Universidad de Alcalá

Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA DE COMUNICACIONES

Trabajo Fin de Grado

MOVIMIENTO EFICIENTE DE DATOS PARA UN SOC BASADO EN ZYNQ

Autor: Javier Pavón Luque

Tutor/es: Raúl Mateos Gil

2015

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA DE COMUNICACIONES

Trabajo Fin de Grado

MOVIMIENTO EFICIENTE DE DATOS PARA UN SOC BASADO EN ZYNQ

Autor: Javier Pavón Luque

Director: Raúl Mateos Gil

Tribunal:

Presidente:

Vocal 1º:

Vocal 2º:

Calificación: .....

Fecha: .....

AGRADECIMIENTOS

Antes de comenzar con el desarrollo del trabajo fin de grado, me gustaría expresar mi más sincero agradecimiento a todas aquellas personas que me han apoyado a lo largo de estos meses de trabajo.

A mi tutor del proyecto, Raúl Mateos, no solo por todos los conocimientos que he adquirido gracias a él y que serán muy útiles en mi vida profesional, sino también por su paciencia, esfuerzo y dedicación durante estos meses a la hora de resolver cualquier duda.

Dar las gracias a mi familia, en especial a mis padres, que sin ellos no hubiera sido posible alcanzar esta meta y lograr una formación académica, por su apoyo en los momentos difíciles y en las decisiones que he tomado y por haber confiado siempre en mí.

A Cristina, por su paciencia y su apoyo durante la realización de este trabajo, y por estar ahí en los momentos de estrés y agobio.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

Este trabajo muestra una solución para la transferencia eficiente de datos entre el procesador y la lógica programable en un SoC basado en Zynq. Esta solución se basa en el uso de un controlador DMA y el puerto ACP, lo que permite descargar al procesador de las tareas de movimiento de datos y mantener la coherencia de la caché.

Para ello se ha diseñado un periférico en la lógica programable que se comporta como un cliente DMA. Este diseño se ha verificado mediante simulación utilizando BFMs para modelar el comportamiento de las interfaces del procesador.

Palabras Clave: Zynq, DMA, BFM

ABSTRACT

This project shows a solution for an efficient data movement between processing system and programmable logic in a Zynq-based SoC. This solution will use a DMA Controller and the ACP port in order to reduce the processor's load due to data movement and maintain the cache coherence.

A peripheral has been designed in the programmable logic so that it behaves as a DMA client. This design has been verified using BFMs in the simulation to model the processor interface behavior.

Keywords: Zynq, DMA, BFM

RESUMEN EXTENDIDO

La familia de dispositivos Zynq de Xilinx son System on Chip que incorporan en un mismo circuito integrado tanto el sistema de procesamiento como la lógica programable. Esto implica diversas mejoras, ya que permite la reducción de tamaño del dispositivo y una disminución de la potencia consumida.

Al disponer de los dos dispositivos en un mismo chip, se consigue la flexibilidad y escalabilidad que aporta la lógica programable junto con la potencia del procesador, lo que permite implementar algoritmos complejos repartidos entre ambas partes. Para que esto sea posible, es necesario realizar una transferencia de datos entre ambas de forma eficiente, de manera que se consiga descargar al procesador de las tareas de movimiento de datos.

Una de las soluciones más eficientes para esta tarea es el empleo de DMA, cuya función es la de transferir datos entre la lógica programable y el procesador, liberando de carga a éste último. El DMA es un IP Core proporcionado por Xilinx que incorpora dos módulos o canales, un canal de transmisión donde los datos procedentes de la memoria se transmiten como un stream de datos hacia un periférico cliente que los absorbe, y un canal de recepción, donde el stream de datos procedente del periférico cliente se transfiere a la memoria. Ambos dispositivos, controlador de DMA y periférico cliente se implementan en la lógica programable.

El DMA permite distintos modos de funcionamiento. El modo de funcionamiento simple permite realizar transferencias de datos entre la memoria y los periféricos o viceversa, de datos contiguos ubicados en una dirección de memoria. Sin embargo, es común que los datos no aparezcan contiguos en memoria, sino que estén dispersos en diferentes posiciones. Para solucionar este problema se configurará el DMA en modo Scatter/Gather, el cual permitirá transmitir datos que se encuentran dispersos en memoria (Gather) o recibir datos y dispersarlos (Scatter). Este modo requiere la utilización de unas estructuras llamadas descriptores o BD (Buffer Descriptor), que están enlazados entre sí como una lista circular, y cada uno contiene la dirección de los datos a transmitir o la posición del buffer donde almacenar los datos recibidos, su tamaño y un puntero al siguiente BD, entre otra información.

Como interfaz entre el procesador y el DMA se utilizará el puerto ACP, cuya principal característica es que permite asegurar la coherencia de datos en las transferencias. El empleo de este puerto permite solucionar el problema de coherencia entre la memoria del dispositivo y la memoria caché cuando los datos de la primera son distintos a los de la segunda.

Para llevar a cabo la transferencia de datos se necesitará un periférico en la lógica programable que actúe como cliente DMA. Para ello se diseñará un IP Core en Vivado con interfaces AXI4-Stream. Este periférico deberá cumplir las especificaciones de una

futura aplicación, en la que en el interior de este IP existirá un core con interfaces de entrada y salida del tipo AXI4-Stream de 16 bits de bus de datos.

Para el máximo aprovechamiento del DMA y el ACP, cuyas interfaces son de 64 bits, habrá que diseñar la lógica encargada de la conversión de anchura de datos. Se dispondrá de un downsizer que se encargará de convertir la anchura de la interfaz de entrada de 64 a 16 bits, mientras que un upsizer realizará el procedimiento inverso para convertir los datos a la salida de 16 a 64 bits. Además el IP a diseñar debe tener una interfaz AXI4-Lite adicional destinada a la escritura y lectura de registros internos de control y estado del futuro core, aunque en este proyecto no se implementará esta funcionalidad, tan solo actuará como un registro cuyos datos a la salida serán los mismos que a la entrada.

El diseño de cada uno de los bloques que componen el IP se verificará mediante simulación utilizando ModelSim. De esta forma se puede comprobar que los protocolos de comunicaciones funcionan correctamente y los datos obtenidos son los esperados. Para ello se deberá cumplir el procedimiento de negociación entre el maestro de la interfaz y el esclavo. Este procedimiento indica que una transferencia tiene lugar cuando las señales TVALID y TREADY están activas, es decir, el maestro pone a la salida un dato válido y el esclavo indica que está disponible para recibir otro dato porque ya ha leído el actual.

Una vez realizada la verificación, el código VHDL se encapsulará junto a una serie de ficheros de control en un IP Core en un proceso llamado empaquetamiento, permitiendo su posterior reutilización en Vivado. Se seguirá el procedimiento indicado en el asistente que proporciona IP Packager, realizando las modificaciones posteriores necesarias para que la funcionalidad del IP sea la deseada. En este proceso de empaquetado del IP será necesario crear una interfaz gráfica para la configuración de los distintos genéricos utilizados, lo que facilitará la reutilización del IP.

Tras la creación del IP, se llevará a cabo una simulación con un componente llamado BFM. Para realizar una simulación del IP Core es necesario modelar el comportamiento del procesador para simular las transferencias de datos, pero esto puede ser una tarea difícil, bien porque no se disponga del modelo, o bien por su complejidad a la hora de simular. Por ello se emplea este componente, que es capaz de modelar el comportamiento de las interfaces de comunicaciones del procesador, de forma que no sea necesario simular el modelo del procesador completo.

El BFM está formado por varias capas, las cuales proporcionan distintas APIs con funciones específicas para cada tipo de interfaz (AXI4-Lite, AXI4-Stream…), cuya finalidad es la configuración del BFM o el envío y recepción de los estímulos necesarios para realizar una transferencia de datos. Estas funciones están escritas en SystemVerilog, luego el testbench a realizar se escribirá también en dicho lenguaje.

El escenario de simulación estará formado por un BFM maestro, encargado de las transferencias de datos hacia el IP bajo test, y un BFM esclavo que recibirá los datos procedentes del IP. Al recibir los datos en este último se comparará el valor de todas las

señales recibidas con los valores esperados, si éstos coinciden se podrá afirmar que la transferencia ha sido correcta.

Una vez verificado el funcionamiento del IP creado, se procederá a integrar todos los componentes en la arquitectura del SoC con el IP Integrator de Vivado. En primer lugar se instanciará el sistema de procesamiento, que debe ser configurado para que el puerto del ACP y los puertos para interrupciones personalizadas procedentes de la lógica programable estén activos.

A continuación se agregará el DMA al diseño. Se conectarán las interfaces mapeadas en memoria al puerto ACP del procesador a través de un AXI Interconnect, y las interrupciones de ambos canales del DMA al puerto correspondiente de las interrupciones procedentes de la lógica programable. En último lugar se añadirá el IP del cliente DMA y se conectarán las interfaces AXI4-Stream a los canales de transmisión y recepción del DMA, y la interfaz AXI4-Lite de configuración al procesador.

Terminada la arquitectura del SoC, se continuará con el desarrollo de la aplicación software que permitirá controlar el DMA. Xlinx proporciona los drivers junto con el IP, los cuales además de las funciones orientadas a la configuración del DMA, proporcionan funciones para llevar a cabo las transferencias deseadas, generación de los anillos de descriptores y control de éstos. La operación del controlador DMA se gestionará mediante interrupciones, en lugar de por sondeo. Cada canal tendrá una interrupción asociada, que será controlada por la aplicación mediante una función de atención a la interrupción.

En el proceso de configuración del DMA, será necesario configurar ambos canales para que operen de la manera deseada, y habrá que crear dos anillos de descriptores, uno para cada canal. Los descriptores del anillo del canal de transmisión estarán enlazados como una lista circular, pero todos sus campos estarán sin inicializar, esperando que la aplicación se encargue de su control para realizar una transferencia. Sin embargo, los descriptores del anillo de recepción serán inicializados con la dirección de memoria del buffer de recepción, y serán enviados al hardware para que se pueda comenzar a recibir datos.

Para realizar una transferencia habrá que configurar uno de estos descriptores con la dirección de memoria donde se encuentran los datos a transmitir, así como el tamaño de éstos. Entonces se enviará al hardware, que se encargará de la trasmisión de los datos indicados, y una vez haya acabado se producirá una interrupción en este canal. En la función de atención a dicha interrupción se volverá a liberar el descriptor para que pueda ser utilizado en el futuro.

Los datos habrán sido transmitidos al cliente DMA, cuya función será transferir dichos datos a la salida tras haber pasado por los distintos bloques comentados anteriormente. Cuando estos datos lleguen de nuevo al canal de recepción del DMA, serán transmitidos hacia la memoria, y se producirá una interrupción en dicho canal cuando se haya acabado de transmitir el dato. En la función de interrupción de dicho canal, habrá que

obtener los descriptores recibidos del hardware para tratar sus datos, y poder liberarlos para mandarlos de nuevo al hardware.

Para probar el funcionamiento del DMA en una situación real se ha diseñado un escenario de prueba que se implementará sobre ZedBoard, una tarjeta de desarrollo de bajo coste que incorpora un SoC Zynq-7020, e incluye diversos periféricos y conectores para facilitar la tarea de desarrollo.

En dicho escenario de prueba será necesario añadir la funcionalidad de comunicaciones Ethernet a ZedBoard, para ello se utilizará la pila de protocolos TCP/IP llamada LwIP, creada especialmente para sistemas empotrados, ya que los recursos necesarios para su uso son muy reducidos. LwIP proporciona una API de bajo nivel llamada “Raw API”, que no necesita de sistema operativo para su funcionamiento y proporciona las funciones necesarias para su control.

En la prueba se enviará un stream de video por un cable Ethernet desde un ordenador hasta la tarjeta de desarrollo ZedBoard, empleando para ello el programa VLC Player. Cuando se reciba un paquete UDP, se hará una llamada a una función callback de recepción, que se encargará de configurar un descriptor del DMA para la transmisión de los datos recibidos hacia la lógica programable.

El proceso de transmisión de datos con el DMA será el mismo que se comentó anteriormente, pero al recibir los datos de vuelta, éstos serán tratados para volver a ser enviados en otro paquete UDP. Esto se llevará a cabo en la función de interrupción del canal de recepción. Al recibir los datos, se empaquetarán en un datagrama UDP, que será enviado mediante las funciones que proporciona la API de LwIP.

Una vez diseñada la aplicación software que incluya LwIP, se podrá probar el escenario de prueba descrito. Con un reproductor VLC Player se enviará el stream de video, que será transmitido con el DMA, y se recibirán los mismos datos de vuelta en otro reproductor para visualizar el resultado.

LISTA DE ACRÓNIMOS

ACP – Accelerator Coherency Port

ADC – Analog to Digital Converter

API – Application Programming Interface

APU – Application Processor Unit

ASIC – Application-Specific Integrated Circuit

BD – Buffer Descriptor

BFM – Bus Functional Model

BSP – Board Support Package

CPU – Central Processing Unit

DMA – Direct Memory Access

DSP – Digital Signal Processor

DUT – Device Under Test

EMIO – Extended MIO

EOF – End of Frame

FIFO – First In First Out

FPGA – Field Programmable Gate Array

FPU – Floating Point Unit

GIC – Generic Interrupt Controller

GPIO – General Purpose Input Output

GUI – Graphical User Interface

HDL – Hardware Description Language

IDE – Integrated Development Environment

IP – Intellectual Property

IPIF – IP Interface

LwIP – Lightweight IP

MIO – Multiplexed Input/Output

MM – Memory Mapped

MM2S – Memory Mapped to Stream

MMU – Memory Management Unit

OCM – On Chip Memory

PBUF – Packet Data Buffer

PCB – Protocol Control Block

PL – Programmable Logic

PLC – Power Line Communications

PS – Processing System

S2MM – Stream to Memory Mapped

SCU – Snoop Control Unit

SIMD – Simple Instruction Multiple Data

SoC – System on Chip

SOF – Start of Frame

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

1 INTRODUCCIÓN

Un System on Chip (SoC) es un dispositivo electrónico que incorpora todos los elementos de un sistema electrónico en un mismo circuito integrado. En la actualidad, los SoC han adquirido gran importancia debido a las diversas ventajas que proporcionan, en comparación con otras arquitecturas.

