimación de movimientos, etc.

Capítulo 7

Referencias

[1] JOHN G. PROAKIS Y DIMITRIS G. MANOLAKIS. ***"Tratamiento Digital de Señales. Principios, algoritmos y aplicaciones"*** Prentice-Hall. 1988.

[2] KESHAB K. PARHI. ***"VLSI Digital Signal Processing Systems"*** John Wiley & Sons, Inc. 1999.

[3] P. P. VAIDYANATHAN. ***"Multirate Systems and filter banks"*** Prentice-Hall. 1993.

[4] DAN E. DUDGEON Y RUSSELL M. MERSEREAU. ***"Multidimensional Digital Signal Processing"*** Prentice-Hall. 1984.

[5] WILLIAM K. PRATT. ***"Digital Image Processing"*** John Wiley & Sons, Inc. 1991.

[6] RAFAEL C. GONZALEZ Y RICHARD E. WOODS. ***"Digital Image Processing"*** Addison Wesley. 1993.

[7] KESHAB K. PARHI Y TAKAO NISHITANI. ***"Digital Signal Processing for Multimedia Systems"*** Marcel Dekker, Inc. 1999.

[8] STEFAN SJOHOLM Y LENNART LINDH. ***"VHDL for Designers"*** Prentice-Hall. 1997.

[9] DAVID W. KNAPP. ***"Behavioral Synthesis"*** Prentice-Hall. 1996.

[10] ROLAND AIRIAU, JEAN-MICHEL BERGÉ Y VINCENT OLIVE. ***"Circuit Synthesis with VHDL"*** Kluwer Academic Publishers. 1994.

[11] PRAN KURUP Y TAHER ABBASI. ***"Logic Synthesis Using Synopsys"*** Kluwer Academic Publishers. 1997.

[12] AL BOVIK. ***"Handbook of Image and Video Processing"*** Academic Press. 2000.

[13] K. R. RAO Y P. C. YIP. ***"The Transform and Data Compression Handbook"*** CRC Press. 2001.

[14] INGRID DAUBECHIES. ***"Ten Lectures on Wavelets"*** CBMS-NSF Regional conference Series In Applied Mathematics. 1992.

[15] STÉPHANE MALLAT. ***"A Wavelet Tour of Signal Processing"*** Academic Press. 1999.

[16] C. SIDNEY BURRUS, RAMESH A. GOPINATH Y HAITAO GUO. ***"Introduction to Wavelets and Wavelet Transforms"*** Prentice Hall. 1998.

[17] RICHARD I. HARTLEY Y KESHAB K. PARHI. ***"Digit-Serial Computation"*** Kluwer Academic Publishers. 1995.

[18] S. J. CHANG, M. H. LEE, Y J. Y. PARK. ***"A High Speed VLSI Architecture of Discrete Wavelet Transform for MPEG-4"*** IEEE transactions on consumer electronics, Vol. 43, Nº. 3, pp. 623-627, 1997.

[19] J. LIN, W. H. KI, T. EDWARDS Y S. SHAMMA. ***"Analog VLSI Implementations of Auditory Wavelet Transforms Using Switched-Capacitor Circuits"*** IEEE transactions on circuits and systems I: Fundamental theory and applications, Vol. 41, Nº. 9, pp. 572-583, 1994.

[20] V. PIURI, E. E. SWARTZLANDER JR, Y F. MARINO. ***"A Parallel Implementation of the 2-D Discrete Wavelet Transform without Interprocessor Communications."*** IEEE transactions on signal processing, Vol. 47, Nº. 11, pp. 3179-3184, 1999.

[21] Y. H. LEE, H. KUNIEDA, T. ISSHIKI Y J. T. KIM. ***"Scalable VLSI Architectures for Lattice Structure-Based Discrete Wavelet Transform."*** IEEE transactions on circuits and systems II:Analog and digital signal processing, Vol. 45, Nº. 8, pp. 1031-1043, 1998.

[22] R. M. OWENS, M. J. IRWIN Y M. VISHWANATH, M. ***"VLSI Architectures for the Discrete Wavelet Transform."*** IEEE transactions on circuits and systems II:Analog and digital signal processing, Vol. 42, Nº. 5, pp. 305-316, 1995.

[23] S. J. CHEN Y C. YU. ***"VLSI Implementation of 2-D Discrete Wavelet Transform for Real-Time Video Signal Processing."*** IEEE transactions on consumer electronics, Vol. 43, Nº. 4, pp. 1270-1279, 1997.

[24] J. FRIDMAN Y E. S. MANOLAKOS. ***"On the synthesis of regular VLSI architectures for the 1-D discrete wavelet transform"*** Proc. of SPIE Conf. On Mathematical Imaging: Wavelet Applications in Signal and Image Processing II, San Diego CA, Julio 1994.

[25] J. FRIDMAN Y E. S. MANOLAKOS. ***"Distributed Memory and Control VLSI Architectures for the 1-D Discrete Wavelet Transform"*** IEEE VLSI Signal Processing, VII, pp. 388-397, 1994.

[26] O. M. NIELSEN Y M. HEGLAND. ***"A Scalable Parallel 2D Wavelet Transform Algorithm"*** Joint Computer Science Technical Report Series, 1997, The Australian National University.

[27] ALEKSANDER GRZESZCZAK, MRINAL K. MANDAL Y SETHURAMAN PANCHANATHAN. ***"VLSI Implementation of Discrete Wavelet Transform"*** IEEE Transactions on VLSI Systems, Vol. 4, Nº 4, pp. 421-433, 1996.

[28] ROBERT D. TURNEY, CHRIS DICK Y ALI M. REZA. ***"Multirate Filters and Wavelets: From Theory to Implementation"*** Xilinx Inc.

[29] ALI M. REZA. ***"Wavelet Characteristics. What Wavelet Should I use?"*** White Paper, 1999.

[30] XUYUN CHEN, TING ZHOU, QIANLING ZHANG, WEI LI Y HAO MIN. ***"A VLSI Architecture For Discrete Wavelet Transform"*** IEEE International Conference on Image Processing, Vol. 2, pp. 1003-1006, 1996.

[31] MOHAN VISHWANATH, ROBERT MICHAEL OWENS Y MARY JANE IRWIN. ***"VLSI Architectures for the Discrete Wavelet Transform"*** IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 42, Nº. 5, pp. 305-316, 1995.

[32] JAVIER VEGA-PINEDA, SERGIO D. CABRERA Y YI-CHIEH CHANG. ***"VLSI Implementation of a Wavelet Image Compression Technique Using Replicated Coding/Decoding Cells"*** IEEE International Symposium on Circuits and Systems, ISCAS'95, Vol. 2, pp. 1173-1176, 1995.

[33] ISIDRO URRIZA, JOSÉ I. ARTIGAS, JOSÉ I. GARCIA, LUIS A. BARRAGÁN Y DENIS NAVARRO. ***"VLSI Architecture for Lossless Compression of Medical Images Using the Discrete Wavelet Transform"*** IEEE Proceeding Design, Automation and Test in Europe, pp. 196-201, 1998.

[34] SEUNG-KWON PAEK Y LEE-SUP KIM. ***"2D DWT VLSI architecture for wavelet image processing"*** Electronics Letters, Vol. 34, Nº. 6, pp. 537-538, 1998.

[35] MATTHIAS SAUER Y JÜRGEN GÖTZE. ***"A VLSI Architecture for Fast Wavelet Computations"*** Proceedings of the IEEE-SP International Symposium, Time-Frequency and Time-Scale Analysis, pp. 407-410, 1992.

[36] A.M. RASSAU, SOO-IK CHAE Y K. ESHRAGHIAN. ***"Simple Binary Wavelet Filters for Efficient VLSI Implementation"*** Electronics Letters, Vol. 35, Nº. 7, pp. 555-557, 1999.

[37] A.B. PREMKUMAR Y A.S. MADHUKUMAR. ***"An Efficient VLSI Architecture for Computation of 1-D Discrete Wavelet Transform"*** IEEE Proceedings of

International Conference on Information, Communications and Signal Processing, ICICS'97, Vol. 2, pp. 1180-1184, 1997.

[38] KESHB K. PARHI Y TAKAO NISHITANI. ***"VLSI Architectures for Discrete Wavelet Transforms"*** IEEE Transactions on VLSI Systems, Vol. 1, Nº. 2, pp. 191-202, 1993.

[39] CHETANA NAGENDRA, MARY JANE IRWIN Y ROBERT M. OWENS. ***"Digit Pipelined Discrete Wavelet Transform"*** IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, pp. II/405-II/408, 1994.

[40] MING-HWA SHEU, MING-DER SHIEH Y SHENG-WEL LIU. ***"A VLSI Architecture Design With Lower Hardware Cost and Less Memory for seaprable 2-D Discrete Wavelet Transform"*** Proceedings of the 1998 IEEE International Symposium on Circuits and Systems, Vol. 5, pp. 457-460, 1998.

[41] LI-MINN ANG, HON NIN CHEUNG Y KAMRAN ESHRAGHIAN. ***"VLSI Architecture for Significance Map Coding of Embedded Zerotree Wavelet Coefficients"*** IEEE Asia-Pacific Conference on Circuits and Systems, pp. 627-630, 1998.

[42] A.S. LEWIS Y G. KNOWLES. ***"VLSI Architecture for 2-D Daubechies Wavelet Transform Without Multipliers"*** Electronics Letters, Vol. 27, Nº. 2, 1991.

[43] GAUTHIER LAFRUIT, FRANCKY CATTHOOR, JAN P. H. CORNELIS Y HUGO J. DE MAN. ***"An Efficient VLSI Architecture for 2-D Wavelet Image Coding with Novel Image Scan"*** IEEE Transactions on VLSI Systems, Vol. 7, Nº. 1, 1999.

[44] G. KNOWLES. ***"VLSI Architecture for the Discrete Wavelet Transform"*** Electronics Letters, Vol. 26, Nº. 15, 1990.

[45] JONGWOO BAE Y VIKTOR PRASANNA. ***"Synthesis of VLSI Architectures for Two-Dimensional Discrete Wavelet Transforms"*** IEEE Proceedings. International Conference on Application Specific Array Processors, pp. 174-181, 1995.

[46] JOHN D. HOYT Y HARRY WECHSLER. ***"The Wavelet Transform - A CMOS VLSI ASIC Implementation"*** IEEE Proceedings. International Conference on Pattern Recognition, Vol. 4, pp. 19-22, 1992.

[47] A. GRZESZCZAK, T.H. YEAP Y S. PANCHANATHAN. ***"VLSI Implementation of Discrete Wavelet Transform"*** Proceedings of the Eighth Annual IEEE International ASIC Conference and Exhibit, pp. 71-74, 1995.

[48] GERARDO GONZALEZ-ALTAMIRANO, ALEJANDRO DIAZ-SANCHEZ Y JAIME RAMIREZ-ANGULO. ***"Fast Sampled-Data Wavelet Transform CMOS VLSI Implementation"*** IEEE 39th Midwest symposium on Circuits and Systems, Vol. 1, pp. 101-104, 1996.

[49] S. GANESAN, S. MAHALINGAM Y S. NAGABHUSHANA. ***"VLSI Synthesis of a Programmable DWT Chip for the Optimal Choice of a Prototype Wavelet"*** IEEE Third Great Lakes Symposium on Design Automation of High Performance VLSI Systems, pp. 127-131, 1993.

[50] R. TIMOTHY EDWARDS Y GERT CAUWENBERGHS. ***"A VLSI Implementation of the Continuous Wavelet Transform"*** IEEE International Symposium on Circuits and Systems, Vol. 4, pp. 368-371, 1996.

[51] TRACY C. DENK Y KESHAB K. PARHI. ***"Systolic VLSI Architectures for 1-D Discrete Wavelet Transforms"*** IEEE Conference on Signals, Systems & Computers, Vol. 2, pp. 1220-1224, 1998.

[52] TRACY C. DENK Y KESHAB K. PARHI. ***"Systematic Design of Architectures for M-ary Tree-Structured Filter Banks"*** IEEE VLSI Signal Processing, Vol. 8, pp. 157-166, 1995.

[53] CHU YU, CHIEN-AN HSIEH Y SAO-JIE CHEN. ***"Design and Implementation of a Highly Efficient VLSI Architecture for Discrete Wavelet Transform"*** IEEE, Custom Integrated Circuits Conference, pp. 237-240, 1997.

[54] H. CHOI, W.P. BURLESON Y D.S. PHATAK. ***"Optimal Wordlength Assignment for the Discrete Wavelet Transform in VLSI"*** IEEE VLSI Signal Processing, Vol. 6, pp. 325-333, 1993.

