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

PROTOTIPADO SOBRE PLATAFORMA FPGA DE UN DEBLOCKING FILTER PARA H.264/SVC

Autor: Omar Espino Santana
Tutor(es): Antonio Núñez Ordóñez
Pedro Pérez Carballo
Fecha: Julio de 2013

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

PROTOTIPADO SOBRE PLATAFORMA FPGA DE UN DEBLOCKING FILTER PARA H.264/SVC

HOJA DE FIRMAS

Alumno:	Omar Espino Santana	Fdo.:
Tutor:	Antonio Núñez Ordóñez	Fdo.:
Tutor:	Pedro Pérez Carballo	Fdo.:

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

PROTOTIPADO SOBRE PLATAFORMA FPGA DE UN DEBLOCKING FILTER PARA H.264/SVC

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Fdo.:

Secretario: Fdo.:

Vocal: Fdo.:

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

Índice de figuras

Índice de tablas

Capítulo 1: Introducción

1 Antecedentes

En la actualidad, la grabación y reproducción de vídeo son procesos cada vez más usuales. Además, debido al avance de los sistemas electrónicos integrados en un único chip, en los últimos años ha habido un fuerte esfuerzo por trasladar estos procesos multimedia a esta nueva tecnología. Con este fin nacen los actuales llamados estándares de codificación de vídeo, que hoy en día se utilizan en una gran variedad de dispositivos como pueden ser ordenadores de sobremesa, móviles, videoconsolas, cámaras de fotos y de vídeo, etc.

Tanto en el proceso de codificar vídeo como en el de decodificar, existen restricciones temporales que deben cumplirse. Los sistemas controlados por este tipo de restricciones se conocen como Sistemas en Tiempo Real (STR). El elemento condicionante suele ser el tiempo máximo de ejecución de un algoritmo que permita tiempos de respuesta del sistema lo más bajos posibles y siempre por debajo de un límite impuesto por el entorno de la aplicación.

La aceleración hardware es una posible solución a las restricciones anteriormente indicadas. Este enfoque implica la realización de una partición de la aplicación software a ejecutar, separándola en módulos, y diseñar aceleradores hardware para procesar aquellos módulos críticos que representen un mayor coste computacional [1].

Las FPGAs (Field Programmable Gate Array) representan un recurso adecuado a utilizar para realizar los bloques descritos. Por su facilidad a la hora de reprogramar y su bajo coste para un bajo número de unidades, serán el principal medio a usar [1].

1.1 Sistemas en Tiempo Real (STR)

Un sistema en tiempo real (STR) es aquel sistema digital que interactúa activamente con su entorno con una dinámica conocida entre sus entradas, salidas y restricciones temporales, para darle un correcto funcionamiento de acuerdo con los conceptos de predictibilidad, estabilidad, controlabilidad y alcanzabilidad [2].

El correcto funcionamiento de estos sistemas depende no sólo del resultado lógico que se genere, sino que también depende del tiempo utilizado en producir dicho resultado. De esta forma, atendiendo al entorno en el que se desarrolle la aplicación, existirá un tiempo límite que el sistema deberá cumplir para que sea exitoso, considerándose todos aquellos resultados obtenidos después del límite como fallos del sistema aun cuando estos sean los valores esperados. Se puede hablar de dos tipos de STR: Hard RT y Soft RT. Para el primero de los casos la restricción es completa, con tiempos de respuesta estrictos. Para el segundo, los tiempos de procesamiento pueden estar comprendidos en un determino margen pero no son críticos cuando la salida sigue una determinada cadencia, aunque con cierta latencia inicial.

Un ejemplo típico de sistema en tiempo real tipo Hard RT es el antibloqueo de las ruedas de un automóvil (ABS). En este caso el tiempo límite en el que debe operar es aquel en el que las ruedas deban liberarse antes de que se bloqueen. Si el sistema libera las ruedas una vez estas ya se han bloqueado, habrá fallado [3]. Un ejemplo de un sistema Soft RT puede ser un sistema de transmisión de vídeo, que es precisamente el caso que nos ocupa en este TFM.

1.2 Servicios multimedia

Debido a que, como ya se comentó anteriormente, es cada vez más usual que dispositivos electrónicos portátiles con diferentes objetivos, incluyan entre su funcionalidad la posibilidad de reproducir vídeo digital, ha existido en los últimos años y se mantiene en la actualidad, un gran

esfuerzo por mejorar las implementaciones de sistemas multimedia de bajo consumo y altas prestaciones.

La decodificación del estándar H.264 está actualmente implementada en una gran cantidad de arquitecturas, debido principalmente a la gran cantidad de videos codificados en este formato. En mayo de 2010, más del 66% de los vídeos en internet estaban codificados en H.264 [4].

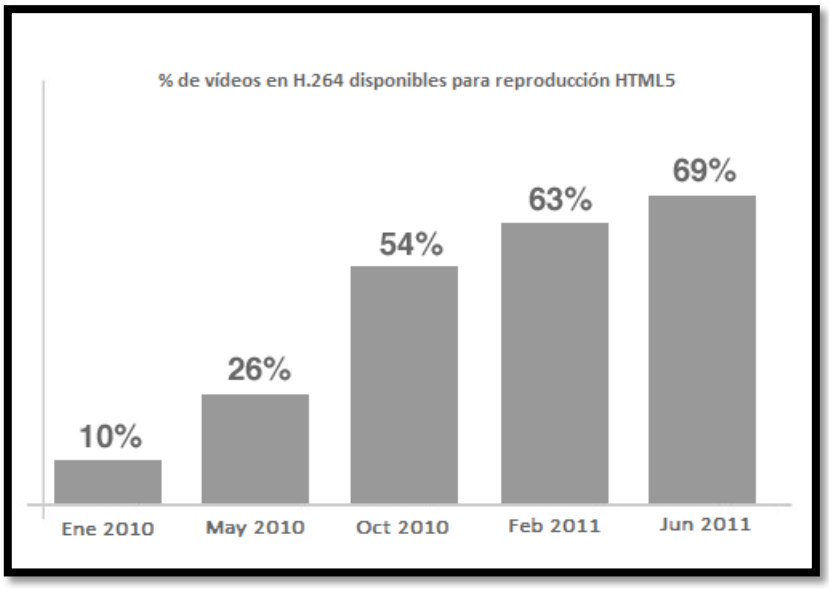

Figura 1. Porcentaje de vídeos disponibles en H.264 [4].

1.3 Estándar H.264

Puesto que el objetivo principal de este TFM es la implementación hardware de un módulo del decodificador de vídeo H.264 en su versión para vídeo escalable, parece oportuno presentar brevemente una historia de los *codecs* de vídeo hasta el actual H.264.

El comienzo por la carrera en la codificación de vídeo digital nació en la década de los 80, mediante la creación del estándar H.120, antecesor del H.261, considerado el primer estándar de vídeo digital. A partir de este, se ha ido implementándo mejoras dando lugar a nuevos estándares como son el MPEG-1 Parte 2 [5], H.262/MPEG-2 Parte 2 [6], H.263/MPEG-4 Parte 2 [7], H.264/MPEG-4 Parte 10 [8] y el más reciente H.265/HEVC [9]. En el presente TFM se trabajará con el estándar H.264, en particular con el anexo G, el cual presenta modificaciones sobre el estándar H.264 original con el fin de implementar vídeo escalable (referido a partir de aquí como estándar H.264/SVC).

H.264 es una norma que define un *codec* de vídeo de alta compresión, desarrollada conjuntamente por el ITU-T Video Coding Experts Group (VCEG) y el ISO/IEC Moving Picture Experts Group (MPEG). La intención del proyecto H.264/AVC fue la de crear un estándar capaz de proporcionar una buena calidad de imagen con tasas binarias notablemente inferiores a los estándares previos (MPEG-2, H.263/MPEG-4 parte 2). Esta mejora se representa en la Figura 2 donde se observa como para una misma tasa binaria se consiguen mejores prestaciones de resolución y/o calidad de la imagen, o como para una misma calidad y resolución se necesita una menor tasa binaria frente a estándares anteriores [10].

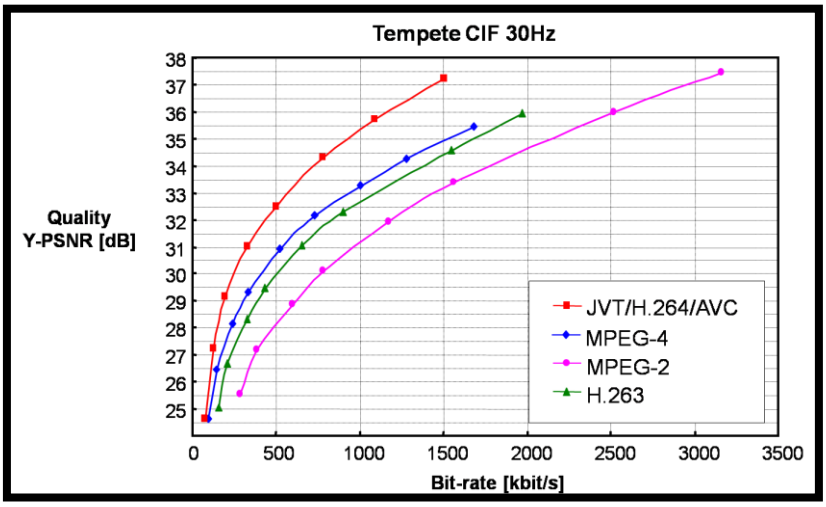

Figura 2. Comparación entre H.264 y estándares anteriores [10].

Para disminuir la cantidad de información a la hora de codificar el vídeo, H.264 se basa en dos tipos de predicciones aprovechando la redundancia de información temporal y espacial que existen en los vídeos. En cuando a la temporal, se basa en que las imágenes consecutivas que producen el movimiento en general contendrán muchas partes de la imagen iguales pero en distinta posición de la misma si se han movido, o en la misma posición si han permanecido estáticas a lo largo del tiempo. Esta cualidad es utilizada por H.264 para, en lugar de enviar cada imagen codificada individualmente, enviar lo que se conoce como un vector de movimiento por cada bloque de la imagen, que indique la dirección y distancia del movimiento de dicho bloque entre dos imágenes cercanas en el tiempo. Cuando esta predicción no tiene el suficiente éxito, se aprovecha la redundancia espacial, es decir, que porciones de la imagen cercanas entre ellas tienen una alta probabilidad de tener los mismos valores, por lo que se podrá codificar un bloque de la misma enviando únicamente la diferencia con alguno de sus bloques vecinos. Estos dos tipos de predicciones se les conocen como INTER e INTRA respectivamente.

Sin embargo, el diseño de la arquitectura tanto del codificador como del decodificador H.264 es complejo. Además de los altos requisitos de cómputo y de acceso a memoria, el camino de codificación es largo, incluyendo predicción, reconstrucción y decodificación de la entropía. Comparándola con otros estándares, la complejidad hardware se incrementa hasta el doble en el caso del decodificador.

1.4 Codificación de vídeo escalable

Scalable Video Coding (SVC) es el nombre del anexo G del estándar H.264 [8], el cual describe un estándar para la compresión de vídeo escalable. Este tipo de codificación se basa en la generación de un flujo de bits (*bitstream*) en el que se encuentren codificadas conjuntamente diferentes capas de vídeo, es decir, diferentes versiones del mismo vídeo con distintos parámetros de calidad, resolución y tasa de imágenes. Este tipo de codificación busca que, la tasa binaria necesaria sea inferior a la suma de generar tantos flujos como capas con codificación independiente. Esta reducción se consigue debido a la clara redundancia al tratarse del mismo vídeo usado como fuente [11].

La necesidad de codificar un vídeo fuente con diferentes parámetros viene dada por la gran diversidad de dispositivos de reproducción existentes, cada uno de ellos con una capacidad de cómputo y visualización diferentes. Por ejemplo, las resoluciones de pantalla para la visualización del vídeo son las que obligan a codificar el vídeo con diferentes resoluciones. Las diferentes capacidades de cómputo de estos dispositivos también restringen la tasa de imágenes y la calidad de las mismas.

Existen multitud de aplicaciones en donde la escalabilidad es explotada, como pueden ser videoconferencias, videovigilancia, difusión de vídeo, etc. En todas estas, el suministrador del vídeo lo codifica con diferentes parámetros con el fin de hacer este disponible a una gran cantidad de dispositivos que deberán decodificar el mismo, usando para ello, en función de sus posibilidades, una u otra capa de mejora.

Existen tres parámetros configurables a la hora de generar las capas de mejora en SVC que son [11]:

- Escalabilidad espacial. Está asociada a la resolución del vídeo decodificado. En función del dispositivo reproductor y de sus características de pantalla, se podrán utilizar diferentes resoluciones en el codificador.
- Escalabilidad temporal. En este caso está asociada a la tasa de imágenes por segundo. Si el dispositivo que debe decodificar el vídeo tiene poca capacidad de

cómputo, tendrá a su vez limitaciones a la hora de decodificar un gran número de imágenes en un corto espacio de tiempo, por lo que usará las capas de menor tasa de bits del *bitstream.* Sin embargo, dispositivos con grandes capacidades aprovecharán este hecho para aplicar todas las capas de mejora y poder optar a mejores tasas de imágenes.

- Escalabilidad de calidad. Este tercer parámetro está asociado al uso de valores del parámetro de cuantización con el fin de mejorar la calidad de la imagen en capas superiores. El uso de parámetros de cuantización más bajo produce *bitstream* más grandes y una necesidad más alta de cómputo.

En la definición de las capas de mejora de SVC se crean escalabilidades combinadas, es decir, que en cada capa pueden verse modificados varios parámetros de los anteriormente descritos. Por ejemplo, en la Figura 3 se puede observar una representación de un *bitstream* en el que existe una capa base y una capa de mejora. En la base, la resolución es más baja, representado por rectángulos de dimensiones inferiores, además de que el número de rectángulos en el tiempo es inferior. La capa de mejora está representada por rectángulos más grande haciendo alusión a resoluciones más altas, así como un mayor número de imágenes explotando así la escalabilidad temporal.

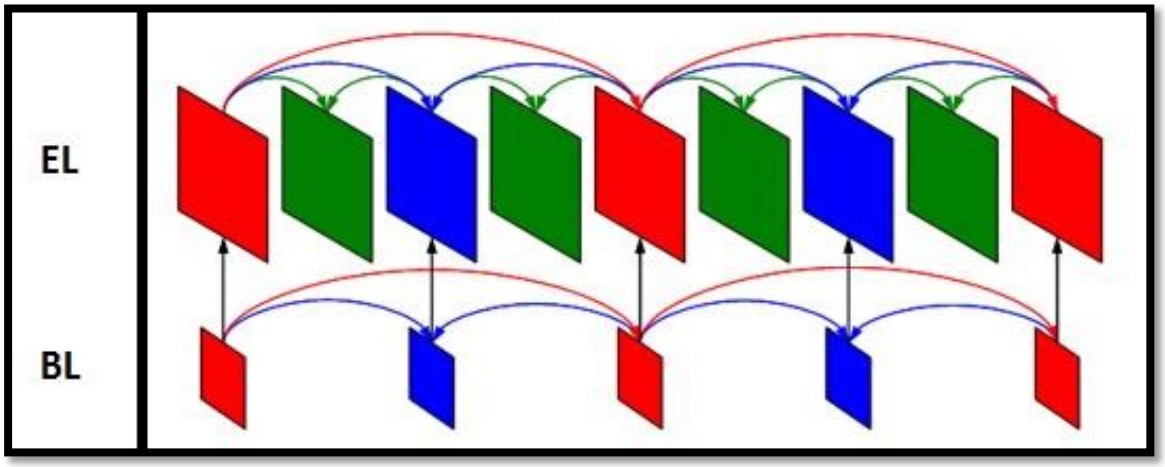

Figura 3. Ejemplo de vídeo escalable con dos capas. (EL = Enhancement Layer, BL = Base Layer).

En este marco de investigación surge el proyecto PCCMuTe (Power Consumption Control in Multimedia Terminals), llevado a cabo por la división SICAD del Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, y el Grupo de Diseño de Electrónica y Microelectrónica de la Universidad Politécnica de Madrid.

1.5 Plataformas FPGA

Con el fin de acelerar una aplicación para cumplir restricciones de tiempo real, por ejemplo, la aceleración hardware es una posible solución ya que hace uso del paralelismo intrínseco del hardware. Esta aceleración puede realizarse mediante diferentes soluciones arquitecturales.

Una posible solución se encuentra en el diseño de circuitos integrados sobre FPGAs (Field Programmable Gate Array), dispositivos semiconductores que contienen bloques de lógica donde su funcionalidad e interconexión son programables. Al contrario de los procesadores de propósito general (GPP), se tratan de soluciones diseñadas de forma optimizada para una aplicación concreta, ganando eficiencia a costa de disminuir la programabilidad del sistema. Otra característica fundamental es la posibilidad de concurrencia real entre tareas, al incluir en el circuito integrado diferentes módulos que realizan distintas operaciones en el mismo instante temporal [12].

En cuanto a la programación, para las aplicaciones a ejecutar sobre un GPP se pueden usar lenguajes de alto nivel como Java, Ada, Basic, C++, etc. Dichos algoritmos serán compilados para transformarlos a lenguaje máquina, es decir, un conjunto de instrucciones entendibles por el procesador.

En las FPGAs, el flujo de desarrollo tiene otra filosofía. Para el diseño de estos sistemas se usan normalmente lenguajes de descripción hardware (HDL) tales como Verilog o VHDL. Los bloques descritos en estos lenguajes son posteriormente sintetizados haciendo uso de las librerías de los fabricantes. En la actualidad se ha incrementado el nivel de abstracción hasta usar lenguajes de descripción de sistemas como puede ser C/C++ o SystemC [13], que tras una síntesis de alto nivel, transforma la descripción algorítmica en una microarquitectura a nivel RTL, que luego es optimizada e implementada siguiendo flujos tradicionales de síntesis [14].

En la actualidad existen ciertas familias de FPGAs que disponen, además de la lógica programable habitual, otros recursos hardware como pueden ser microprocesadores, unidades de coma flotante, controladoras MAC para Ethernet, etc. Este tipo de recursos permite el diseño de SoCs (System-on-Chip) de forma eficiente, al utilizar la lógica programable para la implementación de periféricos hardware así como de coprocesadores hardware para aceleración de ciertas tareas, y el microprocesador para las tareas software del mismo.

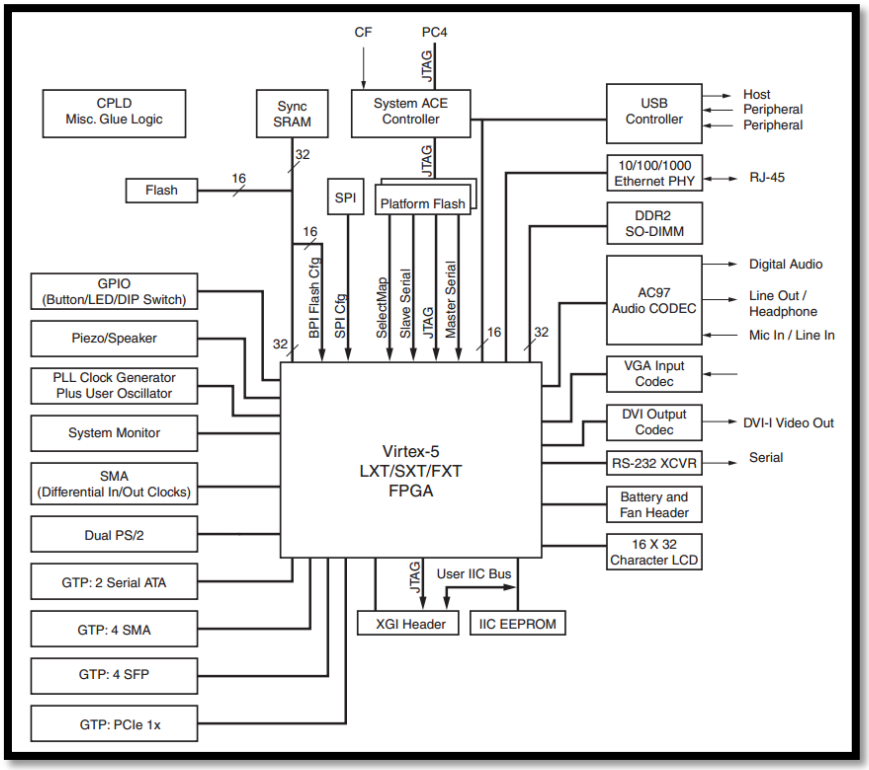

Figura 4. Diagrama de bloques de la placa ML507.

2 Objetivos

El objetivo principal de este TFM es la implementación de un filtro de bucle adaptativo (en inglés *deblocking filter – DF*) de un decodificador H.264/SVC sobre una plataforma FPGA. Este será implementado como parte de un Sistema en Chip (en inglés *System-on-Chip* – SoC) mediante un coprocesador hardware. El resto del SoC se encargará de las tareas de control y representación.

Para esta tarea se definirá un flujo de diseño partiendo de la descripción a nivel algorítmico en SystemC y realizando las tareas de síntesis necesarias hasta alcanzar una versión implementada en FPGA. Además, con el fin de validar el sistema, se diseñará un SoC, que integra el bloque implementado, que envía las tramas de entrada y representa en una pantalla VGA las imágenes tras su filtrado. La descripción del DF así como las tareas asociadas a este TFM forman parte del proyecto PCCMUTE llevado a cabo por la División de Sistemas Industriales y CAD del IUMA.

Las tareas necesarias para alcanzar el objetivo propuesto se describen a continuación:

a. Estudio del funcionamiento del bloque *DF*.
b. Análisis de la arquitectura de la descripción objeto de la implementación.

c. Definición de la metodología de diseño que se llevará a cabo con el fin de alcanzar la versión implementada.
d. Realizar la síntesis del bloque en cuestión, incluyendo las verificaciones funcionales en SystemC así como en RTL.
e. Integración del *DF* en el SoC y validación del mismo.
f. Evaluación de los resultados en cuanto a ocupación y prestaciones del diseño.

3 Peticionario

Actúa como peticionario de este TFM la División de Sistemas Industriales y CAD del Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, en el marco de las líneas de investigación promovidas por la citada división.

Por otro lado, la realización de un Trabajo Fin de Máster es requisito indispensable para la obtención del título de Máster en Tecnologías de Telecomunicación por el Instituto Universitario de Microelectrónica Aplicada perteneciente a la Universidad de Las Palmas de Gran Canaria.

4 Estructura del documento

El presente documento se divide en seis capítulos y dos anexos en los cuales se describe el trabajo realizado:

- **Capítulo 1:** Introducción. Se detallan los antecedentes, objetivos, peticionario y estructura del documento.
- **Capítulo 2:** Dominio de aplicación. Se expone en este capítulo el dominio de aplicación de este trabajo. Se da una descripción del estándar de decodificación H.264, de su anexo G dedicado al vídeo escalable y más específicamente del *DF* utilizado, analizando además su arquitectura interna. Se describe así mismo la situación de partida de este trabajo.
- **Capítulo 3:** Metodología de diseño. Breve introducción a los elementos que forman parte, directa o indirecta, del desarrollo del trabajo, como las herramientas *software* utilizadas, tecnologías y el flujo de diseño seguido.
- **Capítulo 4:** Síntesis del diseño propuesto. Se describe la etapa de síntesis así como los resultados obtenidos y su verificación.

- **Capítulo 5:** Adaptación y validación. Se presenta el diseño de un circuito de adaptación de interfaces así como de la plataforma donde integrar el *DF* para su validación.
- **Capítulo 6:** Conclusiones y líneas futuras. Se exponen las conclusiones generales del presente TFM así como las líneas futuras que nacen de la realización del mismo.

Capítulo 2: Dominio de aplicación

1 Introducción

En este capítulo se detallarán los detalles referentes al diseño con el que se trabajará en el desarrollo de este TFM. En particular se definirán de forma breve los bloques de los que consta un decodificador de vídeo H.264, prestando especial interés al bloque *DF* que es con el que se trabajará en este trabajo. Además de esto se presentará las novedades introducidas en el anexo G de la recomendación de H.264, en el que se explicará la utilidad del vídeo escalable así como sus ventajas en las aplicaciones a las que va destinada.

Para finalizar, se analizará la arquitectura interna de la descripción funcional en SystemC de la que se parte en este TFM, así como a los protocolos de comunicación existentes en los puertos de entrada y salida del bloque.

2 Estándar H.264

El estándar de vídeo H.264 nace de la necesidad de conseguir mayores tasas de compresión en la codificación de vídeo digital frente a los anteriores estándares existentes, con el fin de dar soporte a aplicaciones de transmisión de vídeo en tiempo real como puede ser la difusión televisiva o el *streaming* de vídeo a través de redes de datos.

El estándar H.264/AVC cuenta con los mismos bloques funcionales que sus antecesores, adoptando también un algoritmo híbrido de predicción y transformación para la reducción de la correlación espacial y de la señal residual, control de la tasa binaria, predicción por compensación de movimiento para reducir la redundancia temporal, así como codificación de entropía para reducir la correlación estadística. No obstante, lo que causa que este estándar proporcione una mayor eficiencia en la codificación es el modo en que opera cada bloque funcional. Por ejemplo, H.264/AVC incluye [15]:

- Predicción intra-fotograma (INTRA), característica única de este estándar.
- Transformación por bloques de muestras 4x4, cuyos coeficientes transformados resultan enteros.
- Referencia múltiple para predicción temporal.
- Tamaño variable de los macrobloques a comprimir.
- Precisión de un cuarto de píxel para la compensación de movimiento.
- Filtro de bucle adaptativo *(DF)*, objeto de aceleración de este TFM.
- Codificador de entropía mejorado entre otros.

2.1 Arquitectura de un decodificador H.264 genérico

En la Figura 5 se muestra el diagrama de bloques simplificado de un decodificador H.264. A continuación se explicará brevemente cada uno de los bloques indicados.

2.1.1 Decodificador de longitud variable adaptativo

Las siglas CAVLD responden a *Context-Adaptive Variable Length Decoder*, es decir, es el decodificador de la codificación de longitud variable realizada en el decodificador, y es por lo tanto, el primer bloque del bucle de decodificado.

2.1.2 Transformación y cuantización inversas IQ/IDCT

Corresponde al bloque de cuantización inversa y transformada del coseno discreta inversa. Es conocido comúnmente como IQ/IDCT por su nombre en inglés.

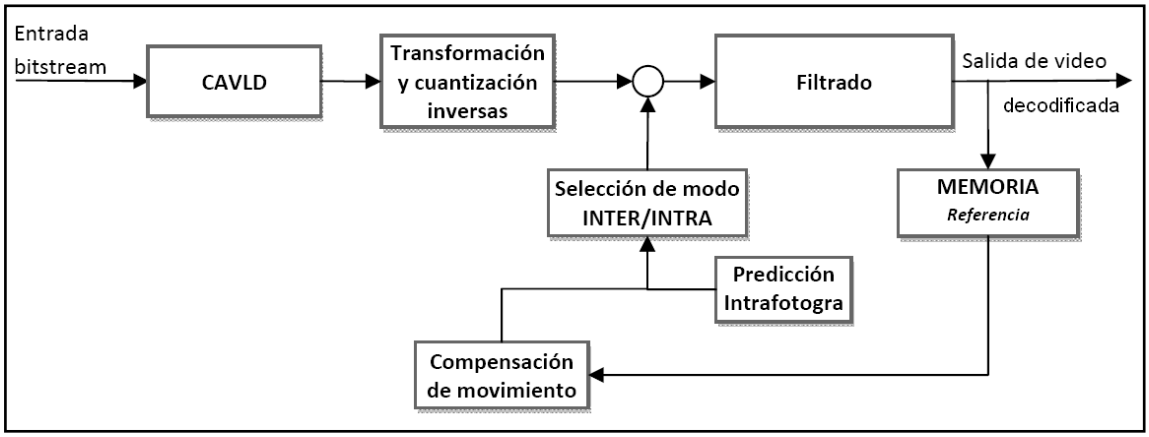

Figura 5. Diagrama de bloques del decodificador H.264.