Debido a la tendencia que tienen los dispositivos electrónicos a reducir su tamaño, los SoC destacan en este campo, ya que al integrar todos los módulos en un mismo chip, permiten reducir sus dimensiones considerablemente. También cobra importancia la disminución de potencia consumida, muy importante en sistemas empotrados portátiles, ya que permiten alargar la duración de la batería.

En este proyecto se utilizará un SoC basado en FPGA, que es un tipo de SoC que proporciona múltiples ventajas en comparación con otras arquitecturas como los ASIC. Por ejemplo, destaca una gran reducción en el coste y una mayor rapidez en el desarrollo, que a su vez implica un menor tiempo de puesta en mercado.

Una de las posibilidades que ofrecen los SoC basados en FPGA, cuyo uso es muy común, es la implementación de parte de los algoritmos en la lógica programable y parte en el procesador. Para que esto sea posible es necesario realizar un movimiento de datos eficiente entre ambas partes, de lo contrario, es posible que los resultados no sean los esperados y no se obtenga ninguna ventaja de esta implementación mixta.

El objetivo de este trabajo será realizar una transferencia de datos entre el sistema de procesamiento y la lógica programable de forma eficiente, de manera que se consiga descargar al procesador de las tareas de movimiento de datos, permitiendo que se puedan implementar algoritmos en él. La solución a este problema se basará en el empleo del DMA (Direct Memory Access), que se encargará de realizar la transferencia de datos sin influir en el rendimiento del procesador.

El diseño se implementará en Zynq, un SoC de Xilinx con un procesador de doble núcleo ARM Cortex-A9 y una lógica programable equivalente a una FPGA Artix-7. Las pruebas se realizarán sobre la tarjeta de desarrollo ZedBoard.

Para llevar a cabo el movimiento de datos será necesario crear un periférico en la lógica programable, que será el encargado de recibir el stream de datos procedente del DMA. En este proyecto no se realizará ningún procesado de dichos datos, sino que éstos serán devueltos al DMA para realizar el movimiento inverso de datos, desde PL hacia PS. Sin embargo, este periférico ha sido diseñado teniendo en cuenta una posible aplicación futura de comunicaciones por la red eléctrica (PLC), cuyas especificaciones se comentarán más adelante.

Hay que destacar el empleo del ACP (Accelerator Coherency Port) en la transferencia de datos entre ambas partes, ya que mediante el uso de este puerto, se asegura la coherencia de las transferencias de datos entre la memoria DDR y la memoria caché.

Como aplicación práctica para demostrar el funcionamiento del diseño realizado, se enviará un stream de video mediante el programa VLC Player desde un ordenador hacia la tarjeta de desarrollo ZedBoard, por medio de un cable Ethernet. Estos datos serán transferidos mediante el DMA hacia el cliente que se encuentra en la PL, que a su vez devolverá los mismos datos para hacer la transferencia inversa hacia el PS. Una vez los datos lleguen de nuevo a la PS, se procederá al envío de un stream para visualizarlo en otro reproductor VLC Player.

La Figura 1-1 muestra el procedimiento de la demostración práctica a realizar.

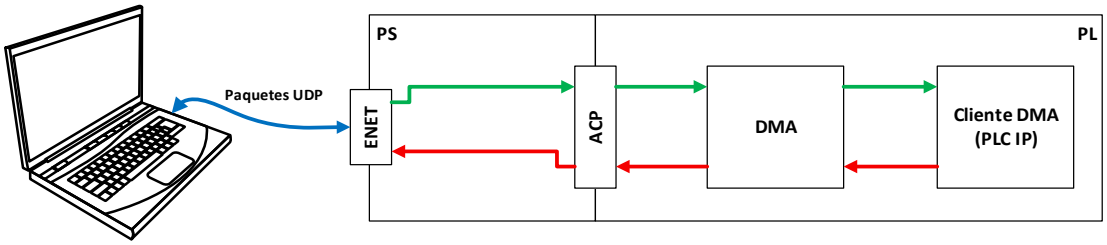

Figura 1-1: Escenario de prueba del diseño a realizar

1.1 Flujo de Trabajo

Una vez definidas las especificaciones del diseño a realizar, el flujo de desarrollo se dividirá en dos ramas, como es común en el diseño de sistemas empotrados y SoCs.

Por una parte se diseñará la arquitectura hardware, donde se instanciarán los periféricos necesarios, se realizarán las interconexiones entre éstos y el procesador, se configurará el sistema de procesamiento y las interfaces necesarias, etcétera. Por otra parte se diseñará la aplicación software teniendo una base del hardware llamada Board Support Package (BSP), que proporciona los drivers necesarios para manejar el hardware desde la aplicación.

El último paso será la integración de ambas partes, donde se comprobará que tanto software como hardware funcionan correctamente en conjunto, aunque hayan sido probadas individualmente con anterioridad.

En este proyecto se comenzará con el diseño de la arquitectura hardware y posteriormente el desarrollo de la aplicación software. A continuación se detallarán ambas partes:

Arquitectura Hardware

En primer lugar se diseñará un cliente para el DMA, capaz de comunicarse con éste mediante buses de comunicaciones AXI4-Stream. Habrá que tener en cuenta la aplicación futura, cuyo módulo encargado de las comunicaciones PLC no tiene la misma

anchura de bus de datos que el DMA, por lo que será necesario diseñar hardware adicional para poder adaptar distintas anchuras de bus.

El código VHDL se encapsulará junto a una serie de ficheros de control en un IP Core en un proceso llamado empaquetado del IP, permitiendo su posterior reutilización en Vivado.

Una vez diseñado el cliente para el DMA y empaquetado en un IP Core, habrá que someterlo a un proceso de simulación para comprobar que el diseño es robusto y no presenta fallos. Para ello se empleará en la simulación un componente llamado BFM (Bus Functional Model), que simulará el comportamiento del procesador en cuanto a interfaces de comunicación.

Una vez superado el proceso de simulación se incorporará el periférico al diseño para definir la arquitectura hardware completa, se configurará adecuadamente y se continuará con el desarrollo software.

Diseño Software

La aplicación software consta de dos partes. La primera, y más importante, es el control del DMA para llevar a cabo su configuración y poder realizar transferencias entre el sistema de procesamiento y la lógica programable, que es el objetivo principal de este proyecto.

Por otra parte, con el objetivo de probar el sistema en una situación real donde se transfiera un gran volumen de datos, se realizará el envío de un streaming de datos desde un ordenador hasta la tarjeta. Para ello se introducirá la pila de protocolos TCP/IP LwIP, que aporta la capacidad de poder transmitir y recibir paquetes Ethernet.

LwIP se utilizará en el escenario de prueba para recibir un stream de datos UDP, que se transmitirá con el DMA al cliente instanciado en la lógica programable, y posteriormente, tras recibir los datos de vuelta del cliente DMA, sean enviados de vuelta al ordenador.

1.2 Objetivos

A continuación se exponen los objetivos a alcanzar en el presente trabajo. Éstos podrán dividirse en tres grupos según las distintas etapas llevadas a cabo: una etapa de establecimiento de las bases de desarrollo de un SoC y aprendizaje, otra etapa de diseño de la arquitectura hardware y verificación de dicho diseño mediante simulaciones basadas en BFM y por último el desarrollo de la aplicación software. Los objetivos a cumplir serán los siguientes:

Bases de desarrollo de un SoC y aprendizaje:

- Aprendizaje del método de diseño de un System on Chip basado en FPGA. Se observará una clara distinción entre diseño hardware y desarrollo software durante la realización del proyecto.
- Adaptación al uso de las herramientas de desarrollo de Xilinx, en este caso Vivado Design Suite, y manejo de las herramientas de simulación que se utilizarán para verificar el diseño realizado (ModelSim y BFM).
- Comprensión del funcionamiento de los protocolos de comunicaciones AXI4-Lite y AXI4-Stream para su posterior implementación en el diseño.

Diseño de la arquitectura hardware del SoC:

- Diseño de un cliente para DMA con interfaces AXI4-Stream y AXI4-Lite. Verificación de su funcionamiento mediante simulación con ModelSim.
- Creación de un periférico (IP Core) de dicho cliente para DMA para su integración en Vivado.
- Verificación del funcionamiento del periférico mediante simulaciones empleando el modelo funcional de bus (BFM) proporcionado por Xilinx.
- Implementación de la arquitectura hardware incluyendo todos los periféricos e interconexiones necesarias.

Desarrollo del software:

- Creación de una aplicación software capaz de controlar el DMA para realizar transferencias de datos entre memoria y periféricos.
- Planteamiento de una demostración práctica en la que se enviará un streaming de video desde un ordenador hacia la placa de desarrollo por el puerto Ethernet, empleando para ello la pila de protocolos TCP/IP LwIP.

1.3 Estructura del documento

Este proyecto se ha estructurado en cuatro bloques principales:

Base Teórica: En este primer apartado se explicarán los conocimientos sobre los que se desarrollará el resto del trabajo, necesarios para comprender el funcionamiento de cada uno de los componentes que conforman el sistema completo.

En los siguientes apartados se explicarán los distintos pasos llevados a cabo para conseguir los objetivos propuestos, detallando el procedimiento utilizado y proporcionando las características de los diseños llevados a cabo, así como de la verificación de su funcionamiento. Por último se comentarán los resultados obtenidos en las pruebas con el diseño realizado.

Diseño Hardware:

- **Diseño del cliente para DMA:** Diseño en VHDL y verificación del cliente DMA que se utilizará en apartados futuros.
- **Creación del IP:** El código VHDL se encapsulará junto a una serie de ficheros de control en un IP Core, en un proceso llamado empaquetamiento del IP, lo que permitirá su posterior reutilización en Vivado.
- **Simulación BFM:** Simulación del IP creado anteriormente empleando componentes BFM para simular los buses de comunicaciones del procesador y verificar el funcionamiento del IP.
- **Definición de la arquitectura hardware:** Se incorporarán al diseño hardware todos los componentes necesarios, que se interconectarán entre ellos para proporcionar la funcionalidad deseada.

Desarrollo Software: Creación de una aplicación software capaz de controlar y configurar el DMA para llevar a cabo las transferencias entre PS y PL. También se diseñará el software empleado en las pruebas experimentales, donde se utilizará la pila de protocolos LwIP para recibir y enviar un stream de video en datagramas UDP.

Pruebas experimentales y resultados: Comprobación del funcionamiento del sistema poniendo a éste bajo prueba en el escenario descrito anteriormente, donde se transmitirá un stream de video recibido desde un ordenador por el puerto Ethernet.

2 BASE TEÓRICA

Antes de comenzar con el desarrollo del trabajo es conveniente detallar las características de los recursos, herramientas y componentes que se utilizarán a lo largo del proyecto, así como su funcionamiento.

En este apartado se introducirán los conceptos teóricos de:

- Zynq-7000, el SoC basado en FPGA que se utilizará en este proyecto. Se dará una visión general del dispositivo, su arquitectura, interfaces, y funciones de cada uno de sus subsistemas.
- ZedBoard, la tarjeta de desarrollo donde se probarán los diseños. Se describirán los distintos puertos, memoria, interruptores y elementos de visualización, entre otros periféricos que proporciona esta tarjeta.
- Descripción de los buses de comunicaciones AXI4 utilizados en la interconexión de bloques funcionales. Se explicará el funcionamiento de los diferentes protocolos y se mostrarán ejemplos de transferencias.
- Herramientas de desarrollo necesarias para el desarrollo del trabajo.
- Características de los distintos periféricos e interfaces utilizadas (DMA, BFM, ACP), así como de librerías utilizadas en el desarrollo software (LwIP).

2.1 Zynq

Zynq es un dispositivo creado por Xilinx que integra un procesador ARM Cortex-A9 de doble núcleo y una FPGA de la serie 7 de Xilinx en un mismo circuito integrado. La arquitectura de Zynq, por tanto, tiene dos partes diferenciadas, que a lo largo de este proyecto se identificarán mediante sus siglas: Sistema de Procesamiento (PS) y Lógica Programable (PL).

Al disponer de una FPGA y un procesador en el mismo chip, se trata de un SoC basado en FPGA, que ofrece la flexibilidad y escalabilidad de tener una lógica programable junto con el rendimiento y potencia de un procesador. Esto permite la implementación de algoritmos en ambas partes, un uso muy común en el diseño de SoCs. Además, en comparación con los ASIC, este tipo de SoC permite reducir considerablemente el tiempo de desarrollo y el coste, ya que no es necesario realizar un gran desembolso inicial.

Una de las características de Zynq es que se puede cortar la energía a la PL para reducir el consumo de potencia, y por otra parte, se puede reducir la frecuencia de los relojes de la PS o incluso desactivarlos para reducir el consumo.

