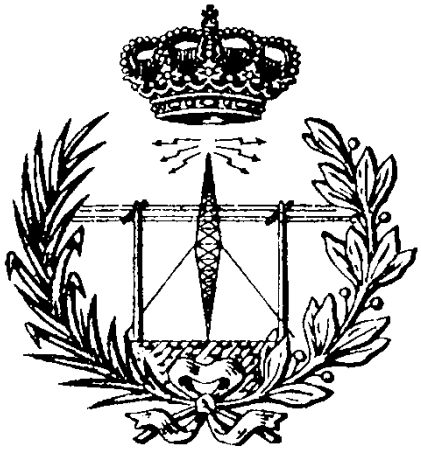
UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



PROYECTO FIN DE CARRERA

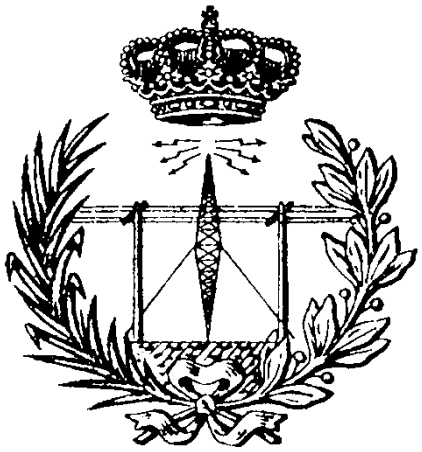
METODOLOGÍA DE SÍNTESIS ESL BASADA EN TLM. APLICACIÓN AL DISEÑO DE UN DECODIFICADOR DE VÍDEO

Titulación: Ingeniero de Telecomunicación
Autor: Paloma Monzón Rodríguez
Tutores: D. Pedro Pérez Carballo
D. Pedro Hernández Fernández

Fecha: Marzo 2015

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



PROYECTO FIN DE CARRERA

METODOLOGÍA DE SÍNTESIS ESL BASADA EN TLM. APLICACIÓN AL DISEÑO DE UN DECODIFICADOR DE VÍDEO

HOJA DE FIRMAS

Alumno/a

Fdo.: Paloma Monzón Rodríguez

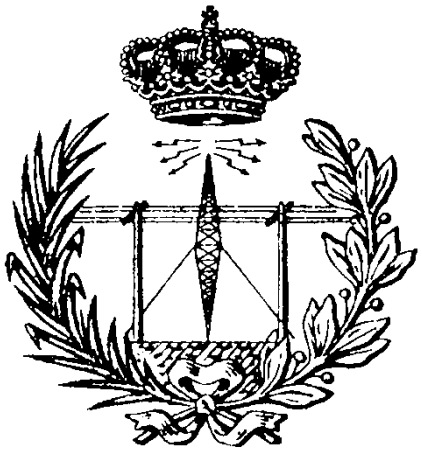
Tutor/a **Tutor/a**

Fdo.: Pedro Pérez Carballo Fdo.: Pedro Hernández Fernández

Titulación: Ingeniero de Telecomunicación
Fecha: Marzo 2015

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



PROYECTO FIN DE CARRERA

METODOLOGÍA DE SÍNTESIS ESL BASADA EN TLM. APLICACIÓN AL DISEÑO DE UN DECODIFICADOR DE VÍDEO

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario/a**

Fdo.: Fdo.:

Titulación: Ingeniero de Telecomunicación

Fecha: .....

Agradecimientos

La realización y presentación de este proyecto supone el fin de una etapa que comenzó con la entrada en la Universidad y en la que, desde entonces, ha habido de todo: palabras de ánimo, consejos, risas, lágrimas, oportunidades, aprendizaje, amistad... Un largo sin fin que me ha ayudado a crecer y estar hoy aquí.

Una etapa por la que han pasado muchas personas. Simplemente, a todos, gracias ☺

Índice general

Índice de figuras

Índice de tablas

Acrónimos

ACK	*Acknowledgement*
AMD	*Advanced Micro Devices*
ASIC	*Application-Specific Integrated Circuit*
ASIP	*Application-Specific Instruction-Set Processor*
AVC	*Advanced Video Coding*
BCA	*Bus Cycle Accurate*
BRAM	*Block RAM*
CABAC	*Context-Adaptive Binary Arithmetic Coding*
CAD	*Computer-Aided Design*
CAVLC	*Contex-Adaptive Variable Length Coding*
CAVLD	*Contex-Adaptive Variable Length Decoder*
CD	*Compact Disc*
CDFG	*Control and Data Flow Graph*
CLB	*Configurable Logic Block*
CMOS	*Complementary Metal-Oxide Semiconductor*
CtoS	*C-to-Silicon Compiler*
DCT	*Discrete Cosine Transform*
DPB	*Decoded Picture Buffer*
DSP	*Digital Signal Processor*
EDIF	*Electronic Design Interchange Format*
E/S	Entrada/Salida
ESL	*Electronic System Level*
FEC	*Forward Error Correction*
FIFO	*First-In First-Out buffer*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*

GB	*Gigabyte*
GIMP	*GNU Image Manipulation Program*
GUI	*Graphical User Interface*
HDL	*Hardware Description Language*
HLS	*High-Level Synthesis*
HP	*Hewlett-Packard*
HW	*Hardware*
I/O	*In/Out*
IC	*Integrated Circuit*
IEEE	*Institute of Electrical and Electronics Engineers*
IGIC	Impuesto General Indirecto Canario
IP	*Intellectual Property*
ISE	*Integrated Software Environment*
IUMA	Instituto Universitario de Microelectrónica Aplicada
LUT	*Look-Up Table*
MB	Macrobloque
MPEG	*Moving Picture Experts Group*
NAL	*Network Abstraction Layer*
OSCI	*Open SystemC Initiative*
PA	*Pin Accurate*
PB	*Picture Buffer*
PSNR	*Peak Signal to Noise Ratio*
QoR	*Quality of Results*
RAM	*Random-Access Memory*
REQ	*Request*
RTL	*Register Transfer Level*
S/N	*Signal to Noise Ratio*
SICAD	Sistemas Industriales y CAD
SLD	*System Level Design*
SoC	*System on Chip*
SW	*Software*
TCL	*Tool Command Language*
TIC	Tecnologías de la Información y las Comunicaciones
TLM	*Transaction-Level Modeling*

TPB	*Temporal Picture Buffer*
UCF	*User Constraint File*
UMC	*United Microelectronics Corporation*
UUT	*Unit Under Test*
VCEG	*Video Coding Experts Group*
VCL	*Video Coding Layer*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
VLC	*Variable Length Coding*
VLSI	*Very Large Scale Integration*
XPE	*Xilinx Power Estimator*
XPA	*XPower Analyzer*

Resumen

RTL es el nivel de abstracción de uso común para el diseño de circuitos integrados, cuya metodología de síntesis y verificación mediante el uso de lenguajes de descripción *hardware* permite reducir los tiempos e incrementar la calidad de los diseños obtenidos. La complejidad en los sistemas electrónicos actuales va en aumento, se requieren sistemas en chip (SoCs) funcionales, económicos, con un bajo consumo de potencia y con un alto nivel de integración que incorporen toda la funcionalidad. Ello dificulta la verificación funcional o la toma de decisiones en fases tempranas del flujo de diseño. Ante este panorama es necesario incrementar el nivel de abstracción para la especificación y verificación del diseño y desarrollar nuevas metodologías.

El modelado a nivel de transacciones (TLM) implica un cambio del nivel de abstracción, donde se separan los aspectos de computación y los de comunicación de un sistema, facilitando el análisis de la arquitectura, reduciendo los esfuerzos en el modelado y aumentando la productividad del diseñador.

El objetivo principal de este proyecto es el desarrollo de una metodología de síntesis basada en el modelado a nivel de transacciones, aplicándola sobre el diseño de uno de los bloques del decodificador de vídeo H.264/AVC perteneciente al Instituto Universitario de Microelectrónica Aplicada (IUMA). Para ello se sigue un flujo de diseño centrado en la síntesis de alto nivel con el fin de obtener modelos sintetizables para una futura implementación *hardware* tanto en ASIC como en FPGA, siendo el primer paso modelar el bloque con TLM.

En definitiva, se pretende conocer en detalle TLM y más concretamente su aplicación en la síntesis de alto nivel, de manera que permita evaluar las cualidades de esta nueva metodología.

Este proyecto se desarrolla en el marco de las líneas de trabajo de la División de Sistemas Industriales y CAD (SICAD) del IUMA.

Abstract

RTL is the commonly used level of abstraction for integrated circuits design, its synthesis and verification through hardware description languages allow reducing design time and increasing the quality of design. The complexity of current electronic systems is increasing, functional, economic, low power consumption and high level of integration systems on chip (SoCs) that incorporate full functionality are required. This complicates the functional verification or decision making in the early stages of the design flow. In this scenario it is necessary to raise the level of abstraction for design specification and verification and develop new design methodologies.

Transaction level modeling (TLM) implies a change in the level of abstraction, where computing and communication aspects of the system are separated, facilitating the analysis of architecture, reducing modeling efforts and increasing productivity of the designer.

The main aim of this project is the development of a synthesis methodology based on transaction level modeling, applied on the design of one of the blocks of H.264/AVC video decoder belonging to the Institute for Applied Microelectronics (IUMA). To achieve this, a design flow centered on high-level synthesis is used in order to obtain synthesizable models for future hardware implementation in both ASIC and FPGA, where the first step is modeling the block with TLM.

In other words, TLM is studied in detail and more specifically its application in high-level synthesis, so as to assess the qualities of this new methodology.

This project is developed in the context of the research lines of the Division of Industrial and Systems CAD (SICAD) of IUMA.

Capítulo 1

Introducción

1.1. Antecedentes

Cada vez más aparecen noticias sobre avances tecnológicos, nuevos dispositivos, nuevos entornos y mayor uso de las Tecnologías de la Información y las Comunicaciones (TIC) en la vida cotidiana. Se trata de la evolución de la Sociedad de la Información, donde el desarrollo y uso de la tecnología van en aumento, siendo diversos sus ámbitos de aplicación [1].

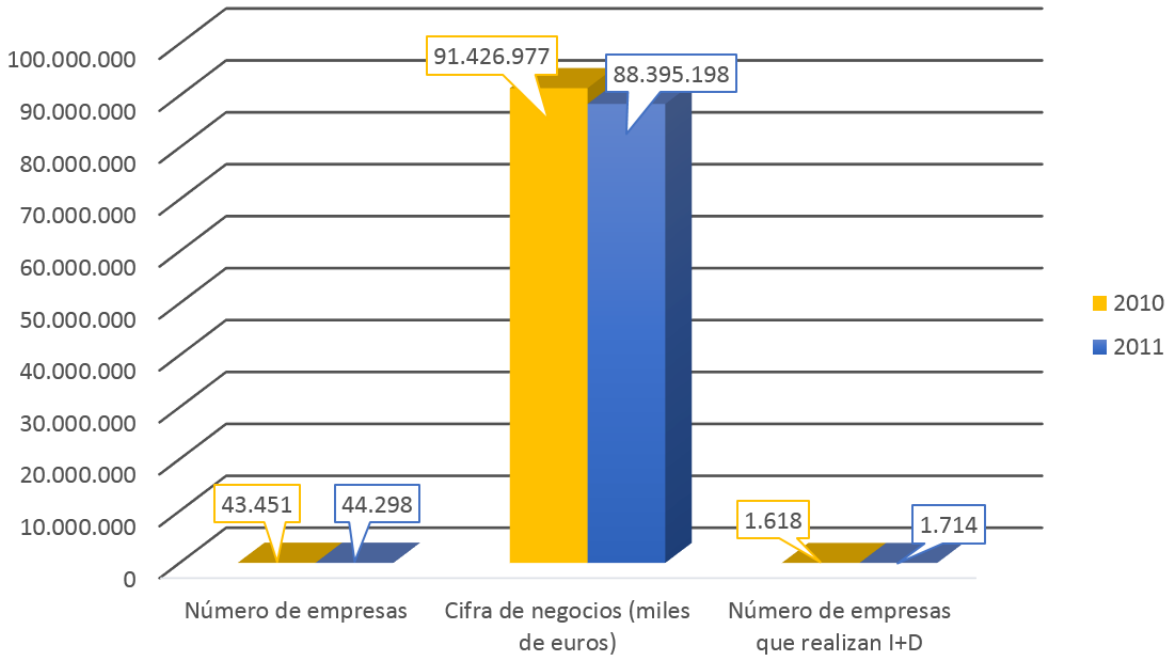


Figura 1.1: Evolución del sector TIC (adaptado de [1])

Los problemas que se abordan son cada vez más arduos, precisan de recursos de cómputo más elevados, con mayor velocidad de procesamiento y grado de paralelismo (redes y comunicaciones, tecnologías médicas, etc.) [2]. Esto conlleva la utilización de sistemas electrónicos más complejos, ya sea desde el punto de vista de la presencia de unidades de procesamiento funcionalmente más complejas y adaptadas a la funcionalidad requerida (ASIP) o la necesidad de incluir en el sistema unidades de procesamiento replicadas para cubrir el procesamiento masivo paralelo.

En ambos casos, las metodologías de diseño se quedan obsoletas y es necesario un cambio de las mismas, añadiendo a las tradicionales basadas en RTL nuevas capacidades de abstracción y procesos de diseño que permitan abordar este incremento de complejidad. En este marco, la literatura científica acepta la necesidad de introducir un nivel de abstracción superior a RTL que facilite al diseñador la creación de modelos del sistema a nivel de transacciones (TLM) [3].

TLM basa el diseño en la **separación de los aspectos de computación y los de comunicación** a incluir en el sistema. Facilitando por una parte, la migración de una aplicación compleja (computación) hacia el sistema *hardware*, y por otra, permitiendo un estudio de la arquitectura de comunicación. Por tanto, ayuda al diseñador con el análisis de dicha arquitectura, reduciendo el esfuerzo de modelado, y aumentando y mejorando la productividad del mismo [4].

1.1.1. Modelado a nivel de transacciones

El diseño a nivel de sistemas (*System Level Design* - SLD) implica un cambio de nivel de abstracción, pasando a crear modelos cuya síntesis se apoya en lenguajes de descripción de sistemas como puede ser SystemC [3], [5]. Este permite describir tanto los componentes *hardware* como *software* de un sistema, poniendo énfasis en su comportamiento y sus comunicaciones [6].

Con RTL es posible reducir los tiempos de diseño aplicando técnicas de reutilización, encapsulando los diseños como IPs. La complejidad del proceso aumenta con la del sistema y se dificulta la verificación funcional y la toma de decisiones sobre la arquitectura, que no pueden ser confirmadas hasta la verificación. Este momento del proceso es crítico para diseños complejos, donde el análisis de aspectos de la arquitectura es clave para la optimización del sistema final [8], [9].

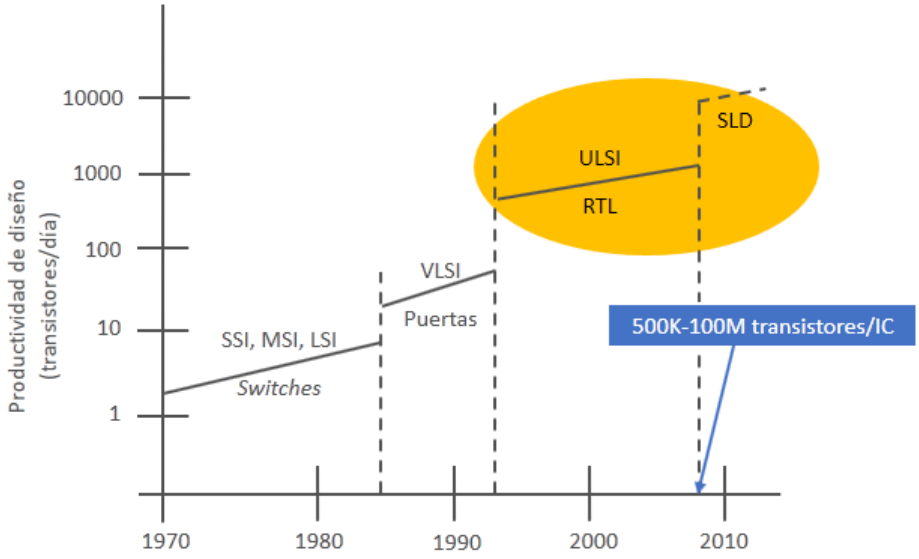


Figura 1.2: Productividad de diseño (adaptado de [7])

El modelado a nivel de transacciones (*Transaction-Level Modeling*) se basa en abstraer las comunicaciones entre las distintas partes de un sistema para permitir trabajar, por un lado con la funcionalidad de cada módulo y por otro, con las comunicaciones. Esta abstracción se representa mediante **transacciones** entre módulos. La comunicación se implementa dentro de **canales**, ocultando los protocolos y exponiendo sus interfaces.

Este nivel de abstracción se ha convertido en un punto de partida en el diseño de sistemas. Cuanto más alto sea el nivel en el que los módulos son diseñados, más rápido será el proceso de simulación y más fácil será la conexión entre ambas partes: funcionalidad y comunicación [10].

En la Figura 1.3 se muestra un flujo de diseño clásico basado en TLM. A partir de los detalles del sistema, generalmente en forma de unas especificaciones ejecutables, se procede a separar el procesamiento de las comunicaciones, para después realizar la descripción TLM de las interfaces, que junto con la funcionalidad, se sintetizará. A la salida de la síntesis se obtendrá la representación RTL, normalmente en términos de unidades de procesamiento y unidades de control con su correspondiente flujo de datos y señales de control. El siguiente paso será realizar la verificación, generalmente a través de la simulación, que permitirá comprobar el correcto funcionamiento del sistema.

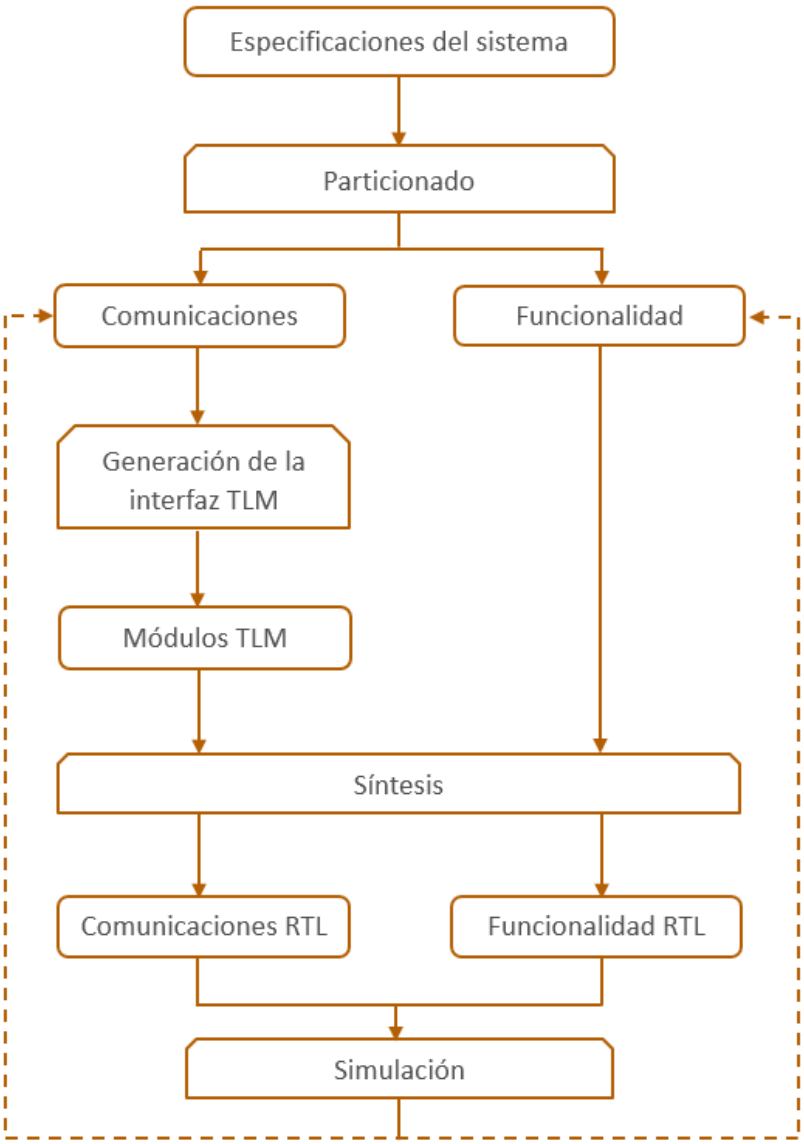


Figura 1.3: Diagrama de flujo de diseño TLM (adaptado de [10])

Mediante la creación de IPs basados en TLM se pretende evitar detalles innecesarios durante las primeras fases del flujo de diseño, con lo que se facilita su creación, así como su reutilización. De este modo, se emplea menos tiempo y esfuerzo en la verificación funcional y se produce un menor número de errores. Además, las decisiones tomadas sobre la arquitectura pueden ser confirmadas antes de la verificación. En definitiva, el resultado de todas estas ventajas será una productividad mucho más alta y unas mejoras significativas en la velocidad de simulación, lo cual permite un mejor análisis del espacio de diseño a nivel arquitectural [9], [11], [12].

Existen diferentes entornos de diseño que soportan metodologías basadas en TLM. A continuación

se referencian algunos de ellos:

- Cynthesizer de Cadence Design Systems [1] es un entorno de síntesis basado en TLM que facilita la labor del diseñador al combinar el modelado del diseño a nivel de transacciones y las interfaces a nivel de pines (PA) [13].
- Cadence C-to-Silicon Compiler (CtoS) [2] proporciona la síntesis del diseño desde una combinación de metodologías de representación, ya sea puramente C++, SystemC o TLM, transformando el diseño hasta una representación RTL [14].
- Calypto Catapult, al igual que CtoS, soporta la síntesis de alto nivel desde una descripción SystemC/C++ [15].
- Vivado Design Suite de Xilinx permite realizar la síntesis de alto nivel partiendo desde SystemC, TLM o C++ [16].

Este proyecto utiliza **Cadence CtoS** como entorno de síntesis TLM, siendo el flujo de referencia a desarrollar el que se presenta en la Figura 1.4.

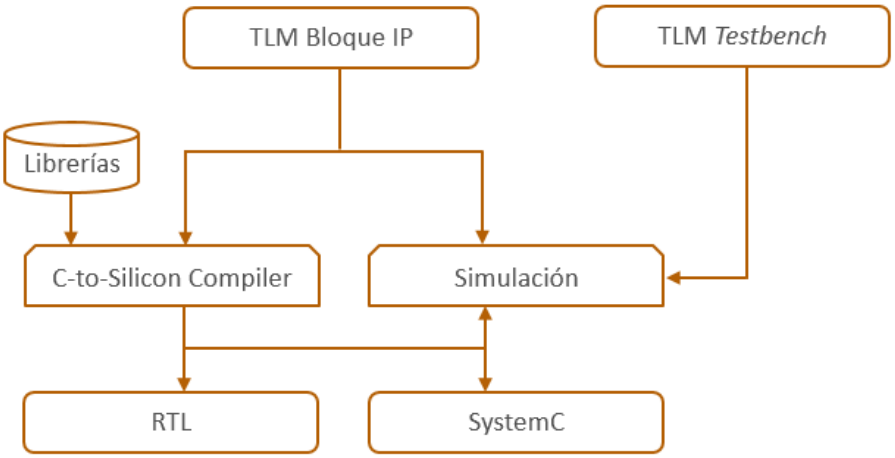


Figura 1.4: Diagrama de flujo de diseño TLM de Cadence CtoS (adaptado de [14])

Pese a que la metodología de diseño TLM puede desarrollarse con distintos lenguajes, **SystemC** se ajusta perfectamente a su estilo de representación ya que soporta de manera natural esta metodología, permitiendo un nivel adecuado de abstracción y aportando la sintaxis y semántica necesaria

[1] Cadence Design Systems, Inc. ha adquirido Forte Design Systems (febrero 2014) [42].

[2] Cadence Design Systems, Inc. está integrando Cynthesizer y CtoS en una nueva herramienta llamada Stratus High-Level Synthesis, que incorpora tecnología de ambas herramientas (febrero 2015).

que permite la separación entre la funcionalidad y la comunicación.

A partir del modelado a nivel de transacciones se plantea la realización de la síntesis de un decodificador de vídeo H.264/AVC. De esta manera se validará la metodología desarrollada y se plantearán las ventajas e inconvenientes de la misma.

1.1.2. Estándar H.264/AVC

La historia sobre la codificación de vídeo se remonta a mediados de la década de los 80, con la creación del estándar H.120, precursor del H.261. Sobre este están basados todos los estándares siguientes: MPEG-1 Parte 2, H.262/MPEG-2 Parte 2, H.263, MPEG-4 Parte 2 y H.264/AVC [17].

El estándar H.264 *Advanced Video Coding*, o MPEG-4 parte 10, define un *códec* de vídeo de alta compresión [18]. Fue publicado en 2003, y es capaz de proporcionar una buena calidad de imagen con tasas binarias inferiores a los estándares previos.

Está compuesto por:

- Un **codificador**, el cual se encarga, a partir de una secuencia de vídeo, de predecir, transformar y codificar, y así obtener una salida apta para ser enviada por un canal o ser almacenada en un fichero.
- Un **decodificador**, que hace la operación inversa: decodifica, realiza la transformada que corresponda y reconstruye la secuencia original de vídeo.

Aunque este *códec* implica ahorros sustanciales en las tasas binarias y mejoras en la calidad de imagen, la complejidad de diseño supera significativamente a la de los estándares anteriores [19], [20]. La Figura 1.5 muestra como para un mismo *bitrate* se obtiene una mejor calidad y resolución con H.264 que con uno de sus predecesores, MPEG-2.

En este proyecto se hará uso de un decodificador de vídeo H.264/AVC para demostrar el uso de la metodología explicada en el apartado anterior. Para ello se utilizará el bloque de predicción temporal (*inter_p*), al que se dotará de interfaces TLM con objeto de comprobar su viabilidad y comparar la calidad de los resultados obtenidos después de la síntesis.

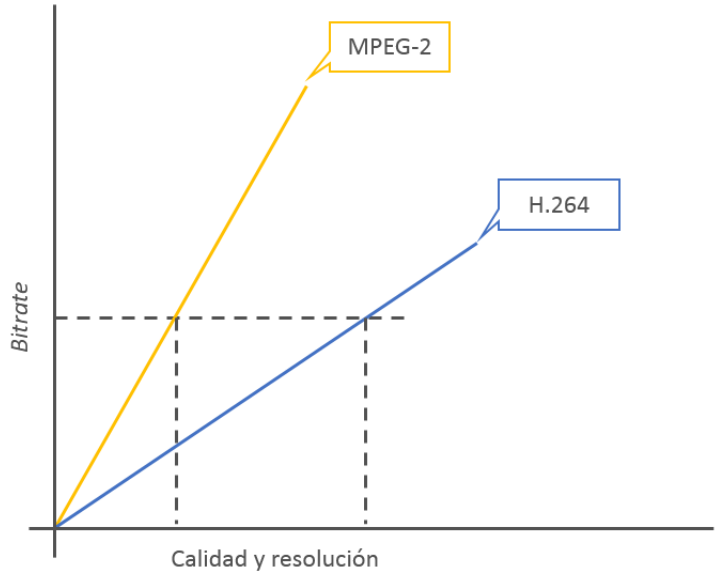


Figura 1.5: Calidad y resolución entre MPEG-2 y H.264 (adaptado de [21])

1.2. Objetivos

El objetivo principal de este proyecto es la **definición de una metodología de diseño de sistemas electrónicos integrados a partir del modelado a nivel de transacciones (TLM)**, con énfasis en la creación de modelos sintetizables. La metodología se validará aplicándola al diseño del bloque *inter_p* de un decodificador de vídeo.

Se partirá del código en SystemC del sistema, tratando los módulos del mismo como “cajas grises” (*grey-box model*), centrándose en las interfaces, y donde la funcionalidad está completamente definida.

En concreto se abordarán los siguientes puntos:

1. Definición de la metodología de modelado, verificación y síntesis TLM.
2. Aplicación a diseños simples.
3. Estudio del *software* de referencia del decodificador H.264/AVC.
4. Descripción incremental a nivel TLM.
5. Síntesis de alto nivel del modelo.
6. Validación de la metodología.

1.3. Peticionario

Actúa como peticionario de este Proyecto Fin de Carrera la División de Sistemas Industriales y CAD (SICAD) 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 este Proyecto Fin de Carrera es requisito indispensable para la obtención del título de Ingeniero de Telecomunicación por la Escuela de Ingeniería de Telecomunicación y Electrónica de la Universidad de Las Palmas de Gran Canaria.

1.4. Estructura del documento

El presente documento, a través de siete capítulos, describe el trabajo realizado. Se estructura de la siguiente manera:

- **Capítulo 1: Introducción.** Se muestra el marco en el que se recoge este Proyecto Fin de Carrera, para el cual se detallan los antecedentes, objetivos, peticionario y estructura del documento.
- **Capítulo 2: Dominio de la aplicación.** Se realiza una descripción del decodificador H.264/AVC, así como de la situación de partida.
- **Capítulo 3: Metodología de diseño.** Se introducen los elementos que forman parte del desarrollo del proyecto, como son las herramientas *software* utilizadas o el flujo de diseño seguido; y se expone el modelado a nivel de transacciones.
- **Capítulo 4: Modelado TLM con *wrapper*.** Se describe el modelado con *wrapper*, detallando las características del modelo.
- **Capítulo 5: Modelado TLM de las interfaces.** Se modela a nivel de transacciones el módulo en estudio.
- **Capítulo 6: Comparativa de los resultados.** Análisis de los resultados obtenidos mediante una comparativa de los mismos.
- **Capítulo 7: Conclusiones y líneas futuras.** Se presentan las conclusiones generales del proyecto y algunas líneas de trabajo que pueden derivarse de su realización.

Capítulo 2

Dominio de la aplicación

2.1. Introducción

El objetivo principal de este Proyecto Fin de Carrera es **estudiar la metodología de síntesis basada en TLM**, aplicándola sobre el diseño de un decodificador de vídeo H.264/AVC, por lo que en los siguientes apartados se presentarán las características principales de dicho decodificador, prestando especial atención a los bloques que lo componen.

Tras tener una visión global del estándar se realizará una descripción detallada de la aplicación de referencia que se va a utilizar.

2.2. Características generales

Ya sea la transmisión de vídeo en tiempo real, *streaming*, o su almacenamiento multimedia, cualquiera que sea la forma, genera la necesidad de ofrecer unas mayores tasas de compresión. Este es el principal motivo por el que nace el estándar H.264/AVC.

Este estándar define un formato y un método de compresión de vídeo, desarrollado conjuntamente por el *Video Coding Experts Group* (VCEG) y el *Moving Picture Experts Group* (MPEG) y con el que se trata de **proporcionar una buena calidad de imagen reduciendo la tasa binaria** respecto a los estándares anteriores [19], [20].

Una secuencia de vídeo se segmenta en fotogramas o *frames*, que se dividen en *slices* y, a su vez, estos son un conjunto de macrobloques, los cuales contienen los datos de luminancia y crominancia (muestras) [20].

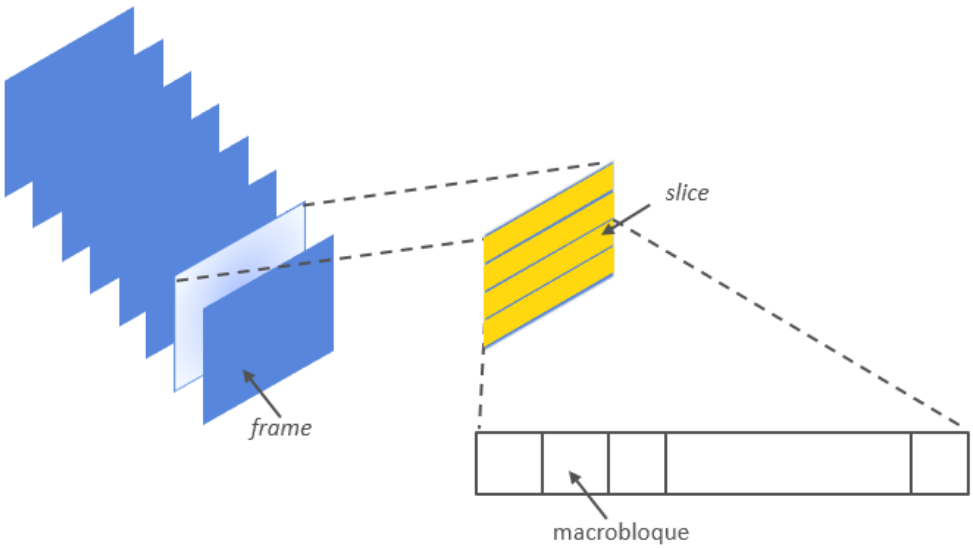


Figura 2.1: Estructura de la secuencia de vídeo

H.264, tanto el codificador como el decodificador, está compuesto por diferentes bloques funcionales como sus antecesores, y adopta un algoritmo híbrido de predicción y transformación para la reducción de la correlación espacial y de la señal residual, control de la velocidad binaria o *bitrate*, predicción por compensación del movimiento para reducir la redundancia temporal, así como codificación de la entropía para reducir la correlación estadística. Sin embargo, los cambios importantes están descritos en los detalles de cada bloque funcional, de manera que lo que proporciona una mayor eficiencia de codificación es **cómo opera cada uno de ellos** [18]. Algunas de las características más destacadas son las siguientes:

- Incluye predicción *intra* fotograma.
- Transformación por bloques de 4x4 muestras, siendo los coeficientes transformados enteros.
- Referencia múltiple para predicción temporal.
- Tamaño variable de los macrobloques a comprimir.
- Precisión de un cuarto de píxel para la compensación de movimiento.

A nivel de decodificación utiliza métodos para incrementar la resistencia a errores, como puede ser el ordenamiento flexible de macrobloques, transmisión de *slices* redundantes de fotograma o particionado de datos [22], [23].

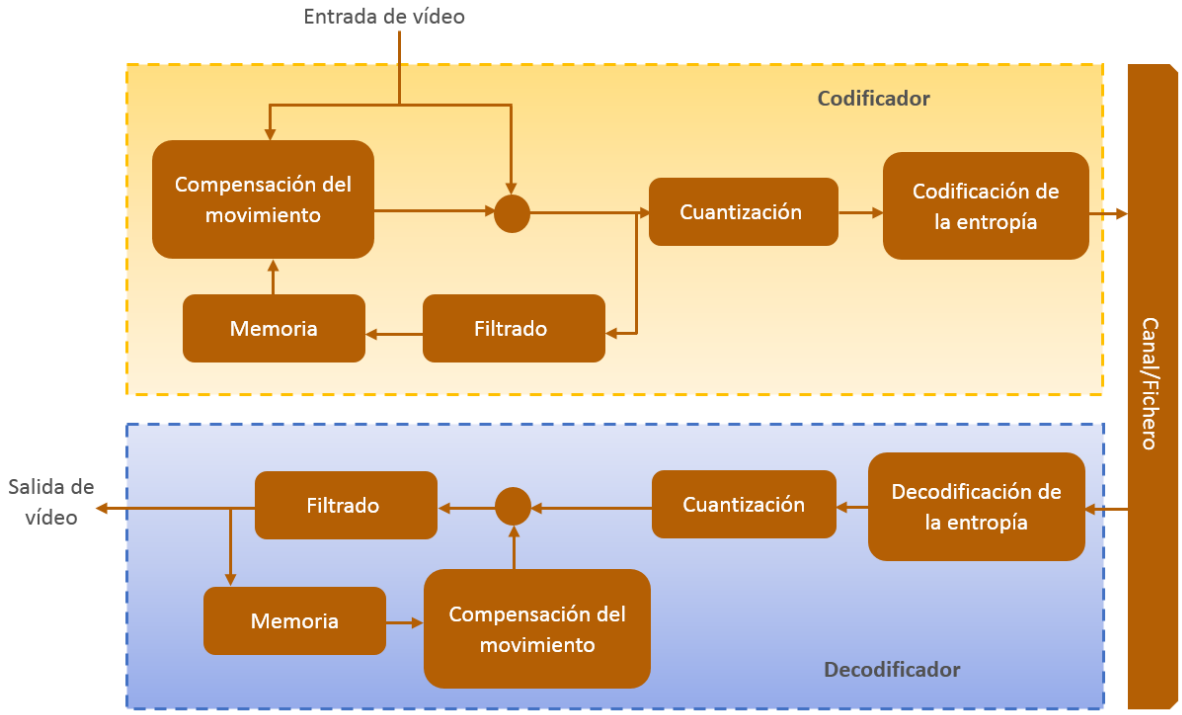


Figura 2.2: Esquema general del estándar H.264/AVC (adaptado de [19])

Las funciones y especificaciones necesarias del codificador y del decodificador se recogen en cada uno de los perfiles que define el estándar (*Baseline*, *Main*, *Extended*). Mientras que las limitaciones de los parámetros (velocidad, tamaño, memoria requerida, etc.) se indican a través de un conjunto de niveles [18].

En cuanto a la estructura, H.264 se compone de dos capas: la capa de abstracción de red (NAL) y la capa de codificación de vídeo (VCL). La primera abstrae los datos para hacer compatible la información de salida del codificador (*bitstream*) con los canales de comunicación o medios de almacenamiento; y la segunda es el núcleo de los datos codificados, la secuencia de vídeo a codificar [24].

En conclusión, H.264/AVC es un estándar nacido a partir de sus predecesores, pero con importantes mejoras respecto a ellos, como son:

- Mejor codificación de la entropía.
- Mejor compensación y predicción del movimiento.
- Bloques pequeños para la codificación.

- Mejor filtro *deblocking*.
- *Bitrate* inferior respecto a los estándares anteriores.
- Mejor calidad de imagen manteniendo la misma relación S/N.

De esta manera se consigue una mayor eficiencia en la compresión, mejor calidad de vídeo, pudiendo ser esta de alta resolución; y flexibilidad en la compresión, transmisión y almacenamiento de vídeo [19]. Todo esto a cambio de un **aumento de la complejidad de la arquitectura *hardware* del sistema**.

2.3. Decodificador de vídeo

El perfil básico del estándar H.264 es el *Baseline*. En este, los coeficientes cuantizados (*bitstream*) se decodifican utilizando un decodificador cuyo código es de longitud variable. Posteriormente, el error residual se descomprime con una transformada y se elimina con la cuantización, para luego realizar la predicción y finalmente el filtrado, reduciendo así las distorsiones introducidas durante los procesos anteriores. En la Figura 2.3 se puede observar el esquema general del decodificador de vídeo.

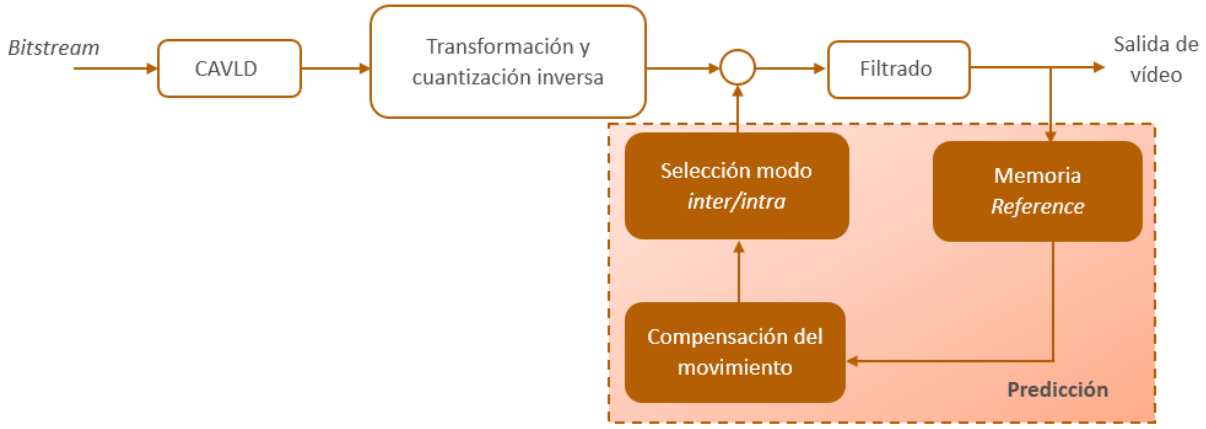


Figura 2.3: Esquema del decodificador H.264/AVC (adaptado de [21])

2.3.1. Decodificación del *bitstream*

En los estándares anteriores la codificación de la entropía se basa en tablas previamente definidas y que contienen los códigos que se van a utilizar, siendo estos de longitud variable. El conjunto de

códigos incluidos en dichas tablas se fundamenta en distribuciones de probabilidad de datos obtenidos en secuencias de vídeo genérico, en lugar de usar la codificación Huffman o aritmética exacta [25].

Sin embargo, H.264/AVC hace uso de diferentes códigos de longitud variable con la finalidad de igualar el símbolo que representa un dato de vídeo con un código basado en las características del contexto en el que se encuentra el mismo. Se realiza una búsqueda en zigzag o una búsqueda alternada (campos de fotograma de vídeo no entrelazados) con el objetivo de leer los datos residuales (coeficientes transformados y cuantizados). Para la decodificación de estos, utiliza un método más avanzado, usando un código adaptativo de longitud variable basado en el contexto (CAVLC) [20].

2.3.2. Transformación y cuantización

En H.264 los procesos de transformación y cuantización están diseñados para minimizar la complejidad de la implementación y que esta pueda realizarse de manera más simple, y por consiguiente, evitar la redundacia espacial. Esto se logra utilizando varias técnicas:

- Reduciendo el tamaño de los pasos de los cuantizadores para así aumentar la relación pico de señal a ruido (PSNR) a niveles que se consideren sin pérdidas desde un punto de vista visual. El rango de los pasos de cuantización se eleva en dos octavas, implicando una redefinición de las tablas de cuantización.
- Usando una transformada entera de 4x4 o de 8x8, que integra el proceso de transformación, cuantización y escalado. Este proceso se denomina transformación entera con post-escalado y disminuye el número de multiplicaciones.

En algunos casos, además, emplea una estructura de transformación jerárquica, es decir, a los coeficientes obtenidos anteriormente se les aplica la DCT y a la salida de esta la transformada de Hadamard [19], [20], [26].

2.3.3. Predicción

Un codificador H.264/AVC hace un uso eficiente de la redundancia temporal y espacial de la información visual que se va codificando, así disminuye la cantidad de información que se necesita para la reproducción del vídeo. En el sentido temporal, procesa cada *slice* para buscar y comparar las texturas de los macrobloques anteriores y siguientes, una técnica que es conocida como estimación de movimiento. Por cada coincidencia que detecta, en el decodificador solo se necesita un vector que apunte

a ella (textura de referencia) y la información requerida para hacer la corrección de cualquier posible diferencia o error. Cuando la estimación de movimiento no proporcione coincidencias suficientes, se pasa a trabajar en el sentido espacial, donde el codificador utiliza la textura de los macrobloques próximos dentro del mismo *slice* para hacer la predicción, almacenando solamente la diferencia entre esta y la textura real [27]. Las imágenes que previamente han sido decodificadas son almacenadas en un *buffer* llamado *Decode Picture Buffer* (DPB), donde se crea la lista con las imágenes de referencia.

Una predicción del macrobloque actual es un modelo que se asemeja al mismo tanto como sea posible y se crea a partir de muestras de una imagen que ya ha sido codificada. Esta predicción es restada al macrobloque actual y el resultado de dicha resta, llamado residuo, se comprime y se transmite al decodificador.

Existe dos tipos de predicción: *intra*, las muestras de un macrobloque se han predicho con muestras pertenecientes al *slice* actual que ya hayan sido codificadas, decodificadas y reconstruidas; e *inter* donde las muestras en un macrobloque se han predicho de muestras previamente codificadas [19], [20].

2.3.4. Filtrado

El proceso de decodificación involucra macrobloques con distintas características, algunos con mayor correlación que otros. Para mantener cierta tasa binaria, los bloques *intra* e *inter* se han cuantizado utilizando diferentes cuantizadores, los cuales introducen una distorsión alrededor de los bloques construidos [28]. La imagen filtrada se usa para compensar el movimiento, por lo que se requiere que sea todo lo posible una reproducción fiel a la original.

El filtro de desbloqueo reduce la distorsión en los bordes del bloque y evita que el ruido acumulado debido a la codificación se propague. En H.264 el filtrado se aplica adaptativamente en varios niveles [18]:

- A nivel de *slice*. El filtrado se puede ajustar a las características individuales de la secuencia de vídeo.
- A nivel de borde del bloque. El filtrado se vuelve independiente de la decisión *intra/inter* de las diferencias de movimiento y de la presencia de residuos codificados en los dos bloques participantes.

- A nivel de muestra. El efecto del filtrado se puede anular dependiendo de los valores de las muestras y de los umbrales del cuantizador.

2.4. Comparativa

La tabla 2.1 muestra un análisis comparativo entre los estándares de codificación de vídeo [28].

2.5. Aplicación de referencia

Como se indicó en el Capítulo 1 el objetivo principal de este proyecto es la **definición de una metodología de síntesis de sistemas electrónicos integrados a partir del modelado a nivel de transacciones**. Para ello se hará uso del decodificador de vídeo H.264/AVC.

Se parte de un diseño en alto nivel descrito en SystemC desarrollado a partir del código de referencia del decodificador H.264/AVC por el Instituto Universitario de Microelectrónica Aplicada [29], [30]. En la Figura 2.4 se tiene un esquema de los módulos que lo componen. En este Proyecto Fin de Carrera se va a utilizar únicamente el bloque ***inter_p***.

