ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Diseño de un Algoritmo de Reducción de Imágenes Astronómicas Basado en Wavelets para el Instrumento FastCam e Implementación sobre un Supercomputador Reconfigurable

AUTOR: Jessica María Nicolás Serrano
DIRECTOR(ES): Francisco Javier Garrigós Guerrero
José Javier Martínez Álvarez

Julio **/** 2010

Autor	Jessica María Nicolás Serrano
E-mail del Autor	jessica_n_s@hotmail.com
Director (es)	Francisco Javier Garrigós Guerrero José Javier Martínez Álvarez
E-mail del Director (es)	javier.garrigos@upct.es jjavier.martinez@upct.es
Título del PFC	*Diseño de un Algoritmo de Reducción de Imágenes Astronómicas Basado en Wavelets para el Instrumento Fastcam e Implementación sobre un Supercomputador Reconfigurable.*
Descriptores	FASTCAM, FPGA, Astronomía, procesado en tiempo real, sistemas coprocesador, ImpulseC, DRC, Hardware reconfigurable, Software

Resumen

FASTCAM, desarrollado por el Instituto de Astrofísica de Canarias en colaboración con la Universidad Politécnica de Cartagena, es un instrumento capaz de obtener imágenes de alta resolución espacial en el espectro visible usando telescopios terrestres. Este instrumento ha obtenido muy buenos resultados hasta la fecha, pero tiene algunas limitaciones que admiten alguna mejora.

En base a esto, se propone un algoritmo de detección de objetos astronómicos que introduce etapas de pre-procesado y post-procesado basado en *wavelets*, que mejora considerablemente la calidad de la imagen comparada con el algoritmo inicial.

En concreto, en este PFC se implementa una arquitectura híbrida hardware/software que calcula en tiempo real la etapa computacionalmente más crítica del algoritmo. Para ello se hace uso de dos herramientas de reciente adquisición por este departamento: un súper-computador reconfigurable *DRC RPU 110-L200* y el entorno de co-diseño CoDeveloper, basado en el modelo de programación de ImpulseC.

Titulación	Ingeniero de Telecomunicación
Departamento	Electrónica, tecnología de computadores y proyectos
Fecha de Presentación	Julio - 2010

Agradecimientos

En primer lugar quiero dar las gracias a Javier Garrigós, director de mi proyecto, por su interés, dedicación y disponibilidad cada vez que lo he necesitado, además de la oportunidad de realizar este proyecto, aportándome facilidades y cercanía que han hecho que estos meses en el laboratorio fueran más agradables.

Igualmente quiero agradecer a José Javier, también director de mi proyecto, sus ideas y aportaciones que han servido de gran ayuda en la realización de este proyecto.

También me gustaría agradecer a Carolina y José Manuel, compañeros de laboratorio, los momentos de evasión, que tanto hacen falta en las largas horas de simulación, creando en todo momento un ambiente agradable.

Quiero dar las gracias a mi familia, especialmente a mis padres y abuelos, por la comprensión y paciencia especialmente en esos periodos de examen en los que era difícilmente soportable.

Por último tengo que agradecerte a ti, Iván tu apoyo a lo largo de estos estos años, y las risas que sólo tú consigues sacarme cada día, haciendo la vida más fácil y divertida.

Tabla de contenido

Capítulo 1

Introducción y Objetivos

1.1. Introducción

En la exploración astronómica del universo la herramienta fundamental es el telescopio y su instrumentación auxiliar. En el espacio, un telescopio puede producir imágenes de alta resolución limitadas básicamente por su diámetro y por la longitud de onda de la luz que incide sobre él. En la tierra sin embargo, la densidad y fluctuación de la atmósfera causan perturbaciones sobre la luz que disminuyen la calidad de las imágenes, volviéndose las mismas más borrosas. Las rápidas fluctuaciones de la atmósfera, del orden de las décimas de milisegundo, producen rápidas variaciones en los patrones de difracción de la luz que ocasionan el desenfoque de las imágenes. Con las técnicas de observación tradicionales este hecho representa un gran problema, debido a que estas compensan lo débil de la luz incidente procedente de los astros celestes con un largo tiempo de exposición.

El uso de cámaras de alta velocidad, como las que utiliza el sistema FastCam [**1**], permite elegir aquellas imágenes que estén menos afectadas por la distorsión (*"Lucky Imaging"*) y combinarlas para formar imágenes de mucha mayor resolución. Con esta técnica, y bajo condiciones atmosféricas normales, un telescopio terrestre de 2.5 metros podría alcanzar la resolución conseguida por el telescopio espacial Hubble1 (2.5 m) de manera muy económica.

Sin embargo, esta técnica requiere de un considerable esfuerzo en el desarrollo de algoritmos para la adecuada combinación de las mejores imágenes (reducción) en una única mejorada. En este contexto, el desarrollo de procesadores de propósito específico, especialmente adaptados a la naturaleza del problema es una técnica comúnmente utilizada para acelerar algoritmos de cómputo con respecto a su ejecución sobre computadoras comerciales. Este es el caso, por ejemplo, de las soluciones hardware para la aceleración de algoritmos de procesamiento y filtrado de imágenes en tiempo real.

El primer prototipo del instrumento FastCam utilizó una versión software del algoritmo de reducción. Una versión mejorada y adaptada para posibilitar su ejecución en tiempo real, utilizando aceleración por hardware mediante dispositivos programables reconfigurables, fue desarrollada en un Proyecto Fin de Carrera [**2**] dirigido por este Grupo (Diseño Electrónico y Técnicas de Tratamiento de Señales) en 2007, en colaboración con el Instituto de Astrofísica de Canarias, y es la que actualmente se encuentra funcionando exitosamente en el Telescopio Carlos Sánchez instalado en el observatorio del Teide en Tenerife.

Lo que se pretende ahora con este PFC, es dar una solución, también acelerada por hardware, para las nuevas versiones de algoritmos, que son cada vez son más complejos.

1.2. Objetivos

Dentro de este marco, el objetivo del presente proyecto es el desarrollo de una arquitectura de co-ejecución hardware-software para la aceleración de un algoritmo de reducción de imágenes estelares basado en *wavelets* apropiado para el instrumento FastCam.

La implementación del algoritmo se realizará utilizando un nuevo tipo de herramientas de diseño a nivel de sistema (ESL, *Electronic System Level*). Estas herramientas utilizan usualmente diferentes variaciones de lenguajes de alto nivel (C, C++, Matlab, etc.) para definir un sistema completo compuesto de software ejecutándose sobre procesadores tradicionales y de hardware específico. Una de las herramientas más avanzadas de este tipo, recientemente licenciada por el Departamento de Electrónica, Tecnología de Computadoras y Proyectos, es CoDeveloper de *Impulse Acelerated Technologies*. Para la ejecución del algoritmo, se dispone también de un nodo de supercomputación híbrido (HPRC, *High Performance Reconfigurable Computer*) modelo DS1002, de la empresa DRC, que permite la co-ejecución software-hardware de un algoritmo sobre procesadores convencionales acelerados mediante coprocesadores hardware desarrollados sobre FPGA.

Así, se buscará comprender y saber aplicar las nuevas características que proporcionan las más avanzadas Herramientas de Diseño Electrónico (EDA) a nivel ESL: paralelización del código, mecanismos de comunicación entre *threads*, pragmas, etc., que permitan definir los estilos de codificación en lenguaje C que generen la implementación hardware más eficiente en términos de rendimiento, área y consumo. También se requiere conocer la arquitectura del HPCR DS1002 de DRC que se utilizará para la ejecución del algoritmo. Se establecerá un flujo de trabajo adecuado con él y una serie de estrategias y técnicas de codificación dependientes de su arquitectura que permitan maximizar el rendimiento de los algoritmos ejecutados.

Una vez diseñado e implementado el algoritmo, se procederá a medir el rendimiento de la solución obtenida mediante co-ejecución software/hardware a partir de la síntesis de descripciones de alto nivel, mediante la herramienta CoDeveloper, frente a soluciones software previamente desarrolladas y trabajos hardware previos a este PFC.

Capítulo 2

Conceptos y Trabajos Previos

Este capítulo está dirigido a introducir conocimientos acerca del estado actual de la investigación en las materias que abarca este PFC. En primer lugar, se expondrá la arquitectura del instrumento FASTCAM, así como la técnica de observación utilizada. A continuación introduciremos la teoría de la transformada *wavelet à trous* necesaria para comprender la utilización de esta transformada en el contexto del procesado de imágenes estelares. Para concluir esta visión actual, se mostrarán también los últimos estudios llevados a cabo por el departamento en la materia, de cuyas conclusiones partimos en esta ocasión, con el fin de desarrollar aspectos que aportaron nuevas vías de mejora al instrumento FASTCAM.

2.1. FastCam

Como ya se ha comentado, FastCam es un instrumento basado en la técnica Lucky-Imaging desarrollado por el IAC (Instituto de Astrofísica de Canarias) en colaboración con la UPCT (Universidad Politécnica de Cartagena) [**1**][**3**] con el objetivo de obtener imágenes de muy alta resolución espacial en el espectro visible usando telescopios terrestres.

Fastcam incluye varias ópticas que se pueden intercambiar para muestrear el límite de difracción en banda I (850 nm) con telescopios de 1.5 a 4.2m de diámetro. El límite de difracción de un telescopio mide la mínima separación angular entre dos fuentes de luz que estén siendo observadas por el mismo, dependiendo éste tanto de la longitud de onda de la luz incidente como del diámetro del espejo primario del telescopio. El ángulo con el que se cuantifica este límite viene dado por la expresión (1), donde λ es la longitud de onda de la luz incidente, *D* es el diámetro del espejo primario del telescopio y *θ* el ángulo de salida:

$$\theta(rad) = \frac{1{,}22 \cdot \lambda(cm)}{D(cm)} \tag{1}$$

A modo de referencia, se suele tener en cuenta que el telescopio espacial *Hubble* (con un espejo de 2.5m y considerando una longitud de onda de 50μm) presentaría un límite teórico de *θ*=0.05 segundos de arco. En la práctica, dicho límite aumenta hasta *θ*=0.1. Este hecho es significativamente importante: el límite de difracción de un telescopio se ve mermado debido a los efectos a los que se someten los rayos de luz incidente. Si esto ocurre para el telescopio *Hubble* que se encuentra orbitando a la Tierra (y que por tanto no se ve sometido al efecto de la atmósfera), qué decir de los telescopios terrestres (que se encuentran bajo la influencia de ésta). La resolución de este problema es la principal aportación de FASTCAM a la observación astronómica, consiguiendo un límite de difracción real muy próximo al teórico en telescopios de hasta cuatro metros consiguiendo gracias a ello posibilitar la observación de objetos estelares nunca antes vistos.

Unas imágenes del dispositivo pueden verse en la Figura 2.1.

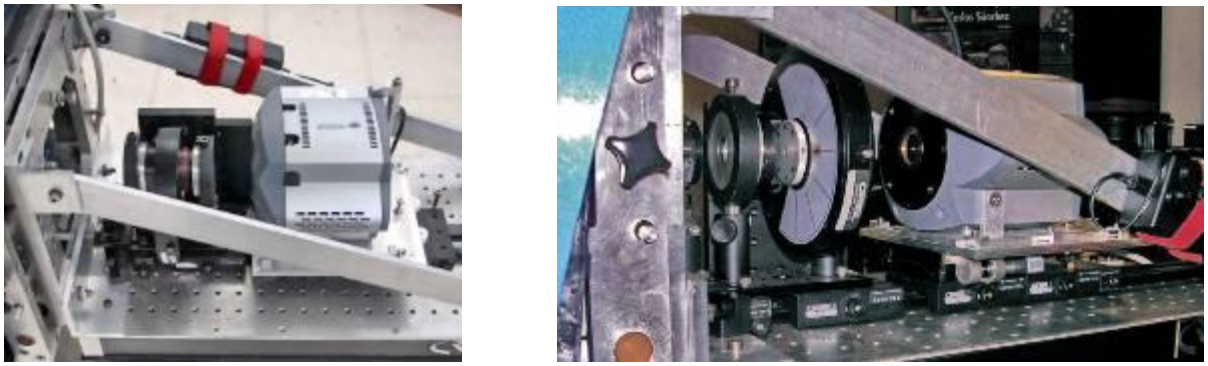

Figura 2.1. FastCam instalado en el foco Cassegrain del Telescopio Carlos Sánchez (1.5m) del Observatorio del Teide(OT)

El instrumento usa una cámara retroiluminada Andor iXon DU-897 [**4**][**5**] que contiene un sensor EMCCD (*"Electron Multiplying Charge Coupled Device"*) de alta sensibilidad y una velocidad de lectura muy rápida. Este tipo de sensores de imagen *EMCCD* [**6**] permiten detectar eventos de un solo fotón manteniendo una alta eficiencia cuántica gracias a una etapa multiplicadora de electrones de estado sólido (denominada *registro EM*, del inglés *electron multiplier*) presente dentro del propio sensor. Este *registro EM* posibilita la amplificación de señales débiles antes de añadir ningún ruido (antes de pasar por el amplificador de salida), contando además de unos cientos de etapas y siendo la magnitud de su amplificación controlable vía software. Por consiguiente, gracias a esta etapa de amplificación dentro del propio detector, estos detectores no están limitados por el ruido del amplificador de salida como ocurre con un *CCD* convencional; permitiendo gracias a ello alcanzar grandes velocidades y posibilitando por tanto este tipo de observaciones que con detectores convencionales serían inviables (al requerir de tiempos de exposición mucho más elevados).

Como se aprecia en la Figura 2.2, el chip posee dos áreas, una para el sensor propiamente dicho que captura la imagen, y un área de almacenamiento, del mismo tamaño, donde la imagen se guarda previa a la lectura. Durante una adquisición, el área del sensor se expone a la luz, y se captura la imagen, después de lo cual, la imagen es automáticamente desplazada al área de almacenamiento donde es leída. Mientras la imagen es leída, el área de captura adquiere la siguiente imagen. Para leer el sensor, la carga es desplazada a través de un registro de lectura en un registro de multiplicación, donde como ya hemos comentado, se realiza la amplificación antes de ser leída por el amplificador de carga.

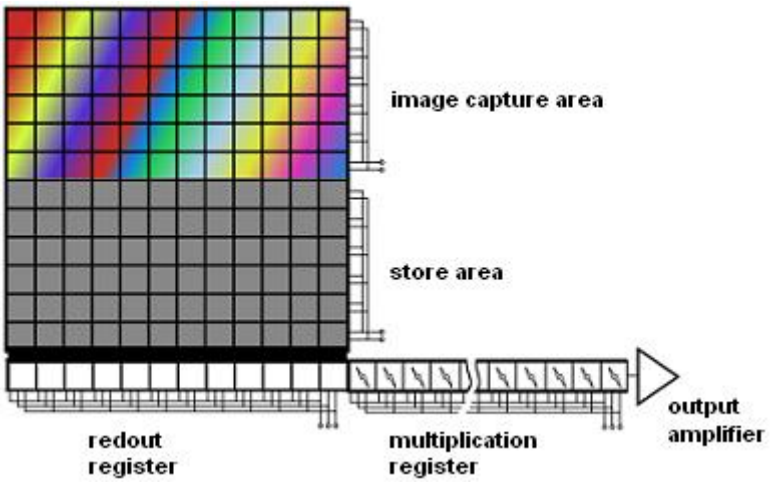

Figura 2.2 Detector EMCCD (incluyendo el registro EM).

En la Figura 2.3 se muestra el esquema físico de Fastcam, que incorpora un sistema de transmisión y evaluación rápida de imágenes utilizando FPGAs y un software propio desarrollado para un procesado eficiente de decenas de miles de imágenes. Así, las imágenes crudas provenientes directamente de la cámara se transmiten en primer lugar a la FPGA, donde se procesan y envían a un monitor VGA que muestra las imágenes crudas, procesadas y un histograma de los factores de calidad calculados, todo en tiempo real. Un ordenador de control conectado a la cámara se encarga de los parámetros de configuración de la misma, como el tiempo de exposición, tasa de imágenes, ganancia, etc. A la FPGA se conectan también dos ordenadores mediante una interfaz Gigabit Ethernet, usando UDP sobre protocolo IP. Uno de éstos controla tanto los parámetros del algoritmo Fastcam como los parámetros correspondientes a las comunicaciones Gigabit Ethernet. Por último, el otro ordenador se encarga de almacenar las imágenes crudas y procesadas que recibe y realizar un post-procesado.

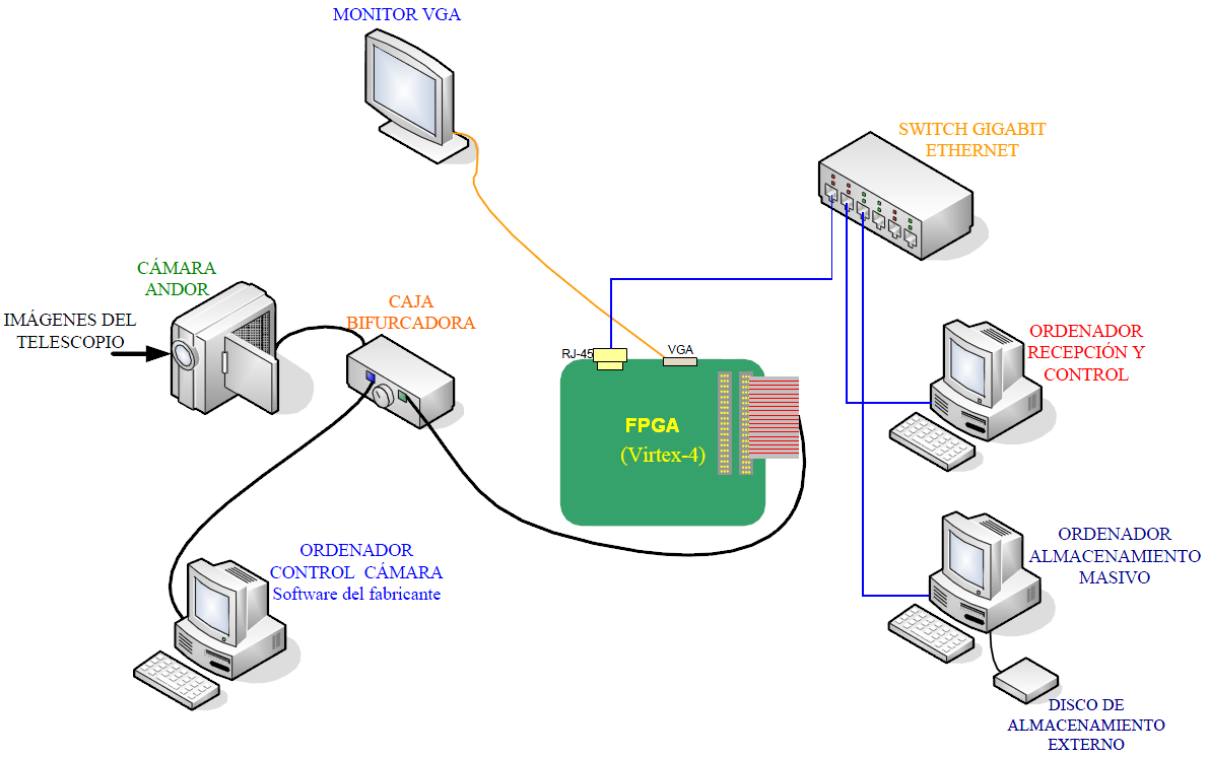

Figura 2.3. Esquema físico del sistema FASTCAM[2].

El algoritmo de reducción de imágenes de corta exposición del instrumento FASTCAM ha sido diseñado en base a que, si se considera un intervalo de tiempo suficientemente corto, se consigue registrar instantes de estabilidad atmosférica, que normalmente duran sólo algunas milésimas de segundo. Conforme a esta consideración, si las imágenes astronómicas son adquiridas con un corto tiempo de exposición (típicamente de 10 ms) usando una cámara de bajo ruido suficientemente rápida, algunas (*lucky images*), de entre miles de imágenes (típicamente un 5%) ofrecerán mucha menor distorsión que las otras. Si sólo se tienen en cuenta estas imágenes y son combinadas, la imagen resultante puede casi alcanzar la máxima resolución del telescopio, de forma que la imagen resultante podrá ofrecer una calidad del mismo orden que si hubiera sido adquirida en ausencia de atmósfera [**7**]. Esta técnica, conocida como *Lucky-Imaging*, fue propuesta originalmente por Fried en 1996 [**8**][**9**][**10**].

El algoritmo FASTCAM actual [**2**], que utiliza la técnica *Lucky-Imaging*, es la pieza clave que hace al instrumento obtener tan buenos resultados; siendo además la parte del sistema que presenta un mayor potencial de mejora. El criterio de calidad seleccionado para la determinación de las imágenes

afortunadas (*lucky*) se basa en el valor del máximo de cada imagen. Se consideran imágenes *lucky* aquéllas cuyo pixel de mayor intensidad supere un umbral (porcentaje) determinado por el mayor máximo de todas las imágenes del objeto. La adecuación de este criterio está justificada por el hecho de que la atmósfera generalmente distorsiona el frente de onda, manteniendo el flujo óptico (cantidad de luz) casi constante. Por tanto, una imagen con un máximo más alto será más nítida (perfilada) que otra con un máximo inferior, que estará más difuminada/distorsionada. Las imágenes afortunadas son posteriormente recentradas con respecto a dicho máximo (píxel más brillante) y acumuladas, procedimiento que se conoce como "reducción" de imágenes.

Un diagrama de bloques del mismo, que realiza dos acumulaciones distintas, puede verse en la Figura 2.4.

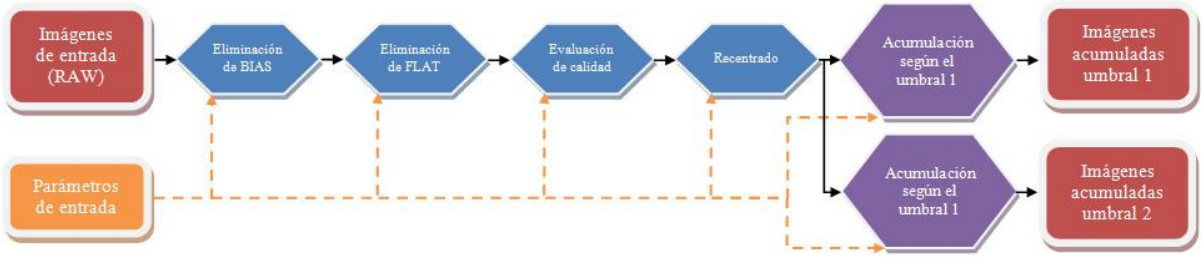

Figura 2.4. Esquema de procesado de imágenes del instrumento FASTCAM.

A continuación se describe cada uno de los bloques que componen el algoritmo:

Entradas: Por un lado se tiene la entrada de imágenes crudas procedentes de la cámara (128 x 128 píxeles de tamaño y 14 bits por píxel de precisión) y por otro los parámetros de configuración del sistema que se listan.

Eliminación de BIAS: Este bloque realiza una resta píxel a píxel entre la imagen de entrada cruda (RAW) y una imagen de BIAS de la cámara que se toma antes de las observaciones y que contiene información sobre la componente continua y de ruido de fondo de la cámara.

Eliminación de FLAT: De forma dual a la imagen de BIAS, la imagen de FLAT se toma antes de las observaciones y se refiere en este caso a la componente de degradado de campo de las imágenes de salida de la cámara, siendo su eliminación efectuada a través de un cociente píxel a píxel entre la imagen cruda de entrada y la de FLAT.

Evaluación de calidad: Esta parte se encarga de obtener un parámetro de calidad para cada una de las imágenes crudas que lleguen al sistema; en una primera implementación este parámetro de calidad se corresponde con el píxel más brillante de la imagen, es decir, el que tenga un valor más alto de píxel.

Recentrado: Conocido el valor de calidad asociado a la imagen, se realiza un centrado de la misma con respecto de la posición del valor máximo, es decir, se deja en la posición central el píxel con mayor calidad.

Acumulación: Finalmente, se acumulan las imágenes centradas si cumplen con un criterio de calidad, que se basa en que el mayor valor de calidad de la imagen supere un cierto umbral configurable.

Cabe destacar, que al tratarse de una implementación hardware para tiempo real, la selección de imágenes consideradas “buenas” se realiza en función a un umbral ya establecido (vía parámetros de

entrada), y se almacenan sólo las que cumplen dicho umbral; en lugar de seleccionar de todas las imágenes recibidas, las que mejor cumplan el criterio de calidad, ya que esto es usual cuando se dispone inicialmente de todas las imágenes, que es lo que ocurre cuando se trabaja en software.

De esta manera, el instrumento ha sido probado con éxito en los telescopios *Telescopio Carlos Sánchez* (1.5m), 2.5m *Telescopio Óptico Nórdico* (2.5m) y *William Herschel Telescope* (4.2m) en los Observatorios del Teide y del Roque de los Muchachos. Se ha logrado alcanzar el límite de difracción con los tres telescopios en banda I (850 nm) y resoluciones comparables en bandas V (550 nm) y R (700 nm).

2.2. Algoritmos DWT

En este Proyecto Fin de Carrera se introducen ciertas modificaciones al algoritmo básico descrito en el epígrafe anterior para mejorar su capacidad. Estas modificaciones consisten en insertar sendas etapas de pre y post procesamiento basadas en la transformada *wavelet*, manteniendo las etapas centrales del algoritmo original.

La utilización de la DWT, y en concreto, del algoritmo *à trous* para la transformada, se justificará convenientemente al final del epígrafe, pero primeramente se presentarán los conceptos básicos sobre la DWT.

Una definición no técnica de una *wavelet* (término inglés que puede traducirse por ondícula) sería la de una onda cuya amplitud comienza en cero, a continuación toma valores positivos y negativos y vuelve a cero. Las áreas positivas y negativas de una *wavelet* son iguales, por lo que la integral de una *wavelet* siempre es cero. Hay *wavelets* de muy diversas formas (como por ejemplo las de la Figura 2.5), todas ellas creadas con el propósito fundamental de ser de utilidad en aplicaciones de procesado de señal. Esto es así porque si se convoluciona una *wavelet* con una señal cualquiera, podemos extraer del resultado información de la señal. En este principio se basa la técnica de procesado de señal denominada *transformada wavelet*. Si los datos sobre los que se aplica la transformada y la propia *wavelet* son datos digitales, la técnica se denomina *transformada wavelet discreta* o DWT (del inglés *discrete wavelet transform*).

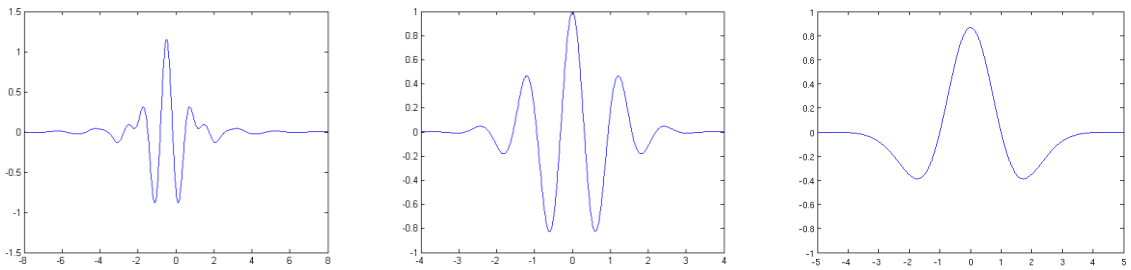

Figura 2.5. De izquierda a derecha: *wavelets "*Meyer", "Morlet*"* y "sombrero mexicano".

La DWT ha sido ampliamente usada a lo largo de los últimos años, especialmente en lo que al análisis de datos astronómicos se refiere; como consecuencia directa del hecho de que este tipo de datos generalmente presenta una estructura jerárquica compleja, que puede ser convenientemente descompuesta a diferentes niveles utilizando métodos multiescala como la DWT. Las técnicas basadas en la DWT son típicamente utilizadas en el procesado de imágenes astronómicas para eliminar ruido, ponderación de información, etc. en cada nivel de descomposición, o para destacar o eliminar artefactos en función de su contenido frecuencial espacial.

Sin embargo, la DWT que mejor se adapta a dichas aplicaciones astronómicas no es la transformada discreta "clásica", sino una aproximación discreta alternativa, denominada de tipo "*à trous"*.

La DWT *à trous* consiste en una descomposición basada en la transformada *wavelet* discreta en la que no se produce submuestreo de las imágenes, de tal forma que las subbandas resultantes de la transformada *wavelet* tendrán las mismas dimensiones que la imagen original.

La aplicación de la DWT *à trous* presenta una ventaja significativa con respecto a la aplicación de una transformada *wavelet* típica. Esto se debe a que una *transformada wavelet "clásica"* es ortogonal, decimada y variante respecto al desplazamiento, lo que tiene el inconveniente de que la imagen reconstruida puede presentar artefactos visuales debido al *aliasing* o a fenómenos *pseudo-Gibbs* [**11**]. Por el contrario, la *transformada wavelet à trous* es isotrópica, no diezmada, e invariante al desplazamiento; por lo el problema anterior no se presenta. De esta forma, los resultados obtenidos sobre datos con un nivel significativo de ruido (como son las imágenes de naturaleza astronómica) son mejores que en el caso de utilizar una *transformada wavelet* "clásica"; siendo ésta una nueva razón para utilizar la DWT *à trous* en este proyecto.

Esta DWT *à trous* permite la descomposición de los datos originales en diferentes escalas *wavelet*, de tal manera que la aplicación de este algoritmo sobre una imagen, da como resultado una serie de aproximaciones a la misma que tienen el mismo tamaño que la original. Cada una de estas aproximaciones o escalas *wavelet* va a contener la información de la imagen original referente a unas determinadas frecuencias espaciales. Así, este hecho posibilita la aplicación de algoritmos de reducción de ruido, ponderación, etc. a cada nivel, y la posterior reconstrucción de los datos originales, ahora mejorados, a partir de los diferentes niveles. Dado que la media de la intensidad de los píxeles de cada escala es cero, es posible multiplicar, dividir, o umbralizar escalas DWT para destacar o eliminar artefactos en función de su contenido frecuencial espacial. Como posteriormente (mediante una suma ponderada) se puede reconstruir la imagen original a partir de estas escalas *wavelet*, este tipo de procesado nos va a permitir un ajuste muy bueno a la hora de aplicar procesado únicamente a determinados artefactos o a la hora de separar determinados artefactos del resto de la imagen.

Para realizar un análisis matemático de esta transformada [**12**][**13**], en primer lugar hay que decir que en lo sucesivo se va a considerar que los datos muestreados, $\{c_0(k)\}$, son el producto escalar, en el pixel k, de la función $f(x)$ con una función de escalado, $\varphi(x)$, que se corresponde con un filtro paso-bajo, de tal forma que:

$$c_0(k) = < f(x), \varphi(x - k) > \tag{2}$$

La función de escalado se elige de forma que cumpla la ecuación de dilatación:

$$\tfrac{1}{2}\varphi\left(\tfrac{x}{2}\right) = \sum_l h(l)\varphi(x-l) \tag{3}$$

Donde h es un filtro paso-bajo discreto asociado con la función de escalado φ. La distancia entre niveles se incrementa en un factor de 2 de una escala a la siguiente. Los datos suavizados $c_i(k)$ en una resolución dada i y en la posición k vienen determinados por la ecuación:

$$c_i(k) = \tfrac{1}{2^i}\left\langle f(x), \varphi\left(\tfrac{x-k}{2^i}\right)\right\rangle \tag{4}$$

Que puede calcularse mediante la convolución:

$$c_i(k) = \sum_l h(l) c_{i-1}\left(k + 2^{i-1} l\right) \tag{5}$$

La transformada *wavelet* discreta en la escala *i*, w_i, es la diferencia entre dos resoluciones consecutivas:

$$w_i(k) = c_{i-1}(k) - c_i(k) \tag{6}$$

Es decir:

$$w_i(k) = \frac{1}{2^i}\left\langle f(x), \psi\left(\frac{x-k}{2^i}\right)\right\rangle \tag{7}$$

Donde la función *wavelet* ψ se define como:

$$\frac{1}{2}\psi\left(\frac{x}{2}\right) = \varphi(x) - \frac{1}{2}\varphi\left(\frac{x}{2}\right) \tag{8}$$

Así, la *transformada wavelet à trous* producirá llegados a este punto un conjunto w_j para cada escala *j*. Este conjunto tendrá el mismo número de píxeles que el conjunto de datos de entrada y contendrá la información del conjunto original para una determinada frecuencia espacial, como ya se ha comentado.

Además, el algoritmo que permite reconstruir la imagen original consiste simplemente en añadir al plano suavizado de la última escala, c_n, todas las diferencias w_i:

$$c_0(k) = c_n(k) + \sum_{j=1}^{n} w_j(k) \tag{9}$$

Los coeficientes {*h*(*k*)} se derivan de la función de escalado (3). Eligiendo una función triángulo (interpolación lineal) para la función de escalado φ, se tiene:

$$\begin{aligned} \varphi(x) &= 1 - |x| \qquad \text{si} \quad x \in [-1,1] \\ \varphi(x) &= 1 - |x| \qquad \text{si} \quad x \notin [-1,1] \end{aligned} \tag{10}$$

Se obtiene,

$$c_1(k) = \frac{1}{4} c_0\left(k - 2^{i-1}\right) + \frac{1}{2} c_0(k) + \frac{1}{4} c_0\left(k + 2^{i-1}\right) \tag{11}$$

Y en general:

$$c_i(k) = \frac{1}{4} c_{i-1}\left(k - 2^{i-1}\right) + \frac{1}{2} c_{i-1}(k) + \frac{1}{4} c_{i-1}\left(k + 2^{i-1}\right) \tag{12}$$

Es decir,

$$h = \left(\frac{1}{4} \quad \frac{1}{2} \quad \frac{1}{4}\right) \tag{13}$$

Los coeficientes wavelet w_i en una escala determinada vienen dados por tanto por:

$$w_i(k) = -\frac{1}{4} c_{i-1}\left(k - 2^{i-1}\right) + \frac{1}{2} c_{i-1}(k) - \frac{1}{4} c_{i-1}\left(k + 2^{i-1}\right) \tag{14}$$

En este proyecto se va a usar una función de escalado de tipo *lineal*, y su *wavelet* correspondiente (ambas en Figura 2.6) por ser una buena aproximación a la forma teórica del núcleo de una estrella

observado a través de un telescopio con una configuración estándar. Esto es debido a que, dada la configuración habitual de los equipos de observación y a la distorsión atmosférica, las estrellas no se muestran en una imagen como puntos de luz definidos, sino más bien como una mancha con una función de dispersión similar a la Gausiana. Por contra, se puede comprobar que el 91.0% de la energía de una estrella está contenida en un área de 5x5 píxeles; mientras que el 83.8% de dicha energía en un área de 3x3 píxeles. La primera y segunda escala de descomposición *wavelet* se corresponden con máscaras de convolución de 3x3 y 5x5 respectivamente. En base a esto, tiene sentido desarrollar un algoritmo basado en la obtención de la primera y segunda escalas de descomposición *Wavelet* (*w1* y *w2*), ya que es en éstas donde se espera encontrar la información más relevante para discernir los núcleos de las estrellas.

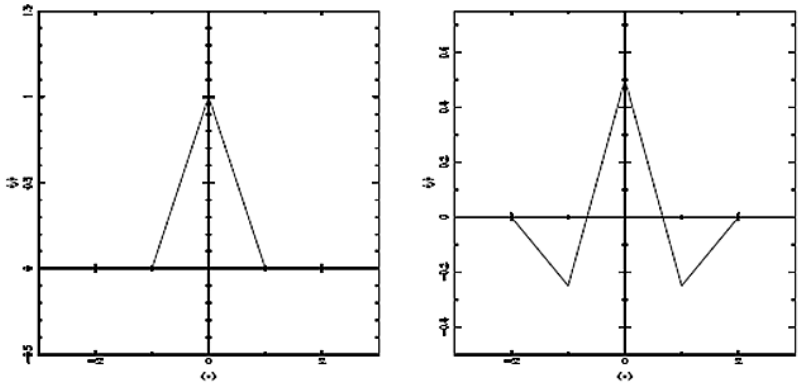

Figura 2.6. Izquierda, función φ de tipo lineal; derecha, *wavelet* ψ correspondiente.

De esta manera, el núcleo de procesado para la implementación de la etapa de preprocesado DWT desarrollado en el presente proyecto, se basará en la descomposición en escalas *wavelet* de las imágenes, procesado individual de las escalas, y reconstrucción de la imagen original.

El esquema que describe la secuencia lógica de operaciones a seguir para el caso de aplicar la DWT *à trous* usando una función de escalado de tipo *lineal* a una imagen de entrada c_0 puede observarse en la Figura 2.7, que también muestra los coeficientes de los filtros que se aplican.

Dado que se va a trabajar con imágenes, el *kernel* de convolución lineal descrito en la ecuación (13) debe extenderse para datos bidimensionales. En dos dimensiones, la función de escalado se obtiene como una función separable a partir de la función de escalado unidimensional [**14**], mediante la siguiente expresión:

$$\varphi_{3x3}(n,m) = \varphi(n)\varphi(m) \tag{15}$$

Como consecuencia directa, el *kernel* de convolución h_{3x3} que se obtiene de la función de escalado bidimensional cumple que:

$$h_{3x3}(n,m) = h(n)h(m) \tag{16}$$

Para la obtención de los coeficientes del filtro 5x5 que se muestran en la figura siguiente, se observa que se ha insertado una fila de ceros entre cada par de filas de h_{3x3} y una columna de ceros entre cada par de columnas de h_{3x3}. De igual forma, para la obtención del filtro 9x9 se expande el filtro inicial h_{3x3} insertando el número de ceros apropiado entre los coeficientes. Este procedimiento caracteriza a la transformada *wavelet à trous*, ya que no submuestrea la imagen original, consiguiendo así, que las distintas escalas obtenidas representen aproximaciones de la imagen original de igual tamaño en las distintas bandas de frecuencia espaciales. Esta característica ha sido otro factor por el que se ha optado por esta transformada, ya que facilita la visualización de los resultados. No ocurre de igual manera en la DWT clásica, en la que los resultados para cada escala

wavelet son sólo coeficientes que no representan versiones de la imagen original, debido al proceso de diezmado. Así, la DWT clásica es más útil en compresión tanto de imágenes como de audio.

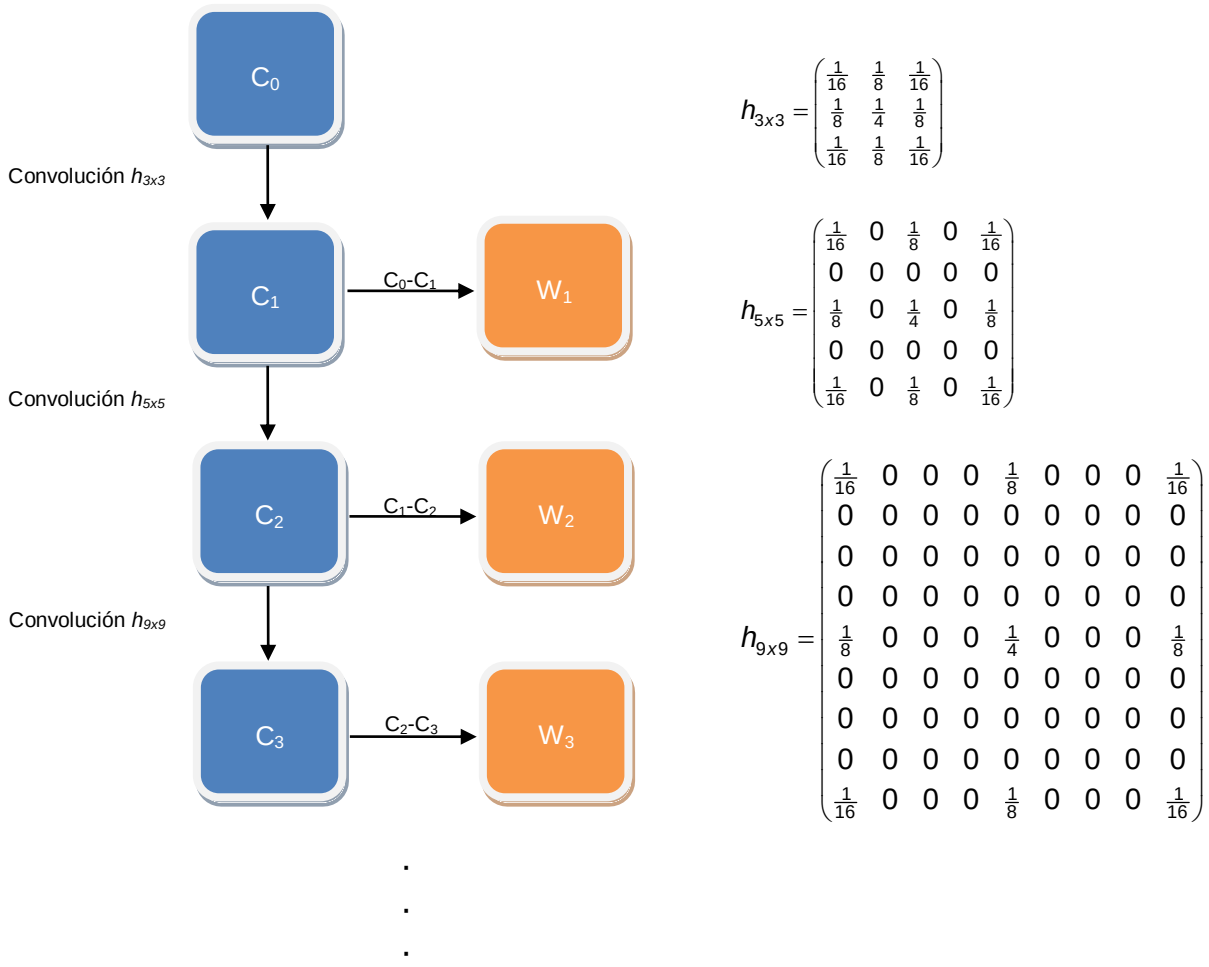

Figura 2.7. DWT *à trous* sobre una imagen de entrada C_0 usando una función de escalado de tipo lineal.

Como se comentó anteriormente, la reconstrucción que se llevará a cabo en este PFC consiste simplemente en la suma de las dos primeras escalas *wavelet* (*w1*+*w2*).

Por último, indicar que en el caso de utilizar otra función de escalado, los coeficientes de los filtros serán distintos. Así, si por ejemplo se utiliza una *spline* bicúbica como función de escalado, los coeficientes {*h*(*k*)} serán los siguientes:

$$h_{B3spline} = \left(\frac{1}{16} \quad \frac{1}{4} \quad \frac{3}{8} \quad \frac{1}{4} \quad \frac{1}{16}\right) \tag{17}$$

En el contexto del espacio bidimensional para procesado de imagen, la función *B3-spline* implicaría la siguiente máscara de convolución 5x5:

$$h_{B3spline} = \frac{1}{256}\begin{pmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix} \tag{18}$$

2.3. Trabajos Previos

En este apartado vamos revisar resultados y conclusiones obtenidas en estudios anteriores realizados por el departamento de *Electrónica, Tecnología de Computadores y Proyectos* de la UPCT, cuya motivación fue mejorar el algoritmo utilizado por FASTCAM.

Estas implementaciones han sido realizadas en una plataforma DS1002 de DRC *Computer Corporation* [**15**], cuya arquitectura se detallará en el siguiente capítulo. Por el momento basta indicar que esta plataforma se compone básicamente de un microprocesador *AMD Opteron*^{TM} *modelo 275,* y de una unidad de procesado reconfigurable basada en una FPGA *Virtex-4 LX200* de *Xilinx.*

Así, en dichos estudios se ha buscado aprovechar la importante capacidad de cálculo del súper-computador DRC para aplicar la DWT a cada imagen, ya que la descomposición en bandas de frecuencias espaciales (escalas *wavelet*, w_i) permite elegir que parte de la información tomar o desechar en cada imagen. Así, tras este filtrado realizado en hardware, la parte software realiza un recentrado de cada imagen recibida, en función del máximo y un promediado. Estos algoritmos programados en CoDeveloper, fueron probados para cubos de imágenes *fits*[1].

De esta forma, en un Proyecto Fin de Carrera [**16**] dirigido por este departamento en Septiembre de 2009, se implementó un sistema de procesado que se denominó "*pseudo-Wavelet*" inspirado en la obtención de algo similar a la w_2, pero sin aplicar el primer filtro paso bajo de 3x3, que eliminaría la más alta frecuencia. Así, considerando la *wavelet* correspondiente a una función de escalado lineal, se realizó el sistema presente en la Figura 2.8, que en primer lugar realiza una convolución con una máscara de radio 2 píxeles y después realiza la resta de la imagen con esta convolución.

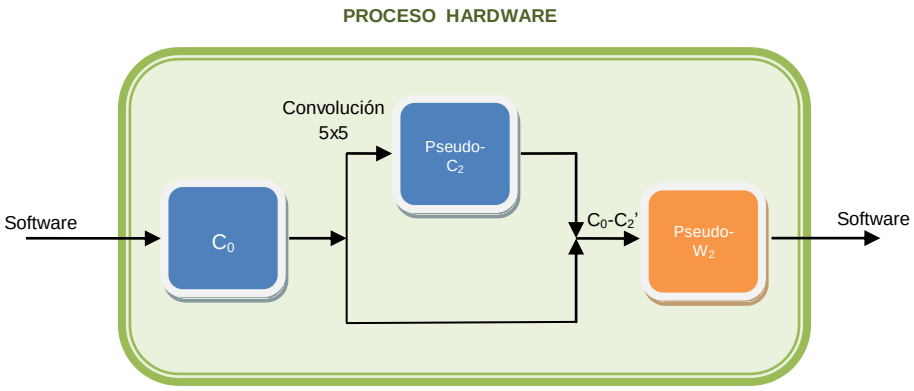

Figura 2.8. Esquema del algoritmo "*pseudo-Wavelet*" implementado.

El motivo de esta implementación no basada de manera exacta en ninguna teoría de descomposición de imágenes, en lugar de un sistema que calcule la suma de las escalas w_1 y w_2, cuyo interés se indica en el apartado 2.2 de este PFC, es la simplicidad de su arquitectura.

Por tanto, aunque los resultados que se obtuvieron nada tienen que ver con una escala de descomposición *Wavelet*, dieron una información similar a la que da w_2 incluyendo información de alta frecuencia. Este resultado se comparó con la obtención de la escala w_2 únicamente, comprobando así la relevancia de la información de alta frecuencia presente en w_1. Se observó también que esta arquitectura introducía un ruido al resultado.

[1] FITS (*Flexible Image Transport System*)[**26**], diseñado a finales de los '70, es el formato de datos más utilizado en el mundo de la astronomía. Un archivo de datos FITS se compone de una secuencia de unidades de datos más una cabecera legible en ASCII.

Con el fin de comprobar si realmente se consiguió una aceleración hardware en la arquitectura anterior implementada en el computador DRC, se realizó la comparativa de tiempos de ejecución que se muestra en la Tabla 2.1. Para la comparativa, en la que se consideraron 90 imágenes, se realizó una medida del tiempo de ejecución del algoritmo para la arquitectura implementada y las dos simulaciones software que se detallan a continuación:

- La simulación mediante el ejecutable que genera CoDeveloper para la depuración de la arquitectura implementada, ejecutado en una máquina IntelCore2 Quad a 2.40GHz.
- La simulación mediante un ejecutable generado tras la compilación de las librerías de ImpulseC y el código C de nuestro proyecto, ejecutado en una máquina IntelCore2 Quad a 2.40GHz.

Especificaciones de la Simulación	**Tiempo (seg)**	**Relación de tiempos con coejecución DRC**
Coejecución Hardware/software en el computador DRC	46,08	
Ejecutable ImpulseC, en una Máquina IntelCore2 Quad a 2.40GHz	274,20	5,95
Ejecutable compilado, en una Máquina IntelCore2 Quad a 2.40GHz	138,49	3,01

Tabla 2.1. Comparativa del tiempo de ejecución del algoritmo implementado.

Como se muestra en los resultados, la co-ejecución del algoritmo, que consume 0,51 segundos/imagen, supone una aceleración hardware efectiva de 3x comparándola con la ejecución en la del ejecutable compilado.

Otros resultados que cabe destacar son los presentados en mayo de este año por este departamento [**7**]. En este artículo se propone un algoritmo desarrollado específicamente para acelerar el instrumento astronómico FASTCAM sobre una plataforma HPRC.

El algoritmo propuesto consiste en cuatro etapas (Figura 2.9). La primera aplica una Transformada *Wavelet* Discreta *à trous* (*DWT à trous*) a cada imagen (denominada *specklegram*) para eliminar la información en frecuencias espaciales fuera del rango de objetos astronómicos de interés. La segunda etapa usa estas imágenes mejoradas para calcular un factor de calidad (QF) usado para determinar si la imagen es o no una *Lucky-Image*. En esta ocasión se propone un nuevo QF basado en la ratio entre el flujo óptico de dos regiones vecinas de radios diferentes, alrededor del centro de masas del objeto. Este QF arrojó resultados similares en calidad al criterio del máximo. Su ventaja con respecto a éste último es que se trata de un parámetro independiente de cada imagen, es decir, permite determinar de forma absoluta la calidad de una imagen, y no de forma relativa (función de la mejor imagen), como hace el máximo. De esta forma, la ratio del flujo óptico entre diferentes vecindades de una estrella es un factor de calidad que puede utilizarse en sistemas que procesan imágenes en flujo de datos, como los aceleradores mediante hardware específico que son el objeto de este proyecto.

En la tercera etapa, las imágenes afortunadas son desplazadas y acumuladas (*shift & add*) utilizando el algoritmo clásico que propone la técnica *Lucky Imaging*. Finalmente la imagen registrada es post-procesada en una cuarta etapa, donde una nueva DWT es calculada, seguida por un perfilado (filtrado *unsharp*) para resaltar los objetos relevantes. Esta última etapa se realiza por software porque, aunque requiere una nueva DWT, se aplica a la imagen reducida (una vez cada 1000 imágenes recibidas, usualmente), en lugar de a todas las imágenes, como en la segunda etapa.

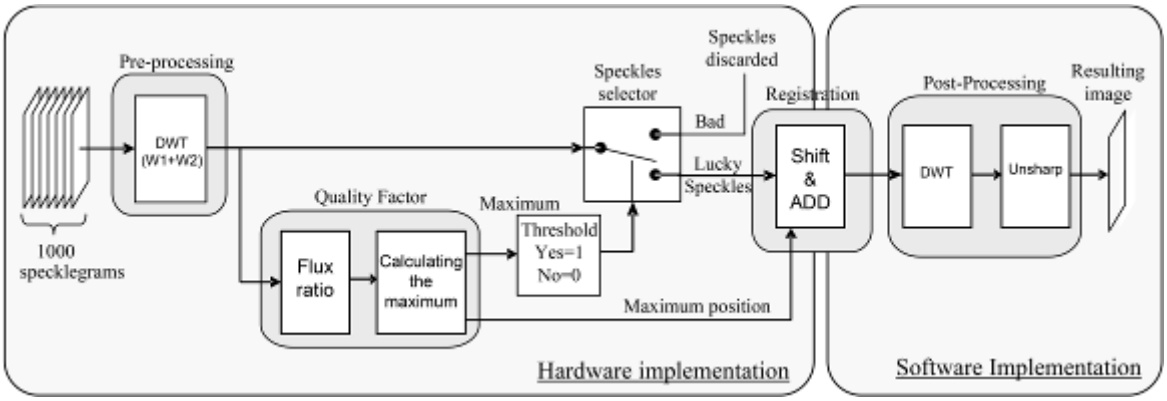

Figura 2.9. Algoritmo hardware propuesto.

En este artículo, se implementa un *kernel* de convolución para la computación de la DWT *à trous*, que utiliza la función *B3-spline* como función de escalado, cuyos coeficientes ya se indicaron en el apartado 2.2.

Al igual que en el PFC descrito anteriormente, se ha hecho uso de la plataforma de DRC descrita al inicio de este apartado. Así, el algoritmo se particionó en varios procesos con el objetivo de ejecutar tareas intensivas de computación (las dos primeras etapas y la parte del desplazamiento de la tercera) sobre hardware dedicado, mientras que las tareas menos costosas (acumulación de imagen, la cuarta etapa, la lectura/escritura de imagen, y el control de rutinas) son ejecutadas como puro software en el microprocesador Opteron.

La Tabla 2.2 resume los recursos hardware consumidos por el algoritmo. El sistema se configuró para procesar imágenes de 512x512 píxeles en formato escala de grises punto fijo de 16 bits.