La Figura 2-1 muestra la arquitectura de Zynq:

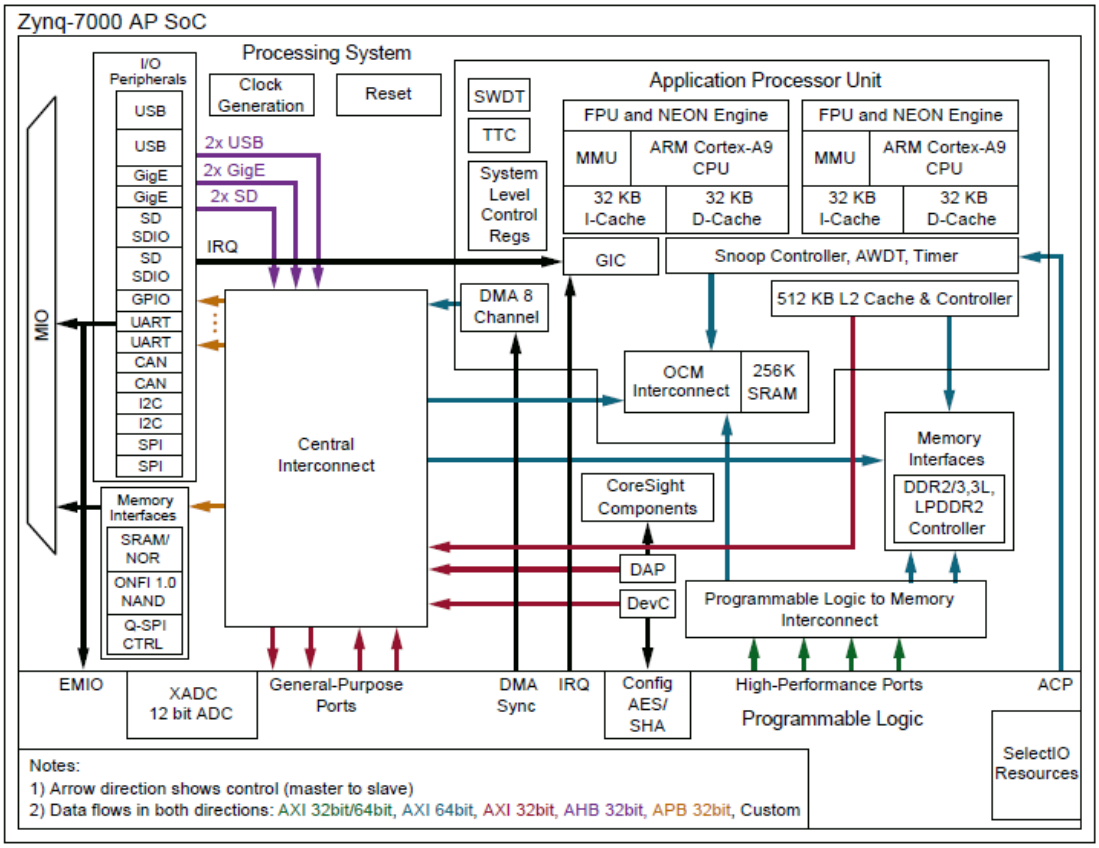

Figura 2-1: Arquitectura de Zynq. Obtenida de [Xilinx, 2015].

A continuación se introducirán los componentes de ambas partes, PL y PS. Para más información sobre los siguientes apartados, u otros aspectos de Zynq, consúltese [Crockett et al., 2014] o [Xilinx, 2015] para información más técnica.

2.1.1 Sistema de Procesamiento (PS)

El PS no sólo incluye el procesador, sino que va acompañado por un conjunto de recursos adicionales que constituyen la Application Processor Unit (APU), interfaces de memoria e interfaces con la PL, periféricos de entrada/salida y un circuito de generación de reloj.

La APU está compuesta por los siguientes submódulos:

- Los dos núcleos del procesador. Cada uno incorpora:
    - **Motor de procesamiento NEON**: Tecnología SIMD (Instrucción Única, Múltiples Datos) que permite trabajar con vectores de datos para ser más eficiente al trabajar con algoritmos:
    - **Unidad de Coma Flotante (FPU):** Provee aceleración hardware en operaciones con coma flotante.

    - **Unidad de gestión de memoria (MMU)**: Cuya función principal es convertir las direcciones de memoria virtuales en direcciones de memoria física, además de controlar los accesos a memoria.
    - **Memorias Caché de nivel 1:** Dos memorias caché, una para instrucciones y otra para datos, ambas de 32 kB.
- **Memoria Caché de nivel 2:** Memoria caché compartida por ambos núcleos del procesador, cuya capacidad es de 512 kB para datos e instrucciones.
- **Acelerator Coherency Port (ACP):** Es una interfaz AXI de 64 bits que permite la conexión entre la PS y la PL manteniendo coherencia de datos entre la memoria caché de nivel 2, la memoria de dispositivo (OCM) y la memoria caché de nivel 1.
- **Snoop Control Unit (SCU):** Módulo conectado a la memoria del dispositivo (OCM) y a las memorias caché de nivel 1 y nivel 2 que se encarga de la interconexión, transferencias de memoria PS-PL a través del ACP y coherencia entre los datos de las memorias caché.

La mayoría de las interfaces entre la PS y el exterior se llevan a cabo por los puertos entrada/salida del PS a través del Multiplexed Input/Output (MIO). Aunque otra opción es utilizar un camino no directo a través de la PL gracias al Extended MIO (EMIO).

En la siguiente lista se detallan las interfaces disponibles:

- GPIO: 54 canales MIO, 64 canales EMIO.
- USB (x2): Puede ser usado como host, device u OTG.
- SPI (x2)
- GigE (x2): Gigabit Ethernet 10Mbps, 100Mbps y 1Gbps.
- I2C (x2)
- CAN (x2)
- SD (x2): Interfaz con tarjeta de memoria SD.
- UART (x2)

2.1.2 Lógica Programable (PL)

La segunda parte de la arquitectura de Zynq es la Lógica Programable (PL). El dispositivo que incorpora la tarjeta de desarrollo a utilizar en este caso es el Zynq-7020, cuya lógica programable está basada en la FPGA de Xilinx Artix-7. Sus características principales son:

- 85K celdas lógicas. Aproximadamente 1.3M de puertas.
- 53.200 Look-up tables.
- 106.400 Flip-Flops.

- 140 BRAMs de 36Kb cada una.
- 220 DSP Slices programables.
- I/Os configurables.
- 2 conversores ADC de 12 bits, con hasta 17 estradas analógicas configurables.

2.1.3 Interfaces de comunicación PS-PL

Las interfaces disponibles entre PS y PL que permiten comunicar ambas partes están basadas en los buses de comunicaciones AXI4, que se explicarán en el apartado 2.3. Estos protocolos facilitan y mejoran la productividad en el proceso de creación de una arquitectura de un SoC, ya que existe una gran cantidad de IP Cores disponibles que los utilizan. Un IP Core es un bloque que constituye un subsistema y proporciona cierta funcionalidad.

Las interfaces existentes son las siguientes y se muestran en la Figura 2-2.

- **GP AXI (General Purpose AXI Ports):** Cuatro puertos de propósito general con un bus de datos de 32 bits cada uno. Dos de ellos son maestros y los dos restantes esclavos.
- **HP AXI (High Performance AXI Ports):** Cuatro puertos de alto rendimiento de 32 o 64 bits de tamaño de bus de datos, donde la PL es el maestro.
- **ACP (Accelerator Coherency Port):** Puerto de 64 bits donde la PL hace de maestro, en el cual existe conexión directa con la SCU de la APU. Este bus permite transacciones con coherencia de caché entre PS y PL.

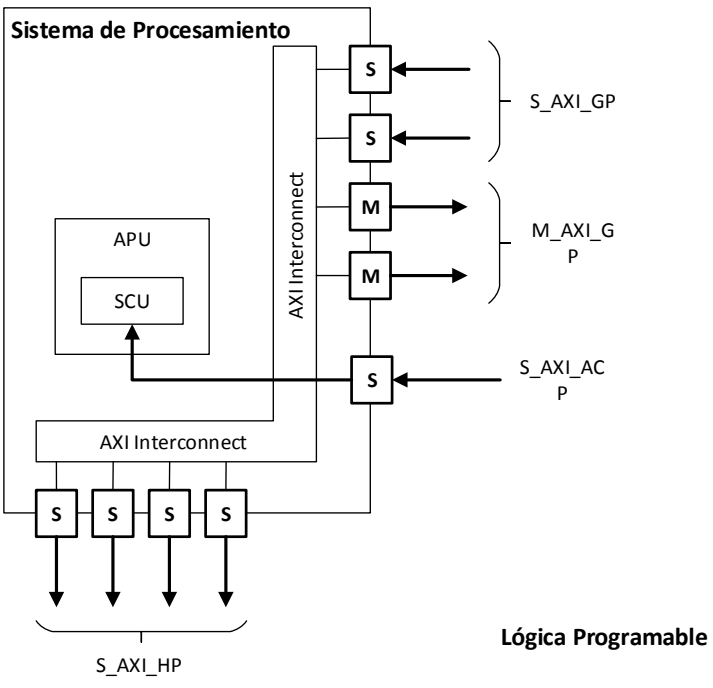

Figura 2-2: Interfaces PS-PL

2.2 ZedBoard

Para probar los diseños realizados durante este proyecto se utilizará la tarjeta ZedBoard. ZedBoard es una placa de evaluación y desarrollo basada en el SoC de Xilinx Zynq-7020, creada en conjunto por Xilinx, Digilent y Avnet, que destaca por su bajo precio y por ser una buena opción para introducirse en el mundo de los SoCs. La tarjeta incorpora gran cantidad de periféricos que facilitan la tarea del desarrollo de una amplia variedad de aplicaciones.

En la Figura 2-3 se puede observar el diseño físico de dicha tarjeta:

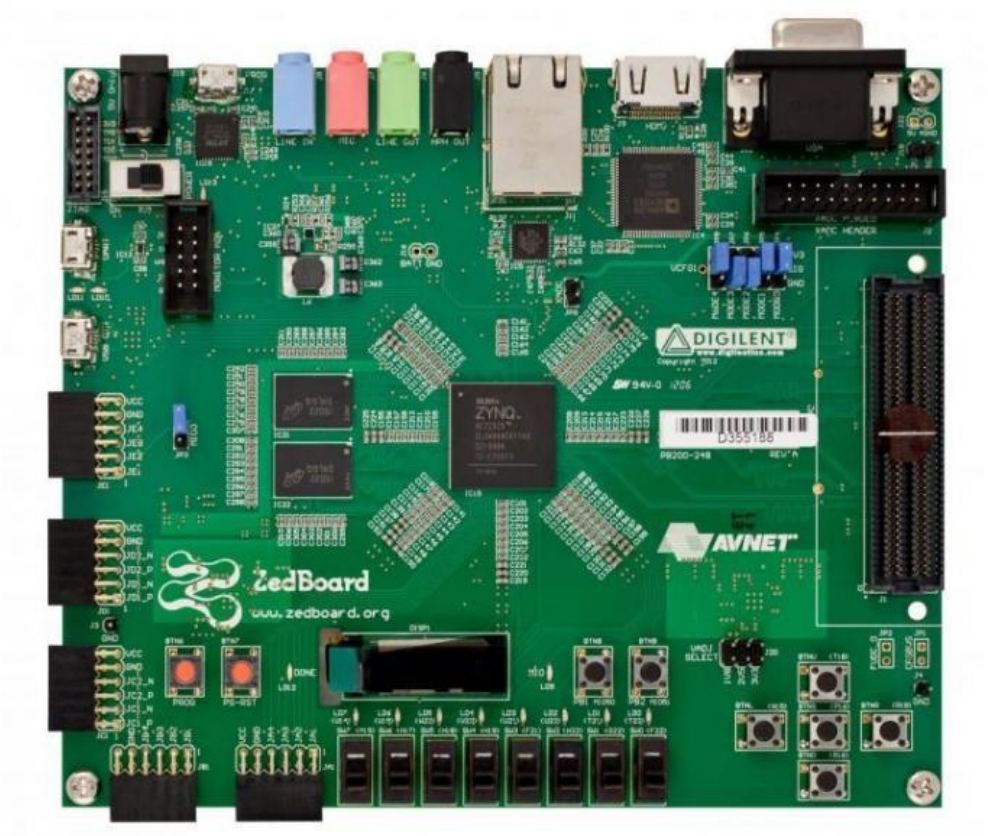

Figura 2-3: Tarjeta de Desarrollo ZedBoard. Obtenida de "zedboard.org".

Las características de ZedBoard son las siguientes:

- SoC: Xilinx Zynq XC7Z020
- Memoria:
  - 512 MB DDR3
  - 256 Mb QSPI Flash
- Osciladores:
  - PS - 33.333 MHz
  - PL – 100 MHz
- Periféricos:
  - GPIO: 9 LEDs, 8 Interruptores, 7 botones

- 10/100/1G Ethernet
- USB-OTG
- USB-JTAG
- USB-UART
- Ranura para tarjeta SD
- HDMI y VGA
- Pantalla OLED de 128x32 pixeles
- Códec de audio con las siguientes entradas/salidas: Line In, Line Out, Entrada de Micrófono y Salida de Auriculares
- 5 Conectores PMOD para tarjetas de expansión
- Conector XADC
- Conector FMC
- Botones de Reset para PS y PL

En la Figura 2-4 se observan todos los periféricos y puertos de dicha tarjeta.

