UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Técnicas de compresión de imágenes hiperespectrales para aplicaciones espaciales

Autor: Aday García del Toro
Tutores: Dr. Roberto Sarmiento Rodríguez
Dr. José Fco. López Feliciano
Fecha: Diciembre 2012

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Técnicas de compresión de imágenes hiperespectrales para aplicaciones espaciales

HOJA DE FIRMAS

Alumno:	Aday García del Toro	Fdo.:
Tutor:	Dr. Roberto Sarmiento Rodríguez	Fdo.:
Tutor:	Dr. José Fco. López Feliciano	Fdo.:

Fecha: Diciembre 2012

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Técnicas de compresión de imágenes hiperespectrales para aplicaciones espaciales

HOJA DE EVALUACIÓN

Calificación: .....

Presidente Fdo.:

Secretario Fdo.:

Vocal Fdo.:

Fecha: Diciembre 2012

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Acrónimos

ASIC	Application-Specific Integrated Circuit
CCSDS	Consultative Committee for Space Data Systems
C-DPMC	Clustered Differential Pulse Code Modulation
DSP	Digital Signal Processor
DUV	Design Under Verification
ESA	European Space Agency
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLS	High Level Synthesis
LCE	Lossy Compression for ExoMars
RTL	Register Transfer Level
UTQ	Uniform-threshold Quantizer
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Memoria

1. Introducción y objetivos

La compresión de imágenes hiperespectrales se ha convertido en un foco de investigación a lo largo de los últimos años debido, principalmente, a la necesidad de adecuar la capacidad de adquisición de los sensores hiperespectrales con las capacidades de comunicación de los enlaces a tierra.

Todo tipo de imágenes espectrales, como multiespectrales, hiperespectrales y ultraespectrales, generan una gran cantidad de información que ha de ser comprimida con el fin de reducir el volumen de datos en la transmisión hacia el segmento terreno [1]. Los diferentes tipos de imágenes espectrales difieren en la cantidad de datos disponibles bien en la dimensión espacial o en la dimensión espectral, requiriendo potencialmente diferentes técnicas de compresión. Por ejemplo, típicamente las imágenes multiespectrales presentan una resolución espacial con alta granularidad mientras que la granularidad en resolución espectral no es tan alta. Por consiguiente, su compresión explota la correlación espacial. El caso opuesto ocurre con las imágenes de tipo hiper- y ultraespectrales, en las que la correlación espectral se presenta como factor dominante.

La compresión resulta de mayor utilidad para sensores espaciales ya que no es posible físicamente desmontar o leer la memoria de almacenamiento que almacena los datos adquiridos. Asimismo, si la resolución del sensor es muy alta o larga la duración de la misión, puede resultar necesaria la aplicación de técnicas de compresión para misiones embarcadas. Las técnicas de compresión de imágenes permiten la transmisión de un volumen mayor de datos en el mismo tiempo. La Figura 1.1 muestra el satélite de teledetección civil SPOT5 y su unidad de compresión. Asimismo, la Figura 1.2 muestra el satélite de observación terrestre PLEIADES-HR y su unidad de compresión.

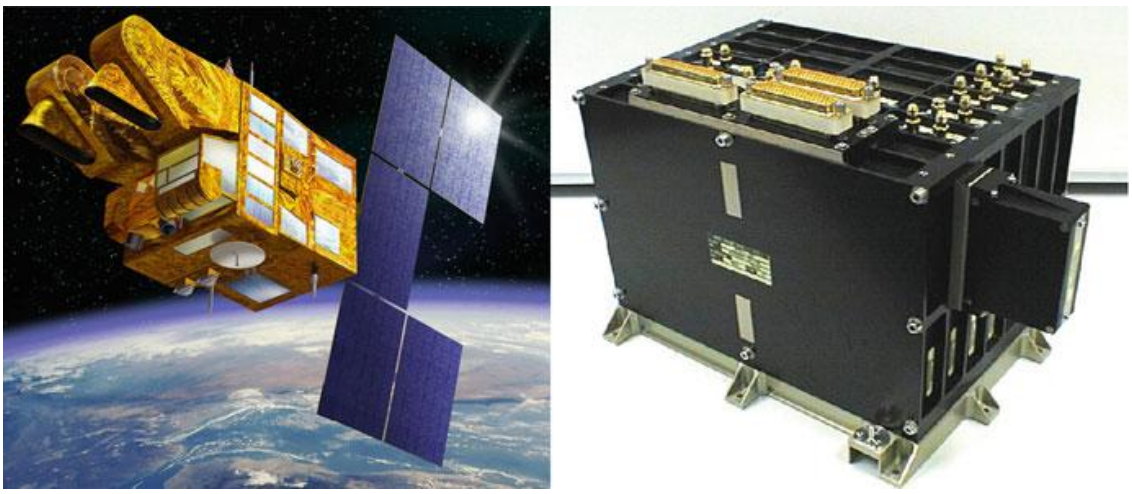

Figura 1.1. Satélite SPOT5 (izquierda) y su unidad de compresión (derecha)

Figura 1.2. Satélite PLEIADES-HR (izquierda) y su unidad de compresión (derecha)

Existen diferentes tipos de técnicas compresión disponibles. En los algoritmos de compresión sin pérdidas (*loseless compression*), la imagen reconstruida es idéntica a la original [2] [3]. En compresión casi-sin pérdidas (*near-lossless compression*), la diferencia máxima absoluta entre la imagen reconstruida y la imagen original no exceden un valor máximo predefinido [4]. Finalmente, en compresión con pérdidas (*lossy compression*), la imagen reconstruida es similar a la imagen original, típicamente se emplea el error cuadrático medio dada una tasa de bits objeto.

Tradicionalmente, la compresión de imágenes hiperespectrales, ha sido realizada mediante algoritmos de compresión sin pérdidas. Estos algoritmos posibilitan que, tras la compresión, los datos sean reconstruidos sin pérdida alguna de información, es decir, la imagen reconstruida es idéntica a la original [2] [3]. La compresión sin pérdidas resulta la deseada ya que se preserva la integridad de toda la información contenida en la imagen. Sin embargo, los ratios de compresión que se obtienen mediante la aplicación de estos algoritmos son limitados. Los mejores algoritmos de compresión sin pérdidas proporcionan ratios de compresión típicamente del orden de 2:1 a 3:1 [5] [6] [7]. Tómese como ejemplo el algoritmo desarrollado en el estándar CCSDS para la compresión sin pérdidas de imágenes hiperespectrales [8].

Sin embargo, la compresión con pérdidas (*lossy compression*) de imágenes hiperespectrales se ha convertido en objeto de estudio dada la necesidad actual de mayores ratios de compresión. Varios algoritmos encaminados a la compresión con pérdidas de imágenes hiperespectrales han sido desarrollados en la actualidad. Incluso, estándares existentes de compresión en dos dimensiones (2D), tales como JPEG2000, han sido adaptados a la naturaleza tridimensional de las imágenes hiperespectrales [9]. Asimismo, el impacto de la

compresión con pérdidas sobre el procesado los datos ha sido evaluado para varios algoritmos de compresión mostrando que, la precisión en los resultados puede ser preservada alcanzado un ratio de compresión superior a los algoritmos de compresión sin pérdidas [1].

Los algoritmos de compresión suelen ser implementados en hardware el cual debe operar en un sistema embarcado, ya sea en una aeronave o en un satélite. A pesar de que la eficiencia en la codificación se presenta como un aspecto clave para la compresión que se realiza en sistemas embarcados, existen otra serie de importantes requisitos a tener en cuenta. La potencia computacional disponible en un satélite, por ejemplo, es limitada, mucho menor que la disponible en un procesador de gama media del sector comercial. Por lo tanto, se requiere de un algoritmo de baja complejidad. Normalmente, el algoritmo de compresión desplegado se implementa sobre una FPGA (*Field Programmable Gate Array*), un ASIC (*Application-Specific Integrated Circuit*) o un DSP (*Digital Signal Processor*). Muchos de los algoritmos de compresión existentes en la actualidad para imágenes en 2D alcanzan un buen ratio de compresión. Sin embargo, suelen ser demasiado complejos para ser implementados en los dispositivos hardware disponible en un satélite. Por este motivo, nuevos algoritmos de compresión de imágenes hiperespectrales han sido desarrollados completamente desde cero. Un ejemplo de esta aproximación, es el algoritmo desarrollado por Magli et. al. [10] denominado, *Lossy Compression for Exomars (LCE) Algorithm*.

1.1. Objetivos

El objetivo del presente Trabajo Fin de Máster (TFM) es la implementación, verificación y síntesis sobre una FPGA del algoritmo con pérdidas LCE. El resultado de este trabajo es por lo tanto un código RTL sintetizable en el lenguaje de descripción hardware VHDL (*VHSIC Hardware Description Language*). Para llevar a cabo este trabajo es necesario el uso de varias herramientas software.

Como entradas a este trabajo, ha sido necesario el estudio, en primer lugar, del estado del arte sobre algoritmos de compresión de imágenes hiperespectrales, en especial el algoritmo LCE. Asimismo, para esta implementación se ha partido del código de referencia proporcionado por la Universidad Politécnica de Torino escrito en el lenguaje de alto nivel C/C++. Este código de referencia ha sido previamente modificado con el fin de lograr una mayor aproximación al hardware empleando tipos de datos con resolución a nivel de bit (*Algoritmic C data types*), reduciendo el número de variables por constantes, evitando las operaciones matemáticas en punto flotante, eliminando accesos a memoria innecesarios y

optimizando bucles. Para mayor información de este trabajo previo sobre el código de referencia consultar [11].

Mediante las entradas descritas, se ha procedido a realizar una separación del código en ANSI C de las funciones que lo componen con el objetivo de obtener mediante la herramienta de síntesis de alto nivel, *High Level Synthesis* (HLS), Catapult C obtener el código RTL sintetizable escrito en VHDL de cada uno de los módulos [12].

En este punto se emplea la herramienta de diseño, simulación y depuración ModelSim con el fin de diseñar y verificar la unidad de control hardware que será la encargada de la conexión y gestión de los flujos de datos de cada uno de los módulos generados.

Finalmente, mediante el empleo de la herramienta Precision Synthesis, se realiza la síntesis de todo el algoritmo de compresión obtenido extrayendo los resultados de la misma. Estos resultados se comparan con los obtenidos en trabajos anteriores [11].

1.2. Estructura de la memoria

Esta memoria de trabajo fin de máster se estructura en este primer capítulo de introducción y objetivos en el que se han descrito una serie de conceptos en ideas básicos que sirven para central al lector en el objeto de este trabajo.

En el segundo capítulo se aborda el estado del arte definiendo, en primer lugar, el concepto de imagen hiperespectral y dentro de este amplio campo de estudio, abordando los algoritmos de compresión.

La primera etapa de trabajo realizado, es decir, el trabajo sobre el código de referencia para su separación en funciones y módulos independientes se describe en el tercer capítulo. En el cuarto capítulo se presenta el proceso de conversión desde ANSI C hacia VHDL de estos módulos independientes, haciendo especial hincapié en las decisiones tomadas en el uso de la herramienta Catapult C. El diseño y verificación de la unidad de control hardware se describe en el quinto capítulo.

En el sexto capítulo, se presentan los resultados obtenidos en este trabajo, su comparación con trabajos anteriores y posibles optimizaciones. Finalmente las conclusiones y bibliografía empleada en el trabajo se presentan en las secciones séptima y octava.

2. Estado del arte

El Estado del Arte en el presente Trabajo Fin de Máster resume y organiza los resultados recientes de investigación sobre lo concerniente a los algoritmos de compresión embarcados de imágenes hiperespectrales. Así, se presentará una mejor perspectiva de la materia y se evaluarán las principales tendencias en éste área. Para ello, se han analizado algoritmos más relevantes publicados en la literatura concerniente a la compresión de imágenes hiperespectrales como área específica.

No se entrará en excesivo detalle en sus respectivas descripciones salvo, que el algoritmo aporte algún concepto novedoso, pues la mayoría de algoritmos estudiados se basan en los mismos principios de funcionamiento.

Además, en este segundo capítulo, se realiza una presentación del concepto de imagen hiperespectral prestando especial atención en la necesidad de la existencia de algoritmos de compresión, especialmente en sistemas embarcados.

Como se describe en la sección anterior, los sistemas embarcados, ya sea en dispositivos tales como aeronaves o satélites, presentan una serie de requisitos exigentes en cuanto a consumo de potencia, cantidad de almacenamiento disponible o ancho de banda de transmisión hacia el segmento terreno. Es por ello que resulta crucial el empleo de algoritmos de compresión para minimizar el impacto que la adquisición de una imagen hiperespectral cause en el resto del sistema embarcado.

2.1. Concepto de imagen hiperespectral

Una imagen hiperespectral es una imagen tomada a diferentes longitudes de onda en la que cada píxel representa la reflectancia de los materiales contenidos en ella. Por consiguiente, si se toman distintas imágenes a diferentes longitudes de onda, cada píxel se encontrará representado no por un único valor, como ocurre en una imagen en blanco y negro, sino por un vector de valores espectrales con la contribución de la luz detectada en ese punto para cada banda en el espectro. De esta forma, una imagen hiperespectral estará formada por un número finito de bandas que puede ir desde las decenas hasta los varios centenares de bandas dependiendo del sensor hiperespectral empleado.

A la hora de representar este tipo de imágenes se emplea, comúnmente, un cubo, denominado hipercubo o cubo hiperespectral, cuyas dos primeras dimensiones representarían la ubicación en el espacio de un píxel determinado de la imagen y una tercera dimensión que representaría la longitud de onda a la que ha sido tomada la

imagen. La Figura 2.1, muestra un ejemplo de representación del cubo hiperespectral para una imagen con 4 bandas.

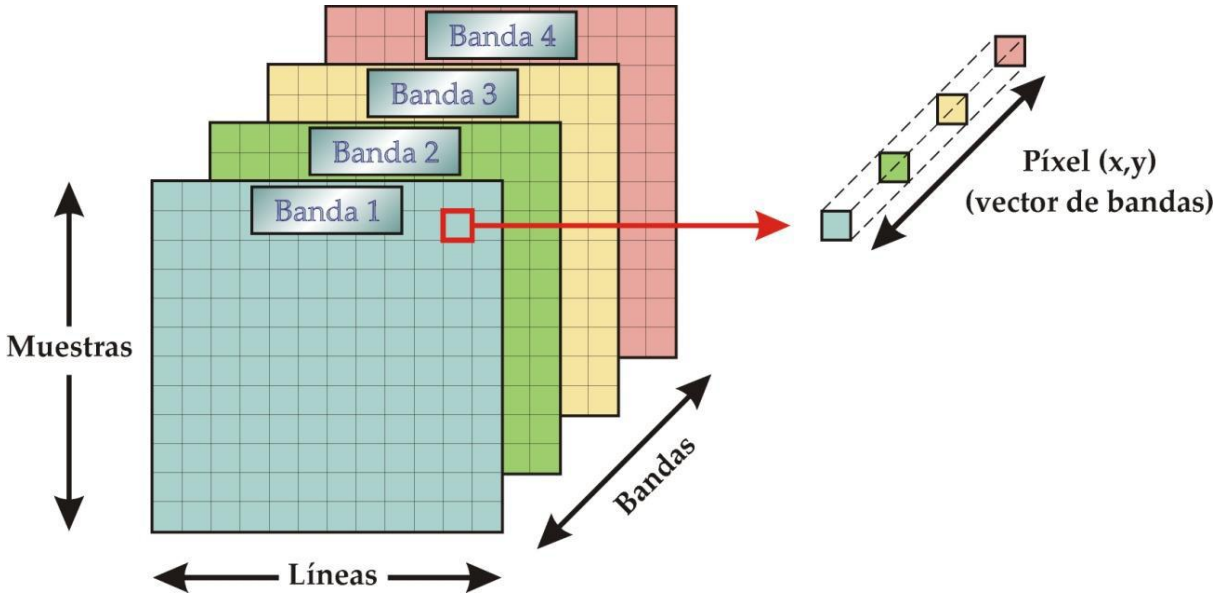

Figura 2.1. Imagen hiperespectral (cubo hiperespectral)

Por su naturaleza, un fenómeno característico de este tipo de imágenes es la presencia de varios tipos de píxeles según su composición en una misma banda a nivel sub-píxel. Existen píxeles llamados píxeles puros en los que únicamente existe la presencia de un único material y píxeles mezcla que contienen distintos materiales [13]. Este fenómeno es producido por la insuficiente resolución espacial de los sensores hiperespectrales para separar materiales espectralmente puros.

Generalmente, la mayoría de los píxeles presentes en una imagen son píxeles mezcla, ya que, independientemente de la escala que se considere, la mezcla se produce a nivel microscópico [14] [15]. En la Figura 2.2, se presenta una imagen hiperespectral en la que se puede observar píxeles puros y mezclas extraídos.

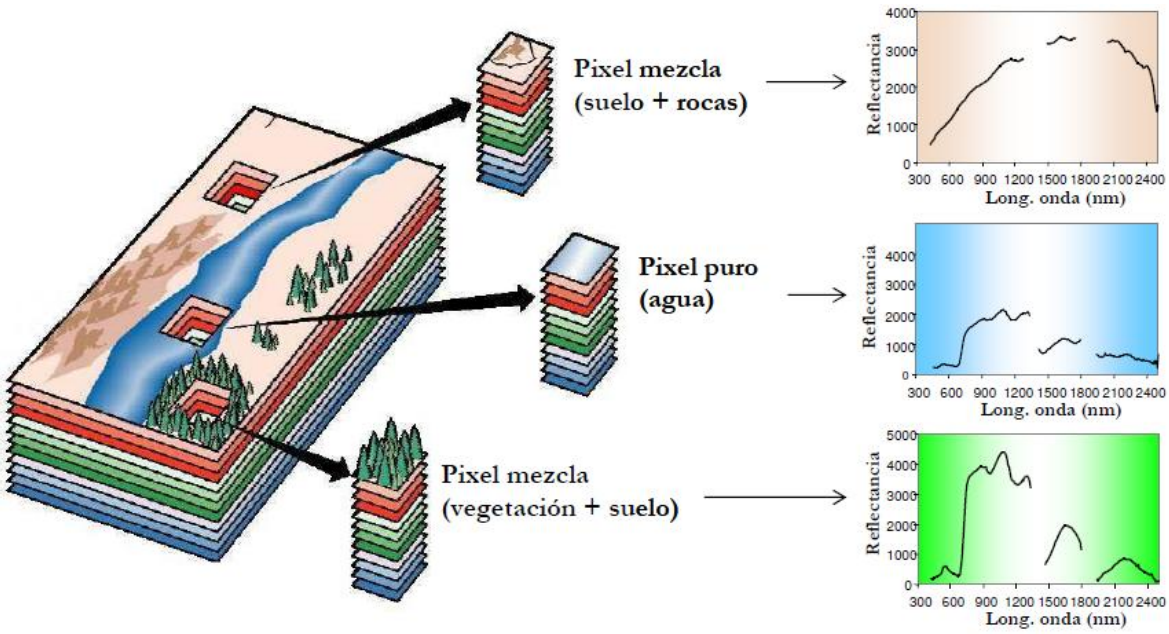

Figura 2.2. Imagen hiperespectral – Tipos de píxeles

2.2. Algoritmos de compresión de imágenes hiperespectrales

Las imágenes hiperespectrales son generalmente capturadas y almacenadas a bordo en un satélite o en una aeronave para, posteriormente, ser transmitidas hacia la estación terrena. Los instrumentos de alta resolución, presentes y futuros, embarcados en misiones espaciales para la adquisición de este tipo de imágenes, contribuyen a hacer necesario que las unidades de procesamiento de datos a bordo sean capaces de gestionar la inmensa cantidad de datos generados. Las limitaciones existentes a bordo en cuanto al ancho de banda disponible para el envío de las imágenes hiperespectrales y la capacidad para el almacenamiento de las mismas concluyen que, en líneas generales, se requiera de la aplicación de algoritmos de compresión de imágenes con el fin de reducir la cantidad de datos almacenados y transmitidos hacia el segmento terreno.