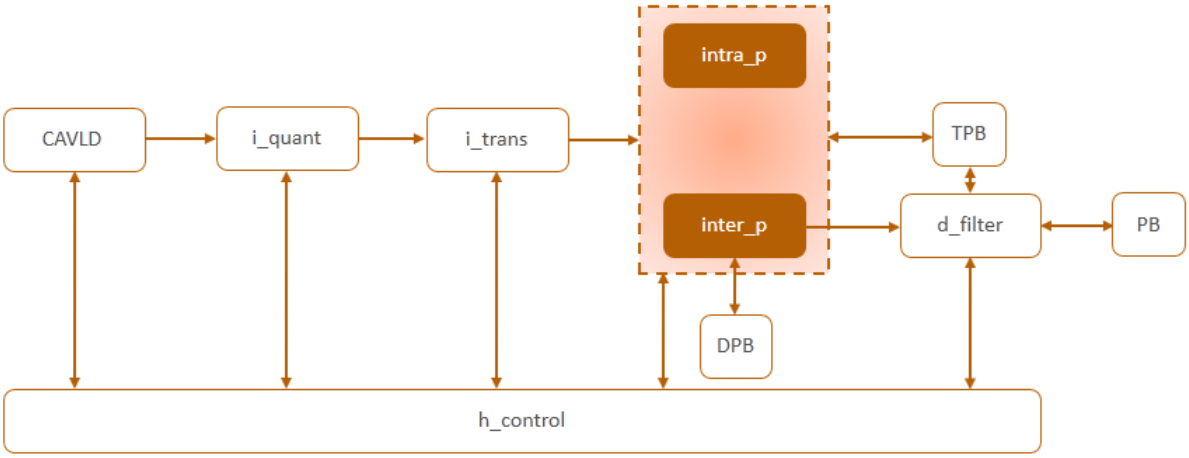


Figura 2.4: Decodificador de vídeo H.264/AVC perteneciente al IUMA (adaptado de [29])

2.5.1. Bloque *inter_p*

El bloque *inter_p* pertenece al flujo de procesado de la predicción *inter*: crea macrobloques de predicción a partir de los vectores de movimiento y del fotograma de referencia.

	MPEG-2	**MPEG-4**	**H.264/AVC**
Tamaño del MB	16x16, 16x8	16x16	16x16
Tamaño de particiones	8x8	16x16, 16x8, 8x8	16x16, 16x8, 8x16, 8x8, 8x4, 4x8, 4x4
Predicción *intra*	No	Dominio de la transformada	Dominio espacial
Transformada	DCT de 8x8	DCT - Wavelet de 8x8	DCT entera 8x8, 4x4; Hadamard 4x4, 2x2
Cuantización	Escalar con tamaño constante de los pasos	Vectorial	Escalar con tamaño entre pasos de 12,5 % del *bitrate*
Codificación de la entropía	VLC	VLC	VLC, CABAC, CAVLC
Exactitud de las muestras	$\frac{1}{2}$ muestra	$\frac{1}{2}$ muestra	$\frac{1}{4}$ muestra
Fotogramas de referencia	1	1	Múltiples
Predicción con peso	No	No	Sí
Filtro de desbloqueo	No	No	Sí
Tipos de fotogramas	I, P, B	I, P, B	I, P, B, SI, SP
Perfiles	5	8	7
Acceso aleatorio	Sí	Sí	Sí
Resistencia a errores	Particionamiento de datos, FEC para transmisión de paquetes por importancia	Sincronización, particionamiento de datos, extensión de encabezados	Particionamiento de datos, ajuste de parámetros, orden flexible de macrobloques, *slices* redundantes
Velocidad de transmisión	2-15Mbps	64Kbps-2Mbps	64Kbps-150Mbps
Complejidad del codificador	Media	Media	Alta

Tabla 2.1: Comparación entre los diferentes estándares de codificación de vídeo

El modelado a nivel de transacciones consiste en abstraer las comunicaciones de la funcionalidad para poder trabajar con cada capa por separado. Por esta razón, se considerará que **la funcionalidad del módulo está completamente definida y el área de interés es la interfaz**, donde se encuentra el protocolo de comunicación.

2.5.2. Protocolo de comunicación

Utiliza un **protocolo de *handshake***, en el que existe un bloque que envía los datos y otro que los recibe. El primero manda una petición (REQ), y el segundo prepara los datos y emite la validación de los mismos (ACK). Tras esto, el bloque que ha pedido los datos, los recibe y elimina la petición. En este caso consta de las señales:

- ***data*,** los datos que se van a procesar.
- ***request*,** solicitud de los datos (REQ).
- ***validate*,** indica que los datos ya están preparados para ser enviados o recibidos (ACK).

En función del módulo con el que se comunique usará uno de los dos esquemas que se presentan a continuación.

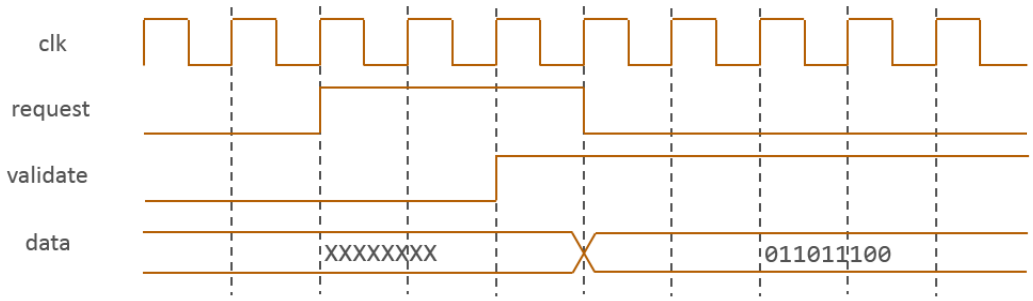


Figura 2.5: Protocolo de comunicación: Recepción de datos

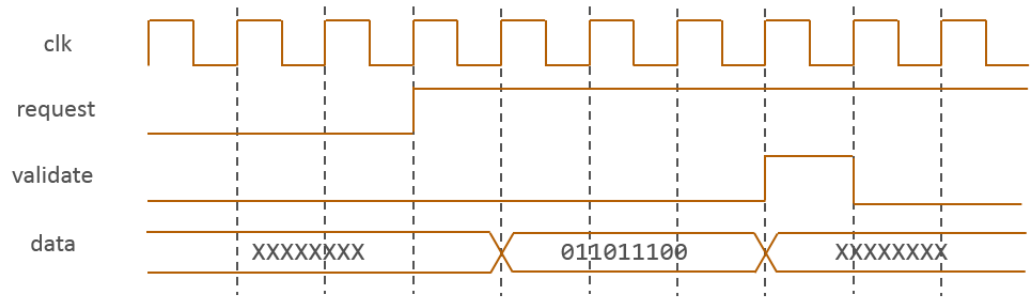


Figura 2.6: Protocolo de comunicación: Envío de datos

En el primer caso, el que se muestra en la Figura 2.5, el módulo solicitante de los datos enviará una solicitud (*request*). Cuando pueda obtener los datos, se activará la validación (*validate*), se desactivará la petición, señal *request*, y en el ciclo de reloj siguiente podrá empezar a recibir los datos.

Por otra parte, para el segundo caso (Figura 2.6), se ha recibido un *request*, por lo que se ha de proceder a mandar los datos. Estos se preparan y cuando estén listos se activa la señal *validate* durante un ciclo de reloj, así el receptor sabrá que puede empezar a leer los datos.

2.5.3. Solución desarrollada

En este proyecto se dotará de interfaces TLM al bloque *inter_p* del decodificador de vídeo. Para ello, con el fin de comprobar la viabilidad de la metodología a nivel de transacciones y comparar los resultados obtenidos después de la síntesis, se seguirán dos líneas de trabajo:

- **Modelado TLM con *wrapper* que adapte la interfaz TLM a la interfaz del bloque**, de manera que internamente el módulo continuará con su protocolo de comunicación, pero externamente estará implementado con TLM. Después de aplicar esta solución el sistema queda como se puede ver en la siguiente figura.

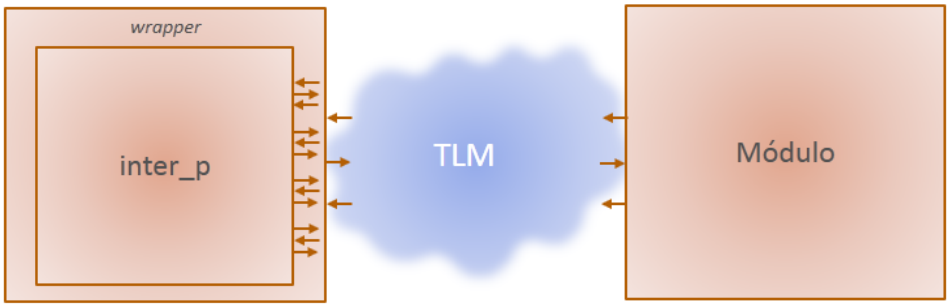


Figura 2.7: Sistema tras la implementación con *wrapper* TLM

- **Modelado TLM de la interfaz**. No será necesario el uso de un protocolo de comunicación ya que TLM lo genera internamente, no visible para el diseñador. El sistema resultante es el que se muestra a continuación.

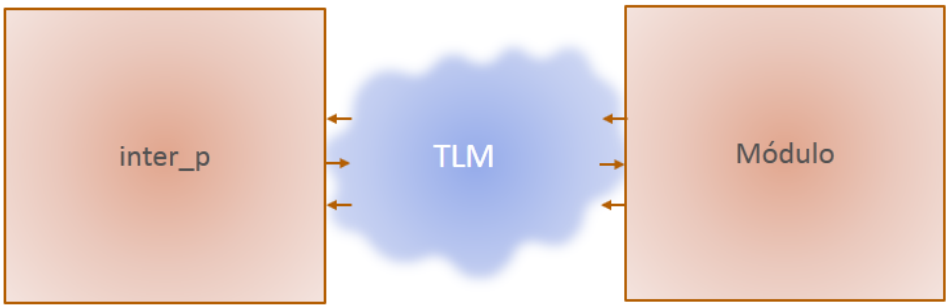


Figura 2.8: Sistema tras la implementación TLM de la interfaz

2.6. Conclusiones

H.264/AVC es uno de lo estándares de compresión de vídeo más usado. Se basa en sus predecesores, donde las diferencias más notables se encuentran en los módulos funcionales que componen el codificador y el decodificador de vídeo. Aunque su arquitectura es compleja ofrece una buena relación calidad de vídeo/*bitrate*, siendo este su punto fuerte, pues le permite ser enviado a través de cualquier canal o ser almacenado en un fichero, es decir, adaptarse a los medios actuales.

Con el desarrollo de nuevas metodologías, flujos de diseño, etc. se busca aumentar y mejorar la productividad de los diseñadores, lo que conlleva reducir el tiempo y el coste de la implementación, como la del decodificador H.264/AVC.

Se seguirán dos líneas de trabajo para verificar la metodología a nivel de transacciones y poder comparar los resultados obtenidos tras la síntesis:

- Modelado TLM con *wrapper*.
- Modelado TLM de la interfaz.

Capítulo 3

Metodología de diseño

3.1. Introducción

El diseño de un sistema electrónico consiste en la transformación de su especificación funcional en una implementación física del mismo. Para realizar esta transformación se utiliza una metodología formada por un conjunto de procesos que pretenden alcanzar unos objetivos concretos, entendiendo como método el procedimiento utilizado para llegar a un fin siguiendo un cierto orden.

Para llevar a cabo la realización de este proyecto, y en consecuencia, lograr los objetivos propuestos, se utiliza **una metodología orientada a la síntesis de alto nivel**, partiendo de una descripción en SystemC, y a la que se le aplicará el modelado a nivel de transacciones (TLM). Con el fin de obtener modelos sintetizables para, si corresponde, una posterior implementación *hardware* tanto en FPGA como en ASIC, se seguirán dos líneas de actuación:

1. Síntesis de alto nivel para ASIC.
2. Síntesis de alto nivel para FPGA.

A continuación se realizará una descripción detallada de la metodología, resaltando el modelado a nivel de transacciones. Además, se presentarán aquellos elementos que caracterizan las distintas etapas en las que consiste el diseño del sistema, como pueden ser los lenguajes o las herramientas necesarias.

3.2. Flujo de diseño

Cabe destacar tres etapas importantes en la metodología o flujo de cualquier diseño, sea *software* o *hardware*: caracterización, diseño y verificación. Se trata de un proceso iterativo en el que se va refinando el sistema hasta obtener la versión a implementar.

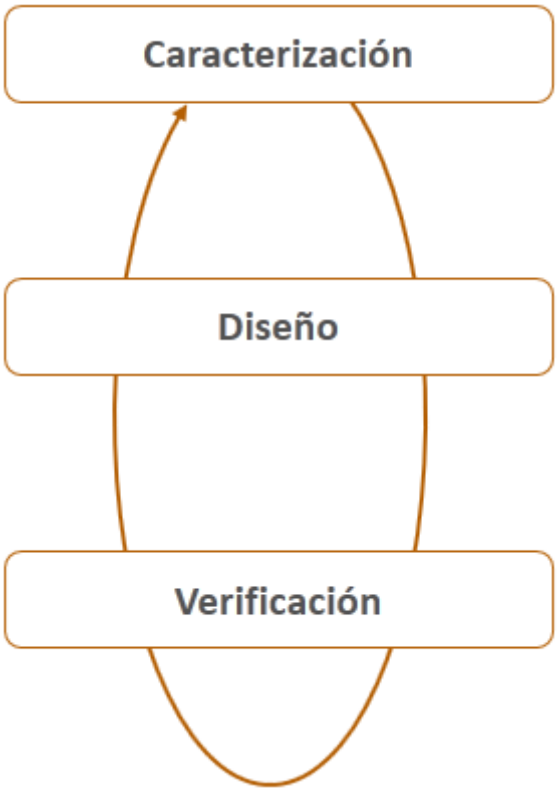


Figura 3.1: Flujo de diseño básico

Antes de comenzar a abordar el diseño de un sistema es necesario poder describirlo de forma que permita establecer su funcionalidad y prestaciones requeridas. Con ello el diseñador puede tomar decisiones como la tecnología a utilizar, lenguajes, herramientas, colocación de los bloques, etc. y así obtener una primera versión funcional. Mediante simulación, se verifica que cumple con las especificaciones definidas, para que en caso de no ser así, aplicar las medidas correctoras necesarias y repetir el ciclo de diseño.

El flujo de diseño tradicional de un ASIC es el que se muestra en la Figura 3.2. En cambio, el flujo de diseño basado en FPGA es más simple, ya que la fase de implementación se reduce a la programación del dispositivo, que cuenta con una arquitectura y recursos predefinidos. Las descripcio-

nes a nivel de sistema son comunes en ambos flujos, mientras que las fases de síntesis RTL y lógica estarán optimizadas para la tecnología destino, pues son las que traducen la descripción en un nivel de abstracción superior a una descripción estructural con componentes del nivel de abstracción inferior.

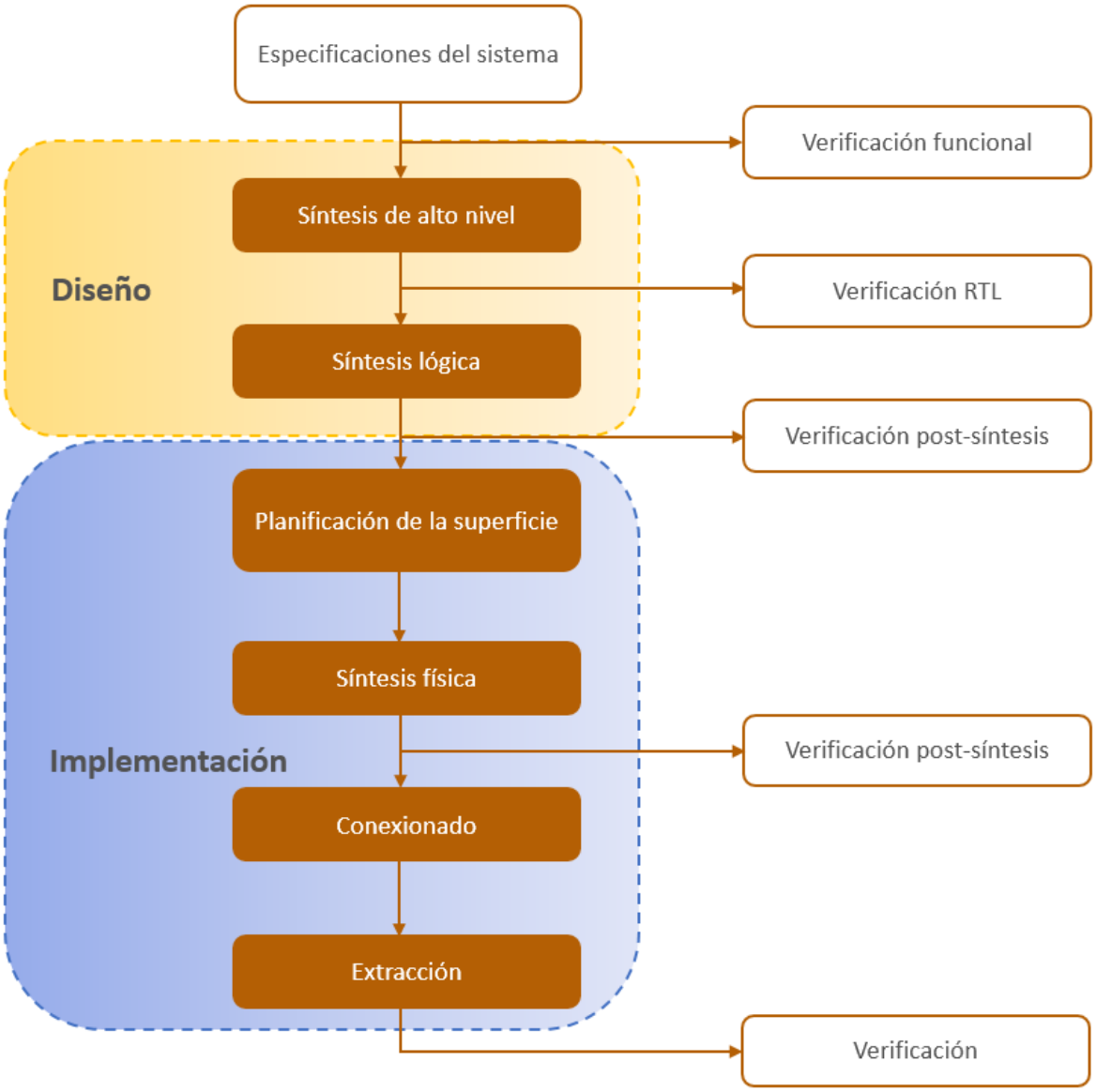


Figura 3.2: Flujo de diseño de un ASIC (adaptado de [31])

La descripción RTL es consecuencia de la planificación temporal y esta se obtendrá en función de los requerimientos de tiempo de ciclo, área y potencia.

3.3. Lenguaje: SystemC

Para realizar el modelado a nivel de transacciones se ha usado **SystemC**, lenguaje que permite la descripción de sistemas electrónicos a nivel de sistemas (ESL) [12], adecuar los niveles de abstracción y proveer los elementos que permiten llevar a cabo la separación entre la funcionalidad y la comunicación.

SystemC es un lenguaje de diseño y verificación de sistemas electrónicos de alto nivel, estandarizado por el IEEE (IEEE 1666-2011) [3] para modelar sistemas. Está implementado como un conjunto de clases de C++ que **soporta concurrencia, noción del tiempo y tipos de datos específicos para *hardware*.** Está consolidado y es ampliamente usado. Su popularidad se atribuye a las siguientes características [32]:

- La tendencia de la industria se inclina hacia la reusabilidad de IPs frente al desarrollo interno.
- Consolidación de los flujos de diseño a nivel de sistemas ya existentes.
- Integración con los diferentes niveles de abstracción.
- Librería de código abierto (*open-source*).

Además, soporta la simulación basada en eventos, la cual presenta unos tiempos de simulación más bajos; y el trazado de formas de onda.

Con SystemC **la funcionalidad puede ser separada de la comunicación**. La primera es descrita usando módulos y procesos, mientras que la segunda mediante canales e interfaces. Esto permite el modelado de ambas partes en diferentes niveles de abstracción [33].

Además de ser una librería de clases C++, SystemC soporta una metodología de diseño que tiene como objetivo acelerar el ciclo de diseño. Entre sus ventajas se encuentra [35]:

- **Refinamiento.** No es necesario realizar una conversión desde un nivel de C/C++ a un lenguaje de alto nivel, sino que el diseño es refinado gradualmente, incluyendo las restricciones temporales y de *hardware*.
- El **tiempo de desarrollo se reduce**, tanto el de diseño como el de verificación.
- SystemC permite describir y verificar el diseño, **y posteriormente sintetizarlo.**

- **Los *testbench* pueden ser reutilizados**.

Las características y ventajas de SystemC hacen de él **un lenguaje adecuado para el diseño a nivel de transacciones**, pues incluye las interfaces y métodos necesarios para el mismo, dispone de FIFOs, interfaces de transporte bloqueantes y no bloqueantes, de acceso directo a memoria, de depurado, modelos de iniciadores y destinos, y modelos de *payload* genéricos; y además, soporta diferentes niveles de modelado temporal (*untimed*, *loosely-time*, *approximately-time* y *cycle-accurate*).

3.3.1. Estructura

Un sistema descrito en SystemC consiste en un conjunto de módulos que se comunican entre ellos a través de canales o eventos. La estructura viene definida por módulos, interfaces, canales y puertos, tal como se describen a continuación [3], [34], [35]:

- **Módulos.** Se usan para modelar la estructura del diseño y contienen procesos concurrentes que describen la funcionalidad del sistema. Acceden a los canales a través de los puertos.

```
SC_MODULE (nombre_módulo) {
    // Cuerpo
}
```

- **Interfaces.** Son un conjunto de métodos que se utilizan para la comunicación entre los canales. Se basan en una clase abstracta de C++, donde únicamente hay métodos puros. Algunas de las interfaces que existen son `sc_signal_in_if`, `sc_signal_inout_if`, `sc_fifo_in_if` o `sc_mutex_if`.

- **Canales.** Representan el camino que sigue la comunicación entre dos procesos. Puede ser una simple señal o una estructura compleja, con procesos (canal jerárquico). Se accede a ellos a través de las interfaces. Las señales son internas a cada módulo y no son visibles desde otros módulos. Un ejemplo de canales primitivos son `sc_signal` o `sc_fifo`.

```
sc_signal<tipo_dato>nombre_señal_1, nombre_señal_2...
```

- **Puertos.** Son el punto de acceso a los módulos. Están relacionados con el tipo de interfaz con el que se conectan.

```
sc_port<tipo_interfaz>nombre_puerto_1, nombre_puerto_2...
```

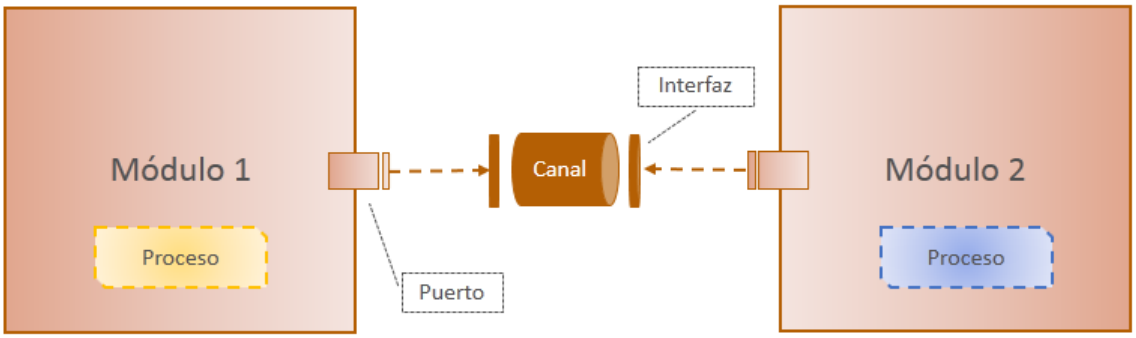


Figura 3.3: Arquitectura de SystemC (adaptado de [33])

3.3.2. Procesos

En ellos se describe la funcionalidad del módulo. Se ejecutan concurrentemente, aunque cada proceso tiene un único hilo de ejecución. Deben ir en un módulo, no se puede crear un proceso dentro de una función. Sin embargo, un módulo puede contener varios procesos.

SystemC distingue tres tipos de procesos [3], [32]:

- `SC_METHOD`. Se ejecutan cuando ocurre un evento en su lista de sensibilidad y no pueden ser suspendidos. Se suelen usar para simular el comportamiento combinacional, ya que se ejecutan en tiempo cero de simulación.

```
SC_MODULE (module){
      SC_METHOD (method);
      sensitive«in;
}

void method (){
      out = in;
}
```

- `SC_THREAD`. Son ejecutados una vez al comienzo de la ejecución del sistema, aunque pueden quedar suspendidos por una llamada a la función `wait()`. En este caso, el proceso se reanudará en el mismo estado en el que fue suspendido cuando se produzca un evento. Si se indica el tiempo de suspensión del proceso, por ejemplo `wait(10_SC_NS)`, entonces la ejecución se reanudaría a los 10 ns. En general, este tipo de procesos suele definirse como un bucle infinito, de forma que cuando termina vuelve a ejecutarse.

```
SC_MODULE (module){
      SC_CTHREAD (thread);
      sensitive«in;
}

void thread (){
      for (i=1;i<10;i++){
            wait();
            out = in;
      }
}
```

- `SC_CTHREAD` o proceso de reloj. Se trata de una modificación sobre el tipo anterior. Su principal diferencia es que en el registro se le añade un evento como sensiblidad, de forma que en la definición pueden realizarse nuevas formas de suspensión. Son usados principalmente en síntesis.

```
SC_MODULE (module) {
      SC_CTHREAD (cthread, clk.pos());
}

void cthread () {
      wait();
      while (true) {
            out = in;
      }
}
```

3.3.3. Relojes

Los relojes son señales con una notación temporal que permite dotar al sistema de relaciones temporales durante la simulación.

La clase `sc_time` representa el tiempo. Es un entero de 64 bits. El tiempo de resolución es programable, tiene que ser potencia de 10 fs, y solo se puede poner una vez, antes de usarla y de simulación; por defecto es 1 ps. Las unidades que se pueden representar son: `SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC` [33].

`sc_clock` es un objeto que genera una forma de onda periódica. Tiene cuatro parámetros importantes: periodo, ciclo de trabajo, primer flanco y si el primer flanco es positivo (`posedge()`) o negativo (`negedge()`) [35].

```
sc_clock nombre_reloj (“nombre”, periodo, unidades_periodo, ciclo_trabajo,
                       desfase, unidades_desfase);
```

3.3.4. Ejemplo

En esta sección se presenta un ejemplo completo del diseño de un sumador (*adder*) descrito en SystemC. El modelo comprende dos ficheros: la cabecera (.h) y la definición (.cpp). En el primero se declaran los puertos y procesos, mientras que en el segundo se describe la funcionalidad.

Para comprobar el correcto funcionamiento del sumador se debe hacer uso de un *testbench*, que será el que genere las entradas, y a su vez recibirá la salida para permitir monitorizarla.

Por tanto, el diseño lo conforman tanto el *testbench* como el *adder* (UUT), formando una estructura jerárquica junto al `sc_main`, el cual conecta los dos módulos e indica el comienzo del procesamiento con `sc_start()`.

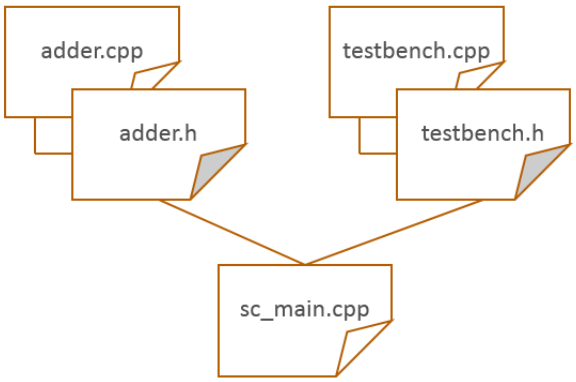


Figura 3.4: Estructura de ficheros del modelo SystemC de un sumador

3.3.4.1. adder.h

```
# include <systemc.h> // Se incluyen las librerías necesarias

SC_MODULE (adder{ // Módulo
        // Puertos
        sc_in<int>in1, in2; // in1 e in2 son dos puertos de entrada de tipo entero
        sc_out<int>out; // out es el puerto de salida de tipo entero

        void add(); // Función
```

```
        // Constructor.
        SC_CTOR (adder){
            // La función add se va a iniciar cuando in1 o in2 sufran algún cambio
            SC_METHOD (add);
            sensitive«in1 «in2;
            }
};
```

El constructor es una función del mismo nombre que la clase, pero sin el tipo de retorno. Se utiliza para crear y para inicializar un módulo y las estructuras de datos del mismo. Además, se registran los procesos y los módulos *instanciados* dentro de él.

3.3.4.2. adder.cpp

```
# include <adder.h> // Se incluye la cabecera

void adder::add() {
    // Operación suma
    out = in1.read() + in2.read();
}
```

3.4. Modelado a nivel de transacciones

La complejidad de los sistemas electrónicos actuales es cada vez mayor. El objetivo es un SoC de pequeño tamaño que incluya diferentes IPs periféricos, buses, múltiples procesadores y memorias. Además, se persigue una mayor velocidad de simulación, desarrollo simultáneo de *software*, poder acceder al mercado lo antes posible (*time-to-market*), y todo esto a un bajo coste. Los diseñadores no son capaces de cubrir las deficiencias de los diseños tradicionales, por lo que tratan de desarrollar nuevas metodologías, como por ejemplo [47]:

- ***Hardware Emulation*:** Es muy costoso y está limitado por las capacidades de depuración. Además, se debe esperar por los diseños RTL.

- ***Cycle Accurate Model*:** Requiere un gran esfuerzo en el modelado y es difícil hacer frente a la evolución RTL.

Ante este panorama se presenta un nuevo nivel de abstracción, superior a RTL, donde **los detalles de tiempo no son considerados y la comunicación entre módulos está modelada por canales** en lugar de cables. Este nivel es llamado nivel de transacciones y el acto de modelar sistemas en él es conocido como **modelado a nivel de transacciones (*Transaction-Level Modeling,* TLM)** [10].

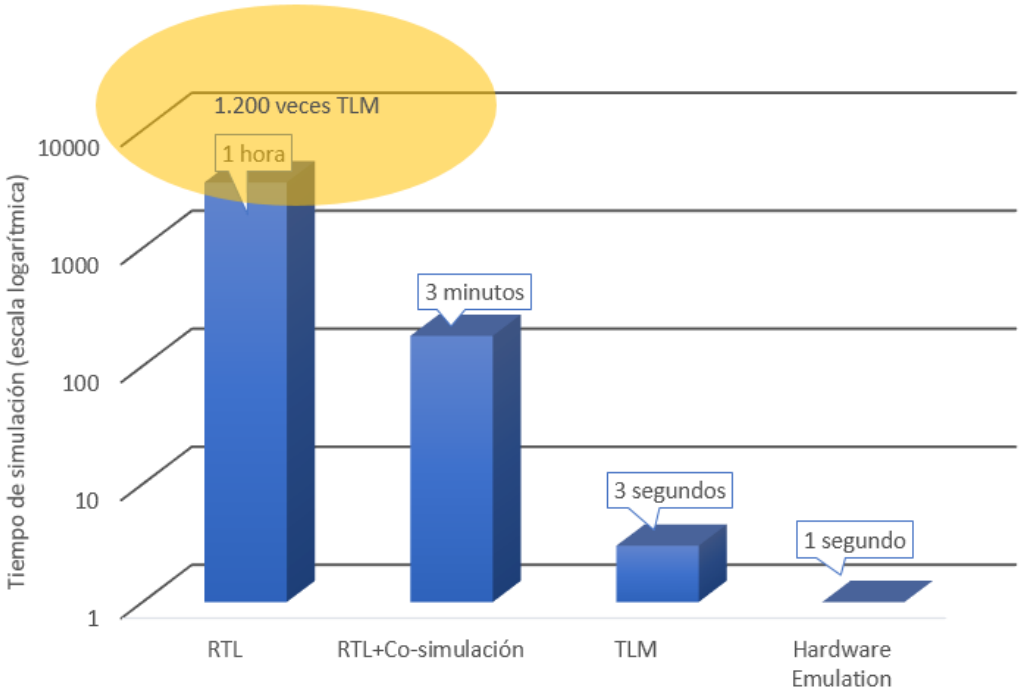


Figura 3.5: Velocidad de simulación de un *frame* de un decodificador MPEG-4 en modelos a diferentes niveles de abstracción (adaptado de [47])

La metodología basada en transacciones fue creada en busca de un nuevo enfoque que pudiera permitir la representación de un diseño en un nivel de abstracción intermedio entre las especificaciones y los modelos RTL. Una solución de alto nivel que integra el diseño a nivel de sistemas electrónicos con la implementación a nivel de bloques.

Actualmente se ha publicado la versión 2.0 de esta metodología [3], donde está incluida la definición de TLM 1.0, siendo este la base del modelado a nivel de transacciones y cuyas características se exponen en los siguientes apartados.

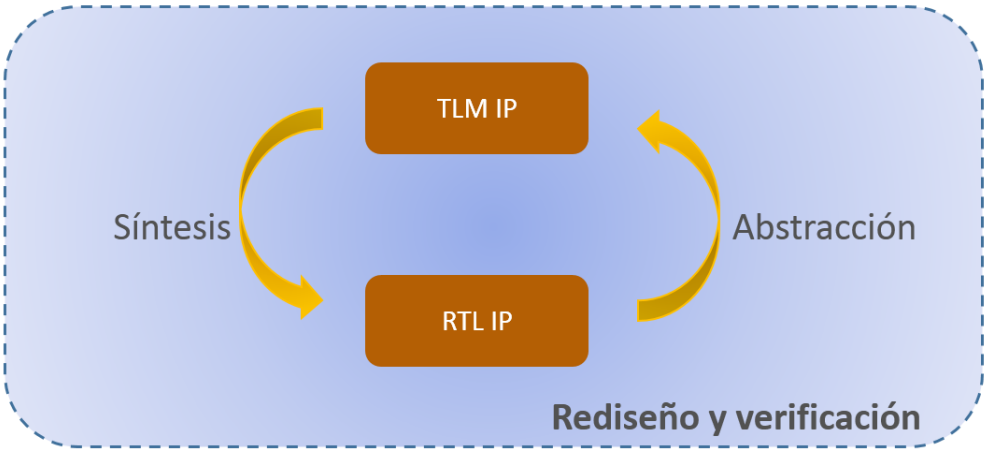


Figura 3.6: Enfoque buscado (adaptado de [51])

3.4.1. Características generales

La idea principal de esta metodología se basa en **separar las comunicaciones de los diferentes módulos que componen un sistema, de la funcionalidad de los mismos**. Para ello, el modelado debe cumplir con una velocidad de simulación y una precisión razonable, así como, debe ser ligero y fácil de desarrollar.

3.4.1.1. Modelos

A continuación se detallan algunos modelos que intentan elevar su nivel de abstracción [4]:

- **Modelo de precisión de ciclos (*Cycle-Accurate*).** Trata de sustituir las partes *hardware*, a excepción del procesador, por un modelo en un nivel superior de abstracción. Puede ser desarrollado utilizando lenguajes como C. Comparado con los modelos RTL es menos preciso. Sin embargo, es sensible a los ciclos de reloj, lo cual es suficiente para la verificación del *software*, aunque no para una versión sintetizable.
- **Modelo temporal.** Aunque los modelos temporales permiten un incremento del nivel de abstracción al centrarse en el análisis del rendimiento del sistema, no se garantiza la precisión a nivel funcional, por lo que no se trata de una buena alternativa para el modelado del sistema.

3.4.1.2. Precisión del modelado

La precisión indica la exactitud del modelo para reproducir el comportamiento y las actividades del sistema a diseñar. Hay dos factores decisivos para determinar el grado de precisión del modelado [3], [4]:

- **Granularidad de los datos.** Refleja el tamaño de las piezas en que se dividen los datos transportados. En función de la exactitud existen tres niveles: paquete de aplicación, paquete de bus y tamaño del bus. Por ejemplo, para el caso de una transferencia de un vídeo IP, si este tiene un algoritmo basado en fotogramas podría aplicarse una granularidad gruesa al paquete de aplicación como una transferencia fotograma a fotograma; una granularidad más fina, a nivel de paquetes de bus, como una transferencia de una línea, una columna o un macrobloque; y a nivel del tamaño del bus sería una transferencia de un píxel de vídeo.

- **Precisión del tiempo.** Determina la fidelidad de un modelo para el comportamiento temporal. Puede ser entendido como una escala de dos extremos: *untimed* (nivel sin tiempo), se representa la funcionalidad de manera pura, sin añadir ningún detalle de la implementación ni comportamiento temporal; y *timed* (nivel de precisión de ciclos), contiene detalles de la implementación, además el tiempo es usado en la ejecución, modelando los retardos y latencias. La transición de un extremo a otro determina la exactitud del modelo y cualquier punto intermedio se considera como *approximately-timed* (nivel temporal aproximado), donde se especifican algunos detalles de la implementación, como puede ser la arquitectura seleccionada.

Usando un alto nivel de abstracción, los detalles innecesarios se pueden evitar en las fases iniciales del flujo de diseño. Además, se obtiene un aumento de la velocidad de simulación y de la productividad de modelado, lo que facilita el diseño de nuevas metodologías. El modelo TLM es el referente común para todos los equipos de trabajo de *software*, *hardware*, análisis de arquitectura y verificación [12].

3.4.2. Estructura

En los modelos TLM se pueden distinguir dos partes [48]:

- El núcleo, provee de la funcionalidad a cada componente individual del SoC. Se conoce como **módulo**.

- Las **interfaces**, que permiten la comunicación entre dos núcleos. Transfieren los datos en forma de transacción a través de los **puertos** que conectan con el núcleo.

Los distintos mecanismos de comunicación se modelan como **canales** y se usan en los módulos con interfaces de SystemC. Una transacción se genera con una **llamada a una función (*function call*)** en dichos canales, que encapsula los detalles de implementación de bajo nivel. De esta forma, las transacciones guardarán detalles como quién genera los datos, a quién van dirigidos, etc. Además, se disminuyen los tiempos de simulación, y permite a los diseñadores de sistemas evaluar distintas

arquitecturas de comunicación de forma sencilla y rápida [49].

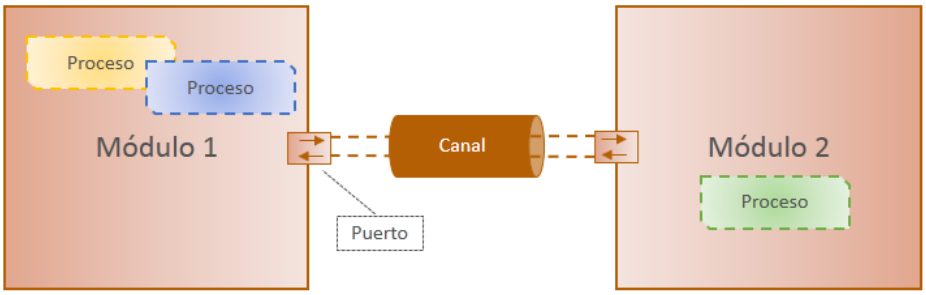


Figura 3.7: Esquema básico TLM (adaptado de [50])

La sincronización del sistema es una acción específica entre dos módulos y necesaria para el correcto funcionamiento del mismo.

El marco de modelado TLM está compuesto por las clases de infraestructura de SystemC [38]:

- `sc_interface`, clase base de las interfaces.
- `sc_module`, clase base de los módulos.
- `sc_export`, permite a un módulo proporcionar explícitamente una interfaz dada.
- `sc_port`, especifica la interfaz requerida por el módulo. Un `sc_port` debe estar enlazado a un `sc_export` o estar conectado directamente a un módulo.

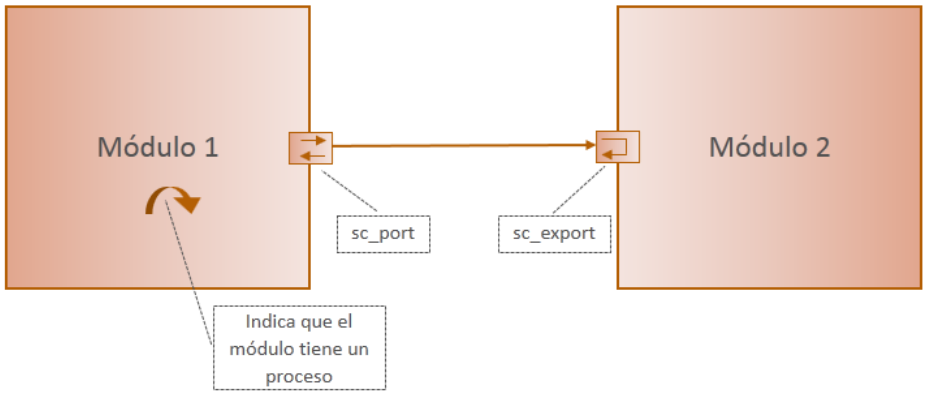


Figura 3.8: Diagrama de la estructura básica de un modelo TLM (adaptado de [38])

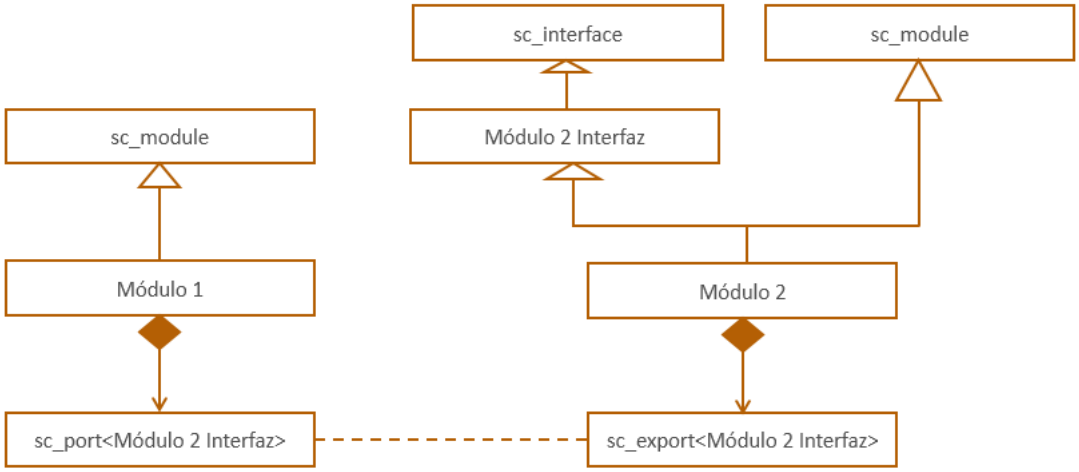


Figura 3.9: Diagrama de relaciones entre dos módulos (adaptado de [38])

3.4.3. Interfaces

La interfaz es la parte más importante del modelo TLM, consiste en uno o más métodos que definen cómo acceder al canal y cómo será el transporte de los datos **(transacción)**. Pueden ser implementadas en un canal específico o directamente en el módulo receptor.

Las clases de las interfaces se basan en SystemC. Son inherentemente más reutilizables, modulares e interoperables [52], [53].

Como ya se había comentado, dos de los tipos básicos de procesos son `SC_THREAD` y `SC_METHOD`. La diferencia radica en la posibilidad de suspender un `SC_THREAD` con una llamada a la función `wait()`, mientras que el `SC_METHOD` solo puede ser sincronizado haciéndolo sensible a un evento externo. En base a esto existen dos tipos de interfaces: ***blocking* (bloqueante)** y ***non-blocking* (no-bloqueante)** [53].

	Contiene `wait()`	Puede ser llamado por
Blocking	Puede implementarlo	`SC_THREAD` únicamente
Non-Blocking	No	`SC_THREAD` o `SC_METHOD`

Tabla 3.1: Interfaces *blocking* y *non-blocking*

Además, las interfaces pueden ser **unidireccionales o bidireccionales**. En el primer caso, la información es transmitida en una única dirección; y en el segundo, en ambas direcciones.