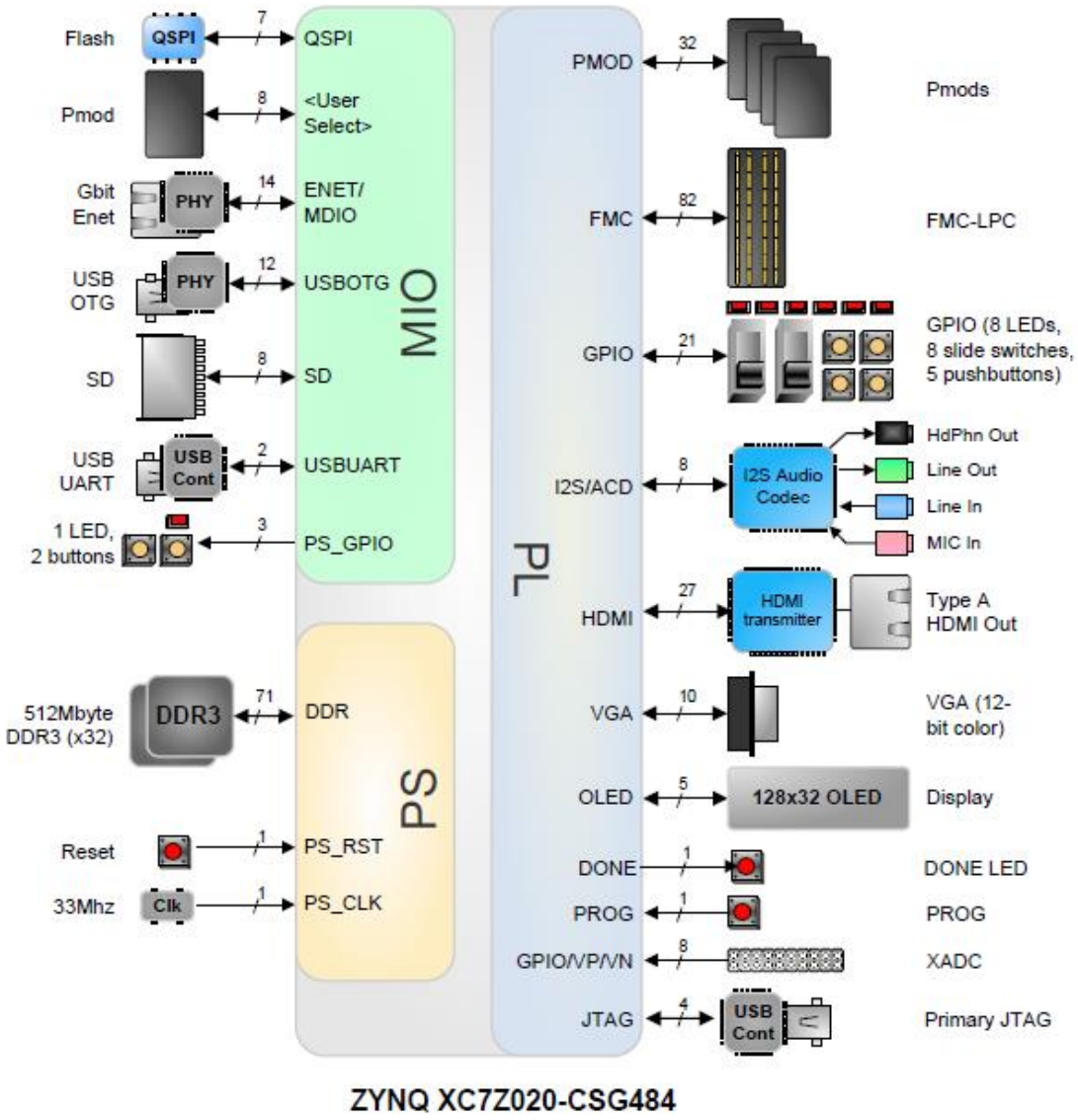

Figura 2-4: Interfaces de la tarjeta ZedBoard. Obtenida de [ZedBoard, 2012].

Para más información sobre la tarjeta de desarrollo ZedBoard consúltese [ZedBoard, 2012].

2.3 Conectividad AXI4

AXI4 es un bus de comunicaciones diseñado por ARM, que constituye la cuarta generación de la especificación AMBA (Advanced Microcontroller Bus Architecture), cuya funcionalidad es la interconexión de bloques funcionales en un SoC.

Este protocolo ha sido adoptado por las herramientas de desarrollo de Xilinx para facilitar la creación de diseños de SoC. El uso de estos buses proporciona varios beneficios, entre ellos una mejora en la productividad de diseño, ya que no es necesario aprender el funcionamiento de diversos protocolos de comunicaciones. Además existe una gran cantidad de IP Cores disponibles para su uso con esta conectividad.

Para ampliar información sobre buses de comunicaciones de un System on Chip consúltese [Pasricha & Dutt., 2008]. Para más información del protocolo AXI4 consúltese [ARM, 2004].

Existen tres tipos de interfaces AXI:

- **AXI4 o AXI4-Full:** Para realizar transferencias mapeadas en memoria con un rendimiento alto, permitiendo enviar ráfagas de datos en una misma transferencia.
- **AXI4-Lite:** Se trata de una simplificación del AXI4 para transferencias mapeadas en memoria, donde tan solo se pueden realizar transferencias simples, es decir, no permite el envío de ráfagas.
- **AXI4-Stream:** Para transferencias tipo streaming de alta velocidad donde un maestro transfiere datos a un esclavo.

2.3.1 Tipos de Interfaces

A continuación se explicará con más detalle el funcionamiento de los distintos tipos de interfaz y se mostrarán ejemplos de su uso. Dichas interfaces emplean diversos canales, los cuales implementan el protocolo de comunicaciones mediante el uso de varias señales de control y datos.

2.3.1.1 Mapeadas en Memoria (MM)

Dentro de este tipo de interfaces, como se explicó anteriormente, se incluyen AXI4 y AXI4-Lite. Estos protocolos requieren el uso de un canal de direcciones que indique la posición de memoria donde se desea realizar una lectura o una escritura. Esto permite la compartición de los canales de este tipo entre varios sistemas, como por ejemplo un

uso típico que consiste en compartir el canal de direcciones, cuya carga es menor que el canal de datos, y utilizar múltiples canales de datos.

Se dispone de los siguientes canales de transmisión:

- **Write Data:** Canal de escritura de datos para realizar una transferencia de datos de un maestro a un esclavo.
- **Read Data:** Canal de lectura de datos para transferir datos de un esclavo a un maestro. Este canal incluye los datos a transferir y una señal que indica el estado de la transferencia.
- **Read Address:** Canal de dirección de lectura. Incluye las señales necesarias para indicar la dirección de lectura de datos, y señales de control.
- **Write Address:** Canal de dirección de escritura. Incluye las señales necesarias para indicar la dirección de escritura de datos, y señales de control.
- **Write Response:** Canal de respuesta a las escrituras que indica el estado de la escritura, es decir, si se ha realizado correctamente o ha habido algún error.

La Figura 2-5 y Figura 2-6 dan una visión general tanto de una transferencia de lectura como una de escritura usando una interfaz mapeada en memoria, donde se pueden observar los cinco canales comentados anteriormente.

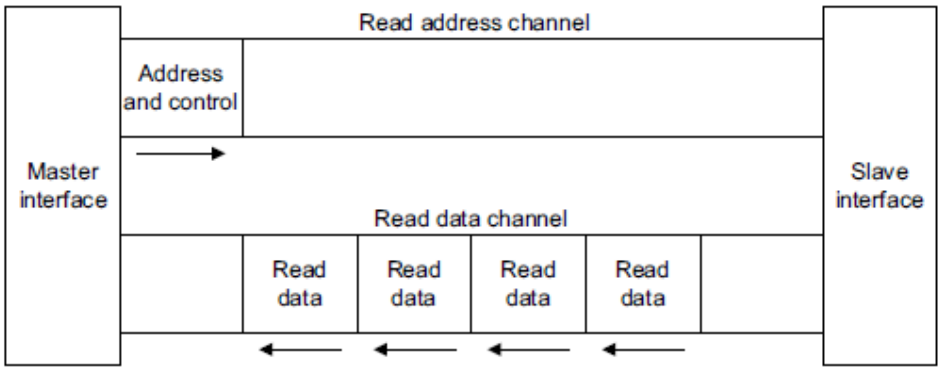

Figura 2-5: Lectura de datos en una interfaz mapeada en memoria

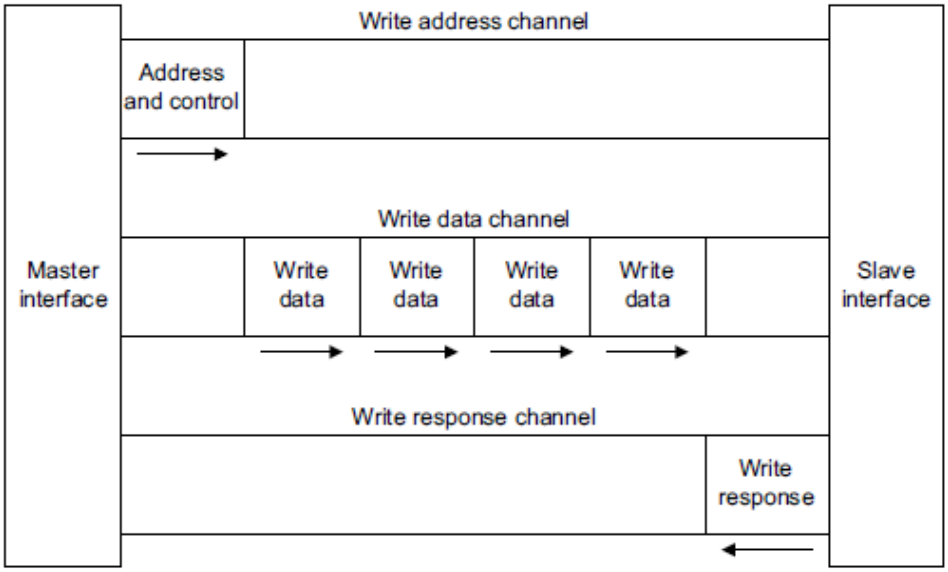

Figura 2-6: Escritura de datos en una interfaz mapeada en memoria

AXI4-Lite es una versión reducida de AXI4, donde se simplifican las señales de los distintos canales, y tan solo se puede realizar una única transferencia con la misma dirección de memoria, es decir, no permite ráfagas de datos. Además el tamaño del bus de datos máximo se reduce, pudiendo ser de tan solo 32 o 64 bits, mientras que en AXI4 puede ser de 32 a 1024 bits. En la Tabla 2-1 se pueden observar las diferencias existentes en las señales empleadas en ambos tipos de interfaz:

Tabla 2-1: Comparación de las señales utilizadas en AXI4 y AXI4-Lite

Write Address			**Read Address**			**Write Data**		
Señal	**AXI4**	**AXI4-Lite**	**Señal**	**AXI4**	**AXI4-Lite**	**Señal**	**AXI4**	**AXI4-Lite**
AWID	✓		ARID	✓		WDATA	✓	✓
AWADDR	✓	✓	ARADDR	✓	✓	WSTRB	✓	✓
AWLEN	✓		ARLEN	✓		WLAST	✓	
AWSIZE	✓		ARSIZE	✓		WUSER	✓	
AWBURST	✓		ARBURST	✓		WVALID	✓	✓
AWLOCK	✓		ARLOCK	✓		WREADY	✓	✓
AWCACHE	✓	✓	ARCACHE	✓	✓			
AWPROT	✓		ARPROT	✓	✓			
AWQOS	✓		ARQOS	✓				
AWREGION	✓		ARREGION	✓				
AWLOCK	✓		ARUSER	✓				
AWUSER	✓		ARVALID	✓	✓			
AWVALID	✓	✓	ARREADY	✓	✓			
AWREADY	✓	✓						

Read Data			Write Response		
Señal	**AXI4**	**AXI4-Lite**	**Señal**	**AXI4**	**AXI4-Lite**
RID	✓		BID	✓	
RDATA	✓	✓	BRESP	✓	✓
RRESP	✓	✓	BUSER	✓	
RLAST	✓		BVALID	✓	✓
RUSER	✓		BREADY	✓	✓
RVALID	✓	✓			
RREADY	✓	✓			

Para más información acerca del uso de este tipo de interfaces, o la funcionalidad de las señales, consúltese [ARM, 2011].

La Figura 2-7 y Figura 2-8 muestran los cronogramas de un ejemplo de escritura y un ejemplo de lectura, respectivamente, con transferencias del tipo AXI4-Lite donde se puede comprobar el funcionamiento de las señales mencionadas.