Tal y como se define en el capítulo 1, en cuanto a la clasificación de los algoritmos de compresión de imágenes hiperespectrales, típicamente, se establecen dos categorías, los algoritmos de compresión sin pérdidas y los algoritmos de compresión con pérdidas.

2.2.1. Algoritmos de compresión sin pérdidas (*lossless compression*)

La compresión de imágenes hiperespectrales sin pérdidas ha estado basada generalmente en el paradigma de la codificación predictiva, en donde cada pixel es predicho mediante datos anteriormente procesados y en donde el error de predicción se codifica entrópicamente [16] [17]. En [18] se introduce el concepto de predicción difusa, en donde se sustituye el predictor por un set predefinido empleando reglas de lógica difusa. En [19] se mejora la predicción mediante el uso del análisis de bordes. En [20] la predicción mediante el empleo de clasificadores proporciona un algoritmo de compresión casi sin pérdidas. La compresión mediante el empleo de clasificadores se desarrolla en mayor profundidad en [21] para compresiones tanto sin pérdidas como casi sin pérdidas. En [22] se emplea el concepto de cuantificación vectorial mientras que en [23] introduce clustering diferencial como base para la codificación proporcionando el concepto C-DPMC (*Clustered Differential Pulse Code Modulation*). En este algoritmo, el espectro de la imagen es agrupado (*clustered*), el predictor es formado para cada clúster y usado para decorrelar el espectro de la imagen. En [24] se introduce el concepto de emplear imágenes previamente procesadas, en el caso de este algoritmo dos, para formar el predictor espectral.

Una vez presentados los algoritmos más representativos de los estudiados en el Estado del Arte para la compresión de imágenes hiperespectrales sin pérdidas se desarrolla, en la siguiente sección, el estándar recomendado por el CCSDS (*Consultative Committee for Space Data Systems*). Este novedoso estándar, publicado en Mayo de 2012, representa el consenso técnico de las agencias participantes en el CCSDS. El CCSDS se postula como la agencia de referencia para organismos tales como la ESA (*European Space Agency*) a la hora de adoptar sus estándares en sus diseños y requerimientos.

2.2.1.1. CCSDS 123.0-B-1 Lossless Multispectral & Hyperspectral Image Compression [8]

El propósito de este estándar es establecer un algoritmo de compresión de datos aplicado al hipercubo tridimensional que forman las imágenes multiespectrales e hiperespectrales adquiridas por los instrumentos situados en la carga útil. Asimismo, especifica el formato de los datos comprimidos [8].

El compresor propuesto lo constituyen dos bloques funcionales representados en la Figura 2.3. Como se puede observar, han sido denominados como predictor y codificador.

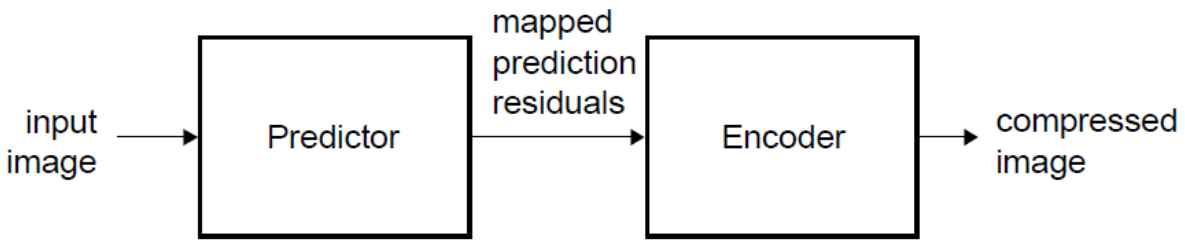

Figura 2.3. Esquema del compresor

Predictor

El predictor es el módulo encargado del cómputo del set de valores de muestras predichas, $\hat{s}_{z,x,y}$, y el mapeo de los residuos predichos, $\delta_{z,x,y}$, a partir de las muestras de la imagen de entrada, $s_{z,x,y}$. La etapa de predicción puede realizarse en un único pase a través de la imagen. La predicción de la muestra $s_{z,x,y}$, es decir, el cómputo de $\hat{s}_{z,x,y}$ y $\delta_{z,x,y}$, depende, en general, de los valores de las muestras vecinas en la banda actual y en las P bandas anteriores, donde $0 \leq P \leq 15$ es un valor a especificar por parte del usuario. La Figura 2.4, ilustra la disposición típica de la vecindad de las muestras usadas para la predicción. Nótese que para $y = 0$, $x = 0$, $x = N_x - 1$ o $z < P$ la vecindad es consecuentemente truncada.

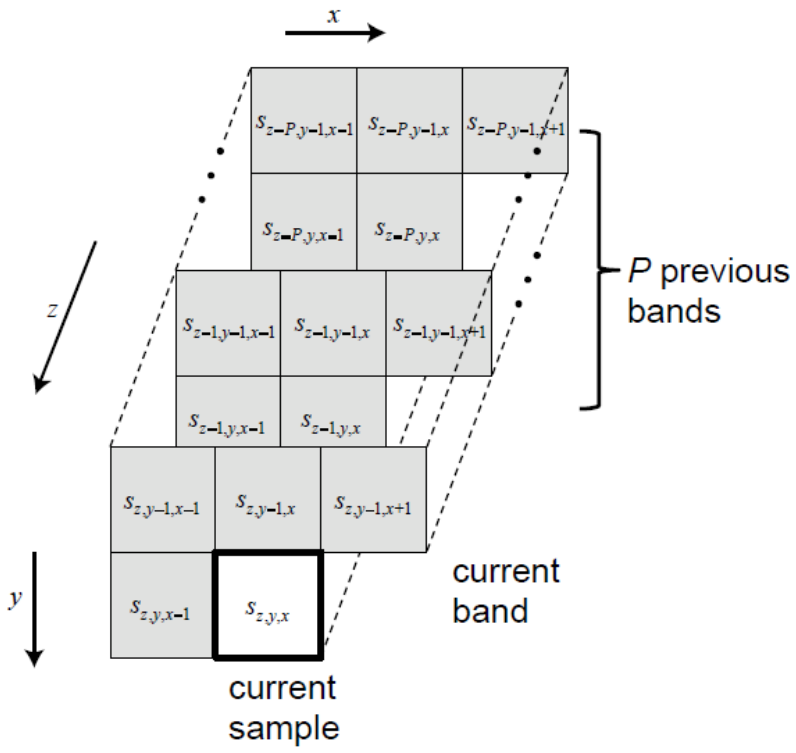

Figura 2.4. Vecindad, disposición típica

Dentro de cada banda espectral, el predictor computa una suma local, $\sigma_{z,x,y}$, de las muestras de valores vecinos. Cada suma local es empleada para calcular la diferencia local. El valor de la muestra predicha se calcula empleando la suma local de la banda espectral actual y una suma ponderada de la diferencia local de la banda actual y las P bandas espectrales anteriores. Los pesos empleados son adaptativamente actualizados. Cada residuo predicho, es decir, la diferencia entre una muestra $s_{z,x,y}$ dada y su correspondiente valor predicho $\hat{s}_{z,x,y}$, se mapea en un entero sin signo denominado $\delta_{z,x,y}$.

Existen varias técnicas, elegibles por parte de usuario, a la hora de seleccionar las muestras vecinas para el cómputo de la suma local. La Figura 2.5 muestra las predicciones denominadas *neighbor-oriented* y *column-oriented*.

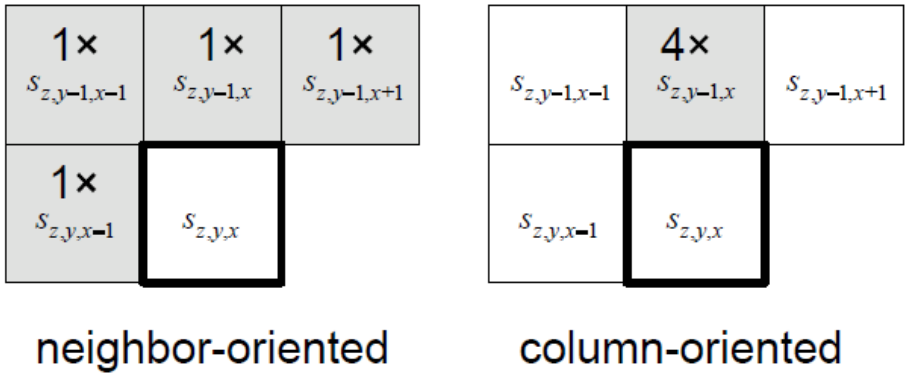

Figura 2.5. Muestras usadas para el cálculo de las sumas locales

Se definen las siguientes ecuaciones para el cómputo de ambas técnicas para el cálculo de la suma local:

- *Neighbor-oriented*

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}, & y > 0, 0 < x < N_x - 1 \\ 4s_{z,y,x-1}, & y = 0, x > 0 \\ 2\left(s_{z,y-1,x} + s_{z,y-1,x+1}\right), & y > 0, x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x}, & y > 0, x = N_x - 1 \end{cases}$$

- *Column-oriented*

$$\sigma_{z,y,x} = \begin{cases} 4s_{z,y-1,x}, & y > 0 \\ 4s_{z,y,x-1}, & y = 0, x > 0 \end{cases}$$

En cuanto al cómputo de la diferencia local, para cada banda espectral, el usuario puede escoger entre el modo reducido y el modo completo. La diferencia local central, $d_{z,x,y}$, es igual a la diferencia entre la suma local y cuatro veces el valor de la muestra $s_{z,x,y}$, en el

modo reducido. Bajo el modo completo, las tres diferencias locales direccionales, $d^N_{z,x,y}$, $d^W_{z,x,y}$, $d^{NW}_{z,x,y}$, son iguales a la diferencia entre $\sigma_{z,x,y}$ y cuatro veces el valor de las muestras denominadas N, W, NW en la Figura 2.6.

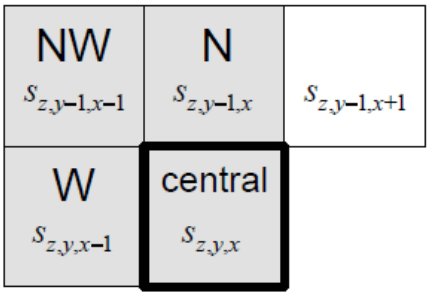

Figura 2.6. Cómputo de las diferencias locales en cada banda espectral

Las siguientes ecuaciones definen el cálculo de la diferencia local en cada caso:

- Diferencia local central

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x}$$

- Diferencias locales direccionales

$$d^N_{z,y,z} = \begin{cases} 4s_{z,y-1,x} - \sigma_{z,y,x}, & y > 0 \\ 0, & y = 0 \end{cases}$$

$$d^W_{z,y,z} = \begin{cases} 4s_{z,y,x-1} - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases}$$

$$d^{NW}_{z,y,z} = \begin{cases} 4s_{z,y-1,x-1} - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4s_{z,y-1,x} - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases}$$

Codificador

Este módulo es el encargado de componer el bit-*stream* de la salida. Éste se encuentra compuesto por una cabecera seguida por el cuerpo del mismo, tal y como se muestra en la Figura 2.7. La cabecera se encuentra compuesta por el metadato de la imagen, metadato del predictor y metadatos del codificador. El cuerpo consiste en la codificación sin pérdidas de los residuos predichos mapeados, $\delta_{z,x,y}$, del predictor.

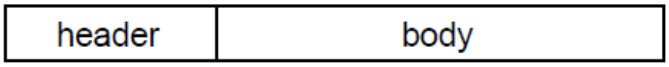

Figura 2.7. Estructura de la imagen comprimida

El usuario puede escoger, para la codificación de los residuos mapeados, el uso de la codificación entrópica *sample-adaptative* o la aproximación *block-adaptative*. La aproximación *sample-adaptative* consigue tamaños de imágenes comprimidas menores que la aproximación *block-adaptative* [25].

Bajo la aproximación de codificación entrópica *sample-adaptative*, cada residuo mapeado predicho se codifica mediante el uso de una palabra de código binaria de longitud variable. Los códigos de longitud variable son seleccionados adaptativamente en base a estadísticas que son actualizadas tras la codificación de cada muestra. Se mantienen estadísticas separadas para cada banda.

Por otra parte, en la aproximación de codificación entrópica *block-adaptative*, la secuencia de residuos mapeados predichos se dividen en bloques de pequeño tamaño, y el método de codificación usado se selecciona independientemente y adaptativamente en cada bloque. Dependiendo del orden de codificación, los residuos mapeados predichos en un bloque pueden ser seleccionados de la misma banda espectral o de una banda diferente. Por consiguiente, si se escoge este método, el tamaño de la imagen comprimida depende del orden de codificación.

2.2.2. Algoritmos de compresión con pérdidas (*lossy compression*)

Los algoritmos de compresión con pérdidas alcanzan mayores ratios de compresión que aquellos sin pérdidas a expensas de una pérdida de información durante el proceso. A pesar de la pérdida de calidad en la imagen reconstruida, este tipo de algoritmos son de gran utilidad especialmente cuando se requieren grandes ratios de compresión. Asimismo, en [1] se evalúa el efecto de dichas pérdidas en imágenes hiperespectrales para aplicaciones específicas tales como, la detección o clasificación de objetivos, mostrando que se pueden alcanzar grandes tasas de compresión con impactos menores con respecto a la funcionalidad objeto.

La importancia del estudio de la compresión con pérdidas queda asimismo reforzada por el hecho de que el CCSDS ha desarrollado y aprobado estándares de relevancia internacional en esta material, en donde se incluye el estándar descrito en la sección anterior para la compresión sin pérdidas de imágenes hiperespectrales y encontrándose actualmente en

fase de definición de un estándar para la compresión de imágenes hiperespectrales con pérdidas.

Existen, en la literatura, numerosos algoritmos desarrollados para la compresión con pérdidas de imágenes hiperespectrales, muchos de ellos resultan generalizaciones de algoritmos existentes para la compresión de imágenes en 2D o algoritmos para la compresión de vídeo. En [26] se propone un algoritmo basado en el estándar JPEG2000 con un decorrelador espectral. En [1] [9] se introduce la transformada de Karhumen-Loève para la compresión del cubo hiperespectral. La transformada mediante wavelets y la descomposición de Tucker se aplican en [27]. Una visión diferente se propone en [28], donde se aplican las técnicas empleadas en el estándar de compresión de video H.264/AVC para la compresión del hipercubo espectral consiguiéndose resultados significativos en términos de tasa de bit – distorsión introducida. Estos algoritmos de compresión presentan unas necesidades en cuanto a memoria de almacenamiento y capacidad de cómputo disponible a bordo resultando demasiado complejos para una misión de larga duración.

Recientemente, se ha formado un nuevo paradigma basado en algoritmos de baja complejidad [10]. Esta técnica se basa en una primera etapa de predicción, una cuantificación, optimización tasa de bits-distorsión y codificación entrópica. Aprovecha la simplicidad y las altas prestaciones de los algoritmos de compresión basados en la predicción empleando pocos accesos a memoria. Asimismo, el empleo de la cuantificación y la optimización tasa de bits-distorsión asegura una tasa de compresión elevada. Así, en la siguiente sección se describe este algoritmo de compresión con pérdidas seleccionado para su implementación en este Trabajo de Fin de Máster.

2.2.2.1. Lossy Compression for Exomars (LCE) Algorithm

El algoritmo de compresión de imágenes hiperespectrales LCE [10], ha sido diseñado con el fin de poder ser desplegado a bordo de un satélite alcanzando altos ratios de compresión. Los objetivos principales en su desarrollo han sido, baja complejidad, resistencia a errores y fácilmente implementable a nivel hardware. A grandes rasgos, el algoritmo consiste en un predictor seguido por un codificador entrópico Golomb potencia de 2.

Con el fin de proporcionar al algoritmo de robustez frente a errores, el algoritmo codifica de manera independiente bloques de imagen de 16 x 16 píxeles. De esta forma, cada bloque es individualmente comprimido con lo que un error en un bloque dado no se propaga al resto de bloques. Asimismo, ha sido diseñado para que su implementación hardware sea fácilmente paralelizable permitiendo mayores prestaciones a sensores con altas tasas de datos.

Predictor

El algoritmo comprime bloques espaciales independientes no solapados de tamaño 16 x 16 píxeles para todas las bandas del cubo hiperespectral. Sea $x_{m,n,i}$ el píxel perteneciente a una imagen hiperespectral en la fila m, columna n y banda i. La Figura 2.8, muestra la ubicación del píxel a codificar y de la vecindad que lo rodea.

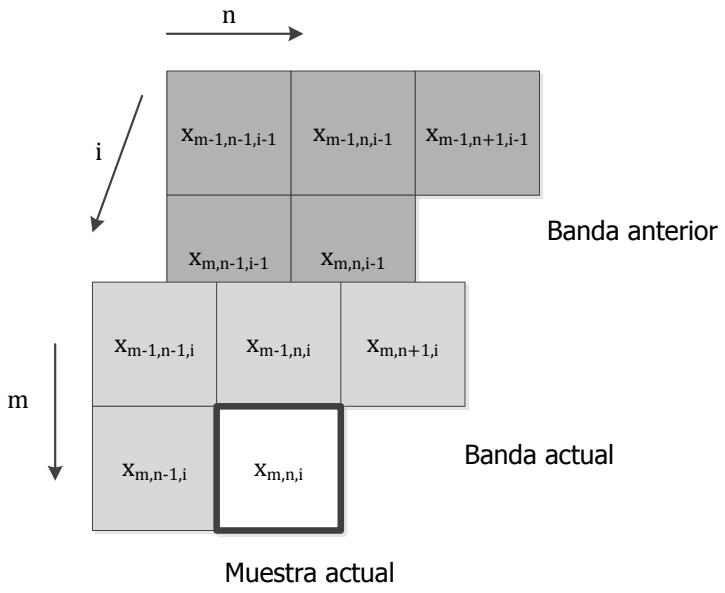

Figura 2.8. Localización de la muestra actual y su vecindad

En la primera banda ($i = 0$), se aplica una compresión denominada 2D (modo INTRA) en la que no se emplea información de ninguna otra banda. Las muestras son procesadas de izquierda a derecha y de arriba hacia abajo, *raster scan order*. En general, el predictor empleado para esta banda es el presentado en la siguiente expresión.

$$\tilde{x}_{m,n,0} = \left(\hat{x}_{m-1,n,0} + \hat{x}_{m.n-1,0}\right) \gg 1$$

Donde $\tilde{x}$ representa al predictor, $\hat{x}$ el valor del píxel reconstruido y >> representa el desplazamiento de un bit hacia la derecha. Por lo tanto, el predictor para la primera banda se basa en la media de los píxeles vecinos (encima e izquierda) del pixel a predecir. Nótese que el primer píxel no es predicho, los píxeles en la primera fila, es decir, $m = 0$, son predichos mediante la siguiente expresión $\tilde{x}_{0,n,0} = \hat{x}_{0,n-1,0}$ y los píxeles en la primera columna, $n = 0$, se predicen como $\tilde{x}_{m,0,0} = \hat{x}_{m-1,0,0}$.