[55] ENCARNACIÓN MOYANO, PASCUAL GONZALEZ, LUIS OROZCO-BARBOSA Y FRANCISCO QUILES. ***"Experimental Study of a Parallel Wavelets Compression Systems for Medical Applications"*** IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, pp. 189-192, 1999.

[56] MARC ANTONINI, MICHEL BARLAUD Y INGRID DAUBECHIES. ***"Image Coding Using Wavelet Transform"*** IEEE, Transactions on Image Processing, Vol. 1, Nº. 2, pp. 205-220, 1992.

[57] INGRID DAUBECHIES. ***"Where Do Wavelets Com From? A Personal Point of View"*** IEEE Proceeding, Vol. 84, Nº. 4, pp. 510-513, 1996.

[58] MISLAV GRGIC, MARIO RAVNJAK Y BRANKA ZOVKO-CIHLAR. ***"Filter Comparision in Wavelet Transform of Still Images"*** Proceedings of the IEEE International Symposium on Industrial Electronics, Vol. 1, pp. 105-110, 1999.

[59] DONG WEI, JUN TIAN, RAYMOND O. WELLS Y C. SIDNEY BURRUS. ***"A new Class of Biorthogonal Wavelet Systems for Image Transform Coding"*** IEEE Transactions On Image Processing, Vol. 7, Nº. 7, pp. 1000-1013, 1998.

[60] U. MEYER-BAESE, J. BUROS, W. TRAUTMANN, Y F. TAYLOS. ***"Fast Implementation of orthogonal wavelet filterbanks using Field-***

Programmble Logic" IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 4, pp. 2119-2122, 1999.

[61] C. SIDNEY BURRUS, Y JAN E ODEGARD. ***"Generalized Coiflet Systems"*** IEEE International Conference on Digital Signal Processing, Vol. 1, pp. 321-324, 1997.

[62] JOHN D. VILLASENOR, BENJAMIN BELZER Y JUDY LIAO. ***"Filter Evaluation and Selection in Wavelet Image Compression"*** IEEE Data Compression Conference, pp. 351-360, 1994.

[63] JOHN D. VILLASENOR, BENJAMIN BELZER Y JUDY LIAO. ***"Wavelet Filter Evaluation for Image Compression"*** IEEE,Transactions on Image Processing, Vol. 4, Nº. 8, pp.1053-1060, 1995.

[64] DAVID B.H. TAY. ***"Families of Binary Coefficient Biothogonal Wavelet Filters"*** ISCAS 2000. IEEE International Symposium on Circuits and Systems, Vol. 3, pp. 371-374, 2000.

[65] C. SIDNEY BURRUS, Y JAN E. ODEGARD. ***"Smooth Biorthogonal Wavelets for Applications in Image Compression"*** IEEE Digital Signal Processing Workshop Proceedings, pp. 73-76, 1996.

[66] W. S. LU Y A. ANTONIOU. ***"Optimized Orthogonal and Biorthogonal wavelets Using Linear Parameterization of Halfband Filters"*** IEEE International Symposium on Circuits and Systems, ISCAS '98. Vol. 5, pp. 102-105, 1998.

[67] DONG WEI, BRIAN L. EVANS, Y ALAN C. BOVIK, ***"Biorthogonal Quincunx Coifman Wavelets"*** IEEE International Conference on Image Processing, Vol. 2, pp. 246-249, 1997.

[68] MANUEL A. SOLA Y SEBASTIÀ SALLENT. ***"The Discrete-Time Biorthogonal Wavelet Transform"*** IEEE International Symposium on Information Theory, p. 430, 1995.

[69] I. DAUBECHIES Y A. J. E. M. JANSSEN. ***"Two Theorems on Lattice Expansions"*** IEEE Transactions on Information Theory, Vol. 39, Nº. 1, pp. 3-6, 1993.

[70] I. DAUBECHIES. ***"Time-Frecuency Localization Operators: Ageometric Phase Space Approach"*** IEEE Transactions on Information Theory, Vol. 34, Nº. 4, pp. 605-612, 1988.

[71] Y. ZHAO Y M. N. S. SWAMY. ***"Technique for Designing Biorthogonal Wavelets Filters With an Aplication to Image Compression"*** Electronics Letters Vol. 35, Nº.18, pp. 1530-1532, 1999.

[72] HYUNGJUN KIM. ***"Image Compression Using Biorthogonal Wavelet Transforms with Multiplierless 2-D Filter Mask Operation"*** IEEE International Conference on Image Processing, Vol. 1, pp. 648-651, 1997.

[73] LOWELL L. WINGER Y ANASTASIOS N. VENETSANOPOULOS. ***"Biortohogonal Modified Coiflet Filters for Image Compression"*** IEEE International Conference on Acoustics, Speech and Signal Processing, Vol. 5, pp. 2681-2684, 1998.

[74] GUILHERME FERREIRA RIBEIRO Y GELSON VIEIRA MENDONÇA. ***"Image Coding Using Biorthogonal Wavelet Transform"*** IEEE Midwest Symposium on Circuits and Systems, Vol. 2, pp. 866-868, 1996.

[75] CAJETAN M. AKUJUOBI. ***"The Effects of Different Wavelets on Image Reconstruccion"*** Proceedings of the IEEE Bringing Together Education, Science and Technology, pp. 293-296, 1996.

[76] T. COOKLEV, A. NISHIHARA, T. YOSHIDA Y M. SABLATASH. ***"Regular Multidimensional Linear Phase FIR Digital Filter Banks and Wavelet Bases"*** IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 2, pp. 1464-1467, 1995.

[77] CHAI W. KIM, RASHID ANSARI Y A. ENIS CETIN. ***"A Class of Linear Phase Regular Biorthogonal Wavelets"*** IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 4, pp. 673-676, 1992.

[78] NIKOLAOS V. BOULGOURIS, ATHANASIOS LEONTARIS Y MICHAEL G. STRINTZIS. ***"Wavelet Compression of 3D Medical Images Using Conditional Arithmetic Coding"*** ISCAS 2000. IEEE International Symposium on Circuits and Systems, Vol. 4, pp. 557-560, 2000.

[79] SHAHID MASUD Y JOHN V. MCCANNY. ***"Rapid VLSI Design of Biorthogonal Wavelet Transform Cores"*** IEEE Workshop on Signal Processing Systems, pp. 291-300, 1999.

[80] CHIEN-YU CHEN, ZHONG-LAN YANG, TU-CHIH WANG Y LIANG-GEE CHEN. ***"A Programmable VLSI Architecture for 2-D Discrete Wavelet Transform"*** ISCAS 2000. IEEE International Symposium on Circuits and Systems, Vol. 1, pp. 619-622, 2000.

[81] MICHAEL L. HILTON, BJÖRN D. JAWERTH Y AYAN SENGUPTA. ***"Compressing Still and Moving Images with Wavelets"*** Multimedia Systems, Vol. 2, Nº. 3, 1994.

[82] ALI N. AKANSU. ***"Wavelets and Filter Banks A Signal Processing Perspective"*** IEEE Circuits & Devices Magazine, Vol. 10, Nº. 6, pp. 14-18, 1994.

[83] ALI N. AKANSU. ***"Multiplierless PR Quadrature Mirror filters for Subband Image Coding"*** IEEE Transactions on Image Processing, Vol. 5, Nº. 9, pp. 1359-1363, 1996.

[84] ETIENNE KUNTZEL Y T.Q. NGUYEN. ***"On Structure of Dyadic Symmetric Wavelets with Integer Coefficients"*** IEEE International Symposium on Circuits and Systems, Vol. 1, pp. 181-184, 1997.

[85] ETIENNE KUNTZEL Y T.Q. NGUYEN. ***"Lossless Image Coder With Low Power Implementation"*** IEEE Conference on Signals, Systems & Computers, Vol. 2, pp. 1460-1464, 1997.

[86] S. SRIRANGANATHAN, D. R. BULL Y D. W. REDMILL. ***"Design of 2-D Multiplierless FIR Filters Using Genetic Algorithms"*** IEE Genetic Algorithms in Engineering Systems: Innovations and Appications, pp. 282-286, 1995.

[87] R. R. MARTIN, D. R. BULL Y D. W. REDMILL. ***"Design of Multiplier Free Linear Phase Perfect Reconstruction Filter Banks Using Transformations and Genetic Algorithms"*** IEE, IPA97, pp. 766-770, 1997.

[88] N. G. KINGSBURY Y D. B. H. TAY. ***"Multidimensinal Wavelet Design Using Generalised Transformations"*** IEE Colloquium on Applications of Wavelet Transforms in Image Processing, Vol. 3, pp. 1-6, 1993.

[89] M. OSLICK, I. R. LINSCOTT, S. MASLAKOVIC Y J. D. TWICKEN. ***"A General Approach to the Generation of Biorthogonal Bases of Compactly-Supported Wavelets"*** IEEE International Conference on Acoustics, Speech and Signal Processing, Vol. 3, pp.1537-1540, 1998.

[90] KONSTANTINOS SLAVAKIS Y ISAO YAMADA. ***"Biorthogonal Bases of Compactly Supported Matrix Valued Wavelets"*** Fifth International Symposium on Signal Processing and its Applications, ISSPA '99, Vol. 2, pp. 981-985, 1999.

[91] J. HERON, D. TRAINOR Y R. WOODS. ***"Image Compression Algorithms Using Re-configurable Logic"*** IEEE Conference on Signals, Systems & Computers, Vol. 1, pp. 399-403, 1997.

[92] CHAITALI CHAKRABARTI, MOHAN VISHWANATH Y ROBERT M. OWENS. ***"A Survey of Architectures for the Discrete and Continuous Wavelet Transforms"*** IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 2849-2852, 1995.

[93] CHAITALI CHAKRABARTI Y MOHAN VISHWANATH. ***"Efficient Realizations of the Discrete and Continuous Wavelet Transforms: From Single Chip Implementations to Mapping on SIMD Array Computers"*** IEEE Transactions on Signal Processing, Vol. 43, Nº. 3, pp. 751-771, 1995.

[94] S. B. ARURU, N. RANGANATHAN Y K. R. NAMUDURI. ***"A VLSI Chip for Image Compression Using Variable Block Size Segmentation"*** IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 500-505, 1996.

[95] A. S. LEWIS Y G. KNOWLES. ***"Video Compression Using 3D Wavelet Transforms"*** Electronics Letters, Vol. 26, Nº. 6, pp. 396-398, 1990.

[96] C. CHAKRABARTI Y C. MUMFORD. ***"Efficient Realizations of Encoders and Decoders Based on the 2-D Discrete Wavelet Transform"*** IEEE

Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 7, Nº. 3, pp. 289-298, 1999.

[97] YOUSEF M. HAWWAR, ALI M. REZA Y ROBERT D. TURNEY. ***"Filtering (Denoising) in the Transform Domain"*** Xilinx, Inc. 1999.

[98] LIXIA LI, WENBO WANG Y DEJUNG WANG. ***"Wavelet Based Spread Spectrum Communication and Performance Analysis"*** International Conference on Signal Processing Applications & Technology, ICSPAT'97, 1997.

[99] ZIXIANG XIONG, KANNAN RAMCHANDRAN, MICHAEL T. ORCHARD Y YA-QIN ZHANG. ***"A Comparative Study of DCT and Wavelet-Based Image Coging"*** IEEE Transactions on Circuits and Systems for Video Technology, Vol. 9, Nº. 5, pp. 692-695, 1999.

[100] C.I. BROWN, N.A. THACKER Y R.B. YATES. ***"A VLSI Device for Low Bit Rate Coding"*** IEE Colloquium on Low Bit Image Coding, Vol. 5, pp. 1-6, 1995.

[101] JEONG-HO PARK; JAE-HO CHOI Y HOON-SUNG KWAK. ***"A New Scheme for Embedded Wavelet Image Coding"*** International Conference on Signal Processing Applications & Technology, ICSPAT'98, 1998.

[102] LES MINTZER. ***"A 100 Megasample/sec FPGA-based DFT"*** Momentum Data Systems, pp. 659-663, 1998.

[103] I. JOUNY Y C. WU. ***"Wavelet Division Multiple Access"*** International Conference on Signal Processing Applications & Technology, ICSPAT'98, pp. 312-316, 1998.

[104] ARMEIN Z. R. LANGI, RUBEN CH. A. LALAMENTIK Y KUDRAT SOEMINTAPURA. ***"A Field Programmable Gate Array (FPGA) Implementation of a Counter-Based Lossless Data Compression"*** International Conference on Signal Processing Applications & Technology, ICSPAT'98, 1998.

[105] A. ASMARI, S. KWATRA Y P. SUKUMAR. ***"Lossless Image Compression using Binary Field Transform"*** International Conference on Signal Processing Applications & Technology, ICSPAT'99, 1999.