Resource	**Quantity (% used)** **Pre-processing + QF + shift**
Slices	919 (2%)
Flip flops	932 (2%)
DSP48	53 (57%)
BRAM	14 (5%)

Tabla 2.2. Recursos hardware. Los porcentajes están referidos a la plataforma DRC-RPU FPGA (Virtex-4 LX200).

La Tabla 2.3 resume el tiempo de ejecución del algoritmo para procesar una imagen en tres plataformas distintas: todo software en un procesador a Core2Duo™ sobre Windows XP™; todo software en el DS1002 con el procesador Opteron™ y el sistema operativo Linux; y la co-ejecución mixta hardware/software en la plataforma DS1002, donde se tomó un reloj de 133 MHz en la generación del hardware para la RPU.

Platform			**Per Frame Runtime (s)**
MicroProccesor	*Operating System*	*Implementation*	
Core2Duo a 2.4GHz (PC)	Win XP	All Software	0.115
Opteron a 2.2GHZ (DS1002)	Linux	All software	0.196
Opteron a 2.2GHZ (DS1002)	Linux	Co-execution H/S	0.022

Tabla 2.3. Tiempos de ejecución del algoritmo propuesto.

A la vista de los resultados queda claro que la solución más rápida es la obtenida mediante la co-ejecución del algoritmo, que consume solamente 0.022 segundos/imagen. Esto es aproximadamente 5 veces más rápido que una implementación software en un PC estándar (Core2Duo@2.4GHz).

Con el fin de evaluar la calidad de los resultados obtenidos con el algoritmo propuesto, el sistema se evaluó usando datos reales procesados en el DS1002. Estos datos consisten en pilas de 1000 imágenes astronómicas adquiridas por telescopios terrestres en el observatorio del Roque de Los Muchachos [**17**]. La Figura 2.10 muestra los resultados obtenidos cuando se intenta resolver dos objetos astronómicos diferentes, una pequeña región de cúmulo globular *M15* (fila superior) y una típica estrella binaria (fila inferior). Para cada objeto se usó una pila de 1000 imágenes.

En ambos casos, la columna de la izquierda muestra una imagen que simula los resultados que se podrían obtener con un instrumento clásico de larga exposición. La columna central muestra la imagen resultante del instrumento FastCam, usando la técnica de corta exposición *Lucky-Image*. Finalmente, la columna de la derecha muestra la imagen obtenida con el nuevo algoritmo propuesto.

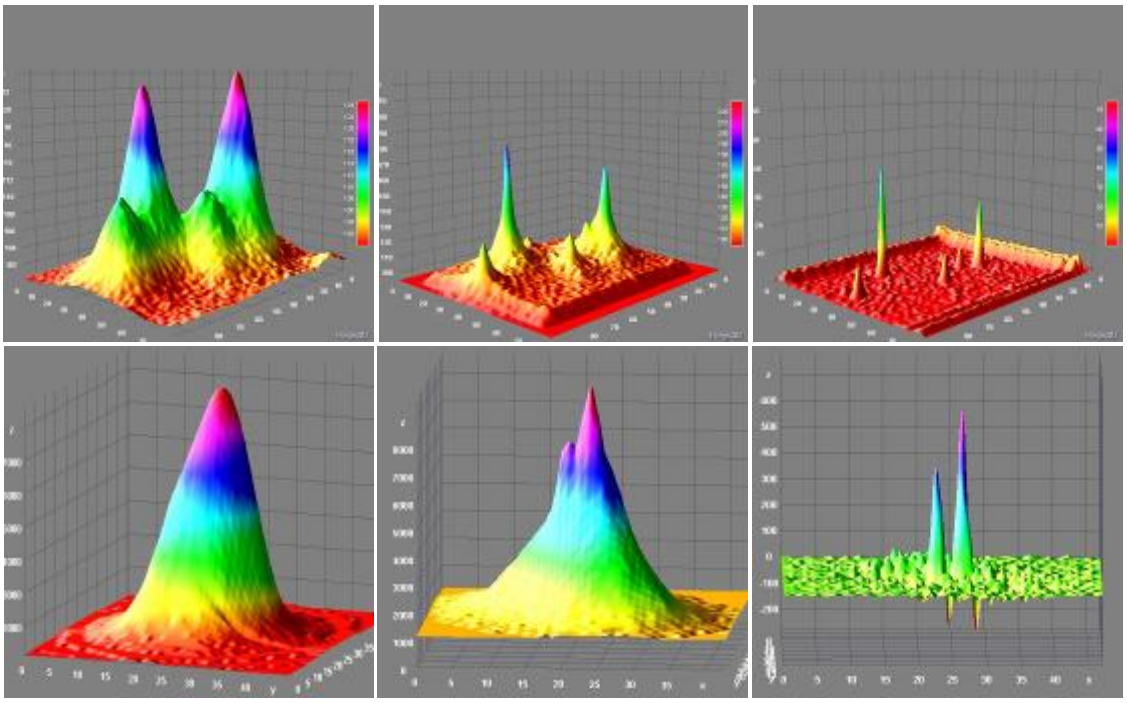

Figura 2.10. Fila superior: Región de cúmulo globular *M15*. Fila inferior: estrella binaria. En ambos ejemplos: columna izquierda, sin FastCam; centro, algoritmo básico de FastCam; derecha, algoritmo propuesto basado en DWT.

A la vista de los resultados, se consigue una mejora considerable tanto en la calidad de la adquisición y detección de objetos astronómicos, como en el rendimiento, gracias a la implementación basada en el HPRC.

Capítulo 3

Herramientas de Desarrollo

Como ya se ha comentado, CoDeveloper ha sido el entorno a nivel de sistema (ESL) elegido para la implementación de la arquitectura. Por tanto, en este apartado se va a describir esta herramienta, así como el modelo de programación utilizado. También se dará a conocer la arquitectura del nodo de supercomputación híbrido (**HPRC**, *High Performance Reconfigurable Computer*) modelo DS1002, de la empresa DRC, donde se ejecutará el algoritmo.

3.1. CoDeveloper y el Modelo de Programación de ImpulseC

3.1.1. Introducción

CoDeveloper, de Impulse Accelerated Technologies [**18**][**19**], es un entorno de desarrollo ESL (*Electronic System Level*) que permite definir sistemas completos compuestos de software ejecutándose sobre procesadores tradicionales y de hardware específico. Este tipo de aplicaciones mixtas hardware/software (en adelante, hw/sw) son bastante comunes, y representan una gran parte de las aplicaciones que requieren aceleración hardware.

CoDeveloper utiliza el modelo de procesos secuenciales comunicantes (CSP[2]) para la descripción de algoritmos. Así, un algoritmo se describe en una o más unidades de procesado, denominadas procesos, que se sincronizan de manera independiente, de tal forma que cada una de ellas va a implementar una etapa de la aplicación bajo desarrollo. A diferencia de las subrutinas software tradicionales, estos procesos no son "llamados" (invocados repetidamente durante la ejecución del programa), sino que se encuentran constantemente respondiendo a la llegada de datos en sus entradas.

La clave para una configuración óptima de este tipo de sistemas (ver Figura 3.1) consiste en implementar en hardware los procesos que requieran elevado coste computacional, y dejar los procesos que manejan la entrada/salida de datos, así como otras tareas cuyas prestaciones no se consideren críticas, para implementaciones software.

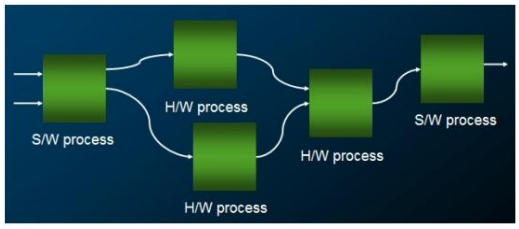

Figura 3.1. Ejemplo de esquema de bloques para un sistema mixto hw/sw.

[2] CSP (del inglés, *communicacting sequencial processes*) según Hoare [**18**], es un modelo de programación utilizado para describir los patrones de interacción entre componentes que operan de manera independiente pero sincronizada, denominados procesos.

Impulse proporciona librerías compatibles con el estándar ANSI C, que basándose en una mínima extensión del lenguaje C (en la forma de nuevos tipos de datos y nuevas funciones) permiten una rápida descripción de programas paralelos y bien sincronizados. De esta forma, la API (***Application Programming Interface***) definida por Impulse, proporciona un reducido número de funciones (unas 25) para soportar la paralelización de los procesos y los mecanismos de comunicación, algo absolutamente necesario para la descripción de hardware, ya que el lenguaje C estándar no soporta la programación concurrente. El lenguaje resultante de añadir estas extensiones al C se denomina ImpulseC.

La comunicación entre procesos se realiza principalmente mediante flujos de datos (***streams***), siendo usual también la utilización de memorias compartidas y de señales, a modo de ***flags***, para comunicaciones asíncronas (ver Figura 3.2).

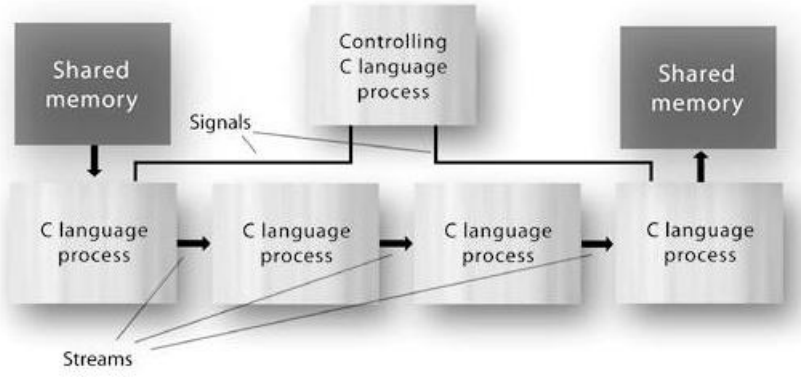

Figura 3.2. El modelo de programación de Impulse C destaca el uso de *streams* para las comunicaciones entre procesos, y también soporta señales y memorias compartidas.

Los datos que son procesados por la aplicación fluyen de proceso en proceso a través de los *streams*, o en algunos casos en forma de de mensajes o zonas de memoria compartida. Los *streams* son canales unidireccionales que comunican procesos concurrentes, y están auto-sincronizados con respecto a los procesos para realizar un *buffering* (si así se ha especificado durante la creación del *stream* en la fase de diseño).

El *buffering* de datos, el cual se implementa mediante FIFOs especificadas y configuradas por el programador, hace posible la escritura de aplicaciones paralelas a un alto nivel de abstracción, sin tener que sincronizar ciclo de reloj a ciclo de reloj, y con muy poco perjuicio para las características finales del sistema.

Las herramientas de Impulse incluyen un compilador, CoDeveloper, que traduce los programas ImpulseC a la representación de bajo nivel apropiada. De esta forma, traduce a C estándar la parte del programa que va a ser compilada sobre un microprocesador, y a una descripción hardware de bajo nivel, como puede ser VHDL, la parte que se sintetizará en una FPGA.

El compilador es capaz de optimizar la generación de la descripción hardware, identificando así un posible uso de paralelismo. De esta forma es capaz de desenrollar bucles (*unrolling loops*) o generar *pipelines* para conseguir el máximo nivel de paralelismo posible en la FPGA. Este entorno también incluye herramientas de simulación que permiten detectar posibles cuellos de botella en el flujo de datos y áreas de optimización.

Cabe destacar que los canales de comunicación entre procesos software (sw/sw), hardware (hw/hw), o mixtos (sw/hw y hw/sw) son inferidos automáticamente por CoDeveloper para un buen número de plataformas soportadas, entre las que se incluye el sistema DS1002.

En base a los párrafos anteriores se puede decir que, aunque hay infinidad de aplicaciones que pueden ser expresadas utilizando ImpulseC, las que mejor se adaptarían son las que presentan todas o algunas de las siguientes características:

- Una elevada tasa de transferencia de datos.
- Los tamaños de los datos son fijos.
- Es necesario realizar múltiples operaciones relacionadas, pero independientes, en un mismo flujo de datos.
- Hay referencias a memorias locales o compartidas para almacenar *arrays* de datos, coeficientes y otras constantes, y para depositar resultados del cómputo.
- Interesa comunicar múltiples procesos independientes principalmente a través de un flujo de datos con sincronización opcional mediante mensajes.

A continuación se describen con más detalle los principales elementos que incorpora ImpulseC, para extender el lenguaje ANSI C con características convenientes/necesarias para poder proyectar algoritmos software sobre una arquitectura hardware diseñada a medida.

3.1.2. Modelo de Programación

- **Procesos y Objetos de Comunicación.**

Procesos

Como ya se ha dejado entrever, un proceso es el resultado de la ejecución independiente de una sección del código ANSI C que describe nuestra aplicación y que está definida por una subrutina llamada *process run fuction*. Un proceso software se diseña para ejecutarse en un procesador convencional, mientras que un proceso hardware se diseña para implementarse en una FPGA u otro elemento de hardware programable.

Un proceso software estará sujeto a las limitaciones del procesador para el que se considere su diseño (que será un RISC común, un DSP particular o un procesador con poca capacidad), mientras que un proceso hardware está típicamente sujeto a unas limitaciones más fuertes (ya que los sistemas de hardware reconfigurable son más específicos). Por consiguiente, en las opciones del programa se deberá especificar de manera exacta el tipo de dispositivo a utilizar.

Las funciones que Impulse C añade a las expresiones estándar de C para realizar la comunicación entre procesos deben estar referenciadas tanto en los procesos hardware como en los procesos software. Estas funciones operan sobre objetos de comunicación compartidos entre procesos (típicamente los ya mencionados *streams*). Por ejemplo, para operaciones de lectura/escritura en un *stream* de datos que conecta dos procesos se utilizan los objetos *co_stream*.

El compilador CoBuilder genera descripciones de hardware compatibles con la síntesis hardware (genera código HDL compatible con las herramientas de síntesis comunes para FPGAs) y, además, genera una colección de procesos (en la forma de código C) para ser implementados en un procesador normal. Estas dos partes de la aplicación generadas por CoBuilder se corresponden con la separación software y hardware realizada en el diseño.

Objetos de comunicación

Impulse C define los siguientes objetos para la comunicación entre procesos. Todos ellos pueden verse en la Tabla 3.1.

Streams	*Streams* de datos con capacidad de *buffering* y de tamaño fijo.
Señales	Sincronización uno-a-uno con opción de transporte de datos
Semáforos	Sincronización uno-a-varios
Registros	Líneas de control sin capacidad de *buffering*
Memorias compartidas	Memorias compartidas entre software y hardware

Tabla 3.1. Objetos para la comunicación entre objetos.

Para usar estos mecanismos, es necesario crear objetos en la función de configuración y pasarlos a los procesos, que llamarán entonces a funciones sobre los objetos. Como ya se ha comentado, CoDeveloper genera automáticamente las interfaces necesarias para comunicar los procesos software con los hardware y viceversa.

- **Tipos de Datos**

Enteros (con y sin signo)

Impulse C proporciona tipos de datos enteros con y sin signo de manera predefinida[**20**], para un rango de bits de 1 a 64. Se utiliza una convención simple para el nombre de estos tipos predefinidos. En tipos con signo se utiliza *co_int* seguido del tamaño en bits, mientras que en tipos sin signo se utiliza *co_uint*. Todo esto puede verse en la Tabla 3.2.

Tipo en Impulse C	Modelado como
co_int2 – co_int8	int8
co_int9 – co_int16	int16
co_int17 – co_int32	int32
co_int33 – co_int64	int64
co_uint1	uint8
co_uint2 – co_uint8	uint8
co_uint9 – co_uint16	uint16
co_uint17 – co_uint32	uint32
co_uint33 – co_uint64	uint64

Tabla 3.2. Tipos de datos enteros (con y sin signo).

Estas variables pueden ser usadas para programar tanto los procesos hardware como los software, pero hay casos en que es obligatorio su uso, como ocurre en el uso de *streams*. Además, es una práctica recomendable programar nuestros procesos hardware utilizando exclusivamente estos tipos de datos.

Tipos enumerados

La palabra clave *enum* es soportada por Impulse C para definir enumeraciones de enteros tanto en procesos hardware como software.

- **Optimización Mediante *Pragmas***

En este apartado se dan a conocer algunos de los *pragmas,* o instrucciones que se pueden dar al compilador, que permiten definir los estilos de codificación en lenguaje C que generen la implementación hardware más eficiente en términos de rendimiento, área y consumo. Estos *pragmas* se expresan usando la directiva #pragma[**18**][**20**], y se aplican a todas las sentencias comprendidas entre la declaración del mismo, y la llave ("}") que finaliza el bloque de código.

CO UNROLL

Este *pragma* se utiliza para desenrollar bucles, para lo que replica el código del bucle tantas veces como sea necesario para implementar la operación descrita. Es importante considerar el tamaño del hardware resultante y desenrollar bucles que tengan un número de iteraciones relativamente pequeño. Las iteraciones del bucle no deben incluir interdependencias que impidan su implementación como una estructura paralela.

CO PIPELINE

Este *pragma*, similar conceptualmente a una cadena de montaje, se utiliza para forzar al optimizador a paralelizar las sentencias dentro del bucle donde se aplique *pipeline*, con el objetivo de reducir el número de ciclos de instrucción requeridos para procesar el *pipeline* completo. Si el *pipeline* no es posible, las etapas[3] se generarán secuencialmente. La generación del número de etapas del *pipeline* puede también controlarse mediante el parámetro *stageDelay*, que especifica el máximo retardo para las etapas generadas.

CO SET

Este *pragma* es utilizado para pasar información de optimización al optimizador a través de sus argumentos (nombre del parámetro y valor).

CO IMPLEMENTATION

Este *pragma* se utiliza para identificar una subrutina en C como un bloque hardware implementado externamente. De esta forma, el compilador no generará hardware para la función o procedimiento donde se aplique este *pragma*, sino que generará una referencia a nivel de *netlist* para el hardware externo.

CO FLATTEN

Este *pragma* es útil para realizar, en una única etapa, la lógica de control generada, consistente en sentencias *case* o cadenas complejas *if-then-else*, que de otra forma requerirían múltiples etapas. El compilador aplicará implícitamente *flattening* cuando se use el *pragma* CO PIPELINE.

CO NONRECURSIVE

Este *pragma* se usa para indicar al optimizador que una variable de tipo *array* no es recursiva. Cuando se accede a un *array* en un *pipeline*, el compilador debe planificar el acceso al *array* de forma que los valores escritos en previas iteraciones del *pipeline,* se lean correctamente en futuras iteraciones.

[3] Todas las sentencias dentro de una etapa se implementan en lógica combinacional en un único ciclo de reloj.

3.1.3. El Entorno CoDeveloper

En esta sección se mostrarán las características del entorno de desarrollo CoDeveloper, apoyadas con la creación de un nuevo proyecto a modo de ejemplo. Para ello comenzaremos accediendo a la herramienta de la forma: *Start -> Programs -> Impulse Accelerated Technologies -> CoDeveloper -> CoDeveloper Application Manager.*

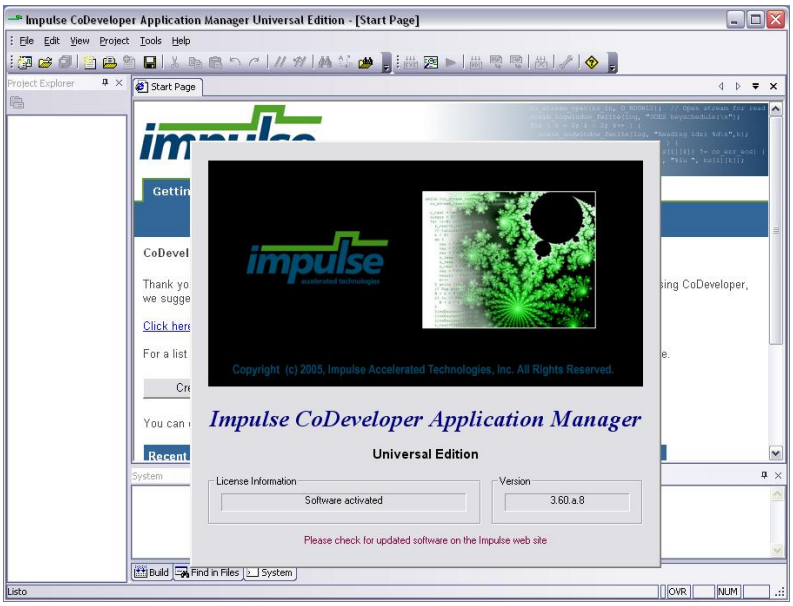

Figura 3.3. Inicio del software CoDeveloper.

- **Creación de un Nuevo Proyecto**

Para crear un nuevo proyecto en CoDeveloper se debe acceder al ítem *New Project* del menú *File*. Aparecerá entonces el cuadro de diálogo de la Figura 3.4, que facilita plantillas de los sistemas híbridos software/hardware más sencillos.

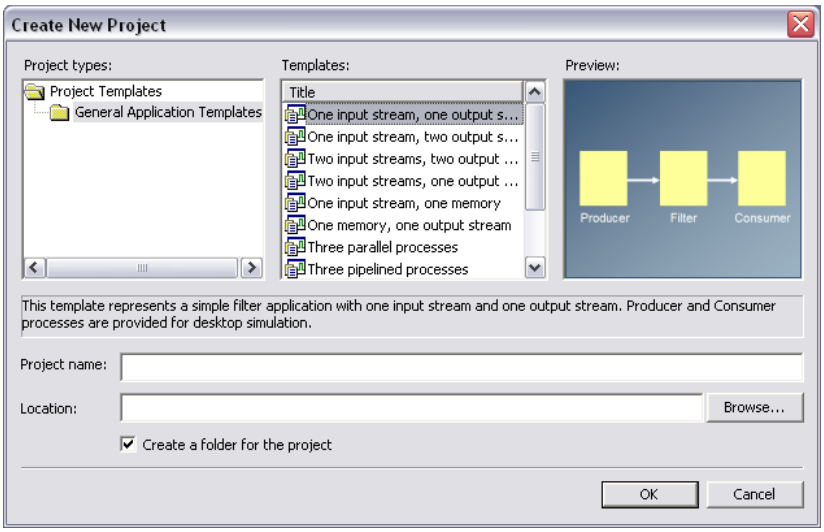

Figura 3.4. Crear un nuevo proyecto en CoDeveloper (I)

En este ejemplo se utilizará la primera arquitectura, definida por dos procesos software, *Producer* y *Consumer*. El primero se encargará de la lectura de datos y el envío de los mismos a la parte

hardware, y el segundo de la recepción de los datos procesados en el hardware y la posterior escritura de ficheros.

En la siguiente ventana se introducirá el nombre del proceso hardware, el nombre del *stream* de entrada al hardware y el nombre del *stream* de salida del hardware. Igualmente, se seleccionará el tamaño de los datos del *stream* y el tamaño del *buffer*. Se pulsa *siguiente* hasta el cuadro de diálogo que finaliza la creación del nuevo proyecto (Figura 3.5).

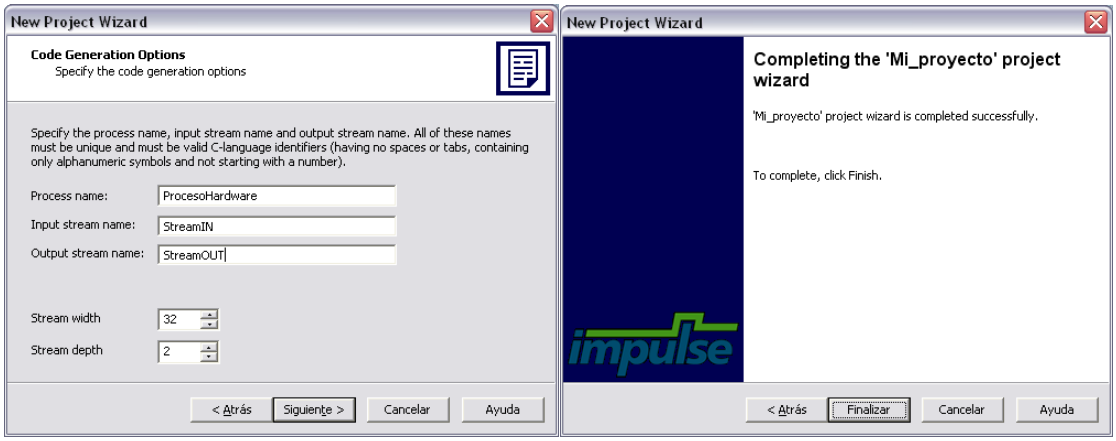

Figura 3.5. Crear un nuevo proyecto en CoDeveloper (II)

De esta forma se obtienen los ficheros que contienen las descripciones en ANSI C para los procesos software y hardware del proyecto creado, así como un fichero cabecera (Figura 3.6).

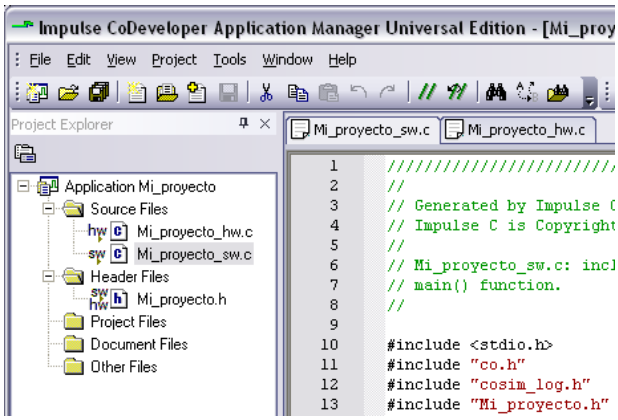

Figura 3.6. Visualización de ficheros en *Project Explorer*.

Estos ficheros, agrupados en la pestaña *Project Explorer* se clasifican de la siguiente forma:

- *Source Files*: son los ficheros que incluyen el código C asociado a la aplicación. Cada fichero ".c" puede representar uno o más procesos que se compilarán en software o hardware. En el nombre del fichero aparece al final un *_hw* o *_sw*, indicando si los procesos que contiene pertenecen a la parte hardware o software de la aplicación.
- *Header Files*: son los ficheros cabecera ".h" necesarios para la compilación. Aquí no se incluyen los ficheros cabecera globales, como el *co.h* o el *stdio.h*, sino únicamente los que son específicos para ese proyecto en concreto.
- *Project Files*: Son los ficheros externos asociados al proyecto.
- *Document Files*: Documentos asociados al proyecto (.doc, .htm, .txt, .pdf, …).
- *Other Files*: Ficheros que no se ajustan a ninguna de las categorías anteriores.

Para configurar las opciones del proyecto se debe acceder al ítem *Options* del menú *Project,* y seleccionar las opciones deseadas en las pestañas correspondientes. Las opciones de la pestaña *Build* (ver Figura 3.7) afectan a la ejecución de la utilidad *Make*. En concreto, en este dialogo hay que especificar los ficheros (*sources* y *headers*) necesarios tanto para la simulación de escritorio, como para la compilación sobre la plataforma final. Si se selecciona la opción *Generate makefile*, CoDeveloper genera automáticamente las dependencias en el *makefile* a partir de la información proporcionada.

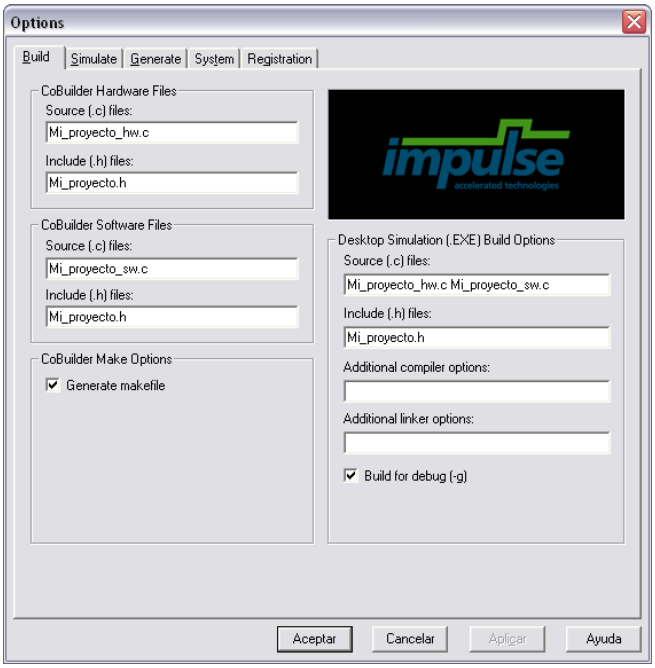

Figura 3.7. Opciones de la pestaña *Build*.

En la pestaña *Simulate* (ver Figura 3.8) se muestran las opciones relativas a la simulación de escritorio (simulación software), simulación hardware usando *Stage Master Debugger*, y generación de un *testbench* HDL usando *CoValidator*.

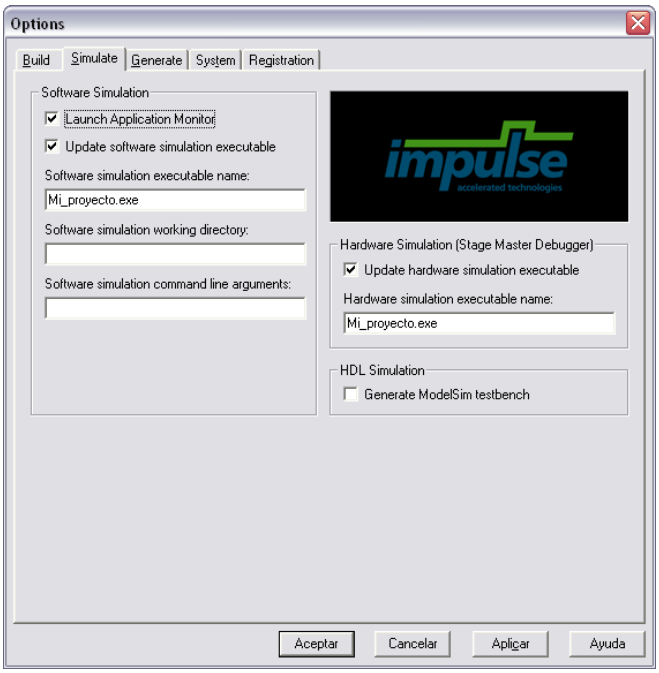

Figura 3.8. Opciones de la pestaña *Simulate*.

En pestaña *Generate* (Figura 3.9, izquierda) se especifican las opciones para generación de la descripción HDL, donde se debe indicar al *CoBuilder* la plataforma sobre la que se va a trabajar.

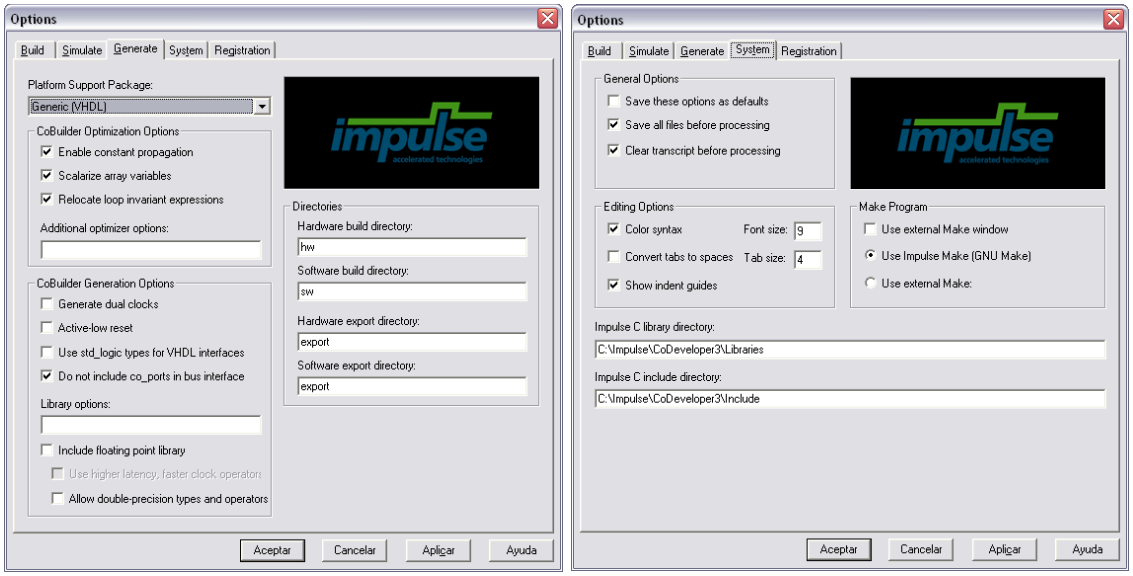

Figura 3.9. Opciones de las pestañas Generate (izquierda) y System (derecha) de CoDeveloper.

En la pestaña de las opciones *System* (Figura 3.9, derecha) se especifican las opciones de comportamiento de CoDeveloper como editor de texto. Además, se especifican las rutas de las librerías de ImpulseC del sistema.

- **Estructura General del Código**

En la Figura 3.10 se muestra la parte del código generado automáticamente en el fichero "Mi_proyecto_sw.c", correspondiente al proceso software *Producer*.

```
void Producer(co_stream StreamIN)
{
	int c;
	IF_SIM(cosim_logwindow log = cosim_logwindow_create("Producer");)

	// Read sample data from a file
	const char * FileName = INPUT_FILE;
	FILE * inFile;
	int32 fileValue;
	co_int32 testValue;

	inFile = fopen(FileName, "r");
	if ( inFile == NULL ) {
		fprintf(stderr, "Error opening input file %s\n", FileName);
		c = getc(stdin);
		exit(-1);
	}
	// Now read and write the waveform data...
	co_stream_open(StreamIN, O_WRONLY, INT_TYPE(STREAMWIDTH));
	IF_SIM(cosim_logwindow_write(log, "Sending test data...\n");)
	while (1) {
		if (fscanf(inFile,"%d",&fileValue) < 1){break;}
		testValue = fileValue;
		co_stream_write(StreamIN, &testValue, sizeof(co_int32));
		IF_SIM(cosim_logwindow_fwrite(log, "Value: 0x%x\n", testValue);)
	}
	IF_SIM(cosim_logwindow_write(log, "Finished writing test data.\n");)
```

```
        co_stream_close(StreamIN);
        fclose(inFile);
}
```

Figura 3.10. Proceso software *Producer*.

Se observa que tras abrir el fichero de entrada y el *stream* de entrada al hardware (*StreamIN*), el proceso software *Producer* escribe los datos del fichero (definido en "Mi_proyecto.h" como "filter_in.dat") en el *stream*. En la Figura 3.11, se observa cómo lectura del *stream* de salida del hardware (*StreamOUT*) se realizará en el proceso software *Consumer*, que también se encarga por tanto, de la escritura del fichero de salida (definido en el fichero cabecera como "filter_out.dat").

```
void Consumer(co_stream StreamOUT)
{
        co_int32 testValue;
        unsigned int count = 0;
        const char * FileName = OUTPUT_FILE;
        FILE * outFile;
        IF_SIM(cosim_logwindow log = cosim_logwindow_create("Consumer");)

        outFile = fopen(FileName, "w");
        if ( outFile == NULL ) {
                fprintf(stderr, "Error opening output file %s\n", FileName);
                exit(-1);
        }

        co_stream_open(StreamOUT, O_RDONLY, INT_TYPE(STREAMWIDTH));

        IF_SIM(cosim_logwindow_write(log, "Consumer reading data...\n");)
        while ( co_stream_read(StreamOUT, &testValue, sizeof(co_int32)) == co_err_none ) {
                fprintf(outFile, "%d\n", testValue);
                IF_SIM(cosim_logwindow_fwrite(log, "Value: 0x%x\n", testValue);)
                printf("Filtered value: 0x%x\n", testValue);
                count++;
        }
        IF_SIM(cosim_logwindow_fwrite(log, "Consumer read %d filtered data values\n", count);)
        co_stream_close(StreamOUT);
        fclose(outFile);
}
```

Figura 3.11. Proceso software *Consumer*.

Para terminar con la descripción del fichero ".sw" queda indicar que al final del mismo se implementa la función *main*, que hace uso de las funciones **co_initialize** y **co_execute** para inicializar y ejecutar los procesos en hilos separados en simulación.

```
int main(int argc, char *argv[])
{
        co_architecture my_arch;
        void *param = NULL;
        int c;
        printf("Impulse C is Copyright(c) 2003-2007 Impulse Accelerated Technologies, Inc.\n");

        my_arch = co_initialize(param);
        co_execute(my_arch);

        printf("\n\nApplication complete. Press the Enter key to continue.\n");
        c = getc(stdin);
        return(0);
}
```

Figura 3.12. Función *main* en "Mi_proyecto.sw".

De la misma forma, tras la creación del proyecto se dispondrá en el fichero “Mi_proyecto_hw.c”, de un código genérico para la descripción hardware, que servirá de patrón para implementaciones más complejas.

En este fichero se genera por tanto el esqueleto del proceso hardware, con los *streams* de entrada y salida definidos en el *wizard* durante la creación del proyecto. En la Figura 3.13 se observa la descripción de este proceso, en la que se indica el lugar donde debe incluirse el procesado de la aplicación, como es lógico, entre la lectura de los datos recibidos de la parte software, y el envío de los datos resultantes a la misma.

```
void ProcesoHardware(co_stream StreamIN, co_stream StreamOUT)
{
        co_int32 nSample;
        IF_SIM(int samplesread; int sampleswritten;)

        IF_SIM(cosim_logwindow log;)
        IF_SIM(log = cosim_logwindow_create("ProcesoHardware");)

        do {    // Hardware processes run forever
                IF_SIM(samplesread=0; sampleswritten=0;)

                co_stream_open(StreamIN, O_RDONLY, INT_TYPE(STREAMWIDTH));
                co_stream_open(StreamOUT, O_WRONLY, INT_TYPE(STREAMWIDTH));

                // Read values from the stream
                while ( co_stream_read(StreamIN, &nSample, sizeof(co_int32)) == co_err_none ) {
                #pragma CO PIPELINE
                        IF_SIM(samplesread++;)

                        // Sample is now in variable nSample.
                        // Add your processing code here.
                                nSample=nSample*2;

                        co_stream_write(StreamOUT, &nSample, sizeof(co_int32));
                        IF_SIM(sampleswritten++;)
                }
                co_stream_close(StreamIN);
                co_stream_close(StreamOUT);
                IF_SIM(cosim_logwindow_fwrite(log,
                        "Closing filter process, samples read: %d, samples written: %d\n",
                        samplesread, sampleswritten);)

                IF_SIM(break;) // Only run once for desktop simulation
        } while(1);
}
```

Figura 3.13. Proceso hardware.

A modo de ejemplo, se ha introducido una sentencia de procesado (destacada en el código anterior), que realizará la multiplicación por dos de cada dato. Así, la parte software recibirá el valor de cada dato enviado multiplicado por dos y escribirá el fichero de salida.

En el código anterior también aparecen sentencias “**IF_SIM()**”, que serán útiles bajo simulación, e ignoradas en la generación del hardware.

Las funciones del tipo **cosim_logwindow** utilizadas tanto en el fichero “.sw” como en el fichero “.hw” se emplean para mostrar indicaciones de depuración a través de la *Application Monitor*, como se indicará en la siguiente sección que tratará las herramientas de simulación.

Al final del fichero “Mi_proyecto_hw.c”, se implementa también la función de configuración donde se define el esqueleto completo de la arquitectura (ver Figura 3.14). Es aquí donde se declaran y crean todos los elementos que componen la arquitectura creada, indicando los parámetros de entrada de los mismos, así como la forma en que están comunicados.

```
void config_Mi_proyecto(void *arg)
{
        co_stream StreamIN;
        co_stream StreamOUT;

        co_process ProcesoHardware_process;
        co_process producer_process;
        co_process consumer_process;

        IF_SIM(cosim_logwindow_init();)

        StreamIN = co_stream_create("StreamIN", INT_TYPE(STREAMWIDTH), STREAMDEPTH);
        StreamOUT = co_stream_create("StreamOUT", INT_TYPE(STREAMWIDTH), STREAMDEPTH);

        producer_process = co_process_create("Producer", (co_function)Producer,1,StreamIN);

        ProcesoHardware_process = co_process_create("ProcesoHardware",
        (co_function)ProcesoHardware,2,StreamIN, StreamOUT);

        consumer_process = co_process_create("Consumer",(co_function)Consumer,1,StreamOUT);

        co_process_config(ProcesoHardware_process, co_loc, "pe0");
}
```

Figura 3.14. Función de configuración.

En el código anterior, para la creación de los *stream* (implementados como buffers FIFO) se ha llamado a la función:

```
co_stream co_stream_create(const char *name, co_type type, int numelements);
```

Así, tamaño del buffer vendrá determinado por el valor del argumento **numelements,** que indica la profundidad del *stream* (*STREAMDEPTH*), multiplicado por el tamaño (en bytes) del tipo de dato especificado en el argumento **type** (esto es, el ancho del *stream* en bytes).

De forma similar, para crear cada proceso se llama a la función:

```
co_process co_process_create(const char *name, co_function run, int argc, ...);
```

En ésta es importante resaltar, que en el argumento **argc**, se indicará el numero de puertos (*streams*, señales, memorias, registros y parámetros), seguido por la lista de los mismos, cuyo orden debe respetarse en la implementación del proceso.

En la última línea (sobre fondo azul) se indica a la función de configuración, a través del atributo **co_loc**, la localización hardware física del proceso especificado. De esta forma, asignando el valor "PE0", se indica que el mismo será compilado en la forma de ficheros de salida HDL.

Por último, en el fichero “.hw” se le indica al compilador que la función de configuración es la definida como *config_Mi_proyecto*, utilizando la siguiente función:

```
co_architecture co_initialize(int param)
{
return(co_architecture_create("Mi_proyecto_arch","Generic",config_Mi_proyecto,(void *)param));
}
```

- **Herramientas de Simulación**

La simulación es un elemento importante del proceso de desarrollo, cuyo objetivo es verificar que la aplicación y sus componentes tienen el funcionamiento esperado, en términos de exactitud y de rendimiento.

Simulación software

Para realizar la simulación funcional se necesitan definir unos estímulos de test y el software necesario para realizar el test. Aquí el desarrollador se puede aprovechar de las ventajas del lenguaje ANSI C para crear de forma fácil un *testbench* tan complejo y completo como quiera. La simulación funcional del sistema sobre la máquina donde se desarrolla la aplicación se denomina simulación de escritorio (*desktop simulation*) o simulación software. En esta simulación funcional, el paralelismo y el funcionamiento de los *streams* implementados se implementan mediante hilos (*threads*) y las bibliotecas asociadas. Esto quiere decir que las relaciones temporales y el comportamiento en paralelo entre procesos puede parecerse poco o nada a lo que se tenga posteriormente en el hardware.

La simulador software se realiza mediante la ejecución del programa *make* (explícitamente o a través de CoDeveloper), que invoca al compilador y enlazador de C, y genera un ejecutable compatible con el sistema operativo.

Así, tras ejecutar (*Project → Launch Simulation Executable*) el código del ejemplo considerado en las secciones previas, aparecen por consola los resultados de la aplicación indicados mediante llamadas a *printf()*. En el ejemplo descrito, también pueden verse los valores recibidos en el fichero "filter_out.dat" generado.

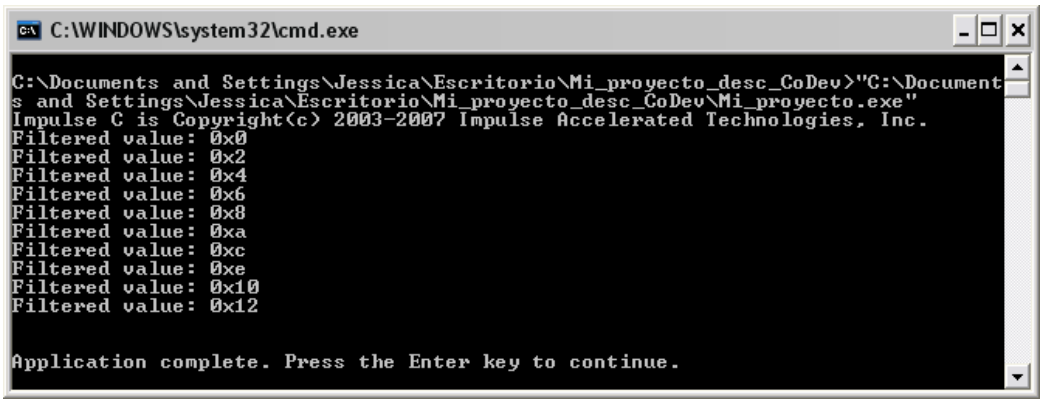

Figura 3.15. Resultado mostrado por consola.

La herramienta *Application Monitor* permite al desarrollador observar el comportamiento de la aplicación durante la simulación software, visualizando la secuencia temporal en la que se generan los mensajes y los datos en los *streams*. La herramienta permite crear una ventana de *logging* por cada uno de los procesos, hardware o software, por separado.

Para utilizar la herramienta *Application Monitor* se debe realizar una llamada a la función **cosim_logwindow_init** en la función de configuración de la aplicación *ImpulseC*.

Para visualizar un proceso (Figura 3.16), se debe realizar una llamada a **cosim_log_window_create** dentro del mismo. Las funciones **cosim_logwindow_fwrite(logwindow, formatstring, arg1, arg2...)** y **cosim_logwindow_write(logwindow, str)** permiten mostrar mensajes de depuración en dicha ventana durante la simulación.

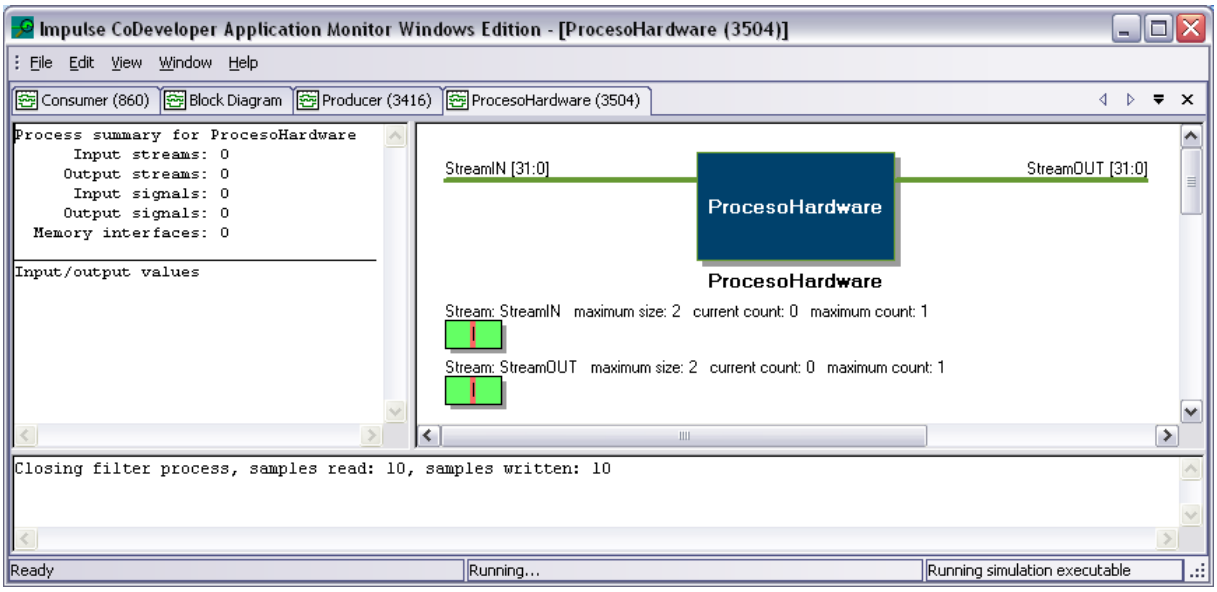

Figura 3.16. Ventana de simulación del proceso *ProcesoHardware*.

Esta *Application Monitor*, aparecerá al inicial la simulación (*Project→ Launch Simulation Executable)* si se ha seleccionado previamente la opción *Launch Application Monitor* en la pestaña *Simulate* del menú *Project→ Options*.

En la Figura 3.17 se muestra que en la *Application Monitor* aparece una pestaña para cada uno de los procesos para los que se ha creado una ventana de simulación, y otra pestaña donde se muestra el diagrama de bloques de la aplicación, útil para comprobar si se han definido correctamente las comunicaciones en la función de configuración.

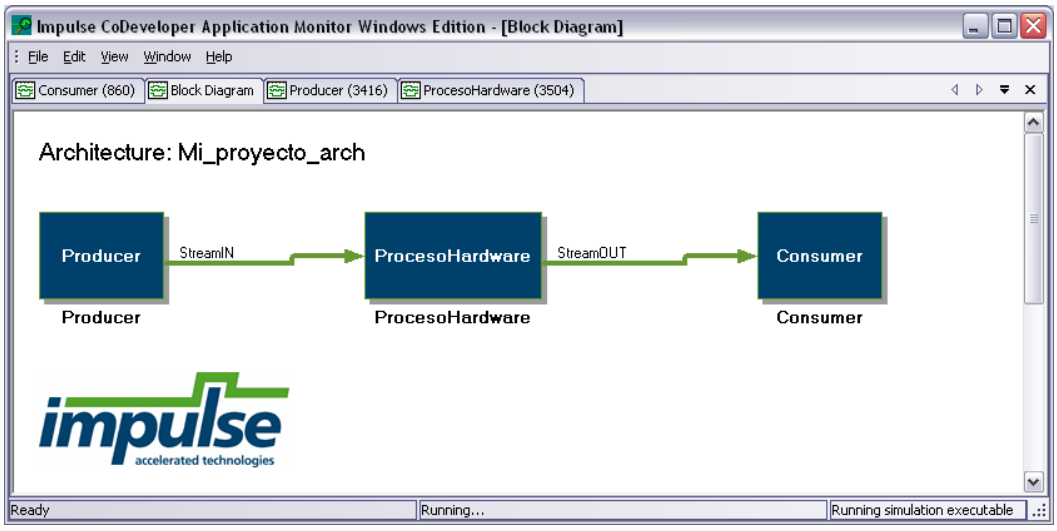

Figura 3.17. Diagrama de bloques de la aplicación.

Simulación hardware

Las siguientes herramientas que se van presentar (*Stage Master Explorer* y *Stage Master Debugger*), requieren que se haya realizado la generación hardware, para ello seleccionar: *Project → Generate HDL*. En la Figura 3.18 aparecen los resultados generados para el ejemplo considerado.

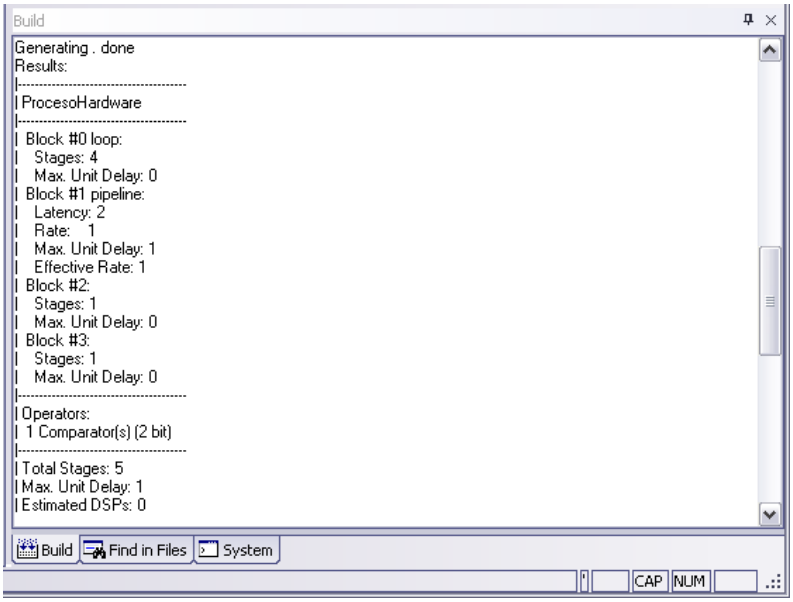

Figura 3.18. Resultados de la *Generación Hardware*.

Una vez hecho esto, se puede acceder al *Stage Master Explorer* seleccionando *Tool→ Stage Master Explorer* y abriendo el fichero .xic proporcionado tras la generación hardware.

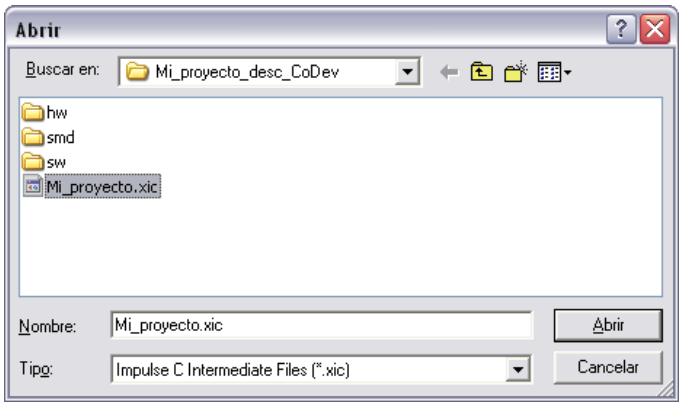

Figura 3.19. Fichero .xic generado.