Interfaz	Descripción	
`tlm_transport_if`	Transporta un *request* y devuelve una respuesta	Bidireccional y bloqueante
`tlm_reset_transport_if`	*Reset*	Bidireccional
`tlm_blocking_put_if`	Escribe	Unidireccional y bloqueante
`tlm_blocking_get_if`	Lee	
`tlm_blocking_peek_if`	Lee	
`tlm_nonblocking_put_if`	Escribe	Unidireccional y no bloqueante
`tlm_nonblocking_get_if`	Lee	
`tlm_nonblocking_peek_if`	Lee	
`tlm_reset_put_if`	*Reset*	Unidireccional
`tlm_reset_get_if`		
`tlm_reset_peek_if`		

Tabla 3.2: Interfaces TLM [3], [38]

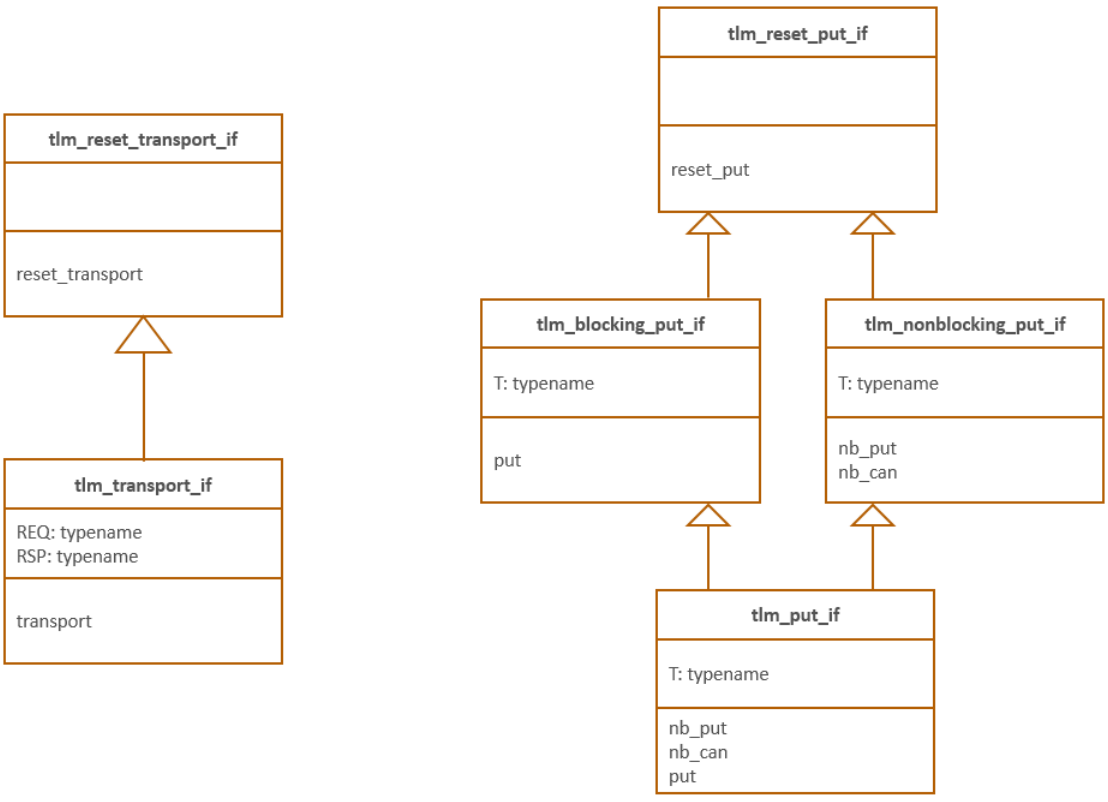


Figura 3.10: Relaciones heredadas de las interfaces TLM (1), *transport* y *put* (adaptado de [52])

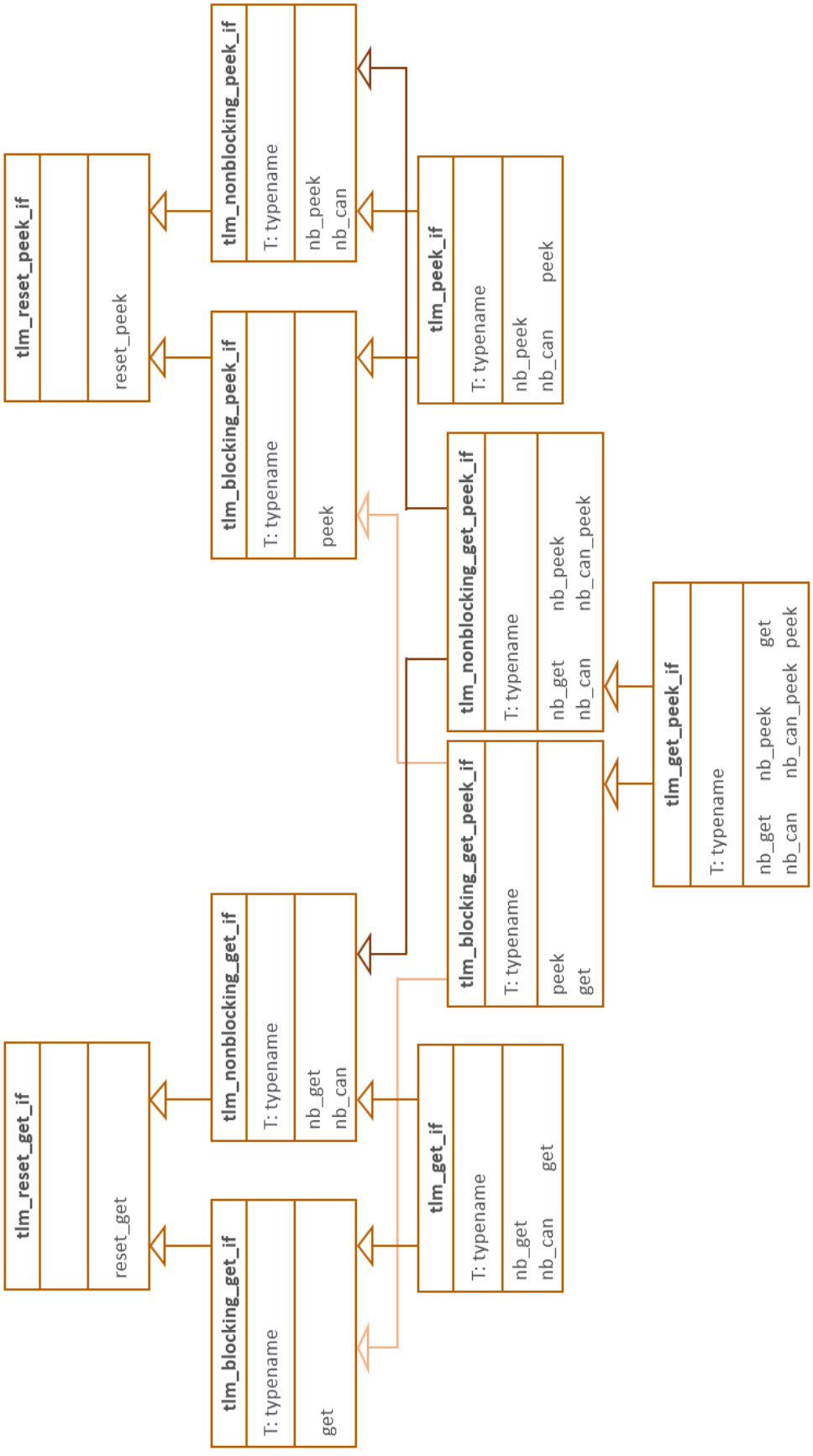


Figura 3.11: Relaciones heredadas de las interfaces TLM (2), *get* y *peek* (adaptado de [52])

También **cuenta con una interfaz de análisis** (`tlm_analysis_if`) que se usa para monitorizar las transacciones entre componentes conectados. Es unidireccional, no bloqueante y no negociante, es decir, el módulo que la recibe no tiene otra elección más que aceptar inmediatamente la transacción pasada como argumento [3].

3.4.4. Canales

TLM implementa tres canales [3], [53]:

- `tlm_fifo`. Es el canal primitivo predefinido de TLM, así como el más usado. Trata de modelar el comportamiento de una FIFO, la cual puede ser redimensionada después de que el objeto haya sido construido. Se utiliza para implementar todas las interfaces unidireccionales bloqueantes y no bloqueantes mencionadas anteriormente.
- `tlm_req_rsp_channel`. Consiste en dos FIFOs, una para la petición que será enviada al receptor y otra para la respuesta del mismo. Pueden ser de cualquier tamaño.
- `tlm_transport_channel`. Se trata de una FIFO que convierte una interfaz bidireccional en dos transacciones unidireccionales, una para la petición y otra para la respuesta. Su tamaño es 1.

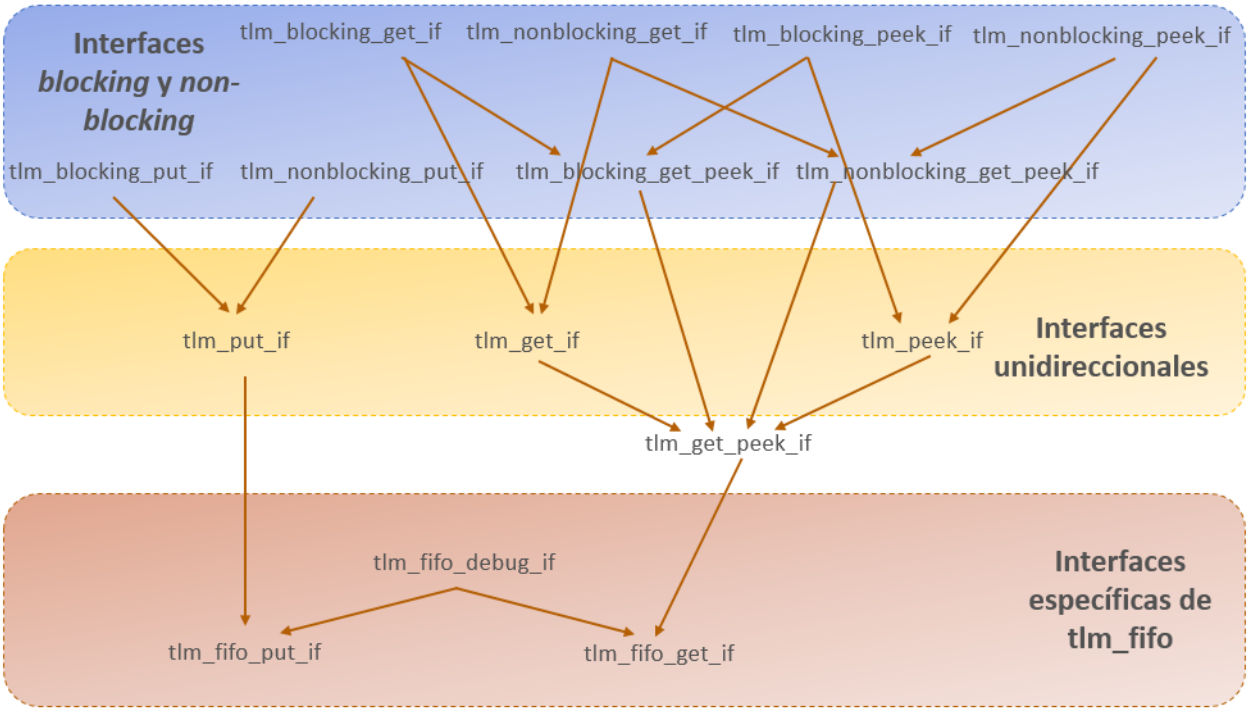


Figura 3.12: Diagrama de relaciones entre interfaces (adaptado de [54])

Con el modelado a nivel de transacciones **no es necesario implementar un protocolo de comunicación entre dos interfaces TLM**, ya que cada transacción, internamente y no visibles para el diseñador, incluye una serie de señales que permiten la sincronización entre módulos.

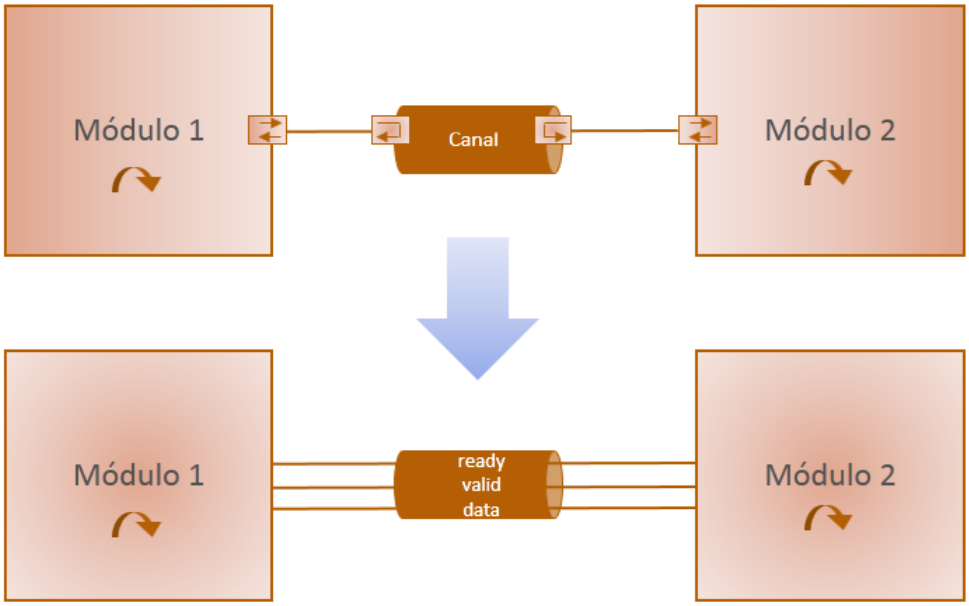


Figura 3.13: Canales: Estructura a nivel de señales (adaptado de [55])

3.4.5. Ventajas

Tras haber presentado esta nueva metodología, basada en modelar los sistemas a nivel de transacciones, se pueden indicar las siguientes ventajas [56]:

- Fácil modelado de las comunicaciones.
- Verificación funcional.
- Evaluación y corrección de errores en las primeras fases de diseño.
- Integración de los modelos HW/SW.
- Precisión en el modelado.
- Simulaciones mucho más rápidas que en RTL.

3.5. Flujo de diseño propuesto

En la Figura 3.14 se muestra el flujo de diseño que se propone seguir para poder conseguir modelos sintetizables, tanto para ASIC como para FPGA, a partir del modelado a nivel de transacciones.

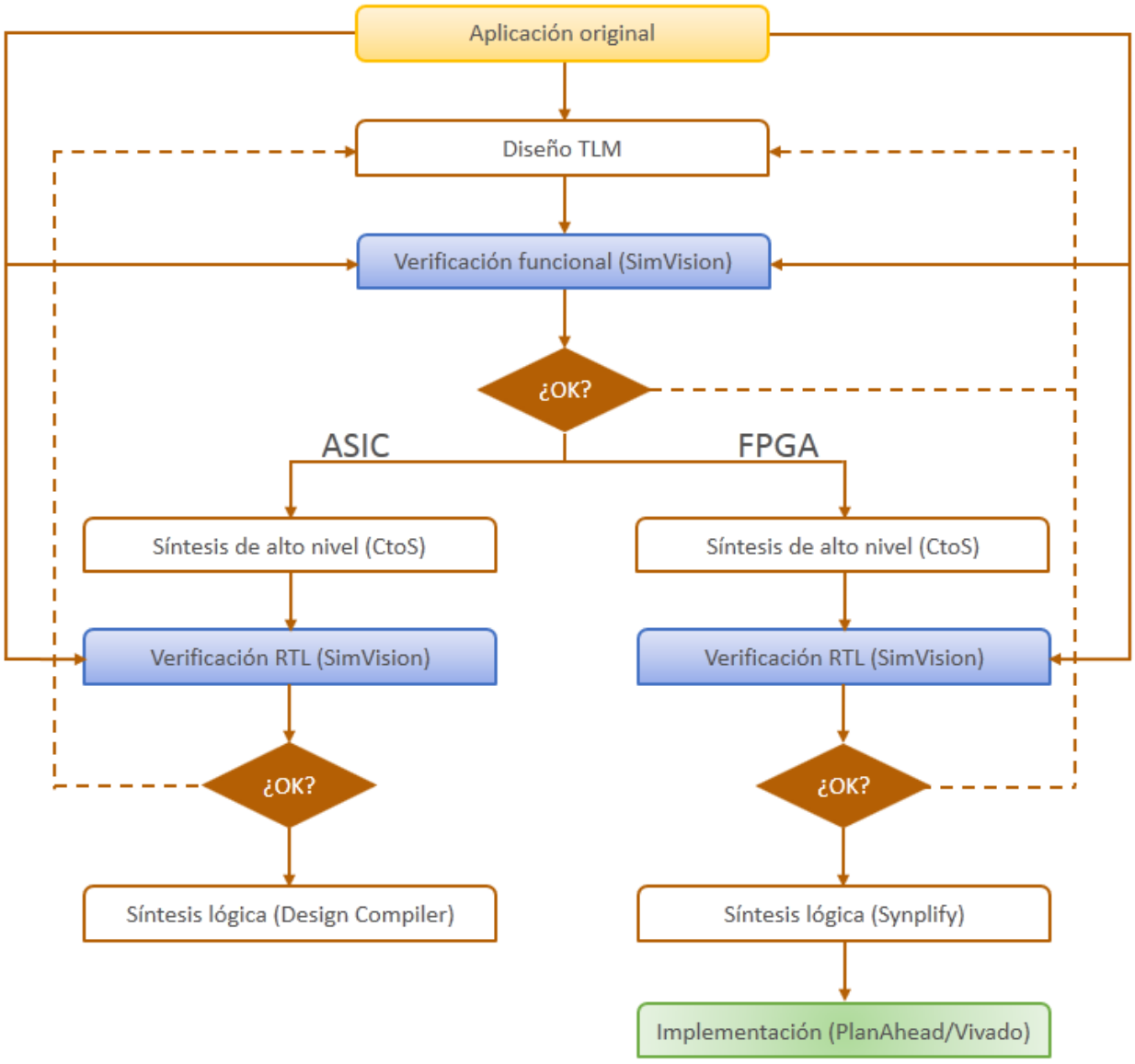


Figura 3.14: Flujo de diseño propuesto

Partiendo del código en SystemC de la aplicación de referencia se procederá a realizar el modelado a nivel de transacciones, adecuando las interfaces de la misma y obteniendo así el diseño TLM. Se verificará su correcto funcionamiento, comparando siempre los resultados con la aplicación original y usando la herramienta de Cadence, SimVision.

Cuando la descripción ESL sea funcionalmente correcta se hará la síntesis de alto nivel mediante Cadence CtoS, indicando si el diseño será para ASIC o FPGA y así obtener el respectivo modelo RTL, del cual se comprobará si cumple con la funcionalidad.

Una vez el paso anterior esté finalizado y se prosiga con el flujo se realizará la síntesis lógica, que dependiendo del modelo RTL obtenido, para ASIC o FPGA, se utilizará Synopsys Design Compiler o Synopsys Synplify Premiere, respectivamente. **En este punto se tendrán modelos sintetizables a partir de la definición de una metodología de síntesis ESL basada en TLM, y por tanto, se habrá conseguido el objetivo principal de este proyecto.**

Por último, para demostrar la viabilidad del diseño y tener una referencia de su posible implementación en una FPGA, se llevará a cabo la síntesis física con PlanAhead y Vivado de Xilinx.

3.6. Herramientas

En este apartado se enumerarán las herramientas que se han utilizado en la elaboración del proyecto, y que permiten reducir el tiempo de diseño en sistemas complejos, así como mejorar la calidad de los resultados (QoR).

3.6.1. Cadence C-to-Silicon Compiler

Cadence C-to-Silicon Compiler (CtoS) es un entorno de diseño que **permite realizar la síntesis de alto nivel** de un sistema electrónico [38]. Además, ha desarrollado una librería de IPs sintetizables para poder usarlos con CtoS, incluye FIFOs, registros, interfaces de bus, operaciones en punto fijo, etc. Estos modelos están diseñados para que sean altamente configurables y provean una buena calidad de los resultados obtenidos durante la síntesis [36].

Hasta ahora la principal barrera para la adopción generalizada del diseño basado en TLM es que la síntesis automatizada no había sido capaz de proporcionar una calidad de los resultados similares a los obtenidos a partir de los modelos RTL codificados manualmente [37]. CtoS soporta las tareas de síntesis de alto nivel. Entre sus beneficios se encuentra [39]:

- Reducción del esfuerzo en la verificación con la introducción de menos errores (*bugs*), facilidad de depurado y simulaciones más rápidas que con RTL.

- Alta productividad y reusabilidad, ya que ofrece mayor rapidez que trabajando con RTL, flexibilidad a la hora de analizar las arquitecturas y facilidad para refinar o recolocar los relojes.

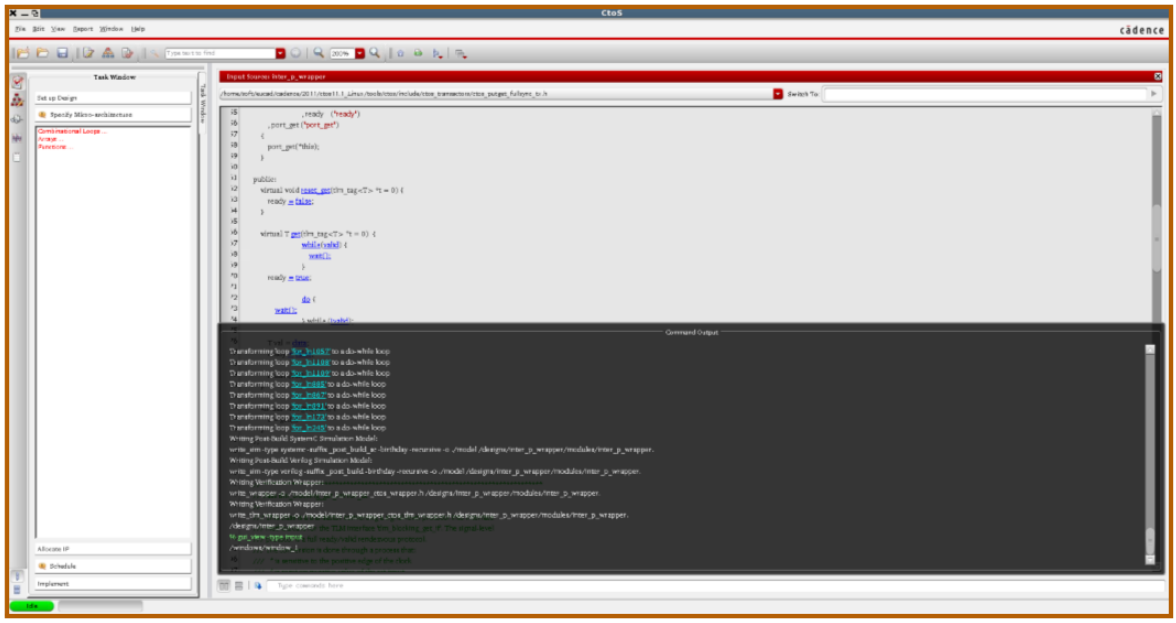


Figura 3.15: Interfaz de usuario de Cadence C-to-Silicon Compiler

3.6.1.1. Características principales

Algunas de las características generales de este entorno son [38]:

- Acepta diseños con un nivel de abstracción alto, como puede ser TLM, descritos en SystemC, C o C++.
- Separa la funcionalidad de las restricciones.
- Permite elegir entre trabajar con interfaz gráfica (GUI) o de texto mediante *scripts* basados en TCL.
- Produce modelos sintetizables a nivel RTL en Verilog.

CtoS acepta el **modelado a nivel de transacciones** y el estándar de OSCI TLM 1.0, del cual ofrece un completo soporte [39]. Por tanto, se ajusta perfectamente a una metodología orientada a la síntesis de alto nivel, permitiendo obtener modelos sintetizables, tanto para FPGA como para ASIC, a partir del modelado a nivel de transacciones.

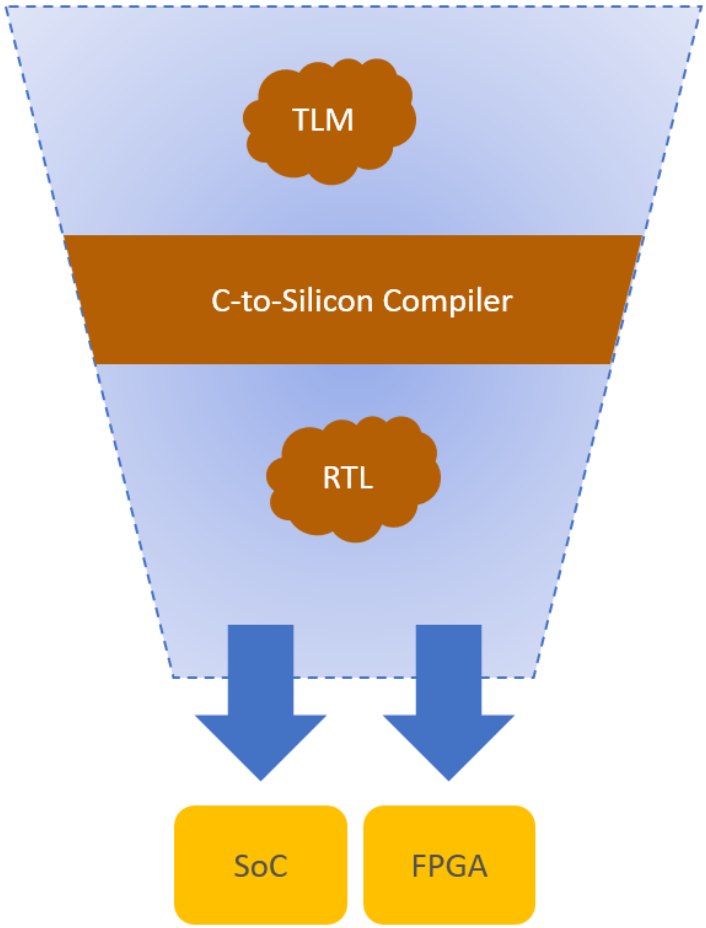


Figura 3.16: Metodología orientada a la síntesis de alto nivel (adaptado de [37])

3.6.1.2. Flujo de diseño

El flujo de diseño de CtoS describe el orden en el que se debería realizar el proceso de síntesis, así como aporta los comandos, informes y modelos necesarios en cada paso. En la Figura 3.17 se muestra el flujo de diseño soportado.

1. **Inicio.** Se crea un nuevo diseño o se abre uno ya existente.

2. **Configuración del diseño.** Se especifica si el diseño es SystemC, C/C++ o TLM. Además, se define el reloj y se incluyen los ficheros fuente. Si se tratara de un diseño TLM también habrá que definir los *transactors*, que son aquellos que generan el protocolo de comunicación una vez se ha sintetizado.

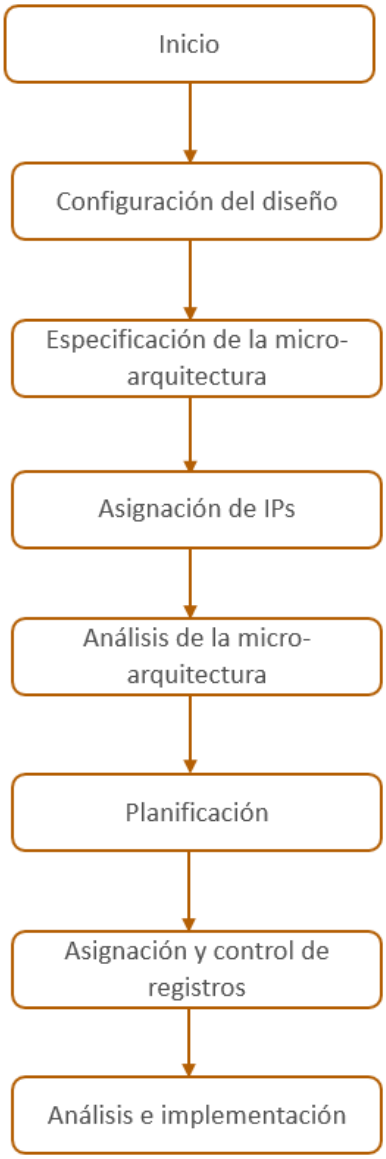


Figura 3.17: Flujo de diseño de CtoS (adaptado de [38])

3. **Especificación de la micro-arquitectura.** Se realizan las transformaciones en la arquitectura necesarias para hacer el diseño sintetizable. Esto implica resolver problemas de realimentación en funciones y bucles, entre otros aspectos.

   Algunas funciones no son sintetizables por tener caminos diferentes que requieren distinta latencia. Se soluciona haciendo las funciones “en línea” (*inline*), es decir, sustituir la llamada por el código que la describe. El efecto es un incremento en los recursos *hardware* necesarios para su implementación.

En el caso de los bucles combinacionales, deben ser eliminados porque estos no pueden ser implementados en *hardware*. Para ello, teniendo en cuenta las restricciones de área y de frecuencia, se puede realizar una de las siguientes acciones:

- Desenrollar el bucle para ser realizado enteramente en tiempo cero. Esto puede ser inviable para bucles de muchas iteraciones, pues sería la ruta crítica, con una latencia muy alta, ya que consume muchos recursos. En cambio, permite obtener mayor velocidad en el proceso.
- Romper el bucle para que cada iteración se realice en un ciclo de reloj, reutilizando recursos.
- Realizar una segmentación del bucle, lo cual significa dividir la ejecución del bucle en un número determinado de etapas, de tal forma que cada iteración se encuentra en una etapa, paralelizando así varias iteraciones.

4. **Asignación de IPs.** Se asignan las variables de tipo vector a memorias.

5. **Análisis de la micro-arquitectura.** Dado que el diseño ya está completamente transformado y CtoS puede realizar las últimas optimizaciones, se puede hacer un análisis temporal, de área o de potencia.

6. **Planificación.** El planificador del CtoS asignará recursos a las operaciones definidas en el diseño. En caso de que no pueda realizar esta operación, el diseñador puede ayudarle controlando:

   - La dependencia con vectores de datos. Esto hará que el planificador conozca las dependencias y pueda añadir estados para resolverlas.
   - Los estados. En caso de que el planificador no pueda resolver los conflictos secuenciales, el diseñador puede añadir manualmente estados para que CtoS pueda encontrar una solución.
   - Los recursos. El diseñador podrá realizar una asignación inicial de recursos sobre los que realizar la asignación de las operaciones del diseño.

7. **Asignación y control de registros.** Se controlan los registros y se le asignan los valores requeridos. Se puede incluir una lógica de *reset* para aquellos registros que no lo tengan, registrar o no todas las salidas de los recursos de lógica combinacional asociados a operaciones del diseño, o minimizar los registros (asociados a los diseños basados en ASIC, donde el coste en área de los registros es alto) o el número de multiplexores (asociado a los diseños basados en FPGAs).

8. **Análisis e implementación.** Se realizan informes de distinto tipo, como puede ser el consumo de recursos o de latencia en ciclos de reloj de cada bloque. También se podrán generar los ficheros de salida (la descripción RTL), un *script* que realice las mismas acciones que se han seguido a través de la interfaz gráfica, o un fichero contenedor en SystemC de la descripción RTL.

3.6.1.3. *Transactors*

Una de las ventajas de CtoS es que soporta TLM y provee un versión sintetizable de la librería TLM 1.0, así como la librería que contiene los *transactors* básicos.

Un *transactor* (TX en la Figura 3.18) permite refinar la interfaz TLM en el protocolo de comunicación a nivel de señales. El *transactor* espejo (MTX en la Figura 3.18) utiliza el protocolo *handshake* con el *transactor*.

Para poder realizar la verificación del sistema, CtoS automáticamente genera el *wrapper* necesario. Este es un módulo que *instancia* el *transactor* espejo y el *wrapper* TLM generado durante la síntesis, el cual define el refinamiento de las interfaces externas en interfaces a nivel de señales.

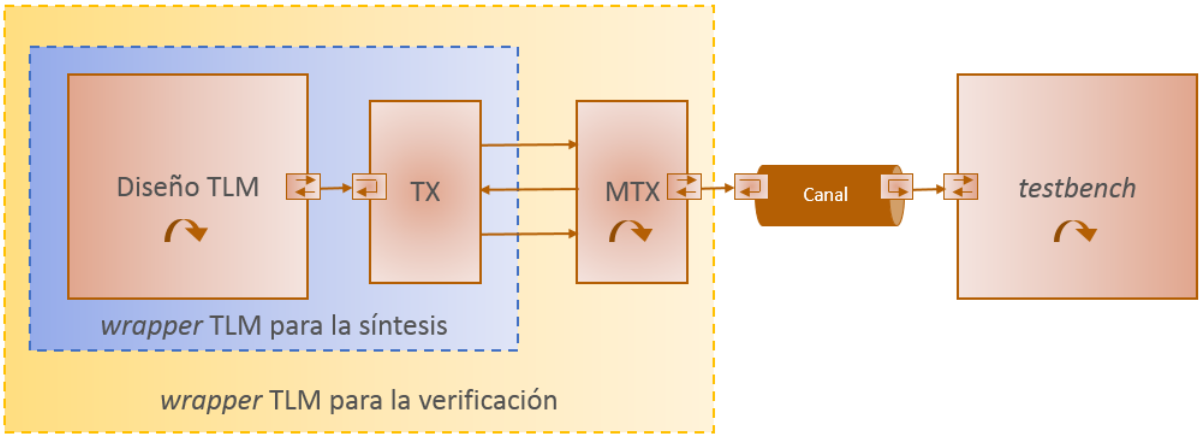


Figura 3.18: *Transactors* (adaptado de [38])

3.6.2. Simulation Analysis Environment SimVision

Con las nuevas tendencias de diseño, cada vez a más alto nivel, las herramientas de simulación deben adecuarse a los nuevos requerimientos. Para ello se hace uso de Cadence Incisive Enterprise Simulator, un entorno de verificación que soporta múltiples lenguajes (SystemVerilog, Verilog, VHDL,

SystemC, C, C++, etc.) y diferentes herramientas para simulación y depuración del diseño en modo gráfico, como es SimVision.

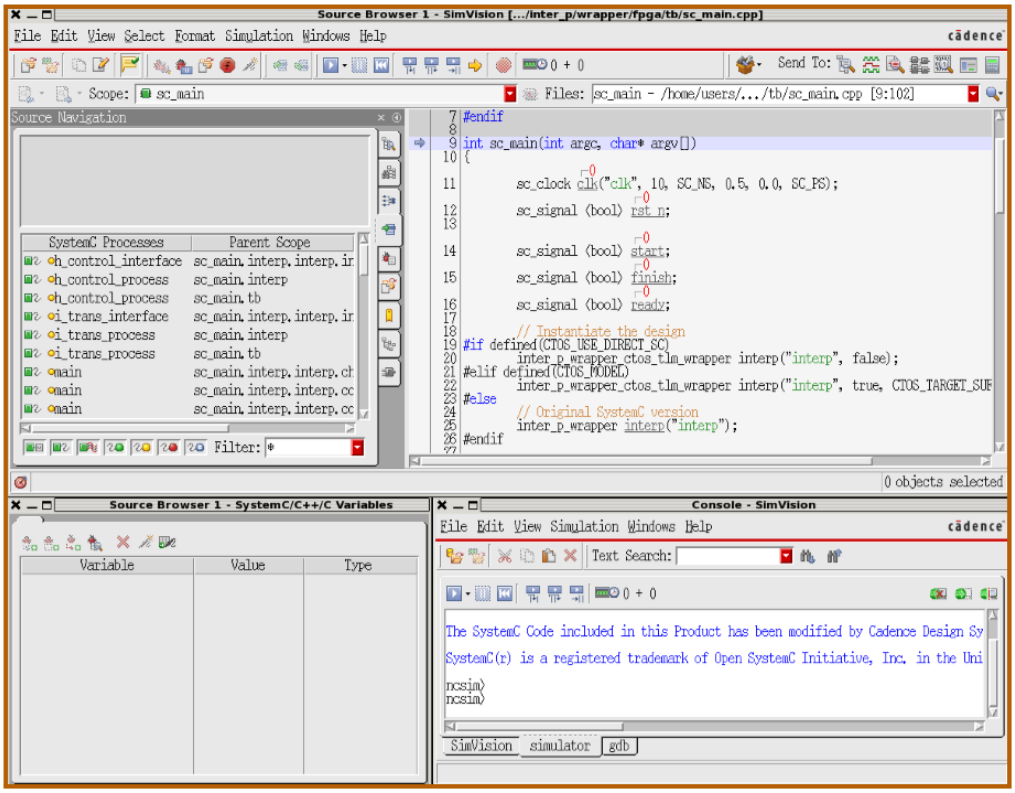


Figura 3.19: Interfaz de usuario de Simulation Analysis Environment SimVision

SimVision se puede utilizar para depurar diseños TLM o RTL, soporta la representación de señales digitales, analógicas o mixtas, así como los lenguajes citados Verilog, System Verilog, VHDL o SystemC [40].

3.6.2.1. Características principales

Las características generales de SimVision son [41], [43]:

- Soporte multi-lenguaje y multi-señal, es decir, acepta todos los estándares del IEEE y Accellera.
- Verificación a todos los niveles de abstracción, desde TLM hasta código HDL.
- Simulación cruzada de sistemas escritos por combinación de los lenguajes soportados.
- Ejecución en línea de comandos (TCL) o en forma de onda.

3.6.3. Synplify Premier with Design Planner

Para la realización de la síntesis lógica, se ha optado por Synopsys Synplify Premier. Se trata de una herramienta enfocada hacia la síntesis lógica para FPGAs. Incorpora técnicas de optimización que extraen el comportamiento del sistema para mejorar el rendimiento del diseño a nivel global.

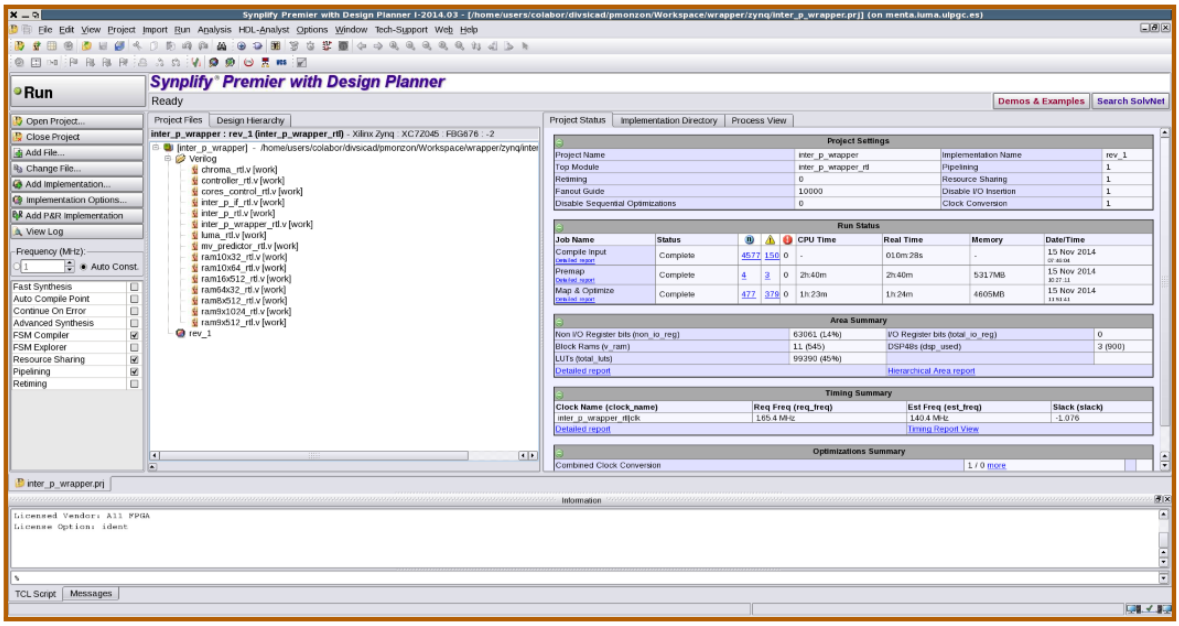


Figura 3.20: Interfaz de usuario de Synplify Premier with Design Planner

3.6.3.1. Características principales

Las principales características de Synopsys Synplify Premier son [44]:

- Gestión de la complejidad mediante la partición del diseño global en subdiseños, con el fin de poder trabajar en paralelo. Permite la realización de proyectos divididos en subproyectos independientes, es decir, un diseño jerárquico, ya sea mediante técnicas de síntesis *top-down* y *bottom-up*.
- Un modo de síntesis rápido (x3) que permite realizar un análisis inicial del sistema.
- Capacidad de multi-proceso usando particiones. Pueden definirse “puntos de compilación” (*compile points)*, particiones del diseño, con el fin de que pueda lanzarse en paralelo la síntesis de cada una de ellas, con el consecuente ahorro de tiempo en dicha etapa.
- Haciendo uso de los puntos de compilación se pueden heredar soluciones de cada partición sintetizada 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.

- El diseño puede representarse en dos vistas, una antes de realizar la síntesis lógica (vista RTL), en la que se representa a modo de bloques la jerarquía de módulos y funcionalidad de la descripción RTL realizada en HDL; y la segunda es la tecnológica, en la que cada bloque se representa como el conjunto de recursos sobre el que ha sido mapeado, ya sean registros, memorias de bloque (BRAM) o LUTs.

3.6.4. Design Vision

Synopsys también cuenta con herramientas para la realización de la síntesis lógica centrada en el diseño sobre de ASICs, dentro del entorno Design Compiler, bajo la interfaz gráfica Design Vision.

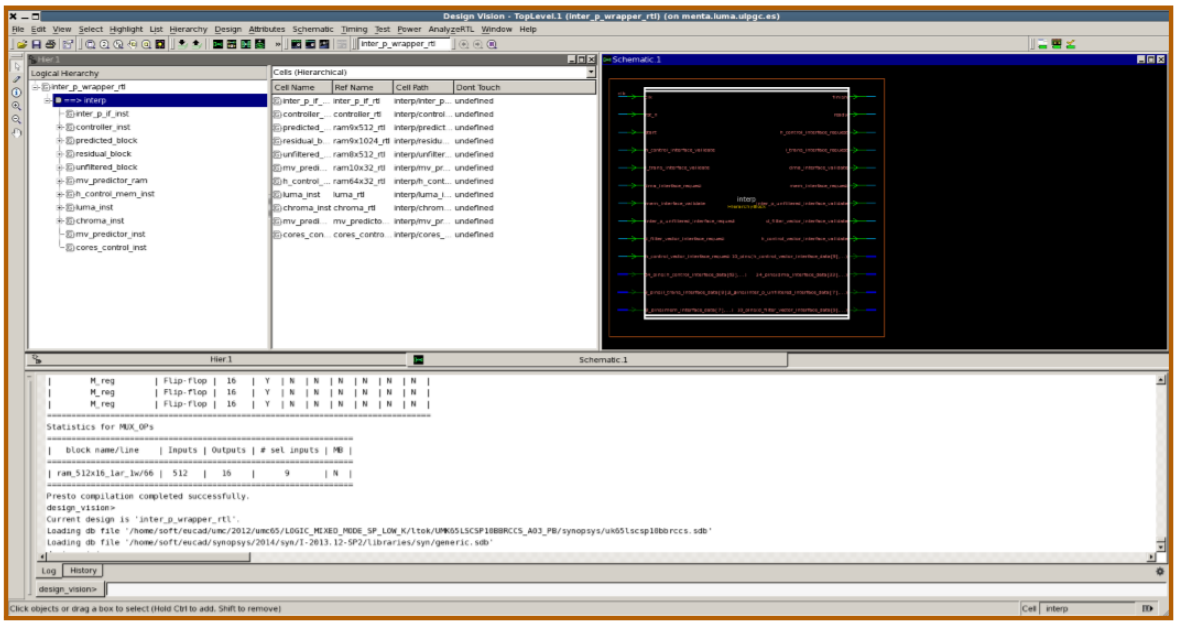


Figura 3.21: Interfaz de usuario de Design Vision

3.6.4.1. Características principales

Sus principales características se indican a continuación [45]:

- Incluye un conjunto de optimizaciones avanzadas ya sea de tipo temporal, de área o de potencia.
- Exploración de alternativas de diseño que posean restricciones mixtas de tiempo, área y potencia, bajo diferentes condiciones de carga, temperatura y voltaje.

- Precisión en la predicción de caminos críticos, pudiendo modificarlos mediante diferentes técnicas tales como *retiming*.
- Soporte de especificaciones combinacionales, secuenciales y jerárquicas de cualquier tipo, así como de estilos de diseño.
- Mapeado para diferentes tecnologías.
- Síntesis y optimización de máquinas de estado finito (FSM), incluyendo minimización y codificación automática de estados.
- Inserción y optimización de *pads* de entrada/salida.
- Optimizaciones de frontera en diseños jerárquicos.
- Diseño automático jerárquico (*top-down* o *bottom-up*).
- Técnicas de diseño incrementales.

3.6.5. PlanAhead

Xilinx PlanAhead es un entorno avanzado que controla las etapas de síntesis e implementación del flujo de diseño de la FPGA de Xilinx. Se pueden introducir diseños en diferentes formatos, como pueden ser esquemáticos, grafos de estado o descripciones *hardware*, y realizar diferentes análisis tras su compilación [46].

Esta herramienta permite al diseñador elegir entre diferentes estrategias predefinidas (optimización global, temporal, optimización por bloques, etc.), o bien crear una propia. La selección de la estrategia apropiada tendrá como resultado una mejora en la optimización realizada y un impacto directo en los tiempos de implementación.

3.6.5.1. Características principales

Algunas de sus características son [46]:

- Mejora del comportamiento del circuito.
- Definición y análisis del diseño RTL.
- Uso de diferentes opciones de implementación.
- Refinamiento de las restricciones temporales.

- Aplicación de restricciones físicas con técnicas de colocación.
- Estimación de los recursos utilizados.
- Modelo jerárquico de datos que permite incrementar la capacidad de diseño.

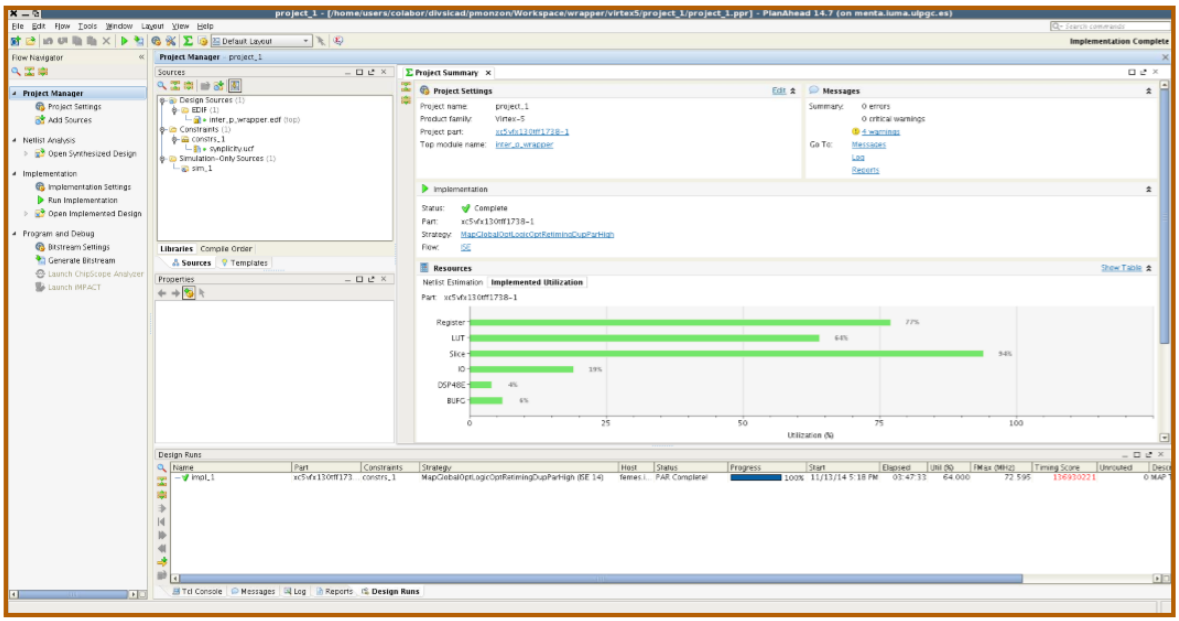


Figura 3.22: Interfaz de usuario de PlanAhead

3.6.6. Vivado Design Suite

Para FPGAs de la serie 7, Zynq-7000 y UltraScale, Xilinx cuenta con un entorno que sustituye a Xilinx ISE Design Suite, Vivado Design Suite [16].

En este proyecto se hace uso del *Place & Route* de Vivado bajo la interfaz de usuario de Synplify Premier.

3.6.6.1. Características principales

Algunas de las características que presenta este entorno son:

- Mayor velocidad de implementación (x4).
- Mejor densidad de diseño.
- Hasta tres velocidades de rendimiento, usando menos energía.

- Depuración de *hardware*.
- Mayor velocidad de verificación (x100) con C, C++ o SystemC.