[106] MATHIAS JOHANSON. ***"Scalable Video Conferencing Using Subband Transform Coding and Layered Multicast Transmission"*** International Conference on Signal Processing Applications & Technology, ICSPAT'99, 1999.

[107] IRINA BOCHAROVA, VICTOR KOLESNIK, BORIS KUDRYASHOV Y ANDREY MALKOV. ***"Two-Dimensional Hierarchical Quantizing and Coding for Wavelet Image Compression"*** International Conference on Signal Processing Applications & Technology, ICSPAT'97, pp. 1233-1237, 1997.

[108] JEONG-HO PARK, BYUNG-HA, JAE-HO CHOI Y HOON-SUNG KWAK. ***"Image Compression by Texture Modeling of Wavelet Coefficients"*** International

Conference on Signal Processing Applications & Technology, ICSPAT'97, pp. 1208-1212, 1997.

[109] TAKAO NISHITANI. ***"Low-Power Architectures for Programmable Multimedia Processors"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 2, pp. 184-196, 1999.

[110] TOMIO KISHIMOTO, HIRONORI YAMAUCHI Y RYOTA KASAI. ***"System Electronics Technologies for Video Processing and Applications"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 2, pp. 197-205, 1999.

[111] KUNITOSHI KOMATSU Y KAORU SEZAKI. ***"Design of Lossless Blocks Transforms and Filter Banks for Image Coding"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 8, pp. 1656-1664, 1999.

[112] HIDEO OHIRA Y FUMITOSHI KARUBE. ***"A Memory Reduction Approach for MPEG Decoding System"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 8, pp. 1588-1591, 1999.

[113] DONGJU LI, LI JIANG Y HIROAKI KUNIEDA. ***"Design Optimization of VLSI Array Processor Architecture for Window Image Processing"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 8, pp. 1475-1484, 1999.

[114] KOICHI KUZUME Y KOICHI NIIJIMA. ***"Design of Time-Varying Lifting Wavelet Filters"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 8, pp. 1412-1419, 1999.

[115] YUSUKE TOKUNAGA Y TAKAHIRO INOUE. ***"A Method for Circular Pattern Recognition in a Binary Image and Its Implementation onto an FPGA"*** IEICE Trans. Fundamentals, Vol. E82-A, Nº. 2, pp. 246-253, 1999.

[116] RICARDO J. COLOM, RAFAEL GADEA Y ANGEL SEBASTIÁ. ***"Study and Analysis of Filters Wavelet for the Compression of Images"*** IX Spanish Symposium on Pattern Recognition and Image Analysis. Vol. II, pp. 311-316, 2001.

[117] RICARDO J. COLOM, RAFAEL GADEA, ANGEL SEBASTIÁ, MARCOS MARTINEZ, VICENTE HERRERO Y VICENTE ARNAU. ***"Implementation of 2-D Discrete Wavelet Transform for Real-Time Video Signal Processing"*** WSES/IEEE International Conference on Speech, Signal and Image Processing. Septiembre 2001.

Apéndice A

Listados Ficheros VHDL

En este apéndice se presentan los listados de los ficheros descritos en lenguaje VHDL, utilizados en la implementación denominada recurrente, que se mostró en el apartado 5.7 (ver pagina 96).

Entidad : Test_General

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Test_General is
end Test_General;

architecture simulacion of Test_General is

component Wavelet_G42
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            pixels : in std_logic_vector(27 downto 0);
            direc_rd : out std_logic_vector(16 downto 0);
            enable_rd : out std_logic;
            pixels_paa_o1 : out std_logic_vector(13 downto 0);
            pixels_pab_o1 : out std_logic_vector(13 downto 0);
            pixels_pba_o1 : out std_logic_vector(13 downto 0);
--          pixels_pbb_o1 : out std_logic_vector(13 downto 0);
            pixels_paa_o2 : out std_logic_vector(13 downto 0);
            pixels_pab_o2 : out std_logic_vector(13 downto 0);
            pixels_pba_o2 : out std_logic_vector(13 downto 0);
--          pixels_pbb_o2 : out std_logic_vector(13 downto 0);
            pixels_paa_o3 : out std_logic_vector(13 downto 0);
            pixels_pab_o3 : out std_logic_vector(13 downto 0);
```

```
            pixels_pba_o3 : out std_logic_vector(13 downto 0);
            pixels_pbb_o3 : out std_logic_vector(13 downto 0));
end component;

component dualmem_img_2B
      port( clk: in std_logic;
            reset : in std_logic;
            enable : in std_logic;
            write : in std_logic;
            direc_a : in std_logic_vector(17 downto 0);
            direc_b : in std_logic_vector(16 downto 0);
            data_in_a : in std_logic_vector(7 downto 0);
            data_out_b : out std_logic_vector(15 downto 0));
end component;

signal clk : std_logic;
signal inic : std_logic;
signal direccion : std_logic_vector(16 downto 0);
signal rst : std_logic;
signal ena_rd : std_logic;
signal writer : std_logic;
signal datos_in : std_logic_vector(7 downto 0);
signal datos_out : std_logic_vector(15 downto 0);
signal pixelb : std_logic_vector(27 downto 0);
signal direccion_a : std_logic_vector(17 downto 0);

signal paa_o1, pab_o1, pba_o1 : std_logic_vector(13 downto 0);
signal paa_o2, pab_o2, pba_o2 : std_logic_vector(13 downto 0);
signal paa_o3, pab_o3, pba_o3, pbb_o3 : std_logic_vector(13 downto 0);

begin

U1 : Wavelet_G42
      port map(
            clock => clk,
            reset => rst,
            inicio => inic,
            pixels => pixelb,
            direc_rd => direccion,
            enable_rd => ena_rd,
            pixels_paa_o1 => paa_o1,
            pixels_pab_o1 => pab_o1,
            pixels_pba_o1 => pba_o1,
--          pixels_pbb_o1 => pbb_o1,
            pixels_paa_o2 => paa_o2,
            pixels_pab_o2 => pab_o2,
            pixels_pba_o2 => pba_o2,
--          pixels_pbb_o2 => pbb_o2,
            pixels_paa_o3 => paa_o3,
            pixels_pab_o3 => pab_o3,
            pixels_pba_o3 => pba_o3,
            pixels_pbb_o3 => pbb_o3);

U2 : dualmem_img_2B
            port map (
                  clk => clk,
```

```
                  reset => rst,
                  enable => ena_rd,
                  write => writer,
                  direc_a => direccion_a,
                  direc_b => direccion,
                  data_in_a => datos_in,
                  data_out_b => datos_out);

pixelb(7 downto 0) <= datos_out(7 downto 0);
pixelb(13 downto 8) <= (others => '0');
pixelb(21 downto 14) <= datos_out(15 downto 8);
pixelb(27 downto 22) <= (others => '0');

estimulos: process
      begin
      rst <= '0';
      inic <= '1';
      writer <= '1';
      datos_in <= (others => '0');
      direccion_a <= (others => '0');
      wait for 100 ns;
      rst <= '1';
      wait for 100 ns;
      inic <= '0';
      wait for 200 ns;
      inic <= '1';
      wait for 200 ns;
      wait;
      end process;

reloj: process
      begin
      clk <='0';
      loop
            wait for 25 ns;
            clk <= not clk;
      end loop;
end process reloj;

end simulacion;

configuration cfg_Test_General of Test_General is
    for simulacion
          for U1 : Wavelet_G42 use configuration work.cfg_Wavelet_G42;
          end for;
          for U2 : dualmem_img_2B use configuration
work.cfg_dualmem_img_2B;
          end for;
    end for;
end cfg_Test_General;
```

Entidad : Dualmem_img_2b

```
library IEEE;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity dualmem_img_2B is
      port( clk: in std_logic;
            reset : in std_logic;
            enable : in std_logic;
            write : in std_logic;
            direc_a : in std_logic_vector(17 downto 0);
            direc_b : in std_logic_vector(16 downto 0);
            data_in_a : in std_logic_vector(7 downto 0);
            data_out_b : out std_logic_vector(15 downto 0));
end dualmem_img_2B;

architecture use_core of dualmem_img_2B is

component dualmem_2B
      port (
      addra: IN std_logic_VECTOR(17 downto 0);
      clka: IN std_logic;
      addrb: IN std_logic_VECTOR(16 downto 0);
      clkb: IN std_logic;
      dia: IN std_logic_VECTOR(7 downto 0);
      wea: IN std_logic;
      enb: IN std_logic;
      rstb: IN std_logic;
      dob: OUT std_logic_VECTOR(15 downto 0));
end component;


begin

U1 : dualmem_2B
      port map (
            addra => direc_a,
            clka => clk,
            addrb => direc_b,
            clkb => clk,
            dia => data_in_a,
            wea => write,
            enb => enable,
            rstb => reset,
            dob => data_out_b);
end use_core;

-- synopsys translate_off

Library XilinxCoreLib;

-- synopsys translate_on

-- synopsys translate_off
configuration cfg_dualmem_img_2B of dualmem_img_2B is
    for use_core
```