El *Stage Master Explorer* es una herramienta gráfica que permite analizar la aplicación determinando proceso a proceso, cómo ha sido de efectivo el compilador al paralelizar el código C. La herramienta presenta información sobre cada uno de los bloques de código C, y es capaz de generar un gráfico de flujo de datos que muestra las relaciones entre las diferentes operaciones generadas en la compilación.

Dicha herramienta ofrece un análisis detallado sentencia a sentencia de los resultados de la optimización de la paralelización. En cada línea expandida de código C informa de la etapa en la que se ejecuta, y ofrece un resumen para cada bloque de código donde aparecen retardos estimados, latencias de los bucles y tasas efectivas de pipeline (ver Figura 3.20).

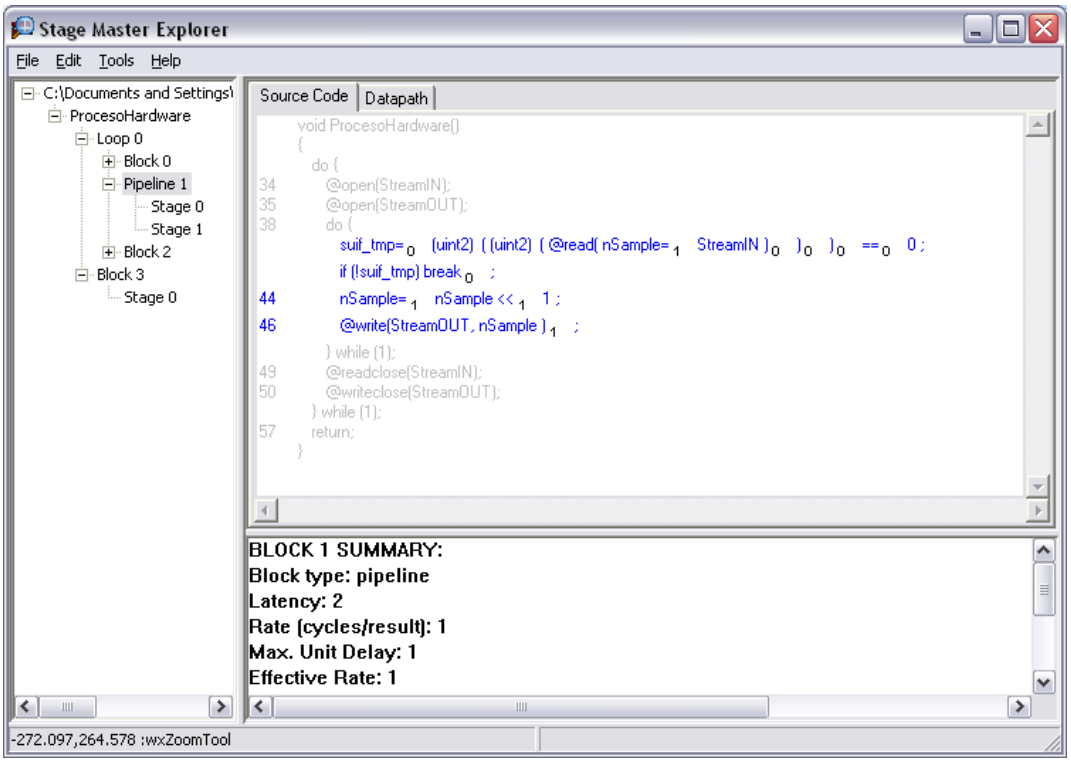

Figura 3.20. Visualización de las etapas requeridas en el código fuente.

Para un análisis más detallado de la paralelización generada del hardware, la pestaña *Datapath* muestra un gráfico completo que representa la estructura paralelizada. En el gráfico se puede observar, etapa a etapa, cómo se han descompuesto las sentencias de código en operaciones en paralelo.

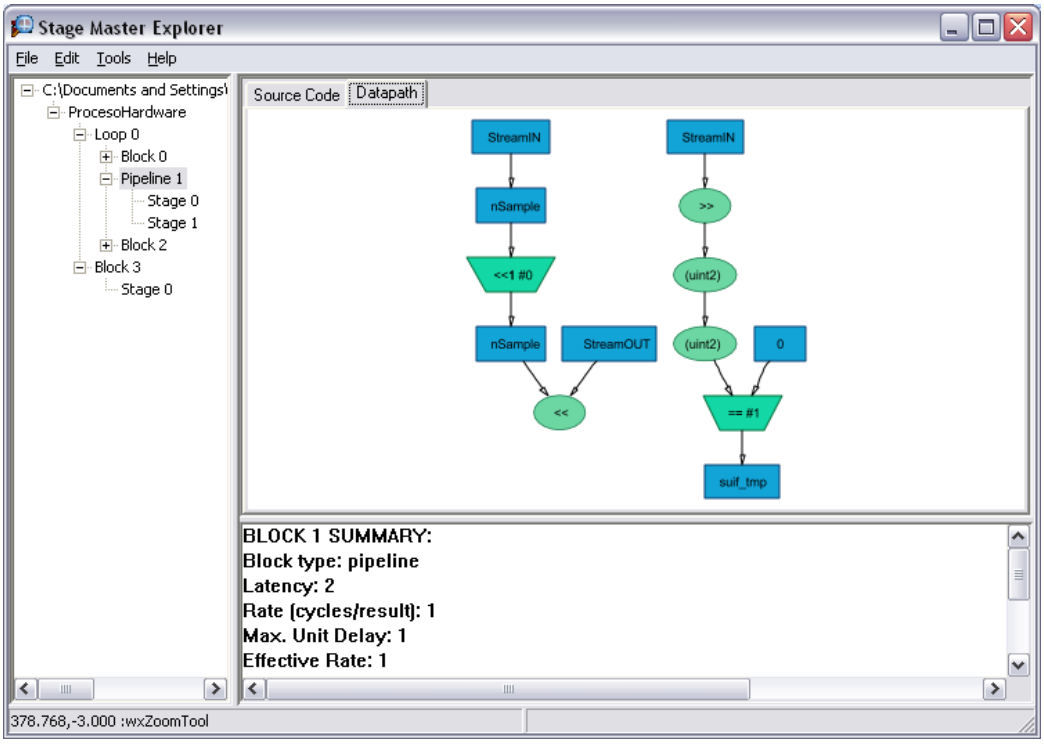

Figura 3.21. Visualización del camino de datos.

En la Figura 3.22 se muestra como al pinchar sobre cada una de las etapas, aparecen activadas en el flujo de datos, las operaciones realizadas en la etapa seleccionada.

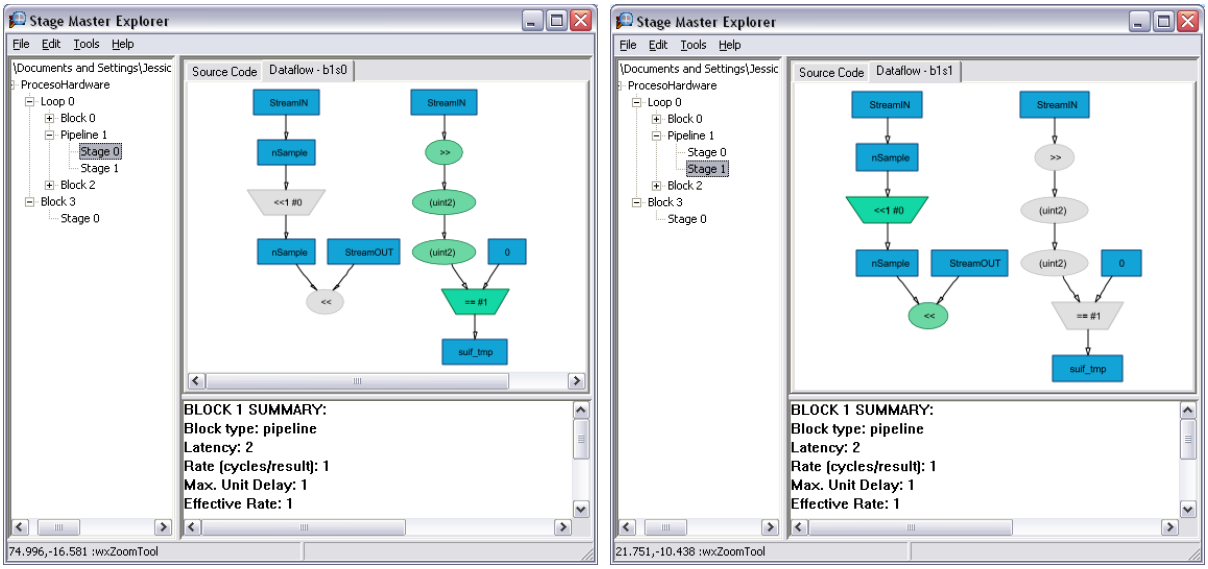

Figura 3.22. Visualización del flujo de datos en la etapa 0 (izquierda) y en la etapa 1 (derecha).

Por último, se verá la herramienta de simulación hardware, *Stage Master Debugger* (accesible a través del menú *Project → Build/Launch Hardware Simulation Executable),* que también se ejecuta sobre la máquina de escritorio. Sin embargo, el funcionamiento hardware de la aplicación se ejecuta ciclo por ciclo comportándose como si estuviera corriendo en la FPGA. Los procesos de la parte software siguen ejecutándose con hilos.

En la Figura 3.23 se muestra el entorno de este *Debugger*, donde para cada ciclo se muestra resaltado las instrucciones ejecutadas, así como es estado de los *streams* y etapa del *pipeline*. Puede observarse que también es posible visualizar el valor de variables en cada ciclo, para lo que se debe pinchar con el botón derecho del ratón sobre la misma, y seleccionar la opción *Watch*.

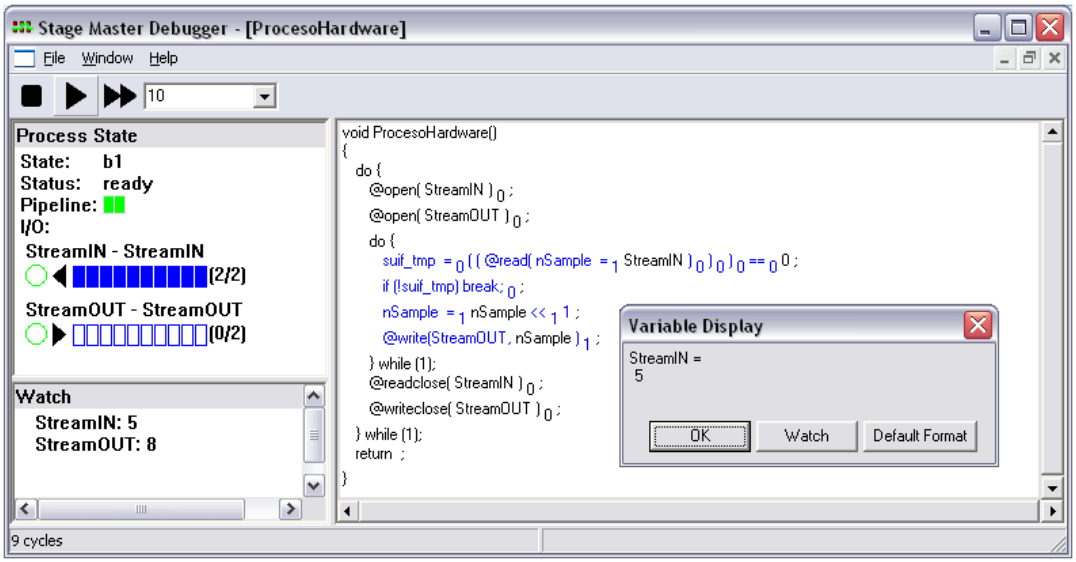

Figura 3.23. Simulación hardware de la aplicación.

3.2. El Nodo de Supercomputación Híbrido DS1002 de DRC

Para la ejecución del algoritmo se ha utilizado un nodo de supercomputación híbrido (**HPRC**, *High Performance Reconfigurable Computer*) modelo DS1002, de la empresa DRC [**15**], que permite la co-ejecución software-hardware de un algoritmo sobre procesadores convencionales acelerados mediante coprocesadores hardware desarrollados sobre FPGA.

Este sistema solventa la mitigación de la aceleración hardware provocada por los largos tiempos de transferencia de datos, en aplicaciones que utilizan una plataforma *FPGA* comunicada con una estación de trabajo *host* a través de una interfaz estándar (como pudiera ser *USB*, *PCI* o *Ethernet*), especialmente si se requiere un gran flujo de datos entre la parte hardware y la parte software de la arquitectura.

El sistema se compone de una estación de trabajo PC basada en *Linux (Kubuntu 6.06 LTS)* estándar dotada de una unidad de proceso reconfigurable (*RPU,* del inglés *Reconfigurable Processor Unit*). Como se muestra en la Figura 3.24, la placa base incluye un microprocesador *AMD™ Opteron* modelo 275 y una *RPU110-L200* basada en una FPGA *Virtex-4 LX200* de *Xilinx*, y dotada con 2GB de DDR2 SDRAM y 128 MB de RLDRAM de baja latencia, entre otros periféricos de comunicación.

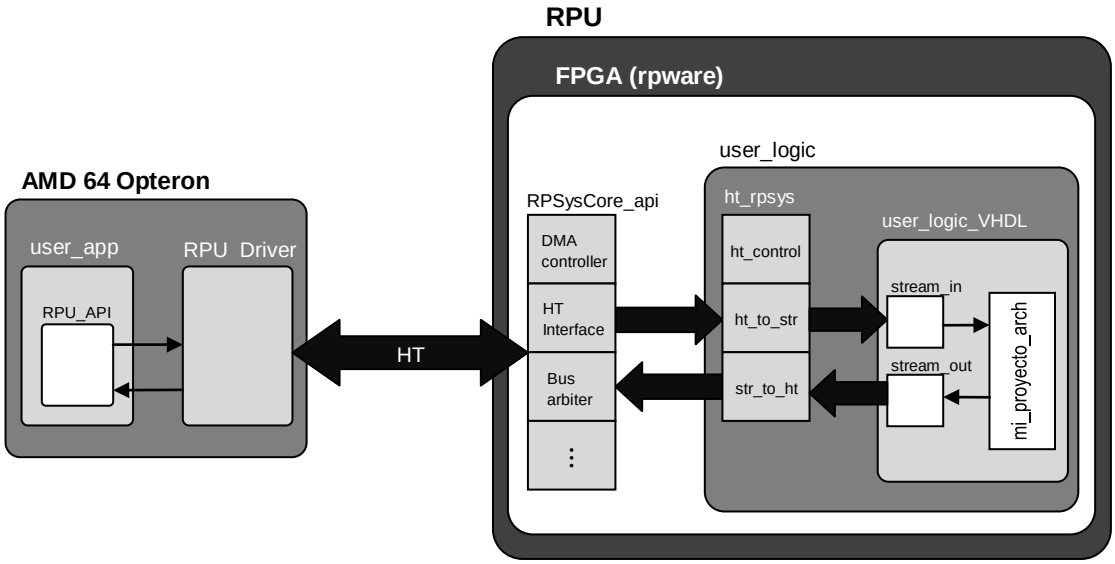

Figura 3.24. Arquitectura hardware interna del DS1002. La plataforma consiste en un microprocesador Opteron, el cual ejecuta los procesos software, y una RPU, que ejecuta los procesos hardware.

La característica más importante de este sistema, como ya se ha comentado, es la solución que proporciona al problema de la comunicación del microprocesador y la *RPU*. Así, dicha comunicación se realiza a través de tres buses de alta velocidad de tipo *HyperTransport* ™ (en lo sucesivo *HT*). Actualmente la interfaz *HT* está limitada a *8bits x 400Mhz* (doble tasa de transmisión), proporcionando un *throughput* teórico de 800MB/s en cada dirección (esto es, un total *1.6GB/s*); lo que se traduce en un ancho de banda total de *4.8GB/s*. Cabe indicar que, para arquitecturas que quieran beneficiarse del soporte que proporciona CoDeveloper para esta plataforma (en forma de drivers de comunicaciones software y hardware), como la que se implementará en este PFC, Impulse impone la restricción de un único bus de comunicación bidireccional. Un resumen de las características del modelo *DS1002* puede verse en la tabla presente en la Tabla 3.3.

	RPU110-L200
FPGA	
Xilinx Virtex™-4	LX 200
Number of LUTs	200,448
RPU Hardware OS use of LUTs(%): *max = HT x3* *plus DDR2 mem ctrl x2*	Min 14,400 (7%) Max 20,000 (10%)
Memory (BRAM w/ECC)	336x18kbits
Physical/Mechanical	
Socket	940 ZIF
Dimensions (mm)	78.7 x 96.5 x 27.9
Power dissipation	10-40 W
HT interface	Bus 0, 1, 2
RPU RLDRAM	128 MB
RPU DDR2 memory	2 x .5 GB
Performance	
HT bus *Per connection* *aggregate*	400MHz x 16bits 3.2 GB/sec 9.6 GB/sec
Memory (motherboard)	128 bit DDR 400 6.4 GB/sec
Memory (RRU DDR2) *Per connection* *aggregate*	16 bit DDR 400 800 MB/sec 1.6 GB/sec
Software/Firmware	
- *Linux Drivers* - *Linux RPU Manager* - *HT interface (s)* - *RPU API* - *RPU Hardware OS:* · *DDR1/DDR2 memory controller* · *RPU RLDRAM controller* · *HT interface(s)* · *RPU Hardware OS API*	

Tabla 3.3. Resumen de las características del modelo DS1002.

Capítulo 4

Procedimientos para la co-ejecución en el nodo de computación DRC

En este capítulo se describirá el procedimiento seguido para crear un nuevo proyecto en CoDeveloper que utilice la plataforma DRC, así como los pasos necesarios para conseguir, a partir del proyecto anterior, el archivo de programación de la RPU y programarla. De igual manera se indicarán las modificaciones pertinentes en la parte software del proyecto implementado en CoDeveloper, para su ejecución en el computador DRC.

4.1. Procedimiento para crear un proyecto de Impulse C que utilice la plataforma DRC

4.1.1. Crear un nuevo proyecto de CoDeveloper

La manera más sencilla de implementar una arquitectura válida para el uso del computador DRC, es generar un proyecto inicial con un único proceso software y un único proceso hardware. A lo largo del diseño podrán añadirse cuantos procesos sean convenientes, siendo lo habitual en este tipo de arquitecturas, gestionar la lectura y recepción de datos en un único proceso software, y realizar el procesado principal del flujo de datos en diversos procesos hardware.

El procedimiento para la creación de un nuevo proyecto comienza accediendo al menú *File* → *New Proyect* de CoDeveloper. Se selecciona la opción *"One-process testbench"* mostrada en la Figura 4.1, y una vez introducido el nombre del proyecto y su localización (importante utilizar un *path* sin espacios ni caracteres especiales) se pulsa en *OK.*

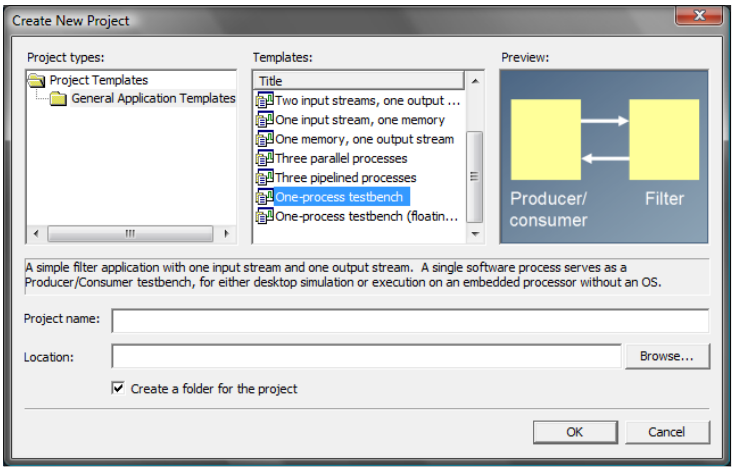

Figura 4.1. Crear un nuevo proyecto en CoDeveloper (I)

Es importante recordar la restricción impuesta por Impulse de un único *stream* bidireccional entre la parte software y la parte hardware, además de no permitir más de un proceso software, en el caso de proyectos que deban ser compilados para la arquitectura DS1002 de DRC, motivo por el que se ha escogido la opción *"One-process testbench"*.

En la siguiente ventana introducir el nombre del proceso hardware, el nombre del *stream* de entrada al hardware y el nombre del *stream* de salida del hardware. Igualmente, seleccionar el tamaño de los datos del *stream* y el tamaño del *buffer*. Pulsar *siguiente* y *siguiente* otra vez (Figura 4.2).

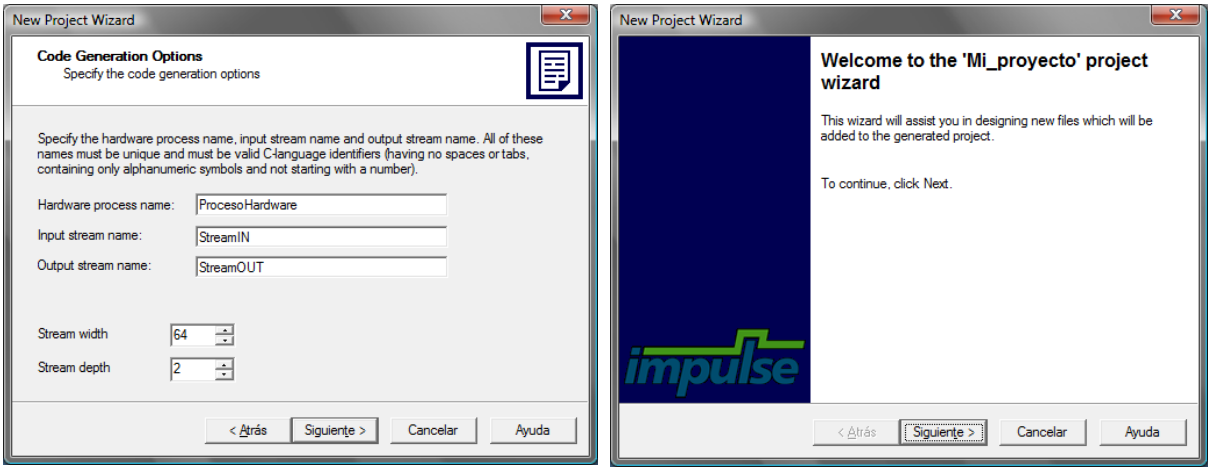

Figura 4.2. Crear un nuevo proyecto en CoDeveloper (II)

4.1.2. Definición de la estructura general de comunicación entre procesos.

Tras el apartado anterior, se deben obtener las descripciones en ANSI C para los procesos software y hardware del proyecto creado, así como un fichero cabecera con las definiciones para las características del *stream* descritas en el *wizard*, y los nombres de los ficheros de entrada/salida (Figura 4.3).

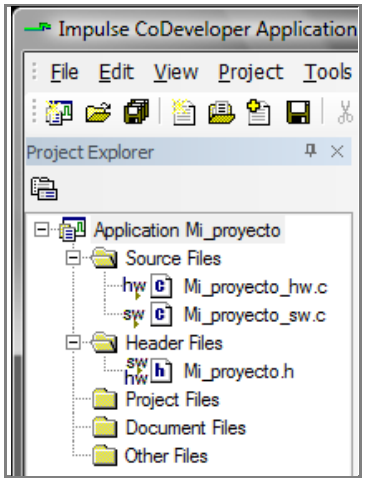

Figura 4.3. Estructura de archivos generada

En la Figura 4.4 se muestra parte del código generado automáticamente en el fichero "Mi_proyecto_sw.c", correspondiente al proceso software, donde se indica la estructura básica que debe respetarse para el correcto funcionamiento de las aplicaciones diseñadas. Se observa que tras

abrir los *streams*, el proceso software *ProducerConsumer* escribe los datos de un fichero (definido en en “Mi_proyecto.h” como “filter_in.dat”) en el *stream* de entrada al hardware (*StreamIN*), de manera no bloqueante. La lectura del *stream* de salida del hardware se realiza en dos partes: la primera lectura cada vez que escribe un dato (por eso lo de escritura no bloqueante) y después una lectura hasta vaciar el *buffer* del *stream*.

```
co_stream_open(StreamIN, O_WRONLY, INT_TYPE(STREAMWIDTH));
co_stream_open(StreamOUT, O_RDONLY, INT_TYPE(STREAMWIDTH));

// Read and write to/from the hardware process
IF_SIM(cosim_logwindow_write(log, "Sending/receiving...\n");)
while ( fscanf(inFile,"%d",&fileValue) == 1 ) {
        outValue = fileValue;
        co_stream_write(StreamIN, &outValue, sizeof(co_int64));
        IF_SIM(cosim_logwindow_fwrite(log, "Sent value: 0x%x\n", outValue);)
        if ( co_stream_read_nb(StreamOUT, &inValue, sizeof(co_int64)) ) {
                fprintf(outFile, "%d\n", inValue);
                IF_SIM(cosim_logwindow_fwrite(log, "Rcvd value: 0x%x\n", inValue);)
                inCount++;
        }
        outCount++;
}
fclose(inFile);
co_stream_close(StreamIN);
IF_SIM(cosim_logwindow_write(log, "Finished sending.\n");)

while ( co_stream_read(StreamOUT, &inValue, sizeof(co_int64)) == co_err_none ) {
        fprintf(outFile, "%d\n", inValue);
        IF_SIM(cosim_logwindow_fwrite(log, "Rcvd value: 0x%x\n", inValue);)
        inCount++;
}
fclose(outFile);
co_stream_close(StreamOUT);
```

Figura 4.4. Envío y recepción en el proceso software.

Como la lectura y escritura se hace en el mismo proceso *ProducerConsumer*, es importante, que tras escribir un dato, la lectura sea no bloqueante. De no ser así, lo habitual será que ocurra lo que se conoce como un *deadlock*[4]. Así, si el hardware necesita más datos de entrada para devolver un resultado, el proceso software no se quedará esperando la lectura de un dato que nunca llega, ya que puede enviar los datos restantes necesarios. Se debe aclarar que la posibilidad de que ocurran *deadlocks* es debida a la unificación en un único proceso software del "Productor" y del "Consumidor", que se encarga tanto de enviar los datos al hardware, como de recibir los resultados de éste. Cuando se utiliza un proceso “Productor” para enviar los datos, y un proceso “Consumidor” para la recepción, la concurrencia entre procesos de ejecución software sigue el paradigma de programación con hilos.[5]

Al igual que en la sección del capítulo 3, donde se introdujo la estructura genérica de un código para una arquitectura mixta (hw/sw) en CoDeveloper, en el fichero generado “Mi_proyecto_hw.c” (Figura 4.5), para el caso de este ejemplo, se ha introducido una única sentencia de procesado, que realiza la multiplicación por dos de cada dato procedente del proceso sofware.

[4] Un *deadlock* ocurre cuando un proceso es incapaz de continuar con sus operaciones hasta que otro proceso haya completado sus tareas y escrito datos en sus salidas. Si los dos procesos son mutuamente dependientes o dependen de algún otro proceso bloqueado, el sistema puede llegar a detenerse.

[5] Se está estudiando la posibilidad de eliminar la restricción de un único proceso (hilo) software para la plataforma DRC en una próxima versión de esta aplicación, cuya implementación queda fuera de este Proyecto Fin de Carrera.

```
void ProcesoHardware(co_stream StreamIN, co_stream StreamOUT)
{
	.
	.
	.
	do {	// Hardware processes run forever
		IF_SIM(samplesread=0; sampleswritten=0;)

		co_stream_open(StreamIN, O_RDONLY, INT_TYPE(STREAMWIDTH));
		co_stream_open(StreamOUT, O_WRONLY, INT_TYPE(STREAMWIDTH));
		// Read values from the stream
		while ( co_stream_read(StreamIN, &nSample, sizeof(co_int64)) == co_err_none ) {
		#pragma CO PIPELINE
			IF_SIM(samplesread++;)

			// Sample is now in variable nSample.
			// Add your processing code here.
				nSample=nSample*2;

			co_stream_write(StreamOUT, &nSample, sizeof(co_int64));
			IF_SIM(sampleswritten++;)
		}
		co_stream_close(StreamIN);
		co_stream_close(StreamOUT);

	} while(1);
}
```

Figura 4.5. Proceso hardware.

Al final del fichero “Mi_proyecto_hw.c”, se implementa la función de configuración donde se define el esqueleto completo de la arquitectura (ver Figura 4.6). Como ya se explicó, en ella se declaran y crean los elementos que componen dicha arquitectura, así como la forma en que están comunicados. Se observa que en este caso sólo hay dos procesos, uno software y otro hardware conectados a través de los *streams,* definidos como *StreamIN* y *StreamOUT,* en el *wizard* de la aplicación.

```
void config_Mi_proyecto(void *arg)
{
	co_stream StreamIN;
	co_stream StreamOUT;

	co_process ProcesoHardware_process;
	co_process testbench_process;

	IF_SIM(cosim_logwindow_init();)

	StreamIN = co_stream_create("StreamIN", INT_TYPE(STREAMWIDTH), STREAMDEPTH);
	StreamOUT = co_stream_create("StreamOUT", INT_TYPE(STREAMWIDTH), STREAMDEPTH);

	testbench_process = co_process_create(
		"ProducerConsumer",(co_function)ProducerConsumer,2,StreamIN,StreamOUT);

	ProcesoHardware_process = co_process_create(
		"ProcesoHardware",(co_function)ProcesoHardware,2,StreamIN,StreamOUT);

	co_process_config(ProcesoHardware_process, co_loc, "pe0");
}
```

Figura 4.6. Función de configuración.

Si se ejecuta, se observará que el fichero de salida generado es el mismo que el obtenido en el ejemplo considerado en el capítulo 3, ya que aunque la arquitectura es distinta, se ha realizado el mismo procesado.

4.1.3. Selección de la plataforma de desarrollo.

A continuación se muestran las opciones de CoDeveloper (a través del menú *Proyect* → *Options*) que deben ser seleccionadas para una correcta simulación y ejecución del proyecto.

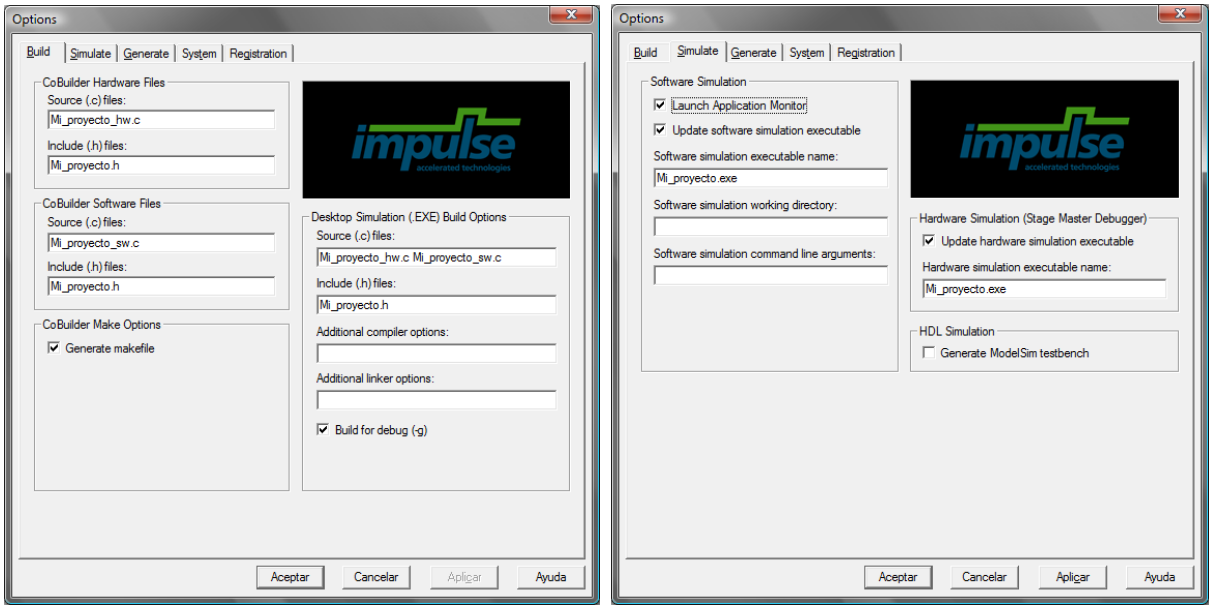

Figura 4.7. Opciones de las pestañas Build (izquierda) y Simulate (derecha) de CoDeveloper.

Es importante no olvidar seleccionar la plataforma de desarrollo, para que a la hora de exportar los archivos en *.vhd* y *.v* estos sean válidos para el computador DRC. Para ello, en la pestaña *Generate* en el campo *"Platform Support Package"* se debe seleccionar *"DRC RPU110-L200 (VHDL)"*; tal y como se ilustra en la Figura 4.8.

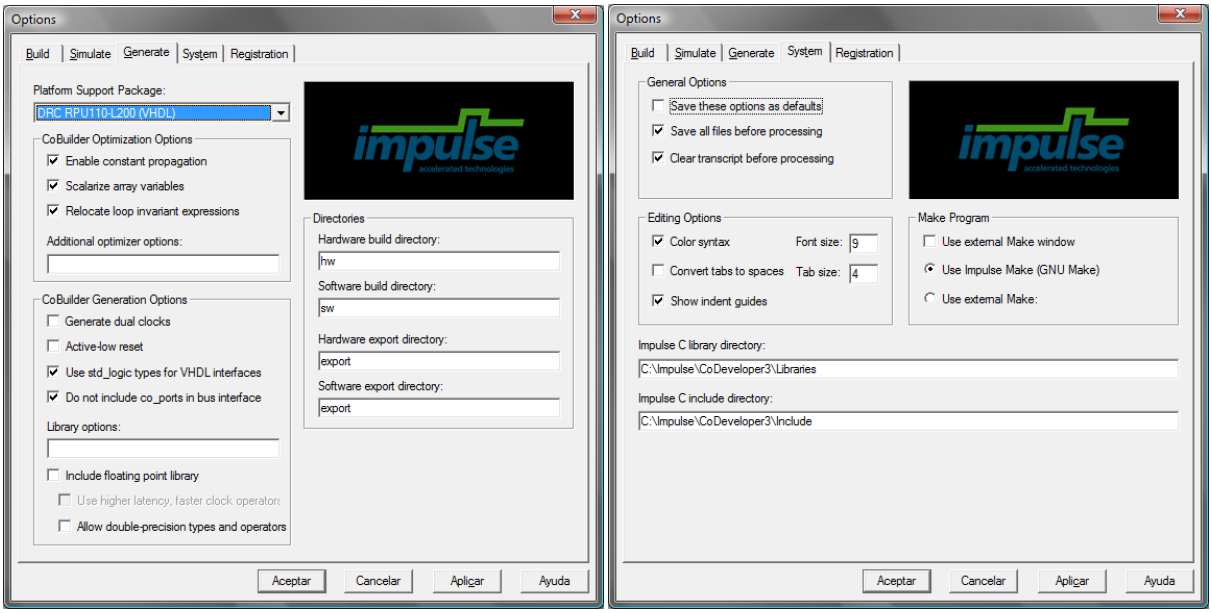

Figura 4.8. Opciones de las pestañas Generate (izquierda) y System (derecha) de CoDeveloper.

4.1.4. Exportar software y hardware generados

Una vez configuradas las opciones necesarias se debe generar el hardware y exportar el software y hardware generado. La Figura 4.9 muestra los iconos que facilitan las operaciones nombradas.

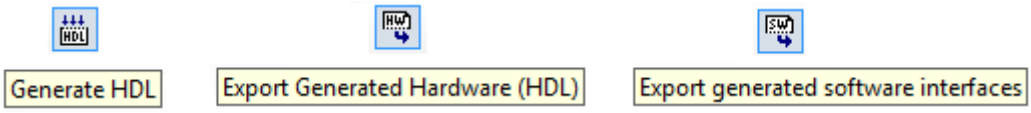

Figura 4.9. De izquierda a derecha, iconos para generar hardware, exportar hardware y exportar software

Si todo ha ido bien, en la carpeta *export* se habrán generado todos los archivos necesarios para generar el hardware y el software, cuyo procedimiento se detallará en el apartado 4.2.

4.2. Generación del archivo de programación de la RPU

En este apartado se expondrán los fundamentos y procedimientos seguidos para conseguir, a partir de un proyecto en CoDeveloper, el archivo *.bit* para volcar en la FPGA de la RPU del computador DRC. Estas indicaciones, pretenden proporcionar una guía útil para proyectos futuros que empleen las herramientas de Impulse para la generación de aplicaciones hardware/software ejecutables en la plataforma de DRC.

Una vez terminada la fase de descripción y compilación de la aplicación en CoDeveloper, si se han seguido todos los pasos indicados en el apartado 4.1 y el desarrollo ha sido exitoso, dentro de la carpeta *export* del proyecto deben aparecer los siguientes cuatro archivos: "*Mi_proyecto_comp.vhd", "Mi_proyecto_top.vhd"*, "*user_logic.v"*, "*ht_rpsys.v"*.

Los archivos *Mi_proyecto_comp.vhd* y *Mi_proyecto_top.vhd* describen los procesos hardware que se han programado en Impulse. Con estos archivos se creará un proyecto llamado *"user_logicVHDL"* en la herramienta ISE de Xilinx, que al sintetizarse generará el *netlist* correspondiente al hardware de la aplicación.

A continuación se deberá generar un segundo proyecto en ISE denominado "*user_logic*", en el que se añadirán los archivos "*user_logic.v"*, "*ht_rpsys.v"*, proporcionados por ImpulseC. Este proyecto actúa de interfaz entre la lógica de usuario y la interfaz genérica proporcionada por DRC para la comunicación entre el hardware y el software a través del bus *HyperTransport*. Es decir, únicamente realiza las conversiones necesarias a los puertos del sistema *user_logicVHDL* (los típicos *stream* de Impulse C)[6] para hacerlos coincidir con los definidos en la interfaz proporcionada en las librerías del DRC. Para ello, en el fichero "*ht_rpsys.v*" se incluyen 3 módulos principales: *ht_control*, *ht_to_stream* y *stream_to_ht*. Por otro lado, la interfaz proporcionada por DRC para la comunicación a través del bus HT se compone de dos partes. Para la parte software proporciona una librería compartida en C que implementa el RPU Driver, y un archivo de cabecera ("RpuC.h") donde se define la RPU API con las funciones que deberemos llamar para comunicarnos con el hardware. La interfaz hardware, denominada RPSysCore, implementa la lógica de comunicaciones de bajo nivel y proporciona una interfaz simplificada para que la user_logic pueda realizar lecturas y escrituras en el bus HT, soportando transferencias unitarias, en modo ráfaga, etc. El RPSysCore se proporciona en un archivo denominado "*rpware-ht0.ngc"* en formato *netlist* (no en un HDL) para la protección de su propiedad intelectual. Más información sodre la interfaz sw y hw proporcionada por DRC puede encontrarse en los capítulos 4 y 5 respectivamente del manual de referencia [**15**].

Por tanto, se debe crear un tercer proyecto en ISE llamado "*rpware*", que incluya el archivo "*rpware-ht0.ngc"*. Esta interfaz proporcionada, incluye fundamentalmente el controlador de acceso al bus *HT* y los buffers de E/S correspondientes. Esto evita tener que lidiar con la complejidad de los protocolos de acceso al bus HT, en la creación del proyecto anterior "*user_logic"*, que se limita únicamente a implementar una etapa de adaptación entre la comunicación mediante streams que se utiliza en ImpulseC y la interfaz para el bus HT que proporciona DRC.

Así, este proyecto "*rpware"* incluirá en la FPGA la interfaz de comunicaciones con la parte software residente en el microprocesador, y será el que generará el archivo "*rpware.bit"* de programación de la FPGA, para lo cual, debe incluir las netlist de los proyectos anteriores antes de sintetizarse. De esta forma, cada proyecto englobará al anterior, sirviendo de interfaz con el siguiente. Esta estructura lógica puede apreciarse gráficamente en la Figura 4.10.

[6] En el flujo de trabajo normal del DRC (si no se crea el proyecto en Impulse C, sino directamente en lenguaje de bajo nivel) sería necesario diseñar en Verilog una entity de nombre user_logic con los puertos y la lógica adecuados para que el hardware diseñado pueda conectarse a la interfaz proporcionada por DRC.

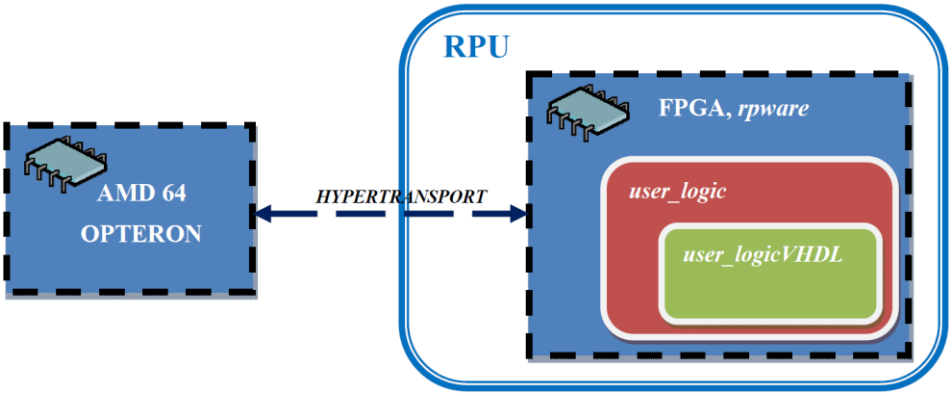

Figura 4.10. Esquema de la estructura lógica de proyectos.

A continuación se describen paso a paso los procedimientos a seguir para la creación de los tres proyectos en ISE y la síntesis e implementación del hardware correspondiente.

Es importante que los proyectos tengan los nombres que se usan en la descripción del procedimiento que sigue a continuación. La estructura de carpetas, así como el nombre del proyecto (que en este PFC se ha denominado *Mi_proyecto*) no influyen en la correcta implementación del hardware.

4.2.1. Sintetizar en ISE los procesos hardware generados por CoDeveloper

Para la creación del proyecto ISE que contiene el hardware de la aplicación diseñada se deben seguir los siguientes pasos:

1. Crear un nuevo proyecto denominado "user_logicVHDL" y localizarlo en "export\user_logicVHDL" (Figura 4.11). Añadir al proyecto todos los archivos .vhd (Figura 4.12) generados en el directorio "export"; estos son "*Mi_Proyecto_comp.vhd*" y "*Mi_proyecto_top.vhd*". Todas las asociaciones de unidades de diseño se deben poner a "*All*", que incluye síntesis/implementación y simulación, si el ISE no las pone automáticamente.

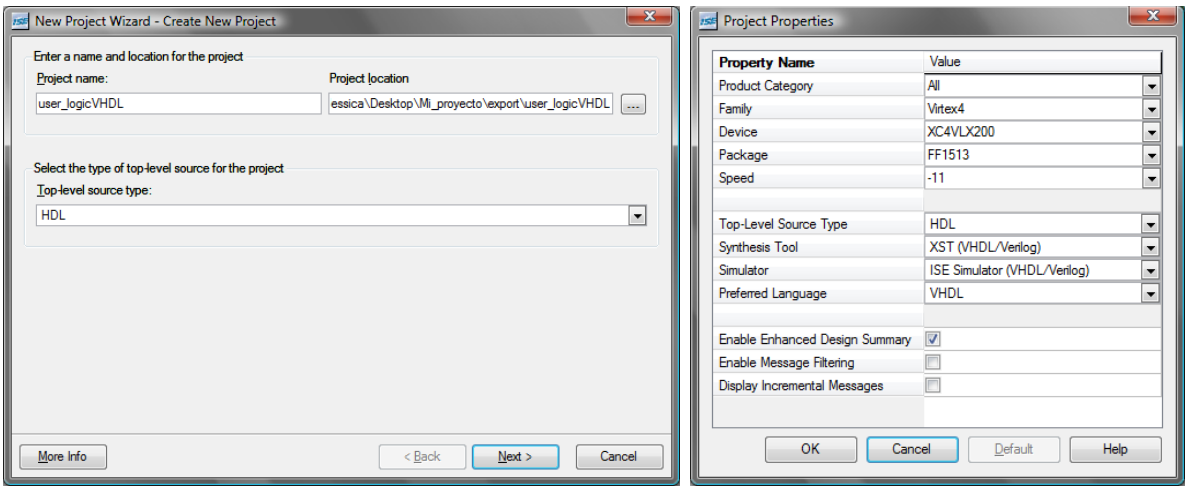

Figura 4.11. Opciones de configuración para la creación del proyecto.

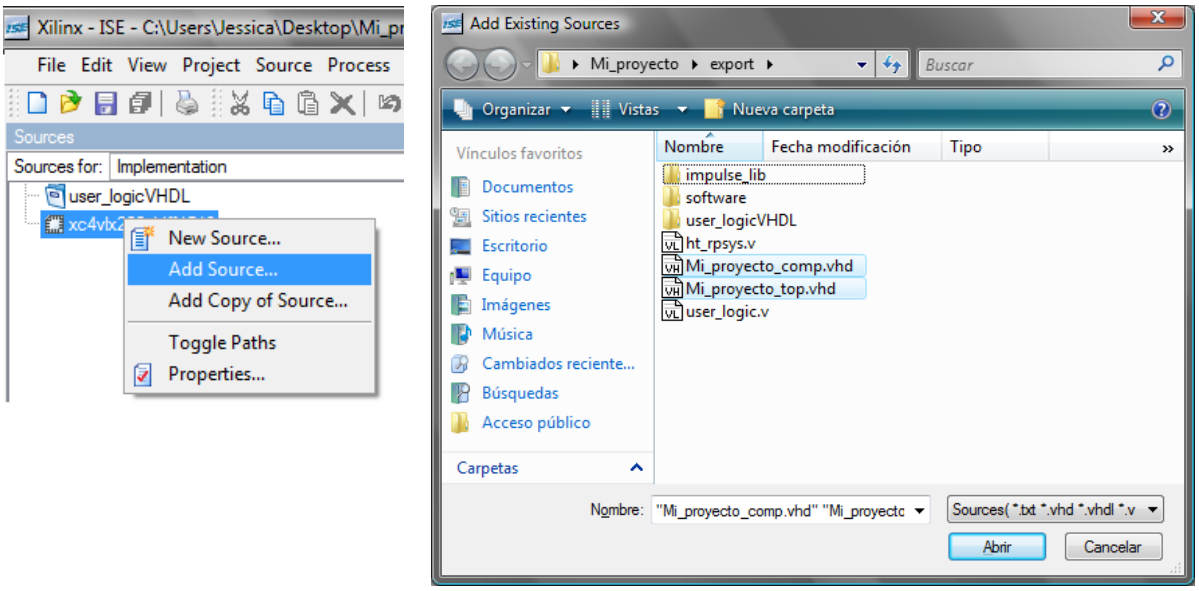

Figura 4.12. Archivos .vhd generados por CoDeveloper.

2. Añadir una librería VHDL al proyecto denominada “impulse” (Figura 4.13) y localizarla en el directorio “export\impulse_lib” creado por CoDeveloper. Se añade a la librería todos los archivos .vhd existentes en “export\impulse_lib (Figura 4.14). Se debe asegurar que todas las asociaciones sean “*All*”. Esta inclusión es necesaria debido a que dichas librerías contienen la descripción hardware de los objetos propios de ImpulseC (tales como *streams*, registros, zonas de memoria compartida, tipos de datos u objetos de señalización), que son llamados desde “*Mi_proyecto_top.vhd”.*

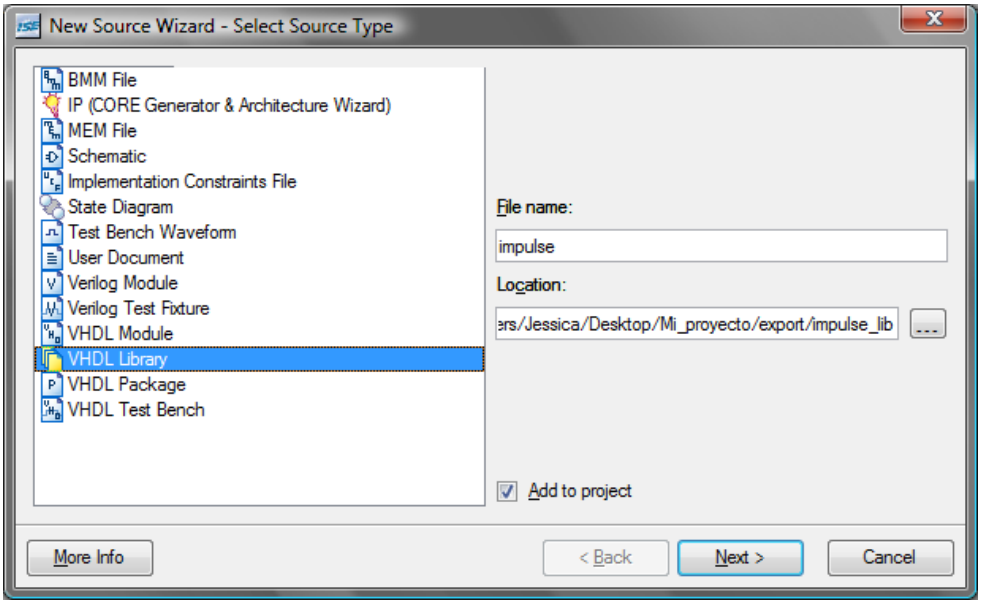

Figura 4.13. Se añade una nueva librería denominada “impulse” localizada en “export\impulse_lib”.

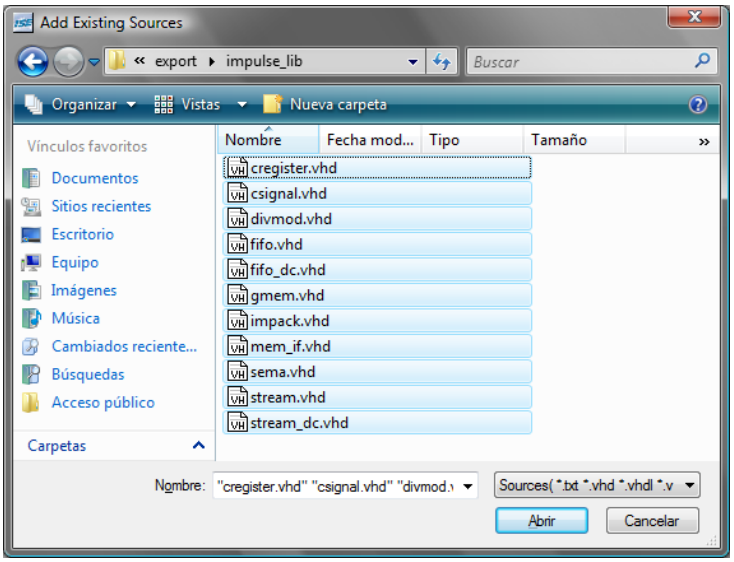

Figura 4.14. Se añaden a la librería todos los archivos .vhd existentes en "export\impulse_lib".

3. Seleccionar el archivo *Mi_proyecto_arch* como toplevel (Figura 4.15) y sintetizar para obtener el archivo *"Mi_proyecto_arch.ngc"* (Figura 4.16). Para ello, se debe deseleccionar previamente en las propiedades de síntesis la opción *"Add I/O buffers"* (Figura 4.17).

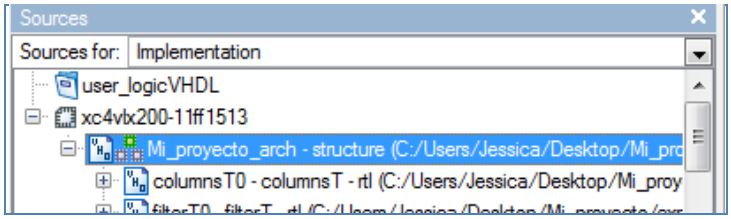

Figura 4.15. Mi_proyecto_arch como toplevel.