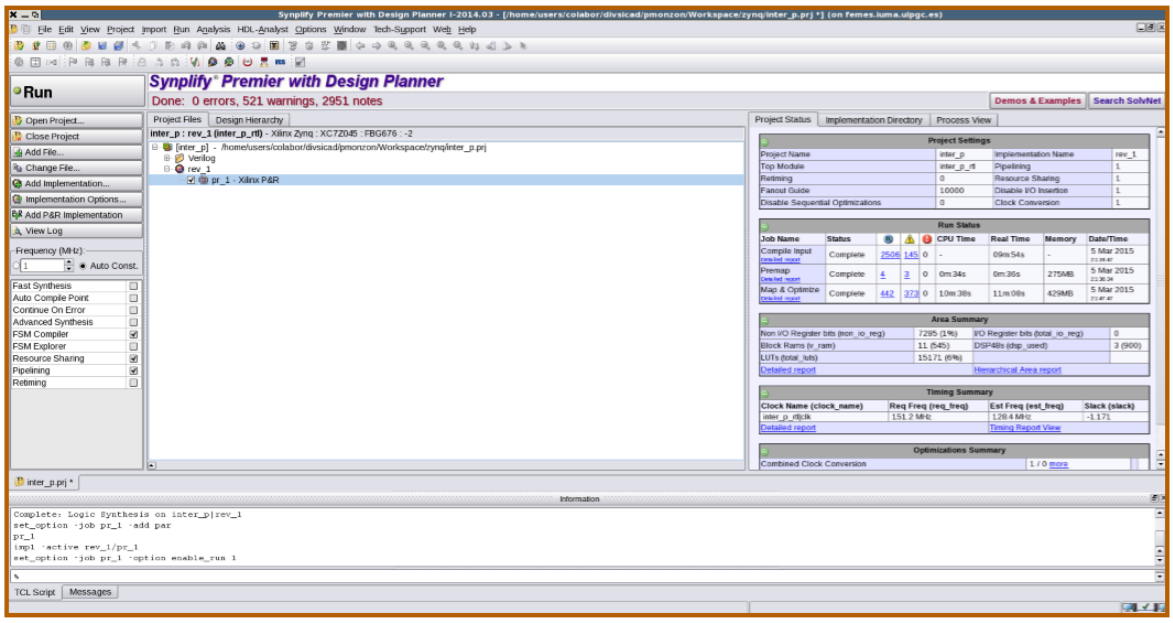


Figura 3.23: *Place & Route* de Vivado bajo la interfaz de usuario de Synplify Premier

3.7. Conclusiones

El modelado a nivel de transacciones se presenta como una metodología que trata de facilitar y agilizar el proceso de diseño. Se basa en separar las comunicaciones del sistema electrónico de su funcionalidad, dando por hecho que esta está completamente definida y “olvidándola”, para centrarse únicamente en la interfaz. Sin embargo, no es necesario un protocolo de comunicación, ya que a nivel externo solo hace transacciones y es a nivel interno donde se encuentra dicho protocolo, no visible para el diseñador.

Se trata de un nivel de abstracción alto, por encima de RTL, donde se puede evitar los detalles innecesarios. No obstante, no se debe olvidar la plataforma donde se implementará el diseño, pues de esta manera se puede hacer un uso eficiente de la misma, además de suponer un aspecto clave para la optimización del sistema final.

Por otro lado, cabe destacar que el diseñador debe conocer qué elementos como el lenguaje, las herramientas, etc. va a utilizar. En este caso, SystemC, un lenguaje de descripción a nivel de sistemas

muy potente, que se ajusta perfectamente a las necesidades de TLM; y un conjunto de herramientas de las que se dispone, permite realizar el diseño del sistema a nivel de transacciones, siguiendo un flujo de diseño adecuado.

Capítulo 4

Modelado TLM con *wrapper*

4.1. Introducción

El modelado de un sistema electrónico transforma las especificaciones del sistema, utilizando un lenguaje existente, para realizar su posterior verificación del correcto funcionamiento mediante simulaciones. El siguiente paso sería la transformación de dicho modelo hacia una implementación, de manera que se irá optimizando el diseño a medida que se va desarrollando durante el proceso.

Como se comentó en el Capítulo 2, se parte del diseño en alto nivel del código de referencia del decodificador H.264/AVC desarrollado por el Instituto Universitario de Microelectrónica Aplicada, descrito en SystemC, para luego modelarlo con TLM.

Por tanto, primero se hará una descripción de la situación inicial de esta etapa, se verá desde dónde se parte, para después presentar la versión modelada a nivel de transacciones del módulo correspondiente con *wrapper*.

4.2. Diseño de referencia

El objeto de estudio de este proyecto es el modelado TLM y la síntesis del bloque *inter_p* del decodificador de vídeo. No obstante, es necesario otro módulo (*testbench*) para poder realizar la verificación del bloque en cuestión. Ambos están interconectados y forman una estructura jerárquica junto al `sc_main`.

En el archivo *sc_main.cpp* se *instancian* los módulos que se quieren conectar. Además, se declaran las señales para poder realizar dicha conexión y se establece el reloj, siendo en este caso un reloj

de nombre “clk”, con un periodo de 10 ns, un ciclo de trabajo del 50 % y sin desfase.

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

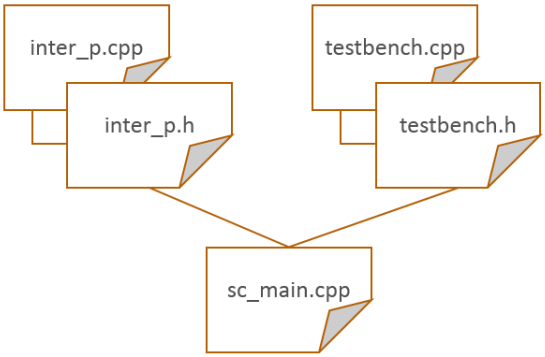


Figura 4.1: Estructura de ficheros del modelo inicial

4.2.1. Módulo *inter_p*

El bloque *inter_p* crea macrobloques de predicción a partir de los vectores de movimiento y del fotograma de referencia.

4.2.1.1. Definición de las interfaces y los puertos

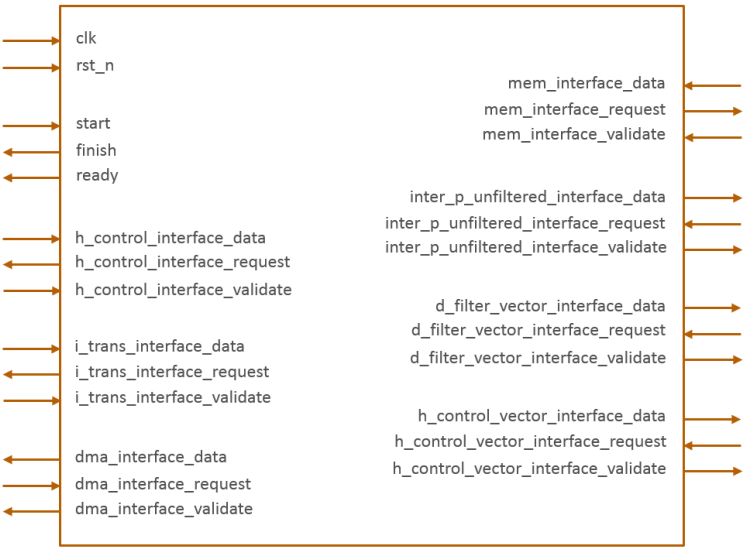


Figura 4.2: Puertos de E/S del bloque *inter_p*

La Tabla 4.1 recoge la descripción de los puertos de entrada y salida del bloque.

Puerto	Ancho	Descripción
`clk`	1	Reloj
`rst_n`	1	*Reset*, activa a nivel bajo
`start`	1	Comienzo de los procesos del *inter_p*
`finish`	1	Finalización de los procesos del *inter_p*
`ready`	1	Señalización de que los datos están preparados
`h_control_interface_data`	64	Protocolo de comunicación con el módulo CAVLD
`h_control_interface_request`	1	
`h_control_interface_validate`	1	
`i_trans_interface_data`	9	Protocolo de comunicación con el módulo *iqit*
`i_trans_interface_request`	1	
`i_trans_interface_validate`	1	
`dma_interface_data`	34	Protocolo de comunicación para el acceso a la memoria
`dma_interface_request`	1	
`dma_interface_validate`	1	
`mem_interface_data`	8	Protocolo de comunicación para la transferencia con la memoria
`mem_interface_request`	1	
`mem_interface_validate`	1	
`inter_p_unfiltered_interface_data`	8	Protocolo de comunicación con el módulo *d_filter*
`inter_p_unfiltered_interface_request`	1	
`inter_p_unfiltered_interface_validate`	1	
`d_filter_vector_interface_data`	10	Vectores de movimiento
`d_filter_vector_interface_request`	1	
`d_filter_vector_interface_validate`	1	
`h_control_vector_interface_data`	10	Vectores de movimiento
`h_control_vector_interface_request`	1	
`h_control_vector_interface_validate`	1	

Tabla 4.1: Descripción de los puertos E/S

En el archivo *inter_p.h* se encuentran definidas tanto las interfaces como los puertos del módulo que se conectarán con el resto de módulos, aunque en este caso, solo lo harán con el *testbench*.

```
SC_MODULE (inter_p) {
    // INTERFACES AND PORTS
    sc_in<bool>clk;
    sc_in<bool>rst_n;

    sc_in<bool>start;
    sc_out<bool>finish;
    sc_out<bool>ready;

    // H_CONTROL interface
    sc_in< sc_uint<64> >h_control_interface_data;
    sc_out<bool>h_control_interface_request;
    sc_in<bool>h_control_interface_validate;

    // I_TRANS interface
    sc_in< sc_int<9> >i_trans_interface_data;
    sc_out<bool>i_trans_interface_request;
    sc_in<bool>i_trans_interface_validate;

    // Sample memory (reference frame) access request
    sc_out< sc_uint<34> >dma_interface_data;
    sc_in<bool>dma_interface_request;
    sc_out<bool>dma_interface_validate;

    // Sample memory (reference frame) transfer
    sc_in< sc_uint<8> >mem_interface_data;
    sc_out<bool>mem_interface_request;
    sc_in<bool>mem_interface_validate;
```

```
        // Output data interface - unfiltered block
        sc_out< sc_uint<8> >inter_p_unfiltered_interface_data;
        sc_in<bool>inter_p_unfiltered_interface_request;
        sc_out<bool>inter_p_unfiltered_interface_validate;

        // Output data interface - motion vectors
        sc_out< sc_int<10> >d_filter_vector_interface_data;
        sc_in<bool>d_filter_vector_interface_request;
        sc_out<bool>d_filter_vector_interface_validate;

        sc_out< sc_int<10> >h_control_vector_interface_data;
        sc_in<bool>h_control_vector_interface_request;
        sc_out<bool>h_control_vector_interface_validate;
        ...
};
```

4.2.1.2. Definición de las funciones y los procesos

El bloque *inter_p* está compuesto por seis bloques funcionales y siete memorias que realizan las operaciones necesarias para poder llevar a cabo su función correctamente.

En este proyecto los módulos se tratan como “cajas grises” (*grey-box model*), es decir, la funcionalidad está completamente definida, y el objetivo es introducir comunicaciones TLM en la interfaz. Por ello solo será de interés el módulo *inter_p_if*, que incluye el protocolo de comunicación.

En el archivo *inter_p_if.h* están definidas las funciones y los procesos que hacen posible el funcionamiento de la interfaz de dicho módulo.

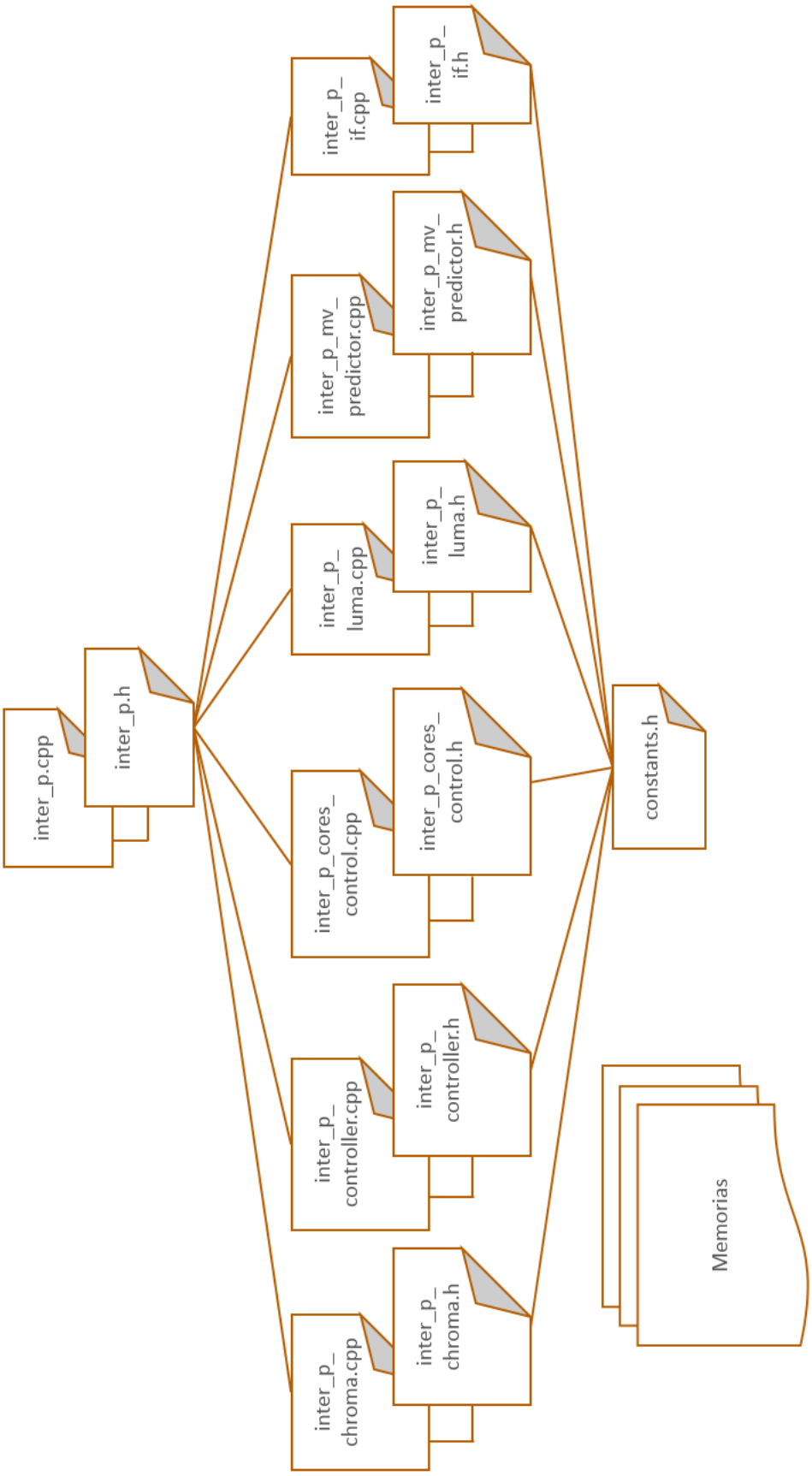


Figura 4.3: Módulo *inter_p*: Ficheros que lo componen

```
SC_MODULE (inter_p_if){

    ...

    // FUNCTIONS AND PROCESS
    void main();
    void read_in();
    void write_out();

    void h_control_interface();
    void i_trans_interface();
    void unfiltered_interface();
    void vector_interface();
    void transfer_ref_frame();

    ...

    SC_CTOR(inter_p_if){

        ...

        // Process registration
        SC_THREAD(main);
        sensitive «clk.pos();

        SC_THREAD(h_control_interface);
        sensitive «clk.pos();

        SC_THREAD(i_trans_interface);
        sensitive «clk.pos();
```

```
                SC_THREAD(unfiltered_interface);
                sensitive «clk.pos();

                SC_THREAD(vector_interface);
                sensitive «clk.pos();

                SC_THREAD(transfer_ref_frame);
                sensitive «clk.pos();
        }
};
```

La Tabla 4.2 recoge la descripción de las funciones.

Función	Descripción
`void main()`	Coordina todo el funcionamiento del módulo. Inicializa las señales de control `ready` y `finish`, e invoca a las funciones de lectura y escritura cuando corresponda
`void read_in()`	Realiza la lectura de datos y genera la señal `ready`
`void write_out()`	Realiza la escritura de los datos y genera la señal `finish`
`void h_control_interface()`	Genera el protocolo de comunicación con el módulo CAVLD
`void i_trans_interface()`	Genera el protocolo de comunicación con el módulo *iqit*
`void unfiltered_interface()`	Genera el protocolo de comunicación con el módulo *d_filter*
`void vector_interface()`	Genera el protocolo de comunicación para los vectores de movimiento
`void transfer_ref_frame()`	Genera el protocolo de comunicación con la memoria

Tabla 4.2: Descripción de las funciones del bloque *inter_p*

4.2.1.3. Protocolo de comunicación

Como se comentó en el Capítulo 2, el protocolo de comunicación que implementa el módulo utiliza un esquema de *handshake*, donde un primer módulo envía una petición a un segundo, y este prepara los datos y emite la validación de los mismos. En definitiva, se utilizan tres señales, siendo generalmente: *request*, *data* y *validate* (petición, datos y validación, respectivamente).

Por cada conexión con cada módulo, el bloque *inter_p* aplica la idea anterior, es decir, para la comunicación con un módulo concreto usará una señal *data*, la señal *request* y la señal *validate*. En la Figura 4.4 se puede observar la comunicación del módulo bajo estudio con el CAVLD. En este caso, el primero es el que inicia la comunicación haciendo una petición al segundo (punto 1). Cuando se recibe la petición, se activa la señal de validación (punto 2) a la vez que inicia el envío de los datos (punto 3) y el *inter_p* desactiva la señal de *request* (punto 4). Cuando la transferencia de datos concluye el CAVLD desactiva la señal *validate* (punto 5).

Figura 4.4: Comunicación entre el bloque *inter_p* y CAVLD

Lo que envía el *testbench* el bloque *inter_p* espera recibirlo, y viceversa. Los datos que se transmiten son:

- 11 palabras de 64 bits (`h_control_interface_data`).
- 256 palabras de 9 bits (`i_trans_interface_data`).
- 1.024 palabras de 8 bits (`mem_interface_data`).

- 1 palabra de 34 bits (`dma_interface_data`).
- 258 palabras de 8 bits (`inter_p_unfiltered_interface_data`).
- 34 palabras de 10 bits (`d_filter_vector_interface_data`).
- 18 palabras de 10 bits (`h_control_vector_interface_data`).

4.2.2. Módulo *testbench*

El *testbench* es el encargado de realizar las operaciones que harían el resto de módulos con los que se comunica el bloque *inter_p* para que este funcione correctamente. Genera las entradas y permite observar las salidas.

En el archivo *testbench.h* se encuentran las definiciones, mientras que en el archivo *testbench.cpp* la funcionalidad.

4.2.2.1. Definición de las interfaces y los puertos

La descripción de los puertos es la misma que la de la Tabla 4.1, aunque la dirección de los puertos cambia según se muestra en la Figura 4.5.

Figura 4.5: Puertos E/S del *testbench*

El modelo SystemC para la definición de las interfaces y los puertos es el siguiente:

```
SC_MODULE (testbench) {

    sc_in<bool>clk;
    sc_out<bool>rst_n;

    sc_out<bool>start;
    sc_in<bool>finish;
    sc_in<bool>ready;

    // H_CONTROL interface
    sc_out< sc_uint<64> >h_control_interface_data;
    sc_in<bool>h_control_interface_request;
    sc_out<bool>h_control_interface_validate;

    // I_TRANS interface
    sc_out< sc_int<9> >i_trans_interface_data;
    sc_in<bool>i_trans_interface_request;
    sc_out<bool>i_trans_interface_validate;

    // Sample memory (reference frame) DMA request
    sc_in< sc_uint<34> >dma_interface_data;
    sc_out<bool>dma_interface_request;
    sc_in<bool>dma_interface_validate;

    // Sample memory (reference frame) transfer
    sc_out< sc_uint<8> >mem_interface_data;
    sc_in<bool>mem_interface_request;
    sc_out<bool>mem_interface_validate;

    // Output data interface - unfiltered block
```

```
        sc_in< sc_uint<8> >inter_p_unfiltered_interface_data;

        sc_out<bool>inter_p_unfiltered_interface_request;
        sc_in<bool>inter_p_unfiltered_interface_validate;

        // Output data interface - motion vectors
        sc_in< sc_int<10> >inter_p_vector_interface_data;
        sc_out<bool>inter_p_vector_interface_request;
        sc_in<bool>inter_p_vector_interface_validate;

        sc_in< sc_int<10> >inter_p_h_control_vector_interface_data;
        sc_out<bool>inter_p_h_control_vector_interface_request;
        sc_in<bool>inter_p_h_control_vector_interface_validate;
        ...
};
```

4.2.2.2. Definición de las funciones y los procesos

La definición de los procesos y funciones asociadas se muestra a continuación:

```
SC_MODULE (testbench) {

    ...

    // Main testbench process
    void thread_process();

    // Data store process
    void store_data();

    // Memory managment process
```

```
                    void memory_process();

                    ...

                    SC_CTOR(testbench){

                        SC_THREAD(thread_process);
                        sensitive_neg «clk;

                        SC_THREAD(memory_process);
                        sensitive_neg «clk;

                        SC_THREAD(store_data);
                        sensitive_neg «clk;
                    }
                };
```

En la siguiente tabla se encuentra la descripción de las funciones.

Función	Descripción
`void thread_process()`	Hilo principal de ejecución. Genera las señales *rst_n* y *start*. Realiza las operaciones relacionadas con los módulos CAVLD e *iqit*
`void memory_process()`	Realiza las operaciones relacionadas con la memoria
`void store_data()`	Realiza las operaciones relacionadas con el módulo *d_filter* y los vectores de movimiento

Tabla 4.3: Descripción de las funciones del *testbench*

4.3. Modelado a nivel de transacciones

Para generar un *wrapper* que adapte la interfaz TLM a la interfaz del bloque a estudiar se deberá desarrollar el protocolo de comunicación entre el *wrapper* y el propio *inter_p*.

Idealmente tras la implementación TLM cada módulo conectado al bloque debería tener la interfaz modelada a nivel de transacciones por lo que, en este caso, **también se modelará la interfaz del *testbench***.

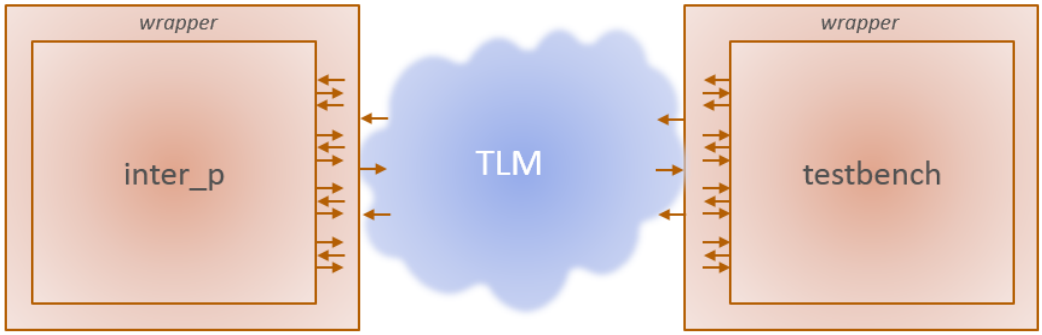


Figura 4.6: Implementación del *wrapper* TLM en el sistema

Ahora la estructura jerárquica transformada se representa en la Figura 4.7.

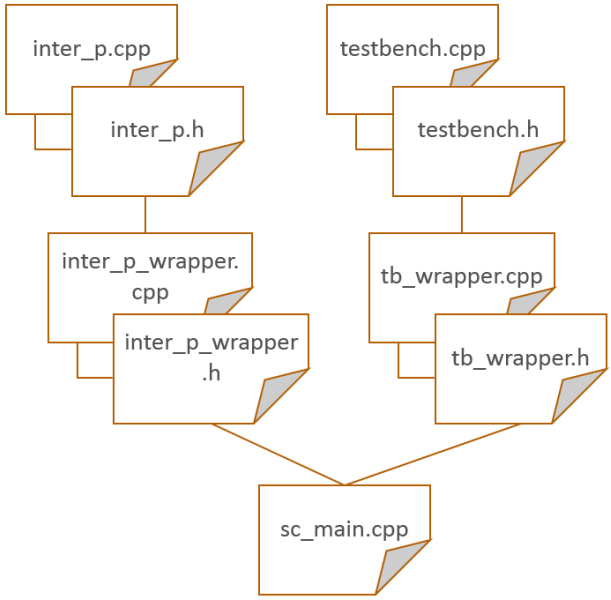


Figura 4.7: Diagrama de ficheros TLM

En el archivo *sc_main.cpp* nuevamente se *instancian* los módulos que se quieren conectar, se declaran las señales para poder realizar dicha conexión y se establece el reloj, el cual se mantiene.

La mayor diferencia radica en la declaración de las señales, ya que el conjunto de tres señales que conforman el protocolo de comunicación se sustituyen por un canal TLM del mismo tipo de datos que los datos que va a transportar. A continuación se presenta la transformación del código para el uso de TLM.

```
sc_clock clk (“clk”, 10, SC_NS, 0.5, 0.0, SC_PS);
sc_signal<bool>rst_n;

sc_signal<bool>start;
sc_signal<bool>finish;
sc_signal<bool>ready;

sc_signal< sc_uint<64> >h_control_interface_data;
sc_signal<bool>h_control_interface_request;
sc_signal<bool>h_control_interface_validate;

sc_signal< sc_int<9> >i_trans_interface_data;
sc_signal<bool>i_trans_interface_request;
sc_signal<bool>i_trans_interface_validate;

sc_signal< sc_uint<34> >dma_interface_data;
sc_signal<bool>dma_interface_request;
sc_signal<bool>dma_interface_validate;

sc_signal< sc_uint<8> >mem_interface_data;
sc_signal<bool>mem_interface_request;
sc_signal<bool>mem_interface_validate;
```

```
sc_signal< sc_uint<8> >inter_p_unfiltered_interface_data;
sc_signal<bool>inter_p_unfiltered_interface_request;
sc_signal<bool>inter_p_unfiltered_interface_validate;

sc_signal< sc_int<10> >d_filter_vector_interface_data;
sc_signal<bool>d_filter_vector_interface_request;
sc_signal<bool>d_filter_vector_interface_validate;

sc_signal< sc_int<10> >h_control_vector_interface_data;
sc_signal<bool>h_control_vector_interface_request;
sc_signal<bool>h_control_vector_interface_validate;
```

```
sc_clock clk (“clk”, 10, SC_NS, 0.5, 0.0, SC_PS);
sc_signal<bool>rst_n;

sc_signal<bool>start;
sc_signal<bool>finish;
sc_signal<bool>ready;
...

// Instantiate FIFOs
tlm_fifo< sc_uint<64> >h_control_fifo(“h_control_fifo”);
tlm_fifo< sc_int<9> >i_trans_fifo(“i_trans_fifo”);
tlm_fifo< sc_uint<8> >unfiltered_fifo(“unfiltered_fifo”);
tlm_fifo< sc_int<10> >h_control_vector_fifo(“h_control_vector_fifo”);
tlm_fifo< sc_int<10> >d_filter_vector_fifo(“d_filter_vector_fifo”);
```

```
tlm_fifo< sc_uint<34> >dma_fifo("dma_fifo");
tlm_fifo< sc_uint<8> >mem_fifo("mem_fifo");
```

Como se vio en el Capítulo 3, TLM implementa tres canales, de los cuales `tlm_fifo` es el que mejor se adapta a la situación de partida, pues el módulo está compuesto por interfaces unidireccionales, descartando así `tlm_transport_channel`. Además, es necesario una única FIFO, prescindiendo de `tlm_req_rsp_channel`.

4.3.1. *Wrapper inter_p*

El objetivo del *wrapper* es poder implementar una interfaz TLM en el bloque y que este se pueda comunicar externamente usando la misma. No obstante, internamente continuará con el protocolo de comunicación propio del módulo *inter_p*.

4.3.1.1. Definición de las interfaces y los puertos

La Tabla 4.4 recoge la descripción de los puertos de entrada y salida de este bloque.

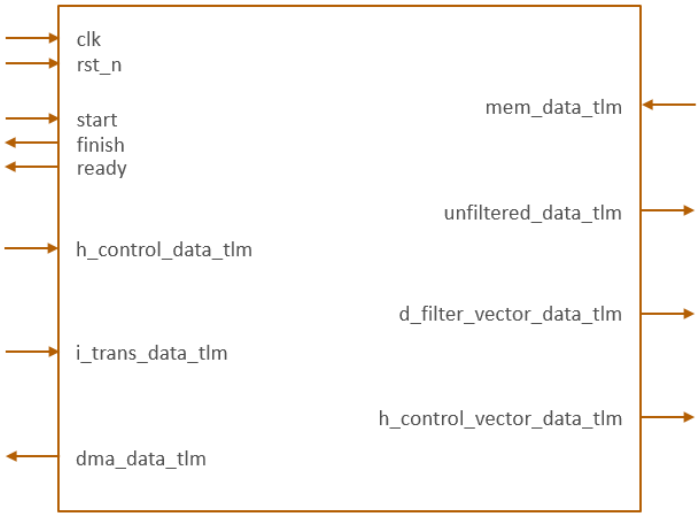


Figura 4.8: Puertos E/S del *wrapper* del bloque *inter_p*

Puerto	Ancho	Descripción
`clk`	1	Reloj
`rst_n`	1	*Reset*, activo a nivel bajo
`start`	1	Comienzo de los procesos
`finish`	1	Finalización de los procesos
`ready`	1	Señalización de que los datos están preparados
`h_control_data_tlm`	64	Comunicación con el módulo CAVLD
`i_trans_data_tlm`	9	Comunicación con el módulo *iqit*
`dma_data_tlm`	34	Comunicación para el acceso a la memoria
`mem_data_tlm`	8	Comunicación para la transferencia con la memoria
`unfiltered_data_tlm`	8	Comunicación con el módulo *d_filter*
`d_filter_vector_data_tlm`	10	Vectores de movimiento
`h_control_vector_data_tlm`	10	Vectores de movimiento

Tabla 4.4: Descripción de los puertos E/S del *wrapper*

En el archivo *inter_p_wrapper.h* se encuentran definidas tanto las interfaces como los puertos. Como ya se dijo al principio de este apartado, **el bloque *inter_p* contiene señales unidireccionales**, por lo que se usan las interfaces `get` y `put` para la lectura y escritura de los datos respectivamente. La interfaz `peek` no está soportada por la herramienta de síntesis CtoS.

Por otro lado, **se utilizan interfaces bloqueantes**, ya que para poder controlar el protocolo de comunicación interno es necesario el uso de la función `wait()`.

```
SC_MODULE (inter_p_wrapper){

    sc_in_clk clk;
    sc_in<bool>rst_n;

    sc_in<bool>start;
    sc_out<bool>finish;
```

```
        sc_out<bool>ready;

        sc_port< tlm_blocking_get_if< sc_uint<64> > >h_control_data_tlm;
        sc_port< tlm_blocking_get_if< sc_int<9> > >i_trans_data_tlm;
        sc_port< tlm_blocking_put_if< sc_uint<8> > >unfiltered_data_tlm;
        sc_port< tlm_blocking_put_if< sc_int<10> > >h_control_vector_data_tlm;
        sc_port< tlm_blocking_put_if< sc_int<10> > >d_filter_vector_data_tlm;
        sc_port< tlm_blocking_put_if< sc_uint<34> > >dma_data_tlm;
        sc_port< tlm_blocking_get_if< sc_uint<8> > >men_data_tlm;

        ...
};
```

4.3.1.2. Definición de las funciones y los procesos

El *wrapper* **por un lado tiene que sincronizarse con el diseño de referencia, y por otro, adaptarse al modelado TLM**, de manera que sus funciones tendrán que escribir o leer los datos enviados a través del canal TLM e implementar el protocolo de comunicación con los diferentes bloques que constituyen el modelo original, incluyendo la lectura o escritura de los datos.

Al introducir el *wrapper*, este debe almacenar cada una de las memorias que conforman las interfaces del modelo de referencia. Por ejemplo, el *inter_p_wrapper* está a la espera de recibir las once palabras correspondientes a `h_control_interface_data`. Sin embargo, el *tb_wrapper* no puede transmitirlas hasta que el protocolo de comunicación con el *testbench* comience, y una vez se ha iniciado, debe almacenar las once palabras para luego enviarlas vía TLM.

A continuación se muestra el código de las funciones y los procesos utilizados en este bloque:

```
SC_MODULE (inter_p_wrapper){

        ...
```

```
        void h_control_process();
        void i_trans_process();
        void unfiltered_process();
        void vector_process();
        void memory_process();

        SC_CTOR(inter_p_wrapper) :

            ...

            SC_THREAD(h_control_process);
            sensitive «clk.pos();

            SC_THREAD(i_trans_process);
            sensitive «clk.pos();

            SC_THREAD(unfiltered_process);
            sensitive «clk.pos();

            SC_THREAD(vector_process);
            sensitive «clk.pos();
            SC_THREAD(memory_process);
            sensitive «clk.pos();

        }
};
```

En la siguiente tabla se recoge la descripción de las funciones.

Función	Descripción
`void h_control_process()`	Lee los datos del puerto `h_control_data_tlm`. Implementa el protocolo de comunicación interno correspondiente al CAVLD
`void i_trans_process()`	Lee los datos del puerto `i_trans_data_tlm`. Implementa el protocolo de comunicación interno correspondiente al *iqit*
`void unfiltered_data()`	Escribe los datos en el puerto `unfiltered_data_tlm`. Implementa el protocolo de comunicación interno correspondiente al *d_filter*
`void vector_process()`	Escribe los datos en los puertos `h_control_vector_data_tlm` y `d_filter_vector_data_tlm`. Implementa el protocolo de comunicación interno correspondiente a los vectores de movimiento
`void memory_data()`	Lee los datos del puerto `mem_data_tlm` y escribe los datos en el puerto `dma_data_tlm`. Implementa el protocolo de comunicación interno correspondiente a la memoria

Tabla 4.5: Descripción de las funciones del *wrapper* del bloque *inter_p*

4.3.2. *Wrapper testbench*

Al igual que el *wrapper* del bloque *inter_p*, el *wrapper* del *testbench* dotará a este de una interfaz TLM.

Las definiciones tanto de las interfaces como de las funciones y los procesos se encuentran en el archivo *tb_wrapper.h*.

4.3.2.1. Definición de las interfaces y los puertos

La descripción de los puertos de entrada y salida es la misma que para el bloque en estudio (Tabla 4.4).

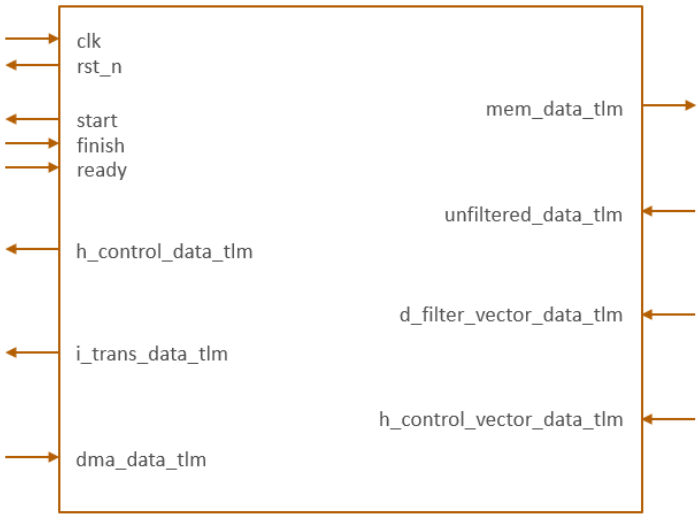


Figura 4.9: Puertos E/S del *wrapper* del *testbench*

En TLM se deben implementar el mismo tipo de interfaces en los distintos módulos que se conecten, es decir, si en el bloque *inter_p* se usaron interfaces bloqueantes, en el *testbench* se utilizará el mismo tipo de interfaz.

```
SC_MODULE (tb_wrapper) {

     sc_in_clk clk;
     sc_out<bool>rst_n;

     sc_in<bool>start;
     sc_out<bool>finish;
     sc_out<bool>ready;

     sc_port< tlm_blocking_put_if< sc_uint<64> > >h_control_data_tlm;
     sc_port< tlm_blocking_put_if< sc_int<9> > >i_trans_data_tlm;
     sc_port< tlm_blocking_get_if< sc_uint<8> > >unfiltered_data_tlm;
     sc_port< tlm_blocking_get_if< sc_int<10> > >h_control_vector_data_tlm;
     sc_port< tlm_blocking_get_if< sc_int<10> > >d_filter_vector_data_tlm;
     sc_port< tlm_blocking_get_if< sc_uint<34> > >dma_data_tlm;
```

```
        sc_port< tlm_blocking_put_if< sc_uint<8> > >men_data_tlm;
        ...
};
```

4.3.2.2. Definición de las funciones y de los procesos

El código de las funciones y los procesos establecidos es el que sigue:

```
SC_MODULE (tb_wrapper){

    ...

    void h_control_process();
    void i_trans_process();
    void unfiltered_process();
    void vector_process();
    void memory_process();

    SC_CTOR(tb_wrapper) :
        ...
        SC_THREAD(h_control_process);
        sensitive «clk.pos();

        SC_THREAD(i_trans_process);
        sensitive «clk.pos();

        SC_THREAD(unfiltered_process);
        sensitive «clk.pos();

        SC_THREAD(vector_process);
        sensitive «clk.pos();
```

```
                SC_THREAD(memory_process);
                sensitive «clk.pos();
            }
        };
```

Como se puede observar las funciones y procesos usados son exactamente los mismos que en el *wrapper* del bloque *inter_p*. Sin embargo, donde en aquel se lee un dato, en este se escribe. En la siguiente tabla se expone la descripción de las funciones.

Función	Descripción
`void h_control_process()`	Escribe los datos en el puerto `h_control_data_tlm`. Implementa el protocolo de comunicación interno correspondiente al CAVLD
`void i_trans_process()`	Escribe los datos en el puerto `i_trans_data_tlm`. Implementa el protocolo de comunicación interno correspondiente al *iqit*
`void unfiltered_data()`	Lee los datos del puerto `unfiltered_data_tlm`. Implementa el protocolo de comunicación interno correspondiente al *d_filter*
`void vector_process()`	Lee los datos de los puertos `h_control_vector_data_tlm` y `d_filter_vector_data_tlm`. Implementa el protocolo de comunicación interno correspondiente a los vectores de movimiento
`void memory_data()`	Escribe los datos en el puerto `mem_data_tlm` y lee los datos del puerto `dma_data_tlm`. Implementa el protocolo de comunicación interno correspondiente a la memoria

Tabla 4.6: Descripción de las funciones del *wrapper* del *testbench*

4.4. Verificación funcional

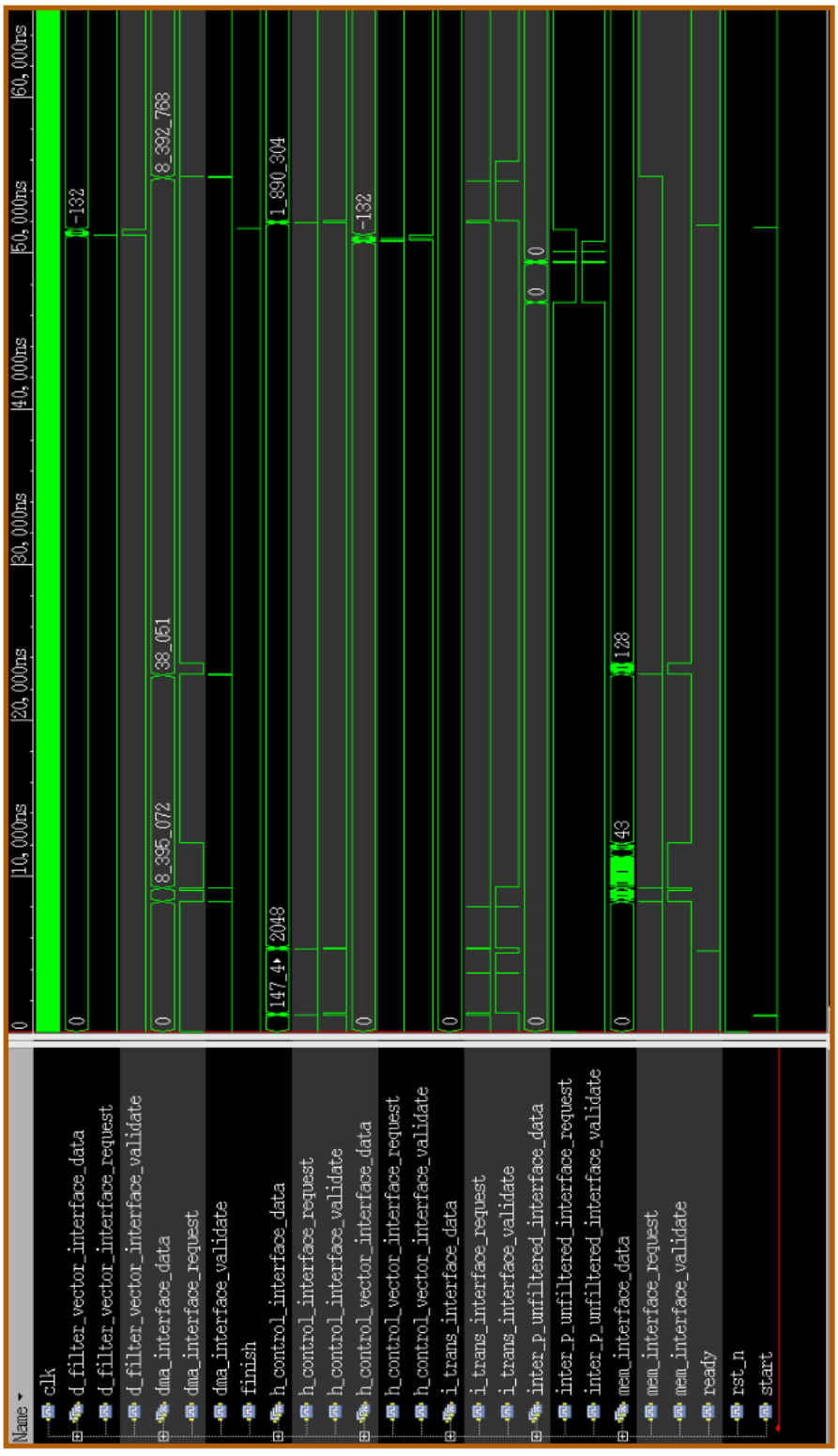


Figura 4.10: Simulación del sistema original en SystemC

Para comprobar el correcto funcionamiento del sistema se debe realizar la verificación funcional. Esto es necesario llevarlo a cabo antes de modelar el diseño a nivel de transacciones, ya que los módulos son “cajas grises” e interesa conocer cómo se desarrolla el comportamiento del sistema; y después del modelado, para corroborar que se sigue cumpliendo dicho comportamiento.

En la Figura 4.10 se muestra una captura del sistema original en funcionamiento. No obstante, es importante observar cómo se produce el protocolo de comunicación entre los diferentes módulos, prestando atención a los distintos conjuntos de señales *request*, *validate* y *data*.

La Figura 4.11 es una captura de la misma simulación pero centrada en la franja de tiempo entre 0,9 μs y 1,2 μs, de manera que se pueda observar con mayor detalle la primera transferencia de datos que se produce, concretamente entre el bloque *inter_p* y el CAVLD.

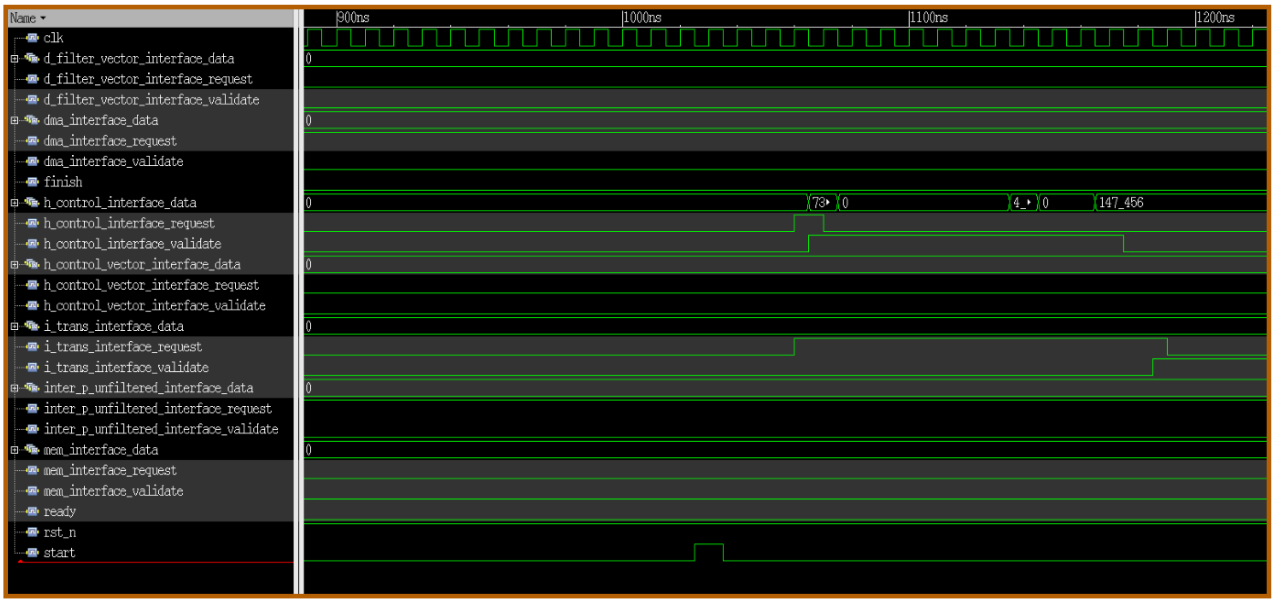


Figura 4.11: Simulación del sistema original. Protocolo de comunicación

Una vez analizado el comportamiento del sistema, se debe simular el mismo tras su modelado a nivel de transacciones, donde no hay ningún protocolo de comunicación externo, únicamente la transferencia de datos.

El entorno gráfico permite observar las transacciones que se están transmitiendo (Figuras 4.12 y 4.13). En ellas se muestra el tipo de operación que está realizando, `put` o `get`; y el tipo de dato que está transportando.