```
      for U1 : dualmem_2B use entity
XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
            generic map(
                  c_depth_b => 131072,
                  c_depth_a => 262144,
                  c_has_web => 0,
                  c_has_wea => 1,
                  c_has_dib => 0,
                  c_has_dia => 1,
                  c_clka_polarity => 1,
                  c_web_polarity => 1,
                  c_address_width_b => 17,
                  c_address_width_a => 18,
                  c_width_b => 16,
                  c_width_a => 8,
                  c_clkb_polarity => 1,
                  c_ena_polarity => 1,
                  c_rsta_polarity => 1,
                  c_has_rstb => 1,
                  c_has_rsta => 0,
                  c_read_mif => 1,
                  c_enb_polarity => 0,
                  c_pipe_stages => 0,
                  c_rstb_polarity => 0,
                  c_has_enb => 1,
                  c_has_ena => 0,
                  c_mem_init_radix => 10,
                  c_default_data => "10",
                  c_mem_init_file => "cuadro_C.mif",
                  c_has_dob => 1,
                  c_generate_mif => 0,
                  c_has_doa => 0,
                  c_wea_polarity => 0);
      end for;
    end for;
end cfg_dualmem_img_2B;
-- synopsys translate_on
```

Entidad : Wavelet_G42

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Wavelet_G42 is
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            pixels : in std_logic_vector(27 downto 0);
            direc_rd : out std_logic_vector(16 downto 0);
            enable_rd : out std_logic;
            pixels_paa_o1 : out std_logic_vector(13 downto 0);
            pixels_pab_o1 : out std_logic_vector(13 downto 0);
            pixels_pba_o1 : out std_logic_vector(13 downto 0);
```

```
--          pixels_pbb_o1 : out std_logic_vector(13 downto 0);
            pixels_paa_o2 : out std_logic_vector(13 downto 0);
            pixels_pab_o2 : out std_logic_vector(13 downto 0);
            pixels_pba_o2 : out std_logic_vector(13 downto 0);
--          pixels_pbb_o2 : out std_logic_vector(13 downto 0);
            pixels_paa_o3 : out std_logic_vector(13 downto 0);
            pixels_pab_o3 : out std_logic_vector(13 downto 0);
            pixels_pba_o3 : out std_logic_vector(13 downto 0);
            pixels_pbb_o3 : out std_logic_vector(13 downto 0));
end Wavelet_G42;

architecture rtl of Wavelet_G42 is

component Generador_G41
      generic ( NN : integer := 512);
      port( clock : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : out std_logic;
            fin_data_A : out std_logic;
            fin_data_B : out std_logic;
            fin_fila_A : out std_logic;
            fin_fila_B : out std_logic;
            direcciones : out std_logic_vector(16 downto 0));
end component;

component Octava_G51
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene: in std_logic;
            fin_data_A: in std_logic;
            fin_data_B: in std_logic;
            fin_fila_A: in std_logic;
            fin_fila_B: in std_logic;
            stop : in std_logic;
            inic_gene: out std_logic;
            enable_rd : out std_logic;
            reg_data : out std_logic;
            pixels : in std_logic_vector(27 downto 0);
            pixels_paa : out std_logic_vector(13 downto 0);
            pixels_pab : out std_logic_vector(13 downto 0);
            pixels_pba : out std_logic_vector(13 downto 0);
            pixels_pbb : out std_logic_vector(13 downto 0));
end component;

component Octava_G61
      port( clock : in std_logic;
            reset : in std_logic;
            inicio_o2 : in std_logic;
            fin_gene_o2: in std_logic;
            fin_data_A_o2: in std_logic;
            fin_data_B_o2: in std_logic;
            fin_fila_A_o2: in std_logic;
            fin_fila_B_o2: in std_logic;
            stop_o2 : in std_logic;
```

```
            inic_gene_o2: out std_logic;
            enable_rd_o2 : out std_logic;
            reg_data_o2 : out std_logic;
            inicio_o3 : in std_logic;
            fin_gene_o3: in std_logic;
            fin_data_A_o3: in std_logic;
            fin_data_B_o3: in std_logic;
            fin_fila_A_o3: in std_logic;
            fin_fila_B_o3: in std_logic;
            stop_o3 : in std_logic;
            inic_gene_o3: out std_logic;
            enable_rd_o3 : out std_logic;
            reg_data_o3 : out std_logic;
            selec_oct : out std_logic;
            pixels : in std_logic_vector(27 downto 0);
            pixels_paa_o2 : out std_logic_vector(13 downto 0);
            pixels_pab_o2 : out std_logic_vector(13 downto 0);
            pixels_pba_o2 : out std_logic_vector(13 downto 0);
            pixels_pbb_o2 : out std_logic_vector(13 downto 0);
            pixels_paa_o3 : out std_logic_vector(13 downto 0);
            pixels_pab_o3 : out std_logic_vector(13 downto 0);
            pixels_pba_o3 : out std_logic_vector(13 downto 0);
            pixels_pbb_o3 : out std_logic_vector(13 downto 0));
end component;

component Guardar_G21
      generic ( NN : integer := 512);
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene : in std_logic;
            reg_data : in std_logic;
            direc : out std_logic_vector(10 downto 0);
            write_mem : out std_logic);
end component;

component Mux2to1_G21
      port( data_in1_A : in std_logic_vector(27 downto 0);
            data_in1_B : in std_logic_vector(27 downto 0);
            selec : in std_logic;
            data_out : out std_logic_vector(27 downto 0));
end component;

component dualmem_pas21
      port( clock: in std_logic;
            reset : in std_logic;
            read : in std_logic;
            write : in std_logic;
            direc_a : in std_logic_vector(10 downto 0);
            direc_b : in std_logic_vector(9 downto 0);
            data_in_a : in std_logic_vector(13 downto 0);
            data_out_b : out std_logic_vector(27 downto 0));
end component;

component Generador_G31
      generic ( NN : integer := 256);
```

```
      port( clock : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : out std_logic;
            fin_data_A : out std_logic;
            fin_data_B : out std_logic;
            fin_fila_A : out std_logic;
            fin_fila_B : out std_logic;
            direcciones : out std_logic_vector(9 downto 0));
end component;

component Control_General_G11
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene_o1 : in std_logic;
            fin_gene_o2 : in std_logic;
            fin_fila_B_o1 : in std_logic;
            fin_fila_B_o2 : in std_logic;
            inicio_octava_1 : out std_logic;
            inicio_octava_2 : out std_logic;
            stop_octava_1 : out std_logic;
            stop_octava_2 : out std_logic);
end component;

constant size_image : integer := 512;

-- Señales Primera Octava
signal inic_d_A_o1, inic_d_B_o1, inic_g_o1, stop_o1, inicio_o1 :
std_logic;
signal fin_g_o1, fin_d_A_o1, fin_d_B_o1, fin_f_A_o1, fin_f_B_o1 :
std_logic;
signal enable_rg_o1, enable_wr_pbb_o1 : std_logic;

--signal pixels_paa_o1, pixels_pab_o1 : std_logic_vector(13 downto 0);
--signal pixels_pba_o1, pixels_pbb_o1 : std_logic_vector(13 downto 0);
signal pixels_pbb_o1 : std_logic_vector(13 downto 0);

signal d_wr_pbb_o1 : std_logic_vector(10 downto 0);

-- Señales Segunda Octava
signal inic_d_A_o2, inic_d_B_o2, inic_g_o2, stop_o2, inicio_o2 :
std_logic;
signal fin_g_o2, fin_d_A_o2, fin_d_B_o2, fin_f_A_o2, fin_f_B_o2 :
std_logic;
signal enable_rg_o2, enable_wr_pbb_o2 : std_logic;

--signal pixels_paa_o2, pixels_pab_o2 : std_logic_vector(13 downto 0);
--signal pixels_pba_o2, pixels_pbb_o2 : std_logic_vector(13 downto 0);
signal pixels_pbb_o2 : std_logic_vector(13 downto 0);

signal read_pbb_o2 : std_logic;
signal direc_rd_pbb_o2 : std_logic_vector(9 downto 0);
signal data_out_pbb_o2 : std_logic_vector(27 downto 0);

signal d_wr_pbb_o2 : std_logic_vector(10 downto 0);
```

```
-- Señales Tercera Octava
signal inic_d_A_o3, inic_d_B_o3, inic_g_o3, stop_o3, inicio_o3 :
std_logic;
signal fin_g_o3, fin_d_A_o3, fin_d_B_o3, fin_f_A_o3, fin_f_B_o3 :
std_logic;
signal enable_rg_o3, enable_wr_pbb_o3 : std_logic;

--signal pixels_paa_o3, pixels_pab_o3 : std_logic_vector(13 downto 0);
--signal pixels_pba_o3, pixels_pbb_o3 : std_logic_vector(13 downto 0);

signal read_pbb_o3 : std_logic;
signal direc_rd_pbb_o3 : std_logic_vector(9 downto 0);
signal data_out_pbb_o3 : std_logic_vector(27 downto 0);

signal d_wr_pbb_o3 : std_logic_vector(10 downto 0);

signal data_out_pbb_o2_o3 : std_logic_vector(27 downto 0);
signal selec_o2_o3 : std_logic;

begin

U1 : Generador_G41
      generic map (NN => size_image)
      port map (
            clock => clock,
            inicio => inic_g_o1,
            pausa => stop_o1,
            fin_gene => fin_g_o1,
            fin_data_A => fin_d_A_o1,
            fin_data_B => fin_d_B_o1,
            fin_fila_A => fin_f_A_o1,
            fin_fila_B => fin_f_B_o1,
            direcciones => direc_rd);

U2 : Octava_G51
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o1,
            fin_gene => fin_g_o1,
            fin_data_A => fin_d_A_o1,
            fin_data_B => fin_d_B_o1,
            fin_fila_A => fin_f_A_o1,
            fin_fila_B => fin_f_B_o1,
            stop => stop_o1,
            inic_gene => inic_g_o1,
            enable_rd => enable_rd,
            reg_data => enable_rg_o1,
            pixels => pixels,
            pixels_paa => pixels_paa_o1,
            pixels_pab => pixels_pab_o1,
            pixels_pba => pixels_pba_o1,
            pixels_pbb => pixels_pbb_o1);
```

```
U4 : Guardar_G21
      generic map (NN => size_image)
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o1,
            fin_gene => fin_g_o1,
            reg_data => enable_rg_o1,
            direc => d_wr_pbb_o1,
            write_mem => enable_wr_pbb_o1);

U5 : dualmem_pas21
      port map(
            clock => clock,
            reset => reset,
            read => read_pbb_o2,
            write => enable_wr_pbb_o1,
            direc_a => d_wr_pbb_o1,
            direc_b => direc_rd_pbb_o2,
            data_in_a => pixels_pbb_o1,
            data_out_b => data_out_pbb_o2);

-- Segunda Octava

U6 : Generador_G31
      generic map (NN => size_image/2)
      port map (
            clock => clock,
            inicio => inic_g_o2,
            pausa => stop_o2,
            fin_gene => fin_g_o2,
            fin_data_A => fin_d_A_o2,
            fin_data_B => fin_d_B_o2,
            fin_fila_A => fin_f_A_o2,
            fin_fila_B => fin_f_B_o2,
            direcciones => direc_rd_pbb_o2);

U7 : Octava_G61
      port map(
            clock => clock,
            reset => reset,
            inicio_o2 => inicio_o2,
            fin_gene_o2 => fin_g_o2,
            fin_data_A_o2 => fin_d_A_o2,
            fin_data_B_o2 => fin_d_B_o2,
            fin_fila_A_o2 => fin_f_A_o2,
            fin_fila_B_o2 => fin_f_B_o2,
            stop_o2 => stop_o2,
            inic_gene_o2 => inic_g_o2,
            enable_rd_o2 => read_pbb_o2,
            reg_data_o2 => enable_rg_o2,
            inicio_o3 => inicio_o3,
            fin_gene_o3 => fin_g_o3,
            fin_data_A_o3 => fin_d_A_o3,
            fin_data_B_o3 => fin_d_B_o3,
            fin_fila_A_o3 => fin_f_A_o3,
```

```
            fin_fila_B_o3 => fin_f_B_o3,
            stop_o3 => stop_o3,
            inic_gene_o3 => inic_g_o3,
            enable_rd_o3 => read_pbb_o3,
            reg_data_o3 => enable_rg_o3,
            selec_oct => selec_o2_o3,
            pixels => data_out_pbb_o2_o3,
            pixels_paa_o2 => pixels_paa_o2,
            pixels_pab_o2 => pixels_pab_o2,
            pixels_pba_o2 => pixels_pba_o2,
            pixels_pbb_o2 => pixels_pbb_o2,
            pixels_paa_o3 => pixels_paa_o3,
            pixels_pab_o3 => pixels_pab_o3,
            pixels_pba_o3 => pixels_pba_o3,
            pixels_pbb_o3 => pixels_pbb_o3);

U8 : Mux2to1_G21
      port map(
            data_in1_A => data_out_pbb_o2,
            data_in1_B => data_out_pbb_o3,
            selec => selec_o2_o3,
            data_out => data_out_pbb_o2_o3);

U9 : Guardar_G21
      generic map (NN => size_image/2)
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o2,
            fin_gene => fin_g_o2,
            reg_data => enable_rg_o2,
            direc => d_wr_pbb_o2,
            write_mem => enable_wr_pbb_o2);

U10 : dualmem_pas21
      port map(
            clock => clock,
            reset => reset,
            read => read_pbb_o3,
            write => enable_wr_pbb_o2,
            direc_a => d_wr_pbb_o2,
            direc_b => direc_rd_pbb_o3,
            data_in_a => pixels_pbb_o2,
            data_out_b => data_out_pbb_o3);

-- Tercera Octava

U11 : Generador_G31
      generic map (NN => size_image/4)
      port map (
            clock => clock,
            inicio => inic_g_o3,
            pausa => stop_o3,
            fin_gene => fin_g_o3,
            fin_data_A => fin_d_A_o3,
            fin_data_B => fin_d_B_o3,
```

```
            fin_fila_A => fin_f_A_o3,
            fin_fila_B => fin_f_B_o3,
            direcciones => direc_rd_pbb_o3);

U14 : Guardar_G21
      generic map (NN => size_image/4)
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o3,
            fin_gene => fin_g_o3,
            reg_data => enable_rg_o3,
            direc => d_wr_pbb_o3,
            write_mem => enable_wr_pbb_o3);

U15 : Control_General_G11
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio,
            fin_gene_o1 => fin_g_o1,
            fin_gene_o2 => fin_g_o2,
            fin_fila_B_o1 => fin_f_B_o1,
            fin_fila_B_o2 => fin_f_A_o2,
            inicio_octava_1 => inicio_o1,
            inicio_octava_2 => inicio_o2,
            stop_octava_1 => stop_o1,
            stop_octava_2 => stop_o2);

U16 : Control_General_G11
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio,
            fin_gene_o1 => fin_g_o2,
            fin_gene_o2 => fin_g_o3,
            fin_fila_B_o1 => fin_f_A_o2,
            fin_fila_B_o2 => fin_f_A_o3,
--          inicio_octava_1 => inicio_o2,
            inicio_octava_2 => inicio_o3,
--          stop_octava_1 => stop_o2,
            stop_octava_2 => stop_o3);

end rtl;

-- synopsys translate_off

configuration cfg_Wavelet_G42 of Wavelet_G42 is
    for rtl
          for all : dualmem_pas21 use configuration
work.cfg_dualmem_pas21;
          end for;
    end for;
end cfg_Wavelet_G42;

-- synopsys translate_on
```