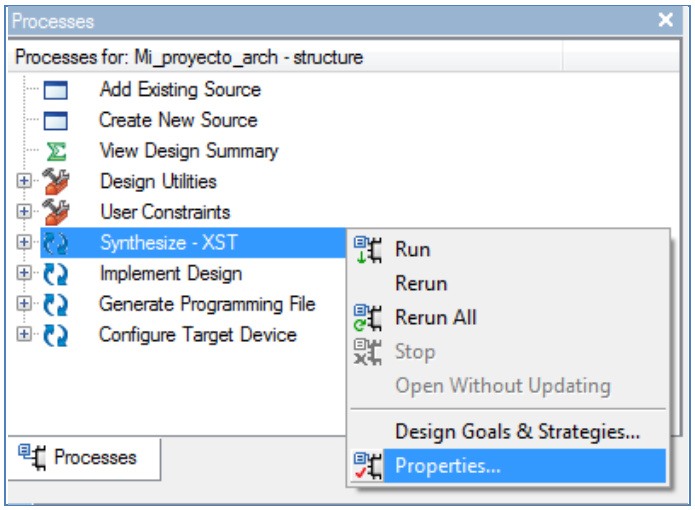

Figura 4.16. Antes de sintetizar se deben seleccionar las opciones correctas en *Propiedades*.

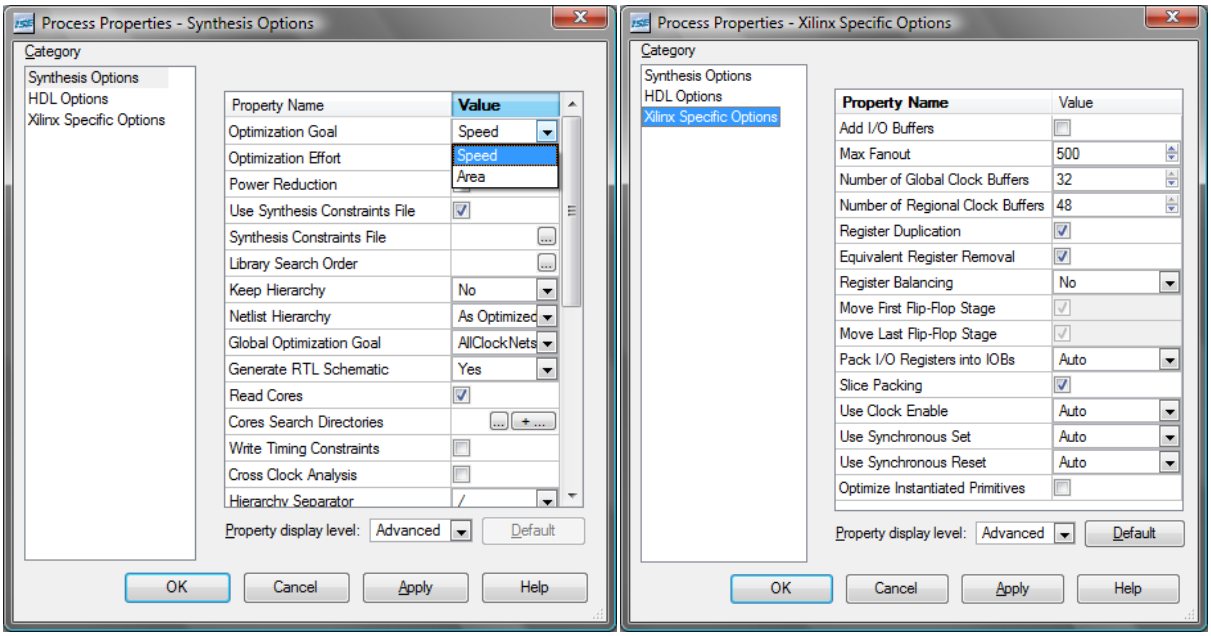

Figura 4.17. Opciones de síntesis. Izquierda: selecciona el objetivo de optimización (área o velocidad) y el esfuerzo de optimización (normal o *high effort*). Derecha: deseleccionar la pestaña *"Add I/O Buffers"*.

4. Comprobar en "*synthetise -> view RTL schematic*" que la arquitectura es correcta, y el informe de tiempos en *synthesis Report* para conocer la máxima velocidad de funcionamiento, imprescindible para la realización del siguiente proyecto *user_logic*. (Figura 4.18).

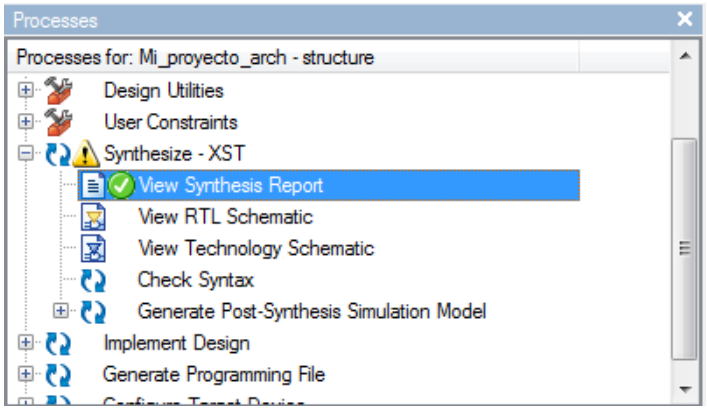

Figura 4.18. Comprobación de los resultados obtenidos en la síntesis.

4.2.2. Sintetizar el módulo de adaptación entre la interfaz de ImpulseC basada en streams y la proporcionada por DRC para el bus HT

En este apartado se enumerarán cada uno de los pasos necesarios para crear y sintetizar el proyecto "*user_logic"*, que servirá de interfaz entre el proyecto anterior "*user_logicVHDL*" y la interfaz proporcionada por DRC.

1. Crear un nuevo proyecto llamado *"user_logic"* en el directorio *"export\user_logic"* (Figura 4.19).

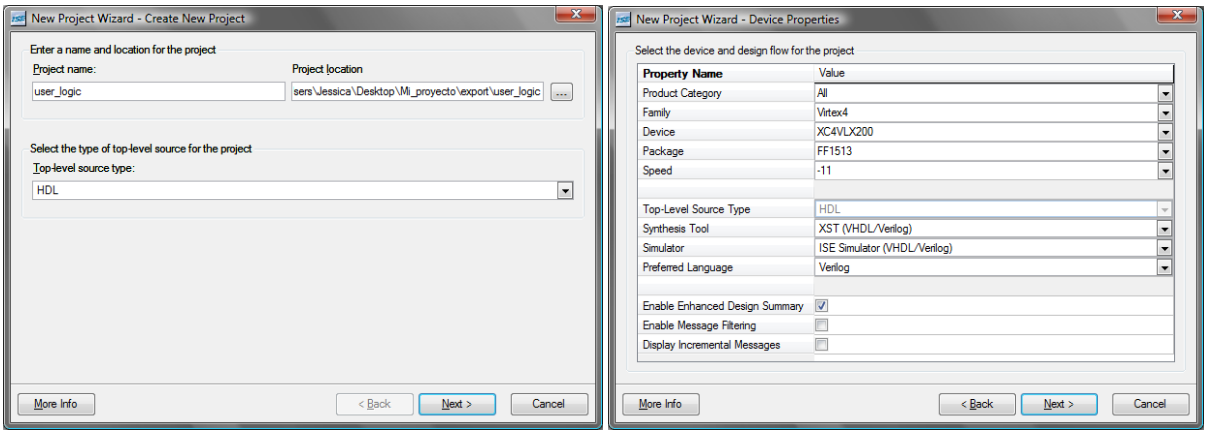

Figura 4.19. Creación de un nuevo proyecto llamado "user_logic" en el directorio "export\user_logic"

2. Añadir al proyecto los archivos "*user_logic.v*" y "*ht_rpsys.v*" (proporcionados por Impulse) descritos en *Verilog*, y seleccionar en ambos la asociación *"All"* (Figura 4.20). En caso de que ISE cambie el *toplevel* del proyecto, volver a seleccionar el "*user_logic*" como *toplevel.*

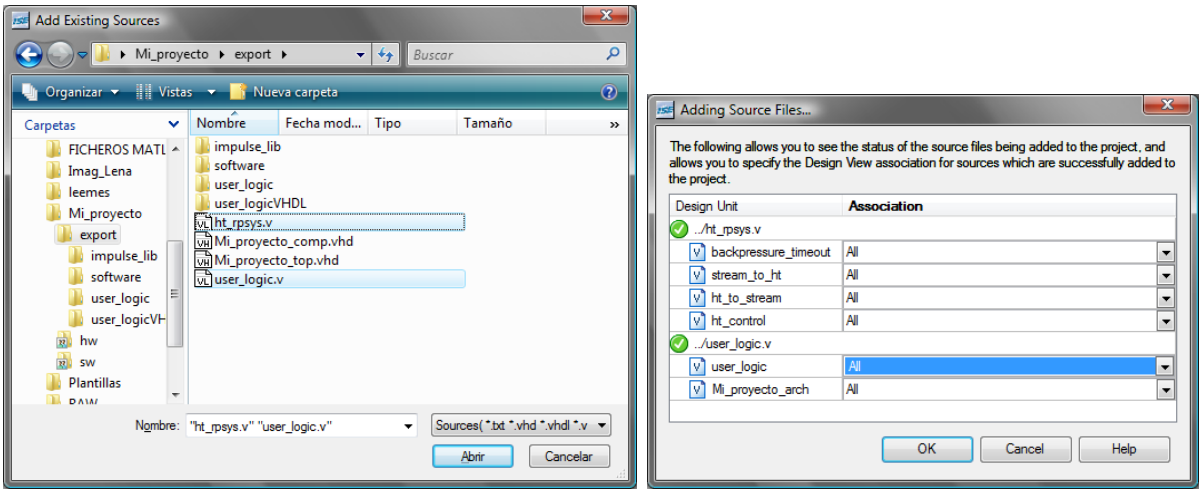

Figura 4.20. Se añaden al proyecto los archivos *user_logic.v* y *ht_rpsys.v*.

3. Seleccionar las siguientes opciones de síntesis:

- Establecer *"Verilog Include Directories"* a *"…\RPU110- L200\include"* (Figura 4.21).
- Deseleccionar en las propiedades de síntesis la opción *"Add I/O buffers"* (Figura 4.17, derecha).

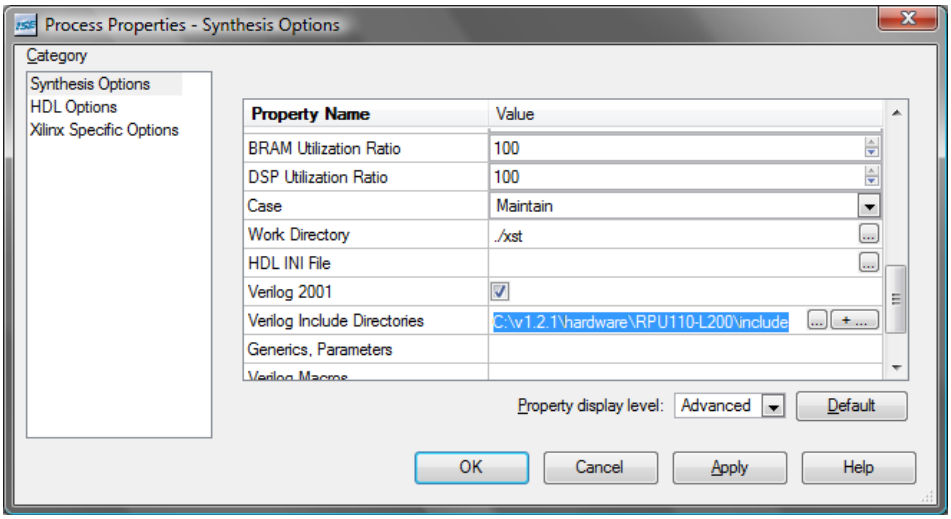

Figura 4.21. Establecer "Verilog Include Dierctories" a "C:\DRC\hardware\RPU110-L200\include\"

El archivo "*ht_rpsys.v*" generado por CoDeveloper, proporciona la interfaz que hará compatible el módulo hardware diseñado, con la interfaz genérica en formato *netlist* proporcionada por DRC. Para ello, proporciona cuatro módulos que van a ser instanciados a fin de conseguir una comunicación exitosa entre los *streams* y el bus *HyperTransport* (*streams_to_ht*) y viceversa (*ht_to_stream*), además de incluir un módulo que permite al software modificar el hardware para el control de las comunicaciones (*ht_control*) y otro que realiza el control de los datos en el buffer (*backpressure_timeout*).

El archivo *"user_logic.v"* generado por CoDeveloper, llama a la *entity* del proyecto *user_logicVHDL* e incluye la "API" proporcionada en el archivo anterior ("*ht_rpsys.v*"), para acoplar así, las entradas/salidas genéricas del proyecto *user_logicVHDL* con la interfaz para las comunicaciones con el bus *HT*.

4. En caso de ser necesario, modificar la línea del archivo *"user_logic.v"* que define la velocidad de reloj, en base a los resultados obtenidos en la síntesis del proyecto anterior (Figura 4.22). Se debe elegir por tanto, una velocidad de reloj que pueda soportar el hardware diseñado, esto es, que no supere a la obtenida en la síntesis del proyecto *user_logicVHDL*. Esto es necesario ya que en este nuevo proyecto, el *user_logicVHDL* se toma como una caja negra, por lo que la síntesis muestra una velocidad máxima mucho mayor a la real. Si todas las líneas están comentadas, se utiliza el valor por defecto para la señal *usr_clk*, que es 200MHz.

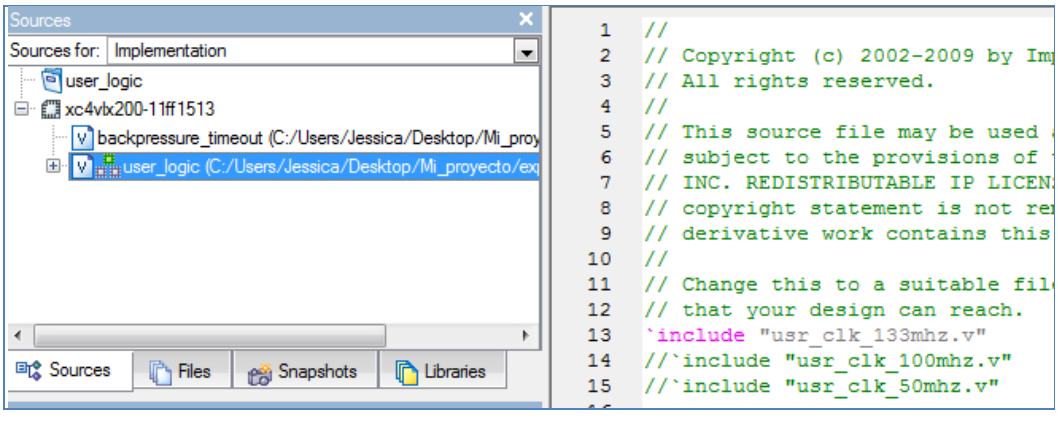

Figura 4.22. Selección de la velocidad de reloj en el archivo "*user_logic.v*".

5. Por último se sintetiza el proyecto para obtener la *netlist*. Si se obtuviera un mensaje de error indicando *"missing modules"*, descomentar las líneas al final del archivo "*user_logic.v*" y volver a sintetizar.

4.2.3. Implementar el RPWARE y generar el archivo de programación

A continuación se muestran los pasos para obtener, a partir de los proyectos anteriores *user_logicVHDL* y *user_logic*, y añadiendo el archivo *netlist* proporcionado por DRC, el archivo ".bit" que programará la FPGA.

1. Crear un nuevo proyecto llamado *rpware* localizado en "*export/rpware"*. Establecer el campo "*Top-Level Source*" a *"NGC/NGO"* (Figura 4.23).

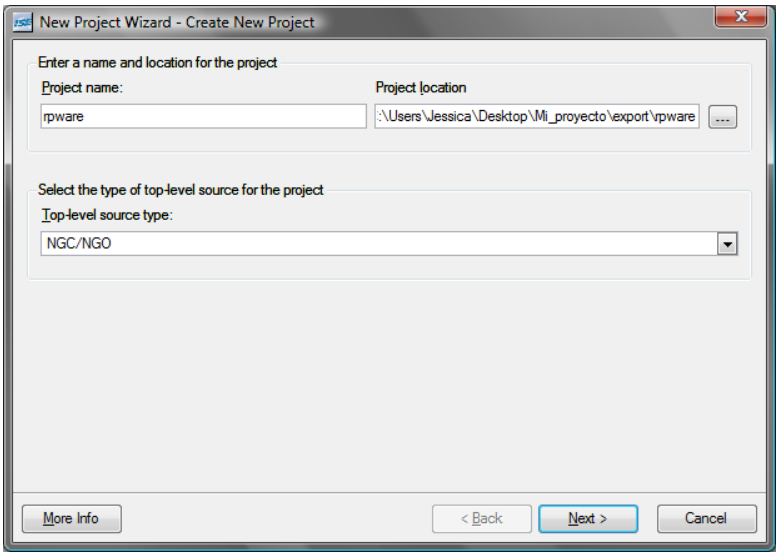

Figura 4.23. Creación del proyecto *rpware*.

2. Añadir como *toplevel* el archivo *netlist* "*rpware-ht0.ngc"*. Se recomienda añadirlo sin copiarlo al directorio para ahorrar espacio. A tal efecto, desmarcar la opción *"Copy the input design to the project directory"*. Establecer las opciones tal y como se muestra en la Figura 4.24.

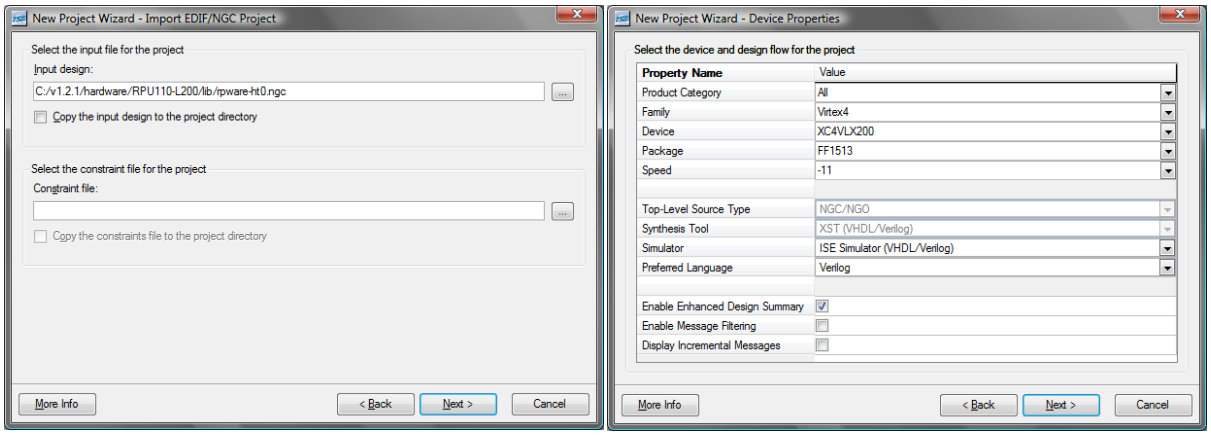

Figura 4.24. Creación del proyecto *rpware* (Opciones).

3. Una vez creado el proyecto, en las propiedades de "*implement design*" poner *"../user_logic | ../user_logicVHDL"* en el campo *"Macro Search Path"* (Figura 4.25). El primero es para que encuentre el *netlist user_logic* al que hace referencia el *rpware*. A su vez, el *user_logic*, hace referencia al *netlist user_logicVHDL* que se generó al principio, y por tanto, también debe ser incluido en el *path*.

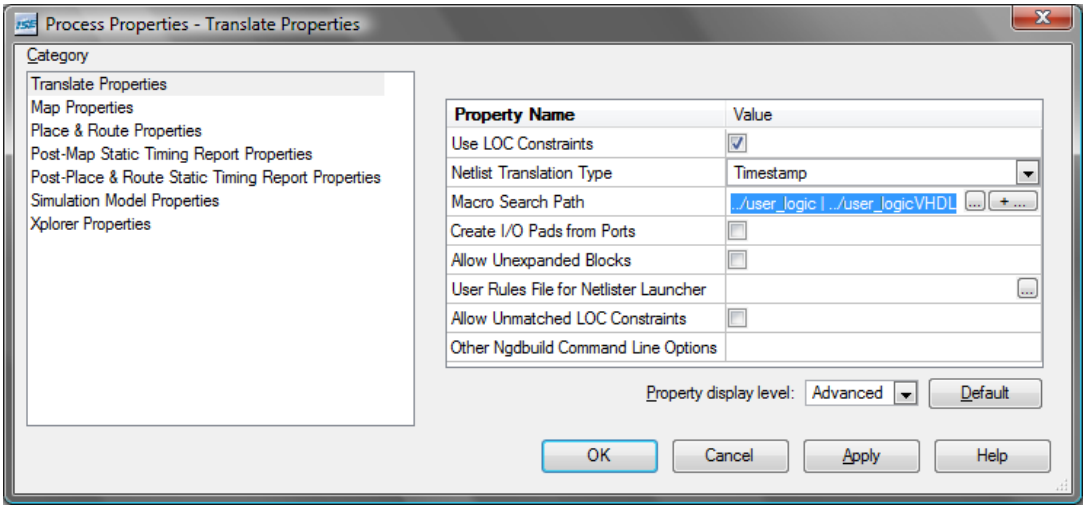

Figura 4.25. Opciones de implementación del proyecto *rpware*.

4. Si implementa con el ISE10.1 (y no con el 9.2) en el proyecto *rpware* hay que añadir un archivo *.ucf* con la restricción "*NET "LDT0_RX_CLK_P[0]" CLOCK_DEDICATED_ROUTE = FALSE;*" (ver Figura 4.26).

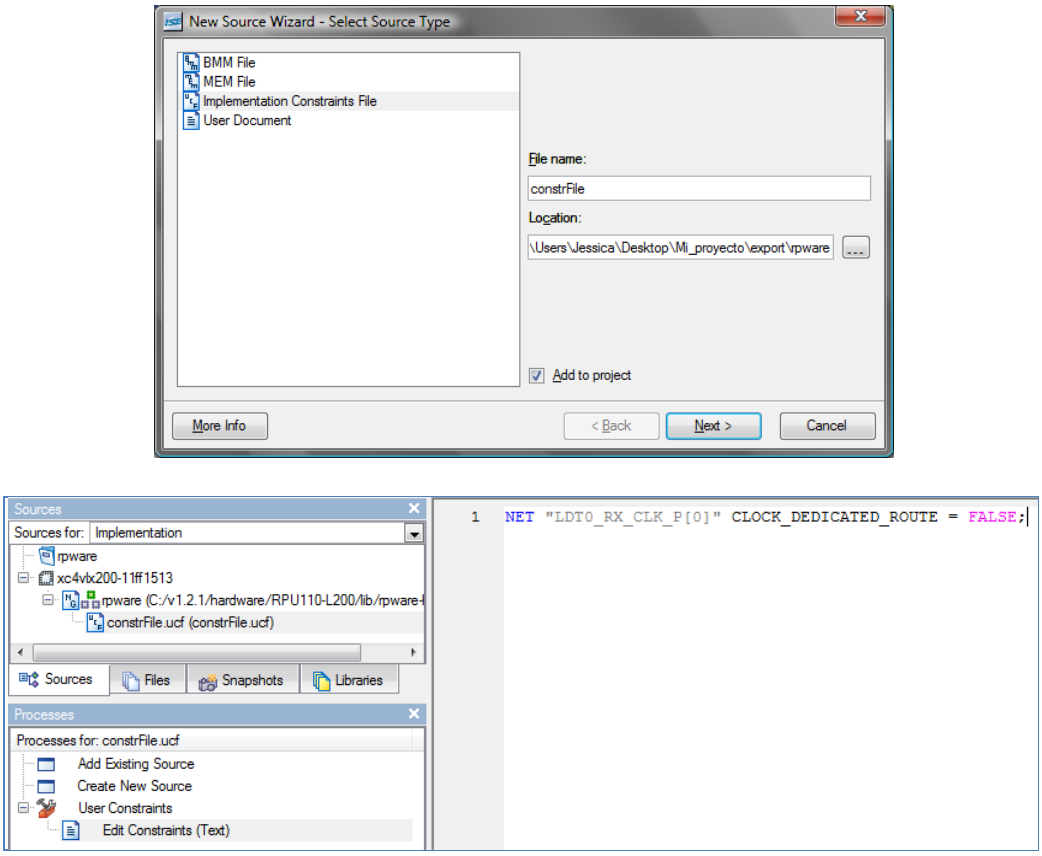

Figura 4.26. Se añade (superior) y edita (inferior) un archivo de restricciones.

5. Por último queda generar el archivo *.bit*. Para ello ir a *"Generate Programming File"*, hacer clic con el botón derecho del ratón y doble clic en *"Rerun all"* (Figura 4.27). Cuando el proceso haya terminado, se obtendrá finalmente el archivo *"rpware.bit".*

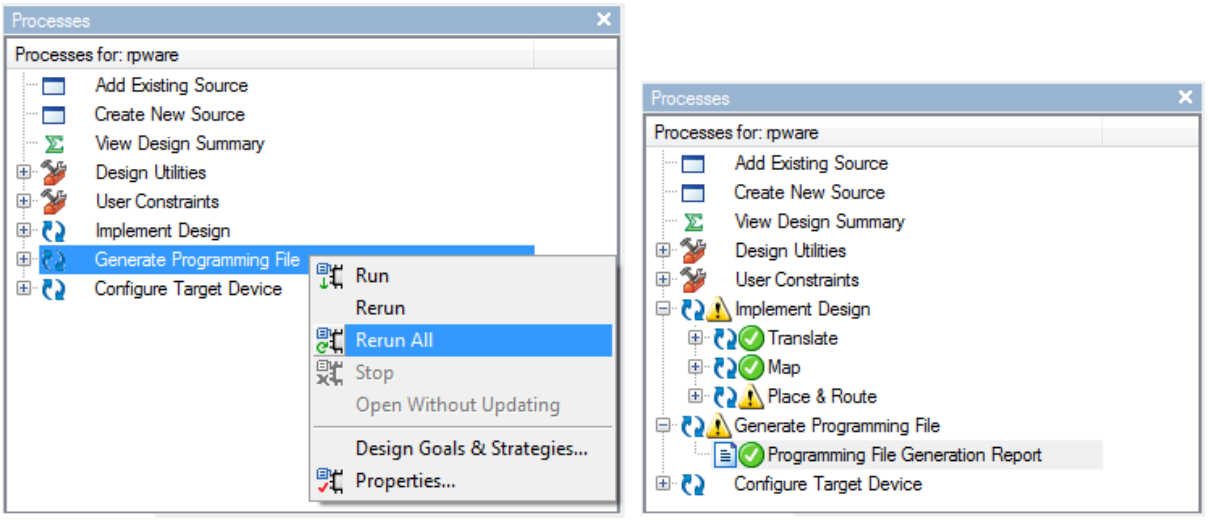

Figura 4.27. Generación del *.bit*.

4.3. Procedimiento para la generación del software para DRC

En este apartado se va a describir el procedimiento a seguir a la hora de obtener los archivos para hacer funcionar la parte software de la arquitectura sobre el SO Linux en el microprocesador AMD del computador DRC.

Los pasos a seguir se muestran a continuación:

1. Crear un directorio *"software"* dentro de *"export"*. Dentro de este crear dos directorios *"threads"* y *"nothreads".* Copiar dentro de "*nothreads*" los archivos: "*co_math.h"*, "*co_types.h"*, "*drc_common.c"*, "*drc_common.h"*, "*Mi_proyecto.h"* y "*Mi_proyecto_sw.c"*. Copiar también un *Makefile* ya hecho de otro ejemplo para modificarlo.

Como es obvio, "*co_math.h*" y "*co_types.h*" serán necesarios en caso de seguir usando algún tipo de datos especial de ImpulseC. Por otro lado, "*Mi_proyecto.h"* y "*Mi_proyecto_sw.c",* contendrán las definiciones y el comportamiento de la parte software de nuestra arquitectura.

Los ficheros "drc_*common.c"* y "drc_*common.h"* (proporcionados por Impulse) contienen las funciones y definiciones necesarias para conectarse y desconectarse de la FPGA, así como para controlar el buffer *HT*. Para esto último, el fichero "drc_*common.h"* define las funciones de alto nivel que usaremos para la lectura y escritura (ver Figura 4.28), ya que DRC no conoce las funciones del tipo "co_stream*" propias de ImpulseC.

```
#define DRC_UINT64_co_stream_read(a,b,c) { Read_Stream(rpu, b, c); }
#define DRC_UINT64_co_stream_read_nb(a,b,c) { Read_Stream_nb(rpu, b, c); }
#define DRC_UINT64_co_stream_write(a,b,c) { Write_Stream(rpu, b, c); }
```

Figura 4.28. Funciones de lectura/escritura definidas en *"drc_common.h".*

Estas funciones, cuya estructura es análoga a la de las funciones “co_stream*”, se traducen en llamadas a las funciones descritas en el fichero “drc_*common.c”*, que a su vez llaman a las de bajo nivel proporcionadas por DRC. Dicha implementación puede verse en las siguientes figuras.

```
int Write_Stream(Rpu_t *rpu, uint64_t *val, int nVal ) {
    if( Rpu_WriteUser(rpu, 0x10000 , val, RPU_UNSIGNED_LONG, nVal,
                  RPU_CHANNEL_AUTO, RPU_ORDERING_STRICT) < nVal ) {
       fprintf( stderr, "Write_Stream() wrote too few elements: %s\n",
            Rpu_GetErrorMessage(rpu) );
         return( -1 );
    } else {
     return( 0 );
    }
}
```

Figura 4.29. Función de escritura implementada en *“drc_common.c”.*

```
int Read_Stream(Rpu_t *rpu, uint64_t *val, int nVal ) {
    int notyet;
    unsigned long status;
    // Wait till data is ready
    notyet = 1;

    while(notyet) {       // will wait indefinitely if no new data is available.
           Read_Status(rpu, &status);
           if((status&2) == 2) notyet = 0;
       }

    if( (Rpu_ReadUser(rpu, 0x10000, val, RPU_UNSIGNED_LONG, nVal,
                    RPU_CHANNEL_POLLING_DMA, RPU_ORDERING_STRICT)) < nVal ) {
        fprintf( stderr, "Read_Stream() read too few elements: %s\n",
            Rpu_GetErrorMessage(rpu) );
         return( -1 );
    } else {
     return( 0 );
    }
}
```

Figura 4.30. Función de lectura implementada en *“drc_common.c”.*

```
int Read_Stream_nb(Rpu_t *rpu, uint64_t *val, int nVal ) {
    unsigned long status;
// Check if NEW data is available
    Read_Status(rpu, &status);      // If no new data is available
    if((status&2) != 2) {           // at this instant, the read exits.
         return (1);                // This corresponds to a "non-blocking" read.
    }

    if( (Rpu_ReadUser(rpu, 0x10000, val, RPU_UNSIGNED_LONG, nVal,
                    RPU_CHANNEL_POLLING_DMA, RPU_ORDERING_STRICT)) < nVal ) {
        fprintf( stderr, "Read_Stream() read too few elements: %s\n",
            Rpu_GetErrorMessage(rpu) );
         return( -1 );
    } else {
     return( 0 );
    }
}
```

Figura 4.31. Función de lectura no bloqueante implementada en *“drc_common.c”.*

2. Modificar el archivo “*Mi_proyecto_sw.c”*. Seguir los siguientes pasos:

- Eliminar todos los *include* referentes a *win32*.

- Eliminar el *header* “*co.h”* y todos los *include* de ImpulseC que no sean necesarios (normalmente solo hay que dejar el “*co_types.h”* y el “*co_math.h”*).

- Adicionalmente en este PFC se incluye “dirent.h”, que servirá para gestionar el acceso directorios para el manejo de ficheros.

En la Figura 4.32, se muestran los ficheros y librerías incluidas en el software desarrollado en este PFC para la ejecución en el DRC.

```
/////////////// SO_Includes ////////////////////
// #ifdef WIN32
// #include <windows.h>
// #include <sys/types.h>
// #include <sys/stat.h>
// #else  // ! WIN32

#include <sys/stat.h>
#include <unistd.h>
// #endif

// /////////////// IMPULSEC_Includes ////////////////////
// #include "co.h"
// #include "cosim_log.h"

///////////////All_platform_Includes////////////////////
#include "co_math.h"
#include "Mi_proyecto.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

///////////////DRC_Includes////////////////////
#include <string.h>
#include <stdint.h>
#include "RpuC.h"
#include "drc_common.h" // DRC/impulse routines
#include "co_types.h"
#include <dirent.h>
```

Figura 4.32. Conjunto de *includes* necesarios en el software para la ejecución en DRC.

- Cambiar la función del proceso software por un *main (*Figura 4.33*)*.

```
/////////////// MAIN or FUNCTION ////////////////////

// extern co_architecture co_initialize(void *);
// void prodcon(co_stream pixels_raw, co_stream pixels_filtered){
int main(int argc, char *argv[])    //main function.

/////////////// CODE ////////////////
{
```

Figura 4.33. Cambio de la función del proceso software por un *main*.

- Añadir las variables que utilizan las funciones *DRC_*_co_stream_** para guardar los datos enviados/recibidos, que pueden verse en la Figura 4.34.

```
/////////////// DRC_declarations ////////////////////
    uint64_t outValue[NUM_ELEMENTS], inValue[NUM_ELEMENTS];
    //uint64_t Optionval=2; //enable write backpressure
    long unsigned int status;
/////////////// END OF DECLARATIONS ////////////////////
```

Figura 4.34. Variables que utilizan las funciones DRC_*_co_stream_* para guardar los datos enviados/recibidos.

Cuando se lean o se pasen datos de distinto tamaño a *uint64_t* a los *streams* hay que llevar cuidado con las funciones de casting: desplazar (opcionalmente) y enmascarar siempre.

- Añadir el código presente en la Figura 4.35 para conectarse y resetear la FPGA, siempre al principio de la aplicación.

```
//////////////////Establish a connection with the RPU.//////////////////////////
    Rpu_t *rpu = Rpu_New( "/dev/drcmod0" );

    // Check userlogic board
    if( Rpu_CheckBoard(rpu) != 0 )
    {
        fprintf( stderr, "Failed board check: %s\n", Rpu_GetErrorMessage(rpu) );
        return( 1 );
    }
    fprintf( stderr, "Estabished a connection with the RPU: running....\n" );

    // Reset the user logic
    fprintf(stderr, "\tReset User Logic with a soft_reset!\n");
    User_Logic_Reset(rpu);
    Read_Status(rpu, &status);
    fprintf(stderr,"\tStatus Value 0x%x_%x\n",(uint32_t)(status>>32),(uint32_t)(status&0xffffffff) );

    //Enable write backpressure <==> blocking write
    //User_Logic_Write_Options(rpu, &Optionval);

////////////////////////////////////////////////////////////////////////////////
```

Figura 4.35. ***Código para conectarse y resetear la FPGA.***

- Añadir el código presente en la Figura 4.36Figura 4.37 para desconectarse de la FPGA al final de la aplicación.

```
// CLOSE CONNECTION WITH THE RPU //////////////////////
   Rpu_Delete(rpu);
```

Figura 4.36. Código para desconectarse de la FPGA.

- Sustituir en el código las llamadas a funciones *co_stream** por las *DRC_UINT64_co_stream_**.

Así, si se quiere modificar la siguiente operación de escritura:

```
co_stream_write(pixels_raw, &outValue[i], sizeof(uint64)*TAMRAFAGA)
```

En el software para el DRC habrá que poner:

```
DRC_UINT64_co_stream_write(pixels_raw, &outValue[i], TAMRAFAGA)
```

Aunque se dijo que la estructura de ambas era análoga, hay que tener cuidado con el tercer parámetro. En CoDeveloper, corresponde al tamaño de los datos a enviar en bytes, esto será el tamaño del dato/paquete empleado, en bytes, multiplicado por el número de datos/paquetes que se envían (tamaño de ráfaga). En cambio, en el computador DRC, tal y como se implementa en *"drc_common.c"*, este parámetro determina el número de datos del tipo uint64_t. De esta forma, si se pretende enviar datos de 64 bits (o paquetes de datos de tamaño 64 bits), este parámetro valdrá 8 en el primer caso, y uno en el segundo.

En el caso de envío a ráfagas, es importante que el tamaño de la ráfaga sea menor que la profundidad del *stream*, ya que la interfaz hardware proporcionada por Impulse no soporta lectura/escritura *BACKPRESURE* (aunque la interfaz software de Impulse tampoco las soporta, se podrían utilizar las funciones de la API de DRC directamente, que si la soportan). Este tema se

abarcará en el capítulo de resultados, donde se mostrarán las pruebas realizadas y conclusiones en torno al envío y recepción de ráfagas en el DRC.

- Eliminar el código de simulación de ImpulseC (*IF_SIM*, *logwindow*…).
- Eliminar los co_stream_open y co_stream_close.
- Eliminar el antiguo *main*.
- Si se usa algún tipo de datos de *ImpulseC* o alguna condición de estado de los *streams* deberá corregirse también.

3. Compilar la aplicación. Para ello:

- Modificar el *Makefile* si fuese necesario.
- Compilar con el comando *make* (>*make –f Makefile*).

4.4. Instalación del archivo .bit y co-ejecución en el DRC

Una vez tenemos el archivo .bit para configurar el hardware por un lado, y el software compilado por otro, sólo resta configurar el hardware y ejecutar la aplicación. Para ello:

- Copiar el archivo "*rpware.bit"* en la máquina DRC.
- Reprogramar la FPGA con el comando *"rpu_reconfigure rpware.bit –r"* (es necesario ser *root).*
- La máquina se reiniciará en este momento, para que se realice el proceso de inicialización (*start up*) de la FPGA en el que se reconfigura con la nueva aplicación almacenada en la memoria FLASH de programación.
- Ejecutar el software para comenzar el procesado: >./Mi_proyecto_sw

Con el fin de agilizar el manejo con el computador DRC en futuros trabajos, se incluyen en el ANEXO I las instrucciones más inmediatas para el acceso remoto al mismo desde el PC de desarrollo para la copia y modificación de los archivos correspondientes.

4.5. Resumen

En el esquema de la Figura 4.37 se muestra de forma resumida la nueva técnica de diseño, englobando todas las etapas descritas en las secciones anteriores.

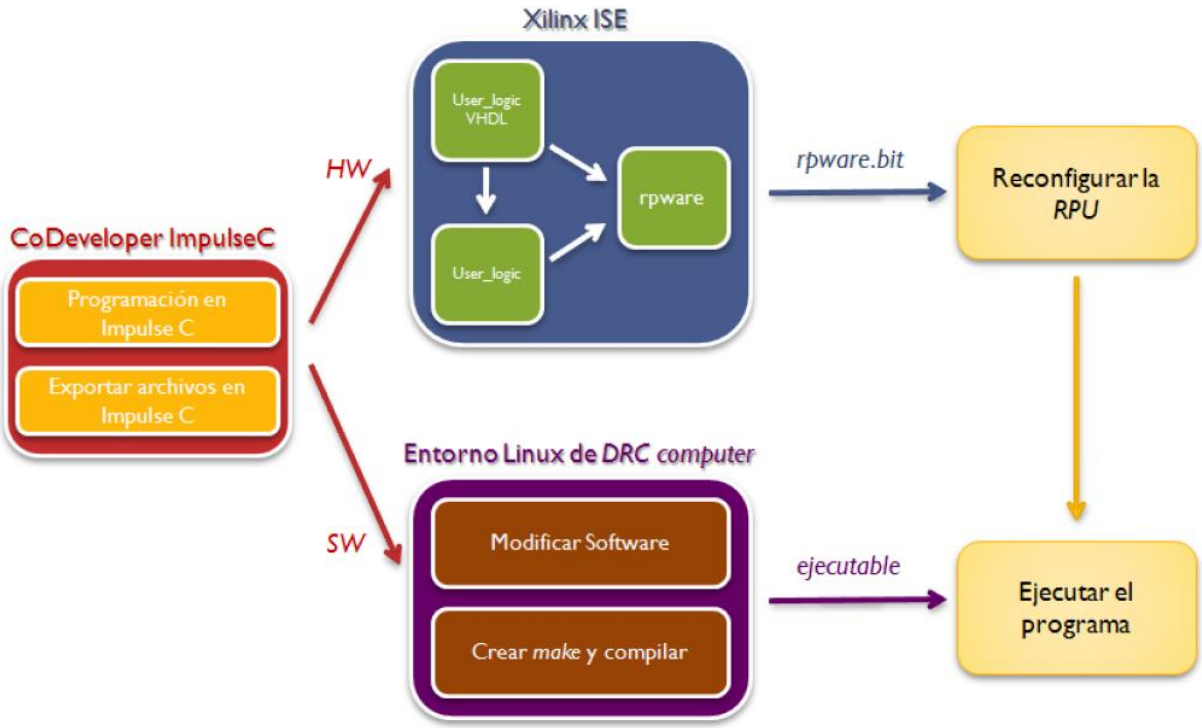

Figura 4.37. Resumen de la metodología de diseño.

Capítulo 5

Diseño e Implementación del Coprocesador

En el apartado 2.3 (Figura 5.1) se presentó el esquema de un algoritmo de reducción de imágenes basado en la descomposición *wavelet à trous* de segunda escala, y se comprobó la mejora que ofrecía respecto al actual algoritmo utilizado por FastCam.

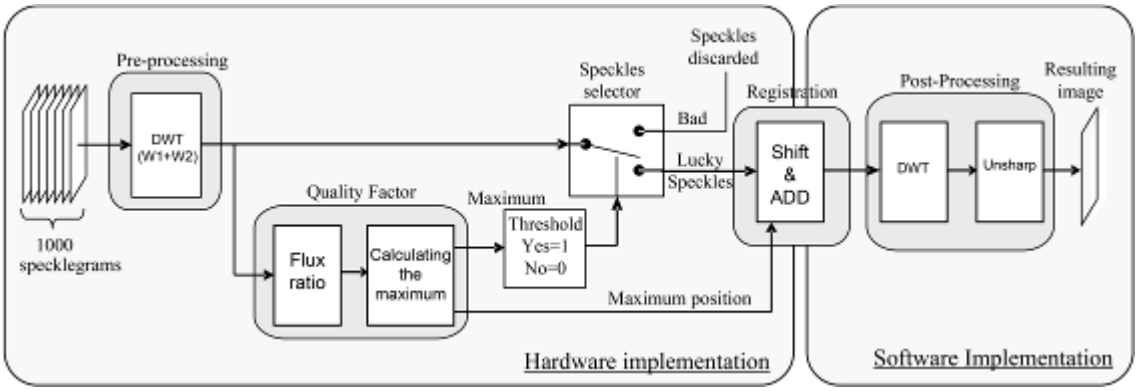

Figura 5.1. Algoritmo hardware propuesto. La etapa 1 realiza un pre_procesado a cada *specklegram* para resaltar la información en las frecuencias relevantes. La etapa 2 evalúa la calidad de la imagen. La etapa 3 registra exclusivamente las *Lucky Images*. La etapa 4 realiza un post-procesado de la imagen acumulada.

En este capítulo se va a implementar la primera etapa de pre-procesado del algoritmo, que como ya se indicó, es la más crítica en tanto que requiere un elevado cómputo para el cálculo de la suma de las dos primeras escalas *wavelets* ($w_1 + w_2$).

5.1. Diseño básico de la etapa de pre-procesado (W1+W2)

En este apartado se pretende introducir el diseño básico que se va a implementar, mostrando únicamente el procesado que se realizará al flujo de datos, sin entrar en detalles respecto a los procesos, *stream* y demás elementos relativos a la programación de Impulse C.

Para sacarle el máximo rendimiento al computador DRC, se va a separar el diseño en dos partes, la parte software, que se ejecutará en el microprocesador, y la parte hardware, que se ejecutará en la FPGA (ver Figura 5.2). La primera se encargará de gestionar la lectura y recepción de datos, mientras que la parte hardware se encargará de las tareas de cálculo intensivo para lo que utilizará varios procesos hardware, que se detallarán a lo largo del capítulo.

Figura 5.2. Esquema general de la arquitectura.

Por tanto, el procesado necesario para el cálculo de la suma de las dos primeras escalas *wavelet* se implementará en la parte hardware. Dicha implementación se basará principalmente en dos *kernel* de convolución en cascada que actuarán a modo *pipeline*. El esquema lógico de la arquitectura puede verse en la Figura 5.3.

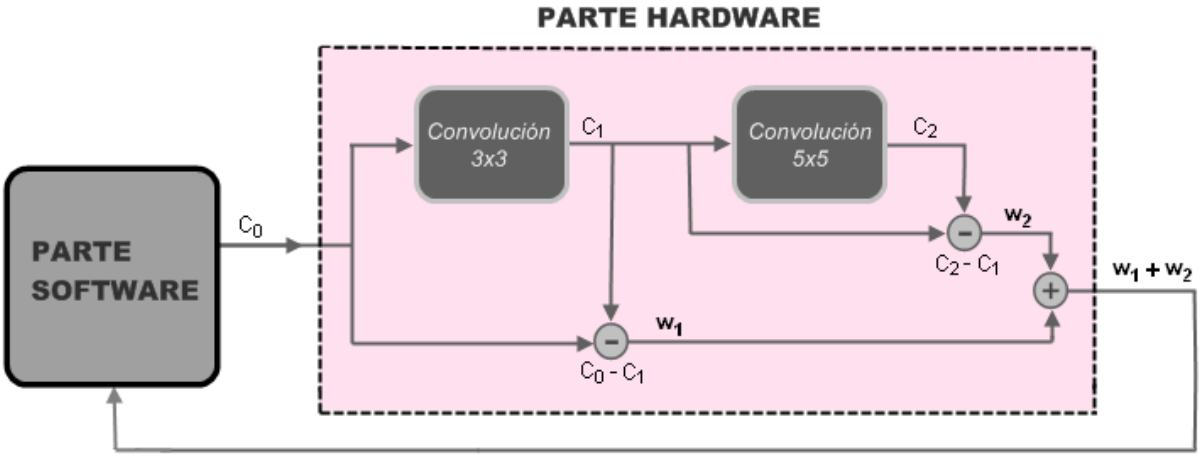

Figura 5.3. Esquema lógico de la arquitectura.

Según la nomenclatura introducida en el apartado de 2.2, que introducía la teoría de *wavelets*, en el esquema se observa que inicialmente se obtendrá la imagen C_1, mediante la convolución de los datos de la imagen original C_0, con una máscara 3x3. A partir de los datos resultantes C_1, se obtendrá la imagen C_2 mediante una convolución con una máscara 5x5. Cada imagen resultante de una convolución, deberá restarse con la imagen previa a dicha convolución. De esta forma se obtendrán las dos primeras escalas *wavelets* como $\mathbf{w_1}=C_1-C_0$ y $\mathbf{w_2}=C_2-C_1$. Por último, la suma de ambas escalas se devolverá a la parte software, que mostrará los resultados.

De esta forma, queda definido el aspecto lógico de la arquitectura del algoritmo, dejando para apartados posteriores la arquitectura exacta adoptada para implementar la parte hardware.

5.2. Implementación de una convolución genérica 3x3

Dado que el cómputo principal de la arquitectura a implementar en este PFC recae sobre dos convoluciones, la primera con una máscara de convolución de 3x3 y la segunda de 5x5, en esta sección se abarcará el procedimiento seguido para la obtención de la primera.

El objetivo principal de esta sección será conseguir una implementación que proporcione el funcionamiento más eficiente en la aplicación de una máscara de convolución de 3x3 a un flujo de datos de imágenes continuo. Posteriormente, esta metodología podrá extenderse al cálculo de la convolución de 5x5 que se aplicará a los datos resultantes de la primera.

En la Figura 5.4 se muestran los procesos y *streams* que componen la arquitectura implementada. En ésta, acatando las restricciones de ImpulseC indicadas en el apartado 3.2, se implementa un único proceso software (*prodcon*), que realizará tan sólo tareas de lectura y escritura de ficheros. Este proceso software se comunica con los procesos hardware (sobre fondo rosa en la Figura 5.4) a través de *input_stream* y *output_stream*, con un ancho de *stream* de 64 bits. Así, el proceso software enviará un flujo continuo de paquetes de datos de 16 bits a la parte hardware, la cual estará compuesta por dos procesos implicados en la convolución (*columnsT* y *filterT*) y un tercero (*Empaq*) que empaquetará los resultados de cuatro en cuatro para enviarlos al software, aprovechando así los 64 bits del *stream*.

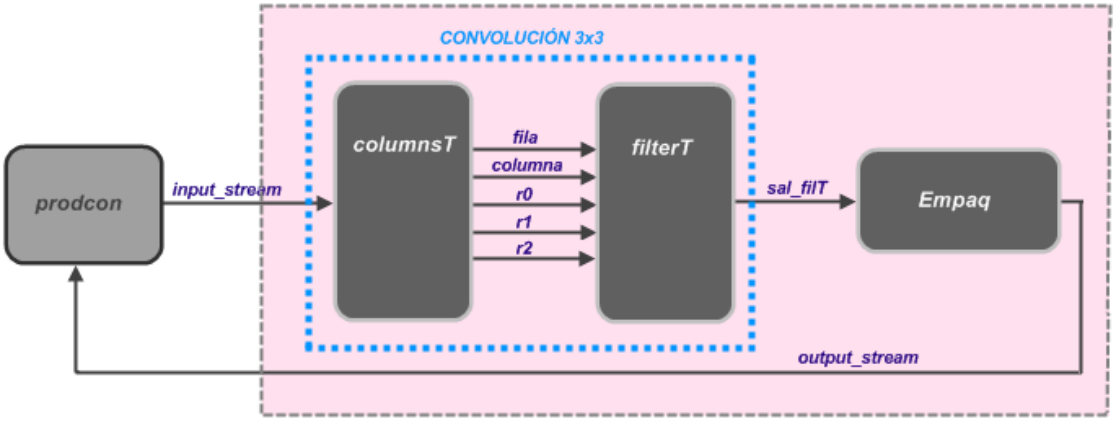

Figura 5.4. Arquitectura para la implementación de una convolución 3x3.

A continuación se explicarán detalladamente tanto la parte software como la parte hardware de esta arquitectura. Ambas partes podrán reutilizarse posteriormente en la implementación de la arquitectura que calcule la suma de las dos primeras escalas *wavelet*.

5.2.1. Implementación del proceso software

Como se ha indicado en la introducción previa a este apartado, el proceso software se ha definido con el nombre ***prodcon***. Los datos de los ficheros que leerá este proceso son de 16 bits, por lo que se van a empaquetar de cuatro en cuatro antes de enviarlos al hardware, aprovechando así todo el ancho del *stream*, que como ya se indicó es de 64 bits. En la Figura 5.5 puede verse cómo se envían y reciben por el bus *HyperTransport*, paquetes de 4 píxeles de 16 bits/píxel.

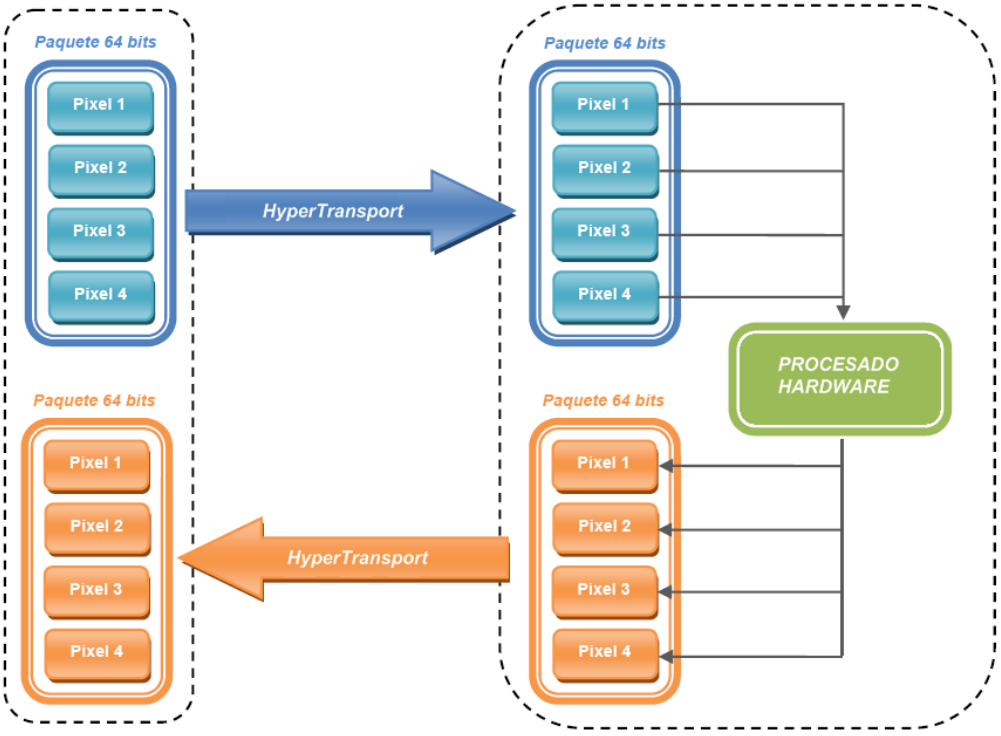

Figura 5.5. Empaquetado de datos.