2.1.3 Predicción intrafotograma

El estándar H.264 hace uso del concepto de predicción intrafotograma (INTRA) para la codificación de bloques o macrobloques (MBs) de referencia y así reducir la cantidad de bits codificados. Para realizar la decodificación de un bloque o MB en este modo, se forma un bloque de predicción basado en un bloque reconstruido previamente dentro del mismo fotograma y sin filtrar. El bloque de luminancia bajo predicción puede construirse por subbloques de 4x4 muestras o por todo el bloque de 16x16. Para cada uno de los bloques de luminancia de 4x4 se selecciona un modo de predicción de entre nueve existentes. Para los bloques de luminancia de 16x16 se elige uno de entre cuatro posibles. Para los bloques de crominancia de 4x4 muestras sólo existe la posibilidad de un modo de predicción. Para información detallada sobre este método de codificación se recomienda las referencias [16, 17].

2.1.4 Compensación de movimiento

Este modo, conocido como INTER, crea un modelo de predicción a partir de uno o más fotogramas de vídeo previamente codificados, usando compensación de movimiento basado en bloques - *block-based motion compensation*. Una diferencia importante con respecto a estándares anteriores, es que H.264 incluye soporte para un rango de tamaños de bloque más amplio, desde 16x16 a 4x4, y sub-muestras de los vectores de movimiento más finos, de un cuarto de píxel de luma. La compensación de movimiento realizada con bloques más pequeños incrementa la ganancia de la codificación, a costa de incrementar el número de datos necesarios para representar la compensación.

2.1.5 Filtrado

Es el bloque con el que se trabajará en este TFM, por lo que se dedicará algo más de espacio a explicar su funcionalidad, tanto a nivel de objetivos que debe cumplir, como etapas de la que consta su ejecución.

El *DF* se encarga de eliminar los efectos no deseados de diferenciación entre los bordes de los bloques, para generar una imagen más suavizada. En el proceso de decodificación la unidad de predicción busca bloques similares al bloque actual, no siendo el bloque encontrado exactamente idéntico, provocando de este modo errores de predicción. Para mejorar la eficiencia en la codificación dichos errores son transformados y cuantizados. Tras la decodificación, el bloque reconstruido es diferente al bloque original, siendo estas diferencias especialmente notables en los bordes de los bloques. Para reducir el grado de discontinuidad se aplica el filtrado del *DF*.

Además, el *DF* tiene una alta carga computacional, dado que han de filtrarse todos los píxeles del borde del bloque adyacente. Esta es la principal razón para hacer una implementación *hardware* de este bloque.

La funcionalidad del *Deblocking Filter* está dividida principalmente en dos etapas:

- Una primera etapa encargada del cálculo de los parámetros de filtrado, asociados a cada bloque.
- La segunda etapa realiza los distintos niveles de filtrado en función de los parámetros calculados en la etapa anterior.

Entrando más en detalle en la primera etapa, los parámetros de filtrado han de ser calculados para cada eje vertical y horizontal, entendiendo como eje la frontera que separa cada pareja de bloques de 4x4. En la Figura 6 se muestran los ejes de filtrado, mientras que la Figura 7 expone las particiones por parámetros de filtrado.

Se suele considerar que el filtrado del *DF* es adaptativo ya que la fuerza de filtrado es dependiente del tipo de macrobloque de entrada, así como de sus valores. Esta fuerza de filtrado se conoce como *Boundary Strength*, y se calcula siguiendo el diagrama de flujo mostrado en la Figura 8.

Una vez calculado el parámetro de BS, los siguientes valores necesarios son α y β, que previenen de filtrar en exceso aquellos ejes correspondientes a bordes de contornos de la imagen, lo cual conllevaría a una pérdida de nitidez en la imagen filtrada. Estos parámetros se calculan en función de los parámetros de cuantización de los bloques a los que afectan siguiendo los 6 pasos que se exponen:

- Paso 1: Se nombran como QPp y QPq los parámetros de cuantización de los bloques 4x4 a filtrar.
- Paso 2: Se calcula el parámetro de cuantización medio como: QPav = (QPp + QPq + 1) >> 1

- Paso 3: Se calcula el índice de la tabla de parámetros α: indexA = Clip3(0, QPav + FilterOffsetA, 51)
- Paso 4: Se calcula el índice de la tabla de parámetros β: indexB = Clip3(0, QPav + FilterOffsetB, 51)
- Paso 5: α = Alpha_Table(indexA)
- Paso 6: β = Beta_Table(indexB)

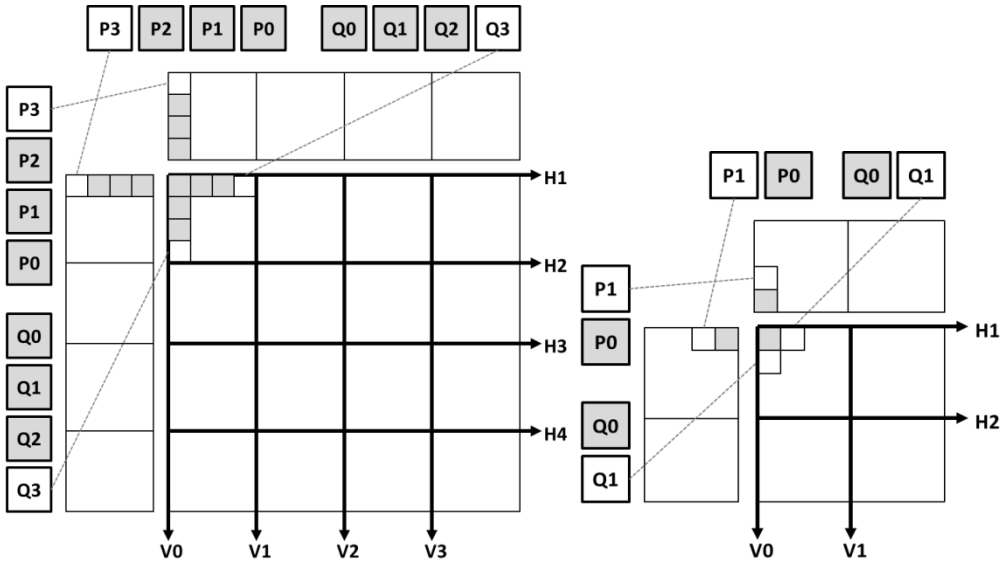

Figura 6. Ejes de filtrado vertical y horizontal.

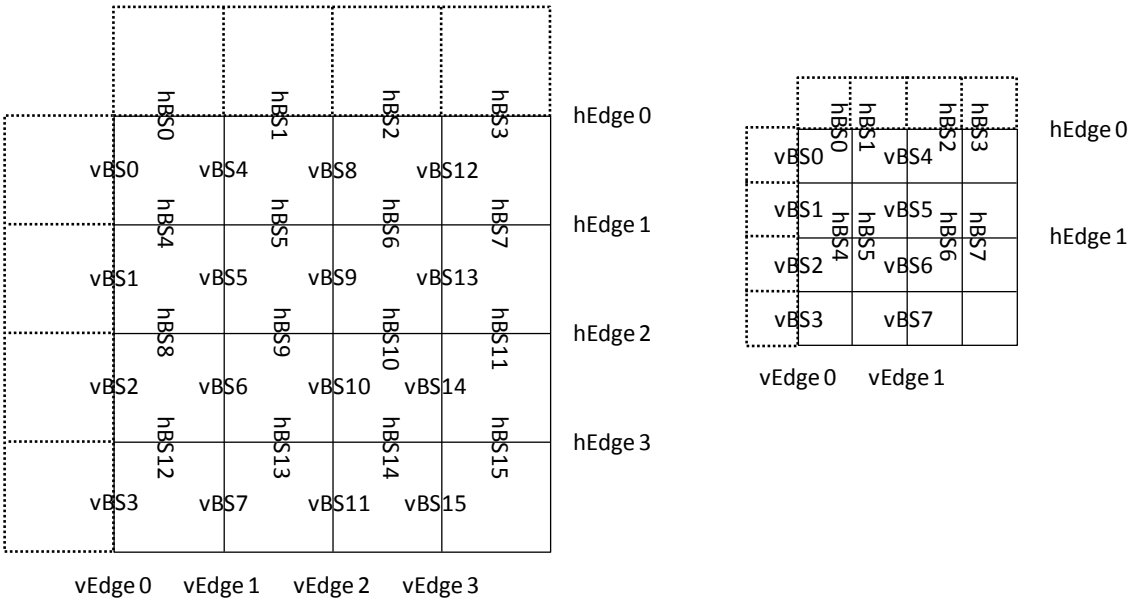

Figura 7. Particiones de parámetros BS de filtrado.

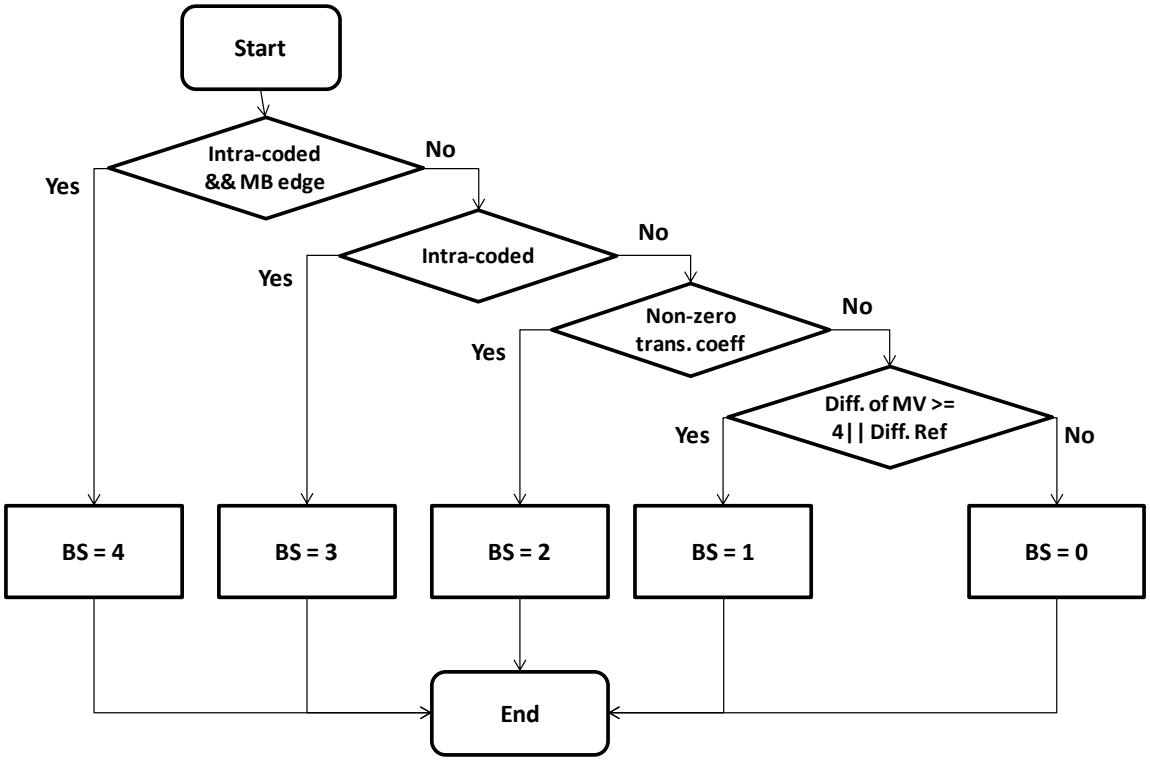

Figura 8. Diagrama para el cálculo del parámetro BS.

La función Clip3 comprueba que un valor esté dentro del rango especificado, en este caso 0 – 51, saturando dicho valor a 0 o 51 en caso de que sea inferior o superior respectivamente.

El último parámetro a calcular es el *flag* de filtrado, que decide en función de los valores previamente calculados y de los valores de las muestras de píxeles si se debe o no filtrar este eje. Las condiciones que se deben cumplir sin excepción para realizar dicho filtrado son:

- Condición 1: BS != 0.
- Condición 2: Abs(p0 – q0) < α.
- Condición 3: Abs(p1 – p0) < β.
- Condición 4: Abs(q1 – q0) < β.

En la segunda etapa se realizan las funciones de filtrado de los bloques de luma y croma. Para ello se toman cuatro muestras de cada uno de los dos bloques a filtrar que tienen frontera común y dependiendo del parámetro de BS se hará un filtrado fuerte o débil, lo cual acarreará ejecutar distintos flujos para cada uno de los casos. Estos flujos están detallados en [18].

2.2 H.264/SVC (*Scalable Video Coding*)

El término SVC hace referencia a la codificación de una fuente de vídeo en diferentes términos de resolución, imágenes por segundo y calidad en un mismo flujo de bits. Ese mismo

término fue utilizado para nombrar la extensión del estándar H.264 centrado en dar solución a estos requerimientos.

Debido a los avances en la codificación de vídeo digital, actualmente existen un gran número de entornos donde se requiere del uso de este tipo de tecnología, desde difusión de televisión hasta videollamadas. Estos diferentes entornos presentan a su vez, diferentes restricciones en función del medio de transmisión utilizado (que implicará la posibilidad de usar un mayor o menor ancho de banda) así como de los dispositivos que deberán decodificar el flujo de bits.

El concepto de codificación de vídeo escalable tiene su máxima utilidad en sistemas donde se requiere codificar una misma fuente de vídeo con diferentes parámetros para diferentes dispositivos receptores y/o diferentes medios de transmisión. El objetivo del estándar H.264/SVC es obtener un flujo de bits para un vídeo de alta calidad, que contiene subgrupos de bits para capas de inferior calidad, con una complejidad de decodificación cercana al estándar H.264/AVC y sin que ello implique un incremento de tamaño de datos.

El vídeo escalable ofrece además otra ventaja. Se puede considerar que, en el concepto de escalabilidad, las capas base son los que aportan una mayor información, o al menos, la información más importante, mientras que las capas de mejora son menos indispensables. Esto implica que un dispositivo de alta capacidad computacional, preparado para decodificar la capa más alta, puede, en caso de pérdida de un paquete, decodificar hasta la penúltima capa en lugar de la última, sin que ello implique la pérdida de la imagen [19].

Un ejemplo de aplicación de gran interés para SVC es el de la videovigilancia, donde además de que el vídeo deberá mostrarse en dispositivos tan heterogéneos como un monitor de alta definición hasta una PDA, este debe también ser almacenado por un tiempo prudencial. Puesto que es importante hacer un uso eficiente de los sistemas de almacenamiento, se podrá realizar un esquema de almacenamiento donde se guardará todo el flujo de bits durante la primera etapa temporal de mayor impacto, para ir borrando las capas de mejora según se vayan cumpliendo ciertos plazos, liberando así una importante cantidad de espacio de los sistemas de almacenamiento [19].

La estandarización de SVC busca básicamente cinco objetivos [19]:

1. Una eficiencia de codificación comparable al de la codificación como una única capa, para cada subconjunto del flujo de bits. Esto es, que el tamaño del vídeo comprimido como la suma de una capa base más varias capas de mejora en SVC, sea comparable al tamaño que tendría si hubiese sido codificado mediante un

codificador AVC, o al menos, que el incremento esté comprendido entre un 10% o 50%.

2. Que el incremento de complejidad para la decodificación no implique un gran impacto frente a sistemas de codificación de única capa o convencionales.
3. Soporte de escalabilidades espacial, temporal y de calidad.
4. Retrocompatibilidad de su capa base con el estándar H.264/AVC.
5. Soporte de adaptaciones simples del flujo de bits tras la codificación.

3 Arquitectura del sistema de referencia

La arquitectura del sistema de referencia de partida está dividida en cinco bloques, atendiendo a las etapas de ejecución previamente explicadas. El diagrama de bloques que representa esta partición se muestra en la Figura 9 [20].

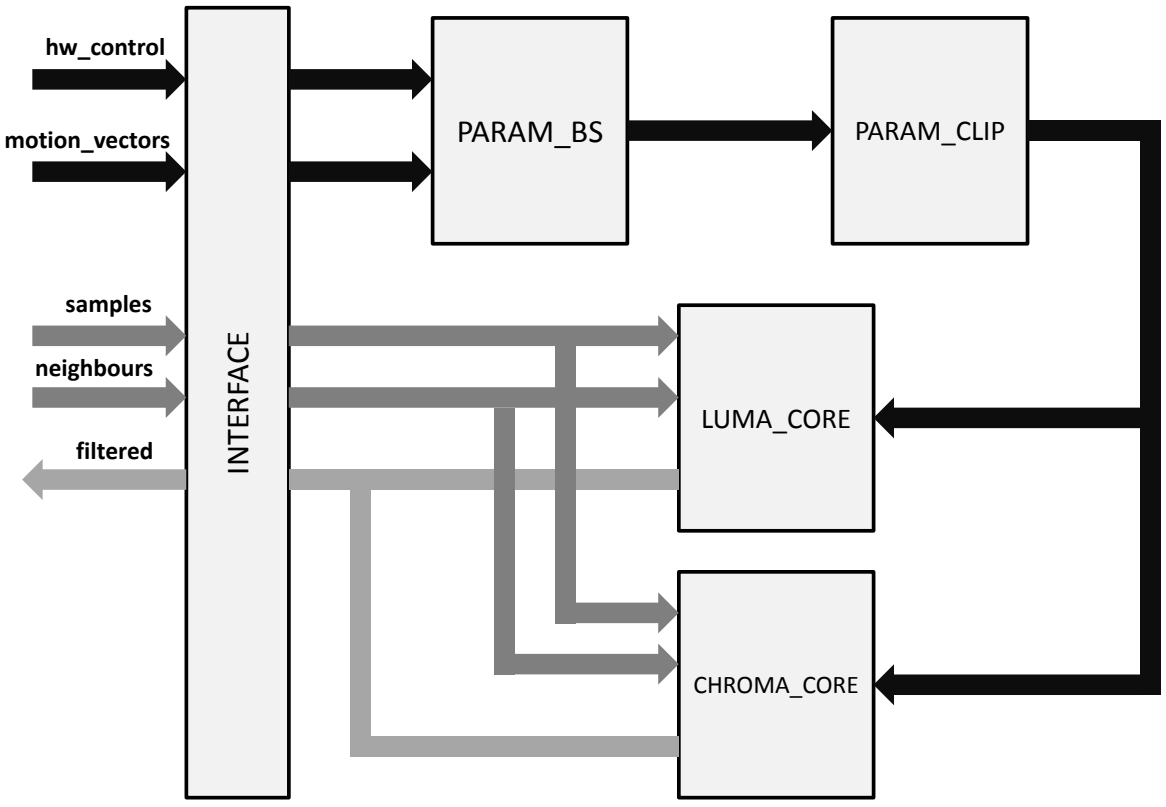

Figura 9. Diagrama de bloques del sistema de referencia [20].

A continuación se definirán los módulos principales que componen el sistema de referencia así como su funcionalidad.

1. Bloque INTERFACE. Realiza las funciones de entrada/salida con el exterior del *DF*, la distribución de los datos hacia el resto de bloques y el control de los mismos. Es el encargado de, una vez que lee los datos de entrada y se los envía al resto de bloques, indicar a estos cuando han sido completamente validados. También será el encargado de enviar hacia el exterior del DBF las muestras filtradas. La comunicación con el exterior se realiza mediante cinco puertos dedicados, uno para datos proveniente de bloques anteriores del decodificador (parámetro de cuantización, tipo de macrobloque, etc.), otro para vectores de movimiento en el caso de bloques de predicción INTER, otro para las muestras del macrobloque a filtrar, otro para las muestras de los vecinos al macrobloque y un último para la salida de las muestras filtradas.
2. Bloque PARAM_BS. Es el encargado de calcular el parámetro *Boundary Strength* para cada uno de los ejes de bloque del macrobloque actual, tanto para la luminancia como para las dos componentes de croma.
3. Bloque PARAM_CLIP. Calcula los parámetros límite α y β para cada uno de los ejes de filtrado tanto de luminancia como de croma.
4. Bloque LUMA_CORE. Es el encargado de comprobar si se cumplen las condiciones de filtrado presentadas anteriormente, y en dicho caso aplicar las ecuaciones de filtrado en la componente de luminancia.
5. Bloque CHROMA_CORE. Realiza las mismas tares que el bloque LUMA_CORE pero a las dos componentes de croma.

La comunicación entre bloques se realiza mediante memorias intermedias debido a la gran cantidad de datos que requieren cada etapa. En la Figura 10 se muestra un diagrama de bloques más detallado del sistema.

3.1 Perfilado del sistema de referencia

Una vez analizada la arquitectura del sistema es interesante conocer la capacidad de filtrado del sistema, así como las aportaciones parciales al retardo total de cada uno de los bloques. Para realizar este estudio se introducen marcas de tiempo en el código de referencia de forma que sea posible medir el tiempo que se tarda en filtrar un macrobloque en su totalidad, es decir, desde que se comienzan a solicitar los primeros datos al exterior hasta que el último pixel filtrado es escrito a través del puerto de salida.

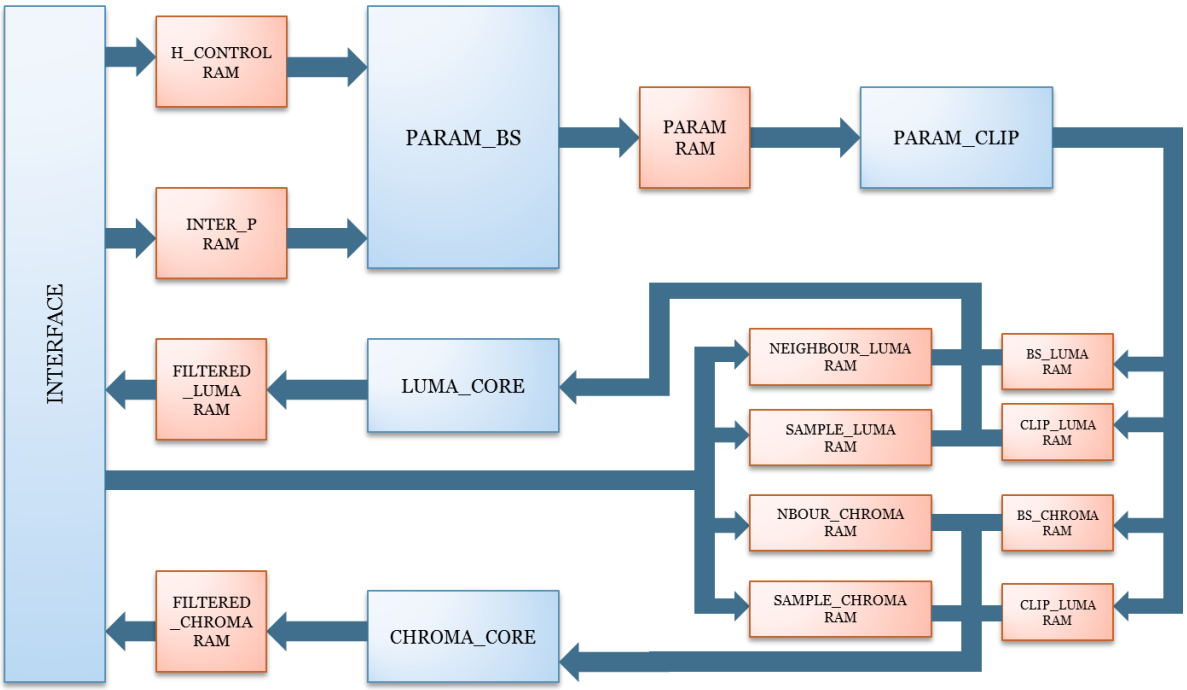

Figura 10. Diagrama detallado de la arquitectura de referencia.

Para realizar dichas simulaciones se han utilizado dos secuencias de vídeo, Foreman y Bus. La secuencia de Foreman consiste en un primer plano del personaje en cuestión hablando, con un fondo muy estático. En el caso de Bus la secuencia sigue a un autobús desde un lateral mientras circula, por lo que el fondo se desplaza lateralmente, a velocidad no constante, a lo largo de a secuencia. Ambas han sido codificadas con dos capas, una capa base en formato QCIF y otra capa de mejora en formato CIF. Así, se podrá medir el impacto que tienen ciertos parámetros en los tiempos de filtrado. A continuación se representa el retardo de filtrado de los primeros 1.400 macrobloques, para la capa base y para la capa de mejora.

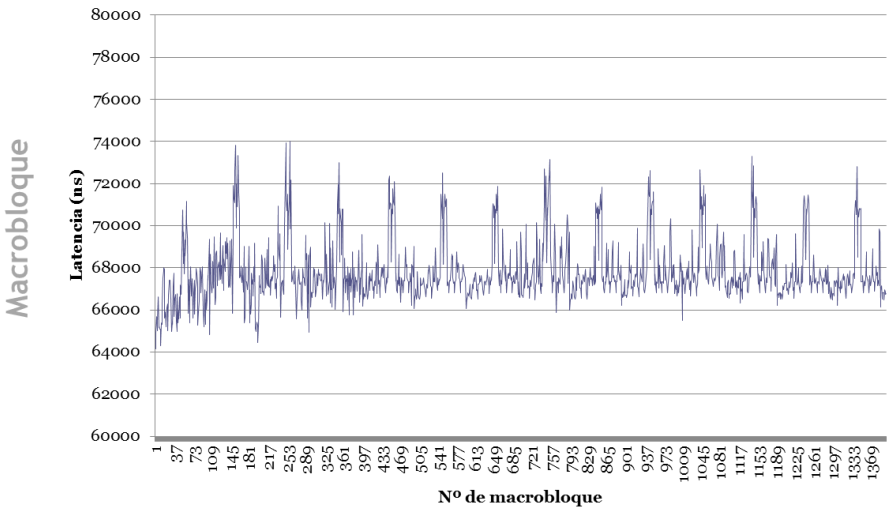

Figura 11. Latencia de filtrado de macrobloque. (Foreman - QCIF - Base Layer).

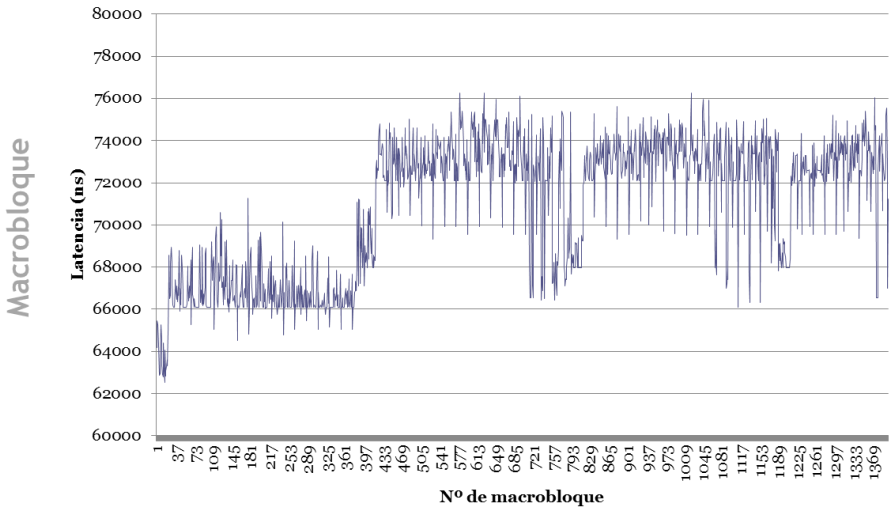

Figura 12. Latencia de filtrado de macrobloque. (Foreman - CIF - Enhancement Layer).

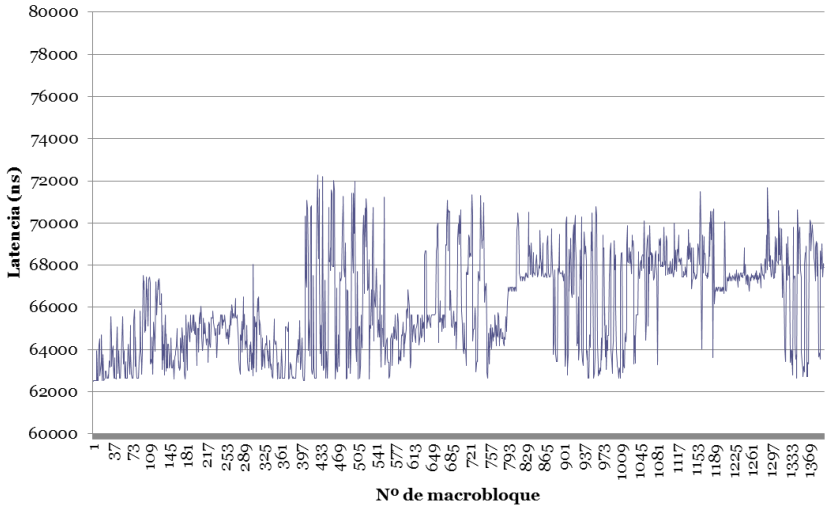

Figura 13. Latencia de filtrado de macrobloque. (Bus - CIF - Enhancement Layer).