Entidad : Generador_G41

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Generador_G41 is
      generic ( NN : integer := 512);
      port( clock : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : out std_logic;
            fin_data_A : out std_logic;
            fin_data_B : out std_logic;
            fin_fila_A : out std_logic;
            fin_fila_B : out std_logic;
            direcciones : out std_logic_vector(16 downto 0));
end Generador_G41;

architecture rtl of Generador_G41 is

-- constant NN : integer := 64;
signal direc : integer range 0 to NN*NN/2+NN;

begin

process (clock)

variable ii : integer range 1 to NN/2;
variable jj : integer range 1 to NN/4+1;
variable j : integer range 1 to 4;
variable i : integer range 1 to 2;

      begin
      if clock'event and clock='1' then
            if inicio = '0' then
                  if pausa = '1' then
                  if (j+4*jj-5)>NN-1 then
                        if (j+2*ii-3)>NN-1 then
                              direc <=(j+2*ii-3-NN)*NN/2+(i+2*jj-3-
NN/2);
                        else
                              direc <=(j+2*ii-3)*NN/2+(i+2*jj-3-NN/2);
                        end if;
                  else
                        if (j+2*ii-3)>NN-1 then
                              direc <=(j+2*ii-3-NN)*NN/2+(i+2*jj-3);
                        else
                              direc <=(j+2*ii-3)*NN/2+(i+2*jj-3);
                        end if;
                  end if;

                  if (i=2 and j=4) then
```

```
      fin_data_A <= '0';
else
      fin_data_A <= '1';
end if;

if (jj>NN/4) then
      fin_fila_A <= '1';
else
      fin_fila_A <= '0';
end if;

if (i=1 and j=4) then
      if jj=1 then
            fin_data_B <= '1';
      else
            fin_data_B <= '0';
      end if;
else
      fin_data_B <= '1';
end if;

if (jj=1 and i<=1 and j<=4) then
      fin_fila_B <= '1';
else
      fin_fila_B <= '0';
end if;

if (i=1 and j=4 and ii=NN/2 and jj=NN/4+1) then
      fin_gene <= '1';
else
      fin_gene <= '0';
end if;

if j=4 then
      j := 1;
      if i=2 then
            i := 1;
            if jj=NN/4+1 then
                  jj := 1;
                  if ii=NN/2 then
                        ii := 1;
                  else
                        ii := ii+1;
                  end if;
            else
                  jj := jj+1;
            end if;
      else
            if jj=NN/4+1 and i=1 then
                  jj := 1;
                  i :=1;
                  if ii=NN/2 then
                        ii := 1;
                  else
                        ii := ii+1;
                  end if;
```

```
                              else
                                    i := i+1;
                              end if;
                        end if;
                  else
                        j := j+1;
                  end if;
                  end if;
            else
                  direc <= 0;
                  fin_gene <= '1';
                  fin_data_A <= '1';
                  fin_data_B <= '1';
                  fin_fila_A <= '1';
                  fin_fila_B <= '1';
                  i := 1;
                  j := 1;
                  ii := 1;
                  jj := 1;
            end if;
      end if;
end process;

direcciones <= conv_std_logic_vector(direc,17);

end rtl;
```

Entidad : Guardar_G21

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Guardar_G21 is
      generic ( NN : integer := 512);
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene : in std_logic;
            reg_data : in std_logic;
            direc : out std_logic_vector(10 downto 0);
            write_mem : out std_logic);
end Guardar_G21;

architecture rtl of Guardar_G21 is

-- constant NN : integer := 64;

type ESTADO is (ESPERA, ESPERA_DATA, PRE_GUARDAR, GUARDAR_PBB,
PREULTIMO, ULTIMO, POST);
signal ESTADO_ACTUAL, ESTADO_SIGUIENTE : ESTADO;

signal count, next_count : integer range 0 to 4*NN+1;
```

```
begin

secuencial: process(clock, reset)
begin
      if reset = '0' then
            ESTADO_ACTUAL <= ESPERA;
      elsif clock'event and clock = '1' then
            ESTADO_ACTUAL <= ESTADO_SIGUIENTE;
            count <= next_count;
      end if;
end process secuencial;

logica_entrada: process(ESTADO_ACTUAL, inicio, fin_gene, reg_data)
begin
      case ESTADO_ACTUAL is
            when ESPERA =>
                  if inicio = '0' then
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  else
                        ESTADO_SIGUIENTE <= ESPERA;
                  end if;
            when ESPERA_DATA =>
                  if reg_data = '0' then
                        ESTADO_SIGUIENTE <= PRE_GUARDAR;
                  else
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  end if;

            when PRE_GUARDAR =>
                  ESTADO_SIGUIENTE <= GUARDAR_PBB;
            when GUARDAR_PBB =>
                  if fin_gene = '1' then
                        ESTADO_SIGUIENTE <= PREULTIMO;
                  else
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  end if;
            when PREULTIMO =>
                  ESTADO_SIGUIENTE <= ULTIMO;
            when ULTIMO =>
                  ESTADO_SIGUIENTE <= POST;
            when POST =>
                  ESTADO_SIGUIENTE <= ESPERA;
      end case;
end process logica_entrada;

logica_salida: process(ESTADO_ACTUAL, count)

variable next_count_v : integer range 0 to 4*NN+1;

begin
      case ESTADO_ACTUAL is
            when ESPERA =>
                  next_count_v := 0;
                  write_mem <= '1';
            when ESPERA_DATA =>
                  next_count_v := count;
```

```
                write_mem <= '1';

          when PRE_GUARDAR =>
                next_count_v := count;
                write_mem <= '0';
          when GUARDAR_PBB =>
                write_mem <= '0';
                if count = 4*NN-1 then
                      next_count_v := NN;
                else
                      next_count_v := count+1;
                end if;
          when PREULTIMO =>
                write_mem <= '1';
                next_count_v := count;
          when ULTIMO =>
                write_mem <= '0';
                next_count_v := count;
          when POST =>
                write_mem <= '0';
                next_count_v := count;
      end case;

next_count <= next_count_v;

end process logica_salida;

direc <= conv_std_logic_vector(count,11);

end rtl;
```

Entidad : Dualmem_pas21

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity dualmem_pas21 is
      port( clock: in std_logic;
            reset : in std_logic;
            read : in std_logic;
            write : in std_logic;
            direc_a : in std_logic_vector(10 downto 0);
            direc_b : in std_logic_vector(9 downto 0);
            data_in_a : in std_logic_vector(13 downto 0);
            data_out_b : out std_logic_vector(27 downto 0));
end dualmem_pas21;


architecture use_core of dualmem_pas21 is

component dualmem_Ato2B
      port (
      addra: IN std_logic_VECTOR(10 downto 0);
```

```
      clka: IN std_logic;
      addrb: IN std_logic_VECTOR(9 downto 0);
      clkb: IN std_logic;
      dia: IN std_logic_VECTOR(13 downto 0);
      wea: IN std_logic;
      enb: IN std_logic;
      rstb: IN std_logic;
      dob: OUT std_logic_VECTOR(27 downto 0));
end component;

begin
U1 : dualmem_Ato2B
            port map (
                  addra => direc_a,
                  clka => clock,
                  addrb => direc_b,
                  clkb => clock,
                  dia => data_in_a,
                  wea => write,
                  enb => read,
                  rstb => reset,
                  dob => data_out_b);
end use_core;

-- synopsys translate_off

Library XilinxCoreLib;

-- synopsys translate_on

-- synopsys translate_off
configuration cfg_dualmem_pas21 of dualmem_pas21 is
    for use_core
      for U1 : dualmem_Ato2B use entity
XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
            generic map(
                  c_depth_b => 1024,
                  c_depth_a => 2048,
                  c_has_web => 0,
                  c_has_wea => 1,
                  c_has_dib => 0,
                  c_has_dia => 1,
                  c_clka_polarity => 1,
                  c_web_polarity => 1,
                  c_address_width_b => 10,
                  c_address_width_a => 11,
                  c_width_b => 28,
                  c_width_a => 14,
                  c_clkb_polarity => 1,
                  c_ena_polarity => 1,
                  c_rsta_polarity => 1,
                  c_has_rstb => 1,
                  c_has_rsta => 0,
                  c_read_mif => 0,
                  c_enb_polarity => 0,
                  c_pipe_stages => 0,
```

```
                  c_rstb_polarity => 0,
                  c_has_enb => 1,
                  c_has_ena => 0,
                  c_mem_init_radix => 16,
                  c_default_data => "0",
                  c_mem_init_file => "dualmem_Ato2B.mif",
                  c_has_dob => 1,
                  c_generate_mif => 0,
                  c_has_doa => 0,
                  c_wea_polarity => 0);
      end for;
    end for;
end cfg_dualmem_pas21;

-- synopsys translate_on
```

Entidad : Generador_G31

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Generador_G31 is
      generic ( NN : integer := 256);
      port( clock : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : out std_logic;
            fin_data_A : out std_logic;
            fin_data_B : out std_logic;
            fin_fila_A : out std_logic;
            fin_fila_B : out std_logic;
            direcciones : out std_logic_vector(9 downto 0));
end Generador_G31;

architecture rtl of Generador_G31 is

-- constant NN : integer := 32;
signal direc : integer range 0 to NN*NN/2+NN;
signal fin_dato_B_int : std_logic;

begin

process (clock)

variable ii : integer range 1 to NN/2;
variable jj : integer range 1 to NN/4+1;
variable j : integer range 1 to 4;
variable i : integer range 1 to 2;
variable cii : integer range 1 to 4;

      begin
      if clock'event and clock='1' then
            if inicio = '0' then
```

```
                  if not(pausa = '0' and fin_dato_B_int = '0') then
                  if (j+4*jj-5)>NN-1 then
                        if (j+2*ii-3)>NN-1 then
                              direc <=(j+2*ii-3-NN)*NN/2+(i+2*jj-3-
NN/2);
                        elsif (j+2*cii-3)>7 then
                              direc <=(j+2*cii-3-6)*NN/2+(i+2*jj-3-
NN/2);
                        else
                              direc <=(j+2*cii-3)*NN/2+(i+2*jj-3-NN/2);
                        end if;
                  else
                        if (j+2*ii-3)>NN-1 then
                              direc <=(j+2*ii-3-NN)*NN/2+(i+2*jj-3);
                        elsif (j+2*cii-3)>7 then
                              direc <=(j+2*cii-3-6)*NN/2+(i+2*jj-3);
                        else
                              direc <=(j+2*cii-3)*NN/2+(i+2*jj-3);
                        end if;
                  end if;

                  if (i=2 and j=4) then
                        fin_data_A <= '0';
                  else
                        fin_data_A <= '1';
                  end if;

                  if (jj>NN/4) then
                        fin_fila_A <= '1';
                  else
                        fin_fila_A <= '0';
                  end if;

                  if (i=1 and j=4) then
                        if jj=1 then
                              fin_data_B <= '1';
                              fin_dato_B_int <= '1';
                        else
                              fin_data_B <= '0';
                              fin_dato_B_int <= '0';
                        end if;
                  else
                        fin_data_B <= '1';
                        fin_dato_B_int <= '1';
                  end if;

                  if (jj=1 and i<=1 and j<=4) then
                        fin_fila_B <= '1';
                  else
                        fin_fila_B <= '0';
                  end if;

                  if (i=1 and j=4 and ii=NN/2 and jj=NN/4+1) then
                        fin_gene <= '1';
                  else
                        fin_gene <= '0';
```

```
                end if;

                if j=4 then
                        j := 1;
                        if i=2 then
                                i := 1;
                                if jj=NN/4+1 then
                                        jj := 1;
                                        if ii=NN/2 then
                                                ii := 1;
                                        else
                                                ii := ii+1;
                                        end if;
                                        if cii= 4 then
                                                cii := 2;
                                        else
                                                cii := cii+1;
                                        end if;
                                else
                                        jj := jj+1;
                                end if;
                        else
                                if jj=NN/4+1 and i=1 then
                                        jj := 1;
                                        i :=1;
                                        if ii=NN/2 then
                                                ii := 1;
                                        else
                                                ii := ii+1;
                                        end if;
                                        if cii= 4 then
                                                cii := 2;
                                        else
                                                cii := cii+1;
                                        end if;
                                else
                                        i := i+1;
                                end if;
                        end if;
                else
                        j := j+1;
                end if;
                end if;
        else
                direc <= 0;
                fin_gene <= '1';
                fin_data_A <= '1';
                fin_data_B <= '1';
                fin_fila_A <= '1';
                fin_fila_B <= '1';
                i := 1;
                j := 1;
                ii := 1;
                jj := 1;
                cii := 1;
                fin_dato_B_int <= '1';
```

```
            end if;
      end if;
end process;

direcciones <= conv_std_logic_vector(direc,10);

end rtl;
```