Para implementar este mecanismo, el proceso software implementado en CoDeveloper leerá inicialmente un fichero de datos de 16 bits, y almacenará todos los paquetes antes de comenzar a enviarlos a la parte hardware. En el apartado 5.4 se describirán diversas modificaciones en este software para la ejecución en el computador DRC. En éstas se dispondrá de distintas versiones de este software que facilitarán la comprobación del hardware implementado, permitiendo que conforme se van leyendo los datos de ficheros de un directorio determinado, se empaqueten y envíen a la parte hardware, así como el envío de ráfagas, etc.

Así, en el software más sencillo, para la simulación en CoDeveloper, tras la lectura del fichero con la función *fread()*, cada dato de 16 bits quedará almacenado en una posición de *Buffer16[i]*, que posteriormente se irán tomando de cuatro en cuatro para formar los paquetes *PKT[k].* En el Cuadro 5.1 se muestra el código que describe el empaquetado de datos en el software, donde *PixelCount* se define como el número total de píxeles en una imagen.

```
fread(Buffer16, sizeof(uint16), PixelCount, inFile);  //Lee el fichero de entrada.

      for(k=0;k<(PixelCount/4); k++){

            PKT[k]= ((((co_uint64)Buffer16[k*4])&0xff)<<48) |
                    ((((co_uint64)Buffer16[k*4+1])&0xff)<<32) |
                    ((((co_uint64)Buffer16[k*4+2])&0xff)<<16) |
                    (((co_uint64)Buffer16[k*4+3])&0xff);
      //printf("PKT[%d]: %d,%d,%d,%d \n", k,
```

```
        (co_uint16)(PKT[k]),(co_uint16)((PKT[k])>>16),(co_uint16)((PKT[k])>>32),
        (co_uint16)((PKT[k])>>48));

        }
```

Cuadro 5.1. Empaquetado de datos en el proceso software.

En el procedimiento seguido en el código anterior, cada dato de 16 bits se convierte a 64 bits, se aplica una máscara, de forma que para cada nuevo dato de 64 bits, sólo sean distintos de cero los 16 bits menos significativos, que se desplazarán en función del orden de lectura. Así, el primer dato de 16 bits se desplazará 48 posiciones, y los sucesivos 32, 16 y cero.

En la Figura 5.6, se representan los datos ya desplazados y preparados para las tres operaciones *OR* que producirán el paquete resultante.

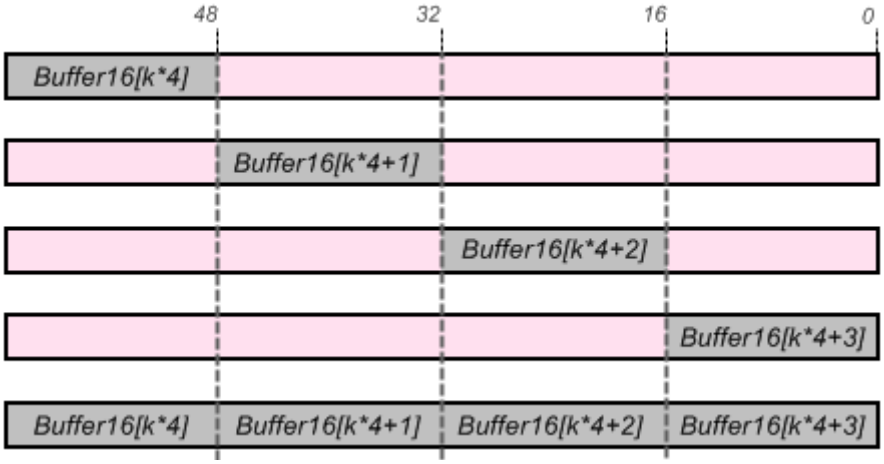

Figura 5.6. Empaquetado de datos de 16 bits en paquetes de 64 bits.

De igual forma, los datos procesados que se reciban de la parte hardware estarán empaquetados, por lo que debe realizarse el proceso inverso de desempaquetado, antes de escribir los ficheros de salida. En el Cuadro 5.2 se resalta el código que realiza el desempaquetado.

```
for(ind2=0; ind2<PixelCount/4; ind2++){

        //DESEMPAQUETADO DE DATOS
        rec[0] = (co_int16)(Resultados[ind2]>>48);
        rec[1] = (co_int16)(Resultados[ind2]>>32);
        rec[2] = (co_int16)(Resultados[ind2]>>16);
        rec[3] = (co_int16)(Resultados[ind2]);

        fprintf(outFile, "%d\t%d\t%d\t%d\t", rec[0],rec[1],rec[2],rec[3]);

        if(ind2==0) printf("%d\t%d\t%d\t%d\t\n",rec[0],rec[1],rec[2],rec[3]);
        if(ind2==PixelCount/4-1) printf("%d\t%d\t%d\t%d\t\n",rec[0],rec[1],rec[2],rec[3]);
        ind5++;
        if(ind5==128) {fprintf (outFile, "\n"); ind5=0;}  //Retorno de carro en cada línea.
}
printf("Ha terminado de escribirse el fichero de salida\n");
```

Cuadro 5.2. Desempaquetado de datos en el proceso software.

El procedimiento para extraer los cuatro datos de cada paquete recibido *Resultados[ind2]*, requiere para cada dato del paquete, desplazar a la derecha hasta alcanzar los 16 bits menos significativos y convertir entonces a enteros de 16 bits, que serán los que se escribirán en el fichero de salida.

De esta forma, se pretende reducir el tiempo invertido en las operaciones de lectura/escritura entre la parte software y la parte hardware, ya que se divide por cuatro el número operaciones de este tipo.

Por último, en el Cuadro 5.3 se muestra el código relativo al envío y recepción de paquetes de 64 bits. Por el momento, el valor de *TAMRAFAGA* se ha fijado a 1, lo que significa que se enviará paquete a paquete.

```
ind3=0;
for (i=0; i < PixelCount/4; i=i+TAMRAFAGA) { //512*512/4=65536 paq en ráf. de "TAMRAFAGA" paq.

        co_stream_write(pixels_raw, &PKT[i], sizeof(uint64)*TAMRAFAGA);
        //printf("rafaga enviada=%d \n",i);

        //LECTURA DE LOS PIXELS RECIBIDOS:
        if (co_stream_read_nb(pixels_filtered, &Resultados[ind3], sizeof(uint64)*TAMRAFAGA)){
            IF_SIM(cosim_logwindow_fwrite(log, " \n Primero de rafaga= %d", Resultados[ind3]);)
            ind3=ind3+TAMRAFAGA;
        }
}
printf("\n Finished writing pixels\n");

//VACIADO DEL BUFFER
while(ind3<PixelCount/4){
        co_stream_write(pixels_raw, &PKT[0], sizeof(uint64)*TAMRAFAGA);
        //printf("rafaga basura enviada \n");

        if(co_stream_read_nb(pixels_filtered, &Resultados[ind3], sizeof(uint64)*TAMRAFAGA)){
                //printf("recibo rafaga ind3=%d \n",ind3);
                IF_SIM(cosim_logwindow_fwrite(log, " \n Resultados=: %d", Resultados[ind3]);)
                ind3=ind3+TAMRAFAGA;
        }
}
```

Cuadro 5.3. Envío y recepción de paquetes de 64 bits.

Como ya se explicó en el capítulo Capítulo 4, al realizarse la lectura y escritura se hace en el mismo proceso *prodcon*, es importante, que tras escribir un dato, la lectura sea no bloqueante para evitar un posible *deadlock*.

En el código sombreado se muestra el vaciado del *buffer*. Se observa que entre cada nueva recepción, se envía el primer paquete, el cual no influirá en ningún resultado, ya que esto se hace una vez que todas las imágenes ya han sido enviadas al hardware. Simplemente se realiza un envío "basura*"* para evitar que el hardware deje de enviar datos procesados, ya que éste esta programado para que trabaje mientras reciba datos.

Por último comentar que en este vaciado del *buffer*, la lectura también se define no bloqueante. Esto no será necesario en las versiones del software para ejecutar en el DRC, ya que el dato siempre estará listo una vez superada la etapa de inicialización. Este tema se abordará en apartados posteriores.

5.2.2. Procesos hardware que implementan la convolución 3x3

Los procesos implementados van a describir una arquitectura hardware para el cómputo de una convolución en modo flujo de datos, buscando la máxima velocidad posible. En la Figura 5.7 se muestra cómo se comenzará procesando el píxel de la esquina superior izquierda de la imagen, P(0,0), por lo que la máscara de convolución debe estar centrada en el mismo. Aplicando la suma de productos de cada uno de los píxeles de la imagen sobre los que se superpone, se obtendrá el resultado del primer píxel de la imagen de salida, $C_1(0,0)$. Para minimizar el retardo el sistema calculará el resultado del píxel P(n,m) en cuanto el sistema disponga del píxel P(n+1,m+1).

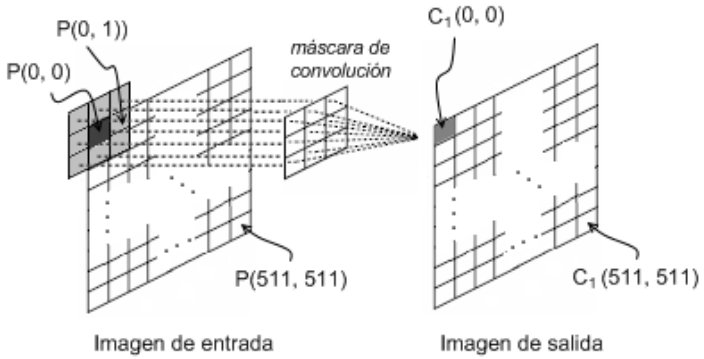

Figura 5.7. Convolución 3x3.

Como se comentó en la introducción, en el cálculo de la convolución de una imagen se utilizan dos procesos (*columnsT* y *filterT*). El primero se encargará de preparar columnas de tres datos de la imagen que enviará al proceso *filterT*, el cual, calculará un resultado conforme disponga de los datos necesarios para la obtención del mismo. Así, la filosofía de este método consiste en que una vez superado un periodo de inicialización, el proceso *columnsT* envíe una columna a *filterT* en cada ciclo de reloj, y este último saque el resultado previo en dicho ciclo. Este mecanismo debe funcionar de forma continua para cuantas imágenes se reciban del proceso software, sin necesitar ningún tiempo adicional de inicialización entre las mismas.

En el modo de operación normal (una vez superada la inicialización), el proceso *columnsT* tendrá que tener almacenadas dos filas de la imagen, de forma que para cada nuevo dato recibido (píxel P(n+1,m+1)) del software, pueda enviar una columna de tres datos al proceso *filterT*. En la Figura 5.8 se muestra de forma gráfica el procedimiento seguido. Las dos filas necesarias en el proceso *columnsT* se almacenarán en dos arrays `A[]` y `B[]` de 512 elementos (ancho de la imagen).

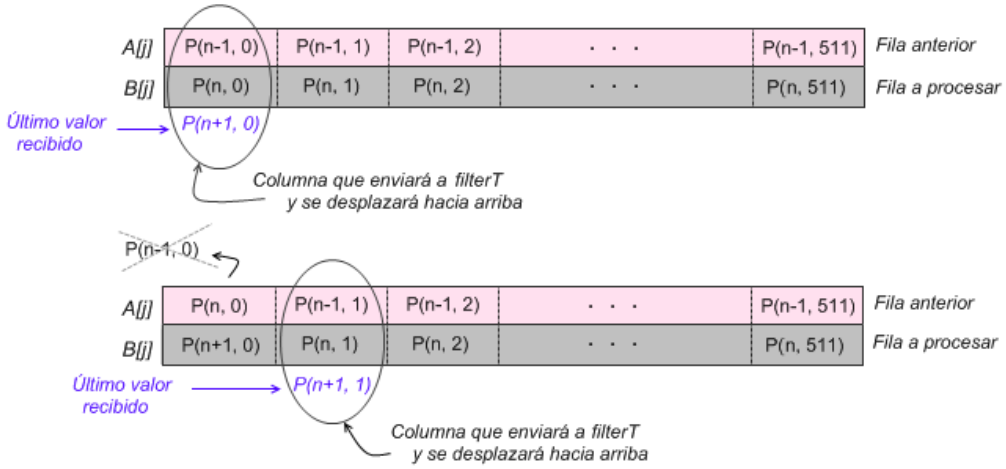

Figura 5.8. Funcionamiento del proceso *columnsT*.

Por su parte, el proceso *filterT* tendrá almacenados en 9 variables, los píxeles que se multiplicarán por la máscara de convolución en cada ciclo. Así, almacenará cada nueva columna de 3 píxeles recibida, en las variables `data0`, `data1`, `data2`, y se actualizarán las variables que contienen los 6 pixeles recibidos en los 2 ciclos anteriores, para que la suma de productos produzca el resultado del píxel P(n,m) correspondiente. Dichas variables se denominan `p11`, `p12`,…, `p33`. La actualización de las variables sigue el siguiente esquema:

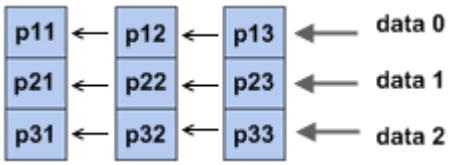

Figura 5.9. Núcleo de convolución en *filterT*.

Una vez actualizadas las variables se realiza la suma de productos correspondientes a la convolución. Los productos[7] (`p11*F9`, `p12*F8`, …) se realizarán en un mismo ciclo de reloj. Como resultado se generan 9 multiplicadores en paralelo. Los coeficientes del *kernel* han sido previamente definidos siguiendo la siguiente nomenclatura:

$$h_{3x3} = \frac{1}{16} \cdot \begin{pmatrix} F1 & F2 & F3 \\ F4 & F5 & F6 \\ F7 & F8 & F9 \end{pmatrix} \tag{19}$$

Los coeficientes (`F1`, `F2`, `F3`, …) contienen números enteros para realizar los productos en el proceso *filterT* con datos de tipo `co_uint16`. El factor multiplicativo 1/16 es un término de normalización que se implementará directamente en hardware y su valor proviene de la máscara de coeficientes empleados en el cálculo de DWT *à trous* (ver sección 2.2).

La condición de contorno para tratar con los píxeles que bordean la imagen consiste en suponer ceros alrededor de la misma. Una solución que evita rellenar con ceros, se consigue empleando una máscara binaria (`m11`, `m12`, `m13`,…) 3x3 que decida en cada momento que elementos se multiplicarán por los coeficientes del filtro en cada ciclo de reloj. Esta solución se muestra en la Figura 5.10.

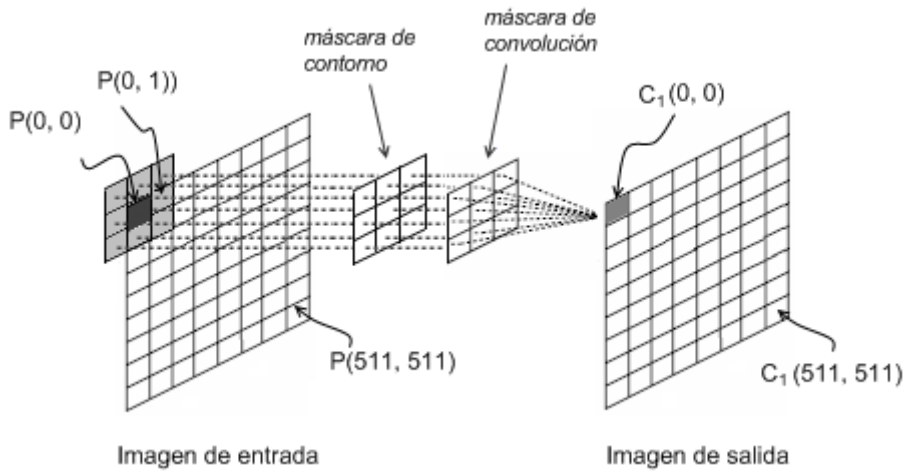

Figura 5.10. Convolución 3x3 considerando condiciones de contorno.

[7] En las multiplicaciones se ha invertido el orden natural de los coeficientes de la máscara de convolución de acuerdo a la fórmula de la convolución lineal.

Con esta máscara binaria se realiza una operación AND elemento a elemento de la forma que se indica en la Figura 5.11:

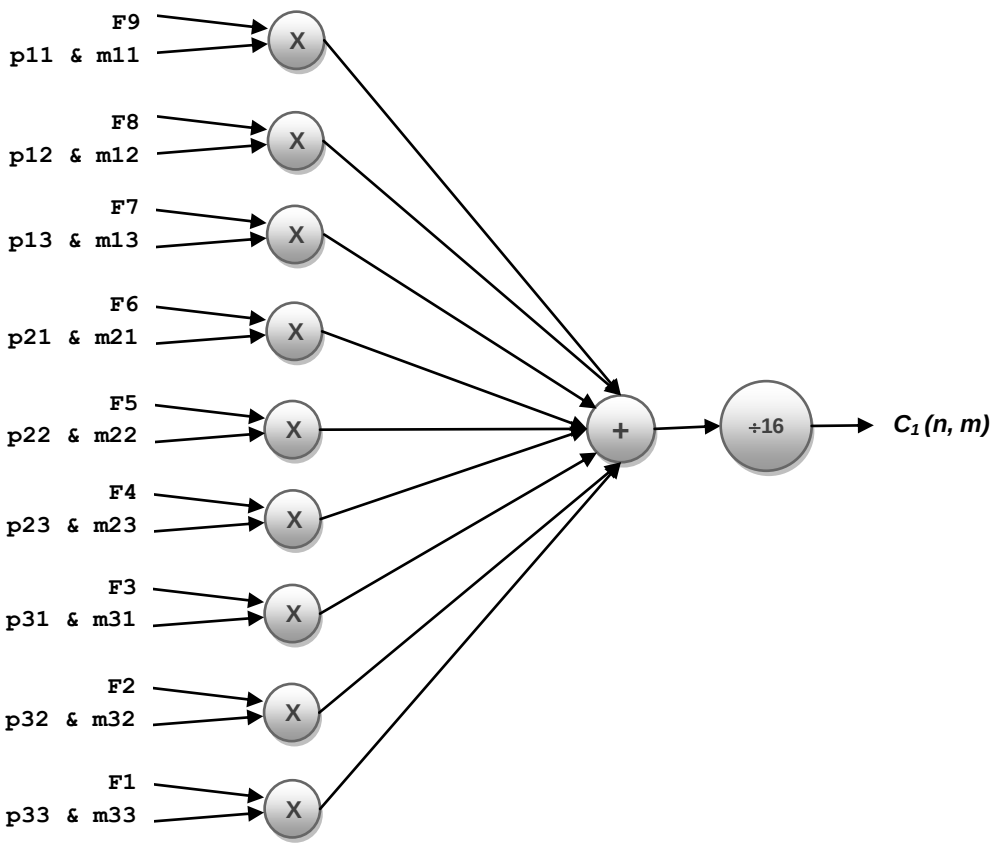

Figura 5.11. Cálculo de la convolución para el píxel contenido en la variable `p22`.

Como ya se ha comentado, antes de comenzar a procesar la primera imagen se requiere una fase de inicialización en el proceso *columnsT*, que permita el envío de una columna por ciclo de reloj. Esta fase consiste en almacenar filas de la imagen hasta que la fila del píxel a procesar se sitúe en el centro de la columna enviada al proceso *filterT*. Por lo tanto, el *array* `B[]` debe rellenarse con la primera fila de la imagen (ver Figura 5.12) de forma que en cuanto empiecen a llegar píxeles de la segunda fila, se pueda enviar la primera columna al proceso *filterT*. Los datos procedentes del *array* `A[]`serán indiferentes, puesto que la máscara binaria de condiciones de contorno hará que sea ignorado en los cálculos.

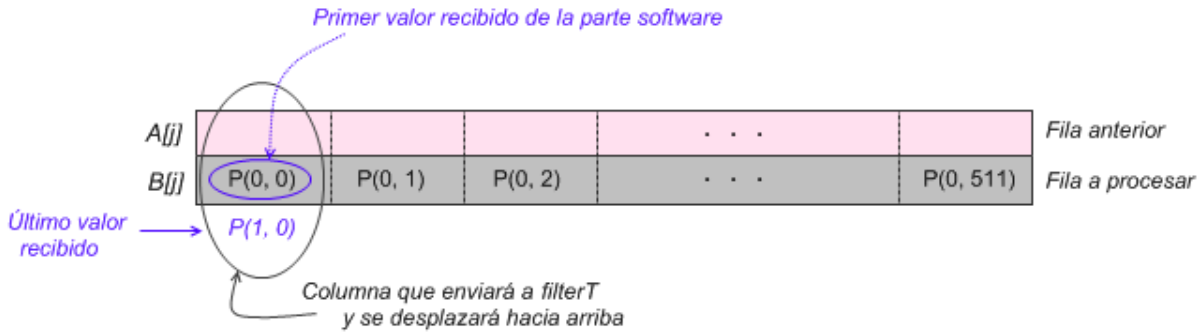

Figura 5.12. Inicialización del proceso *columnsT*.

5.2.2.1 Proceso ColumnsT

En esta sección se van a describir los segmentos de código más relevantes del proceso *columnsT*. Dicho proceso tiene como argumentos los 6 objetos de tipo `co_stream` mostrados en la siguiente línea de código:

```
void columnsT (co_stream input_stream, co_stream r0, co_stream r1, co_stream r2, co_stream
fila, co_stream columna){
```

Por el `input_stream` se recibirán los datos procedentes del software. Los *streams* de salida `r0`, `r1` y `r2` enviarán al proceso *filterT* la columna de 3 datos explicada anteriormente. Por último, los *streams* de salida fila y columna informarán a *filterT* de las coordenadas del píxel a procesar en ese ciclo de reloj, necesarias para obtener las condiciones de contorno.

La inicialización del proceso *columnsT* se muestra en la siguiente porción de código, donde mediante un bucle `for` se leen los paquetes de cuatro datos (ver sección 5.2.1) del software necesarios para rellenar el *array* `B[]`.

```
// Lee toda una fila del stream y se almacena en el array B[].
IF_SIM(cosim_logwindow_fwrite(log, "\n Valores almac en array B: ");)

for ( j = 0; j < WIDTH/4; j++ ){

     co_stream_read(input_stream, &vInter, sizeof(co_uint64));

     B[j*4] = (co_uint16)(vInter>>48);
     B[j*4+1] = (co_uint16)(vInter>>32);
     B[j*4+2] = (co_uint16)(vInter>>16);
     B[j*4+3] = (co_uint16)(vInter);

     IF_SIM(cosim_logwindow_fwrite(log," %d, %d, %d, %d ",B[j*4],B[j*4+1],B[j*4+2],B[j*4+3]);)
}
IF_SIM(cosim_logwindow_fwrite(log," \n ");)
```

El desempaquetado en cada iteración del bucle se realiza siguiendo el proceso inverso al de empaquetado descrito en la Figura 5.6.

A continuación comienzan los cálculos que se realizan durante el funcionamiento normal de la aplicación. Estas operaciones se sitúan dentro de una estructura `do{ … } while(err==co_err_none)`, que funcionará mientras se reciban datos. Mediante los bucles `for(i)` y `for(j)` se lleva la cuenta de los píxeles recibidos de cada imagen.

La primera operación a realizar es el cálculo de las coordenadas del píxel a procesar, que se almacenarán en las variables `ip` y `jp` y se envían al proceso *filterT*.

```
do {
   for ( i=0; i < HEIGHT; i++ ) {
       for (j=0; j < WIDTH; j++) {

//Envía a filterT la fila y columna del pixel a procesar para calcular la máscara de contorno.
               if(j>0) {ip=i; jp=j-1;}
               else if(j<1 & i!=0) {ip=i-1; jp=WIDTH-1; }
               else {ip=HEIGHT-1; jp=WIDTH-1;}

               co_stream_write(fila, &ip, sizeof(co_uint16));
               co_stream_write(columna, &jp, sizeof(co_uint16));
```

A continuación se rellena la columna de tres datos que se envía al proceso *filterT*, la cual se identifica a través de las variables `p13`, `p23` y `p33`. En cada iteración las variables `p13` y `p23` se actualizarán con el elemento `j` de los *arrays* `A[]` y `B[]` respectivamente. La variable `p33` se actualizará con el dato adecuado del último paquete recibido del software. La lectura del `input_stream` se realizará cada vez que el contador `itera` se reinicie a cero, de forma que se reciba un nuevo paquete en cuanto se hallan enviado los 4 datos del paquete anterior al proceso *filterT*.

Por último se enviará la columna de tres datos (`p13`, `p23`, `p33`) al proceso *filterT* a través de los *streams* `r0`, `r1` y `r2`, y se actualizan los *arrays* `A[]` y `B[]` de la forma indicada en la Figura 5.8. La directiva de preprocesador `#pragma CO PILELINE`, descrita en la sección 3.1.2, fuerza a solapar las instrucciones de código en medida de lo posible para que se realicen en el menor número de ciclos de reloj.

```
                #pragma CO PIPELINE
                #pragma CO set stageDelay 64

                p13 = A[j];
                p23 = B[j];

                if(itera==0) {err=co_stream_read(input_stream, &vInter, sizeof(co_uint64));}

                p33= (co_uint16)(vInter>>((3-itera)*16));
                itera++; if (itera >3) itera=0;

                co_stream_write(r0, &p13, sizeof(co_uint16));
                co_stream_write(r1, &p23, sizeof(co_uint16));
                co_stream_write(r2, &p33, sizeof(co_uint16));

        //Desplazamiento de los buffers.
                A[j] = p23;
                B[j] = p33;
        }
    }
} while (err==co_err_none);
```

5.2.2.2 Proceso filterT

En esta sección se van a describir los segmentos de código más relevantes del proceso *filterT*. Dicho proceso tiene como argumentos los 6 objetos de tipo `co_stream` mostrados en la siguiente línea de código:

```
void filterT (co_stream fila, co_stream columna, co_stream r0, co_stream r1, co_stream r2,
co_stream sal_filT){
```

Los *streams* de entrada `fila` y `columna` contendrán las coordenadas del siguiente píxel a procesar. Los *streams* de entrada `r0`, `r1` y `r2` contendrán la columna de 3 datos que completa la matriz 3x3 que se multiplica por la máscara de convolución para el cálculo del píxel indicado por los *streams* `fila` y `columna`. El *stream* de salida `sal_filT` enviará el resultado de la convolución del píxel anterior al proceso *Empaq*.

Los cálculos que se realizan en modo de funcionamiento normal se sitúan dentro de una estructura `do{ … } while(1)`, que funcionará mientras se reciban datos. Las directivas `#pragma CO PIPELINE` y `#pragma CO set stageDelay`, descritas en la sección 3.1.2, se incluyen para forzar el *pipeline* en el hardware.

Se comienza leyendo los *streams* de entrada `r0`, `r1`, `r2`, `fila` y `columna`, almacenando su valor en las variables locales `data0`, `data1`, `data2`, `fil` y `col` respectivamente.

```
do {
        #pragma CO PIPELINE
        #pragma CO set stageDelay 40  //Máx. ud de retardo=35  y 9 etapas.

        //Almacena en dataX los datos q lee de los stream de entrada.
       err  = co_stream_read(r0, &data0, sizeof(co_uint16));
       err &= co_stream_read(r1, &data1, sizeof(co_uint16));
       err &= co_stream_read(r2, &data2, sizeof(co_uint16));
       err &= co_stream_read(fila, &fil, sizeof(co_uint16)); //Lee cuando no está vacío.
       err &= co_stream_read(columna, &col, sizeof(co_uint16));
       if (err != co_err_none) break;

        IF_SIM(cosim_logwindow_fwrite(log, " \n Posicion fil col leida: %d, %d", fil,col);)
```

A continuación se calculan los coeficientes de la máscara binaria de condiciones de contorno. Para el caso de la matriz 3x3 las condiciones de contorno afectarán a la primera y última columna de la imagen, así como a la primera y última fila de la imagen. Así, si el píxel a procesar pertenece a la primera columna (`col==0`), la máscara binaria pondrá a cero los coeficientes de su primera columna (`m11=m21=m31=0`). Si el píxel es de la última columna (`col==WIDTH-1`), la máscara pondrá su última columna a cero (`m13=m23=m33=0`). En cuanto a las condiciones de contorno de las filas de la imagen, si el píxel pertenece a la primera, la máscara pondrá a cero los coeficientes de su primera fila (`m11=m12=m13=0`). Por último, si el píxel pertenece a la última fila, la máscara pondrá los coeficientes de su última fila (`m31=m32=m33=0`) a cero.

```
//MÁSCARA:
//-------------------------------------------------------------------------------------------
        m11 = 0xFFFF; m12 = 0xFFFF; m13 = 0xFFFF;
        m21 = 0xFFFF; m22 = 0xFFFF; m23 = 0xFFFF;
        m31 = 0xFFFF; m32 = 0xFFFF; m33 = 0xFFFF;

        if (col==0) {  //cond contorno izqda. (011)
                              m11 = 0;
                              m21 = 0;
                              m31 = 0;
        }
        else if (col==WIDTH-1) {   //cond contorno drcha. (110)
                              m13 = 0;
                              m23 = 0;
                              m33 = 0;
        }
        if (fil==0) { m11 = 0; m12 = 0; m13 = 0; }
        else if (fil==HEIGHT-1){ m31 = 0; m32 = 0; m33 = 0; }
```

Una vez calculados los coeficientes de la máscara de contorno, se actualizan los elementos de la matriz de datos (`p11`, `p12`,..., `p33`), añadiendo en su última columna, la recibida del proceso *columnsT*. Para empezar a calcular resultados de la convolución, el píxel a procesar debe encontrarse en el centro de la matriz de datos (esto es en la variable `p22`), lo que ocurre a partir de que la variable `iter` sea mayor que 1. La división por 16 de la Figura 5.11 se implementa mediante el desplazamiento lógico a la derecha insertando 4 ceros. Por último, se envía el resultado obtenido, almacenado en la variable `result`, al proceso *Empaq* a través del *stream* `sal_filT`.

```
//Desplaza la matriz P a la izqda para meter una nueva col x la drcha. Se pierde la 1ªcol.
        p11 = p12; p12 = p13;
        p21 = p22; p22 = p23;
        p31 = p32; p32 = p33;

//Valores de la última columna (j=3)
        p13 = data0;
        p23 = data1;
        p33 = data2;

//Como entran los datos ordenados del primer al último pixel, hay que dar la vuelta al filtro:
       if(iter>1){
              sop =   ((p11&m11)*F9 + (p12&m12)*F8 + (p13&m13)*F7
                      + (p21&m21)*F6 + (p22&m22)*F5 + (p23&m23)*F4
                      + (p31&m31)*F3 + (p32&m32)*F2 + (p33&m33)*F1)>>4;

              result=(co_uint16)sop;
              //IF_SIM(cosim_logwindow_fwrite(log, " \n result_c1=: %d", result);)
              co_stream_write(sal_filT, &result, sizeof(co_uint16));
              iter++;
       }
       else iter++;
}while(1);
```

5.2.2.3 Proceso Empaq

En esta sección se van a describir los segmentos de código más relevantes del proceso *Empaq*. Dicho proceso tiene como argumentos 2 objetos de tipo `co_stream` mostrados en la siguiente línea de código:

```
void Empaq(co_stream sal_filT, co_stream output_stream){
```

El *stream* `sal_filT` contendrá el resultado calculado en el proceso *filterT*. El *stream* de salida `output_stream` contendrá el paquete de 4 datos que se enviarán al proceso software *prodcon*.

Los cálculos que se realizan en modo de funcionamiento normal se sitúan dentro de una estructura `do{ … } while(1)`, que funcionará mientras se reciban datos. Las directivas `#pragma CO PIPELINE` descrita en la sección 3.1.2, se incluye para forzar el *pipeline* en el hardware.

Los datos procedentes del proceso *filterT* se almacenan en la variable `sal_fil`. A continuación se inserta el dato en la posición del paquete actual en la posición marcada por la variable `k`, siguiendo el proceso de empaquetado descrito en la sección 5.2.1. Cada nuevo paquete se almacenará en la variable `result`, que una vez sea completado (`k==3`) se enviará al proceso software.

```
do{
       #pragma CO PIPELINE

       if(co_stream_read(sal_filT, &sal_fil, sizeof(co_uint16))==co_err_none){

              //EMPAQUETA
              if(k==0) {s0=(co_uint16) sal_fil; k++;}
              else if (k==1) {s1=(co_uint16) sal_fil; k++;}
              else if(k==2) {s2=(co_uint16) sal_fil; k++;}
              else if(k==3)  {
                             k=0;
                             s3=(co_uint16) sal_fil;
                             result=
```

```
                                        ( ((co_uint64)s0)<<48) |
                                        ( ((co_uint64)s1<<32) ) |
                                        ( ((co_uint64)s2<<16) ) |
                                        ( (co_uint64)s3);

co_stream_write(output_stream, &result, sizeof(co_uint64));  //Escribe el pixel filtrado.
IF_SIM(cosim_logwindow_fwrite(log, " \n result=: %d", result);)
              }
       }

} while(1);
```

5.2.2.4 Función de Configuración

En la función de configuración se declaran y se crean los *streams* y procesos involucrados en la arquitectura del sistema, siguiendo la metodología descrita bajo la Figura 3.14.

En la creación de los *streams* se establece el ancho y profundidad de los búferes que implementan, mediante variables definidas en el fichero "Mi_proyecto.h" del proyecto. La variable `STREAMWIDTH` define un ancho de 64 bits para los *streams* que realizan la comunicación entre procesos sw/hw y hw/sw. La variable `STREAMWIDTHi`, cuyo valor es de 16, define el ancho en bits de los *streams* que comunican procesos hw/hw. Por último, la variable `STREAMDEPTH` establece la profundidad de los *streams* a 512, valor que posibilitará el envío a ráfagas. Las consideraciones relativas al envío de ráfagas se indicarán en el Capítulo 6 de de pruebas y resultados.

```
input_stream = co_stream_create(" input_stream", INT_TYPE(STREAMWIDTH),STREAMDEPTH);
fila = co_stream_create("fila", UINT_TYPE(16),STREAMDEPTH);//De columnsT a filterT.
columna = co_stream_create("columna", UINT_TYPE(16),STREAMDEPTH);  //De columnsT a filterT.
r0 = co_stream_create("r0", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
r1 = co_stream_create("r1", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
r2 = co_stream_create("r2", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
sal_filT = co_stream_create("sal_filT", INT_TYPE(STREAMWIDTHi),STREAMDEPTH);
output_stream = co_stream_create("output_stream", INT_TYPE(STREAMWIDTH),STREAMDEPTH);
```

Los procesos se conectan a través de los *streams* formando la arquitectura descrita en la Figura 5.4.

```
columnsT_process = co_process_create("columnsT", (co_function)columnsT, 6, input_stream, r0,
r1, r2, fila, columna);

filterT_process = co_process_create("filterT", (co_function)filterT, 6, fila, columna, r0, r1,
r2, sal_filT);

Empaq_process = co_process_create("Empaq", (co_function)Empaq, 2, sal_filT, output_stream);

prodcon_process = co_process_create("prodcon", (co_function)prodcon, 2, input_stream,
output_stream);

co_process_config(columnsT_process, co_loc, "PE0");
co_process_config(filterT_process, co_loc, "PE0");
co_process_config(Empaq_process, co_loc, "PE0");
```

A continuación se usa la función `co_stream_config()` para asignar "`bram`" al atributo `co_kind`, que define los atributos físicos de los *streams*. De esta forma, cuando el proyecto en ISE genere la memoria necesaria para los *streams* (FIFOs) lo hará utilizando memoria de tipo *blockram*, en lugar de memoria distribuida. Esta última, a diferencia de la anterior, no utiliza memorias reales existentes en

la FPGA, sino que se construye a base de otros recursos hardware, por lo que para evitar posibles problemas, es más recomendable utilizar *blockram*.

```
co_stream_config(input_stream, co_kind , "bram");
co_stream_config(fila, co_kind , "bram");
co_stream_config(columna, co_kind , "bram");
co_stream_config(r0, co_kind , "bram");
co_stream_config(r1, co_kind , "bram");
co_stream_config(r2, co_kind , "bram");
co_stream_config(sal_filT, co_kind , "bram");
co_stream_config(output_stream, co_kind , "bram");
```

Especialmente esto es conveniente si se va a realizar un envío a ráfagas, ya que de esta forma se conseguiría un funcionamiento más rápido de la memoria.

5.3. Implementación del proyecto completo en CoDeveloper para el cálculo de la suma de las dos primeras escalas *Wavelet à trous*

En este apartado se detallará el procedimiento seguido a la hora de programar en CoDeveloper, una arquitectura que implemente el diseño del algoritmo descrito en el esquema lógico del apartado 5.1. Se incluirán por tanto indicaciones relativas al modelo de programación de ImpulseC, así como justificaciones que aclaren el por qué de esta implementación.

Dada la extensión de este capítulo, y con el fin de no perder la perspectiva, se comenzará exponiendo la arquitectura final implementada (ver Figura 5.13), y se irá describiendo la programación conforme al procesado que sigue el flujo de datos. Esto no significa, que éste haya sido el orden llevado a cabo en la realización, ya que en arquitecturas complejas es más sencillo y prudente dividir el problema en cómputos sencillos, y una vez comprobados los resultados intermedios, unirlos para realizar el procesado completo. No obstante, se tratará de indicar y justificar cada decisión tomada, con el fin de orientar y prevenir de errores sufridos, en futuros trabajos.

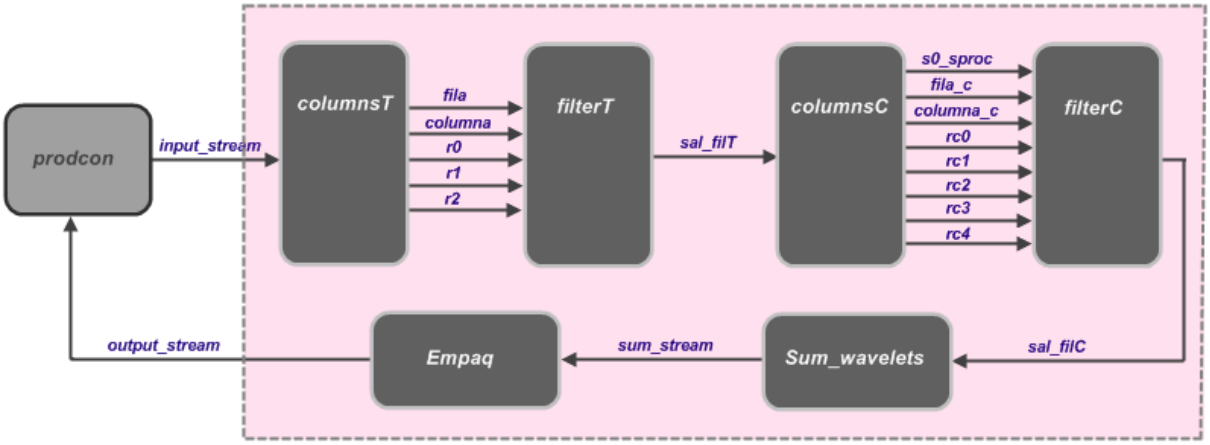

Figura 5.13. Arquitectura implementada que muestra los procesos y *stream* utilizados en la consecución del algoritmo que calcula la suma de las dos primeras escalas *wavelet*.

A groso modo se observa que la parte hardware consta de dos pares de procesos *columns*X-*filter*X, donde la X hace referencia al tamaño del *kernel* de convolución. El primer par, *columnsT* y *filterT*, corresponde a la convolución con un *kernel* de tamaño 3x3 descrita en la sección 5.2, con la salvedad de que el *stream* de salida se conectará al segundo par, *columnsC* y *filterC*, encargado de la convolución con un *kernel* de 5x5 coeficientes. Otro proceso, denominado *Sum_wavelets* calculará cada una de las escalas *wavelet* y la suma de ambas. Por último, se incluye un proceso *Empaq* que preparará los resultados recibidos de *Sum_wavelets* para enviarlos al proceso software, el cual escribirá los ficheros de salida con los datos procesados.

Cabe resaltar, que la principal dificultad radica en conseguir un funcionamiento para un flujo continuo de datos en tiempo real, por lo que el hardware deberá optimizarse al máximo con el fin de conseguir un resultado por ciclo de reloj, sin tener que esperar entre una imagen y la siguiente.

5.3.1. Descripción del proceso software

El proceso software será el mismo que el descrito para la implementación de una convolución genérica 3x3, que se puede consultar en la sección 5.2.1.

5.3.2. Implementación de los procesos que realizan la convolución 3x3 para el cálculo de la imagen aproximación de la primera escala de la DWT à trous

La imagen aproximación de la primera escala de la DWT *à trous*, denominada C_1, se obtiene mediante la convolución con el kernel de tamaño 3x3 de la ecuación de la Figura 2.7. En la sección 5.2 se describe el código que implementa los procesos *columnsT* y *filerT* utilizados para realizar dicha convolución.

En la ecuación (20) se muestra el valor de las variables (`F1`, `F2`,…,`F9`) utilizadas en el proceso *filterT* , que representan los coeficientes del filtro, de acuerdo a la nomenclatura descrita en la sección 5.2.

$$h_{3x3} = \frac{1}{16} \cdot \begin{pmatrix} \texttt{F1} & \texttt{F2} & \texttt{F3} \\ \texttt{F4} & \texttt{F5} & \texttt{F6} \\ \texttt{F7} & \texttt{F8} & \texttt{F9} \end{pmatrix} = \frac{1}{16} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{20}$$

En la Figura 5.3 se observa que para el cálculo de los coeficientes wavelet de la primera escala se necesitan los píxeles de la imagen original (C_0). Este cálculo se lleva a cabo en el proceso *Sum_wavelet* tal como se indicó anteriormente. El píxel de la imagen $C_0(n,m)$ y el resultado $C_1(n,m)$, deben estar disponibles simultáneamente en dicho proceso. Para evitar problemas de sincronización se ha optado por que los píxeles de la imagen de entrada recorran el mismo camino que los resultados de la convolución correspondientes a los mismos. Por tanto, el píxel de la imagen de entrada deberá atravesar los procesos intermedios hasta llegar al proceso *Sum_wavelet,* de acuerdo al esquema de procesos mostrado en la Figura 5.13.

Para asegurar dicha sincronización, en el *stream* de salida `sal_filT` se ha empaquetado el resultado de la convolución, almacenado en la variable `result`, junto con el píxel para el cual se ha obtenido dicho resultado, disponible en la variable `p22`.

```
sal_fT= (((co_uint32)p22)<<16) | (((co_uint32)result)&0xff);
co_stream_write(sal_filT, &sal_fT, sizeof(co_uint32));
```

El *stream* de salida `sal_fT` del proceso *filterT* se redefine con un ancho de 32 bits en lugar de los 16 indicados en la sección 0, de forma que acomode el valor de las dos variables.

5.3.3. Implementación de los procesos que realizan la convolución 5x5 para el cálculo de la imagen aproximación de la segunda escala de la DWT à trous

En esta sección se va a describir el procedimiento realizado para calcular la imagen aproximación de la segunda escala de la DWT *à trous*, denominada C_2. Dicha imagen se calcula como la convolución de la aproximación de la primera escala por el *kernel* de tamaño 5x5 de la Figura 2.7. De forma análoga a la convolución 3x3 se definen los procesos *columnsC* y *filterC* para realizar dicho cálculo.

En la siguiente figura se muestran los cálculos realizados para obtener el resultado de la convolución para el píxel $C_1(n,m)$, en la que las variables F00, F01, …, F44 definen los coeficientes del filtro.

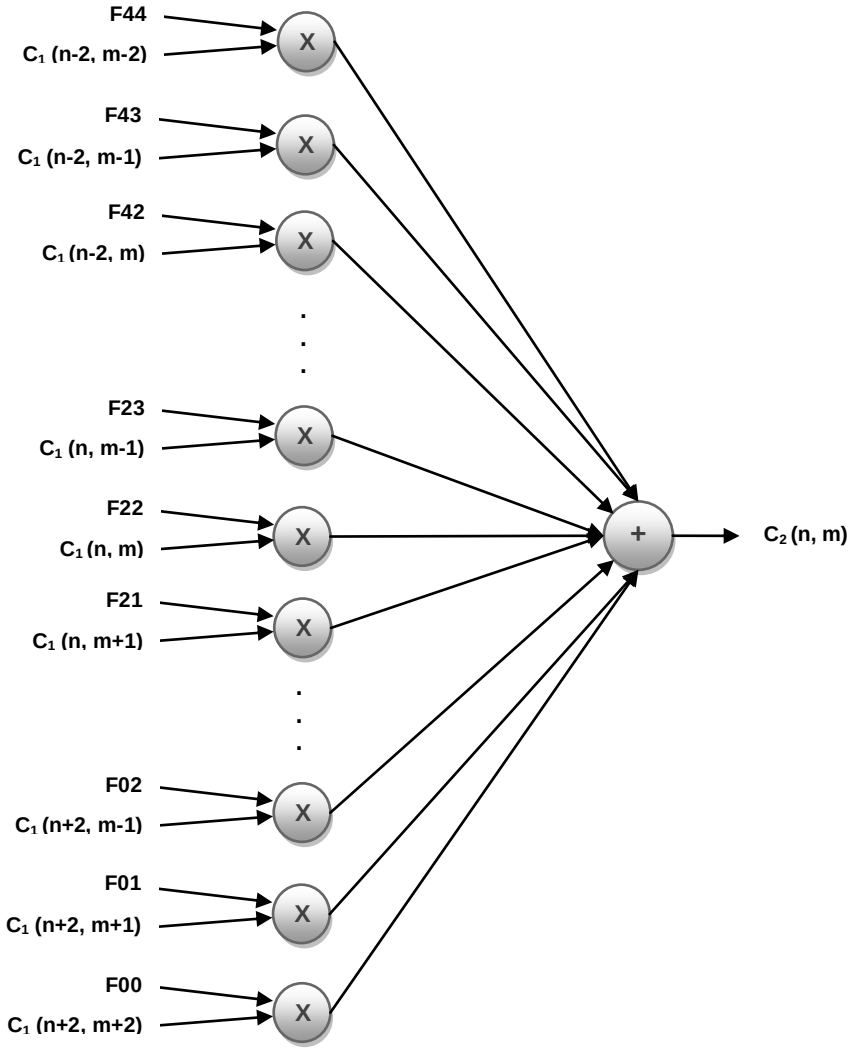

Figura 5.14. Cálculo genérico de una convolución 5x5 para el píxel $C_1(n, m)$.

Como se observa en la Figura 2.7, los coeficientes del *kernel* de convolución h_{5x5} para calcular la aproximación de esta escala, se obtienen insertando ceros entre los coeficientes del *kernel* utilizado en la escala anterior, denominado h_{3x3}, de forma que el valor central esté a distancia 2 del resto de coeficientes. En la siguiente ecuación se muestran los coeficientes para el nuevo *kernel* de tamaño 5x5, además de su relación con los coeficientes del *kernel* 3x3, utilizados en el proceso *filterT*.

$$h_{5x5} = \begin{pmatrix} F00 & F01 & F02 & F03 & F04 \\ F10 & F11 & F12 & F13 & F14 \\ F20 & F21 & F22 & F23 & F24 \\ F30 & F31 & F32 & F33 & F34 \\ F40 & F41 & F42 & F43 & F44 \end{pmatrix} = \begin{pmatrix} \frac{1}{16} & 0 & \frac{1}{8} & 0 & \frac{1}{16} \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{8} & 0 & \frac{1}{4} & 0 & \frac{1}{8} \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{16} & 0 & \frac{1}{8} & 0 & \frac{1}{16} \end{pmatrix} = \frac{1}{16} \cdot \begin{pmatrix} F1 & 0 & F2 & 0 & F3 \\ 0 & 0 & 0 & 0 & 0 \\ F4 & 0 & F5 & 0 & F6 \\ 0 & 0 & 0 & 0 & 0 \\ F7 & 0 & F8 & 0 & F9 \end{pmatrix} \quad (21)$$

De forma análoga al proceso *columnsT*, el proceso *columnsC* prepara columnas de 5 datos (en este caso correspondientes a los coeficientes C_1), que serán enviadas al proceso *filterC* para el cálculo de la convolución.

En el modo de operación normal (una vez superada la inicialización), el proceso *columnsC* tendrá que tener almacenadas 4 filas de la imagen, de forma que para cada nuevo dato recibido (píxel $C_1(n+2,m+2)$) del proceso *filterT*, pueda enviar una columna de 5 datos al proceso *filterC*. En la Figura 5.15 se muestra de forma gráfica el procedimiento seguido. Las 4 filas necesarias en el proceso *columnsC* se almacenarán en cuatro *arrays* `A[]`, `B[]`, `C[]` y `D[]` de 512 elementos (ancho de la imagen).

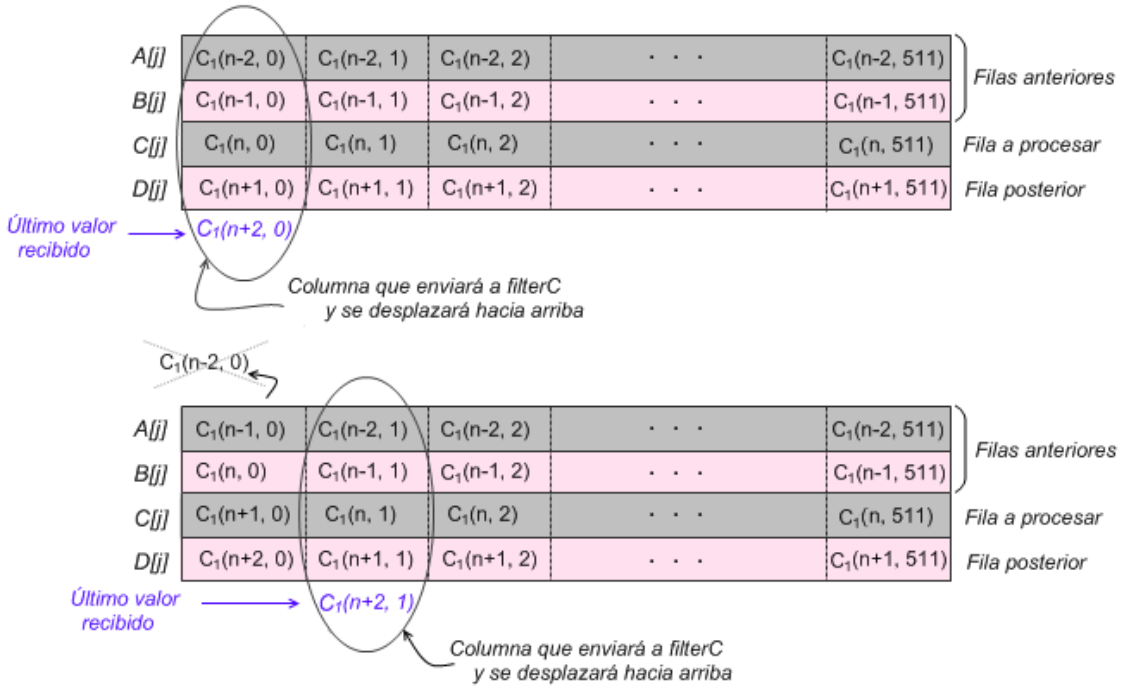

Figura 5.15. Actualización de los *arrays* en el proceso *columnsC*.

Por su parte, el proceso *filterC* tendrá almacenados en 25 variables, los píxeles que se multiplicarán por la máscara de convolución en cada ciclo. Así, almacenará cada nueva columna de 5 píxeles recibida, en las variables `data0`, `data1`, `data2`, `data3` y `data4`, y se actualizarán las variables que contienen los 20 pixeles recibidos en los 4 ciclos anteriores, para que la suma de productos produzca el resultado del píxel $C_1(n,m)$ correspondiente. Dichas variables se denominan `p00`, `p01`,..., `p44`. La actualización de las variables sigue el siguiente esquema:

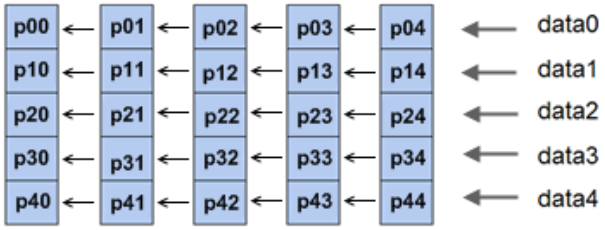

Figura 5.16. Actualización de la matriz de datos en el proceso *filterC*.

A continuación se lleva a cabo la suma de productos relativa a la convolución. Aprovechando la relación entre las matrices de coeficientes h_{3x3} y h_{5x5} mostrada en la ecuación (21), se puede

simplificar de 25 a 9 el número de productos siguiendo el siguiente esquema, en el que se mantiene la nomenclatura de los coeficientes del proceso *filterT*.