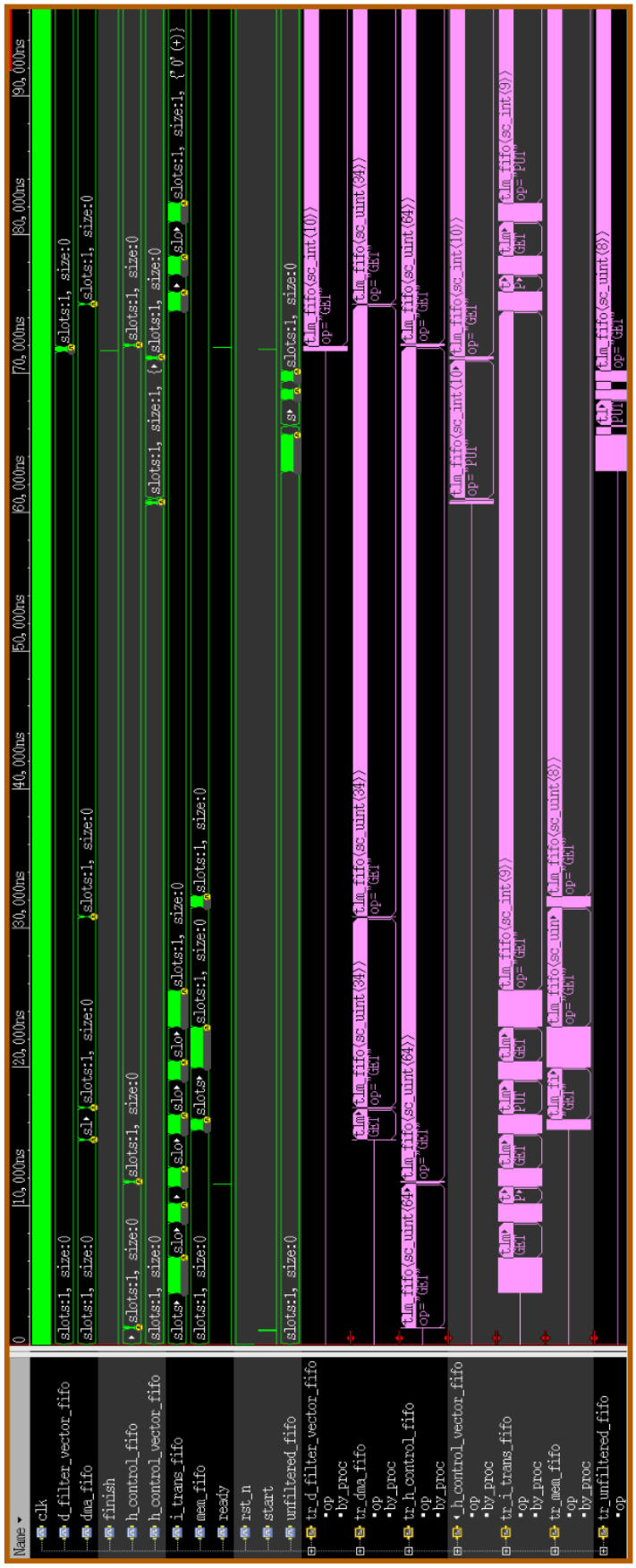


Figura 4.12: Simulación del diseño modelado a nivel de transacciones

Figura 4.13: Simulación del diseño TLM. Transacciones

Además, se pueden ir visualizando los datos que se transmiten en la consola de la herramienta, ya que en el código se han dispuesto sentencias *cout* que permiten monitorizar las entradas y salidas del sistema.

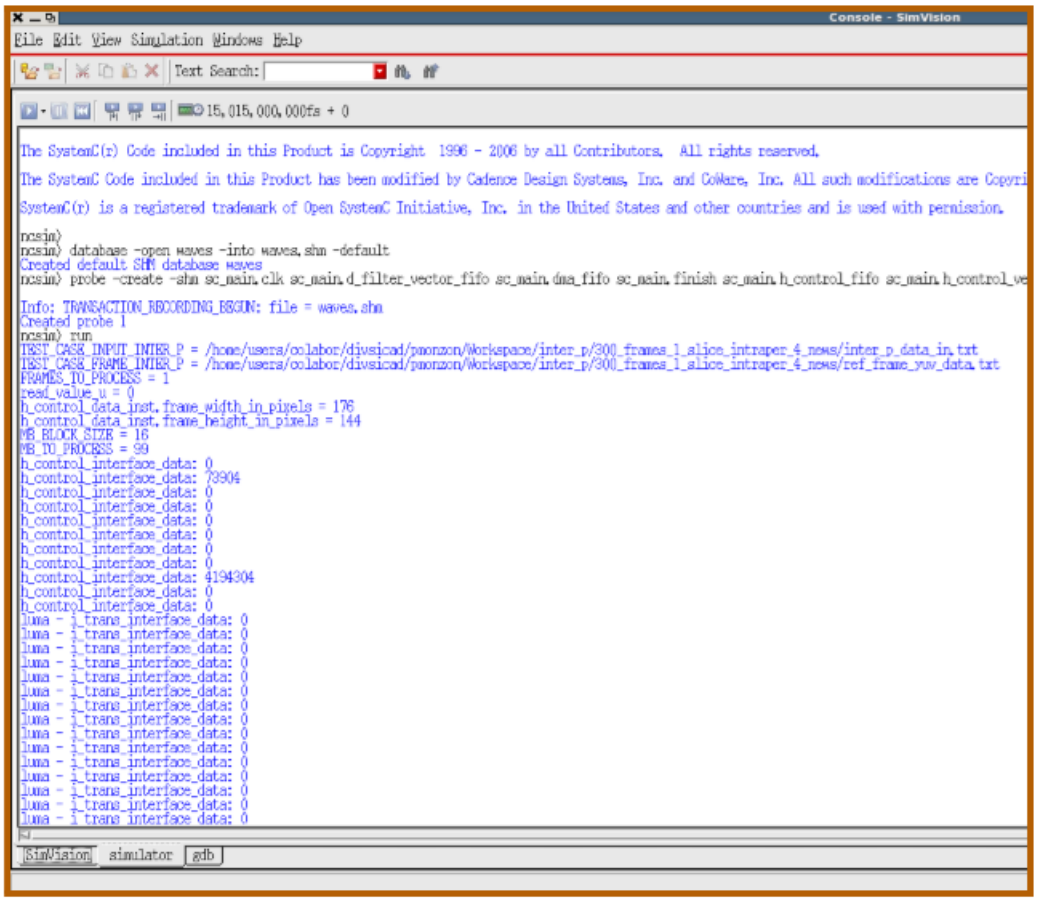


Figura 4.14: Consola de la herramienta Simulation Analysis Environment SimVision

Al igual que para el sistema original, se ha centrado la simulación en una franja de tiempo concreta (entre 1 μs y 1,3 μs), de manera que se pueda observar con mayor detalle la primera transferencia de datos que se produce entre el bloque *inter_p* y el CAVLD.

Como se puede ver en la imagen, el protocolo de comunicación que se observaba en la Figura 4.11 desaparece y únicamente se tienen las transacciones, característica de esta metodología donde el protocolo de comunicación es interno y no visible para el diseñador.

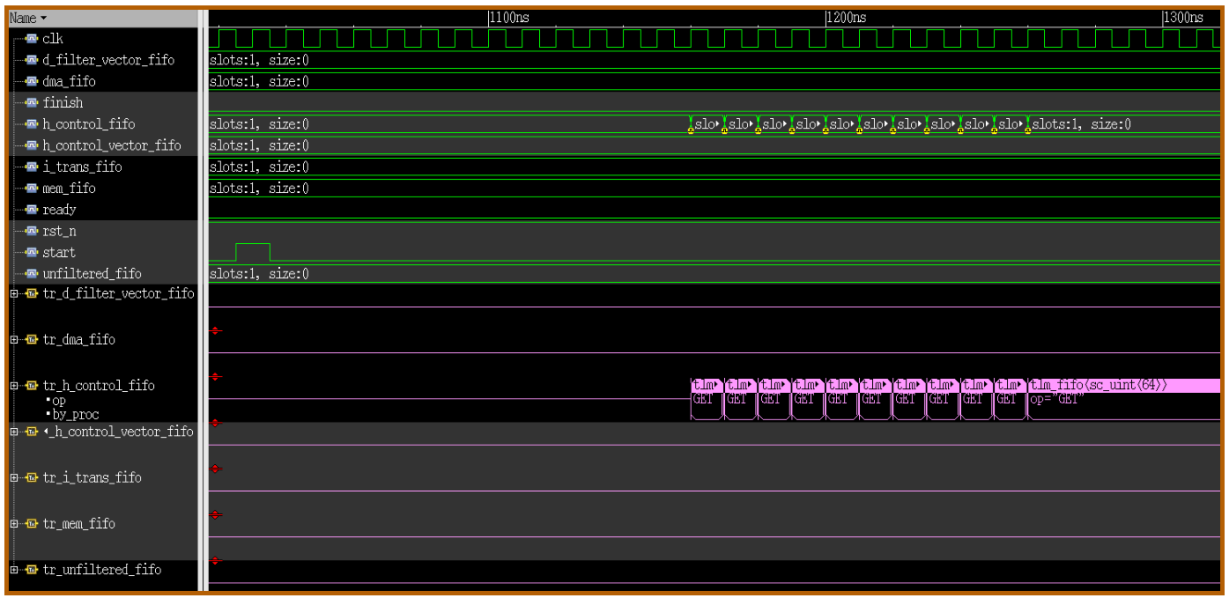


Figura 4.15: Diseño modelado. Transferencia de datos

Se puede apreciar que después de la implementación TLM se ha introducido un retardo en el sistema. En las simulaciones (Figuras 4.16 y 4.17) se han añadido dos marcadores (*TimeA* y *TimeB*) en los instantes en los que se activa la señal *start* (inicio de la simulación) con el fin de que se pueda observar el tiempo de dicho momento en cada simulación.

La latencia en el nuevo diseño se debe principalmente a las llamadas a la función `wait()`, cuya tarea es introducir un ciclo de reloj. Por un lado, en TLM las interfaces bloqueantes las implementan, y por otro, permiten coordinar los protocolos de comunicación internos que han sido diseñados.

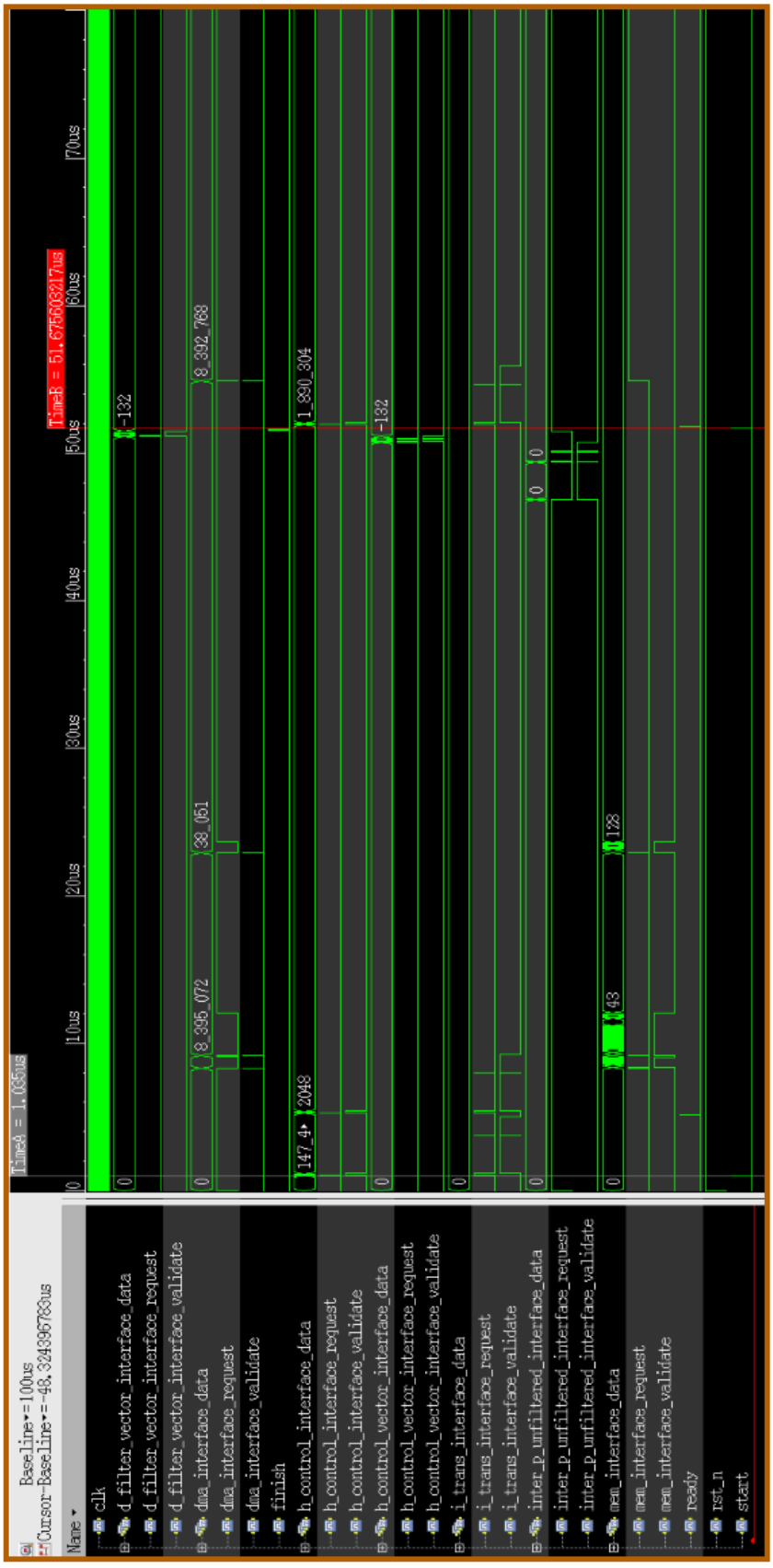


Figura 4.16: Simulación del sistema original

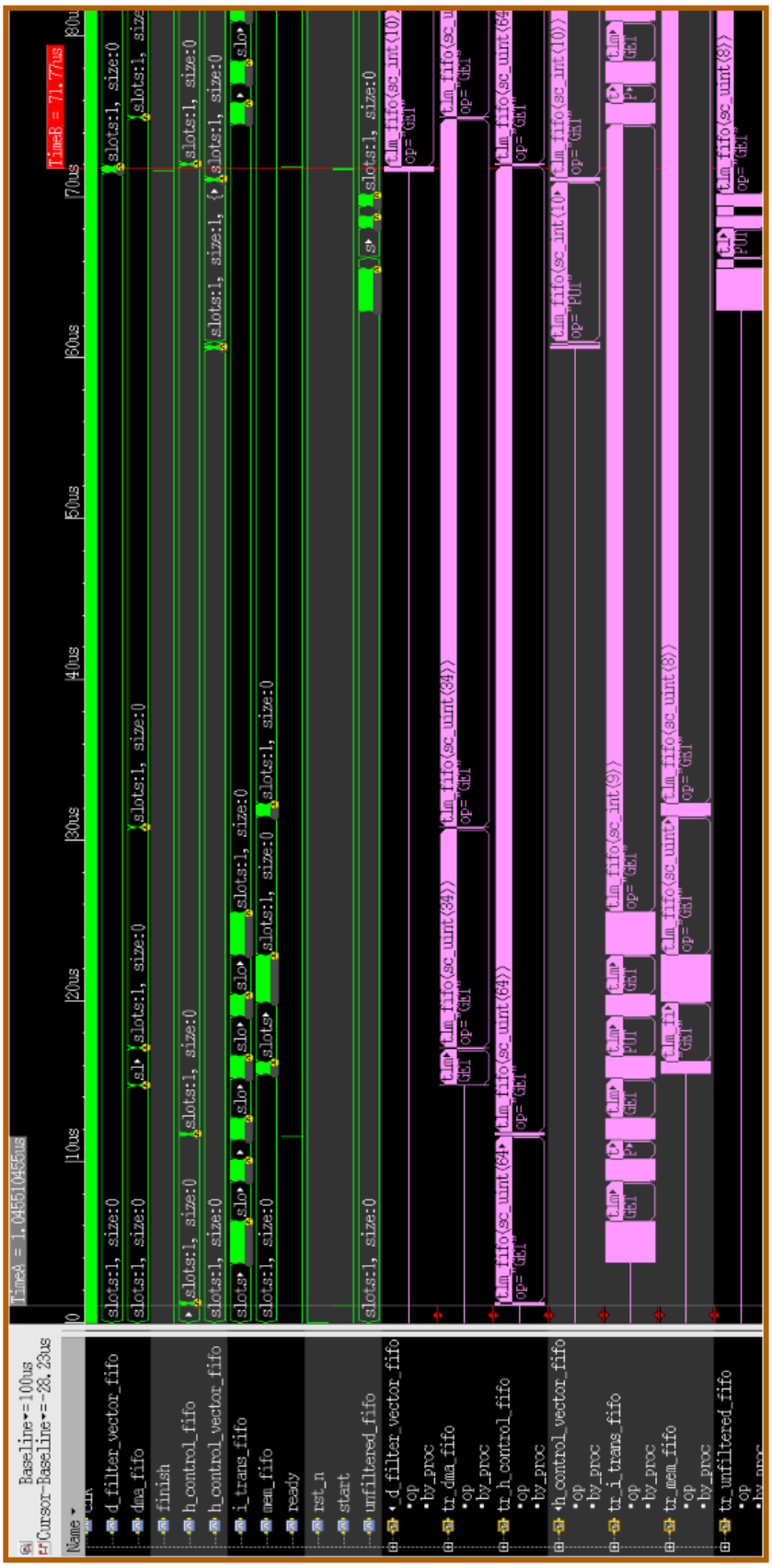


Figura 4.17: Simulación del diseño modelado a nivel de transacciones

4.5. Síntesis

Tras el modelado y la verificación, la siguiente etapa del flujo de diseño es la síntesis. Consiste en convertir o traducir un modelo descrito en un nivel de abstracción alto a una descripción del mismo a un nivel inferior, añadiendo detalles de implementación al sistema, por ejemplo, tras la síntesis de un modelo en HDL se obtendría su descripción a nivel de puertas.

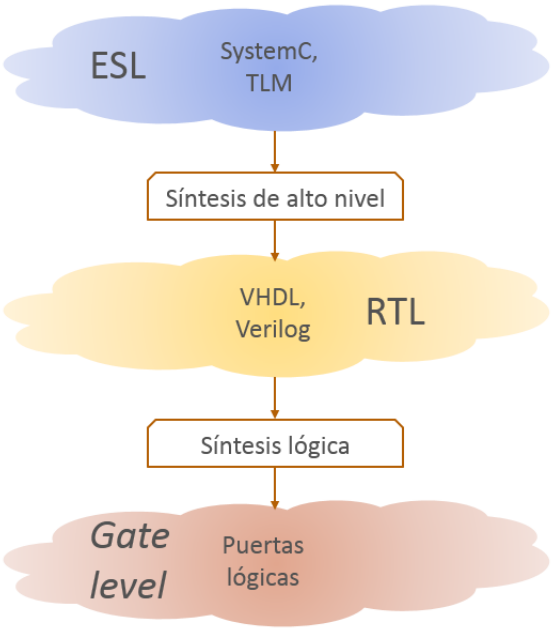


Figura 4.18: Flujo de la síntesis

El objetivo de este apartado es la presentación del proceso de síntesis, partiendo del modelado TLM y centrándose en la síntesis de alto nivel, a fin de obtener la microarquitectura a nivel RTL.

4.5.1. Consideraciones previas

En la síntesis de alto nivel existen muchos grados de libertad a la hora de implementar una determinada función, intervienen variables como el consumo, el área ocupada, la velocidad de operación, etc. Dado que normalmente no es posible obtener un sistema sintetizado que optimice todos estos parámetros, es necesario establecer un compromiso entre los mismos o determinar cuál de ellos es el que más interesa. De ahí que el proceso de síntesis siempre vaya unido al de optimización [57].

Los tres parámetros básicos a optimizar son el área o recursos ocupado, la latencia, que es el número de ciclos necesarios para completar una función, y el tiempo de ciclo, que vendrá determinado

por el bloque más lento del sistema. Estos parámetros pueden usarse como **restricciones del diseño** y se deberán tener en cuenta a la hora de efectuar la optimización.

Claramente, el objetivo principal de este proyecto es obtener modelos sintetizables a partir del modelado a nivel de transacciones. Para ello se tendrán en cuenta los parámetros indicados como restricciones de diseño, asegurando que el sistema tras la síntesis continúe funcionando correctamente.

Por otro lado, existen dos enfoques basados en la partición del sistema en módulos más simples, que al interconectarlos consiguen la funcionalidad principal. Al módulo que integra y conecta al resto se le conoce como *top* del diseño. Las estrategias de síntesis que se pueden adoptar son las siguientes:

1. *top-down*, consiste en crear un único proyecto para la síntesis, cuyo módulo principal es el *top* y este *instancia* los demás bloques. El sistema se sintetiza a partir del *top* del diseño.
2. *bottom-up*, se basa en sintetizar cada bloque del sistema por separado y finalmente integrarlos todos mediante el módulo *top*, el cual está dedicado a definir la conectividad entre ellos.

En este caso **se va a utilizar la primera estrategia, *top-down***, ya que permite obtener mejores resultados de síntesis para los bloques del diseño.

4.5.2. Síntesis de alto nivel

La síntesis de alto nivel o HLS (*High-Level Synthesis*) se basa en el principio de que todo sistema puede modelarse mediante una serie de operaciones y sus dependencias [57]. El primer paso de este proceso consiste en traducir la especificación que el diseñador propone utilizando uno de los lenguajes funcionales a nivel de sistemas, típicamente C/C++ o SystemC, en una representación RTL descrita en un HDL.

Cadence C-to-Silicon Compiler (CtoS) es un entorno de trabajo que **realiza la síntesis del diseño desde SystemC o TLM, transformando el mismo hasta una representación RTL** en Verilog.

Antes de comenzar con la síntesis se debe incluir una interfaz de *reset* en todos los procesos, requisito obligatorio para que los procesos se puedan implementar como sistemas síncronos. La forma de introducir la especificación del *reset* mantiene la compatibilidad del modelo con las herramientas de simulación y con otras de síntesis. Para ello se hace uso del preprocesador de C y de las macros

definidas por CtoS (`#if defined(__CTOS__) || defined(CTOS_MODEL)`).

```
SC_THREAD (memory_process);
sensitive «clk.pos();
reset_signal_is (rst_n, false);


void inter_p_wrapper::memory_process(){
// Reset
#if defined(__CTOS__) || defined(CTOS_MODEL)
dma_data_tlm->reset_put();
mem_data_tlm->reset_put();
#endif
...
```

4.5.2.1. Flujo de diseño CtoS

Aunque en el Capítulo 3 se mostraba cuál era el flujo de diseño de la herramienta, ahora se describen en detalle los pasos seguidos. Además, CtoS permite escribir en un *script* las mismas acciones que se siguen a través de la interfaz gráfica y así facilitar el proceso de síntesis, con el consecuente ahorro de tiempo, por lo que también se indicará la sintaxis TCL de las operaciones llevadas a cabo.

1. **Configuración del diseño**

   El primer paso a realizar es la configuración del diseño, que incluye la selección del dispositivo o de las librerías tecnologícas, reloj, código fuente, macros definidos por el usuario, y otras opciones:

   - **Se especifica que se trata de un diseño TLM.**
   - **Se indica el dispositivo de prototipado final** y sus características. Para el caso del ASIC se referencian las librerías necesarias en formato Liberty, habiéndose usado en este proyecto UMC de 65 nm.

	Librería	Tensión (V)	Temperatura (°C)
Caso Típico	uk65lscsp10bbrccs_100c25_tc	1,0	25
Mejor Caso	uk65lscsp10bbrccs_110c0_bc	1,1	0
Peor Caso	uk65lscsp10bbrccs_090c125_wc	0,9	125

Tabla 4.7: Características de las librerías UMC de 65 nm

Para el caso de la FPGA se identifica el fabricante y la familia de la misma. El diseño se implementará en tres FPGAs:

Familia	Dispositivo	Registros	BRAMs	LUTs	DSPs
Xilinx	Virtex5 xc5vfx130tff1738-1	81.920	298	81.920	320
	Zynq xc7z045fbg676-2	437.200	545	437.200	900
	Spartan6 xc6slx9csg225-2	11.440	32	5.720	16

Tabla 4.8: Dispositivos FPGA utilizados en este proyecto y sus recursos disponibles

- **Se define el reloj,** donde se indica la frecuencia y el ciclo de trabajo, el cual se da en términos de periodo de reloj y *offset* del flanco de bajada. Los valores se indican en pico-segundos (ps).

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

Se establece una señal de reloj de nombre “clk”, frecuencia de 100 MHz (periodo de 10 ns) y ciclo de trabajo del 50 %. El nombre del reloj debe ser el mismo que el usado en el diseño, pues luego será utilizado en el refinamiento de los puertos.

- **Se definen los *transactors*,** uno por cada interfaz.

```
define_tlm_transactor
```

```
-name nombre_transactor
-class {clase_transactor}
-source_file ctos_transactors/ctos_transactors.h
-clock nombre_reloj
-reset nombre_reset
-default_mirror_transactor {clase_transactor_espejo}
```

Un ejemplo sería:

```
define_tlm_transactor
     -name h_control_data_tlm_tx
     -class {tlm_blocking_put_tx< sc_uint<64> > >}
     -source_file ctos_transactors/ctos_transactors.h
     -clock clk
     -reset rst
     -default_mirror_transactor {tlm_blocking_put_mirror_tx<
          sc_uint<64> > >}
```

- **Se refinan los puertos.**

```
refine_tlm_interface
     -port nombre_puerto
     -transactor {nombre_transactor}
     -clock nombre_reloj
     -reset nombre_reset
     -mirror_transactor {clase_transactor_espejo}
```

Por ejemplo:

```
refine_tlm_interface
      -port h_control_data_tlm
      -transactor {h_control_data_tlm_tx}
      -clock clk
      -reset rst_n
      -mirror_transactor {tlm_blocking_put_mirror_tx< sc_uint<64> > >}
```

Al igual que el nombre de la señal de reloj, el nombre de la señal de *reset* también debe ser el mismo que el especificado durante el modelado.

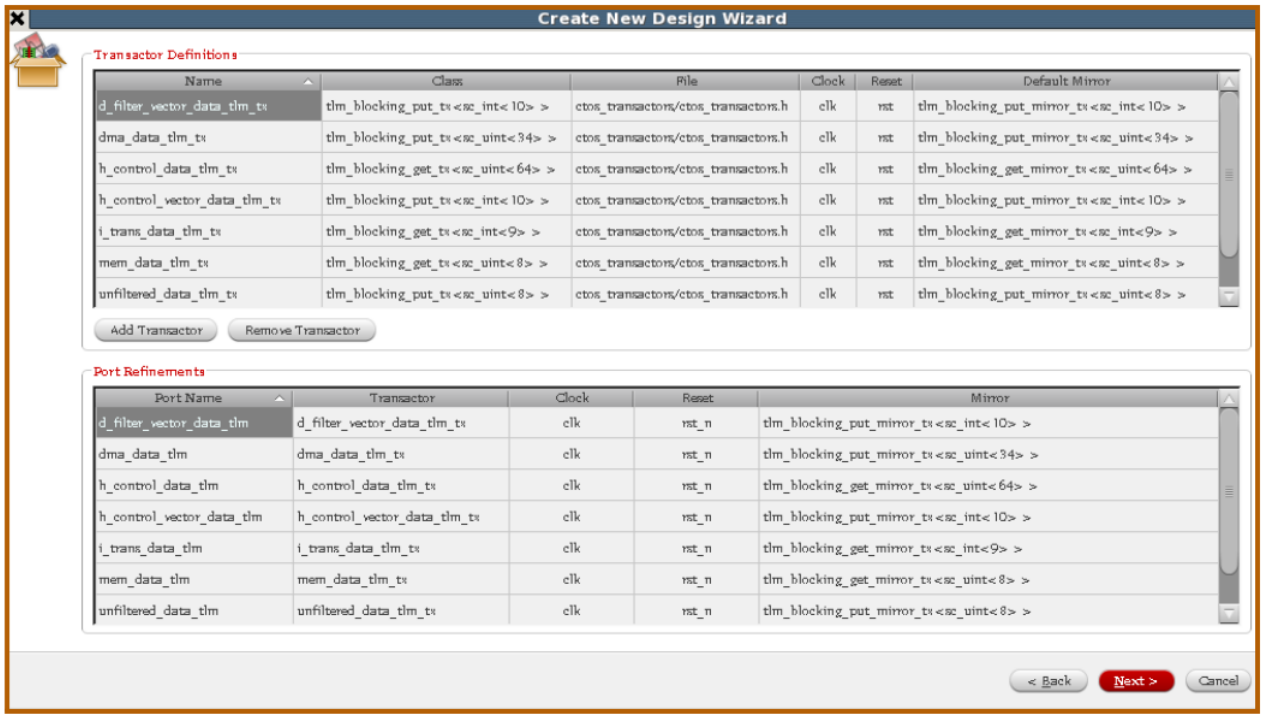


Figura 4.19: CtoS. Definición de los *transactors* y refinamiento de los puertos

2. **Especificación de la micro-arquitectura**

   El siguiente paso es definir la microarquitectura del sistema:

   - **Resolución de los bucles combinacionales.** Se puede optar por una de las siguientes estrategias:
     - Desenrollar el bucle. Se ejecutan las iteraciones en paralelo, en un único ciclo, ya que replica la lógica interna del bucle una vez por cada iteración. Se minimiza el número de

ciclos a costa de aumentar los recursos *hardware* necesarios y el periodo de reloj. El desenrollado puede ser completo o parcial.

- Romper el bucle. Se añade una llamada a la función `wait()` al final de cada iteración, de manera que cada una se realice en un ciclo de reloj. Es la acción por defecto. En este caso se reutiliza el *hardware* y se procesa de forma secuencial cada iteración del bucle.
- Realizar una segmentación del bucle o *pipeline*. Rompe el bucle de forma que se ejecuta en varios ciclos o etapas, y en cada una de ellas se lleva a cabo una iteración, paralelizando así varias. Se mejora el rendimiento para los mismos con gran carga en cada iteración, pero es ineficiente para aquellos sencillos pues añade lógica adicional. Depende de su tamaño y de la dependencia de datos.

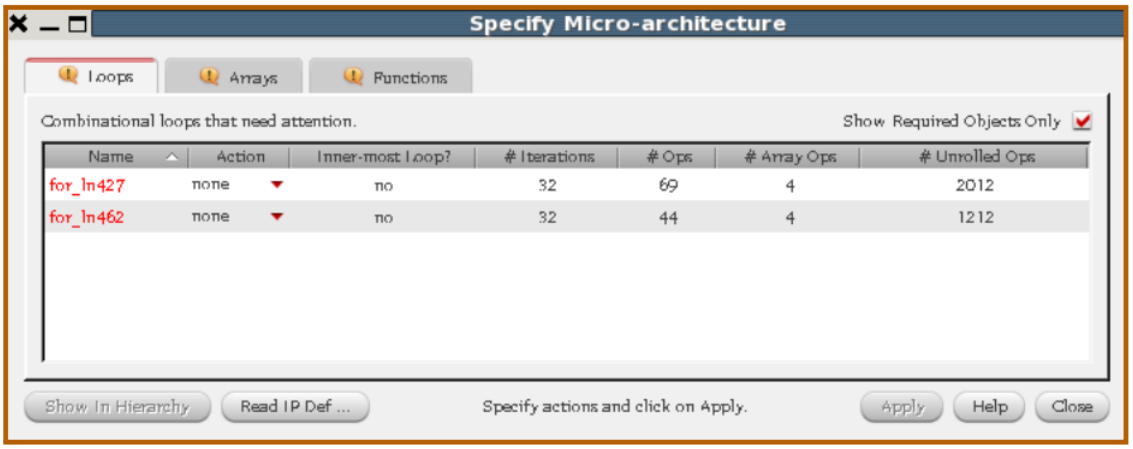


Figura 4.20: CtoS. Resolución de los bucles combinacionales

El diseño presenta únicamente dos bucles combinacionales, por lo que la ruptura de ambos es la opción más acertada.

```
set combo_loops [find_combinational_loops]
foreach loop $combo_loops {
      puts “breaking loop: $loop”
      break_combinational_loop $loop
}
```

- **Resolución de las funciones no planificables.** Algunas funciones no son sintetizables,

debido a latencias no constantes o por la llamada a la función desde distintos procesos concurrentes, entre otros casos. Para solucionarlo se hace la función “en línea” (*inline*), es decir, se sustituye la llamada por el código que la describe, replicando la función. Como consecuencia directa de esta acción aumenta el consumo de los recursos.

```
inline /designs/proyecto/modules/modulo/behaviors/funcion
```

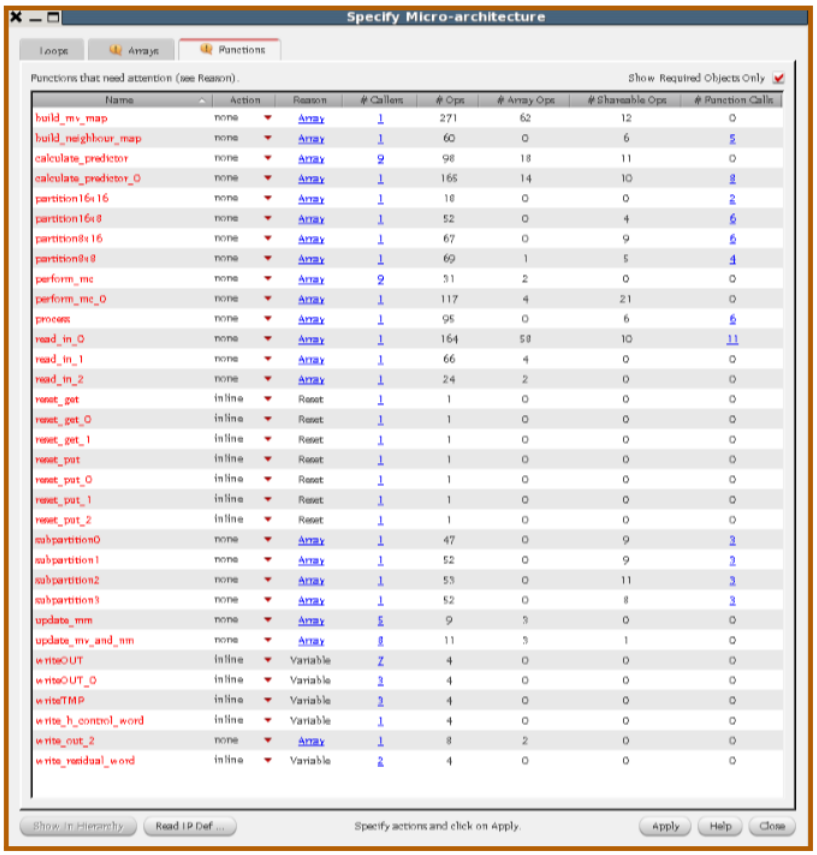


Figura 4.21: CtoS. Resolución de las funciones no planificables

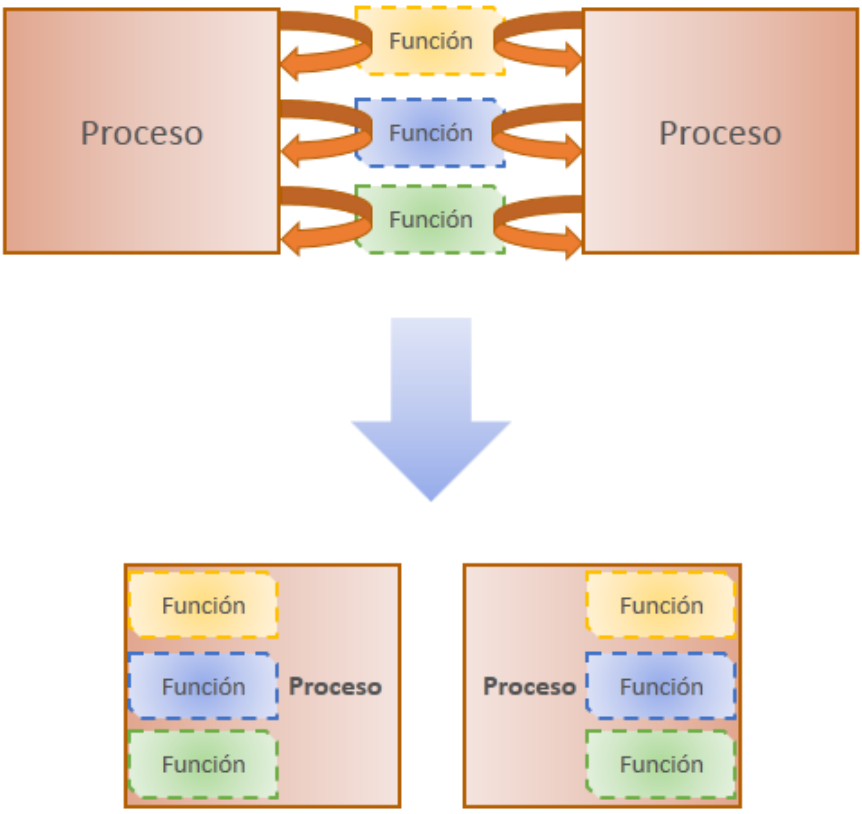


Figura 4.22: *Inline* (adaptado de [39])

3. **Asignación de IPs**

   En el caso de las variables de tipo vector se debe decidir qué tipo de recursos se utilizará para el almacenamiento *hardware* de los datos. Las posibles opciones son:

   - **RAM interna.** Implementa una memoria lógica, bloques BRAM o LUTs. Es muy útil para mapear una RAM en una FPGA.
   - **Registros.** Los datos se almacenan en registros. Es viable para casos de vectores de pocos elementos, y que estos no sean excesivamente grandes en términos de longitud de bits.
   - **RAM de terceros.** Especifica que un *array* de gran tamaño será una memoria en la implementación *hardware* final.

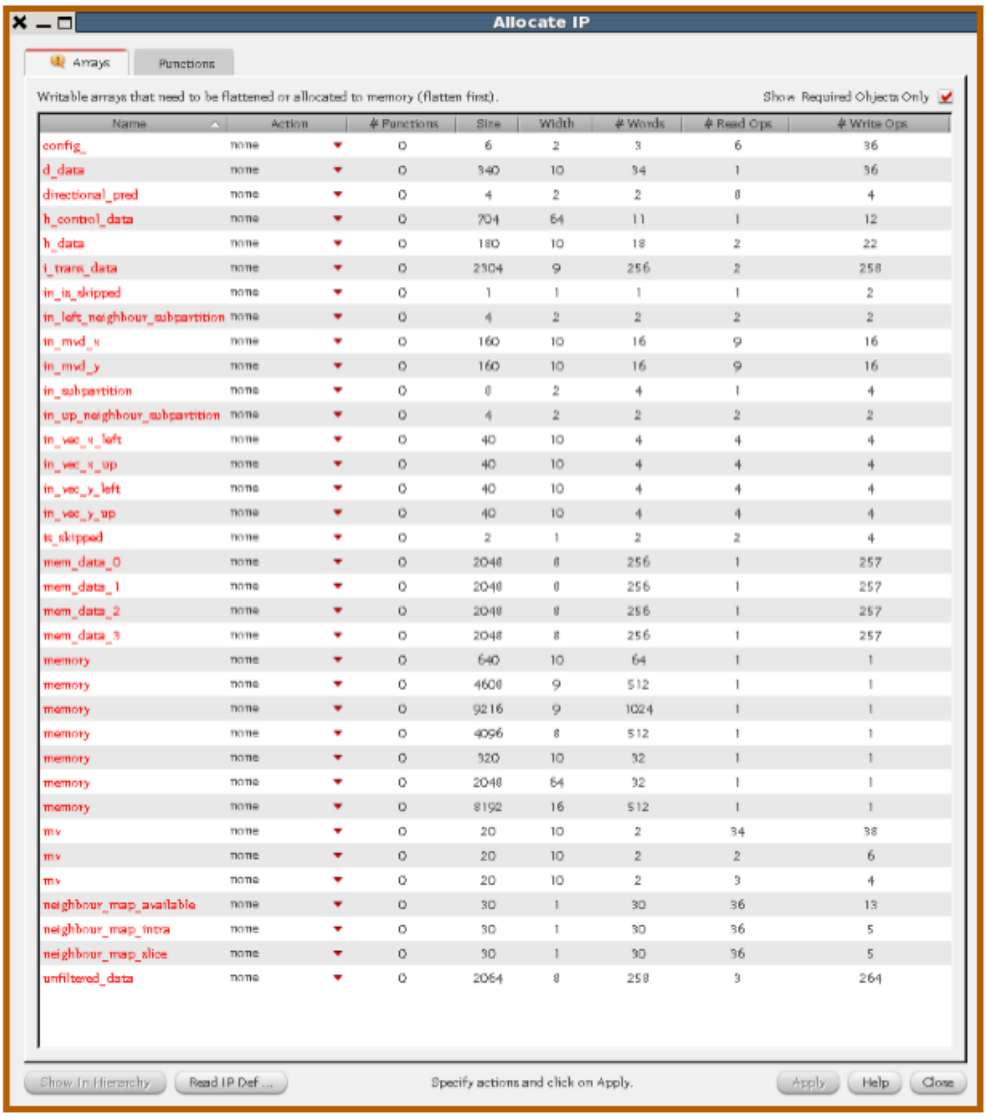


Figura 4.23: CtoS. Resolución de las variables de tipo vector

La elección de una u otra opción dependerá de los objetivos de una específica implementación del diseño, y de cómo sea este. Utilizar registros es la mejor decisión para vectores pequeños, mientras que una RAM ofrecerá una implementación más eficiente para aquellos más grandes, por lo que las siete memorias con las que cuenta el diseño, se mapearán en una RAM, y para el resto de vectores se usarán registros.

```
allocate_builtin_ram /designs/proyecto/modules/modulo/arrays/memoria
```

```
flatten_array /designs/proyecto/modules/modulo/arrays/vector
```

4. **Planificación**

   El planificador de CtoS ofrece una serie de opciones para poder obtener una mejor calidad de los resultados (QoR) de la síntesis del diseño [39], como por ejemplo:

   - Selección del esfuerzo, entre bajo o alto.
   - Realizar una optimización tras la planificación.

   Además, si el dispositivo de prototipado final es una FPGA se puede indicar el uso de bloques DSPs. Esta decisión la tomará el diseñador en función del diseño que haya realizado y si este tiene operaciones que puedan ser optimizadas usando un DSP, como por ejemplo sumadores, multiplicadores o unidades de multiplicación-acumulación. Para este proyecto se ha hecho uso de esta opción.

```
use_dsp /designs/modulo
```

   Por otro lado, CtoS permite incrementar el número de ciclos de latencia de una función cuando esta no es planificable con el número de ciclos propuesto por el diseñador. No obstante, esta acción no se debe llevar a cabo en funciones dedicadas a implementar los protocolos de comunicación, como es el caso de utilizar modelos precisos a nivel ciclos de bus (BCA). Para poder controlar este aspecto, el planificador hace uso del analizador de ciclos, especificando si no se quiere usar, si se quiere un análisis simple o bien que sea preciso.

4.5.2.2. Análisis

Con la propia herramienta de síntesis se pueden efectuar los primeros análisis: grafo de control de flujo, análisis de ciclos y análisis de área, siendo los parámetros a optimizar la latencia, el tiempo de ciclo y el área, respectivamente.

1. **Grafo de control y de flujo de datos**

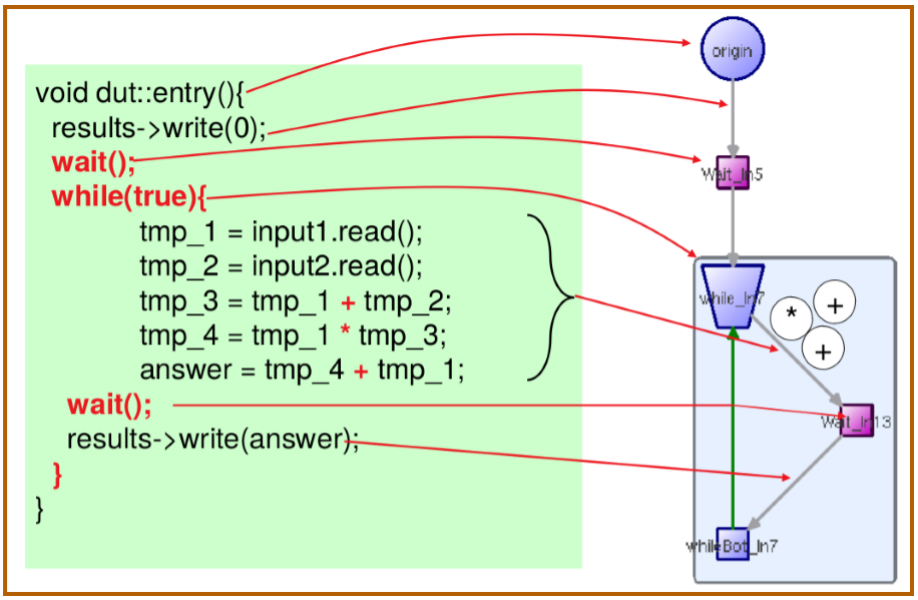


Figura 4.24: Grafo de control y de flujo de datos [39]

En el CDFG (*Control and Data Flow Graph*) se representan los distintos caminos que puede seguir la ejecución del bloque diseñado y las diferentes acciones que realiza en cada etapa. Además, se pueden ver los bucles, sus condiciones y las bifurcaciones generadas en base a alguna condición. Sirve para cuantificar las latencias del sistema para cada posible camino alternativo.

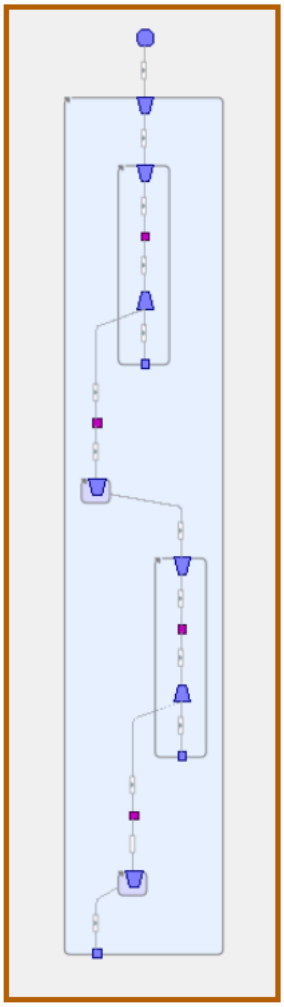


Figura 4.25: Grafo de control y de flujo de datos del modelo TLM

2. **Análisis de ciclos**

   Este análisis representa la ruta crítica del bloque, indica los retardos de cada uno de los elementos que lo componen y el *slack* de dicha ruta, tiempo máximo que se puede retrasar en el inicio sin que afecte al tiempo estimado para terminar la ejecución. También incluye los retardos debidos al *fan-out* de la señal representada (número máximo de entradas que es posible conectar a una puerta), el tiempo de *hold* del *flip-flop* origen de la ruta crítica y los retardos de la propagación de la señal.