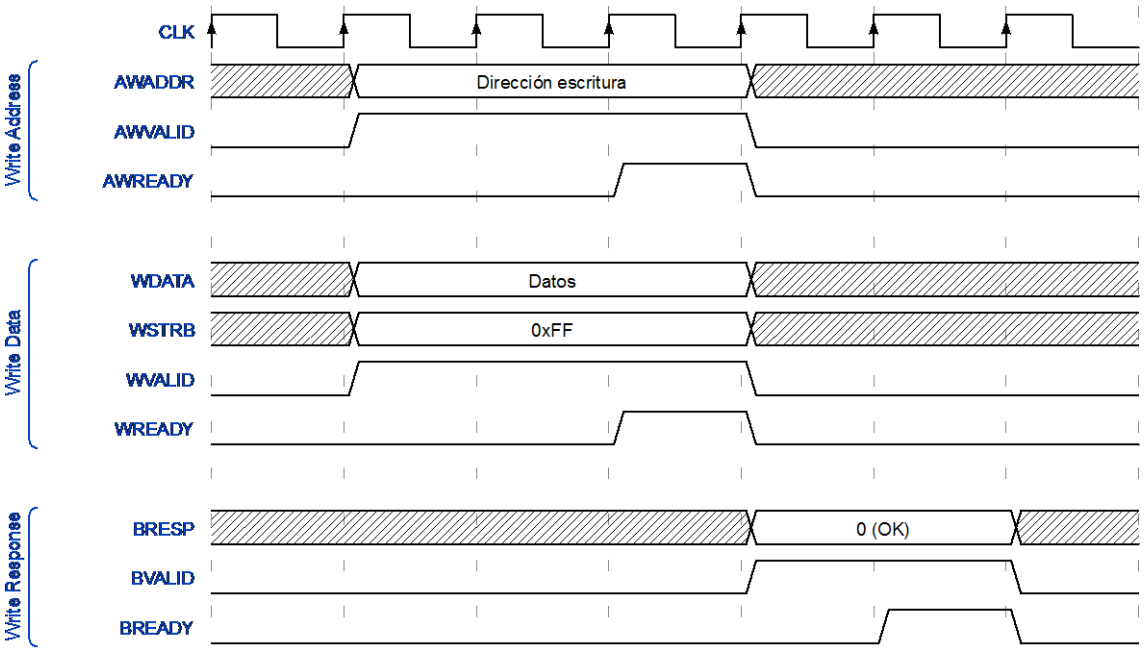

Figura 2-7: Cronograma de una operación de escritura AXI4-Lite

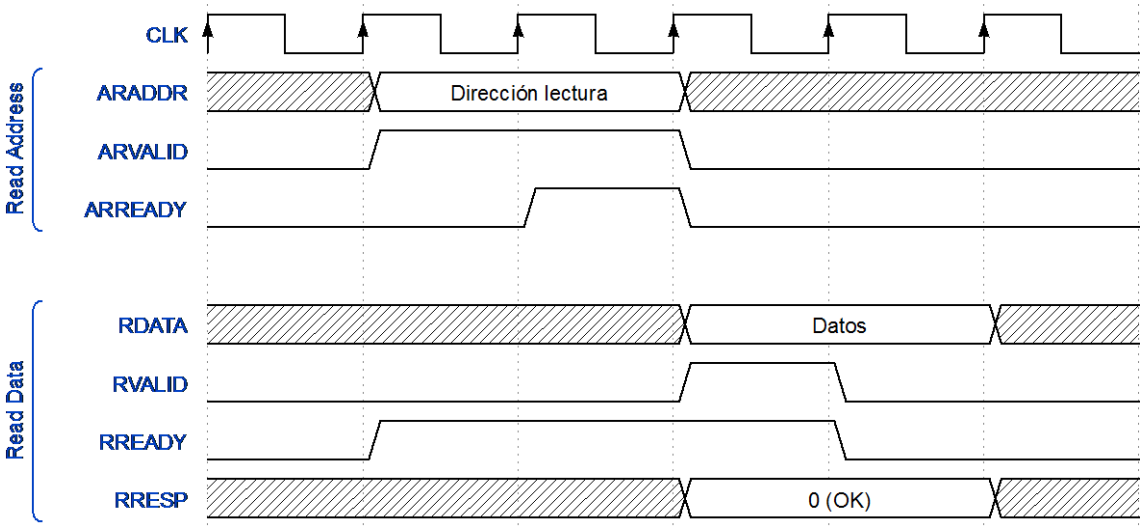

Figura 2-8: Cronograma de una operación de lectura AXI4-Lite

2.3.1.2 AXI4-Stream

AXI4-Stream se diferencia de los protocolos mapeados en memoria principalmente en dos aspectos: Que las transferencias de datos son punto a punto a través de un canal unidireccional, es decir, de un maestro a un esclavo. Y que el canal no puede ser compartido como ocurría en los casos anteriores.

La Figura 2-9 muestra el escenario donde se produce una transferencia del tipo AXI4-Stream:

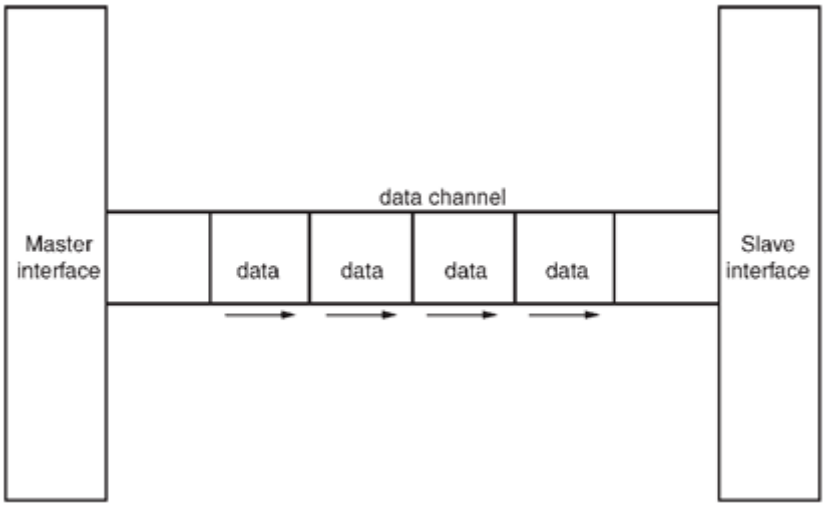

Figura 2-9: Transferencia de datos en una interfaz AXI4-Stream

Las señales utilizadas en este tipo de interfaz se pueden consultar en la Tabla 2-2:

Tabla 2-2: Señales utilizadas en el protocolo AXi4-Stream

Señal	Fuente	Descripción
ACLK	Reloj	Señal de reloj global. Las señales se muestrean en el flanco de subida de ACLK.
ARESETn	Reset	Señal de reset global activa a nivel bajo.
TDATA[(8n-1) : 0]	Master	Bus de datos de entrada.
TVALID	Master	Señal que indica que los datos de entrada son válidos.
TREADY	Slave	Señal que indica que el esclavo puede aceptar un nuevo dato.
TSTRB[(n-1) : 0]	Master	Indica si los bytes de TDATA son bytes de datos (datos válidos) o son bytes de posición (datos no válidos).
TKEEP[(n-1) : 0]	Master	Indica los bytes de TDATA que son válidos y deben ser transferidos.
TLAST	Master	Indica el final del paquete.
TID[(d-1) : 0]	Master	Es el indicador de stream de datos. Cada stream tendrá un valor diferente.
TDEST[(d-1) : 0]	Master	Aporta información de la ruta del stream de datos.
TUSER[(u-1) : 0]	Master	Para enviar información de usuario sobre la transacción

Nota:

- n: Número de bytes del bus de datos
- i: Número de bites de TID
- u: Número de bites de TUSER
- d: Número de bites de TDEST

La Figura 2-10 muestra un cronograma de ejemplos de transferencia del tipo AXI4-Stream donde se puede comprobar el funcionamiento de las señales anteriormente descritas. Se puede observar el caso donde se tiene un dato válido pero el esclavo no está disponible para recibir un dato en el Dato 3, o el caso donde no se dispone de un dato válido en el Dato 4. El Dato 5 indica el final de paquete, donde además el dato no tiene todos sus bytes válidos.

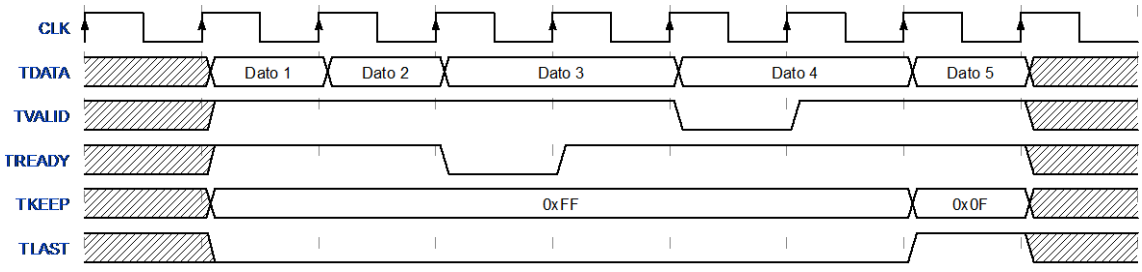

Figura 2-10: Cronograma de una transferencia AXI4-Stream

Para más información sobre el protocolo AXI4-Stream consúltese [ARM, 2010].

2.3.2 Proceso de Negociación (Handshaking)

En las transferencias de datos del protocolo AXI4, las señales TVALID y TREADY son utilizadas para el proceso de negociación como método de control del flujo de datos. Como se ha visto en el apartado anterior, estas señales son utilizadas en todos los canales de AXI4, AXI4-Lite y AXI4-Stream.

La señal TVALID es generada por el maestro cuando los datos que se transmiten por el bus TDATA son válidos, de forma que el esclavo pueda leerlos. El esclavo utiliza la señal TREADY para indicar al maestro que ha terminado de leer los datos anteriores y está disponible para recibir una nueva transferencia.

Una transferencia tiene lugar cuanto las señales TVALID y TREADY están activas en el mismo flanco de reloj. Se puede dar el caso de que las dos señales se activen a la vez, o que primero se active TVALID o TREADY.

La Figura 2-11 muestra un ejemplo de uso de estas dos señales. Una vez que el maestro activa la señal TVALID indicando que el dato es válido, no debe cambiar los datos o señales de control hasta que el esclavo active la señal TREADY.

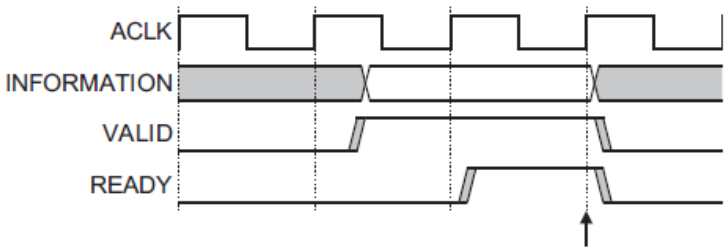

Figura 2-11: Proceso de negociación en AXI4. Obtenida de [ARM, 2004].

2.4 Herramientas de desarrollo

2.4.1 Xilinx Vivado Design Suite

Vivado Design Suite es un entorno de desarrollo integrado (IDE) que sustituye a las herramientas de la suite Xilinx ISE (ChipScope, XST, Coregen, PlanAhead, Simulator...), incorporándolas todas en una misma interfaz gráfica, permitiendo mejorar la productividad en los diseños, ya que el cambio de una herramienta a otra se realiza rápidamente y dentro del mismo programa.

Esta suite permite realizar diseños en FPGA y SoCs para las familias de la serie 7 de Xilinx: Ultrascale, Virtex-7, Kintex-7, Artix-7, y Zynq -7000. Además permite implementarlos fácilmente mediante la interconexión de IP Cores.

Vivado proporciona una interfaz gráfica sencilla, muy útil para nuevos usuarios, que permite la creación de diseños complejos de forma visual. Se pueden crear arquitecturas de SoC fácilmente con IP Integrator mediante la configuración e interconexión de distintos bloques funcionales (IPs) previamente creados, tanto los creados por Xilinx, como los creados por el usuario. Para más información sobre la creación de diseños con IP Integrator consúltese [Xilinx, 2013].

La Figura 2-12 muestra el entorno de desarrollo de Vivado Design Suite.

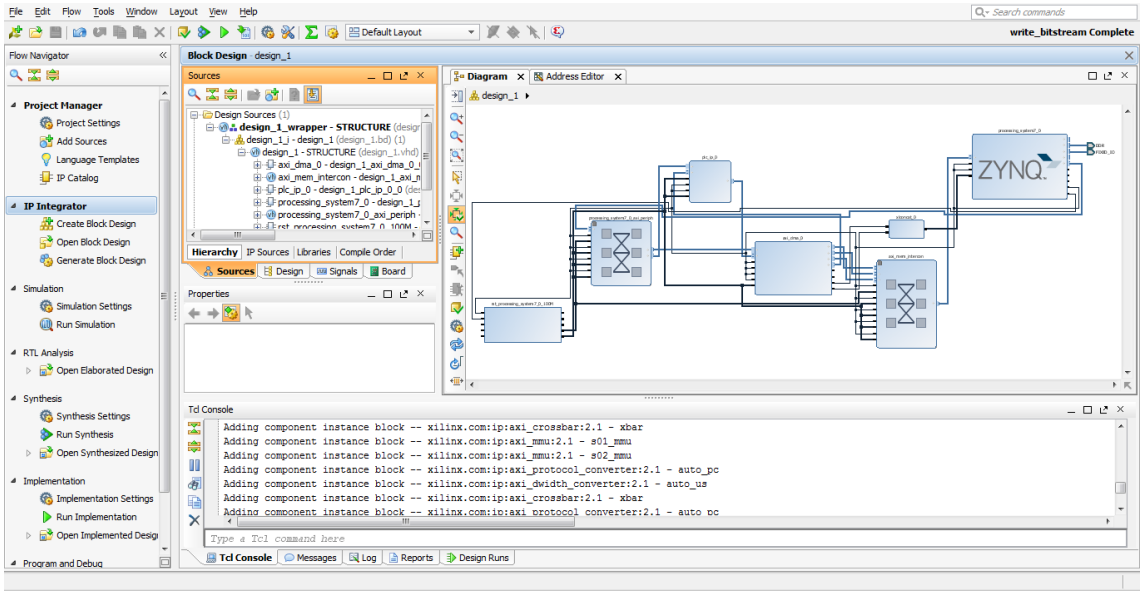

Figura 2-12: Vivado Design Suite

Por otra parte se dispone de IP Packager, otro componente de Vivado que proporciona la capacidad de empaquetar un diseño completo con la finalidad de que éste pueda ser reutilizado en un futuro. Se hablará más en detalle acerca del IP Packager en el apartado 3.2, donde el periférico del cliente DMA creado es empaquetado.

2.4.2 ModelSim

ModelSim es un simulador de hardware descrito mediante lenguajes HDL, creado por Mentor Graphics, que permite la verificación de diseños para FPGA. Soporta el uso de los siguientes lenguajes de descripción de hardware: VHDL, Verilog, Verilog 2001, SystemVerilog y SystemC.

Gracias a este simulador, se podrán crear bancos de prueba (testbench) que simulen el comportamiento de los diseños realizados. En los ficheros de testbench se llevará a cabo el control de las distintas señales que intervienen en el diseño y con ModelSim se comprobará si el funcionamiento es correcto o no observando en el diagrama de tiempos los resultados obtenidos ante dichos estímulos.

La Figura 2-13 muestra el entorno de simulación de ModelSim.

