UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una Unidad de Despacho para la Plataforma SoC DPI basada en FPGA Xilinx Zynq

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación. Sistemas Electrónicos

Autor: Irene González Crespo

Tutores: Pedro Pérez Carballo
Benjamín Vega del Pino
Pedro Hernández Fernández

Fecha: Julio de 2016

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una Unidad de Despacho para la Plataforma SoC DPI basada en FPGA Xilinx Zynq

HOJA DE FIRMAS

Alumna:

Fdo.: Irene González Crespo

Tutor: **Tutor:** **Tutor:**

Fdo.: Pedro Pérez Carballo Fdo.: Benjamín Vega del Pino Fdo.: Pedro Hernández Fernández

Fecha: Julio de 2016

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de una Unidad de Despacho para la Plataforma SoC DPI basada en FPGA Xilinx Zynq

HOJA DE EVALUACIÓN

Calificación: _____

Presidente:

Fdo.:

Vocal:

Fdo.:

Secretario/a:

Fdo.:

Fecha:

AGRADECIMIENTOS

A mis tres tutores, Pedro Pérez Carballo, Benjamín Vega del Pino y Pedro Hernández Fernández, debo agradecer especialmente su tiempo, paciencia y dedicación. Gracias por enseñarme tanto.

También quiero agradecer a mi familia y amigos el apoyo y confianza prestados a lo largo de estos años. Sin ellos este Trabajo de Fin de Grado no sería hoy una realidad.

RESUMEN

En este Trabajo Fin de Grado se realiza el diseño, verificación e implementación de un bloque de comunicaciones en chip, al que se ha llamado Unidad de Despacho, encargado de realizar el envío de paquetes de datos entre una unidad MAC y una serie de bloques procesadores de datos. La implementación del bloque de comunicaciones se ha realizado en una placa de desarrollo ZedBoard basada en un dispositivo de la familia Zynq-7000 de Xilinx. El objetivo del Trabajo Fin de Grado es obtener una comunicación eficiente entre la unidad de red y los bloques procesadores de datos, liberando al bloque de acceso directo a memoria (DMA) de realizar las lecturas y escrituras en la memoria RAM de los datos proporcionados por la unidad de red.

Inicialmente, se describen todos los recursos, tanto hardware como software, utilizados para la realización del Trabajo Fin de Grado y se exponen sus características principales y su funcionamiento orientados al desarrollo del trabajo. A continuación, se explica el diseño del bloque IP utilizando síntesis de alto nivel, describiendo su arquitectura y funcionamiento y los pasos seguidos para la obtención del mismo. También se explican las fases de diseño y síntesis necesarias para su desarrollo. Una vez obtenido el bloque IP, se procede a explicar las fases de implementación e integración en el dispositivo Zynq-7000. Finalmente, se exponen las fases de verificación y validación del sistema. Se analizan los recursos consumidos y la latencia de la UD, así como se comprueban los tiempos de ejecución en situaciones diferentes.

Como conclusión, se puede observar que es posible la obtención de un mayor ancho de banda sin necesidad de comprometer la eficiencia del sistema. El envío y la recepción de los paquetes de datos proporcionados por la unidad de red se realizan para velocidades de Gigabit. Por este motivo, se puede afirmar que el sistema diseñado para la plataforma empotrada es funcional y gracias a ello se obtiene un sistema fiable y efectivo.

ABSTRACT

In this end-of-degree project, the design, verification and implementation of a DPI communication block is accomplished. The so called Dispatch Unit (UD) is used to send data packages between a network unit and a series of data processing blocks. The implementation of the communication block has been done on a ZedBoard development board, based on a Xilinx Zynq-7000 FPGA. The aim of this project is to obtain an efficient communication between the network unit and the data processing blocks. Consequently, the Direct Memory Access (DMA) block is released of accomplishing reads and writes in the RAM memory of the data provided by the network unit.

Firstly, all resources used for the end-of-degree project, both hardware and software, are introduced, exposing its main characteristics and performance-oriented development work. Then the IP block design is explained using high-level synthesis, describing its architecture and operation, and the steps for obtaining it. Likewise, the phases of design and synthesis flow necessary for its development is explained. After creating the IP block, the phases of the implementation and integration into the Zynq-7000 platform are explained. Finally, the verification and validation phases are exposed in order to obtain the utilization and latency parameters of the UD, as well as execution times in different critical situations.

In conclusion, after analyzing the results is concluded that it is possible to obtain a higher bandwidth without compromising system efficiency. Sending and receiving data packages provided by the network drive are performed at Gigabit speeds. For this reason the designed system on an embedded platform is functional and, as a result, a reliable and effective system.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ACRÓNIMOS

ACP	Accelerator Coherency Port
AMBA	Advanced High-Performance Bus
AMP	Asymmetrical Multiprocessing
ALU	Arithmetic Logic Unit
APB	Advanced Peripheral Bus
APU	Application Processor Unit
AXI	Advanced Extensible Interface
AXI_ACP	AXI Accelerator Coherency Port
AXI_GP	AXI General Purpose
AXI_HP	AXI High Performance Port
BRAM	Block Random Access Memory
BSP	Board Support Package
BST	Behavior-Structure-Timing
CABA	Cycle Accurate Bit Accurate
CAN	Controller Area Network
CFS	Constraint-Functionality Separation
CLB	Configurable Logic Block
CPU	Central Processing Unit
CtoS	C-to-Silicon Compiler
DCP	Design Checkpoint
DDR	Double Data Rate
DDR3L	Double Data Rate 3 Low Voltage
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
ELS	Embedded Logic Synthesis
EMIO	Extendable Multiplexed I/O
FCLK	Frequency-Programmable Clocks
FHM	Fast Hardware Model
FIFO	First In First Out
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

GIC	General Interrupt Controller
GMII	Gigabit Media-Independent Interface
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HDMI	High-Definition Multimedia Interface
HDL	Hardware Description Language
HLS	High Level Synthesis
HP	High Performance
HR	High Range
I2C	Inter-Integrated Circuit
IDE	Integrated Design Environment
ILA	Integrated Logic Analyzer
IOP	Input/Output Peripherals
IP	Intellectual Property
IP	Internet Protocol
IUMA	Instituto Universitario de Microelectrónica Aplicada
LPDDR-2	Low Power Double Data Rate 2
LUT	Look-Up Tables
LVCMOS	Low-Voltage CMOS
LVDS	Low-Voltage Differential Signaling
MIO	Multiplexed Input/Output
OS	Operating System
PCAP	Processor Configuration Access Port
PL	Programmable Logic
PLL	Phase-Locked Loop
PPI	Private Peripheral Interrupt
PS	Processing System
RGMII	Reduced Gigabit Media-Independent Interface
RTL	Register Transfer Level
SCU	Snoop Control Unit
SDC	Synopsys Design Constraints
SD	Secure Digital
SDIO	Secure Digital Input/Output
SDK	Software Development Kit
SEU	Single Event Upset
SGI	Software Generated Interrupt

SGMII	Serial Gigabit Media-Independent Interface
SICAD	Sistemas Industriales y CAD
SMC	Static Memory Controller
SMP	Symmetrical Multiprocessing
SLCRs	System-Level Control Registers
SoC	System on Chip
SPI	Shared Peripheral Interrupt
SRAM	Static Random Access Memory
TFG	Trabajo Fin de Grado
UART	Universal Asynchronous Receiver/Transmitter
ULPGC	Universidad de Las Palmas de Gran Canaria
UD	Unidad de Despacho
USB	Universal Serial Bus
USB OTG	USB On-The-Go
VIO	Virtual Input/Output
XADC	Analog-to-Digital Converter
XDC	Xilinx Design Constraints
XMD	Xilinx Microprocessor Debugger
XSDB	Xilinx System Debugger

Capítulo 1. Introducción

A lo largo de este primer capítulo se justifica la realización de este proyecto, indicando los antecedentes que llevan a su realización, así como los objetivos planteados y la estructura del documento.

1.1. Antecedentes

Este trabajo está basado en las demandas cada vez más exigentes de los sistemas electrónicos, que van desde una necesidad de un mayor ancho de banda, sin comprometer la eficiencia del sistema, hasta una necesidad constante de tener sistemas con comunicaciones fiables y efectivas.

A su vez, es necesaria la implementación de estas mejoras dentro de las características de las FPGA. Esto se debe a la predisposición de estos sistemas a cambios, su flexibilidad y las facilidades de reprogramación, así como de integración y depurado.

1.1.1. Demandas en los sistemas electrónicos

En un mundo interconectado, las necesidades de comunicaciones de los clientes van en aumento y, por consiguiente, la industria busca maximizar las capacidades de los sistemas electrónicos para cumplir con estos requerimientos. Por ello, la demanda de un mayor ancho de banda se ha vuelto una de las bases fundamentales en el desarrollo de cualquier sistema electrónico. Sin embargo, el aumento del ancho de banda de los dispositivos no debe ir en deterioro de su flexibilidad y eficiencia. Si se tiene en cuenta el ancho de banda, la flexibilidad y la eficiencia se puede obtener un sistema electrónico con capacidades de comunicación elevadas, siendo fiable a la vez que eficaz [1].

La ley de Nielsen muestra esta realidad. Se basa en la suposición de que cada año el ancho de banda utilizado en Internet crecerá un 50% respecto al año anterior. En la Figura 1 se muestra el crecimiento exponencial del ancho de banda, según la ley de Nielsen citada con anterioridad. Cada punto del diagrama muestra mediciones reales de las velocidades obtenidas en el uso de internet [1].

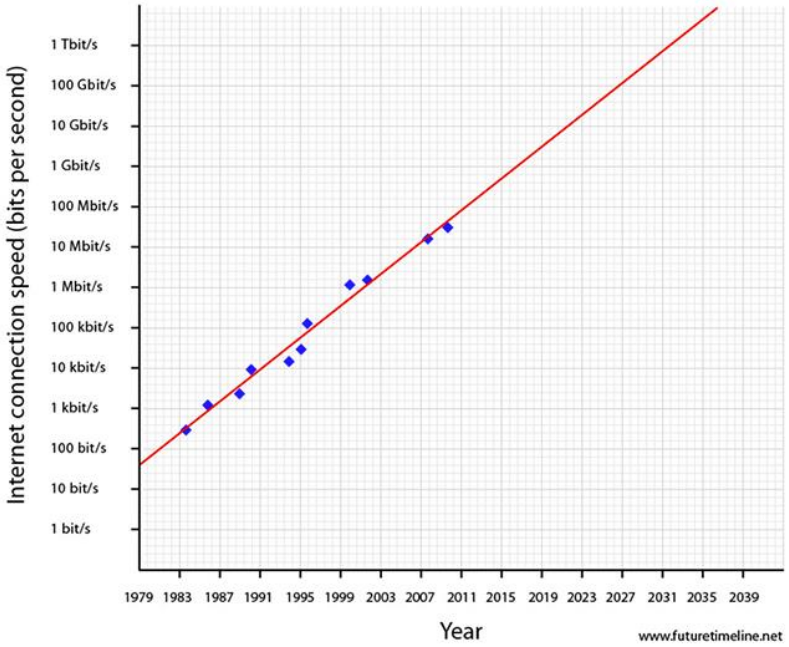

Figura 1: Gráfico de la ley de Nielsen1

Para conseguir este objetivo, además de incrementar la capacidad de procesamiento de los sistemas electrónicos, es preciso generar esquemas de comunicación en chip adecuados para los bloques de un sistema electrónico. La arquitectura de comunicaciones ha sido objeto de estudio debido al impacto que tiene en las prestaciones finales. Con el paso de los años los requisitos se vuelven más exigentes en cuanto a prestaciones, haciendo cada vez más necesario realizar un análisis exhaustivo del sistema para generar diseños competentes, con un mayor ancho de banda y que reduzcan el consumo de recursos del sistema.

1.1.2. Plataformas basadas en FPGAs

Debido a la demanda indicada se busca desarrollar una unidad de despacho capaz de conseguir implementar estos requisitos en la comunicación de una FPGA determinada. Para este proyecto se decide trabajar con el sistema en chip ZYNQ de Xilinx [2, 3], que dispone de una gran cantidad de elementos para la implementación de la unidad de despacho. En particular, se usa la placa de desarrollo ZedBoard basada en el dispositivo Xilinx® XC7Z020-1CLG484C Zynq-7000 AP SoC.

La unidad de despacho se ha diseñado para formar parte de un sistema dedicado a la inspección profunda de paquetes (DPI). En este tipo de sistemas las latencias de captura y transmisión

[1] Ver: Future of the Internet – 8 Expanding Dimensions (http://www.futuristspeaker.com/business-trends/future-of-the-internet-8-expanding-dimensions/)

de paquetes a los aceleradores *hardware* juegan un papel central en el éxito de la aplicación. Este TFG se plantea con el objetivo de reducir dichas latencias.

Debido al funcionamiento general del sistema en chip Zynq 7000 de Xilinx y a la comunicación existente entre la unidad de red y los diferentes bloques destinatarios, surge la necesidad de crear una unidad de despacho para poder realizar la comunicación de forma más eficiente, minimizando el uso de recursos del sistema. Para ello, se deben tener en cuenta las restricciones necesarias que permiten que la comunicación se desarrolle de forma eficiente.

La captura de datos provenientes de la interfaz de red Ethernet normalmente requiere de un *stack* TCP/IP [4] que se implementa en un microprocesador (ARM Cortex A9 para el dispositivo utilizado), en este caso empotrado en la FPGA. En el caso de la FPGA Zynq de Xilinx la unidad de red se comunica directamente con un DMA (*Direct Memory Access*) para poder guardar los datos en la memoria RAM. A continuación, cuando alguno de los datos almacenados debe ser transmitido a los bloques IP, otro DMA accede a la memoria RAM para tomar estos datos y ser capaz de enviarlos a los diferentes IPs de procesamiento [5]. Sin embargo, como se ha comentado anteriormente, este proceso implica la presencia del microprocesador para ejecutar la implementación *software* del *stack* TCP/IP indicado. Esta solución es menos eficiente que si se dispone de un bloque dedicado a la gestión directa de los datos entrantes en el sistema a través de la interfaz de red. Para velocidades crecientes en las tasas de datos esta solución *hardware* se hace obligatoria.

Se propone por tanto el diseño de un bloque IP *hardware* que sea capaz de capturar los paquetes de datos entrantes por la interfaz Ethernet [6, 7] y los dirija hacia nodos de procesamiento internos dedicados a la inspección profunda de paquetes de datos. Este bloque IP, que se puede observar en la Figura 2, se denominará unidad de despacho (UD).

Sin embargo, algo que se debe tener en cuenta en el desarrollo de este tipo de sistemas de comunicación es la planificación necesaria. El proyecto se centra en un planificador que sea capaz de enviar los datos recibidos a diferentes unidades de cómputo. El objetivo es obtener un flujo de datos continuo de manera flexible y eficiente, sin la necesidad de realizar almacenamiento en memoria externa ni de utilizar una gran cantidad de recursos [8]. De esta manera se controla el flujo de datos en la FPGA evitando las transferencias de los paquetes con el microprocesador a través de un DMA. Esta alternativa no solo reduce las latencias de comunicación con la RAM sino que además permite prescindir de una librería *software* específica que implemente el *stack* TCP/IP.

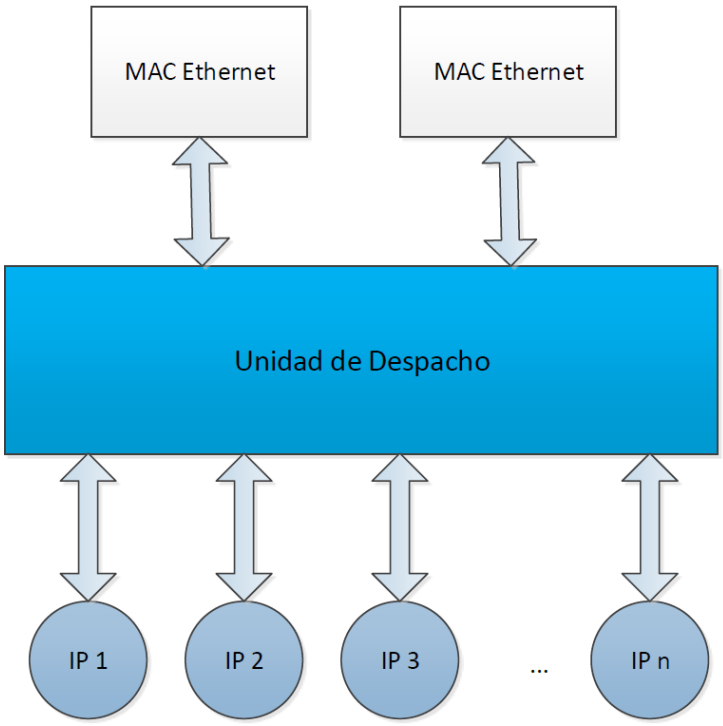

Figura 2: Esquema de bloques de interconexión de la UD

1.2. Objetivos

Los objetivos de este trabajo se centran en conseguir una unidad de despacho orientada al flujo de datos [9, 10] capaz de realizar una comunicación eficiente y ágil de interfaz entre la unidad de red y los bloques aceleradores IP del sistema en chip Zynq de Xilinx.

La unidad de despacho conectada entre la unidad de red y los bloques de destino, realizará el envío de los paquetes de datos teniendo la inteligencia necesaria para detectar diferentes protocolos, definir diferentes reglas en función del protocolo y disparar acciones concretas sobre los paquetes de datos. El objetivo principal de este componente es reducir el uso de recursos y liberar al DMA del envío/recepción hacia/desde la memoria RAM de los datos proporcionados por la unidad de red. La UD poseerá diferentes modos de operación en lo que se refiere a la comunicación entre la interfaz Ethernet y los IPs de procesamiento. Se implementarán tres modos de operación básicos diferentes: *broadcast*, *unicast* y segmentación.

Los objetivos operativos en los que se organiza el proyecto son los siguientes:

O1. Estudiar la arquitectura de buses presente en el SoC FPGA Zynq.
O2. Estudiar los procedimientos de diseño necesarios para utilizar los bloques que componen el bus del sistema y su integración en la plataforma.

O3. Diseñar la arquitectura de la unidad de despacho, realizar su modelado y simulación y analizar sus prestaciones a nivel de transacciones (TLM).
O4. Integrar la unidad de despacho en la plataforma hacia su implementación y validar su funcionalidad y sus prestaciones.
O5. Documentar el trabajo realizado.

1.3. Peticionario

Actúa como peticionario la División de Sistemas Industriales y CAD (SICAD) del Instituto Universitario de Microelectrónica Aplicada de la ULPGC el diseño de un bloque IP específico a ser incluido en su plataforma DPI.

Igualmente actúa como peticionario de este Trabajo de Fin de Grado la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE), para cumplir con los requisitos de la asignatura Trabajo Fin de Grado, dentro del Grado en Ingeniería en Tecnologías de la Telecomunicación.

1.4. Estructura del documento

El documento está formado por un total de siete capítulos. El primer capítulo introduce los antecedentes, así como los objetivos del proyecto y otros aspectos formales. El segundo capítulo se centra en introducir el SoC Zynq a utilizar, incluyendo sus características generales y la arquitectura del dispositivo. También, se explica el funcionamiento de la placa de prototipado ZedBoard.

En el tercer capítulo se introduce el flujo de diseño del sistema, explicando brevemente cada una de las herramientas que se han utilizado para la realización del diseño y su cometido. A continuación, se trata el diseño del bloque IP en el capítulo cuarto, teniendo en consideración la arquitectura utilizada, su funcionamiento y ciertas características fundamentales para su correcto desarrollo. No obstante, también se incluyen los pasos necesarios para poder obtener el bloque IP listo para su implementación.

El siguiente capítulo explica la integración e implementación realizada para la obtención del sistema deseado en la placa de desarrollo. La verificación y la validación del sistema se encuentran en el capítulo seis, que incluye los datos reales de funcionamiento del sistema. Para cerrar el documento, en el capítulo séptimo se presentan las conclusiones del Trabajo, exponiendo los resultados obtenidos y propuestas de mejoras para futuras líneas de trabajo. Finalmente, se incluyen las referencias del documento, así como el presupuesto del mismo y el pliego de condiciones del proyecto.

Capítulo 2. El sistema en chip Zynq

Para la realización de este TFG es necesaria la utilización de una plataforma, capaz de cumplir con las necesidades de un acelerador *hardware*. Para ello, se ha seleccionado como placa de desarrollo la placa ZedBoard de Avnet, basada en el dispositivo Xilinx® XC7Z020-1CLG484C Zynq-7000 AP SoC. Se trata de un dispositivo que combina una unidad de procesamiento constituida por dos procesadores ARM Cortex A9, una unidad aceleradora SIMD, memoria L2 y unidades de depurado. Todo ello es lo que se conoce como unidad de procesamiento o PS. Además el SoC incluye una unidad lógica tipo FPGA, en lo que se conoce como Lógica programable o PL: Con ello se dispone de un sistema flexible y configurable, tanto desde el dominio *software* como desde el dominio *hardware*. Por este motivo, se pueden obtener sistemas con diseño de bajo coste, así como de alto rendimiento y bajo consumo de potencia. A su vez, la seguridad y fiabilidad del sistema son más robustos que en otras placas con características similares [11].

Se han realizado diversas comparaciones con otras placas SoC con características similares, y los resultados obtenidos afirman que el dispositivo Zynq-7000 es mejor en rendimiento y potencia, siendo un 25% más rápido el procesador utilizado, un 66% más rápida la lógica FPGA utilizada y con un consumo de potencia 55% menor [12].

También se debe tener en cuenta las herramientas que Xilinx aporta para facilitar la síntesis y la implementación en sus dispositivos. Para la plataforma a utilizar existen las herramientas de diseño llamadas Vivado, que se verán en el siguiente capítulo. Estas herramientas permiten, entre otras características, niveles de abstracción elevados y mejoras para el diseño e implementación de sistemas complejos [11].

2.1. Características generales

Cada dispositivo de Xilinx Zynq está compuesto por un sistema de procesamiento (PS), basado en un procesador *dual-core* ARM Cortex-A9, y una lógica programable (PL). Lo que caracteriza especialmente este sistema es la unión de ambas partes en un único dispositivo,

proporcionando las prestaciones de la computación *hardware* y la flexibilidad aportada por la capacidad de reprogramabilidad del dispositivo [13].

Es de especial importancia la conexión existente entre la lógica programable y el sistema de procesamiento, pues son dos bloques fundamentales que se verán con detenimiento cuando se explique la arquitectura de la plataforma. El ancho de banda máximo acumulado para la comunicación de ambos bloques es de 100 Gbps y se realiza a través de puertos de 64 bits AXI AXP para permitir que la aceleración *hardware* se realice a prestaciones máximas.

La existencia de diversos modos de operación posibilita la mejora de ciertas características del sistema como pueden ser el ahorro de energía, así como la posibilidad de reconfiguración parcial, proporcionando una reducción notable en la lógica programable del dispositivo. También se posibilita la utilización de sistemas operativos en tiempo real (RTOS) dentro del dispositivo Zynq, entre los que se pueden resaltar Linux, FreeRTOS, o Android. A su vez, es posible utilizar dos estructuras diferentes para el PL, pudiendo ser una estructura FPGA Artix-7 destinada al bajo consumo y coste o una estructura FPGA Kintex-7 para mejorar el precio, el rendimiento o la potencia.

No se debe olvidar la capacidad de memoria del sistema, que proporciona una serie de memorias integradas capaces de soportar DDR3-1866 y dispone de una memoria de 512 KB L2 caché y una memoria *on-chip* de 256 KB. Algunos de los periféricos integrados más destacables que se incluyen dentro del dispositivo de Xilinx Zynq son dos USB, dos Tri-mode Gigabit Ethernet, dos SD/SDIO, dos UART, dos CAN 2.0B, dos I2C, dos SPI y una GPIO de 32b.

2.2. Arquitectura

En la Figura 3 se puede observar la arquitectura del dispositivo Zynq, la cual muestra muchas de las características mencionadas en el apartado anterior. Se trata de una arquitectura heterogénea que se ve caracterizada principalmente por dos bloques, el bloque de lógica programable (PL) y el bloque de sistema de procesamiento (PS).

2.2.1. Lógica Programable (PL)

El bloque de lógica programable se centra en aquellas características que son configurables por el usuario. El dispositivo de Xilinx Zynq-7000 viene caracterizado por incluir el mismo PS pero variar el PL y los recursos de entrada/salida en función de la placa seleccionada. El PL para los dispositivos inferiores, como son 7z010, 7z015 y 7z020, está basado en una lógica FPGA Artix-7, mientras que para los dispositivos superiores la lógica FPGA utilizada es Kintex-7. Sin embargo, no siempre se realiza su configuración al inicio de la ejecución y puede ser configurada totalmente o

reconfigurada de forma parcial a lo largo del funcionamiento normal del dispositivo. También dispone de un modo de operación en el que se deshabilita por completo su funcionamiento, proporcionando así al usuario un ahorro de consumo considerable [13].

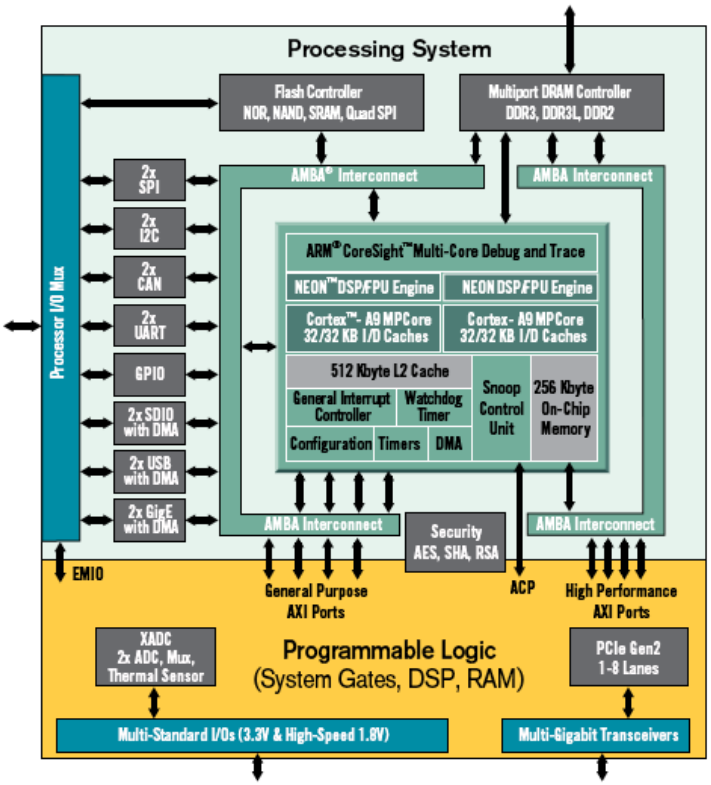

Figura 3: Diagrama de bloques del dispositivo Xilinx Zynq

Dentro de las características más resaltables se encuentran los bloques de lógica configurables (CLB) que incluyen 6 LUTs y capacidad de memoria entre las LUTs. También se incluye una memoria BRAM de doble puerto que puede tener un ancho de 72 bits y puede ser programable como FIFO al introducir lógica adicional. Cada Zynq puede tener entre 60 o 465 BRAMs implementadas, y cada una de estas memorias tiene dos puertos independientes para realizar la lectura y la escritura. Una característica fundamental de las BRAM es que, aparte de ser síncronas, necesitan un único ciclo para realizar una escritura, pero necesitan de dos ciclos para poder obtener el dato de una lectura.

El manejo del reloj es fundamental en la correcta implementación del PL, por ello se debe tener en cuenta la síntesis de frecuencia, así como el desplazamiento de fase. Los *buffers* de alta velocidad permiten aumentar la rapidez del sistema y las transacciones. No se debe olvidar el uso de las entradas y salidas configurables, pues proporcionan un alto rendimiento, a la vez que una alta frecuencia de los condensadores de desacoplamiento. Cuando el sistema en chip Zynq se encuentra en alto rango (HR) las entradas y salidas son capaces de soportar de 1,2 V a 3,3 V. Sin embargo,

cuando se encuentra en alto rendimiento (HP) las entradas y salidas solo soportan de 1,2 V a 1,8 V [14].

También existe un procesamiento de señales digitales con DSP48E1, que permite optimizar aplicaciones con filtros simétricos y permite incluir *pipelining*, ALU y buses dedicados para realizar una estructura en cascada. Los transmisores en HP pueden llegar a velocidades de hasta 12,5 Gbps, excepto el dispositivo 7z015 que solo puede alcanzar 6,25 Gbps. Por último, pero no menos importante, cabe destacar la existencia de un convertidor analógico/digital (XADC) que dispone de 17 entradas analógica flexibles y configurables por el usuario, y de un acceso de JTAG ininterrumpido para comprobar las medidas obtenidas por el convertidor.

2.2.2. Sistema de procesamiento (PS)

En la Figura 4 se aprecia una vista general del sistema de procesamiento del sistema en chip Xilinx Zynq, incluyendo muchas de las características mencionadas anteriormente. En particular, se pueden observar cuatro bloques principales que incluyen las funcionalidades del sistema de procesamiento (PS). Estos bloques son la Unidad de Procesamiento de Aplicación (APU), las interfaces de memoria, los periféricos de entrada/salida (IOP) y la central de interconexiones [13]. A continuación se expondrán las características y funcionalidades más destacables de cada uno de estos bloques.

2.2.2.1. Unidad de Procesamiento de Aplicación (APU)

El sistema de procesamiento de la Zynq incluye no solo el procesador ARM, sino también un conjunto de recursos de procesamiento denominados Unidad de Procesamiento de Aplicación (APU), que es la encargada de proporcionar las características de alto rendimiento y, a su vez, capacita al sistema para cumplir con los estándares correspondientes. La APU está compuesta por un procesador *dual-core* con tecnología ARM Cortex-A9, capaz de proporcionar la posibilidad de configurar el sistema para ejecutar un solo procesador u obtener un multiprocesamiento asimétrico (AMP) o simétrico (SMP). También, permite una gran variedad de modos de operación como pueden ser supervisor, sistema y usuario, ajustando los niveles de aplicación y seguridad de la APU a las necesidades de cada uno de los modos [14].

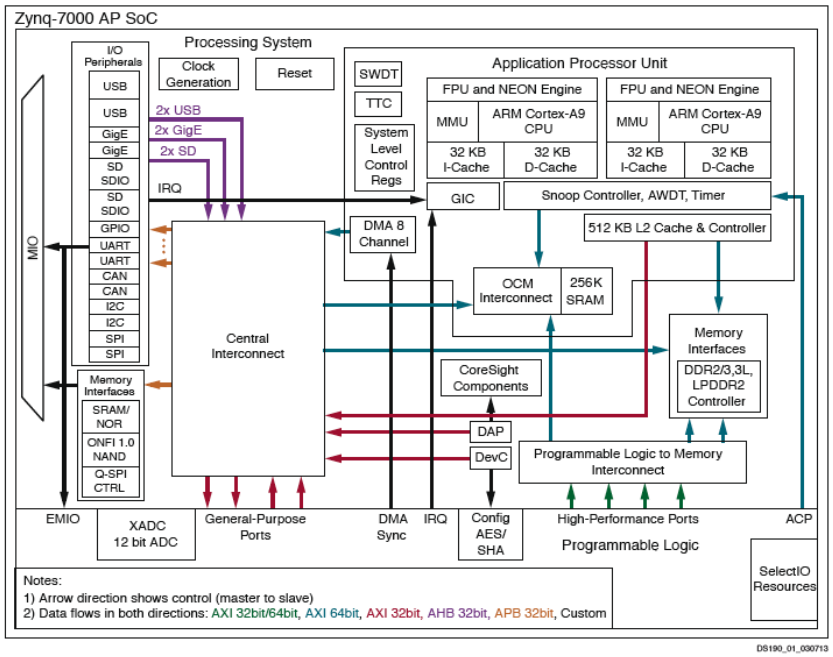

Figura 4: Sistema de procesamiento del dispositivo Xilinx Zynq

Dentro de la APU existen una gran cantidad de características que determinan su funcionalidad. Entre ellas, cabe destacar los siguientes módulos:

- Los Registros de Control de Nivel de Sistema (SLCRs) que permiten controlar por completo el comportamiento del PS.
- La Unidad de Control de *Snoop* (SCU) permite la transparencia de datos y la coherencia entre los procesadores.
- El Puerto de Coherencia del Acelerador (ACP) que realiza la conexión entre los bloques PL y PS, siendo PL el bloque maestro y PS el bloque esclavo.
- El Controlador de Interrupciones Generales (GIC) encargado del manejo de las priorización de interrupciones y de proporcionar las máscaras a las interrupciones individuales. Dispone de interrupciones de periféricos privados (PPI), interrupciones generadas por *software* (SGI), e interrupciones de periféricos compartidos (SPI).
- Temporizadores privados y temporizadores *watchdog*.
- Un controlador DMA que dispone de cuatro canales para PS y otros cuatro canales independientes para PL.
- Una memoria SRAM *on-chip* de 256 KB con paridad y de doble puerto.

2.2.2.2. Interfaces de Memoria

El siguiente bloque que forma parte del PS son las interfaces de memoria, formadas por el controlador DDR, el núcleo controlador DDR y el planificador de transacciones, el controlador Quad-SPI y el controlador de memoria estática (SMC). El controlador DDR es capaz de soportar

memorias del tipo DDR3, DDR3L, DDR2 y LPDDR-2. A su vez, proporciona un ancho de 16 bits o 32 bits y permite el acceso exclusivo de dos IDs diferentes por cada puerto.

El planificador de transacciones se utiliza para conseguir un ancho de banda y una latencia optimizadas, cumpliendo en todo momento con las reglas y estándares AXI. También proporciona una eficiencia para accesos a memoria, tanto para lectura y escritura continua como aleatoria.

El controlador Quad-SPI puede ser individual o dual y dispone de una interfaz APB 3.0 de 32 bits que permite programación, lectura y configuración de las entradas y salidas. También dispone de una interfaz AXI de 32 bits para mapeado de direcciones lineales para operaciones de lectura, gracias a la cual se pueden implementar los modos serie, dual y quad-SPI, así como permite la utilización de dos memorias *flash* SPI. La frecuencia máxima de funcionamiento es de 100 MHz para el modo maestro e incluye una memoria FIFO de entrada de 252 bytes para mejorar la eficiencia y rendimiento del sistema en las lecturas.

Por último, el SMC puede tener diferentes dispositivos de arranque los cuales son: un controlador NAND o un controlador SRAM/NOR paralelo. El controlador NAND se caracteriza por disponer de un ancho para las entradas y salidas de 8 bits o 16 bits. El controlador SRAM/NOR paralelo se caracteriza por un ancho de bus de datos de 8 bits y se puede seleccionar un único chip con 26 señales de direcciones o seleccionar dos chips con 25 señales de direcciones. También dispone de un modo de operación de memoria asíncrona en caso de que sea necesario. Sin embargo, tienen ciertas características en común, pues ambos cuentan con FIFOs de lectura y escritura de 16 palabras, así como de un tiempo de ciclo de entrada y salida programable [14].

2.2.2.3. Periféricos de Entrada y Salida

Los periféricos de entrada y salida (IOP) son un conjunto de interfaces para comunicación de datos externa. Estas interfaces están basadas en estándares de la industria y son las siguientes [14]:

a. GPIO
b. 2 controladores Gigabit Ethernet
c. 2 controladores USB
d. 2 controladores SD/SDIO
e. 2 controladores SPI
f. 2 controladores CAN
g. 2 controladores UART
h. 2 controladores I2C
i. Entradas/Salidas PS MIO

2.2.3. Interconexión entre el PS y el PL

La interconexión entre el sistema de procesamiento y la lógica programable puede variar en función de las tecnologías utilizadas, para la optimización del sistema utilizado, teniendo en cuenta la comunicación implementada y la funcionalidad de los bloques. Las conexiones entre ambos bloques superan las 3000 conexiones y gracias a ello se permite la integración de aceleradores *hardware* en el PL, capaces de acceder a los recursos necesarios en el PS. La interfaz PS-PL es la responsable de la interconexión entre ambos bloques y contiene todas las señales necesarias para la integración de ambos bloques. Estas interfaces son de dos tipos fundamentalmente: interfaces funcionales, disponibles en el PL para conectar con los bloques IP, y señales de configuración, conectadas el PL a lógica fija y proporcionando el control de PS [14].

Las interfaces funcionales son las encargadas de la interconexión de las señales AXI, así como de las interfaces MIO para la mayoría de periféricos de entrada/salida, interrupciones, control del flujo del DMA, relojes e interfaces de depurado. A su vez, las señales de control están compuestas por el puerto de acceso de configuración del procesador (PCAP), el estado de la configuración, SEU y *program/done/init*.

Las interfaces funcionales AXI son de especial importancia y se pueden clasificar en: AXI_ACP, AXI_HP y AXI_GP. AXI_ACP tiene un puerto maestro coherente a PL y se conecta con la SCU para comprobar la coherencia de caché entre la CPU y el PL. AXI_HP dispone de cuatro puertos maestros de alto rendimiento en el PL y AXI_GP dispone de cuatro puertos de propósito general, dos con interfaces maestro y dos con interfaces esclavo. En la Tabla 1 se puede ver un resumen de estas interfaces [13].

Tabla 1: Interfaces AXI de Xilinx Zynq

Descripción de la Interfaz	Nombre de la interfaz	Maestro	Esclavo
AXI de propósito general (AXI_GP)	M_AXI_GP0	PS	PL
	M_AXI_GP1	PS	PL
	S_AXI_GP0	PL	PS
	S_AXI_GP1	PL	PS
Puerto de Coherencia del Acelerador AXI (AXI_ACP)	S_AXI_ACP	PL	PS
Puerto de Alto Rendimiento AXI (AXI_HP)	S_AXI_HP0	PL	PS
	S_AXI_HP1	PL	PS
	S_AXI_HP2	PL	PS
	S_AXI_HP3	PL	PS

Por una parte, el reloj y su funcionamiento son indispensables para que la interconexión entre ambos bloques sea efectiva. El bloque PS a través de los relojes de frecuencia programable (FCLKs) proporciona cuatro relojes al PL que se encuentran físicamente conectados entre ambos bloques. Una ventaja que proporcionan estos relojes es que pueden ser controlados de forma independiente y pueden ser utilizados como una fuente de frecuencia al conectarse con los buffers de reloj proporcionados por el PL. Por otra parte, el *reset* no debe ser olvidado. El PS proporciona, al igual que con el reloj, cuatro señales de *reset* programables conectadas al PL que se programan de forma independiente y tienen total autonomía con respecto a las señales de reloj [13]. Por último, se puede observar en la Figura 5 el subsistema de memoria utilizado en la interconexión de PS y PL y su funcionamiento [3].

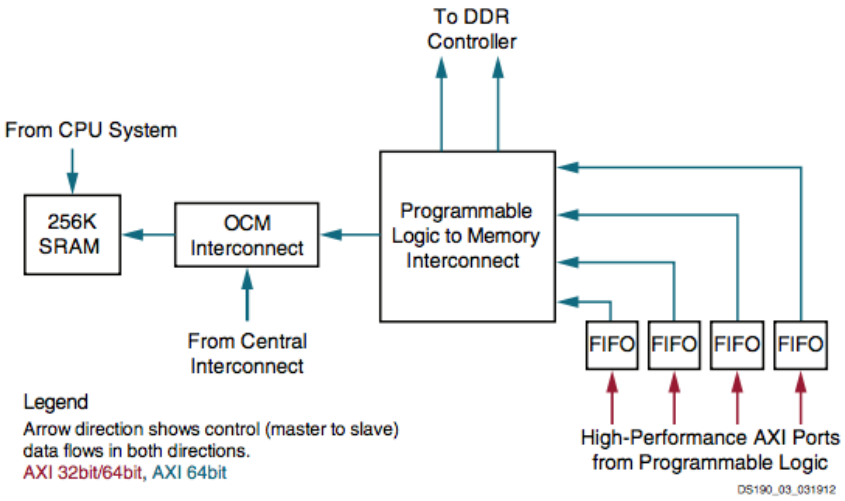

Figura 5: Subsistema de memoria de PS y PL

2.3. ZedBoard

En este Trabajo fin de Grado se ha utilizado la placa ZedBoard como placa de prototipado para comprobar el correcto funcionamiento del sistema. ZedBoard o *Zynq Evaluation and Development Board* es una placa que incluye el dispositivo XC7Z020. La elección de esta placa de prototipado ha sido por su reducido coste y las prestaciones ofrecidas, pues dispone de una gran gama de herramientas de diseño y depurado, así como una facilidad de interconexión con los periféricos implementados. Las posibilidades de la ZedBoard son muy amplias, pues ofrece desde procesamiento de video y control de motores, hasta aceleración *software*, procesamiento de sistemas empotrados y desarrollo de RTOS. En la Figura 6 se muestra una imagen de la placa de prototipado ZedBoard donde se resaltan sus principales componentes [16].

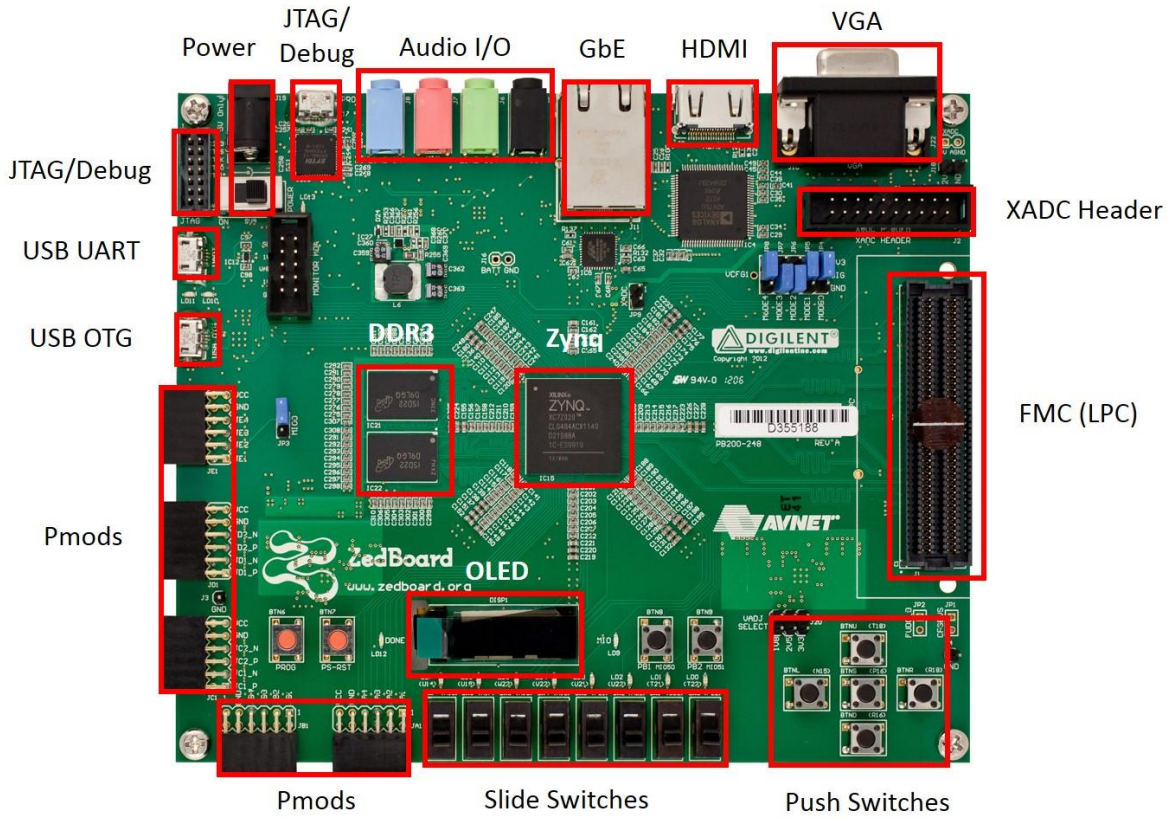

Figura 6: Distribución de los componentes en la placa de prototipado ZedBoard

En la Figura 7 se muestra el diagrama de bloques de la ZedBoard y sus componentes principales. A continuación, se mencionarán algunas de las características más notables de la placa, que pueden diferir con las características vistas en los apartados anteriores. Por ejemplo, las memorias disponibles son una memoria DDR3 de 512 MB, una memoria *flash* Quad SPI de 256 Mb y una tarjeta SD de 4 GB. Es de especial importancia el puerto Ethernet que ofrece velocidades de 10 Mbps, 100 Mbps y 1000 Mbps utilizando RGMII, pues es fundamental en el desarrollo de este TFG [17].