A partir de las gráficas aportadas se pueden observar algunos fenómenos destacables como son los siguientes:

a. El filtrado de un macrobloque perteneciente a una capa de mejora (EL) presenta, en general, mayor coste computacional y tiene un mayor retardo.
b. El filtrado de macrobloques tipo INTRA requiere de menor tiempo de filtrado, lo cual puede verse reflejado en los primeros 99 macrobloques de la capa base (QCIF) o en los primeros 396 de la capa de mejora (CIF). Aunque se pudo observar que el parámetro *Boundary Strength* es siempre mayor en los macrobloques de tipo INTRA, también es importante hacer notar que el cálculo de dicho parámetro es más costoso cuanto más condiciones deban comprobarse, haciendo que en los de tipo INTER con vectores de

movimiento bajo o nulo (caso del fondo de la secuencia Foreman que permanece en estático entre imágenes) su cálculo sea muy costoso.

Los tiempos medios de filtrado de macrobloque para cada caso, que quedan como:

Tabla 1. Tiempo de filtrado de macrobloque.

	QCIF - BL	CIF - EL
Tiempo de filtrado (ns)	53.700	58.400

A partir de estos datos, y conociendo el número de macrobloques que contiene un *frame* para cada caso (QCIF - 99 y CIF - 396) se puede calcular el *frame rate* capaz de filtrar el diseño de referencia.

Tabla 2. Frame rate obtenido para la capa base y la capa de mejora.

	QCIF - BL	CIF - EL
Frame rate (f/s)	188	43

Otro dato importante es conocer la distribución de tiempo consumido en el filtrado entre los diferentes bloques. Esta distribución queda reflejada en la Figura 14 y en la Tabla 3.

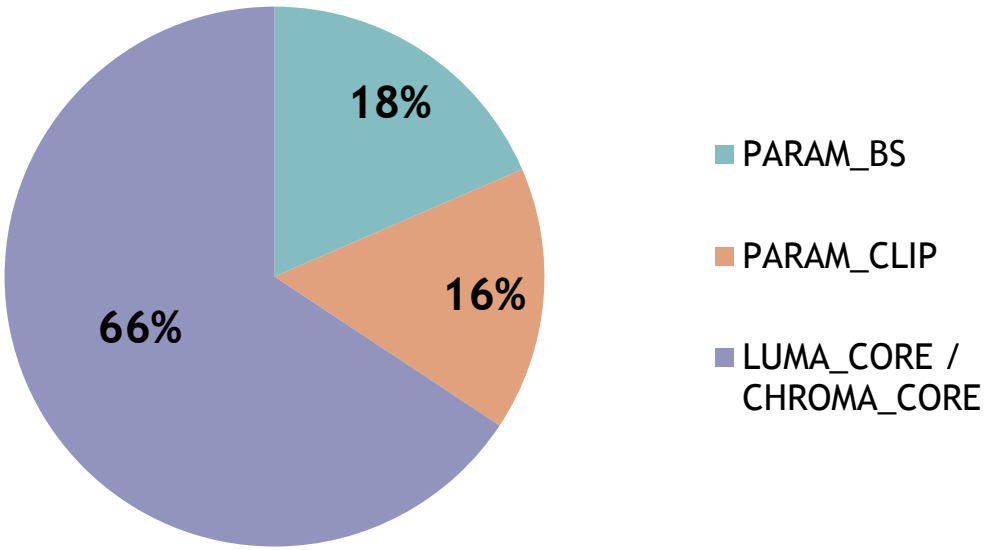

Figura 14. Distribución de latencia por bloques.

Tabla 3. Latencia media por bloque.

PARAM_BS	PARAM_CLIP	LUMA_CORE	CHROMA_CORE
10,8 µs	9,2 µs	38,4 µs	34,3 µs

Es importante tener en cuenta que el filtrado tanto de las componentes de croma como de luminancia se realizan en paralelo, siendo la de luminancia la que tiene un consumo mayor y la que afecta a la ruta crítica del sistema. Así, se puede observar que dos tercios del tiempo de ejecución del bloque son dedicados al propio filtrado, mientras que una tercera parte se dedica al cálculo de parámetros de filtrado.

4 Conclusiones

En este capítulo se ha presentado el dominio de aplicación en el que se encuentra el presente Trabajo Fin de Máster.

Se ha explicado de forma superficial la arquitectura típica de un decodificador H.264, así como las mejoras introducidas por el anexo SVC para vídeo escalable del estándar.

Por último se ha presentado la arquitectura del sistema de referencia y se ha realizado una tarea de *profiling* con el fin de comprender cuales de las tareas consumen un mayor tiempo y poder realizar optimizaciones a la misma.

La arquitectura presentada está formada por cinco bloques principales: INTERFACE, PARAM_BS, PARAM_CLIP, LUMA_CORE y CHROMA_CORE, además de un conjunto de memorias auxiliares. El análisis realizado muestra que es capaz de procesar 188 frames/s para la capa base en QCIF y 43 frames/s en la capa de mejora en CIF.

Capítulo 3: Metodología de diseño

1 Introducción

La metodología se define como el conjunto de métodos que se siguen en una investigación científica, entendiendo un método como el modo de decir o hacer con orden [21].

Para cumplir los objetivos de este Trabajo Fin de Máster descritos con anterioridad, en este capítulo se describirá la metodología usada y como consecuencia de esta, los lenguajes, herramientas y tecnologías necesarias para su realización.

Se ha seguido una metodología orientada a la síntesis de alto nivel, partiendo de una descripción en lenguaje SystemC. Dentro de los niveles de alto nivel definidos, nos encontramos en una precisión temporal de ciclo, introducido por el diseñador durante la descripción del modelo, y una granularidad de datos de bus, al no utilizar sentencias TLM para la comunicación entre bloques. Con esto, y atendiendo a las definiciones de Ghenassia, nos encontraríamos en un nivel CA (*cycle-accurate*) [22]. En la Figura 15 se muestra una representación de estos niveles.

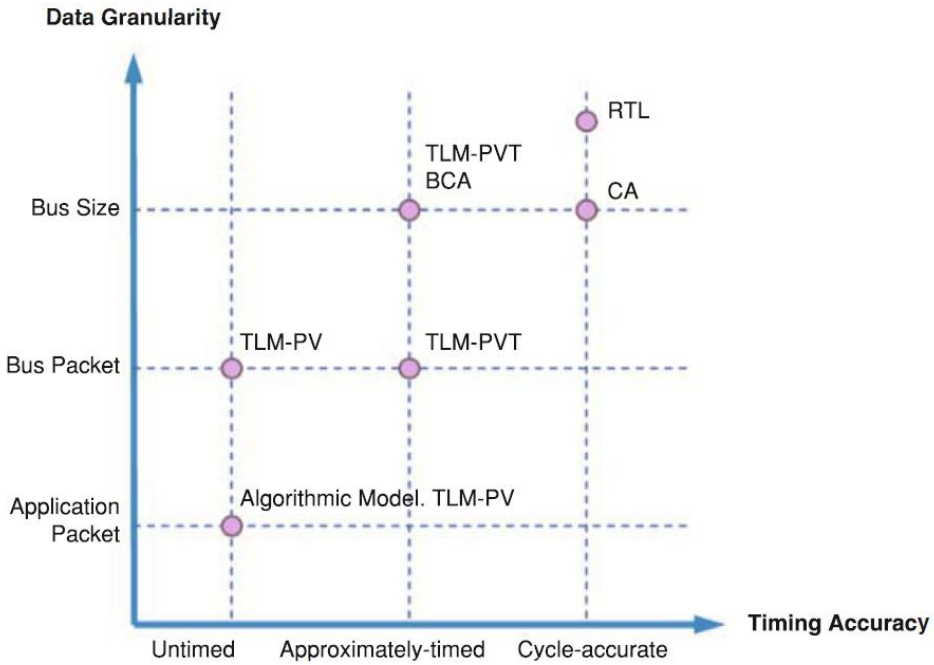

Figura 15. Niveles definidos por Guenassia [22].

2 Metodología de diseño

En todo diseño, ya sea hardware, software o de cualquier índole, se pueden enumerar tres fases claves en la metodología de diseño, a saber: especificación, diseño y verificación.

En la primera fase se analizan las especificaciones del sistema a realizar, ya sean estas funcionales, temporales o de cualquier otra índole que se haya dado. Una vez definidas y acotadas, se procederá al diseño del sistema en el que se decide la tecnología, las etapas de desarrollo, la partición del diseño (si la hubiese), etc. Ello produce una primera versión funcional del sistema que se utiliza para su verificación.

Con una primera versión terminada, se verifica que cumple con las especificaciones definidas inicialmente, para adoptar las medidas correctoras necesarias en fases tempranas del ciclo de diseño, repitiendo el proceso hasta obtener una versión que cumpla con las prestaciones esperadas.

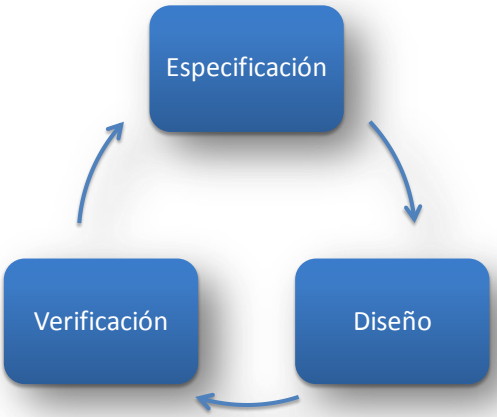

Figura 16. Flujo de diseño.

3 Lenguajes

Debido a la creciente complejidad de los diseños de sistemas electrónicos, se hace necesaria la aparición de nuevos métodos para realizar la captura del diseño. En la actualidad, hay una tendencia cada vez mayor a la descripción del sistema electrónico a nivel de sistemas (ESL)[23], por lo que es necesario disponer de lenguajes que soporten un nivel de abstracción a nivel de sistema tal como SystemC.

3.1 SystemC

Es el lenguaje de descripción de alto nivel en el cual está diseñado el sistema a tratar en el presente TFM. Se detallan a continuación algunos conceptos de interés para la correcta comprensión de las innovaciones que aporta este lenguaje en las metodologías de diseño hardware.

Para profundizar en las características de SystemC se recomienda la lectura de [24], el manual de referencia de la última versión de la librería, así como con guías de diseño y ejemplos [13].

3.1.1 Definición

SystemC es una librería de clases de C++ desarrollada conjuntamente con una nueva metodología de diseño y verificación funcional para describir sistemas electrónicos complejos a nivel de sistemas (ESL).

El uso de SystemC, conjuntamente con las herramientas estándar de desarrollo para C++, tiene como objeto poder modelar a nivel de sistema sistemas electrónicos, y poder validarlo con los menores costes temporales posibles, además de conseguir un modelo que cumpla las especificaciones con el fin de que la siguientes fases de diseño puedan usarlo como referencia de funcionalidad.

La razón de elegir C++ como lenguaje para el desarrollo de la librería SystemC es que se trata de un lenguaje de alto nivel de uso muy extendido, además de que sus compiladores generan códigos muy eficientes.

SystemC proporciona aquellas características que no están soportadas por C++, necesarias para el desarrollo de sistemas electrónicos:

- Noción temporal: C++ no permite conocer los instantes lógicos en los que se producen los eventos.
- Concurrencia: C++, y en general cualquier lenguaje orientado al diseño de software no está preparado para describir sistemas concurrentes, algo que es intrínseco en el diseño de hardware.
- Tipos de datos: Los tipos de datos que soportan los lenguajes de programación no contemplan factores necesarios en el hardware como son el valor de alta impedancia Z en una señal.

La librería de SystemC, a través de la definición de nuevas clases de objetos en C++, da solución a estos requerimientos sin la necesidad de redefinir sintácticamente el lenguaje. Esto sumado a que se permite insertar código en C y C++ en el diseño, permite realizar una coverificación HW/SW sin un incremento notable en el esfuerzo de validación.

3.1.2 Elementos principales

A continuación se detallan algunos elementos de relevancia en el modelado de sistemas con SystemC.

1. Módulos

Es la clase usada por SystemC para modelar la estructura del diseño, dando soporte a conceptos tales como jerarquía, interconexión, etc. Un módulo encapsula un componente del diseño, que puede contener a su vez un conjunto de módulos interconectados a través de canales.

```
SC_MODULE (adder_reg) {
  …
};
```

2. Puertos

En cada módulo podrán definirse puertos de entrada, de salida y bidireccionales, con el fin de interconectar módulos entre sí, o si fuese el módulo jerárquicamente superior, para representar las entradas y salidas del diseño. Además del sentido del puerto, cada uno podrá tener un tipo de datos, que podrá ser cualquier tipo de datos soportado por C/C++, tipos definidos por el usuario, o tipos soportados por la librería SystemC.

```
sc_in<bool> clk;
sc_in<sc_int<8> > a;
sc_in<sc_int<8> > b;
sc_out<sc_int<9> > c;
```

3. Señales/canales

Conjuntamente con los puertos, sirven para comunicar varios módulos. De igual forma se utilizan para comunicar varios procesos. A diferencia de los puertos, las señales son internas a cada módulo y no son visibles desde otros módulos externos al contenedor de la señal en cuestión. Como los puertos, los datos pueden ser de tipos de C/C++, de usuario o de SystemC.

```
    sc_signal<sc_int<9> > temp;
```

4. Procesos

En ellos se describe las funcionalidad de un módulo, que consistirá en un código escrito en C/C++ haciendo uso tanto de funciones del lenguaje raíz, como de métodos de las clases de la librería SystemC. Existen tres tipos de procesos: SC_METHOD, SC_THREAD, SC_CTHREAD.

SC_THREAD

Son procesos que solo se ejecutan una vez al comienzo de la ejecución del sistema. Sin embargo, permiten suspender el proceso con el fin de ser reanudado en el mismo estado en el que fue suspendido cada vez que se produce un evento en su lista de sensibilidad. Cada vez que la ejecución del proceso encuentre una secuencia wait(); entrará en suspensión. En general, los procesos de este tipo suelen definirse como un bucle infinito, de forma que cuando termine vuelva a ejecutarse, y se define su latencia como el número de eventos necesario para una ejecución del bucle. Puede usarse para definir la ejecución de un camino de datos segmentados, por ejemplo.

A continuación se muestra un ejemplo de declaración, definición, registro y lista de sensibilidad de un proceso de tipo SC_THREAD. Sin embargo, se recomiendo que la definición del mismo se encuentre separado en un fichero *.cpp, y el resto en un fichero de cabecera *.h [25].

```
SC_MODULE (adder_reg) {
…
void reg();
…
void adder_reg::reg() {
while (true) {
…
wait();
…
  }
}
…
SC_THREAD(reg);
sensitive << clk.pos();
```

SC_CTHREAD

Se trata de una modificación sobre el tipo SC_THREAD. Su principal diferencia es que en el registro se le añade un evento como sensibilidad, de forma que en la definición pueden realizarse nuevas formas de suspensión. Un ejemplo es la realización de una suspensión que se reanude un número de eventos más tarde, en lugar de en el siguiente evento. Otra es la posibilidad de suspender el proceso hasta que se cumpla una condición (y se genere el evento).

Estas formas de suspensión pueden también conseguirse con el proceso SC_THREAD como se muestra a continuación, pero requiere que el kernel reanude el proceso para volver a ponerlo en reposo en cada evento, haciendo la ejecución más lenta.

A continuación se muestran los modos de suspensión comentados, a la izquierda en su implementación para un SC_THREAD y su equivalente en un SC_CTHREAD a la derecha.

`for (i=0; i!=N; i++) wait();`	`wait(N);`
`do wait() while(!expr);`	`wait_until(expr.delay);`

SC_METHOD

Los procesos de tipo SC_METHOD se ejecutan cada vez que se produce un evento en su lista de sensibilidad y no pueden ser suspendidos. En general se usan para simular un comportamiento combinacional ya que se ejecuta en tiempo cero y no se suspende para volver a reanudarse. Al igual que en el tipo SC_THREAD se debe indicar en la lista de sensibilidad los eventos que provocarán la ejecución del método.

```
SC_MODULE (adder_reg) {
…
void add();
…
void adder_reg::add() {
  …
}
…
SC_METHOD(add);
sensitive << a << b;
```

A continuación se detalla un ejemplo de diseño de un sumador con la salida registrada en el que se hace uso de dos procesos en paralelo. Uno de ellos se encargará del proceso combinacional de realizar la suma y otro de registrar la señal [26].

```
#include "systemc.h"

SC_MODULE(adder_reg) {
  sc_in<sc_int<8> > a;
  sc_in<sc_int<8> > b;
  sc_out<sc_int<9> > c;
  sc_in<bool> clk;

  sc_signal<sc_int<9> > temp;

  void add() { temp = a + b; }
  void reg() { while (1) {c = temp; wait();} }

  SC_CTOR (adder_reg) {
    SC_METHOD(add);
    sensitive << a << b;

    SC_THREAD(reg);
    sensitive << clk.pos();
  }
};
```

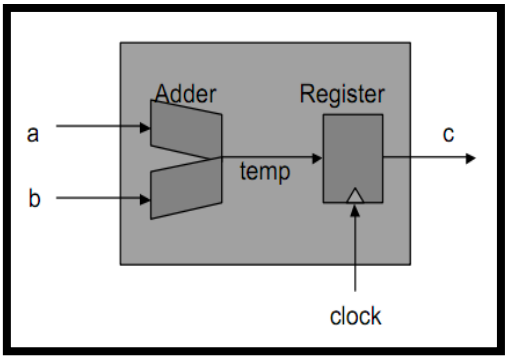

Figura 17. Diseño de ejemplo de uso de procesos en SystemC

Como puede observarse en el ejemplo anterior, por un lado existe un proceso de tipo SC_METHOD que se ejecuta en tiempo cero cada vez que se produzca un evento en alguno de los puertos de entrada (a y/o b), realizando la suma de ambas y almacenándola en temp. Por otro lado, en cada flanco de subida del reloj, el valor almacenado en temp se escribe en el puerto de

salida c. Esta descripción cumple funcionalmente con el diseño propuesto en la Figura 17, incluyendo que en un ciclo hayan varias modificaciones de las señales de entrada, almacenándose en la salida la última modificación antes del flanco de subida del reloj (el proceso SC_METHOD se ejecutará, tantas veces como cambios haya en los puertos de entrada, pero solo se escribirá en el puerto de salida el valor que tuviese en el momento en el que se reanuda el proceso SC_THREAD).

5. Relojes

Los relojes son objetos con una notación temporal, que permite dotar al sistema de relaciones temporales en su simulación. Se pueden definir varios relojes, con distintos tiempos de ciclo, ciclos de trabajo, así como fase relativa arbitraria.

```
sc_clock clk("clk", 10, SC_NS, 0.5, 0.0, SC_PS);
```

En el ejemplo anterior se ha declarado un reloj llamado clk, con un periodo de 10 ns, un ciclo de trabajo del 50% y un desfase de 0 ps.

Las variables temporales en SystemC se almacenan como un entero de 64 bits y las unidades en las que se pueden representar son: SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC.

3.1.3 Características principales

A continuación se detallan algunas características fundamentales de SystemC que hacen de este lenguaje una buena alternativa en el diseño hardware de alto nivel.

a. **Simulación basada en eventos.** Al contrario que en una simulación por ciclos de reloj, la simulación basada en eventos presenta unos tiempos de simulación mucho más bajos, ya que no precisa de la ejecución de todos los procesos paralelos del sistema en cada ciclo de reloj, sino que pueden haber varios suspendidos en espera de que se cumpla uno de sus eventos.

b. **Múltiples niveles de abstracción.** SystemC permite realizar modelos de diferente precisión temporal, desde *untimed* a *cycle accurate*, con varios niveles de abstracción, desde funcional hasta RTL [22]. Esto permite refinar iterativamente el código hasta llegar a un modelo RTL que cumpla la funcionalidad y usarlo como fuente para pasar a un diseño en VHDL o Verilog y de ahí seguir el flujo estándar de diseño electrónico.

c. **Trazado de formas de onda.** Con SystemC es también posible el trazado de formas de ondas en los formatos VCD, WIF e ISDB.

3.1.4 Estructura de capas

La Figura 18 muestra la arquitectura del lenguaje SystemC, siendo los bloques centrales el núcleo de este lenguaje. Como puede observarse, está construido sobre C++ [13].

Las capas superiores son librerías y estándares de diseño que el usuario puede decidir usar o no. Gracias a esta estructura se pueden seguir añadiendo librerías sobre el núcleo sin tener que modificar el mismo.

El núcleo del lenguaje está formado por el simulador orientado por eventos, junto a los elementos ya comentados (puertos, módulos, procesos, etc.). Los tipos de datos son necesarios para el modelado *hardware* y ciertos tipos de desarrollo *software*. Los canales primarios son los canales incorporados que tienen un amplio uso tal como las señales y las FIFOs.

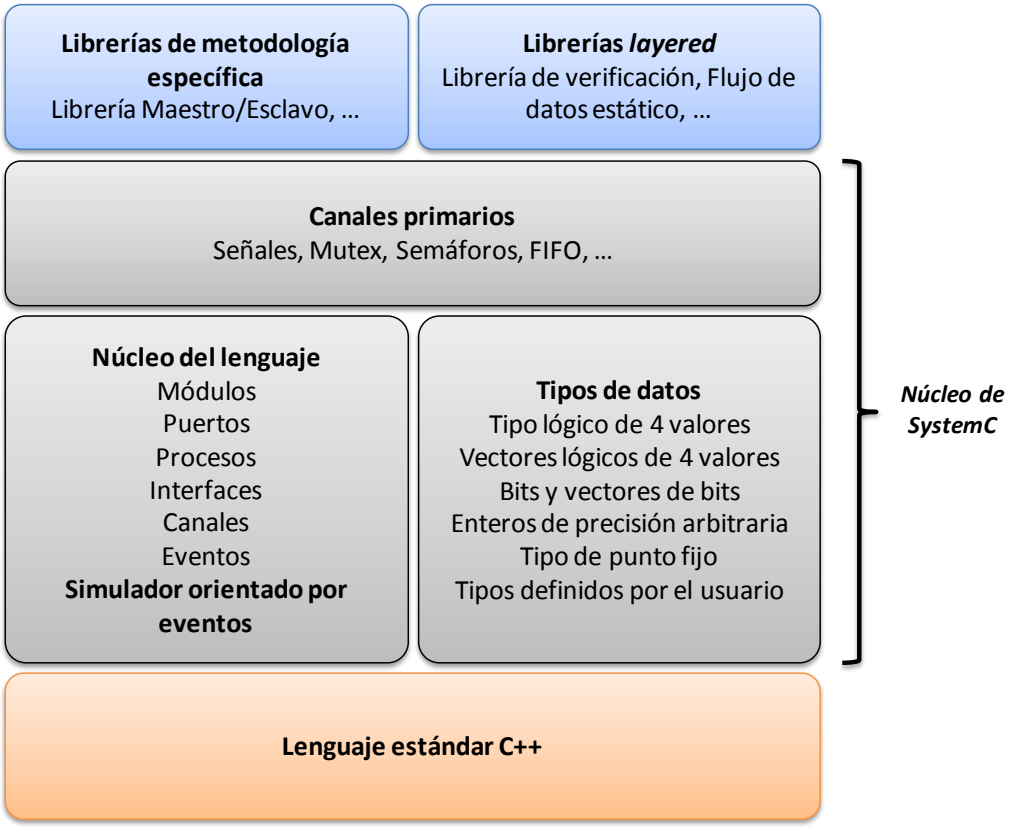

Figura 18. Arquitectura de capas de SystemC [24].

3.1.5 Metodología de diseño

Como ya se indicó anteriormente SystemC representa, además de una librería de clases, una metodología de diseño que tiene como objetivo acelerar las fases de diseño a nivel funcional así como su verificación.

En un flujo tradicional de diseño electrónico es una práctica habitual realizar un diseño funcional en C/C++ con el fin de crear un modelo de alto nivel que cumpla las especificaciones tras

varias iteraciones de diseño y optimización, para luego realizar una traducción a un lenguaje de descripción *hardware* a nivel RTL (VHDL o Verilog). Este tipo de metodología presente inconvenientes conocidos como pueden ser:

a. Traducir el código C/C++ a VHDL/Verilog puede producir errores difíciles de depurar. Aún con un modelo en C/C++ que cumpla con las especificaciones, es difícil realizar una traducción sistemática.
b. Caducidad del modelo C/C++. Una vez el diseño se realiza a nivel RTL, todas las optimizaciones se realizarán sobre este, lo que llevará a que el modelo en alto nivel quede desfasado.
c. Imposibilidad de reutilizar los test de diseño. La naturaleza de los tests de una aplicación en C/C++ son muy diferentes a un *testbench* de un sistema *hardware*, lo que obligará a rediseñar los sistemas de test para la versión RTL del diseño.

La metodología que presenta SystemC pretende dar solución a los problemas anteriormente citados y se representa en el diagrama de flujo de la Figura 19.

Las principales ventajas de este flujo modificado pueden resumirse en los siguientes puntos:

a. La metodología de refinamiento permite escribir una primera versión del sistema en C/C++ e ir incluyendo características de SystemC al diseño en varias iteraciones.
b. El realizar el sistema en SystemC para luego sintetizarlo hace que no sea necesario conocer otros lenguajes de descripción *hardware*.
c. El tiempo de desarrollo se reduce, tanto en diseño como, lo que es más importante, en verificación.
d. Los *testbenchs* pueden ser reutilizados desde la primera versión hasta la versión más refinada y cercana a RTL.

Una parte importante de la metodología implica la verificación de que la solución obtenida sea equivalente a la especificada. En este caso, se debería volver a realizar la verificación del sistema, adaptando el *testbench* inicial a los requerimientos de latencia presente en el modelo preciso a nivel de ciclos, obtenidos durante la síntesis de alto nivel. Otra forma de realizar la verificación es utilizar métodos de comparación formal entre las diferentes representaciones.

4 Herramientas

Este apartado recoge las principales herramientas de ayudas al diseño (EDA) utilizadas en el proyecto. Son utilizadas para cubrir uno o varios pasos del flujo de diseño, ya sea en sus fases de análisis, síntesis o verificación del mismo. En lo posible se han utilizado lenguajes y formatos

estandarizados, asegurando la independencia del flujo de diseño de las herramientas concretas utilizadas.

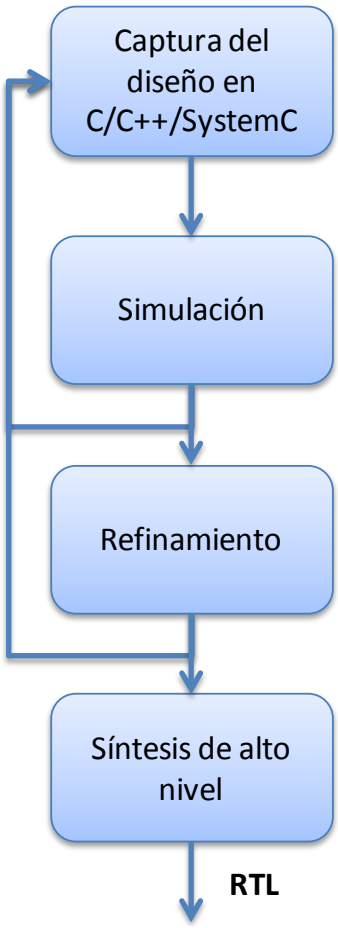

Figura 19. Metodología de SystemC

4.1 C-to-Silicon Compiler

Cadence CtoS (C-to-Silicon Compiler) es una herramienta de síntesis de alto nivel. Este tipo de herramientas tiene como objeto reducir el tiempo de diseño en sistemas complejos y, en la mayoría de los casos, mejorar la calidad de los resultados en comparación con las traducciones hechas a mano por los diseñadores.