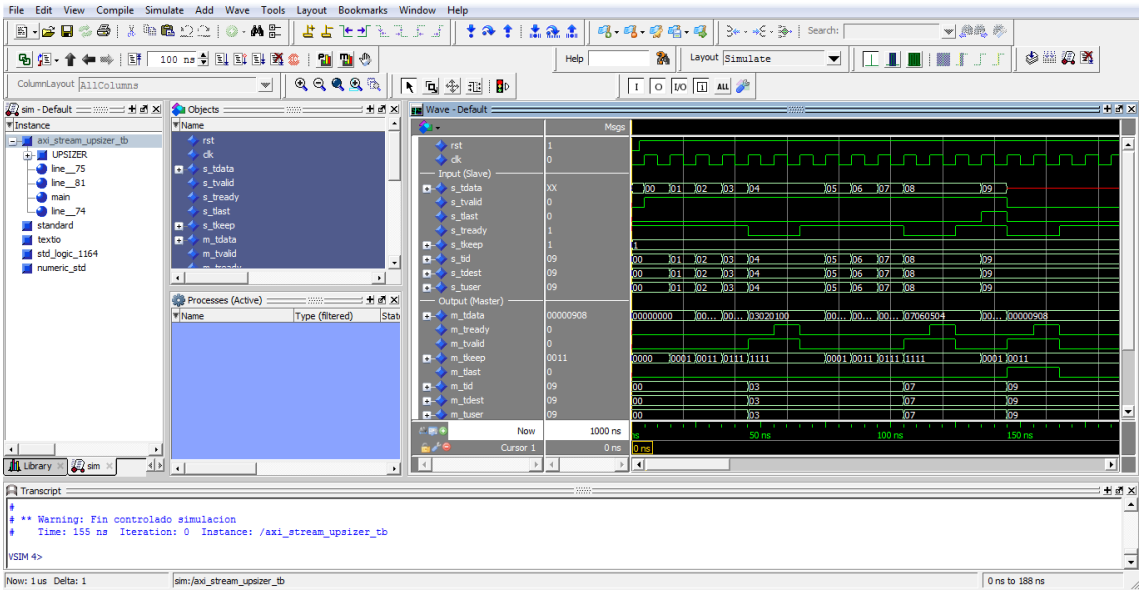

Figura 2-13: Entorno de trabajo de ModelSim

2.5 AXI DMA

AXI DMA es un IP Core proporcionado por Xilinx que permite realizar transferencias de datos entre memoria, a través de una interfaz AXI4 Memory Mapped, y periféricos con interfaz AXI4-Stream.

Uno de los motivos del empleo del DMA para realizar un movimiento de datos es evitar la sobrecarga del procesador en dicho proceso. El DMA de Xilinx proporciona diversos modos de funcionamiento, algunos de ellos son: Micro DMA, multicanal, transferencias en dos dimensiones, etc. Sin embargo, en este trabajo se centrará la atención en el empleo del modo Scatter/Gather, que permite reducir la carga del procesador cuando hay que enviar gran cantidad de datos o éstos están dispersos en memoria.

La Figura 2-14 muestra la estructura interna del IP del DMA.

En este apartado se llevará a cabo una introducción al DMA y su funcionamiento. Para más información consúltese [Xilinx, 2014b].

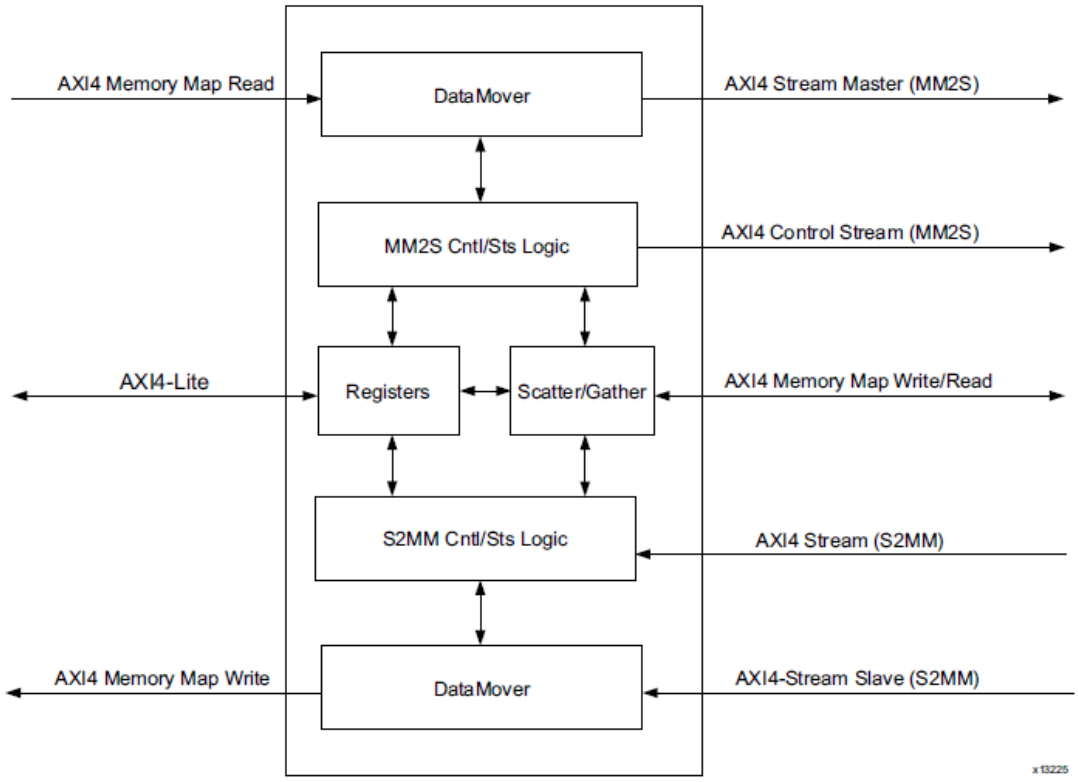

Figura 2-14: Estructura del DMA. Obtenida de [Xilinx, 2014b].

Para una mejor comprensión del DMA, se definirán dos tipos de canales que se identificarán mediante los acrónimos MM2S y S2MM. Estos canales estarán asociados a los conceptos de transmisión y recepción como se explica a continuación:

- **MM2S**: Canal de transmisión donde los datos procedentes de la memoria (a través de una interfaz mapeada en memoria) se transmiten como un stream de datos hacia la PL
- **S2MM**: Canal de recepción donde los datos procedentes de la PL en forma de stream se transmiten a memoria a través de una interfaz mapeada en memoria.

Tal como se muestra en la Figura 2-14, algunas de las interfaces están formadas por varios canales. Estas interfaces son:

- **Interfaz MM2S:** Formada por un canal de datos y un canal de control, ambos AXI4-Stream.
    - **AXI4-Stream Master:** Canal por el que se transmiten los paquetes de datos procedentes de la memoria, en formato stream.
    - **AXI4 Control Stream:** Canal por el que se envía información de control y datos de usuario al cliente DMA en la PL.

- **Interfaz S2MM**: Formada por un canal de datos y un canal de control, ambos AXI4-Stream.
    - **AXI4-Stream Slave:** Canal de por el que se recibe un stream de datos procedente de la PL que se desea enviar a memoria.
    - **AXI4 Stream (Control):** Canal de tipo stream por el que se recibe información de control e información de usuario procedentes del cliente DMA.
- **Interfaz AXI-Lite:** Canal destinado a la configuración de los registros del DMA.
- **Interfaz AXI Scatter/Gather:** Encargada de la configuración de registros específicos del modo de funcionamiento Scatter/Gather.
- **Interfaces AXI4 Memory Map:** Canales de lectura y escritura destinados a la comunicación entre el DMA y la memoria.

2.5.1 Modo de funcionamiento Scatter/Gather

El DMA puede configurarse en varios modos de funcionamiento, dependiendo de las necesidades de la aplicación. El modo de funcionamiento simple permite realizar transferencias de datos entre la memoria y los periféricos o viceversa de datos contiguos ubicados en una dirección de memoria. Sin embargo, es común que los datos no aparezcan contiguos en memoria, sino que están dispersos en diferentes posiciones.

Para solucionar esto se emplea el método Scatter/Gather, el cual ofrece la opción de transmitir paquetes de datos que se encuentran en distintas zonas de memoria (Gather = Recolectar), o por el contrario, recibir paquetes de datos que se almacenarán en distintas zonas de memoria (Scatter = Dispersar).

Para este modo de funcionamiento cobran importancia los descriptores, llamados Buffer Descriptors (BD). Los BD son estructuras de datos que están enlazadas entre sí como una lista circular, que aportan información de los datos que hay que transmitir o recibir.

Por ejemplo, en el caso de un descriptor de transmisión (canal MM2S) se indica la dirección de memoria donde se encuentran los datos que se desean transmitir, el tamaño de la transmisión y puntero al siguiente descriptor, entre otros. Igualmente en el caso del descriptor de recepción (canal S2MM), se dispone de la dirección donde se desean almacenar los datos del stream de datos recibido y su tamaño.

En el caso expuesto anteriormente, donde un dato se encuentra disperso en la memoria, este dato estará descrito por más de un BD. Para ello, cobran gran importancia los campos SOF (Start of frame) y EOF (End of frame) de los descriptores.

El bit SOF indica que los datos a procesar de dicho descriptor suponen el inicio del paquete. El bit EOF informa de que dicho descriptor marca el final del paquete. En el caso de que se desee mandar tan solo un paquete de datos definido en un descriptor, ambos bits aparecerán activos simultáneamente.

2.5.2 Descritores Scatter/Gather

A continuación se describirá la estructura de un descriptor Scatter/Gather tanto para el canal de transmisión (MM2S) como para el de recepción (S2MM). Un descriptor está formado por 13 palabras de 32 bits, cuyos campos se pueden observar en la Figura 2-15:

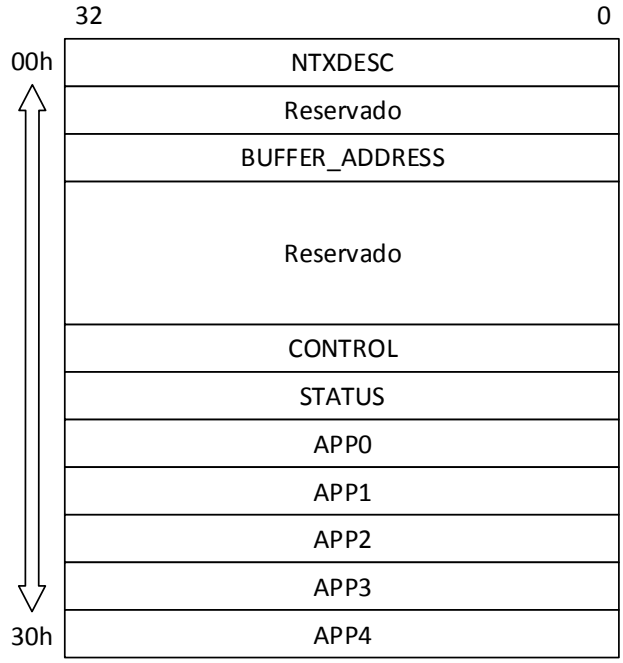

Figura 2-15: Estructura de un descriptor Scatter/Gather

La funcionalidad de cada uno de los campos se explicará a continuación tanto para el caso de un descriptor MM2S como para uno S2MM.

NTXDESC: Puntero que apunta al inicio del siguiente BD del anillo de descriptores.

BUFFER_ADDRESS: Puntero que en el caso del descriptor MM2S apunta a los datos a transferir, y en el caso de S2MM apunta a la dirección de memoria disponible para almacenar los datos recibidos.

CONTROL: Registro de control de las transferencias entre PS y PL que permite la siguiente configuración:

- **Tamaño del Buffer:** Indica el tamaño en número de bytes del dato a transmitir por el canal MM2S o del tamaño disponible para recibir datos por el canal S2MM.
- **SOF:** Inicio de paquete. Bit que indica que el descriptor actual es el inicio de un paquete de datos a transmitir o recibir.
- **EOF:** Fin de paquete. Bit que indica que el descriptor actual es el final de un paquete de datos a transmitir o recibir.

STATUS: Registro de estado que proporciona la siguiente información sobre los canales del DMA.

- **Bytes Transferidos:** Para un descriptor MM2S indica el número de bytes que han sido enviados, este valor debe coincidir con el tamaño del buffer visto anteriormente. Para un descriptor S2MM indica el número de bytes que han sido recibidos.
- **RXEOF:** Solo disponible en los descriptores del canal S2MM. Indica que dicho descriptor es el último de un paquete.
- **RXSOF:** Solo disponible en los descriptores del canal S2MM. Indica que dicho descriptor es el primero de un paquete.
- **DMAIntErr:** Error interno del DMA que ocurre cuando el tamaño del buffer especificado es de 0 bytes.
- **DMASlvErr:** Error que ocurre cuando la lectura de la interfaz MM realizada por un esclavo provoca un error.
- **DMADecErr:** Error de decodificación que se produce cuando la dirección del puntero BUFFER_ADDRESS no es válida.
- **Completed:** Bit que indica si la transferencia de dicho descriptor se ha completado.

APP0 – APP4: Campos de información de usuario. En un descriptor MM2S los datos de estos campos son transmitidos al bus AXI4 Control Stream cuando SOF=1. En un descriptor S2MM la información del bus AXI4 Status Stream se almacena en los campos APP0 – APP3 cuando EOF=1. El campo APP4 en un descriptor S2MM puede funcionar como los otros, o para indicar los bytes recibidos de un paquete.

2.6 Accelerator Coherency Port (ACP)

El Accelerator Coherency Port (ACP) es una interfaz esclava de 64 bits de bus de datos conectada directamente a la SCU que proporciona accesos a la memoria con coherencia de caché.

En los diseños de SoCs basados en FPGA es común tener que realizar transferencias de datos desde el PS hacia la PL, para ello se suele utilizar el DMA. Al ser el procesador quien modifica los datos a transferir, es posible que los datos existentes en la memoria caché no se correspondan con los que hay almacenados en la memoria del dispositivo, luego los datos no son coherentes. Es aquí donde cobra importancia el uso del ACP para asegurar la coherencia de datos entre ambas memorias.