Para el resto de bandas, $i \neq 0$, las muestras $x_{m,n,i}$ se predicen mediante las muestras decodificadas, $\hat{x}_{m,n,i-1}$, es decir, mediante las muestras del mismo bloque en la banda

anterior. Se computa sobre el bloque un estimador LMS (*Least Mean Square*) definido como $\propto = \frac{\propto_N}{\propto_D}$, siendo $\propto_N$ y $\propto_D$:

$$\propto_N = \sum_{m,n} (\hat{x}_{m,n,i-1} - m_{i-1})(x_{m,n,i} - m_i)$$

$$\propto_D = \sum_{m,n} (\hat{x}_{m,n,i-1} - m_{i-1})(x_{m,n,i-1} - m_{i-1})$$

Donde, m_i y m_{i-1} son las medias, dentro del conjunto de números enteros, computadas de los bloques anteriores decodificados en las bandas i e $i-1$.

$$m_{i/i-1} = round\left(\frac{1}{P}\sum_{m,n} x_{m,n,i/i-1}\right)$$

En la ecuación anterior P se define como el número de muestras computadas en el bloque actual.

Las versiones cuantificadas de $\propto$ y m_i, denominadas $\widehat{\propto}$ y $\widehat{m}_i$, se generan mediante el empleo de un cuantificador escalar. Finalmente, los valores de las muestras predichas se calculan para todas las muestras pertenecientes a un bloque como:

$$\tilde{x}_{m,n,i} = \widehat{m}_i + \widehat{\propto}\left(\hat{x}_{m,n,i-1} - m_{i-1}\right)$$

Por consiguiente, el error de predicción se calcula mediante la siguiente expresión:

$$e_{m,n,i} = x_{m,n,i} - \tilde{x}_{m,n,i}$$

Optimización tasa de bits-distorsión

Antes de proceder a la cuantificación del error de las muestras predichas, se verifica si el valor de la predicción se aproxima al valor real del pixel, permitiendo de esta forma evitar la codificación de las muestras de error. Si este es el caso, se fija al valor '1' lógico un flag que indica que el bloque actual de muestras de errores de predicción es cero. Esta condición se denomina, *zero_block condition*. Para la toma de esta decisión, se computa la energía del error de predicción mediante la siguiente expresión:

$$D_0 = \frac{1}{256}\sum_{m,n} e_{m,n,i}^2$$

Si D_0 es menor que el umbral definido D_T, es decir, $D_0 < D_T$, se aplica la condición *zero_block*.

Cuantificación y mapeo

Las muestras de los errores de predicción son cuantificadas en valores enteros, $eq_{m,n,i}$, para, posteriormente, ser decuantificadas a valores reconstruidos, $er_{m,n,i}$. Dentro de la primera banda, $y = 0$, este proceso se realiza pixel a pixel empleando un cuantificador escalar uniforme. Para el resto de bandas, $y \neq 0$, es posible escoger entre el cuantificador escalar uniforme y el cuantificador de umbral uniforme, UTQ (*Uniform-threshold Quantizer*) presentado en [29]. Finalmente, los valores reconstruidos son mapeados a números enteros no negativos empleando la siguiente aproximación.

$$Sm,n,i = \begin{cases} 2|eq_{m,n,i}| - 1, si\ eq_{m,n,i} > 0 \\ 2|eq_{m,n,i}| \quad\quad , si\ eq_{m,n,i} \leq 0 \end{cases}$$

Cuantificación y mapeo

El bloque de 16 x 16 residuos se codifican, en *raster scan order*, empleando la codificación Golomb potencia de 2, exceptuando la primera muestra de cada bloque. Éste, se codifica empleando un código Golomb exponencial potencia de 0. La primera muestra cuantificada de la primera banda no se codifica, por lo tanto, se almacena en la imagen comprimida usando 16 bits.

Formato del fichero de salida

El bit-*stream* comprimido a la salida se encuentra compuesto por la concatenación de bloques codificados. Éstos han de ser leídos espacialmente en *raster scan order*. Cada bloque es codificado para todas las bandas. En cuanto a la información de salida escrita en el archivo comprimido para todos los bloques se tiene: a) para todas las bandas salvo la primera, los parámetros $\propto$ y m; b) un bit correspondiente al indicador *zero_block condition*; c) para aquellos bloques en los que no se de la condición zero_block, las muestras de error de predicción cuantificadas de cada bloque escritas en *raster scan order*.

```
block_0 [BAND0 → encoded pred error;
         BAND1→  ∝ + μ + zero_block flag + encoded pred error]
block_1 … block_N
```

Figura 2.9. Formato del fichero comprimido generado por el algoritmo LCE

3. Desarrollo de la arquitectura implementada

Este tercer capítulo de la memoria se centra en la descripción del trabajo realizado en el presente Trabajo Fin de Máster. Se aborda en primer lugar el diseño a nivel de software del algoritmo *Lossy Compression for Exomars* (LCE) describiendo los módulos que lo componen, flujos de datos y consideraciones a la hora de separar en módulos independientes dicho código de referencia. Posteriormente, se presentan los trabajos realizados en la herramienta de síntesis Catapult C para la obtención del código VHDL a nivel RTL sintetizable. Finalmente, se presenta el módulo de control hardware diseñado en VHDL, implementado y verificado que permite la conexión de los módulos independientes del algoritmo.

3.1. Implementación del algoritmo LCE en el software de referencia

El software proporcionado [30] implementa el algoritmo en lenguaje ANSI C operando sobre cada bloque independiente de 16 x 16 píxeles de la imagen hiperespectral de entrada de forma secuencial, es decir, bloque a bloque. Cada bloque se procesa conjuntamente con todas sus bandas espectrales, antes de que el siguiente bloque sea procesado.

La imagen hiperespectral de entrada es almacenada en un buffer de memoria denominado `data` de tipo entero. Durante el procesamiento de cada bloque, sus muestras son almacenadas en un buffer de menor tamaño, 256 posiciones de 16 bits, denominado `curr_block`. El bloque usado como referencia a la hora de realizar la predicción se copia de igual forma en el buffer denominado `ref_block` (256 x 16 bits) Las palabras código de salida del algoritmo se almacenan en el buffer de salida denominado `block_out` (256 x 32 bits).

Asimismo, se incluyen tres variables que proporcionan información relevante a la hora realizar las escrituras en el buffer de salida. La variable `pp` almacena temporalmente los datos codificados y comprimidos. Cada vez que una nueva palabra de código es generada se almacena, temporalmente, en esta variable hasta que sus 32 bits sean escritos. Una vez se llene este buffer por la escritura de sus 32 bits, la palabra resultante se escribe en el buffer de salida `block_out`. El número actual de bits escritos en la variable `pp` es controlado por la variable denominada `m`. Finalmente, la variable `filecount` indica el número de palabras de 32 bits escritas en el buffer de salida `block_out`. Durante la codificación de cada banda este valor es actualizado, fijándose nuevamente a su valor inicial de '0' lógico al inicio de cada banda.

El estudio del software de referencia, ha permitido la extracción de los estímulos y así como la verificación de resultados en la etapa de verificación de la implementación del algoritmo. Los estímulos serán las entradas al banco de pruebas (*testbench*) diseñado y su salida será posteriormente comparada con los resultados proporcionados por el código de referencia.

En este punto cabe destacar que, el código de entrada en este Trabajo Fin de Máster contiene las siguientes modificaciones implementadas en [11] que facilitan su implementación a nivel hardware. Nótese que en [11] se evalúa el impacto de estas modificaciones sobre el código original no hallándose diferencias significativas entre el código inicial de referencia y el código a la salida de [11].

Como paso inicial en [11] se aisló en el código la parte del algoritmo que se implementaría a nivel hardware de lo que formaría parte del banco de pruebas. En este punto, se introdujeron tipos de datos con precisión a nivel de bit. Para ello, se emplearon datos de tipo Algorithmic C. De esta forma, la función denominada `pred1block()` contiene los módulos objeto de la implementación. La Figura 3.1, muestra en pseudo-código la estructura de la función `pred1block()`.

```
pred1block ()
{
          if (i == 0)
              2D-prediction (INTRA-mode)
              entropy coding
          else{
                spectral prediction
                rd-optimization
                quantization
                entropy coding
           }
}
```

Figura 3.1. Pseudo-código de la función `pred1block()`

Por otra parte, se redujo el número de operaciones matemáticas realizadas mediante el empleo de operadores lógicos tales como, el desplazador, funciones and/or, cuando resultó posible. Además la división entera propuesta para el cálculo del parámetro de ganancia del predictor $\propto$, fue sustituida por una búsqueda dicotómica. En la misma línea, se eliminaron y minimizaron las operaciones de lectura/escritura en los buffers de entrada/salida.

En cuanto a la llamada a funciones, se introdujeron en el código aquellas necesarias para la implementación hardware del algoritmo. Finalmente, se optimizaron con vistas a su implementación hardware los bucles empleados en el algoritmo.

3.2. Separación del código de referencia en módulos funcionalmente independientes

Como primer paso a la hora de abordar la implementación del algoritmo LCE, a partir del código de referencia proporcionado, se ha decidido dividir el mismo en módulos funcionalmente independientes. Para ello, se han identificado en el código de referencia aquellas funciones o sentencias de código asociadas a cada a cada una de las etapas del algoritmo LCE. Como salidas de esta tarea dispondremos de dos ficheros para cada módulo independiente, un fichero con extensión *.cpp* que contiene el cuerpo de cada función y un fichero con extensión *.h* que contiene las cabeceras de las funciones empleadas.

El objetivo de esta primera etapa es el de obtener módulos funcionalmente independientes que sirvan de entrada a la herramienta Catapult C. La idea principal que subyace bajo esta aproximación es la de conocer dentro de cada módulo su ruta crítica para así poder conocer los módulos con una mayor carga computacional asociada.

Los ficheros de salida correspondientes a módulos funcionalmente independientes son los que se citan a continuación:

- adgolomb.cpp y adgolomb.h
- calc_perr.cpp y calc_perr.h
- estimationls.cpp y estimationls.h
- init_output.cpp y init_output.h
- mean.cpp y mean.h
- pred_2D.cpp y pred_2D.h
- write_alphamu.cpp y write_alphamu.h
- zero_block.cpp y zero_block.h

Los siguientes apartados presentan cada uno de los módulos ANSI C independientes obtenidos describiendo su funcionalidad e interfaces de entrada/salida.

Dentro del desarrollo de esta tarea, se analizan cada una de las entradas/salidas resultantes de los módulos independientes con el fin de servir como entradas a las siguientes etapas del trabajo realizado. Resulta fundamental este análisis ya que las entradas o salidas que se definan se convertirán posteriormente en puertos de entrada, de salida o bidireccionales en el lenguaje de descripción hardware VHDL.

A nivel de código, la función `pred1block()` presenta, una vez finalizada esta tarea, la implementación mostrada en la Figura 3.2. Como puede observarse, sólo existen llamadas a funciones o decisiones a nivel de control. En ellas se decide la llamada o no de una función dada.

```
void pred1block(ac_int<16, true> cur_block[256], ac_int<16, true> ref_block[256],
                int bx, ac_int<6, false> *m, ac_int<32, false> *pp,
                ac_int<32, false> *filecount, ac_int<32, true> block_out[256],
                ac_int<22, true> *ym){

    ac_int<9, false> alpha, a_i = 0;
    ac_int<9, false> j = 0;
    ac_int<1, false> k0 = 1;
    ac_int<1, false> INTRA =0;
    ac_int<22, true> xm = 0;
    ac_int<17, true> p_err[256];
    ac_int<18, true> pred[256];
    /*created to make cur_block only input ++ parallelizing ++ memory*/
    ac_int<16, true> aux_cur_block[256];
    ac_int<32, false> D0 = 0;

    if(bx == 0) INTRA = 1;
    else INTRA = 0;

    init_output (filecount, block_out);

    if (INTRA){
       // 2D Prediction of the block
       xm = pred_2D(cur_block, ref_block, scan, m, pp, block_out, filecount,
                    INTRA, ym);
    }
    else{
        //Calculate MEAN of current block
        xm = mean (cur_block);

        //Estimate, quantize and dequantize alpha
        estimation_ls(cur_block, aux_cur_block, ref_block, &alpha, scan, &a_i,
                      xm, ym);

        //Rate-distortion optimization & prediction error
        D0 = calc_perr (p_err, pred, alpha, ref_block, aux_cur_block);

        if(D0 > 0) k0 = 1; /*Zero-block option not selected-1 bit size*/
        else k0 = 0; /*Zero-block option selected*/

        // Output update with alpha and xm
        write_alphamu (xm, a_i, k0, m, pp, block_out, filecount);

        if(k0 == 0){
            //If zero-block, decoded block is identical to the predictor
            zero_block (ref_block, pred, xm);
        }
        else if (k0 == 1){
            //Entropy coder
            ad_golomb_code_perr(p_err, m, pp, block_out, filecount, INTRA,
                                ref_block, pred, xm);
        }
    } /* end of else for bands other than first */
} // end of pred1block
```

Figura 3.2. Código ANSI C de la función `pred1block()`

La Figura 3.3, presenta el flujo de datos y las dependencias entre los mismos resultado de la implementación de estas modificaciones en el código de referencia. Se observa que la primera banda, $i = 0$, es procesada de forma distinta al resto de bandas, tal y como se define a nivel teórico en el algoritmo LCE.

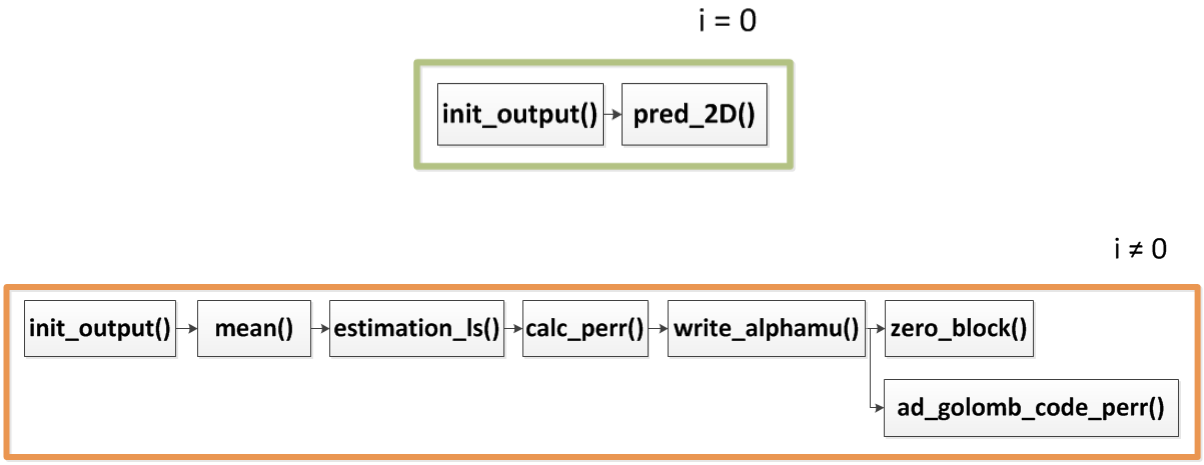

Figura 3.3. Flujos de datos

Cabe destacar que, durante la realización de esta tarea se ha verificado en todo momento que las modificaciones introducidas en el código de referencia no afectan al funcionamiento del algoritmo LCE, esto es, el código ANSI C obtenido como salida de esta tarea es funcionalmente equivalente al código de referencia. Esta verificación se ha realizado mediante la comparación entre la imagen comprimida tras las modificaciones realizadas y una imagen comprimida con anterioridad mediante el uso del software de referencia con las modificaciones aportadas en [11].

3.2.1. Módulo adgolomb.cpp

En el interior de este módulo se han incluido las funcionalidades siguientes:

- Realizar la cuantificación del error de predicción
- Cómputo del bloque de referencia, es decir, los valores reconstruidos, que han de estar disponibles para la siguiente banda
- Implementar la codificación entrópica del error de predicción
- Realizar el empaquetado a nivel de bit de las palabras código resultantes de la compresión

Las conexiones de entrada y salida relacionadas con el módulo `adgolomb.cpp` extraídas del código resultante se pueden dividir en tres grupos: control y datos.

Los parámetros de control permiten identificar los elementos necesarios para la gestión del módulo, mientras que, el grupo de datos se refiere a valores de píxel. Ambos se encuentran referidos a un bloque y se pueden observar en la Tabla 3.1.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	m	I/O	6 bits	Número de bits escritos en pp
	pp	I/O	32 bits	Buffer intermedio de valores codificados
	filecount	I/O	32 bits	Número de palabras escritas en data_out
	INTRA	I	1 bit	Indicador de predicción 2D
	xm	I	22 bits	Valor medio de los píxeles del bloque actual
DATOS	dat	I	256 x 17 bits	Array de errores de predicción
	data_out	O	256 x 32 bits	Array de palabras código
	ref_block	O	256 x 16 bits	Array de valores reconstruidos
	pred	I	256 x 18 bits	Array de valores predichos

Tabla 3.1. Interfaz del módulo `adgolomb.cpp`

3.2.2. Módulo calc_perr.cpp

En este modulo, se computan tanto el predictor como el error de predicción teniendo en cuenta el valor del parámetro $\propto$ decuantificado. Asimismo, retorna el estimador LMS ($D0$) con lo que a la salida de este módulo puede decidirse si la condición *zero_block* es empleada.

En cuanto a entradas/salidas del módulo se agrupan de la misma forma que en la sección anterior, esto es, en control y datos. Así, la Tabla 3.2, presenta la interfaz de este módulo.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	alpha	I	9 bits	Valor decuantificado de $\propto$
	D0	O	41 bits	Valor energía del error de predicción
DATOS	p_err	O	256 x 17 bits	Array de errores de predicción
	aux_cur_block	I	256 x 32 bits	Array auxiliar del bloque actual
	ref_block	I	256 x 16 bits	Array de valores reconstruidos
	pred	O	256 x 18 bits	Array de valores predichos

Tabla 3.2. Interfaz del módulo `calc_perr.cpp`

3.2.3. Módulo estimationIs.cpp

El cálculo de los parámetros $\propto_N$ y $\propto_D$, además del propio $\propto$, se incluyen en este módulo. La resta del valor medio del bloque actual y el bloque de referencia también ha sido incluida en el módulo.

Por otra parte, el cómputo del valor cuantificado de $\propto$ mediante una búsqueda dicotómica se añade como parte de este módulo. Finalmente, antes de retornar de la función, se calcula el valor decuantificado del parámetro $\propto$.

La interfaz de este módulo se presenta en la Tabla 3.3, en donde se observa la agrupación entre control y datos de los parámetros.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	xm	I	22 bits	Valor medio de los píxeles del bloque actual
	ym	I/O	22 bits	Valor medio de los píxeles del bloque anterior
	ax	O	9 bits	Valor de cuantificado de $\propto$
	alpha	O	9 bits	Valor decuantificado de $\propto$
DATOS	scan	I	64 x 8 bits	Índices a procesar en el bloque actual
	cur_block	I	256 x 16 bits	Array de valores del bloque actual
	z	I/O	256 x 16 bits	Array auxiliar del bloque actual

	t	O	256 x 16 bits	Array de valores reconstruidos

Tabla 3.3. Interfaz del módulo `estimationls.cpp`

3.2.4. Módulo init_output.cpp

Este módulo es el encargado de la inicialización a 0xFFFFFFFF de las primeras 128 posiciones del buffer de palabras código de salida del algoritmo. En la Tabla 3.4 se representa la interfaz del módulo.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	filecount	I	32 bits	Número de palabras escritas en block_out
DATOS	block_out	O	256 x 32 bits	Array de palabras código

Tabla 3.4. Interfaz del módulo `init_output.cpp`

3.2.5. Módulo mean.cpp

El valor medio de las muestras pertenecientes a una banda se computa en este módulo. Por lo tanto, la función retorna un valor entero de 22 bits con dicha media. La Tabla 3.5 presenta la interfaz perteneciente a este módulo.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	xm	O	22 bits	Valor medio de los píxeles del bloque actual
DATOS	cur_block	I	256 x 16 bits	Array de valores del bloque actual

Tabla 3.5. Interfaz del módulo `mean.cpp`

3.2.6. Módulo pred_2D.cpp

La predicción en dos dimensiones (2D), realizada para la primera banda, i=0, se implementa en este módulo. Asimismo, la codificación entrópica y el cómputo del valor medio de las muestras del bloque actual para ser utilizadas en la siguiente banda son implementados.