La tensión de funcionamiento del dispositivo es de 12 V. Sin embargo, la programación se realiza a través de un puerto USB-JTAG, desde el cual también se realizan depuraciones del diseño. También existen opciones de configuración auxiliares, que incluyen un JTAG en cascada y una tarjeta SD. La selección de la configuración del sistema viene determinada por una serie de pulsadores, así como *interruptores* y *jumpers* disponibles en la placa. La ZedBoard dispone de una serie de periféricos entre los que destacan los diferentes puertos USB, que son USB OTG 2.0, USB-UART y *bridge* USB-UART, sin olvidar los conectores de expansión FMC, que realizan conexiones directas con interfaces de E/S y el bloque PL. A su vez, la placa dispone de salidas de video HDMI, OLED y una VGA a color de 12 bits, al igual que un *display* OLED de 128x32. Por último, cabe

destacar la posibilidad de utilizar EMIO para realizar la conexión entre los controladores directamente con el bloque PL [16, 17].

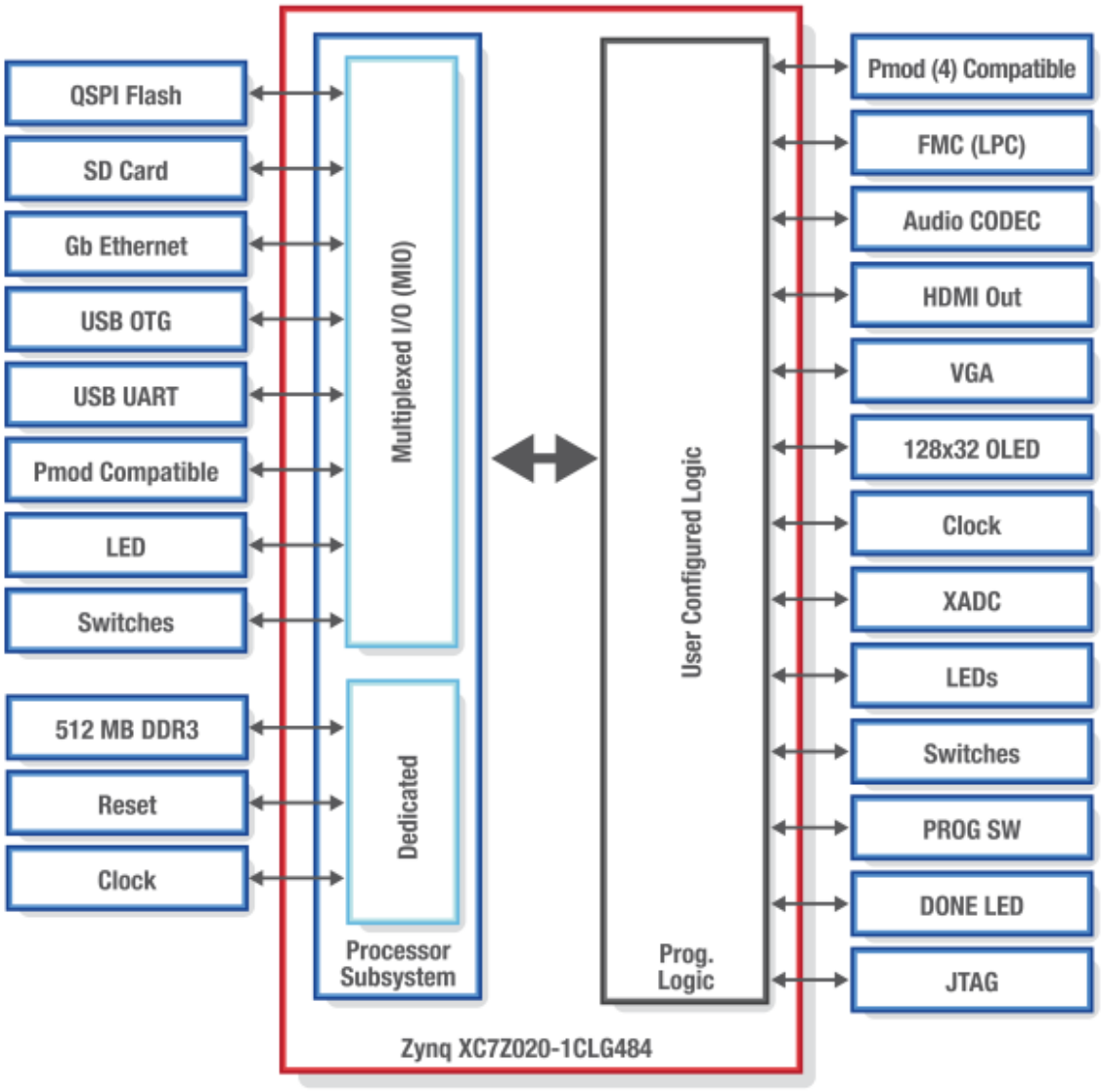

Figura 7: Diagrama de bloques de la ZedBoard

2.4. Conclusiones

Como conclusión de este apartado cabe resaltar la gran versatilidad del dispositivo programable seleccionado. Es capaz de soportar un procesador de doble núcleo ARM Cortex-A9 y el procesamiento del sistema necesario, así como conseguir aplicaciones empotradas optimizadas gracias a un rápido prototipado. Permite conectividad con periféricos USB, UART, IIC y CAN entre otros, la implementación de aplicaciones de video gracias a una salida HDMI incorporada y el desarrollo de aplicaciones de red de 10/100/1000 Mbps.

Al tratarse de un SoC con un alto grado de integración, los costes de producción se ven reducidos, así como facilitan la programación al ya estar realizadas las interconexiones entre los componentes. Existen diseños de referencia, así como *hardware*, bloques IP y herramientas de diseño para facilitar el desarrollo de cualquier aplicación deseada.

El SoC permite la integración de sistemas programables, personalizables y flexibles, adaptándose así a las características requeridas por la aplicación. La división del sistema en los bloques PS y PL y su estrecha interconexión permiten que el sistema sea capaz de tener un alto rendimiento con un coste de potencia muy reducido, debido a la capacidad de reconfiguración que proporciona el PL. Gracias a este estudio se puede realizar la implementación del sistema, teniendo en cuenta sus prestaciones, así como sus limitaciones.

Por último, cabe destacar el uso de la placa ZedBoard como placa de prototipado. Gracias a su reducido coste y las amplias prestaciones disponibles se ajusta a los requerimientos del proyecto. Incluye el dispositivo XC7Z020 de la serie 7 de la familia Zynq y ofrece la posibilidad de usar una gran cantidad de herramientas de soporte para la realización del flujo de diseño del sistema.

Capítulo 3. Flujo de diseño

A lo largo de este capítulo se expondrán las principales características de las herramientas utilizadas para la realización de este proyecto. Estas herramientas son Xilinx Vivado HLS, Cadence SimVision, Cadence CtoS, Synopsys Synplify Premier y Xilinx Vivado Design Suite. Tras describir cada una de las herramientas y sus funcionalidades se podrá exponer el flujo de diseño empleado.

3.1. Xilinx Vivado HLS

La herramienta Vivado High Level Synthesis (HLS) de Xilinx transforma la especificación C en una implementación RTL que se puede sintetizar en una FPGA, permitiendo mejorar la implementación del algoritmo. Las especificaciones C se pueden escribir en los lenguajes de programación C, C++ o SystemC. Vivado HLS es capaz de extraer el control y el *dataflow* del código fuente, permitiendo implementar el diseño en función de las características, las restricciones y las directivas utilizadas. Gracias a ello, se obtiene un diseño *hardware* tras realizar la planificación y las interconexiones necesarias.

Es de especial importancia para la herramienta Vivado HLS encontrar un balance entre el rendimiento y la frecuencia de funcionamiento del sistema. Una vez realizado este balance, ya se centra en minimizar la latencia y el área de utilización. Puede aumentar el paralelismo del diseño para que el rendimiento del sistema sea mayor o realizar un cambio en el tipo de datos y su comunicación para reducir el área utilizada. Sin embargo, el objetivo principal de Vivado HLS es sintetizar una función C en un bloque IP que pueda ser integrado en el sistema *hardware*. En la Figura 8 se muestra el flujo de diseño de la herramienta Vivado HLS, el cual puede describirse en los siguientes pasos a seguir para poder obtener el bloque IP deseado [18, 19]:

1. Compilar, simular y depurar el algoritmo C.
2. Sintetizar el algoritmo C en una implementación RTL.
3. Generar informes detallados y analizar el diseño.
4. Verificar la implementación RTL.
5. Empaquetar la implementación RTL en un bloque IP.

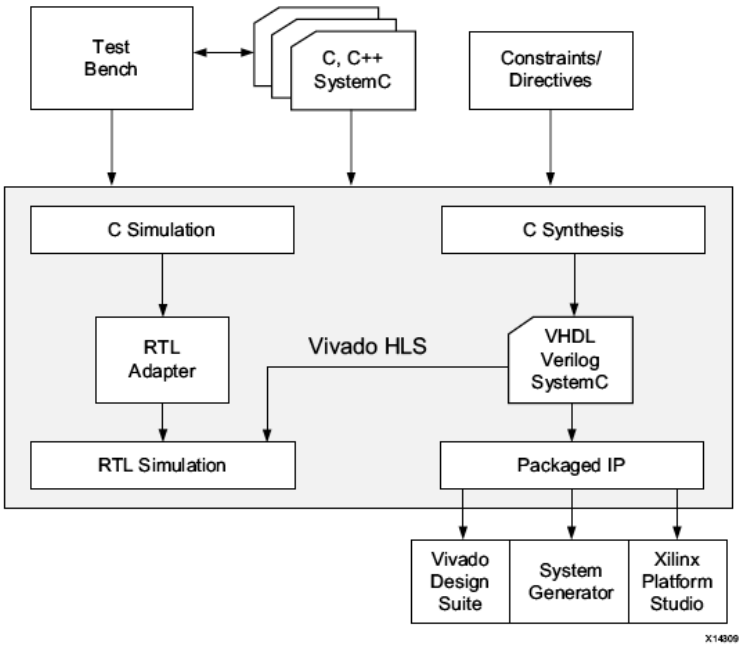

Figura 8: Flujo de diseño de Vivado HLS

Algunas de las características más destacables de la herramienta Vivado HLS incluyen mejorar el rendimiento del sistema a nivel *software* y mejorar la productividad del sistema a nivel *hardware*. Esto se debe a que al mejorar la frecuencia de funcionamiento del sistema, se obtiene un diseño optimizado que tiene en cuenta las directivas, las restricciones y la funcionalidad especificadas para el sistema, y como consecuencia aumenta el nivel de abstracción en el diseño *hardware*. También ofrece la posibilidad de realizar una implementación RTL en VHDL o en Verilog. Gracias a ello se puede realizar el uso de la síntesis lógica, obteniendo la síntesis del diseño RTL en un implementación *gate-level* o un archivo *bitstream* para FPGA. Este último incluye los archivos de diseño de Verilog y VHDL.

Es necesario realizar un proyecto para la utilización de Vivado HLS. Cada proyecto puede contener múltiples soluciones en función de las directivas de optimización, así como las restricciones utilizadas. Esto permite la implementación de diferentes sistemas desde el mismo código fuente. Las directivas de optimización disponibles son las siguientes:

- Realizar *pipeline* para que diferentes ejecuciones de una misma tarea puedan ejecutarse simultáneamente sin necesidad de esperar a que la primera ejecución termine.
- Determinar la latencia de determinadas secciones del código.
- Determinar el número de recursos máximos que pueden ser utilizados.
- Permitir unas operaciones específicas, sin tener en cuenta las dependencias existentes.
- Especificar el protocolo de entrada/salida utilizado.

Para poder realizar la síntesis, optimización y análisis del proyecto es necesario crear un proyecto con la solución inicial y asegurarnos a través de los diferentes *testbenchs* utilizados de su correcto funcionamiento. A continuación, se debe realizar la síntesis y analizar los resultados obtenidos. Tras este paso se pueden realizar cambios en las restricciones del sistema y en las directivas de optimización para obtener diferentes soluciones con exacta validez.

El comienzo de la síntesis en Vivado HLS se caracteriza por una planificación, la cual determinará la frecuencia de funcionamiento, el periodo de reloj y las directivas de optimización del sistema. También es necesario realizar el interconexionado y seleccionar los recursos *hardware* que deben encargarse de cada operación. Por último, es obligatoria la extracción de la lógica de control para obtener una máquina de estados (FSM) requerida para el diseño RTL.

Para la elaboración del diseño RTL cada función del código fuente pasa a ser un bloque RTL, pudiendo ser un módulo en Verilog o una entidad en VHDL. Sin embargo, aquellas funciones que no requieren de procesamientos o recursos elevados, podrán ser enlazadas para permitir que la jerarquía pueda ser más flexible. A través de los tipos de operadores utilizados se definen los recursos *hardware* necesarios para la implementación. Los *arrays* pasan a ser BRAMs de forma automática y los puertos de entrada/salida disponible son los argumentos de la función principal del sistema.

Por último, la validación del sistema es fundamental para el correcto funcionamiento del mismo. Por este motivo existen dos etapas que se realizan antes y después de la síntesis. Como se ha explicado anteriormente, antes de realizar la síntesis del diseño es necesario verificar el correcto funcionamiento del código fuente utilizando *testbench*. Tras la realización de la síntesis y la obtención del diseño RTL la verificación del diseño se lleva a cabo a través de una verificación RTL que se centra en crear una cosimulación del nuevo diseño con el *testbench* utilizado inicialmente.

La herramienta Vivado HLS permite obtener un diseño y una implementación RTL a partir de un código fuente descrito en C. Gracias a la síntesis del diseño, se puede obtener un análisis del rendimiento, el área y la latencia del sistema, permitiendo realizar optimizaciones en el sistema [18].

3.2. Cadence SimVision

Cadence SimVision es un entorno de depurado gráfico unificado para simulación de sistemas electrónicos. La principal ventaja que ofrece es la posibilidad de realizar depuración interactiva de cualquier simulación. Hace uso de Cadence Incisive Enterprise Simulator para la elaboración de múltiples simulaciones y gracias a ello se puede realizar el análisis y el depurado de diferentes

simulaciones de forma simultánea. La cosimulación permite realizar una verificación del sistema mucho más extensa y determinar si la funcionalidad del sistema es la esperada. Por ello, es de especial importancia remarcar la capacidad de SimVision para soportar múltiples lenguajes de descripción de sistemas electrónicos (Verilog, VHDL, SystemC, SystemVerilog o una combinación de ellos) y permitir la depuración de diseños digitales, analógicos y de señal mixta [20, 21].

SimVision se encuentra fundamentado en un entorno capaz de efectuar verificaciones exhaustivas de las simulaciones del diseño. Como consecuencia, permite realizar comprobaciones del comportamiento de los *testbench* en cualquier fase del proceso de verificación, así como de las APIs disponibles. A su vez, SimVision permite el uso de flujos basados en transacciones, así como de flujos de diseño y de verificación. Estos últimos incluyen la búsqueda de código específico, el control del análisis *hardware* del diseño y el análisis de formas de ondas de transacciones y señales mixtas.

En la Figura 9 se aprecia el entorno de trabajo de la herramienta SimVision, el cual está compuesto por una serie de ventanas que se detallan a continuación:

- *Waveform window*: la ventana de formas de onda es una de las ventanas fundamentales en la herramienta SimVision, pues muestra las diferentes señales y variables de forma gráfica a través de un diagrama de ejes X e Y, que muestran los datos y el tiempo. Gracias a esta ventana es posible observar y monitorizar las diferentes señales del sistema.
- *Source Browser window*: la ventana del buscador fuente muestra el código fuente del sistema y permite acceder a él.
- *Properties window*: la ventana de propiedades permite el manejo de los objetos de depurado.
- *Design Browser window*: la ventana del buscador del diseño proporciona la posibilidad de controlar las señales y variables del diseño y monitorizar sus valores.
- *Schematic Tracer window*: la ventana de trazas de esquemáticos muestra el esquemático del diseño y permite definir señales sobre él.
- *Memory Viewer window*: la ventana de visualización de la memoria permite observar los cambios de valores producidos en las memorias del diseño.
- *Watch window*: la ventana de vigilancia es similar a *Design Browser* pero permite realizar el control de las señales y variables del diseño de forma más concreta.
- *Register window*: la ventana de registro permite la utilización de un editor de gráficas para guardar diferentes datos obtenidos en la simulación.
- *Expression Calculator window*: la ventana del calculador de expresiones permite definir expresiones que combinan señales y forman buses o señales virtuales.

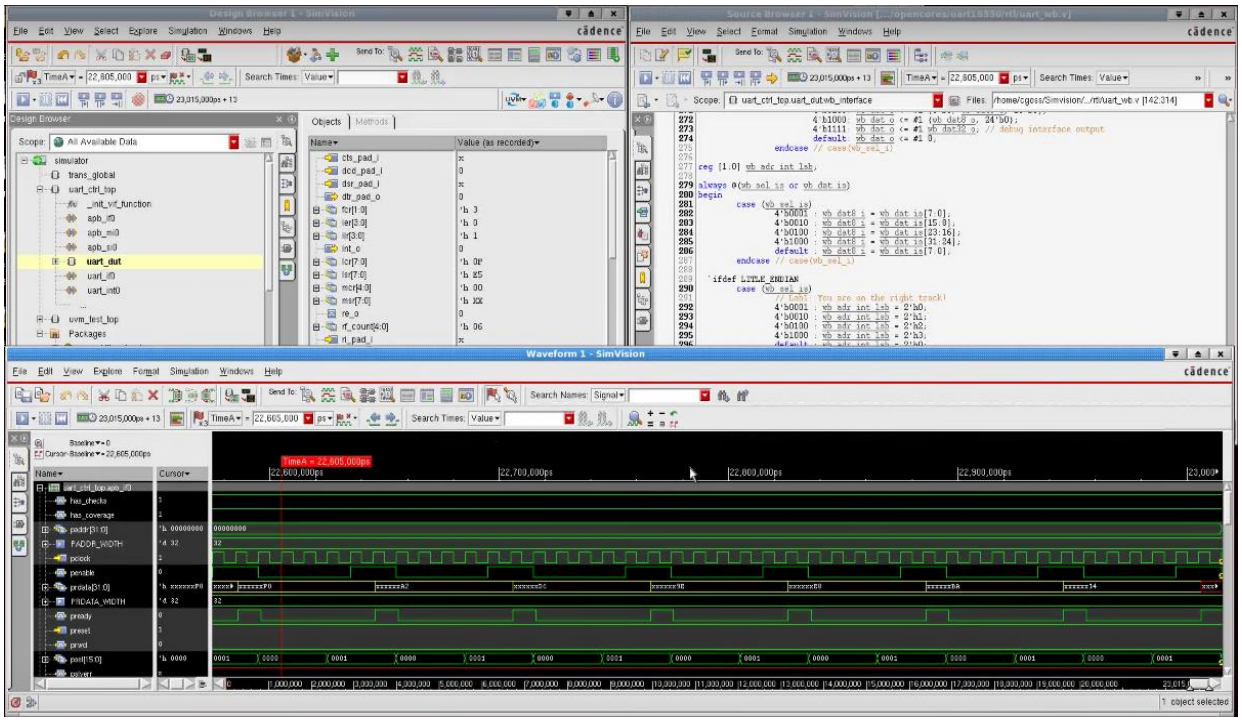

Figura 9: Entorno de la herramienta SimVision

3.3. Cadence CtoS

Cadence C-to-Silicon Compiler o CtoS es un compilador capaz de confeccionar la síntesis RTL y obtener un diseño RTL de un sistema determinado. A partir de códigos descritos en C, C++ o SystemC permite realizar la síntesis de la funcionalidad de la unidad de control o de la ruta de datos o *datapath*. Una de las ventajas más importantes que ofrece CtoS es la posibilidad de crear diseños a partir de niveles de abstracción elevados. Esto produce un aumento de la productividad del diseño, ya que cumple con los requisitos de complejidad y eficiencia del diseño. Gracias al diseño RTL obtenido, la reusabilidad del sistema se ve maximizada debido a la separación de funcionalidades del sistema. Esto permite que las funcionalidades y las restricciones del diseño se encuentren separadas, facilitando la implementación del sistema en diferentes dispositivos sin variar su funcionalidad.

En la Figura 10 se puede observar el flujo de síntesis de alto nivel de la herramienta CtoS, que permite la obtención de un diseño RTL, un diseño FHM[2], y un *wrapper* en SystemC. Los modelos FHM permiten mejorar la coverificación del *hardware* y el *software*, así como reducir el tiempo de desarrollo del *software*. El *wrapper* obtenido se genera de forma automática y permite la verificación del diseño RTL utilizando *testbench* descritos en SystemC. También, se aprecia cómo se pueden obtener *scripts* para realizar flujos de integración y testeado en Calypso SLEC.

[2] Un modelo FHM es un modelo simplificado equivalente al modelo RTL creado para aumentar la velocidad de simulación del sistema.

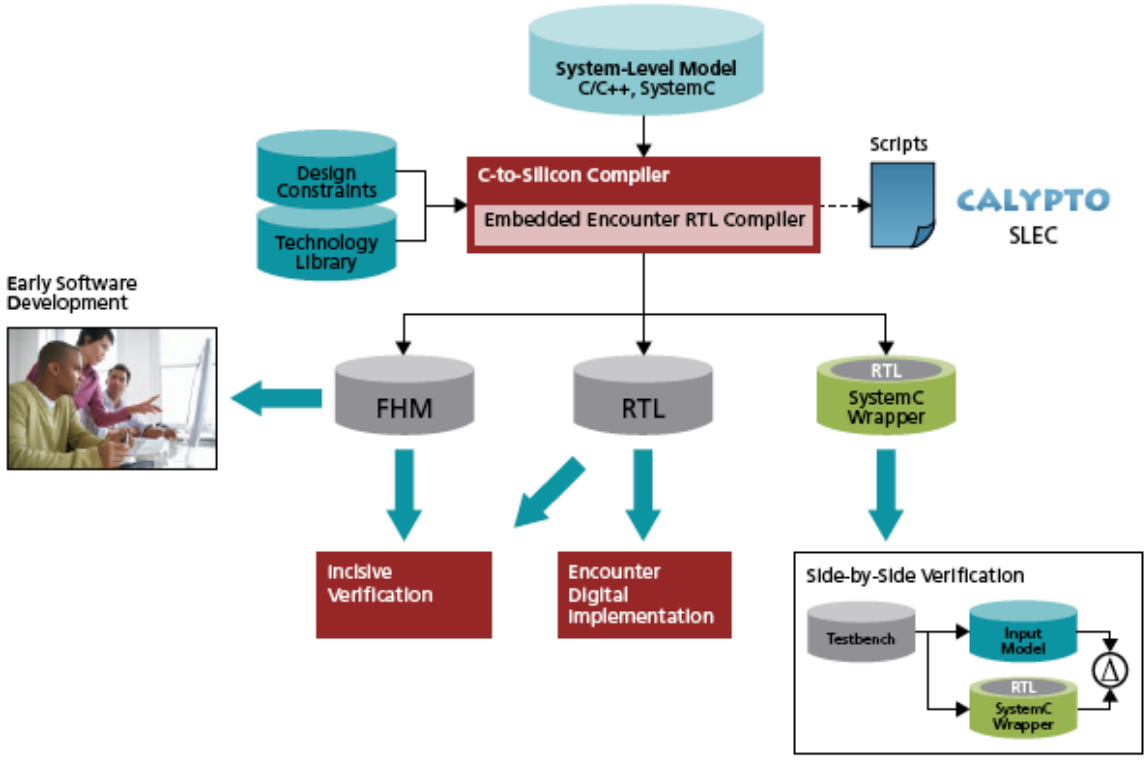

Figura 10: Flujo de síntesis de alto nivel en CtoS

CtoS permite la optimización del sistema durante la síntesis y crea una serie de archivos que permiten el análisis del sistema y la síntesis incremental, incluyendo gráficos de control y *dataflow*, e informes de la temporización del sistema. No obstante, hay cuatro características del CtoS que son de especial importancia [22]:

- La síntesis ELS permite la optimización paralela de la lógica de control y del *datapath*. Gracias a ello, verifica la correcta realización de la síntesis y ahorra tiempo de cómputo.
- La base de datos BST permite una síntesis incremental y una reducción de tiempo en el diseño y la verificación del sistema.
- CFS proporciona la reutilización de múltiples aplicaciones y tecnologías.
- Los modelos FHM son autogenerados en la síntesis, que permiten la coverificación del sistema.

3.4. Synopsys Synplify Premier

Synopsys Synplify Premier es un entorno que permite la síntesis lógica y el análisis de diseños orientados a ser implementados en FPGA. Su objetivo principal es acelerar la implementación de los diseños y prototipos basados en FPGA. Gracias a la síntesis lógica se realizan mejoras en el funcionamiento del sistema y en su diseño, optimizando el tiempo de ciclo, un mayor

rendimiento en cuanto a la reducción del área utilizada y la reducción de potencia. También da la posibilidad de utilizar metodología de diseño jerárquica y proporciona un depurado eficiente.

En la Figura 11 se muestra el flujo de diseño de la herramienta Synplify Premier. A partir de diseños RTL, diseños *third-party* o ficheros de diseños IP anteriores se puede realizar la síntesis lógica. Se deben tener en cuenta las restricciones de diseño y de tiempo del sistema para la realización de la síntesis. Posteriormente, se puede efectuar el empaquetado y conexionado del sistema sobre la FPGA deseada para poder obtener finalmente el *bitstream* necesario para la programación de la plataforma.

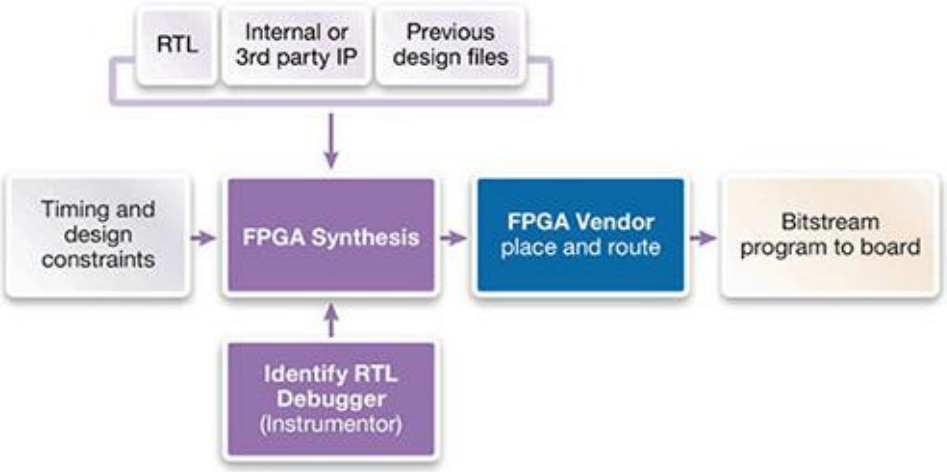

Figura 11: Flujo de diseño de Synplify Premier

Las características más destacables de Synplify Premier son las siguientes [23, 24]:

- Permite realizar síntesis distribuida, que realiza la síntesis de dispositivos múltiples o individuales.
- Dispone de un depurador RTL que permite la detección de errores funcionales.
- Contiene un depurado y un análisis de diseño avanzado a través de un analizador HDL y flujos de depurado jerárquicos.
- Proporciona la posibilidad de manejar múltiples implementaciones de diseño.
- Mejora la síntesis lógica, y a su vez los resultados temporales.
- Permite realizar diseños de alta fiabilidad para sistemas automáticos y sistemas donde la seguridad es crítica.
- Dispone del estándar SDC (Synopsys Design Constraints) que es óptimo para las restricciones temporales.
- Para mejorar los resultados, incluye extracción, optimización y depurado FSM que puede ser controlado por el usuario.

- Soporta una gran variedad de lenguajes, donde destacan VHDL, Verilog y SystemVerilog.
- Proporciona en consumo de potencia de forma automática para las BRAMs de la FPGA que estén inactivas.
- Realiza optimización de potencia de forma automática para las primitivas de Xilinx DSP48.

3.5. Xilinx Vivado Design Suite

Xilinx Vivado Design Suite es un entorno de herramientas de diseño que permite la implementación de sistemas descritos en lenguaje de alto nivel en plataforma *hardware* tipo FPGA. Una de las bases fundamentales de este entorno es la optimización y la búsqueda de la mejora del uso de recursos, así como la maximización del rendimiento del sistema. Vivado Design Suite facilita el diseño de sistemas para múltiples dominios de aplicación, con una rápida implementación para aquellos sistemas que utilizan la serie 7 de Xilinx, las placas de desarrollo Zynq-7000 o los dispositivos UltraScale, teniendo en cuenta la productividad del diseño, la integración y la implementación de los sistemas.

Las tareas que incluye Vivado Design Suite son la partición del *hardware* y el *software*, el desarrollo de un diseño para la plataforma *hardware*, la obtención de *software* empotrado, así como la integración, la programación y el depurado del sistema en el dispositivo seleccionado. Teniendo en cuenta lo amplio que es este entorno de herramientas de diseño, solo se hará hincapié en aquellos aspectos fundamentales para el desarrollo de este proyecto. Por este motivo, se explicará a continuación el flujo de diseño, los bloques IP, la plataforma y el prototipado y la herramienta SDK [25].

3.5.1. Flujo de diseño

Para la integración de la solución deseada es necesaria la utilización de una interfaz de usuario gráfica (GUI), que en el entorno de Vivado Design Suite es un entorno de diseño integrado (IDE). El flujo de diseño tradicional suele basarse en obtener el *bitstream* para la configuración de la FPGA a partir del diseño RTL, pero Vivado Design Suite añade otras posibilidades que se centran en obtener bloques IP (Intellectual Property) para realizar el diseño. Por ello, la principal función del IDE de Vivado es proporcionar al usuario la posibilidad de realizar el ensamblaje, la implementación y la validación de su sistema y de los bloques IP. El IDE de Vivado permite generar el análisis del sistema y asignar restricciones a través del proceso de diseño e incluye la posibilidad de crear un diseño abierto en memoria, incluso tras haber realizado la síntesis e implementación RTL. Gracias a ello, se pueden efectuar cambios en cada etapa del diseño en las restricciones, en la lógica y en la configuración del dispositivo. En la Figura 12 se observa el flujo de diseño en alto nivel para Vivado Design Suite [25].

El flujo de diseño se centra en el diseño del bloque IP y su implementación e integración, incluyendo su configuración y su verificación. Los pasos que conforman el flujo de diseño son la síntesis, la implementación, el análisis temporal y de potencia y la generación del *bitstream*.

La fase de implementación comienza con la simulación lógica, la asignación de recursos, tanto físicos como lógicos, y la síntesis lógica. Finalmente, tras obtener el *bitstream* necesario para la exportación a SDK se puede llevar a cabo la programación y el depurado en la plataforma [26].

Sin embargo, Vivado Design Suite proporciona dos tipos diferentes de flujos de diseño: flujo RTL o flujo de síntesis con herramientas externas que permite la síntesis e implementación a partir de un archivo "EDIF" o Verilog estructural. El flujo RTL proporciona la síntesis e implementación para los archivos de código del tipo Verilog, VHDL, SystemVerilog, XDC o basados en C.

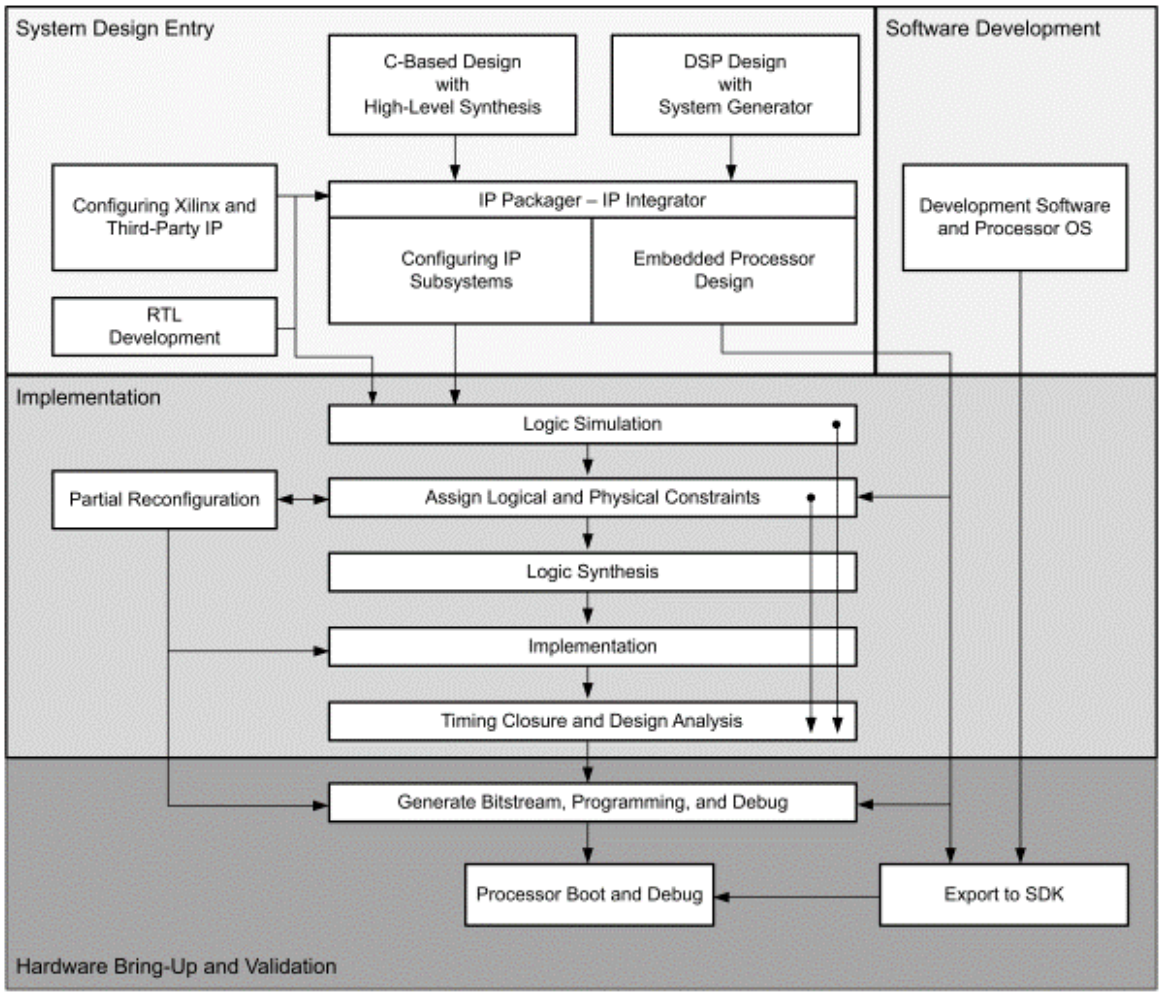

Figura 12: Flujo de diseño de Vivado Design Suite

3.5.2. Diseños basados en bloques IP

Uno de los flujos de diseño que permite utilizar Vivado Design Suite está basado en el uso de bloques IP y en la estandarización de la arquitectura de comunicaciones. Se centra en realizar el diseño del sistema incluyendo diferentes bloques IP, como pueden ser periféricos y procesadores concretos o de uso general. Para la obtención del diseño RTL deseado. Con este flujo de diseño es necesaria la utilización de diferentes herramientas como son IP Packager e IP Integrator, entre otras.

Vivado Design Suite proporciona la posibilidad de añadir diferentes módulos IP al diseño a partir de diseños DSP, de diseños generados en Vivado HLS, a partir de un archivo "EDIF" o diseñados como bloque IP en la propia herramienta.

El entorno de desarrollo ya proporciona una amplia gama de bloques IP estandarizados que se encuentran en el catálogo de Xilinx. En la Figura 13 se puede observar el flujo de diseño basado en bloques IP. Tras la adición de los paquetes IP al sistema, se pasa al empaquetado de los bloques IP y a su integración. Para ello, se debe utilizar la herramienta IP Packager que es la encargada de realizar la conversión de los diferentes módulos o modelos incorporados en bloques IP aptos para su uso en el entorno de desarrollo. Después, es necesario el uso de la herramienta IP Integrator, que proporciona un entorno de trabajo para la generación del esquemático del sistema. Se incluyen los bloques IP deseados y se realizan las interconexiones correspondientes para obtener el sistema completo que posteriormente será implementado en la plataforma seleccionada. A continuación, ya es posible realizar la generación de los productos de salida, los cuales incluyen la configuración del bloque IP, incluyendo las restricciones del sistema, el HDL y los objetivos concretos para la simulación. Por último, tras la obtención de los bloques IP configurados también es posible elaborar el diseño RTL [27].

Vivado Design Suite permite realizar subsistemas IP utilizando el protocolo AMBA AXI4. Gracias a la estandarización de este protocolo se permite la creación de sistemas de forma rápida y flexible, acorde a las necesidades del mercado. La creación de subsistemas se basa en incluir varios IP dentro de un único bloque IP, para facilitar su utilización e implementación. Tras realizar la validación y empaquetar los diferentes IP, obtenemos un solo bloque IP. Todo ello se efectúa de forma interactiva aplicando conexiones, al igual que en un esquemático.

3.5.3. Plataforma y prototipado

Para poder obtener el diseño final en la plataforma y confeccionar su prototipado es necesario realizar algunos pasos después de la obtención del diseño RTL. En la Figura 14 se puede observar el flujo de diseño de Vivado Design Suite. Las primeras etapas del diseño ya han sido explicadas con

anterioridad, hasta llegar a la obtención del diseño RTL, el cual debe pasar por las fases de síntesis, implementación, programación y depurado.

La implementación del sistema en la plataforma seleccionada también es posible gracias a Vivado Design Suite. Recordamos que los únicos dispositivos con los que trabaja son UltraScale y la serie 7 de Xilinx. A partir de diseños RTL, diseños *Netlist* o diseños basados en bloques IP se puede realizar la implementación. Para ello es necesario tener un diseño sintetizado en memoria para iniciar la implementación.

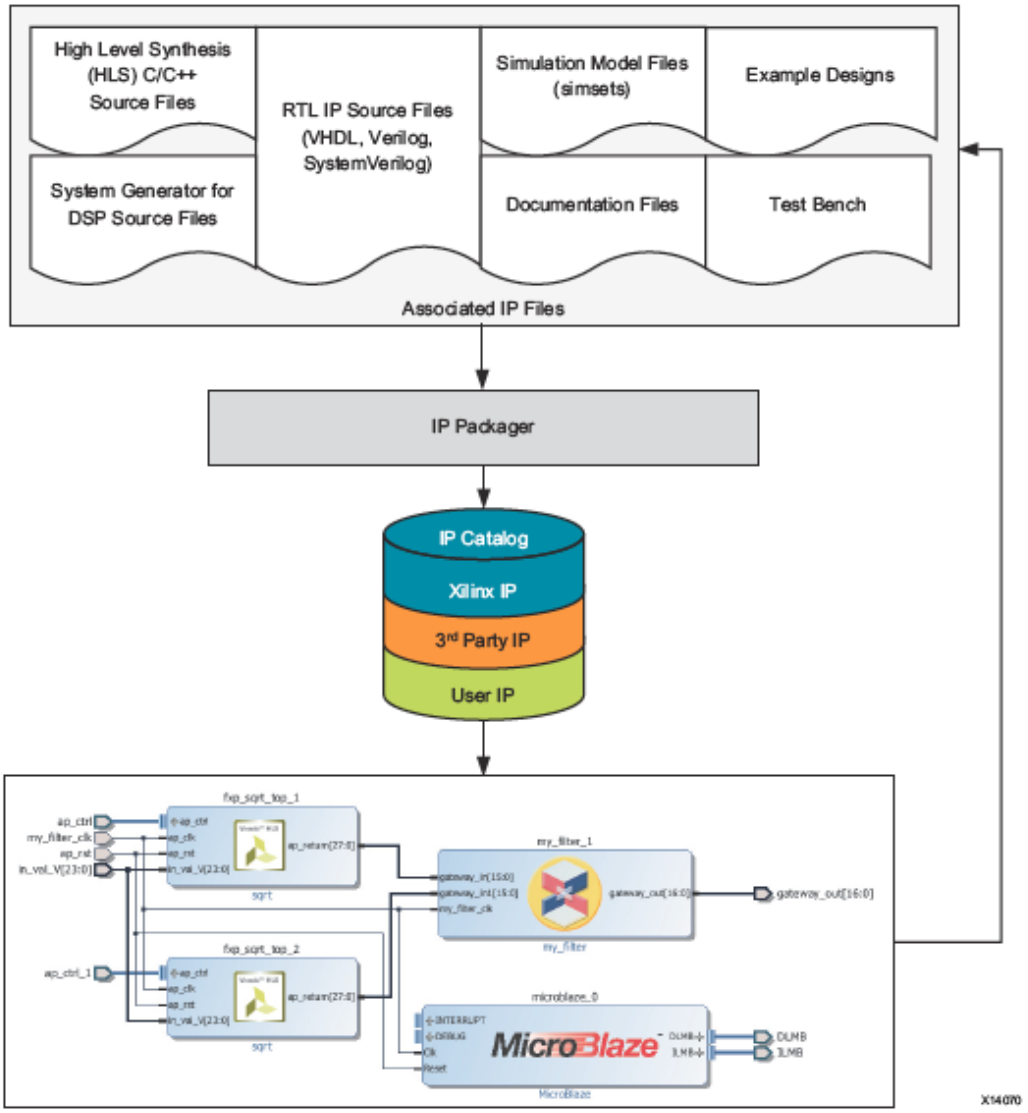

Figura 13: Flujo de diseño basado en bloques IP

En este paso se puede comenzar el flujo de implementación, pero antes se deben determinar las características del sistemas y sus restricciones, haciendo uso del estándar SDC y de XDC. Sin embargo, no es necesario determinar las características del sistema si no es requerido por el programador. Para llevar a cabo la implementación es necesario incluir los resultados de la síntesis.

En proyectos cuya síntesis ha sido realizada en Vivado no es necesario añadir estos archivos, pues ya se encuentran disponibles; pero para aquellos diseños que no lo tengan será necesario añadir una lista de conexiones sintetizada, que pueden ser de tipo Verilog estructural, SystemVerilog estructural, EDIF y DCP sintetizado. Finalmente, tras elaborar la implementación se obtiene un diseño completo del sistema, válido para la generación del *bitstream* necesario para la programación y el depurado del sistema [28].

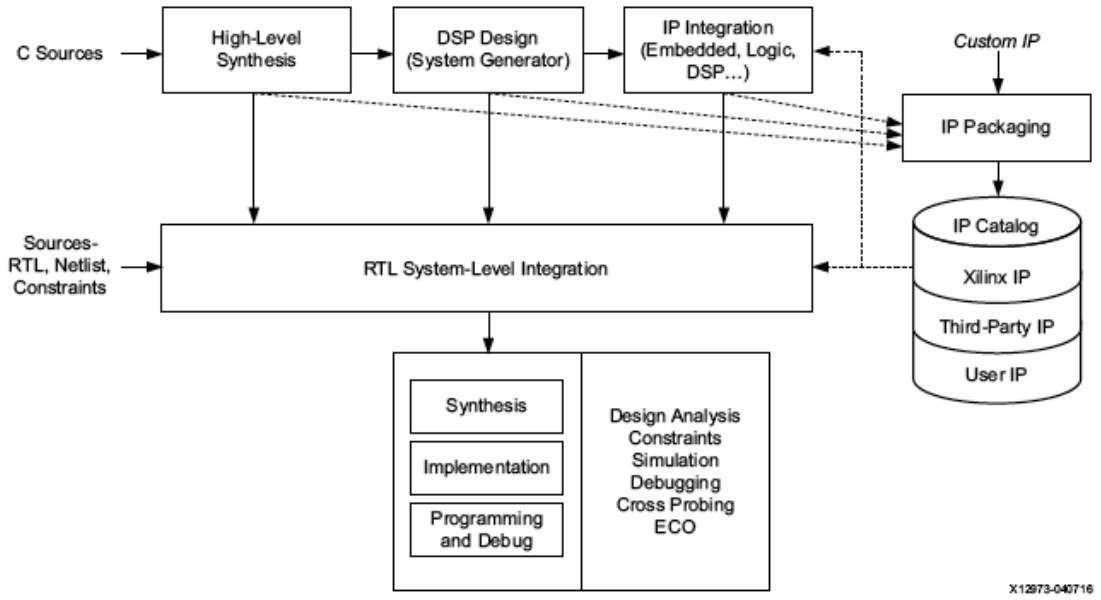

Figura 14: Flujo de diseño de alto nivel de Vivado Design Suite

3.5.4. SDK

Vivado Design Suite proporciona un entorno de desarrollo de *software* empotrado, conocido como SDK. Tras la implementación del diseño es necesario pasar a la programación de la plataforma FPGA seleccionada y a su depuración. SDK proporciona la posibilidad de crear aplicaciones que posteriormente puedan ser integradas en el dispositivo seleccionado. Algunas de las características más destacables de la herramienta son un entorno de compilación y un editor de código aptos para C y C++, un controlador de proyectos, facilidad de depurado y elaboración de perfiles para sistemas empotrados y control sobre la versión del código fuente.