Además de la ventaja de tener accesos coherentes a la memoria, el uso del ACP mejora el rendimiento del sistema y la energía consumida disminuye. Esto es debido a que al mantener la coherencia de caché, no es necesario realizar limpiados de memoria (flush), siendo más eficiente.

En la Figura 2-16 se muestra la conexión entre la SCU, las memorias caché y de dispositivo y el puerto ACP.

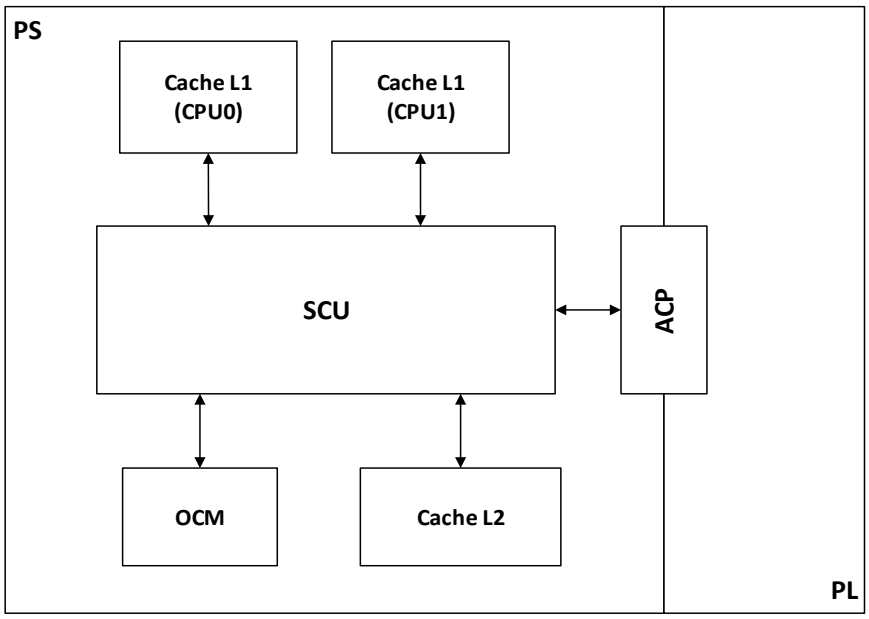

Figura 2-16: Conexión del ACP a la SCU

El procedimiento del ACP para realizar una lectura de datos de la memoria de forma coherente sigue los siguientes pasos. Para ello es necesario que las siguientes señales de la interfaz AXI tengan los valores: **ARUSER[0]** = 1 y **ARCACHE[1]** = 1.

1. Comprobar las memorias caché de nivel 1 del procesador.
2. Si los datos que se desean leer no se encuentran en dichas memorias, se comprueba la memoria caché de nivel 2.
3. Si tampoco están los datos en esta memoria caché se realiza una lectura de la memoria del dispositivo.

Para el proceso de escritura en memoria de forma coherente, también es necesario **ARUSER[0]** = 1 y **ARCACHE[1]** = 1. El procedimiento es el siguiente:

1. Si los datos se encuentran en la memoria caché del procesador, primero se invalidan dichos datos.
2. Una vez invalidados los datos (o en el caso de que no existiesen) se realiza la escritura.

En la nota de aplicación [Bagni et al., 2013], se puede observar el empleo del puerto ACP en una aplicación realizada para Zynq.

2.7 Modelo Funcional de Bus (BFM)

En la simulación de periféricos implementados en la PL y conectados al PS mediante buses de comunicaciones, es necesario simular el comportamiento del procesador, ya que es éste el encargado de realizar las transferencias tanto de lectura como de escritura. Modelar el comportamiento del procesador puede ser complicado, debido a que no existe su modelo, o bien lleve mucho tiempo realizar la simulación incluyéndolo debido a su complejidad. La solución a estos problemas se soluciona mediante el empleo del BFM.

El BFM (Modelo funcional de bus) es un componente capaz de modelar el comportamiento de los buses de comunicaciones del procesador, de forma que no sea necesario simular el modelo del procesador completo. El BFM consta de una máquina de estados encargada de realizar los ciclos de bus para generar los estímulos en la interfaz, y de una API que facilita la configuración del BFM y la generación de las operaciones deseadas.

El BFM a utilizar en este proyecto es el proporcionado por Xilinx, que permite la simulación de los buses AXI4, AXI4-Lite, AXI4-Stream y AXI3. Xilinx también proporciona varios ejemplos de uso con los distintos protocolos de bus, en los que se incluyen tareas que permiten comparar los estímulos recibidos con los esperados, de forma que se genere un mensaje de error en el caso en el que no coincidan.

La arquitectura del IP Core AXI BFM consta de cuatro capas fundamentales, como se muestra en la Figura 2-17:

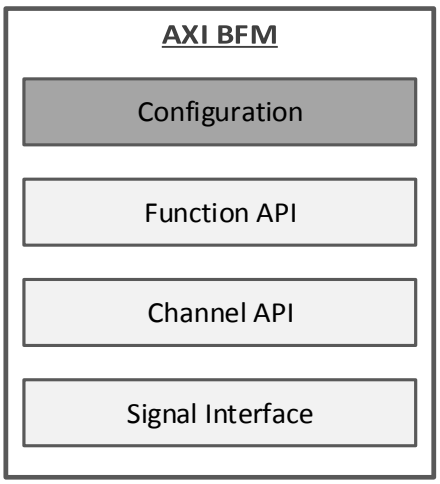

Figura 2-17: Estructura de capas del BFM. Adaptada de [Xilinx, 2014a].

- **Signal Interface:** Define los diferentes puertos de entrada y salida del core BFM.
- **Channel API:** Constituido por un conjunto de funciones que definen las transferencias de datos básicas del protocolo AXI.

- **Function API:** Esta capa dispone de un control completo a nivel de transacciones. Por ejemplo una función para enviar una ráfaga de datos, que empleará funciones de la capa inferior para ello.
- **Configuration:** API que proporciona las funciones necesarias para realizar la configuración de los cores BFM.

Las funciones de las API comentadas anteriormente están escritas en SystemVerilog, ya que los cores BFM están implementados en este lenguaje. Debido a esto, el testbench y la instanciación de componentes debe hacerse también en este lenguaje.

2.7.1 Parámetros BFM

El IP core AXI BFM dispone de varios parámetros de configuración, que se podrán especificar una vez instanciado el componente en el diseño de bloques de Vivado. Sin embargo, estos parámetros pueden ser modificados durante la simulación mediante el empleo de funciones de la "Utility API". Consúltese el apartado Utility API Tasks/Functions de la guía de uso de AXI BFM Core [Xilinx, 2014a] para más información.

Existen diferentes parámetros dependiendo del protocolo de comunicaciones utilizado (AXI4, AXI4-Lite, AXI4-Stream, AXI3). A continuación se explicarán aquellos que se utilizarán en la simulación del BFM AXI4-Stream, que será la que se llevará a cabo en el presente trabajo.

Consúltese [Xilinx, 2014a] para más información sobre los parámetros de otros tipos de buses.

2.7.1.1 Parámetros AXI4-Stream BFM Maestro

En la Tabla 2-3 se describen los distintos parámetros de configuración que pueden ser utilizados en un BFM Maestro.

Tabla 2-3: Parámetros del BFM AXI4-Stream Maestro

Parámetro	Descripción	Por defecto
NAME	Nombre del BFM utilizado en los mensajes provenientes de él.	"MASTER_0"
DATA_BUS_WIDTH	Tamaño del bus de datos en número de bits.	32
ID_BUS_WIDTH	Tamaño del bus TID en número de bits.	8
DEST_BUS_WIDTH	Tamaño del bus TDEST en número de bits.	4
USER_BUS_WIDTH	Tamaño del bus TUSER en número de bits.	8
MAX_PACKET_SIZE	Tamaño máximo de un paquete. Indica el número de transferencias de tamaño DATA_BUS_WIDTH que caben en un paquete.	10

MAX_OUTSTANDING_TRANSACTIONS	Número máximo de transferencias pendientes. No se podrá generar más tráfico si se ha alcanzado el límite, a no ser que se acabe alguna transferencia pendiente.	8
STROBE_NOT_USED	Deshabilita el uso de la señal TSTRB.	0 (Habilitado)
KEEP_NOT_USED	Deshabilita el uso de la señal TKEEP.	0 (Habilitado)
PACKET_TRANSFER_GAP	Número de ciclos de reloj de espera entre transferencias de un paquete.	0
RESPONSE_TIMEOUT	Número de ciclos de reloj en el que se considera una transferencia como perdida.	500
STOP_ON_ERROR	Si está habilitada y ocurre un error se parará la simulación.	1 (Habilitada)
CHANNEL_LEVEL_INFO	Habilita la opción de imprimir mensajes de información a nivel de canal.	1 (Habilitado)
CLEAR_SIGNALS_AFTER_HANDSHAKE	Si está activa se resetearán las señales de salida entre transferencias.	0 (Inactivo)
INPUT_SIGNAL_DELAY	Usado para retrasar las señales de entrada respecto al reloj.	0
TASK_RESET_HANDLING	0: Ignorar reset y continuar procesando tarea. 1: Esperar hasta el final del reset. 2: Informar de error y parar. 3: Avisar y continuar.	0

2.7.1.2 Parámetros AXI4-Stream BFM Esclavo

En la Tabla 2-4 se describen los distintos parámetros de configuración que pueden ser utilizados en un BFM Maestro.

Tabla 2-4: Parámetros del BFM AXI4-Stream Esclavo

Parámetro	Descripción	Por defecto
NAME	Nombre del BFM utilizado en los mensajes provenientes de él.	"SLAVE_0"
DATA_BUS_WIDTH	Tamaño del bus de datos en número de bits.	32
ID_BUS_WIDTH	Tamaño del bus TID en número de bits.	8
DEST_BUS_WIDTH	Tamaño del bus TDEST en número de bits.	4
USER_BUS_WIDTH	Tamaño del bus TUSER en número de bits.	8

MAX_PACKET_SIZE	Tamaño máximo de un paquete. Indica el número de transferencias de tamaño DATA_BUS_WIDTH que caben en un paquete.	10
MAX_OUTSTANDING_TRANSACTIONS	Número máximo de transferencias pendientes. No se podrá generar más tráfico si se ha alcanzado el límite, a no ser que se acabe alguna transferencia pendiente.	8
STROBE_NOT_USED	Deshabilita el uso de la señal TSTRB.	0 (Habilitado)
KEEP_NOT_USED	Deshabilita el uso de la señal TKEEP.	0 (Habilitado)
RESPONSE_TIMEOUT	Número de ciclos de reloj en el que se considera una transferencia como perdida.	500
DISABLE_RESET_VALUE_CHECKS	Deshabilita la comprobación de los valores correctos de las señales de entrada tras el reset.	0 (Inactivo)
STOP_ON_ERROR	Si está habilitada y ocurre un error se parará la simulación.	1 (Habilitada)
CHANNEL_LEVEL_INFO	Habilita la opción de imprimir mensajes de información a nivel de canal.	1 (Habilitado)
INPUT_SIGNAL_DELAY	Usado para retrasar las señales de entrada respecto al reloj.	0
TASK_RESET_HANDLING	0: Ignorar reset y continuar procesando tarea. 1: Esperar hasta el final del reset. 2: Informar de error y parar. 3: Avisar y continuar.	0

2.7.2 Funciones de la API para realizar transferencias

Para controlar las transacciones tanto de entrada como de salida en los cores BFM se utilizarán funciones incluidas en las API de las capas Function y Channel. Estas últimas permiten enviar o recibir una transferencia de datos, mientras que las funciones de la Function API se encuentran un nivel por encima y utilizan las funciones de la Channel API para realizar transferencias de ráfagas o paquetes.

Existen diferentes funciones dependiendo del protocolo de comunicaciones utilizado (AXI4, AXI4-Lite, AXI4-Stream, AXI3). Las funciones básicas que se utilizarán en la simulación del BFM AXI4-Stream se explicarán a continuación. Para más información sobre las funciones específicas de otros tipos de bus consúltese [Xilinx, 2014a].

Las funciones de la Channel API específicas para el bus AXI4-Stream serán las encargadas de realizar transferencias de datos y paquetes en el caso del BFM maestro, y lecturas de datos y paquetes en el BFM esclavo. Estas funciones son las siguientes:

2.7.2.1 Funciones AXI4-Stream BFM Maestro

En la Tabla 2-5 se detallan las distintas funciones de la API cuya función es el envío de una transferencia en un BFM maestro.

Tabla 2-5: Funciones del BFM AXI4-Stream Maestro

Función	**Definición**	
SEND_TRANSFER	Crea una transferencia simple con los datos y parámetros indicados como entrada a dicha función.	
	Parámetros de Entrada	**Parámetros de Salida**
	- ID - DEST - DATA - STRB - KEEP - LAST - USER	Ninguno

Función	**Definición**	
SEND_PACKET	Envía un paquete de datos empleando para ello varias llamadas a la función SEND_TRANSFER	
	Parámetros de Entrada	**Parámetros de Salida**
	- ID - DEST - DATA: Vector de datos. - DATASIZE: Tamaño en bytes del vector de datos. - USER	Ninguno

2.7.2.2 Funciones AXI4-Stream BFM Esclavo

En la Tabla 2-6 se detallan las distintas funciones de la API cuya función es la recepción de una transferencia en un BFM esclavo.