A nivel de interfaz, el módulo presenta los parámetros expuestos en la Tabla 3.6. En ella, pueden observase tanto los parámetros de control como de datos presentes en dicha interfaz.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	m	I/O	6 bits	Número de bits escritos en pp
	pp	I/O	32 bits	Buffer intermedio de valores codificados
	filecount	I/O	32 bits	Número de palabras escritas en data_out
	INTRA	I	1 bit	Indicador de predicción 2D
	xm	O	22 bits	Valor medio de los píxeles del bloque actual
	ym	O	22 bits	Valor medio de los píxeles del bloque anterior
DATOS	cur	I	256 x 17 bits	Array de errores de predicción
	scan	I	64 x 8 bits	Índices a procesar en el bloque actual
	ref	O	256 x 16 bits	Array de valores reconstruidos
	data_out	O	256 x 32 bits	Array de palabras código

Tabla 3.6. Interfaz del módulo `pred_2D.cpp`

3.2.7. Módulo write_alphamu.cpp

En este módulo se realiza la escritura en el buffer de palabras código del valor cuantificado de $\propto$, la media del bloque actual y el indicador de la condición *zero_block*.

La Tabla 3.7, presenta la interfaz de este módulo, así como, la descripción de cada uno de estos parámetros.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	m	I/O	6 bits	Número de bits escritos en pp
	pp	I/O	32 bits	Buffer intermedio de valores codificados
	filecount	I/O	32 bits	Número de palabras escritas en data_out
	xm	I	22 bits	Valor medio de los píxeles del bloque actual
	a_i	I	9 bits	Valor de cuantificado de $\propto$

	k0	I	1 bits	Indicador *zero_block contidion*
DATOS	data_out	O	256 x 32 bits	Array de palabras código

Tabla 3.7. Interfaz del módulo `write_alphamu.cpp`

3.2.8. Módulo zero_block.cpp

Si se activa la condición zero_block, es decir, si el valor de la predicción se aproxima al valor real del pixel, se llama a esta función. Por lo tanto, se evitan tanto la cuantificación, la codificación entrópica y el empaquetado de las palabras código.

La interfaz del módulo se presenta en la Tabla 3.8.

Grupo	Parámetro	I/O	Tamaño	Descripción
CONTROL	xm	O	22 bits	Valor medio de los píxeles del bloque actual
DATOS	pred	I	256 x 18 bits	Array de valores predichos
	ref_block	O	256 x 16 bits	Array de valores reconstruidos

Tabla 3.8. Interfaz del módulo `zero_block.cpp`

3.3. Generación del lenguaje de descripción hardware VHDL

Catapult C [12] es una herramienta de sintetizado hardware de alto nivel. Se trata de un software comercial producido por Mentor Graphics, adquirido por la compañía Calypto Design Systems. Permite la utilización de código C/C++ y la generación de salidas RTL en código de descripción hardware VHDL y Verilog, entre otros, para la síntesis hardware de FPGAs y ASICs. Catapult C permite a los usuarios indicar las restricciones de tiempo y área indicando la frecuencia de reloj a utilizar y la tecnología de destino.

La utilización básica de Catapult no resulta complicada y sólo se requiere de la inserción de los archivos .cpp para iniciar el proceso. Posteriormente, Catapult C comprueba la corrección del código fuente. Una vez que el código es correcto, se continúa con la configuración del diseño, en donde se selecciona el dispositivo a sintetizar y la frecuencia de reloj que se quiere utilizar. El siguiente paso, Catapult genera las restricciones de la arquitectura seleccionada. Una vez finalizado este paso se pasa al organizador de la síntesis y para finalizar se realiza el paso de generación del lenguaje de descripción hardware RTL seleccionado.

Catapult C proporciona acceso a la herramienta Precision que permite realizar los pasos necesarios para la síntesis del código RTL generado. Gracias a que obtiene los datos necesarios del dispositivo seleccionado de Catapult C solo se ha de pulsar el botón de sintetizado para generara automáticamente todo el proceso.

Se debe destacar que, en determinados casos, los tiempos de generación del código RTL y de la síntesis posterior son muy prolongados pudiéndose necesitar días para la finalización de este proceso.

En la Figura 3.4 y Figura 3.5 se muestra el aspecto que presenta el programa Catapult C y Precision RTL Synthesis, empleada para el emplazamiento y ruteo.

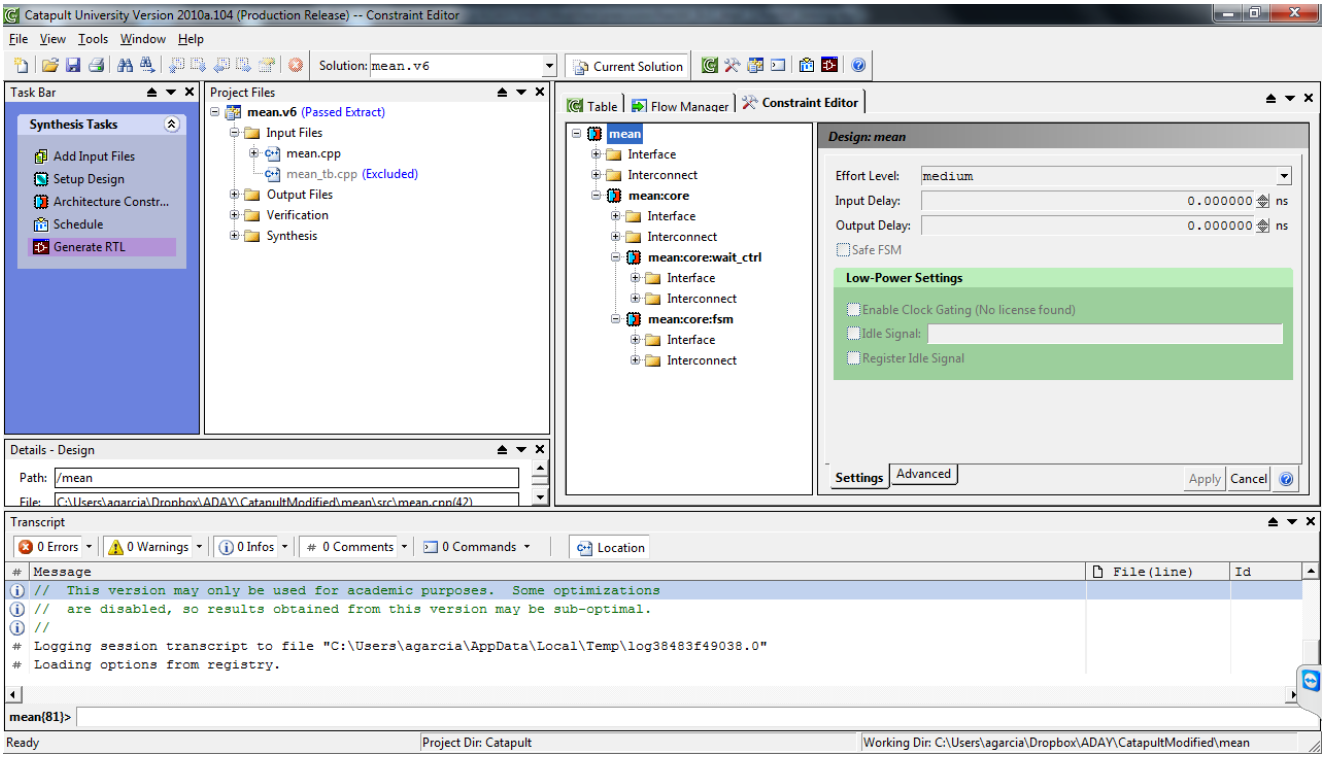

Figura 3.4. Aspecto de la interfaz de la herramienta Catapult C

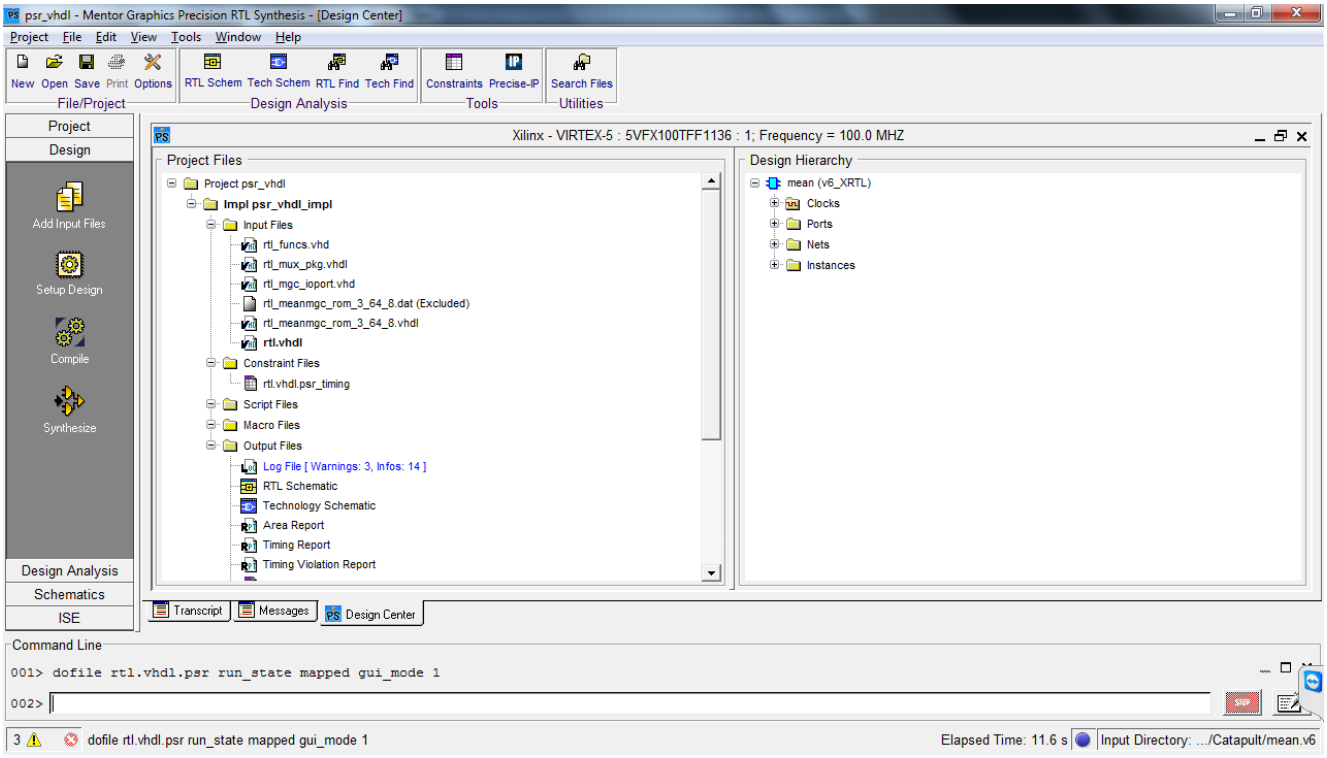

Figura 3.5. Aspecto de la interfaz de la herramienta Precision RTL Synthesis

3.3.1. Metodología de diseño en la herramienta Catapult C

En esta sección se presenta una breve descripción de la metodología de diseño que emplea la herramienta de síntesis Catapult C, para mayor información consultar [12]. Conocer esta metodología, así como en qué medida afecta un cambio en los parámetros escogidos en el hardware generado, resulta fundamental para poder realizar su integración en tareas posteriores.

3.3.1.1. Paso 1. Escritura y verificación del código C/C++

El algoritmo escrito, entrada a la herramienta, afecta al hardware resultante en mayor medida que cualquier otro paso en la metodología. Por lo tanto, la implementación de alto nivel del algoritmo debe hacerse cuidadosamente.

Es probable que, cualquier código escrito en C/C++ e introducido en la herramienta Catapult C genere un hardware en código VHDL o Verilog sintetizable. Sin embargo, los resultados que se obtendrán, en términos de latencia, área y transferencia de datos, probablemente, no cumplirán con los requisitos impuestos y, lo que es peor, será complicado cómo aumentar las prestaciones del sistema obtenido a partir de un código en C/C++ no escrito para ser usado en herramientas de síntesis de alto nivel.

En el caso de este Trabajo Fin de Máster, como ha quedado descrito, se parte de un código de referencia en ANSI C modificado para ser empleado en herramientas de síntesis de alto nivel. Asimismo, se han realizado las modificaciones necesarias con el fin de obtener módulos funcionalmente independientes.

3.3.1.2. Paso 2. Configuración de las restricciones hardware a nivel global

En este paso, se especifica, por parte del usuario, la frecuencia de reloj y la polaridad y presencia o ausencia de pines de control tales como *reset* y *enable*. Además, es en este paso donde se especifica la tecnología objeto de síntesis.

Por otra parte, se ha de especificar el módulo de más alto nivel (top module) de los comprendidos en la implementación. La herramienta, para el resto de funciones que sean llamadas desde el *top module*, creará su implementación hardware. Mientras que, aquellas que queden fuera se considerarán parte de banco de pruebas con lo que no formarán parte del diseño hardware generado.

3.3.1.3. Paso 3. Especificación de las restricciones arquitecturales

Catapult C proporciona abstracciones de alto nivel proporcionando grados de libertad al usuario con el fin de decidir la manera en que el diseño será implementado a bajo nivel. En este paso, las principales características disponibles son:

- Posibilidad de mapeo de arrays a recursos de memoria internos
- Decidir la forma en que se mapean las memorias en el diseño
- Parámetros para la optimización de bucles en el diseño. Los más relevantes son: *loop unrolling* (U), permitiendo la eliminación del bucle expandiéndolo completamente o dividiéndolo un número finito de veces, o *loop pipelining*, permitiendo el comienzo de la segunda iteración antes de la finalización de la anterior, esto se fija mediante la variable *initialization interval* (II)
- Identificación de entradas, salidas y recursos a nivel global (memorias ROMs, etc.)
- Gestión de las interfaces de entrada/salida

3.3.1.4. Paso 4. Planificación del diseño

En este paso, Catapult C planifica el diseño hardware de acuerdo a las restricciones introducidas en los pasos 2 y 3. De forma gráfica proporciona un diagrama de Gantt en donde se representan las dependencias de datos existentes. De esta forma, el usuario obtiene una visión de cómo es planificado el diseño por la herramienta.

Asimismo, la herramienta proporciona información de utilidad con el fin de conocer cómo las restricciones impuestas en el paso 3 pueden ser optimizadas.

Por último, ya en este punto se obtienen estimaciones en cuanto a área, latencia y transferencia de datos.

3.3.1.5. Paso 5. Generación del RTL

Es en este paso donde la herramienta genera los ficheros RTL del diseño. Se generan, además del código RTL en VHDL o Verilog, esquemáticos por lo que es posible identificar la ruta crítica del diseño.

En cuanto a los ficheros generados cabe destacar que se disponen de dos tipos de descripciones de bajo nivel del diseño:

- cycle.vhdl, el cual contiene una descripción a nivel de comportamiento del diseño útil a la hora de realizar simulaciones
- rtl.vhdl, descripción de diseño sintetizable, es la que se debe usar en la síntesis del mismo

Tras la generación del código RTL, la herramienta proporciona un enlace directo a la herramienta Precision para realizar el emplazamiento y ruteo del diseño generado. Estimaciones con mayor detalle del área, latencia y transferencia de datos se proporciona a la salida de esta herramienta.

3.3.1.6. Flujo de verificación SCVerify

Complementariamente a la metodología de síntesis propuesta por Catapult C, existen una serie de flujos de verificación disponibles con el objetivo de garantizar el correcto comportamiento del código RTL generado.

De esta forma, el flujo de verificación SCVerify proporciona un entorno de trabajo para la validación de las salidas de Catapult C frente a la entrada original en C/C++ empleando un banco de pruebas proporcionado por el usuario en C/C++.

Estímulos idénticos se aplican tanto al diseño de alto nivel original y al código de bajo nivel generado mediante comparación se valida que las salidas de ambos diseños es idéntica.

A lo largo de la realización de esta tarea se ha creado, para cada uno de los módulos funcionalmente independientes, un banco de pruebas con valores aleatorios con el fin de garantizar a nivel de simulación que los módulos generados por la herramienta proporcionan los mismos resultados a estímulos idénticos a la entrada. La Figura 3.6, representa el último paso ya en la herramienta de simulación y depuración ModelSim en donde se observa que el número de errores es nulo. Este proceso ha sido repetido para todos y cada uno de los módulos obtenidos.

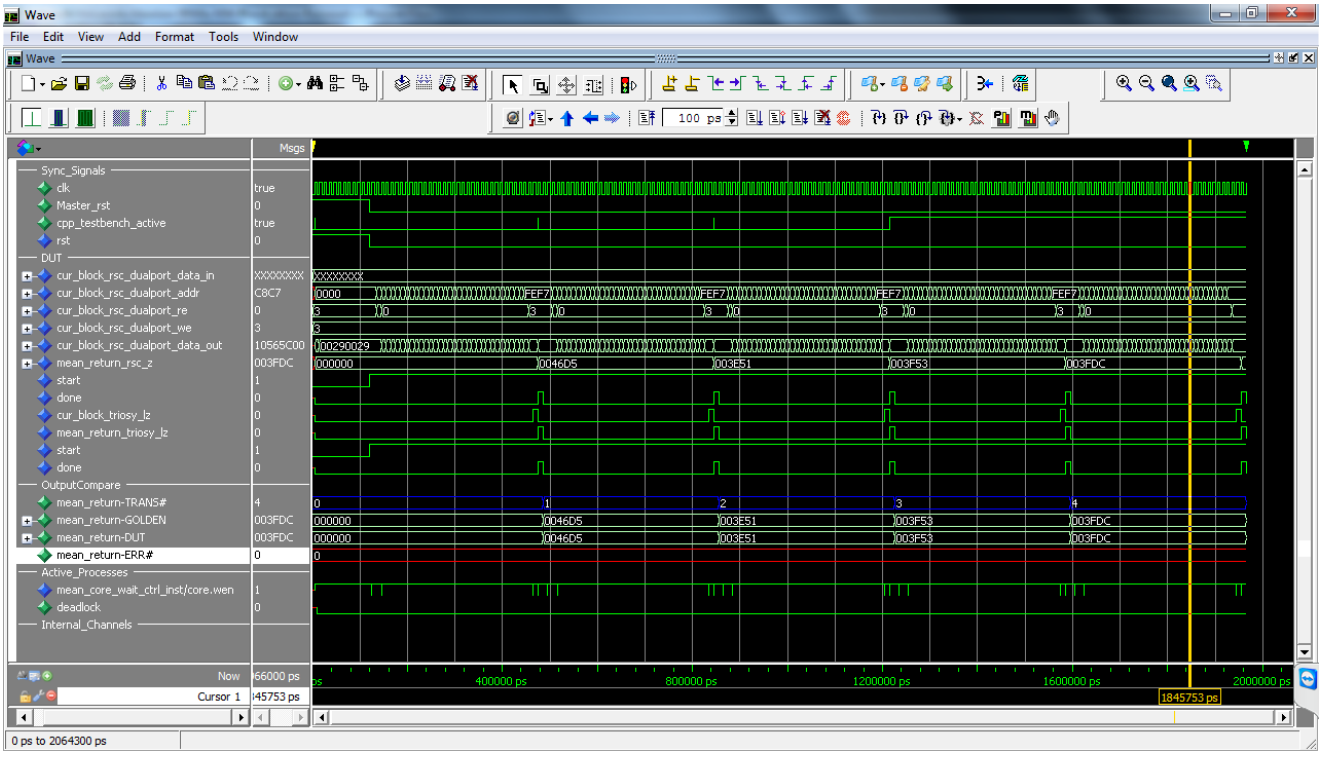

Figura 3.6. Aspecto de la interfaz de la herramienta ModelSim tras su llamada mediante Catapult C

Esta verificación a nivel de bloque permite ir con una mayor seguridad a tareas posteriores en las que se verifica con estímulos extraídos del software de referencia el correcto funcionamiento de la implementación del algoritmo LCE.