Sin embargo, existen ciertas peculiaridades de la herramienta SDK que permiten el uso de depuradores XMD y XSDB, así como la programación de FPGA utilizando el *bitstream* o la programación de dispositivos *flash*.

El flujo de trabajo de SDK está formado por una serie de pasos. Primero se debe invocar al SDK y crear un nuevo espacio de trabajo. Posteriormente, se realiza la creación de un archivo BSP que contiene la colección de drivers y librerías utilizadas por la aplicación. Tras disponer del *bitstream* y el BSP ya es posible programar la placa de desarrollo y comprobar su correcto funcionamiento. No obstante, también es posible realizar una verificación del sistema antes de integrarlo en la plataforma y comprobar que funcione de forma correcta. Una vez se desea efectuar la programación de la plataforma se puede generar un *bootloader* que será el encargado de iniciar la aplicación una vez programada en la plataforma [29].

3.6. Determinación del flujo de diseño

A lo largo de este capítulo se han explicado las diferentes herramientas de diseño e implementación utilizadas para la realización del proyecto, incluyendo Vivado HLS, SimVision, CtoS, Synplify Premier y Vivado Design Suite. A continuación se describe el flujo de diseño utilizado para este proyecto y la utilización de cada una de las herramientas mencionadas anteriormente. Pero antes, se debe aclarar qué herramienta de síntesis de alto nivel va a utilizarse. Existen dos herramientas que se encargan de la síntesis del diseño y la obtención del diseño RTL: Vivado HLS y CtoS.

Tras el estudio de ambas herramientas se ha decidido utilizar la herramienta CtoS para la síntesis del diseño debido a las ventajas y mejoras que presenta sobre Vivado HLS. Para el diseño del modelado se ha utilizado una metodología CABA (Cycle Accurate Bit Accurate), donde la precisión a nivel de ciclo es de especial importancia para la implementación del protocolo. Por ello, las nociones temporales pasan a ser indispensables en el desarrollo del bloque UD y se realiza la selección de SystemC frente a C o C++ como lenguaje de programación de alto nivel. CtoS permite un mejor soporte para el modelo diseñado en SystemC cuando se precisa tener un control estricto sobre los ciclos de las interfaces de entrada y salida.

Una vez aclarada esta elección, se procede a explicar el flujo de diseño utilizado. Tras la obtención del código fuente y los *testbench* correspondientes, que han sido descritos en SystemC, se procede a la verificación del código con la herramienta SimVision. Tras comprobar el correcto funcionamiento del sistema se realiza la síntesis del sistema en CtoS, obteniendo así el diseño RTL. Antes de continuar con las siguientes fases del flujo de diseño, se vuelve a utilizar la herramienta SimVision para realizar la cosimulación del sistema en SystemC y RTL. Tras verificar su correcto funcionamiento es posible efectuar la síntesis lógica del diseño en Synplify Premier, obteniendo como resultado un archivo EDIF, necesario para la creación del bloque IP en Vivado Design Suite. Una vez obtenido el bloque IP, se procede a realizar el diseño de la plataforma incluyendo nuestro

bloque, los bloques necesarios para su funcionamiento y las interconexiones pertinentes en Vivado Design Suite. Finalmente, se obtiene el *bitstream* del sistema, que será implementado en la herramienta SDK para su programación en la plataforma y su posterior verificación. Todo este flujo de diseño puede verse de forma más gráfica en la Figura 15.

3.7. Conclusiones

A lo largo de este capítulo se ha elaborado un estudio de las diversas herramientas necesarias para la realización del flujo de diseño del sistema. Inicialmente, la herramienta Cadence SimVision realiza la simulación y cosimulación del sistema para permitir su verificación. Cadence CtoS permite realizar la obtención del diseño RTL y Synopsys Synplify Premier la elaboración de la síntesis lógica. Por último, Xilinx Vivado Design Suite permite realizar el diseño de la plataforma para su posterior programación sobre la misma. Con todas estas herramientas se ha conseguido un flujo de diseño integrado que como consecuencia ha permitido la realización de este proyecto.

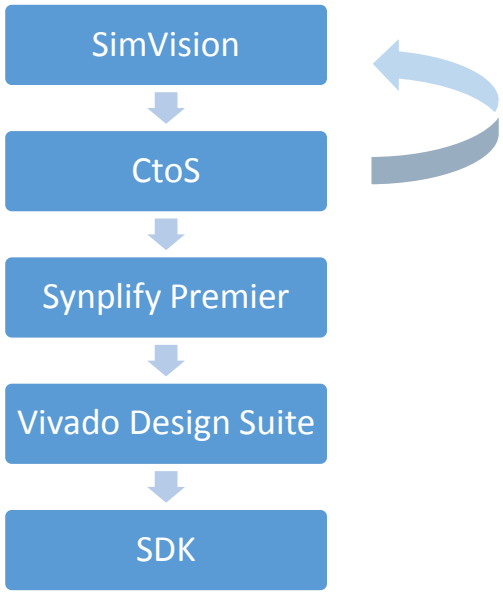

Figura 15: Flujo de diseño de la UD

Capítulo 4. Diseño del bloque IP

Este capítulo está dedicado a la descripción del funcionamiento de la Unidad de Despacho (UD) diseñada, incluyendo la arquitectura elegida para su desarrollo. También se realizará un análisis de los recursos utilizados y de la frecuencia de funcionamiento, y se explicarán las comunicaciones del bloque, para finalmente explicar los pasos a seguir necesarios para la obtención del bloque IP.

4.1. Arquitectura de la Unidad de Despacho

Debido a la complejidad del bloque, se ha decidido desarrollar dos Unidades de Despacho independientes pero idénticas a nivel funcional y estructural. Las unidades de red con las que se quiere realizar la conexión son bloques MAC Ethernet, como se observa en la Figura 16. Cada una de las UD puede estar conectada a un bloque MAC Ethernet emisor y un bloque MAC Ethernet receptor. Esta decisión se ha tomado para facilitar el desarrollo de la Unidad de Despacho y conseguir rapidez y eficiencia en el funcionamiento normal del bloque. Por ello, la unidireccionalidad de la UD permitirá la independencia de cómputo, sin necesidad de entrelazar sus procesos o esperar a que se encuentre libre la UD para permitir la transferencia de información. Gracias a esto se consigue un código más sencillo, simple y dinámico, a la vez que eficaz y rápido.

La arquitectura de la UD se puede observar en la Figura 16. En ella se muestra cómo existe una entrada única para el bloque MAC Ethernet emisor y una salida única para el bloque MAC Ethernet receptor. También existen múltiples entradas y salidas para los bloques IP aceleradores que procesarán los paquetes almacenados en la UD. Por cada bloque IP acelerador que se desee integrar en la plataforma debe existir un puerto de entrada y un puerto de salida independientes al módulo de la UD. Por último, existe una entrada independiente para el puerto de configuración encargado de determinar el modo de operación de la UD y los parámetros necesarios para que esta sea capaz de funcionar con normalidad. Las comunicaciones de todos estos puertos se realiza mediante buses que incluyen el protocolo AXI4, que se explicará más adelante en este capítulo. Para los bloques MAC Ethernet y los bloques IP aceleradores se utilizará el protocolo AXI4-Stream, mientras que para el puerto de configuración será necesario implementar el protocolo AXI4-Lite.

También es necesario incluir dos memorias FIFOs al diseño, las cuales se han denominado FIFO1 y FIFO2. Estas memorias se encuentran conectadas a la UD a través de puertos de entrada y salida, para poder realizar las operaciones de lectura y escritura. Para cada UD implementada serán necesarias las dos memorias, pues almacenarán los datos recibidos de cada bloque MAC Ethernet emisor. La memoria FIFO1 será de 2048 posiciones de 33 bits, que incluyen 32 bits de datos y un bit que indica la finalización o no del paquete. La memoria FIFO2 tendrá la mitad de posiciones, 1024, pero almacenará palabras de 65 bits, 64 bits de datos y un bit que cumple con la misma función que en la memoria anterior. Finalmente, los puertos de entrada y salida del bloque deben incluir una cola FIFO independiente al bloque que se quiere desarrollar. Esta cola FIFO irá almacenando los datos recibidos por el bloque MAC Ethernet emisor para permitir al sistema la posibilidad de funcionar a una frecuencia diferente a la del bloque MAC Ethernet. Al ser necesarias dos UD para realizar las comunicaciones de forma exitosa, cada una de ellas dispondrá de una cola FIFO externa propia.

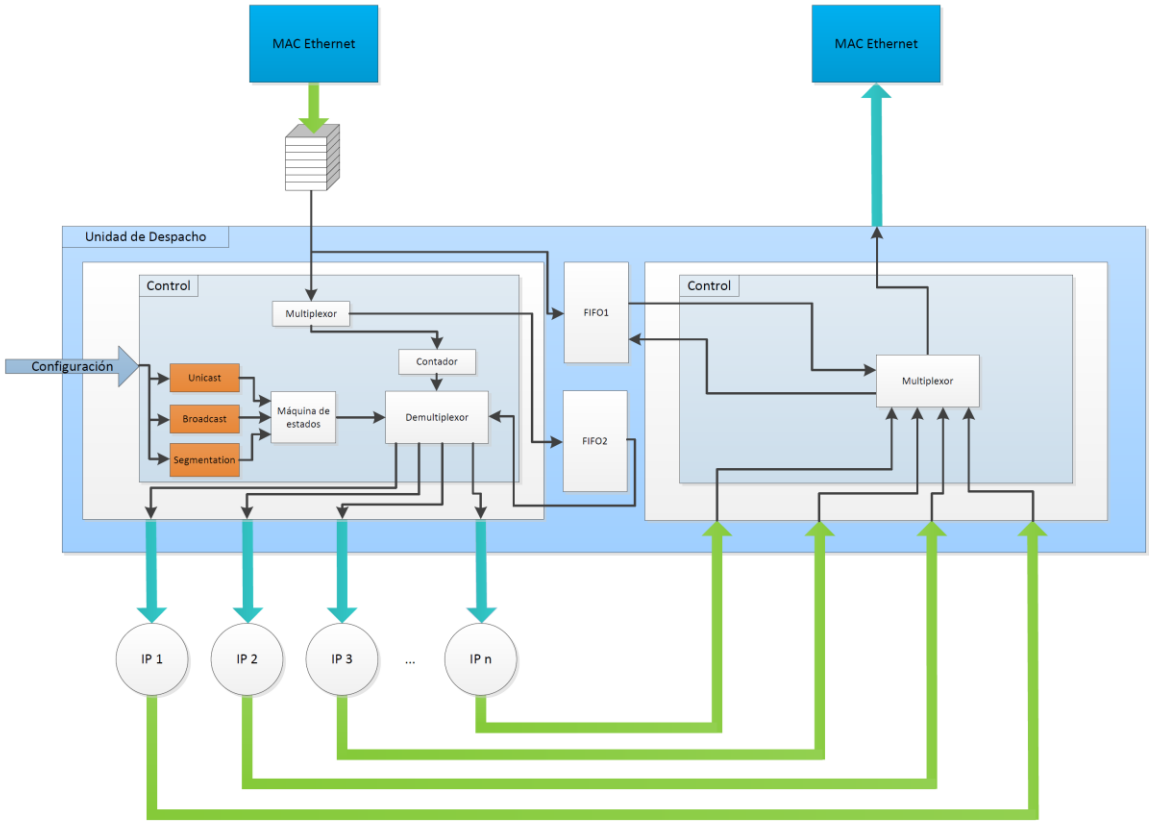

Figura 16: Arquitectura de la UD

Inicialmente los datos tras ser almacenados en la FIFO externa y entrar en la UD son almacenados sin realizar modificaciones en la FIFO1 y gracias a un multiplexor se combinan dos palabras y se almacenan en la memoria FIFO2. Esto es necesario, porque los datos recibidos del bloque MAC Ethernet emisor son de 32 bits, pero los bloques IP aceleradores tienen puertos de

entrada y salida que requieren datos de 64 bits. A continuación, los datos pasarán al contador, que en función del modo de operación actuará de dos maneras diferentes. Si el modo de operación seleccionado es *Unicast* o *Broadcast*, simplemente dejará pasar los datos al demultiplexor sin realizar ningún cambio sobre ellos. Sin embargo, si el modo de operación es *Segmentation*, se encargará de dividir el bloque en los segmentos necesarios para luego enviarlos al demultiplexor.

No obstante, dentro de la UD se incluye una máquina de estados finita que será la encargada de determinar el modo de operación seleccionado por el usuario, a través del puerto de configuración, e implementar las modificaciones pertinentes en el módulo. Gracias a las señales producidas por la máquina de estados se determinan los bloques IP aceleradores a los que se debe enviar el paquete, cuya transmisión se producirá a través de un bloque demultiplexor y se comunicará directamente con el contador para así poder determinar cómo debe operar este.

Posteriormente el demultiplexor, se encargará de realizar la distribución de los paquetes a los diferentes bloques IP aceleradores, según se haya determinado por la máquina de estados y las direcciones transmitidas. Para enviar los paquetes se accederá a la memoria FIFO2 para poder enviar datos con palabras de 64 bits. Los bloques IP aceleradores analizarán los datos recibidos y devolverán una respuesta que la UD deberá procesar. Las respuestas de cada bloque IP serán almacenadas dentro de su propia cola de salida tipo FIFO. Gracias a ello, no será necesaria una cola FIFO independiente para cada bloque IP y se podrán ahorrar recursos *hardware*.

Por último, será necesario el uso de un multiplexor que se encargue de acceder a la memoria FIFO1 para poder transmitir los diferentes mensajes almacenados del bloque MAC Ethernet emisor. Si tras el procesamiento de los diferentes bloques aceleradores IP se determina que se debe enviar el mensaje al bloque MAC Ethernet receptor, se obtiene de la memoria FIFO1 el mensaje correcto y se realiza su transmisión. En caso contrario, el mensaje se desecha liberando espacio de memoria para posteriores mensajes.

4.2. Configuración de la UD

La configuración de la UD es una de las partes fundamentales que aporta flexibilidad al sistema. Está soportada mediante una interfaz AXI4-Lite por la que se recibe la configuración desde el procesador y por un conjunto de registros y máquinas de estado de la UD. Uno de los parámetros principales de la configuración es el modo de operación con el que la UD ejecutará su funcionamiento. Los modos de operación disponibles e implementados son tres:

- **Modo de operación *Unicast***. Es el modo de operación más sencillo incluido dentro de la UD. Únicamente necesita la dirección del bloque acelerador IP al que se desea enviar todos

los mensajes recibidos. Esto implica una comunicación única y exclusiva con un solo bloque acelerador IP que procesará la información recibida.

- **Modo de operación *Broadcast***. Su funcionamiento es similar al del modo de operación *Unicast*, sin embargo envía los mensajes entrantes a todos los bloques aceleradores IP conectados a la UD. De esta manera, la UD debe esperar la respuesta de cada uno de ellos antes de decidir si el paquete debe ser enviado al bloque MAC Ethernet receptor o debe ser descartado. La ventaja de este modo de operación es que, en prácticamente la misma latencia, se ha podido procesar el mensaje en búsqueda de múltiples coincidencias y se ha conseguido realizar un análisis más extenso y complejo aprovechando el paralelismo *hardware* incluido en el sistema. También es posible enviar la información a aquellos bloques aceleradores IP conectados a la UD definidos por el usuario en la configuración. Con ello se incluye selectividad en el envío de paquetes y por tanto, gracias a ello, se podrá habilitar o deshabilitar ciertos bloques aceleradores IP a demanda.

- **Modo de operación *Segmentation.*** Es el modo de operación más complejo de los tres disponibles. Se encarga de dividir el paquete completo en diferentes segmentos o *flits* que se enviarán a diferentes bloques aceleradores IP para su procesamiento de forma independiente. Gracias a este modo de operación se puede ahorrar tiempo al dividir el paquete en fragmentos más pequeños que necesitan menos tiempo de procesamiento. Al igual que en el modo de operación *Broadcast,* la UD debe esperar a recibir la respuesta de todos los bloques aceleradores IP implicados antes de decidir qué debe hacer con el mensaje recibido. Por ello, este modo de operación necesita una señal con las direcciones de los bloques aceleradores IP a las que se desean enviar los mensajes, así como una señal individual para cada bloque con el número de segmentos que se desean enviar. Para la implementación de este modo de operación, existe una restricción en nuestro diseño, pues es necesario conocer con anterioridad el tamaño del paquete que se recibe a través del bloque MAC Ethernet emisor. Este paquete debe ser fijo en todo momento para que la distribución asignada a las señales que determinan el número de segmentos para cada bloque IP pueda ser correcta. La suma de estas señales debe ser igual a la totalidad del paquete.

Es necesario incluir valores que determinen el modo de operación deseado, así como las direcciones de los bloques aceleradores IP a los que se desea enviar la información. Al igual, es necesario incluir el número de segmentos que se desea enviar a cada bloque IP en caso de decir que el modo de operación es *Segmentation*. Por ello existirán tres señales que determinen el modo de operación utilizado. Estas señales son:

- *operation_mode*: determina el modo de operación, "0" para *Unicast*, "1" para *Broadcast* y "2" para *Segmentation*.

- *addr*: determina los bloques IP aceleradores que se encuentran activos y desactivos.

- *segments*: existirá una señal individual para cada bloque acelerador IP y determinará el número de segmentos que se deben enviar a cada bloque IP en caso de estar activo el modo de operación *Segmentation*. La suma de todas las señales *segment*s deberá ser igual al total del paquete Ethernet de entrada.

4.3. Comunicaciones del bloque

AMBA AXI es una especificación de protocolo de buses en chip generada por ARM. AXI4 se centra en proporcionar productividad, flexibilidad y disponibilidad [30]. Gracias a ser un estándar industrial es uno de los protocolos más utilizados para la interconexión de bloques aceleradores IP, siendo soportado por una gran cantidad de proveedores IP. Asimismo existe un gran conjunto de herramientas de diseño y verificación que dan soporte a este protocolo. En concreto Xilinx ha estandarizado AXI como el protocolo de referencia para sus IPs y está soportado en el entorno Vivado.

En este diseño, las interfaces de entrada y salida de la UD se realizan a través de protocolos de comunicaciones AXI específicos. Estos son AXI4-Stream para datos y AXI4-Lite para control. Esto se debe a que tanto la unidad de red, como los bloques procesadores solo aceptan este tipo de protocolos de entrada y salida. Por ello, se ha realizado un estudio de los diferentes modos de operación (lectura y escritura) para su correcta implementación.

En primer lugar, AXI4-Stream es necesario para realizar las comunicaciones con la unidad de red y los bloques procesadores. AXI4-Stream se utiliza para efectuar el *streaming* de datos. Para cumplir con el protocolo de comunicaciones AXI4-Stream necesitamos de un total de ocho entradas y salidas para cada dispositivo utilizado: cuatro para realizar la recepción de datos y otras cuatro para realizar el envío. Estas cuatro señales son iguales para la entrada y para la salida, siendo las siguientes: TDATA, TLAST, TREADY y TVALID. Las dos primeras señales son necesarias para el envío de datos (TDATA) y para determinar cuándo termina de enviarse cada paquete (TLAST). Sin embargo, TREADY y VALID son necesarias para poder iniciar la comunicación. TREADY es activada por el dispositivo esclavo para avisar de que se encuentra disponible para la recepción de datos, mientras que TVALID será activada por el dispositivo maestro para confirmar que el dato que se encuentra en TDATA es válido.

En segundo lugar, se encuentra el protocolo de comunicaciones AXI4-Lite, que es necesario para la configuración de la UD a través del puerto de configuración. Este protocolo de comunicación, a diferencia del anterior, utiliza un esquema de direcciones y datos (*memory-mapped*). Requiere cinco canales de comunicación, que son:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

Los datos pueden moverse en ambas direcciones entre el maestro y el esclavo de forma simultánea, y los tamaños de transferencia de datos pueden variar. Sin embargo AXI4-Lite permite una única transferencia de datos por transacción. La Figura 17 muestra la arquitectura del canal para las lecturas y la Figura 18 la arquitectura del canal para las escrituras [30].

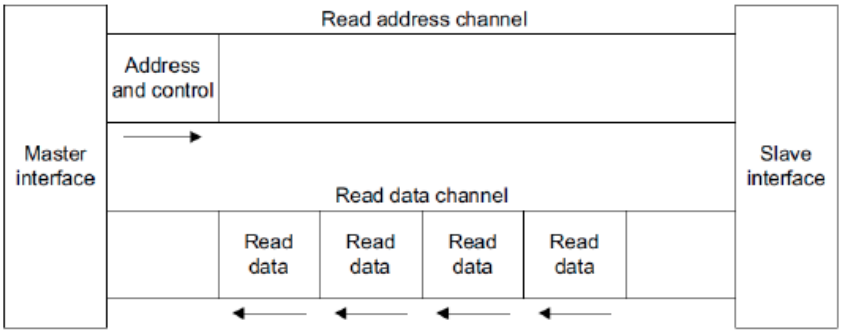

Figura 17: Arquitectura del canal para la lectura AXI4-Lite

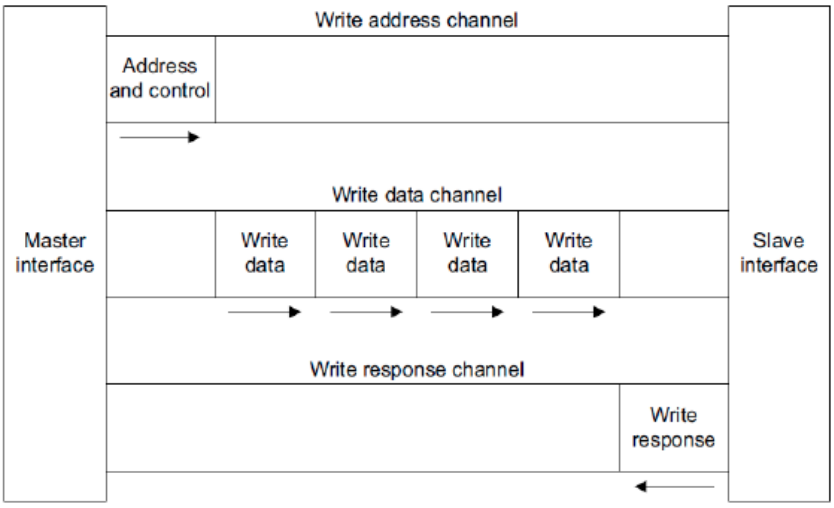

Figura 18: Arquitectura del canal para la escritura AXI4-Lite

A continuación, se observa en la Tabla 2 las diferentes señales que componen el protocolo AXI4-Lite para realizar una lectura y cuál es su cometido. A su vez, en la Tabla 3 se muestran las señales correspondientes a la escritura del AXI4-Lite y su finalidad [30].

Tabla 2: Señales correspondientes a la lectura AXI4-Lite

Canal	Señal	Tamaño	Maestro(M) / Esclavo(S)	Funcionalidad
Read Address Channel	S_AXI_ARADDR	32 bit	M	Dirección donde se desea realizar la lectura.
	S_AXI_ARVALID	1 bit	M	Asegura que la dirección es correcta.
	S_AXI_ARREADY	1 bit	S	Indica que el esclavo está listo para recibir la dirección.
Read Data Channel	S_AXI_RDATA	32 bit	S	Dato pedido por el maestro.
	S_AXI_RVALID	1 bit	S	Asegura que el dato es correcto.
	S_AXI_RREADY	1 bit	M	Indica que el maestro está listo para recibir el dato.
	S_AXI_RRESP	2 bit	S	Determina si la transacción ha sido correcta o hay un error.

Tabla 3: Señales correspondientes a la escritura AXI4-Lite

Canal	Señal	Tamaño	Maestro(M) / Esclavo(S)	Funcionalidad
Write Address Channel	S_AXI_AWADDR	32 bit	M	Dirección donde se desea realizar la escritura.
	S_AXI_AWVALID	1 bit	M	Asegura que la dirección es correcta.
	S_AXI_AWREADY	1 bit	S	Indica que el esclavo está listo para recibir la dirección.
Write Data Channel	S_AXI_WDATA	32 bit	M	Dato que se desea escribir en el esclavo.
	S_AXI_WVALID	1 bit	M	Asegura que el dato es correcto.

Canal	Señal	Tamaño	Maestro(M) / Esclavo(S)	Funcionalidad
	S_AXI_WREADY	1 bit	S	Indica que el esclavo está listo para recibir el dato.
	S_AXI_WSTRB	4 bit	M	Determina que bytes del dato son válidos.
Write Response Channel	S_AXI_BREADY	1 bit	M	Indica que el maestro está listo para recibir el dato.
	S_AXI_BRESP	2 bit	S	Determina si la transacción ha sido correcta o hay un error.
	S_AXI_BVALID	1 bit	S	Asegura que el BRESP es correcto.

Cabe destacar que para la escritura del AXI4-Lite, tanto la dirección como el dato deben estar disponibles en el mismo ciclo de reloj. En la Figura 19 se aprecia un ejemplo de escritura AXI4-Lite. Cuando las señales VALID y READY de la dirección y el dato están activas es posible realizar la escritura en dos ciclos. Sin embargo, para llevar a cabo la lectura es necesario primero enviar la dirección donde se quiere realizar la lectura para que, en el siguiente ciclo de reloj, la señal de datos S_AXI_RDATA disponga del dato que se desea leer. En la Figura 20 se observa un ejemplo de la lectura AXI4-Lite, que requiere de un mínimo de tres ciclos para obtener el dato deseado, siempre y cuando todas las señales se activen nada más sean requeridas [31].

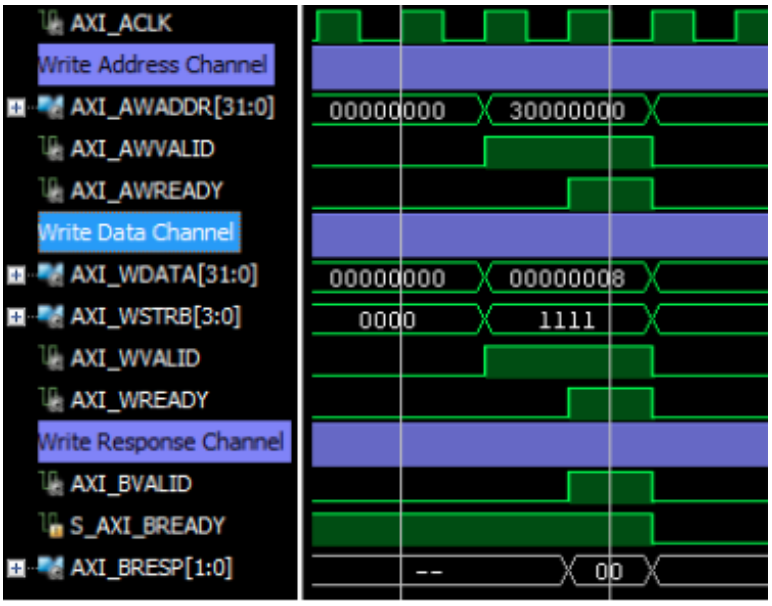

Figura 19: Ejemplo de escritura AXI4-Lite

Figura 20: Ejemplo de lectura AXI4-Lite

4.4. Funcionamiento de la Unidad de Despacho

La UD ha sido modelada en SystemC, siguiendo la arquitectura mostrada en la Figura 16. Para el desarrollo del sistema final se ha decidido incluir ocho bloques IP aceleradores, aunque esto puede variar en función de los requisitos del sistema final en el que se desee integrar la UD, del usuario y de la plataforma utilizada.

4.4.1. Puertos de entrada/salida

Los puertos de entrada y salida para la generación del módulo son numerosos, pero la utilización de un nivel de modelado más abstracto simplifica el problema. Sin embargo, si se muestran los puertos de entrada y salida en función de buses que contienen los protocolos de comunicación, los puertos utilizados son los siguientes:

- data_in_ethernet: realiza la conexión con el bloque MAC Ethernet emisor para realizar la lectura de los datos.

- data_out_ethernet: es necesario para realizar la escritura de los datos en el bloque MAC Ethernet receptor.

- data_in_accelX: existe un bus por cada bloque IP acelerador, por lo que habrá un total de ocho. Realiza la lectura del bloque IP acelerador seleccionado.

- data_out_accelX: al igual que con la señal anterior existirán ocho señales en el diseño, una por cada bloque IP acelerador implementado. Es el encargado de la operación de escritura en el bloque seleccionado.

- read_fifoX: para cada una de las dos FIFOs necesarias existirá un bus que incluye las señales de lectura de la memoria: *re* habilita la lectura, *dout* transmite el dato leído y *empty* avisa de que la memoria se encuentra vacía.

- write_fifoX: al igual que con la lectura de la FIFO, existirá un bus para cada una de las dos FIFOs implementadas. Incluye las señales de escritura en la memoria: *we* habilita la escritura, *din* contiene el dato a almacenar y *almostfull* indica que la memoria está casi llena.

- S_AXI: incluye el protocolo AXI4-Lite necesario para la implementación del puerto de configuración. Permite la lectura y escritura de datos relacionados con el modo de operación a petición del usuario.

En la Figura 21 se puede observar el diseño del bloque de la UD con las interfaces de entrada y salida descritas.

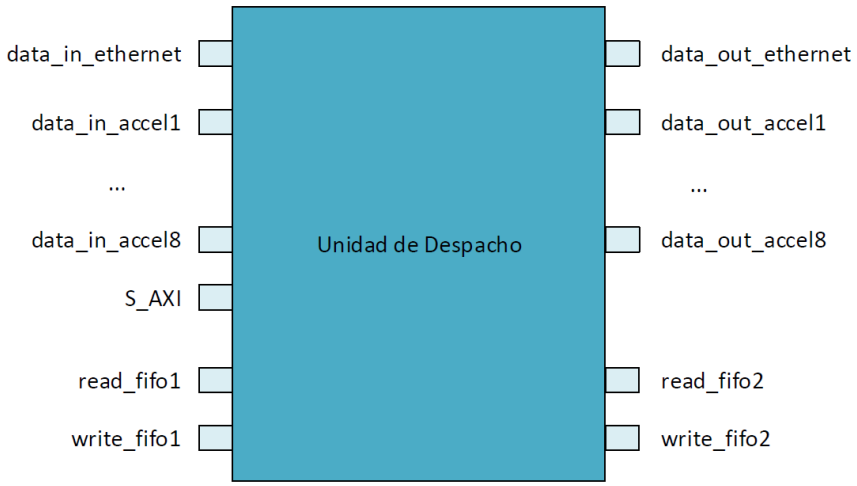

Figura 21: Diseño del bloque UD

4.4.2. Señales

Una vez determinadas las entradas y salidas del bloque, se deben introducir las señales internas utilizadas para la interconexión de las funciones que componen el diseño de la UD. La funcionalidad de estas señales es comunicar las funciones, sincronizarlas y producir el intercambio de información entre ellas. Las señales implementadas en el diseño son las siguientes:

- *waiting_IP*: una señal *unsigned int* de 3 bits que indica que el módulo está esperando una respuesta de un bloque IP acelerador, pues se ha iniciado el envío de datos a uno o varios de los bloques IP aceleradores. La selección de 3 bits se debe a que se utilizan ocho bloques IP aceleradores (N = 8) y al aplicar "$\log_2(N)$" obtenemos el ancho en bits del sistema.

- *pending*: una señal *unsigned int* de 3 bits (ancho de bits del sistema) que indica que se ha recibido una respuesta completa de los bloques IP aceleradores y se debe analizar su valor.

- *start_demux*: una señal de tipo *boolean* que indica cuándo la función demux() puede empezar a realizar el envío de datos a los bloques IP aceleradores.

- *operation_mode*: una señal *unsigned int* de 2 bits, que indica el modo de operación escogido por el usuario, siendo "0" *Unicast*, "1" *Broadcast* y "2" *Segmentation*.

- *addr*: una señal *unsigned int* de 8 bits, pues son 8 los bloques IP aceleradores implementados en el diseño. Indica cuáles están habilitados poniendo un "1" en la posición correspondiente, siguiendo un orden progresivo, siendo el bit menos significativo el primer bloque IP acelerador y el bit más significativo el octavo bloque IP acelerador.

- *segments1…x*: por cada bloque IP acelerador existe una señal *unsigned int* de 8 bits que indica el número de segmentos que serán enviados a ese bloque. Para el diseño realizado se han generado ocho señales y únicamente se aplicarán cuando el modo de operación sea *Segmentation*.

- *change_om*: una señal de tipo *boolean* que avisa a la UD de que el usuario desea realizar un cambio de modo de operación.

- *ready_change_om*: una señal de tipo *boolean* que indica a la función state_machine() que el sistema está listo para realizar el cambio de modo de operación.

- *transaction_om*: una señal *unsigned int* de 2 bits que indica al usuario si el cambio de operación realizado ha sido correcto o no. Tiene tres posibles valores: "0" OK, "1" ERROR y "2" en proceso.

A continuación, en la Figura 22 se mostrará el código del archivo de cabecera con la función "top" realizado para el desarrollo de la UD. Aparecen todas las interfaces de entrada y salida y las señales descritas. También se incluyen las funciones que componen la UD que se verán con detenimiento en el siguiente apartado.

```
#ifndef _UNIDADDESPACHO_H_
#define _UNIDADDESPACHO_H_

// Include files
…

SC_MODULE (UnidadDespacho) {
   sc_in<bool> clk;
   sc_in<bool> rst_n;
   //---------------------------------------------------------------------------
   // Interfaces

   // DATA IN Signals needed to perform a AXI4_Stream transmission.
   sc_in<sc_uint<32> >          data_in_ethernet_data;
   sc_in<bool>                  data_in_ethernet_valid;
   sc_in<bool>                  data_in_ethernet_last;
   sc_out<bool>                 data_in_ethernet_ready;

   sc_in<sc_uint<64> >          data_in_accel1_data;
   …

  // DATA OUT Signals needed to perform a AXI4_Stream transmission.
   sc_out<sc_uint<32> >         data_out_ethernet_data;
   …
   sc_out<sc_uint<64> >         data_out_accel1_data;
   …

   // Interface with external FIFO1
   sc_out<DATA>                 din1;
   sc_out<bool>                 we1;
   sc_in<bool>                  almostfull1;
   sc_in<DATA>                  dout1;
   sc_out<bool>                 re1;
   sc_in<bool>                  empty1;

   // Interface with external FIFO2
   …

   // DATA Signals needed to perform a AXI4_Lite transmission.
   // Write Address Channel
   sc_in<sc_uint<WORD_SIZE> >   S_AXI_AWADDR;
   sc_in<bool>                  S_AXI_AWVALID;
   sc_out<bool>                 S_AXI_AWREADY;

   // Write Data Channel
   …
   // Read Address Channel
   …
   // Read Data Channel
   …
   // Write Response Channel
   …

//---------------------------------------------------------------------------
//Variables
   sc_uint<2>     accel_finish[8];
```

```
// Functions
    void ethernet_in_copy();
    void ethernet_out_send();
    void accel_read_answer();
    void mux_64();
    void demux();
    void state_machine();
    void read_om();

//--------------------------------------------------------------------------------
// Internal Signals
    sc_signal<sc_uint<3> >          waiting_IP;
    sc_signal<sc_uint<3> >          pending;
    sc_signal<bool>                 start_demux;
    sc_signal<sc_uint<2> >          operation_mode;
    sc_signal<sc_uint<8> >          addr;
    sc_signal<sc_uint<8> >          segments1;
    …
    sc_signal<sc_uint<8> >          segments8;
    sc_signal<bool>                 change_om;
    sc_signal<bool>                 ready_change_om;
    sc_signal<sc_uint<2> >          transaction_om;

//--------------------------------------------------------------------------------
//Constructor
  SC_CTOR (UnidadDespacho) :
            clk("clk"),
            rst_n("rst_n"),

            // DATA IN Signals needed to perform a AXI4_Stream transmission.
            data_in_ethernet_data("data_in_ethernet_data"),
            …
            // DATA OUT Signals needed to perform a AXI4_Stream transmission.
            data_out_ethernet_data("data_out_ethernet_data"),
            …
            //FIFOs
            din1("din1"),
            …
            // DATA Signals needed to perform a AXI4_Lite transmission.
            S_AXI_AWADDR("S_AXI_AWADDR"),
            …
            // Internal Signals
            waiting_IP("waiting_IP"),
            …
    {

//----------------- Modules' Interconections---------------------------------
            SC_CTHREAD(ethernet_in_copy, clk.pos());
            reset_signal_is(rst_n, false);
            …
    }
};


#endif /* _UNIDADDESPACHO_H_ */
```

Figura 22: Archivo de cabecera "UnidadDespacho.h"

4.4.3. Funciones

Se han generado un total de seis funciones que implementan la funcionalidad de la UD diseñada.

void ethernet_in_copy().

Es la función principal del módulo UD, pues es la que permite que el resto de funciones puedan empezar a ejecutarse. Su objetivo es copiar el paquete recibido del bloque MAC Ethernet emisor en la memoria FIFO1 y unificar dos mensajes de 32 bits en una única palabra de 64 bits para almacenarla en la memoria FIFO2. Como la UD es un bloque encargado de realizar las comunicaciones se encontrará siempre que sea posible activo para la recepción de paquetes, a excepción de que las memorias FIFOs se encuentren completas o de que un cambio de modo de operación esté teniendo lugar.

Para el desarrollo de esta función se ha tenido en cuenta el protocolo de comunicación AXI4-Stream y el protocolo para realizar escrituras en memorias tipo FIFO. Tras recibir dos palabras consecutivas se activa la señal *start_demux* que permite a la función "demux()" iniciar el envío de datos a los bloques IP aceleradores. Una consideración a tener en cuenta en este bloque es que, una vez se inicia la recepción de un paquete, no permite iniciar el cambio de modo de operación, sino que espera que se termine la recepción del paquete para deshabilitar la recepción de datos hasta que el nuevo modo de operación haya sido implementado.

void demux().

Esta función realiza la lectura de datos de la memoria FIFO2 y la escritura de dichos datos en los bloques IP aceleradores correspondientes. Por este motivo, debe tener en cuenta las señales que determinan el modo de operación del sistema (*operation_mode*, *addr* y *segments1…8*).

Antes de poder realizar sus objetivos principales, debe comprobar que la memoria a la que quiere acceder para la lectura no se encuentre vacía y que la señal *start_demux* esté activa. En caso afirmativo aumenta el valor de la señal *waiting_IP* para indicar que se ha iniciado una transmisión de paquetes a uno o varios bloques IP aceleradores.

Posteriormente, en función del modo de operación y de las direcciones activas, realiza la lectura de la FIFO2 y la escritura de los datos obtenidos en los correspondientes bloques IP aceleradores, comprobando anteriormente que se encuentren activos para la recepción de datos y gestionando el número de mensajes que debe recibir cada bloque en función del modo de operación.

Para la realización de esta función no solo ha sido necesario implementar el protocolo de comunicaciones AXI4-Stream, sino también incluir el protocolo de lectura de la FIFO. La lectura de la FIFO tiene una peculiaridad, pues para acceder a un dato por primera vez son necesarios dos ciclos para su obtención, pero para las lecturas posteriores solo será necesario un ciclo. Sin embargo, si la memoria se encuentra vacía durante varios ciclos consecutivos será necesario volver a repetir este proceso.

En la Figura 23 se muestra un extracto del código utilizado para esta función, pues dentro de las funciones desarrolladas es la de mayor complejidad.

```
void UnidadDespacho::demux() {
  ... // Reset Phase: Signal values
  wait();

  while (true) {

    tstart_demux = start_demux.read();
    taddr = addr.read();
    tempty2 = empty2.read();
    tsegments1 = segments1.read();
    ...
    tsegments8 = segments8.read();

    if ((tstart_demux == true) || (tempty2 == false)) {

      u++;
      waiting_IP.write(u);

      do {
        if ((empty2.read() == true) && (first_time2 == false)) {
          if (next == true) {
              first_time = true;
              next = false;
          }
          wait();
          while (empty2.read() == true) {
              first_time = true;
              wait();
          }
        } else {
          first_time2 = true;
          next = false;
        }

        if (first_time == true) {
          re2.write(true);
          first_time = false;
          first_time2 = false;
          next = false;
          wait();
        }

        re2.write(true);
```

```
if ((empty2.read() == true) && (first_time2 == true)) {
  re2.write(false);
  first_time2 = false;
}
ip1 = (cnt1 < tsegments1);
...
ip7 = (cnt7 < tsegments7);
wait();

if (empty2.read() == true) {
  next = true;
}
ip8 = (cnt8 < tsegments8);
temp = dout2.read();
tlast = temp.tlast;
re2.write(false);

if (operation_mode.read() != 2) {          // Unicast or Broadcast

  while (((data_out_accel1_ready.read() != true) & taddr.range(0,0)) &&
      ...
      ((data_out_accel8_ready.read() != true) & taddr.range(7,7))) {
    re2.write(false);
    wait();
  }

  if (taddr.range(0, 0) == 1) {
    data_out_accel1_valid.write(true);
    data_out_accel1_data.write(temp.data);
    data_out_accel1_last.write(temp.tlast);
  }
  …
  if (taddr.range(7, 7) == 1) {
    data_out_accel8_valid.write(true);
    data_out_accel8_data.write(temp.data);
    data_out_accel8_last.write(temp.tlast);
  }
} else {                  // Segmentation

  while (((data_out_accel1_ready.read() != true) & taddr.range(0,0)) ||
      ...
      ((data_out_accel8_ready.read() != true) & taddr.range(7,7))) {
    re2.write(false);
    wait();
  }

  if (ip1 == true) {
    data_out_accel1_valid.write(true);
    data_out_accel1_data.write(temp.data);
    data_out_accel1_last.write(temp.tlast);
    cnt1++;
    if (cnt1 == tsegments1) {
      data_out_accel1_last.write(true);
    }
  }
  ...
  } else if (ip8 == true) {
    data_out_accel8_valid.write(true);
    data_out_accel8_data.write(temp.data);
```

```
            data_out_accel8_last.write(temp.tlast);
            cnt8++;
            if (cnt8 == tsegments8) {
              data_out_accel8_last.write(true);
            }
          }
        }
        wait();
        data_out_accel1_valid.write(false);
        ...
        data_out_accel8_valid.write(false);

      } while ((tlast == false));

      first_time2 = true;
      cnt1 = 0;
      ...
      cnt8 = 0;
      re2.write(false);
      wait();

    } else {
      if (empty2.read() == true) {
        wait();
        if (empty2.read() == true) {
          first_time = true;
        }
      }
      re2.write(false);
      wait();
    }
  }
}
```

Figura 23: Función demux() de la Unidad de Despacho

Para realizar la escritura de los datos en los bloques IP aceleradores son necesarios dos ciclos. Se ha diseñado el bloque de esta manera, para poder realizar la escritura de datos de forma simultánea a la recepción de datos de la unidad de red, pues son necesarios dos ciclos para obtener un paquete de 64 bits, tal y como se explicó en la función "ethernet_in_copy()".