Entidad : Control_General_G11

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Control_General_G11 is
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene_o1 : in std_logic;
            fin_gene_o2 : in std_logic;
            fin_fila_B_o1 : in std_logic;
            fin_fila_B_o2 : in std_logic;
            inicio_octava_1 : out std_logic;
            inicio_octava_2 : out std_logic;
            stop_octava_1 : out std_logic;
            stop_octava_2 : out std_logic);
end Control_General_G11;

architecture rtl of Control_General_G11 is

type ESTADO_O1 is (ESPERA_OCTAVA1, INIT_OCTAVA1, OCTAVA1, MAS_OCTAVA1,
                  OTRA_OCTAVA1, FIN_OCTAVA1);
signal ESTADO_ACTUAL_O1, ESTADO_SIGUIENTE_O1 : ESTADO_O1;
signal count1, next_count1 : integer range 0 to 8;

type ESTADO_O2 is (ESPERA_OCTAVA2, INIT_OCTAVA2, OCTAVA2,
STOP_OCTAVA2,
            POSTSTOP_OCTAVA2, MAS_OCTAVA2, OTRA_OCTAVA2, FIN_OCTAVA2);
signal ESTADO_ACTUAL_O2, ESTADO_SIGUIENTE_O2 : ESTADO_O2;
signal count2, next_count2 : integer range 0 to 3;

begin

secuencial: process(clock, reset)
begin
      if reset = '0' then
            ESTADO_ACTUAL_O1 <= ESPERA_OCTAVA1;
            ESTADO_ACTUAL_O2 <= ESPERA_OCTAVA2;
      elsif clock'event and clock = '1' then
            ESTADO_ACTUAL_O1 <= ESTADO_SIGUIENTE_O1;
            count1 <= next_count1;
            ESTADO_ACTUAL_O2 <= ESTADO_SIGUIENTE_O2;
            count2 <= next_count2;
      end if;
```

```
end process secuencial;

logica_octava1: process(ESTADO_ACTUAL_O1, inicio, fin_gene_o1,
fin_fila_B_o1, count1)
variable next_count1_v : integer range 0 to 8;
begin
      case ESTADO_ACTUAL_O1 is
            when ESPERA_OCTAVA1 =>
                  if inicio = '0' then
                        ESTADO_SIGUIENTE_O1 <= INIT_OCTAVA1;
                  else
                        ESTADO_SIGUIENTE_O1 <= ESPERA_OCTAVA1;
                  end if;
                  next_count1_v := 0;
            when INIT_OCTAVA1 =>
                  if count1 = 6 then
                        ESTADO_SIGUIENTE_O1 <= MAS_OCTAVA1;
                        next_count1_v := 0;
                  elsif fin_fila_B_o1 = '1' then
                        ESTADO_SIGUIENTE_O1 <= INIT_OCTAVA1;
                        next_count1_v := count1;
                  else
                        ESTADO_SIGUIENTE_O1 <= OCTAVA1;
                        next_count1_v := count1;
                  end if;
            when OCTAVA1 =>
                  if fin_fila_B_o1 = '1' then
                        ESTADO_SIGUIENTE_O1 <= INIT_OCTAVA1;
                        next_count1_v := count1+1;
                  else
                        ESTADO_SIGUIENTE_O1 <= OCTAVA1;
                        next_count1_v := count1;
                  end if;
            when MAS_OCTAVA1 =>
                  if count1 = 2 then
                        ESTADO_SIGUIENTE_O1 <= MAS_OCTAVA1;
                        next_count1_v := 0;
                  elsif fin_fila_B_o1 = '1' then
                        ESTADO_SIGUIENTE_O1 <= MAS_OCTAVA1;
                        next_count1_v := count1;
                  else
                        ESTADO_SIGUIENTE_O1 <= OTRA_OCTAVA1;
                        next_count1_v := count1;
                  end if;
            when OTRA_OCTAVA1 =>
                  if fin_fila_B_o1 = '1' then
                        if fin_gene_o1 = '1' then
                              ESTADO_SIGUIENTE_O1 <= FIN_OCTAVA1;
                        else
                              ESTADO_SIGUIENTE_O1 <= MAS_OCTAVA1;
                        end if;
                        next_count1_v := count1+1;
                  else
                        ESTADO_SIGUIENTE_O1 <= OTRA_OCTAVA1;
                        next_count1_v := count1;
                  end if;
```

```
            when FIN_OCTAVA1 =>
                  ESTADO_SIGUIENTE_O1 <= ESPERA_OCTAVA1;
                  next_count1_v := count1;
      end case;
next_count1 <= next_count1_v;
end process logica_octava1;

logica_octava2: process(ESTADO_ACTUAL_O2, count1, fin_gene_o1,
fin_gene_o2, fin_fila_B_o2, count2)
variable next_count2_v : integer range 0 to 3;
begin
      case ESTADO_ACTUAL_O2 is
            when ESPERA_OCTAVA2 =>
                  if count1 = 6 then
                        ESTADO_SIGUIENTE_O2 <= INIT_OCTAVA2;
                  else
                        ESTADO_SIGUIENTE_O2 <= ESPERA_OCTAVA2;
                  end if;
                  next_count2_v := 0;
            when INIT_OCTAVA2 =>
                  if count2 = 2 then
                        ESTADO_SIGUIENTE_O2 <= STOP_OCTAVA2;
                        next_count2_v := 0;
                  elsif fin_fila_B_o2 = '1' then
                        ESTADO_SIGUIENTE_O2 <= INIT_OCTAVA2;
                        next_count2_v := count2;
                  else
                        ESTADO_SIGUIENTE_O2 <= OCTAVA2;
                        next_count2_v := count2;
                  end if;
            when OCTAVA2 =>
                  if fin_fila_B_o2 = '1' then
                        ESTADO_SIGUIENTE_O2 <= INIT_OCTAVA2;
                        next_count2_v := count2+1;
                  else
                        ESTADO_SIGUIENTE_O2 <= OCTAVA2;
                        next_count2_v := count2;
                  end if;
            when STOP_OCTAVA2 =>
                  if count1 = 2 then
                        ESTADO_SIGUIENTE_O2 <= POSTSTOP_OCTAVA2;
                  else
                        ESTADO_SIGUIENTE_O2 <= STOP_OCTAVA2;
                  end if;
                  next_count2_v := 0;
            when POSTSTOP_OCTAVA2 =>
                  ESTADO_SIGUIENTE_O2 <= MAS_OCTAVA2;
                  next_count2_v := 0;
            when MAS_OCTAVA2 =>
                  if fin_fila_B_o2 = '1' then
                        ESTADO_SIGUIENTE_O2 <= MAS_OCTAVA2;
                  else
                        ESTADO_SIGUIENTE_O2 <= OTRA_OCTAVA2;
                  end if;
                  next_count2_v := 0;
            when OTRA_OCTAVA2 =>
```

```
                  if fin_fila_B_o2 = '1' then
                        if fin_gene_o2 = '1' then
                              ESTADO_SIGUIENTE_O2 <= FIN_OCTAVA2;
                        elsif fin_gene_o1 = '1' then
                              ESTADO_SIGUIENTE_O2 <= MAS_OCTAVA2;
                        else
                              ESTADO_SIGUIENTE_O2 <= STOP_OCTAVA2;
                        end if;
                  else
                        ESTADO_SIGUIENTE_O2 <= OTRA_OCTAVA2;
                  end if;
                  next_count2_v := 0;
            when FIN_OCTAVA2 =>
                  ESTADO_SIGUIENTE_O2 <= ESPERA_OCTAVA2;
                  next_count2_v := 0;
      end case;
next_count2 <= next_count2_v;
end process logica_octava2;

logica_salida_o1: process(ESTADO_ACTUAL_O1)
begin
      case ESTADO_ACTUAL_O1 is
            when ESPERA_OCTAVA1 =>
                  inicio_octava_1 <= '1';
                  stop_octava_1 <= '1';
            when INIT_OCTAVA1 =>
                  inicio_octava_1 <= '0';
                  stop_octava_1 <= '1';
            when OCTAVA1 =>
                  inicio_octava_1 <= '1';
                  stop_octava_1 <= '1';
            when MAS_OCTAVA1 =>
                  inicio_octava_1 <= '1';
                  stop_octava_1 <= '1';
            when OTRA_OCTAVA1 =>
                  inicio_octava_1 <= '1';
                  stop_octava_1 <= '1';
            when FIN_OCTAVA1 =>
                  inicio_octava_1 <= '1';
                  stop_octava_1 <= '1';
      end case;
end process logica_salida_o1;

logica_salida_o2: process(ESTADO_ACTUAL_O2)
begin
      case ESTADO_ACTUAL_O2 is
            when ESPERA_OCTAVA2 =>
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '1';
            when INIT_OCTAVA2 =>
                  inicio_octava_2 <= '0';
                  stop_octava_2 <= '1';
            when OCTAVA2 =>
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '1';
            when STOP_OCTAVA2 =>
```

```
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '0';
            when POSTSTOP_OCTAVA2 =>
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '0';
            when MAS_OCTAVA2 =>
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '1';
            when OTRA_OCTAVA2 =>
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '1';
            when FIN_OCTAVA2 =>
                  inicio_octava_2 <= '1';
                  stop_octava_2 <= '1';
      end case;
end process logica_salida_o2;

end rtl;
```

Entidad : Mux2to1_G21

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Mux2to1_G21 is
      port( data_in1_A : in std_logic_vector(27 downto 0);
            data_in1_B : in std_logic_vector(27 downto 0);
            selec : in std_logic;
            data_out : out std_logic_vector(27 downto 0));
end Mux2to1_G21;

architecture rtl of Mux2to1_G21 is

begin

      with selec select
            data_out <=
                  data_in1_A when '0',
                  data_in1_B when others;

end rtl;
```

Entidad : Octava_G51

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Octava_G51 is
      port( clock : in std_logic;
            reset : in std_logic;
```

```
            inicio : in std_logic;
            fin_gene: in std_logic;
            fin_data_A: in std_logic;
            fin_data_B: in std_logic;
            fin_fila_A: in std_logic;
            fin_fila_B: in std_logic;
            stop : in std_logic;
            inic_gene: out std_logic;
            enable_rd : out std_logic;
            reg_data : out std_logic;
            pixels : in std_logic_vector(27 downto 0);
            pixels_paa : out std_logic_vector(13 downto 0);
            pixels_pab : out std_logic_vector(13 downto 0);
            pixels_pba : out std_logic_vector(13 downto 0);
            pixels_pbb : out std_logic_vector(13 downto 0));
end Octava_G51;

architecture rtl of Octava_G51 is

component Data_Path_G11
      port( clock : in std_logic;
            inicio : in std_logic;
            pixel : in std_logic_vector(27 downto 0);
            pbb, pba, pab, paa : out std_logic_vector(13 downto 0));
end component;

component Control_Path_G31
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene : in std_logic;
            fin_fila_A : in std_logic;
            fin_fila_B : in std_logic;
            fin_dato_B : in std_logic;
            stop : in std_logic;
            inicio_data_A : out std_logic;
            inicio_data_B : out std_logic;
            inicio_gene : out std_logic;
            enable_mem : out std_logic);
end component;

component Registrar_G01
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : in std_logic;
            fin_data_A : in std_logic;
            fin_data_B : in std_logic;
            write_reg : out std_logic;
            selec : out std_logic);
end component;

component Registro_G01
      port( clock : in std_logic;
            reset : in std_logic;
```

```
            enable : in std_logic;
            data_in : in std_logic_vector(13 downto 0);
            data_out : out std_logic_vector(13 downto 0));
end component;

component Mux2to1_G01
      port( data_in1_A : in std_logic_vector(13 downto 0);
            data_in1_B : in std_logic_vector(13 downto 0);
            selec : in std_logic;
            data_out : out std_logic_vector(13 downto 0));
end component;

signal inic_d_A, inic_d_B : std_logic;
signal enable_rg, selec : std_logic;

signal pbb_int_A, pba_int_A, pab_int_A, paa_int_A :
std_logic_vector(13 downto 0);
signal pbb_int_B, pba_int_B, pab_int_B, paa_int_B :
std_logic_vector(13 downto 0);
signal pbb_int, pba_int, pab_int, paa_int : std_logic_vector(13 downto
0);


begin

U1 : Data_Path_G11
      port map(
            clock => clock,
            inicio => inic_d_A,
            pixel => pixels,
            pbb => pbb_int_A,
            pba => pba_int_A,
            pab => pab_int_A,
            paa => paa_int_A);

U2 : Data_Path_G11
      port map(
            clock => clock,
            inicio => inic_d_B,
            pixel => pixels,
            pbb => pbb_int_B,
            pba => pba_int_B,
            pab => pab_int_B,
            paa => paa_int_B);

U3 : Control_Path_G31
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio,
            fin_gene => fin_gene,
            fin_fila_A => fin_fila_A,
            fin_fila_B => fin_fila_B,
            fin_dato_B => fin_data_B,
            stop => stop,
            inicio_data_A => inic_d_A,
```

```
            inicio_data_B => inic_d_B,
            inicio_gene => inic_gene,
            enable_mem => enable_rd);

U4 : Registrar_G01
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio,
            pausa => stop,
            fin_gene => fin_gene,
            fin_data_A => fin_data_A,
            fin_data_B => fin_data_B,
            write_reg => enable_rg,
            selec => selec);