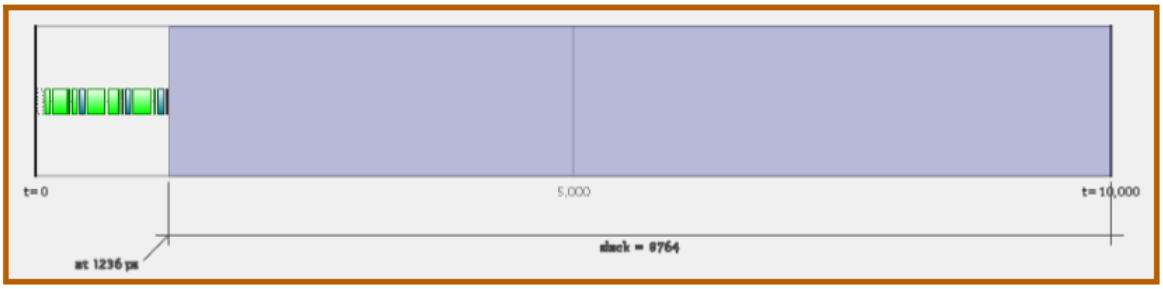


Figura 4.26: ASIC. Análisis de ciclos

El diseño está sintetizado con una frecuencia de funcionamiento de 100 MHz, es decir, un periodo de reloj de 10 ns. En la Figura 4.26 se puede observar que existe un *slack* positivo de 8,764 ns, lo que implica que el límite de frecuencia a la que puede funcionar el diseño no se ha alcanzado, pudiendo ser aumentada significativamente. En cambio, para FPGA, Figura 4.27, se tiene un *slack* negativo de 0,193 ns, por lo que la frecuencia seleccionada no es alcanzable por el diseño. La solución es reducir la frecuencia o partir la ruta crítica añadiendo llamadas a la función `wait()`. También es posible incluir restricciones de ciclos en el CDFG.

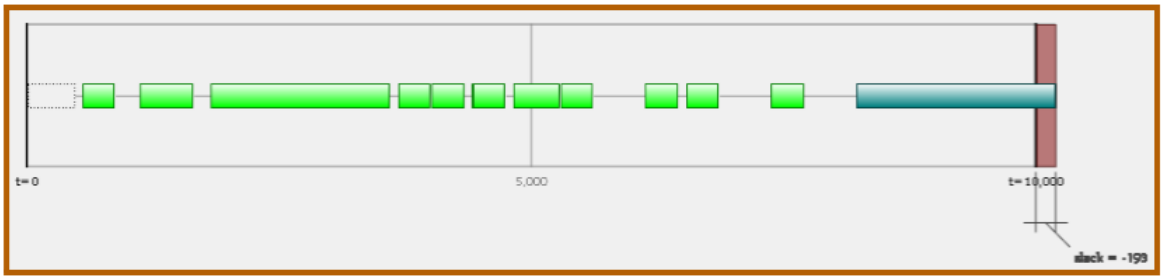


Figura 4.27: FPGA. Análisis de ciclos

3. **Análisis de área**

El consumo del área se representa en base a los elementos que forman parte del diseño y para una función en concreto. Para un ASIC es posible utilizar distintas librerías de células estándar, representando las categorías de celdas que constituyen el área total. En cambio, para FPGA este análisis permite identificar el conjunto de bloques básicos configurables (CLB, *slices*, BRAM, etc.) que conforman su estructura.

Figura 4.28: ASIC. Análisis de área

4.5.3. Verificación RTL

Después de realizar la síntesis de alto nivel se obtienen los módulos definidos en Verilog, de los cuales habrá que comprobar si mantienen el comportamiento original. Para ello se efectúa la simulación con las mismas entradas que para el diseño en alto nivel, es decir, se reutiliza el mismo *testbench*.

CtoS genera un *wrapper* en SystemC que **instancia** el modelo en Verilog obtenido, de manera que se pueda ejecutar la simulación con el *testbench* modelado anteriormente. Además, incluye dos *instancias* más: una del diseño TLM modelado en alto nivel y otra del diseño sintetizado en SystemC con los *transactors*. Esto queda reflejado en el archivo `sc_main` del diseño de la siguiente manera:

```
#if defined(CTOS_USE_DIRECT_SC)
     inter_p_wrapper_ctos_tlm_wrapper interp(“interp”, false);
#elif defined(CTOS_MODEL)
     inter_p_wrapper_ctos_tlm_wrapper interp(“interp”, true,
          CTOS_TARGET_SUFFIX(CTOS_MODEL));
#else
     inter_p_wrapper interp(“interp”);
#endif
```

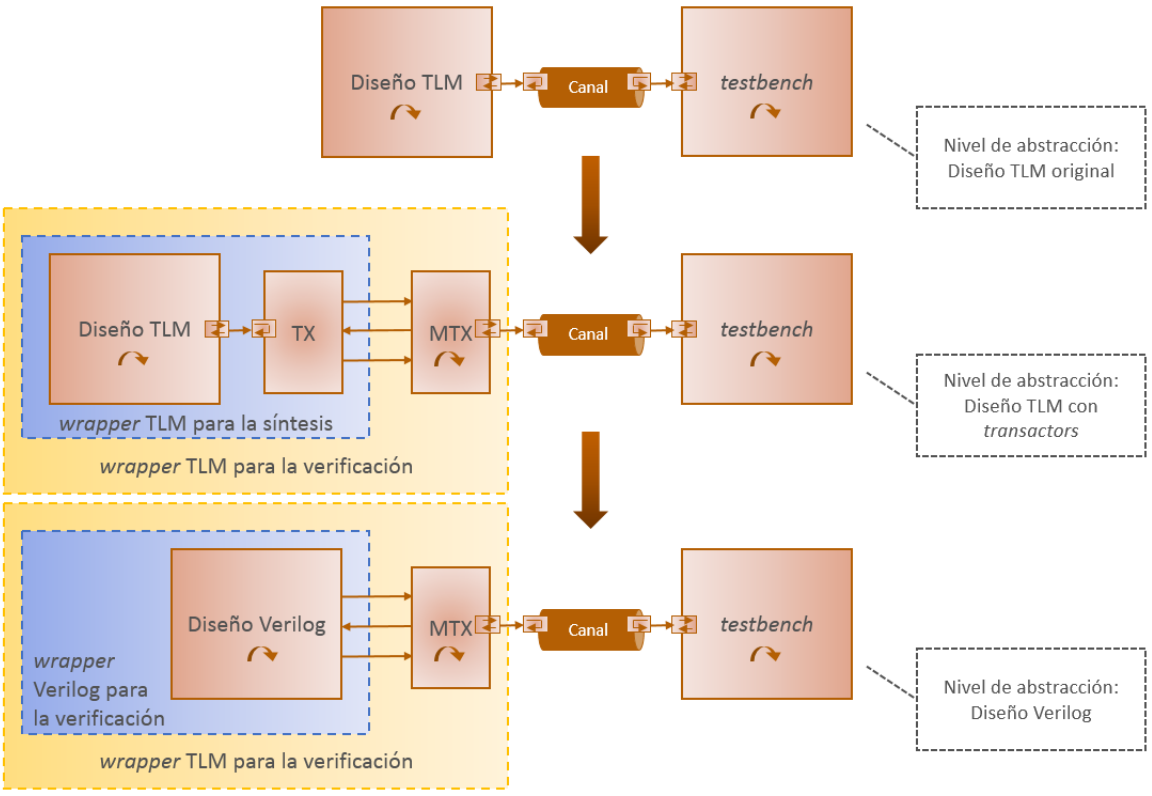


Figura 4.29: Opciones de simulación RTL (adaptado de [38])

La simulación y consecuente verificación del diseño generado por CtoS se ha llevado a cabo con la herramienta de Cadence Incisive y su interfaz de usuario SimVision. Además, se ha usado el *wrapper* con la *instancia* del modelo en Verilog.

En las Figuras 4.30 y 4.31 se muestra una captura de las simulaciones realizadas con los dos diseños, tanto para ASIC como para FPGA. Comparándolas con la del diseño original, pues en la del modelado solo se pueden observar las transacciones, **se puede verificar que el sistema es funcionalmente correcto.**

De forma ideal se espera que la herramienta de síntesis genere un diseño RTL cuyo comportamiento sea idéntico al diseño de entrada en alto nivel. No obstante, puede ocurrir que tenga diferentes interpretaciones o que sus descripciones no sean sintetizables, provocando discordancias.

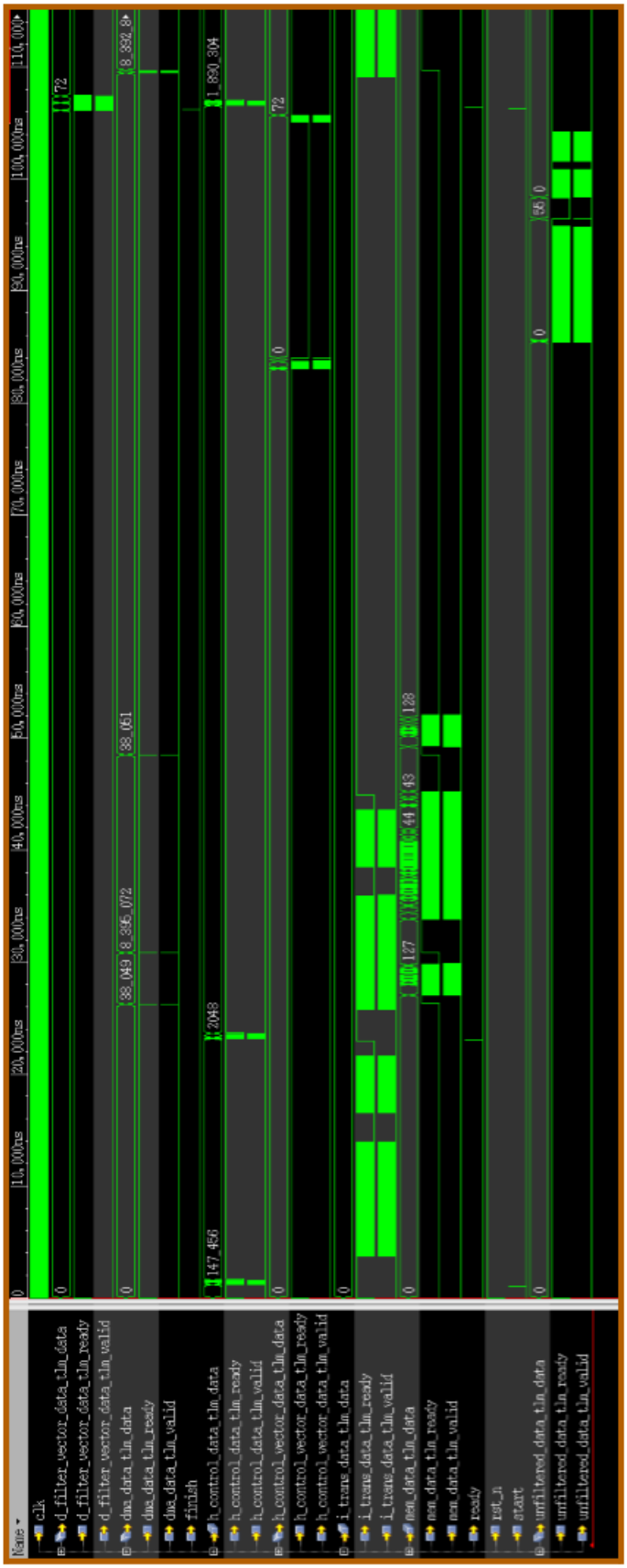


Figura 4.30: ASIC. Verificación RTL

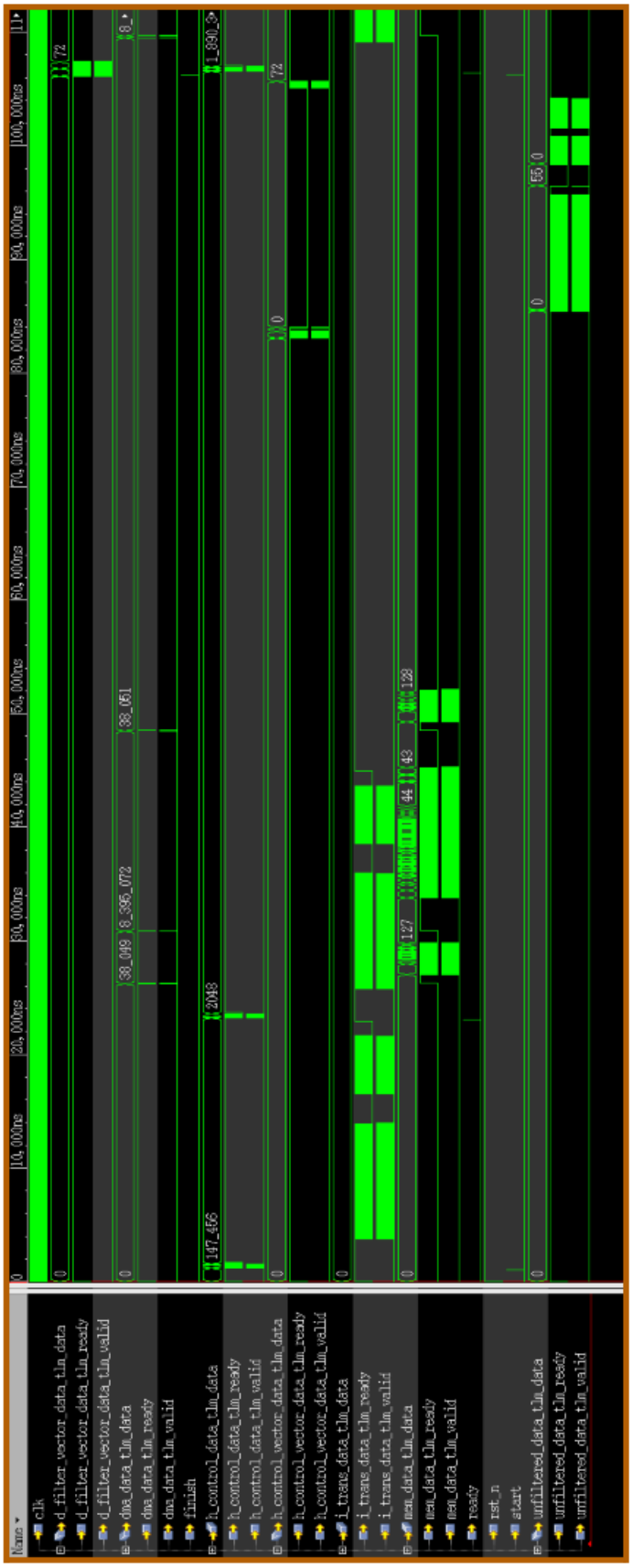


Figura 4.31: FPGA. Verificación RTL

Algunos aspectos que pueden dar lugar a diferencias entre el diseño en alto nivel y el sintetizado son:

- Uso de variables globales. CtoS soporta las variables globales dentro de un mismo proceso, de forma que una variable global es compartida tanto a la función principal como a las rutinas que sean llamadas desde la misma. Sin embargo, en los módulos donde existan dos o más hilos paralelos, el uso de esta no está permitido, provocando errores. En estos casos es necesario usar señales.
- Relajación de la latencia. Esta opción permite al planificador incrementar la latencia de un bloque para poder cumplir las restricciones temporales (ciclo de reloj) añadiendo registros intermedios. En las funciones de acceso a los puertos de salida o a las señales de control puede ocurrir que el diseño no cumpla las restricciones temporales impuestas por el protocolo de E/S. Por ejemplo, si se permite relajar la latencia en una función de control de lectura o escritura de una FIFO puede provocar que se inserten datos duplicados en la cola, o que se extraigan varios elementos.
- Inicialización de las memorias. Dado que las opciones para inicializar las memorias con SystemC implica acciones no soportadas en la síntesis, como por ejemplo usar bucles combinacionales, se obtienen memorias sin valores definidos. Para solucionar esto, en Verilog se ha usado la directiva `$readmemb` que permite la inicialización a través de un fichero externo que contenga los datos, separados por saltos de línea para cada posición.

4.5.4. Síntesis lógica

Para obtener un mejor análisis del diseño, es necesario descender en el nivel de abstracción y realizar la síntesis lógica a partir de la descripción RTL obtenida.

Dependiendo del dispositivo de prototipado final, Synopsys ofrece dos entornos de trabajo, Synplify para FPGA y Design Compiler para ASIC. En ambos casos los ficheros de entrada serán los generados por CtoS, tanto la descripción RTL en Verilog como las restricciones de entrada al entorno de síntesis.

4.5.4.1. Síntesis lógica para ASIC

Los pasos a realizar del flujo de diseño de Design Compiler son los siguientes:

- **Se cargan las librerías.** Como se nombró anteriormente, se utilizan las librerías UMC de 65 nm. Se debe seleccionar aquella para el caso en el que se trabajará, pudiendo ser *wc, worst case* (peor caso); *bc, best case* (mejor caso) o *tc, typical case* (caso típico).

- **Se lee el diseño,** siendo los archivos RTL en Verilog obtenidos en la síntesis de alto nivel.
- **Se define el reloj.**
- **Se especifica el entorno de diseño** en el que va a operar, como el modelo de carga debido al interconexionado.
- **Se definen las restricciones de diseño,** por ejemplo el área máxima. Con las restricciones de diseño se le indica a la herramienta los objetivos de optimización.
- **Se realiza la síntesis.** Para compilar se pueden usar diferentes esfuerzos, dependiendo de la precisión que se quiera obtener. En este caso se deja la opción por defecto: esfuerzo medio, compromiso entre calidad y tiempo de optimización.
- **Se obtienen los resultados.** Como se puede apreciar en la Tabla 4.9, no se presentan variaciones significativas según el caso, ya que el tiempo de ciclo se mantiene, siendo aproximadamente igual a 10 ns; y el área está en torno a los 114 μm. La potencia en ninguno de los casos llega a 1 W.

	Área (μm^2)	**Potencia (mW)**	**Tiempo de ciclo (ns)**
Caso Típico	114.169,0695	47,0771	9,95
Mejor Caso	114.106,6095	102,7504	9,97
Peor Caso	113.876,9295	40,0152	9,96

Tabla 4.9: ASIC. Resultados obtenidos en la síntesis lógica. Diseño con *wrapper*

En la tabla anterior se puede observar un incremento de aproximadamente 60 mW para el mejor caso respecto a los otros dos, que se corresponde con la menor frecuencia obtenida (100,3 MHz). Esto se debe a la potencia de pérdidas, ya que un aumento de la tensión de alimentación eleva significativamente el valor de las corrientes de fuga del transistor, y por consiguiente la potencia disipada por pérdidas.

	Potencia dinámica (mW)	**Potencia de pérdidas (mW)**
Caso Típico	41,021	6,056
Mejor Caso	53,008	49,709
Peor Caso	32,028	7,963

Tabla 4.10: Potencia dinámica y de pérdidas. Diseño con *wrapper*

4.5.4.2. Síntesis lógica para FPGA

El flujo de diseño de Synplify Premier permite obtener la implementación de la FPGA desde la entrada RTL en Verilog. Las etapas del mismo se explican a continuación:

- **Se introducen los ficheros fuente,** que serán los archivos RTL de la microarquitectura en Verilog obtenidos tras la síntesis de alto nivel.
- **Se especifican los criterios de implementación,** como la FPGA usada: Virtex5, Spartan6 o Zynq de la familia Xilinx. Además, se activan las siguientes opciones:
  - *Disable I/O Insertion,* impide la inserción de bloques de entrada/salida de la FPGA para las señales de entrada y salida del *top* del diseño. Esto se realiza cuando se está trabajando con bloques internos del diseño que permiten generar IPs para ser implementados en flujos más avanzados de la FPGA.
  - *FSM Compiler,* optimiza las máquinas de estado siguiendo una estrategia diferente de codificación de estados en función del número de estos.
  - *Resource Sharing,* comparte recursos con el fin de reducir el consumo de los recursos del dispositivo.
  - *Pipelining,* permite que varias operaciones se realicen a la vez sobre el mismo recurso, partiendo dicha ejecución en etapas, donde cada dato se encuentra en una de ellas.
  - *Enable Advanced LUT Combining,* prepara el *netlist* (fichero que describe la conectividad del diseño) de salida para una posterior combinación de LUTs en los diseños.
- **Se añaden las restricciones de tiempo** si las hubiera. En este caso se deja la opción por defecto, automático, ya que optimiza el diseño hasta obtener la máxima frecuencia.
- **Se genera la síntesis.**
- **Se obtienen los resultados.** El valor de la frecuencia de trabajo se supera tanto para la Virtex5 como para la Zynq, pues son FPGAs más avanzadas y por tanto, ofrecen mejores prestaciones. Sin embargo, la Spartan6 no llega a dicha frecuencia, así como en cuanto a la estimación de la ocupación del diseño en el dispositivo sobrepasa sus límites.

	Registros	BRAMs	LUTs	DSPs	Frecuencia (MHz)
Virtex5	63.287	11	71.780	14	122,6
Zynq	63.061	11	99.390	3	140,4
Spartan6	63.135	12	66.997	12	84,0

Tabla 4.11: FPGA. Resultados obtenidos en la síntesis lógica. Diseño con *wrapper*

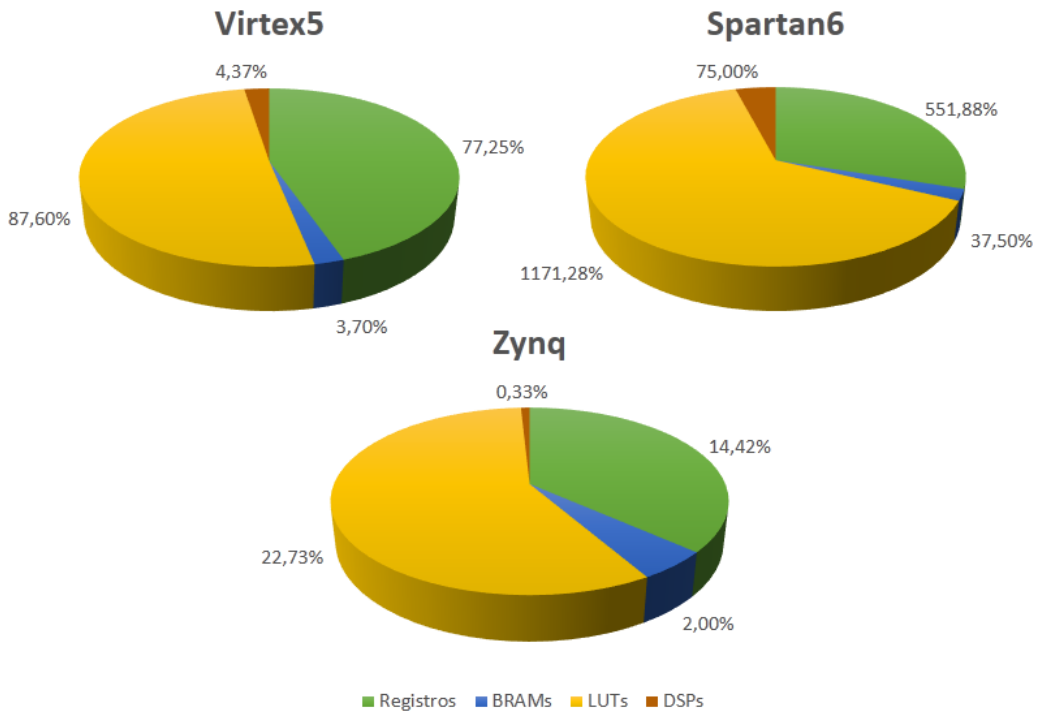


Figura 4.32: FPGA. Ocupación del diseño con *wrapper* tras la síntesis lógica

Synplify Premiere ofrece la posibilidad de representar los diseños en dos vistas, RTL y tecnológica. En la primera no se muestran componentes específicos, sino genéricos como RAMs, puertas lógicas o *flip-flops*; mientras que en la segunda son las células básicas de la FPGA con la que se implementa el diseño. Esta vista, además, permite ver la ruta crítica.

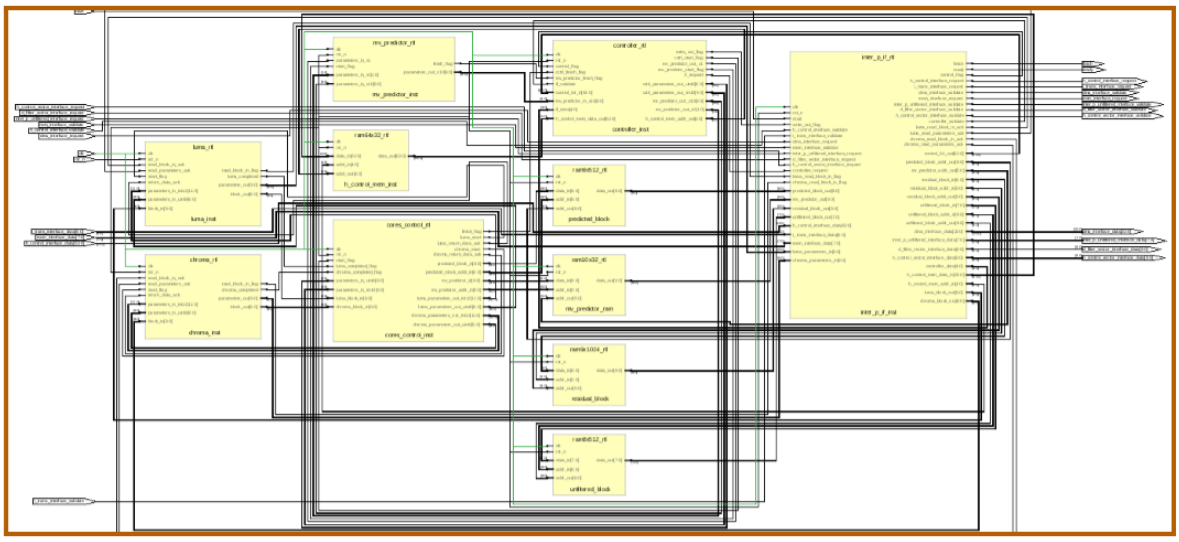


Figura 4.33: Vista RTL. Diseño original

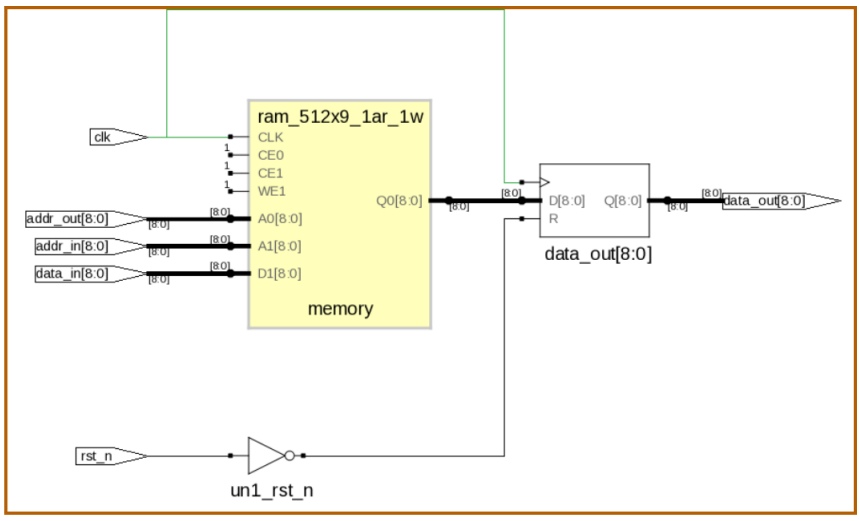


Figura 4.34: Esquema de la memoria

4.6. Implementación en FPGA

El siguiente paso es realizar la implementación del diseño sobre las FPGAs elegidas. Se utilizarán las herramientas avanzadas de Xilinx, tanto PlanAhead como Vivado.

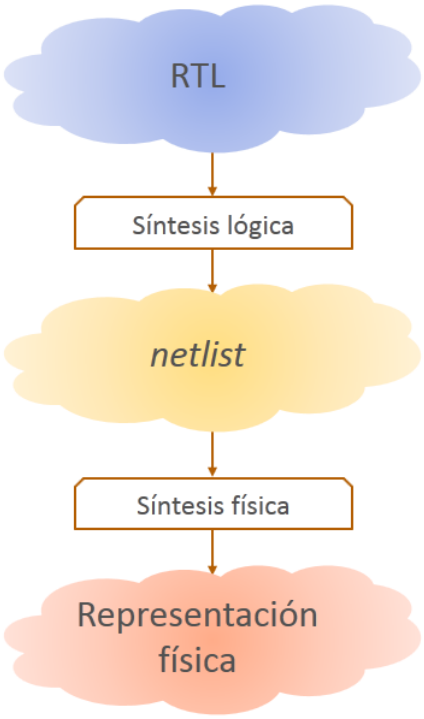


Figura 4.35: Flujo de diseño: últimas etapas

Para la implementación del sistema en la FPGA Zynq se utilizó el entorno de trabajo Vivado, que como ya se comentó en la presentación de las herrramientas en el Capítulo 3, en este proyecto este se usa bajo la interfaz de usuario de Synplify Premier. Una vez completada la síntesis lógica, tan solo se debe añadir la opción de *Place & Route* al diseño sintetizado y se genera la síntesis física.

Mientras que para las FPGAs Spartan6 y Virtex5 PlanAhead de Xilinx es el entorno de diseño, que a partir de un fichero de tipo *netlist* implementa el diseño en el dispositivo que se le indique, proporcionando informes de los recursos utilizados así como de la frecuencia de funcionamiento.

El flujo de diseño es el siguiente:

- **Se crea el proyecto.** Se indica la etapa de diseño desde la que se parte, siendo en este caso de ficheros que ya están sintetizados (EDIF); y se selecciona el dispositivo para trabajar.

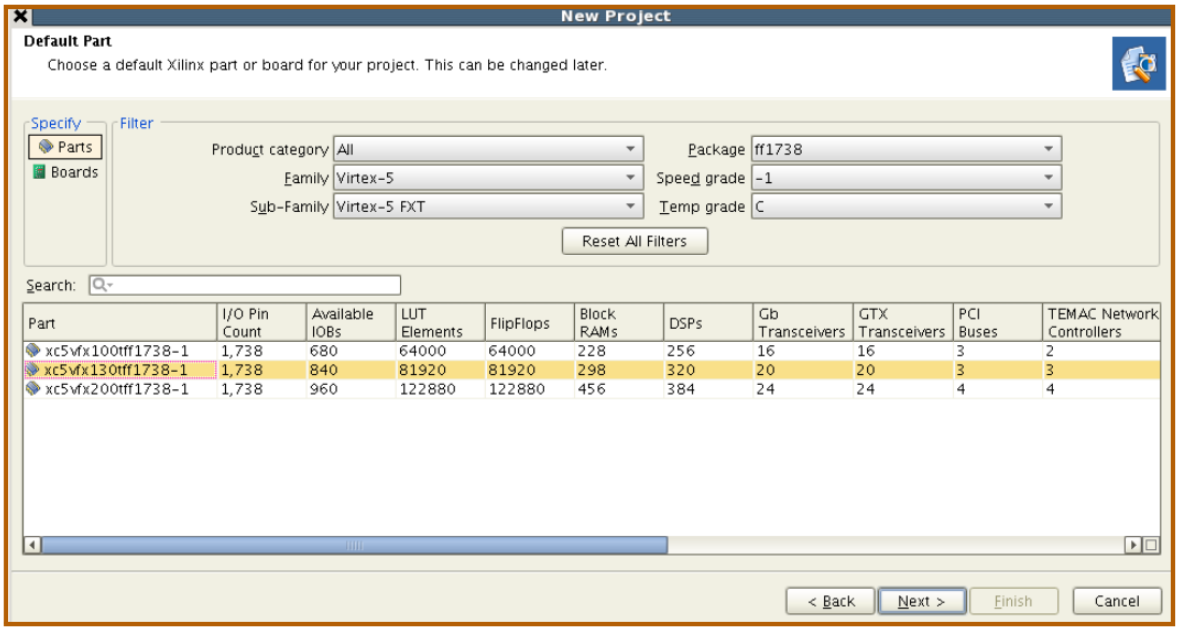


Figura 4.36: Selección del dispositivo. Ejemplo Virtex5

- **Se añade el fichero fuente (EDIF)** obtenido en Synopsys Synplify Premier.
- **Se introduce el fichero de restricciones del usuario (UCF),** también generado por Synopsys Synplify Premier.
- **Se elige la estrategia.** PlanAhead ofrece la posibilidad de lanzar varias implementaciones con diferentes estrategias para luego poder optar por aquella que dé un mejor comportamiento temporal. Además, permite poder ejecutar en paralelo dichas estrategias, reduciendo así los tiempos de síntesis.

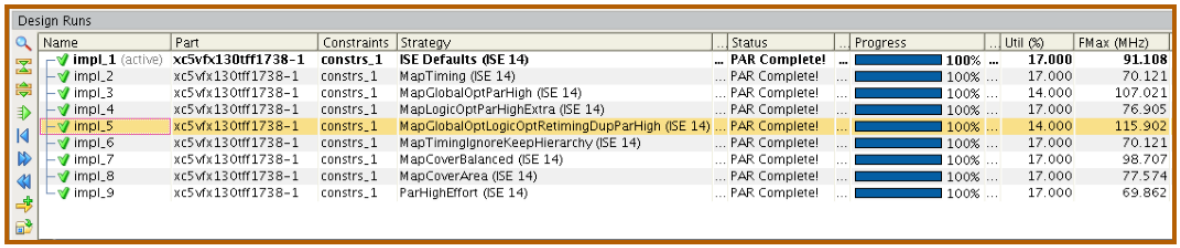


Figura 4.37: Elección de la estrategia

- **Se genera la síntesis.**
- **Se obtienen los resultados.** Con el fin de poder tener una mejor visión de estos también se recogen los datos conseguidos con Vivado.

Mientras que para la Spartan6 no se puede realizar la implementación, para las otras dos FPGAs los valores estimados en la síntesis lógica se mantienen, solo desciende el número de BRAMs finalmente implementadas. En cuanto a la frecuencia de funcionamiento desciende para el caso del dispositivo Virtex5 y aumenta para la Zynq.

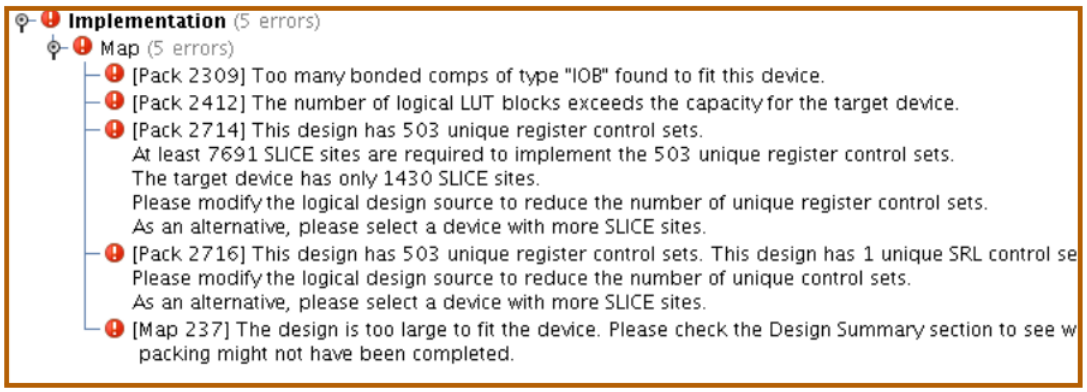


Figura 4.38: Spartan6. Error de implementación

Para obtener la potencia y poder realizar un análisis de la misma de un diseño implementado, por un lado para la FPGA Virtex5, PlanAhead cuenta con la herramienta XPower Analyzer (XPA). Mientras que para la FPGA Zynq, Xilinx pone a disposición del diseñador Xilinx Power Estimator (XPE).

	Registros	**BRAMs**	**LUTs**	**DSPs**	**Frecuencia (MHz)**	**Potencia (W)**
Virtex5	63.287	7	71.829	14	72,595	4,407
Zynq	63.061	6	99.195	3	165,399	1,782

Tabla 4.12: FPGA. Resultados obtenidos en la síntesis física. Diseño con *wrapper*

Esta herramienta permite visualizar el dispositivo y resalta con otro color el diseño, de manera que se pueda observar la ubicación del mismo.

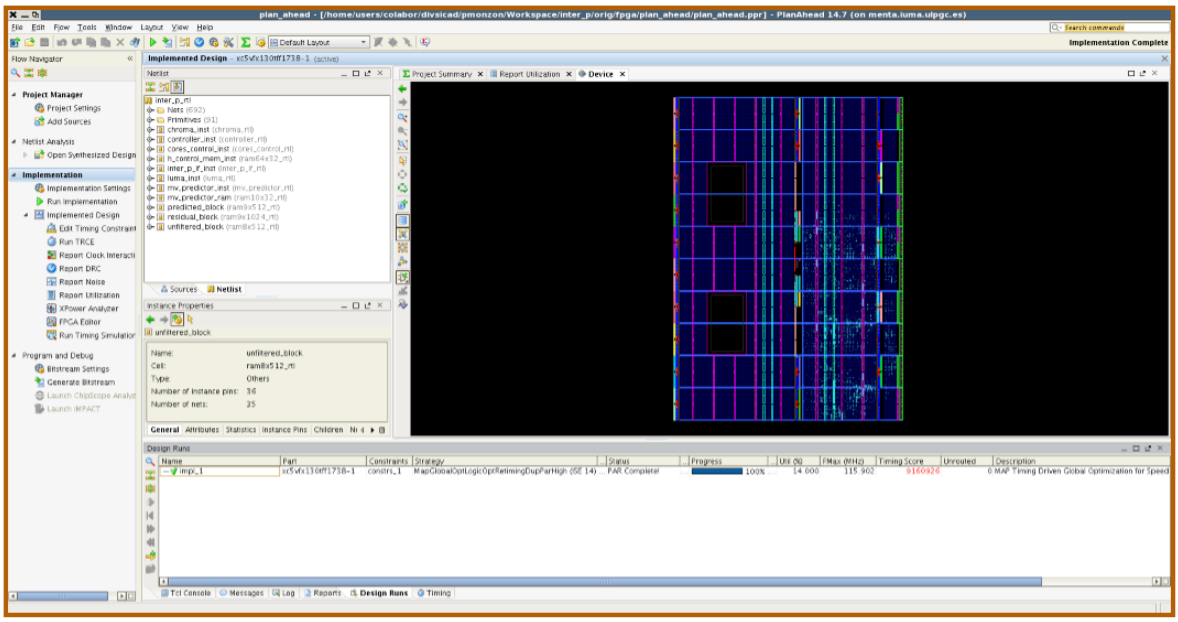


Figura 4.39: Visualización del dispositivo

En la Figura 4.40 la zona coloreada en rojo es el sistema implementado en la FPGA Virtex5.

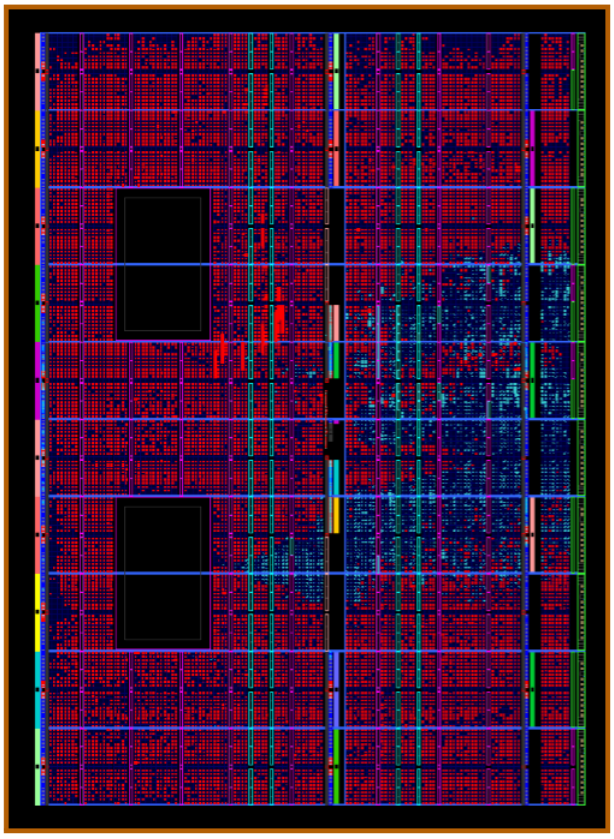


Figura 4.40: Virtex5. Ocupación del diseño

Llama la atención el **alto consumo de los recursos por parte del diseño**, teniendo en cuenta que se trata de un único bloque del decodificador de vídeo H.264/AVC.

La herramienta de síntesis lógica, Synopsys Synplify Premier, genera un informe del área del dispositivo en función de los diferentes módulos que componen el sistema, como muestra la Figura 4.41. Se puede observar que el diseño sin modelar consume 16.886 LUTs de los 71.780 que ocupa el sistema modelado, habiendo una diferencia de 54.894 LUTs, pertenecientes al *wrapper*. Ocurre lo mismo con los registros: el diseño original ocupa 7.492, mientras que el modelo TLM 63.287, siendo la diferencia de 55.795 registros.

Area Summary : Hierarchical Area report

Module name	LUTS	REGISTERS	SYNC RAMS	DSP48
inter_p_wrapper_rtl	71780	63287	11	14
inter_p_rtl	16886	7492	11	14
chroma_rtl	4919	1516	2	7
controller_rtl	5864	3205	1	0
cores_control_rtl	437	238	0	0
inter_p_if_rtl	515	504	0	0
luma_rtl	4367	1549	3	7
mv_predictor_rtl	784	480	0	0
ram10x32_rtl	0	0	1	0
ram64x32_rtl	0	0	1	0
ram8x512_rtl	0	0	1	0
ram9x1024_rtl	0	0	1	0
ram9x512_rtl	0	0	1	0

Figura 4.41: Virtex5. Informe del área de manera jerárquica con Synplify

La razón de esta situación se debe a que el *wrapper* genera unas memorias que en la asignación de IPs del CtoS, por dependencias, se mapean en registros en lugar de en BRAMs.

Analizando el problema planteado, como se explicó en el Capítulo 4, lo que envía el *testbench* el bloque *inter_p* espera recibirlo, y viceversa. Al introducir el *wrapper*, este debe almacenar cada una de las memorias que conforman las interfaces del modelo de referencia, ya que **por un lado tiene que sincronizarse con el diseño original, y por otro, adaptarse al modelado TLM**.

Los datos que se transmiten son:

- 11 palabras de 64 bits (`h_control_interface_data`).
- 256 palabras de 9 bits (`i_trans_interface_data`).

- 1.024 palabras de 8 bits (`mem_interface_data`).
- 1 palabra de 34 bits (`dma_interface_data`).
- 258 palabras de 8 bits (`inter_p_unfiltered_interface_data`).
- 34 palabras de 10 bits (`d_filter_vector_interface_data`).
- 18 palabras de 10 bits (`h_control_vector_interface_data`).

Por ejemplo, el *inter_p_wrapper* está a la espera de recibir las 1.024 palabras correspondientes a `mem_interface_data`. Sin embargo, el *tb_wrapper* no puede transmitirlas hasta que el protocolo de comunicación con el *testbench* comience, y una vez se ha iniciado, debe almacenar las 1.024 palabras para luego enviarlas vía TLM. Esto explica el alto consumo de recursos, pues en la asignación de IPs en la síntesis de alto nivel, por dependencias de datos, las 1.024 palabras se mapean en registros. Lo mismo ocurre con el resto de memorias.

El problema es la existencia de conflicto entre los dos protocolos de comunicación, ya que la sincronización entre ellos es inexistente, debido a que el protocolo que implementa TLM es interno y no visible al diseñador.

4.7. Conclusiones

Partiendo de un código desarrollado en SystemC se plantea la implementación TLM a través de un *wrapper*. Cuando los módulos funcionan con un protocolo de comunicación propio este deberá ser incluido en el nuevo diseño, a la par que la interfaz TLM de cara a la comunicación externa.

El modelado a nivel de transacciones supone un mayor nivel de abstracción. Lo esencial son las interfaces y los canales, ya que él mismo se encarga de generar el protocolo de comunicación entre los diferentes bloques a conectar. Esta es la clave de esta metodología y lo que ofrece una larga lista de ventajas.

Con la simulación se puede comprobar que después del modelado TLM el diseño mantiene su funcionalidad, y esta se ejecuta correctamente.

Una vez se tiene el diseño modelado se entra en la fase donde las herramientas son quiénes cobran mayor importancia, pues se introducen los archivos fuente y son ellas quiénes ofrecen resultados de calidad. Si bien se debe conocer las características del diseño y el fin del mismo, pues en base a las configuraciones (restricciones, condiciones, etc.) que el propio diseñador introduzca, la herramienta operará de una manera u otra.

Con las herramientas de síntesis lógica se pueden conseguir resultados cuantitativos: cuánta potencia, qué área, cuál es la frecuencia de funcionamiento, qué recursos utiliza, etc. y así tener una visión real del diseño en el dispositivo. Mientras que con el CtoS se obtienen los modelos RTL, siendo estos el objetivo principal de este proyecto.

Por otro lado, los resultados sobre la FPGA demuestran un alto consumo de los recursos (BRAMs y registros) del *wrapper* debido a que la asignación de IPs en la síntesis de alto nivel es ineficiente, lo que hace desestimable la estrategia de crear un *wrapper* TLM para este tipo de diseños.

Capítulo 5

Modelado TLM de la interfaz

5.1. Introducción

Las diferentes etapas del flujo de diseño brindan resultados que permiten analizar el comportamiento y la caracterización del sistema. Sin embargo, como se comentó en el Capítulo 3, el desarrollo e implementación de un diseño es un proceso interactivo en el que se va refinando el sistema hasta obtener la versión final, por lo que en muchas ocasiones se deben plantear soluciones alternativas con diferentes tipos de optimizaciones.

Por lo que en este capítulo se describe la segunda línea de trabajo planteada: modificar la aplicación de referencia y desarrollar las interfaces modeladas a nivel de transacciones, tanto para el bloque *inter_p* como para el *testbench*, de manera que el sistema resultante quedaría como se puede ver en la Figura 5.1.