CtoS genera una descripción RTL del diseño en el estándar de IEEE Verilog, de forma que el diseño puede seguir la ruta de síntesis mediante herramientas de síntesis lógica disponibles, ya sea la propia de Cadence (Encounter RTL Compiler) o cualquier otra de las que existen en el mercado (Xilinx Synthesis Technology, Synplify Pro/Premier, etc.). Además del código RTL, CtoS genera descripciones de comportamiento, tanto en SystemC como en Verilog, para su simulación funcional [27].

Existen dos modos de ejecución de la síntesis en CtoS, ya sea en modo interactivo mediante el uso de una interfaz gráfica y en modo *batch* mediante el uso de scripts en TCL. La utilización del modo interactivo facilita la toma de decisiones por parte del diseñador con la ayuda de las herramientas gráficas que posee el entorno. Cada decisión tomada en ella, generará una o varias órdenes de consola, que pueden consultarse para escribir el script de síntesis correspondiente para ser utilizados en modo *batch,* eliminando procesos tediosos cuando las decisiones de diseño ya han sido tomadas.

4.1.1 Flujo de diseño

En la Figura 20 se representa el flujo de diseño en CtoS que incluye las etapas que se enumeran y describen a continuación.

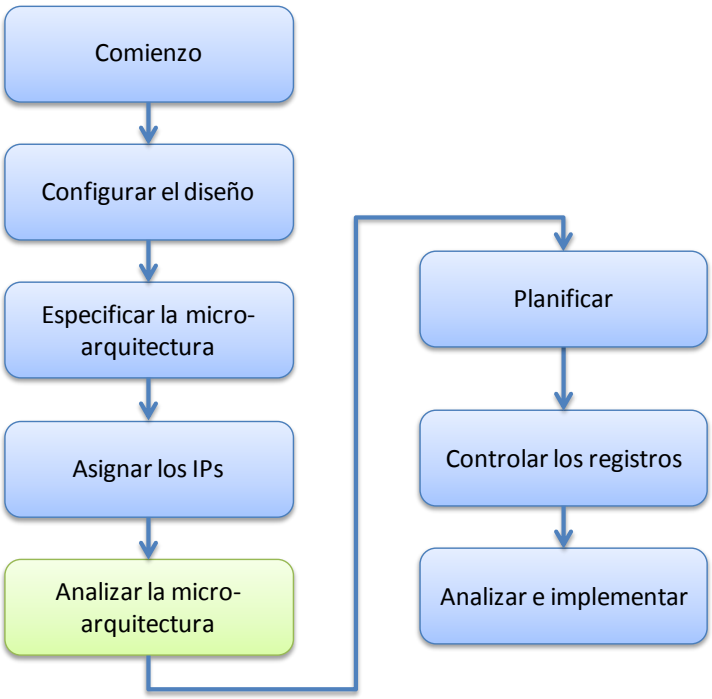

Figura 20. Flujo de diseño de CtoS

1. Crear/cargar el diseño

El primer paso a realizar consiste en crear un nuevo diseño o abrir uno existente. Existen diferentes alternativas, ya sea el diseño basado en C/C++, SystemC o TLM.

2. Configuración del diseño

Aquí habrá que indicar si el diseño será SystemC, TLM o C/C++. Se definirá aquí la fuente de reloj, con sus propiedades de nombre, frecuencia, ciclo de trabajo, etc. Se incluirán los ficheros fuente, así como aquel que corresponde al *top* del diseño.

3. Especificación de la micro-arquitectura

En este apartado se realizan las transformaciones arquitecturales para hacer el diseño sintetizable. Los elementos a transformar son: funciones y bucles.

Algunas funciones no son sintetizables por tener dos caminos diferentes con distinta latencia, lo cual impide al que hace la llamada conocer la espera que debe realizar. Otra causa puede ser que acceda a vectores donde se pueda escribir. Esto se soluciona haciendo las funciones "en línea" (*inline*), es decir, sustituir la llamada por el código que la describe.

Por otro lado, los bucles combinacionales (bucles que se realizan idealmente en tiempo cero) no son sintetizables, por lo que puede decidirse entre varias formas de solucionarlo:

a. Desenrollar el bucle para ser realizado enteramente en tiempo cero. Esto puede ser inviable para bucles de muchas iteraciones, pues haría del bucle la ruta crítica con una latencia muy alta.

b. Romper el bucle para que cada iteración se realice en un ciclo de reloj.

c. Realizar una segmentación el bucle. Se divide la ejecución del bucle en un número determinado de etapas, de tal forma que cada iteración del bucle se encuentra en una etapa del mismo simultáneamente, paralelizando así varias iteraciones.

Decidir la solución tomar es responsabilidad del diseñador, y deberá tener en cuenta restricciones tanto de área como de frecuencia. La Figura 21 muestra una imagen de CtoS durante este paso.

4. Asignar los IPs

En este paso se asignarán las variables de tipo vector/*array* del diseño a bancos de registros, a memorias internas de la FPGA, o a descripciones HDL de memorias.

5. Analizar la micro-arquitectura (opcional)

Una vez asignados los IPs, el diseño está completamente transformado, a falta de que el CtoS realice las optimizaciones oportunas, por lo que puede realizarse un análisis temporal, de área y

de potencia (menos exacto que el que se realiza tras el planificador) con el fin de modificar algunas de las decisiones tomadas anteriormente.

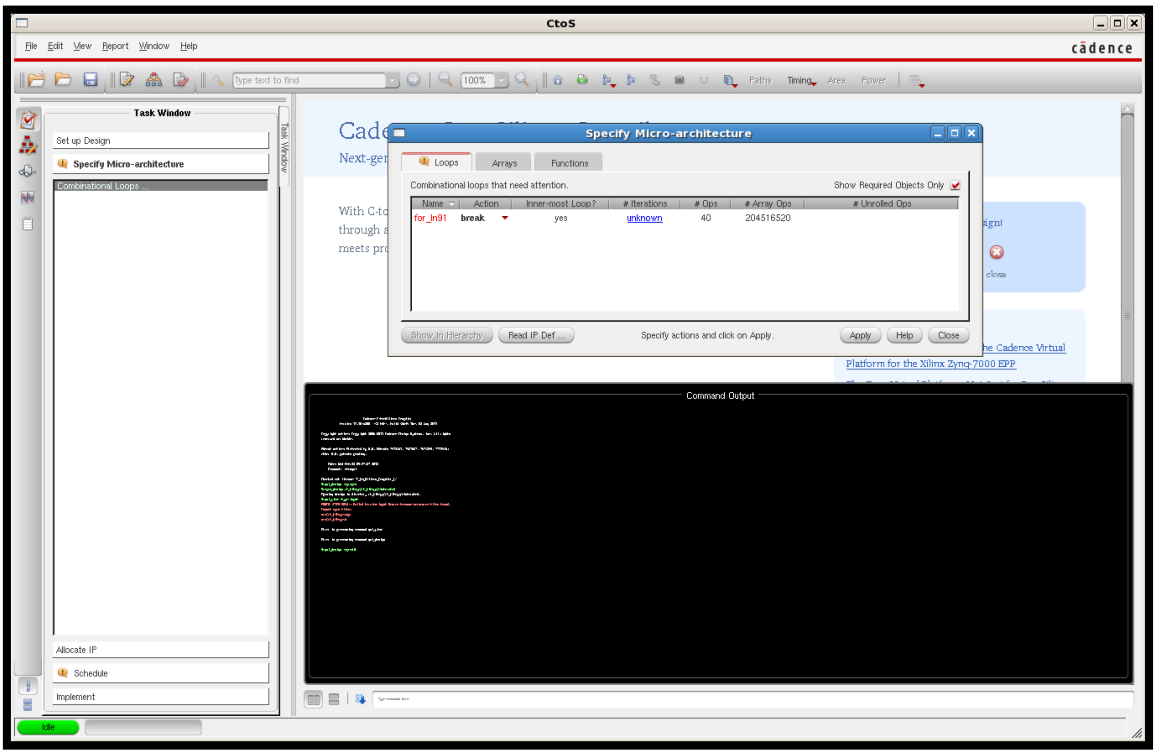

Figura 21. Etapa de especificación de micro-arquitectura en CtoS.

6. Planificar

En este punto, el planificador de CtoS asignará recursos a las operaciones del diseño fuente. En caso de que el planificador no pueda resolver esta tarea, se presentan algunas acciones que faciliten la obtención de una solución:

a. Controlar la dependencia con vectores de datos. Esto hará que el planificador conozca las dependencias y pueda añadir estados para resolverlas.

b. Controlar los estados. En caso de que el planificador no pueda resolver conflictos secuenciales, el diseñador puede añadir manualmente estados para guiar a CtoS a encontrar una solución a dichos conflictos.

c. Controlar los recursos. El diseñador podrá a priori, crear unos recursos iniciales sobre los que mapear las operaciones del diseño.

7. Asignación y control de registros

En esta etapa puede decidirse si incluir una lógica de *reset* adicional para aquellos registros que por defecto no lo tengan. También puede decidirse si registrar o no todas las salidas de los recursos de lógica combinacional asociados a operaciones del diseño. Aquí se debe decidir entre

minimizar registros (decisión normalmente asociada a diseño ASIC donde el coste en área de los registros es muy alto) o minimizar el número de multiplexores (asociado normalmente al diseño basado en FPGAs).

8. Análisis e implementación

En este paso se podrán realizar informes de distinto tipo, tales como consumo de recursos, o de latencia en ciclos de reloj de cada bloque.

También se podrán generar aquí los ficheros de salida, como puede ser la descripción RTL, un script que realice la mismas acciones que las realizadas a través de la interfaz gráfica, o un fichero contenedor en SystemC de la descripción RTL, con el fin de poder realizar una simulación de esta, usando el *testbench* en SystemC con el que se verificó el modelo en alto nivel. La Figura 22 muestra el grafo de control y datos (CDFG) generado por CtoS.

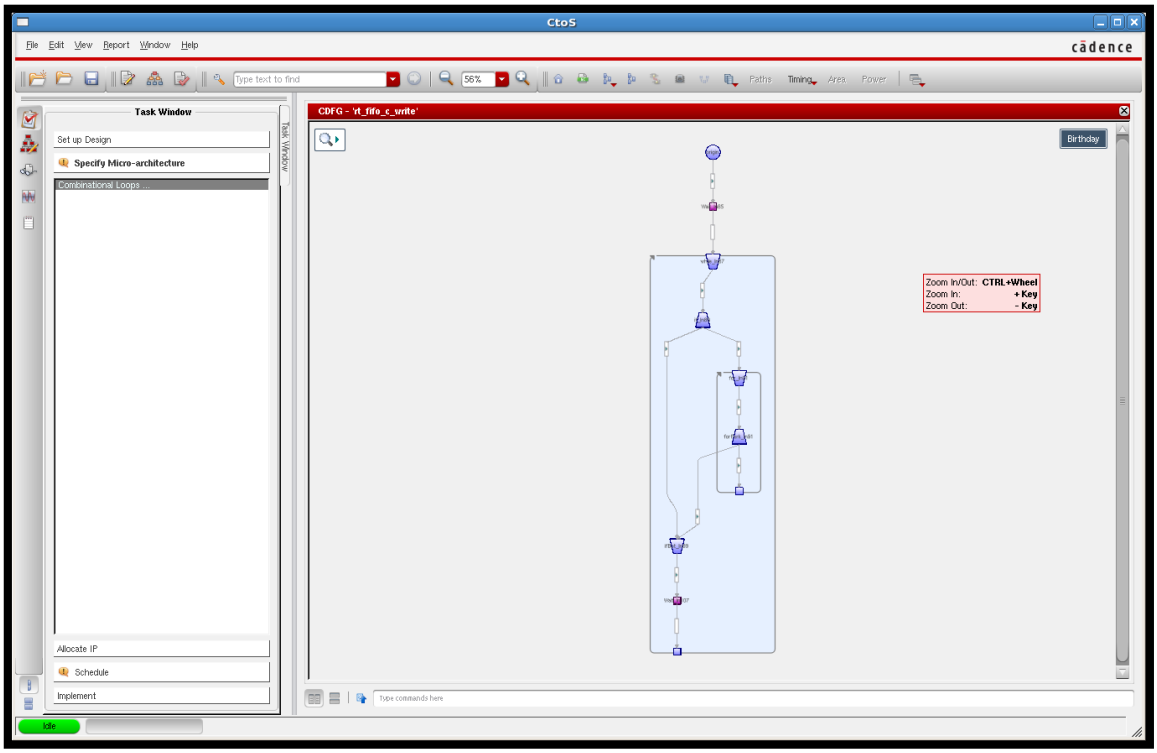

Figura 22. Grafo de control y datos en CtoS

4.1.2 Características soportadas de SystemC

CtoS soporta el subconjunto sintetizable de SystemC. Sin embargo, como se comentó en el apartado de lenguaje SystemC, este está pensando no sólo para la realización de síntesis de alto nivel, sino también para descripciones puramente de comportamiento, y es por ello que muchas de sus funcionalidades no son sintetizables por las herramientas de síntesis de alto nivel.

Tabla 4. Características de SystemC no soportadas por CtoS.

Tipo de características	Características no soportadas	
Clases del lenguaje	get_child_objects sc_process_handle sc_event_and_list sc_event_or_list sc_event sc_time register_port	default_event sc_fifo sc_buffer sc_mutex sc_semaphore sc_event_queuesc_clock
Tipos de datos	get_child_objects sc_generic_base	sc_numrep float
Utilidad de clases	sc_trace sc_report sc_report_handler sc_exception	sc_copyright sc_version sc_release
Anclaje de puertos	Solo podrán hacerse estáticamente en el constructor del módulo.	
Punteros	Solo se podrán usar para apuntar a variables de forma que estáticamente cada puntero pueda saberse a que variable apunta.	

4.2 Synplify Premier

Synopsys Synplify Premier es una herramienta para la realización de la síntesis lógica del diseño. Está especialmente centrada en el diseño sobre FPGA [28].

Synplify Premier soporta las siguientes características necesarias en la fase de síntesis de diseño hardware:

a. **Diseño jerárquico.** Debido a la creciente complejidad de los diseños y a su tamaño, cada vez más es típica la división en subdiseños con el fin de trabajar en paralelo. Synplify Premier permite la realización de proyectos divididos en subproyectos independientes. Permite tanto diseño *top-down* como *bottom-up*.

b. **Modo rápido**. Es un modo de síntesis que se realiza con una aceleración de hasta x3, permitiendo realizar un análisis inicial del sistema.

c. **Multiprocesado usando particiones.** Pueden definirse lo que Synplify llama puntos de compilación (*Compile Points*) que son en realidad particiones del diseño con el fin de que pueda lanzarse en paralelo la síntesis de cada una de las particiones, con el consecuente ahorro de tiempo de síntesis.

d. **Diseño incremental.** Haciendo uso de los puntos de compilación, se pueden heredar soluciones de cada partición sintetizadas con anterioridad, para que únicamente sea necesario resintetizar la partición que haya sido modificada. Así, durante la fase de optimización, se puede ahorrar una gran cantidad de tiempo de procesamiento si

toda la optimización se realiza en un número reducido de particiones. Además, estas divisiones pueden transferirse a las herramientas de emplazamiento y ruteado.

4.2.1 Vistas

En Synplify Premier puede representarse el diseño en dos vistas principales, una antes de realizar la síntesis lógica, que se conoce como vista RTL, en la que se representa a modo de bloques la descripción de los ficheros fuentes HDL. La Figura 23 muestra la vista RTL.

La segunda vista es la tecnológica en la que cada bloque se representa como el conjunto de recursos sobre el que ha sido mapeado, ya sean estos registros, memorias de bloque RAM o LUTs. Esta vista se presenta en la Figura 24.

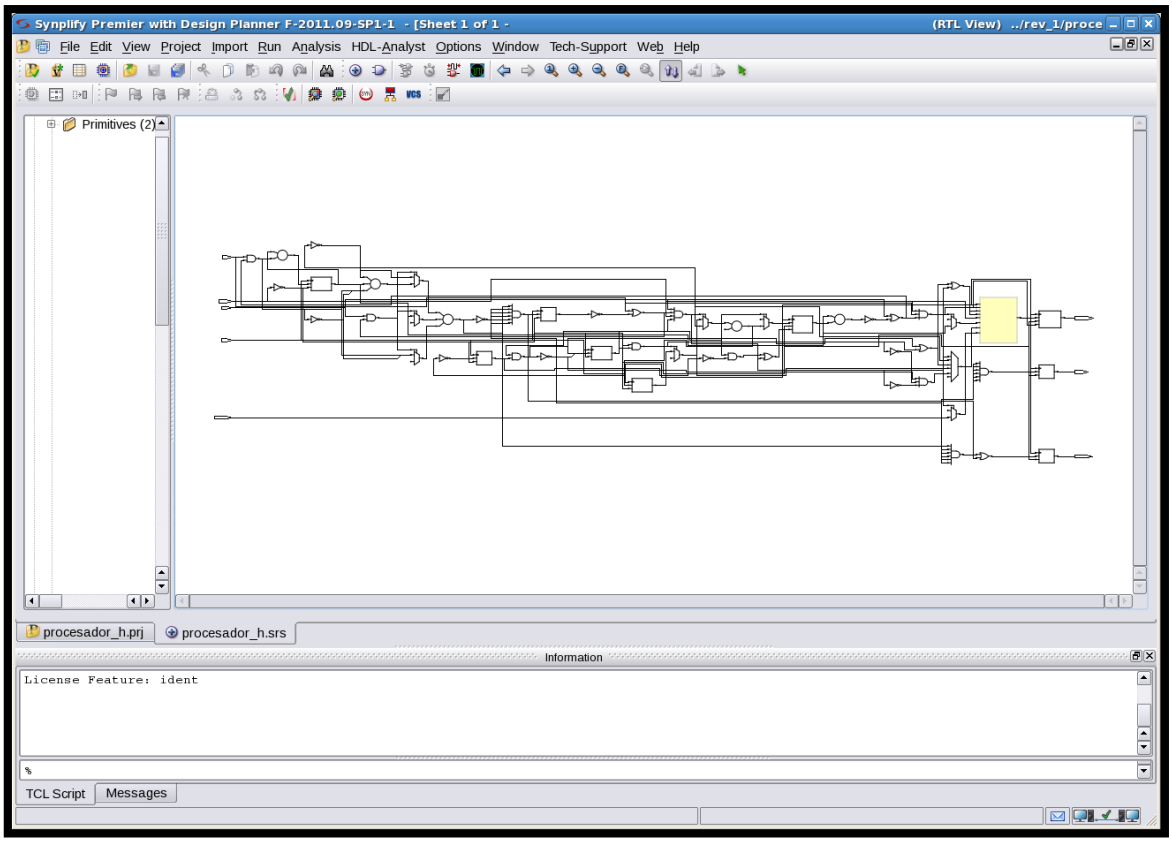

Figura 23. Vista RTL de Synplify Premier

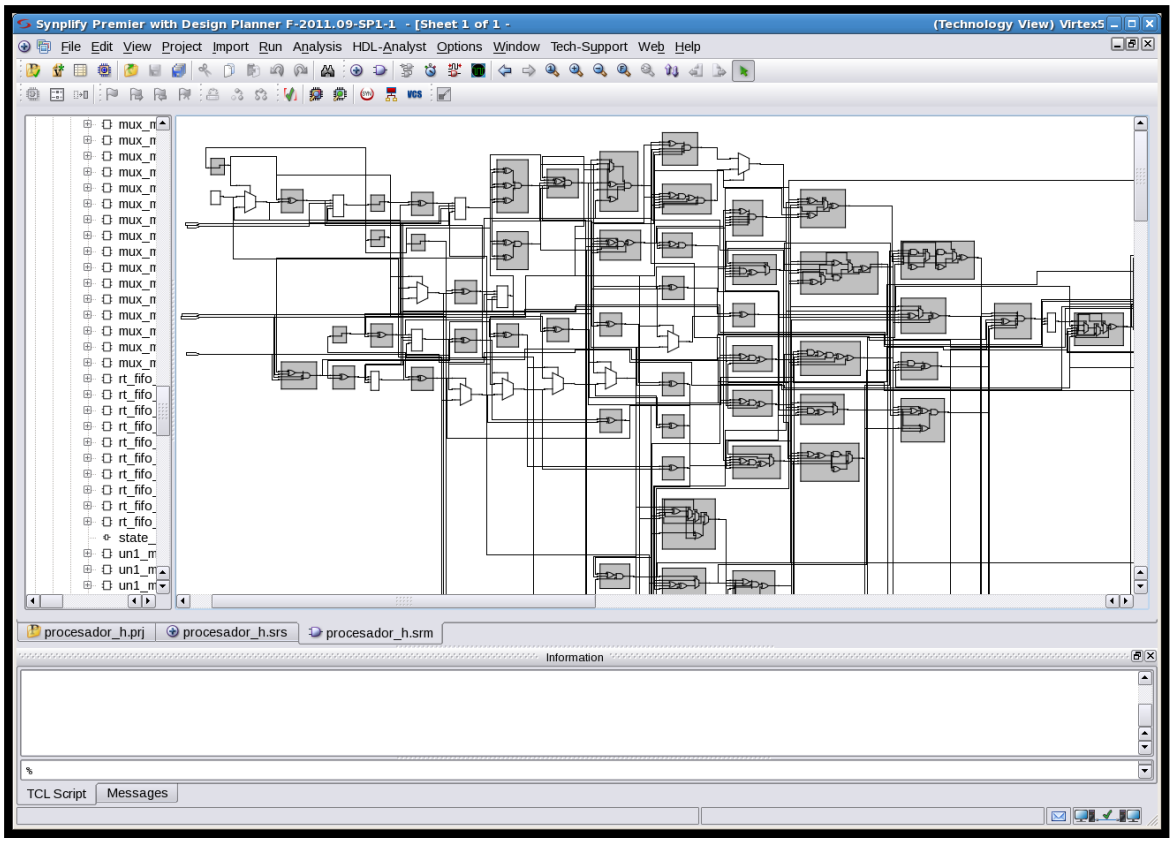

Figura 24. Vista tecnológica de Synplify Premier

4.3 Xilinx Platform Studio

Xilinx Platform Studio es la herramienta que Xilinx proporciona para la realización de plataformas *hardware* en las que se dispone de uno o de varios microprocesadores tanto Microblaze como PowerPC.

En conjunto con otras herramientas de Xilinx, XPS permite crear a nivel de plataforma un diseño formado por el microprocesador y un conjunto de periféricos, ya sean importados de la propia librería de IPs que proporciona Xilinx como diseñados por el usuario, que podrán ser importados en formatos HDL o netlist.

El objetivo de XPS es dar facilidad al diseñador a realizar instancias e interconectar distintos bloques de gran complejidad de forma sencilla y gráfica, pudiendo tener una visión global del sistema completo sin perderse en tener que entender a nivel de bit cada una de las interfaces de los distintos IPs.

La aplicación integra otras herramientas de Xilinx para poder implementar el diseño. De esta forma, una vez realizado el diseño, puede realizarse todos los pasos hasta la programación de la

FPGA sin tener que exportar el diseño a otras herramientas. Para ello, hace uso de otras herramientas [29], que se describen en la Tabla 5.

Tabla 5. Herramientas de Xilinx

Herramienta	Descripción
Xilinx CORE Generator System	Genera la descripción HDL de bloques IP parametrizables por el diseñador.
Platform Generator	Genera la descripción HDL de la plataforma completa a partir de los distintos ficheros sinstetizados o no de cada bloque IP y del fichero de especificaciones hardware (*MHS file*).
PlanAhead	El PlanAhead es una herramienta que se divide en tres etapas (Map, Place&Route y BitGen). Aunque XPS no realiza llamadas al PlanAhead, sí que lo hace a las tres etapas independientemente.
Xilinx Synthesis Technology (XST)	Es la herramienta de síntesis de Xilinx. Realiza la síntesis lógica que traduce la descripción HDL a una descripción estructural.
Software Development Kit (SDK)	Es un entorno de desarrollo software basado en Eclipse que Xilinx ha desarrollado para la realización y depurado de la aplicación software que se ejecutará en los microprocesadores de la plataforma.
Library Generator	Crea las librerías necesarias en función de las especificaciones software del proyecto (*MSS file*). Estas librerías son las que contendrán aquellas funciones que se hayan requerido en la elaboración de la aplicación del microprocesador.
GNU Compiler	Es el compilador que traduce la aplicación a lenguaje máquina del microprocesador usado en la plataforma. Se incluyen las versiones para Microblaze y PowerPC.
Xilinx Microprocessor Debugger	Se trata de un *Shell* con el que configurar el microprocesador de la placa. Con él se descarga el fichero ejecutable así como se dan las instrucciones para el arranque de la aplicación y la interrupción de detención.
System ACE File Generator	Genera el fichero de configuración Xilinx System ACE, que congrega el fichero *bitstream* de configuración *hardware* de la FPGA y el fichero *software* ejecutable del microprocesador.

4.4 ChipScope Analyzer

ChipScope Analyzer es una herramienta que facilita el depurado del diseño *hardware* mediante la utilización de mecanismos de tipo ILA (In-Circuit Logic Analyzer). Permite controlar los bloques de analizador lógicos internos en la FPGA, obteniendo la forma de onda de los mismos. Durante la fase de diseño de la plataforma, existen bloques IP contenidos en la librería de depurado de Xilinx, que tienen por objetivo almacenar muestras de los puntos de prueba a los

que se conecten. Durante su instanciación se deberá configurar el ancho de cada punto de prueba, así como la longitud en número de muestras que podrá almacenar cuando comience la captura (las muestras se almacenarán en memorias de bloque internas, por lo que deberá prestarse interés a dicha cantidad) [30].

Una vez la FPGA ha sido configurada, con bloques de análisis lógico internos conectados a aquellas señales que se quieran capturar, es cuando entra en juego el *software* ChipScope, el cual permitirá configurar las condiciones de disparo, es decir, las condiciones bajo las cuales comenzarán a capturarse datos.

La conexión entre el software de análisis lógico y los bloques *hardware* de captura de datos se realiza a través del puerto de configuración JTAG de la placa de prototipado, que tiene conexión directa con los puertos de configuración de la FPGA.

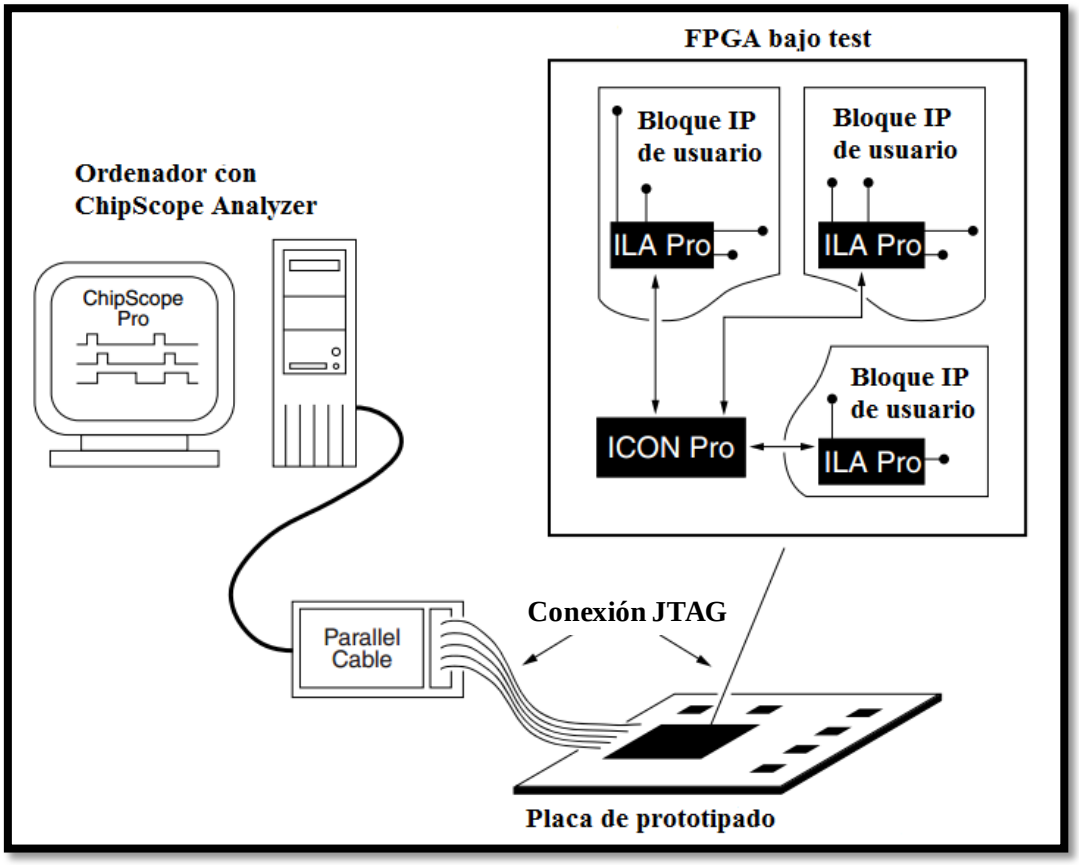

Figura 25. Conexión para analizador lógico ChipScope.