3.3.2. Implementación de los módulos mediante la herramienta Catapult C

Una vez presentada, tanta la metodología de diseño empleada en la herramienta, como el flujo de verificación usado, se introduce, en esta sección, la implementación de los distintos módulos funcionalmente independientes obtenidos tras la ejecución de la tarea descrita con anterioridad.

La tecnología objeto de la implementación realizada con la herramienta es una FPGA de la familia Virtex-5 del fabricante XILINX. En concreto se ha empleado el modelo XC5FX100T con *speed grade* -1 y encapsulado FF1136.

Asimismo, se ha definido como objetivo, conseguir una frecuencia de funcionamiento por encima de los 75MHz para cada uno de los módulos.

La implementación de los módulos se ha realizado siguiendo la metodología descrita para la herramienta. A parte de las restricciones impuestas a nivel global, se han habilitado las siguientes señales a nivel de interfaz:

- Señal *start* habilitada.

  Señal activa a nivel alto. El módulo permanece a la espera de la activación de esta señal para comenzar su actividad.

- Señal *done* habilitada.

  Señal activa a nivel alto. Indica la finalización del procesado por parte del módulo. Éste permanece a la espera de una nueva señal de *start*.

- Señal *rst* habilitada.

  Señal activa a nivel alto. Inicializa las señales pertenecientes al módulo.

3.3.2.1. Implementación del módulo `adgolomb.cpp`

A nivel arquitectural, este módulo contiene un bucle principal denominado MAP. Asimismo, anidados dentro de éste, contiene tres bucles adicionales denominados abs_adgolomb:for, find_lead_one:for y CALCK. Para el bucle principal se ha escogido para su pipeline II=1, mientras que los bucles adicionales han sido expandidos completamente, es decir U=1, con lo que su ejecución se ha paralelizado completamente. El bucle principal contiene 256 iteraciones mientras que el resto contiene 18 iteraciones para el bucle abs_adgolomb:for, 18 iteraciones para el bucle find_lead_one:for y 22 iteraciones para el bucle CALCK. Notar que se han aplicado varias opciones de optimizaciones de bucles obteniéndose mejores resultados con las configuraciones que se comentan.

A nivel de interfaces, el bloque queda descrito en la Figura 3.7. En ella se observa que en cuanto a las entradas al módulo se han inferido dos memorias RAM, una de 256 x 17 bits para la variable de entrada `dat` y una de 256 x 18 bits para la variable `pred`, un registro de 22 bits para `xm` y uno de 1 bit para `INTRA`. Además, las variables bidireccionales, es decir, de entrada-salida, `filecount`, `pp` y `m` han sido inferidas en registros con anchos de 32 bits para las dos primeras y 6 bits para la última. Finalmente, para las variables de salida se han inferido dos memorias RAM, una de 256 x 32 bits para la variable `data_out` y una de 256 x 16 bits para la variable `ref_block`. Por lo tanto, se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

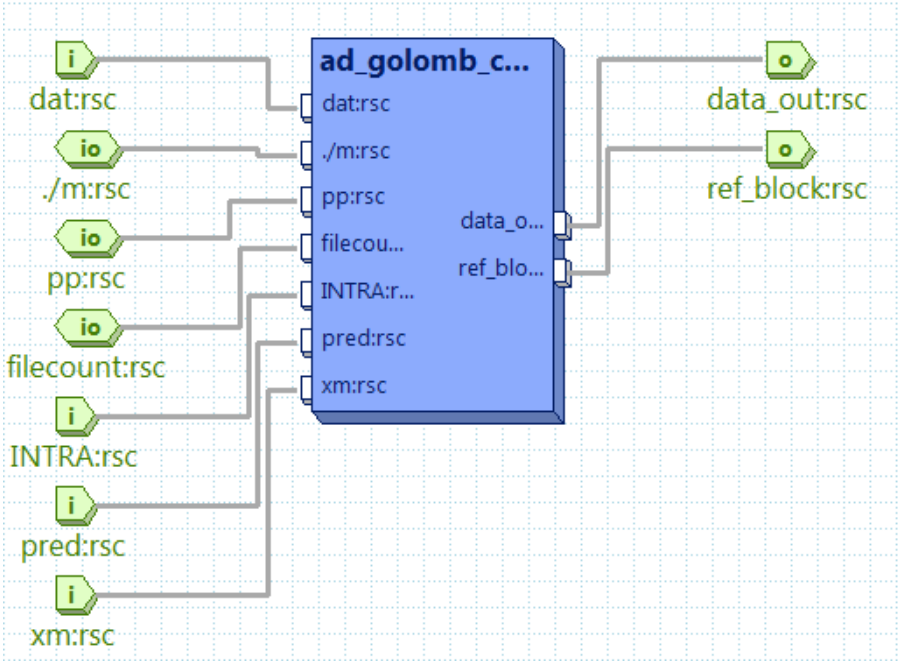

Figura 3.7. Interfaz módulo `adgolomb.cpp`

Las estimaciones dadas por la herramienta a su salida son los presentados en la Tabla 3.9.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	2	266	1 muestra/ciclo	4146,73	1,42

Tabla 3.9. Estimaciones para el módulo `adgolomb.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.10.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	2502 (3,05%)	626 (3,06%)	1 RAMB18 (0,34%)	5 (1,56%)	85,020MHz	11,762ns

Tabla 3.10. Resultados obtenidos tras el emplazamiento y ruteo del módulo `adgolomb.cpp`

3.3.2.2. Implementación del módulo `calc_perr.cpp`

Este módulo contiene un bucle denominado PERR1, anidado dentro de éste existe un bucle denominado abs_perr:for. El bucle principal contiene 256 iteraciones y se ha escogido U=2 e II=1. Por otro lado, el bucle anidado se ha expandido completamente, es decir, U=1. Notar que se han aplicado varias configuraciones obteniéndose mejores resultados con la configuración que se presenta.

A nivel de interfaces, el bloque queda descrito en la Figura 3.8. En ella se puede observar que se ha inferido dos memorias RAMs de doble puerto, por una parte, una de 256 x 16 bits para la variable de entrada `ref_block`, y una de 256 x 16 bits para la variable de entrada `aux_cur_block`. Además, se infiere, un registro de 9 bits para la variable de entrada `alpha`. En cuanto a variables de salida, se infieren dos memorias RAM de doble puerto, una de 256 x 17 bits para la variable `p_err`, y una de 256 x 18 bits para la variable `pred`. Asimismo, para la variable retornada por la función `calc_perr.return` se infiere un registro de 41 bits. Por lo tanto, se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

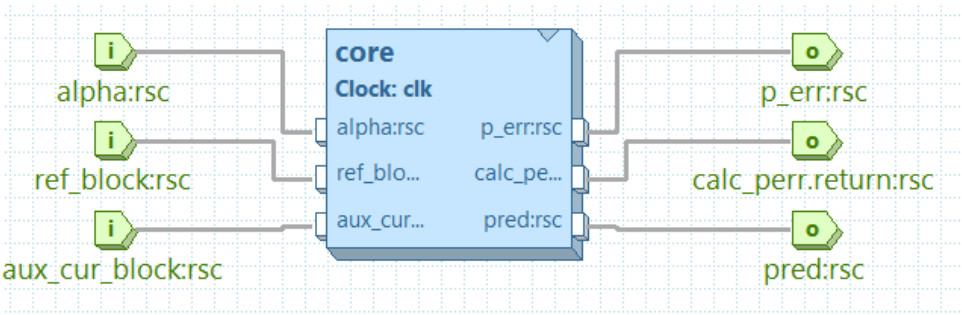

Figura 3.8. Interfaz módulo `calc_perr.cpp`

Las estimaciones dadas por la herramienta a su salida para esta solución son los presentados en la Tabla 3.11.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	2	135	1 muestra/ciclo	3125,60	2,11

Tabla 3.11. Estimaciones para el módulo `calc_perr.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.12.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	893 (1,40%)	224 (1,40%)	---	10 (3,91%)	108,707MHz	9,199ns

Tabla 3.12. Resultados obtenidos tras el emplazamiento y ruteo del módulo `calc_perr.cpp`

3.3.2.3. Implementación del módulo `estimationls.cpp`

Dentro del módulo denominado `estimationls.cpp` nos encontramos ante los bucles EST, abs_estls:for y alpha_quant:for. Asimismo, dentro de este último podemos encontrar a su vez dos bucles adicionales denominados abs_estls#1:for y abs_estls#2:for. El bucle EST presenta 256 iteraciones y se ha empleado II=1. El bucle abs_estls:for ha sido completamente expandido. En cuanto al bucle alpha _quant:for ha sido configurado con un II=3 para que pueda ser planificado por la herramienta. Finalmente, los bucles anidados abs_estls#1:for y abs_estls#2:for se han expandido completamente aumentando de esta forma el paralelismo en sus iteraciones.

A nivel de interfaces, el bloque se presenta en la Figura 3.9. En ella observa que se han inferido dos memorias RAM de doble puerto, una de 256 x 16 bits para la variable de entrada `cur_block`, y una de 64 x8 bits para la variable de entrada `scan`. Un registro ha sido inferido de la misma forma para la variable de entrada `xm`. En cuanto a las variables bidireccionales, se infiere, un registro para la variable `ym` y una memoria RAM de doble puerto de 256 x 16 para la variable `z`. Las variables de salida `alpha` y `ax` han sido mapeadas en registros de 9 bits mientras que la variable `t` ha sido mapeada en una memoria RAM de doble puerto de 256 x 16 bits.

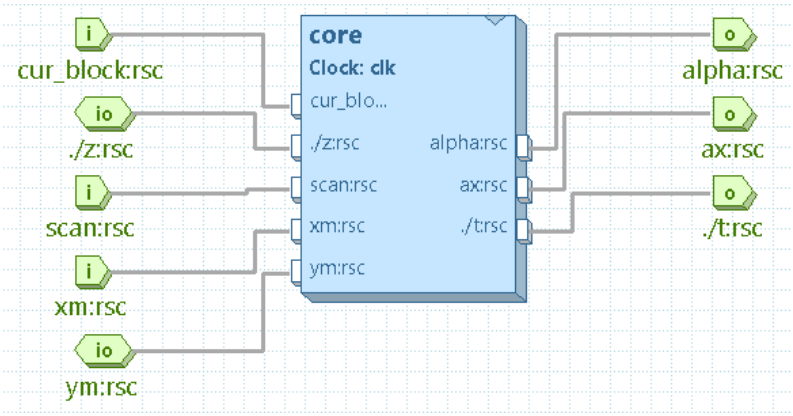

Figura 3.9. Interfaz módulo `estimationls.cpp`

Las estimaciones dadas por la herramienta a su salida para esta solución son los presentados en la Tabla 3.13.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	3	281	1 muestra/ 3 ciclos	4861,29	1,08

Tabla 3.13. Estimaciones para el módulo `estimationls.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.14.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	1187 (1,85%)	297 (1,86%)	---	8 (3,13%)	89,055MHz	11.229ns

Tabla 3.14. Resultados obtenidos tras el emplazamiento y ruteo del módulo `estimationls.cpp`

3.3.2.4. Implementación del módulo `init_output.cpp`

Este módulo contiene un único bucle denominado INIT1. Contiene 128 iteraciones y se ha escogido U=2 e II=1, con lo que se ha paralelizado su ejecución al dividirlo en 2 bucles que se ejecutan en paralelo. Notar que se han aplicado varias configuraciones obteniéndose mejores resultados con la configuración que se comenta.

A nivel de interfaces, el bloque queda descrito en la Figura 3.10. En ella se puede observar que se ha inferido una memoria RAM de doble puerto de 256 x 32 bits para la variable de salida `block_out` y un registro de 32 bits para la variable de entrada `filecount`. Con lo que se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

Figura 3.10. Interfaz módulo `init_output.cpp`

Las estimaciones dadas por la herramienta a su salida son los presentados en la Tabla 3.15.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	2	63	1 muestra/ciclo	98,83	6,56

Tabla 3.15. Estimaciones para el módulo `init_output.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.16.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	71 (0,11%)	18 (0,39%)	---	---	356,761MHz	2,803ns

Tabla 3.16. Resultados obtenidos tras el emplazamiento y ruteo del módulo `init_output.cpp`

3.3.2.5. Implementación del módulo `mean.cpp`

A nivel arquitectural, este módulo contiene dos bucles denominados ACC y abs_mean:for. El primero de ellos contiene 64 iteraciones y se ha escogido U=2 e II=1. Por otra parte, el segundo bucle contiene 32 iteraciones escogiéndose U=1 con lo que el bucle se ha expandido completamente. Notar que se han aplicado varias configuraciones obteniéndose mejores resultados con las configuraciones que se comentan.

A nivel de interfaces, el bloque queda descrito en la Figura 3.11. En ella se puede observar que se ha inferido una memoria RAM de 256 x 16 bits para la variable de entrada `cur_block`, una memoria ROM de 64 x 8 bits para la variable de entrada `scan_mean` y un registro de 22 bits para la variable de salida `mean.return`. Con lo que se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

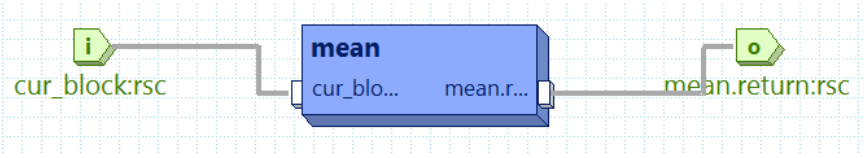

Figura 3.11. Interfaz módulo `mean.cpp`

Las estimaciones dadas por la herramienta a su salida son los presentados en la Tabla 3.17.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	2	33	1 muestra/ciclo	436,32	1,80

Tabla 3.17. Estimaciones para el módulo `mean.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.18.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	251 (0,39%)	63 (0,39%)	---	---	181,686MHz	5,504ns

Tabla 3.18. Resultados obtenidos tras el emplazamiento y ruteo del módulo `mean.cpp`

3.3.2.6. Implementación del módulo `pred_2D.cpp`

A nivel arquitectural, este módulo contiene un bucle principal denominado ROW que contiene anidados dos bucles adicionales abs_18b:for y CALCK. Asimismo, el bucle abs_pred_2D:for es llamado en el interior del módulo. Para el bucle principal se ha escogido II=10, con lo que se obtendrá una muestra cada 10 ciclos de reloj. Por otra parte, los bucles anidados han sido expandidos completamente, es decir U=1, con lo que su ejecución se ha paralelizado completamente. El bucle principal contiene 256 iteraciones

mientras que el resto contiene 18 iteraciones para el bucle abs_18b:for, 18 iteraciones para el bucle abs_pred_2D:for y 22 iteraciones para el bucle CALCK. Notar que se han aplicado varias configuraciones obteniéndose mejores resultados con las configuraciones que se comentan.

A nivel de interfaces, el bloque queda descrito en la Figura 3.12. En ella se observa que en cuanto a las entradas al módulo se han inferido dos memorias RAM, una de 256 x 16 bits para la variable de entrada `cur` y una de 64 x 8 bits para la variable `scan` y un registro de 1 bit para `INTRA`. Además, las variables bidireccionales, es decir, de entrada-salida, `filecount`, `pp` y `m` han sido inferidas en registros con anchos de 32 bits para las dos primeras y 6 bits para la última. Asimismo, una memoria RAM de doble puerto de 256 x 16 bits ha sido inferida para la variable `ref`. Finalmente, para las variables de salida se ha inferido una memoria RAM de doble puerto de 256 x 32 bits para la variable `data_out`. Además, se han inferido dos registros de 22 bits para las variables de salida `ym` y `pred_2D.return`.Por lo tanto, se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

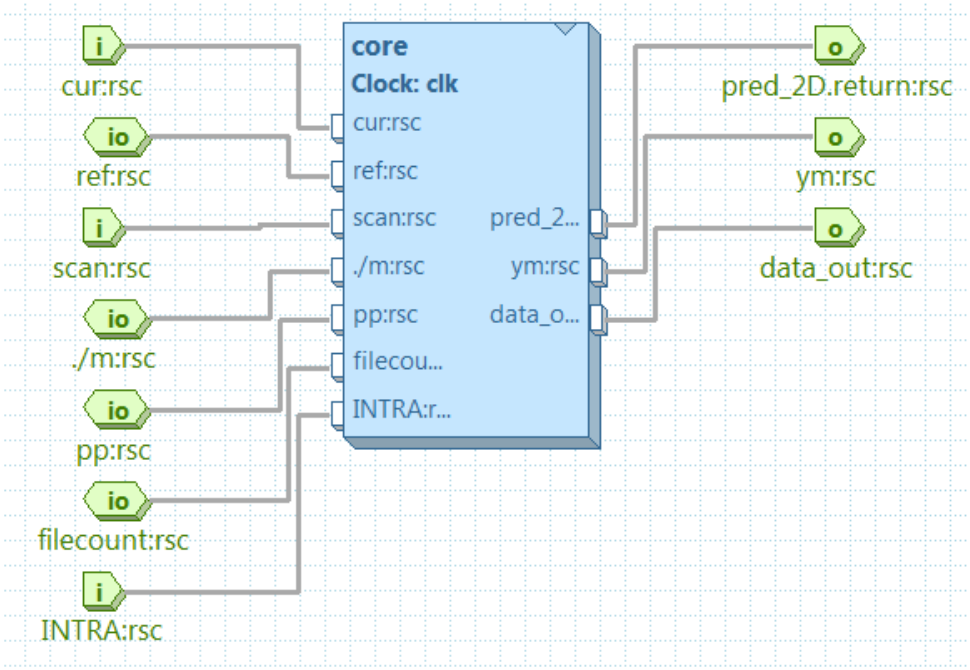

Figura 3.12. Interfaz módulo `pred_2D.cpp`

Las estimaciones dadas por la herramienta a su salida son los presentados en la Tabla 3.19.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	10	2566	1 muestra/ 10 ciclos	4466,15	0,14

Tabla 3.19. Estimaciones para el módulo `pred_2D.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.20.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	2888 (4,51%)	722 (4,51%)	---	2 (0,78%)	81,433MHz	12,280ns

Tabla 3.20. Resultados obtenidos tras el emplazamiento y ruteo del módulo `pred_2D.cpp`

3.3.2.7. Implementación del módulo `write_alphamu.cpp`

Dentro del módulo denominado `write_alphamu.cpp` no existen bucles en su interior. Por lo tanto, salvo dependencias de datos, su ejecución puede realizarse completamente en paralelo. Dado que se realiza un desplazamiento a la variable de entrada `xm` antes de su escritura en memoria junto con $\propto$, su ejecución se realiza en 2 ciclos de reloj.