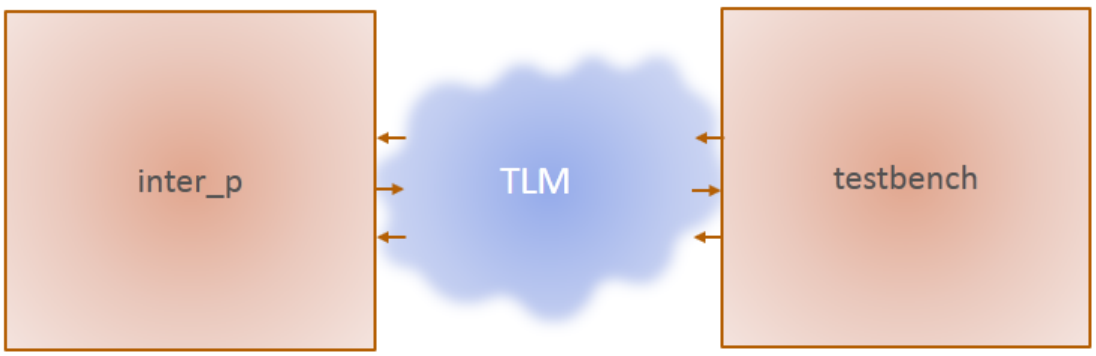


Figura 5.1: Sistema tras la implementación TLM

5.2. Modelado a nivel de transacciones

El esquema de ficheros se mantiene como el diagrama inicial (Figura 4.1), cambiando el *sc_main* por el usado en el *wrapper*.

```
sc_clock clk (“clk”, 10, SC_NS, 0.5, 0.0, SC_PS);
sc_signal<bool>rst_n;

sc_signal<bool>start;
sc_signal<bool>finish;
sc_signal<bool>ready;
...

// Instantiate FIFOs
tlm_fifo< sc_uint<64> >h_control_fifo(“h_control_fifo”);
tlm_fifo< sc_int<9> >i_trans_fifo(“i_trans_fifo”);
tlm_fifo< sc_uint<8> >unfiltered_fifo(“unfiltered_fifo”);
tlm_fifo< sc_int<10> >h_control_vector_fifo(“h_control_vector_fifo”);
tlm_fifo< sc_int<10> >d_filter_vector_fifo(“d_filter_vector_fifo”);
tlm_fifo< sc_uint<34> >dma_fifo(“dma_fifo”);
tlm_fifo< sc_uint<8> >mem_fifo(“mem_fifo”);
```

Tanto la declaración de las interfaces y los puertos como la definición de las funciones y los procesos del *wrapper* se trasladan a los ficheros *inter_p_if.h* y *testbench.h*.

Por último, para implementar TLM se reescriben las interfaces incluidas en los archivos de definición *testbench.cpp* y *inter_p_if.cpp*, donde se pasa de `read()` y `write()` a `get()` y `put()` de TLM, respectivamente; y **se establece un protocolo de comunicación a nivel puramente TLM, sin necesidad del *handshake* de señalización** indicado en capítulos anteriores.

A continuación se muestra el código correspondiente a la interfaz (`h_control_interface_data`):

```
while (h_control_interface_validate.read() == true){

     write_h_control_word(k, h_control_interface_data.read());
     k++;
     wait();
}
```

```
for (int i=0; i<11; i++){

     h_control_data = h_contro_data_tlm->get();
     write_h_control_word(k, h_control_data);
     k++;
     wait();
}
```

5.3. Verificación funcional

Para comprobar el correcto funcionamiento del sistema tras el modelado TLM se realiza la verificación funcional, simulando el diseño con la herramienta de Cadence, SimVision, Figura 5.2.

Como el *wrapper* introducía un retardo en el sistema, se han añadido dos marcadores (*TimeA* y *TimeB*) con el fin de ver qué ocurre con el mismo tras la implementación TLM del módulo *inter_p*. Comparando la Figura 5.3 con la simulación del sistema original, no solo no introduce retardo sino que **el sistema es más rápido que el diseño de referencia.**

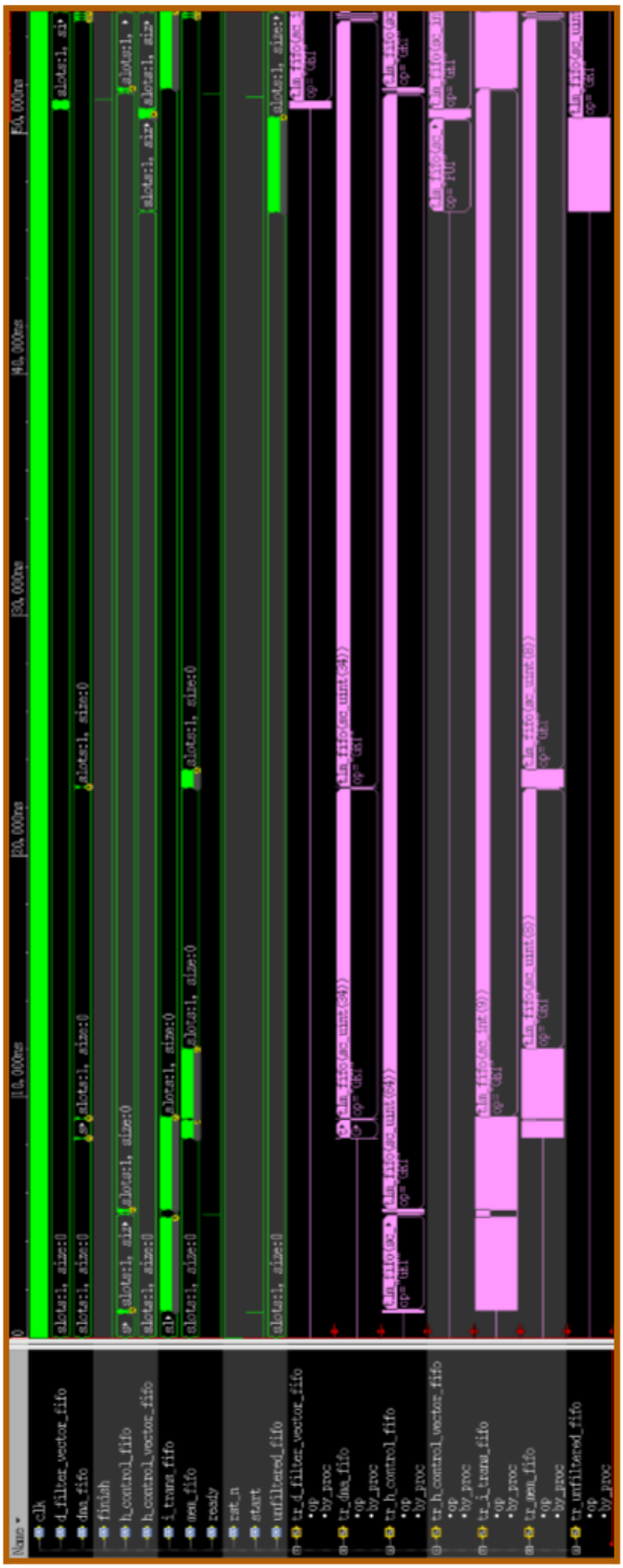


Figura 5.2: Verificación funcional tras la implementación TLM (1)

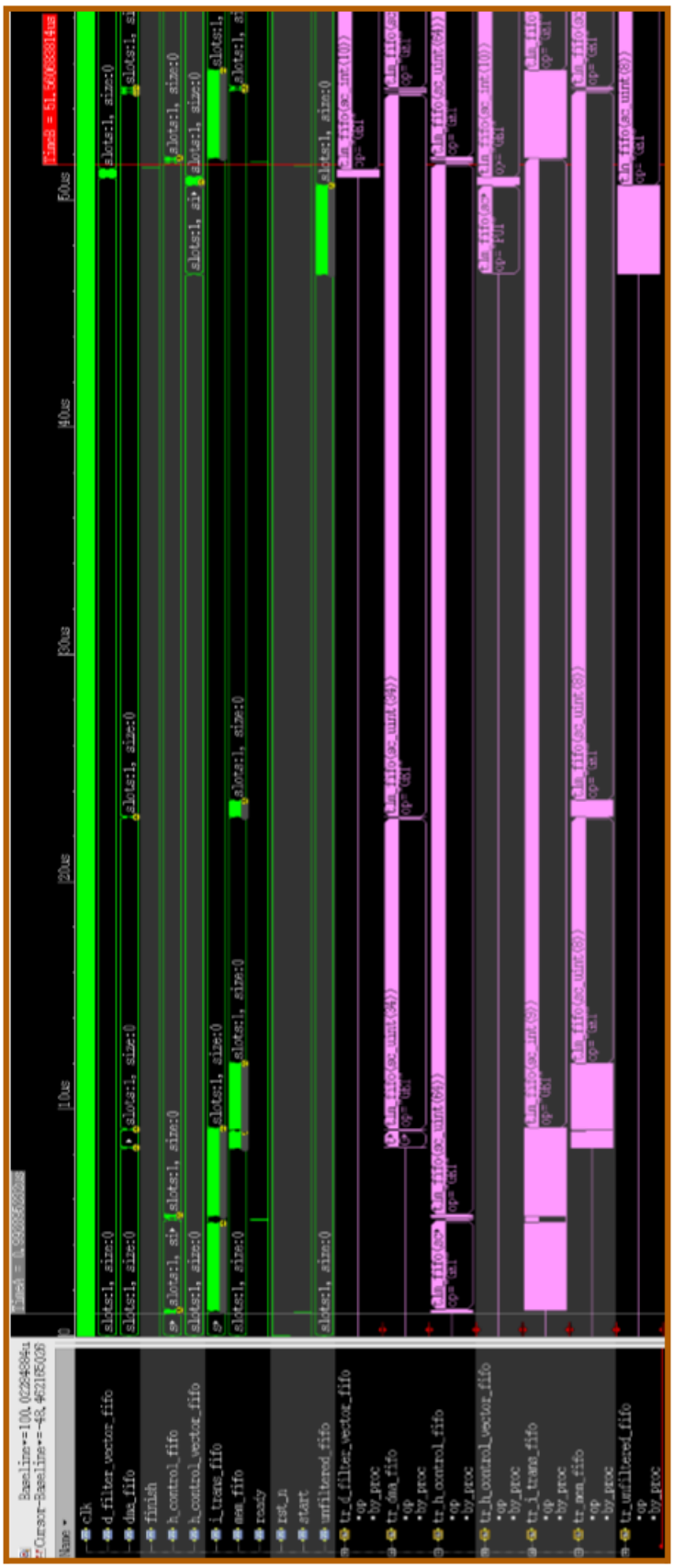


Figura 5.3: Verificación funcional tras la implementación TLM (2)

5.4. Síntesis

Una vez modelado el sistema, y tras haber comprobado que **es funcionalmente correcto**, se procede a efectuar la síntesis, donde se mantendrán las consideraciones previas de la misma del Capítulo 4.

5.4.1. Síntesis de alto nivel

Nuevamente, para la síntesis de alto nivel se debe añadir una interfaz de *reset* en todos los procesos:

```
SC_THREAD (memory_process);
sensitive «clk.pos();
reset_signal_is (rst_n, false);


void inter_p_wrapper::memory_process(){
// Reset
#if defined(__CTOS__) || defined(CTOS_MODEL)
dma_data_tlm->reset_put();
mem_data_tlm->reset_put();
#endif
...
```

Las configuraciones, resoluciones, etc. indicadas en el flujo de diseño del CtoS son iguales que para el *wrapper*.

Una vez se obtienen los modelos sintetizables se pueden realizar los primeros análisis. En las siguientes imágenes se puede observar un *slack* positivo para los dos dispositivos, ASIC y FPGA, lo que implica que **la frecuencia de trabajo propuesta (100 MHz) es adecuada para el diseño en ambos casos,** mientras que para la descripción basada en *wrapper* se obtiene un *slack* negativo para FPGA habiendo usado la misma frecuencia.

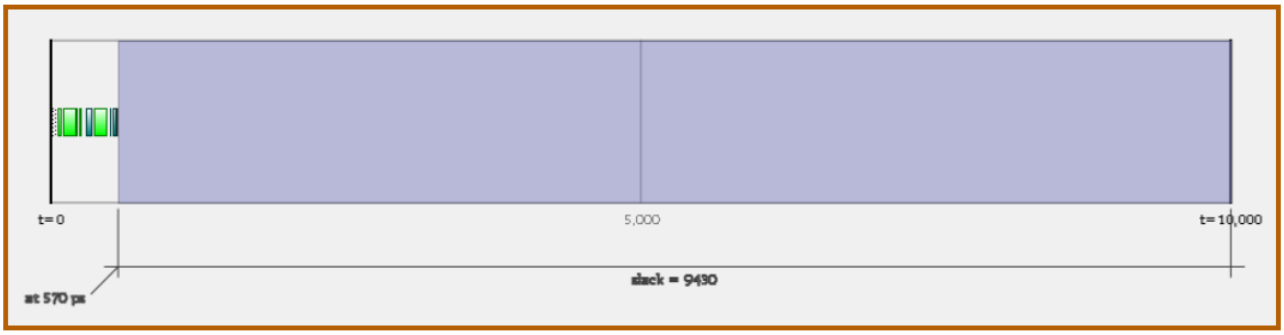


Figura 5.4: ASIC. Análisis de ciclos

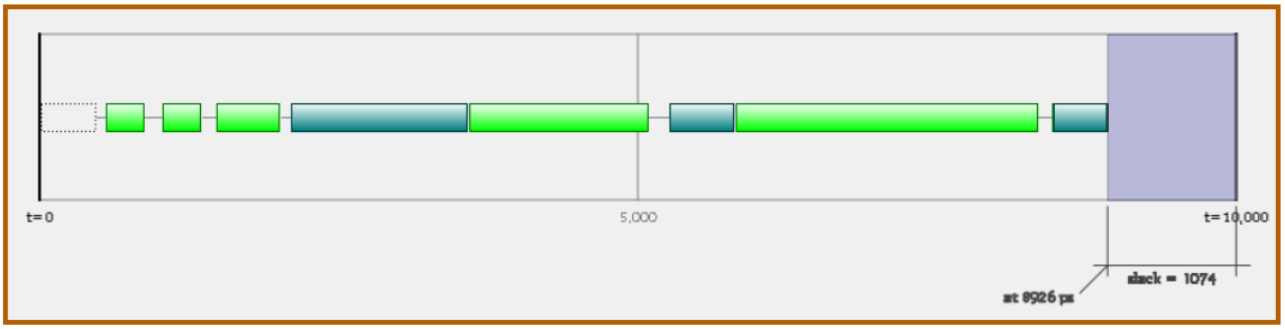


Figura 5.5: FPGA. Análisis de ciclos

La funcionalidad de los modelos sintetizables obtenidos debe continuar siendo la misma que la del diseño original. Por lo que en este punto habrá que efectuar la verificación RTL.

En las siguientes figuras se muestra una captura de las simulaciones llevadas a cabo tanto para ASIC como para FPGA. Comparándolas con la del diseño de referencia **se puede verificar que el modelo es funcionalmente correcto.**

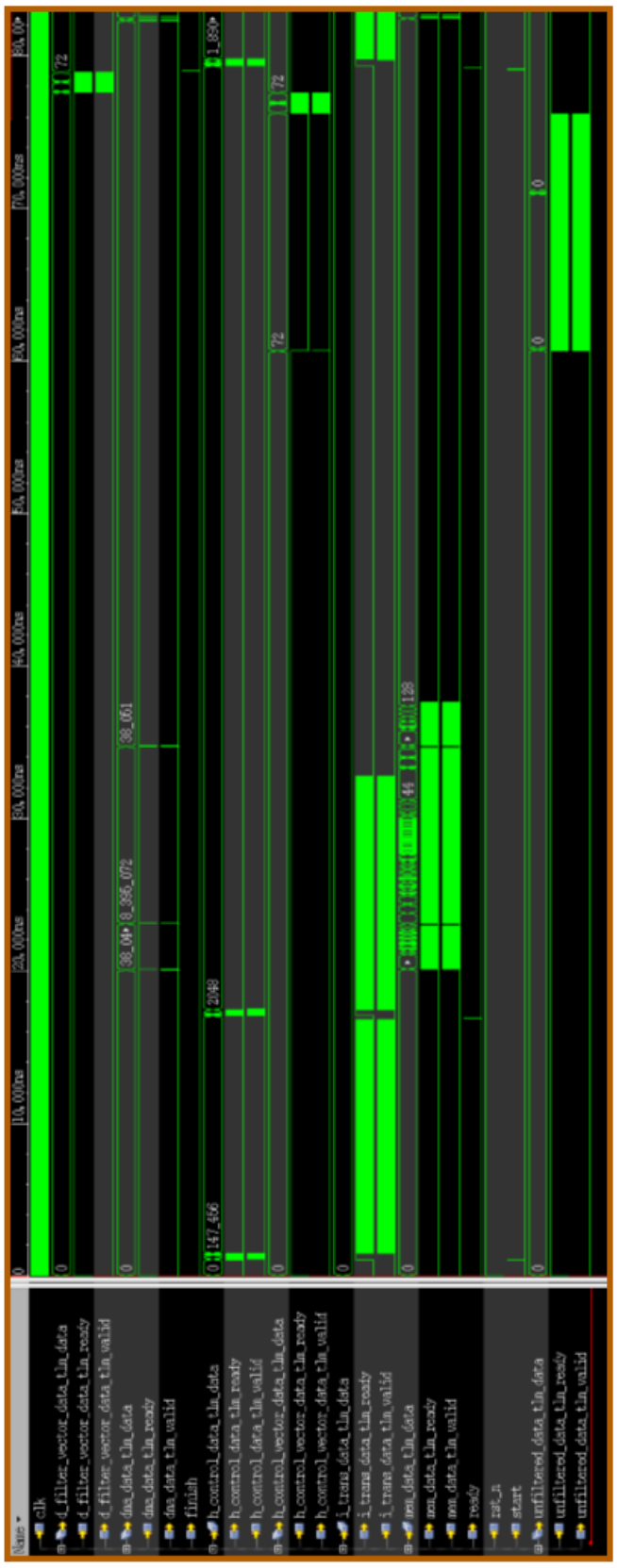


Figura 5.6: ASIC. Verificación RTL

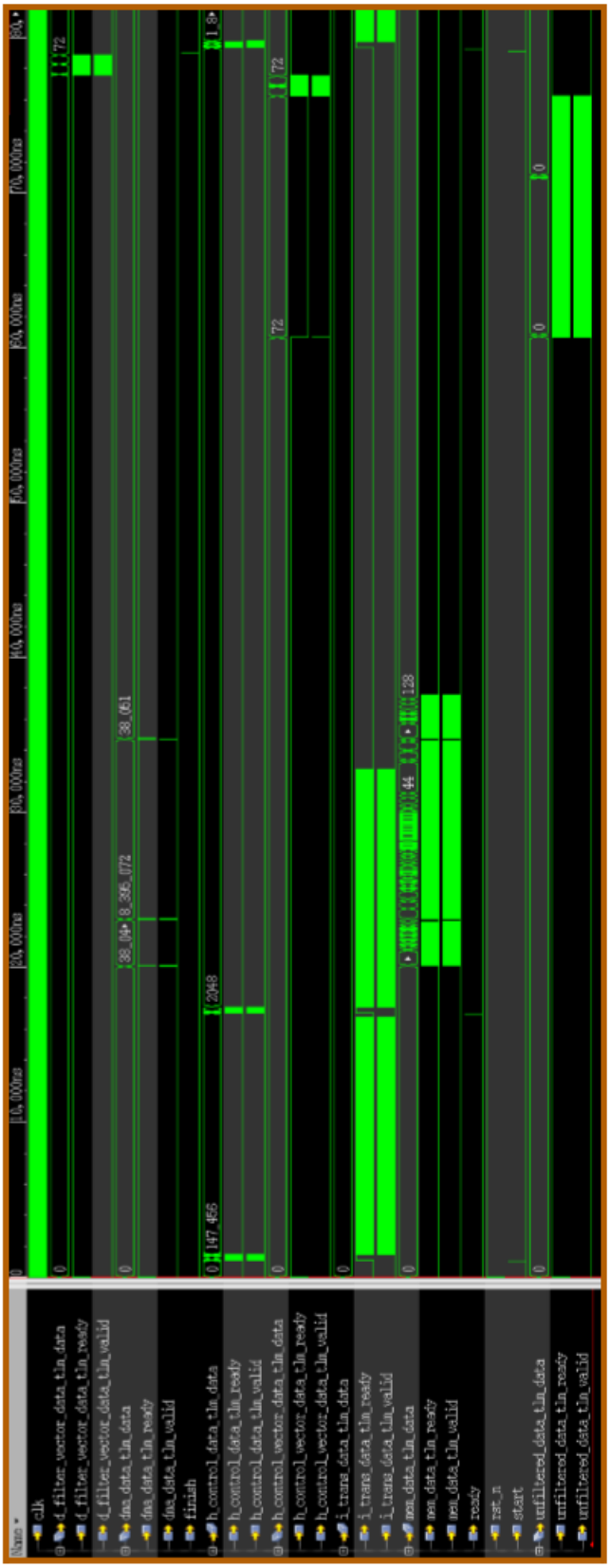


Figura 5.7: FPGA. Verificación RTL

5.4.2. Síntesis lógica

El flujo de diseño utilizado, tanto para el ASIC como para FPGA, es el mismo que se ha explicado con anterioridad, manteniendo las mismas restricciones.

En el caso del ASIC, el tiempo de ciclo es de 10 ns y el área disminuye, siendo de 80 μm. La potencia también se reduce, continuando por debajo de 1 W; y se mantiene la mayor disipación de potencia con la menor frecuencia obtenida.

	Área (μm^2)	**Potencia (mW)**	**Tiempo de ciclo (ns)**
Caso Típico	80.442,613	37,598	9,95
Mejor Caso	80.371,045	81,578	9,99
Peor Caso	80.368,633	31,809	9,96

Tabla 5.1: ASIC. Resultados obtenidos en la síntesis lógica. Diseño TLM

	Potencia dinámica (mW)	**Potencia de pérdidas (mW)**
Caso Típico	32,949	4,615
Mejor Caso	42,571	38,994
Peor Caso	25,798	5,994

Tabla 5.2: Potencia dinámica y de pérdidas. Diseño TLM

Por otro lado, el número de registros y LUTs utilizados en las FPGAs presenta mejores resultados, debidos a la eliminación de la necesidad de almacenar los datos localmente. La frecuencia de trabajo es mayor a la propuesta tanto para la Virtex5 como para la Zynq, mientras que la Spartan6 no llega a dicha frecuencia y además, el límite de LUTs lo vuelve a superar.

	Registros	BRAMs	LUTs	DSPs	Frecuencia (MHz)
Virtex5	8.180	11	14.783	14	131,1
Zynq	7.585	11	15.937	3	130,9
Spartan6	7.558	12	14.613	12	84,2

Tabla 5.3: FPGA. Resultados obtenidos en la síntesis lógica. Diseño TLM

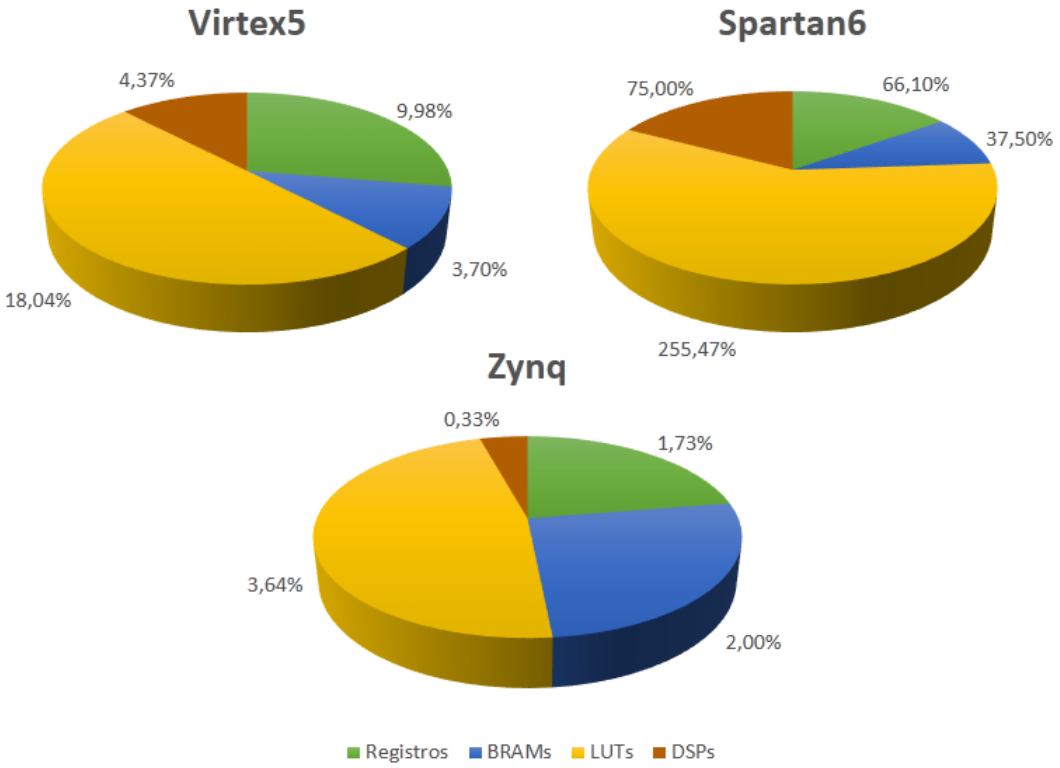


Figura 5.8: Ocupación del diseño TLM

5.5. Implementación en FPGA

Las especificaciones del flujo de diseño seguido por PlanAhead y Vivado se mantienen.

	Registros	BRAMs	LUTs	DSPs	Frecuencia (MHz)	Potencia (W)
Virtex5	7.998	7	12.624	14	115,354	2,932
Zynq	7.585	6	15.831	3	153,420	0,540

Tabla 5.4: FPGA. Resultados obtenidos en la síntesis física. Diseño TLM

En la Figura 5.9 la zona en una tonalidad azul claro es el sistema implementado en la FPGA Virtex5, donde se puede observar que la ocupación del diseño en el dispositivo ha disminuido considerablemente en comparación con la Figura 4.40.

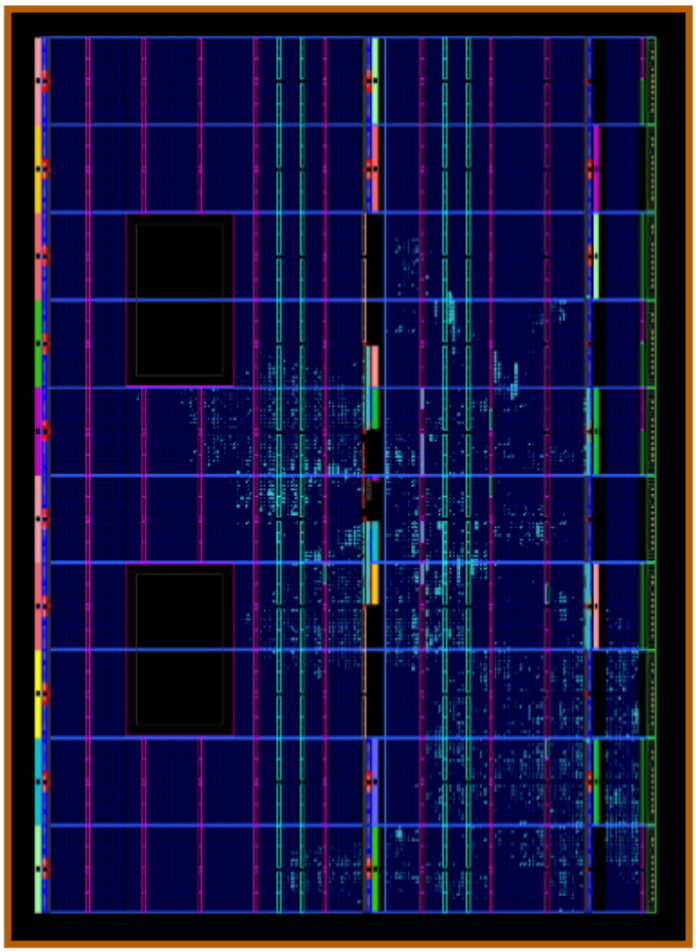


Figura 5.9: Virtex5. Ocupación del diseño TLM

5.6. Conclusiones

A la hora de crear modelos TLM es necesario tener en cuenta las características de dicha metodología, pues pueden aparecer incompatibilidades con los protocolos de comunicación ya implementados.

Esta solución **produce un impacto positivo en las prestaciones del sistema**. Además de obtenerse mejores resultados en cuanto a recursos, frecuencia, área, etc., el sistema es más rápido que el modelo original, cumpliendo en todo momento con la funcionalidad.

Capítulo 6

Comparativa de los resultados

6.1. Introducción

Este Proyecto Fin de Carrera partía de un código en SystemC del bloque *inter_p* del decodificador de vídeo H.264/AVC, para el cual se proponía su modelado a nivel de transacciones usando un *wrapper* y la implementación de las interfaces del diseño en TLM. Por tanto, en los siguientes apartados se pesenta una comparativa de ambos diseños.

Además, se ha seguido el flujo de diseño propuesto con el bloque original y así tenerlo como referencia.

6.2. Comparativa ASIC

	Área (μm^2)	**Potencia (mW)**	**Tiempo de ciclo (ns)**
Caso Típico	80.295,337	37,615	9,95
Mejor Caso	80.191,621	81,598	9,97
Peor Caso	80.058,241	31,794	9,96

Tabla 6.1: ASIC. Resultados obtenidos en la síntesis lógica. Diseño original

Con el sistema modelado directamente en TLM se obtienen mejores resultados que los conseguidos con el *wrapper*, y se mantienen prácticamente iguales que los del diseño original.

	Potencia dinámica (mW)	Potencia de pérdidas (mW)
Caso Típico	33,007	4,608
Mejor Caso	42,624	38,947
Peor Caso	25,799	5,979

Tabla 6.2: Potencia dinámica y de pérdidas. Diseño original

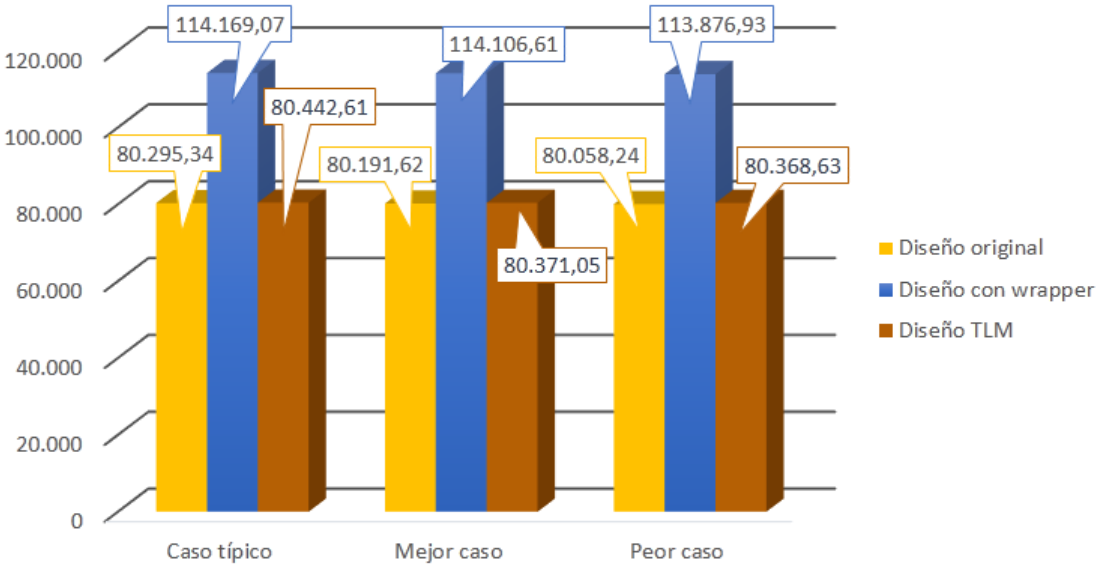


Figura 6.1: Comparativa área (μm^2)

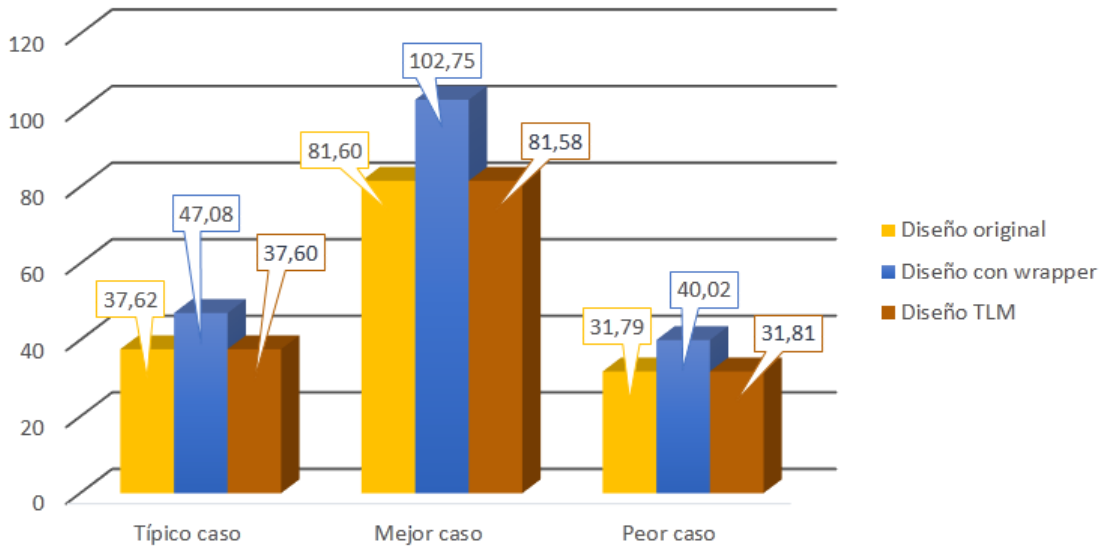


Figura 6.2: Comparativa potencia (mW)

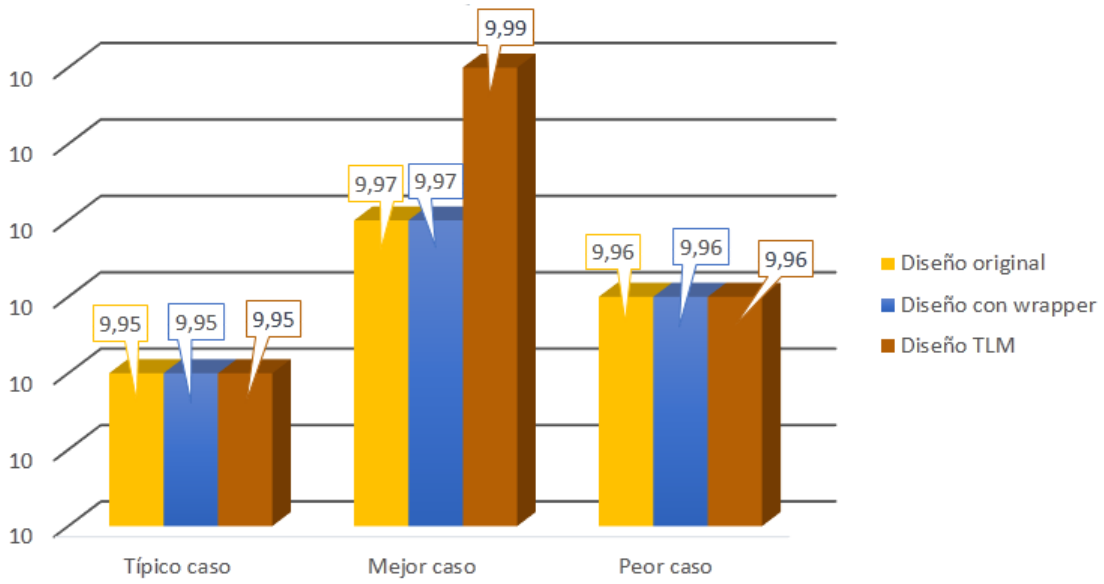


Figura 6.3: Comparativa tiempo de ciclo (ns)

6.3. Comparativa FPGA

	Registros	BRAMs	LUTs	DSPs	Frecuencia (MHz)
Virtex5	7.730	11	14.072	14	129,9
Zynq	7.285	11	15.171	3	128,4
Spartan6	7.215	12	13.831	12	87,2

Tabla 6.3: FPGA. Resultados obtenidos en la síntesis lógica. Diseño original

En este caso también el número de LUTs usados en la Spartan6 sobrepasa su límite y por tanto, no se puede realizar la implementación del diseño en ella.

	Registros	BRAMs	LUTs	DSPs	Frecuencia (MHz)	Potencia (W)
Virtex5	7.534	7	15.968	14	115,902	2,845
Zynq	7.285	6	15.064	3	151,194	0,511

Tabla 6.4: FPGA. Resultados obtenidos en la síntesis física. Diseño original

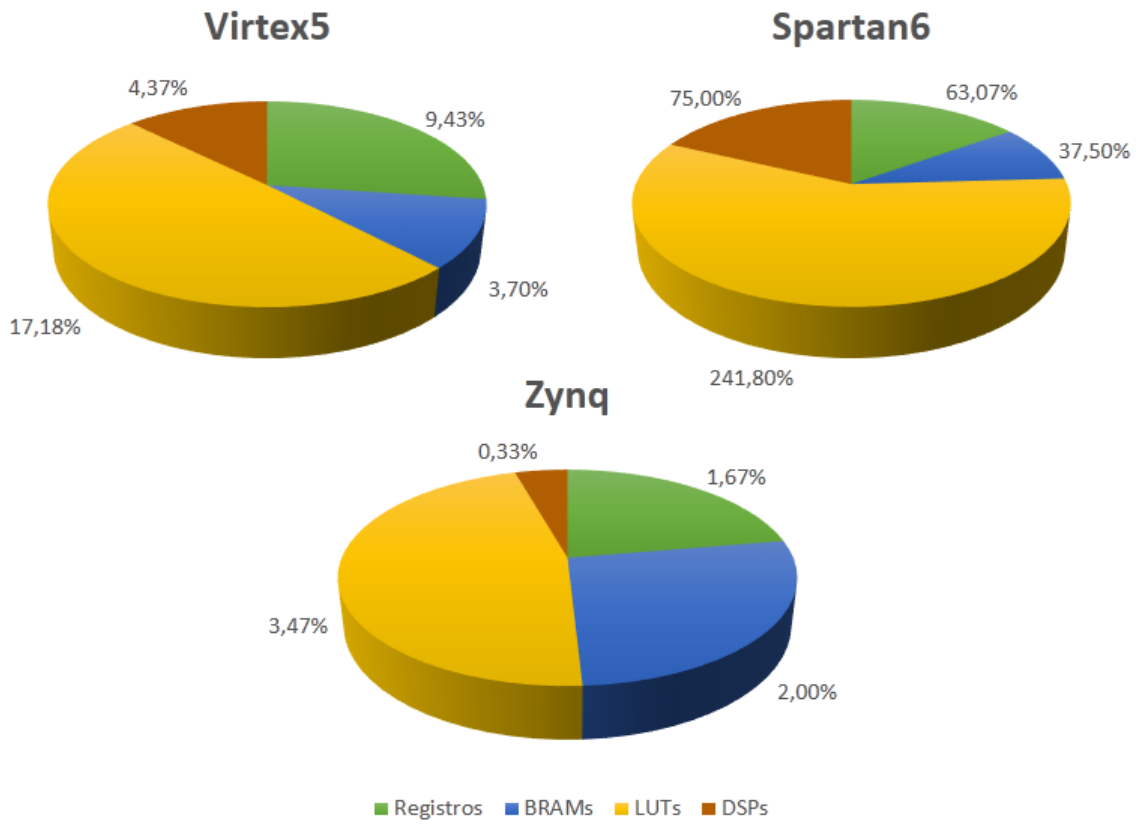


Figura 6.4: Ocupación del diseño original

A continuación se presentan las gráficas con los resultados obtenidos en la síntesis lógica.

Se puede observar que independientemente del dispositivo utilizado, el diseño que implementa el *wrapper* consume un alto número de recursos de la FPGA, concretamente registros y LUTs, debido al mapeado de las memorias en registros en la asignación de IPs en la síntesis de alto nivel. En cambio, con el diseño TLM se consigue reducir drásticamente los mismos, asemejándose al consumo de recursos del modelo original.

Además, en ninguno de los casos la Spartan6 llega a la frecuencia de funcionamiento propuesta (100 MHz), mientras que tanto la Virtex5 como la Zynq la superan.