Se pueden configurar varias ventanas de captura, con la consecuente disminución de número de muestras que se puedan capturar por ventana. Cada señal conectada a un puerto de disparo, puede usarse tanto para definir condiciones como datos de captura. Se pueden formar condiciones complejas en forma de composiciones AND y/o OR.

5 Tecnologías

En este apartado se presentan las tecnologías y dispositivos FPGA con las que se trabajará en este TFM.

5.1 Familia de FPGA Xilinx Virtex-5

La familia Virtex-5 de Xilinx usa la segunda generación de la arquitectura basada en columnas ASMBL™ (*Advanced Silicon Modular Block*), estando dividida en cinco sub-familias con el fin de cubrir todas las posibles necesidades de los diseños a implementar. Además de las células lógicas programables, las FPGAs de la familia Virtex-5 contienen una gran cantidad de bloques de tipo *Hard-IP* (bloques *hardware* no programables embebidos en la FPGA) como pueden ser, BlockRAMs y FIFOs de 36Kbit, DSPs 24 x 18, tecnología *SelectIO™* con impedancia controlada digitalmente (DCI), bloques de interfaz *ChipSync™ source-synchronous*, funcionalidad de monitorización del sistema, generadores de reloj interno mediante DCMs (*Digital Clock Manager*) y PLLs (*Phase-Locked-Loop*) y opciones avanzadas de configuración, incluyendo reconfiguración parcial. Otras funcionalidades de la FPGA están soportadas por la presencia de bloques transceptores de alta velocidad y bajo consumo, bloques integrados de interfaz *PCI Express®*, MACs (*Media Access Controllers*) Ethernet tri-modo (10/100/1000 Mbps) y microprocesadores PowerPC® 440 de altas prestaciones. Está fabricada con tecnología de 65 nm [31].

5.1.1 Resumen de características de la familia Virtex-5

A continuación se listan los principales aspectos de la familia de FPGAs Virtex-5 de Xilinx. Se referencia al lector a la hoja de características para una descripción más completa [32].

- Cinco subfamilias:
  - Virtex-5 LX: Para aplicaciones de lógica general.
  - Virtex-5 LXT: Para aplicaciones de lógica general que requieran conectividad serie avanzada.
  - Virtex-5 SXT: Para aplicaciones de procesado de señales que requieran conectividad serie avanzada.
  - Virtex-5 TXT: Para sistemas que requieran conectividad serie avanzada de doble densidad.
  - Virtex-5 FXT: Para sistemas empotrados con conectividad serie avanzada.
- Compatibilidad entre plataformas:
  - Las familias LXT, SXT y FXT usan encapsulados compatibles, pudiéndose reutilizar diseños de PCBs modificando únicamente el voltaje.

- Estructura de lógica de altas prestaciones:
  - Tecnología de LUTs de 6 entradas.
  - Opción de LUTs de 5 entradas duales.
  - Rutas reducidas mejoradas.
  - Opción de RAM distribuida de 64 bits.
  - LUTs con registros de desplazamiento de 32 bits / opción dual de LUTs con registros de desplazamiento de 16 bits
- Gestión de reloj CMT:
  - Bloques DCM para retardo de *buffer* nulo, síntesis de frecuencia y desfase de reloj.
  - Bloques PLL para filtrado de *jitter*, retardo de *buffer* nulo, síntesis de frecuencia y división de reloj enganchado en fase.
- Bloques FIFO y RAM integrada de 36 Kbit:
  - Bloques de RAM de dos puertos.
  - Lógica de FIFO programable.
  - Programable:
    - Anchos de x36 para memorias de dos puertos.
    - Anchos de x72 para memorias de un puerto.
  - Circuitería interna opcional de corrección de errores.
  - Opcionalmente se puede programar cada bloque como dos bloques independientes de 18 Kbit.
- Tecnología *SelectIO™*:
  - Operaciones de entrada/salida con voltaje de 1,2 V a 3,3V.
  - Interfaz con tecnología *source-synchronous ChipSync™.*
  - Impedancia controlada digitalmente (DCI)
  - Bancos de E/S de granularidad fina.
  - Soporte para interfaz de memoria de alta velocidad.
- *Slices* DSP48E:
  - Multiplicadores 24 x 18 en complemento a dos.
  - Sumador, restador y acumulador opcional.
  - Segmentación opcional.
  - Funcionalidad de lógica a nivel de bits opcional.
  - Conectividad en cascada dedicada.

- Opciones de configuración flexibles:
  - Interfaces SPI y *Parallel* FLASH.
  - Soporte *multi-bitstream* con lógica de reconfiguración dedicada.
  - Detección automática de ancho de bus.
- Capacidad de monitorizar el sistema en todos los dispositivos:
  - Monitorización de temperatura on-chip/off-chip.
  - Monitorización de tensión de alimentación on-chip/off-chip.
  - Acceso por JTAG a todos los valores monitorizados.
- Bloques *Endpoint* integrados para diseños de PCI *Express*®.
  - Subfamilias LXT, SXT, TXT y FXT.
  - Acorde con la especificación base v1.1 de PCI *Express*®.
  - Soporte en cada bloque de pista x1, x4 o x8
  - Funcionalidad compartida conjuntamente con los transceptores *RocketIO*™.
- MACs Ethernet tri-modo 10/100/1000 Mbps.
  - Subfamilias LXT, SXT, TXT y FXT.
  - Los transceptores *RocketIO™* pueden ser usados como PHY o conectados a PHY externos usando las diferentes soluciones MII (*Media Independent Interface*).
- Transceptores *RocketIO™* GTP con tasas de 100 Mbps a 3,75 Gbps.
  - Subfamilias LXT y SXT.
- Transceptores *RocketIO™* GTX con tasas de 150 Mbps a 6,5 Gbps.
  - Subfamilias TXT y FXT.
- Microprocesadores PowerPC 440:
  - Solamente están soportados en la subfamilia FXT. Presenta Arquitectura RISC con un pipeline de 7 etapas, caches de instrucciones y de datos de 32 KB, con una Estructura de interfaz de procesador optimizada.
- Tecnología de 65 nm de cobre.
- Voltaje de núcleo de 1,0 V.
- Encapsulados Flip-Chip estándar y opciones libres de plomo.

5.2 Kit de desarrollo ML507

La placa ML507 [33] incluye la FPGA XCV5FX70T. El diagrama de bloques del dispositivo se muestra en la Figura 26.

Entre los recursos que se representan, de relevancia para el presente proyecto son la memoria RAM DDR2 SO-DIMM, el controlador SystemACE, el controlador RS-232 para comunicación serie y la interfaz de salida DVI.

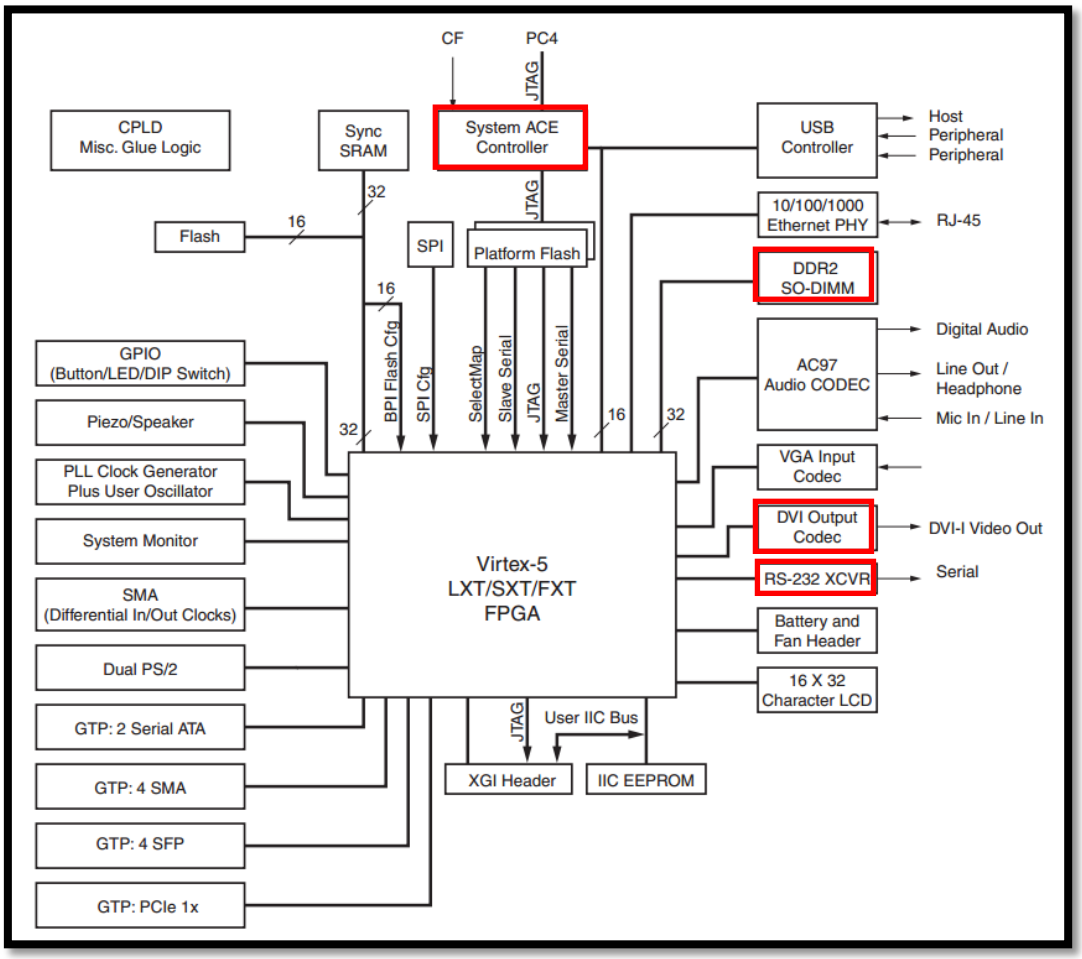

Figura 26. Diagrama de bloques de la placa ML507.

6 Flujo de diseño propuesto

A continuación, y a partir de los lenguajes, herramientas y tecnologías presentadas en los apartados anteriores, se concreta un flujo de diseño especificando la herramienta a usar en cada etapa del mismo. Este flujo se presenta en la Figura 27.

En un primer paso se hará una verificación funcional del sistema. Una vez comprobado que el sistema funciona según las especificaciones dadas, se continúa con síntesis de alto nivel del mismo.

El siguiente paso será la verificación funcional a nivel RTL del código generado por la herramienta de síntesis de alto nivel. Cuando esta fuere exitosa, se pasa a realizar la síntesis lógica con el fin de obtener métricas que permitan el estudio del diseño propuesto.

Para finalizar, este diseño será adaptado para ser prototipado sobre una plataforma FPGA. Será necesario diseñar un bloque interfaz que permita conectar el *DF* con un microprocesador que suministre las tramas de entrada y que represente la salida en una pantalla a través de la interfaz DVI.

La síntesis, junto con las validaciones necesarias, serán tratadas en el Capítulo 4 de este TFM y la adapación y validación del mismo en el Capítulo 5.

Figura 27. Flujo de diseño propuesto.

7 Conclusiones

En el presente capítulo se ha presentado el lenguaje, la tecnología, herramientas y metodología y técnicas fundamentales para la correcta realización de este Trabajo Fin de Máster.

En un primer apartado, se ha presentado SystemC, el lenguaje de descripción *hardware* de alto nivel usado en este desarrollo.

Posteriormente, se han descrito las herramientas a las que se hará alusión durante la descripción del desarrollo del trabajo, explicando sus objetivos y a qué etapa del flujo pertenecen.

Por último, se ha visto la tecnología de implementación, que para el Trabajo Fin de Máster corresponde a la familia de FPGAs Virtex-5 de Xilinx. Se ha destacado de esta, aquellas propiedades que las hacen de interés para el desarrollo de este trabajo.

Para terminar, se ha presentado el flujo de diseño propuesto, describiendo brevemente cada una de las etapas que lo componen.

Como conclusión de este capítulo se ha demostrado la necesidad de disponer de un flujo de diseño optimizado y orientado al diseño ESL, a partir de descripciones algorítmicas en SystemC, y que se conecte sin costuras a un conjunto de herramientas que faciliten el prototipado del sistema en una plataforma FPGA. Se trata de un flujo complejo, donde es necesario utilizar diferentes lenguajes y formatos estándares (SystemC, Verilog, VHDL, TCL, lenguajes para la captura de restricciones, etc.) orientados al diseño hardware, aparte de la utilización de C/C++ y librerías específicas para la utilización de los diferentes dispositivos a utilizar en la plataforma de prototipado. Se valora que la utilización de estos flujos de diseño requiere una formación avanzada en el diseño de SoCs basados en FPGA.

Capítulo 4: Síntesis del diseño propuesto

1 Introducción

En este capítulo se describirán las etapas de síntesis y verificación del diseño propuesto. En particular se tratarán las síntesis tanto de alto nivel por la herramienta Cadence CtoS como la lógica por la herramienta Synplify Premier, detallando opciones de síntesis y flujo seguido, así como los resultados obtenidos. Además, se describirá brevemente las etapas de verificación funcional previas a cada síntesis, realizadas mediante el entorno de simulación de Cadence, tanto a nivel ESL como RTL.

El punto de partida del trabajo descrito en este capítulo es el diseño modelado en SystemC que describe la funcionalidad y jerarquía de un *DF* para H.264/SVC basado en el software de referencia Open SVC Decoder [11].

2 Verificación funcional a nivel ESL

La verificación funcional del modelo del DF se ha realizado mediante simulación. Se han planificado las simulaciones necesarias sobre el diseño en SystemC con el fin de garantizar que la descripción realizada coincide con las especificaciones de partida, es decir, que ante unos estímulos de entrada genera las mismas salidas que el software de referencia.

Para realizar esta verificación se ha creado un testbench que suministra los datos de entrada al DF atendiendo a las necesidades de protocolo de éste, descritas anteriormente como cinco buses independientes, y almacenando la salida del mismo en un fichero de texto.

Este fichero de salida podrá compararse con la salida generada por el software de referencia con el fin de asegurar que cada pixel en la imagen filtrada por ambos códigos coincide.

Los ficheros de entrada se generan instrumentalizando el código de referencia OpenSVC Decoder para que almacene en ficheros de texto cada uno de los parámetros necesarios para el filtrado así como la propia imagen sin filtrar antes de realizar dicho proceso. Además, tras el proceso de filtrado se almacena la salida en otro fichero con el fin de ser usado como salida de referencia. El esquema de verificación explicado se representa en la Figura 28.

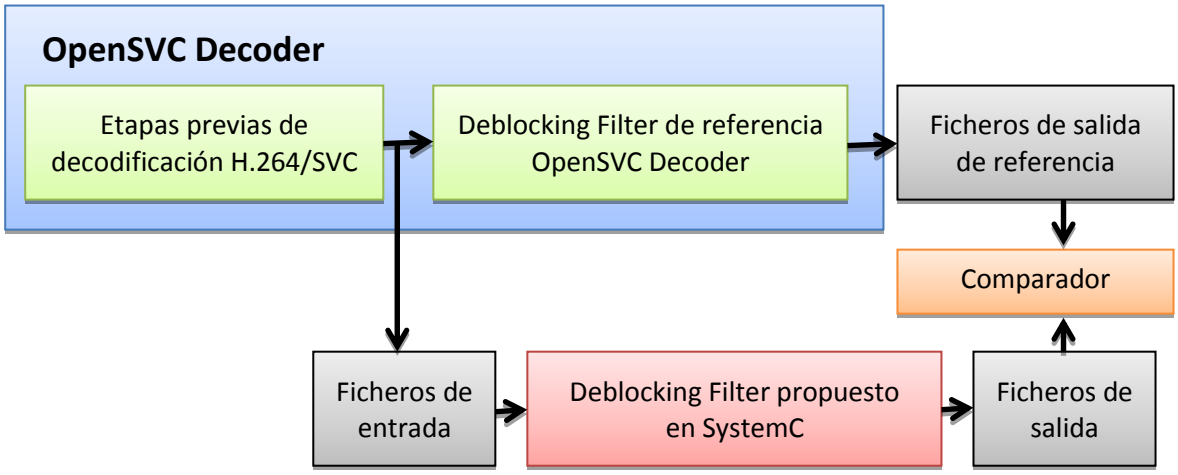

Figura 28. Esquema de verificación.

Una vez comprobado que el modelo del DF en alto nivel realizado en SystemC genera los parámetros e imagen idénticos en los ficheros de salida a los del software de referencia, se considera que el diseño es funcionalmente correcto, pasando a realizar la síntesis de alto nivel. Para todo el proceso de modelado se ha tenido en cuenta el subconjunto de síntesis soportado por el entorno CtoS, protegiendo, donde ha sido necesario mediante directivas del

preprocesador, aquella parte del modelo no soportada por la síntesis u orientada a otras herramientas no incluidas en el flujo de diseño explicado anteriormente (#ifndef __CTOS__ … #endif)

3 Síntesis de alto nivel

Para la síntesis de alto nivel, tal como se explicó en el capítulo anterior, se ha optado por hacer uso de la herramienta C-to-Silicon (CtoS) de Cadence, la cual ofrece la posibilidad de realizar la síntesis partiendo de un diseño en SystemC, y obteniendo como resultado la descripción RTL del sistema en Verilog.

Como ya se adelantó en el capítulo de metodología, las herramientas de síntesis requieren de una guía que les proporcione cierta información sobre el objetivo del sistema, pues existen muchas posibles soluciones que cumplen con los requisitos de funcionalidad. Es por ello, durante esta fase, CtoS exige que se especifiquen restricciones de diseño y directivas de síntesis, que se comentarán en el siguiente apartado.

3.1 Restricciones y directivas de la síntesis

A continuación se enumerarán aquellas restricciones de diseño y directivas de síntesis que deben especificarse durante la etapa de síntesis de alto nivel para la herramienta CtoS. Además se indicará la sintaxis para realizar esta especificación en la consola del programa o, lo que es más importante, para un script TCL (*Tool Command Language*) para futuras ejecuciones.

1. Frecuencia de reloj

Al comienzo de la etapa de síntesis es necesario indicar la frecuencia de reloj, así como su ciclo de trabajo (los valores se dan en términos de periodo de reloj y offset de flanco de bajada). Esta restricción guiará a la herramienta de síntesis para añadir más o menos ciclos de latencia así como a replicar o no lógica con el fin de alcanzar el objetivo de frecuencia de funcionamiento.

```
define_clock -name nombrereloj -period periodo -rise 0 -fall flancobajada
```

Los valores de periodo, flanco de subida y flanco de bajada se indicarán en picosegundos, como a continuación se muestra en el ejemplo para una señal de reloj llamada *clock* que tiene un periodo de 10ns (frecuencia de reloj de 100MHz) y un ciclo de trabajo del 50%.

```
define_clock -name clock -period 10000 -rise 0 -fall 5000
```

2. Bucles combinacionales

En el diseño en SystemC pueden existir, y de hecho son simulables, bucles enteramente combinacionales que, según su definición, deberían ejecutarse en su totalidad en un único ciclo de reloj. Es por ello que ha de indicarse que hacer en estos casos. Puede optarse por varias soluciones:

- **Desenrollar el bucle**. Realiza todas las iteraciones una detrás de la otra en un único ciclo, a costa de replicar la lógica interna del bucle una vez por cada iteración. Favorece una minimización del número de ciclos, aumentando los recursos *hardware* necesarios, así como la duración del periodo de reloj.
- **Romper el bucle.** Es el equivalente a añadir una sentencia `wait()` al final de cada iteración. Así, cada iteración se realiza en un ciclo de reloj. Es normalmente la acción por defecto.
- **Realizar un *pipeline*.** Rompe el bucle de forma que se tardan varios ciclos en realizarse, pero además, varias iteraciones se realizan a la vez, pero cada una ocupando una etapa del bucle. Esta solución mejora el rendimiento para bucles con gran carga en cada iteración, pero es ineficiente para bucles sencillos al añadir la lógica adicional para el control del *pipeline*.

Para el presente trabajo se realizará una acción de rotura del bucle en todos aquellos bucles combinacionales existentes, para lo cual a continuación se muestra la sintaxis TCL.

```
set combo_loops [find_combinational_loops]
foreach loop $combo_loops {
    break_combinational_loop  $loop
}
```

3. Funciones no planificables

En la descripción del diseño en alto nivel se permiten las llamadas a funciones globales. En general estas se resuelven por el planificador de la herramienta mediante la síntesis de un bloque independiente y un mecanismo de comunicación mediante un protocolo de petición y respuesta. Sin embargo, pueden presentarse ocasiones en las que esto no ocurre por diversas causas, como puede ser una latencia no constante o la llamada a la función desde distintos procesos concurrentes.

En estos casos, la herramienta obliga a realizar una acción de *inline*, es decir, reemplazar la llamada a la subrutina por una copia íntegra del código asociado a dicha función. Esta acción genera un aumento en el consumo de recursos. Ello requiere una modificación del código fuente en algunos casos, pero en otros es inevitable.

También se permite realizar un *inline* de las funciones que no producen estos conflictos. Sin embargo, llevar esta acción sobre todas las funciones del diseño suele conducir a una utilización

elevada de recursos de cara a la implementación hardware en un dispositivo físico. A continuación se muestra la sintaxis para realizar *inline* de una función específica.

```
inline /designs/nombre_proyecto/modules/nombre_modulo/behaviors/funcion1
```

4. Asignación de memorias

Si existen vectores de datos en la descripción SystemC del diseño, se debe decidir los recursos a utilizar para almacenarlos. Existen diferentes opciones, que se enumeran a continuación:

- **RAM interna**. Se usa como recursos los bloques BRAM o las LUTs de los SLICEM para almacenar la información. El uso de uno u otro depende de los ciclos de espera introducidos entre la disposición de la dirección y la lectura del dato.
- **Registros**. Todos los datos se almacenan en registros. Esta solución solo debe realizarse en casos de vectores de pocos elementos y en los cuales los elementos no son excesivamente grandes en términos de longitud de bits. Presenta como ventaja que es posible acceder a todos los elementos del vector de forma simultánea.
- **RAM de terceros**. Pueden imponerse como recurso de memoria la descripción HDL de una RAM de terceros.

El uso de registros o BRAM/LUTs dependerá de la disponibilidad de recursos del dispositivo y del uso que haga el resto del diseño.

5. Uso de DSPs

Para cada módulo puede indicarse si debe o no usar DSPs. En general, y dependiendo de la naturaleza del diseño, puede ser interesante que un módulo haga uso de los bloques DSPs disponibles en la FPGA para aquellas operaciones para los que han sido diseñados, y otros, menos importantes, usen lógica programable para realizarlas a costa de un mayor número de ciclos de latencia.

```
use_dsp /designs/$modulo
```

6. Relajación de la latencia

Esta opción permite a CtoS incrementar el número de ciclos de latencia de una función, con el fin de facilitar la planificación de la misma. Es importante no permitir esta acción sobre aquellas funciones dedicadas a implementar los protocolos de comunicación. Por ejemplo, no es conveniente permitir relajar la latencia de las funciones que hacen accesos a memoria, o que se dedican a leer y escribir sobre los puertos de control de comunicación, ya que añadir estados en medio de una de estas rutinas puede significar que el protocolo deje de ser funcionalmente

correcto. A continuación se muestra como desactivar la relajación para todas las funciones del diseño.

```
set behaviours [find $top_path/modules/*/behaviors/*]
foreach beh $behaviours {
    set_attr relax_latency "false" $beh
}
```

3.2 Automatización de la síntesis

Como se adelantó anteriormente CtoS permite lanzarse en modo *batch*, haciendo uso de *scripts* TCL. De esta forma, realizar la síntesis durante la etapa de depuración es mucho más rápido, sin la necesidad de elegir todos los parámetros anteriormente descritos en cada síntesis.

Para lanzar la herramienta en modo *batch*, se especificará el alias de la aplicación en dicho modo, el *script* a ejecutar y, opcionalmente, un fichero de *log* donde almacenar la salida del proceso.

```
ctos scripts/ctos.tcl -log log/ctos.log
```

En el fichero de *script* TCL existirán dos etapas diferenciadas. La primera es la que configura el proyecto de CtoS, indicando los ficheros fuente, la frecuencia de reloj, así como el dispositivo de prototipado final.

```
new_design $modulo
set_attr auto_write_models "true" /designs/$modulo
define_sim_config -model_dir "./model" /designs/$modulo
set_attr source_files [list src/$modulo.cpp] /designs/$modulo
set_attr compile_flags " -w -I../../include -I./src" /designs/$modulo
set_attr auto_save_dir $modulo /designs/$modulo
define_clock -name clock -period 10000 -rise 0 -fall 5000
set_attr implementation_target FPGA [get_design]
set_attr fpga_target [list Xilinx Virtex5 xc5vfx70t-1-ff1738]
[get_design]
set_attr fpga_install_path [exec which xst] [get_design]
set_attr fpga_work_dir "./fpga_work" [get_design]
```

La segunda parte del *script* es la que define las diferentes etapas de la síntesis previamente citadas en este apartado. A continuación se muestra un ejemplo.

```
set combo_loops [find_combinational_loops]
foreach loop $combo_loops {
    break_combinational_loop  $loop
}
foreach beh [find $top_path/modules/*/behaviors/*] {
    set_attr relax_latency "false" $beh
}
schedule -effort high -passes 200 /designs/$modulo
allocate_registers
write_rtl -recursive -o [concat ./[string trim $model]] $top_path
/modules/$modulo
```

3.3 Resultados en pasos tempranos del flujo de diseño

En flujos de diseño complejo como el presentado en este TFM, donde el proceso de síntesis y verificación puede necesitar mucho tiempo, es de especial importancia disponer de resultados en los primeros pasos realizados, de forma que puedan modificarse parámetros de los mismos sin tener que llegar al último punto.

Es por ello que, desde la prima etapa de la síntesis, es posible estimar el consumo de recursos y la frecuencia máxima de funcionamiento por parte de la herramienta de síntesis de alto nivel.

De esta forma en la Tabla 6, se presentan los datos de ocupación y frecuencia obtenidos por la herramienta CtoS tras la síntesis de alto nivel.

Tabla 6. Resultados obtenidos por CtoS.

Recurso/Parámetro	Cantidad/Valor
LUTs	149.920
Flip Flops	23.958
DSPs	1
Frecuencia	69,85 MHz

El valor de LUTs es en general bastante pesimista, ya que no tiene en cuenta la gran cantidad de recursos complejos existentes en la FPGA, suponiendo que toda la lógica es implementada mediante LUTs.

En general todos los valores dan una estimación de peor caso, que mejorará tras la etapa de síntesis lógica como se verá en los siguientes apartados. Sin embargo, esta información es de especial interés cuando los valores obtenidos en frecuencia y Flip Flops está muy fuera del rango esperado, obligando al diseñador a volver atrás, modificar el diseño y/o los parámetros de síntesis antes de seguir en la siguiente etapa del flujo de diseño.

4 Verificación funcional a nivel RTL

Una vez realizada la síntesis de alto nivel, el diseño pasa a estar constituido por un número de bloques en Verilog HDL. Para comprobar que la síntesis ha sido correcta, esto es, que el diseño escrito en HDL se comporta como se esperaba, se deben realizar simulaciones con el mismo

testbench de entrada que el diseño en alto nivel. Si todo es correcto, deben generarse los mismos patrones de salida.