A nivel de interfaces, el bloque queda descrito en la Figura 3.13. En ella se puede observar que se ha inferido una memoria RAM de doble puerto de 256 x 32 bits para la variable de salida `block_out`. En cuanto a variables de entrada se infiere: un registro de 22 bits para `xm`, uno de 9 bits para `a_i` y uno de 1 bit para `k0`. Finalmente, se infieren tres registros de entrada-salida de 6 bits para `m`, de 32 bits para `pp` y de 32 bits para `filecount`. Por lo tanto, se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

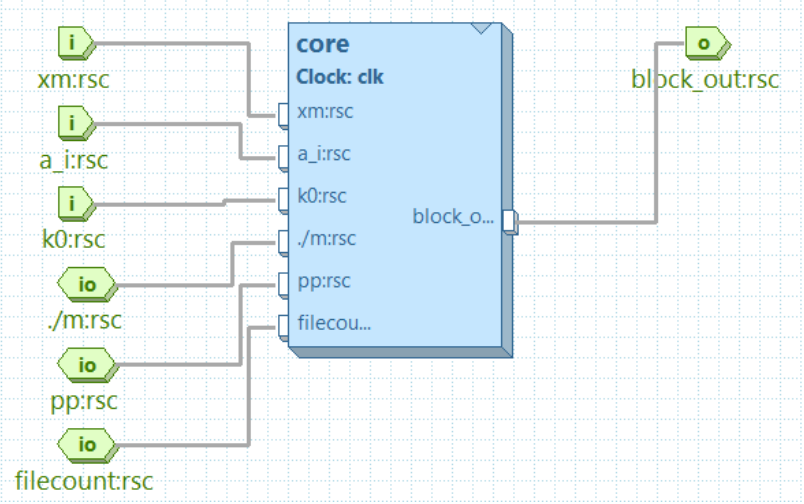

Figura 3.13. Interfaz módulo `write_alphamu.cpp`

Las estimaciones dadas por la herramienta a su salida para esta solución son los presentados en la Tabla 3.21.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	2	2	1 muestra/ciclo	688,12	4,61

Tabla 3.21. Estimaciones para el módulo `write_alphamu.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.22.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	503 (0,79%)	126 (0,79%)	---	---	283,046MHz	3.533ns

Tabla 3.22. Resultados obtenidos tras el emplazamiento y ruteo del módulo `write_alphamu.cpp`

3.3.2.8. Implementación del módulo `zero_block.cpp`

Este módulo contiene un único bucle denominado ZEROB. Contiene 256 iteraciones y se ha escogido U=2 e II=1. Notar que se han aplicado varias configuraciones obteniéndose mejores resultados con la configuración que se comenta.

A nivel de interfaces, el bloque queda descrito en la Figura 3.14. En ella se puede observar que se ha inferido dos memorias RAMs de doble puerto, por una parte, una de 256 x 16 bits para la variable de salida `ref_block`, y, por otra parte, una de 256 x 18 bits para la variable de entrada `pred`. Además, se infiere, un registro de 22 bits para la variable de entrada `xm`. Por lo tanto, se observa la trazabilidad entre las interfaces presentadas en la sección 3.2 a nivel software y las obtenidas a la salida de la herramienta.

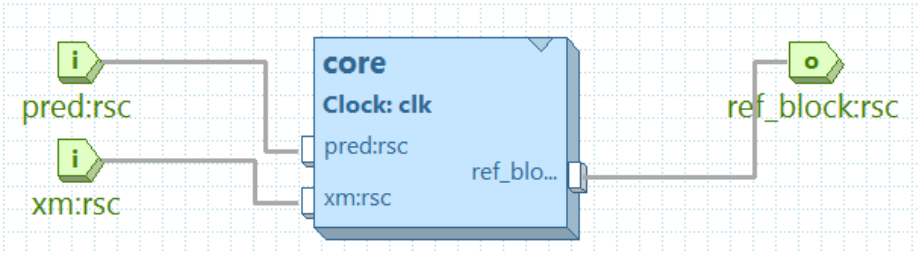

Figura 3.14. Interfaz módulo `zero_block.cpp`

Las estimaciones dadas por la herramienta a su salida para esta solución son los presentados en la Tabla 3.23.

Tecnología	Ciclos en la primera iteración	Ciclos totales	Transferencia de datos	Área (puertas)	Margen (ns)
5VFX100 (-1)	2	127	1 muestra/ciclo	213,97	4,39

Tabla 3.23. Estimaciones para el módulo `zero_block.cpp`

Asimismo, los resultados tras el emplazamiento y ruteo son dados en la Tabla 3.24.

Tecnología	LUT (*Look-up table*)	*Slices* usadas	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
5VFX100 (-1)	152 (0,22%)	38 (0,22%)	---	---	298,191MHz	3,354ns

Tabla 3.24. Resultados obtenidos tras el emplazamiento y ruteo del módulo `zero_block.cpp`

3.4. Diseño, implementación y verificación del controlador hardware

A lo largo de esta sección, se describe funcionalmente la arquitectura propuesta para la implementación del algoritmo de compresión de imágenes hiperespectrales LCE. La descripción se realizará siguiendo una estructura *top-down*, es decir, en primer lugar se presenta la arquitectura final, para luego bajar de nivel detallando cada uno de los módulos que la componen.

Para la implementación del algoritmo LCE se ha diseñado, implementado y verificado un módulo denominado controlador hardware, en adelante hwctrl (*Hardware Controller*). Dentro de sus funciones principales destacan la conexión a bajo nivel de cada uno de los módulos obtenidos en la tarea anterior y gestionar el flujo de datos presente. Para ello ha de proporcionar adecuadamente los datos a la entrada de cada uno de dichos módulos así como almacenar sus valores de salida. Asimismo, ha de inferir internamente las memorias necesarias para el correcto funcionamiento de los módulos.

3.4.1. Diseño del módulo hwctrl

El esquema general del bloque funcional encargado de la implementación del algoritmo LCE se muestra en la Figura 3.15. En términos generales, presenta las siguientes características:

- El uso de memorias tanto a nivel de interfaces como interno es compartido por cada uno de los módulos. Esto se traduce en una mayor sencillez a la hora de abordar su implementación a costa de perder paralelismo en su ejecución.
- Empleo de señales de *start* y *done* en cada módulo. Esto facilita conocer en qué estado se encuentra cada módulo, es decir, si se encuentra a la espera de la activación de la señal de *start* para comenzar su procesado o si ha terminado con el mismo si la señal *done* es activada.
- Empleo de una señal de *reset* asíncrona para todos los módulos y registros.
- Señal de reloj, denominada *clk*, activa a nivel alto.
- Se han añadido señales para la validación de todos los datos disponibles a la salida del módulo, así como, una señal pulsada que indica la finalización del procesado de cada banda.
- Internamente se encuentra compuesto por ocho módulos que se corresponden con aquellos obtenidos tras su implementación mediante la herramienta Catapult C.

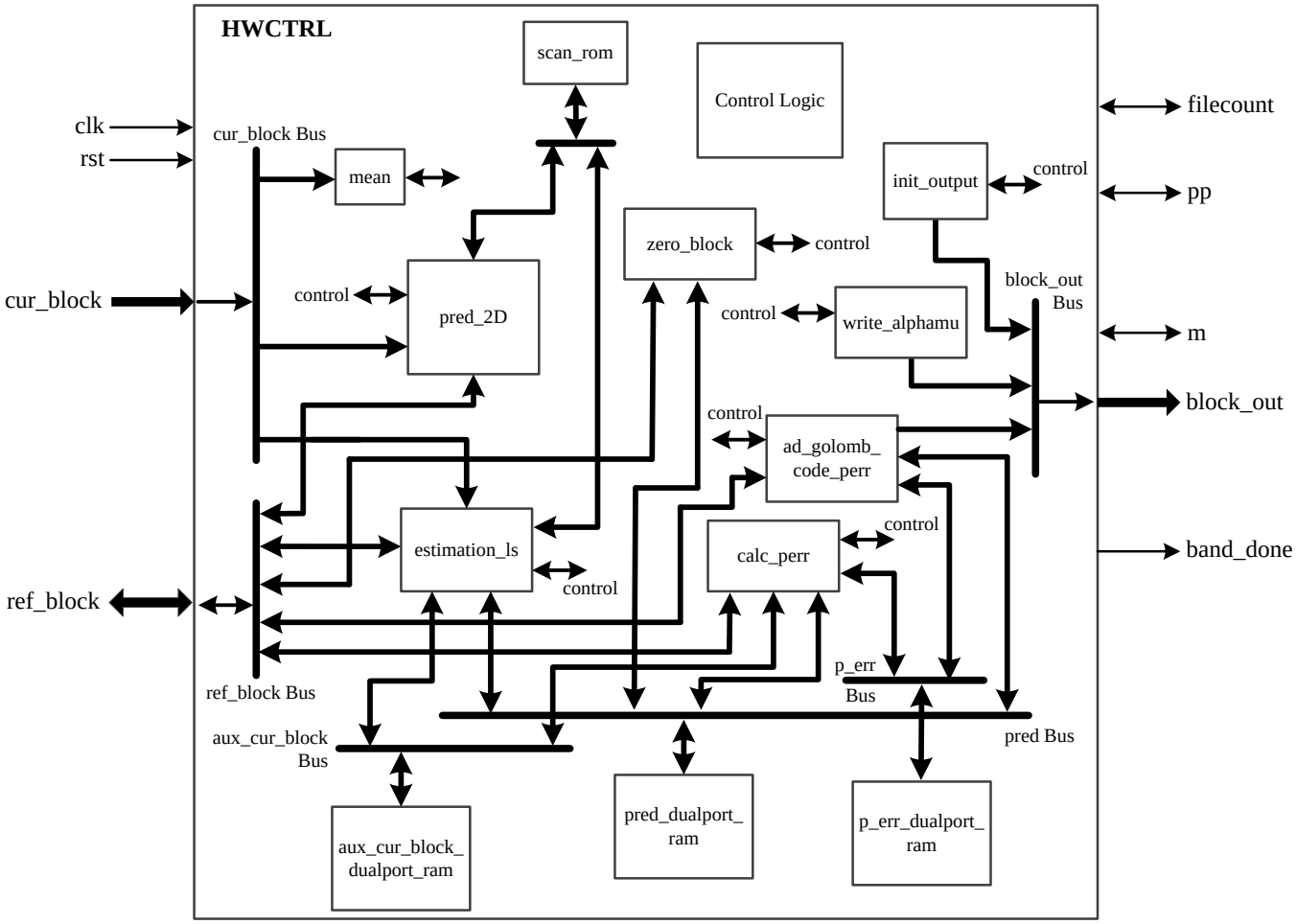

Figura 3.15.Controlador hardware (hwctrl), esquema general

A nivel de interfaces, visto el módulo como caja negra (*black-box*), el controlador hardware diseñado presenta la configuración mostrada en la Figura 3.16.

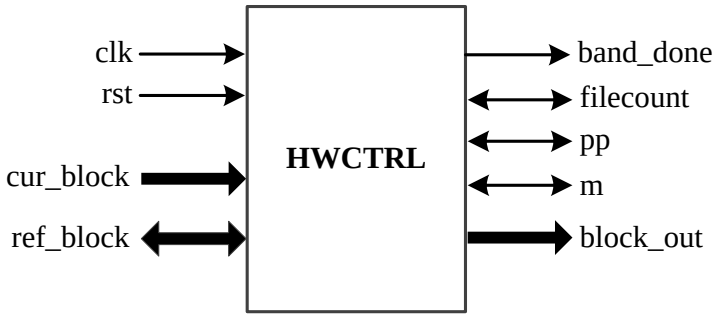

Figura 3.16. Interfaces del módulo controlador de hardware (hwctrl)

Por consiguiente, su interfaz de comunicaciones presenta las señales mencionadas en la Tabla 3.25. Ésta, recoge, asimismo, la descripción de las señales de entrada/salida del hwctrl diseñado.

Nombre	Ancho	Tipo
clk	**1 bit**	**Entrada**
Reloj del sistema activo a nivel alto.		
rst	**1 bit**	**Entrada**
Reset asíncrono activo a nivel alto.		
band_done	**1 bit**	**Salida**
Señal activa a nivel alto que indica la finalización del procesado de una banda. Su duración es de un ciclo de la señal de reloj.		
cur_block_data_in	**32 bits**	**Salida**
Bus de datos perteneciente a la memoria RAM de doble puerto que contiene el bloque actual.		
cur_block_data_addr	**16 bits**	**Salida**
Bus de direcciones perteneciente a la memoria RAM de doble puerto que contiene el bloque actual.		
cur_block_data_re	**2 bits**	**Salida**
Bus de habilitación de lectura perteneciente a la memoria RAM de doble puerto que contiene el bloque actual.		
cur_block_data_we	**2 bits**	**Salida**
Bus de habilitación de escritura perteneciente a la memoria RAM de doble puerto que contiene el bloque actual.		
cur_block_data_out	**32 bits**	**Entrada**
Bus de datos perteneciente a la memoria RAM de doble puerto que contiene el bloque actual.		
ref_block_data_in	**32 bits**	**Salida**
Bus de datos perteneciente a la memoria RAM de doble puerto que contiene el bloque de referencia.		
ref_block_data_addr	**16 bits**	**Salida**
Bus de direcciones perteneciente a la memoria RAM de doble puerto que contiene el bloque de referencia.		
ref_block_data_re	**2 bits**	**Salida**
Bus de habilitación de lectura perteneciente a la memoria RAM de doble puerto que contiene el bloque de referencia.		
ref_block_data_we	**2 bits**	**Salida**
Bus de habilitación de escritura perteneciente a la memoria RAM de doble puerto que contiene el bloque de referencia.		
ref_block_data_out	**32 bits**	**Entrada**

Bus de datos perteneciente a la memoria RAM de doble puerto que contiene el bloque de referencia.		
filecount_in	**32 bits**	**Entrada**
Número de palabras de 32 bits escritas en la memoria de salida.		
filecount_out	**32 bits**	**Salida**
Número de palabras de 32 bits escritas en la memoria de salida.		
filecount_valid	**32 bits**	**Salida**
Señal que, activa a nivel alto, indica la validez de un nuevo valor de filecount. Su duración es de un ciclo de la señal de reloj.		
pp_in	**32 bits**	**Entrada**
Buffer intermedio de palabras código.		
pp_out	**32 bits**	**Salida**
Buffer intermedio de palabras código.		
pp_valid	**32 bits**	**Salida**
Señal que, activa a nivel alto, indica la validez de un nuevo valor de pp. Su duración es de un ciclo de la señal de reloj.		
m_in	**6 bits**	**Entrada**
Indicador de bits escritos en el buffer intermedio pp.		
m_out	**6 bits**	**Salida**
Indicador de bits escritos en el buffer intermedio pp.		
m_valid	**6 bits**	**Salida**
Señal que, activa a nivel alto, indica la validez de un nuevo valor de m. Su duración es de un ciclo de la señal de reloj.		
block_out_data_in	**64 bits**	**Salida**
Bus de datos perteneciente a la memoria RAM de doble puerto que contiene el bloque de palabras código como resultado de la aplicación del algoritmo.		
block_out_data_addr	**16 bits**	**Salida**
Bus de direcciones perteneciente a la memoria RAM de doble puerto que contiene el bloque de palabras código como resultado de la aplicación del algoritmo.		
block_out_data_re	**2 bits**	**Salida**
Bus de habilitación de lectura perteneciente a la memoria RAM de doble puerto que contiene el bloque de palabras código como resultado de la aplicación del algoritmo.		
block_out_data_we	**2 bits**	**Salida**
Bus de habilitación de escritura perteneciente a la memoria RAM de doble puerto que contiene el bloque de palabras código como resultado de la aplicación del algoritmo.		

block_out_data_out	64 bits	Entrada
Bus de datos perteneciente a la memoria RAM de doble puerto que contiene el bloque de palabras código como resultado de la aplicación del algoritmo.		

Tabla 3.25. Interfaz del módulo hwcrtrl

3.4.2. Implementación del módulo hwctrl

Definida la funcionalidad de la arquitectura hwctrl, el paso siguiente es su implementación. La implementación realizada en este Trabajo Fin de Máster se basa en su descripción en código HDL (*Hardware Description Language*) para su modelado hardware. En esta descripción se ha definido la funcionalidad del circuito al nivel de abstracción RTL (*Register Transfer Level*), con el fin de garantizar que el código resultante sea completamente sintetizable. El lenguaje de descripción VHDL [31] ha sido el utilizado para la implementación de este diseño. VHDL es un lenguaje de descripción hardware ampliamente usado para describir sistemas electrónicos. Este lenguaje soporta el diseño, prueba e implementación de circuitos analógicos, digitales y mixtos a diferentes niveles de abstracción.

En esta sección se presenta la implementación, del módulo hwctrl diseñado. La descripción de cada uno de los módulos funcionalmente independientes implementados mediante la herramienta Catapult C ya se aborda con anterioridad en la sección 3.3.

3.4.2.1. Módulo hwctrl

El módulo encargado de la correcta conexión de los módulos obtenidos mediante la herramienta Catapult C es el denominado hwctrl. Asimismo, a nivel de flujo de datos desempeña un papel fundamental ya que es este módulo el que controla el mismo durante la ejecución del algoritmo LCE. De este modo, ha de realizar la conexión de las siguientes instancias:

- ad_golomb_code_perr_inst
- zero_block_inst
- write_alphamu_inst
- calc_perr_inst
- estimation_ls_inst
- mean_inst

- pred_2D_inst
- init_output_inst

Para la implementación de estas tareas se ha diseñado una máquina de estados finitos, FSM (*Finite State Machine*). El diagrama de estados de la FSM perteneciente a este módulo es el que se representa en la Figura 3.17, y consta de nueve estados. La explicación de la funcionalidad presente en cada estado es descrita en la Tabla 3.26.