No obstante, al disponer de dos ciclos para la escritura de datos se puede dividir la lógica introducida en esta función y así no alargar la duración de los periodos de reloj, manteniendo una frecuencia de funcionamiento superior a 200 MHz. Se puede observar en la Figura 23 que en la función se ha introducido una variable de tipo *boolean* "first_time2" que es necesaria para subsanar la pérdida de un dato de la memoria FIFO2.

Esto se debe a que al realizar las operaciones de lectura de la memoria FIFO2 en dos ciclos, cuando se lee el último dato antes de vaciarse por completo se activa a nivel alto la señal *empty2* de la memoria FIFO2, pero el dato está listo para ser leído. Por ello si la señal "first_time2" se encuentra

activa y la memoria está vacía, permite realizar la escritura del dato leído en los bloques IP aceleradores.

Sin embargo, solo será válido una vez, pues se desactiva la señal "first_time2" al solo ser necesaria en este caso particular. Como consecuencia de introducir esta señal se debe introducir otra llamada "next" de tipo *boolean*, que se activa al encontrarse la memoria FIFO2 vacía. Únicamente, tendrá validez esta señal cuando la memoria FIFO2 se encuentre vacía y la señal "first_time2" se encuentra desactiva. La funcionalidad de "next" es asegurarse de que al comenzar la lectura de la FIFO2, tras esta encontrarse vacía, no se obtengan datos que no existen.

void accel_read_answer().

Esta función siempre se encontrará activa y a la espera de recibir respuestas de los bloques IP aceleradores. Sin embargo, cuando el paquete de datos ha sido enviado a más de un bloque debe esperar la respuesta de todos ellos. Por ello, tiene que acceder a la señal *addr* y comprobar que se han recibido todas las respuestas antes de poder enviar la respuesta final a la función "ethernet_out_send()". Realiza una suma de todas las señales obtenidas y guarda el resultado en una variable interna *result*. Si el resultado es "0" significa que los bloques IP aceleradores no han tenido éxito en su búsqueda, lo que representa que el paquete de datos es apto para continuar su transmisión a la unidad de red correspondiente.

Para ello, se ha creado una variable llamada *accel_finish[]* que es un *array* de 8 posiciones de 2 bits. Si el resultado obtenido es positivo escribe un "1" para indicar que el paquete debe ser descartado, y en caso contrario, si la respuesta obtenida es negativa escribe un "2" para indicar que el paquete debe ser transmitido. También incrementa el valor de la señal *pending* una vez guardado el resultado obtenido en el *array*, para avisar a la función "ethernet_out_send()" de que tiene una nueva respuesta que procesar.

Una característica particular de esta función es que tiene en cuenta si aún no se han recibido las respuestas de los paquetes enviados a los bloques IP aceleradores, gracias a la señal *waiting_IP*. Esto se debe a que una vez no haya más respuestas pendientes y un usuario haya solicitado un cambio de modo de operación, se activa la señal *ready_change_om* para permitir realizar el cambio de operación. En este punto, todas las funciones que necesitan conocer el modo de operación de la UD para su correcto funcionamiento se encuentran detenidas y es posible realizar un cambio de modo de operación. Hasta que el nuevo modo de operación no se haga efectivo la función se queda esperando y posteriormente retoma su normal funcionamiento.

void ethernet_out_send().

La función se encuentra continuamente a la espera de una respuesta de los bloques IP aceleradores para poder analizarla. La señal *pending* determina si existe alguna. Antes de poder realizar la lectura de datos en la memoria FIFO1 se debe comprobar que la señal *empty1* no se encuentre activa, para poder acceder a los datos recibidos de la unidad de red. A continuación, se lee en el *array accel_finish[]* el resultado obtenido. Si el resultado es igual a "2" se procede a la lectura de la FIFO1 y su escritura en el bloque MAC Ethernet receptor.

Sin embargo, si la respuesta es diferente a "2" simplemente se realiza la lectura de la memoria pero no se lleva a cabo la escritura, descartando de esta manera el paquete y liberando espacio de memoria para permitir el almacenamiento de nuevos paquetes. Cuando se procede a la lectura de la memoria FIFO1 en esta función también es necesario tener en cuenta los dos ciclos de lectura iniciales explicados en la función "demux()"

void state_machine().

Cumpliendo con el protocolo de comunicaciones AXI4-Lite esta función permite determinar el modo de operación, las direcciones de los bloques IP aceleradores activos y el número de segmentos. El bloque se encuentra continuamente a la espera de que se realice una escritura en los puertos AXI4-Lite para realizar un cambio de modo de operación. Una vez detecta una escritura activa la señal *change_om* para avisar al resto de funciones de la intención de cambiar el modo de operación del sistema. Mientras las funciones activas terminan de realizar los procesos activos se debe esperar a recibir las tres escrituras necesarias para la obtención de todas las señales que determinan el modo de operación de la UD.

Tras obtener tanto el modo de operación, como las direcciones de los bloques IP aceleradores activos y el número de segmentos de cada bloque IP se procede a comprobar que el nuevo modo de operación sea válido para su implementación en la plataforma. En caso de que sea correcto, solamente se espera a que la señal *ready_change_om* se active para hacer efectivo dicho cambio. En caso contrario, se fuerza el modo de operación por defecto del sistema tras un *reset* y se espera a la señal *ready_change_om* para implementarlo. Por último, desactiva la señal *change_om* para permitir que la UD siga funcionando de forma normal con el nuevo modo de operación. También, cabe destacar el uso de la señal *transaction_om* que se actualizará a lo largo de esta función, para determinar si se está realizando un cambio de modo de operación o si el nuevo modo de operación ha sido implementado de forma satisfactoria. En la Figura 24 se muestra el código utilizado para desarrollar esta función.

```
void UnidadDespacho::state_machine() {
… // Reset Phase: Signal values
wait();

while (true) {
    if ((S_AXI_AWVALID.read() == true) && (S_AXI_WVALID.read() == true)) {
        change_om.write(true);
        transaction_om.write(2);   // In process

        S_AXI_AWREADY.write(true);
        S_AXI_WREADY.write(true);
        S_AXI_BVALID.write(true);
        S_AXI_BRESP.write(0);
        twrite_addr = S_AXI_AWADDR.read().range(15,0);
        twrite_data = S_AXI_WDATA.read();
        wait();

        switch (twrite_addr) {

        case DATA_0:
            toperation_mode = twrite_data.range(1,0);
            taddr = twrite_data.range(9,2);
            i++;
            break;
        case DATA_1:
            tsegments1 = twrite_data.range(7,0);
            tsegments2 = twrite_data.range(15,8);
            tsegments3 = twrite_data.range(23,16);
            tsegments4 = twrite_data.range(31,24);
            tsegments_a = tsegments1 + tsegments2 + tsegments3 + tsegments4;
            i++;
            break;
        case DATA_2:
            tsegments5 = twrite_data.range(7,0);
            tsegments6 = twrite_data.range(15,8);
            tsegments7 = twrite_data.range(23,16);
            tsegments8 = twrite_data.range(31,24);
            tsegments_b = tsegments5 + tsegments6 + tsegments7 + tsegments8;
            i++;
            break;
        default:
            break;
        }

        S_AXI_BVALID.write(false);
        S_AXI_AWREADY.write(false);
        S_AXI_WREADY.write(false);
        if (i == 3) {
            end_transmission = true;
            i = 0;
        } else {
            end_transmission = false;
        }
        wait();
    }
    // Checks if the new Operation Mode is correct or not
    if (end_transmission == true) {
        if (toperation_mode == 2) {
            tsegments = tsegments_a + tsegments_b;
```

```
                    if (tsegments == 0xBC) {
                        OK = true;
                        wait();
                        if (((taddr.range(0,0) == 1) == (tsegments1 != 0))
                            && ((taddr.range(1,1) == 1) == (tsegments2 != 0))
                            …
                            && ((taddr.range(7,7) == 1) == (tsegments8 != 0))) {
                                OK = true;
                        } else {
                            OK = false;
                        }
                    } else {
                        OK = false;
                    }
                } else {
                    for (int u = 0; u < 8; u++) {
                        if (taddr.range(u,u) == 1) {
                            a++;
                        }
                        wait();
                    }
                    if (a != 0) {
                        OK = true;
                    } else {
                        OK = false;
                    }
                    a = 0;
                }
                wait();
                if (OK == false) {                  // If the new Operation Mode is not correct,
                    toperation_mode = 0x00;  // the Unicast Mode is used
                        taddr = 0x00000001;
                    tsegments1 = 0x00000000;
                    …
                    tsegments8 = 0x00000000;
                }
                wait();
                while (ready_change_om.read() == false) {
                    wait();
                }

                operation_mode.write(toperation_mode);
                addr.write(taddr);
                segments1.write(tsegments1);
                …
                segments8.write(tsegments8);
                change_om.write(false);
                end_transmission = false;
                if (OK == false) {
                    transaction_om.write(1);   // ERROR
                } else {
                    transaction_om.write(0);   // OK
                }
            }
            wait();
        }
}
```

Figura 24: Función "state_machine()" de la UD

void read_om().

Por último, esta función es la encargada de gestionar las lecturas exigidas por el usuario, cumpliendo con el protocolo de comunicaciones AXI4-Lite. En función de la dirección de lectura irá enviando al usuario los datos del modo de operación efectivo en la UD, siendo las mismas direcciones utilizadas para la lectura y para la escritura de estos datos. Sin embargo, también añade una dirección adicional para poder leer la señal *transaction_om* y verificar el estado del cambio de modo de operación.

Finalmente, se puede observar en la Figura 25 un esquemático que incluye todas las funciones utilizadas y la interconexión de ellas mediante las señales internas.

Se ha añadido un fichero de cabecera con el nombre "registers_param.h" que será el encargado de almacenar las direcciones de escritura y lectura del protocolo de comunicaciones AXI4-Lite. Para este trabajo, solo han sido necesarias la creación de cuatro direcciones para incluir el modo de operación, las direcciones utilizadas, los segmentos asignados a cada bloque IP y el estado del cambio de modo de operación. En la primera dirección, a la que se ha llamado "DATA_0" se incluirán las señales *operation_mode* y *addr*, introduciéndolas de bit menos significativo a bit más significativo. De igual manera, en "DATA_1" se almacenan los valores de *segments1* hasta *segments4*; almacenando los valores de *segmentes5* a *segments8* en "DATA_2". Por último, "DATA_3" únicamente incluirá el valor de *transaction_om*. La Tabla 4 muestra las diferentes direcciones utilizadas para el protocolo AXI4.Lite, teniendo en cuenta que no existen espacios entre las señales incluidas.

Tabla 4: Direcciones del AXI4-Lite para la UD

Dirección	Valor de dirección	Contenido			
		MSBs		LSBs	
DATA_0	0x00000000	-	-	addr	operation_mode
DATA_1	0x00000020	segments4	segments3	segments2	segments1
DATA_2	0x00000040	segments8	segments7	segments6	segments5
DATA_3	0x00000060	-	-	-	transaction_om

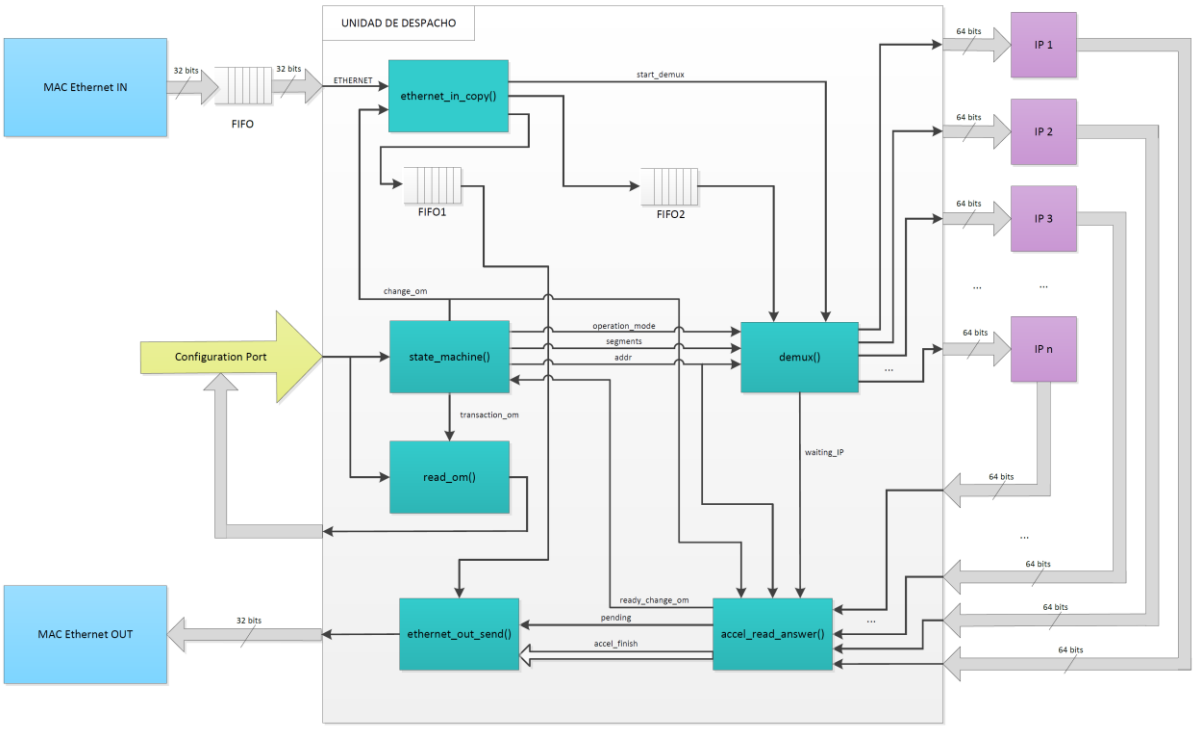

Figura 25: Esquemático de las funciones de la UD

4.5. Análisis de la memoria y la frecuencia de funcionamiento

En este apartado se realizará un análisis de la memoria y la frecuencia de funcionamiento requeridas para el correcto funcionamiento de la Unidad de Despacho. Para ello es necesario tener en cuenta ciertos factores iniciales. Antes que nada, se debe considerar el funcionamiento normal de los bloques MAC Ethernet y la velocidad con que los datos son enviados. Para ello, se supone que el bloque MAC Ethernet trabaja a una frecuencia constante de 200 MHz, lo que implica un régimen binario de 6,4 Gbps. El bloque MAC Ethernet envía paquetes de tamaño de 1500 bytes de forma constante. Estos paquetes se descomponen en 375 palabras de 32 bits que se envían desde el bloque MAC Ethernet a la UD. Por tanto, cada 5 ns se recibe una palabra de 32 bits.

Antes de la UD existirá una cola FIFO externa encargada de ser la intermediaria entre el bloque MAC Ethernet emisor y la propia UD. De esta manera, no es necesario que el bloque MAC Ethernet y la UD tengan la misma frecuencia de funcionamiento, pudiendo trabajar a dos frecuencias diferentes.

Para poder realizar la recepción de datos en la UD es necesaria la implementación de varias memorias FIFOs dentro de la UD para ir almacenando los paquetes antes de que sean leídos. Se ha decidido utilizar dos memorias tipo FIFO en la UD, una que almacene los datos de 32 bits que llegan desde el bloque MAC Ethernet emisor, y otra que almacene datos de 64 bits, al incluir dos palabras

de 32 bits en una única de 64 bits. Estas memorias tipo FIFO serán bloques independientes que se implementarán como parte del código. Gracias a ello, la portabilidad del código se realizará con mayor facilidad y permitirá la reusabilidad del código en otros dispositivos. Como ya se ha comentado en apartados anteriores, las memorias tipo FIFO necesitan únicamente un ciclo de escritura. Sin embargo, necesitan dos ciclos de lectura la primera vez que desea realizar una lectura, y después únicamente necesitarán un ciclo. A excepción de que la memoria se vuelva a encontrar vacía, por lo que necesitará otra vez de los dos ciclos de lectura iniciales.

La memoria FIFO de 32 bits es necesaria para posteriormente, tras haber procesado el mensaje, ser capaz de enviarlo al bloque MAC Ethernet receptor en caso de que así se requiera. Su tamaño será de 2048 palabras de 32 bits, incluyendo para cada palabra un *boolean* que determinará si es la última palabra del paquete. De esta forma, se podrán almacenar más de 5 paquetes de 1500 bytes. La memoria FIFO de 64 bits es necesaria para poder almacenar los paquetes si los bloques IP aceleradores no se encuentran disponibles para recibirlos. También incluirá un *boolean* por cada palabra de 64 bits, y tendrá un total de 1024 palabras, así que será capaz de almacenar la misma cantidad de paquetes que la otra memoria FIFO.

A su vez, hay que tener en cuenta que los bloques IP aceleradores, con los que se desea trabajar, funcionan a una velocidad de 1 Gbps, por lo que su latencia será de 5 µs. También disponen de dos colas tipo FIFO; una de entrada y otra de salida. Debido a esto no es necesario implementar colas FIFO a la salida de los bloques, ya que a nivel *hardware* su implementación no es rentable y ya disponen de colas FIFO de salida propias.

Por otra parte, se debe tener en cuenta la cantidad de ciclos que tardará la placa de desarrollo ZedBoard en procesar los datos entre las entradas y salidas de la UD desarrollada. Será necesario un ciclo de escritura para guardar la palabra de 32 bits entrante, así como otro ciclo adicional, para poder obtener la palabra de 64 bits y guardarla en su correspondiente memoria. Igualmente, se necesitan dos ciclos para realizar la lectura de la memoria FIFO de 64 bits, y su escritura en los bloques IP aceleradores correspondientes. Suponemos una latencia de 5 µs para dichos bloques tras recibir la totalidad del paquete. Tras dicha latencia, se decide si se envía o no el mensaje al bloque MAC Ethernet receptor. En caso de que el bloque IP determine que se puede proseguir con la comunicación se realiza el envío desde la memoria FIFO de 32 bits de la UD al bloque MAC Ethernet destinatario. Teniendo en cuenta que la UD trabajará a una frecuencia de 200 MHz y que son necesarios 375 ciclos para poder transmitir el mensaje, se tardará 1,88 µs en realizar el envío del mensaje completo. Son necesarios 375 ciclos para realizar la lectura de la memoria FIFO1, salvo en el caso de que la memoria esté vacía que precisa un ciclo adicional.

Este sería el mejor caso, siempre y cuando no fuera necesario esperar a que los bloques IP aceleradores o el bloque MAC Ethernet receptor estén disponibles. Por lo que al menos serán necesarios 8,78 µs desde que se empiece a recibir el paquete desde el bloque MAC Ethernet emisor hasta que el paquete completo haya sido enviado al bloque MAC Ethernet receptor. En la Figura 26 se pueden ver los pasos explicados anteriormente gracias a un cronograma.

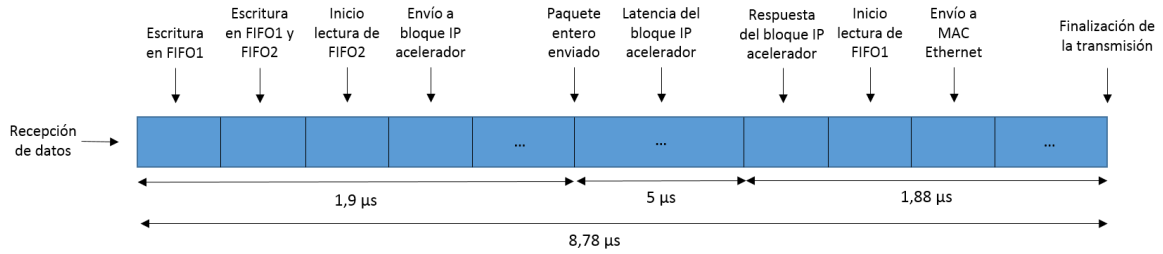

Figura 26: Cronograma de transmisión de datos en UD

4.6. Simulación en SystemC

Para la simulación del módulo desarrollado en SystemC es necesaria la elaboración de una serie de *testbench* que contengan todos los posibles casos. Para la comprobación de la UD ha sido necesaria la utilización de tres *testbench* que se implementan e interconectan con el bloque simulado de la UD gracias a un archivo "sc_main", al que se ha llamado "sc_main_UnidadDespacho". En este archivo se incluyen todos los puertos de entrada y salida necesarios para la implementación de la UD, para posteriormente poder efectuar las interconexiones pertinentes con los *testbench* realizados.

Inicialmente, se ha creado un *testbench* llamado "UnidadDespacho_test" que simula la actividad de la unidad de red. Este es el encargado de realizar el envío de datos a la UD y de comprobar que los datos devueltos por la UD coinciden con los esperados, teniendo en cuenta el protocolo de comunicaciones AXI4-Stream. Con la intención de obtener los paquetes a enviar a la UD se realiza una lectura de un archivo .txt que almacena un paquete de red. A su vez, se debe tener en consideración el estado de la UD y esperar a que este se encuentre activo antes de proceder al envío de paquetes de datos. Dentro de este *testbench* también se incluye el puerto de configuración del sistema, realizando escrituras en la UD para realizar cambios en el modo de operación, así como lecturas del estado del sistema. Para ello, ha sido necesario implementar el protocolo de comunicaciones AXI4-Lite maestro.

El siguiente *testbench* realizado ha sido el "IPblock_test" que simula el comportamiento de un bloque IP acelerador. Es necesario inicializar y conectar con la UD en el archivo "sc_main" tantos "IPblock_test" como bloques IP aceleradores hayan sido descritos en el diseño. Como se ha

comentado anteriormente, para la elaboración de este proyecto se ha decidido incluir ocho bloques aceleradores para comprobar el correcto funcionamiento de la UD. Ya que no se dispone del diseño del bloque IP acelerador, se ha decidido optar por realizar un *testbench* sencillo, que únicamente espere a la recepción del paquete para posteriormente devolver como resultado un valor predeterminado. Si el bloque devuelve un "0" como respuesta significa que el paquete puede ser enviado a la unidad de red. Sin embargo, si el resultado es "1" el paquete deberá ser descartado.

Finalmente, es necesario incluir un *testbench* que simule el funcionamiento de las memorias FIFOs implementadas en el diseño de la UD. El *testbench* "fifo" simula el funcionamiento de las lecturas y escrituras en la memoria FIFO y almacena los datos siempre que sea obligatorio. Para su inicialización, es necesario incluir en el "sc_main" las posiciones disponibles para cada memoria, así como el tipo de datos a almacenar y el número de bits utilizado para su asignación. En la Figura 27 se puede observar cómo se ha efectuado dicha inicialización para las memorias FIFO1 y FIFO2, incluyendo en primer lugar el número de bits, en segundo lugar el tipo de dato a guardar y en último lugar el tamaño de la memoria:

```
fifo<11, DATA, 2048> fifo1_tb("fifo1_tb");
fifo<10, DATA2, 1024> fifo2_tb("fifo2_tb");
```

Figura 27: Inicialización de las memorias FIFO en la simulación

Los tipos de estructura de datos DATA y DATA2 se incluyen en un fichero de cabecera llamado "data_struct.h". Ambos se encuentran conformados por un *unsigned int* que almacena los datos y un *boolean* que indica si el paquete ha finalizado o no. DATA dispone de un *unsigned int* de 32 bits, mientras que el de DATA2 es de 64 bits.

Tras saber todos los *testbench* necesarios, ya se pueden explicar las diferentes simulaciones llevadas a cabo y proceder al análisis de sus resultados. Para la correcta verificación del diseño, se ha decidido realizar un *testbench* que envíe un total de 16 paquetes, efectuando paradas en el envío de los mismos, e incluyendo cambios de modo de operación. Todo ello con el fin de verificar cada uno de ellos, incluyendo cambios de operación en mitad de una transmisión de datos. Por último, también se han implementado lecturas del modo de operación del sistema. Con el objetivo de facilitar la comprobación de los paquetes se ha decidido alternar el envío de paquetes predefinidos, uno de ellos con diferentes valores obtenidos del paquete de red y otro con un único valor: "0". Asimismo, se debe aclarar que todos los paquetes enviados son de 375 palabras de 32 bits, incluyendo en cada palabra un *boolean* que determina la finalización del paquete. De esta forma, se verifica que la UD

es capaz de recibir y enviar grandes paquetes, pues coincide con el tamaño de paquetes que la unidad de red distribuye.

De esta forma ya es posible introducir los resultados obtenidos en la simulación en SystemC. Se ha decidido realizar el re-envío de todos los paquetes de entrada, por lo que los bloques IP aceleradores siempre devuelven "0". La Figura 29 se obtiene al realizar la simulación en la herramienta SimVision, que permite observar las formas de onda de las señales en la simulación. Se pueden apreciar las señales internas del sistema, incluyendo las que caracterizan el modo de operación, y cómo varían las señales que determinan el modo de operación en función de las escrituras realizadas a través del puerto de configuración. Las señales *change_om* y *ready_change_om* señalan cuándo se inicia el cambio de operación y en qué momento se hace efectivo, respectivamente. Sin embargo, también se han generado una serie de impresiones por pantalla, para verificar el correcto funcionamiento de la UD. La Figura 28 muestra un ejemplo de una transacción utilizando el modo de operación *Broadcast*. Se realiza el envío de un paquete de datos a la UD y se muestra de qué manera es enviado a los bloques IP aceleradores y posteriormente enviado como respuesta a la unidad de red, comprobando que coincide con el paquete original.

```
Sending message to be forwarded.
…
Message from Ethernet = 4239004788
Last from Ethernet = 0
Message from Ethernet = 1642070023
Last from Ethernet = 0
…
Message to accel1 = 7052637050765972596
Last to accel = 0
…
Message to accel8 = 7052637050765972596
Last to accel = 0
…
Message from Ethernet = 93990065
Last from Ethernet = 1
…
Message to accel1 = 93990065
Last to accel = 1
…
Message to accel8 = 93990065
Last to accel = 1
Original message = 4239004788
Message from UD = 4239004788
…
Original message = 93990065
Message from UD = 93990065
OK
Test 1 Finished OK
```

Figura 28: Ejemplo de una Transacción en modo Broadcast

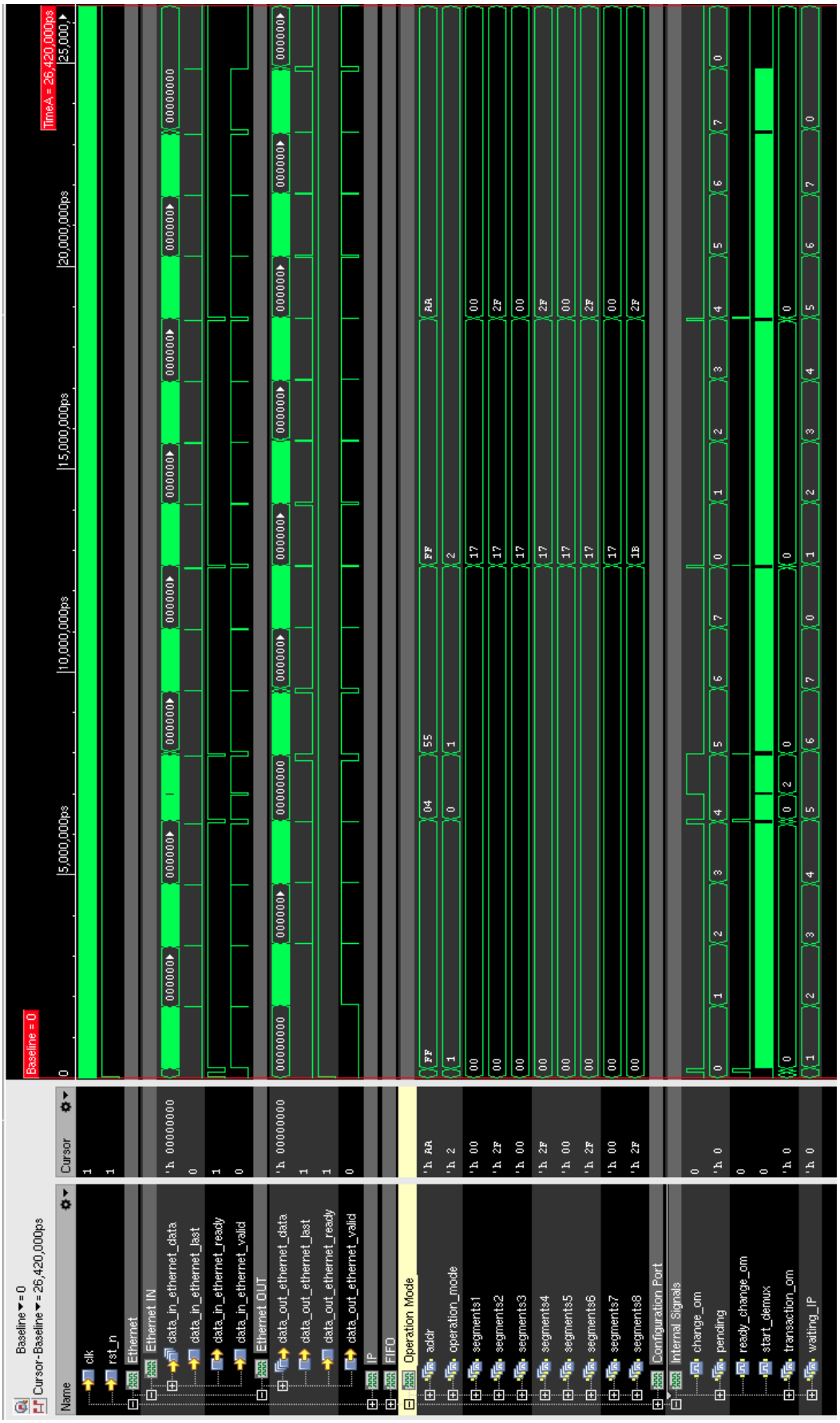

Figura 29: Simulación de SystemC en SimVision

En la Figura 30 se presenta un ejemplo de un cambio de modo de operación en la UD. Los canales de escritura de la UD se activan y almacenan los datos necesarios para el nuevo modo de operación. Para ello hacen falta tres escrituras que se realicen de forma concurrente. Se puede ver como la señal "data_in_ethernet_ready" se desactiva hasta que el nuevo modo de operación se hace efectivo en la UD. Más tarde se vuelve a activar, permitiendo la entrada de paquetes del bloque MAC Ethernet emisor.

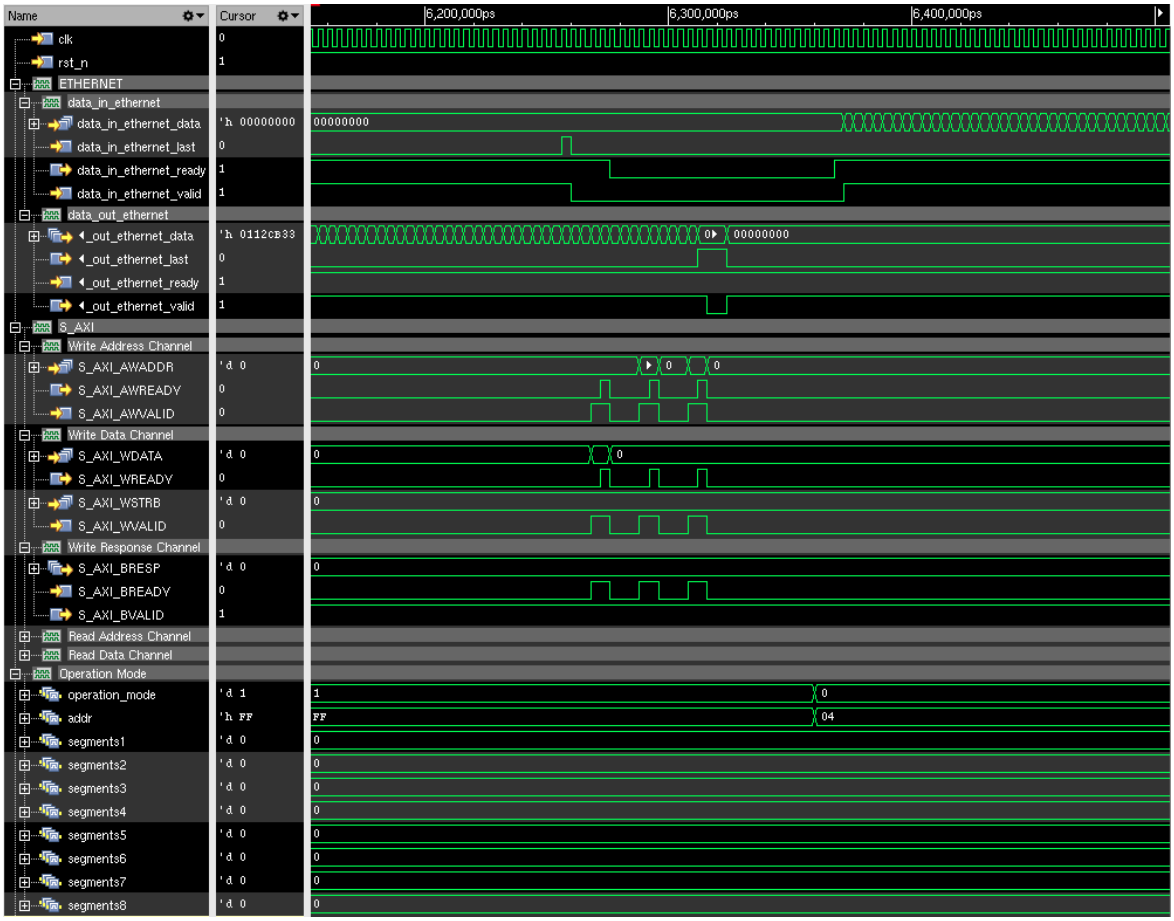

Figura 30: Simulación en SystemC de un cambio de modo de operación

Para realizar una lectura del modo de operación y del estado de la UD es necesario utilizar los canales *Read Address Channel* y *Read Data Channe*l del puerto de configuración. La Figura 31 muestra cómo al activarse la señal "AR_VALID" se inicia la lectura de la dirección seleccionada y se obtiene: en la primera lectura el modo de operación y las direcciones de los bloques IP aceleradores; en la segunda lectura los segmentos correspondientes a los primeros cuatro bloques IP aceleradores; y en la tercera lectura el resto de segmentos.

De igual forma, en la Figura 32 se presenta un ejemplo de transmisión de datos a los bloques IP aceleradores cuando el modo de operación es *Segmentation* y solo hay cuatro bloques activos. Se

puede observar cómo el paquete es dividido y enviado a cada uno de los cuatro bloques, y hasta que no finaliza el paquete y no se devuelve la respuesta a la UD no empieza a realizarse la transmisión al bloque MAC Ethernet receptor.

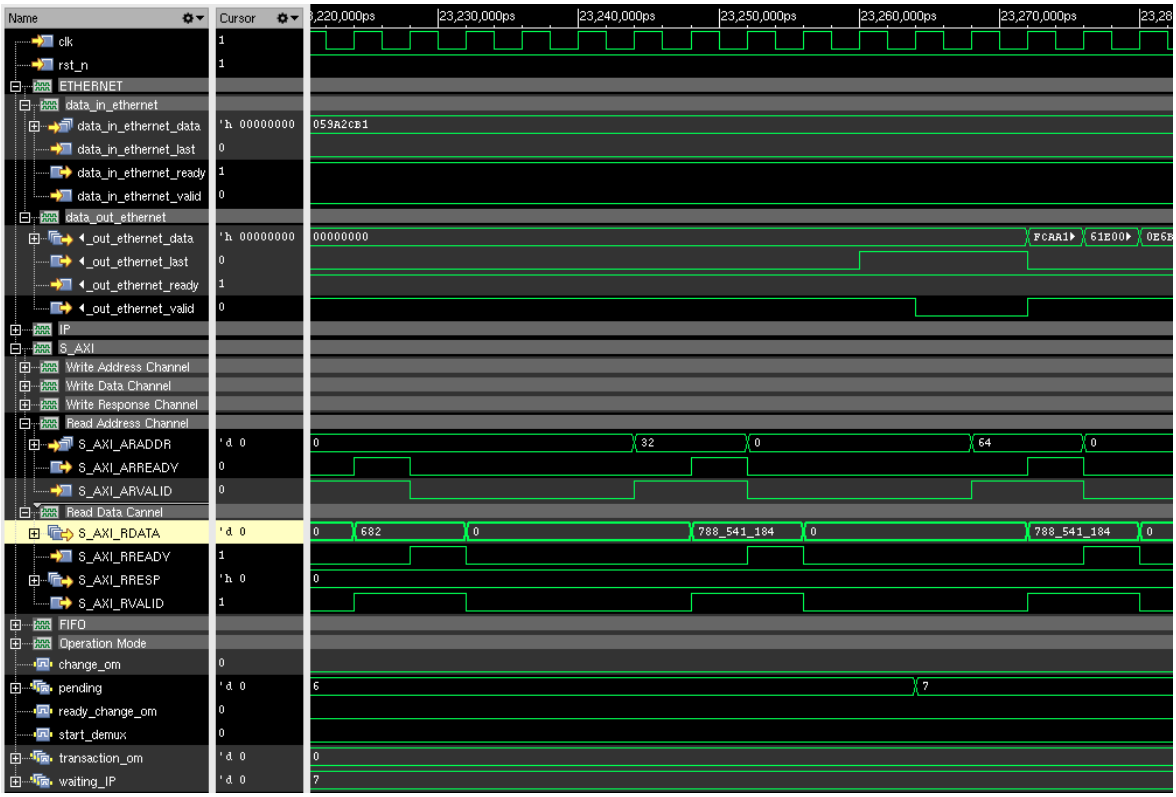

Figura 31: Simulación en SystemC de una lectura a través del puerto de configuración

Al realizar la simulación en SystemC también se ha comprobado la correcta respuesta de los bloques IP aceleradores, así como de las memorias FIFOs implementadas. Además, se ha efectuado una simulación muy similar a la mostrada con anterioridad, pero no permitiendo siempre el envío de los paquetes a la unidad de red. Para ello, se ha realizado una modificación en el *testbench* "IPblock_test", de forma que no siempre devuelva un "0" como respuesta del bloque IP acelerador, sino que alterne valores.

De esta manera, se comprueba que la UD es capaz de descartar los bloques cuando es necesario y continuar con su ejecución normal, sin perder ningún paquete. Por último, se ha realizado una simulación en la que los bloques IP aceleradores, los cuales no siempre se encuentran activos para la recepción de datos. Por este motivo, la UD debe comprobar y esperar a que se encuentren activos los bloques IP aceleradores necesarios para el envío de paquetes. Estas dos simulaciones se han realizado siguiendo la misma estructura que en la simulación mostrada en las figuras anteriores, en lo referido al envío de paquetes y la realización de lecturas y escrituras por parte del puerto de configuración.

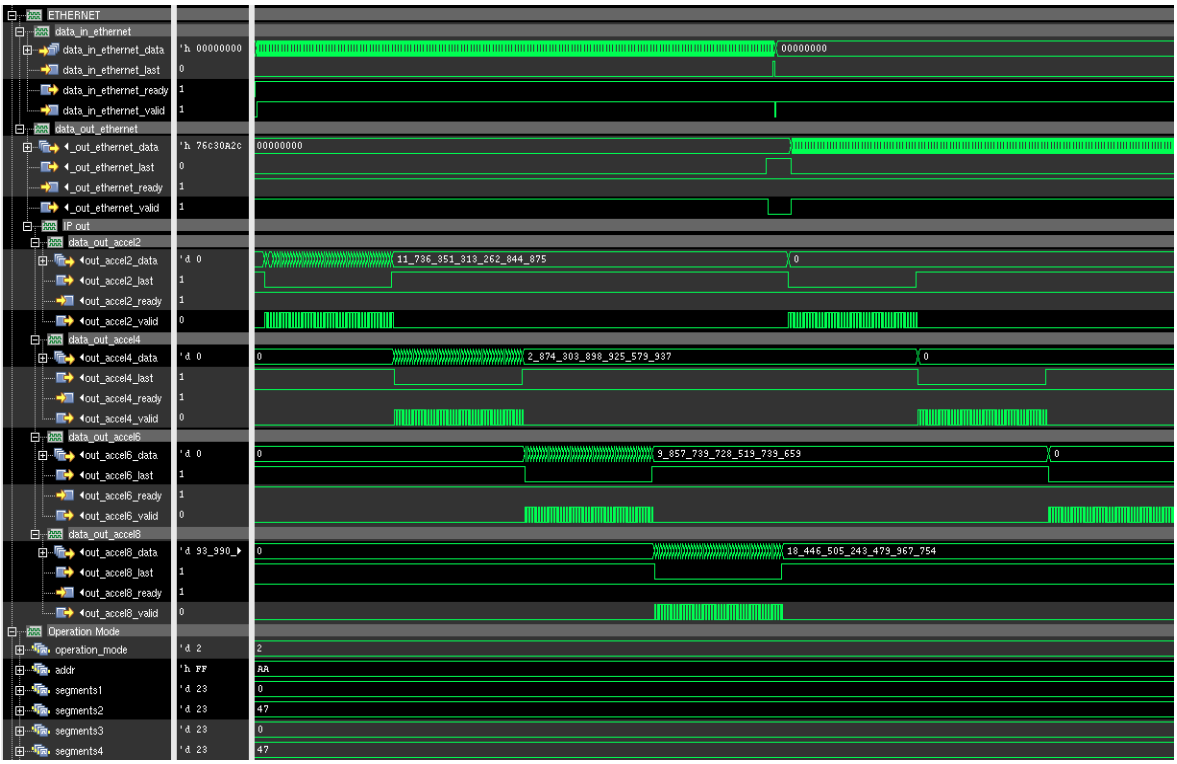

Figura 32: Simulación en SystemC de transmisión de datos en modo Segmentation

Gracias al conjunto de todos estos *testbench* realizados y los distintos casos estudiados se puede asumir el correcto funcionamiento de la UD, procediendo a realizar la cosimulación del diseño.

4.7. Verificación: Cosimulación en SystemC y RTL

El siguiente paso a realizar tras comprobar en simulación el correcto funcionamiento del sistema, es verificarlo a través de una cosimulación. La cosimulación nos permite comprobar si el diseño generado en un lenguaje de alto nivel, como es SystemC, y un diseño RTL obtenido a partir del mismo, funcionan de igual manera. Gracias a ello, se obtiene la verificación del sistema, asegurando la funcionalidad del mismo.

El diseño RTL se ha obtenido a partir de la herramienta CtoS, tras realizar la síntesis del sistema, como se explicó en el capítulo 4. Para realizar la verificación de la UD se ha utilizado el mismo *testbench* explicado en el apartado anterior. El motivo para seleccionar dicho *testbench* es su complejidad para poder realizar todas las verificaciones de posibles modos de operación del sistema y determinadas situaciones, con tan solo un par de modificaciones en los archivos de *testbench*. En las siguientes imágenes se puede observar las señales correspondientes a la simulación en SystemC y las señales correspondientes al diseño RTL generado. Al mostrarse las señales de datos y control sobre un eje temporal es fácilmente verificable el diseño. En la Figura 33 se muestran las señales de entrada y salida de la UD a las unidades de red, donde se verifica que todos los paquetes son enviados

y recibidos de forma exacta. La Figura 34 a su vez muestra las señales de escritura del puerto de configuración, mostrando los diferentes cambios de modo de operación realizados.