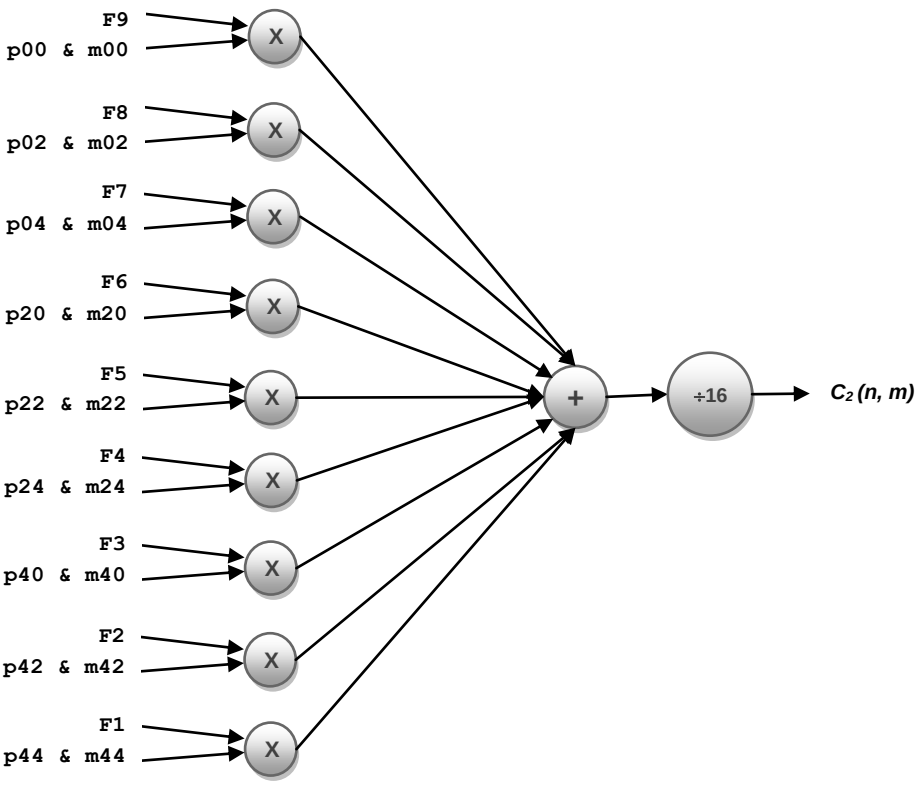

Figura 5.17. Cálculo de la convolución en el proceso *filterC*.

Análogamente a la convolución 3x3, las variables `m00`, `m01`, …, `m44`, constituyen una máscara binaria de condiciones de contorno para tratar con los píxeles que bordean la imagen, que deciden qué elementos se multiplicarán por los coeficientes del filtro en cada ciclo de reloj. Dicha máscara se calcula teniendo en cuenta que el nuevo *kernel* de convolución es de tamaño 5x5.

Al igual que en el proceso *columnsT*, antes de comenzar a procesar la primera imagen se requiere una fase de inicialización en el proceso *columnsC*, que permita el envío de una columna por ciclo de reloj. Esta fase consiste en almacenar filas de la imagen hasta que la fila del píxel a procesar se sitúe en el centro de la columna enviada al proceso *filterC*. Por lo tanto, el *array* `C[]` debe rellenarse con la primera fila de la imagen C_1 y el *array* `D[]` con la segunda (ver Figura 5.18), de forma que en cuanto empiecen a llegar píxeles de la tercera fila, se pueda enviar la primera columna al proceso *filterC*. Los datos procedentes de los *arrays* `A[]`y `B[]` serán indiferentes, puesto que la máscara binaria de condiciones de contorno hará que sean ignorados en los cálculos.

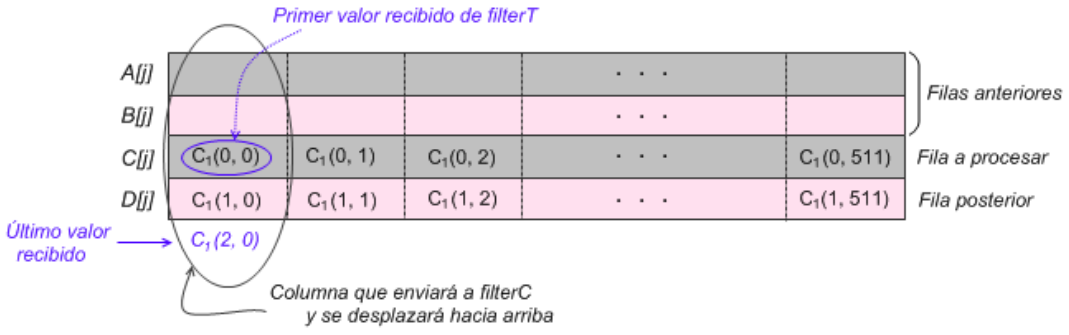

Figura 5.18. Inicialización de los *arrays* en el proceso *columnsC*.

5.3.3.1 Proceso columnsC

En esta sección se van a describir los segmentos de código más relevantes del proceso *columnsC*. Dicho proceso tiene como argumentos los 9 objetos de tipo `co_stream` mostrados en la siguiente línea de código:

```
void columnsC (co_stream sal_filT, co_stream rc0, co_stream rc1, co_stream rc2, co_stream rc3,
               co_stream rc4, co_stream fila_c, co_stream columna_c, co_stream c0_sproc)
```

El *stream* de entrada `sal_filT` contendrá el píxel de la imagen C_1, empaquetado junto con el píxel correspondiente de la imagen inicial C_0, tal y como se indicó en la sección 5.3.2. Los *streams* de salilda `rc0`, `rc1`, `rc2`, `rc3`, `rc4`, `fila_c`, `columna_c` y `c0_sproc` contendrán respectivamente la columna de 5 píxeles que se enviará al proceso *filterC*, las coordenadas del píxel C_1(n,m) a procesar en dicho proceso y el píxel correspondiente de la imagen inicial C_0.

En las siguientes líneas de código se muestra la inicialización de los *arrays* `C[]` y `D[]`indicada en la Figura 5.18. Como se ha recordado en el párrafo anterior, el *stream* `sal_filT` no sólo contiene el píxel de la imagen C_1, sino que empaqueta con éste el píxel correspondiente de la imagen de entrada C_0. Por este motivo, se definen los *arrays* `C0a[]` y `C0b[]`, que almacenarán las dos primeras filas de la imagen C_0 correspondientes a los *arrays* `C[]` y `D[]`.

```
// Lee toda una fila ( WIDTH píxeles) del stream y se almacena en el array C[].
    for ( j = 0; j < WIDTH; j++ ){
                err = co_stream_read(sal_filT, &vInter, sizeof(co_uint32));
                C[j] = (co_uint16)(vInter);
                C0a[j] = (co_uint16)(vInter>>16);
                if (err != co_err_none) break;
        }

// Lee la 2ª fila de la imagen y la almacena en D[].
    for ( j = 0; j < WIDTH; j++ ){
                err = co_stream_read(sal_filT, &vInter, sizeof(co_uint32));
                D[j] = (co_uint16)(vInter);
                C0b[j] = (co_uint16)(vInter>>16);
                if (err != co_err_none) break;
        }
```

Los cálculos que se realizan en estado de funcionamiento normal se sitúan dentro de una estructura `do{ … } while(err==co_err_none)`, que funcionará mientras se reciban datos. Mediante los bucles `for(i)` y `for(j)` se lleva la cuenta de los píxeles recibidos de cada imagen. La primera operación a realizar es el cálculo de las coordenadas del píxel a procesar, que se almacenarán en las variables `ip` y `jp` y se envían al proceso *filterC*, teniendo en cuenta el nuevo tamaño del *kernel* de convolución.

La directiva de preprocesador `#pragma CO PILELINE` fuerza a solapar las instrucciones de código en medida de lo posible para que se realicen en el menor número de ciclos de reloj. Esta directiva se describe en la sección 3.1.2.

A continuación se rellenará la columna de cinco datos que se envía al proceso *filterC*, la cual se identifica a través de las variables `p04`, `p14`, `p24`, `p34` y `p44`. De la misma forma, se definen las variables `C01`, `C02` y `C03`, que almacenan la columna de píxeles de la imagen C_0 correspondiente a los datos C_1 de las variables `p24`, `p34` y `p44`. Esta columna, únicamente se utiliza para sincronizar el dato C_0 almacenado en `C01`, con su resultado C_1 correspondiente almacenado en la variable `p24` (elemento central de la columna enviada al proceso *filterC*).

En las siguientes líneas de código se muestra la actualización de las variables `p04`, `p14`, `p24` y `p34` con el elemento `j` de los *arrays* correspondientes en cada iteración. Se actualizarán de igual forma las variables `C01` y `C02`, con el elemento `j` de los *arrays* `C0a[]` y `C0b[]`.La variable `p44` se actualizará con el dato C_1 adecuado del último paquete recibido del proceso *filterT* y la variable `C03` se actualizará con el dato C_0 recibido en dicho paquete.

```
#pragma CO PIPELINE
#pragma CO set stageDelay 17

p04 = A[j];
p14 = B[j];
p24 = C[j];  // Éste es mi píxel central c1, q al mult x la máscara dará c2.
p34 = D[j];
c01=C0a[j];
c02=C0b[j];

err = co_stream_read(sal_filT, &vInter, sizeof(co_uint32));
p44 = (co_uint16)(vInter);
c03 = (co_uint16)(vInter>>16);
```

Por último se enviará en un mismo ciclo de reloj, la columna de cinco datos (`p04`, `p14`, `p24`, `p34` y `p44`) y el dato almacenado en la variable `C01` al proceso *filterC* a través de los *streams* `rc0`, `rc1`, `rc2`, `rc3`, `rc4`, y `c0_sproc` respectivamente. Esto se describe en las siguientes líneas de código.

```
co_stream_write(rc0, &p04, sizeof(co_uint16));
co_stream_write(rc1, &p14, sizeof(co_uint16));
co_stream_write(rc2, &p24, sizeof(co_uint16));  // El central.
co_stream_write(rc3, &p34, sizeof(co_uint16));
co_stream_write(rc4, &p44, sizeof(co_uint16));
co_stream_write(c0_sproc, &c01, sizeof(co_uint16));

//Actualización de los buffers.
                        A[j] = p14;
                        B[j] = p24;
                        C[j] = p34;
                        D[j] = p44;
                        C0a[j] =c02;
                        C0b[j] =c03;
```

Al final del código anterior se muestra la actualización de los *arrays* `A[]`, `B[]`, `C[]` y `D[]`, tal y como se indicó en la Figura 5.15. Al mismo tiempo y de forma análoga se actualizan también los *arrays* `C0a[]` y `C0b[]`.

5.3.3.2 Proceso filterC

En esta sección se van a describir los segmentos de código más relevantes del proceso *filterC*. Dicho proceso tiene como argumentos los 9 objetos de tipo `co_stream` mostrados en la siguiente línea de código:

```
void filterC(co_stream fila_c, co_stream columna_c, co_stream c0_sproc, co_stream rc0,
co_stream rc1, co_stream rc2, co_stream rc3, co_stream rc4, co_stream sal_filC){
```

Los *streams* de entrada `fila_c` y `columna_c` contendrán las coordenadas del siguiente píxel a procesar. El *stream* de entrada `c0_sproc` contiene el píxel de la imagen inicial C_0, cuyo resultado de la convolución de 3x3 se encuentra en el píxel central de la columna de 5 datos recibida en los *streams* de entrada `rc0`, `rc1`, `rc2`, `rc3` y `rc4`. Dicha columna completa la matriz 5x5 que se

multiplica por la máscara de convolución para el cálculo del píxel indicado por los *streams* `fila_c` y `columna_c`. El *stream* de salida `sal_filC` enviará el resultado de la convolución del píxel anterior al proceso *Empaq*.

Los cálculos que se realizan en modo de funcionamiento normal se sitúan dentro de una estructura `do{ … } while(1)`, que funcionará mientras se reciban datos. Las directivas `#pragma CO PIPELINE` y `#pragma CO set stageDelay`, descritas en la sección 3.1.2, se incluyen para forzar el *pipeline* en el hardware.

Se comienza leyendo simultáneamente los *streams* de entrada `rc0`, `rc1`, `rc2`, `rc3`, `rc4`, `c0_sproc`, `fila_c` y `columna_c`, almacenando la lectura en las variables locales `data0`, `data1`, `data2`, `data3`, `data4`, `data_c0`, `fil` y `col` respectivamente.

A continuación se calculan los coeficientes de la máscara binaria de condiciones de contorno. Para el caso de la matriz 5x5 las condiciones de contorno afectarán a las dos primeras y últimas columnas de la imagen, así como a las dos primeras y últimas filas de la misma.

```
m00 = 0xFFFF; m01 = 0xFFFF; m02 = 0xFFFF; m03 = 0xFFFF; m04 = 0xFFFF;
m10 = 0xFFFF; m11 = 0xFFFF; m12 = 0xFFFF; m13 = 0xFFFF; m14 = 0xFFFF;
m20 = 0xFFFF; m21 = 0xFFFF; m22 = 0xFFFF; m23 = 0xFFFF; m24 = 0xFFFF;
m30 = 0xFFFF; m31 = 0xFFFF; m32 = 0xFFFF; m33 = 0xFFFF; m34 = 0xFFFF;
m40 = 0xFFFF; m41 = 0xFFFF; m42 = 0xFFFF; m43 = 0xFFFF; m44 = 0xFFFF;

if (col==0) {  //cond contorno izqda.
                                m00 = 0; m01 = 0;
                                m10 = 0; m11 = 0;
                                m20 = 0; m21 = 0;
                                m30 = 0; m31 = 0;
                                m40 = 0; m41 = 0;
                            }
else if (col==1) {     //cond contorno izqda.
                                m00 = 0;
                                m10 = 0;
                                m20 = 0;
                                m30 = 0;
                                m40 = 0;
                            }
else if (col==WIDTH-2) {      //cond contorno drcha.
                                m04 = 0;
                                m14 = 0;
                                m24 = 0;
                                m34 = 0;
                                m44 = 0;
                            }
else if (col==WIDTH-1) {      //cond contorno izqda.
                                m03 = 0; m04 = 0;
                                m13 = 0; m14 = 0;
                                m23 = 0; m24 = 0;
                                m33 = 0; m34 = 0;
                                m43 = 0; m44 = 0;
                            }
if (fil==0){
                                m00 = 0; m01 = 0; m02 = 0; m03 = 0; m04 = 0;
                                m10 = 0; m11 = 0; m12 = 0; m13 = 0; m14 = 0;
                            }
else if (fil==1){ m00 = 0; m01 = 0; m02 = 0; m03 = 0; m04 = 0;}
else if (fil==HEIGHT-2){ m40 = 0; m41 = 0; m42 = 0; m43 = 0; m44 = 0;}
else if (fil==HEIGHT-1){
                                m30 = 0; m31 = 0; m32 = 0; m33 = 0; m34 = 0;
                                m40 = 0; m41 = 0; m42 = 0; m43 = 0; m44 = 0;
                                }
```

Una vez calculados los coeficientes de la máscara de las condiciones de contorno, se actualizan los elementos de la matriz de datos (p00, p01,..., p44), añadiendo en su última columna, la recibida del proceso *columnsC*.

```
//Desplaza la matriz P a la izqda para meter una nueva columna x la drcha.

        p00 = p01; p01 = p02; p02 = p03; p03 = p04;
        p10 = p11; p11 = p12; p12 = p13; p13 = p14;
        p20 = p21; p21 = p22; p22 = p23; p23 = p24;
        p30 = p31; p31 = p32; p32 = p33; p33 = p34;
        p40 = p41; p41 = p42; p42 = p43; p43 = p44;

       c0a = c0b; c0b=c0c;

//Valores de la última columna (j=4)
        p04 = data0;
        p14 = data1;
        p24 = data2;
        p34 = data3;
        p44 = data4;

       c0c = data_c0;
```

Las variables c0a, c0b y c0c se utilizan para mantener la sincronización del píxel de la imagen inicial C_0 recibido en el *stream* c0_sproc, con la columna recibida (r0, r1, r2, r3 y r4) simultáneamente. Esto es así porque el píxel central de la misma (almacenado en la variable data2) corresponde al resultado de la convolución 3x3 para el dato leído del *stream* c0_sproc, y ambos serán requeridos simultáneamente en el proceso *Sum_wavelet* para el cálculo de la primera escala *wavelet* w1, como se indicará en dicho proceso. Por tanto, el valor leído del *stream* c0_sproc, almacenado en la variable data_c0, actualizará la variable c0c, y la actualización de las variables c0a y c0b se realizará al tiempo que la actualización de la matriz de datos.

Para empezar a calcular resultados de la convolución, el píxel a procesar debe encontrarse en el centro de la matriz de datos (esto es en la variable p22), lo que ocurre a partir de que la variable iter sea mayor que 2. La división por 16 de la Figura 5.17 se implementa mediante el desplazamiento lógico a la derecha insertando 4 ceros.

```
// Como entran los datos ordenados del primer al último pixel, hay q dar la vuelta al filtro:
       if(iter>2){

               sop =   ((p00&m00)*F9 + (p02&m02)*F8 + (p04&m04)*F7
                       + (p20&m20)*F6 + (p22&m22)*F5 + (p24&m24)*F4
                       + (p40&m40)*F3 + (p42&m42)*F2 + (p44&m44)*F1)>>4;

               b=(co_uint16)sop;

               //EMPAQUETO:

       sal_fC=(((co_uint48)c0a)<<32)|((((co_uint48)p22)&0xff)<<16)|(((co_uint48)b)&0xff);
       co_stream_write(sal_filC, &sal_fC, sizeof(co_uint48));
//IF_SIM(cosim_logwindow_fwrite(log, " \n result_c2 = %d, c1_sp=%d, c0_sp=%d  \n",b,p22,c0a);)
               iter++;

       }
       else iter++;

} while (1);
```

En el código anterior se observa que la variable `sal_fC` de 48 bits que se enviará al proceso *Empaq*, contiene tres datos empaquetados: `c0a`, `p22` y `b`. Éstos se corresponden con los datos $C_0(n, m)$, $C_1(n, m)$ y $C_2(n, m)$, respectivamente, de forma que el píxel de la imagen aproximación C_2, será el resultado de la convolución 5x5 para el píxel C_1, que a su vez será resultado de la convolución 3x3 para el píxel de la imagen inicial C_0.

5.3.4. Descripción del proceso Sum_wavelet

Este proceso es el encargado de calcular para cada píxel, las dos primeras escalas wavelet y la suma de ambas. Tiene como argumentos los dos objetos de tipo `co_stream` mostrados en la siguiente línea de código:

```
void Sum_wavelets (co_stream sal_filC, co_stream sum_stream){
```

Como se indicó en el esquema de la Figura 5.3, para el cálculo de la primera escala `w1` se restará a cada píxel de la imagen inicial C_0, el correspondiente de la imagen aproximación C_1 de la primera escala, y para el cálculo de `w2`, se restará a este último, el píxel correspondiente de la imagen aproximación C_2. Tal y como se ha procedido, estos 3 píxeles (`c0`, `c1` y `c2`) llegan empaquetados al proceso *Sum_wavelet* por el *stream* `sal_filC`, por lo que no hay necesidad de incluir ningún mecanismo que sincronice la llegada de los mismos, y pueden por tanto realizarse las operaciones en el mismo ciclo de reloj, una vez se desempaqueta el dato leído.

```
do{
        #pragma CO PIPELINE

        err  = co_stream_read(sal_filC, &vInter, sizeof(co_uint48));
        c2 = (co_uint16)(vInter);
        c1 = (co_uint16)(vInter>>16);
        c0 = (co_uint16)(vInter>>32);
         if (err != co_err_none) break;

        //Cálculo de las dos primeras escalas wavelet y la suma de ambas:
        w1=(co_int16)c0-(co_int16)c1;
        w2=(co_int16)c1-(co_int16)c2;
        suma = w1+w2;

        co_stream_write(sum_stream, &suma, sizeof(co_uint16));  //Escribe el pixel procesado.

}while(1);
```

Por último, se envía para cada píxel, el resultado de la suma de las dos primeras escalas al proceso *Empaq*, que preparará los datos para enviarlos al software.

5.3.5. Proceso Empaq

El proceso *Empaq* será análogo al descrito para la implementación de una convolución genérica 3x3, que se puede consultar en la sección 5.2.2.3. En esta arquitectura, leerá cada resultado del *stream* `sum_stream` procedente del proceso *Sum_wavelet* e irá empaquetando paquetes de cuatro datos que enviará al proceso software *prodcon* por el *stream* de salida `output_stream`.

5.3.6. Función de configuración

En la función de configuración se declaran y se crean los *streams* y procesos involucrados en la arquitectura del sistema, siguiendo la metodología descrita bajo la Figura 3.14.

En la creación de los *streams* se establece el ancho y profundidad de los búferes que implementan. Al igual que en la implementación de una convolución 3x3 genérica, se emplean las variables `STREAMWIDTH`, `STREAMWIDTHi` y `STREAMDEPTH`, definidas en el fichero "Mi_proyecto.h".

```
input_stream = co_stream_create(" input_stream", INT_TYPE(STREAMWIDTH),STREAMDEPTH);
fila = co_stream_create("fila", UINT_TYPE(16),STREAMDEPTH);
columna = co_stream_create("columna", UINT_TYPE(16),STREAMDEPTH);
r0 = co_stream_create("r0", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
r1 = co_stream_create("r1", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
r2 = co_stream_create("r2", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
sal_filT = co_stream_create("sal_filT", INT_TYPE(32),STREAMDEPTH);
fila_c = co_stream_create("fila_c", UINT_TYPE(16),STREAMDEPTH);
columna_c = co_stream_create("columna_c", UINT_TYPE(16),STREAMDEPTH);
rc0 = co_stream_create("rc0", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
rc1 = co_stream_create("rc1", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
rc2 = co_stream_create("rc2", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
rc3 = co_stream_create("rc3", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
rc4 = co_stream_create("rc4", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
c0_sproc = co_stream_create("c0_sproc", INT_TYPE(STREAMWIDTHi), STREAMDEPTH);
sal_filC = co_stream_create("sal_filC", INT_TYPE(48),STREAMDEPTH);
sum_stream = co_stream_create("sum_stream", INT_TYPE(STREAMWIDTHi),STREAMDEPTH);
output_stream = co_stream_create("output_stream", INT_TYPE(STREAMWIDTH),STREAMDEPTH);
```

Como se observa en el código anterior, se define un ancho de 64 bits para los *streams* que realizan la comunicación entre procesos sw/hw y hw/sw. El ancho de los *streams* que comunican procesos hw/hw depende del tamaño de la variable que circula por los mismos en cada caso, tal como se ha indicado en la implementación de cada proceso. Por último, la profundidad de los *streams* se fija a 512, para posibilitar el envío a ráfagas, que se analizará en el Capítulo 6 de de pruebas y resultados.

Los procesos se conectan a través de los *streams* formando la arquitectura descrita en la Figura 5.13.

```
columnsT_process = co_process_create("columnsT", (co_function)columnsT, 6, input_stream, r0,
r1, r2, fila, columna);
filterT_process = co_process_create("filterT", (co_function)filterT, 6, fila, columna, r0, r1,
r2, sal_filT);
columnsC_process = co_process_create("columnsC", (co_function)columnsC, 9, sal_filT, rc0, rc1,
rc2, rc3, rc4, fila_c, columna_c, c0_sproc);
filterC_process = co_process_create("filterC", (co_function)filterC, 9, fila_c, columna_c,
c0_sproc, rc0, rc1, rc2, rc3, rc4, sal_filC);
Sum_wavelets_process = co_process_create("Sum_wavelets", (co_function)Sum_wavelets, 2,
sal_filC, sum_stream);
Empaq_process = co_process_create("Empaq", (co_function)Empaq, 2, sum_stream, output_stream);
prodcon_process = co_process_create("prodcon", (co_function)prodcon, 2, input_stream,
output_stream);
```

Por último, y de forma análoga a la función de configuración empleada para la implementación de una convolución genérica de 3x3 descrita en la sección 5.2.2.4, se usa la función `co_stream_config()` para asignar "`bram`" al atributo `co_kind`, que define los atributos físicos de los *streams*. De esta forma, cuando el proyecto en ISE genere la memoria necesaria para los *streams* (FIFOs) lo hará utilizando memoria de tipo *blockram*, en lugar de memoria distribuida, tal como se explicó.

5.4. Generación del software de la aplicación para el DRC.

En esta sección se mostrará el código de la parte software del sistema, modificado para ejecutar la aplicación en el computador DRC DS1002.

En la sección 4.3 se enumeran los pasos a seguir para adaptar el código software desarrollado en CoDeveloper a software adecuado para la co-ejecución de una aplicación sw/hw sobre esta plataforma. En esta implementación, además de cumplir dichas directrices se ha optado por añadir funcionalidades al código software empleado en CoDeveloper descrito en la sección 5.2.1.

En resumen, este software procesará las imágenes disponibles en el directorio "RAW", que deberá localizarse en el directorio raíz donde se encuentre el ejecutable del software ya compilado. Para ello, enviará ráfagas de paquetes de cuatro píxeles secuencialmente a la RPU. Las ráfagas recibidas de la RPU se escribirán en ficheros de resultados dentro del directorio de nombre "ProcRaw", localizado en el mismo directorio raíz.

Dentro de la function main, se define la función `funcion_filtrado`, cuya funcionalidad consiste en indicar si la extensión del archivo pasado como argumento es de tipo ".raw". Esta función es utilizada por la función scandir para obtener el número de imágenes .raw disponibles en el directorio "RAW", que se almacena en la variable `n_imag`.

```
   int funcion_filtrado(struct dirent *archivo){
        if(strstr(archivo->d_name, ".raw")==NULL) return 0;
        else return 1;
   }
//-------------------------------------------------------------
    struct dirent **resultados=NULL;
    char *directorio="./RAW/";

    n_imag=scandir(directorio, &resultados, funcion_filtrado, alphasort);
    printf("Numero de imagenes encontradas = %d\n", n_imag);
//-------------------------------------------------------
```

En las siguientes líneas de código se abren iterativamente los ficheros de extensión ".raw". Además se crean y abren sus respectivos ficheros de salida, a los que se les asigna el mismo identificador, precedido del prefijo "sal_", y de extensión ".txt", que contendrán los datos procesados en la RPU. En los *arrays* `inFiles[]` y `outFiles[]` de tipo `*FILE`, se almacenan los descriptores de los ficheros de entrada y salida respectivamente.

```
///////  Se abren todos los ficheros de lectura y escritura ///////////////
//-------------------------------------------------------------------------------------------
    for(r=0; r<n_imag; r++){

       char direc[40] = "./RAW/";
       inFileName = strcat(direc, resultados[r]->d_name);
       printf("inFileName = %s \n", inFileName);
       char cadena[40] = "./ProcRaw/sal_";

       // CAMBIO DE LA EXTENSION DE LOS FICHEROS DE SALIDA DE .RAW A .TXT
       char s1[18]="";
       strncpy(s1, resultados[r]->d_name, 15);
       char ext[3]= "txt";
       strcat(s1, ext);

       outFileName = strcat(cadena, s1);
       printf("outFileName = %s \n",outFileName);

       inFiles[r] = fopen(inFileName, "r");
```

```
        if ( inFiles[r] == NULL ) {
                fprintf(stderr, "Error opening input file %s for reading\n", inFileName);
                exit(-1);
        }
        outFiles[r] = fopen(outFileName, "w");
        if ( outFiles[r] == NULL ) {
                fprintf(stderr, "Error opening output file %s for writing\n", outFileName);
                exit(-1);
        }
    }
```

A continuación, se utiliza un bucle `for(n)` para recorrer los ficheros del array `inFiles[]`. Para cada fichero, un bucle `for(i)` se encarga de leer en cada iteración el número de datos de 16 bits que se enviarán en una ráfaga a la RPU, y los almacenará en la variable `*Buffer16`. La variable `TAMRAFAGA` indica el número de paquetes de 4 datos que contiene una ráfaga. Por tanto, el número de datos que se leen en una iteración `i` es de `TAMRAFAGA*4`. Antes de enviar los datos al hardware, se insertan en paquetes de 64 bits, que se almacenan en posiciones contiguas de memoria. El puntero `*PKT` indica el comienzo de la zona de memoria reservada para almacenar los paquetes de la ráfaga. A continuación se envía, a través de la función `DRC_UINT64_co_stream_write` explicada en la sección 4.3, la ráfaga de paquetes de datos. Tras cada envío se realizará una lectura no bloqueante de una ráfaga recibida de la RPU, que será almacenada en la zona de memoria indicada por el puntero `*Resultados[]`. Al tratarse de una lectura no bloqueante, si no hay una ráfaga disponible, se procederá a enviar la siguiente ráfaga del fichero de entrada. La lectura no bloqueante resulta necesaria ante la posibilidad de que se produzca un *deadlocks*, tal y como se explicó en la sección 4.1.2. La variable `ind3` lleva la cuenta del número de paquetes de 64 bits recibidos de la RPU.

```
////////////// SE ENVIAN LOS DATOS A LA FPGA ////////////////////////////
   ind3=0;
   for (n=0; n < n_imag; n++) {

        inFile = inFiles[n];

        for(i=0; i<PixelCount/4; i=i+TAMRAFAGA){ //De 0 al total de pq, en saltos de 128pq/raf.

                //Lee 1raf de TAMRAFAGA*4 datos de 16 bits.
                fread(Buffer16, sizeof(uint16), TAMRAFAGA*4, inFile);

//Empaqueta los datos almacenados en Buffer16[128*4] en 128 paq de 64 bits con 4 pixel/paq.
                s=0;
                for(k=0;k<TAMRAFAGA; k++){
                        PKT[s]= ((((co_uint64)Buffer16[k*4])&0xff)<<48) |
                                ((((co_uint64)Buffer16[k*4+1])&0xff)<<32) |
                                ((((co_uint64)Buffer16[k*4+2])&0xff)<<16) |
                                (((co_uint64)Buffer16[k*4+3])&0xff);
                        s++;
                }

                DRC_UINT64_co_stream_write(pixels_raw, &PKT[0], TAMRAFAGA);

                //LECTURA DE LOS PIXELS RECIBIDOS:
           if ((DRC_UINT64_co_stream_read_nb(pixels_filtered,&Resultados[ind3],TAMRAFAGA))==0){
                        ind3=ind3+TAMRAFAGA;
           }
        }
   }
```

Una vez se han terminado de enviar las ráfagas de todos los ficheros de entrada, se realizará una operación de vaciado de buffer que complete la lectura de los datos procesados en la RPU. Debido a la implementación de la arquitectura del hardware, la RPU enviará resultados mientras reciba datos,

por lo que en dicha operación de vaciado de buffer, será necesario el envío de ráfagas “basura” para que la RPU termine de sacar todos resultados. En este caso la lectura será de tipo bloqueante, puesto que la RPU ya dispone en todo momento de los datos necesarios para calcular los resultados, y no existe posibilidad de *deadlocks*.

```
//VACIADO DEL BUFFER
while(ind3<PixelCount/4*n_imag){

	DRC_UINT64_co_stream_write(pixels_raw, &PKT[0], TAMRAFAGA);
	//printf("rafaga 'basura' enviada \n");

	if((DRC_UINT64_co_stream_read(pixels_filtered, &Resultados[ind3], TAMRAFAGA))==0){
		//printf("recibo rafaga ind3=%d \n",ind3);
		ind3=ind3+TAMRAFAGA;
	}
}
```

Por último se desempaquetan los datos procesados, almacenados en las posiciones de memoria, apuntadas por la variable `*Resultados`, y se escriben en los ficheros de salida.

```
//PRESENTACION DE RESULTADOS:
	f=0;
	outFile = outFiles[f]; //outFiles[f] apunta al FILE de salida correspondiente.
	ind5=0;
	ind3=0;

	printf("Comienzan a escribirse los ficheros de salida \n");

	for(ind2=0; ind2<PixelCount/4*n_imag; ind2++){

		//DESEMPAQUETO
		rec[0] = (co_int16)(Resultados[ind2]>>48);
		rec[1] = (co_int16)(Resultados[ind2]>>32);
		rec[2] = (co_int16)(Resultados[ind2]>>16);
		rec[3] = (co_int16)(Resultados[ind2]);

		if(ind5<127) {
			fprintf(outFile, "%d\t%d\t%d\t%d\t", rec[0],rec[1],rec[2],rec[3]);
			ind5++; ind3++;
		}
		else if(ind5==127 & ind3<(PixelCount/4)-1) {
			fprintf(outFile, "%d\t%d\t%d\t%d\n", rec[0],rec[1],rec[2],rec[3]);
			ind5=0; ind3++;
		}
		else {
			fprintf(outFile, "%d\t%d\t%d\t%d", rec[0],rec[1],rec[2],rec[3]);
			ind3=0; ind5=0;
			fclose(outFile);
			printf("Ha terminado de escribirse el fichero sal_fichero%d\n", f+1);
			f++;
			outFile = outFiles[f]; //outFiles[f] apunta al FILE de salida corresp.
		}
	}
```

5.5. Manejo de ficheros de entrada/salida en ImageJ

En este apartado se explicará el procedimiento a seguir para adecuar los ficheros de imágenes de entrada, al formato que leerá el software en el DRC, así como la forma de visualizar los ficheros ya procesados.

ImageJ es un programa de procesado de imagen de distribución gratuita y de código abierto. Esta herramienta, ampliamente conocida en el campo de trabajo que abarca este PFC, soporta gran variedad de formatos y permite la automatización de tareas, así como la creación de herramientas personalizadas usando macros en Java.

Como ya se comentó, en astrofísica se trabaja con ficheros *fits* pero como este proyecto tenía como objetivo principal la aceleración hardware, por falta de tiempo, se ha trabajado con formatos estándar y se deja como trabajo futuro las modificaciones pertinentes para el trabajo con este tipo de ficheros.

Por tanto, según se ha definido en el software del DRC, las imágenes de entrada deben consistir en ficheros de tamaño 512x512 en formato *.raw*[8] de 16 bits por dato.

El DRC leerá los ficheros *raw* según el formato *Little-Endian*[9], por lo que en ImageJ, los ficheros de entrada deberán escribirse en el mismo formato para que en la lectura se interpreten correctamente. Para ello, se debe seleccionar la opción "*Save TIFF and Raw in Intel Byte Order"* en el diálogo de la Figura 5.19, a través de los menús: *Edit → Options → I/O options.*

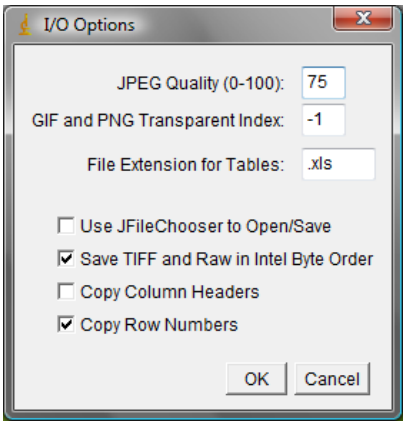

Figura 5.19. Opciones de I/O en ImageJ.

En caso de no seleccionar esta opción, los ficheros de entrada al DRC se habrán generado según *Big-Endian*, y por tanto, en la lectura no se interpretará correctamente el orden de los bytes y los datos se leerán al revés. Así, si un píxel tiene el valor 3 (en binario con un byte: *00000011*), en formato *Little-Endian* con 2 bytes se escribiría como *00000011 00000000 (LSB-MSB),* y en formato *Big-Endian* como *00000000 00000011 (LSB-MSB).* Por tanto, si se lee el dato con el formato contrario al que se han escrito, se leerá el dato 768 en lugar de 3.

[8] Una imagen *.raw* (cruda) consiste en un fichero sin cabecera, donde se escriben los datos "en crudo" como su nombre indica. En este caso, cada dato será un entero sin signo de 16 bits.

[9] *Little-Endian* y *Big-Endian* hacen referencia a la forma en que se guardan en memoria los números que ocupan más de un byte. El primer formato, guarda primero el LSB y aunque no es lo más intuitivo, es utilizado en la mayoría de ordenadores PC. Por el contario, el segundo almacena primero el MSB y lo utilizan procesadores de la familia PowerPc (típicos en ordenadores Apple). Por esto, es importante conocer la plataforma en que se crea o lee un fichero de datos.

En este PFC, se dispone inicialmente de 100 imágenes en formato *.bmp* de 8 bits de tamaño 512x512, por lo que a continuación se indicarán los pasos para transformar dichas imágenes al formato adecuado para el software del DRC.

1. Para evitar modificarlas una a una, se abrirán todas las imágenes a la vez. Para ello recorrer la siguiente lista de menús: *File* → *Import* → *Image Sequence* (Figura 5.20). Se selecciona la primera, se pulsa "*Abrir*", y se eligen las opciones correspondientes al número de imágenes. En la Figura 5.21 se muestran las opciones seleccionadas.

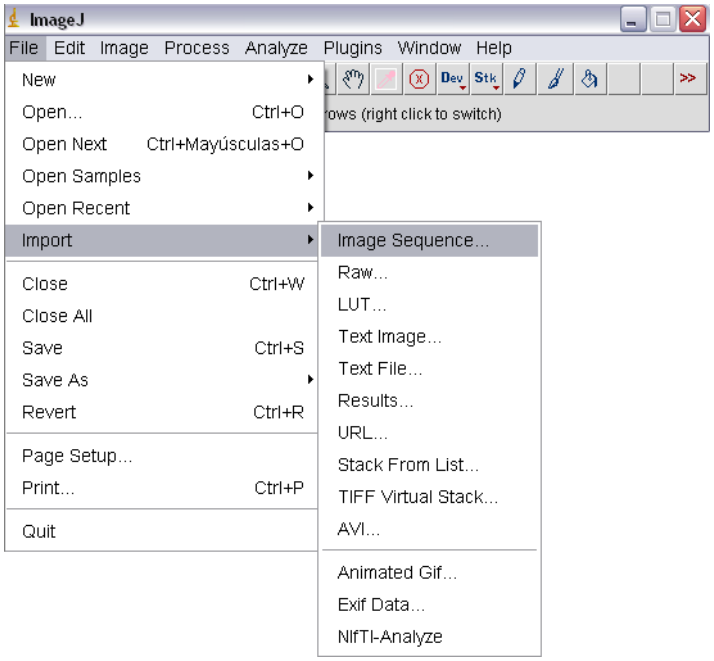

Figura 5.20. Procedimiento para abrir una secuencia de imágenes.

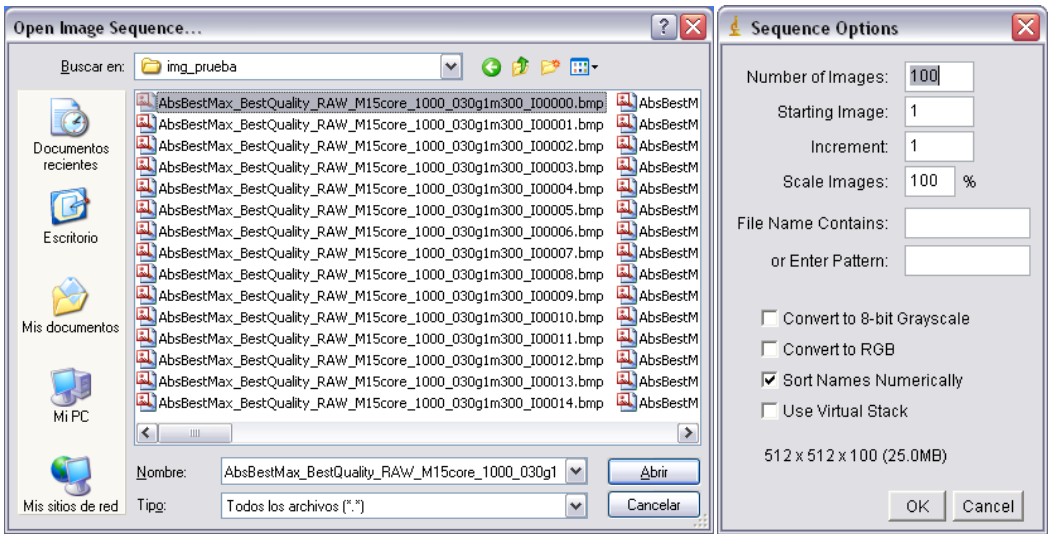

Figura 5.21. Opciones para abrir una secuencia de imágenes.

En ImageJ se habrá abierto un stack (o pila) de imágenes que pueden separarse haciendo: *Images* → *stacks* → *stack to images*. En esta ocasión no se hará, ya que se pretende modificar todas las imágenes igual manera.

2. El siguiente paso será pasar de 8 bits a 16 bits por dato, opción disponible en el menú: *Image* → *Type* →*16 bits*, como se aprecia en la Figura 5.22.

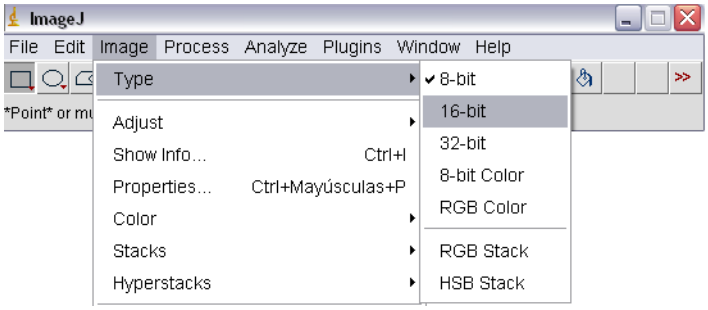

Figura 5.22. Modificación del número de bits por píxel.

3. Guardar la pila de imágenes en formato *raw*. Para ello recorrer los siguientes menús: *File* → *Save as* → *Raw Data*. Si no se modifica, se guardará con el nombre de la carpeta donde se encuentra la secuencia de imágenes (en este caso "*img_prueba.raw*"). Este paso en realidad es opcional, por si queremos guardar todas las imágenes en un único fichero, aunque en realidad, el paso necesario es el siguiente.

4. Guardar la pila anterior como imágenes separadas en una capeta de nombre *RAW* haciendo: *File* → *Save as* → *Image Sequence* → *raw data* y seleccionar los parámetros indicados en la Figura 5.23. De esta forma se guardarán en la carpeta *RAW* 100 imágenes de nombres "*img_pruebaXXXX.raw*", donde *XXXX* será código de identificación para cada imagen.

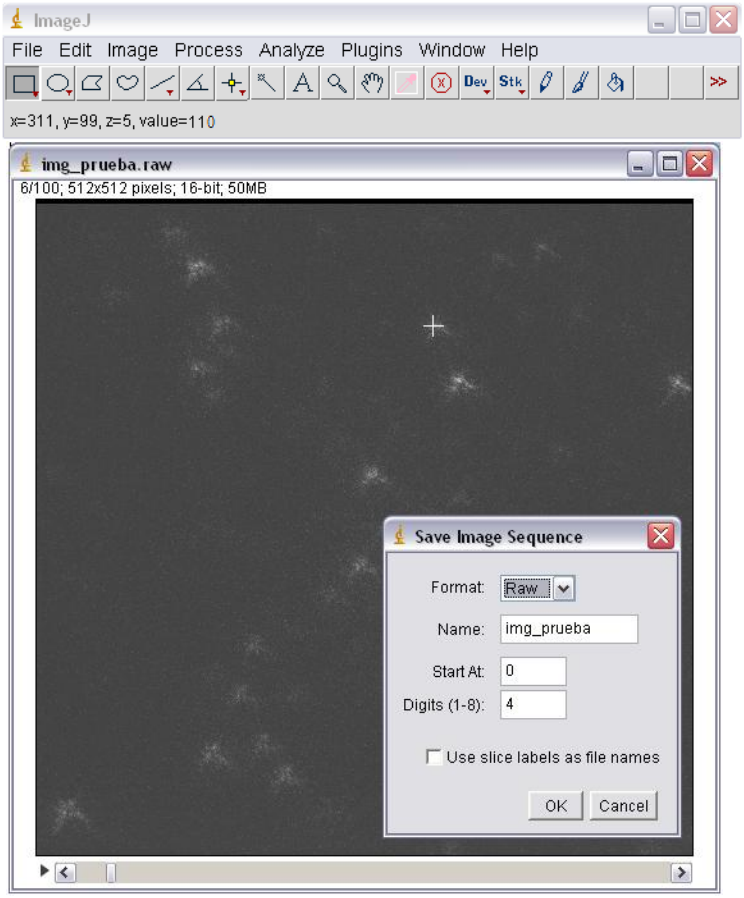

Figura 5.23. Opciones para guardar una secuencia de imágenes.

Para comprobar que se han guardado correctamente, se puede abrir la imagen "*img_prueba0006.raw*", por ejemplo, seleccionando las opciones de la Figura 5.24.

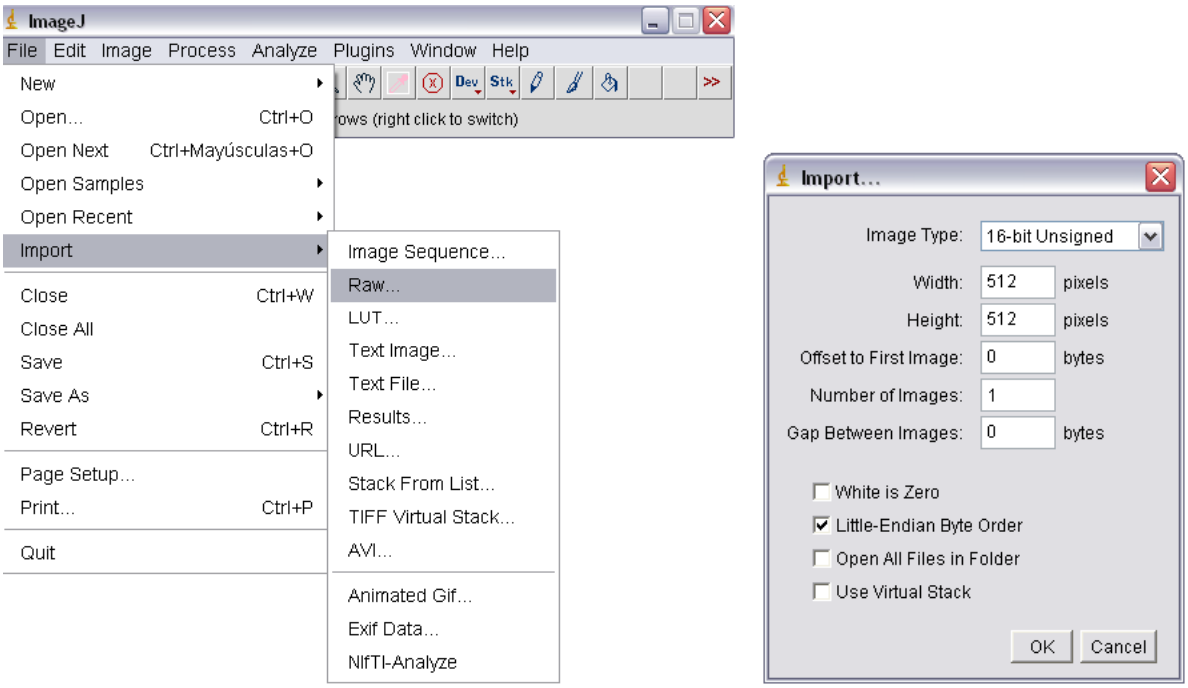

Figura 5.24. Procedimiento para abrir una imagen cruda.

Por último, indicar que para visualizar las imágenes resultantes, procedentes de la FPGA, se deben recorrer los menús: *File* → *Import* → *Text Image*. Se ha utilizado este formato, para poder comprobar los resultados numéricos de los ficheros de salida visualmente, ya que si se escriben en "crudo" esto no es posible ya que los datos se escriben en formato binario, en lugar de formato ascii.

Capítulo 6

Pruebas y Resultados

En este capítulo se expondrán resultados obtenidos en dos casos: (1) en la implementación de la arquitectura para la realización de una convolución genérica de 3x3 descrita en la sección 5.2, y (2) en la implementación de la arquitectura que calcula la suma de las dos primeras escalas DWT *à trous*, descrita en la sección 5.3.

6.1. Resultados de la implementación de la arquitectura para el cálculo de una convolución genérica de 3x3

Antes de evaluar los resultados obtenidos es necesario aclarar que, para poder realizar un análisis sobre la implementación de una convolución genérica de forma correcta (independiente de la aplicación objeto de este Proyecto Fin de Carrera), en la elección de los coeficientes se ha tenido en cuenta que no fueran potencias de 2, para obligar al sintetizador de hardware a emplear multiplicadores en el cálculo de la suma de productos, evitando el uso de desplazamientos lógicos. Por tanto, se definen los siguientes coeficientes de la máscara de convolución:

$$h_{3x3} = \frac{1}{16} \cdot \begin{pmatrix} F1 & F2 & F3 \\ F4 & F5 & F6 \\ F7 & F8 & F9 \end{pmatrix} = \frac{1}{16} \cdot \begin{pmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \\ 7 & 7 & 7 \end{pmatrix} \tag{22}$$

6.1.1. Validación de la arquitectura utilizando las herramientas de CoDeveloper

En esta sección se va a validar la correcta conexión entre procesos a través de los *streams*, tal y como se definen en la función de configuración, así como el correcto funcionamiento del sistema. También se evaluará la eficiencia del compilador, a la vista de los informes generados sobre recursos hardware inferidos y su utilización en tiempo, o latencia. Para ello se utilizará la herramienta *Application Monitor* de CoDeveloper.

La compilación del proyecto genera un ejecutable denominado "de escritorio" (para el sistema operativo del computador de desarrollo) que permite realizar una simulación software del sistema. Al ejecutar la simulación desde el entorno de CoDeveloper, si se ha seleccionado previamente la opción *Launch Application Monitor* en la pestaña *Simulate* del menú *Project→ Options*, se lanza también la herramienta *Application Monitor* (AP), lo que permite depurar la aplicación durante su ejecución. En la pestaña *Block Diagram* del AP se muestra el siguiente diagrama de procesos:

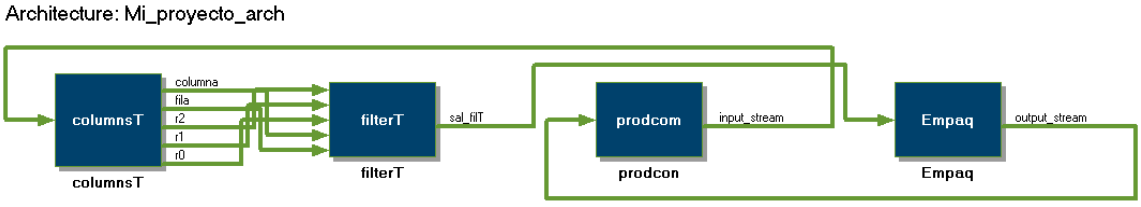

Figura 6.1. Diagrama de la arquitectura que muestra la comunicación de los procesos a través de los *streams*.

Analizando visualmente los *streams* que comunican los distintos procesos, se puede comprobar que la descripción del sistema se ha llevado a cabo de forma correcta.

Los resultados tras la generación del hardware con CoDeveloper, eligiendo como plataforma la opción Xilinx Generic (VHDL), se muestran en la Tabla 6.1. En la misma puede observarse cómo el optimizador (*Stage Master*) divide el código C en bloques básicos y asigna cada operación de cada bloque a una o más etapas. Las sentencias pertenecientes a una etapa se implementan como lógica combinacional ejecutándose en un único ciclo de reloj. En todos los procesos, el primer y último bloque tienen asignadas las operaciones de abrir y cerrar los *streams*, respectivamente.

```
|--------------------------------------
| columnsT
|--------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 loop:
|    Stages: 5
|    Max. Unit Delay: 17
|  Block #2 loop:
|    Stages: 7
|    Max. Unit Delay: 0
|  Block #3 loop:
|    Stages: 5
|    Max. Unit Delay: 0
|  Block #4 pipeline:
|    Latency: 3
|    Rate:    1
|    Max. Unit Delay: 64
|    Effective Rate: 64
|  Block #5:
|    Stages: 1
|    Max. Unit Delay: 17
|  Block #6:
|    Stages: 1
|    Max. Unit Delay: 1
|  Block #7:
|    Stages: 1
|    Max. Unit Delay: 0
|--------------------------------------
| Operators:
|  3 Adder(s)/Subtractor(s) (9 bit)
|  6 Adder(s)/Subtractor(s) (16 bit)
|  1 Adder(s)/Subtractor(s) (32 bit)
|  1 Comparator(s) (2 bit)
|  8 Comparator(s) (17 bit)
|  1 Comparator(s) (32 bit)
|--------------------------------------
| Total Stages: 14
| Max. Unit Delay: 64
| Estimated DSPs: 0
|--------------------------------------
```

```
|--------------------------------------
| filterT
|--------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 pipeline:
|    Latency: 10
|    Rate:    1
|    Max. Unit Delay: 36
|    Effective Rate: 36
|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|--------------------------------------
| Operators:
|  10 Adder(s)/Subtractor(s) (32 bit)
|  9 Multiplier(s) (32 bit)
|  1 Comparator(s) (9 bit)
|  4 Comparator(s) (17 bit)
|  1 Comparator(s) (32 bit)
|--------------------------------------
| Total Stages: 12
| Max. Unit Delay: 36
| Estimated DSPs: 36
|--------------------------------------
|--------------------------------------
| Empaq
|--------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 pipeline:
|    Latency: 2
|    Rate:    1
|    Max. Unit Delay: 16
|    Effective Rate: 16
|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|--------------------------------------
| Operators:
|  3 Adder(s)/Subtractor(s) (16 bit)
|  1 Comparator(s) (2 bit)
|  4 Comparator(s) (17 bit)
|--------------------------------------
| Total Stages: 4| Max. Unit Delay: 16
| Estimated DSPs: 0
|--------------------------------------
```

Tabla 6.1. Las sentencias de código C se dividen en bloques y etapas.