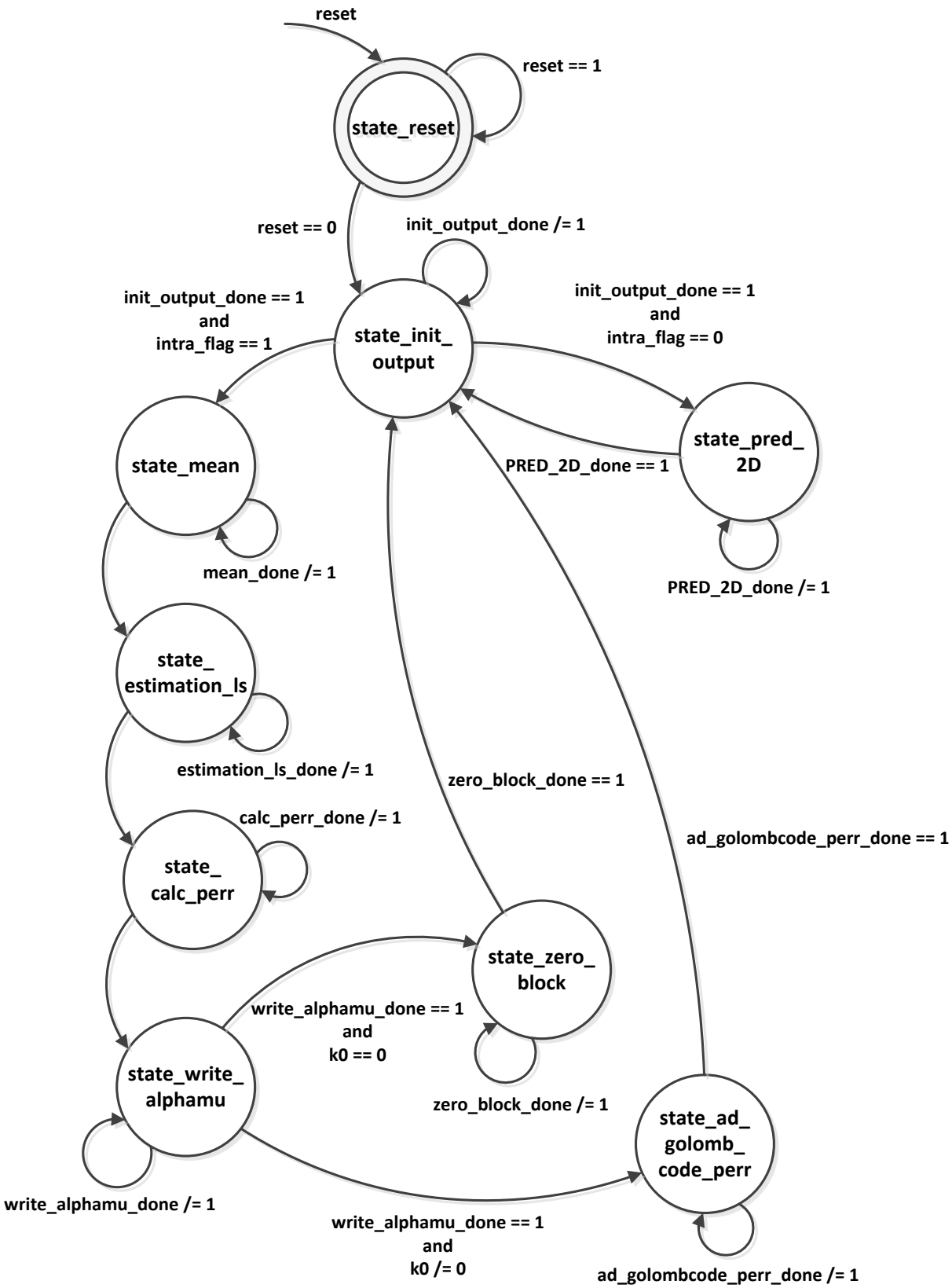

Figura 3.17. Diagrama de estados de la FSM perteneciente al bloque funcional hwctrl

Estado	Descripción
state_reset	Estado inicial. Se entra en él después de un reset. Se permanece en este estado hasta que la señal de reset se sitúa a nivel bajo, '0' lógico.
state_init_output	Estado de inicialización de la memoria de salida. En este estado se activa el módulo `init_output`. Se permanece en este estado hasta que la señal `init_output_done` se active durante un ciclo de la señal de reloj.
state_pred_2D	Predicción en 2 dimensiones o INTRA. En este estado se activa el módulo `pred_2D`. Se permanece en este estado hasta que la señal `pred_2D_done` sea activada durante un ciclo de la señal de reloj.
state_mean	Cálculo de la media del bloque actual. En este estado se activa el módulo `mean`. Se permanece en este estado hasta que la señal `mean_done` sea activada durante un ciclo de la señal de reloj.
state_estimation_ls	Cálculo de $\propto$. En este estado se activa el módulo `estimation_ls`. Se permanece en este estado hasta que la señal `estimation_ls_done` sea activada durante un ciclo de la señal de reloj.
state_calc_perr	Cálculo del error de predicción. En este estado se activa el módulo `calc_perr`. Se permanece en este estado hasta que la señal `calc_perr_done` sea activada durante un ciclo de la señal de reloj.
state_write_alphamu	Escritura de $\propto$, media del bloque actual e indicador *zero_block*. En este estado se activa el módulo `write_alphamu`. Se permanece en este estado hasta que la señal `write_alphamu_done` sea activada durante un ciclo de la señal de reloj.
state_zero_block	Módulo de condición *zero_block*. En este estado se activa el módulo `zero_block`. Se permanece en este estado hasta que la señal `zero_block_done` sea activada durante un ciclo de la señal de reloj.

state_ad_golomb_code_perr	Escritura de palabras código. En este estado se activa el módulo `ad_golomb_code_perr`. Se permanece en este estado hasta que la señal `ad_golomb_code_perr_done` sea activada durante un ciclo de la señal de reloj.

Tabla 3.26. Estados de la FSM del módulo hwctrl

La integración se ha llevado a cabo siguiendo la siguiente estrategia. Primero, se ha realizado la integración obviando los controles con las interfaces de entrada y salida. Al resultado de esta integración se le añadide el control de las interfaces del módulo. Posteriormente, se añaden al diseño las memorias necesarias para el funcionamiento del diseño. La integración se ha desarrollado de esta forma ya que así se plantea un diseño fácilmente adaptable a otro tipo de tecnologías.

Tanto las memorias RAM internas del módulo denominadas pred_dualport_ram, p_err_dualport_ram y aux_cur_block_dualport_ram como la memoria ROM scan_rom, necesarias para la implementación del bloque funcional hwctrl han sido generadas a partir de las librerías disponibles en la herramienta Catapult C para memorias RAM de doble puerto y memorias ROM. Para esta generación, es necesario especificar el ancho del bus de cada memoria, la cantidad de palabras, el número de puertos sus características (entrada/salida, sólo entrada, sólo salida, etc.) y definir las señales de control de cada una (*enable*, *reset*, *write enable*, etc.). El resultado es un código VHDL sintetizable del que se dispone de la interfaz para poder interactuar con la macrocélula correspondiente a la memoria de la FPGA, siendo únicamente necesario instanciar el código generado en la descripción del diseño que utilizará dicha memoria. Las características que necesitan especificarse de cada memoria en la generación de éstas se presentan en la Tabla 3.27. Cabe destacar que todas las memorias empleadas registran el bus de direcciones.

Tipo Memoria	Ancho de bus	Nº de palabras	Puertos	Señales de control
RAM (pred_dualport_ram)	18 bits	256	2 puertos duales	***reset*, *clock*, *write enable*, *read enable***
RAM (p_err_dualport_ram)	17 bits	256	2 puertos duales	***reset*, *clock*, *write enable*, *read enable***

RAM (aux_cur_block_dualport_ram)	16 bits	256	2 puertos duales	***reset*, *clock*, *write enable*, *read enable***
ROM (scan_rom)	8 bits	64	1 puerto	---

Tabla 3.27. Especificación de características de las memorias internas en hwctrl

La utilización de las memorias RAM y ROM que forman parte del bloque funcional de hwctrl es la siguiente:

- La memoria pred_dualport_ram almacena los valores de píxeles predichos, tiene una capacidad de almacenamiento de 576 bytes, todos ellos efectivos.
- La memoria p_err_dualport_ram almacena los valores del error de predicción, tiene una capacidad de almacenamiento de 544 bytes, todos ellos efectivos.
- La memoria aux_cur_block_dualport_ram almacena los valores de píxeles del bloque actual, tiene una capacidad de almacenamiento de 512 bytes, todos ellos efectivos.
- La memoria scan_rom almacena los índices de los valores a ser procesados, tiene una capacidad de almacenamiento de 64 bytes, todos ellos efectivos.

3.4.3. Verificación del módulo hwctrl

El proceso de verificación tiene como objetivo detectar los posibles errores de diseño que puedan ser originados en cada una de las etapas que atraviesa un sistema durante su concepción y desarrollo. No debe confundirse la verificación funcional del diseño, que se realiza durante su implementación, con la fase de test, consistente en la comprobación del circuito físico, cuyo fin es detectar errores producidos durante la fabricación. La importancia de la verificación está justificada por el considerable coste de realizar modificaciones una vez fabricado el circuito, costes de ingeniería no recurrente (NRE).

Durante la verificación funcional se toma como referencia la especificación previamente realizada para el diseño que es comparada con la implementación que se desea validar, esperando que ambas describan exactamente el mismo sistema. En nuestro caso, para hacer la verificación se sustituirán las especificaciones dadas en el algoritmo, por el software de referencia descrito en la sección 3.1 y que constituirá el modelo de referencia

(*golden-reference*) en nuestra verificación. Este modelo se utilizará fundamentalmente para la verificación a nivel de bloque funcional (IP).

La escritura de un entorno de simulación puede llegar a ser tan compleja como la codificación RTL del sistema a verificar (DUV, *Design Under Verification*). Sin embargo, una de las ventajas que presenta su escritura es la posibilidad de no tener que ser sintetizable y, por tanto, no ser descrito a nivel RTL. Para codificar un entorno de simulación, el diseñador debe tener siempre presente las especificaciones de la arquitectura, en las que quedan reflejadas las funciones del diseño y, por tanto, las funciones a verificar. En la Figura 3.18, se muestra la estructura genérica del entorno de simulación empleado, donde se observa cómo el módulo *Testbench* es el que genera los impulsos de entrada al DUV y analiza sus salidas.

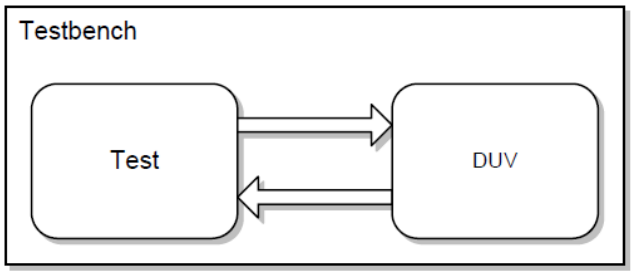

Figura 3.18. Estructura genérica del entorno de simulación

La verificación del módulo hwctrl se ha realizado a nivel de sistema utilizando una metodología denominada *black-box*, tomando como *golden-reference* la implementación en lenguaje ANSI C del algoritmo LCE. Para la correcta verificación del módulo se han extraído, del software de referencia, los bloques de 16 x 16 píxeles de la imagen hiperespectral de entrada.

En cuanto a los módulos funcionalmente independientes extraídos del software de referencia y presentados en la sección 3.2 del presente documento, cabe destacar que han sido previamente verificados haciendo uso del flujo de verificación SCVerify integrado en la herramienta Catapult C. Así pues, se garantiza su correcto desempeño durante su instanciación en el módulo hwctrl.

De este modo, la metodología seguida se basa los siguientes pasos:

1. Simulación mediante el flujo de verificación SCVerify de los módulos extraídos del software de referencia.
2. Simulación del software de referencia con el fin de obtener los estímulos que se aplican a la descripción RTL del bloque funcional y los resultados correctos.

3. Someter al módulo diseñado a los mismos estímulos que se han obtenido a partir del código de referencia

4. En el caso de obtener las mismas salidas se podrá asegurar que la implementación del diseño coincide con lo establecido en el algoritmo LCE, si no es así se itera depurando el diseño implementado para, posteriormente volver al paso 3.

A continuación, se muestran una serie de capturas de la herramienta ModelSim empleada para la verificación y depuración del módulo hwctrl. En la Figura 3.19, se muestra el inicio de la ejecución del algoritmo presentando los primeros estados donde se realizan las transiciones hasta llegar al estado donde se obtiene predicción en dos dimensiones para la primera banda $i = 0$. La Figura 3.20 presenta las distintas fases del algoritmo para bandas posteriores, es decir, $i \neq 0$. Por último, la Figura 3.21 muestra los valores de las señales internas más representativas del algoritmo.