Figura 33: Cosimulación de las señales de entrada y salida en System-C y RTL

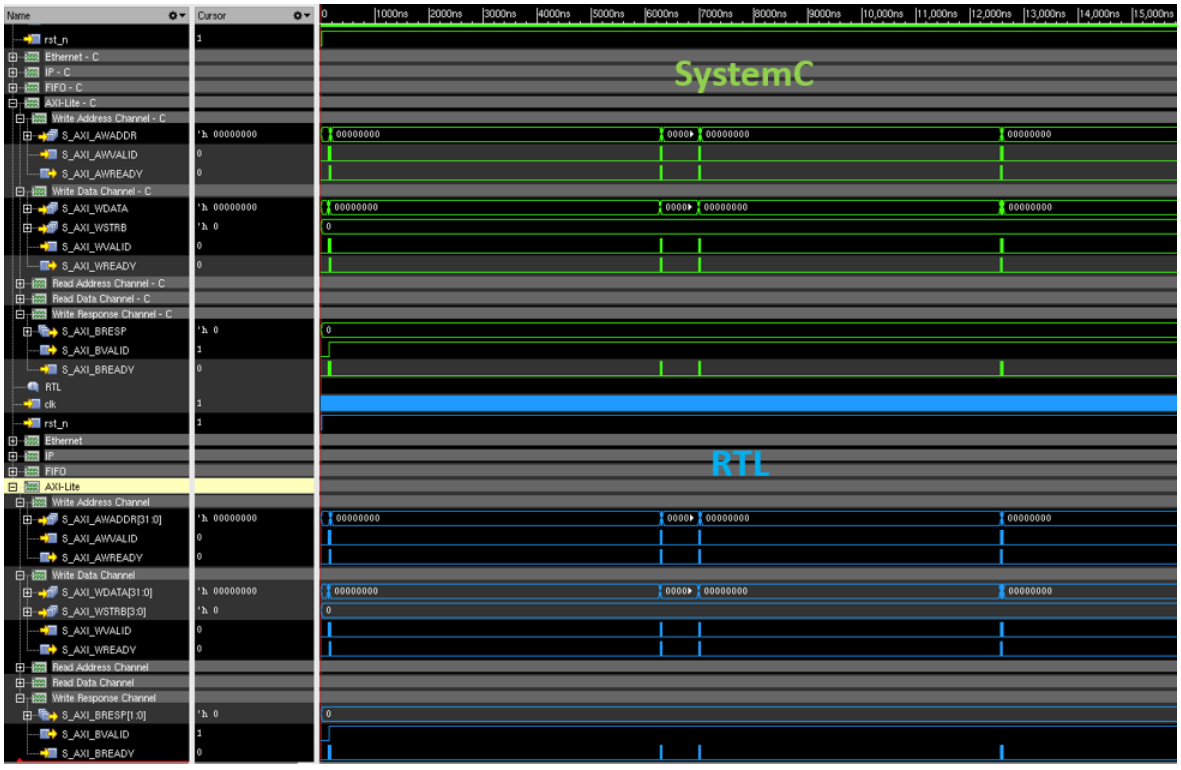

Figura 34: Cosimulación de la escritura del puerto de configuración en System-C y RTL

4.8. Obtención del bloque IP

A continuación, se mostrarán los pasos seguidos desde la escritura del código fuente hasta la obtención del bloque IP. Para ello, se hará uso de las herramientas descritas anteriormente, que incluyen SimVision, CtoS, Synplify y Vivado Design Suite. Antes de nada, se debe explicar el flujo de trabajo que se ha utilizado. Para la realización de este trabajo se ha partido de la creación de una serie de archivos que facilitan el uso de las diferentes herramientas y proporcionan un flujo de trabajo más rápido y sencillo de utilizar. Se basa en el uso de un *workspace* que incluye diferentes carpetas con los diferentes archivos necesarios para cada herramienta de trabajo. También incluye un *makefile* que permite la llamada a las diferentes herramientas, sin necesidad de incluir todos los archivos en cada herramienta, pues gracias al *workspace* se realiza de forma inmediata.

Como consecuencia, los pasos a seguir para la obtención del bloque IP con la utilización del *workspace* son los siguientes:

1. Una vez escrito el código fuente del módulo deseado, así como sus correspondientes *testbench* para comprobar su correcto funcionamiento, se pueden realizar las modificaciones pertinentes en el *workspace* para su posterior utilización. En la Figura 35 se observa la distribución de carpetas que conforman el entorno de trabajo de nuestro proyecto.

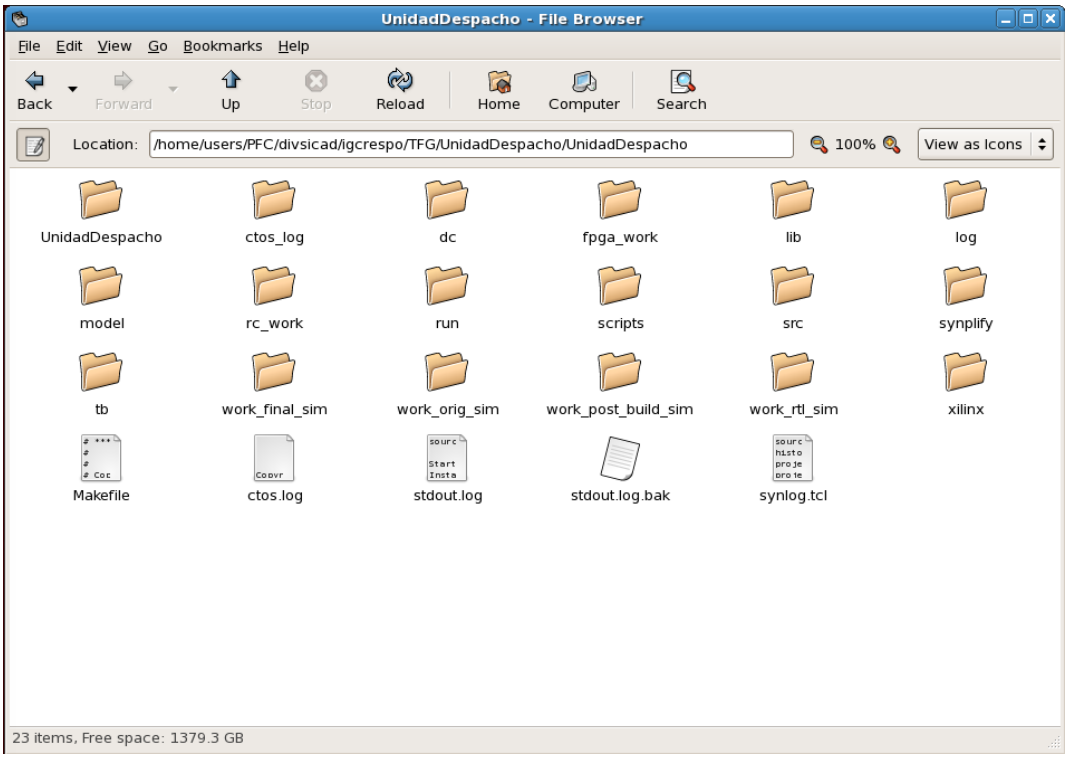

Figura 35: Conjunto de carpetas del módulo UD

2. Antes de efectuar las modificaciones en el *workspace* se debe realizar un cambio en el nombre de la carpeta principal por el nombre del nuevo módulo. En este caso el nombre es "UnidadDespacho".
3. A continuación, se debe acceder al archivo "Makefile" que se encuentra dentro de la carpeta principal del *workspace* y, también, se debe acceder a la carpeta "scripts", cuyo contenido se puede ver en la Figura 37. Dentro de esta carpeta existen tres archivos que se deben modificar: "config_global.tcl", "ctos.tcl" y "ctos_setup.tcl". En estos cuatro archivos, incluyendo el "Makefile", se reemplazará el nombre del módulo anterior por el nuevo módulo a utilizar. En este caso será "UnidadDespacho". También se debe realizar otra cambio en "ctos_setup.tcl", pues deben incluirse los nombre de los archivos fuente y los archivos de cabecera, tal y como se muestran en la Figura 36. En el archivo "Makefile" se realiza algo parecido, pero incluyendo un listado de los archivos fuente y de cabecera del *testbench*, al modificar la siguiente línea: "DESIGN_TB_FILES = ./tb/UnidadDespacho_test.cpp ./tb/IPblock_test.cpp ./tb/sc_main_UnidadDespacho.cpp ".

```
set_attr source_files [list src/UnidadDespacho.cpp] /designs/$modulo
set_attr header_files [list src/UnidadDespacho.h src/data_struct.h] /
designs/$modulo
```

Figura 36: Listado de los archivos fuente y de cabecera de "ctos_setup.tcl"

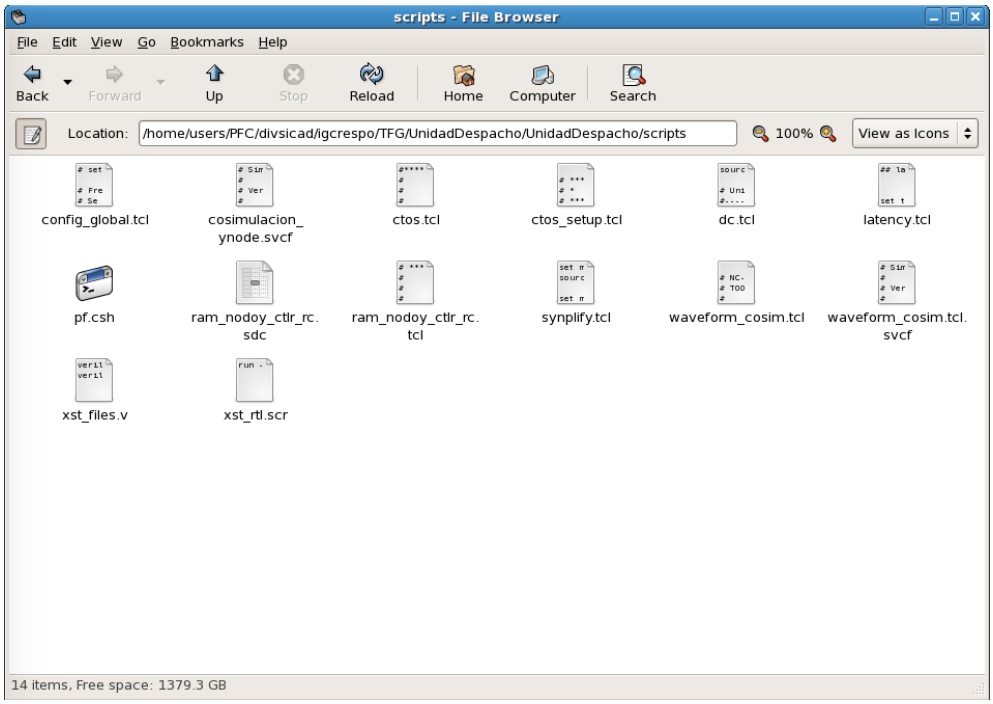

Figura 37: Carpeta "scripts" del entorno de trabajo

4. Tras realizar estas modificaciones, ya se pueden incorporar los archivos de nuestro módulo, incluyendo el código fuente y los *testbench* correspondientes. Para ello, se accede a la carpeta "src" y se incluyen únicamente aquellos archivos que correspondan al código fuente del módulo, teniendo en cuenta que todos los archivos de cabecera requeridos para su implementación también deben encontrarse en esta carpeta. Posteriormente, se accede a la carpeta "tb" y se incluyen todos los archivos correspondientes al *testbench* del módulo, teniendo en cuenta la misma consideración con los archivos de cabecera que con la carpeta anterior.
5. Tras la realización de los cuatro pasos anteriores se puede proceder a la verificación del módulo y a la utilización del *workspace*. Para la verificación del sistema se accede a la herramienta SimVision, que permite la visualización de las señales del sistema, facilitando su interpretación y su correcto funcionamiento. Para ello, simplemente se debe abrir un terminal y asegurar que se encuentra en la carpeta principal del *workspace*, que en este caso coincide con "UnidadDespacho", para poder hacer una llamada al *makefile*. En la Figura 38 se pueden observar las diferentes tareas que puede ejecutar el *makefile* y los comandos correspondientes. Posteriormente, se debe introducir el comando "make_orig_gui", el cual realizará automáticamente el arranque de la herramienta SimVision con el módulo seleccionado permitiendo la simulación del mismo utilizando la interfaz GUI.

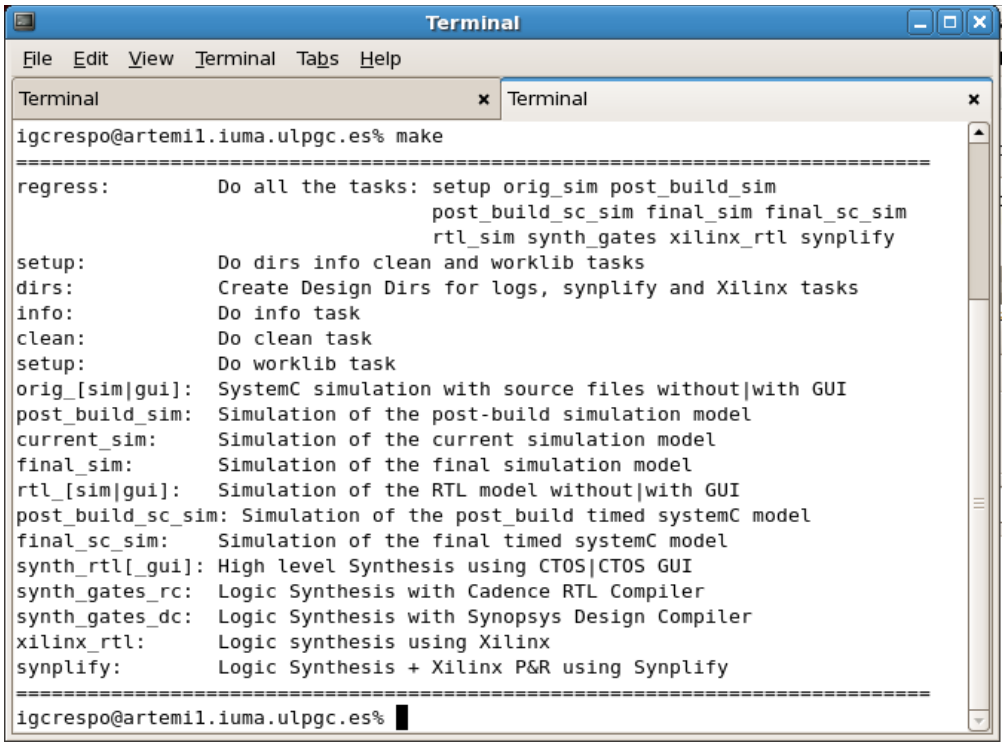

Figura 38: Lista de comandos proporcionados por el makefile

6. Tras realizar las diferentes comprobaciones que sean necesarias en el SimVision y asegurar el correcto funcionamiento del módulo, se debe cerrar la herramienta SimVision y acceder a la herramienta CtoS para realizar la planificación y la síntesis necesaria del módulo y posteriormente poder obtener la cosimulación del sistema con RTL. Para ello, se vuelve a acceder a un terminal y este se debe encontrar en la carpeta principal para poder introducir el comando "make_synth_rtl_gui". Esto permitirá realizar la síntesis en alto nivel utilizando la herramienta CtoS y la interfaz GUI.
7. Una vez abierta la herramienta CtoS debemos asegurarnos que el dispositivo seleccionado coincida con el deseado. Se ha seleccionado, la placa de desarrollo Zynq como se explicó anteriormente y como se puede observar en la Figura 39.
8. En la herramienta CtoS se debe realizar la construcción del módulo y posteriormente, seleccionar las memorias a utilizar y cuál será su comportamiento. En la Figura 41 se muestran las opciones seleccionadas para la construcción del módulo. Gracias al *workspace* no es necesario añadir los archivos fuente y de cabecera, pues se realiza de forma automática, al igual que la selección del módulo principal. Para la selección de las memorias en este caso únicamente existe una llamada "accel_finish", como se puede observar en la Figura 40. Finalmente, en la Figura 42 se determina la memoria a utilizar para "accel_finish", siendo esta una memoria con lectura síncrona que se encuentra en RAM.

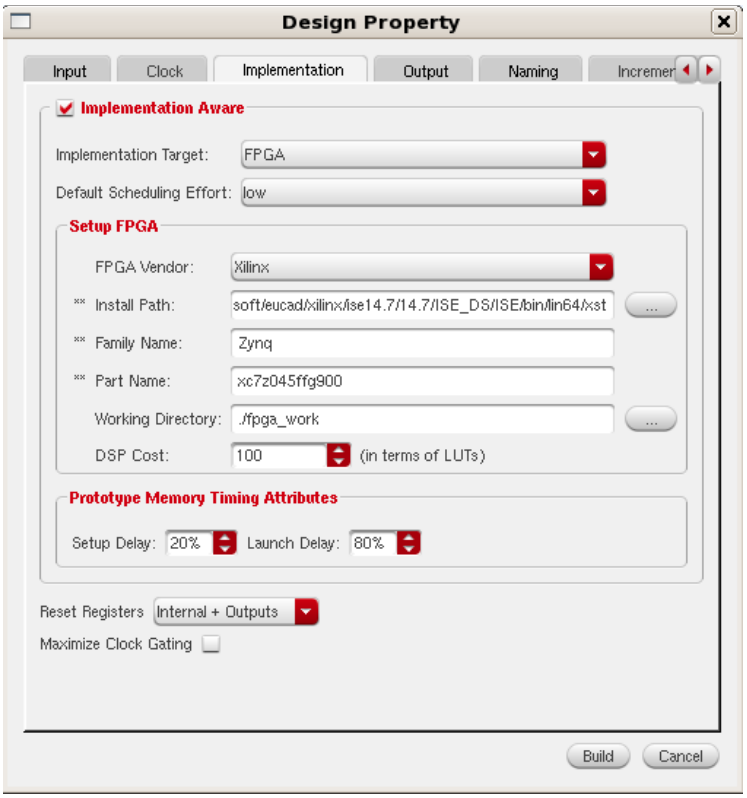

Figura 39: Propiedad del diseño en CtoS: Selección de la FPGA

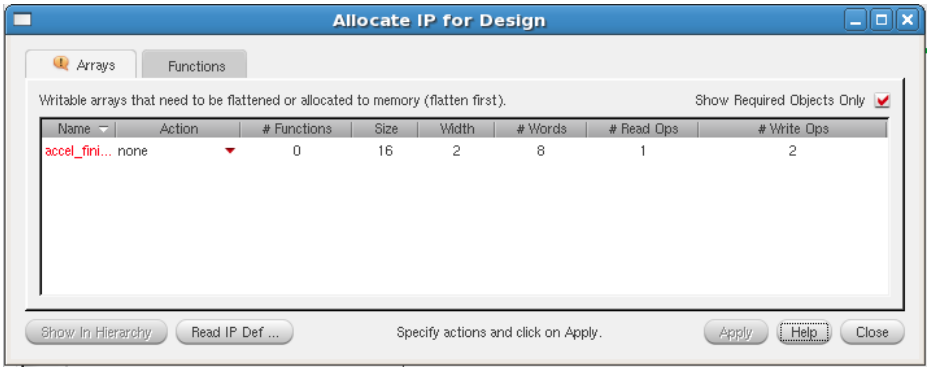

Figura 40: Arrays disponibles en el diseño en CtoS

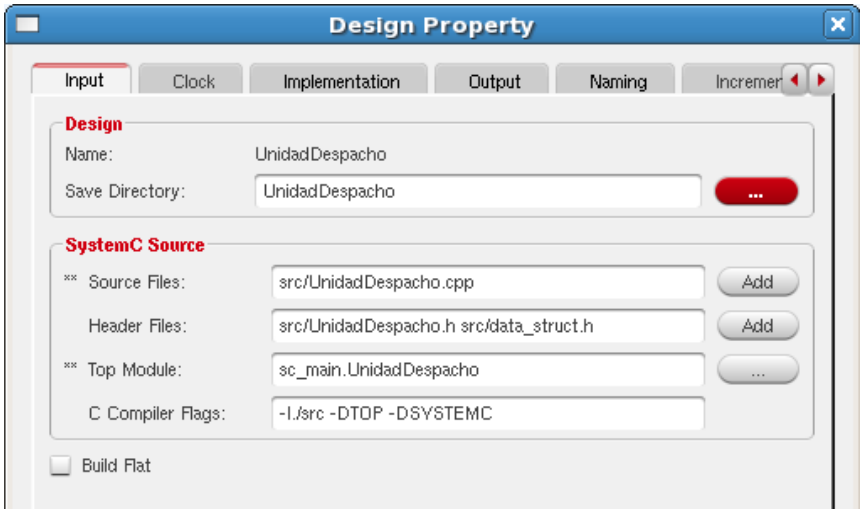

Figura 41: Opciones para la construcción del módulo en CtoS

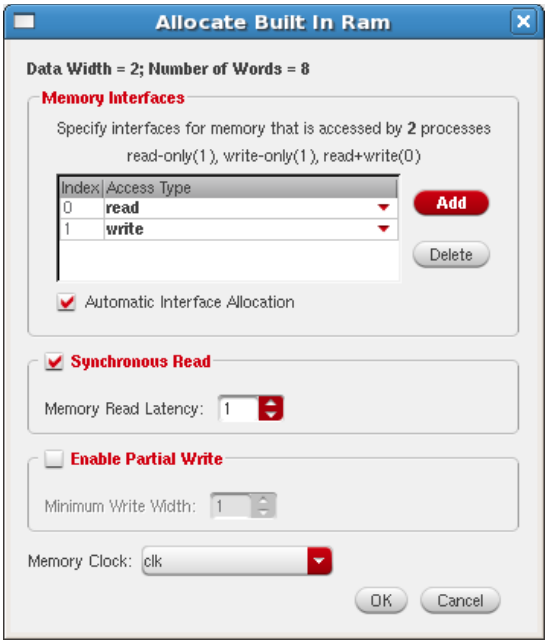

Figura 42: Selección de la memoria para "accel_finish" en CtoS

9. Tras haber realizado estos pasos, se puede iniciar la planificación y la síntesis, teniendo en cuenta que se puede efectuar con dos filosofías diferentes: "Top-down" o "Bottom-up". "Top-down" se realiza directamente sobre el módulo, mientras que "Bottom-up" permite la planificación individual de cada una de las funciones del módulo antes de obtener la síntesis global. En la Figura 43 se muestra la configuración realizada para este proyecto, utilizando metodología "Top-down" y con los valores predeterminados por la herramienta CtoS.

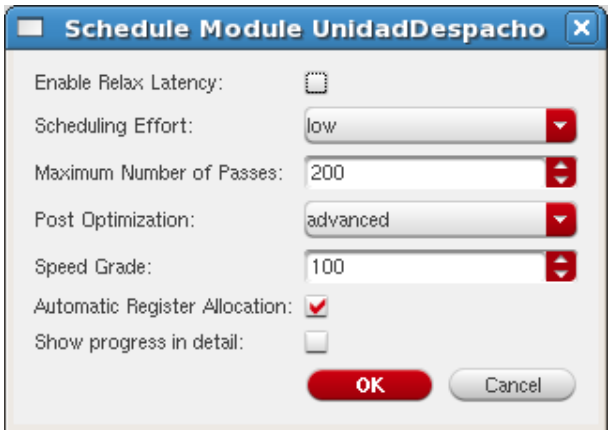

Figura 43: Configuraciones de la planificación y la síntesis en CtoS

10. Tras una correcta planificación, ya se obtiene el diseño RTL, que permite la ejecución de la cosimulación en SimVision, y su posterior uso en Synplify. Simplemente, se debe guardar el diseño RTL en un solo archivo, como se muestra en la Figura 44, y salir correctamente de la herramienta CtoS, pues ya no será necesaria.
11. A continuación, vuelve a ser necesario el uso de un terminal que se encuentre en la carpeta principal como se ha explicado con anterioridad. Se introduce el comando "make rtl_gui" y se procede a abrir de forma automática nuevamente la herramienta SimVision. Sin embargo, esta vez no solo se podrá observar la simulación en el lenguaje de programación seleccionado, sino también en RTL. Gracias a ello, se procede a la cosimulación y a la verificación del sistema. Una vez comprobado que ambas simulaciones funcionen de forma exacta se puede cerrar el SimVision, pues ya no será necesario.
12. El siguiente paso a realizar tras la verificación del sistema es acceder a la herramienta Synplify que permite realizar la síntesis lógica del módulo sobre RTL. Para ello, se debe volver a acceder a un terminal e introducir el comando "synplify_premier_dp". Antes de nada, se debe seleccionar el fichero RTL correcto para la correcta síntesis lógica del bloque. En laFigura 45 se muestra la selección del fichero y donde se encuentra almacenado. La configuración seleccionada ha sido "Rev_5" como se observa en la Figura 46.

Figura 44: Generación del archivo RTL en CtoS

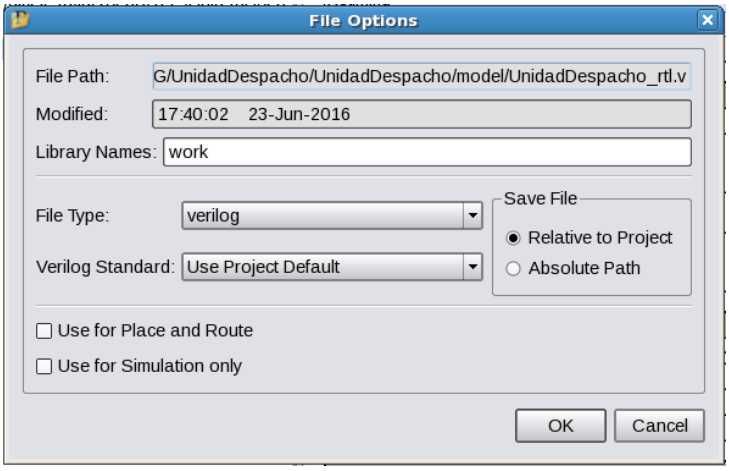

Figura 45: Selección del fichero RTL en Synplyfy

Figura 46: Selección de la configuración en Synplify

13. También, son importantes las características escogidas para el diseño. Se ha seleccionado la realización de síntesis avanzada y de síntesis de 64-bits, compilación FSM, compartición de recursos, *pipelining* y *retiming* como se puede observar en la Figura 47. No se debe olvidar, asegurar la selección del dispositivo seleccionado para la implementación del sistema. En la Figura 48 se muestra el dispositivo utilizado y las características deseadas para la implementación, que en este caso son la posibilidad de combinar LUTs, la comprobación de lectura y escritura en RAM y el análisis de las propiedades.

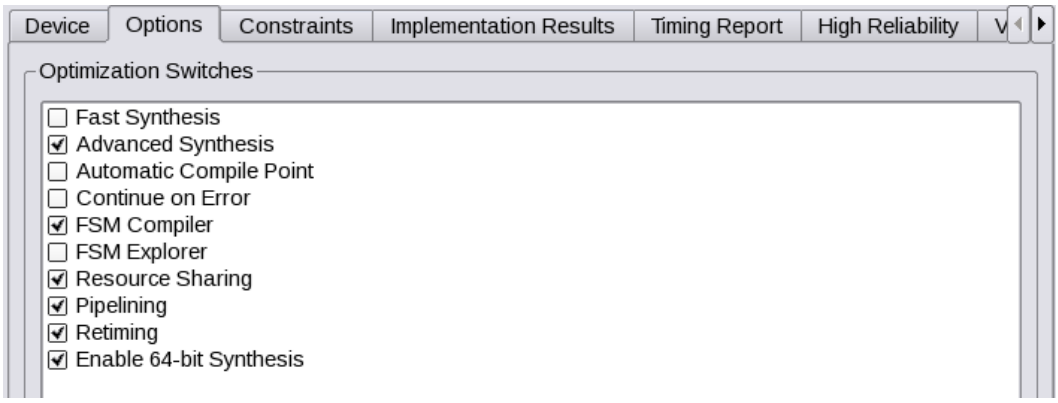

Figura 47: Selección de opciones en Synplify

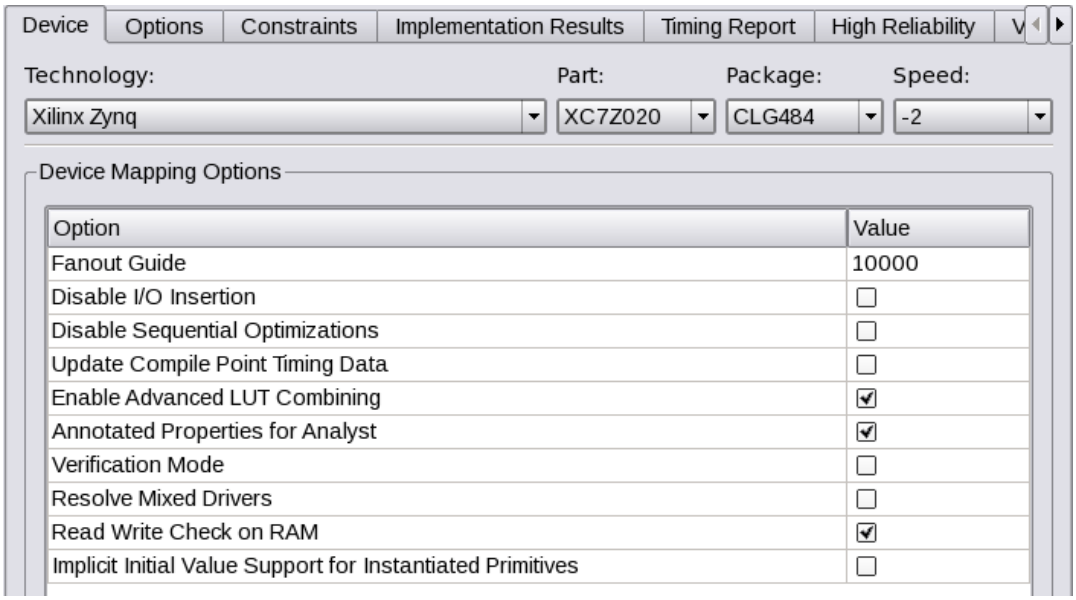

Figura 48: Selección del dispositivo y sus opciones en Synplify

14. A continuación, se realiza la compilación, el premapeado y, finalmente, el mapeado final y la optimización del bloque. En la Figura 49 se puede ver los resultados finales del bloque "UnidadDespacho", entre los que destacan una frecuencia de funcionamiento de 241,6 MHz, un uso del 1.464 LUTs y ninguna BRAM.

Area Summary			
I/O ports (io_port)	1494	Non I/O Register bits (non_io_reg)	2125 (1%)
I/O Register bits (total_io_reg)	0	Block Rams (v_ram)	0 (140)
DSP48s (dsp_used)	0 (220)	LUTs (total_luts)	1464 (2%)
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	**Req Freq (req_freq)**	**Est Freq (est_freq)**	**Slack (slack)**
UnidadDespacho_rtl|clk	284.3 MHz	241.6 MHz	-0.621
Detailed report		Timing Report View	

Figura 49: Resultados de la síntesis lógica en Synplify

15. La ruta crítica del sistema es de especial importancia, pues determina la frecuencia máxima a la que puede funcionar el bloque. En este caso el periodo de trabajo de la UD es de 4,14 ns, encontrándose la ruta crítica en la función "demux()", explicada con anterioridad, desde la obtención del dato al realizar la lectura en la FIFO2 hasta su escritura en los bloques IP aceleradores correspondientes.
16. Si los resultados obtenidos coinciden con las características que debe cumplir el bloque, ya es posible obtener un archivo EDIF que incluirá la solución del bloque tras la síntesis lógica. Este archivo es esencial para la obtención del bloque final necesario para a implementación. Una vez obtenido el archivo, se cierra la herramienta Synplify.
17. Ya solo es necesario usar la herramienta Vivado Design Suite para poder obtener el bloque IP listo para la implementación en la placa de desarrollo. Para ello, se abre la herramienta Vivado Design Suite y se crea un nuevo proyecto RTL. En la Figura 50 se puede ver la creación del proyecto, donde se introduce el nombre "UnidadDespacho" y se decide donde se ubicará el proyecto. No se debe olvidar añadir el archivo EDIF en los archivos fuente al realizar la creación del proyecto como se observa en la Figura 51.
18. También, se selecciona la placa de desarrollo que se va a utilizar. Para este caso se va a emplear la placa ZedBoard, que es el kit de desarrollo y evaluación de Zynq. Finalmente, Vivado Design Suite nos muestra un resumen de las características del diseño para comprobar que todo sea correcto, como se muestra en la Figura 52.

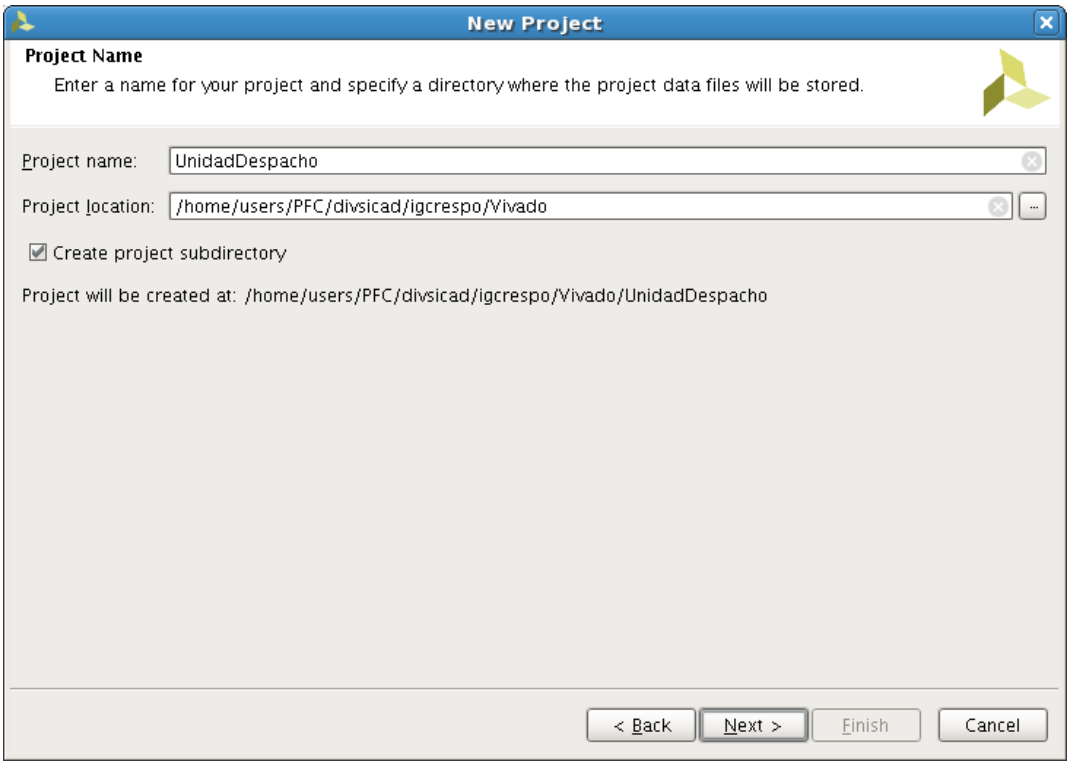

Figura 50: Creación de un nuevo proyecto en Vivado

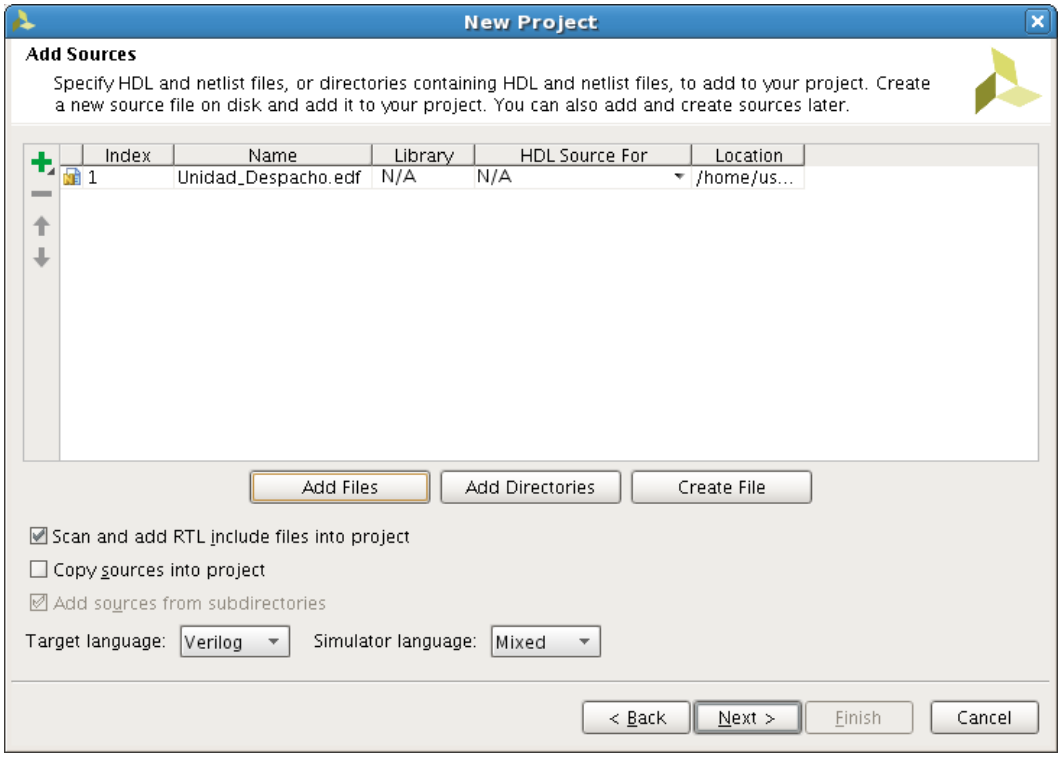

Figura 51: Adición de archivo EDIF en Vivado

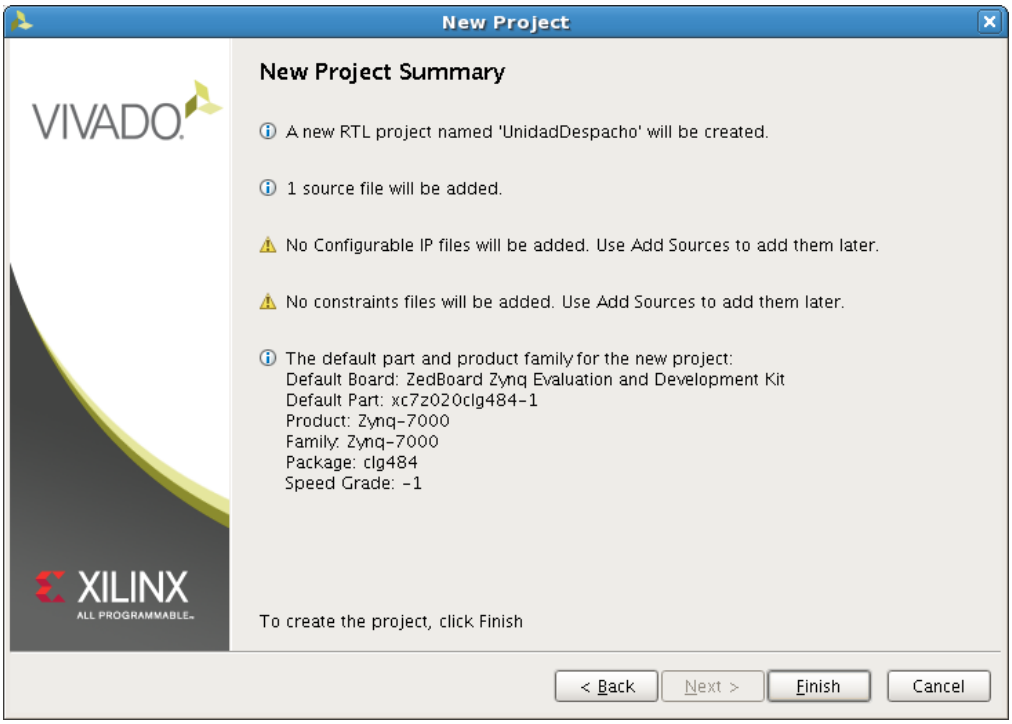

Figura 52: Resumen del nuevo proyecto en Vivado

19. A continuación, se realiza la creación de un nuevo bloque IP, para ello se accede a "Create and Package New IP" y se deja por defecto la selección de empaquetar el proyecto actual. En la Figura 53 se muestra la ubicación seleccionada, que debe coincidir con donde se ha almacenado el archivo EDIF del sistema.

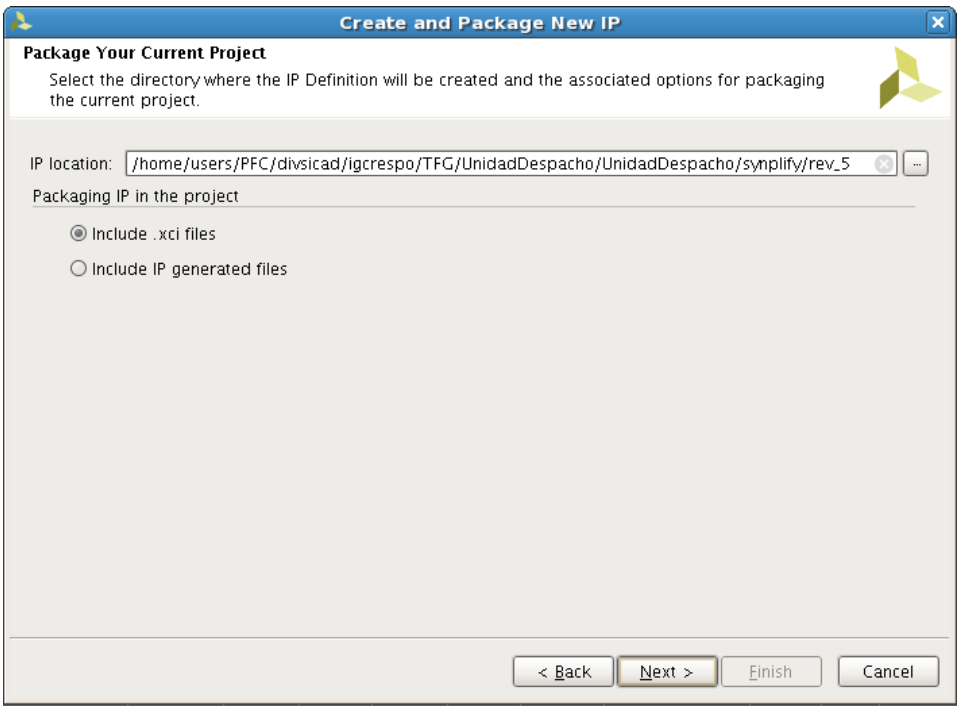

Figura 53: Creación de un nuevo bloque IP en Vivado

20. Una vez realizadas las configuraciones necesarias para el nuevo bloque IP se abre un sumario del proyecto. En este proyecto los puertos de entrada y salida del módulo son bastante numerosos. Esto se debe a la utilización de protocolos de comunicación, como son AXI4-Stream y AXI4-Lite. Por ello, se generan diferentes interfaces que junten cada conjunto de señales relacionadas. Para ello, se debe seleccionar crear una nueva interfaz, teniendo en cuenta el tipo de interfaz que es, el protocolo de comunicación que utiliza y si es maestro o esclavo. Posteriormente, se debe realizar un mapeado de los puertos, para unir los puertos lógicos de la interfaz con los puertos físicos. Para asegurarnos de no cometer errores activamos las casillas de "Filter Incompatible Physical Ports" y "Hide Mapped Ports", para no realizar conexiones incompatibles a nivel físico, ni conectar dos puertos lógicos a uno físico. En la Figura 54 se puede observar un ejemplo de cómo se crean las interfaces para una conexión AXI4-Stream. Tras elaborar las interfaces para todos los puertos de entrada y salida posibles, las entradas y salidas del bloque quedan tal y como se muestra en la Figura 55. No se debe olvidar seleccionar el reloj que utilizará cada una de las interfaces, que en este caso será el mismo para todas.