Partiendo de que el diseño en SystemC era correcto, es decir, que para los parámetros de entrada e imágenes sin filtrar proporcionados por el *testbench*, las salidas correspondían con las del código de referencia, la verificación RTL pasa a basarse en una comparación entre las señales del modelo en alto nivel y el de nivel RTL.

Precisamente por esto, si los estímulos de entrada coinciden con los del *testbench* en alto nivel, la depuración del diseño a nivel RTL se simplifica notablemente, además de que se acelera, ya que basta con comparar las salidas y las formas de onda de ambas simulaciones.

Para este tipo de verificación, CtoS genera un *wrapper* en SystemC que instancia al modelo en Verilog generado, permitiendo luego realizar una simulación usando el mismo *testbench* que en SystemC. Este *wrapper* además, realiza dos instancias del diseño en cuestión, una del modelo original en alto nivel en SystemC y otro del sintetizado en Verilog. De esta forma, pueden presentarse en pantalla ambas señales y compararlas en el tiempo, haciendo que la etapa de verificación sea mucho más rápida.

Esta etapa implicará un proceso de múltiples iteraciones "verificación RTL ↔ síntesis de alto nivel" hasta conseguir que, el diseño sintetizado sea válido. Existen diferentes causas para que el diseño sintetizado no concuerde con la simulación de la descripción de alto nivel. Algunas de estas causas están descritas en el capítulo de síntesis de [34].

En la Figura 29 se muestra un ejemplo de simulación en el que se visualiza la entrada de datos en el *DF* y la salida de los datos filtrados, por los distintos puertos previamente comentados.

5 Síntesis lógica

Una vez comprobado el correcto funcionamiento del diseño a nivel RTL, el siguiente paso para poder implementarlo es realizar la síntesis lógica, consistente en realizar una transformación desde un nivel RTL al nivel de puertas lógicas, en este caso primitivas de Xilinx interconectadas formando un *netlist* completo en formato EDIF.

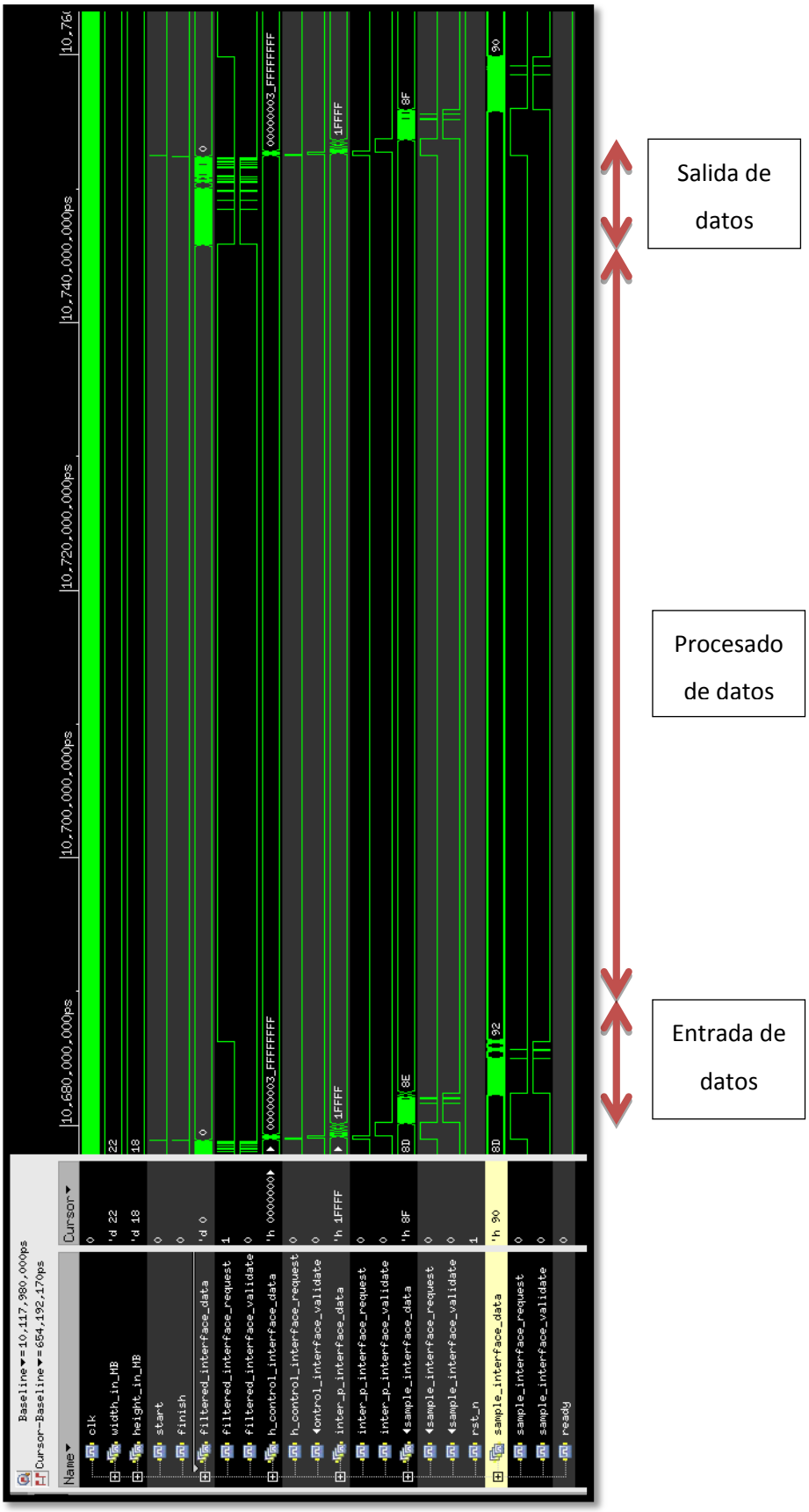

Figura 29. Verificación RTL del DF.

Este paso de síntesis lógica dentro del flujo de diseño está soportado dentro del entorno de Xilinx por la herramienta XST (Xilinx Synthesis Tecnology) [35] que permite realizar la síntesis lógica del CE en la plataforma generada desde XPS directamente a nivel RTL. Sin embargo, los resultados obtenidos por XST son, en general, peores a los de otros entornos especializados como es el caso de Synopsys Synplify Premier. Es por ello que se optado por el uso de esta herramienta.

5.1 Opciones de síntesis

A continuación se describen aquellas opciones a activar en las opciones de síntesis de la herramienta, con el fin de obtener los mejores resultados para el modelo de referencia con el que se trabaja.

Disable I/O Insertion

Consiste en impedir la inserción de bloques de entrada/salida (IOB) de la FPGA para las señales de entrada y salida del top del diseño. Para el caso que nos ocupa, es importante activarlo ya se trata de sintetizar un bloque IP que será integrado en la plataforma final que se diseña siguiendo otro flujo de diseño diferente. Por tanto las E/S del IP estarán conectadas a otras señales internas de la plataforma.

Por defecto, las herramientas de síntesis dispondrán de IOBs para los puertos del bloque de mayor nivel de jerarquía (top), entendiendo que dichas señales estarán asociadas a pines de la FPGA, con el fin de conectarlas a dispositivos externos.

FSM Compiler

Se trata de un optimizador de máquinas de estado. Synplify ofrece la posibilidad de optimizar la lógica de estado siguiente de las instancias de máquinas de estado del diseño que encuentre, siguiendo una estrategia diferente de codificación de estados en función del número de estos, según la siguiente tabla.

Tabla 7. Codificación de estados de FSM Compiler.

Número de estados	Tipo de codificación
< 5	Secuencial
5 – 24	One-Hot
> 24	Gray

Resource Sharing

Permite compartir recursos con el fin de reducir el consumo de recursos del dispositivo de prototipado, a costa de reducir la frecuencia.

Pipelining

Permite que varias operaciones se realicen a la vez sobre el mismo recurso, partiendo dicha ejecución en etapas, y permitiendo que cada dato se encuentre en una de ellas. En las tecnologías Virtex-5 de Xilinx, esta optimización va asociada a las memorias ROMs y a los multiplicadores del diseño.

Enable Advanced LUT Combining

Prepara el fichero *netlist* de salida para una posterior combinación de LUTs en los diseños sobre FPGAs de Xilinx.

6 Resultados

En este apartado se presentarán los resultados de área, frecuencia y potencia obtenidos de la síntesis total del diseño, es decir, tras la síntesis lógica. Para los resultados de área y frecuencia, estos son datos que se obtienen de la propia herramienta de síntesis. Para la medida de potencia consumida se hará uso de una herramienta específica del fabricante de la FPGA, Xilinx XPower Analyzer.

6.1 Resultados de área

En este apartado se estudiarán los recursos consumidos por el diseño, así como la distribución de los mismos en función de los bloques internos del *Deblocking Filter*.

En la Tabla 8 se encuentra, de forma jerárquica, el consumo de LUTs, registros, DSPs, y bloques de memoria.

Tabla 8. Consumo de recursos del *Deblocking Filter*.

	LUTs	Registros	BRAMs	DSPs
INTERFACE	637	446	0	0
PARAM_BS	9.552	9.523	0	0
PARAM_CLIP	1.911	877	0	1
LUMA_CORE	9.832	4.170	1	0
CHROMA_CORE	4.470	3.099	1	0
Memorias intermedias	790	244	8	0
Total *Deblocking Filter*	27.192	18.359	10	1

En la Figura 30 se representa el uso de LUTs de cada uno de los bloques que está compuesto el *DF* así como una representación relativa del uso de cada uno.

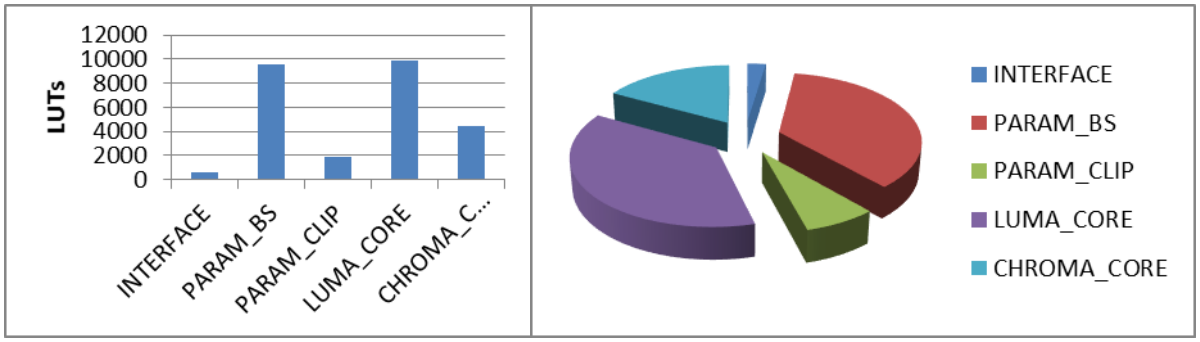

Figura 30. Uso de LUTs por bloque.

Se puede observar que los bloques que realizan las tareas de filtrado, LUMA_CORE y CHROMA_CORE consumen una gran parte de los recursos, y que, el cálculo de parámetros de Boundary Strengh está muy cerca de ser el bloque de mayor consumo de lógica.

En la Figura 31 se representa el mismo esquema para el uso de registros.

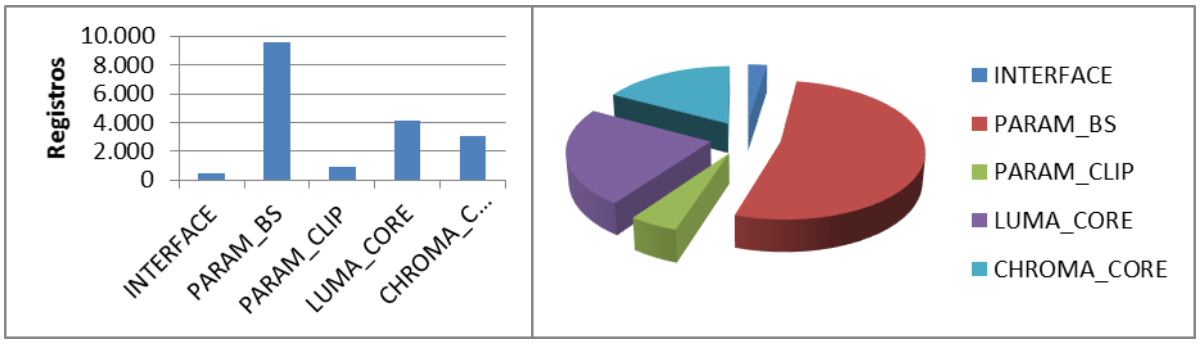

Figura 31. Uso de registros por bloque.

Se observa que, en el caso de registros el uso más grande se da en el bloque de cálculo de parámetros Boundary Strength. Esto se debe, en parte, a que las memorias internas a este bloque no son del tamaño suficiente como para que la herramienta de síntesis les dedicase una memoria de bloque RAM interna, y por lo tanto ha sido mapeada sobre registros.

En las figuras Figura 32 y Figura 33 se observa como el uso de BRAMs se limita a los bloque de LUMA_CORE y CHROMA_CORE ya que tienen memorias lo suficientemente grandes como para almacenar un macrobloque y sus vecinos. En el caso de los DSP, es únicamente usado en el bloque PARAM_CLIP ya que requiere de un cálculo de número de macrobloque por cuadro en función del ancho y alto en macrobloques del mismo.

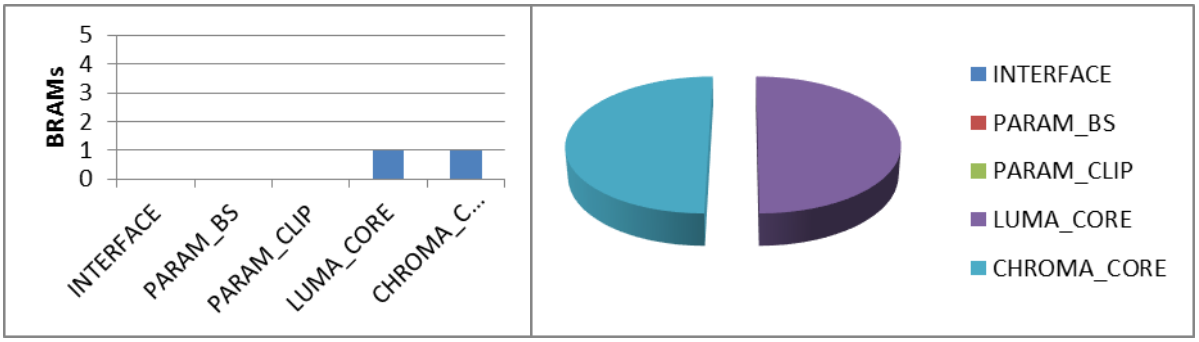

Figura 32. Uso de BRAMs por bloque.

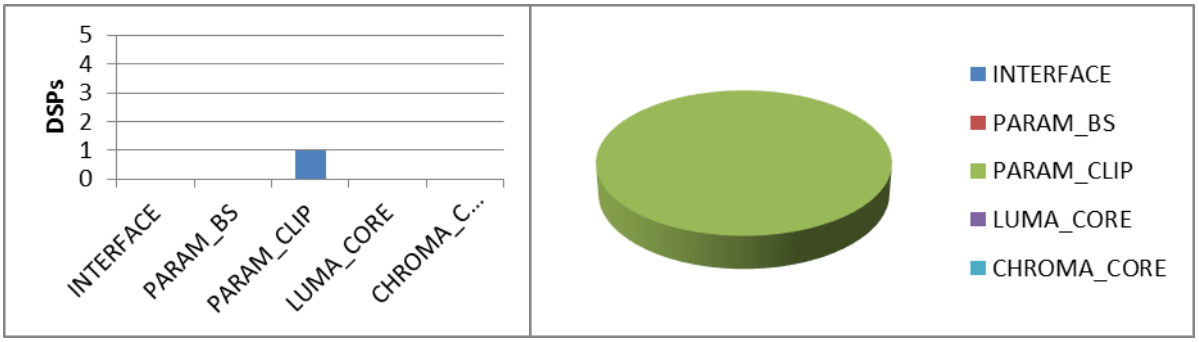

Figura 33. Uso de DSPs por bloque.

6.2 Resultados de frecuencia

En la fase de *profiling* de la descripción SystemC de partida, se tomó como frecuencia nominal de la simulación un reloj de 100 MHz. Esta definición, en las simulaciones, aceptan cualquier frecuencia y su única interpretación es que, entre flancos del diseño, se contabilizará dicho periodo. Esto implica que, en la descripción SystemC, a la hora de realizar la simulación, se podrá suponer relojes que durante la implementación resulten inviables debido a rutas críticas.

Tras la síntesis lógica se dispone de una frecuencia máxima de funcionamiento estimada por la herramienta de síntesis en cuestión. Cabe destacar que, tras la implementación física (*Place & Route* en FPGA) esta frecuencia puede disminuir en función de colocación de los recursos y de la congestión de ruteado.

Así, la frecuencia máxima de funcionamiento así como datos relativos a la ruta crítica de describen en la Tabla 9.

Tabla 9. Resultados de frecuencia.

Frecuencia máxima (MHz)	120,3
Periodo mínimo (ns)	8,315
Bloque de la ruta crítica	LUMA_CORE
Tipo de ruta crítica	BRAM to Register

Independientemente de este resultado, y visto el alto consumo de recursos existente, se tomará 100 MHz como frecuencia de funcionamiento para la implementación física, a fin de evitar rutas demasiado críticas que dificulten las tareas de *Place & Route*.

6.3 Resultados de potencia

Para la realización de la medida del consumo de potencia por parte del bloque correspondiente al *DF* se ha hecho uso de la herramienta Xilinx XPower Analyzer.

Con el fin de mejorar la precisión de las medidas, se ha proporcionado como dato de entrada el diseño completamente ruteado, así como información de actividad en los nodos obtenidos a partir de la simulación a nivel RTL del diseño. Para disponer del diseño ruteado, se ha realizado la síntesis física o implementación con PlanAhead.

El diseño ruteado se ha proporcionado en el formato propietario de Xilinx NCD, mientras que la información relativa a la actividad de los nodos en formato VCD (*Value Change Dump*).

Accediendo a los resultados obtenidos detallados por jerarquía, se puede aislar los concernientes únicamente al *DF*. En la Tabla 10 se especifican las medidas obtenidas.

Tabla 10. Consumo de potencia del *Deblocking Filter*.

Bloque	Consumo (mW)
Deblocking Filter	18,69

6.4 Comparativas

Una vez obtenidos los resultados es importante realizar una comparativa con otros trabajos relacionados. En particular se realizará una comparativa con los resultados obtenidos de la síntesis, siguiendo la misma metodología de diseño, para un Deblocking Filter para H.264/AVC. De esta forma se podrá realizar una estimación de cuanto incrementa en complejidad la escalabilidad al filtrado de H.264.

Los resultados han sido obtenidos del trabajo [36], el cual utilizó una descripción en SystemC escrita por el mismo diseñador y realizando la síntesis con las mismas herramientas que el presente trabajo.

En la Tabla 11 se muestran los valores de recursos utilizados y frecuencia máxima de funcionamiento para ambos diseños.

Tabla 11. Comparativa entre *DF* para SVC y AVC.

	LUTs	FlipFlops	BRAMs	DSPs	Frecuencia (MHz)
Este trabajo	27192	18359	10	1	120,3
[36]	9550	9882	3	0	112,2

Como puede observarse de los valores obtenidos, existe un aumento importante de consumo de recursos, multiplicándose en un factor de entre x2 y x3 tanto para LUTs, Flip Flops como BRAMs. Este incremento se ve representado en la Figura 34.

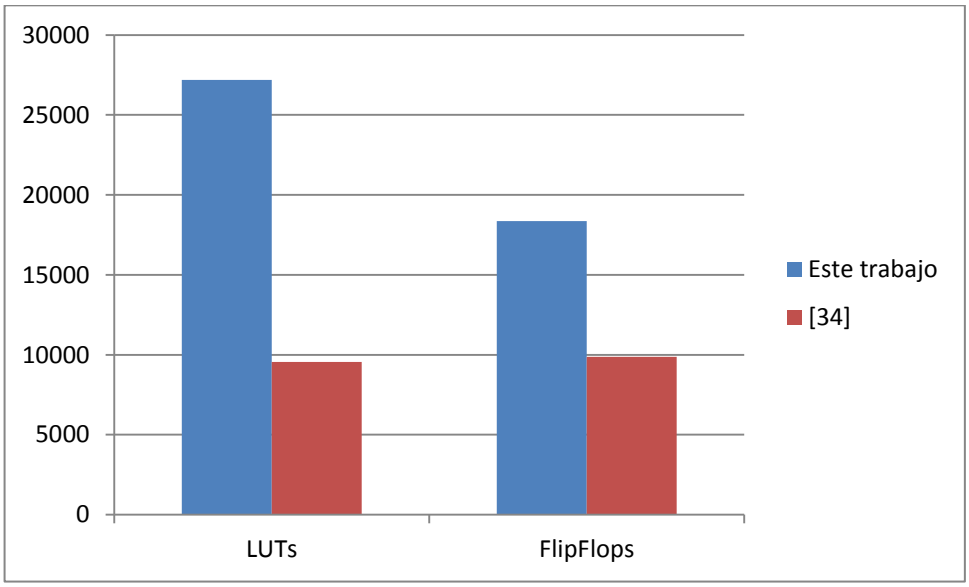

Figura 34. Incremento en LUTs y FFs de AVC a SVC.

Puesto que el objetivo principal de PCCMuTe es controlar el consumo en terminales multimedia portátiles, es de especial interés comparar los resultados de consumo del diseño utilizado con el de otros diseños del estado del arte.

Por ello, se presenta en la Tabla 12 los resultados de consumo y frecuencia de funcionamiento de este trabajo y otros trabajos similares, así como sus tecnologías.

Tabla 12. Comparativa de consumo de potencia.

Referencia	Tecnología	Frecuencia (MHz)	Consumo de potencia (mW)
[37] Nadeem et al.	Xilinx Virtex-2 (CMOS 130nm)	76	43.00
[38] Parlak et al.	Xilinx Virtex-2 (CMOS 130nm)	72	259.13
Este trabajo	Xilinx Virtex-5 (CMOS 65nm)	100	18.69

Se puede concluir de la tabla que el diseño propuesto presenta resultados de consumo óptimos al ser comparados con diseños orientados al bajo consumo. Debe tenerse en cuenta en todo caso que, al tratarse de tecnologías diferentes, el consumo se ve afectado. De [39] se puede obtener que existe una disminución del consumo del 50% entre tecnologías de 130nm y 65nm.

7 Conclusiones

En este capítulo se ha descrito el proceso de síntesis de la descripción de partida en SystemC en un fichero EDIF de *netlist*. Esta transformación ha permitido obtener datos cuantitativos referentes tanto al diseño, como a la metodología empleada.

Las estimaciones más importantes que nos permiten hacer los resultados obtenidos es que la mayor complejidad del *DF* se divide entre el filtrado de las muestras y la obtención de la fuerza de filtrado, estando prácticamente en igualdad de recursos. Esta información es muy importante para futuras fases de optimización así como de base de conocimiento para futuros diseños e implementaciones de este bloque del decodificador.

El *DF* funciona a una frecuencia máxima de 120,3 MHz, utilizando el 60% de la FPGA y consumiendo 18,69 mW. El *DF* presenta una latencia máxima de 74 µs por MB y su frecuencia máxima de funcionamiento viene limitada por el bloque LUMA_CORE.

Se obtiene además que existe un incremento en consumo de recursos de hasta un 200% entre implementaciones AVC y SVC del *DF*.

Capítulo 5: Adaptación y validación

1 Introducción

En este capítulo se presenta el diseño de una plataforma sobre FPGA que permita la validación del bloque *DF* sintetizado con anterioridad. Las necesidades de este tipo de plataformas son, principalmente, la posibilidad de leer y suministrar los datos de entrada al bloque a validar, así como recoger los datos de salida para almacenarlos y/o representarlos.

De esta forma, para este trabajo, se presentará una plataforma capaz de leer los datos de entrada de un medio de almacenamiento de ficheros como es una tarjeta CompactFlash, el almacenamiento de la salida en el mismo formato, y la representación de las tramas filtradas en una pantalla.

También se presentan los pasos necesarios para el depurado del sistema en ejecución en tiempo real mediante el uso de analizadores lógicos, y el software asociado. En ello se engloba, tanto la interconexión del analizador lógico a los pines que se quieran representar, así como al uso del software de captura y representación, donde se configuran además las condiciones de disparo.

2 Arquitectura de la plataforma

La plataforma diseñada está compuesta de un microprocesador que sirve de unidad de control al resto de bloques. Además existen otros bloques que sirven de comunicación con dispositivos externos como la tarjeta de memoria flash, memoria DDR, etc. De esta forma, los bloques que compondrán la plataforma se enumeran a continuación:

1. **Bloque de procesador**. Se trata de un IP que se encuentra empotrado en la FPGA. El bloque en cuestión contiene, además del PowerPC 440, otros módulos útiles que se han incluido con el fin de no consumir lógica adicional programable del resto del dispositivo. Para mejorar los accesos a memoria por parte del procesador y de otros posibles periféricos, la conexión con esta se realiza a través de una matriz de interconexión (*crossbar switch*), la cual acepta peticiones de transferencia de las cachés de datos e instrucciones del procesador, así como de cuatro bloques DMA y dos interfaces esclavas PLB, que se encuentran dentro del bloque de procesador. Estas transferencias pueden tener como destino la interfaz del controlador de memoria o una interfaz maestra PLB.
2. **Controlador de memoria DDR2**. Este bloque tiene por objeto comunicarse directamente con la memoria externa de la FPGA y que servirá de memoria principal para el PowerPC 440 de la plataforma. El bloque de procesador se conecta al controlador a través de la interfaz de controlador de memoria (Memory Controller Interface – MCI).
3. **Controlador de interrupciones**. El PowerPC 440 tiene el inconveniente de únicamente tener un puerto de un bit de ancho de interrupción. Por lo tanto, se hace necesaria la inclusión de un controlador que tenga como entrada varias interrupciones de dispositivos periféricos, y genere una salida que sirva de señal de interrupción para el bloque de procesador.
4. **Controlador SysACE**. El controlador System ACE representa la interfaz entre el bus PLB y el microprocesador de interfaz (MPU) del periférico System ACE Compact Flash. El circuito integrado encargado de acceder a la tarjeta se encuentra disponible en la placa de prototipado, siendo el controlador descrito el encargado de servir de interfaz entre dicho circuito externo y el bus integrado en la FPGA. Esta tarjeta de memoria es usada como sistema de ficheros para el almacenamiento de los datos de entrada del *DF*, así como para la escritura de los resultados.

5. **Transceptor serie UART**. Para poder monitorizar el sistema, se ha optado por usar una comunicación serie. Así, cualquier mensaje del sistema será enviado a través del puerto serie a un equipo de monitorización. Este periférico, como los demás, tiene una interfaz esclava PLB, y dispone de diferentes atributos configurables, como son por ejemplo, la velocidad de transmisión, el uso de paridad, etc.
6. **Analizador lógico integrado**. Se trata de un bloque, implementado a partir de lógica configurable, que sirve para monitorizar las señales internas del sistema, como lo haría un analizador lógico. Al ser un bloque síncrono, se le aplicarán las mismas restricciones temporales que al resto del diseño. Las señales del sistema se conectan a las entradas del bloque ILA (*Integrated Logic Analyzer*), y estas se capturan en tiempo real, a la frecuencia del diseño que las contenga. Antes de que el sistema sea implementado, se seleccionarán el número de señales a capturar así como el número de muestras a capturar. La comunicación con el bloque ILA se realiza a través del puerto JTAG de la FPGA y a través del bloque de control ICON, que realiza la tarea de comunicación entre los bloques del analizador lógico integrado (ILA) y el bloque JTAG *Boundary Scan* de la FPGA, para poder controlar dichos bloques a través del puerto de programación del dispositivo. Este analizador estará conectado al bus LocalLink de comunicación entre el *DF* y el bloque DMA de forma que puedan capturarse los datos de entrada y de salida del mismo.
7. **Controlador de pantalla**. Es el bloque encargado de generar las señales de control para la representación de imágenes a través de la pantalla TFT usando una interfaz DVI. Para ello, se reserva una zona de memoria del doble de tamaño de una imagen para implementar un buffer de memoria *ping-pong* (*double buffering*), donde almacenar las muestras en formato RGB. Este bloque dispone de una interfaz maestra PLB que le permite, a través de una de las interfaces esclavas PLB del procesador, acceder a la memoria principal off-chip SDRAM.