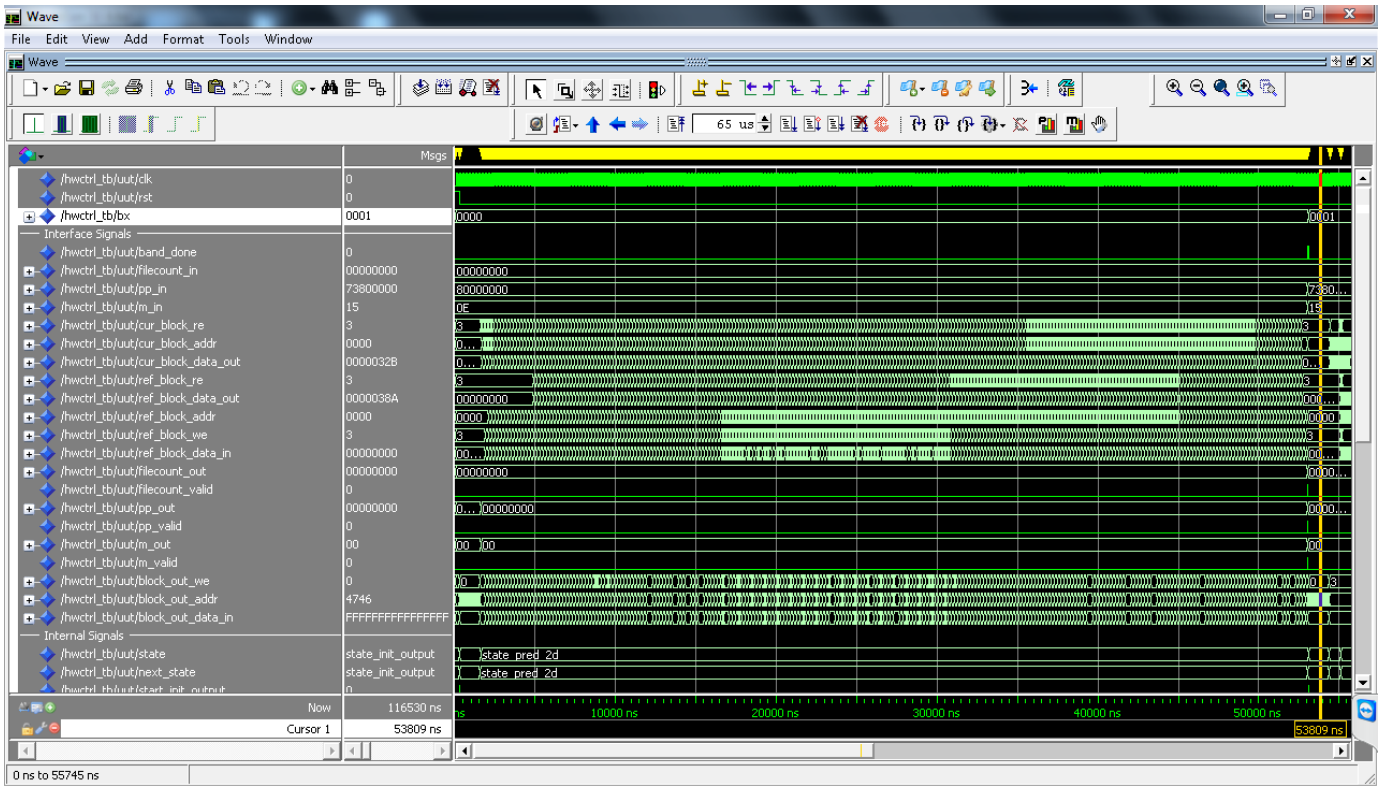

Figura 3.19. Simulación hwctrl banda 0, $i = 0$

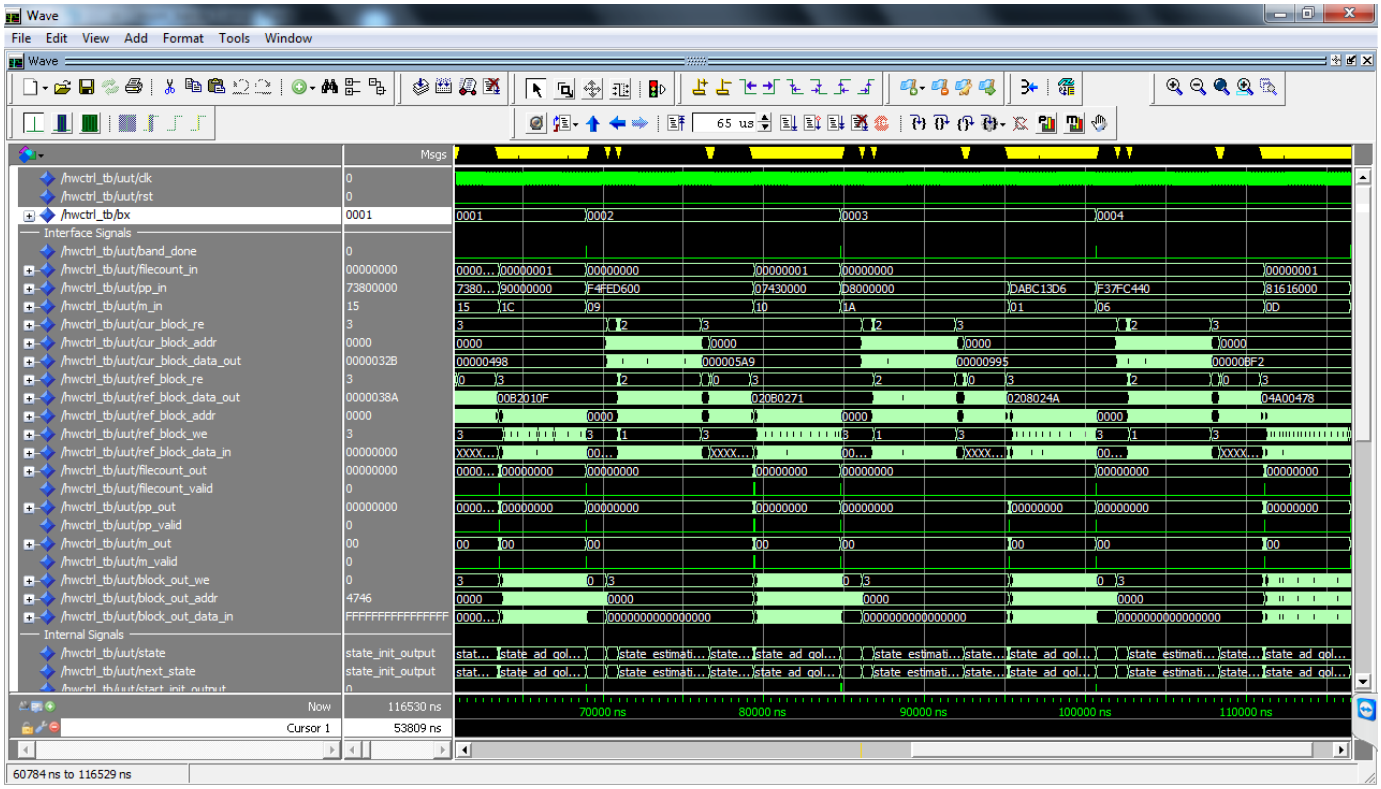

Figura 3.20. Simulación hwctrl resto de bandas, $i \neq 0$

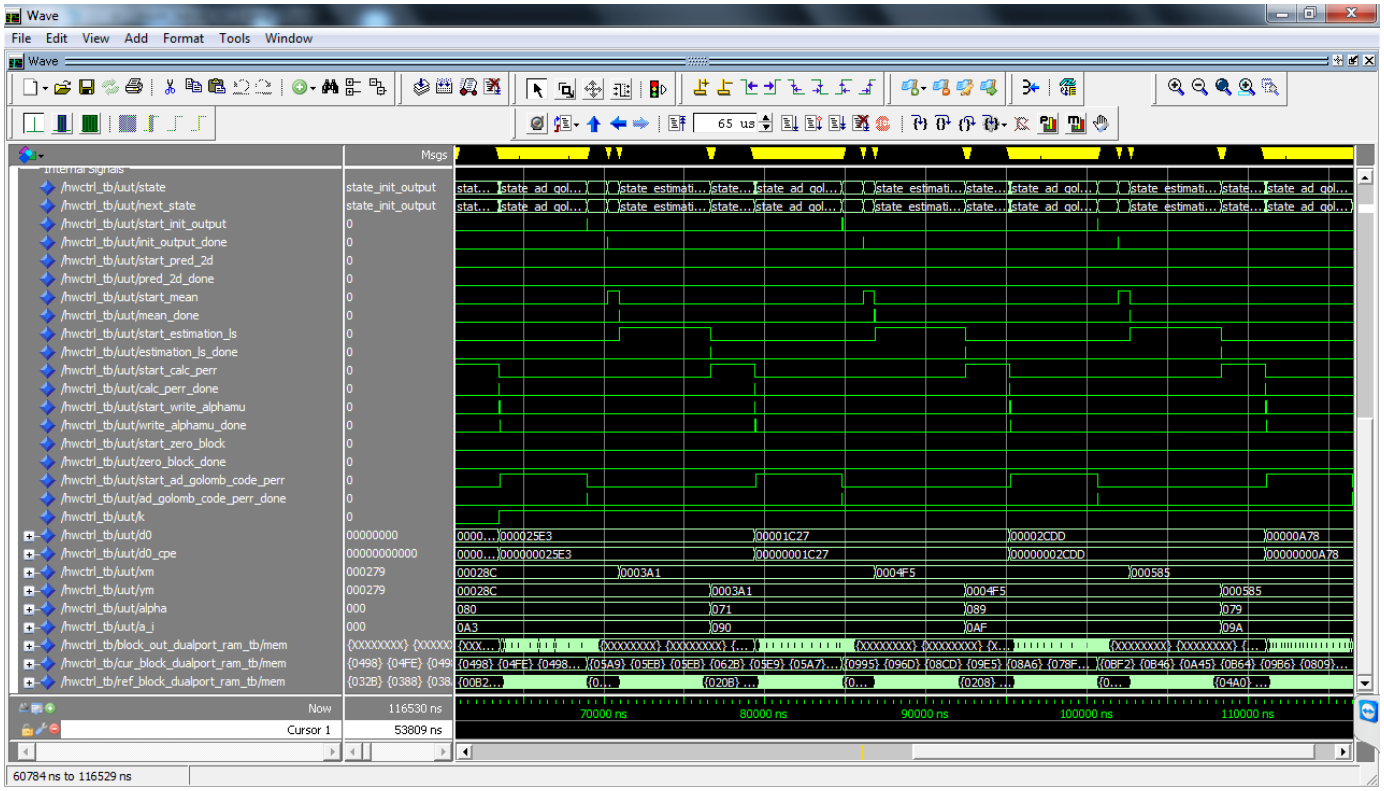

Figura 3.21. Simulación hwctrl señales internas representativas

3.4.3.1. Resultados de la verificación del módulo hwctrl

Una vez visto el procedimiento seguido en la verificación, se va a proceder a analizar sus resultados. En primer lugar hay que mencionar que los objetivos especificados en la metodología de verificación han sido cubiertos en su totalidad. Es decir, se ha comprobado mediante las imágenes hiperespectrales de entrada del sensor AVIRIS, Indian Pines y World Trade Center el correcto desempeño de la implementación para cada una de las bandas. Con ello se puede afirmar que el diseño funcional del bloque funcional hwctrl es correcto.

No obstante, los entornos de verificación disponibles permiten generar cuantas comprobaciones sean necesarias en el caso de que el IP diseñado sea fabricado mediante una librería de células y una tecnología determinada.

4. Resultados y conclusiones

El objetivo principal de este Trabajo Fin de Máster es el desarrollo de un bloque funcional o IP, que implemente el algoritmo de compresión de imágenes hiperespectrales LCE. Este objetivo se ha alcanzado de manera satisfactoria, tal y como se ha mostrado en la sección 3 de este documento. En este sentido, los requisitos de generación de una descripción hardware de la funcionalidad del IP y su comprobación mediante el proceso de verificación estarían cumplidos.

Sin embargo, es necesario comprobar que la descripción ofrecida es sintetizable, aspecto que ya se ha tenido en cuenta en la implementación del diseño a nivel de descripción RTL, y que cumple con los requisitos previos que se han establecido para el bloque funcional. Este capítulo se dedicará a presentar los resultados obtenidos en el proceso de síntesis de la descripción VHDL realizada sobre una FPGA. La FPGA empleada es el modelo XC5VFX100T de la familia de FPGAS Virtex-5 del fabricante XILINX.

4.1. Síntesis sobre FPGA

Para el prototipado del diseño sobre la FPGA, se ha utilizado la herramienta Precision RTL de Mentor Graphics, que proporciona resultados de síntesis para el fabricante escogido. Proporciona soporte para los fabricantes Actel, Altera, Xilinx y Lattice.

La Tabla 4.1 resume los resultados presentados de la síntesis para los módulos extraídos del software de referencia para la FPGA 5VFX100TFF con *speed grade* -1. Como se puede comprobar en ella, la frecuencia máxima, cuando la optimización se realiza en frecuencia, es superior a los 75 MHz fijados como objetivo, por lo que se ha cumplido con el requisito establecido.

Módulo	LUT (*Look-up table*)	*Slices*	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
`adgolomb`	2502 (3,05%)	626 (3,06%)	1 RAMB18 (0,34%)	5 (1,56%)	85,020MHz	11,762ns
`calc_perr`	893 (1,40%)	224 (1,40%)	---	10 (3,91%)	108,707MHz	9,199ns

estimationls	1187 (1,85%)	297 (1,86%)	---	8 (3,13%)	89,055MHz	11.229ns
init_output	71 (0,11%)	18 (0,39%)	---	---	356,761MHz	2,803ns
mean	251 (0,39%)	63 (0,39%)	---	---	181,686MHz	5,504ns
pred_2D	2888 (4,51%)	722 (4,51%)	---	2 (0,78%)	81,433MHz	12,280ns
write_alphamu	503 (0,79%)	126 (0,79%)	---	---	283,046MHz	3.533ns
zero_block	152 (0,22%)	38 (0,22%)	---	---	298,191MHz	3,354ns

Tabla 4.1. Resultados de la síntesis de los módulos de manera independiente

Asimismo, los resultados de la síntesis del bloque funcional hwctrl sobre la FPGA 5VFX100 con *speed grade* -1, se muestran en la Tabla 4.2. Este módulo infiere el resto de módulos además de incorporar lógica de control interna y para el control de las interfaces.

Módulo	LUT (*Look-up table*)	*Slices*	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
hwctrl	7746 (12,1%)	1937 (12,11%)	4 RAMB18 (0,88%)	25 (9,77%)	86,964MHz	11,499ns

Tabla 4.2. Resultados de la síntesis sobre FPGA del bloque funcional hwctrl

Con el fin de proporcionar una comparación, se ha realizado la implementación del algoritmo únicamente empleando la herramienta Catapult C. Por lo tanto, como entrada a la herramienta se introduce el módulo `pred1block.cpp` completamente, así como, los módulos que son dependencia del mismo. Cabe destacar que para que la comparación pueda ser de utilizad se han de fijar en la herramienta idénticos parámetros

arquitecturales que los escogidos para la implementación en el presente Trabajo Fin de Máster. Los resultados obtenidos tras la síntesis se presentan en la Tabla 4.3.

Módulo	LUT (*Look-up table*)	*Slices*	Bloques de RAM	DSP48	Max. Frecuencia	Ruta crítica
`pred1block`	8688 (13,57%)	2172 (13,57%)	4 RAMB18 (0,88%)	17 (6,64%)	75,781MHz	13,196ns

Tabla 4.3. Resultados de la síntesis sobre FPGA del módulo `pred1block` al completo

Como puede observarse se obtienen mejores resultados en cuanto a ocupación y frecuencia de trabajo mediante la aproximación tomada en el presente Trabajo Fin de Máster. Esto se debe a que la implementación realizada por la herramienta no resulta la óptima. De forma cualitativa, se puede afirmar que, cuanta mayor implementación se realice mediante la herramienta menor grado de optimización se obtiene a su salida. Por otra parte, cabe destacar como aspecto relevante la reducción del tiempo de desarrollo mediante el empleo de la herramienta Catapult C frente a un diseño totalmente realizado por un diseñador hardware.

La Tabla 4.4, compara los resultados de obtenidos en este trabajo con los resultados de la implementación de [32]. A pesar de su mayor complejidad, los resultados de implementación obtenidos son comparables a los resultados obtenidos en [32]. La principal diferencia apreciable se encuentra en el número de DSP usados, que resulta mayor en la implementación del algoritmo LCE. Este hecho, se debe, al mayor número de operaciones matemáticas que componen este algoritmo con pérdidas. Destacable resulta el número de memoria RAM empleada en la implementación del algoritmo LCE.

	Algoritmo LCE		ESA sin pérdidas [32]
	4VLX200 (-11)	5VFX100 (-1)	4VLX200 (-11)
LUT	9283	7746	10306
Slices	4642	1937	6312
RAM Blocks	4	4	9
DSP48	25	25	9
Max. Frequency	75,844 MHz	86,964 MHz	81 MHz

Tabla 4.4. Comparativa implementación algoritmo LCE frente a [32]

4.2. Estrategias de optimización

Con el fin de evaluar las prestaciones de la implementación hardware del algoritmo desarrollada, se establece como parámetro objeto, el cálculo de muestras por segundo a la salida del codificador.

Para ello, se examina, a nivel de simulación, el módulo controlador hardware (hwctrl) hallándose el número de muestras por ciclo alcanzado. La Tabla 4.5 muestra tanto el número de ciclos requerido para la predicción de la primera banda como para el resto de bandas. Además, se muestra la latencia del diseño para cada banda. Dado que para la primera banda, $i = 0$, se realiza la predicción en dos dimensiones, el número de ciclos para esta banda difiere con respecto del número de ciclos requeridos para el cálculo del resto de bandas, $i \neq 0$.

Banda	Latencia	Muestras por ciclo
Primera banda (predicción 2D, $i = 0$)	2640 ciclos	1 muestra / 10 ciclos
Resto de bandas ($i \neq 0$)	793 ciclos	1 muestra / 3 ciclos

Tabla 4.5. Latencia y número de muestras por segundo obtenidas

La Tabla 4.6, para cada bloque de 16 x 16 muestras, presenta el número de muestras por ciclo que se alcanza como media para las imágenes de los sensores AIRS, AVIRIS y MODIS. Asimismo, se presenta el número de muestras por segundo que se alcanzable tras el emplazamiento y ruteo a la máxima frecuencia de operación obtenida.

Sensor	Número de bandas	Muestras por ciclo (media)	Muestras por segundo (86,964MHz)
AIRS	1501	1 muestra / 3,114 ciclos	$27{,}927 \times 10^6$ muestras/s
AVIRIS	224	1 muestra / 3,142 ciclos	$27{,}677 \times 10^6$ muestras/s
MODIS	17	1 muestra / 3,535 ciclos	$24{,}6 \times 10^6$ muestras/s

Tabla 4.6. Muestras por ciclo media para imágenes de los sensores AIRS, AVIRIS y MODIS

El límite teórico alcanzable para la implementación es de 1 muestra por ciclo de reloj. Con el fin de aproximar el diseño a este límite teórico se puede paralelizar el diseño interfaz a interfaz, variable a variable, para que, en el momento que todas las variables necesarias a la entrada de un módulo estén disponibles activar el procesado del mismo sin esperar a la finalización de la ejecución del módulo anterior.

Asimismo, no todas las variables de entrada a un módulo son necesarias al inicio de la ejecución del mismo con lo que es posible adelantar la activación del mismo para que cuando se lea esta variable de entrada ya se encuentre disponible.

4.3. Conclusiones

En el presente Trabajo Fin de Máster se ha desarrollado un módulo que implementa el algoritmo de compresión de imágenes hiperespectrales LCE. Este núcleo IP se ha diseñado con el fin de realizar una caracterización del algoritmo en cuanto a recursos necesarios para su desempeño empleando una aproximación mediante el empleo de la herramienta Catapult C.

Como paso natural a la finalización del presente trabajo se presenta la optimización del algoritmo mediante las estrategias presentadas en la sección anterior con el objeto de aproximar la implementación al objetivo de 1 muestra/ciclo. Asimismo, resulta de interés el estudio y la implementación del algoritmo en su totalidad.

A lo largo de este proyecto se han seguido los siguientes pasos:

- Se estudiaron las bases de la compresión de imágenes hiperespectrales, adquiriéndose conocimientos sobre el funcionamiento de los diferentes tipos de compresión. Se examinó con un mayor grado de detalle el algoritmo LCE implementado.
- Se analizó el código de referencia ANSI C del algoritmo, para determinar y conocer las funciones que lo componen.
- Se dividió el código de referencia y se estableció la funcionalidad de cada uno de los diferentes bloques funcionales que lo conforman. A continuación, se procedió a su implementación y verificación mediante la herramienta Catapult C. Para ello, se impusieron las restricciones tecnológicas y arquitecturales requeridas.
- Se diseño un bloque funcional para la conexión de cada uno de los módulos implementados elaborándose la descripción en código VHDL de la arquitectura diseñada, verificándose para comprobar su correcto funcionamiento.

- Se analizaron los resultados de área, frecuencia y transferencia de datos que se obtienen al sintetizar el diseño sobre una FPGA XC5VFX100T de Xilinx, comprobándose que se cumplen los requisitos fijados en cuanto frecuencia, frecuencia mínima de 75 MHz.

5. Bibliografía

[1] B. Penna, T. Tillo, E. Magli, and G. Olmo, “Transform coding techniques for lossy hyperspectral data compression,” IEEE Transactions on Geoscience and Remote Sensing, vol. 45, no. 5, pp. 1408–1421, May 2007.

[2] R.E. Roger and M.C. Cavenor, “Lossless compression of AVIRIS images,” IEEE Transactions on Image Processing, vol. 5, no. 5, pp. 713–719, May 1996.

[3] J. Mielikainen, A. Kaarna, and P. Toivanen, “Lossless hyperspectral image compression via linear prediction,” Proc. SPIE, vol. 4725, 2002.

[4] B. Aiazzi, L. Alparone, and S. Baronti, “Near-lossless compression of 3-D optical data,” IEEE Transactions on Geoscience and Remote Sensing, vol. 39, no. 11, pp. 2547–2557, Nov. 2001.

[5] Reichel, J., Menegaz, G., Nadenau, M.J., Kunt, M.: Integer wavelet transform for embedded lossy to lossless image compression. IEEE Trans. Image Processing 10(3), 383–392 (2001)

[6] Said, A., Pearlman, W.A.: An image multiresolution representation for lossless and lossy compression. IEEE Trans. Image Processing 5(9), 1303–1310 (1996)

[7] Lei Wang, Jiaji Wu, Licheng Jiao, and Guangming Shi. Lossy-to-lossless hyperspectral image compression based on multiplierless reversible integer TDLT/KLT. IEEE Geosci. Remote Sens. Lett., 6(3):587–591, July 2009.

[8] Lossless Multispectral & Hyperspectral image compression. Recommended standard CCSDS 123.0-B-1. Blue Book. May 2012

[9] Du and J. E. Fowler, “Hyperspectral image compression using JPEG2000 and principal component analysis”, IEEE Geosci. Remote Sen. Lett., vol. 4, no. 2, pp. 201-205, 2007.

[10] A. Abrardo, M. Barni, E. Magli, “Low-complexity predictive lossy compression of hyperespectral and ultraspectral images”, in Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE Int. Conf. on, May 2011, pp. 797-800.

[11] L. Santos, LCE GPU Implementation Report. ESA Standard Document.

[12] Catapult C. http://www.calypto.com/en/products/catapult/overview (último acceso: 29 de noviembre de 2012).

[13] M. Zortea, A. Plaza. A quantitative and comparative analysis of different implementations of N-FINDR: A fast endmember extraction algorithm. pp. 787-791, s.l.: IEEE Geoscience and Remote Sensing Letters, 2009.

[14] J. Plaza, A. Plaza, D. Valencia y A. Paz, "Massively Parallel Processing of Hyperspectral Images", SPIE Optics and Photonics, Satellite Data Compression, Communication, and Processing Conference, San Diego, CA, 2009.

[15] A. Plaza, J. Plaza, S. Sánchez y A. Paz, "Optimizing a Hyperspectral Image Processing Chain Using Heterogeneous and GPU-Based Parallel Computing Architectures", 9th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'09), Gijón, 2009.

[16] R.E. Roger and M.C. Cavenor, "Lossless compression of AVIRIS images," IEEE Transactions on Image Processing, vol. 5, no. 5, pp. 713–719, May 1996.

[17] J. Mielikainen, A. Kaarna, and P. Toivanen, "Lossless hyperspectral image compression via linear prediction," Proc. SPIE, vol. 4725, 2002.

[18] B. Aiazzi, P. Alba, L. Alparone, and S. Baronti, "Lossless compression of multi/hyperspectral imagery based on a 3-D fuzzy prediction," IEEE Transactions on Geoscience and Remote Sensing, vol. 37, no. 5, pp. 2287–2294, Sept. 1999.

[19] S.K. Jain and D.A. Adjeroh, "Edge-based prediction for lossless compression of hyperspectral images," Proc. IEEE Data Compression Conference, 2007.

[20] B. Aiazzi, L. Alparone, and S. Baronti, "Near-lossless compression of 3-D optical data," IEEE Transactions on Geoscience and Remote Sensing, vol. 39, no. 11, pp. 2547–2557, Nov. 2001.

[21] B. Aiazzi, L. Alparone, S. Baronti, and C. Lastri, "Crisp and fuzzy adaptive spectral predictions for lossless and near-lossless compression of hyperspectral imagery," IEEE Geoscience and Remote Sensing Letters, vol. 4, no. 4, pp. 532–536, Oct. 2007.

[22] M.J. Ryan and J.F. Arnold, "The lossless compression of AVIRIS images by vector quantization," IEEE Transactions on Geoscience and Remote Sensing, vol. 35, no. 3, pp. 546–550, May 1997

[23] J. Mielikainen and P. Toivanen, Clustered DPCM for the lossless compression of hyperspectral images, IEEE Transactions on Geoscience and Remote Sensing, vol. 41, no. 12, pp. 2943–2946, Dec. 2003.

[24] E Magli, G Olmo, and E. Quacchio, “Optimized onboard lossless and near-lossless compression of hyperspectral data using CALIC,” IEEE Geoscience and Remote Sensing Letters, vol. 1, no. 1, pp. 21–25, Jan. 2004

[25] Lossless Multispectral & Hyperspectral Image Compression. Draft Report Concerning Space Data System Standards, CCSDS 120.2-G-0. Draft Green Book. Under development

[26] D. S. Taubman and M. W. Marcellin, JPEG2000: Image Compression Fundamentals, Standards, and Practice, Kluwer, 2001.

[27] A. Karami, M. Yazdi, and G. Mercier, “Compression of hyperspectral images using discrete wavelet transform and tucker decomposition”, Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal, vol. 5, no. 2, pp. 444-450, April 2012.

[28] L. Santos, S. Lopez, G. M. Callico, J. F. López, and R. Sarmiento, “Performance evaluation of the H.264/AVC video coding standard for lossy hyperspectral image compression”, Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal, vol. 5, no. 99, pp. 1, 2011.

[29] G. J. Sullivan, “On embedded scalar quantization”, in Acoustics, Speech, and Signal Processing, 2004. Proceedings, ICASSP ’04, IEEE International Conference on, May 2004, vol. 4, pp. iv-605.

[30] Lossy Compression for Exomars (LCE), Software Design Document. Reference: LCE-SDD-POLITO-002. September 2011.

[31] IEEE Standard VHDL Language Reference Manual. 1076-2008. September 2008.

[32] Lossless Data Compression, Green Book, CCSDS 120.0-G-1, May 1997.