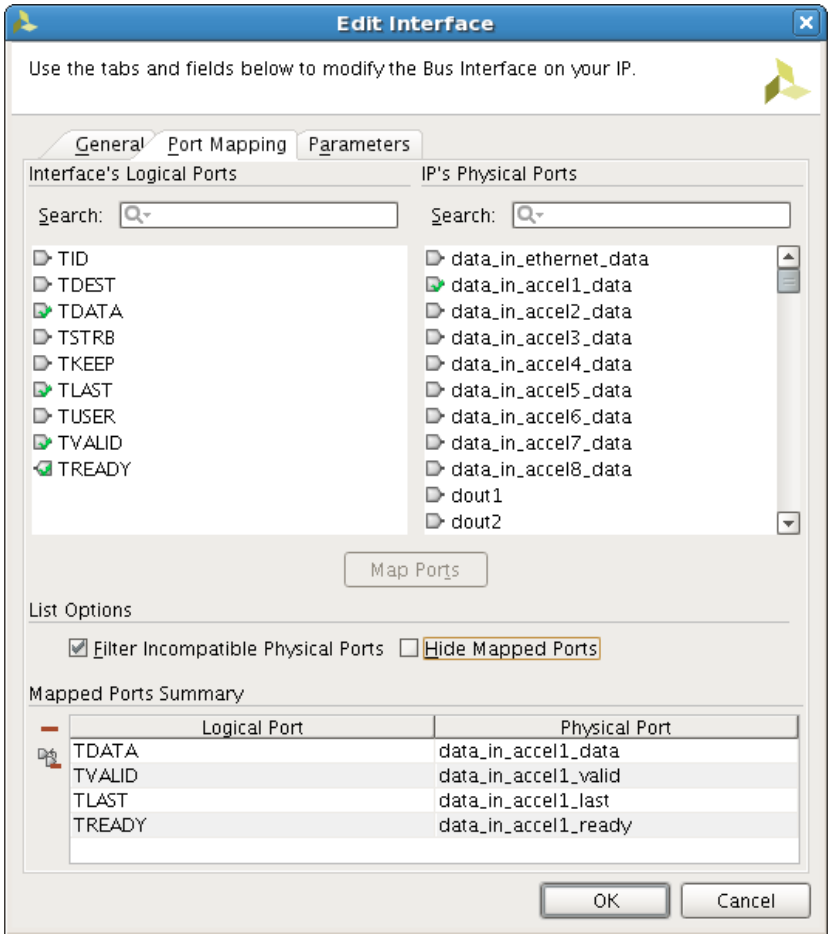

Figura 54: Creación de una interfaz para el nuevo bloque IP en Vivado

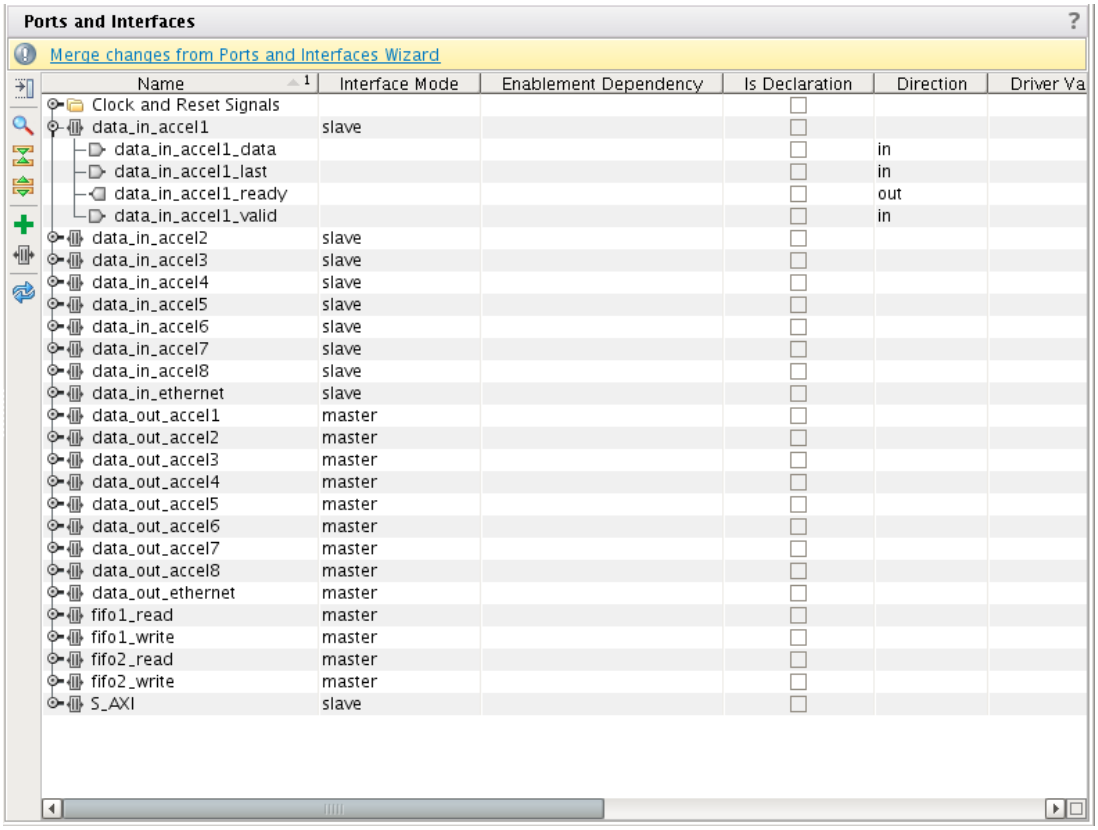

Figura 55: Puertos e interfaces del nuevo bloque IP en Vivado

21. En la Figura 56 se muestra la distribución del bloque tras los cambios realizados. Finalmente, para la obtención del bloque IP solo se debe accede a "Review and Package" y seleccionar "Package IP" para exportar el bloque IP que contiene el módulo con el que se está trabajando, como se muestra en la Figura 57.

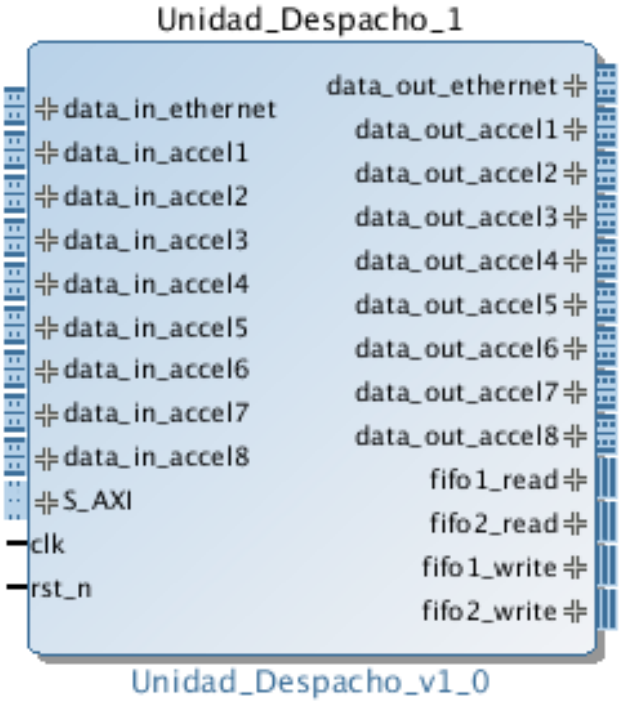

Figura 56: Bloque IP de la UD

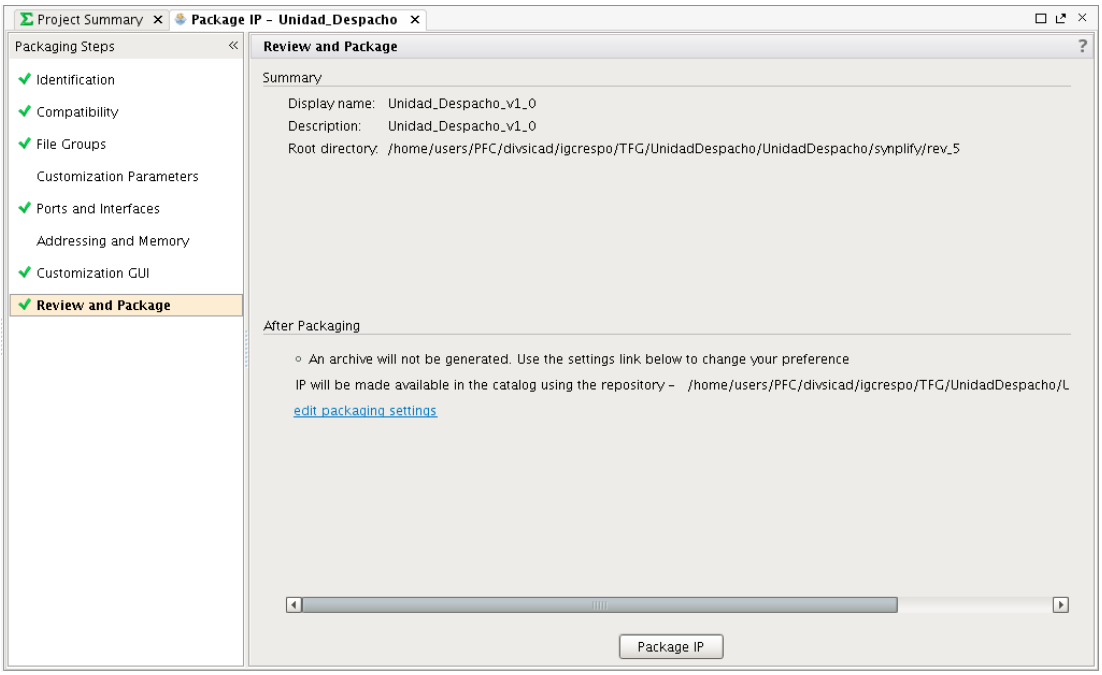

Figura 57: Obtención del nuevo bloque IP en Vivado

Capítulo 5. Integración e implementación

A lo largo de este capítulo se explicarán los pasos seguidos para la integración e implementación del sistema en la plataforma seleccionada, que en este trabajo se trata de la placa de prototipado ZedBoard. Inicialmente, se describirá el diagrama de bloques diseñado para la verificación del sistema en el dispositivo, y posteriormente se explicarán los pasos seguidos para la obtención del *bitstream* necesario para la programación de la placa de desarrollo.

5.1. Diseño de la plataforma

Tras obtener el bloque IP de la UD ya se puede realizar el diseño de la plataforma, a través de un diagrama de bloques. Pero antes de ello, se lleva a cabo un pequeño análisis de los bloques IP que se han utilizado. De esta manera, se aclara el funcionamiento de cada bloque y se explica por qué es necesaria su integración en la plataforma para la implementación del bloque UD y su verificación. Posteriormente, se explican las interconexiones realizadas y se muestra el diagrama de bloques final, que será utilizado en la validación final del sistema.

5.1.1. Bloques IP

Para la realización del proyecto, se han empleado cinco bloques diferentes, incluyendo el bloque IP de la UD descrito en el capítulo 4. Algunos de estos bloques se utilizan más de una vez en el diseño de la plataforma. Estos bloques son el bloque IP FIFO, el bloque IP de PS del Zynq, el bloque IP para el DMA y el bloque IP del acelerador. Todos ellos se encuentran integrados en el catálogo de bloques IP de Xilinx, excepto el bloque IP del acelerador y el bloque IP de la UD. A continuación, se mostrará en detalle cada uno de estos bloques, incluyendo su funcionamiento, los puertos de entrada y salida disponibles y los parámetros escogidos para su ejecución en el sistema diseñado.

- *ZYNQ7 Processing System*. Inicialmente es necesario incluir el bloque IP que incorpora el sistema de procesamiento de Zynq. Es el bloque central de la plataforma que permite instanciar el PS en la FPGA y realizar la comunicación con el PL y el resto de elementos de

la placa de prototipado, incluyendo la placa ZedBoard. En la Figura 58 se muestran los diferentes puertos de entrada y salida de los que dispone el bloque, incluyendo la señal de reloj y de *reset*. Como se observa, se hace uso del protocolo de comunicaciones AXI para realizar conexiones entre el PS y el PL. La configuración escogida para el bloque ha sido activar un reloj del PL, el "FCLK_CLK0", y ajustar su frecuencia de funcionamiento a 150 MHz, pues el DMA que se utilizará para la verificación no puede funcionar a la frecuencia máxima del bloque, que es 200 MHz. También, se ha activado una interfaz AXI esclava (S_AXI_HPO) para conectar el bloque con el DMA explicado a continuación.

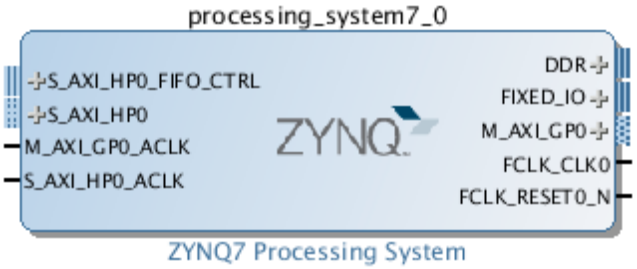

Figura 58: Bloque IP del sistema de procesamiento de Zynq

- *AXI Direct Memory Access*. Es el acceso a memoria directo o DMA, que utiliza el protocolo de comunicaciones AXI. Su función es permitir transacciones al generar un acceso a memoria directo, sin necesidad de transitar la CPU del sistema. El PS es el encargado de enviar los datos a la UD diseñada, pero no puede ejercer control sobre esta debido a la necesaria utilización del DMA para poder realizar la transacción. Para la validación del sistema se utiliza este bloque junto al PS para realizar el envío de datos predeterminados, como sustitución a la unidad de red para la cuál no sería necesario el uso del DMA. La estructura del bloque IP se muestra en la Figura 59, incluyendo una configuración simple. Esta configuración se puede observar en la Figura 60, donde se habilitan los canales de lectura y escritura y se seleccionan 32 bits para las direcciones y 14 bits para los registros del *buffer*.

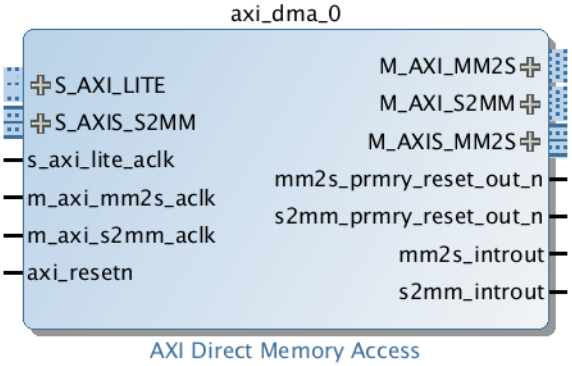

Figura 59: Bloque IP del DMA con protocolo AXI

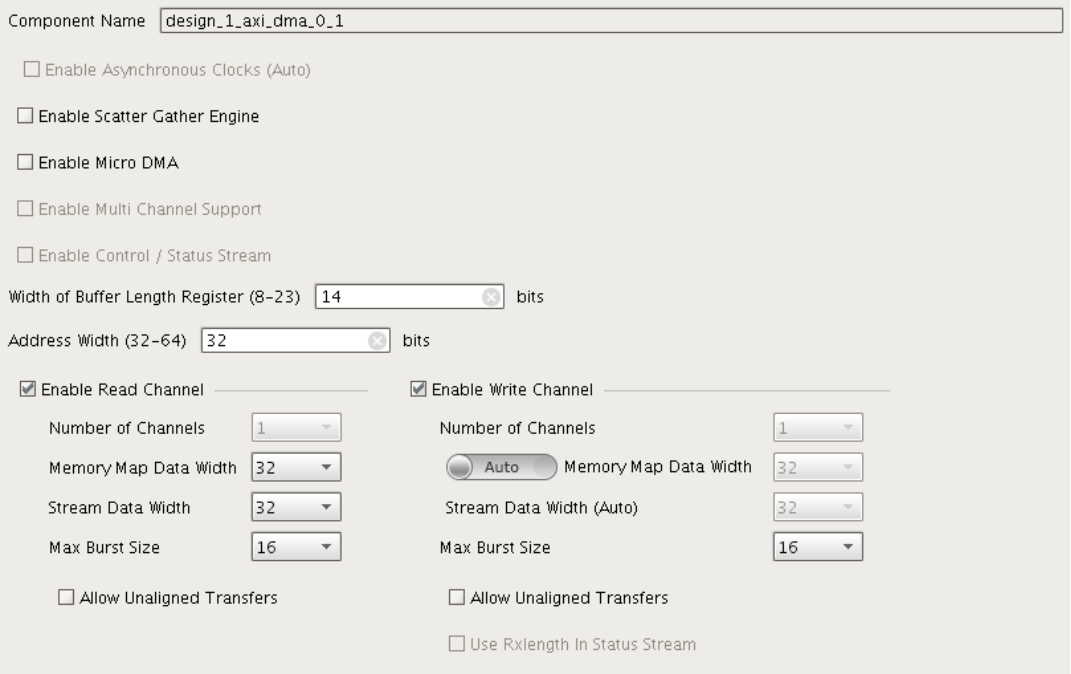

Figura 60: Configuración del bloque IP DMA

- *FIFO Generator*. Para el correcto funcionamiento del bloque IP UD, es necesario el uso de dos memorias tipo FIFO. Para ello, se utiliza un generador de memorias FIFO incluido en el repositorio de bloques IP de Vivado. En la Figura 61 se muestran los puertos de entrada y salida del bloque. La configuración utilizada para ambos bloques es la misma, y se basa en un tipo de interfaz nativo que utiliza el reloj común del BRAM e implementa una FIFO estándar. Sin embargo, para cada una de ellas se debe seleccionar unos parámetros diferentes para el puerto de datos. Para la primera FIFO se selecciona un ancho de lectura y escritura de 33 bits (32 bits de datos, más un *boolean* que indica la finalización del bloque) y un total de 2048 posiciones disponibles. Para la segunda FIFO se selecciona un ancho de lectura y escritura de 65 bits (64 bits de datos y un *boolean*) y un total de 1048 posiciones disponibles. La señal de *reset* se debe desactivar y se debe activar el *flag* requerido para la señal *almostfull*.

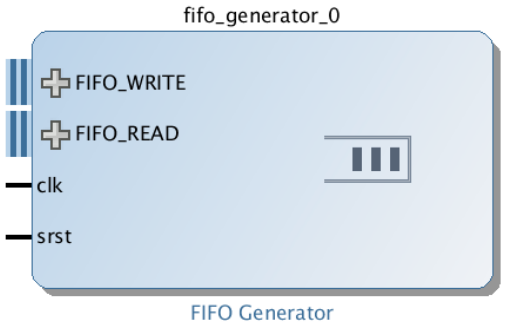

Figura 61: Bloque Ip del generador FIFO

- *AccelIP_rtl_v1_0*. Por último, es necesaria la implementación del bloque acelerador encargado de procesar los datos que recibe la UD. Sin embargo, este bloque no se encuentra desarrollado, por lo que ha sido necesaria desarrollar un bloque que cumpla con los requisitos necesarios para la validación del bloque diseñado. Para ello, se ha creado un módulo en SystemC que dispone de una entrada y una salida de datos que cumplen con el protocolo AXI4-Stream, un señal de reloj, una señal de *reset* y un señal de 64 bits "answer", tal y como se muestra en la Figura 62. El funcionamiento de este bloque es sencillo. Siempre se encuentra esperando la recepción de un paquete de datos y una vez comienza la recepción espera hasta obtener el paquete completo, para posteriormente devolver como respuesta un único paquete que coincidirá con la señal "answer" obtenida a través del puerto de entrada. Esta señal vendrá determinada por un bloque IP VIO que se explicará más adelante. Una vez terminado el desarrollo del módulo acelerador y realizada su verificación, es necesario seguir los pasos explicados en el capítulo 4 para la obtención del bloque IP. Finalmente, ya se dispone del bloque listo para la validación del diseño.

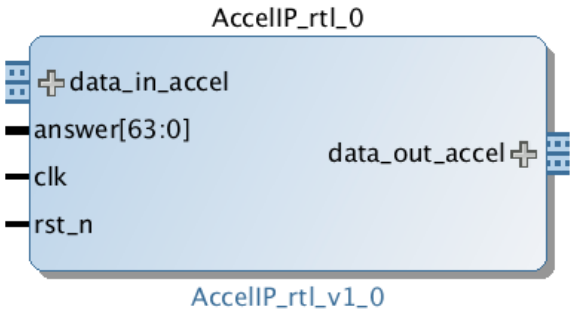

Figura 62: Bloque IP del acelerador

Sin embargo, también son necesarios ciertos bloques que son generados automáticamente al realizar la conexión del DMA y el PS de Zynq. Estos bloques son los siguientes:

- *Processor System Reset*. Es el *reset* del PS de Zynq y el encargado de gestionar todas las señales de *reset* de los bloques integrados en la plataforma. En la Figura 63 se puede ver los puertos de entrada y salida de los que dispone el bloque

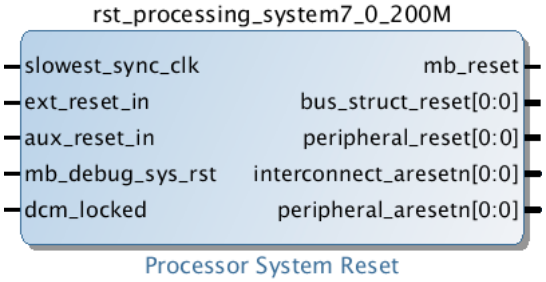

Figura 63: Bloque IP del reset del PS

- *AXI Interconnect*. Se encarga de realizar las interconexiones a los periféricos, cumpliendo con el protocolo de comunicaciones AXI. Su función es configurar y establecer las conexiones entre los diferentes bloques IP que conforman el diseño de la plataforma. Es capaz de gestionar diferentes conexiones de dispositivos maestros y esclavos, aunque dispongan de anchos de banda y frecuencias de reloj diferentes. Sin embargo, no es capaz de integrar el bus AXI4-Stream. No obstante, este módulo está compuesto por dos bloques IP diferentes; uno es "processing_system7_0_axi_periph" y "axi_mem_intercon". En las Figura 64 y Figura 65 se pueden observar ambos bloques y las señales de entrada y salida de las que dispone.

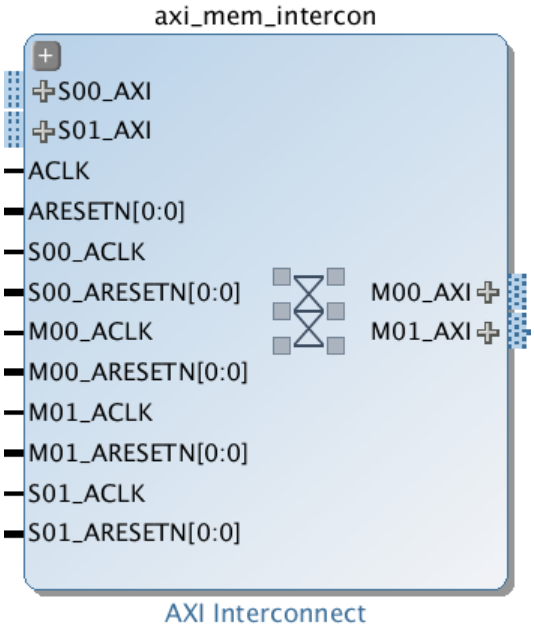

Figura 64: Bloque IP "axi_mem_intercon" de la interconexión AXI

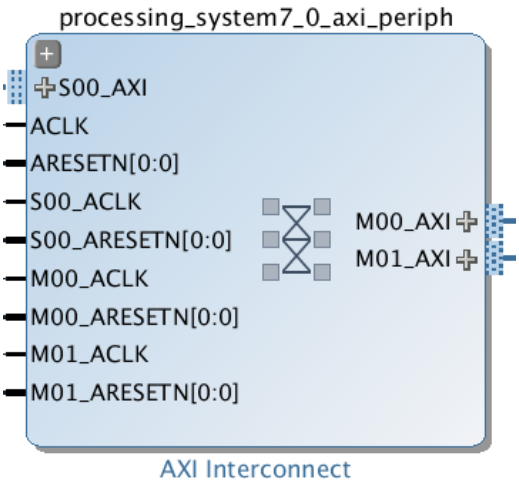

Figura 65: Bloque IP "axi_periph" de la interconexión AXI

Por último, no se debe olvidar la integración de los bloques de depurado, que son necesarios para comprobar el correcto funcionamiento del sistema:

- *VIO (Virtual Input/Output)*. Es una entrada y salida virtual, más conocida como VIO, utilizada para incidir sobre la señal. Es necesaria para la realización de este diseño debido a que el bloque IP acelerador no existía y el desarrollado para la verificación de este proyecto no realiza el procesado de los paquetes de datos. Por ello, se forzará la respuesta deseada a través del VIO, decidiendo si la respuesta del acelerador será "0x00000001" o "0x00000000". Esto se debe a la configuración realizada en la UD, en referencia a la respuesta esperada del acelerador, tal y como se explicó en el capítulo 4. En la Figura 66 se puede ver el aspecto del bloque, que dispone de una señal de reloj. Existen ocho puertos de salida, uno por cada bloque IP acelerador que se encuentra en el diseño.

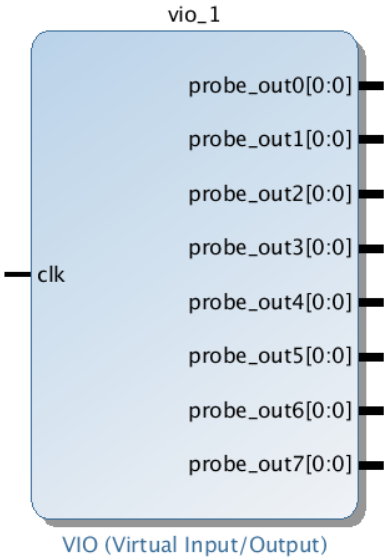

Figura 66: Bloque IP del VIO

- *ILA (Integrated Logic Analyzer)*. El analizador lógico integrado o ILA es necesario para observar el flujo de datos del diseño. La configuración del bloque es la que viene por defecto; solamente se debe cambiar el tipo de protocolo AXI utilizado. En nuestro caso, solo se utiliza AXI4-Stream y AXI4-Lite. En la Figura 67 se puede ver un ejemplo del bloque IP utilizado.

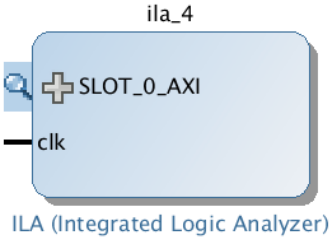

Figura 67: Bloque IP del ILA

5.1.2. Diagrama de bloques

Tras haber explicado todos los bloques involucrados en el diseño del sistema, se presenta la realización del diagrama de bloques. A continuación, se explicarán los pasos seguidos hasta su obtención, para la posterior implementación del sistema en la plataforma.

1. Es necesario usar la herramienta Vivado Design Suite. Se genera un nuevo proyecto RTL con el nombre "DispatchUnit". No se añade ningún tipo de archivo fuente o directorio, ni ningún archivo IP configurable. Tampoco es necesario añadir excepciones, pero si se debe seleccionar el dispositivo sobre el que se desea realizar el sistema, siendo en este caso la placa de prototipado ZedBoard. En la Figura 68 se pueden observar las características mencionadas anteriormente.

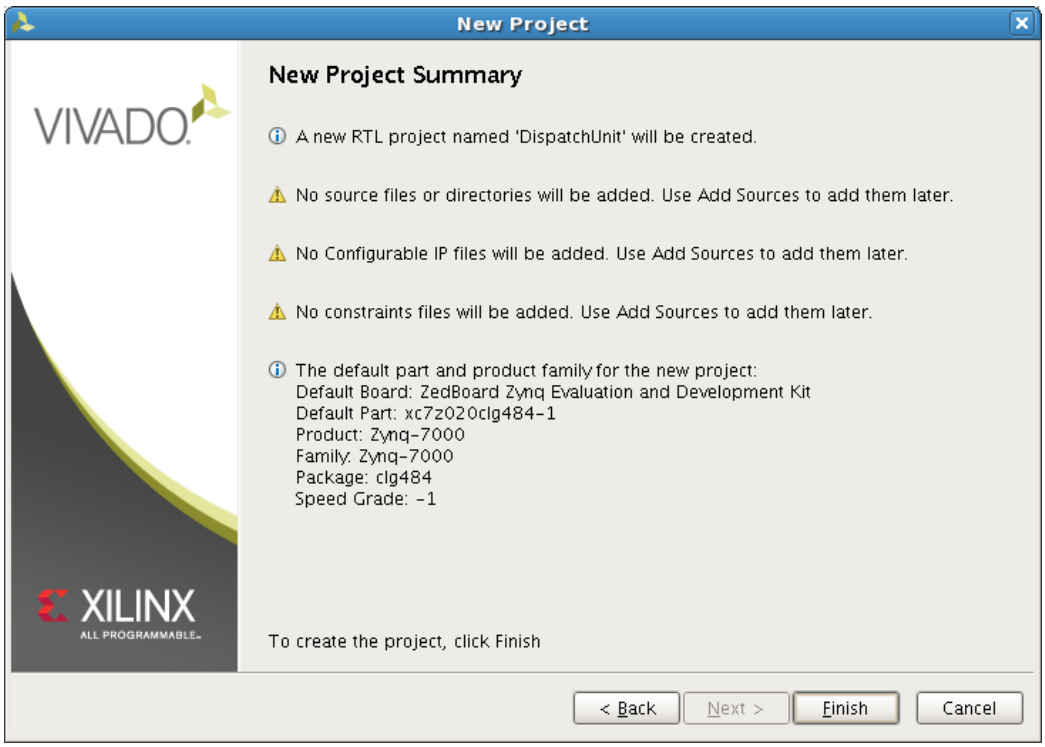

Figura 68: Propiedades del proyecto RTL "DispatchUnit"

2. Tras ello, se crea un nuevo diseño de bloque accediendo a "Create Block Design", al que se le ha dado el nombre de "design_1" y será almacenado en el directorio del proyecto, dentro de la carpeta "Design Sources".
3. Se deben importar los bloques IP creados a la librería, incluyendo el bloque de la UD y el bloque acelerador, para su utilización. Para ello se accede a las características IP del proyecto y se selecciona "IP Packager". Se debe añadir cada bloque IP por separado, yendo al directorio donde se guardó tras su creación, tal y como se observa en la Figura 69.

4. Una vez todos los bloques IP necesarios se encuentren en la librería se puede comenzar el diagrama de bloques. Primero se debe añadir el bloque del PS de Zynq. Para ello, sobre el diseño de bloque creado se accede a la ventana "Diagram" y se añade el bloque IP seleccionado "Add IP" y buscando el nombre "ZYNQ7 Processing System". Sin embargo, se deben seleccionar las propiedades del bloque explicadas anteriormente. Para ello, se accede a "Re-customize IP" y se selecciona en "Clock Configuration" la frecuencia de 150 MHz para PCLK_0 y en "PS-PL Configuration" la interfaz "S AXI HPO Interface."
5. El siguiente paso, es añadir el bloque IP correspondiente al DMA, para ello se vuelve a añadir un nuevo bloque IP introduciendo "AXI Direct Memory Access". Para introducir los cambios necesarios solo se debe acceder a "Re-customize IP" y completar los campos como se mostró en la Figura 60. Para todos aquellos bloques cuya configuración difiera de la que viene por defecto, tal y como se explicó en el capítulo anterior, se deben realizar los mismos pasos.

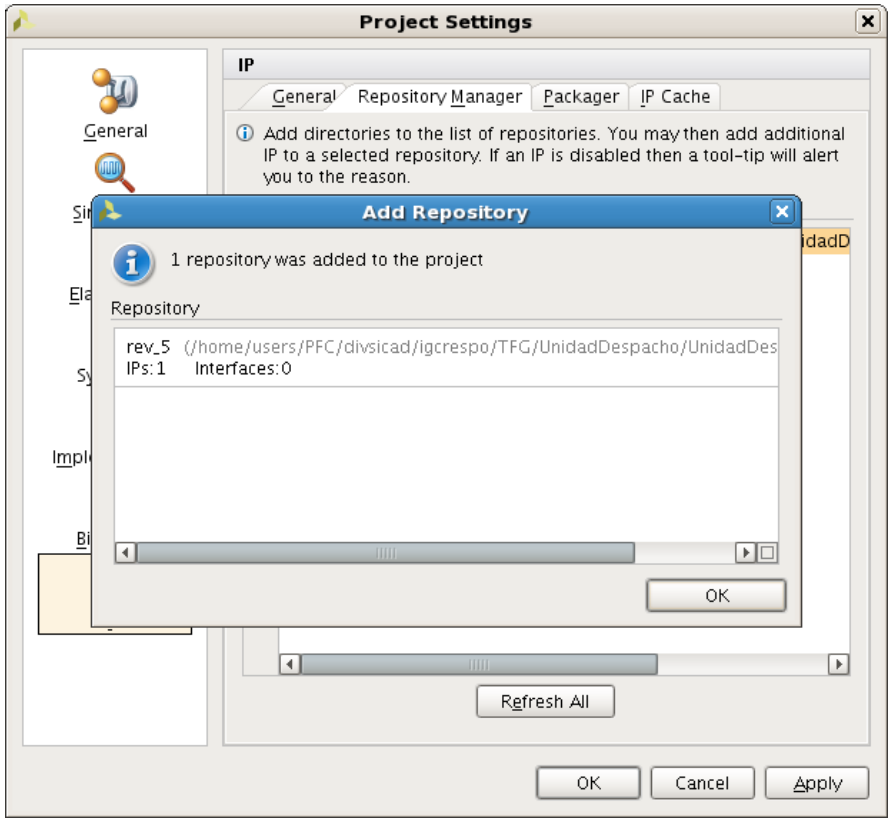

Figura 69: Insertar nuevo bloque IP en Vivado Design Suite

6. Una vez ambos bloques se encuentran en el entorno de trabajo se procede a la interconexión de ambos. Para ello se accede a "Run Connection Automation" y como se observa en la Figura 70 se selecciona la señal de reloj utilizada para ambos bloques, que será "FCLK_CLK0".

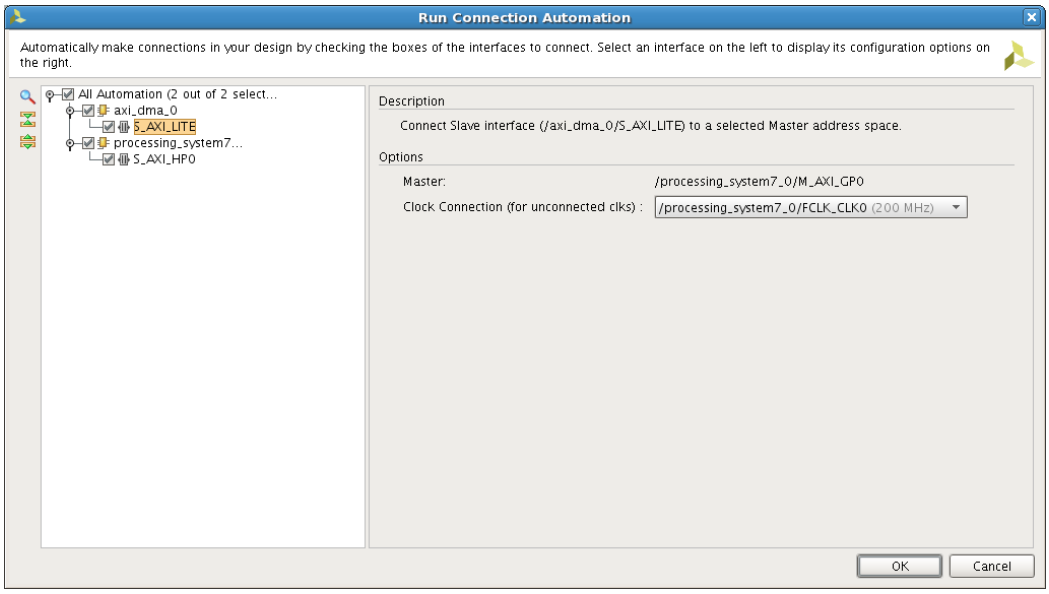

Figura 70: Conexionado Automático de los bloques Zynq PS y DMA

7. Nada más terminar de realizarse el conexionado aparecerán tres bloques nuevos en nuestro diseño. Estos son "axi_mem_intercon", "processing_sytem7_0_axi_periph" y "rst_procesing_system7_0_200M".
8. Ahora ya es posible añadir el bloque IP de la UD diseñado. Para ello se añade un nuevo bloque IP al buscar "Unidad_Despacho_v1_0". Se procede a realizar la conexión automática "Run Connection Automation" para que las señales de reloj y *reset* se conecten de forma automática y el bus "S_AXI" del bloque UD se conecte al "M01_AXI" de "processing_system7_0_axi_periph". También es necesario conectar la entrada de datos "data_in_ethernet" de la UD al bloque DMA del diseño a través de "M_AXIS_MM2S", y la salida de datos de la UD "data_out_ethernet" a la señal "S_AXIS_S2MM". Gracias a esta conexión se podrá realizar la transmisión de datos desde el DMA y la comprobación, de que los datos devueltos al mismo coinciden con los enviados.
9. También se deben añadir las dos FIFOs necesarias en el sistema. Al introducir "FIFO Generator" al añadir un nuevo IP se obtiene el bloque deseado. Por tanto, se procede a configurar ambas memorias y a realizar las conexiones necesarias. La señal de reloj se conecta para ambas FIFOs a "FCLK_CLK0", al igual que en el punto número 8, y tanto el bus de lectura como el de escritura se conectan a la UD en sus correspondientes buses.
10. A continuación, se incluyen ocho bloques aceleradores al buscar "AccelIP_rtl_v1_0" al añadir un nuevo bloque IP. Para que el diagrama de bloques no se vea saturado, se decide crear una jerarquía donde se añaden todos los bloques aceleradores y queda tal y como se muestra en la Figura 71. Las conexiones de este bloque se realizan principalmente con la UD, a excepción de las señales de reloj y *reset*, que se conectan al igual que en los bloques

anteriores. Las señales "asnwer" se conectarán al bloque VIO que se detallará a continuación. Todos las señales "data_in_accel" se conectan de forma secuencial a las "data_out_accel" del bloque UD y de forma inversa con las señales "data_out_accel".

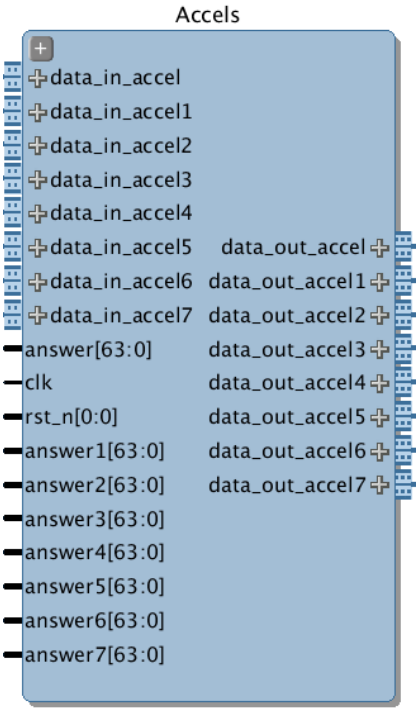

Figura 71: Jerarquía de los bloques IP aceleradores

11. Por último, se deben añadir los bloques de depurado antes de poder terminar el diseño del diagrama de bloques. En primer lugar, se añade el bloque VIO, al introducir "VIO" al añadir un nuevo bloque IP, y se realiza su configuración. También se debe conectar la señal de reloj a "FCLK_CLK0" y las señales de salida "probe_out" de forma secuencial a las señales "answer" de los bloques IP aceleradores. En segundo lugar, son necesarios cicno bloques ILA, que se añaden al buscar la palabra "ILA" y se conectan todas las señales de reloj con la señal de reloj principal. El primer ILA se conecta al puerto de configuración de la UD que corresponde al bus "S_AXI". El segundo y el tercer ILA se conectan a la entrada y salida del ethernet de la UD, respectivamente. Y por último, el cuarto ILA se conecta a la salida del primer bloque acelerador y el quinto ILA a la salida del octavo bloque acelerador.
12. Una vez finalizado el diagrama de bloques se puede observar en la Figura 72 el resultado final del diseño.

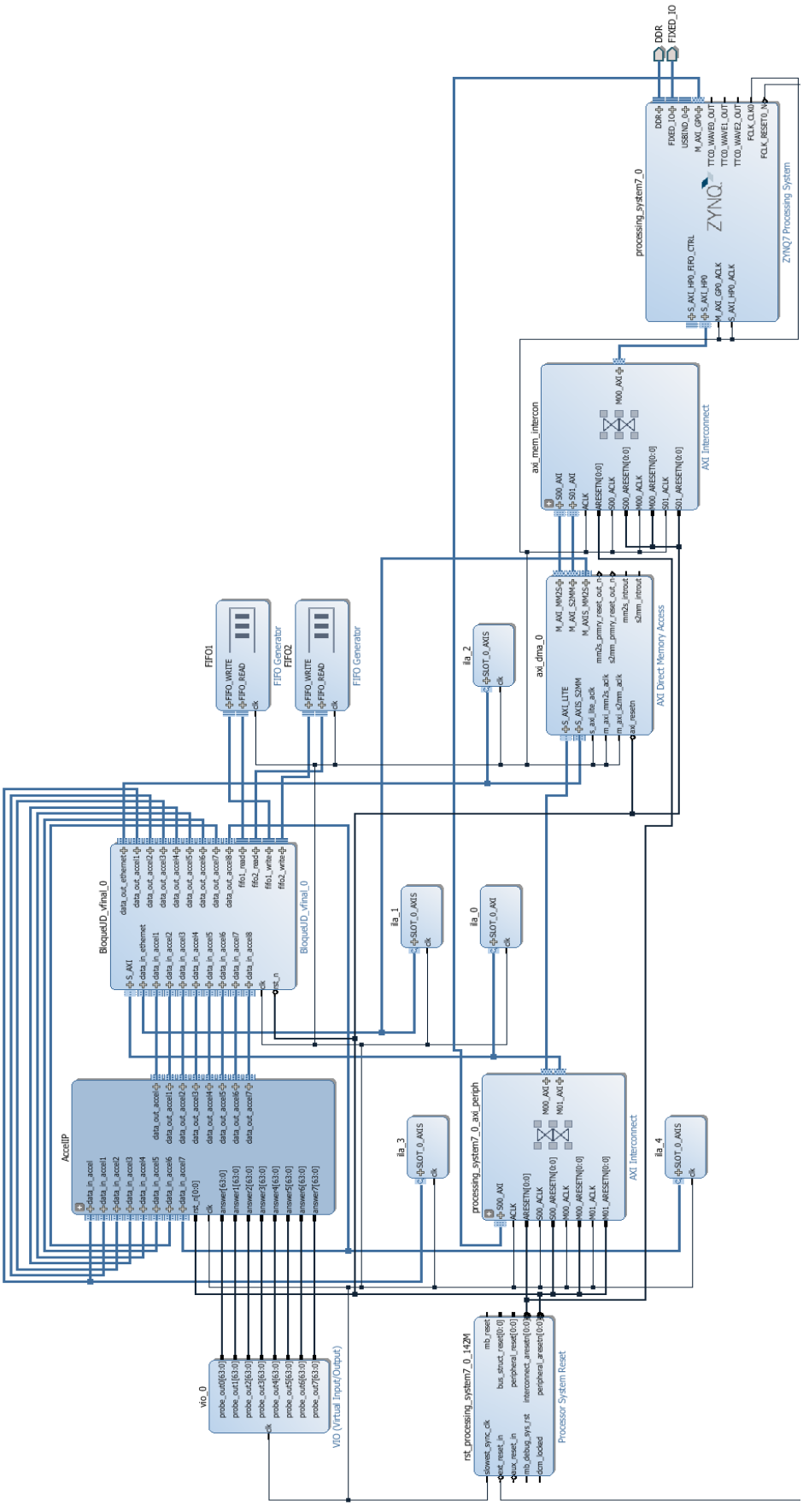

Figura 72: Diagrama de la plataforma final incluyendo ILAs

5.2. Implementación en la plataforma

Para la implementación en la plataforma del sistema realizado será necesaria la obtención del *bitstream* para la programación del dispositivo y el desarrollo de un proyecto en el SDK para poder realizar la validación en la plataforma.

5.2.1. Obtención del *bitstream*

A continuación, se procede a la síntesis, la implementación y la exportación del *hardware*, obteniendo como resultado el *bitstream* necesario para la programación del dispositivo. Tras todos estos pasos, la herramienta Vivado Design Suite seguirá siendo necesaria para poder realizar el depurado del sistema mediante los ILAs y el VIO incluidos en el diseño.

Antes que nada debe validarse el diseño realizado. En la ventana "Diagram" se debe seleccionar "Validate Design" y solo cuando esté se encuentre validado correctamente se puede proseguir con los siguientes pasos. Para ello debemos acceder a la ventana "Sources" incluida dentro de "Block Design" y seleccionar el diseño generado. El siguiente paso es generar los productos de salida del diagrama realizado, para ello seleccionamos "Generate Output Products" y tal y como se muestra en la Figura 73, realizar una síntesis global y ejecutarlo en el servidor local.