Para conectar el bloque a validar con el resto de la plataforma se ha decidido hacer uso de una interfaz de tipo DMA, de forma que no se congestione el bus de comunicaciones global. A su vez, el controlador de pantalla se conecta a la memoria principal a través del *crossbar switch* del bloque de procesador, mediante una conexión PLB dedicada donde dicho *crossbar* hace de esclavo.

Un diagrama de bloques donde se encuentran todos estos componentes así como su conexión se detalla en la Figura 35.

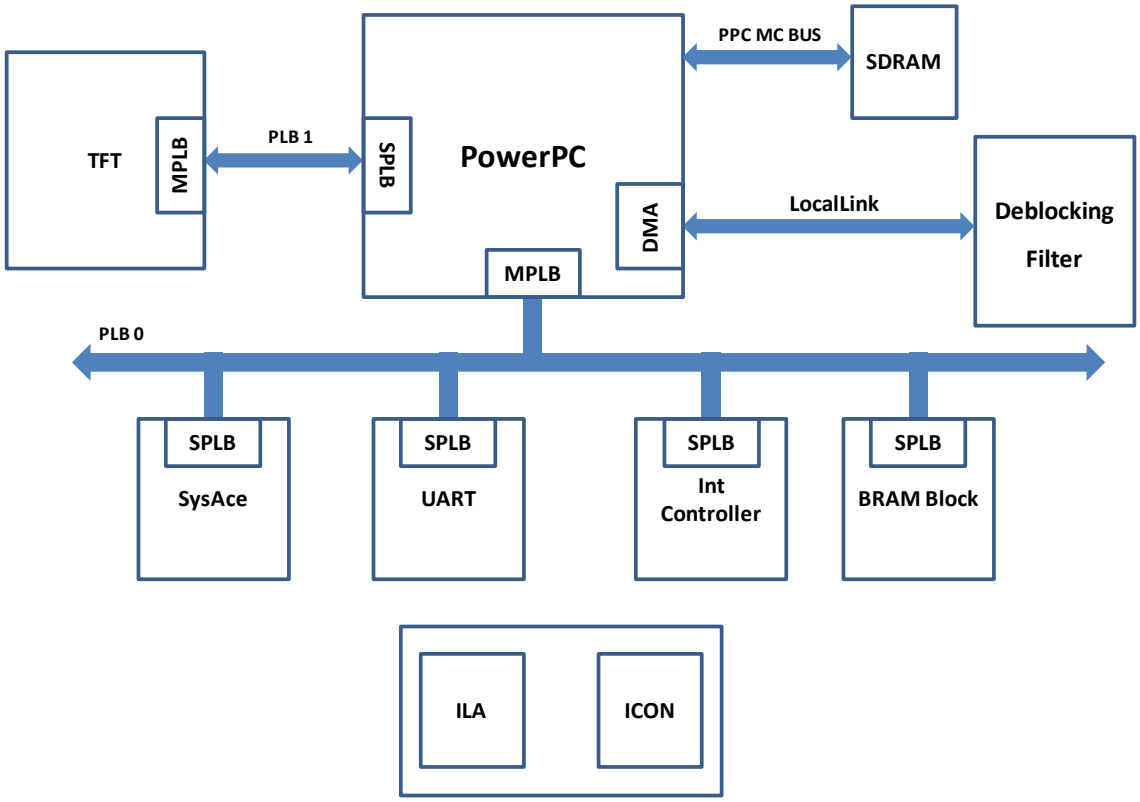

Figura 35. Diagrama de bloques de la plataforma de validación.

3 Adaptación de interfaces

Como ya se adelantó anteriormente, la interfaz del bloque *DF* con el exterior se basa en cinco buses independientes, con un protocolo optimizado específicamente para mejorar las transferencias de parámetros y contenido de la imagen. Debido a ello, para poder conectarlo al resto de la plataforma es necesario diseñar un módulo de adaptación o *wrapper*.

El *wrapper* está compuesto de tres partes principales. En primer lugar debe responder a las peticiones del *Deblocking Filter*, es decir, suministrar los datos que este solicite. Además deberá recibir la salida del *DF* y hacer lo apropiado con ella.

Por otro lado, la conexión a los bloques DMA de la FPGA Virtex 5 utilizada se basa en conexiones punto a punto con protocolo LocalLink, centrada en ráfagas. Teniendo en cuenta que, cada envío de una ráfaga por parte del DMA requiere de una cierta señalización por parte del microprocesador, es de especial interés realizar el mínimo de transacciones. Es por ello que se ha decidido realizar una transacción DMA por cada *frame*, en lugar de por cada macrobloque.

Además, esto permitirá responder a las peticiones de datos de vecinos de cada macrobloque, al disponer de los datos de todo un *frame* en el bloque de adaptación de interfaces.

A partir de lo que se ha comentado, la estructura del módulo está compuesta de tres bloques principales: uno que atienda al protocolo definido en la interfaz del *Deblocking Filter*, uno que interprete y aplique el protocolo LocalLink para la comunicación con el bloque DMA, y una memoria intermedia con tamaño suficiente para almacenar los datos asociados a un *frame*. Esta arquitectura se representa en la Figura 32.

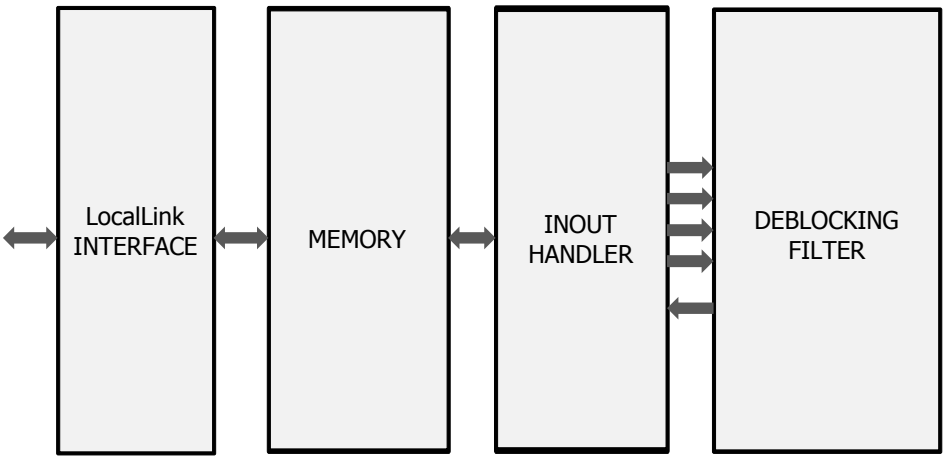

Figura 36. Arquitectura de adaptación de interfaces.

3.1 Arquitectura de adaptación

Como se adelantó anteriormente, la arquitectura de adaptación está compuesta por tres módulos. Los módulos con funcionalidad, es decir, la interfaz LocalLink y el manejador de peticiones de entrada y salida del *DF* fueron escritos en SystemC y sintetizados con el mismo flujo que el resto del *DF*. La memoria fue directamente escrita en Verilog con el fin de minimizar el *overhead* de recursos introducidos por las herramientas de síntesis. A continuación se presentará detalladamente el diseño de cada uno de estos.

3.1.1 LocalLink INTERFACE

Es el encargado de implementar el protocolo LocalLink y así poder comunicarse con el bloque DMA. El protocolo LocalLink es un sencillo protocolo de envío de ráfagas con cabecera o cola (en función del sentido de la comunicación). En la Tabla 13 se muestra el conjunto de puertos de los que dispone el módulo en cuestión.

Tabla 13. Puertos del bloque LocalLink INTERFACE.

Entrada / Salida	Tipo	Nombre	Bloque con el que comunica
sc_in	bool	clk	
sc_in	bool	rst_n	
sc_out	bool	ll_dst_rdy_tx_n	DMA
sc_in	bool	ll_src_rdy_tx_n	DMA
sc_in	bool	ll_sof_tx_n	DMA
sc_in	bool	ll_sop_tx_n	DMA
sc_in	bool	ll_eof_tx_n	DMA
sc_in	bool	ll_eop_tx_n	DMA
sc_in	sc_uint<32>	ll_data_tx	DMA
sc_in	sc_uint<4>	ll_rem_tx	DMA
sc_in	bool	ll_dst_rdy_rx_n	DMA
sc_out	bool	ll_src_rdy_rx_n	DMA
sc_out	bool	ll_sof_rx_n	DMA
sc_out	bool	ll_sop_rx_n	DMA
sc_out	bool	ll_eof_rx_n	DMA
sc_out	bool	ll_eop_rx_n	DMA
sc_out	sc_uint<32>	ll_data_rx	DMA
sc_out	sc_uint<4>	ll_rem_rx	DMA
sc_out	sc_uint<17>	addr	MEMORY
sc_out	sc_uint<32>	data_wr	MEMORY
sc_out	bool	ce	MEMORY
sc_out	bool	we	MEMORY
sc_in	sc_uint<32>	data_rd	MEMORY
sc_in	bool	read_handler_request	INOUT HANDLER
sc_out	bool	read_handler_validate	INOUT HANDLER
sc_in	bool	write_handler_request	INOUT HANDLER
sc_out	bool	write_handler_validate	INOUT HANDLER
sc_out	sc_uint<16>	width_in_MB	DF/INOUT HANDLER
sc_out	sc_uint<16>	height_in_MB	DF/INOUT HANDLER

La ejecución de la interfaz LocalLink consiste en esperar las peticiones de lectura o escritura del INOUT HANDLER. Estas peticiones indicarán si la interfaz debe recibir un nuevo *frame* sin filtrar del DMA, o si debe enviar el actual *frame* ya filtrado al DMA. Además, mantendrá en los puertos de salida la altura y anchura en MB del *frame* actualmente en memoria como información necesaria para el DF y para el INOUT HANDLER. Por último tiene una interfaz con la memoria de lectura/escritura. La memoria fue diseñada con un ancho de 32 bits con el fin de hacerla coincidir con el ancho del bus LocalLink del DMA y evitar usar otros dominios de reloj.

Información sobre la implementación de LocalLink así como su funcionamiento puede encontrarse en [40] y en [34].

3.1.2 MEMORY

El bloque de memoria consiste en una memoria dimensionada con el fin de almacenar toda la información necesaria para el filtrado de un *frame*, es decir, la imagen sin filtrar y todos los datos que el *DF* necesita para realizar su labor.

Como se indicó en el apartado anterior, es una memoria de ancho 32 bits. Sin embargo, debido a que el filtrado de la croma requiere de realizar escrituras de tamaño 16 bits (durante el filtrado de un MB de croma, son modificados además dos píxeles de sus vecinos superior e izquierdo). Es por ello que se diseñó la memoria como cuatro memorias de tamaño 8 bits concatenadas, permitiendo así la escritura sobre cualesquiera de ellas. En la Tabla 14 se detallan los puertos de este bloque.

Tabla 14. Puertos del bloque MEMORY.

Entrada / Salida	Tipo	Nombre	Bloque con el que comunica
input		CLK	
input		RSTn	
input		CE1	INOUT HANDLER
input		CE0	LocalLink INTERFACE
input		WE1	INOUT HANDLER
input		WE0	LocalLink INTERFACE
input	[3:0]	BE1	INOUT HANDLER
input	[31:0]	D1	INOUT HANDLER
input	[31:0]	D0	LocalLink INTERFACE
input	[16:0]	A1	INOUT HANDLER
input	[16:0]	A0	LocalLink INTERFACE

output	[31:0]	Q1	INOUT HANDLER
output	[31:0]	Q0	LocalLink INTERFACE

Como se observa en la Tabla 14 la señal BE1, de ancho 4 bits, es la que indica que bytes de los cuatro que componen una palabra, deben sobrescribirse. Esta señal no está presente del lado de la interfaz LocalLink ya que esta siempre escribe las palabras de 32 bits recibidas del DMA.

3.1.3 INOUT HANDLER

Es el bloque más complejo de los tres necesarios para la adaptación, así como el de mayor inteligencia. Es el encargado de conocer el número de macrobloque actual, así como de si ya se ha terminado de filtrar el *frame* actual, en cuyo caso así se lo indicará a la interfaz LocalLink para que este envíe al DMA el *frame* actual y solicite los datos del siguiente.

La mayor complejidad de este bloque reside en el cálculo de direcciones. Téngase en cuenta que los datos en memoria están ordenados a nivel de frame, de izquierda a derecha y de arriba abajo. Esto quiere decir que, para acceder a los píxeles de un macrobloque, se debe dar un salto en memoria hasta la siguiente línea del frame. Un ejemplo para un frame QCIF queda reflejado en la Figura 37.

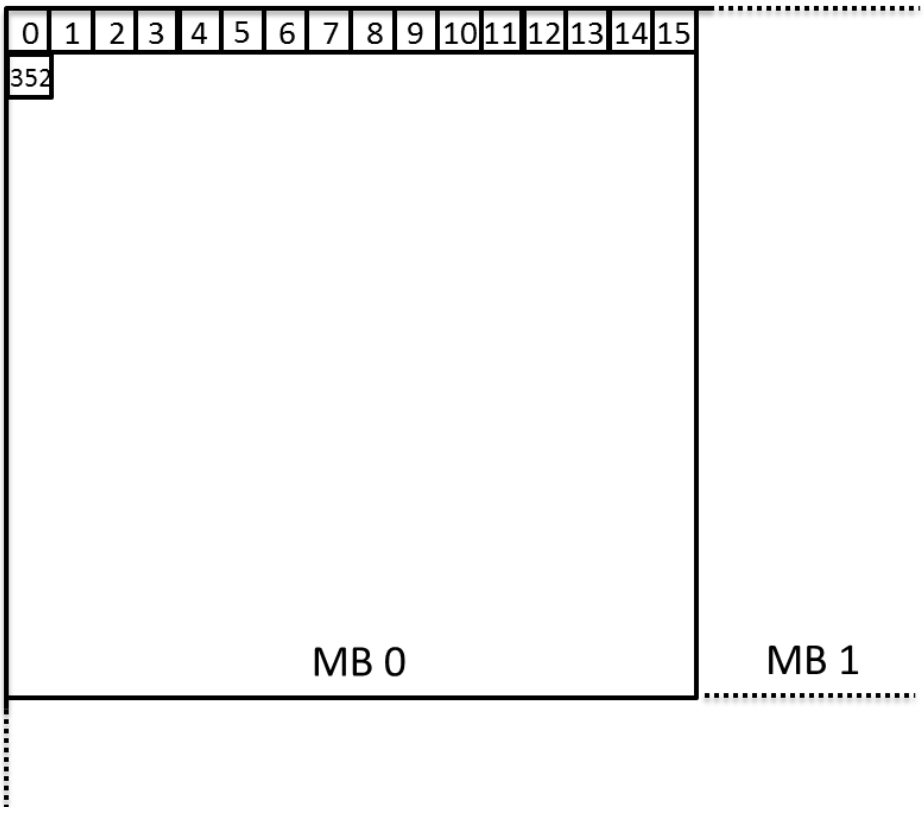

Figura 37. Disposición de datos en memoria.

Como puede observarse en esta figura, el decimosexto dato del primer macrobloque de la imagen, no se encuentra consecutivo con el decimoquinto. Estos cálculos de direcciones de memoria se hacen incluso más complejos en la búsqueda de vecinos.

En la Tabla 15 se enumeran los puertos de este bloque.

Tabla 15. Puertos del bloque INOUT HANDLER.

Entrada / Salida	Tipo	Nombre	Bloque con el que comunica
sc_in	bool	clk	
sc_in	bool	rst_n	
sc_out	bool	start	DF
sc_in	bool	finish	DF
sc_in	bool	ready	DF
sc_out	sc_uint<64>	h_control_interface_data	DF
sc_in	bool	h_control_interface_request	DF
sc_out	bool	h_control_interface_validate	DF
sc_out	sc_uint<17>	inter_p_interface_data	DF
sc_in	bool	inter_p_interface_request	DF
sc_out	bool	inter_p_interface_validate	DF
sc_out	sc_uint<8>	sample_interface_data	DF
sc_in	bool	sample_interface_request	DF
sc_out	bool	sample_interface_validate	DF
sc_out	sc_uint<8>	neighbour_sample_interface_data	DF
sc_in	bool	neighbour_sample_interface_request	DF
sc_out	bool	neighbour_sample_interface_validate	DF
sc_out	sc_uint<8>	filtered_interface_data	DF
sc_in	bool	filtered_interface_request	DF
sc_out	bool	filtered_interface_validate	DF
sc_out	bool	ce	MEMORY
sc_out	bool	we	MEMORY
sc_out	sc_uint<4>	be	MEMORY
sc_out	sc_uint<17>	addr	MEMORY
sc_in	sc_uint<32>	data_rd	MEMORY

sc_out	sc_uint<32>	data_wr	MEMORY
sc_out	bool	read_handler_request	LocalLink INTERFACE
sc_in	bool	read_handler_validate	LocalLink INTERFACE
sc_out	bool	write_handler_request	LocalLink INTERFACE
sc_in	bool	write_handler_validate	LocalLink INTERFACE
sc_in	sc_uint<16>	width_in_MB	LocalLink INTERFACE
sc_in	sc_uint<16>	height_in_MB	LocalLink INTERFACE

Como se observa en la tabla los puertos se dividen en tres señales de control con el *DF*, las cinco interfaces ya descritas del *DF* a las que el bloque responde, y la interfaz con la memoria. Como ya se ha adelantado, la principal función de este módulo es, cuando recite una petición de datos por el DF, buscar dichos datos en memoria para lo que debe calcular la dirección de memoria equivalente y suministrarlos. Los datos filtrados que recibe del *DF* son escritos en memoria sobrescribiendo la imagen sin filtrar.

4 Proceso de validación

En este apartado se presentará el proceso de validación realizado sobre la plataforma FPGA. Por un lado se explica la aplicación *software* que realizará el control del envío de datos al *DF* así como decidir qué hacer con los datos de salida.

4.1 Recorrido de los datos

Se explica a continuación el recorrido realizado por los datos en la plataforma definida, desde que son leídos desde la CF hasta que se muestran en la TFT una vez filtrados:

1. El primer movimiento de datos consiste en la lectura de los datos de un *frame* desde la memoria flash de la tarjeta externa y su almacenamiento en memoria principal, que para el caso de la arquitectura propuesta se encuentra en memoria DDR. Este primer paso queda representado en la Figura 38.
2. El segundo paso en el recorrido de datos, después de que estos hayan sido ordenados por el PowerPC, consiste en enviarlos a través de la interfaz LocalLink por parte del DMA al *DF*, representado en la Figura 39. Los datos se filtran en el DF.

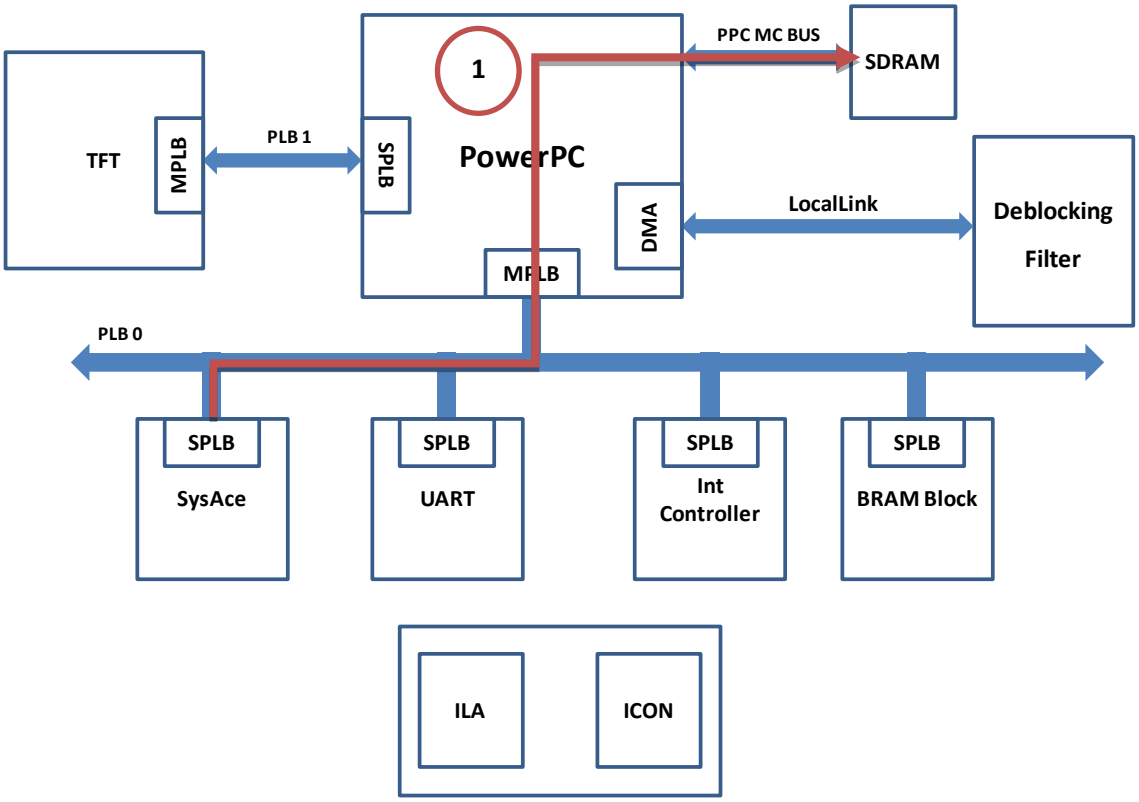

Figura 38. Primer paso del recorrido de datos.

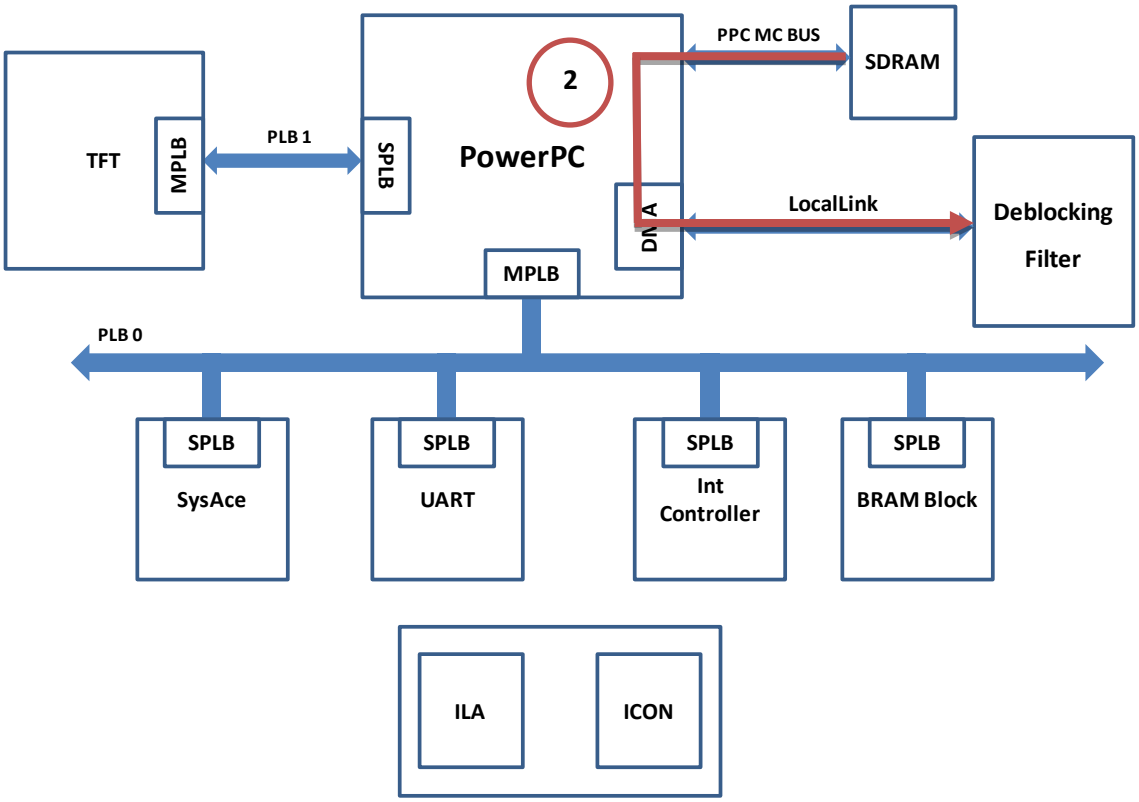

Figura 39. Segundo paso del recorrido de datos.

3. El tercer paso consistirá en recoger la salida filtrada que será almacenada por el bloque DMA en memoria principal, tal y como se representa en la Figura 40.
4. Por último, el cuarto y último paso puede diferir en función de si se quiere mostrar los datos por pantalla o si se quiere almacenar la salida en la memoria flash. Ambas opciones se representan en la Figura 41 y Figura 42 respectivamente.

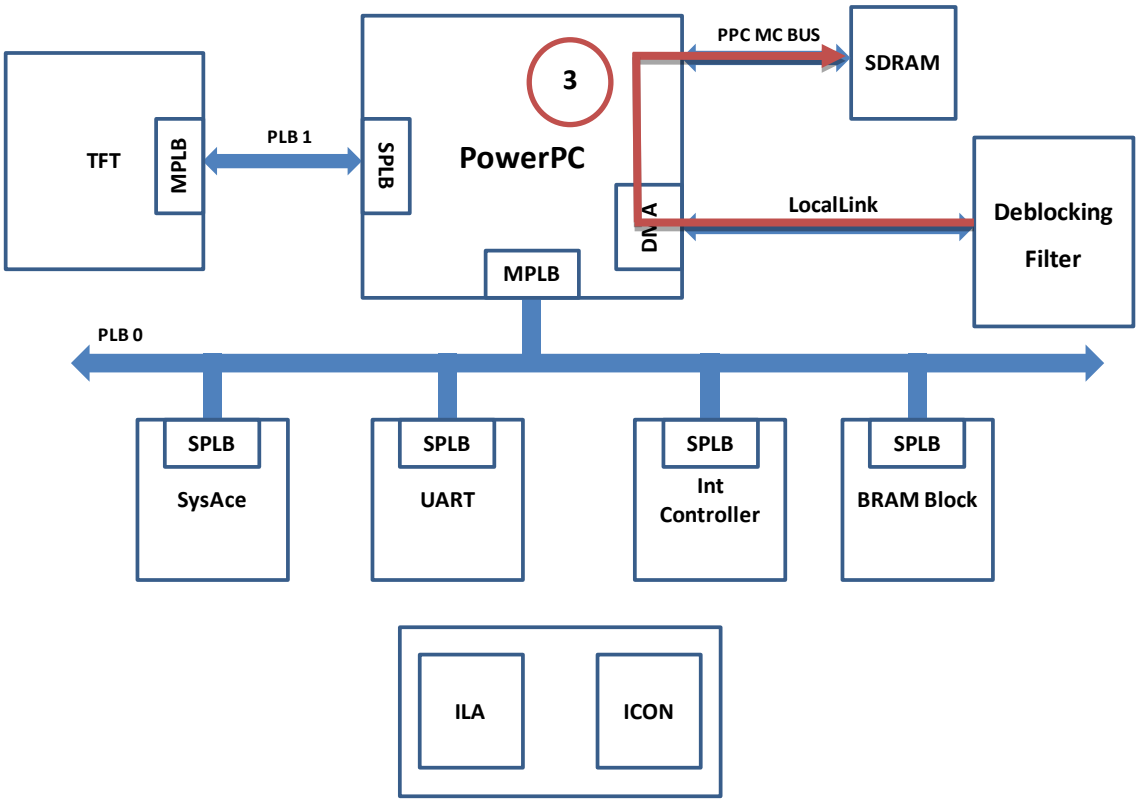

Figura 40. Tercer paso del recorrido de datos.