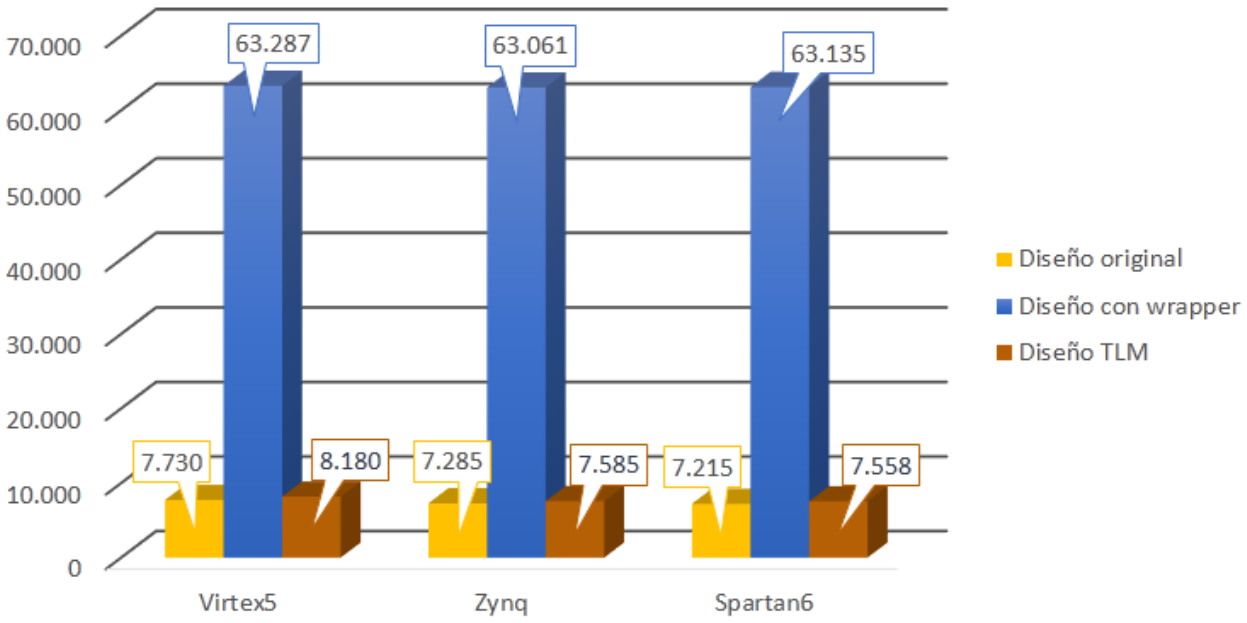


Figura 6.5: Síntesis lógica. Comparativa registros

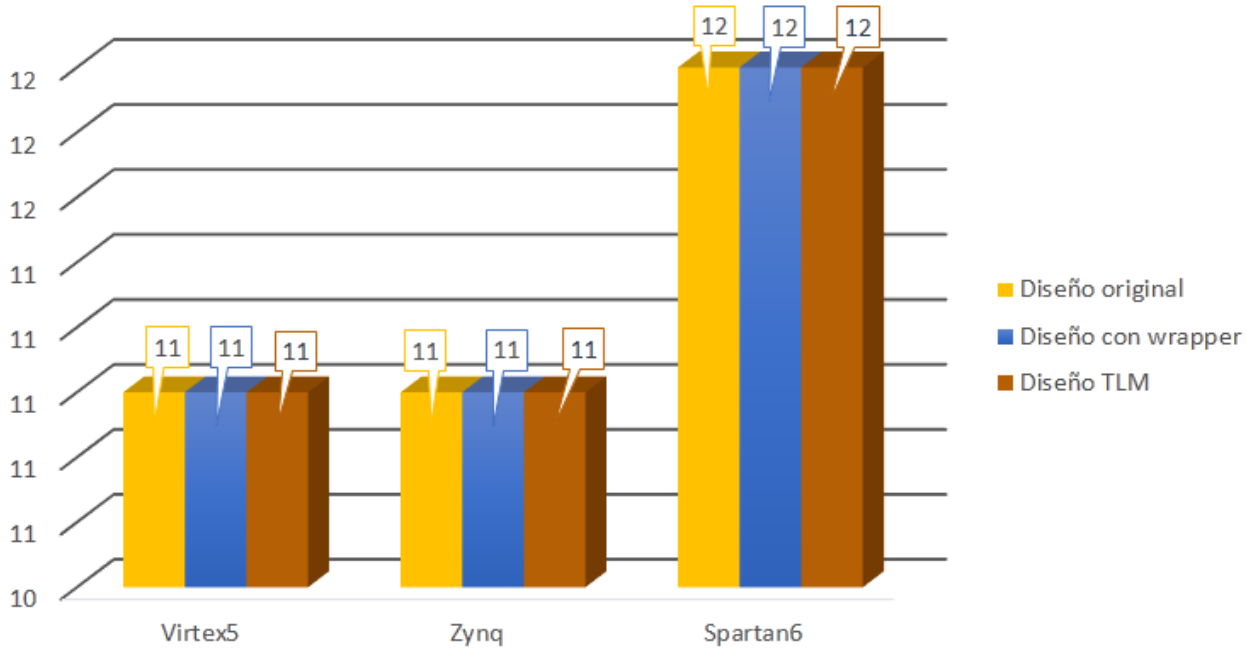


Figura 6.6: Síntesis lógica. Comparativa BRAMs

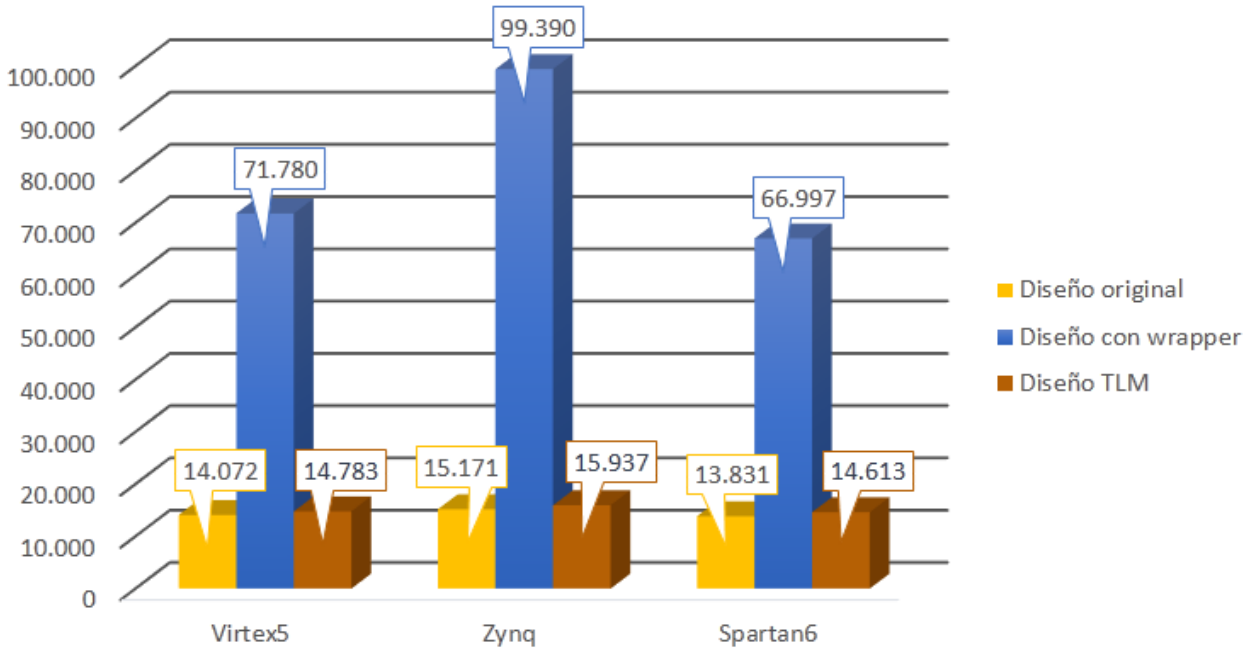


Figura 6.7: Síntesis lógica. Comparativa LUTs

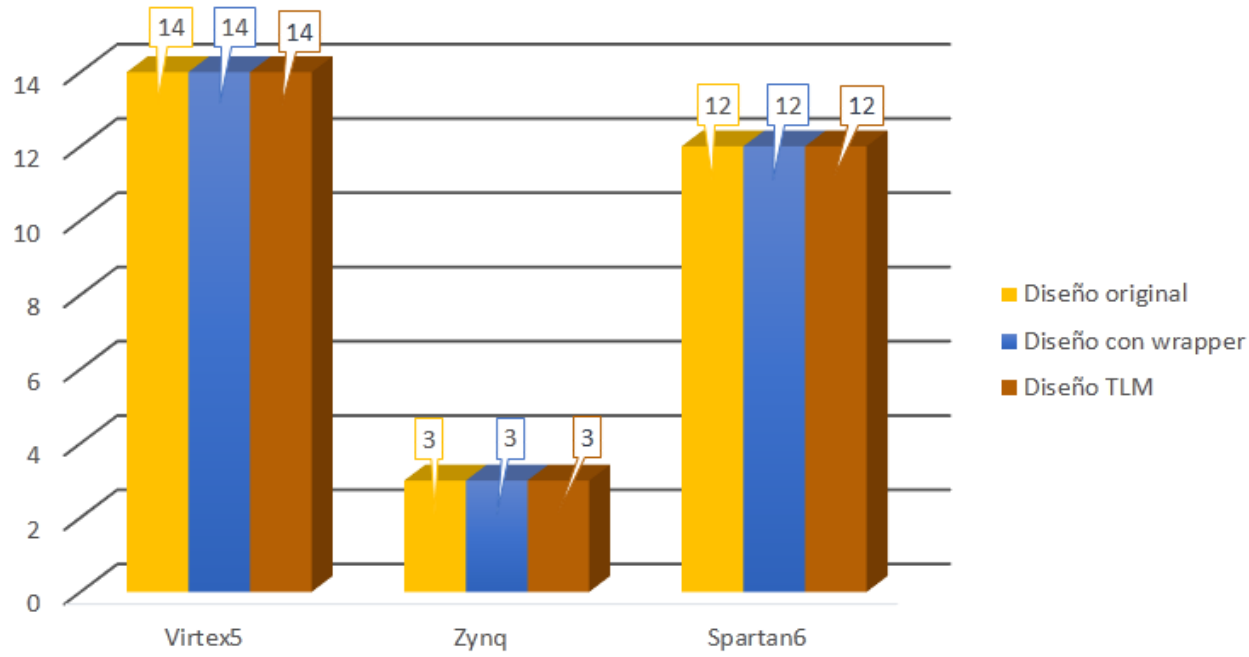


Figura 6.8: Síntesis lógica. Comparativa DSPs

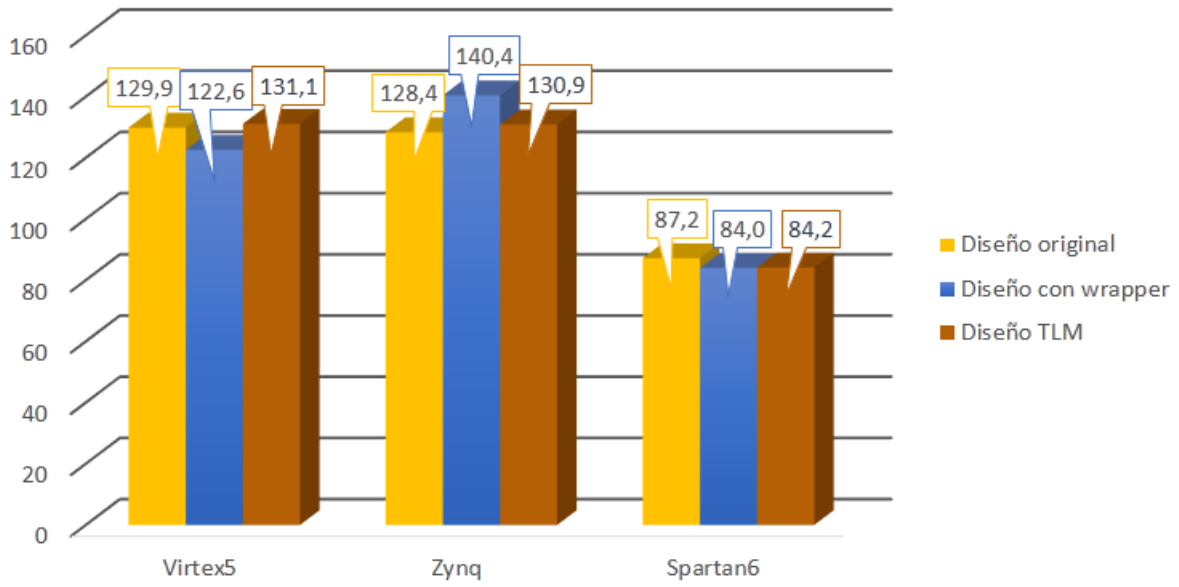


Figura 6.9: Síntesis lógica. Comparativa frecuencia de funcionamiento (MHz)

En las siguientes gráficas se muestran los resultados obtenidos en la síntesis física. Cabe destacar la frecuencia de funcionamiento, disminuyendo en la FPGA Virtex5 y aumentando en la FPGA Zynq, aunque en ambos casos continúan superando la propuesta (100 MHz); y la potencia, la cual para el diseño original y el de TLM disminuye respecto a la obtenida con el *wrapper*.

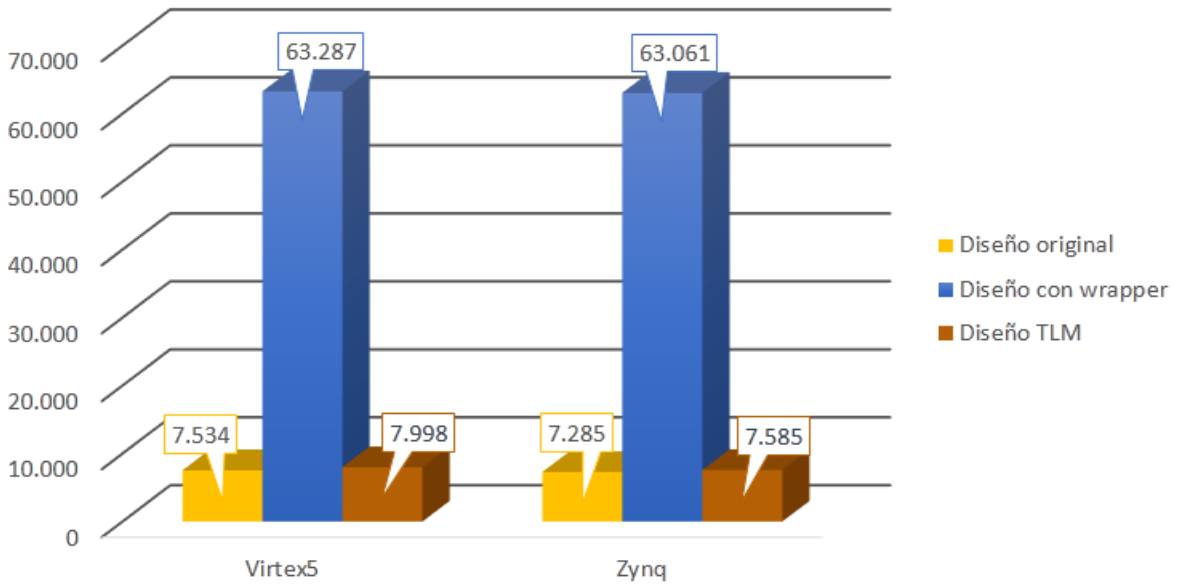


Figura 6.10: Síntesis física. Comparativa registros

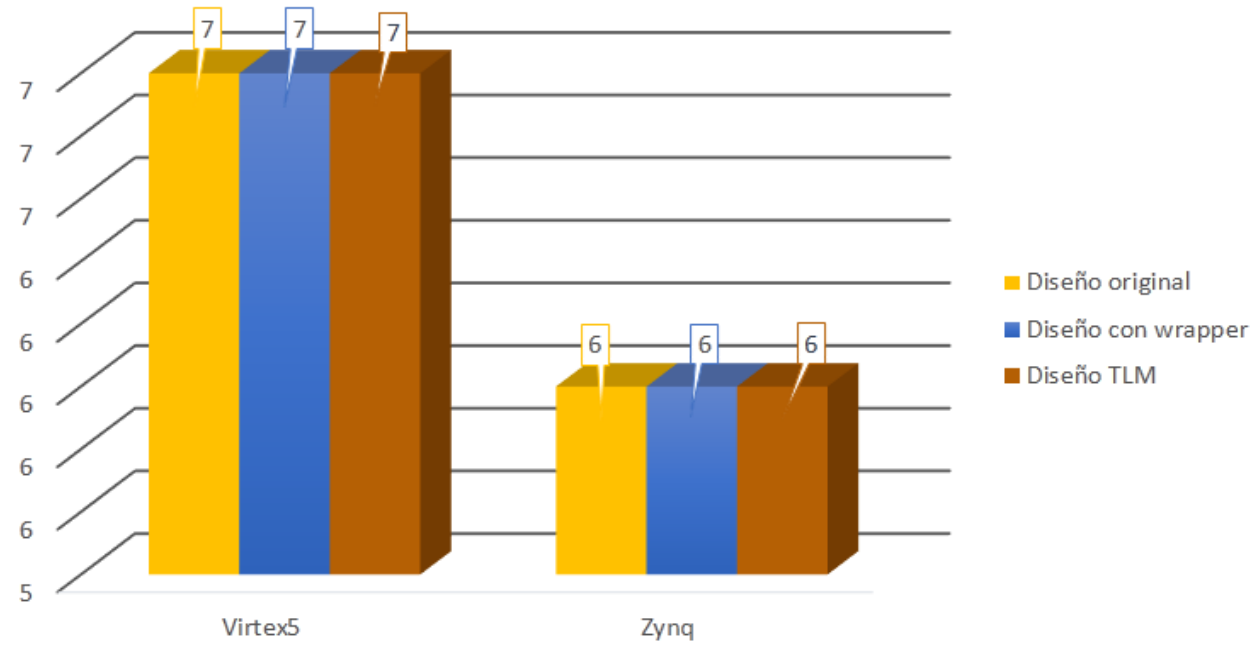


Figura 6.11: Síntesis física. Comparativa BRAMs

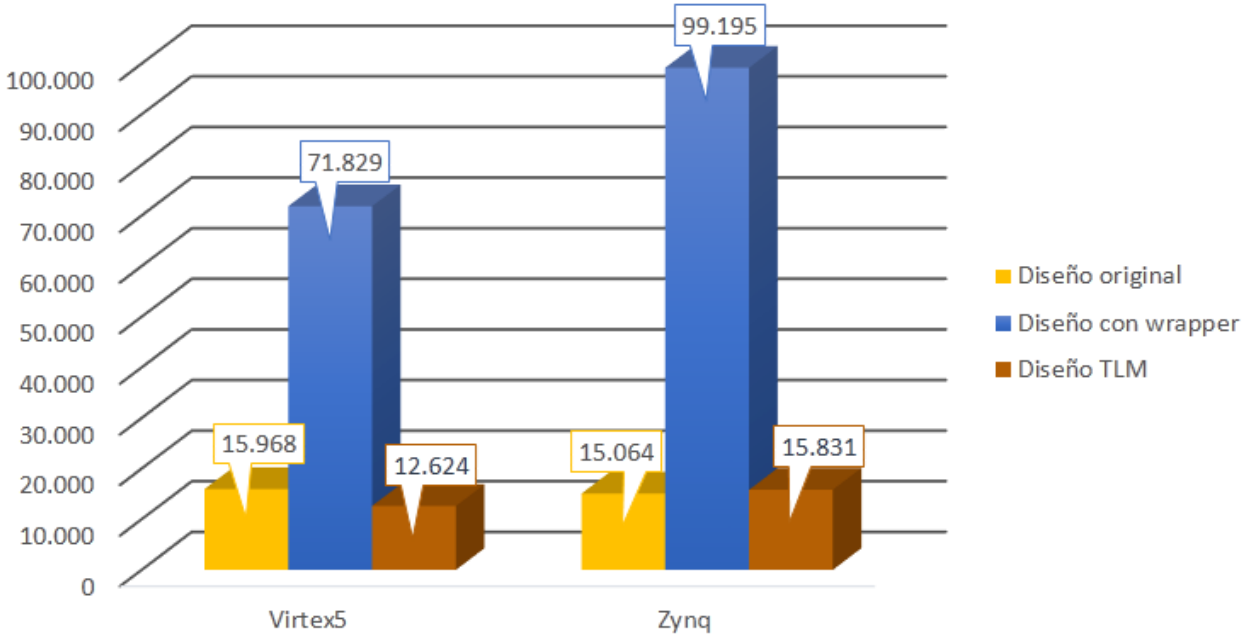


Figura 6.12: Síntesis física. Comparativa LUTs

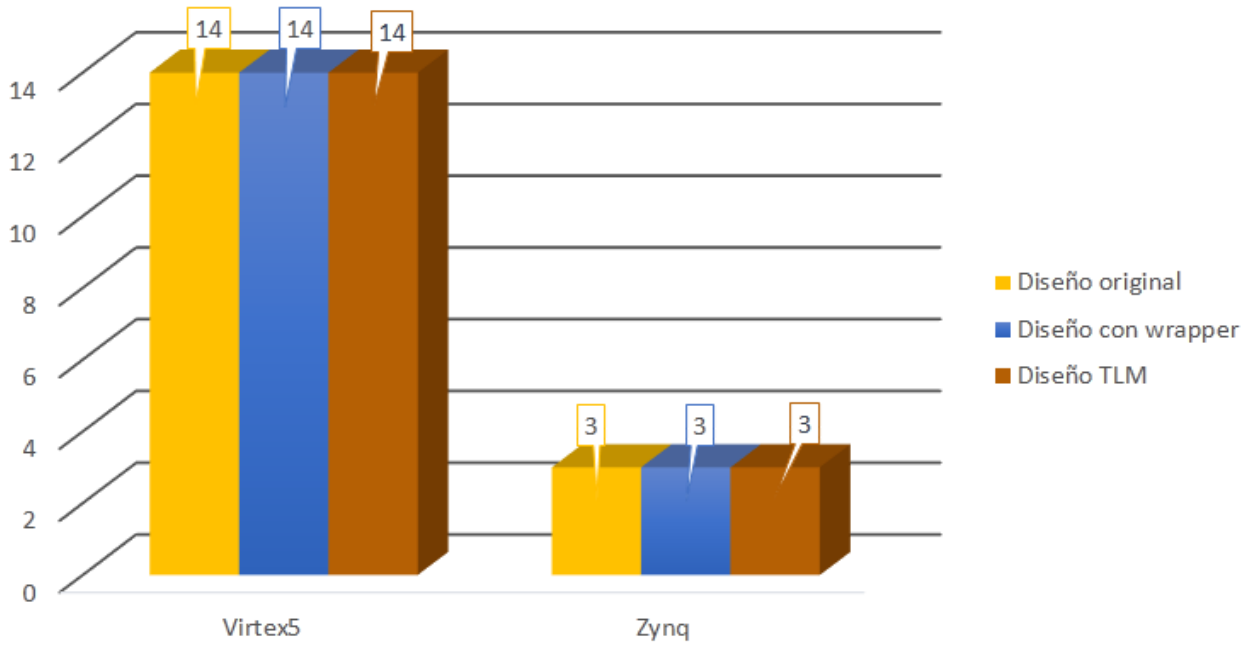


Figura 6.13: Síntesis física. Comparativa DSPs

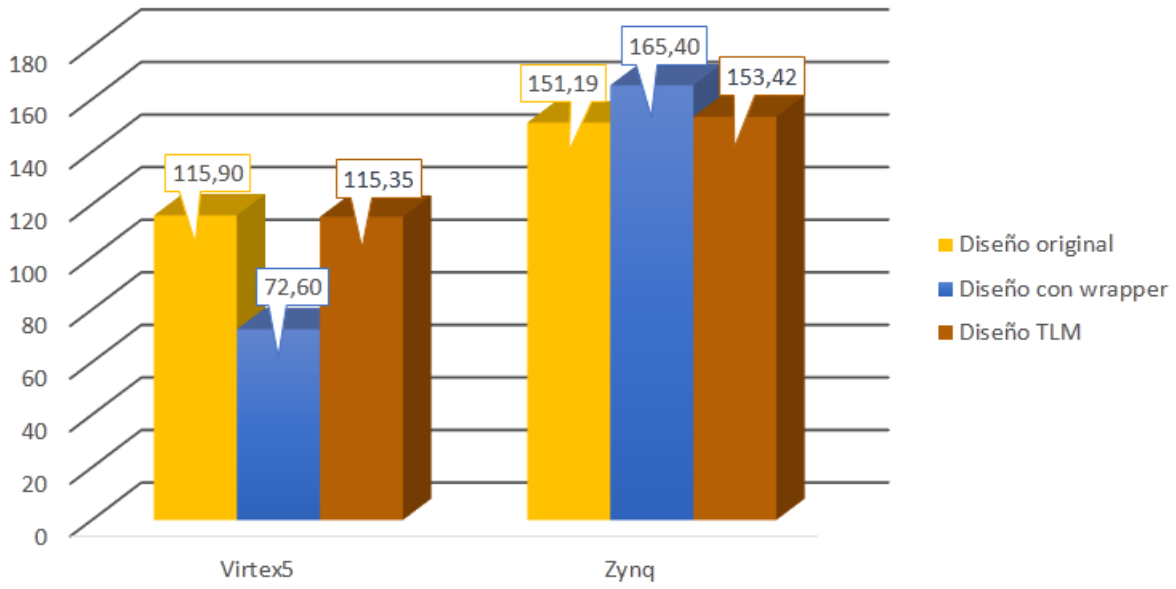


Figura 6.14: Síntesis física. Comparativa frecuencia de funcionamiento (MHz)

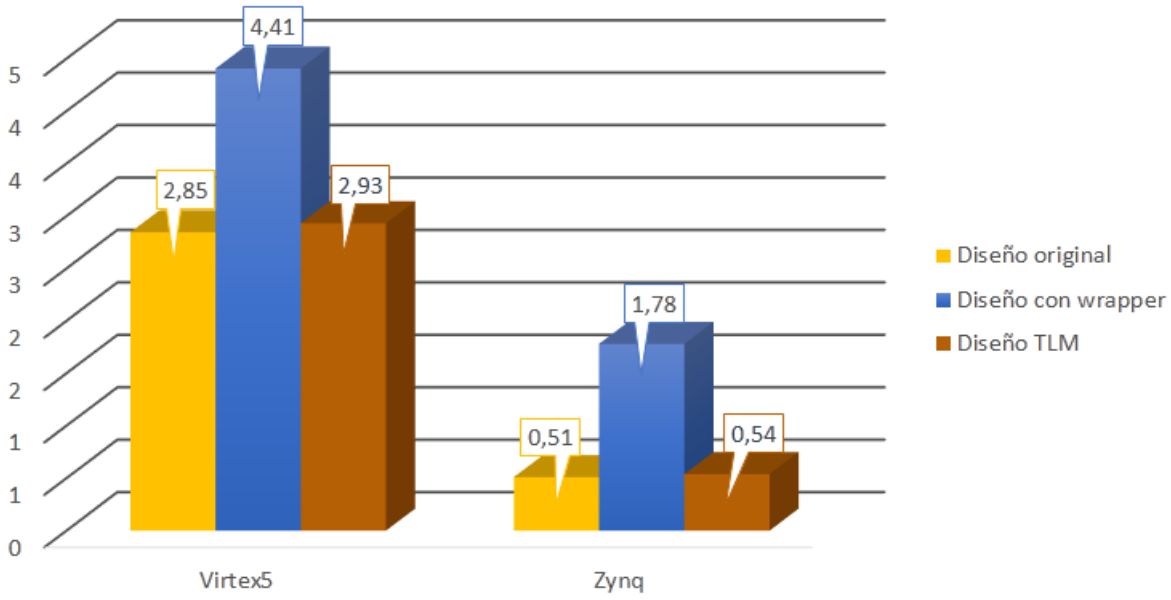


Figura 6.15: Síntesis física. Comparativa potencia (mW)

Por último, se ha buscado la frecuencia límite de cada dispositivo según el diseño.

Frecuencia especificada	**Frecuencia obtenida (MHz)**		
(MHz)	**Diseño original**	**Diseño con *wrapper***	**Diseño TLM**
100	129,9	113,3	131,1
150	139,9	116,1	139,9

Tabla 6.5: Virtex5. Frecuencia límite

Frecuencia especificada	**Frecuencia obtenida (MHz)**		
(MHz)	**Diseño original**	**Diseño con *wrapper***	**Diseño TLM**
100	128,8	140,4	130,9
150	156,2	151,8	153,4
200	182,3	184,7	182,9
250	213,9	189,1	209,1

Tabla 6.6: Zynq. Frecuencia límite

6.4. Conclusiones

Con las dos soluciones desarrolladas se ha podido evaluar la metodología a nivel de transacciones, pudiendo observar los posibles problemas asociados y las ventajas de la misma. Por un lado, la complejidad de la sincronización de dos protocolos: TLM y BCA, donde principalmente se tiene la necesidad de almacenamiento de datos para poder establecer la comunicación entre ambos, lo que supone un alto consumo de los recursos de las FPGAs. Si bien este problema no se presenta durante la simulación se debe tener en cuenta a la hora de realizar la implementación del sistema.

Por otro lado, la importancia de abstraer hacia TLM y no en la otra dirección, hasta ahora la estrategia más común. Aplicando esta metodología directamente sobre las interfaces de un diseño **ha permitido obtener resultados óptimos, y además, un modelo más rápido que el diseño de referencia.**

Capítulo 7

Conclusiones y líneas futuras

7.1. Introducción

Con el proyecto finalizado solo queda exponer las conclusiones y plantear posibles líneas futuras de trabajo, que complementen o mejoren el trabajo realizado.

7.2. Conclusiones

El objetivo principal de este Proyecto Fin de Carrera es la definición de una metodología de diseño de sistemas electrónicos a partir del modelado a nivel de transacciones, obteniendo modelos sintetizables. Para ello se ha utilizado el bloque *inter_p* del decodificador de vídeo H.264/AVC perteneciente al IUMA.

El modelado a nivel de transacciones se basa en abstraer las comunicaciones de la funcionalidad, donde además, **no es necesario que el diseñador implemente un protocolo de comunicación,** ya que este se encuentra a nivel interno y externamente solo hace transacciones.

Se partió del código en SystemC del módulo, tratando el mismo como una “caja gris” (*grey-box model*), es decir, centrándose en las interfaces, y donde la funcionalidad está completamente definida. Se comenzó desde el principio básico de TLM, separación de las comunicaciones de la funcionalidad, y se desarrolló un *wrapper* que adaptara el sistema a una comunicación externa vía TLM.

El modelado a nivel de transacciones es muy básico y fácil de usar, solo hay que prestar atención a las interfaces y los canales, pues él mismo se encarga de generar el protocolo de comunicación. Además, se trata de un nivel de abstracción alto, por encima de RTL, por lo que se pueden evitar los

detalles innecesarios y **se facilita el diseño de un sistema.**

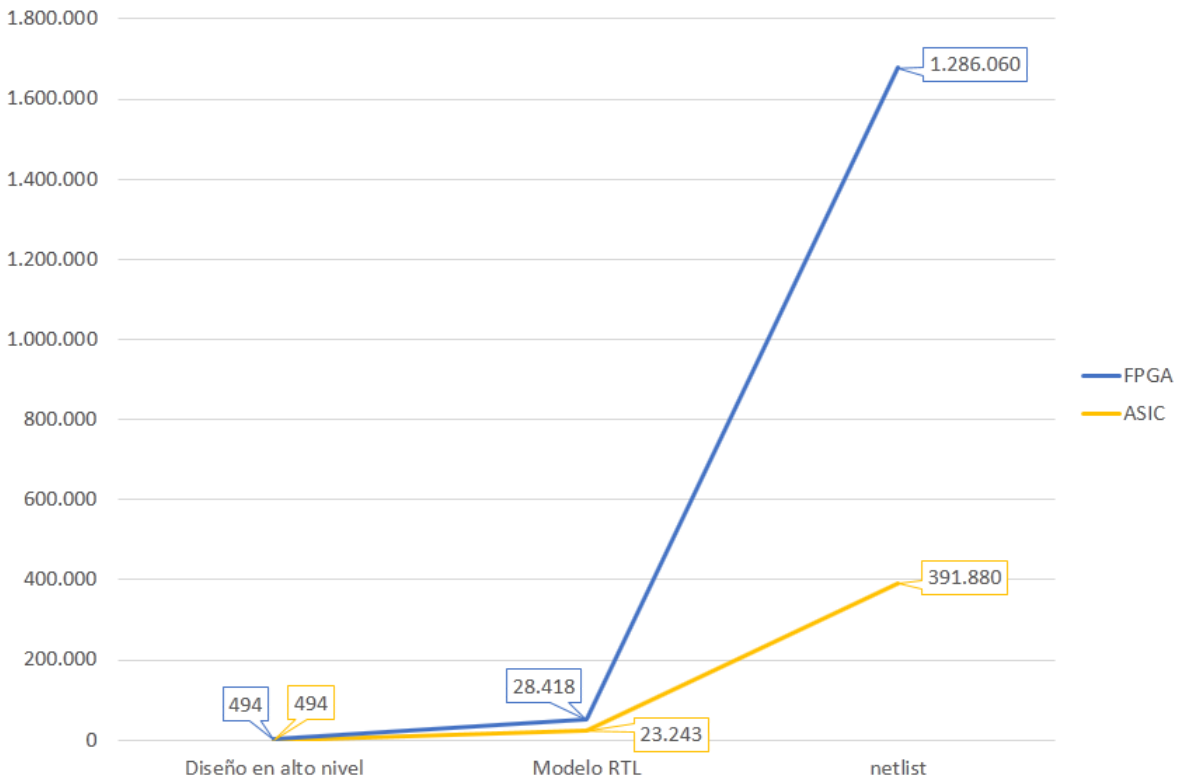


Figura 7.1: Líneas de código

Esta metodología permite aumentar y mejorar la productividad del diseñador, y por tanto reducir los tiempos de diseño e implementación.

La utilización de un *wrapper* no siempre es la mejor solución ya que depende de la naturaleza del bloque desarrollado. En estos casos es necesario adoptar metodologías que modifiquen directamente la interfaz del módulo, con esquemas tipo “caja blanca” (*white-box model*). En este proyecto esta adopción se simplifica por la separación de interfaces y funcionalidad del modelo original. Para el problema estudiado, el *wrapper* desarrollado consume una gran cantidad de recursos de los dispositivos utilizados, debido a que TLM crea su propio protocolo de comunicación y el sistema original también, generando la necesidad de almacenar un conjunto elevado de datos para poder realizar la interfaz entre ambos protocolos.

Parece, por tanto, que la solución natural es implementar directamente las interfaces con TLM, lo cual **no añade más *hardware* al sistema, tanto la frecuencia de funcionamiento como los recur-**

sos consumidos se mantienen comparables a los obtenidos en el diseño original, y además, el módulo TLM es más rápido.

En ambos casos, **se tiene un sistema completamente funcional y que responde al comportamiento esperado,** habiéndose conseguido los modelos sintetizables requeridos como objetivo de este proyecto.

Se ha demostrado que las mejoras en cuanto al incremento en el nivel de abstracción introducidas usando el modelado TLM **no suponen un coste adicional ni afectan a la calidad de los resultados obtenidos ni al uso de los recursos.** Ello muestra la eficiencia de la metodología basada en transacciones para el modelado de sistemas complejos.

7.3. Líneas futuras

A partir de las conclusiones obtenidas se expone una serie de posibles líneas futuras que permita mejorar el trabajo realizado.

1. Desarrollar el sistema con TLM 2.0, aprovechando las ventajas y novedades que implementa. No obstante, TLM 2.0 no está soportado en muchos entornos de síntesis, incluido el utilizado en el presente proyecto, por lo que será necesario revisar la metodología de diseño propuesta.

2. Realizar una exploración más exhaustiva del espacio de diseño que permita mostrar con detalle los límites tecnológicos para los dispositivos y tecnologías elegidas, mejorando la calidad de los resultados.

3. Estudiar otros entornos de síntesis de alto nivel que soporten TLM y evaluar su impacto en la implementación del sistema.

Bibliografía

[1] “Indicadores del sector de tecnologías de la información y las comunicaciones (TIC)”, Instituto Nacional de Estadística, 2011. [En línea], consultado en 2014.

[2] “International Technology Roadmap for Semiconductor. 2012 Update Overview”, International Roadmap Committee, 2012.

[3] “IEEE Standard for Standard SystemC Language Reference Manual”, IEEE Std 1666-2011, 2011.

[4] F. Ghenassia, “Transaction-Level Modeling with SystemC”, Springer, Lightning Source UK Ltd, 2005.

[5] D. C. Black, J. Donovan, B. Bunton, A. Keist, “SystemC: From the Ground Up”, 2nd ed., Springer, 2010.

[6] L. Wang, Y. Chang, K. Cheng, “Electronic Design Automation: Synthesis, Verification and Test”, The Morgan Kaufmann Series in Systems on Silicon, Wayne Wolf, 2009.

[7] J. Erickson, “Developing the Skill Set Required for SystemC TLM-Based Hardware Design and Verification”, Cadence Design Systems, INC., 2013. [En línea], consultado en 2014.

[8] S. Brown, “TLM-Driven Design and Verification – Time for a Methodology Shift”, Cadence Design Systems, INC., 2009.

[9] M. Glasser, “Open Verification Methology Cookbook”, Springer, 2009.

[10] M. Sedghi, A. Shahabi, Z. Navabi, “TLM Studio for Transaction Level Simulation and Synthesis”, DATE 2009, University of Tehran, 2009.

[11] B. Bailey, “A TLM-Driven Design and Verification”, Brian Bailey Consulting, 2010.

[12] S. Rigo, R. Azevedo, L. Santos, “Electronic System Level Design”, Springer, 2011.

[13] “TLM Synthesis”, Forte Design Synthesis, 2011.

[14] S. Brown, “TLM-Driven Design and Verification Solution”, Cadence Design Systems, INC., 2009.

[15] “Catapult C Synthesis”, Calypto, 2011.

[16] “Vivado Design Suite. User Guide. High-Level Synthesis”, Xilinx, 2014.

[17] R. A. Velásquez, E. M. Hernández, “Arquitectura de Decodificación de MPEG-4 para Sistemas Portátiles Inalámbricos”, Universidad de Antioquia, 2005.

[18] “Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding”, ISO/IEC 14496-10:2012, 2012.

[19] I. E. G. Richardson, “The H.264 Advanced Video Compression Standard”, Vcodex Limited, 2nd ed., Wiley, 2010.

[20] I. E. G. Richardson, “H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia”, Wiley, 2003.

[21] A. Domínguez, P. P. Carballo, A. Núñez, “Metodología de Codiseño HW/SW para un Decodificador H.264/AVC”, Universidad de Las Palmas de Gran Canaria, 2008.

[22] Y. W. Huang, “Analysis, Fast Algorithm, and VLSI Architecture Desing for H.264/AVC Intra Frame Coding”, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, 2005.

[23] M. Wien, “Variable Block-Size Transform for H.264/AVC”, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, 2005.

[24] T. Wiegand, G. J. Sullivan, G. Bjontegaard, A. Luthra, “Overview of the H.264/AVC Video Coding Standard”, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, 2003.

[25] M. Flierl, B. Girod, “Generalized B Pictures and the Draft H.264/AVC Video-Compression Standard”, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, 2003.

[26] H. Ochoa, K. R. Rao, “A New Modified Version of the HDWTSVD Coding System for Monochromatic Images”, WSEAS Transaction on Systems, Vol. 5, 2006.

[27] D. Sarasúa, “Aproximación al análisis de secuencias de vídeo codificadas en H.264”, Universidad Autónoma de Madrid, 2010.

[28] M. Nadeem, S. Wong, G. Kuzmanov, A. Shabbir, M. F. Nadeem, F. Anjam, “Low-Power High-Troughput Deblocking Filter for H.264/AVC”, System on Chip (SoC) International Symposium, 2010.

[29] M. Thadani, T. Szydzik, P. P. Carballo, P. Hernández, G. Marrero, A. Núñez, “ESL quantitative assessment of the optimization steps in an ESL flow for a hardware implementation of a H.264/AVC decoder”, 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools , 2009.

[30] M. Thadani, P. P. Carballo, P. Hernández, G. Marrero, A. Núñez, “ESL flow for a hardware H.264/AVC decoder using TLM-2.0 and high level synthesis: a quantitative study”, Proceedings of SPIE Vol. 7363, VLSI Circuits and Systems IV, 2009.

[31] H. Bhatnagar, “Advanced ASIC Chip Synthesis”, 2nd ed., Conexant Systems, INC., Kluwer Academic Publishers, 2002.

[32] T. Wieman, B. Bhattacharya, T. Jeremiassen, C. Schröder, B. Vanthournout, “An Overview of Open SystemC Initiative Standards Development”, IEEE Design & Test of Computers, Vol. 29, 2012.

[33] “Fundamentals of SystemC”, Doulos Ltd., 2007.

[34] S. A. Huss, “System Modeling. Transaction Level Modeling”, Technische Universität Darmstadt, 2007.

[35] W. López, J. Santa Ana, A. González, “Introducción a SystemC”, Universidad Técnica Federico Santa María, 2003.

[36] S. Swam, Q. Zhu, X. Li, “Moving to SystemC TLM for design and verification of digital hardware”, Cadence Design Systems, INC., 2013.

[37] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox, Y. Watanabe, “TLM-Driven Design and Verification Methodology”, Cadence Design Systems, INC., 2010.

[38] “Cadence C-to-Silicon Compiler User Guide”, Cadence Design Systems, INC., 2011.

[39] M. Winterholer, “Europractice/Cadence TLM Flow Information Day”, Cadence Design Systems, INC., 2011.

[40] “SimVision User Guide”, Cadence Design Systems, INC., 2009.

[41] “Cadence SimVision Debug”, Cadence Design Systems, INC. [En línea], consultado en 2014.

[42] “Cadence to Enhance High-Level Synthesis Offering with Acquisition of Forte Design Systems”, Cadence Design Systems, INC., 2014. [En línea], consultado en 2014.

[43] “Introduction to SimVision”, Cadence Design Systems, INC., 2011.

[44] “Synplify Premier User Guide”, Synopsys, INC., 2011.

[45] “Design Compiler Graphical”, Synopsys, INC., 2014.

[46] “PlanAhead User Guide”, Xilinx, 2013.

[47] S. Dutta, “Transaction Level Modeling with SystemC”, RICE University, 2011.

[48] N. Hatami, P. Prinetto, A. Trapanese, “Hardware Design Methodology to Synthesize Communication Interfaces from TLM to RTL”, Automation Quality and Testing Robotics (AQTR), 2010 IEEE International Conference on, Vol. 2, 2010.

[49] M. Montón, B. Martínez, J. Carrabina, “Síntesis de canales TLM para procesador Nios-II”, Jornadas de Computación Reconfigurable y Aplicaciones, 2008.

[50] Z. Yuan, “Transaction Level Modeling (TLM)”, Seminar Algorithms and Tools for Computer Aided Circuit Design, 2011.

[51] N. Bombieri, F. Fummi, V. Pravadelli, S. Vinco, “Redesign and Verification of RTL IPs through RTL-to-TLM Abstraction and TLM Synthesis”, 13th International Workshop on Micropocessor Test and Verification, University of Verona, 2012.

[52] “TLM Interfaces, Ports, and Exports”, Mentor Graphics Corp., 2011. [En línea], consultado en 2014.

[53] A. Rose, S. Swan, J. Pierce, J. M. Fernandez, “Transaction Level Modeling in SystemC”, OSCI TLM Working Group, 2005.

[54] S. Swan, “A Tutorial Introduction to the SystemC TLM Standard”, Cadence Design Systems, INC., 2006.

[55] P. Bishop, “A Look Into How a High-Level Synthesis Design Flow Benefits Verification Turnaround”, Cadence Design Systems, INC., 2013.

[56] Z. Navabi, “The Role of SystemC in the Evolution of Hardware Design”, Worcester Polytechnic Institute, 2012.

[57] J. A. Rubio, “Diseño de Circuitos y Sistemas Integrados”, Universidad Politécnica de Catalunya, 2003.

Presupuesto

1. Introducción

Este presupuesto recoge el estado de costes tanto parciales como totales de este Proyecto Fin de Carrera, realizando un informe detallado sobre los recursos materiales empleados, los costes de ingeniería, el material fungible y los costes que ha conllevado la edición del mismo.

2. Recursos materiales

A continuación se presentan los costes producidos por el uso de las herramientas *software*, el material *hardware* y el mantenimiento de estos. El coste se ha establecido en función al periodo de tiempo en el que han sido utilizados.

2.1. Recursos *hardware*

Los equipos de desarrollo lo componen un ordenador de sobremesa y un portátil para las labores de diseño y documentación; y un *cluster* de servidores de cómputo para las tareas de síntesis. Además, para la impresión de la documentación se hizo uso de las impresoras del Instituto Universitario de Microelectrónica Aplicada. Sus características son:

- **Ordenador de sobremesa:** Intel Pentium 3GHz, 3,24GB RAM. Ejecuta como sistema operativo Ubuntu 10.04.
- **Ordenador portátil:** Dell Latitude E6400 (Intel Core Duo 2,53GHz, 4GB RAM). Ejecuta como sistema operativo Ubuntu 14.04.
- **Servidores de cómputo:** SunFire X2200 (AMD Opteron 2,2GHz, 4GB RAM DDR3). Ejecutan como sistema operativo Red Hat Enterprise 5.
- **Impresora blanco y negro:** HP LaserJet 4350dtn.

- **Impresora color:** HP LaserJet 4700dn.

Todos los equipos están conectados en red a los recursos de cómputo y almacenamiento del Instituto Universitario de Microelectrónica Aplicada mediante red Gigabit Ethernet, y gestionados por el Servicio de Infraestructura de Red (SIR). Se incluyen los costes de operación y amortización.

Descripción	Coste	Factor de amortización		Meses de uso	Coste asociado
		Meses	Usuarios		
Ordenador de sobremesa	800,00€	12	1	12	800,00€
Ordenador portátil	1.000,00€	12	1	12	1.000,00€
Servidores de cómputo	12.504,00€	12	5	9	1.875,60€
Total					**3.675,60€**

Tabla 1: Coste recursos *hardware*

2.2. Herramientas de diseño y acceso a librerías

Las herramientas de automatización del diseño electrónico (EDA) y librerías tecnológias usadas en este Proyecto Fin de Carrera pertenecen al IUMA, siendo gestionadas por el Servicio de Tecnologías y Herramientas (STH). Se incluyen los costes de adquisición y mantenimiento.

Descripción	Tipo de licencia	Coste de licencia	Mantenimiento anual
Paquete Cadence (IC & Systems Packages)	Universitaria	1.926,00€	1.968,80€
Paquete Synopsys (Front End and Verification)	Universitaria	1.926,00€	1.123,50€
Paquete Xilinx (Vivado System Edition)	Universitaria	Donación	214,00€
Suscripción a herramientas	Universitaria	–	535,00€
Suscripción a kits de diseño	Universitaria	–	642,00€
Total		**3.852,00€**	**4.483,30€**

Tabla 2: Coste recursos *software*

El coste total asociado a los recursos materiales se recoge en la siguiente tabla.

Descripción	Coste
Coste de amortización y operación de recursos *hardware*	3.675,60€
Coste proporcional del uso de recursos de herramientas y librerías (15 %)	1.250,30€
Total	**4.925,90€**

Tabla 3: Coste recursos materiales

3. Costes de ingeniería

En la realización de este Proyecto Fin de Carrera se ha invertido un total de 12 meses, siendo los seis primeros a media jornada y los seis restantes a jornada completa. Durante este tiempo se han llevado a cabo las tareas de formación, desarrollo y documentación.

Para conocer el coste de un ingeniero se ha contemplado la Tabla de Contrataciones publicada en el Boletín Oficial de la Universidad de Las Palmas de Gran Canaria (Año III, núm. 11, 4 de noviembre de 2010), que atribuye un coste mensual de 1.921,64€ para un técnico en proyecto con titulación de Ingeniero. Durante los seis primeros meses, al haber trabajado a media jornada se estima que el salario corresponde a 960,82€/mes.

Descripción	Meses	Coste mensual	Coste total
Etapa de estudio del problema, formación y *training*	2	960,82€	1.921,64€
Desarrollo del proyecto (tiempo parcial)	4	960,82€	3.843,28€
Desarrollo del proyecto (tiempo completo)	5	1.921,64€	9.608,20€
Documentación final	1	1.921,64€	1.921,64€
Total			**17.294,76€**

Tabla 4: Costes de ingeniería

4. Material fungible

Se considera material fungible todo aquel cuya vida útil es inferior a un año e importe unitario no sobrepasa una cantidad determinada, generalmente de reducido valor. En este caso, los gastos asociados a consumibles utilizados durante el desarrollo del proyecto como pueden ser impresiones de documentación o CDs, entre otros, han sido de **180,00€**.

5. Costes de edición

El coste de la edición de la memoria: papel, impresión y encuadernación, ha sido de **400,00€**.

6. Resumen de costes

Ítem	Descripción	Coste
1.	Recursos materiales (*hardware* y herramientas)	4.925,90€
2.	Costes de ingeniería	17.294,76€
3.	Material fungible	180,00€
4.	Costes de edición	400,00€
	Subtotal	22.800,66€
	7 % IGIC	1.596,05€
	Total	**24.396,70€**

Tabla 5: Tabla resumen de costes

Dña. Paloma Monzón Rodríguez declara que el Proyecto Fin de Carrera *"Metodología de Síntesis ESL basada en TLM. Aplicación al Diseño de un Decodificador de Vídeo"* asciende a una cantidad total de **veinticuatro mil trescientos noventa y seis euros con setenta céntimos (24.396,70€).**

Fdo. Paloma Monzón Rodríguez

En Las Palmas de Gran Canaria, a 11 de marzo de 2015.

Pliego de condiciones

1. Introducción

Durante la realización de este Proyecto Fin de Carrera se ha hecho uso de un conjunto de herramientas de automatización del diseño electrónico y equipos *hardware*, cuyas características y versiones se listan en los siguientes apartados, así como la garantía de cada uno de ellos.

Se garantiza que los diseños son compilables y se pueden simular en las versiones de las herramientas indicadas, bajo los sistemas operativos de los equipos *hardware* listados.

2. Equipos *hardware*

Equipo de trabajo	Descripción
Ordenador de sobremesa	Intel Pentium 3GHz, 3,24GB RAM bajo Ubuntu 10.04
Ordenador portátil	Dell Latitude E6400 (Intel Core Duo 2,53GHz, 4GB RAM) bajo Ubuntu 14.04
Servidores de cómputo	SunFire X2200 (AMD Opteron 2,2GHz, 4GB RAM DDR3) bajo Red Hat Enterprise Linux Server 5.3
Impresora blanco y negro	HP LaserJet 4350dtn
Impresora color	HP LaserJet 4700dn

Tabla 1: Equipos *hardware*

3. Herramientas *software*

Todos los recursos *software* han sido utilizados bajo la plataforma UNIX, bien sea con el sistema operativo Red Hat Enterprise para las tareas de síntesis, o bien con Ubuntu para las tareas de modelado

del sistema y desarrollo de la documentación del proyecto; excepto el paquete ofimático de Microsoft, que ha sido utilizado bajo el sistema operativo Windows 7.

Herramienta	Versión	Fabricante	Descripción
Simulation Analysis Environment SimVision	09.20-s022	Cadence	Entorno de simulación
Cadence C-to-Silicon Compiler	11.20-s200	Cadence	Entorno de síntesis de alto nivel
Synplify Premier with Design Planner	I-2014.03	Synopsys	Entorno de síntesis lógica para FPGAs
Design Vision	I-2013.12-SP2	Synopsys	Entorno de síntesis lógica para ASICs
PlanAhead	14.7	Xilinx	Entorno de síntesis e implementación
Vivado Design Suite	2014.04	Xilinx	Entorno de síntesis e implementación
UMC 65nm	2012	UMC	Librería tecnológica para ASICs
Texmaker	3.4	*Software* libre	Editor de texto con entorno de trabajo LaTeX
GIMP	2.8.4	*Software* libre	Editor de imágenes
Microsoft Office	2013	Microsoft	Paquete ofimático

Tabla 2: Herramientas *software*

4. Garantía

El autor del proyecto lo presenta **"AS IS" (tal cual), sin garantía implícita de ningún tipo**. Tampoco se responsabiliza de los daños que pueda causar el código presentado o sus archivos derivados a cualquier equipo o del uso que hagan de ellos terceras personas.

5. Cláusula de confidencialidad

El presente Proyecto Fin de Carrera se desarrolla en el marco de las líneas de trabajo de la División de Sistemas Industriales y CAD del Instituto Universitario de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, incluyendo sus distintos proyectos de investigación y convenios con empresas. Por ello, **el material usado, producido y/o referenciado en este proyecto puede estar sujeto a cláusulas de confidencialidad**, lo que se declara, a los efectos reglamentarios.