U5 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg,
            data_in => pbb_int,
            data_out => pixels_pbb);

U6 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg,
            data_in => pba_int,
            data_out => pixels_pba);

U7 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg,
            data_in => pab_int,
            data_out => pixels_pab);

U8 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg,
            data_in => paa_int,
            data_out => pixels_paa);


U9 : Mux2to1_G01
      port map(
            data_in1_A => pbb_int_A,
            data_in1_B => pbb_int_B,
            selec => selec,
            data_out => pbb_int);
U10 : Mux2to1_G01
```

```
      port map(
            data_in1_A => pba_int_A,
            data_in1_B => pba_int_B,
            selec => selec,
            data_out => pba_int);
U11 : Mux2to1_G01
      port map(
            data_in1_A => pab_int_A,
            data_in1_B => pab_int_B,
            selec => selec,
            data_out => pab_int);
U12 : Mux2to1_G01
      port map(
            data_in1_A => paa_int_A,
            data_in1_B => paa_int_B,
            selec => selec,
            data_out => paa_int);

reg_data <= enable_rg;

end rtl;
```

Entidad : Octava_G61

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Octava_G61 is
      port( clock : in std_logic;
            reset : in std_logic;
            inicio_o2 : in std_logic;
            fin_gene_o2: in std_logic;
            fin_data_A_o2: in std_logic;
            fin_data_B_o2: in std_logic;
            fin_fila_A_o2: in std_logic;
            fin_fila_B_o2: in std_logic;
            stop_o2 : in std_logic;
            inic_gene_o2: out std_logic;
            enable_rd_o2 : out std_logic;
            reg_data_o2 : out std_logic;
            inicio_o3 : in std_logic;
            fin_gene_o3: in std_logic;
            fin_data_A_o3: in std_logic;
            fin_data_B_o3: in std_logic;
            fin_fila_A_o3: in std_logic;
            fin_fila_B_o3: in std_logic;
            stop_o3 : in std_logic;
            inic_gene_o3: out std_logic;
            enable_rd_o3 : out std_logic;
            reg_data_o3 : out std_logic;
            selec_oct : out std_logic;
            pixels : in std_logic_vector(27 downto 0);
            pixels_paa_o2 : out std_logic_vector(13 downto 0);
```

```
            pixels_pab_o2 : out std_logic_vector(13 downto 0);
            pixels_pba_o2 : out std_logic_vector(13 downto 0);
            pixels_pbb_o2 : out std_logic_vector(13 downto 0);
            pixels_paa_o3 : out std_logic_vector(13 downto 0);
            pixels_pab_o3 : out std_logic_vector(13 downto 0);
            pixels_pba_o3 : out std_logic_vector(13 downto 0);
            pixels_pbb_o3 : out std_logic_vector(13 downto 0));
end Octava_G61;

architecture rtl of Octava_G61 is

component Data_Path_G11
      port( clock : in std_logic;
            inicio : in std_logic;
            pixel : in std_logic_vector(27 downto 0);
            pbb, pba, pab, paa : out std_logic_vector(13 downto 0));
end component;

component Control_Path_G31
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene : in std_logic;
            fin_fila_A : in std_logic;
            fin_fila_B : in std_logic;
            fin_dato_B : in std_logic;
            stop : in std_logic;
            inicio_data_A : out std_logic;
            inicio_data_B : out std_logic;
            inicio_gene : out std_logic;
            enable_mem : out std_logic);
end component;

component Registrar_G01
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : in std_logic;
            fin_data_A : in std_logic;
            fin_data_B : in std_logic;
            write_reg : out std_logic;
            selec : out std_logic);
end component;

component Registro_G01
      port( clock : in std_logic;
            reset : in std_logic;
            enable : in std_logic;
            data_in : in std_logic_vector(13 downto 0);
            data_out : out std_logic_vector(13 downto 0));
end component;

component Mux2to1_G01
      port( data_in1_A : in std_logic_vector(13 downto 0);
            data_in1_B : in std_logic_vector(13 downto 0);
```

```
            selec : in std_logic;
            data_out : out std_logic_vector(13 downto 0));
end component;

signal inic_d_A, inic_d_B : std_logic;
signal pbb_int_A, pba_int_A, pab_int_A, paa_int_A :
std_logic_vector(13 downto 0);
signal pbb_int_B, pba_int_B, pab_int_B, paa_int_B :
std_logic_vector(13 downto 0);

signal inic_d_A_o2, inic_d_B_o2 : std_logic;
signal enable_rg_o2, selec_o2 : std_logic;
signal pbb_int_o2, pba_int_o2, pab_int_o2, paa_int_o2 :
std_logic_vector(13 downto 0);

signal inic_d_A_o3, inic_d_B_o3 : std_logic;
signal enable_rg_o3, selec_o3 : std_logic;
signal pbb_int_o3, pba_int_o3, pab_int_o3, paa_int_o3 :
std_logic_vector(13 downto 0);

begin

U1 : Data_Path_G11
      port map(
            clock => clock,
            inicio => inic_d_A,
            pixel => pixels,
            pbb => pbb_int_A,
            pba => pba_int_A,
            pab => pab_int_A,
            paa => paa_int_A);

inic_d_A <= inic_d_A_o2 and inic_d_A_o3;

U2 : Data_Path_G11
      port map(
            clock => clock,
            inicio => inic_d_B,
            pixel => pixels,
            pbb => pbb_int_B,
            pba => pba_int_B,
            pab => pab_int_B,
            paa => paa_int_B);

inic_d_B <= inic_d_B_o2 and inic_d_B_o3;

selec_oct <= inic_d_A_o2 and inic_d_B_o2;

U3 : Control_Path_G31
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o2,
            fin_gene => fin_gene_o2,
            fin_fila_A => fin_fila_A_o2,
            fin_fila_B => fin_fila_B_o2,
```

```
            fin_dato_B => fin_data_B_o2,
            stop => stop_o2,
            inicio_data_A => inic_d_A_o2,
            inicio_data_B => inic_d_B_o2,
            inicio_gene => inic_gene_o2,
            enable_mem => enable_rd_o2);

U4 : Registrar_G01
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o2,
            pausa => stop_o2,
            fin_gene => fin_gene_o2,
            fin_data_A => fin_data_A_o2,
            fin_data_B => fin_data_B_o2,
            write_reg => enable_rg_o2,
            selec => selec_o2);

U5 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o2,
            data_in => pbb_int_o2,
            data_out => pixels_pbb_o2);

U6 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o2,
            data_in => pba_int_o2,
            data_out => pixels_pba_o2);

U7 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o2,
            data_in => pab_int_o2,
            data_out => pixels_pab_o2);

U8 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o2,
            data_in => paa_int_o2,
            data_out => pixels_paa_o2);

U9 : Mux2to1_G01
      port map(
            data_in1_A => pbb_int_A,
            data_in1_B => pbb_int_B,
            selec => selec_o2,
```

```
            data_out => pbb_int_o2);

U10 : Mux2to1_G01
      port map(
            data_in1_A => pba_int_A,
            data_in1_B => pba_int_B,
            selec => selec_o2,
            data_out => pba_int_o2);

U11 : Mux2to1_G01
      port map(
            data_in1_A => pab_int_A,
            data_in1_B => pab_int_B,
            selec => selec_o2,
            data_out => pab_int_o2);

U12 : Mux2to1_G01
      port map(
            data_in1_A => paa_int_A,
            data_in1_B => paa_int_B,
            selec => selec_o2,
            data_out => paa_int_o2);

reg_data_o2 <= enable_rg_o2;

-- Tercera Octava

U13 : Control_Path_G31
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o3,
            fin_gene => fin_gene_o3,
            fin_fila_A => fin_fila_A_o3,
            fin_fila_B => fin_fila_B_o3,
            fin_dato_B => fin_data_B_o3,
            stop => stop_o3,
            inicio_data_A => inic_d_A_o3,
            inicio_data_B => inic_d_B_o3,
            inicio_gene => inic_gene_o3,
            enable_mem => enable_rd_o3);

U14 : Registrar_G01
      port map(
            clock => clock,
            reset => reset,
            inicio => inicio_o3,
            pausa => stop_o3,
            fin_gene => fin_gene_o3,
            fin_data_A => fin_data_A_o3,
            fin_data_B => fin_data_B_o3,
            write_reg => enable_rg_o3,
            selec => selec_o3);

U15 : Registro_G01
      port map(
```

```
            clock => clock,
            reset => reset,
            enable => enable_rg_o3,
            data_in => pbb_int_o3,
            data_out => pixels_pbb_o3);

U16 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o3,
            data_in => pba_int_o3,
            data_out => pixels_pba_o3);

U17 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o3,
            data_in => pab_int_o3,
            data_out => pixels_pab_o3);

U18 : Registro_G01
      port map(
            clock => clock,
            reset => reset,
            enable => enable_rg_o3,
            data_in => paa_int_o3,
            data_out => pixels_paa_o3);

U19 : Mux2to1_G01
      port map(
            data_in1_A => pbb_int_A,
            data_in1_B => pbb_int_B,
            selec => selec_o3,
            data_out => pbb_int_o3);

U20 : Mux2to1_G01
      port map(
            data_in1_A => pba_int_A,
            data_in1_B => pba_int_B,
            selec => selec_o3,
            data_out => pba_int_o3);

U21 : Mux2to1_G01
      port map(
            data_in1_A => pab_int_A,
            data_in1_B => pab_int_B,
            selec => selec_o3,
            data_out => pab_int_o3);

U22 : Mux2to1_G01
      port map(
            data_in1_A => paa_int_A,
            data_in1_B => paa_int_B,
            selec => selec_o3,
```

```
            data_out => paa_int_o3);

reg_data_o3 <= enable_rg_o3;


end rtl;
```

Entidad : Data_Path_G11

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.biorto.all;

entity Data_Path_G11 is
      port( clock : in std_logic;
            inicio : in std_logic;
            pixel : in std_logic_vector(27 downto 0);
            pbb, pba, pab, paa: out std_logic_vector(13 downto 0));
end Data_Path_G11;

architecture rtl of Data_Path_G11 is

constant bb : MATRIX_BIORTO3:= MATRIZ_BB_BI3;
constant ba : MATRIX_BIORTO3:= MATRIZ_BA_BI3;
constant ab : MATRIX_BIORTO3:= MATRIZ_AB_BI3;
constant aa : MATRIX_BIORTO3:= MATRIZ_AA_BI3;

begin

process (clock)

variable pbb1,pba1,pab1,paa1 : std_logic_vector(21 downto 0);
variable pbb2,pba2,pab2,paa2 : std_logic_vector(21 downto 0);
variable pbb3,pba3,pab3,paa3 : std_logic_vector(21 downto 0);
variable i : integer range 1 to 12;

      begin
      if clock'event and clock='1' then
            if inicio = '0' then
                  if i = 1 then
                        pbb1 := (4=>'1', others => '0');
                        pba1 := (4=>'1', others => '0');
                        pab1 := (4=>'1', others => '0');
                        paa1 := (4=>'1', others => '0');
                  end if;

                  pbb2 := pixel(13 downto 0)*bb(i);
                  pbb3 := pixel(27 downto 14)*bb(i+4);
                  pbb1 := pbb3+pbb2+pbb1;
                  pba2 := pixel(13 downto 0)*ba(i);
                  pba3 := pixel(27 downto 14)*ba(i+4);
                  pba1 := pba3+pba2+pba1;
                  pab2 := pixel(13 downto 0)*ab(i);
```