4.2 Rutinas software

Para poder enviar los datos a través del DMA así como recibirlos, leer datos de la CompactFlash o representar los datos en la pantalla conectada por DVI se han tenido que utilizar ciertas funciones de librería que serán descritas a continuación.

- XLlDma_Initialize(). Es la función que inicializa un bloque DMA y lo asocial a una variable que sera el identificador virtual del bloque *hardware*.

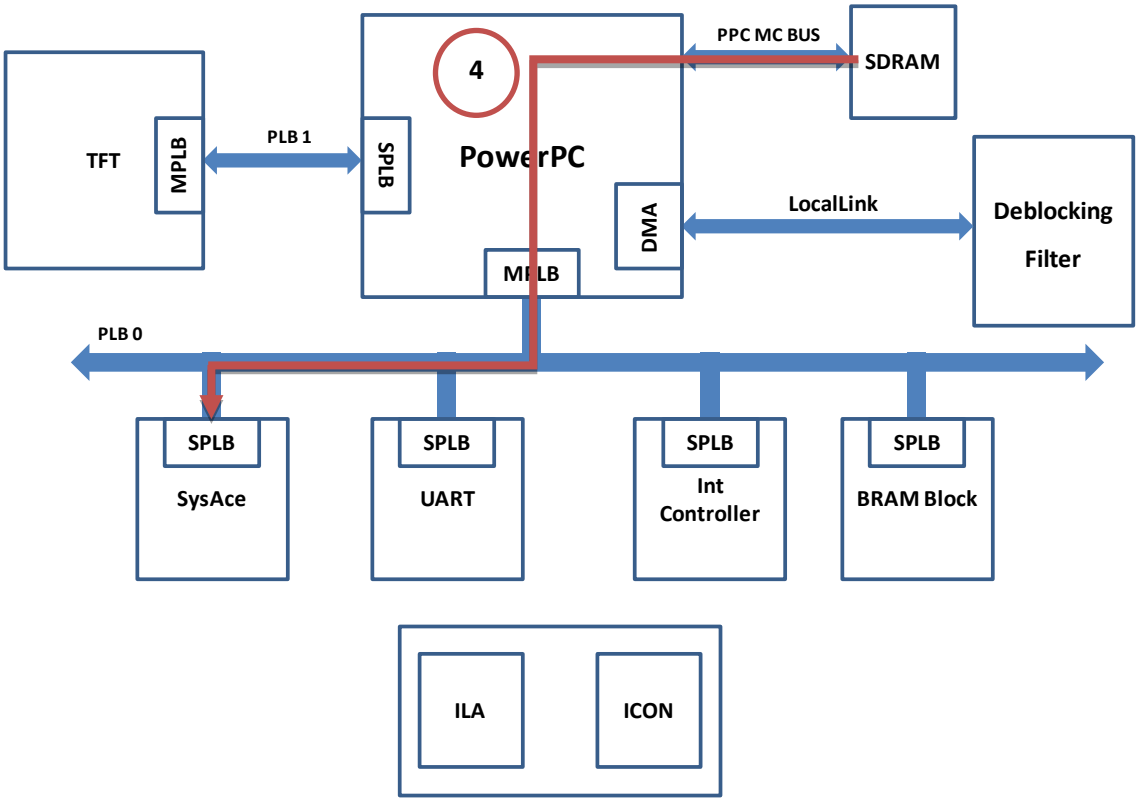

Figura 41. Cuarto paso del recorrido de datos, para el almacenamiento en memoria flash.

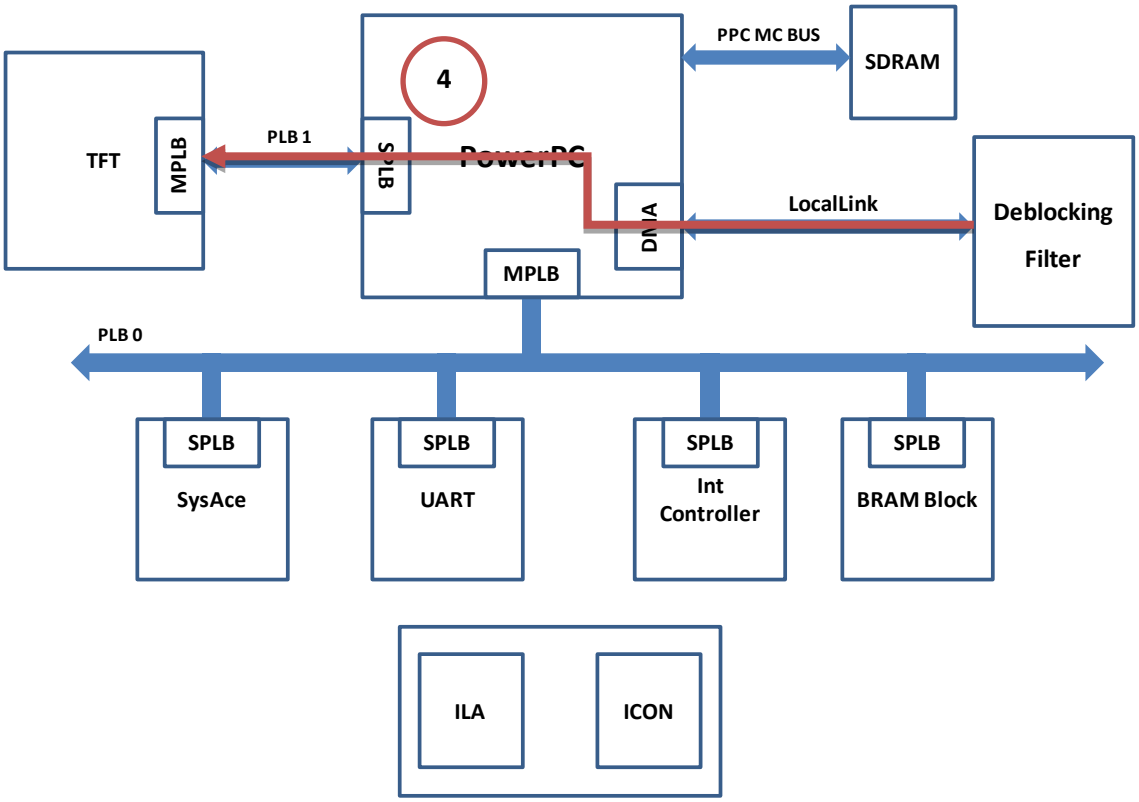

Figura 42. Cuarto paso del recorrido de datos, para representación en pantalla.

- XLlDma_BdRingCreate(). Sirve para crear un anillo de descriptores. Un descriptor es una estructura de datos que se utiliza para identificar transferencias a través del DMA. Cada vez que se desea enviar una ráfaga de datos a través del DMA, se usa uno de los descriptores libres del anillo como medio de comunicación con el DMA. En el descriptor se indican datos como dirección de memoria donde están los datos, longitud de la ráfaga, etc.
- XIntc_Connect(). Es la encargada de vincular rutinas de interrupción a los vectores de interrupción correspondientes. La ejecución de una aplicación que realiza transferencias usando un DMA no es, en general, lineal. Debido a que el DMA realizará dichas transferencias en paralelo a la ejecución de otras tareas de la CPU, es necesario disponer de interrupciones cuando una transferencia haya finalizado.
- XIntc_Enable(). Habilita un vector de interrupción en particular.
- RxHandler() y TxHandler(). Son las rutinas de interrupción de las interrupciones de trama recibida y trama enviada con éxito. En ellas se liberarán los descriptores asociados a dichas tramas y se prepararán los siguientes. En el caso de la recepción, en dicha rutina de interrupción se almacenarán los datos en la CompactFlash y se realizará una copia en memoria de los datos a la zona del buffer de visualización.
- XLlDma_BdRingIntEnable(). Indica al DMA que genere interrupciones cuando termine de enviar o recibir una trama.
- sysace_fopen(), sysace_fclose(), sysace_fread(), sysace_fwrite(). Son las funciones utilizadas para leer y escribir en la CompactFlash.
- convert_to_444(). Como ya se ha comentado, el DF trabaja con una representación 4:2:0, es decir, que la croma mide la mitad en alto y en ancho. Esta función es la encargada de pasar a una representación 4:4:4 donde las tres componentes coinciden en dimensiones.
- convert_to_RGB(). Pasa de una representación YUV a una representación RGB para su almacenamiento en el buffer de representación.

Una descripción más detallada así como una explicación más extensa de cómo utilizar estas rutinas software se pueden encontrar en [41] y en [34].

4.3 Depuración física

La depuración de un sistema *software* es un proceso muy documentado y sin dificultades tecnológicas, pues consiste en ejecutar paso a paso las instrucciones del código desarrollado, para lo cual los procesadores del mercado dan soporte.

Sin embargo, la depuración de un sistema digital durante su funcionamiento en tiempo real, requiere de tecnología adicional de captura de datos en los puntos de medida. Los dispositivos diseñados para este fin se conocen como analizadores lógicos, como son el caso de los bloques analizadores integrados (Integrated Logic Analyzer – ILA) instanciados en la plataforma.

El proceso de medida a través de un analizador lógico, consiste en la definición de unas condiciones de disparo, mediante sentencias lógicas de las señales de disparo y valores asignados por el diseñador. Cuando se cumpla la condición de disparo, el analizador comenzará a almacenar los valores, de forma síncrona al reloj con el que se alimenta (en analizadores lógicos de alta gama es común usar relojes de frecuencia más alta que la de las señales a capturar, con el fin de sobremuestrar la señal y capturar los cambios de estado de la misma, durante el tiempo de estabilización de la señal).

ChipScope permite dos modos de captura, el simple, y el de ventana. En el modo simple, cuando se cumple la condición, captura el número máximo de muestras que le permite el tamaño de su memoria interna (para este proyecto se configuró con un tamaño de 1024 muestras). En el modo ventana, se puede configurar un número de ventanas de captura, pero con un número de muestras inferior, dependiendo del número de ventanas seleccionadas. El tamaño de las ventanas de captura siempre será un múltiplo de dos (tanto para tres ventanas, como para cuatro ventanas, la longitud de la trama capturada será de 256 muestras).

Otro parámetro importante en la captura, es el desplazamiento de la muestra de disparo. Este valor, indica que posición ocupará la muestra en la que se produzco el disparo, respecto a la primera muestra capturada. Por ejemplo, puede darse el caso que, cuando se produce un evento en una señal, se quiera capturar los datos a partir de la muestra 100 después de dicho evento. En ese caso, el valor del desplazamiento (*Position*, en ChipScope) será de -100. Pueden además, indicarse desplazamientos positivos, si se quieren capturar las 100 muestras anteriores a que se produzca el evento (para ello, el analizador lógico se encuentra en un estado de escritura continua de los datos, siendo el evento el que indica que no los sobrescriba con los que se produzcan a partir de él).

En la Figura 37 se observa la configuración de las condiciones de disparo. En la parte superior, se indican los valores lógicos asociados a cada señal (1, 0 ó X). En la inferior, se enumeran las condiciones de disparo. En este caso, a la señal DMATXIRQ (M14), se le asocia el valor lógico 1 (ya que es una señal activa a nivel alto), y en la condición de disparo se indica M14 (se pueden indicar condiciones compuestas como M14 && M13).

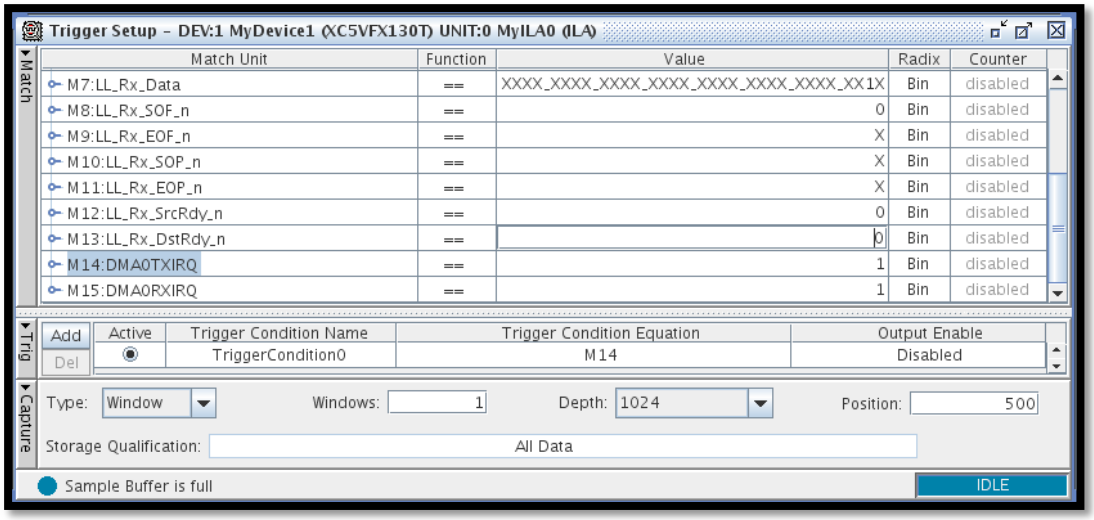

Figura 43. Condiciones de disparo en ChipScope.

4.4 Esquema de validación

Para la visualización en tiempo real del proceso de filtrado se ha dispuesto de una pantalla conectada a la plataforma FPGA donde se representa la imagen filtrada. Este esquema queda representado en la Figura 38. En la figura se puede observar una muestra decodificada de la secuencia Bus y a la derecha el entorno de desarrollo de Xilinx utilizado. Para comprobar que los datos representados en la pantalla son los correctos, en un segundo modo de operación estos datos son almacenados en un fichero de texto de la memoria flash de la tarjeta CompactFlash, comparado luego con el generado por el *software* de referencia.

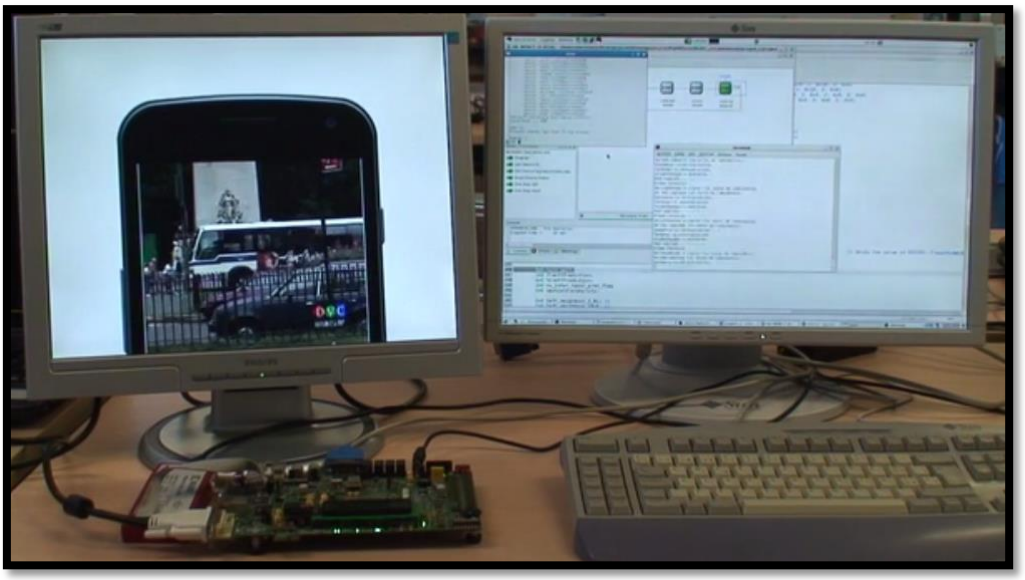

Figura 44. Esquema de validación.

5 Conclusiones

En este capítulo se ha presentado la plataforma utilizada para la validación del bloque diseñado prototipado sobre la FPGA. Se ha descrito su composición así como el proceso por el cual los datos son leídos, enviados, procesados, representados y almacenados.

Además se ha definido el proceso de depuración física de un dispositivo hardware mediante el uso de analizadores lógicos, atendiendo a la metodología de captura de datos y su representación.

Con ello se ha validado el sistema funcionando sobre la plataforma FPGA obteniendo unos valores de píxeles de salida idénticos a los del software de referencia Open SVC Decoder. Además se ha podido comprobar el funcionamiento en tiempo real del sistema suministrándole una gran cantidad de datos previa lectura en memoria DDR. Además, este trabajo ha permitido generar una plataforma de validación que puede utilizarse con otros bloques IP de distinta índole con muy pocas modificaciones, facilitando así futuros trabajos de validación.

Capítulo 6: Conclusiones y líneas futuras

1 Conclusiones

En este Trabajo Fin de Máster se realiza el prototipado sobre una plataforma FPGA de un *Deblocking Filter* para H.264/SVC partiendo de una descripción del mismo en alto nivel SystemC.

Con el fin de alcanzar este objetivo se define un flujo de diseño basado en la síntesis de alto nivel, frente a los comunes flujos basados en la realización de un diseño a nivel RTL directamente escrito por el diseñador.

Como punto de comienzo se realizada un estudio de la arquitectura de un decodificador de vídeo H.264, prestando especial interés al *DF*, su funcionalidad y sus distintas etapas. Además, se presentan las novedades introducidas por la sección SVC del estándar H.264.

Conociendo el funcionamiento del bloque, se estudia la descripción del mismo usado como partida, descrita en SystemC, y comprendiendo la división jerárquica realizada y su equivalencia en cuanto a etapas de la ejecución del mismo.

Esta descripción ha sido verificada en alto nivel mediante la comparación de la salida generada con el código de referencia utilizado en su elaboración, el Open SVC Decoder, para luego comenzar con el flujo de síntesis.

El flujo consiste en una primera etapa de síntesis de alto nivel que da lugar a una descripción RTL del modelo, verificada utilizando técnicas de cosimulación que comparan los resultados obtenidos desde el modelo funcional con el modelo RTL y con el fin de verificar que el proceso de síntesis se haya realizado correctamente. Comprobado esto, la síntesis lógica nos permite realizar medidas de ocupación, frecuencia y potencia con el fin de conocer cómo se distribuye dichos costes entre los bloques que lo componen.

Se realiza también el prototipado del mismo sobre una FPGA Virtex 5. Para ello se ha diseñado una plataforma de validación que permite enviar y recibir los datos del modelo en cuestión, almacenar estos datos en un fichero de texto con el fin de compararlo con el generado por el *software* de referencia, así como de representar la salida mediante una pantalla y poder así comprobar el filtrado de las imágenes de forma visual.

En este trabajo se pretende, por un lado demostrar la viabilidad de los nuevos flujos de diseños basados en síntesis de alto nivel, sin necesidad de escribir toda la microarquitectura de un diseño en lenguajes de descripción *hardware* de nivel RTL. Por otro lado, se presenta cómo realizar la validación de un bloque hardware diseñado mediante el uso de una plataforma en una FPGA, teniendo en cuenta para ello que bloques son necesarios, qué métodos de interconexión utilizar y qué rutinas *software* son necesarias para la correcta validación del mismo.

Este trabajo ha sido incluido en las siguientes publicaciones enviadas a congresos:

- Implementation of Scalable Video Coding Deblocking Filter from High-Level SystemC Description. SPIE Microtechnologies 2013, Grenoble, Francia. [20]
- Scalable Video Coding Deblocking Filter FPGA and ASIC implementation using High-Level Synthesis Methodology. EUROMICRO DSD 2013, Santander, España. [42]

2 Líneas de trabajo futuras

De la realización de este Trabajo Fin de Máster pueden nacen diferentes ideas con el fin de continuar estudiando tomando como punto de partida la finalización de este.

Por un lado, dadas las nuevas arquitecturas de las familias Virtex 7 de Xilinx, en los modelos Zynq, es de interés la validación de diseños *hardware* haciendo uso de estas, utilizando para ello los nuevos microprocesadores ARM y los buses de comunicación AMBA/AXI con los que vienen integrados.

Otra fuente de desarrollo es la optimización del diseño utilizado como entrada en este flujo. Habiéndose obtenido unos resultados de síntesis y prototipado, es de especial interés utilizar esta información para optimizar aquellos bloques de mayor ocupación para reducir su coste de recursos u optimizar los bloques de mayor latencia para que realicen la misma funcionalidad en un menor número de ciclos aplicando paralelización o modificando parte de la arquitectura. Por ejemplo, existe en el diseño actual un gran número de ciclos empleados en el movimiento de datos entre los diferentes bloques de memoria del diseño. Esto se debe en parte a la arquitectura basada en el uso de memorias como paso de datos entre bloques. Este punto debería ser el de principal optimización. Otro punto podría ser el estudio de la posibilidad de realizar un pipeline entre las distintas etapas del filtrado, teniendo en cuenta para ello la dependencia de datos entre macrobloques debido al uso de datos de vecinos.

Por último, dada la naturaleza que se trata en este TFM, se hace necesaria la comunicación del bloque *DF* con el resto del decodificador de vídeo. En el proyecto en el que está definido este TFM, el resto de etapas de la decodificación es realizada por un SoC de Texas Instruments basado en un microprocesador ARM y un DSP, en particular un OMAP 3530. Es por ello que se hace necesario el desarrollo de una interfaz de comunicación que permita la transmisión de datos entre el OMAP 3530 y la FPGA que implementará el *DF*. Este trabajo está siendo objeto de desarrollo en un proyecto fin de carrera, implementando una interfaz esclava del protocolo General Pupose Memory Controller (GPMC) de Texas Instruments.

Bibliografía

[1] Garcia, P.; Salgado, F.; Cardoso, P.; Cabral, J.; Ekpanyapong, M.; Tavares, A., "A FPGA based C runtime hardware accelerator," *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on* , vol., no., pp.805,809, 26-29 July 2011.

[2] Levi, S.; Agrawala, A. K., *Real-Time System Design* . Ed. McGraw-Hill Pub. Co., 1990.

[3] Mauer, G.F., "A fuzzy logic controller for an ABS braking system", *IEEE Transactions on Fuzzy Systems*, Noviembre 2009.

[4] Lawler, R., "H.264 is still winning the codec war," GIGAOM. [En línea]. Disponible: http://gigaom.com/2011/07/07/h-264-winning-the-codec-war/.

[5] International Organization of Standardization - ISO, "ISO/IEC 11172-2:1993 Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 2: Video ," 1993.

[6] International Telecommunication Union, ITU-T, "H.262. Information Technology - Generic coding of moving pictures and associated audio information: Video", 1995.

[7] International Telecommunication Union, ITU-T, "H.263. Video coding for low bit rate communication", 1996.

[8] International Telecommunication Union, ITU-T, "H.264. Series H: Audiovisual and multimedia systems. Infrastructure of audiovisual services - Coding of moving video. Advanced video coding for generic audiovisual services.", 2003.

[9] International Telecommunication Union, ITU-T, "H.265. Series H: Audiovisual and multimedia systems. Infrastructure of audiovisual services - Coding of moving video. High efficiency video coding.", 2013.

[10] Song, Y., "Introduction to H.264/AVC," University of Arizona. [En línea]. Disponible: http://www2.engr.arizona.edu/~yangsong/h264.htm.

[11] Blestel, M.; Raulet, M., "Open SVC Decoder: a Flexible SVC Library," *Proceedings of the international conference on Multimedia*, Italy, 2010.

[12] Nedjah, N.; Mourelle, L. M., "Co-Design for System Acceleration : A Quantitative Approach." ed. London, UK, Springer, 2007. Disponible: http://dx.doi.org/10.1007/978-1-4020-5546-1.

[13] Grötker, T., *System Design with SystemC*, Boston: Kluwer Academic Publishers, 2002.

[14] Coussy, P.; Morawiec, A., *High-Level Synthesis: From Algorithm to Digital Circuit,* ed. Springer, 2008.

[15] Neris, R., *Implementación y caracterización de un IP decodificador de vídeo H.264/AVC en tecnologías CMOS DSM*, Proyecto Fin de Carrera, Universidad de Las Palmas de Gran Canaria, 2012.

[16] Parlak, M.; Adibelli, Y.; Hamzaoglu, I., "A novel computational complexity and power reduction technique for H.264 intra prediction". *Consumer Electronics, IEEE Transactions on 54(4),* pp. 2006-2014. 2008.

[17] Wang, S. B.; Zhang, X. L.; Yao, Y.; Wang, Z., "H.264 intra prediction architecture optimization". *Multimedia and Expo, 2007 IEEE International Conference on*, 2007.

[18] Youn-Long, S. L.; Chao-Yang, K.; Hung-Chih, K.; Jian-Wen, K., *VLSI Design for Video Coding: H.264/AVC Encoding from Standard Specification to Chip*, Springer-Verlag New York Inc. 107-124 (2010).

[19] D. M. Heiko Schwarz, "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard," .

[20] Carballo, P. P.; Espino, O.; Neris, R.; Hernández-Fernández, P.; Szydzik, T. M.; Nunez, A., "Implementation of scalable video coding deblocking filter from high-level SystemC description", *Proc.* SPIE8764, VLSI Circuits and Systems VI, 876408 (May 28, 2013).

[21] Real Academia Española. *Diccionario De La Lengua Española* (22º ed.) 2001.

[22] Guenassia, F., *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. ed. Springer-Verlag New York, Inc. Secaucus, NJ, USA ISBN:0387262326 2005.

[23] Rigo, S.; Azevedo, R.; Santos, L., *Electronic System Level Design: An Open-Source Approach.* ed. Springer, 2011.

[24] IEEE Computer Society, *IEEE Standard for Standard SystemC® Language Reference Manual,* 2012.

[25] Black, D. C.; Donovan, J.; SpringerLink. *SystemC: From the Ground Up,* 2004. Disponible: http://www.springerlink.com/openurl.asp?genre=book&isbn=978-0-387-29240-3.

[26] Ganguly, M., *SystemC Overview*, Synopsys, Inc., 2001.

[27] *Cadence C-to-Silicon Compiler: User Guide*, Cadence Design Systems, Inc., 2011.

[28] *Synplify Premier User Guide*, Synopsys, Inc., 2011.

[29] *Embedded System Tools Reference Manual,* Xilinx, Inc., 2011.

[30] *ChipScope Pro Software and Cores: User Guide*, Xilinx, Inc., 2011.

[31] *Virtex-5 Family Overview*, Xilinx, Inc., 2009.

[32] *Virtex-5 FPGA User Guide,* Xilinx, Inc., 2012.

[33] *ML505/ML506/ML507 Evaluation Platform User Guide*, Xilinx, Inc., 2011.

[34] Espino, O., *Acelerador Hardware para el Procesado de Eventos en Tiempo Real*, Proyecto Fin de Carrera, Universidad de Las Palmas de Gran Canaria, 2012.

[35] *XST User Guide*, Xilinx, Inc., 2009.

[36] Neris, R., *Síntesis de alto nivel de un decodificador H.264/AVC sobre plataforma FPGA*, Trabajo Fin de Máster, Instituto Universitario de Microelectrónica Aplicada, Universidad de Las Palmas de Gran Canaria, 2012.

[37] Nadeem, M.; Wong, S.; Kuzmanov, G.; Shabbir, A.; Nadeem, M.F.; Anjam, F., "Low-power, high-throughput deblocking filter for H.264/AVC," *System on Chip (SoC), 2010 International Symposium on* , vol., no., pp.93,98, 29-30 Sept. 2010.

[38] Parlak, M.; Hamzaoglu, I., "A Low Power Implementation of H.264 Adaptive Deblocking Filter Algorithm," *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, vol., no., pp.127,133, 5-8 Aug. 2007.

[39] Kamel, D.; Standaert, O.-X.; Flandre, D., "Scaling trends of the AES S-box low power consumption in 130 and 65 nm CMOS technology nodes," Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on, vol., no., pp.1385,1388, 24-27 May 2009.

[40] *LocalLink Interface Specification*, Xilinx, Inc., 2005.

[41] Lucero, J., *Reference System: Designing an EDK Custom Peripheral with a LocalLink Interface*, Xilinx, Inc., 2008.

[42] Carballo, P. P.; Espino, O.; Neris, R.; Hernández-Fernández, P.; Szydzik, T. M.; Nunez, A., "*Scalable Video Coding Deblocking Filter FPGA and ASIC implementation using High-Level Synthesis Methodology*," *Proc.* EUROMICRO DSD 2013, 2013.