Además de la información relativa a los bloques y etapas necesarias en los mismos, bajo las etiquetas *Operators* se muestran los recursos hardware requeridos en cada proceso implementado. Esta información es útil a la hora de comprobar si se han obtenido los recursos esperados o sería conveniente expresar de otro modo alguna operación en el código para que su proyección hardware resultase en un circuito más eficiente. Se puede comprobar, por ejemplo, que para realizar la suma de productos relativa a la convolución, en el proceso *filterT*, se utilizan 9 multiplicadores de 32 bits y por consiguiente 10 sumadores de 32 bits, como era de esperar, ya que se pretende realizar dicha suma de productos en un único ciclo de reloj.

El optimizador trata de determinar el mínimo número de etapas en la implementación de cada bloque básico, pero esto no siempre es la mejor opción. Por otro lado, para una buena optimización del código C, es importante que a la hora de programar, se tenga en cuenta la posible interpretación del por parte del optimizador de CoDeveloper. Por tanto, a continuación se van a indicar las diversas medidas tomadas que han permitido un mayor control sobre la optimización hardware conseguida.

Se va a comenzar analizando la implementación de las operaciones del proceso *filterT* descritas bajo la directiva `#pragma CO PIPELINE`. El optimizador asigna 2 etapas a dicho *pipeline*, una (*stage* 0) para la lectura de los *streams* de entrada, y otra (*stage* 1) para el cálculo de la máscara de condiciones de contorno, suma de productos y escritura de resultados en el *stream* de salida. Esto supone una máxima unidad de retardo (medida como una estimación de retardos de puerta) de 292, lo que podría limitar considerablemente la velocidad del reloj. Por este motivo, se optó por incluir la directiva `#pragma CO set stageDelay`, que limite el máximo retardo de una etapa. A la hora de elegir un valor máximo, se ha hecho uso de la herramienta *Stage Master Explorer*, descrita en la sección 3.1.3, que proporciona un gráfico a través del menú *Tools* → *Pipeline Graph*, que muestra el rendimiento en función del retardo de puerta asignado. Dicho gráfico se muestra en la siguiente figura:

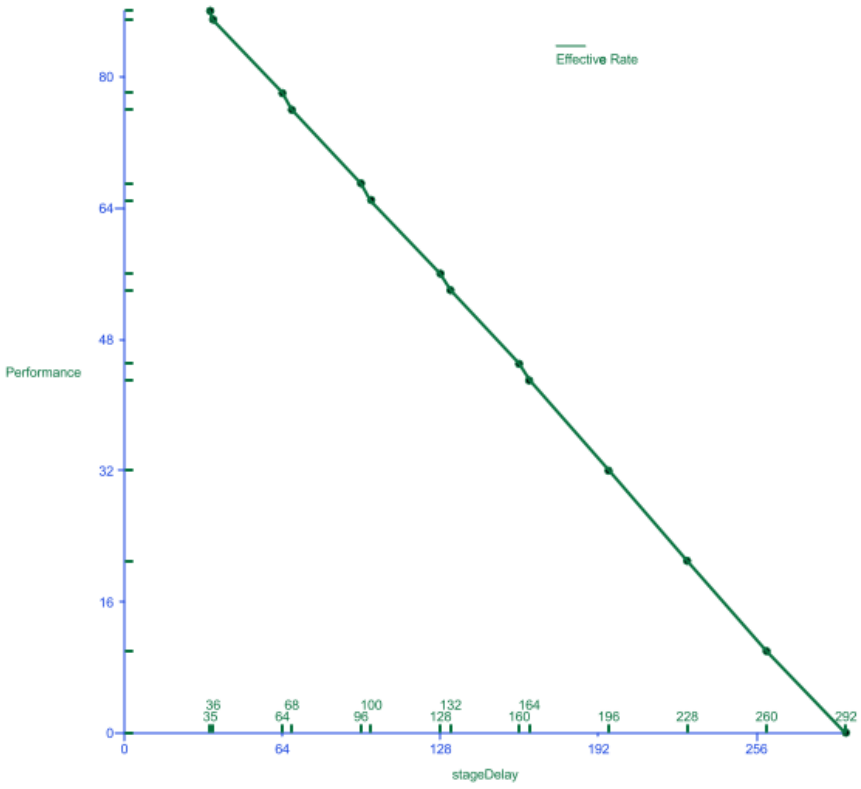

Figura 6.2. *Pipeline Graph* del proceso *filterT*.

Considerando la gráfica anterior, se ha tomado un retardo de puerta máximo de 40 (`#CO set stageDelay 40`), de forma que siguiendo esta indicación, el optimizador asigna ahora a este *pipeline* 10 etapas y una tasa efectiva de 36, como puede verse en la Tabla 6.1.

Como ya se explicó en la sección 3.1.2, el uso *pipeline* dentro de un bucle permite el comienzo de la ejecución en paralelo de la siguiente iteración antes de completar la iteración actual. Por lo tanto, en este caso se realizarán en paralelo múltiples iteraciones del bucle de forma que a partir del primer resultado de salida, que requerirá las 10 etapas indicadas, se obtendrá un resultado por ciclo, como se indica en la tabla anterior (*Rate*:1), a una velocidad superior a la que obtendríamos cada nuevo resultado en el caso descrito inicialmente de sólo 2 etapas. Estas 10 etapas corresponden a la latencia del *pipeline*, término que determina el número de ciclos requeridos para completar una iteración del bucle.

De forma análoga se han fijado los valores máximos de retardo de puerta en los procesos *columnsT* y *Empaq*.

Otra consideración a destacar para reducir la tasa (ciclos/resultado) de *pipeline*, consiste en evitar varios accesos a un mismo *array*. Esta situación se ha dado en la implementación del proceso *columnsT* a la hora de preparar las columnas a enviar al proceso *filterT*. En esta implementación se almacenó una fila de la imagen en el *array* `A[512]` y la siguiente en el *array* `B[512]`, con el objetivo de acceder al elemento `j` de ambos simultáneamente y junto con el nuevo dato leído del *stream* de entrada, enviar la columna al proceso *filterT*. Así, se observa en la siguiente imagen de la herramienta *Stage Master Explorer*, que la lectura de ambos *arrays* se realiza en la misma etapa (*Stage* 1), y en esa misma etapa se ha recibido también el nuevo dato (que se almacena en la variable `p33`) y se envía la columna a través de los *arrays* `r0`, `r1` y `r2`.

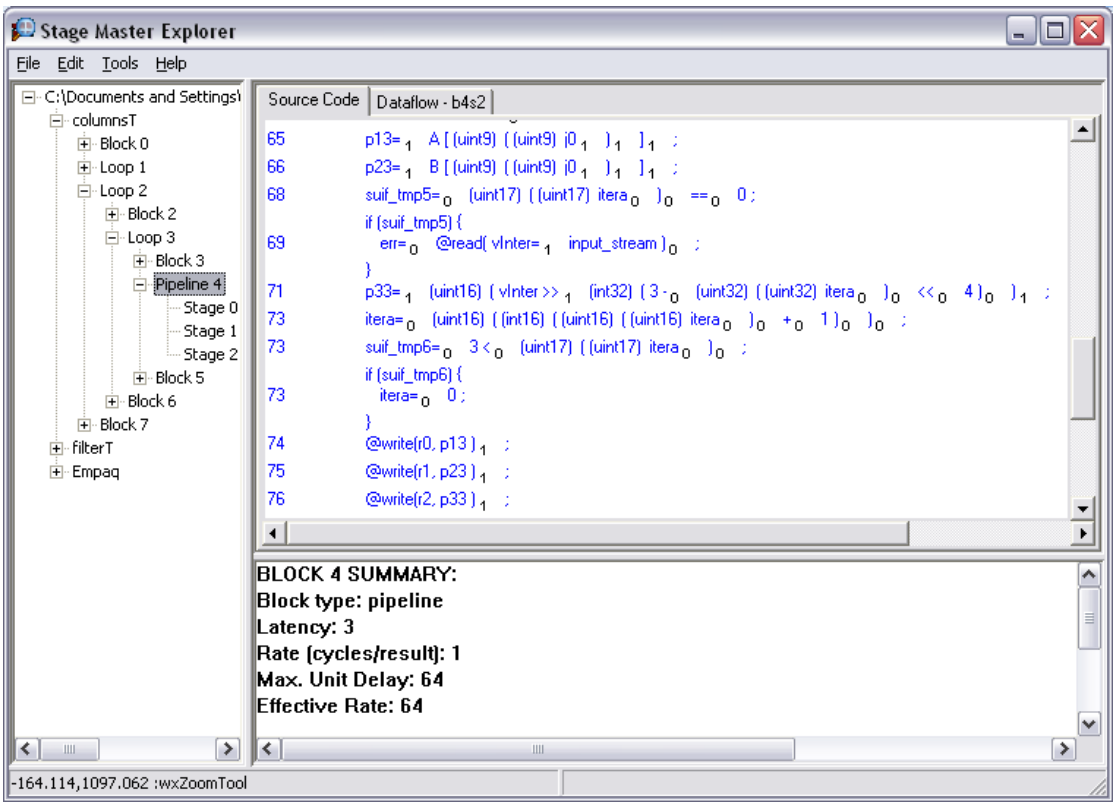

Figura 6.3. Etapas del proceso *columnsT*. Se indica con '0' o '1' la etapa a la que pertenece cada operación.

Si en lugar de esta opción, se hubiera considerado un array de dos dimensiones `S[512][2]`, el acceso simultáneo a `S[j][1]` y `S[j][2]` no sería posible y el optimizador incluiría una etapa distinta para cada lectura, ya que no se puede acceder a un mismo bloque de RAM en el mismo ciclo de reloj.

Para concluir esta sección, se analizarán los bloques y etapas del proceso *Empaq*, que empaqueta los datos antes de enviar los resultados al software. En este proceso también se ha hecho uso de la directiva `#pragma CO PIPELINE`, para conseguir así, que mientras se lee un nuevo dato del *stream* de entrada `sal_filT`, se inserte en un paquete el resultado de la iteración anterior, y se envíe por el *stream* de salida en caso de que el paquete de datos esté completo. De esta forma, este proceso ha resultado en 3 bloques, el primero y último para abrir y cerrar los *streams* como se indicó al comienzo de esta sección. El segundo, que consta de 2 etapas, tiene asignadas las operaciones del bucle que se realizan en *pipeline*. En los siguientes diagramas de flujo de datos se observan las operaciones asignadas a cada etapa y que por tanto se realizan en paralelo.

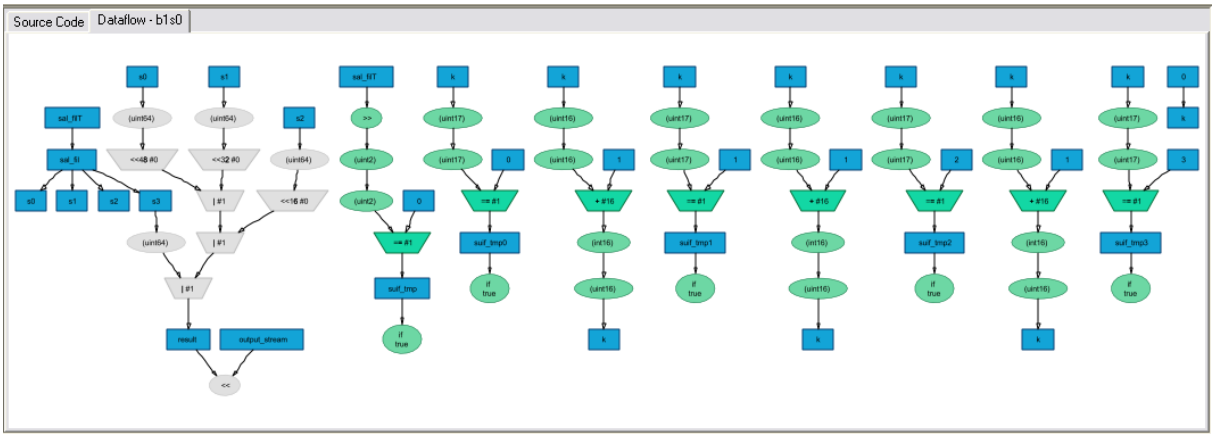

Figura 6.4. Operaciones del proceso *Empaq* del bloque 1 asignadas a la etapa 0 (bloques activos en verde).

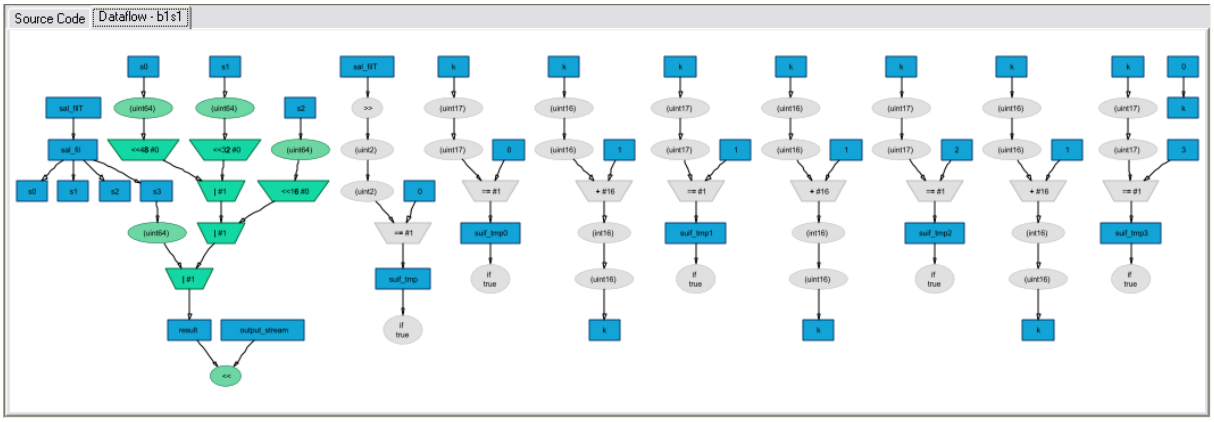

Figura 6.5. Operaciones del proceso *Empaq* del bloque 1 asignadas a la etapa 1 (bloques activos en verde).

Como se muestra en el gráfico de la Figura 6.4, en la etapa 0 se realizan de forma paralela la lectura del *stream* de entrada y las operaciones que comprueban en la posición del paquete en la que se insertará el dato leído. En la etapa 1, mostrada en la Figura 6.5, se inserta el dato en el paquete correspondiente y se envía por el *stream* de salida si dicho paquete está completo.

6.1.2. Resultados de la síntesis e implementación del hardware en el entorno ISE de Xilinx sobre una plataforma FPGA Virtex4

6.1.2.1 Resultados de síntesis

En la síntesis del sistema, se traducirá el hardware programado a un modelo a nivel de transferencia de registros (RTL). En el proceso de síntesis se distinguen tres fases: una optimización a nivel RTL (análisis de código, agrupación de constantes, etc), otra optimización a nivel lógico (optimización de ecuaciones booleanas para reducir el *fan-out* y el área del circuito), y una última optimización a nivel de puerta, en función de la tecnología sobre la que se implementará el sistema.

Tras la fase de síntesis se dispone de una representación a nivel RTL, en términos de símbolos genéricos (tales como sumadores, multiplicadores, contadores, puertas AND y OR). Para analizar dicha representación se debe acceder a la opción *View RTL Schematic* de la pestaña *Synthesize* del entorno ISE. En la siguente Figura 6.6 se observa dicha representación, donde pueden identificarse los *streams* de entrada/salida del sistema. Para analizar los distintos niveles de representación se dispone de las opciones "*Back a Schematic"* y *"Forward a Schematic"*.

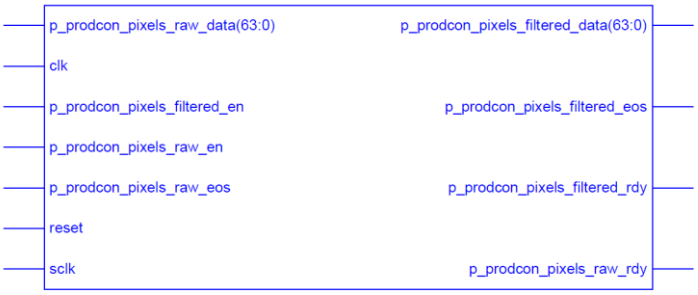

Figura 6.6. Esquemático del sistema.

Al sintetizar también se genera el informe de síntesis (*Synthesis Report*), donde se muestran estimaciones de área y velocidad. En esta implementación se ha sintetizado seleccionando inicialmente la velocidad como objetivo de optimización en las opciones de síntesis, y posteriormente el área, con el fin de comparar resultados.

- **Resultados de síntesis para optimización en velocidad**

A continuación se muestra el informe de síntesis HDL (*HDL Synthesis Report*) donde se indican los recursos hardware requeridos:

```
HDL Synthesis Report

Macro Statistics
# RAMs                                          : 12
 512x16-bit dual-port RAM                      : 4
 512x17-bit dual-port RAM                      : 6
 512x65-bit dual-port RAM                      : 2
# Multipliers                                   : 9
 32x32-bit multiplier                          : 9
# Adders/Subtractors                            : 35
 16-bit adder                                  : 4
 16-bit subtractor                             : 2
 32-bit adder                                  : 10
```

```
 9-bit adder                                       : 19
# Counters                                         : 16
 9-bit up counter                                  : 16
# Registers                                        : 141
 1-bit register                                    : 38
 10-bit register                                   : 1
 16-bit register                                   : 24
 17-bit register                                   : 12
 2-bit register                                    : 1
 3-bit register                                    : 1
 32-bit register                                   : 47
 4-bit register                                    : 1
 64-bit register                                   : 2
 65-bit register                                   : 4
 8-bit register                                    : 3
 9-bit register                                    : 7
# Comparators                                      : 23
 17-bit comparator greater                         : 2
 17-bit comparator less                            : 4
 32-bit comparator greater                         : 1
 9-bit comparator equal                            : 16
# Multiplexers                                     : 1
 32-bit 4-to-1 multiplexer                         : 1
```

A la vista de estos resultados puede decirse que el código generado por Codeveloper está bastante bien, ya que la traducción en ISE se aproxima bastante a los recursos hardware, en cuanto a número de multiplicadores, sumadores y demás, indicados por CoDeveloper mostrados en la Tabla 6.1.

En el siguiente informe generado, *Advanced HDL Synthesis Report*, se reconocen e implementan, además, registros de desplazamiento dinámico, máquinas de estado y multiplicadores *pipeline*.

```
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                             : 10
 512x16-bit dual-port block RAM                    : 2
 512x17-bit dual-port block RAM                    : 6
 512x65-bit dual-port block RAM                    : 2
# Multipliers                                      : 9
 32x32-bit registered multiplier                   : 9
# Adders/Subtractors                               : 35
 16-bit adder                                      : 4
 16-bit subtractor                                 : 2
 20-bit adder                                      : 1
 28-bit adder                                      : 1
 32-bit adder                                      : 8
 9-bit adder                                       : 19
# Counters                                         : 16
 9-bit up counter                                  : 16
# Registers                                        : 1921
 Flip-Flops                                        : 1921
# Comparators                                      : 23
 17-bit comparator greater                         : 2
 17-bit comparator less                            : 4
 32-bit comparator greater                         : 1
 9-bit comparator equal                            : 16
# Multiplexers                                     : 1
 32-bit 4-to-1 multiplexer                         : 1
```

En la siguiente figura se muestra el número de recursos utilizados de la FPGA seleccionada, así como el porcentaje de utilización de cada tipo, lo que permite de una forma bastante aproximada conocer el tamaño de la implementación en la FPGA.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1570	89088	1%
Number of Slice Flip Flops	1875	178176	1%
Number of 4 input LUTs	2490	178176	1%
Number of bonded IOBs	136	960	14%
Number of FIFO16/RAMB16s	12	336	3%
Number of GCLKs	1	32	3%

Figura 6.7. Estimación de los recursos utilizados en la FPGA tras la síntesis para optimización en velocidad.

Al final de la síntesis se genera un informe de tiempos (*TIMING REPORT*), que únicamente muestra una estimación, cuya bondad se podrá medir en el informe generado tras la fase *Place&Route* de la implementación del diseño, que ya muestra resultados precisos.

```
Timing Summary:
---------------
Speed Grade: -11

   Minimum period: 6.560ns (Maximum Frequency: 152.439MHz)
   Minimum input arrival time before clock: 2.890ns
   Maximum output required time after clock: 7.996ns
   Maximum combinational path delay: No path found
```

Este informe estima el periodo mínimo del ciclo de reloj a 6.56 ns, asignando el 39.7% del mismo a la lógica, y el 60.3% al rutado.

```
    Total                     6.560ns (2.605ns logic, 3.955ns route)
                                      (39.7% logic, 60.3% route)
```

- **Resultados de síntesis para optimización en área**

En el informe de síntesis HDL (*HDL Synthesis Report*), donde se indican los recursos hardware requeridos, así como el siguiente informe generado (*Advanced HDL Synthesis Report*), se muestran los mismos resultados que para la optimización por tiempo.

Respecto a la utilización de recursos de la FPGA, se estiman un menor número de *Slices*, *Slice Flip-Flops* y *LUTs* de 4 entradas, pero la disminución es tan pequeña que no repercute en el porcentaje de utilización.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	1389	89088	1%
Number of Slice Flip Flops	1873	178176	1%
Number of 4 input LUTs	2113	178176	1%
Number of bonded IOBs	136	960	14%
Number of FIFO16/RAMB16s	12	336	3%
Number of GCLKs	1	32	3%

Figura 6.8. Estimación de los recursos utilizados en la FPGA tras la síntesis para optimización en área.

En el informe de tiempos generado (*TIMING REPORT*), como era de esperar, la máxima frecuencia de reloj se reduce respecto a la síntesis optimizada por velocidad.

```
Timing Summary:
---------------
Speed Grade: -11

   Minimum period: 8.618ns (Maximum Frequency: 116.036MHz)
   Minimum input arrival time before clock: 3.014ns
   Maximum output required time after clock: 7.742ns
   Maximum combinational path delay: No path found
```

Este informe estima el periodo mínimo del ciclo de reloj a 8.61 ns, asignando el 49.6% del mismo a la lógica, y el 50.4% al rutado.

```
    Total                     8.618ns (4.273ns logic, 4.345ns route)
                                      (49.6% logic, 50.4% route)
```

6.1.2.2 Resultados de Implementación

La implementación incluye dos tareas principales, llevadas a cabo en su última fase denominada *Place&Route*. Éstas son, la selección de la ubicación de cada una de las celdas que componen el circuito sobre la superficie disponible de la FPGA, y el rutado de las conexiones existentes entre las mismas y con las celdas que se conectan a las patillas del circuito integrado. Por tanto, tras la implementación tendremos los resultados precisos de utilización de la FPGA seleccionada, así como el periodo mínimo de reloj. A continuación se mostrarán dichos resultados para el caso de optimización por velocidad y por área.

- **Resultados para optimización en velocidad**

A continuación se muestra una tabla resumen con los recursos de la FPGA Virtex 4 utilizados en la implementación del sistema:

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	1,875	178,176	1%	
Number of 4 input LUTs	2,242	178,176	1%	
Logic Distribution				
Number of occupied Slices	1,781	89,088	1%	
Number of Slices containing only related logic	1,781	1,781	100%	
Number of Slices containing unrelated logic	0	1,781	0%	
Total Number of 4 input LUTs	2,474	178,176	1%	
Number used as logic	2,242			
Number used as a route-thru	232			
Number of bonded IOBs	136	960	14%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number of FIFO16/RAMB16s	12	336	3%	
Number used as RAMB16s	12			

Figura 6.9. Recursos utilizados en la FPGA tras la implementación.

Si se comparan estos resultados con la estimación mostrada tras la síntesis del proyecto, se observa que ésta era bastante buena.

Por el contrario, el informe *Static Timing Report*, generado tras *Place&Route*, muestra un periodo de reloj de 9.756 ns, bastante superior al estimado en la síntesis (6.56 ns).

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    9.756|         |         |         |
---------------+---------+---------+---------+---------+
```

- **Resultados para optimización en área**

En la siguiente tabla se muestran los recursos de la FPGA Virtex 4 utilizados en la implementación del sistema. Se observa nuevamente que la estimación realizada tras la síntesis es bastante aproximada.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	1,873	178,176	1%	
Number of 4 input LUTs	1,776	178,176	1%	
Logic Distribution				
Number of occupied Slices	1,611	89,088	1%	
Number of Slices containing only related logic	1,611	1,611	100%	
Number of Slices containing unrelated logic	0	1,611	0%	
Total Number of 4 input LUTs	2,136	178,176	1%	
Number used as logic	1,776			
Number used as a route-thru	360			
Number of bonded IOBs	136	960	14%	
IOB Flip Flops	1			
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number of FIFO16/RAMB16s	12	336	3%	
Number used as RAMB16s	12			

Figura 6.10. Recursos utilizados en la FPGA tras la implementación.

Por último, el informe *Static Timing Report*, generado tras *Place&Route*, muestra un periodo de reloj de 10.537 ns, que supera al estimado en la síntesis (8.618 ns).

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |   10.537|         |         |         |
---------------+---------+---------+---------+---------+
```

6.1.3. Análisis de tiempos de co-ejecución del sistema que calcula la convolución genérica de 3x3 sobre la plataforma DRC-DS1002

En esta sección se van a mostrar los resultados temporales obtenidos en la ejecución del sistema híbrido, que implementa la convolución genérica 3x3, sobre la plataforma DRC-DS1002. Estos resultados consistirán en la medición del tiempo requerido para calcular la convolución de 100 imágenes de 512x512 píxeles, y obtener, a partir de este dato, el tiempo medio empleado por imagen. Para la programación del hardware y la generación del software sobre dicha plataforma, se han seguido los pasos descritos en el capítulo 4.

Siguiendo la estimación indicada en la síntesis (máxima velocidad de reloj de 152.439MHz) se selecciona un reloj de 133MHz. Tras la implementación del proyecto en ISE "rpware" descrito en la sección 4.2, el informe *Static Timing Report*, generado en el proceso de *Place&Route*, indica que la velocidad máxima (120.265 MHz) es inferior a la seleccionada, por lo que la estimación en la fase de síntesis era más optimista. A pesar de de esto, se comprueba que el sistema funciona con el reloj de 133MHz, ya que las estimaciones del ISE en la implementación son bastante "conservadoras".

```
Timing summary:
---------------
Timing errors: 0  Score: 0
Constraints cover 46036 paths, 0 nets, and 12446 connections

Design statistics:
   Minimum period:   8.315ns   (Maximum frequency: 120.265MHz)
   Maximum path delay from/to any node:   8.315ns
```

Con el fin de que la lectura y escritura de ficheros no interfiera en las mediciones, el software empleado almacena previamente las 100 imágenes a procesar, y de la misma forma, la escritura de ficheros de resultados se realizará una vez termina el procesado de todas ellas. Por tanto, la medida de tiempos se realizará desde que comienzan a enviarse los datos hasta que se recibe el último resultado.

Se ha comprobado el correcto funcionamiento del sistema para el envío a ráfagas de 1, 64 y 256 paquetes de 4 datos de 16 bits. Para ráfagas de tamaño 32 y 128, los resultados no son los esperados. Esto es posible que se deba a que en la comuninación hw/sw no se soporta *back-pressure*, lo que supone que a la hora de escribir en el *stream*, el sistema únicamente se indica si hay espacio en el búffer, pero no si éste es suficiente para contener una nueva ráfaga completa. Esto puede provocar, por tanto, una pérdida de datos.

Los resultados temporales muestran, como era de esperar, que el menor tiempo de ejecución se obtiene para ráfagas del mayor tamaño (256 paquetes en este caso). Esto es debido a que el envío de una ráfaga se realiza de forma monolítica por parte del proceso software (no cede el uso del bus HT hasta que transmite la ráfaga completa), mientras que al transmitir varias ráfagas, el software tiene que obtener y ceder el control del bus en cada ocasión. En la siguiente figura aparecen los resultados que el proceso software imprime en la consola, donde se observa que el tiempo total obtenido para las 100 imágenes es de 0.25 segundos.

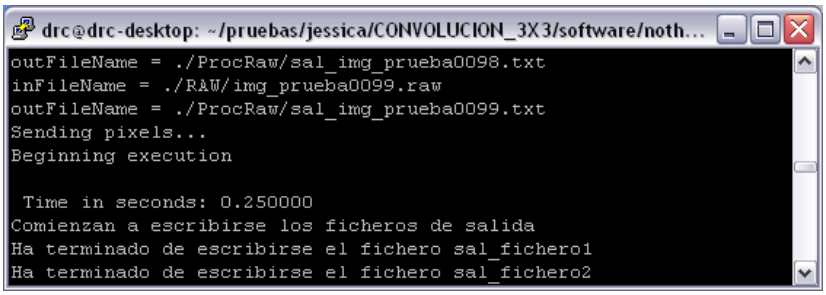

Figura 6.11. Resultados que el proceso software imprime por pantalla.

Esto supone un tiempo de procesado medio por imagen de 0.0025 segundos, lo que equivale a una velocidad de 400 imágenes por segundo.

Si se compara este resultado temporal con el que se obtendría realizando la convolución de 3x3 en matlab, puede comprobarse que se consigue una aceleración hardware importante. En un PC estándar (Core2Duo a 2.4GHz) se consiguen procesar en matlab unas 37 imágenes por segundo, frente a las 400 que se obtienen con la arquitectura implementada.

Para tener una idea aproximada de la eficiencia del *pipeline* del hardware programado, se ha calculado el número medio de ciclos de reloj requeridos por el sistema para ofrecer un resultado. Según la ecuación (23), se consume 1,268 ciclos de reloj por píxel.

$$Tasa = \frac{(tiempo\ /\ imagen)(seg)}{(píxeles\ /\ imagen)\cdot periodo\_reloj(seg)} = \frac{0{,}0025}{(512\cdot 512)\cdot 7{,}52e^{-9}} = 1{,}268 \qquad (23)$$

Considerando que la solución ideal sería un ciclo por píxel, el resultado obtenido se puede considerar bueno. La diferencia entre este resultado y el ideal, puede atribuirse al tiempo necesario para el envío de cada ráfaga, ya que para tamaños de ráfagas menores (lo que equivale a un mayor envío de ráfagas), aumenta el número de ciclos por píxeles.

6.2. Resultados de la implementación de la arquitectura para el cálculo de la suma de las dos primeras escalas de la DWT *à trous*

6.2.1. Co-simulación (simulación software) en CoDeveloper

En esta sección se va a validar la correcta conexión entre procesos a través de los *streams*, tal y como se definen en la función de configuración. Para ello se utilizará la herramienta *Application Monitor* de CoDeveloper.

Siguiendo el mismo procedimiento que en el test de la convolución 3x3, realizamos una compilación software del proyecto, la cual genera un ejecutable “de escritorio” para realizar una simulación software del sistema. En el lanzamiento de la simulación, si se ha seleccionado previamente la opción *Launch Application Monitor* en la pestaña *Simulate* del menú *Project→ Options*, aparece la herramienta *Application Monitor*. En la pestaña *Block Diagram* se muestra el siguiente diagrama de procesos:

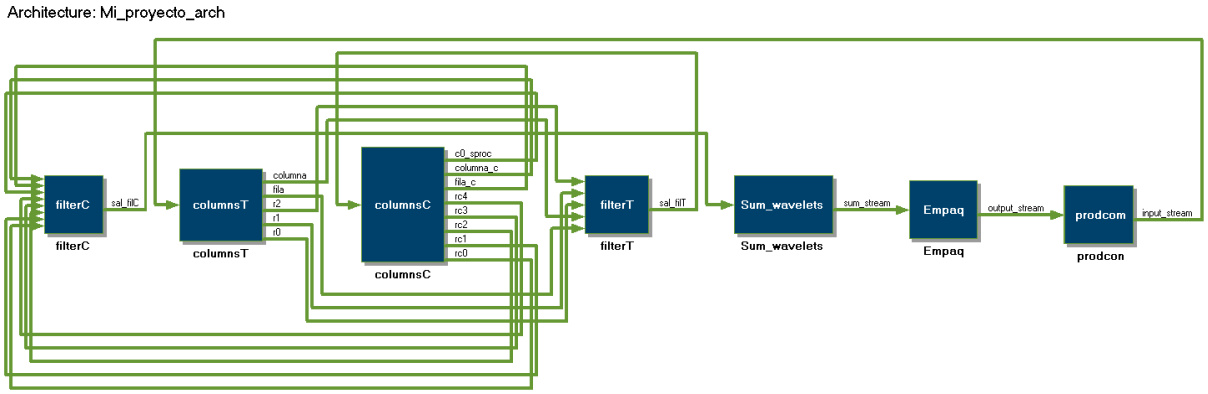

Figura 6.12. Diagrama de la arquitectura que muestra la comunicación de los procesos a través de los *streams*.

Analizando visualmente los *streams* que comunican los distintos procesos, se puede comprobar que la descripción del sistema se ha llevado a cabo de forma correcta.

Si se selecciona la pestaña correspondiente al proceso *Sum_wavelet* pueden comprobarse los resultados de las variables leídas del proceso *filterC* (c0, c1 y c2) y el resultado de la suma de las dos primeras escalas *wavelet* à trous (w1+w2), así como la evolución dinámica de la carga de datos en los stream, lo que nos puede ayudar a determinar el tamaño adecuado de sus respectivas FIFOs (identificando posibles saturaciones o esperas de datos).

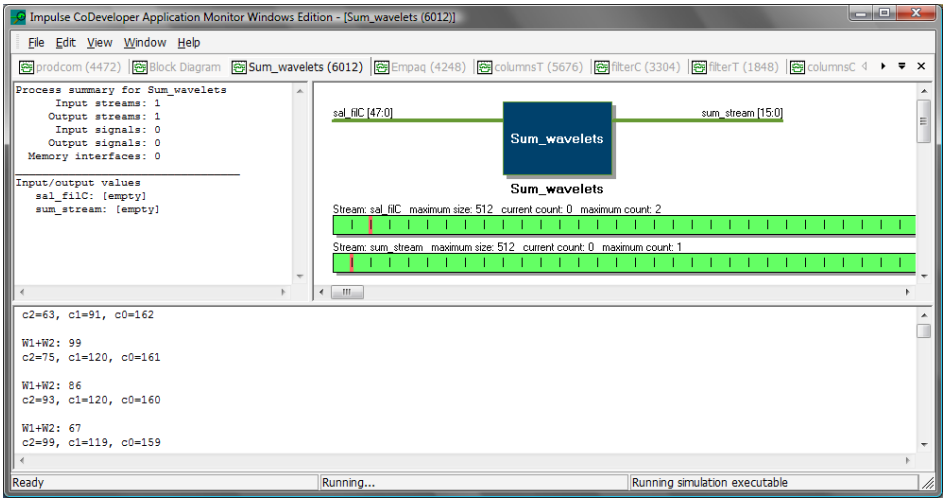

Figura 6.13. Proceso *Sum_wavelets*.

6.2.2. Resultados de la generación hardware en CoDeveloper para la plataforma DRC-RPU110-L200

Como se indicó en la sección 4.1.3 del capítulo 4, para la generación hardware debe elegirse la plataforma “DRC RPU110-L200(VHDL)”. En la generación hardware el optimizador (*Stage Master*) divide el código C en bloques básicos y asigna cada operación de cada bloque a una o más etapas. Las sentencias pertenecientes a una etapa se implementan como lógica combinacional ejecutándose en un único ciclo de reloj. En todos los procesos, el primer y último bloque tienen asignadas las operaciones de abrir y cerrar los *streams*, respectivamente. Los resultados para cada proceso se muestran a continuación:

```
|---------------------------------------
| columnsT
|---------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 loop:
|    Stages: 5
|    Max. Unit Delay: 17
|  Block #2 loop:
|    Stages: 7
|    Max. Unit Delay: 0
|  Block #3 loop:
|    Stages: 5
|    Max. Unit Delay: 0
|  Block #4 pipeline:
|    Latency: 3
|    Rate:    1
|    Max. Unit Delay: 64
|    Effective Rate: 64
|  Block #5:
|    Stages: 1
|    Max. Unit Delay: 17
|  Block #6:
|    Stages: 1
|    Max. Unit Delay: 1
|  Block #7:
|    Stages: 1
|    Max. Unit Delay: 0
|---------------------------------------
| Operators:
|  3 Adder(s)/Subtractor(s) (9 bit)|  6
Adder(s)/Subtractor(s) (16 bit)
|  1 Adder(s)/Subtractor(s) (32 bit)
|  1 Comparator(s) (2 bit)
|  8 Comparator(s) (17 bit)
|  1 Comparator(s) (32 bit)
|---------------------------------------
| Total Stages: 14
| Max. Unit Delay: 64| Estimated DSPs: 0
|---------------------------------------
```

```
|---------------------------------------
| columnsC
|---------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0|  Block #1 loop:
|    Stages: 3
|    Max. Unit Delay: 17
|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #3 loop:
|    Stages: 3
|    Max. Unit Delay: 17
|  Block #4 loop:
|    Stages: 7
|    Max. Unit Delay: 0
|  Block #5 loop:
|    Stages: 5
|    Max. Unit Delay: 0
|  Block #6 pipeline:
|    Latency: 3
|    Rate:    1
|    Max. Unit Delay: 17
|    Effective Rate: 17
|  Block #7:
|    Stages: 1
|    Max. Unit Delay: 17
|  Block #8:
|    Stages: 1
|    Max. Unit Delay: 1
|  Block #9:
|    Stages: 1
|    Max. Unit Delay: 0
|---------------------------------------
| Operators:
|  8 Adder(s)/Subtractor(s) (16 bit)
|  3 Comparator(s) (2 bit)
|  7 Comparator(s) (17 bit)
|  1 Comparator(s) (32 bit)
|---------------------------------------
| Total Stages: 16
| Max. Unit Delay: 17
| Estimated DSPs: 0
|---------------------------------------
```

```
|---------------------------------------
| filterT
|---------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 pipeline:
|    Latency: 9
|    Rate:    1
|    Max. Unit Delay: 34
|    Effective Rate: 34
|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|---------------------------------------
```

```
|---------------------------------------
| filterC
|---------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 pipeline:
|    Latency: 9
|    Rate:    1
|    Max. Unit Delay: 34
|    Effective Rate: 34
|  Block #2:|    Stages: 1
|    Max. Unit Delay: 0
|---------------------------------------
| Operators:
```

```
| Operators:
|  10 Adder(s)/Subtractor(s) (32 bit)
|  1 Comparator(s) (9 bit)
|  4 Comparator(s) (17 bit)
|  1 Comparator(s) (32 bit)
|---------------------------------------
| Total Stages: 11
| Max. Unit Delay: 34
| Estimated DSPs: 0
|---------------------------------------
```

```
|  10 Adder(s)/Subtractor(s) (32 bit)
|  1 Comparator(s) (9 bit)
|  8 Comparator(s) (17 bit)
|  1 Comparator(s) (32 bit)
|---------------------------------------
| Total Stages: 11
| Max. Unit Delay: 34
| Estimated DSPs: 0
|---------------------------------------
```

```
|---------------------------------------
| Sum_wavelets
|---------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 pipeline:
|    Latency: 3
|    Rate:    1|    Max. Unit Delay: 32
|    Effective Rate: 32
|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|---------------------------------------
| Operators:
|  3 Adder(s)/Subtractor(s) (16 bit)
|  1 Comparator(s) (2 bit)
|---------------------------------------
| Total Stages: 5
| Max. Unit Delay: 32
| Estimated DSPs: 0
|---------------------------------------
```

```
|---------------------------------------
| Empaq
|---------------------------------------
|  Block #0:
|    Stages: 1
|    Max. Unit Delay: 0
|  Block #1 pipeline:
|    Latency: 2
|    Rate:    1
|    Max. Unit Delay: 16
|    Effective Rate: 16
|  Block #2:
|    Stages: 1
|    Max. Unit Delay: 0
|---------------------------------------
| Operators:
|  3 Adder(s)/Subtractor(s) (16 bit)
|  1 Comparator(s) (2 bit)
|  4 Comparator(s) (17 bit)
|---------------------------------------
| Total Stages: 4
| Max. Unit Delay: 16| Estimated DSPs: 0
|---------------------------------------
```

Además de la información relativa a los bloques y etapas necesarias en los mismos, bajo las etiquetas *Operators* se muestran los recursos hardware requeridos en cada proceso implementado. Esta información es útil a la hora de comprobar si se han inferido los recursos esperados o sería conveniente expresar de otro modo alguna operación en el código C.

Puede observarse que a diferencia de la implementación de la convolución de 3x3 genérica, en esta implementación, para realizar la suma de productos relativa a la convolución, tanto en el proceso *filterT* como en el *filterC*, no se utilizan multiplicadores. Esto se debe a que los coeficientes de las máscaras de convolución (definidos en el capítulo 5) son potencias de 2, y por tanto, las multiplicaciones se realizan mediante desplazamientos lógicos.

El optimizador trata de determinar el mínimo número de etapas en la implementación de cada bloque básico, pero esto no siempre es la mejor opción. Por otro lado, para una buena optimización del código C, es importante que a la hora de programar, se tenga en cuenta la posible interpretación del por parte del optimizador de CoDeveloper. En este sentido, las medidas tomadas que han permitido un mayor control sobre la optimización hardware en el proceso *columnsT*, son las mismas que se describen en la sección 6.1.1 para el caso de la convolución genérica. En cuanto a los procesos *filterT* y *filterC*, las medidas tomadas también son análogas a las indicadas para el proceso *filterT* de la convolución genérica.

Por tanto, van a describirse los resultados en cuanto a bloques y etapas asignadas en la implementación del proceso *columnsC*. En la tabla anterior de resultados de generación hardware puede observarse que este proceso tiene la misma latencia y tasa que *columnsT*, pero una unidad máxima de retardo de 17, lo que es bastante inferior a la de *columnsT*. En este proceso, la lectura de datos requiere menos lógica de control ya que se lee del *stream* procedente del *filterT* en cada ciclo. En el caso de *columnsT*, se leen del *stream* procedente del software, paquetes de 4 datos, por lo que la lectura debe realizarse cada 4 ciclos de reloj.

En la Figura 6.14 se muestran las operaciones del proceso *columnsC* asignadas a las dos primeras etapas del *pipeline*.

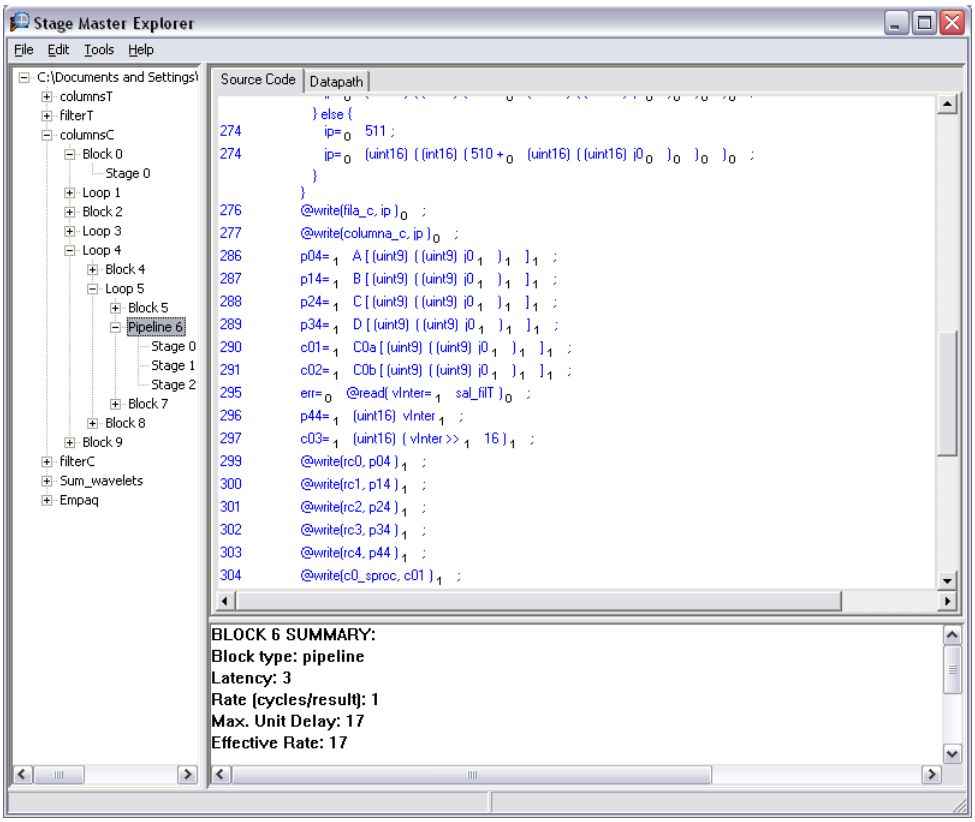

Figura 6.14. Etapas del proceso *columnsC*. Se indica con '0' o '1' la etapa a la que pertenece cada operación.

En la etapa 0 se calcula el valor de las variables `ip` y `jp` y se envían al proceso *filterC* para el cálculo de la máscara de condiciones de contorno. En la etapa 1 se lee la posición `j` de los arrays `A[]`, `B[]`, `C[]`, `D[]`, `C0a[]` y `C0b[]`, y los nuevos datos del *stream* procedente de *filterT*, que se almacenan en las variables `p44` y `c03`, y se envía la columna correspondiente al proceso *filterC*, junto el dato `c01`, por los *streams* `rc0`, `rc1`, `rc2`, `rc3`, `rc4` y `s0_cproc`, respectivamente.

En la Figura 6.15 se muestra que la actualización de los *arrays* se realiza de forma paralela en la etapa 2.

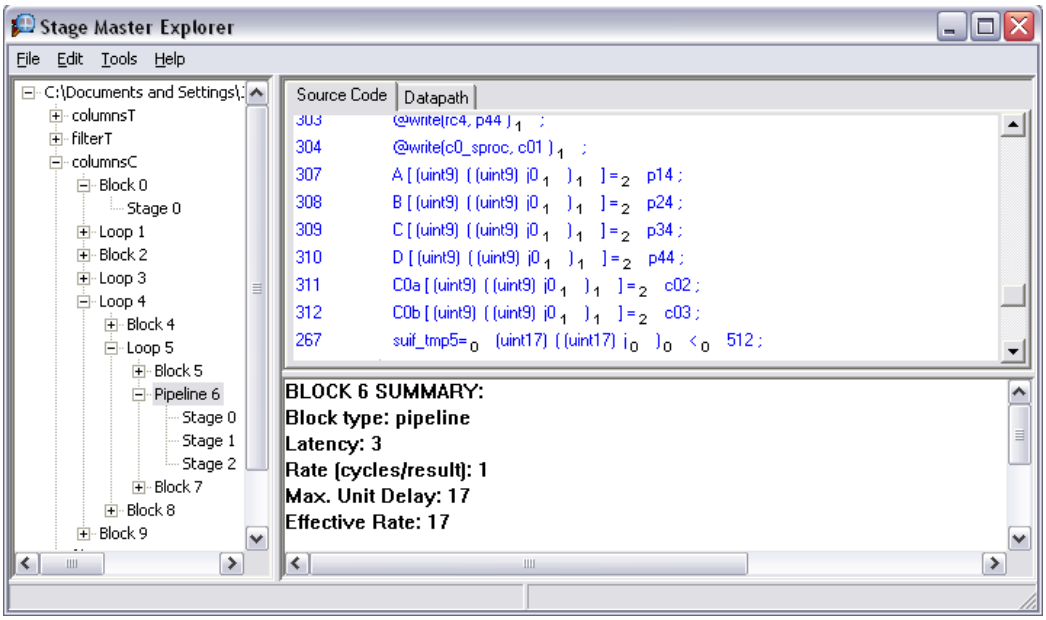

Figura 6.15. Etapas del proceso *columnsC*. Se indica con '0', '1' o '2' la etapa a la que pertenece cada operación.

Por último se van a mostrar las etapas asignadas a las operaciones del proceso *Sum_wavelet* que se realizan en *pipeline*. Se observa que en la etapa 1 se leen del *stream* procedente del proceso *filterC* los valores `c0`, `c1` y `c2`, necesarios para el cálculo de la suma de las dos primeras escalas wavelet, `w1+w2`, que se realiza en la etapa 2.

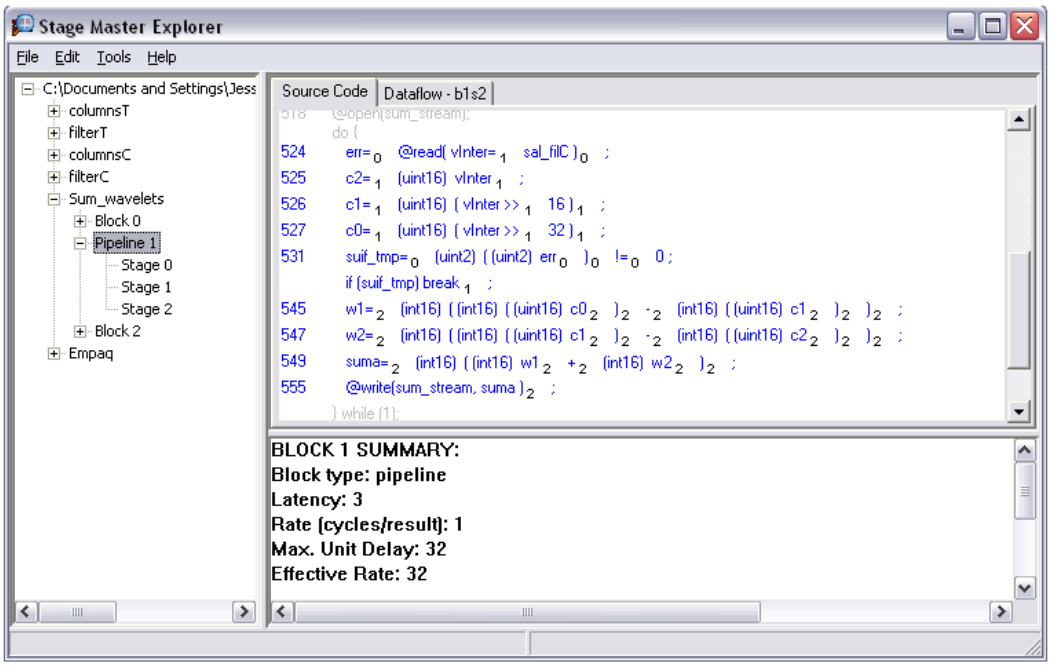

Figura 6.16. Etapas del proceso *Sum_wavelet*. Se indica con '0', '1' o '2' la etapa a la que pertenece cada operación.

6.2.3. Resultados de la síntesis e implementación del hardware en el entorno ISE de Xilinx sobre una plataforma FPGA Virtex4

En esta sección se mostrarán los resultados de la síntesis del proyecto *"user_logicVHDL.ise"*, que tal y como se indicó en la sección 4.2.1 contiene los procesos hardware generados en CoDeveloper. Los resultados de implementación se obtendrán tras la fase de *place&route* del proyecto *"rpware.ise"*, que como se indicó en la sección 4.2.3, incluye además de los procesos de la arquitectura hardware, la interfaz de comunicaciones con la parte software residente en el microprocesador.

6.2.3.1 Resultados de Síntesis

En la síntesis del sistema, se traducirá el hardware programado en un modelo a nivel de transferencia de registros (RTL).