Una vez finalizado, también es necesario crear el *wrapper* HDL del diseño al seleccionar "Create HDL Wrapper". Se debe seleccionar la opción que permite a Vivado manejar el *wrapper* de manera independiente y actualizarlo. Tras ello, el diseño ya está listo para realizar la síntesis, la implementación y la exportación del *hardware*.

Para ello, únicamente se debe seguir la ruta "Flow Navigator → Program and Debug → Generate Bitstream". El sistema mostrará un aviso de que no se ha realizado la síntesis ni la implementación del sistema y que para poder proseguir deben realizarse estos pasos. Por ello, se selecciona realizarlos todos de forma consecutiva hasta la obtención del *bitstream*.

Tras finalizar, se escoge abrir el diseño implementado y se observa el *layout* de la plataforma, tal como se muestra en la Figura 74. El *layout* muestra los recursos utilizados, el cuál es bastante extenso debido a todos los recursos que son necesarios implementar para validar el correcto funcionamiento de la UD. En la figura se observan remarcados los recursos utilizados por el bloque UD y las FIFOs necesarias para su implementación.

En la Figura 75 se presentan los resultados del análisis temporal del diseño, obteniendo un *slack* positivo, al utilizar una frecuencia de funcionamiento de 150 MHz para el bloque PL.

Un PLL es el encargado de generar la frecuencia en valores discretos. Por este motivo la frecuencia real del sistema será aproximada, en este caso de 142,857 MHz. El valor del *slack* obtenido es de 0,497 ns y permite aún un pequeño aumento de la frecuencia utilizada, pero para la validación del bloque no es necesario realizar cambios.

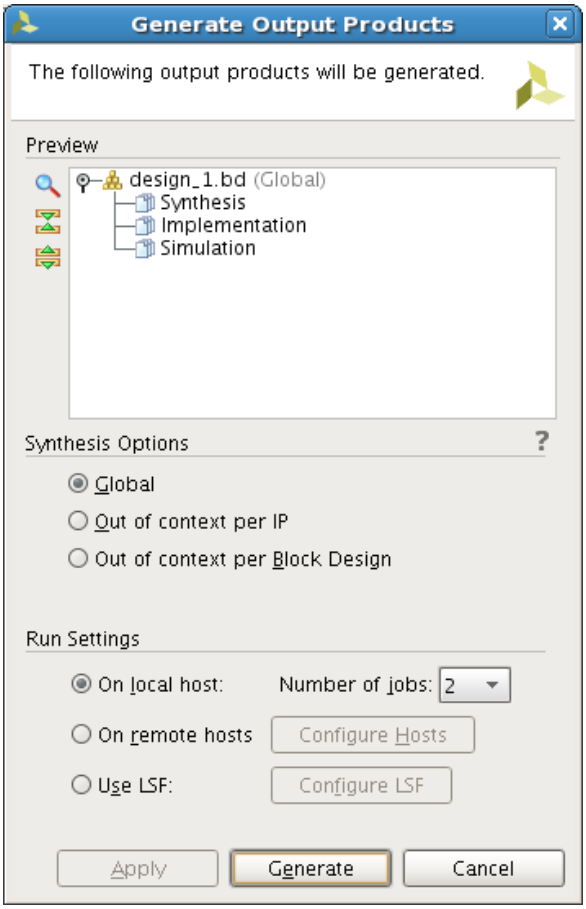

Figura 73: Generación de los productos de salida en Vivado Design Suite

También se puede observar en la Figura 76 un gráfico que indica los porcentajes de recursos utilizados en la implementación del sistema, tras realizar la implementación. Las LUTs utilizadas ascienden a 10394, siendo un 20% de las disponibles en el dispositivo XC7Z020. También se hace uso de 1182 LUTRAMs (7%), 15588 FFs (15%), 18,5 BRAMs (13%) y 3 BUFGs (9%).

Sin embargo, los recursos utilizados por la UD y las memorias FIFO, que incluye para su correcto funcionamiento, son mucho menores. En la Tabla 5 se observa una comparativa de los recursos totales del sistema implementado y los recursos de la UD, que incluyen las memorias FIFO.

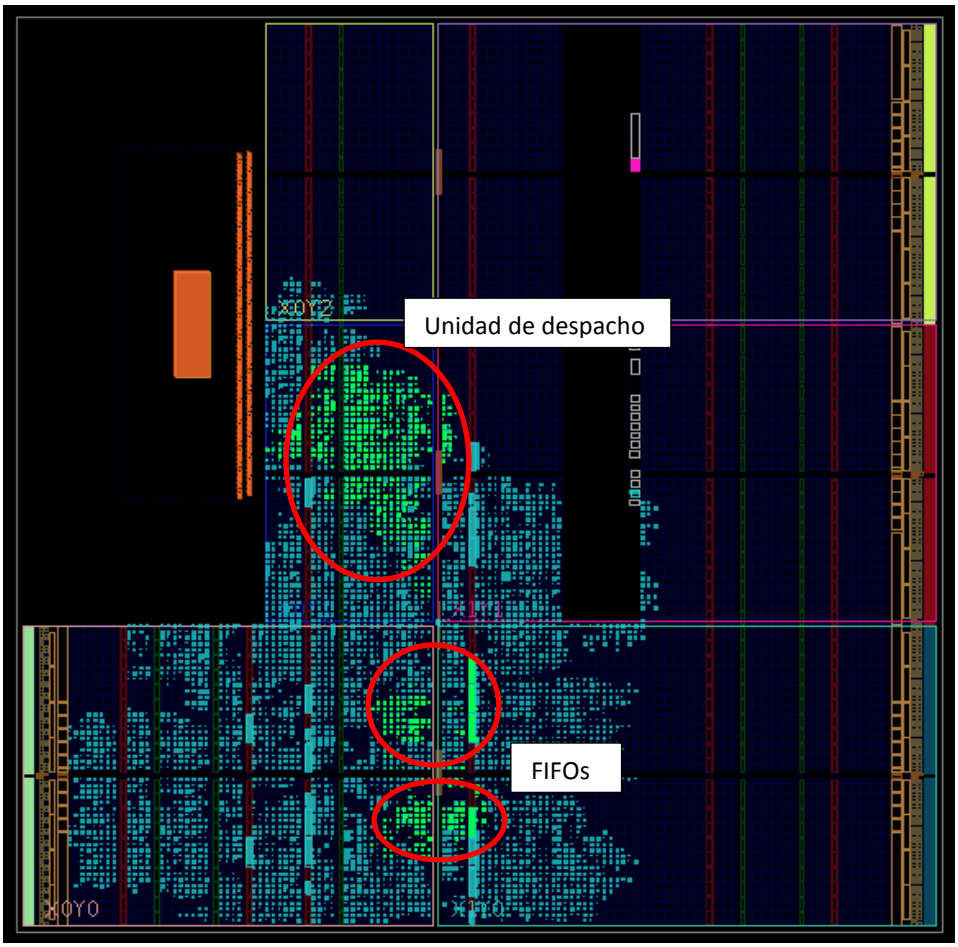

Figura 74: Layout de la plataforma

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.497 ns	Worst Hold Slack (WHS):	0.023 ns	Worst Pulse Width Slack (WPWS):	2.250 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	34020	Total Number of Endpoints:	34020	Total Number of Endpoints:	17649

All user specified timing constraints are met.

Figura 75: Resultados de temporización del diseño

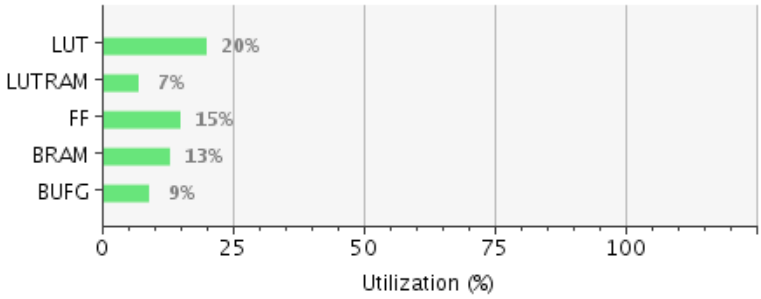

Figura 76: Gráfico de utilización de recursos tras la implementación

Tabla 5: Utilización de Recursos del Sistema y de la UD

	Sistema completo		UD + FIFOs	
LUTs	10.394	19,54 %	1.557	2,93 %
LUTRAMs	1.182	6,79 %	0	0,00 %
FF	15.588	14,65 %	1.854	1,74 %
BRAM	18,5	13,21 %	4	2,86 %
BUFG	3	9,38 %	0	0,00 %

Por último, cabe resaltar los resultados obtenidos al realizar un análisis de la potencia consumida por el sistema implementado. En la Figura 77 se muestra el consumo de potencia de la plataforma, siendo igual a 1,847 W. Sin embargo, el consumo de la UD no supera los 16 mW como se observa en la Tabla 6.

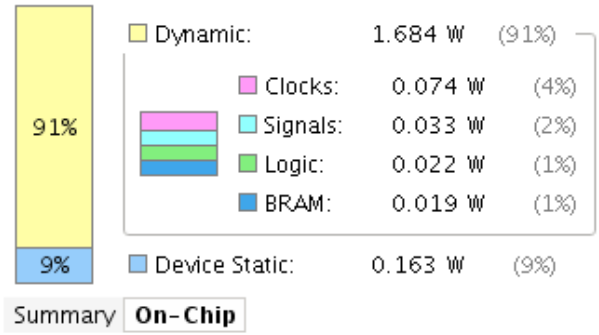

Figura 77: Consumo de potencia de la plataforma

Tabla 6: Consumo de potencia de la UD

Utilización (mW)	Nombre	Reloj (mW)	Señales (mW)	Datos (mW)	Lógica (mW)	BRAM (mW)
13	BloqueUD_vfinal	7	2	2	2	<0.1
2	FIFO1	1	<0.1	<0.1	<0.1	1
1	FIFO2	<0. 1	<0.1	<0.1	<0.1	<0.1

5.2.2. Herramienta SDK

Por último, es necesario utilizar la herramienta SDK para poder realizar la programación del *software* empotrado. Se comienza con la creación del proyecto, su programación en la FPGA deseada y la validación y testeado del mismo. Para ello, en la herramienta Vivado Design Suite se debe exportar el *hardware* obtenido en el desarrollo del diseño de la plataforma. Simplemente se accede

a "File → Export → Export Hardware" y se incluye el *bitstream* generado con anterioridad. Finalmente, se selecciona "File → Launch SDK" y se abre automáticamente la herramienta deseada.

Al abrir la herramienta SDK directamente desde Vivado Design Suite, el directorio utilizado en el entorno de trabajo es el mismo que el que contiene el *hardware* exportado, incluyendo el *bitstream*. A continuación, se deben crear dos proyectos en SDK, uno que incluya el paquete de soporte del dispositivo y otro con la aplicación a implementar en la plataforma. Inicialmente se selecciona crear un nuevo proyecto BSP con el nombre "BSP", que incluye todo el soporte necesario para la placa de desarrollo.

Las configuraciones se realizan por defecto, pero se debe comprobar que en el *hardware* seleccionado, la plataforma utilizada sea la desarrollada en el apartado anterior, que tiene por defecto el nombre de "design_1_wrapper_hw_platform_0". También, se debe asegurar que la CPU utilizada coincida con la de la plataforma, que es "ps7_cortexa9_0".

A continuación, ya es posible generar el proyecto que incluirá la aplicación para verificar el correcto funcionamiento del sistema. Para ello se realiza un nuevo proyecto "Aplication Project" con el nombre "Verification" y se tienen en cuenta las mismas restricciones para la configuración que con el proyecto anterior, incluyendo el BSP generado. Posteriormente, es necesario determinar el tipo de aplicación que se quiere crear en "Templates" y para la verificación de este proyecto se ha seleccionado un proyecto de aplicación vacío. En la Figura 78 se observa el entorno de trabajo de la herramienta SDK de Xilinx.

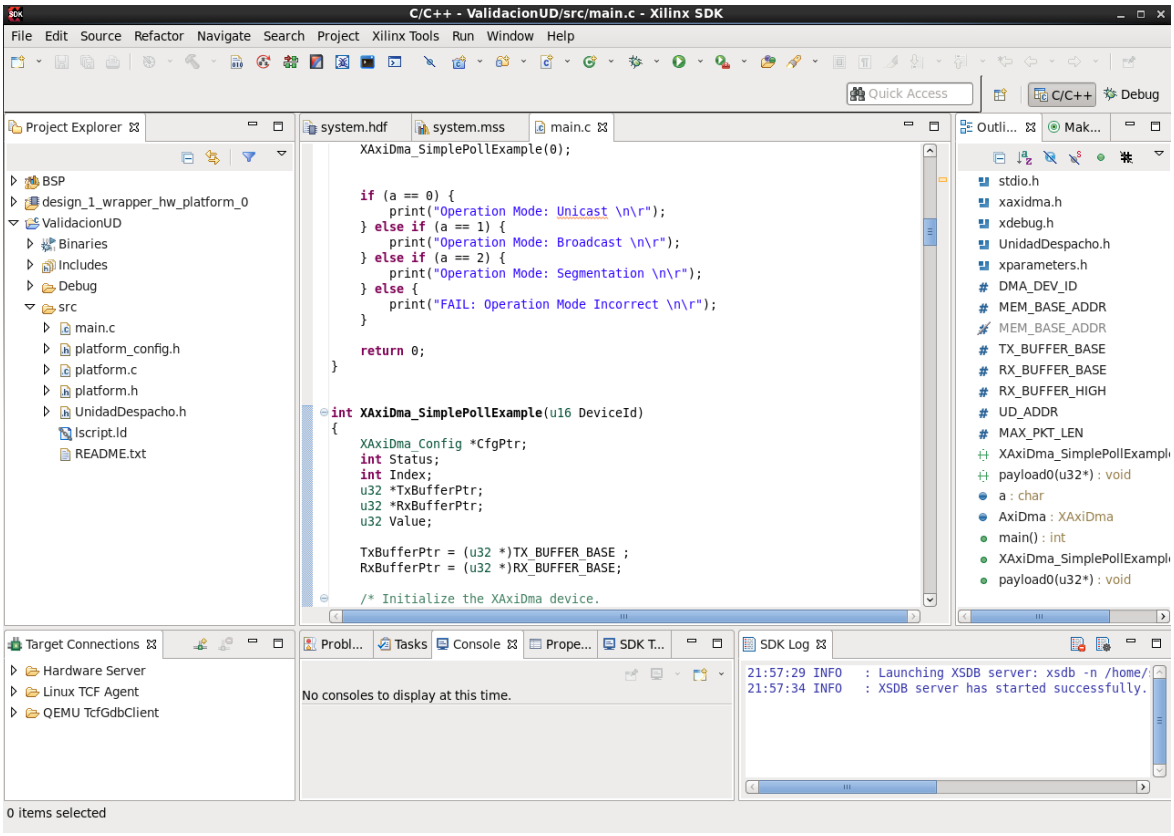

Figura 78: Entorno de trabajo de Xilinx SDK

Capítulo 6. Validación

Tras obtener la implementación del sistema ya es posible realizar la validación del mismo. Se procede a la validación del sistema sobre la plataforma ZedBoard gracias al *bitstream* generado y la herramienta SDK que permite su programación. Asimismo, se utiliza la herramienta Vivado Design Suite, que gracias al configurador de *hardware* que incorpora permite realizar comprobaciones de los flujos de datos del sistema a través de los ILAs integrados en el diseño. También se puede seleccionar si la UD debe enviar o descartar los paquetes, en función de la respuesta del acelerador que forzamos a través del VIO. Por último, el uso de la utilidad Minicom es requerida para poder observar a través del terminal los mensajes. Estos muestran, a medida que se ejecuta el código de validación, si su comportamiento es correcto o si algún error ha sido encontrado.

6.1 Procedimiento de validación

En el capítulo 5 se explicaron los diferentes proyectos necesarios para la comprobación del *hardware*, incluyendo uno de aplicación que inicialmente se encuentra vacío. En este proyecto se incluye el funcionamiento del DMA y se realiza la transmisión de datos con la UD para poder validar su correcto funcionamiento. Por ello, se ha creado un archivo "main.c" que incorpora el comportamiento general del DMA, el envío de datos, la comprobación de su recepción y las lecturas y escrituras del puerto de configuración, tal y como se observa en la Figura 79. Antes que nada se debe conocer la dirección base asignada de forma automática a la UD en Vivado Design Suite, a la que posteriormente se llamará UD_ADDR en el archivo "main.c". Para obtener el valor de la dirección base se debe acceder a "xparameters.h", donde se encuentra:

```
#define XPAR_BLOQUEUD_VFINAL_0_BASEADDR 0x43C00000
```

En primer lugar, se genera una función con el nombre "XAxiDma_SimplePollExample" que realiza la inicialización del DMA con configuración simple y es manejado a través de *polling*. También se ejecuta el envío de un paquete almacenado en memoria, del cual se sabe su dirección y su valor. Se utiliza un paquete de datos conocido (y no uno aleatorio o de la red) para poder comprobar que se realiza la comunicación de forma correcta. El paquete es enviado a través del DMA a la UD y el DMA también espera la respuesta de la UD. Comprueba que cada palabra del paquete

se corresponde con la esperada y, en caso contrario, muestra un mensaje ERROR que se puede ver en la pantalla de la utilidad Minicom.

En segundo lugar, se comprueba el correcto funcionamiento del puerto de configuración, tanto al realizar lecturas como escrituras. Gracias al puerto de configuración es posible validar todos los modos de operación implementados en la UD. A tal fin, se ha creado un archivo "UnidadDespacho.h", el cual incluye una serie de funciones que permiten la escritura o lectura en cada una de las direcciones del AXI4-Lite habilitadas en la UD. Para las funciones de lectura solo es necesario incluir la dirección base para poder obtener los datos deseados. Se han de tener en cuenta las direcciones creadas en el archivo de cabecera "registers_param.h" para añadirlos a la dirección base de la UD y acceder a los datos de forma correcta. Para las escrituras del sistema, sin embargo, no solo es necesaria la dirección base, sino también los datos del nuevo modo de operación que se quiera implementar. La función será la encargada de realizar la escritura en las posiciones correctas para coincidir con los datos esperados por la UD.

```
// Include & define files
…
int XAxiDma_SimplePollExample(u16 DeviceId);
void payload0(u32* payload);
char a;
/************************** Variable Definitions *****************************/
/* Device instance definitions */
XAxiDma AxiDma;

int main() {
   printf("Init Configuration \n\r");
   // Change Operation Mode or make a Read from AXI4-Lite
   …
   XAxiDma_SimplePollExample(0);

   if (a == 0) {
         print("Operation Mode: Unicast \n\r");
   } else if (a == 1) {
         print("Operation Mode: Broadcast \n\r");
   } else if (a == 2) {
         print("Operation Mode: Segmentation \n\r");
         } else {
         print("FAIL: Operation Mode Incorrect \n\r");
   }
   return 0;
}

int XAxiDma_SimplePollExample(u16 DeviceId) {
   // Variables
   …
   TxBufferPtr = (u32 *)TX_BUFFER_BASE ;
   RxBufferPtr = (u32 *)RX_BUFFER_BASE;
   /* Initialize the XAxiDma device. */
   CfgPtr = XAxiDma_LookupConfig(DeviceId);
```

```
    if (!CfgPtr) {
        xil_printf("No config found for %d\r\n", DeviceId);
        return XST_FAILURE;
    }

    Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
    if (Status != XST_SUCCESS) {
        xil_printf("Initialization failed %d\r\n", Status);
        return XST_FAILURE;
    }

    if (XAxiDma_HasSg(&AxiDma)){
        xil_printf("Device configured as SG mode \r\n");
        return XST_FAILURE;
    }

    /* Disable interrupts, we use polling mode */
    XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
    u32 payload[400];
    payload0(&payload);

    for (Index = 0; Index < 375; Index ++) {
        TxBufferPtr[Index] = payload[Index];
    }

    /* Flush the SrcBuffer before the DMA transfer, in case the Data Cache is enabled */
    Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN*4);
    Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) RxBufferPtr,
    MAX_PKT_LEN*4, XAXIDMA_DEVICE_TO_DMA);

    if  (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) TxBufferPtr,
                        MAX_PKT_LEN*4, XAXIDMA_DMA_TO_DEVICE);
    if  (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    while ((XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) ||
                    (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
        /* Wait */
    }
    u32 *RxPacket;
    RxPacket = (u32 *) RX_BUFFER_BASE;

    /* Invalidate the DestBuffer before receiving the data, in case the Data Cache is enabled */

    Xil_DCacheInvalidateRange((UINTPTR)RxPacket, MAX_PKT_LEN);
    for (Index = 0; Index < MAX_PKT_LEN; Index++) {
        xil_printf("Data[%d]: %x\r\n", Index, (unsigned int)RxPacket[Index]);
        if  (payload[Index] != RxPacket[Index]) xil_printf("ERROR\r\n");
    }
/* Test finishes successfully */
    return XST_SUCCESS;
}
```

Figura 79: Archivo "main.c" con el código principal para la validación de la UD

En último lugar, es necesario configurar las opciones de depuración para la programación de la plataforma y la validación del sistema. Para ello se accede a la ruta "Debug Configurations → Xilinx C/C++ application (GDB) → New Debug". En la Figura 80 se pueden observar los parámetros introducidos en "Target Setup", donde se selecciona como tipo de depurado uno de aplicación independiente con conexión local. Asimismo, se escoge la plataforma *hardware* deseada, siendo "design_1_wrapper_hw_platform_0", que concuerda con el generado en el Vivado Design Suite a través del diagrama de bloques. El archivo *bitstream* será "design_1_wrapper.bit" y el fichero de inicialización "ps7_init.tcl", ambos generados de forma automática por la herramienta Vivado Design Suite. Por otra parte, en la pestaña de aplicación se debe incluir la aplicación creada para validar el sistema, que en este caso coincide con "Verification.elf".

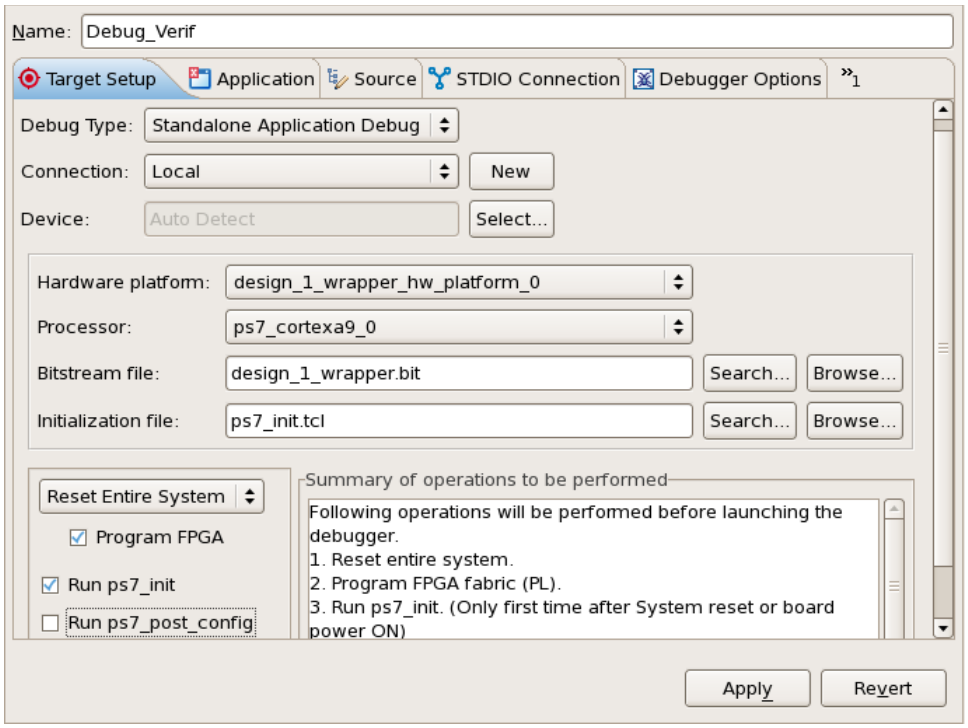

Figura 80: Opciones de depuración en SDK

6.2. Validación del modo de operación *Unicast*

En este momento ya es posible validar el sistema en la placa de desarrollo. Para ello, se ejecuta la depuración creada y se muestra por el terminal del Minicom los resultados obtenidos. La primera validación que se hace al sistema es comprobar la correcta transmisión de datos en el modo de operación por defecto, que es *Unicast* al primer bloque IP acelerador conectado. Se ha realizado el envío de dos paquetes de 375 mensajes de 32 bits de forma consecutiva, sin realizar cambios en el modo de operación. La Figura 81 muestra los datos devueltos por la UD. En esta se puede ver cómo al finalizar el envío de datos se presenta el modo de operación utilizado. Al no aparecer ningún

mensaje de error, se valida el funcionamiento del modo de operación *Unicast* en la ZedBoard. Además, se ha probado a realizar cambios en el VIO para que los mensajes no sean devueltos por la UD, confirmando que el DMA se queda a la espera de los datos tras realizarse la transmisión.

```
Init Configuration
Data[0]: FCAA1474
Data[1]: 61E00007
…
Data[373]: 84EE9322
Data[374]: 7DF48D82
Data[0]: FCAA1474
Data[1]: 61E00007
…
Data[373]: 84EE9322
Data[374]: 7DF48D82
Operation Mode: Unicast
```

Figura 81: Transmisión de paquetes en modo Unicast sobre la placa ZedBoard

En las siguientes páginas se presentan diferentes imágenes donde se refleja el comportamiento de la UD en la ZedBoard. Gracias a los ILAs incorporados al diseño es posible observar el flujo de datos a las entradas y salidas de la UD. Uno de los ILAs se encuentra conectado al bus "data_in_ethernet", el cual permite realizar un control sobre las señales de entrada a la UD enviadas por la unidad de red. En la Figura 82 se pueden apreciar estas señales, donde se observa que el paquete es enviado de forma correcta y las señales actúan según lo esperado.

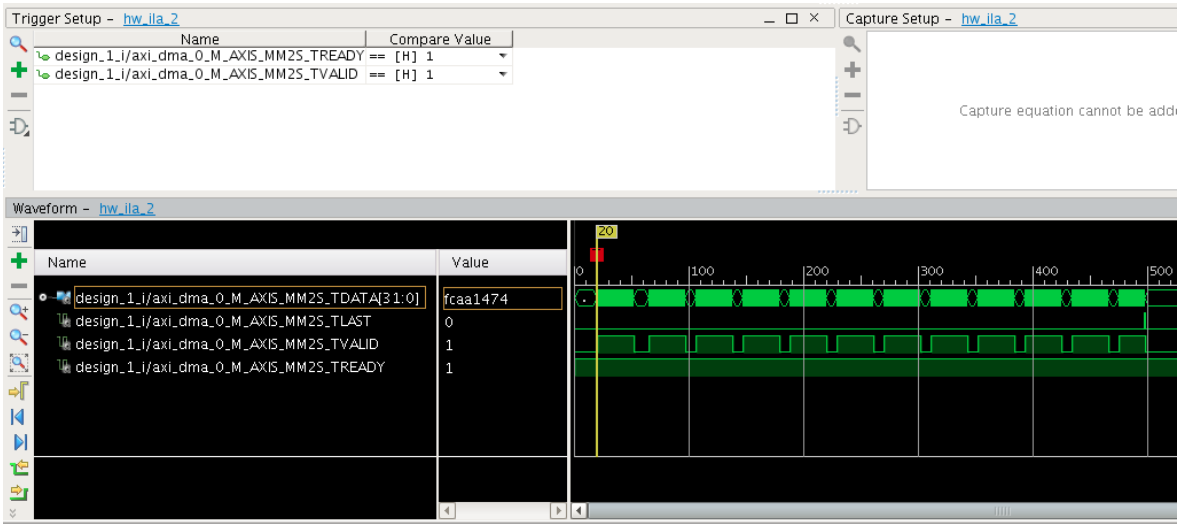

Figura 82: Validación de la UD modo Unicast - Señales "data_in_ethernet"

A continuación, en la Figura 83 se muestran las señales pertenecientes al bus "data_out_accel1" de la UD. Utiliza el modo de operación por defecto, que coincide con el modo *Unicast* al primer bloque IP acelerador conectado a la UD. Gracias a este ILA se puede apreciar

cómo se realiza la transmisión de paquetes del bloque UD al bloque IP acelerador correspondiente, realizando envíos de mensajes de datos de 64 bits.

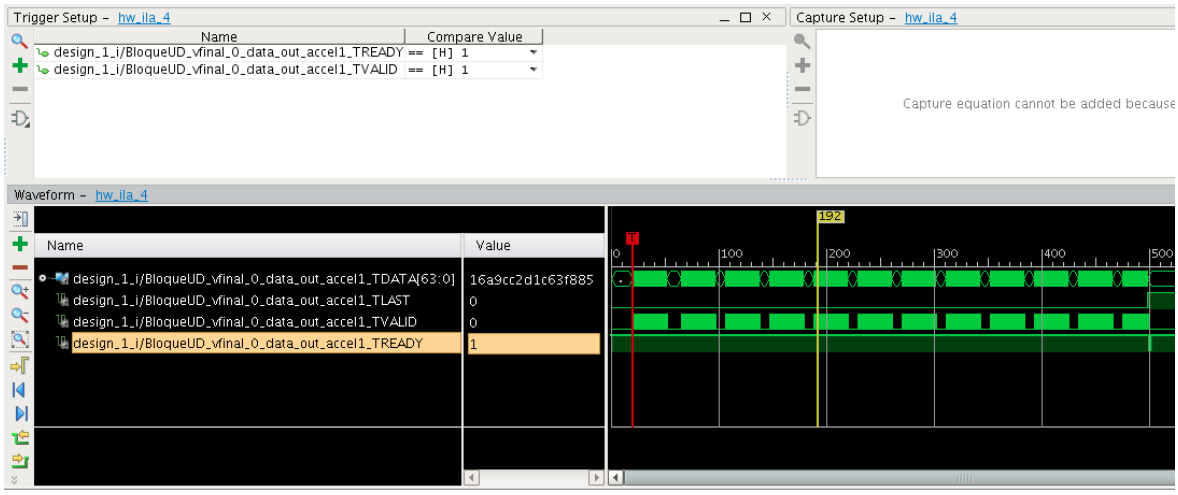

Figura 83: Validación de la UD modo Unicast - Señales "data_out_accel1"

Por último, para comprobar que el modo *Unicast* es válido sobre la placa de desarrollo ZedBoard se ha realizado un análisis de las señales pertenecientes al bus "data_out_ethernet", tal y como se observa en la Figura 84. Se muestra cómo se reciben los paquetes de forma satisfactoria, ejecutando también la comprobación a través del terminal. Este fenómeno también se refleja en la Figura 81.

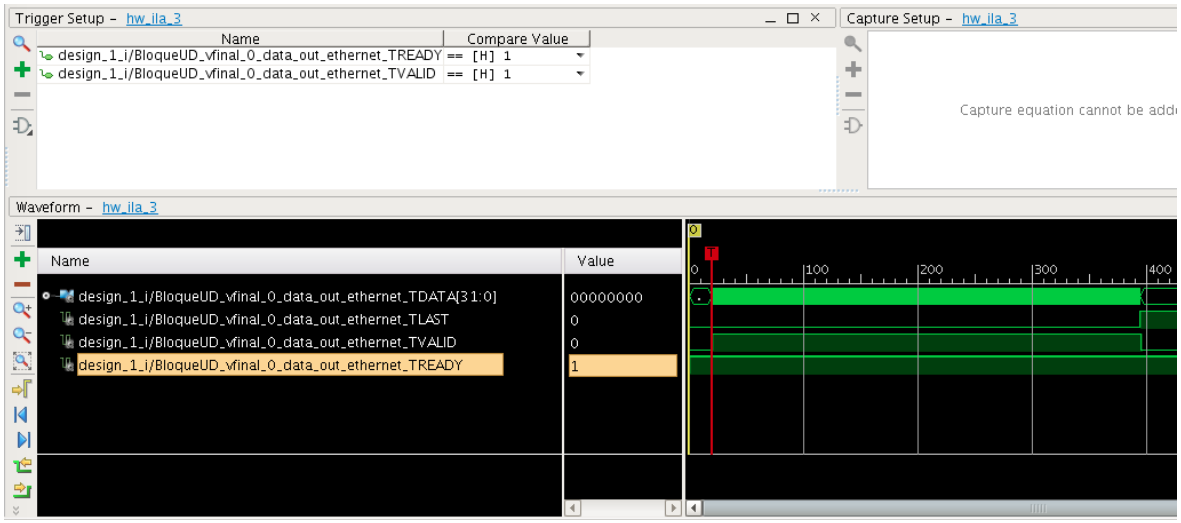

Figura 84: Validación de la UD modo Unicast - Señales "data_out_ethernet"

6.3. Validación del modo de operación *Broadcast*

Tras realizar la validación del modo de operación *Unicast*. Ya es posible comprobar el correcto funcionamiento del resto de modos de operación. A continuación se mostrarán las escrituras

realizadas en el puerto de configuración para poder realizar el cambio de modo de operación a *Broadcast*. En la Figura 85 se muestra la escritura en la primera dirección asignada al puerto de configuración, donde se incluyen los valores de las señales "operation_mode" y "addr". Al ser 0x3FD el valor asignado, indica que el modo de operación será "1", que coincide con *Broadcast*, y se realizará el envío de datos a todos los bloques aceleradores disponibles, pues el valor asignado a "addr" será 0xFF.

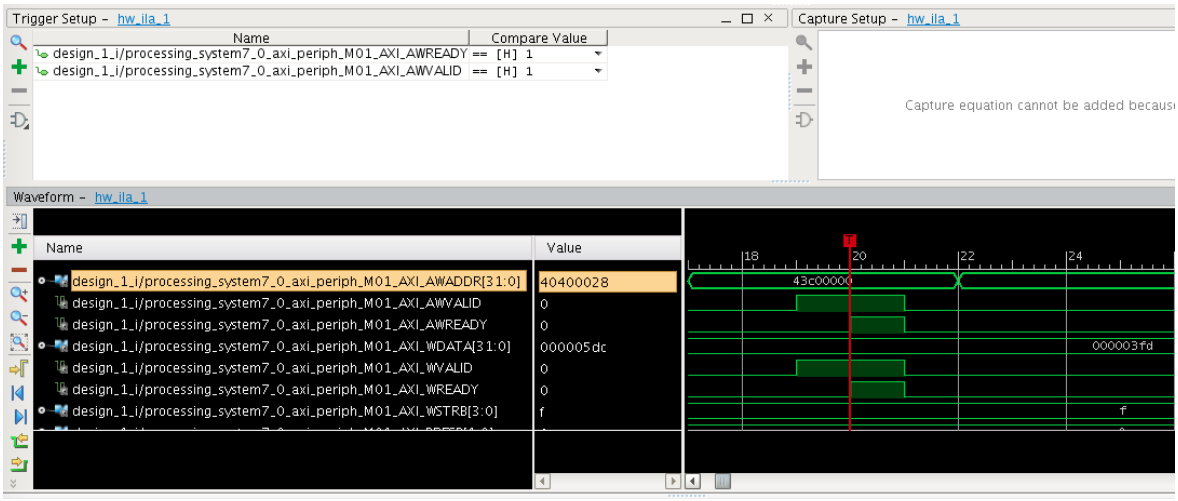

Figura 85: Validación de la UD modo Broadcast - Señales "S_AXI" I

En la Figura 86 y la Figura 87 se muestran dos escrituras en el puerto de configuración de la UD desarrollada. En la Figura 86 se realiza la escritura de las señales "segments" para los cuatro primeros bloques aceleradores, y en la Figura 87 se realiza la escritura para las señales "segments" de los cuatro bloques aceleradores restantes. Al ser el modo de operación escogido *Broadcast* no serán necesario realizar división de los paquetes de datos y todas las señales "segments" valdrán 0.

A continuación, ya es posible iniciar la transferencia de paquetes de datos. Se realiza el envío de dos paquetes de 375 mensajes de 32 bits de forma consecutiva, sin realizar cambios en el modo de operación, al igual que al comprobar el modo de operación *Unicast*. Los mensajes obtenidos a través del terminal gracias a Minicom son exactamente iguales a los mostrados en el apartado anterior, simplemente cambia el modo de operación utilizado. Al no aparecer ningún mensaje de error, se valida el funcionamiento del modo de operación *Broadcast* en la ZedBoard. También se han realizados cambios en el VIO para comprobar que el DMA queda a la espera de los datos tras realizarse la transmisión y que la UD no los envía de vuelta.

En la Figura 88 se muestra un paquete de datos recibido a la entrada del bloque UD, gracias al ILA conectado al bus "data_in_ethernet". El paquete es enviado y recibido acorde al diseño realizado.

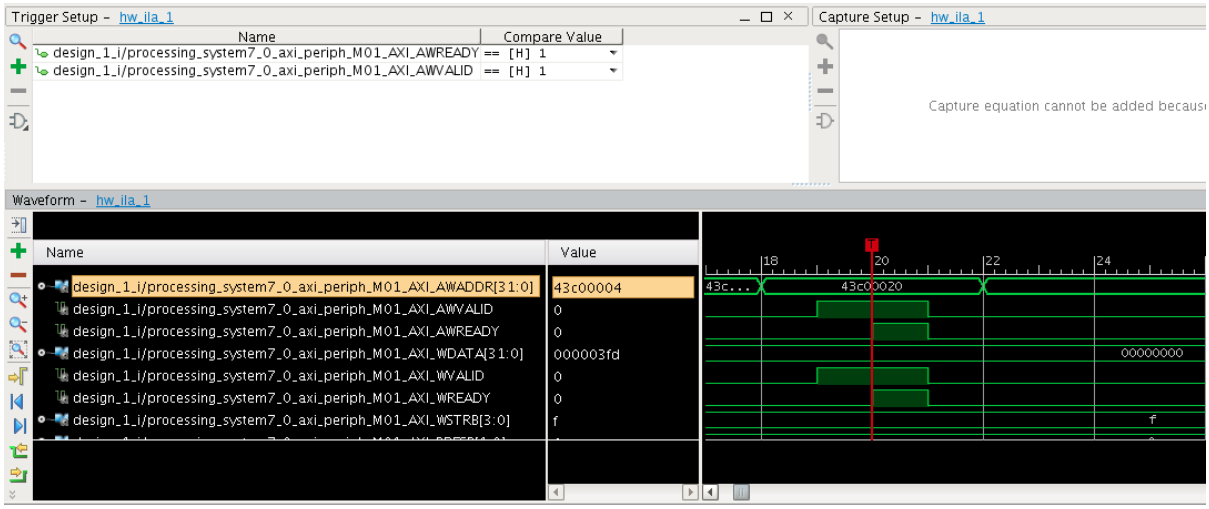

Figura 86: Validación de la UD modo Broadcast - Señales "S_AXI" II

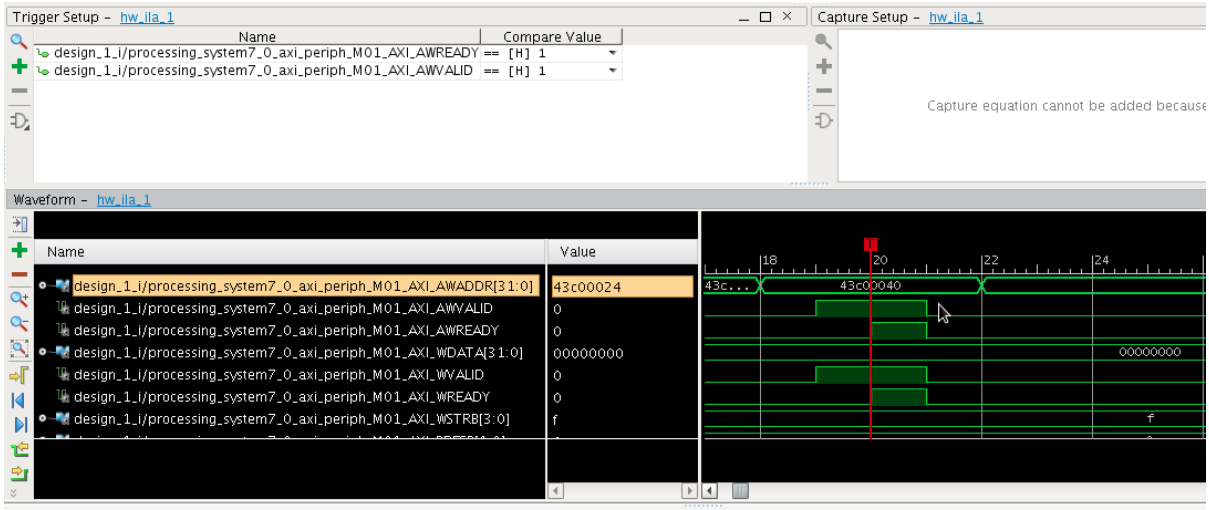

Figura 87: Validación de la UD modo Broadcast - Señales "S_AXI" III

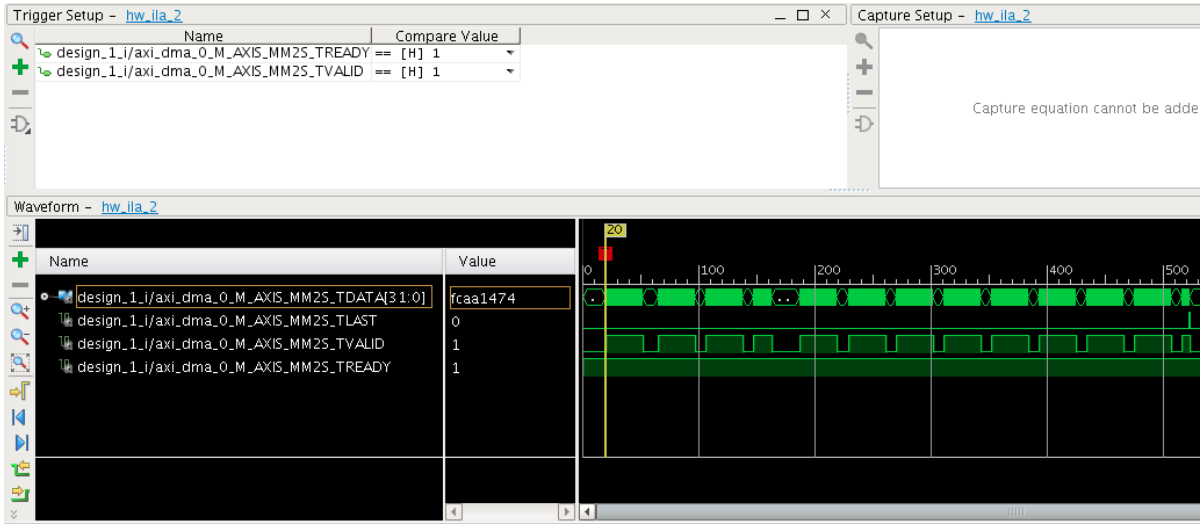

Figura 88: Validación de la UD modo Broadcast - Señales "data_in_ethernet"

El siguiente paso es comprobar que los datos sean enviados a los bloques aceleradores de forma correcta. En la Figura 89 y la Figura 90 se muestran el primer y el octavo bloque aceleradores. Al ser el modo de operación seleccionado *Broadcast* los datos transmitidos a ambos bloques deben ser iguales, y por tanto el comportamiento de las señales de los buses "data_out_accel1" y "data_out_accel8" debe coincidir en todo momento. Gracias a comprobar estos dos buses de datos, es posible asegurar el correcto funcionamiento de la transmisión de datos a los bloques aceleradores desde la UD.