```
                pab3 := pixel(27 downto 14)*ab(i+4);
                pab1 := pab3+pab2+pab1;
                paa2 := pixel(13 downto 0)*aa(i);
                paa3 := pixel(27 downto 14)*aa(i+4);
                paa1 := paa3+paa2+paa1;

                if i = 12 then
                      i:=1;
                elsif i=4 then
                      i:=i+1+4;
                else
                      i:=i+1;
                end if;
          else
                pbb1 := (others => '0');
                pba1 := (others => '0');
                pab1 := (others => '0');
                paa1 := (others => '0');
                i:= 1;
          end if;

          pbb <= pbb1(18 downto 5);
          pab <= pab1(18 downto 5);
          pba <= pba1(18 downto 5);
          paa <= paa1(18 downto 5);

      end if;

end process;

end rtl;
```

Entidad : Control_Path_G31

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Control_Path_G31 is
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            fin_gene : in std_logic;
            fin_fila_A : in std_logic;
            fin_fila_B : in std_logic;
            fin_dato_B : in std_logic;
            stop : in std_logic;
            inicio_data_A : out std_logic;
            inicio_data_B : out std_logic;
            inicio_gene : out std_logic;
            enable_mem : out std_logic);
end Control_Path_G31;

architecture rtl of Control_Path_G31 is
```

```
type ESTADO is (ESPERA, INIT_GENE, INIT_MEM, INIT_DATA_A, INIT_DATA_B,
            STOP_DATA_A, PREPAUSA, PAUSA, CALCULA_FIN);
signal ESTADO_ACTUAL, ESTADO_SIGUIENTE : ESTADO;
signal count, next_count : integer range 0 to 4;

begin

secuencial: process(clock, reset)
begin
      if reset = '0' then
            ESTADO_ACTUAL <= ESPERA;
      elsif clock'event and clock = '1' then
            ESTADO_ACTUAL <= ESTADO_SIGUIENTE;
            count <= next_count;
      end if;
end process secuencial;

logica_entrada: process(ESTADO_ACTUAL, inicio, fin_gene, count,
fin_fila_A,
                  fin_fila_B, stop, fin_dato_B)
begin
      case ESTADO_ACTUAL is
            when ESPERA =>
                  if inicio = '0' then
                        ESTADO_SIGUIENTE <= INIT_GENE;
                  else
                        ESTADO_SIGUIENTE <= ESPERA;
                  end if;
            when INIT_GENE =>
                  ESTADO_SIGUIENTE <= INIT_MEM;
            when INIT_MEM =>
                  ESTADO_SIGUIENTE <= INIT_DATA_A;
            when INIT_DATA_A =>
                  if count = 3 then
                        ESTADO_SIGUIENTE <= INIT_DATA_B;
                  else
                        ESTADO_SIGUIENTE <= INIT_DATA_A;
                  end if;
            when INIT_DATA_B =>
                  if fin_fila_A = '0' then
                        ESTADO_SIGUIENTE <= INIT_DATA_B;
                  else
                        ESTADO_SIGUIENTE <= STOP_DATA_A;
                  end if;
            when STOP_DATA_A =>
                  if fin_gene = '1' then
                        ESTADO_SIGUIENTE <= CALCULA_FIN;
                  elsif stop = '0' and fin_dato_B = '0' then
                        ESTADO_SIGUIENTE <= PREPAUSA;
                  elsif fin_fila_B = '0' then
                        ESTADO_SIGUIENTE <= STOP_DATA_A;
                  else
                        ESTADO_SIGUIENTE <= INIT_DATA_A;
                  end if;
            when PREPAUSA =>
```

```
                        ESTADO_SIGUIENTE <= PAUSA;
                  when PAUSA =>
                        if stop = '1' then
                              ESTADO_SIGUIENTE <= INIT_MEM;
                        else
                              ESTADO_SIGUIENTE <= PAUSA;
                        end if;
                  when CALCULA_FIN =>
                        ESTADO_SIGUIENTE <= ESPERA;
            end case;
end process logica_entrada;

logica_salida: process(ESTADO_ACTUAL, count)

variable next_count_v : integer range 0 to 4;

begin
      case ESTADO_ACTUAL is
            when ESPERA =>
                  inicio_data_A <= '1';
                  inicio_data_B <= '1';
                  inicio_gene <= '1';
                  enable_mem <= '1';
                  next_count_v := 0;
            when INIT_GENE =>
                  inicio_data_A <= '1';
                  inicio_data_B <= '1';
                  inicio_gene <= '0';
                  enable_mem <= '1';
                  next_count_v := count;
            when INIT_MEM =>
                  inicio_data_A <= '1';
                  inicio_data_B <= '1';
                  inicio_gene <= '0';
                  enable_mem <= '0';
                  next_count_v := count;
            when INIT_DATA_A =>
                  inicio_data_A <= '0';
                  inicio_data_B <= '1';
                  inicio_gene <= '0';
                  enable_mem <= '0';
                  next_count_v := count+1;
            when INIT_DATA_B =>
                  inicio_data_A <= '0';
                  inicio_data_B <= '0';
                  inicio_gene <= '0';
                  enable_mem <= '0';
                  next_count_v := 0;
            when STOP_DATA_A =>
                  inicio_data_A <= '1';
                  inicio_data_B <= '0';
                  inicio_gene <= '0';
                  enable_mem <= '0';
                  next_count_v := count;
            when PREPAUSA =>
                  inicio_data_A <= '1';
```

```
                  inicio_data_B <= '0';
                  inicio_gene <= '0';
                  enable_mem <= '0';
                  next_count_v := count;
            when PAUSA =>
                  inicio_data_A <= '1';
                  inicio_data_B <= '1';
                  inicio_gene <= '0';
                  enable_mem <= '0';
                  next_count_v := count;
            when CALCULA_FIN =>
                  inicio_data_A <= '1';
                  inicio_data_B <= '0';
                  inicio_gene <= '1';
                  enable_mem <= '0';
                  next_count_v := count;
      end case;

next_count <= next_count_v;

end process logica_salida;

end rtl;
```

Entidad : Registrar_G01

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Registrar_G01 is
      port( clock : in std_logic;
            reset : in std_logic;
            inicio : in std_logic;
            pausa : in std_logic;
            fin_gene : in std_logic;
            fin_data_A : in std_logic;
            fin_data_B : in std_logic;
            write_reg : out std_logic;
            selec : out std_logic);
end Registrar_G01;

architecture rtl of Registrar_G01 is

type ESTADO is (ESPERA, ESPERA_DATA, PRE_REGISTRAR_A, REGISTRAR_A,
            PRE_REGISTRAR_B, REGISTRAR_B, STOP);
signal ESTADO_ACTUAL, ESTADO_SIGUIENTE : ESTADO;

begin

secuencial: process(clock, reset)
begin
      if reset = '0' then
            ESTADO_ACTUAL <= ESPERA;
```

```
      elsif clock'event and clock = '1' then
            ESTADO_ACTUAL <= ESTADO_SIGUIENTE;
      end if;
end process secuencial;

logica_entrada: process(ESTADO_ACTUAL, inicio, fin_gene, fin_data_A,
fin_data_B, pausa)
begin
      case ESTADO_ACTUAL is
            when ESPERA =>
                  if inicio = '0' then
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  else
                        ESTADO_SIGUIENTE <= ESPERA;
                  end if;
            when ESPERA_DATA =>
                  if fin_data_A = '0' then
                        ESTADO_SIGUIENTE <= PRE_REGISTRAR_A;
                  elsif fin_data_B = '0' then
                        ESTADO_SIGUIENTE <= PRE_REGISTRAR_B;
                  else
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  end if;

            when PRE_REGISTRAR_A =>
                  ESTADO_SIGUIENTE <= REGISTRAR_A;
            when REGISTRAR_A =>
                  if fin_gene = '1' then
                        ESTADO_SIGUIENTE <= ESPERA;
                  else
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  end if;

            when PRE_REGISTRAR_B =>
                  ESTADO_SIGUIENTE <= REGISTRAR_B;
            when REGISTRAR_B =>
                  if fin_gene = '1' then
                        ESTADO_SIGUIENTE <= ESPERA;
                  elsif pausa = '0' then
                        ESTADO_SIGUIENTE <= STOP;
                  else
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  end if;
            when STOP =>
                  if pausa = '0' then
                        ESTADO_SIGUIENTE <= STOP;
                  else
                        ESTADO_SIGUIENTE <= ESPERA_DATA;
                  end if;
      end case;
end process logica_entrada;

logica_salida: process(ESTADO_ACTUAL)
begin
      case ESTADO_ACTUAL is
            when ESPERA =>
```

```
                write_reg <= '1';
                selec <= '0';
            when ESPERA_DATA =>
                write_reg <= '1';
                selec <= '0';

            when PRE_REGISTRAR_A =>
                write_reg <= '1';
                selec <= '0';
            when REGISTRAR_A =>
                write_reg <= '0';
                selec <= '0';

            when PRE_REGISTRAR_B =>
                write_reg <= '1';
                selec <= '1';
            when REGISTRAR_B =>
                write_reg <= '0';
                selec <= '1';
            when STOP =>
                write_reg <= '1';
                selec <= '0';
      end case;

end process logica_salida;

end rtl;
```

Entidad : Registro_G01

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Registro_G01 is
      port( clock : in std_logic;
            reset : in std_logic;
            enable : in std_logic;
            data_in : in std_logic_vector(13 downto 0);
            data_out : out std_logic_vector(13 downto 0));
end Registro_G01;

architecture rtl of Registro_G01 is

begin

process(clock, reset)
begin
      if reset = '0' then
            data_out <= (others => '0');
      elsif clock'event and clock = '1' then
            if enable = '0' then
                  data_out <= data_in;
            end if;
```

```
      end if;
end process;

end rtl;
```

Entidad : Mux2to1_G01

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity Mux2to1_G01 is
      port( data_in1_A : in std_logic_vector(13 downto 0);
            data_in1_B : in std_logic_vector(13 downto 0);
            selec : in std_logic;
            data_out : out std_logic_vector(13 downto 0));
end Mux2to1_G01;

architecture rtl of Mux2to1_G01 is

begin

      with selec select
            data_out <=
                  data_in1_A when '0',
                  data_in1_B when others;

end rtl;
```

Package : Biorto

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_arith.all;
use IEEE.Std_Logic_signed.all;

package BIORTO is

type MATRIX_BIORTO3 is array(1 to 16) of std_logic_vector(7 downto 0);

--------------------------------------
function MATRIZ_BB_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
function MATRIZ_BA_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
function MATRIZ_AB_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
```

```
function MATRIZ_AA_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
function MATRIZ_IBB_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
function MATRIZ_IBA_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
function MATRIZ_IAB_BI3 return MATRIX_BIORTO3;
--------------------------------------

--------------------------------------
function MATRIZ_IAA_BI3 return MATRIX_BIORTO3;
--------------------------------------

end BIORTO;

package body BIORTO is

type BIORTO3 is array(1 to 4) of integer;

constant b3cl : BIORTO3:=(-2, 6, 6, -2);
constant b3ch : BIORTO3:=(1, -3, 3, -1);
constant b3icl : BIORTO3:=(1, 3, 3, 1);
constant b3ich : BIORTO3:=(-2, -6, 6, 2);

--------------------------------------
function MATRIZ_BB_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable bb : MATRIX_BIORTO3;
variable bb1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  bb1:=b3cl(J)*b3cl(I);
                  bb(4*I+J-4):=conv_std_logic_vector(bb1,8);
            end loop;
      end loop;
      return bb;
end MATRIZ_BB_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_BA_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable ba : MATRIX_BIORTO3;
variable ba1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  ba1:=b3ch(J)*b3cl(I);
                  ba(4*I+J-4):=conv_std_logic_vector(ba1,8);
```

```
            end loop;
      end loop;
      return ba;
end MATRIZ_BA_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_AB_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable ab : MATRIX_BIORTO3;
variable ab1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  ab1:=b3cl(J)*b3ch(I);
                  ab(4*I+J-4):=conv_std_logic_vector(ab1,8);
            end loop;
      end loop;
      return ab;
end MATRIZ_AB_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_AA_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable aa : MATRIX_BIORTO3;
variable aa1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  aa1:=b3ch(J)*b3ch(I);
                  aa(4*I+J-4):=conv_std_logic_vector(aa1,8);
            end loop;
      end loop;
      return aa;
end MATRIZ_AA_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_IBB_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable ibb : MATRIX_BIORTO3;
variable ibb1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  ibb1:=b3icl(J)*b3icl(I);
                  ibb(4*I+J-4):=conv_std_logic_vector(ibb1,8);
            end loop;
      end loop;
      return ibb;
end MATRIZ_IBB_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_IBA_BI3 return MATRIX_BIORTO3 is
```

```
--------------------------------------
variable iba : MATRIX_BIORTO3;
variable iba1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  iba1:=b3ich(J)*b3icl(I);
                  iba(4*I+J-4):=conv_std_logic_vector(iba1,8);
            end loop;
      end loop;
      return iba;
end MATRIZ_IBA_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_IAB_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable iab : MATRIX_BIORTO3;
variable iab1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  iab1:=b3icl(J)*b3ich(I);
                  iab(4*I+J-4):=conv_std_logic_vector(iab1,8);
            end loop;
      end loop;
      return iab;
end MATRIZ_IAB_BI3;
--------------------------------------

--------------------------------------
function MATRIZ_IAA_BI3 return MATRIX_BIORTO3 is
--------------------------------------
variable iaa : MATRIX_BIORTO3;
variable iaa1: integer;
begin
      for I in 1 to 4 loop
            for J in 1 to 4 loop
                  iaa1:=b3ich(J)*b3ich(I);
                  iaa(4*I+J-4):=conv_std_logic_vector(iaa1,8);
            end loop;
      end loop;
      return iaa;
end MATRIZ_IAA_BI3;
--------------------------------------

end BIORTO;
```