Para analizar dicha representación se debe acceder a la opción *View RTL Schematic* de la pestaña *Synthesize*. Los resultados se muestran en la Figura 6.17, donde pueden identificarse los *streams* de entrada/salida del sistema. Para analizar los distintos niveles de representación se dispone de las opciones "*Back a Schematic"* y "*Forward a Schematic"*.

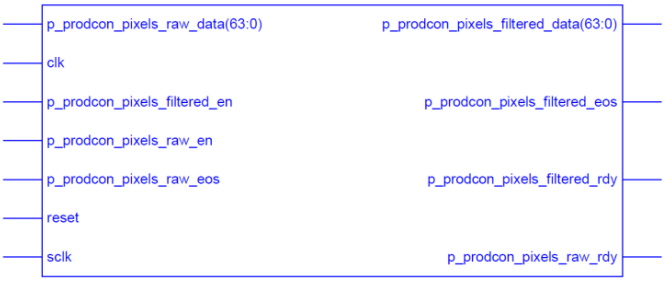

Figura 6.17. Esquemático del sistema.

Al sintetizar también se genera el informe de síntesis (*Synthesis Report*), donde se muestran estimaciones de área y velocidad. En esta implementación se ha sintetizado seleccionando inicialmente la velocidad como objetivo de optimización en las opciones de síntesis.

A continuación se muestra el informe de síntesis HDL (HDL Synthesis Report) donde se indican los recursos hardware requeridos:

```
HDL Synthesis Report

Macro Statistics
# RAMs                                          : 34
 512x16-bit dual-port RAM                      : 16
 512x17-bit dual-port RAM                      : 14
 512x33-bit dual-port RAM                      : 1
 512x49-bit dual-port RAM                      : 1
 512x65-bit dual-port RAM                      : 2
# Adders/Subtractors                           : 72
 16-bit adder                                  : 8
 16-bit subtractor                             : 6
 32-bit adder                                  : 19
 9-bit adder                                   : 39
# Counters                                     : 36
 9-bit up counter                              : 36
# Registers                                    : 304
```

```
 1-bit register                                    : 75
 16-bit register                                   : 62
 17-bit register                                   : 28
 2-bit register                                    : 1
 3-bit register                                    : 3
 32-bit register                                   : 85
 33-bit register                                   : 2
 4-bit register                                    : 3
 48-bit register                                   : 8
 49-bit register                                   : 2
 64-bit register                                   : 2
 65-bit register                                   : 4
 8-bit register                                    : 6
 9-bit register                                    : 23
# Comparators                                      : 50
 17-bit comparator greater                         : 3
 17-bit comparator less                            : 9
 32-bit comparator greater                         : 2
 9-bit comparator equal                            : 36
# Multiplexers                                     : 2
 32-bit 4-to-1 multiplexer                         : 2
```

En el siguiente informe generado, *Advanced HDL Synthesis Report*, se reconocen e implementan además registros de desplazamiento dinámico, máquinas de estado y multiplicadores *pipeline*.

```
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                             : 26
 512x16-bit dual-port block RAM                    : 8
 512x17-bit dual-port block RAM                    : 14
 512x33-bit dual-port block RAM                    : 1
 512x49-bit dual-port block RAM                    : 1
 512x65-bit dual-port block RAM                    : 2
# Adders/Subtractors                               : 72
 12-bit adder                                      : 2
 16-bit adder                                      : 8
 16-bit subtractor                                 : 6
 28-bit adder                                      : 1
 32-bit adder                                      : 16
 9-bit adder                                       : 39
# Counters                                         : 36
 9-bit up counter                                  : 36
# Registers                                        : 3379
 Flip-Flops                                        : 3379
# Comparators                                      : 50
 17-bit comparator greater                         : 3
 17-bit comparator less                            : 9
 32-bit comparator greater                         : 2
 9-bit comparator equal                            : 36
# Multiplexers                                     : 2
 32-bit 4-to-1 multiplexer                         : 2
```

En la siguiente figura se muestra el número de recursos utilizados de la FPGA seleccionada, así como el porcentaje de utilización de cada tipo, lo que permite de una forma bastante aproximada conocer el tamaño de la implementación en la FPGA.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2631	89088	2%
Number of Slice Flip Flops	3117	178176	1%
Number of 4 input LUTs	4226	178176	2%
Number of bonded IOBs	0	960	0%
Number of FIFO16/RAMB16s	29	336	8%

Figura 6.18. Estimación de los recursos utilizados en la FPGA tras la síntesis.

Al final de la síntesis se genera un informe de tiempos (*TIMING REPORT*), que muestra una estimación, cuya bondad se podrá medir en el informe generado tras la fase *Place&Route* de la implementación del diseño, que ya muestra resultados precisos. En base a esta estimación se escoge el reloj de 133 MHz.

```
Timing Summary:
---------------
Speed Grade: -11

   Minimum period: 6.777ns (Maximum Frequency: 147.549MHz)
   Minimum input arrival time before clock: 1.697ns
   Maximum output required time after clock: 4.082ns
   Maximum combinational path delay: No path found
```

Este informe estima el periodo mínimo del ciclo de reloj a 6.56 ns, asignando el 39.7% del mismo a la lógica, y el 60.3% al rutado.

```
    Total                      6.777ns (2.114ns logic, 4.663ns route)
                                       (31.2% logic, 68.8% route)
```

6.2.3.2 Resultados de Implementación

Tras la implementación del proyecto en ISE "rpware" descrito en la sección 4.2, el informe *Static Timing Report*, generado en el proceso de *Place&Route*, indica que la velocidad máxima (113.417 MHz) es inferior a la seleccionada (133MHz), por lo que la estimación en la fase de síntesis, cuyos resultados se muestran en el apartado previo, era más optimista. A pesar de de esto, se comprueba que el sistema funciona con el reloj de 133MHz, ya que las estimaciones del ISE en la implementación son bastante "conservadoras".

```
Timing summary:
---------------
Timing errors: 0  Score: 0

Constraints cover 46036 paths, 0 nets, and 12447 connections

Design statistics:
   Minimum period:   8.817ns   (Maximum frequency: 113.417MHz)
   Maximum path delay from/to any node:   8.817ns
```

A continuación se muestra una tabla resumen con los recursos de la FPGA Virtex 4 utilizados en la implementación del sistema:

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	17,326	178,176	9%	
Number of 4 input LUTs	18,928	178,176	10%	
Logic Distribution				
Number of occupied Slices	14,995	89,088	16%	
Number of Slices containing only related logic	14,995	14,995	100%	
Number of Slices containing unrelated logic	0	14,995	0%	
Total Number of 4 input LUTs	20,282	178,176	11%	
Number used as logic	17,611			
Number used as a route-thru	1,354			
Number used for Dual Port RAMs	1,282			
Number used as Shift registers	35			
Number of bonded IOBs	624	960	65%	
IOB Flip Flops	509			
IOB Dual-Data Rate Flops	242			
IOB Master Pads	57			
IOB Slave Pads	57			
Number of BUFG/BUFGCTRLs	15	32	46%	
Number used as BUFGs	15			
Number of FIFO16/RAMB16s	71	336	21%	
Number used as RAMB16s	71			
Number of DCM_ADVs	5	12	41%	
Number of PMCDs	2	8	25%	
Number of BUFRs	3	48	6%	
Number of ISERDESs	11	960	1%	
Number of OSERDESs	19	960	1%	
Number of BSCAN_VIRTEX4s	1	4	25%	
Number of IDELAYCTRLs	32	32	100%	
Number of BUFIOs	1	64	1%	

Figura 6.19. Recursos utilizados en la FPGA tras la implementación.

6.2.4. Análisis de tiempos de co-ejecución del sistema sobre la plataforma DRC-DS1002 y comparación con versiones All-software

En esta sección se van a mostrar los resultados temporales obtenidos en la ejecución del sistema sobre la plataforma DRC-DS1002. Estos resultados consistirán en la medición del tiempo requerido en el cálculo de la suma de las dos primeras escalas de la DWT *à trous* para 100 imágenes de 512x512 píxeles, y obtener, a partir de este dato, el tiempo medio empleado por imagen. Por último se realizará un comparativa de tiempos que evalúe la aceleración hardware conseguida, así como la mejora respecto a arquitecturas mixtas hw/sw implementadas en trabajos previos.

- **Análisis de tiempos de co-ejecución en la plataforma DRC-DS1002**

Con el fin de que la lectura y escritura de ficheros no interfiera en las mediciones, el software empleado almacena previamente las 100 imágenes a procesar, y de la misma forma, la escritura de ficheros de resultados se realizará una vez termina el procesado de todas ellas. Por tanto, la medida de tiempos se realizará desde que comienzan a enviarse los datos hasta que se recibe el último resultado.

Se comprueba que si se fija el valor de la variable `TAMRAFAGA` a 1 (un paquete de 4 datos de 16 bits por cada envío), se obtienen los ficheros de resultados correctos para las 100 imágenes.

A continuación, se prueba el funcionamiento del sistema para envíos mediante ráfagas de distintos tamaños. Se comprueba que para ráfagas de 32 y 64 paquetes de datos, el sistema funciona correctamente, ya que los ficheros de resultados contienen los datos correctos.

Para el envío de ráfagas de mayor tamaño ha sido necesario introducir un retardo entre el envío de cada ráfaga a la FPGA, ya que se ha comprobado que de otra forma los resultados no son los esperados. Como ya se indicó en los resultados temporales para el cálculo de una convolución genérica de 3x3, en la comuninación hw/sw no se soporta *back-pressure*. Esto puede explicar por tanto, la necesidad de dicho retardo entre el envío de ráfagas, ya que con una espera adecuada antes de introducir una nueva ráfaga en el *stream*, da tiempo a que el búffer del *stream* se vacíe lo suficiente para contener una ráfaga completa, evitando así una pérdida de datos. Estos retardos se introducen en el software mediante bucles *dummies* de la forma: '`for(a=0;a<nit;a++)b=b+1;`'. El número de iteraciones del bucle `nit` se calcula de forma iterativa analizando los resultados obtenidos.

Se ha comprobado, que para ráfagas de tamaño 128, el número de iteraciones del bucle de retardo entre el envío de cada ráfaga debe ser de 700 como mínimo, ya que de otra forma los resultados devueltos no son correctos. Con este retardo mínimo, se procesan las 100 imágenes en 0.36 segundos (0.0036 seg/imagen), lo que supone una velocidad de 277 imágenes por segundo. Se observa que si el retardo entre el envío de ráfagas es excesivo también devuelve datos erróneos. El retardo máximo admitido los constituyen unas 6000 iteraciones del bucle, ya que retardos superiores devuelven datos erróneos.

Si las ráfagas son de tamaño 256, el retardo mínimo viene dado por unas 3500 iteraciones del bucle `for` descrito, y el máximo por unas 10000 iteraciones.

Los mejores resultados temporales se obtienen para ráfagas de 256 paquetes, considerando el retardo mínimo necesario entre una ráfaga y la siguiente. En la siguiente figura aparecen los resultados que el proceso software imprime en la consola, donde se observa que el tiempo total obtenido para las 100 imágenes es de 0.33 segundos.

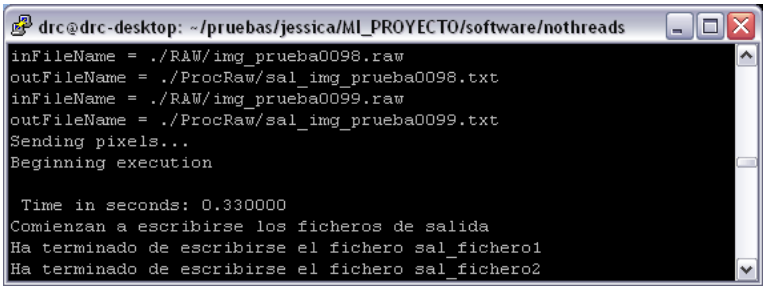

Figura 6.20. Resultados que el proceso software imprime por pantalla.

Esto supone un tiempo de procesado medio por imagen de 0.0033 segundos, lo que equivale a una velocidad de 303 imágenes por segundo.

Para tener una idea aproximada de la eficiencia del *pipeline* del hardware programado, se ha calculado el número medio de ciclos de reloj requeridos por el sistema para ofrecer un resultado. Según la ecuación (24), se consume 1,674 ciclos de reloj por píxel.

$$Tasa = \frac{(tiempo\ /\ imagen)(seg)}{(píxeles\ /\ imagen)\cdot periodo\_reloj(seg)} = \frac{0{,}0033}{(512\cdot 512)\cdot 7{,}52e^{-9}} = 1{,}674 \qquad (24)$$

En vista a los resultados se puede decir que conforme el tamaño de las ráfagas es mayor, el tiempo de procesado por imagen disminuye y consecuentemente, el número medio de ciclos por resultado también disminuye. Esto demuestra que el tiempo de escritura de ráfaga es considerable y por tanto, es preferible aumentar el tamaño de la ráfaga disminuyendo así el número de operaciones de escritura.

A continuación se muestra un gráfico que muestra el tiempo medio de procesado por imagen, en función del número de imágenes enviadas y el tamaño de ráfaga. Mediante la inversa de este tiempo se obtiene por tanto, el número de imágenes que el sistema es capaz de procesar en un segundo.

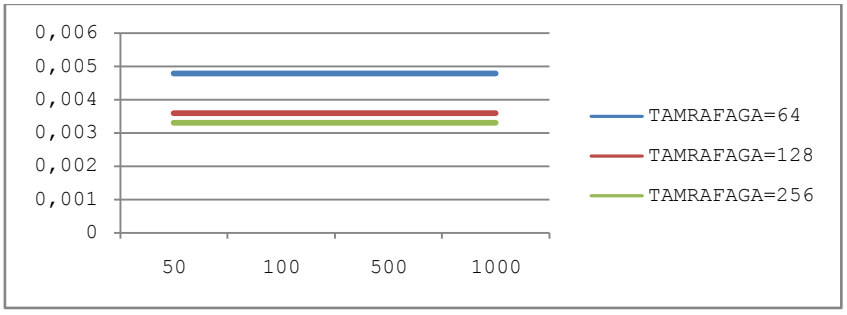

Figura 6.21. Tiempo (s) por imagen (eje vertical) en función del número de imágenes enviadas (eje horizontal) para distintos tamaños de ráfaga (en azul, rojo y verde).

Si en lugar de seleccionar el reloj de 133 MHz, se elige un reloj de 100MHz, el tiempo necesario para procesar las 100 imágenes aumenta. Este reloj, requiere la introducción de un retardo entre el envío de cada ráfaga mayor al que se calculó en el caso del reloj de 133MHz. Así, para ráfagas de 128 paquetes, la máxima velocidad obtenida es de 0.42 segundos por 100 imágenes (0.0042seg/imagen), lo que equivale a procesar 238 imágenes por segundo. En este caso ha sido necesario un retardo mínimo de 2500 iteraciones del bucle `for`, y se ha comprobado que para retardos superiores a 9000 iteraciones de dicho bucle, los resultados de los ficheros de salida no son los esperados. Por tanto, para relojes más lentos es necesario introducir retardos mayores entre el envío de ráfagas para obtener resultados correctos.

- **Comparativa de tiempos y aceleración hardware conseguida**

Con el fin de medir la aceleración hardware conseguida, se ha medido el tiempo de procesado por imagen en la herramienta CoDeveloper, instalada en una máquina *IntelCore2 Quad* a 2.40GHz. Para ello basta con enviar los datos paquete a paquete, ya que en la ayuda proporcionada por CoDeveloper se advierte que en simulación software, las ráfagas se implementan en múltiples operaciones, de forma que no se envía más de un paquete simultáneamente. Los resultados muestran que en media se tardan 12 segundos en procesar cada imagen de 512x512 píxeles. Este tiempo de procesado es excesivo, y puede deberse a que la ejecución en CoDeveloper utiliza los hilos (*threads*) de Windows, que no parecen muy eficientes a la hora de comunicar procesos.

Por este motivo se va a comprobar el tiempo en el que matlab es capaz de procesar las imágenes, utilizando funciones ya desarrolladas, también en la misma máquina (*IntelCore2 Quad* a 2.40GHz). Para ello se programa la siguiente función, que será llamada pasando como parámetro el número de iteraciones del bucle `for`, que determina el número de veces que se procesa la imagen.

```
function [] = tiempos(K)

X=imread('lena_grey.bmp');
C0=uint16(X);
c0=double(C0);
%-------------------------------------------------------------------------
coef_w2=double([1 0 2 0 1; 0 0 0 0 0; 2 0 4 0 2; 0 0 0 0 0; 1 0 2 0 1]);
coef_w1=double([1 2 1; 2 4 2; 1 2 1]);
%-------------------------------------------------------------------------
K = str2double(K);
tic

for i = 1:K

    C1=uint16(uint32(floor(conv2(c0,coef_w1,'same')/16)));
    c1=double(C1);
    C2=uint16(uint32(floor(conv2(c1,coef_w2,'same')/16)));

    %W1+W2
    w1=int16(C0)-int16(C1);
    w2=int16(C1)-int16(C2);
    suma=w1+w2;
end

t = toc;
%-------------------------------------------------------------------------
% Resultados
disp('Tiempo total: ');
t
disp('Por imagen:');
t/K
```

El tiempo medio obtenido por imagen es de 0.091 segundos, por lo que se pueden procesar 10,99 imágenes por segundo.

```
>> tiempos('20')

Tiempo total:
t = 1.8272

Por imagen:
ans =  0.0914
```

```
>> tiempos('100')

Tiempo total:
t = 9.1312

Por imagen:
ans = 0.0913
```

En la Tabla 6.2 se resumen los resultados de la comparativa, donde para los tiempos de la co-ejecución hw/sw se ha escogido el mejor caso, que se obtiene enviando ráfagas de 256 paquetes.

Especificaciones de la Ejecución			**Tiempo por Imagen (s)**
Microprocesador	*Sistema Operativo*	*Implementación*	
IntelCore2 Quad a 2.40GHz (PC)	Win XP	All Software (Ejecución en CoDeveloper)	12
IntelCore2 Quad a 2.40GHz (PC)	Win XP	All Software (Ejecución en Matlab)	0,091
Opteron a 2.2GHZ (DS1002)	Linux	Co-ejecución hw/sw	**0,0033**

Tabla 6.2. Comparativa del tiempo de ejecución del algoritmo implementado.

A la vista de los resultados queda claro que la solución más rápida es la obtenida mediante la co-ejecución del algoritmo, que consume solamente 0.0033 segundos/imagen. Esto es aproximadamente 27 veces más rápido que una implementación software en un PC estándar (Core2 Quad a 2.4GHz).

Por último también se pueden comparar los tiempos obtenidos en la co-ejecución hw/sw con los obtenidos en los trabajos previos sobre la misma plataforma cuyos algoritmos se especifican en la sección 2.3.

Trabajos Previos	Tiempo por Imagen (s)	Aceleración con la nueva implementación
PFC [**16**]	0.512	155x
Paper [**7**]	0.022	6,6x

Tabla 6.3. Comparativa con algoritmos de trabajos previos.

6.2.5. Análisis de resultados numéricos de la arquitectura que calcula la suma de las dos primeras escalas DWT à trous

En esta sección se verificará que los resultados obtenidos en los ficheros de salida son los correctos. Para ello se calcula en matlab la suma de las dos primeras escalas de la *DWT à trous* para la imagen '`lena_grey.bmp`' como se muestra en las siguientes líneas de código.

```
clc; close all; clear all;

%*********************************   CÁLCULOS MATLAB   ************************************
X=imread('lena_grey.bmp');
C0=uint16(X);
c0=double(C0);
%-------------------------------------------------------------------------
coef_w2=double([1 0 2 0 1; 0 0 0 0 0; 2 0 4 0 2; 0 0 0 0 0; 1 0 2 0 1]);
coef_w1=double([1 2 1; 2 4 2; 1 2 1]);
%-------------------------------------------------------------------------

C1=uint16(uint32(floor(conv2(c0,coef_w1,'same')/16)));
c1=double(C1);
C2=uint16(uint32(floor(conv2(c1,coef_w2,'same')/16)));

%W1+W2
w1=int16(C0)-int16(C1);
w2=int16(C1)-int16(C2);
sumaW=w1+w2;
```

A continuación se lee el fichero de salida '`sal_lena_grey16_le.txt`' en la co-ejecución sw/hw del sistema para dicha imagen de entrada, y se almacena en la matriz `sal_DRC`.

```
%*********************************   RESULTADOS DRC   ************************************
fid = fopen('sal_lena_grey16_le.txt', 'r');
sal_lena_DRC = fscanf(fid, '%g', [512 512]);
fclose(fid);
sal_DRC=int16(sal_lena_DRC');

%*****************************    COMPARACIÓN DE RESULTADOS  *****************************
COMPROBACION=sumaW-sal_DRC;
```

Finalmente se restan elemento a elemento los resultados obtenidos en la co-ejecución, con los obtenidos en matlab y el resultado de esta operación se almacena en la matriz `COMPROBACION`. Se comprueba así que los resultados obtenidos en la co-ejecución coinciden con los obtenidos con matlab y por tanto son correctos, ya que la matriz `COMPROBACION` sólo contiene ceros.

En la Figura 6.22 pueden analizarse visualmente los resultados de la suma de las dos escalas 1 y 2 de la DWT *á trous* (“sal_lena_grey16_le.txt”) junto a la imagen original (“lena_grey16_le.raw”) visualizadas en ImageJ.

Figura 6.22. A la izquierda: imagen original. A la derecha: imagen procesada.

De forma análoga se han comprobado los resultados para la pila de 100 imágenes dada correspondiente a un cúmulo globular M15. En la siguiente figura se muestra una de las imágenes de dicha pila (“img_prueba0076.raw”) junto con la imagen procesada correspondiente (“sal_img_prueba0076.txt”).

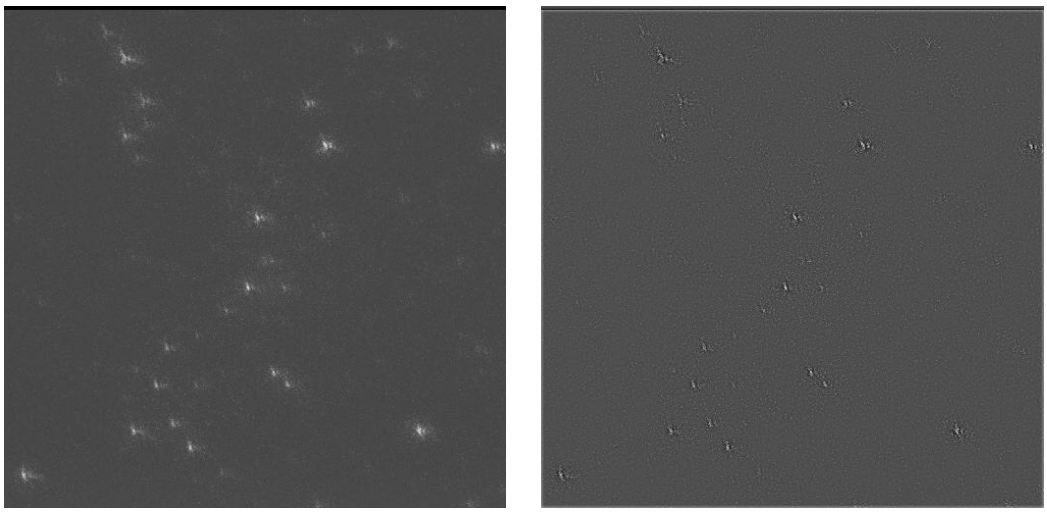

Figura 6.23. A la izquierda: imagen original. A la derecha: imagen procesada.

Capítulo 7

Conclusiones y Perspectivas de Futuro

- **Resumen y Conclusiones**

En el presente proyecto se implementa una arquitectura de co-ejecución hardware-software para la aceleración de un algoritmo de reducción de imágenes estelares basado en *wavelets*, apropiado para el instrumento FastCam.

En la implementación y ejecución del algoritmo se han utilizado dos herramientas punteras de gran utilidad en el desarrollo de aplicaciones mixtas hw/sw. La herramienta CoDeveloper de *Impulse Acelerated Technologies*, aún en fase de desarrollo, ha agilizado considerablemente el diseño de aplicación mixta hw/sw, gracias a la programación en C, evitando gestionar las tediosas sincronizaciones inherentes a la programación en VDHL. Por su parte, el nodo de supercomputación híbrido HPRC, modelo DS1002, de la empresa DRC, ha pemitido la co-ejecución hw/sw del algoritmo sobre un procesador convencional acelerado mediante un coprocesador hardware desarrollado sobre una FPGA virtex 4 de Xilinx.

En el capítulo 4 se ha establecido un flujo de trabajo detallado, dirigido a la implementación de futuras arquitecturas mixtas hw/sw sobre las herramientas de desarrollo descritas, que permiten maximizar el rendimiento de los algoritmos ejecutados.

En el capítulo 5 se han descrito en detalle dos arquitecturas mixtas hw/sw a nivel ESL. La primera, para el cálculo de una convolución genérica 3x3, y la segunda, más compleja, para el cálculo de la suma de las dos primeras escalas de la DWT *à trous*. En las arquitecturas propuestas se ha perseguido maximizar la paralelización del hardware con el fin de conseguir la máxima eficiencia en términos de rendimiento. Para ello se ha hecho uso de las ventajas ofrecidas por el modelo de programación de CoDeveloper, en cuanto a *pragmas* y comunicación entre procesos, y además se ha requerido un alto conocimiento de la arquitectura del HPRC DS1002 de DRC.

Finalmente en el capítulo 6 se ha procedido a cuantificar el rendimiento de las soluciones implementadas, de forma aislada y en comparación tanto con arquitecturas mixtas hw/sw previamente desarrolladas, como con soluciones software ejecutadas sobre un procesador común.

En cuanto al rendimiento de la primera arquitectura mixta sw/hw ejecutada sobre el computador HPRC DS1002, que realiza el cálculo de una convolución genérica 3x3, cabe destacar que se ha conseguido una aceleración hardware de 10,4x con respecto a la ejecución software en matlab. Se ha obtenido una velocidad de procesado de 400 imágenes por segundo.

La aceleración hardware conseguida en la segunda arquitectura, encargada del cálculo de la suma de las dos primeras escalas de la DWT *à trous*, ha sido de 27x respecto de la ejecución software en matlab. Además se han mejorado los resultados de rendimiento obtenidos en los trabajos previos, consiguiendo un procesado 155 veces superior a la arquitectura implementada en el PFC [**16**], y 6,6 veces más rápida a la implementada en el *paper* [**7**].

Estos resultados en términos de velocidad se complementan también con buenas prestaciones en cuanto al área del circuito. Una convolución consume en realidad menos del 1% de los recursos de la FPGA, mientras que el sistema completo con las dos convoluciones consume un 16%. Teniendo en cuenta por tanto que un 13-14% de recursos están invertidos en la interfaz de comunicaciones, que permanecerá constante, los recursos ofrecidos por el hardware permitirían la implementación de sistemas con una complejidad superior en varios ordenes de magnitud. Estos sistemas, al estar proyectados sobre circuitos hardware que funcionan de forma concurrente, seguirían procesando datos a una velocidad similar a los sistemas mostrados en este proyecto, al contrario que lo que ocurriría en su implementación software, que iría incrementando el tiempo de procesamiento de forma lineal. En estas circunstancias, sería viable conseguir incrementos de velocidad muy importantes, del orden de 100x o 200x.

- **Perspectivas de Futuro**

En este apartado se indicarán las líneas de mejora principales que se abren tras la implementación de esta arquitectura.

En primer lugar, se debería profundizar en el funcionamiento del envío a ráfagas entre la parte software y la parte hardware de la arquitectura, con el ánimo de evitar el uso de retardos entre ráfagas, motivados en principio por la falta de soporte de *back-pressure* en los mecanismos de comunicación hw/sw. Dado que ImpulseC, como se indica en los manuales, no soporta ráfagas, en este PFC no ha sido posible avanzar más en el tema. Está previsto que Impulse proporcione una nueva versión de su PSP (*Platform Support Package*) que soporte ráfagas, que permitirá una comunicación más eficiente.

De forma simultánea a este PFC, se han estado desarrollando nuevos algoritmos de reducción de imágenes basados en *wavelets*, que pueden implementarse con pequeñas modificaciones de la arquitectura aquí presentada, como estudiadas combinaciones lineales de escalas *wavelet*, la inclusión de un filtrado para eliminar ruido, etc. Por tanto, una futura línea de trabajo consistiría en extender las arquitectura mixta hw/sw para adaptarla a las nuevas versiones.

Como último apunte, convendría la inclusión de funcionalidades para el manejo de ficheros *FITS* para la entrada y salida de datos (imágenes), ampliamente utilizados en tratamiento de imágenes astronómicas, en lugar de archivos tipo *raw*, en base al objetivo final del sistema que se engloba este PFC.

Bibliografía

[1] "FastCam". [Online]. http://www.iac.es/proyecto/fastcam.

[2] Juan José Piqueras (autor) José Javier Martínez Álvarez (Director), "FastCam: Sistema de adquisición de imágenes astronómicas y procesado en tiempo real sobre hardware reconfigurable," UPCT, Proyecto Fin de Carrera (2007).

[3] L. Rodriguez-Ramos, Y. Martin, J.J. Martinez-Alvarez, J. Piqueras, "FastCam: Real-Time Implementation of the Lucky Imaging Technique using FPGA," *SPL*, pp. 26-28, 2008.

[4] Manual Fastcam. [Online]. http://www.iac.es/telescopes/Manuales/manualfastcam.pdf

[5] ANDOR Technology. [Online]. http://www.andor.com/scientific_cameras/ixon/models/default.aspx?iProductCodeID=3

[6] EMCCD. [Online]. http://www.emccd.com

[7] J. Javier Martínez, Isidro Villó, F. Javier Toledo, J. Manuel Ferrández Javier Garrigós, "Acceleration of a DWT-based Algorithm for Short Exposure Stellar Images Processing on a HPRC Platform," *fccm*, pp. 113-116, Mayo 2010, The 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines.

[8] Fried D. L., "Limiting Resolution Looking Down Through the Atmosphere," *J. Opt. Soc. Am.*, vol. 56, pp. 1380-1384, 1966.

[9] Fried D. L., "Optical Resolution Through a Randomly Inhomogeneous Medium for Very Long and Very Short Exposures," *J. Opt. Soc. Am.*, vol. 56, pp. 1372-1379, 1966.

[10] Fried D. L., "Probability of getting a lucky short-exposure image throughturbulence," *J. Opt. Soc. Am.*, vol. 68, pp. 1651-1658, 1978.

[11] Alastair R. Allen Stewart I. Fraser, "A Speckle Reduction Algorithm using The A'Trous Wavelet Transform," in *Proceedings of the IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001)*, Marbella, Spain, September 3-5, 2001.

[12] M.J. Shensa, "The Discrete Wavelet Transform: Wedding the A Trous and Mallat Algorithms," *IEEE Transactions on Signal Processing*, vol. 40 , no. 10, pp. 2464-2482, 1992.

[13] P. Tchamitchian, M. Holschneider, *"Les ondelettes"*. Ed. P.G. Lemarié, Springer-Verla, 1990.

[14] Stephane G. Mallat, "A Theory for Multiresolution Signal Decomposition," vol. II, no. 7, pp. 679-693, 1989.

[15] DRC Computers. (2009) DS1002 User Guide. http://www.drccomputer.com.

[16] José Javier (director), David Montoro Mouzo (autor), "Diseño de un nuevo algoritmo de reducción de imágenes astronómicas para el instrumento FastCam y su implementación sobre un supercomputador reconfigurable para ejecución en tiempo real," UPCT, Proyecto Fin de Carrera, 2009.

[17] Observatory Roque de los Muchachos. [Online]. http://www.iac.es/

[18] Scott Thibault, David Pellerin, *Practical FPGA Programming in C*.: Prentice-Hall, 2005.

[19] Impulse Accelerated Technologies Inc. (2009) [Online]. http://www.impulsec.com

[20] Impulse Accelerated Technologies, CoDeveloper Universal C to FPGA Compiler User and Help Guides, 2009.

[21] Dennis Ritchie, Brian Kernighan, *El lenguaje de programación C*.: Prentice Hall, 1988.

[22] Meyer-Baese, *Digital signal processing with field programmable gate arrays.*: Springer, 2007.

[23] Xilinx Inc. (2009) Application Notes –several numbers-. www.xilinx.com.

[24] Impulse Accelerated Tech. (2009) Application Notes –several numbers-. www.impulsec.com.

[25] Toledo, Martínez, Garrigós, *Síntesis de Sistemas Digitales con VHDL*.: UPCT, 2002.

[26] High Energy Astrophysics Science Archive Research Center. [Online]. http://heasarc.gsfc.nasa.gov/docs/heasarc/fits_overview.html

ANEXO I

Acceso Remoto
al Computador DRC

En este anexo se mostrarán una serie de herramientas que permitirán el acceso remoto al computador DRC, con el fin de agilizar la metodología de trabajo en el mismo. Esto sólo pretende ser una guía práctica con procedimientos básicos, pudiendo hallar más información en los manuales disponibles en la web, así como en la ayuda aportada por las aplicaciones.

- **Acceso remoto en modo línea de comandos y para transferencia de ficheros.**

El servidor DRC tiene por defecto instalado un servidor Telnet y SSH, así como un servidor FTP y SCP que se ejecutan durante el inicio del sistema. Por tanto, una forma sencilla de acceder a la máquina es utilizando cualquier aplicación cliente con soporte para estos protocolos.

En nuestro caso, utilizamos PuTTY, un cliente Telnet y SSH de código abierto para el acceso remoto al DRC en modo comando.

También se ha instalado WinSCP, cliente SFTP, FTP y SCP de código abierto cuya interfaz se puede seleccionar, similar al Explorer o a Total Commander (ésta es la mostrada en las figuras). Así, se muestran dos paneles (ver Figura 24), el izquierdo para el directorio local y el derecho para el remoto, de forma que es posible arrastrar ficheros de uno a otro fácilmente, así como visualizarlos.

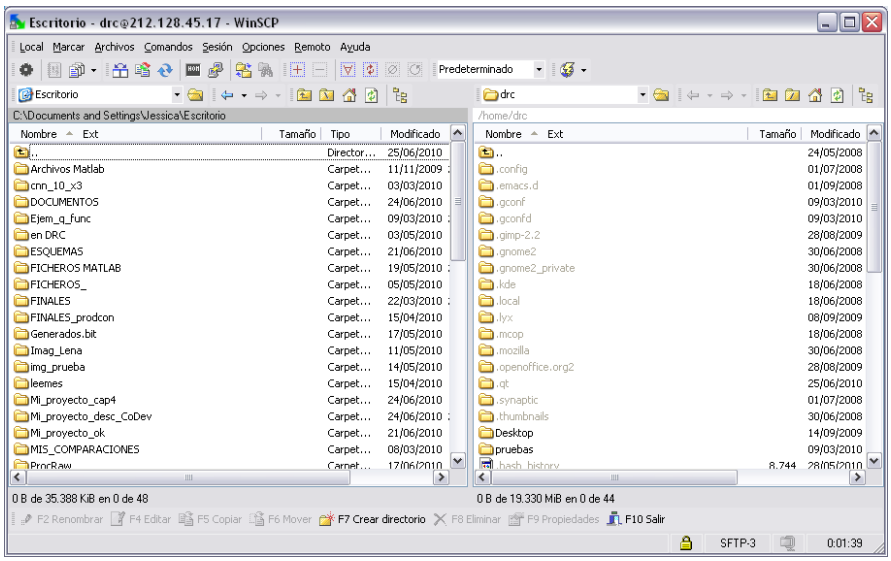

Figura 24. Interfaz de WinSCP.

Como se muestra en la siguiente figura, se debe comenzar configurando una sesión con la dirección IP del computador DRC, así como los parámetros seleccionados.

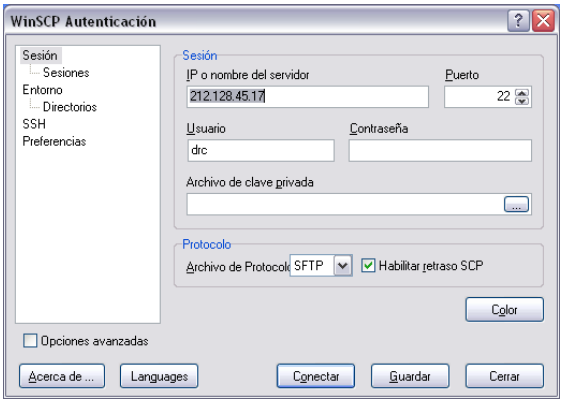

Figura 25. Configuración relativa a la sesión.

Una vez guardados los datos de la conexión, se pulsa en conectar y se introduce la contraseña “pk2a+a2” en el cuadro especificado (ver Figura 26). Previamente debe haberse iniciado sesión en el computador DRC. Para ello habrá que introducir como usuario "drc" y contraseña "pk2a+a2". El carácter "+" debe introducirse con el teclado numérico.

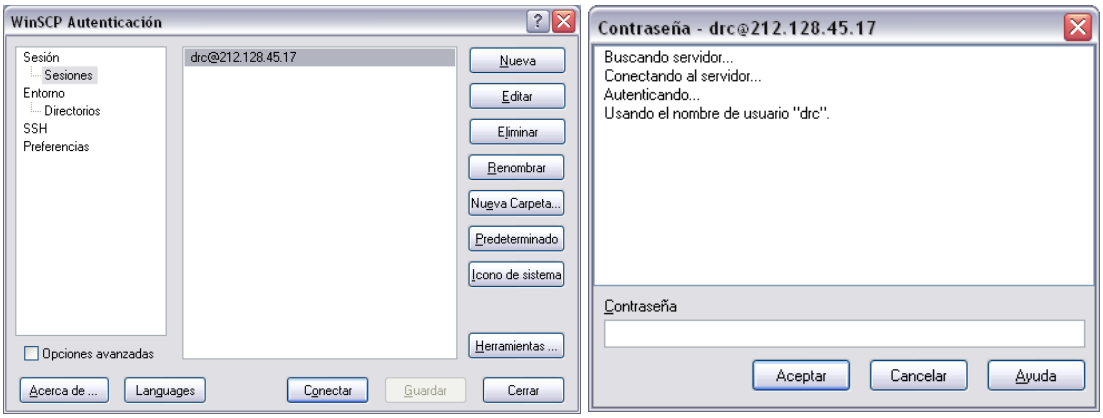

Figura 26. Conexión al DRC con WinSCP.

A través de este software se podrá acceder a PuTTY pinchando sobre el acceso directo situado en la barra de herramientas (ver Figura 27).

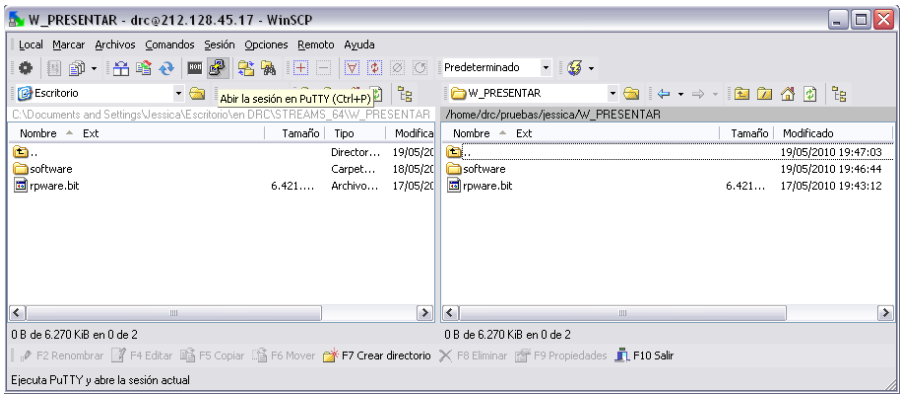

Figura 27. Acceso a PuTTY desde WinSPC.

La posibilidad de importar sesiones desde PuTTY, se habilita en la opción “Herramientas” del cuadro de diálogo inicial “WinSCP Autenticación” (Figura 28, izquierda). En el cuadro de la derecha de la Figura 28 se muestran las opciones del programa a través de la pestaña Preferencias. Se observa la configuración para integrar PuTTY como una herramienta externa.

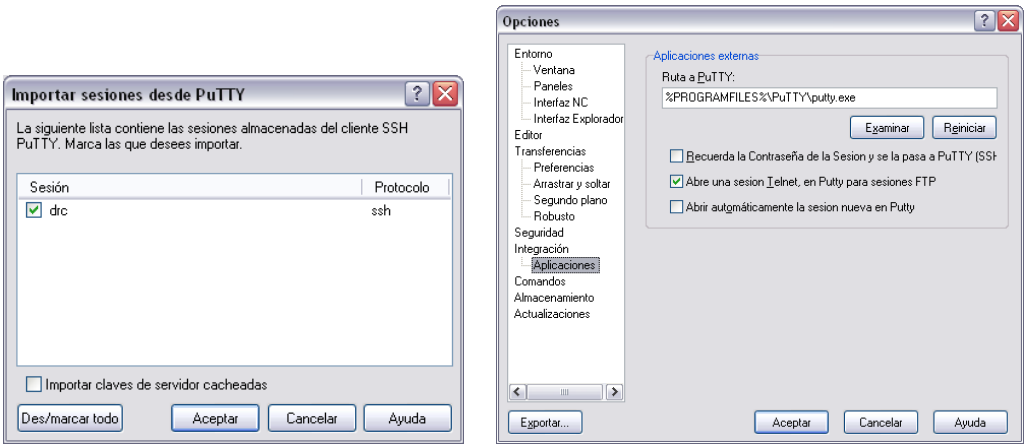

Figura 28. Configuración para la integración de PuTTY en WinSCP.

Una vez abierta la sesión en PuTTY habrá que introducir la contraseña “pk2a+a2” (Figura 29).

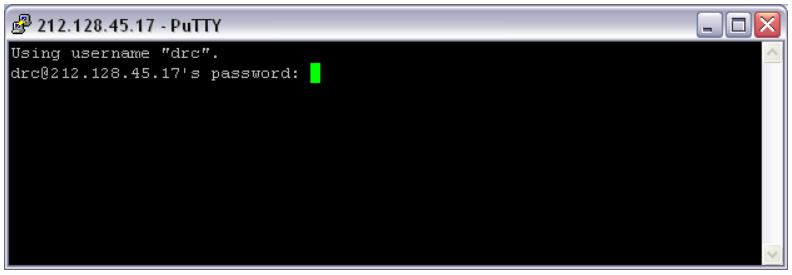

Figura 29. Conexión remota con PuTTY.

Para programar la RPU, como ya se indicó en el capítulo correspondiente, se deben seguir los siguientes pasos:

1) Copiar el fichero “*rpware.bit*” en el DRC, utilizando WinSCP.

2) Reconfigurar la RPU con el fichero “.bit”, abriendo para ello una sesión con PuTTY y ejecutando en ella los siguientes comandos:

    a. Ser root:
        $ su
        Password: drc

    b. Programar la fpga utilizando el comando: $ rpu_reconfigure rpware.bit –r

Como se observa en la Figura 30, se perderá la conexión, ya que la máquina necesita reiniciarse con la nueva programación de la FPGA.

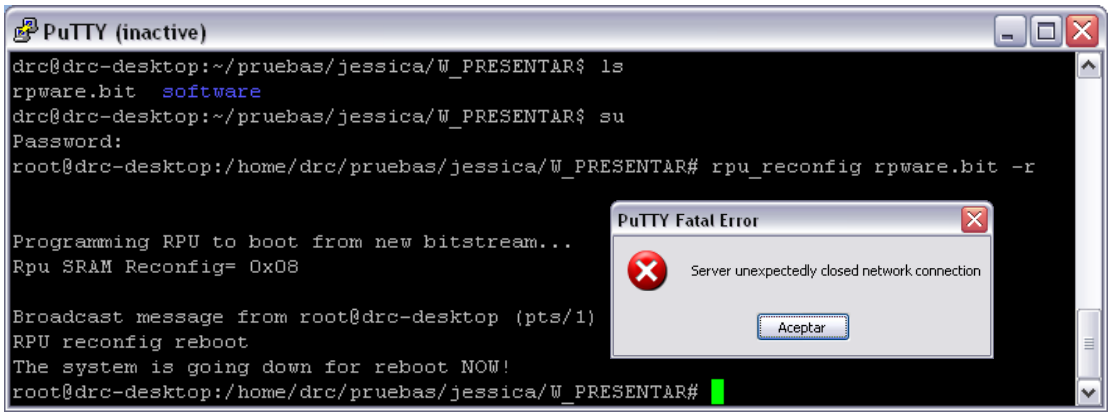

Figura 30. Programación remota de la FPGA.

Se vuelve a establecer la conexión con PuTTY, una vez reiniciado el DRC, y ya puede ejecutarse el software previamente compilado (‘make –f Makefile’), que se comunicará con el hardware que acabamos de configurar. Para ello introducir el comando ‘./Mi_proyecto_sw’ (ver Figura 31).

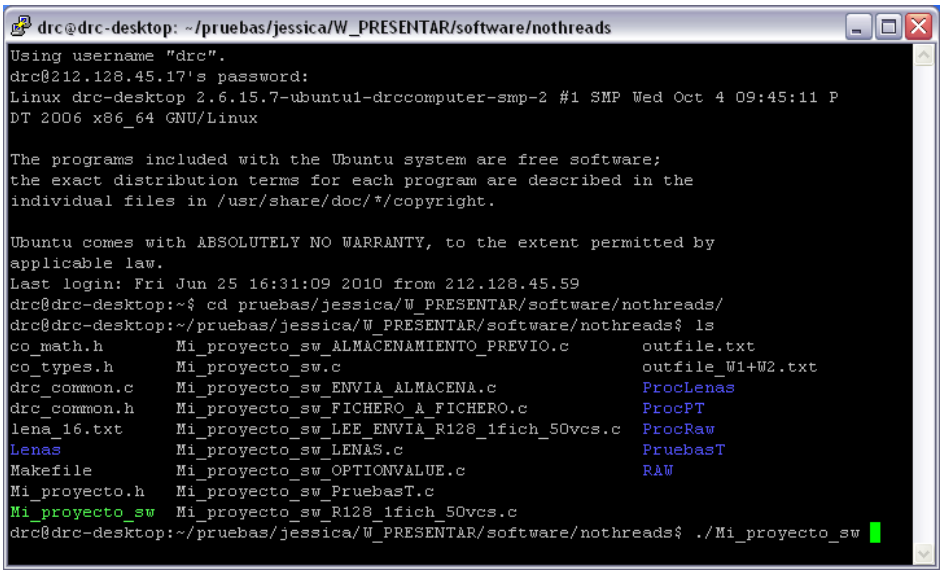

Figura 31. Ejecución de la aplicación desde un PC remoto.

- **Acceso remoto mediante VNC**

Otra forma más directa de acceder al servidor DRC sería mediante alguna utilidad que soporte los protocolos VNC (Virtual Network Computing). Para ello basta con configurar y lanzar la aplicación servidor en el DRC introduciendo la siguiente línea de comandos en la consola:

'x11vnc -shared -passwd kk'.

Una vez termine la configuración, nos avisará por consola con el siguiente mensaje:

```
The vnc desktop is drc-desktop:1
PORT=5901
```

En este momento podemos conectarnos desde otro PC, utilizando para ello cualquiera de los clientes VNC existentes. En nuestro caso,seleccionamos la herramienta de libre distribución UltraVNC, con la que podemos conectarnos introduciendo la dirección IP del DRC y el puerto indicado, tal y como se muestra en la Figura 32.

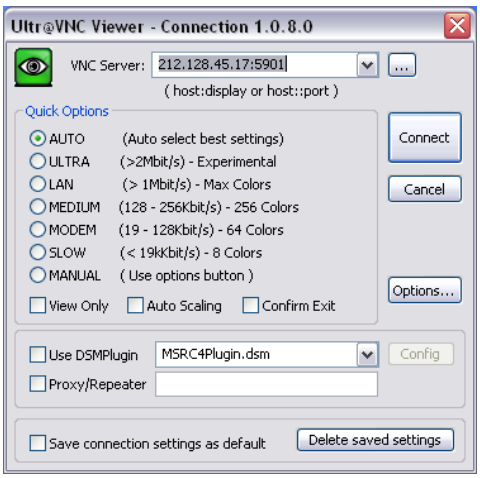

Figura 32. Conexión con VNC Viewer.

En la Figura 33 se muestra cómo nos pedirá la contraseña necesaria para la conexión solicitada (“kk”). Una vez aceptada nuestra solicitud aparecerá en la pantalla del ordenador el escritorio del computador DRC tal como puede verse en la Figura 34.

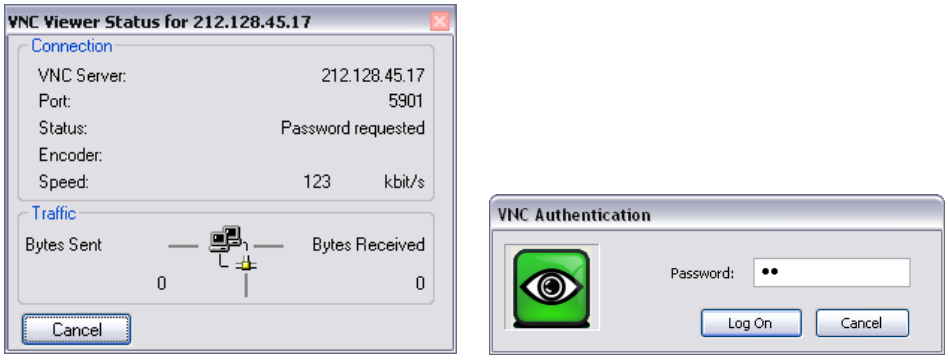

Figura 33. Petición de contraseña para la conexión solicitada.

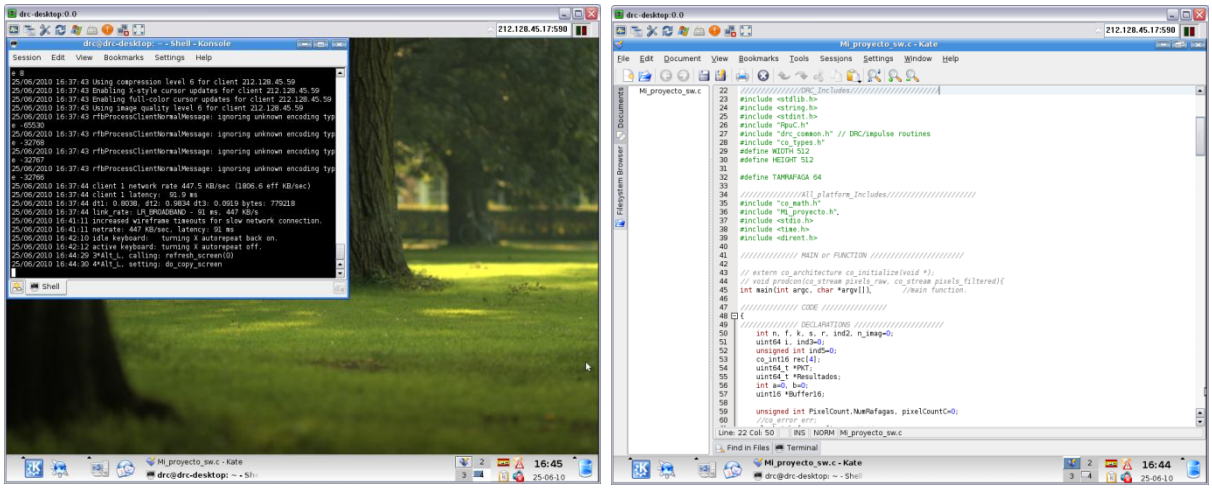

Figura 34. Visualización del computador DRC en un PC remoto.

De esta forma, será mucho más cómodo modificar el fichero software de nuestra aplicación a la hora de realizar pruebas, ahorrándonos la transferencia de archivos en cada cambio. Así se podrán realizar modificaciones, compilar y ejecutar como si estuviéramos delante del DRC, sin necesidad de una conexión con PuTTY. Ahora bien, cabe recordar que cada vez que se programe la FPGA, se reiniciará el DRC y se perderá la conexión. El problema con este método en ese caso es que, inmediatamente después del reinicio, ningún usuario ha hecho un *login* en el DRC, por lo que las librerías x11 no están cargadas, y resulta imposible lanzar la aplicación x11vnc que hace uso de ellas.

Por tanto, si después de hacer un reinicio no se tiene acceso físico al servidor DRC (para hacer un *login* que cargaría el entorno gráfico x11 automáticamente), habrá que utilizar el acceso mediante PuTTY, únicamente en modo línea de comandos (y WinSCP para la transferencia de ficheros).

Como conclusión se podría decir que VNC es más recomendable para las pruebas en el código del fichero software. Sin embargo, las conexiones con PuTTY son más rápidas si lo que se va a probar son distintos ficheros “.bit” que programen la FPGA.