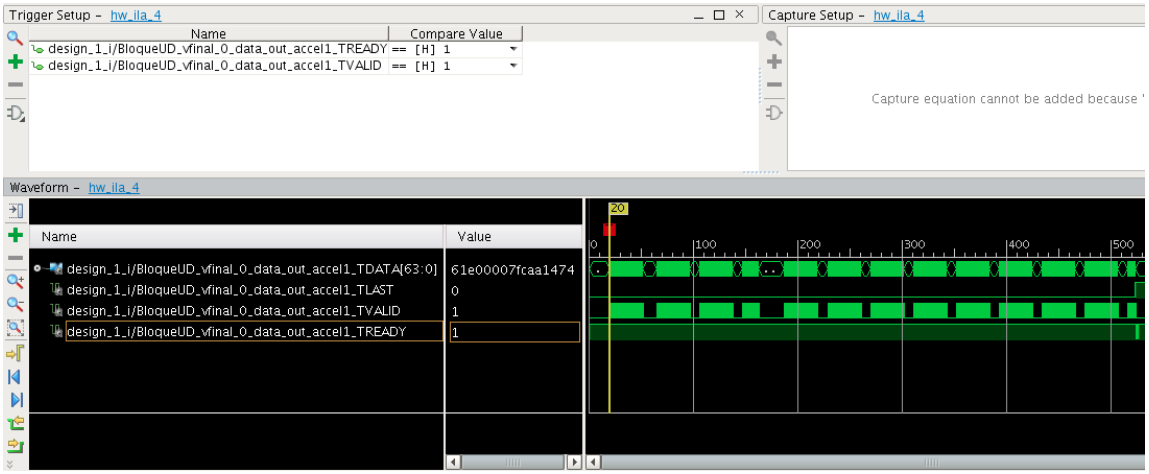

Figura 89: Validación de la UD modo Broadcast - Señales "data_out_accel1"

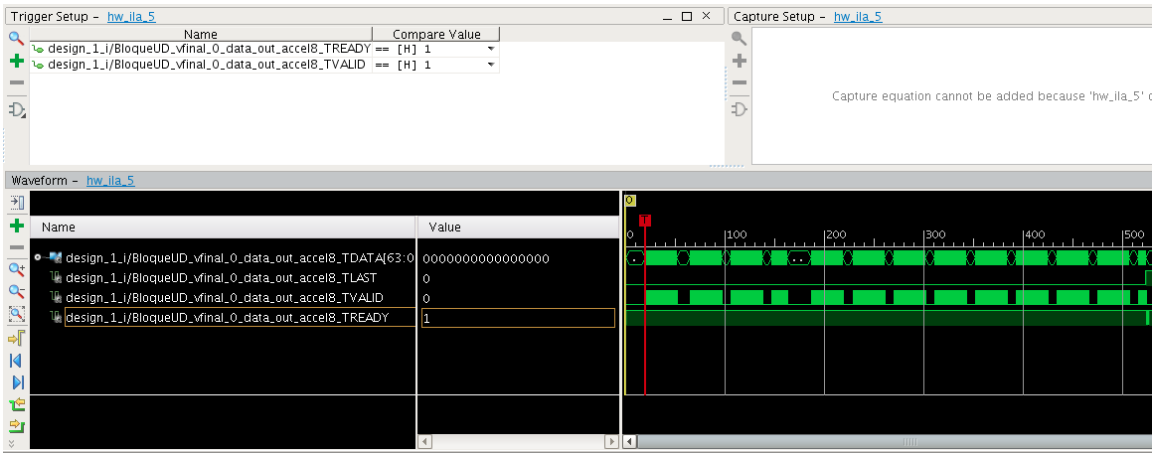

Figura 90: Validación de la UD modo Broadcast - Señales "data_out_accel8"

Finalmente, para asegurar el correcto funcionamiento de la UD en modo *Broadcast* es necesario verificar que los datos son devueltos, sin obtener errores. El bus que muestra las señales de "data_out_ethernet" se encuentra en la Figura 91. Estos serán los datos devueltos a la unidad de red desde la UD, que en este caso será el DMA que realizó el envío de los paquetes. Como consecuencia, se puede asegurar que el modo de operación *Broadcast* en la UD funciona acorde a lo

esperado al verificar a través del terminal de la utilidad Minicom que no se ha producido ningún error en la recepción de datos.

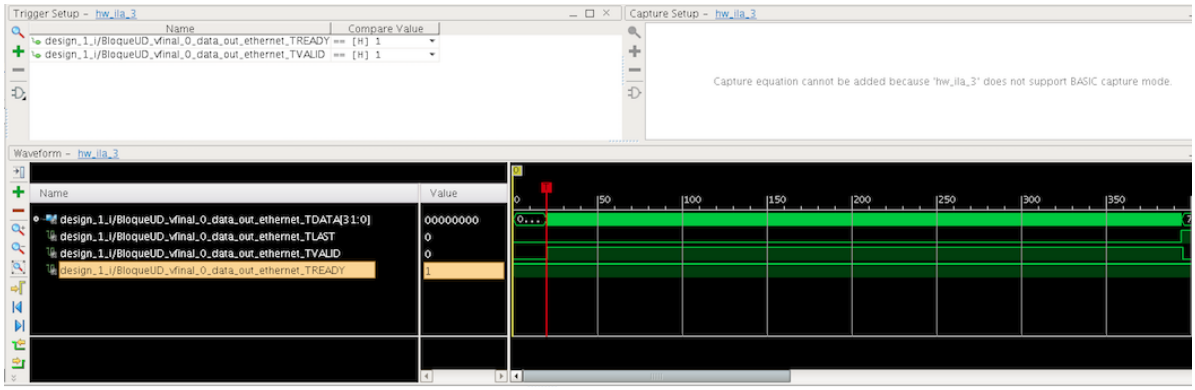

Figura 91: Validación de la UD modo Broadcast - Señales "data_out_ethernet"

6.4. Validación del modo de operación *Segmentation*

El último modo de operación implementado en la UD desarrollado es el *Segmentation*, que se encarga de realizar divisiones de los paquetes de datos recibidos para que los bloques aceleradores no necesiten realizar el procesado de un paquete entero, sino de segmentos. Al validar este modo de operación, se habrán verificado y validado todos los modos de operación implementados en la UD. Para ello, se realiza el mismo procedimiento que con los modos de operación anteriores. Se realiza el envío de dos paquetes de datos de 375 palabras de 32 bits y a través del terminal proporcionado por la utilidad Minicom, se observa que los datos enviados coinciden con los recibidos al no mostrar ningún mensaje de “ERROR”.

Inicialmente, es necesario realizar escrituras en el puerto de configuración para que el nuevo modo de operación se haga efectivo sobre la UD. En la Figura 92 se muestra la escritura de las señales “operation_mode” y “addr”. Al introducir el valor “0x3FE” se asigna un “2” al modo de operación, que coincide con el modo de operación *Segmentation* en la UD, y se activan todas las direcciones de los bloque aceleradores al asignar “0xFF” a la señal “addr”.

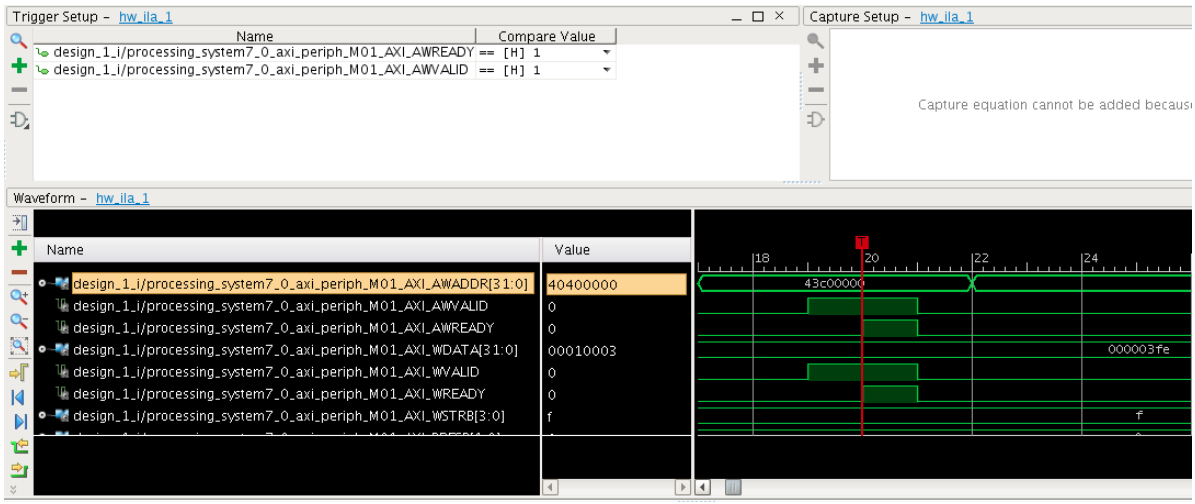

Figura 92: Validación de la UD modo Segmentation - Señales "S_AXI" I

A continuación, en Figura 93 y la Figura 94 se muestran los valores asignados a las señales "segments" de los bloques aceleradores. En la Figura 93 se realiza la escritura para los cuatro primeros bloques aceleradores. El valor asignado es "0x17171717" eso significa que cada uno de los cuatro bloques deberá recibir 23 palabras de 32 bits. Por ello, la señal "segments" para cada uno de estos bloques será 23. En la Figura 94 se realiza la escritura para los cuatro bloques aceleradores restantes conectados a la UD. En este caso el valor asignado es "0x1B171717", lo que significa que los bloques aceleradores 5, 6 y 7 dispondrán de 23 segmentos al igual que los bloques anteriores. Sin embargo, al bloque acelerador 8 se le asignarán 27 segmentos. Esto se debe a que la suma total de segmentos debe ser igual a 188, pues es igual a la mitad del número total de palabras que componen el paquete de datos enviados. Esto se debe a que a los bloques aceleradores se les envían palabras de 64 bits, no de 32 bits.

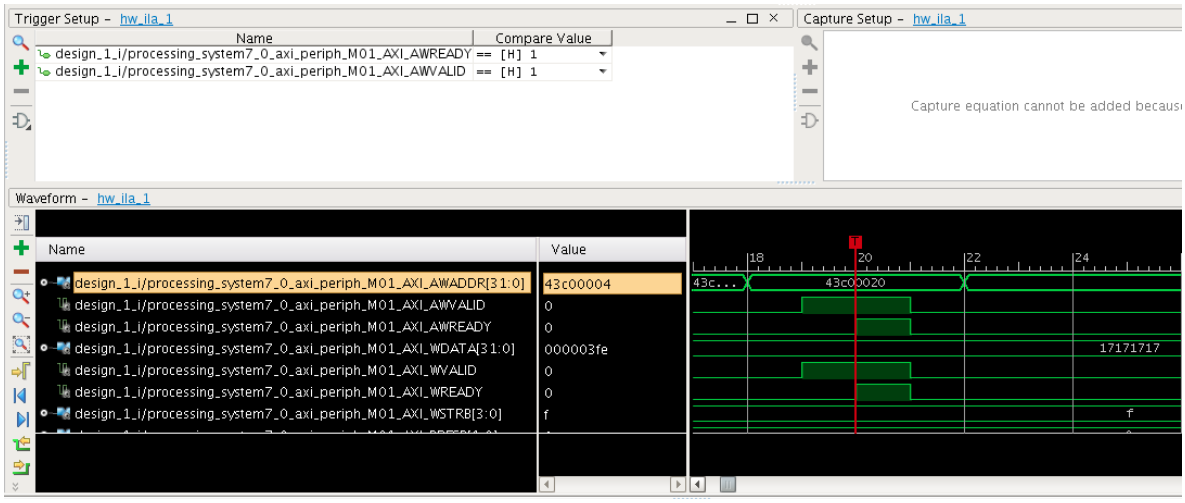

Figura 93: Validación de la UD modo Segmentation - Señales "S_AXI" II

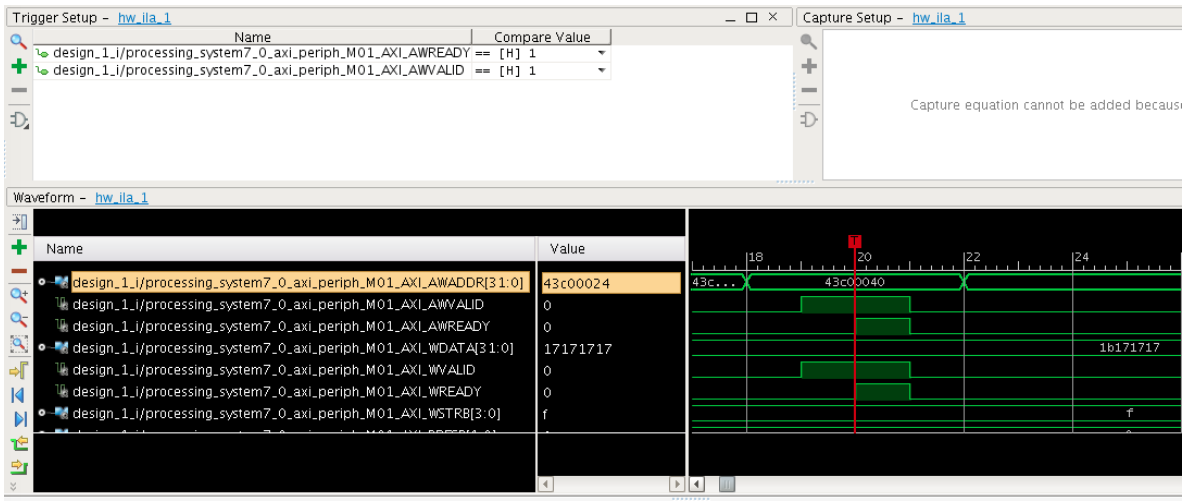

Figura 94: Validación de la UD modo Segmentation - Señales "S_AXI" III

Tras realizar el cambio de modo de operación ya es posible realizar el envío de datos y comprobar a través de los ILAs introducidos en el diseño que se realizan las transmisiones de datos de forma correcta. En la Figura 95 se muestra el bus de datos "data_in_ethernet" que contiene los datos recibidos a través de la unidad de red, en este caso el DMA. Se comprueba que la UD recibe todos los datos de forma precisa y sin incidentes.

Como consecuencia, se realiza el envío a los bloques aceleradores activos. En este caso son los ocho incluidos en la UD. En primer lugar, en la Figura 96 se muestra el bus de datos de "data_out_accel1" que coincide con los datos enviados al primer bloque acelerador. Como se puede observar se envían un total de 23 palabras de 64 bits, tal y como se configuró en el nuevo modo de operación. En segundo lugar, en la Figura 97 se muestran los datos enviados al octavo bloque acelerador conectado a la UD con el bus de datos de "data_out_accel8". En este caso, se envían 27 palabras de 64 bits como estaba previsto.

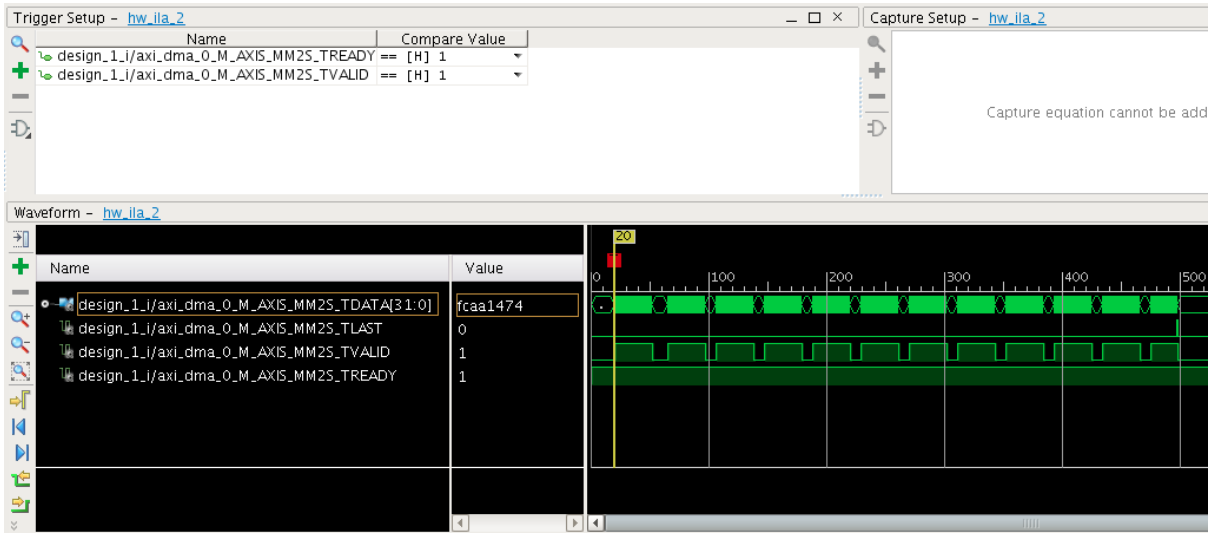

Figura 95: Validación de la UD modo Segmentation - Señales "data_in_ethernet"

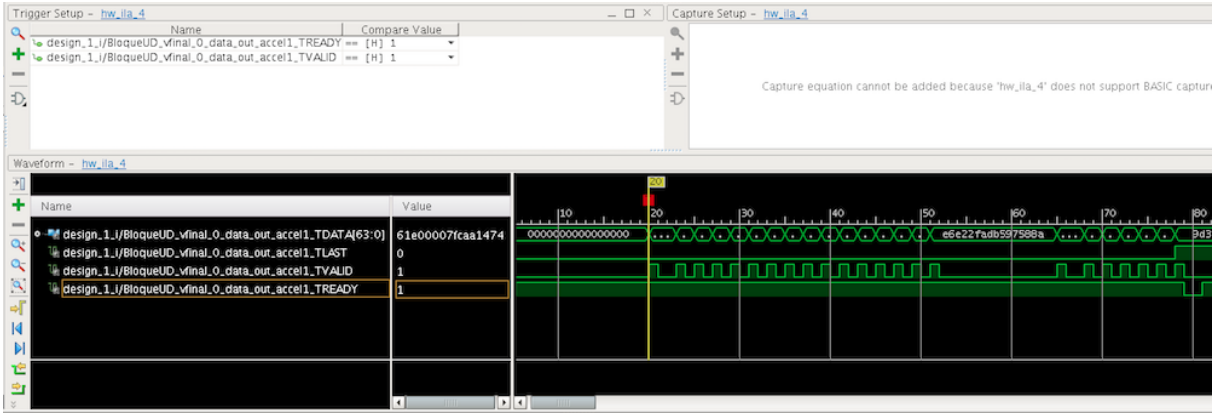

Figura 96: Validación de la UD modo Segmentation - Señales "data_out_accel1"

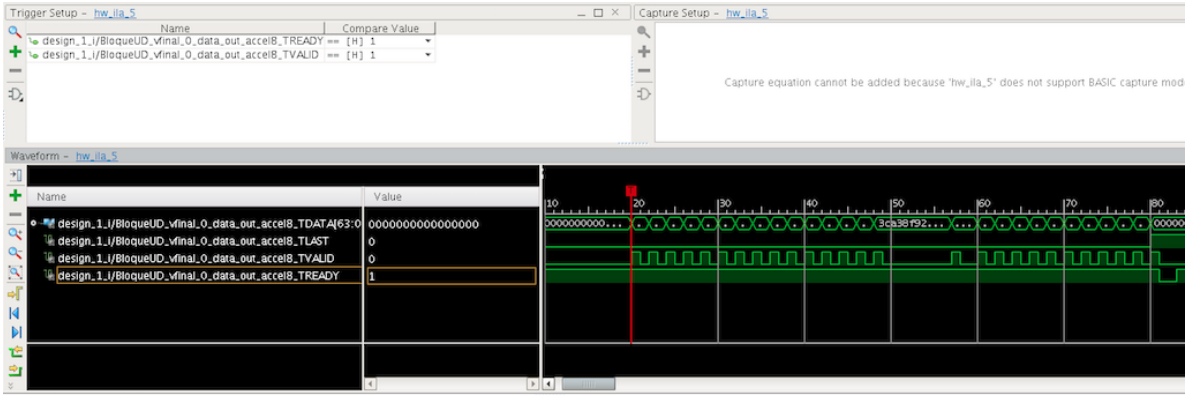

Figura 97: Validación de la UD modo Segmentation - Señales "data_out_accel8"

Por último, solamente es necesario comprobar que los datos son devueltos a la unidad de red, en este caso el DMA, de forma correcta y sin pérdidas de información. En la Figura 98 se muestra como los datos son devueltos a través del bus "data_out_ethernet". Al verificar los paquetes mediante los ILAs y el terminal del Minicom se puede asegurar el correcto funcionamiento de la UD para el modo de operación Segmentation.

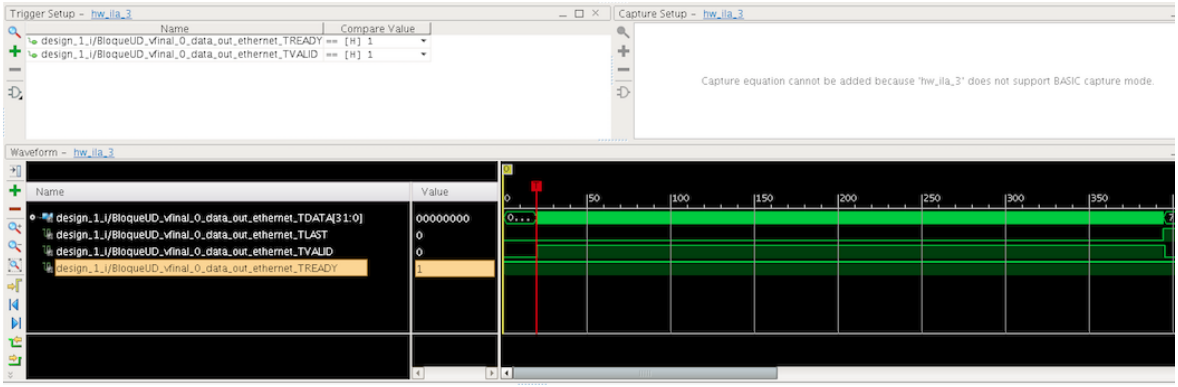

Figura 98: Validación de la UD modo Segmentation - Señales "data_out_ethernet"

6.5. Validación de la lectura del puerto de configuración

Para poder validar la funcionalidad total de la UD, no se deben olvidar las lecturas del puerto de configuración que determinan el modo de operación actual del bloque y si se está produciendo un cambio de modo de operación. En las siguientes figuras se ha muestra una lectura del puerto de configuración realizada sin introducir ningún cambio de modo de operación. Esto implica que el modo de operación por defecto es el que se obtendrá en las lecturas. En la Figura 99 se muestra la lectura de la primera dirección asignada a la UD, que devuelve los valores de las señales "operation_mode" y "addr". El resultado devuelto es "0x4" lo que implica que el modo de operación utilizado es "0" o *Unicast* y que solo se encuentra activo el primer bloque acelerador, tal y como se ha configurado el modo de operación por defecto en la UD.

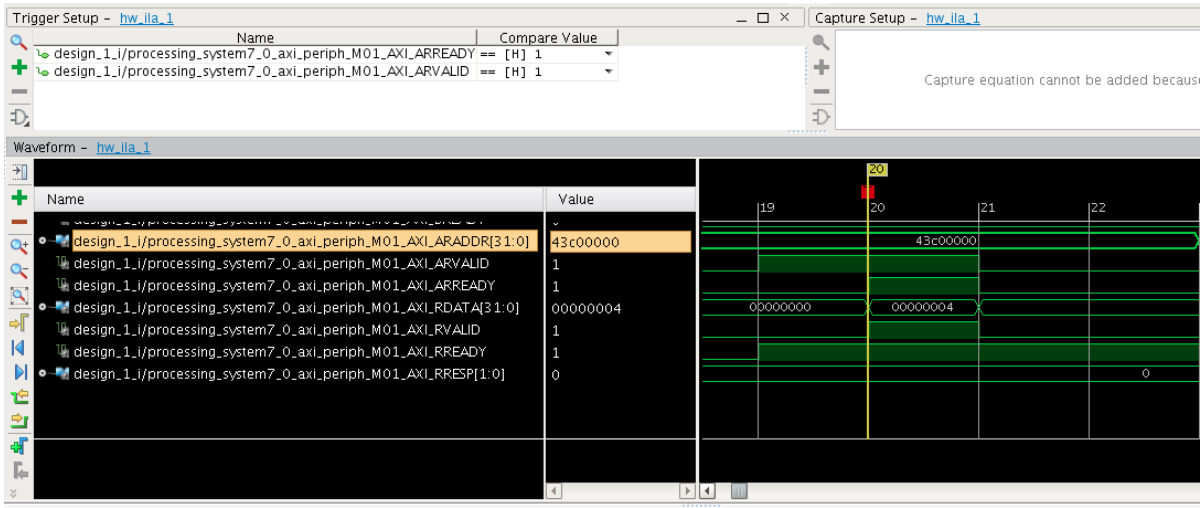

Figura 99: Validación de la UD - Lectura del puerto de configuración I

A continuación, se realiza la lectura de las tres direcciones siguientes asignadas a la UD. Dos de ellas muestran el número de segmentos de cada uno de los bloques aceleradores incluídos en la UD y la última muestra el estado de la UD, avisando al usuario de si un cambio de modo de operación está teniendo lugar o si el nuevo modo de operación implementado se ha realizado de forma correcta. Estas tres lecturas nos devuelven el mismo valor que es "0x0" lo que implica que no existen segmentos para ningún bloque acelerador y que no se está efectuando ningún cambio de modo de operación y que no se ha producido ningún error. Por ello, solo se mostrará un ejemplo en la Figura 100 que coincide con la última dirección de lectura para saber el estado de la UD. Gracias a ello, se asegura el correcto funcionamiento de las lecturas en la UD.

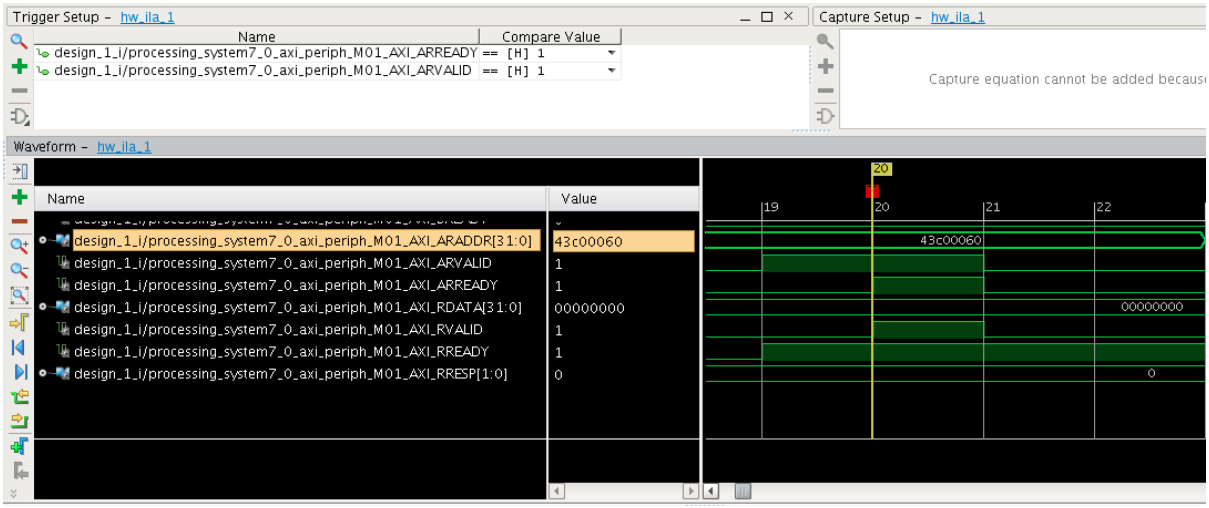

Figura 100: Validación de la UD - Lectura del puerto de configuración II

6.6. Conclusiones

Para concluir este capítulo, cabe resaltar la importancia de obtener resultados simulados que permitan verificar el sistema y resultados reales sobre la placa de desarrollo con el fin último de validarlo. Gracias a ambos factores es posible ir verificando el código a lo largo de su desarrollo, para posteriormente, tras ser implementado sobre el dispositivo, poder asegurarnos de su correcto funcionamiento, sin la necesidad de realizar demasiados casos de "prueba-error".

Inicialmente, se ha comprobado con una simple simulación en el lenguaje de programación SystemC el correcto funcionamiento de la UD. Se han tenido en cuenta casi todos los posibles casos a los que se puede ver sometido el sistema y se ha comprobado cada uno de ellos, haciendo el código apto para su verificación. Tras conseguir unos resultados favorables, se ha realizado la cosimulación del sistema con el diseño RTL obtenido. Gracias a ello, se ha podido verificar el sistema y la funcionalidad de la UD. Por último, gracias a la validación del sistema sobre la placa de desarrollo, se ha asegurado que el bloque desarrollado es apto para implementarse en otros sistemas más complejos, cumpliendo con las funciones de comunicación entre la unidad de red y los bloques aceleradores que procesan los paquetes de datos.

Capítulo 7. Conclusiones y trabajos futuros

En este último capítulo se exponen las conclusiones generales del proyecto, explicando los aspectos esenciales de cada uno de los capítulos tratados a lo largo de este trabajo. También se realiza un análisis de los resultados obtenidos con respecto a los esperados. Posteriormente, se presentan una serie de propuestas que pueden ser útiles para la realización de trabajos futuros, siguiendo la misma línea de trabajo.

7.1. Conclusiones del Proyecto

Tras realizar un análisis del desarrollo de la Unidad de Despacho diseñada y una verificación y validación de su funcionamiento, se puede confirmar su funcionalidad y su validez para ser implementado en la placa de desarrollo Zedboard. Los objetivos generales definidos para el desarrollo del proyecto han sido cumplidos, obteniendo como resultado un bloque IP funcional, que permite realizar las comunicaciones entre la unidad de red y los bloques procesadores.

Inicialmente, se ha llevado a cabo un análisis de la necesidad de mejorar la velocidad de transmisión de paquetes recibidos por la red y la capacidad de las FPGAs para soportar el desarrollo de dispositivos rápidos, eficientes y fiables. Especialmente, se han explicado las características que permiten el uso de la placa de desarrollo Zynq para poder implementar la UD encargada de las comunicaciones de la unidad de red del dispositivo. Para ello, ha sido necesario realizar el estudio de las herramientas que permiten un rápido desarrollo, síntesis, implementación, integración, programación, verificación y validación del bloque desarrollado.

El flujo de diseño utilizado permite volver atrás en el desarrollo de la UD, para efectuar los cambios necesarios y cumplir con los requisitos de funcionamiento. Al obtener el diseño en el lenguaje de programación de alto nivel SystemC se permite verificar el código de forma más exhaustiva, obteniendo una síntesis más completa del sistema y dirigida a optimizar los recursos deseados. El flujo de diseño escogido, incluye inicialmente realizar una verificación del sistema en SystemC en la herramienta SimVision, para posteriormente proceder a la síntesis en la herramienta CtoS y obtener un diseño RTL de la UD. Antes de realizar la síntesis lógica del diseño se ha optado

por efectuar una cosimulación que incluya el diseño en SystemC y el diseño RTL obtenido, para asegurar el correcto funcionamiento del sistema antes de conseguir la implementación del sistema. Para la realización de la síntesis lógica se ha optado por utilizar la herramienta Synplify Premier y obtener el fichero EDIF necesario para la generación del *bitstream* en Vivado Design Suite. Por último, la programación y la verificación del diseño se han realizado utilizado la herramienta SDK.

Tras la realización del diseño del bloque IP que integra el módulo de la UD creada, se ha conseguido un bloque con una frecuencia de funcionamiento de 241,6 MHz, permitiendo una velocidad de transmisión de 7,7 Gbps.

La implementación realizada sobre la ZedBoard es capaz de funcionar a 200 MHz. El factor limitante de la plataforma de prototipado utilizada son los bloques DMA que pueden funcionar hasta 150 MHz. Para la implementación de la plataforma, el uso de recursos de la UD es reducido, siendo el uso de las memorias FIFOs el factor más sensible en función de la implementación final (tamaño de las FIFOs, tamaño de la palabra). Finalmente, tras realizar la verificación de todos los modos de operación en diferentes situaciones y en diferentes fases del flujo de diseño del sistema, se puede afirmar que la UD es un bloque completamente funcional que cumple con todos los requerimientos planteados para su desarrollo.

Cabe destacar, la portabilidad y la flexibilidad de la UD al haber sido desarrollada en lenguaje de alto nivel (SystemC) y no hacer uso directo de las memorias de la placa de prototipado, sino hacer uso de memorias FIFOs que pueden ser implementadas en múltiples dispositivos. Esto se debe, a que la reusabilidad del diseño ha sido una de las premisas en el desarrollo de la UD. Igualmente, se ha definido como objetivo maximizar la frecuencia de funcionamiento para permitir la realización de comunicaciones en sistemas cuya tasa binaria sea elevada. Por este motivo, es posible la reutilización de la UD como un bloque IP, a través de un diseño RTL, en Verilog o VHDL, o como un diseño de alto nivel desarrollado en SystemC.

7.2. Trabajos futuros

A continuación se tratarán una serie de propuestas para futuros trabajos, que buscan la mejora del funcionamiento de la UD y de su rendimiento. Inicialmente, se debe plantear la posibilidad de mejorar la frecuencia de funcionamiento del sistema. En el desarrollo de este proyecto ha sido posible conseguir una frecuencia de funcionamiento de 241,6 MHz, lo que supone una tasa binaria de 7,77 Gbps. Sin embargo, sería óptimo conseguir que la UD pueda ser implementada en otros dispositivos o en plataformas que redes con tasas binarias superiores, siendo óptimo llegar a los 10 Gbps. De esta

forma se permitiría la implementación del bloque UD apto para realizar las comunicaciones en sistemas con altas velocidades de transmisión de datos.

Por otra parte, el uso de recursos del sistema también puede ser reducido, intentando reducir el tamaño de las memorias FIFO implementadas se conseguiría liberar espacio sobre la plataforma de desarrollo para poder integrar el bloque UD en sistemas más complejos que requieran de un mayor uso de recursos.

Tal y como se vio en el capítulo 4 al realizar el diseño del bloque IP, existen ciertas restricciones para la utilización del modo de operación *Segmentation*. Esto se debe a que es necesario que los datos recibidos en la UD sean de un tamaño fijo y conocido con anterioridad por el usuario, para poder obtener la programación de las señales que determinan el número de segmentos para cada bloque. Por ello, se propone mejorar el modo de operación *Segmentation* para permitir la recepción de paquetes de diferentes tamaños y que se pueda realizar de forma dinámica la distribución de segmentos entre los bloques IP aceleradores.

Por último, se pueden incluir nuevos modos de operación para ampliar la funcionalidad de la Unidad de Despacho diseñada. De esta manera se pueden cumplir con requerimientos de nuevos sistemas en donde se quiera efectuar la integración del bloque diseñado.

Referencias

[1] Nielsen, J. (2014). *Nielsen's Law of Internet Bandwidth*. Available: https://www.nngroup.com/articles/law-of-bandwidth/.

[2] Xilinx Inc., "Zynq-7000 All Programmable SoC Product Advantages," 2016.

[3] Xilinx Inc. (2016). *Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit*. Available: http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html#overview.

[4] Xilinx Inc., "lwIP Library (v2.00.a)," 08/01/2007, 2007.

[5] Johnson, J. (08/12/2015). *Using AXI Ethernet Subsystem and GMII-to-RGMII in a Multi-port Ethernet design*. Available: http://www.fpgadeveloper.com/2015/12/using-axi-ethernet-subsystem-and-gmii-to-rgmii-in-a-multi-port-ethernet-design.html.

[6] Opsero Electronic Design Inc. (2015). *Ethernet FMC. Example Designs.* Available: http://ethernetfmc.com/exampledesigns/.

[7] Opsero Electronic Design Inc. (2015). *Ethernet FMC*. Available: http://ethernetfmc.com/product/ethernet-fmc/.

[8] S. Fleming and D. Thomas, "FPGA based Control for Real Time Systems," 2013.

[9] Xilinx Inc., "AXI4-Stream Infrastructrure IP Suite: LogiCORE IP Product Guide," 01/04/2015, .

[10] Xilinx Inc., "AXI4-Stream Interconnect v1.1: LogiCORE IP Product Guide," 18/11/2015, .

[11] Xilinx Inc., "A generation ahead for smarter systems: 9 reasons why the Xilinx Zynq-7000 All Programmable SoC platform is the smartest solution," 2014.

[12] Xilinx Inc., "Zynq-7000 AP SoC," 2015.

[13] Xilinx Inc., "Zynq-7000 All Programmable SoC Overview," 20/01/2016, 2016.

[14] Xilinx Inc., "Zynq-7000 All Programmable SoC. Technical Reference Manual," 23/05/2015, 2015.

[15] Xilinx Inc., "ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC. User Guide," 04/09/2015, 2015.

[16] AVNET, "Avnet Product Brief ZedBoard," 2014.

[17] AVNET, "ZedBoard (Zynq Evaluation and Development) Hardware User's Guide," 27/01/2014, 2014.

[18] Xilinx Inc., "Vivado Design Suite User Guide: High-Level Synthesis," 01/10/14, 2014.

[19] Xilinx Inc., "Introduction to High-Level Synthesis with Vivado HLS: Vivado HLS 2013.3 Version," 2013.

[20] T. Manikas and M. Thornton, "Tutorial for Cadence SimVision Verilog Simulator Tool," 13/06/2013, 2013.

[21] ePM, "SimVision User Guide," 2005.

[22] Cadence, "Cadence C-To-Silicon Compiler: High-Level Synthesis," 2008.

[23] Synopsis, "Synplify Pro and Premier: Fast, Reliable FPGA Implementation and Debug," 2015.

[24] Synopsis (2016). *Synplify Premier: Accelerate Implementation of FPGA Designs and FPGA-based Prototypes*. Available: https://www.synopsys.com/tools/implementation/fpgaimplementation/pages/synplify-premier.aspx.

[25] Xilinx Inc., "Vivado Design Suite User Guide: Getting Started," 30/09/2016, 2015.

[26] Xilinx Inc., "Vivado Design Suite User Guide: Design Flows Overview," 06/04/2016, 2016.

[27] Xilinx Inc., "Vivado Design Suite User Guide: Designing with IP," 30/09/2016, 2015.

[28] Xilinx Inc., "Vivado Design Suite User Guide: Implementation," 06/04/2016, 2016.

[29] Xilinx Inc., "Software Development Kit (SDK)," 2015.

[30] Xilinx Inc., "Vivado Design Suite: AXI Reference Guide," 21/06/2015, 2015.

[31] R. Griffin, "Design a Custom AXI-lite Slave Peripheral," Julio 2014, 2014.

[32] IUMA (2016). *Resumen y costes. Costes de mantenimiento de las herramientas.* Available: sth.iuma.ulpgc.es/index.php/resumen-y-costes.

[33] Microsoft (2016). *Microsoft Office para el Hogar, Estudiantes y Profesionales - Microsoft Store*. Available: https://www.microsoftstore.com/store/mseea/es_ES/cat/Office/categoryID.66226700?s_kwcid=AL!4249!3!119758906361!e!!g!!microsoft%20office&WT.mc_id=pointitsem+Google+Adwords+Office+Generic+-+ES&ef_id=V3wJIgAAACkeleQ-:20160705192314:s.

[34] ULPGC, "BOULPGC: Boletión Oficial de la Universidad de Las Palmas de Gran Canaria," 04/11/2010, 2010.

Presupuesto

Para la obtención del presupuesto debemos tener en cuenta todos los recursos utilizados para la realización del proyecto. Los recursos necesarios han sido humanos y materiales en cada una de las etapas del desarrollo del bloque IP y su implementación, integración, verificación y validación. A continuación se mostrarán los costes asociados al proyecto, dividiéndoles en recursos hardware, software y humanos, y haciendo un presupuesto total del mismo, teniendo en cuenta el material fungible y los gastos de edición del proyecto.

1. Recursos *hardware*

Para la realización de este proyecto ha sido necesario el uso de diferentes herramientas de trabajos, así como de la placa de desarrollo utilizada. Estos recursos *hardware* conllevan un coste que se describe en la Tabla 7 que se puede observar a continuación. Se debe tener en cuenta que los costes son estimativos, teniendo en cuenta el uso realizado del equipo y el ciclo de amortización del mismo.

Tabla 7: Costes de recursos hardware

Recurso	Coste del equipo	Tiempo de utilización	Coste para el TFG
ZedBoard	424,30 €	60 horas	424,30 €
Workstation de diseño y programación	1.800,00 €	170 horas	184,34 €
Cables y equipo auxiliar	40,00 €	60 horas	40,00 €
TOTAL:			648,64 €

2. Recursos *software*

Los recursos *software* utilizados para la realización de este proyecto han sido todas las herramientas necesarias para la realización del bloque IP, su implementación y verificación, así como los recursos de edición del proyecto utilizados. Los costes de las herramientas de diseño y síntesis utilizadas Xilinx Vivado Design Suite, Cadence CtoS, Cadence Simvision y Synopsis Simplify requieren únicamente de un mantenimiento anual debido a la licencia universitaria [32]. En la Tabla 8 se puede observar los costes *software* detallados, sin olvidar que al igual que con los recursos *hardware* los valores serán estimados, pues debemos tener en cuenta el ciclo de amortización y el periodo de utilización [33].

Tabla 8: Costes de Recursos Software

Recurso	Tipo de licencia	Coste de licencia	Mantenimiento Anual	Coste para el TFG
Xilinx Vivado Deisgn Suite	Universitaria	Donación	214,00 €	35,18 €
Cadence CtoS y SimVision	Universitaria	Donación	1.968,80 €	323,64 €
Synopsys Synplyfy	Universitaria	Donación	1.123,50 €	184,69 €
Entorno de programación Eclipse C/C++/SystemC	Licencia Pública	-	-	-
Microsoft Office 2016	Hogar y Estudiantes	149,00 €	-	149,00 €
TOTAL:				692,51 €

3. Recursos humanos

A continuación, en la Tabla 9 se consideran los recursos humanos necesarios para el desarrollo de este proyecto. Como referencia se ha tomado el coste por hora de un ingeniero técnico de telecomunicaciones con un contrato anual, teniendo en cuenta que se especializa en realización de proyectos [34].

4. Material fungible

El material fungible se encuentra formado por las impresoras y el material de papelería necesario para la impresión de la memoria del proyecto. También incluye los discos necesarios para realizar las copias de seguridad necesarias del proyecto. El coste del material fungible es estimado y tiene un valor de 200,00 €.

Tabla 9: Coste de Recursos Humanos

Tarea	Coste por hora	Horas	Coste
WP1. Estudio de la documentación	16,65 €	60	999,00 €
WP2. Diseño de la arquitectura	16,65 €	50	832,50 €
WP3. Desarrollo de la Unidad de Despacho	16,65 €	60	999,00 €
WP4. Integración e implementación	16,65 €	30	499,50 €
WP5. Validación	16,65 €	30	499,50 €
WP6. Documentación final	16,65 €	50	832,55 €
WP7. Control y seguimiento del proyecto	16,65 €	20	333,00 €
TOTAL:			4995,00 €

5. Costes de edición del TFG

Para la impresión y encuadernación de la memoria del proyecto debemos tener en cuenta un coste de 80,00 €, teniendo en cuenta todos los materiales necesarios para ello.

6. Coste total del TFG

Para la finalización del presupuesto se puede observar en la Tabla 10 los valores de cada uno de los apartados vistos anteriormente y el coste total del TFG.

Tabla 10: Coste total del Proyecto

Recursos	Coste
1. Recursos *hardware*	648,64 €
2. Recursos *software*	692,51 €
3. Recursos humanos	4995,00 €

Recursos	Coste
4. Material fungible	200,00 €
5. Coste de edición del proyecto	80,00 €
Subtotal:	6616,15 €
IGIC (7%)	463,13 €
TOTAL:	7079,28 €

Dª. Irene González Crespo declara que el presupuesto del presente proyecto asciende a siete mil setenta y nueve euros y veintiocho céntimos (7079,28 €).

Las Palmas de Gran Canaria, a 22 de julio de 2016,

Fdo.: Irene González Crespo

Pliego de condiciones

Todos los resultados obtenidos a lo largo de este proyecto, así como el desarrollo del mismo y el procedimiento realizado son válidos para los recursos *software*, *hardware* y de edición del proyecto listados a continuación:

RECURSOS *SOFTWARE*

- Workstation de diseño y programación:
    - OS Red Hat versión 5
- Herramientas de diseño y síntesis:
    - Cadence CtoS
    - Cadence SimVision
    - Synopsys Synplify
    - Xilinx Vivado Design Suite 2015.4

RECURSOS *HARDWARE*

- Plataforma de prototipado:
    - ZedBoard Zynq-7000 ARM/FPGA SoC

RECURSOS DE EDICIÓN DEL PROYECTO

- Office 2016 sobre OS Windows7 y OS Mac X El Capitan 10.11.5