Tabla 2-6: Funciones del BFM AXI4-Stream Esclavo

<table>
<tr><th>Función</th><th colspan="2">Definición</th></tr>
<tr><td rowspan="3">RECEIVE_TRANSFER</td><td colspan="2">Recibe una transferencia de datos especificada por los parámetros de entrada ID y DEST si están validados por IDValid y DESTValid. Si estos últimos dos parámetros son 0, se tomará la primera transferencia recibida.</td></tr>
<tr><td>Parámetros de Entrada</td><td>Parámetros de Salida</td></tr>
<tr><td>- ID
- IDValid: Bit que indica si el parámetro ID debe ser usado.
- DEST
- DESTValid: Bit que indica si el parámetro DEST debe ser usado.</td><td>- ID
- DEST
- DATA
- STRB
- KEEP
- LAST
- USER</td></tr>
</table>

<table>
<tr><th>Función</th><th colspan="2">Definición</th></tr>
<tr><td rowspan="3">RECEIVE_PACKET</td><td colspan="2">Recibe un paquete de datos compuesto por varias transferencias, por tanto utiliza la función RECEIVE_TRANSFER. Si los parámetros de entrada IDValid y DESTValid son 0, se ignoran los valores de ID y DEST y se tomará el primer paquete recibido.</td></tr>
<tr><td>Parámetros de Entrada</td><td>Parámetros de Salida</td></tr>
<tr><td>- IDValid: Bit que indica si el parámetro ID debe ser usado.
- DEST
- DESTValid: Bit que indica si el parámetro DEST debe ser usado.</td><td>- PID
- PDEST
- DATA: Datos del paquete.
- DATASIZE: Tamaño de los datos válidos del paquete en bytes.
- USER</td></tr>
</table>

2.8 LwIP

LwIP (Lightweight IP) es una implementación de la pila de protocolos TCP/IP creada especialmente para sistemas empotrados, ya que los recursos necesarios para su uso son muy reducidos, necesitando muy poca memoria y poco código para hacerla funcionar. Esta pila de protocolos está disponible como software libre, y se puede utilizar en variedad de plataformas, con la opción de utilizar sistema operativo o no.

A continuación se introducirán las características básicas de LwIP. Para ampliar dicha información consúltese [LwIP, 2004] o [Dunkels, 2001].

LwIP está estructurado en varias capas, donde cada una implementa un protocolo de comunicaciones. Sin embargo, en algunas de ellas se comparten datos con otras capas para mejorar el rendimiento tanto en procesamiento como en recursos de memoria, pues es posible la reutilización de buffers. Además, se puede especificar el contenido de éstos mediante punteros a memoria, de forma que no haga falta realizar copias de datos, y por tanto reduciendo los recursos de memoria utilizados.

LwIP utiliza varios tipos de buffers para su manejo. Uno de los buffers utilizados es el PBUF (Packet Data Buffer), una estructura que representa a un paquete de datos. Los PBUFs pueden estar enlazados en forma de cadena, permitiendo que un paquete pueda utilizar varios PBUFs, como se muestra en la Figura 2-18.

La estructura de este buffer consta de los campos mostrados en la Tabla 2-7.

Tabla 2-7: Estructura de un PBUF

Campo	Descripción
Next	Puntero al siguiente PBUF, en el caso en el que exista una cadena de PBUFs.
Payload	Puntero que indica la dirección en memoria donde comienzan los datos del PBUF. Existen distintos tipos de PBUFs dependiendo de la posición en memoria donde se encuentren los datos del paquete.
Len	Tamaño de los datos del PBUF.
Tot_Len	Suma de todos los campos Len de los PBUFs de la cadena.
Flags	Indican el tipo de PBUF.
Ref	Contador que indica el número de punteros que apuntan a dicho PBUF.

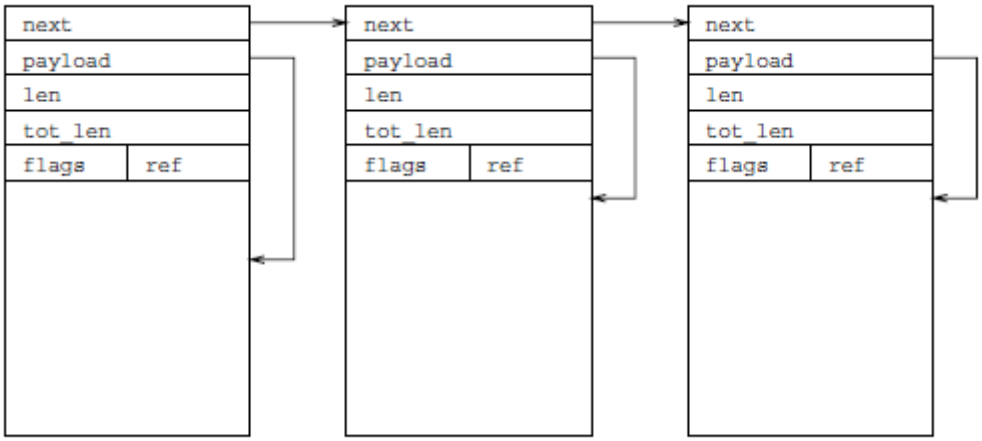

Figura 2-18: Ejemplo de cadena de PBUFs. Obtenida de [Dunkels, 2001].

Las interfaces de red del dispositivo también son representadas por una estructura en LwIP llamada netif. Al igual que con los PBUFs, esta estructura también se almacena como una lista enlazada cuyos elementos representan a diferentes interfaces. Los campos de esta estructura se indican en la Tabla 2-8.

Tabla 2-8: Estructura de un buffer netif

Campo	Descripción
Next	Puntero al siguiente elemento de la cadena de netif.
Name	Nombre de la interfaz.
Num	Número utilizado para diferenciar varias interfaces con el mismo nombre.
IP_addr	Dirección IP fija de la interfaz.
Netmask	Máscara de red.
GW	Puerta de enlace.
Input	Puntero que indica la función que debe ser llamada tras recibir un nuevo paquete.
Output	Puntero que indica la función del driver del dispositivo que transmite un paquete.
State	Estado de la interfaz de red.

Otra de las estructuras que se utilizan en LwIP es la llamada PCB (Bloque de Control de Protocolo), cuya función es definir los parámetros de la sesión. Más adelante, en el diseño software de la aplicación, se comentarán los parámetros de la PCB específica para el protocolo UDP.

3 DISEÑO HARDWARE

Una vez introducidos los conceptos teóricos básicos necesarios para la realización del proyecto, se procederá a explicar detalladamente los distintos pasos llevados a cabo durante su desarrollo.

Tal como se indicó en la introducción, el objetivo principal será realizar una transferencia de datos entre la PS y la PL de manera eficiente, para lo que se empleará el DMA. En primer lugar será necesario diseñar un cliente de DMA que se instanciará en la PL y será el encargado de recibir las transferencias de datos procedentes de la PS, y de transmitir datos al DMA para hacer una transferencia hacia la PS.

Tras diseñar el cliente DMA y verificar su funcionamiento mediante simulación, se empaquetarán los ficheros VHDL junto con otros ficheros de control en un IP Core para permitir su reutilización en Vivado. Este IP Core se someterá a un proceso de verificación en el que se emplearán BFMs, que simularán el comportamiento de las interfaces AXI4-Stream del procesador para realizar transferencias de datos. Verificado el funcionamiento del IP Core, se diseñará la estructura hardware del SoC necesaria para la aplicación a desarrollar.

3.1 Cliente para DMA

Con el objetivo de realizar un movimiento de datos de manera eficiente entre la PS y la PL, se necesitará emplear un controlador DMA que se encargue de realizar las transferencias de datos entre ambas partes. Para que esto sea posible, en la PL debe existir un periférico que sea cliente del DMA, y por lo tanto debe estar dotado de las interfaces típicas de un cliente DMA.

Se diseñará un cliente de DMA teniendo en cuenta las especificaciones requeridas por una futura aplicación de comunicaciones por la red eléctrica (PLC), donde la entidad encargada de la modulación y demodulación de los datos utiliza interfaces AXI4-Stream de 16 bits para la entrada y salida de datos, y un bus AXI4-Lite para su configuración y reporte de estado.

La Figura 3-1 muestra una vista general la arquitectura del SoC, y proporciona una idea de las interfaces necesarias del cliente DMA:

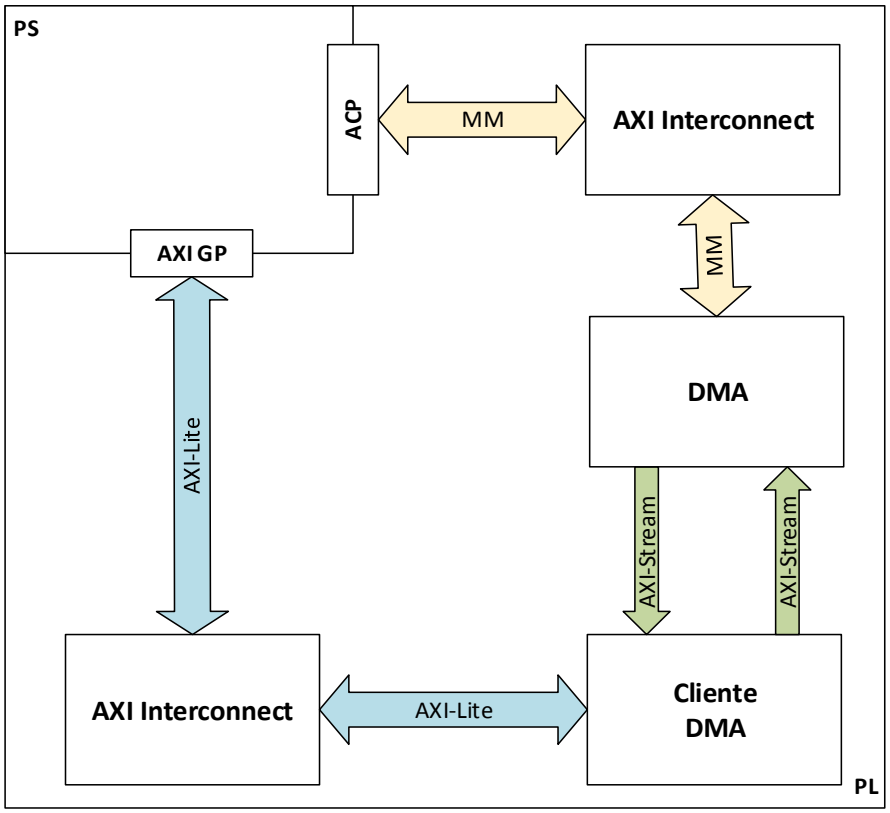

Figura 3-1: Visión general de la arquitectura del SoC

El DMA utilizará la interfaz ACP para comunicarse con el PS, con la finalidad de tener coherencia de datos en las transferencias a realizar. Para aprovechar al máximo el bus de datos de este puerto, que es de 64 bits, el DMA se configurará para trabajar con el mismo número de bits en sus interfaces de los canales de transmisión y recepción. Como consecuencia de esto y debido a que el bloque encargado de las comunicaciones PLC trabaja con interfaces AXI4-Stream de 16 bits, será necesario emplear sendos módulos de adaptación a la entrada y a la salida denominados respectivamente downsizer y upsizer.

Un downsizer es un módulo a cuya entrada recibe un stream de datos de cierto número de bits y genera un stream de salida de menor anchura. Cada dato de entrada estará formado por N datos de salida, siendo N la relación entre la anchura de entrada y la de salida. En la Figura 3-2 se muestra el cronograma del funcionamiento básico de este bloque, teniendo en cuenta el procedimiento de negociación del protocolo Axi4-Stream.

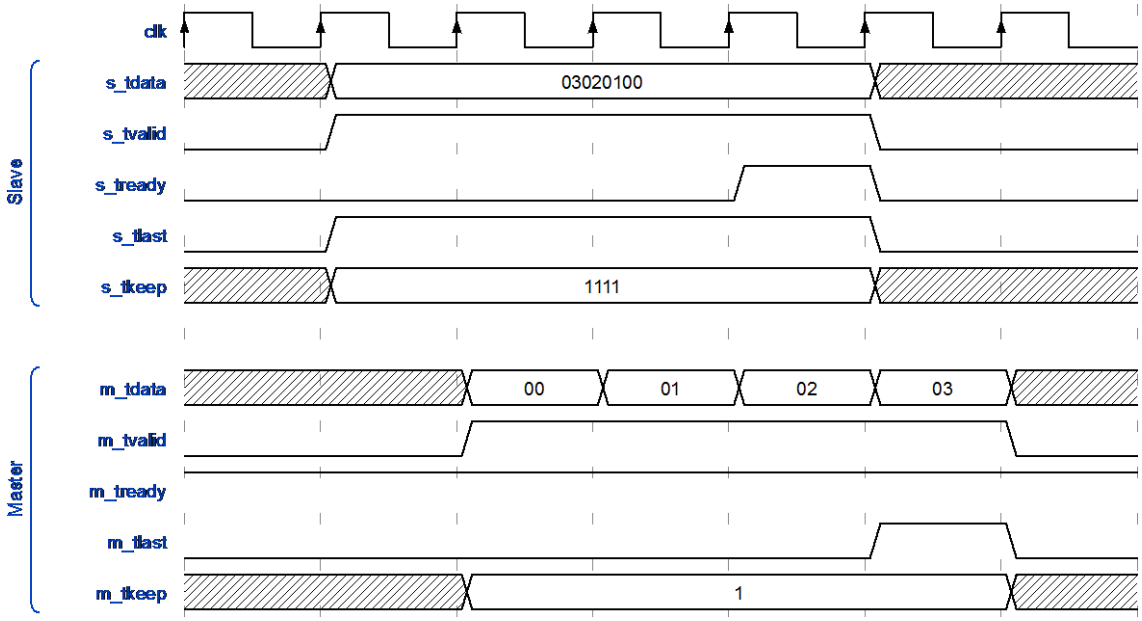

Figura 3-2: Cronograma del funcionamiento del Downsizer

Por otra parte, un upsizer es un módulo que recibe un stream de datos de cierto número de bits y los empaqueta en un registro para generar un stream de salida de mayor anchura. Cada dato de salida estará formado por N datos de entrada, siendo N la relación entre la anchura de salida y la de entrada. En la Figura 3-3 se muestra el cronograma del funcionamiento básico de este bloque, teniendo en cuenta el procedimiento de negociación del protocolo Axi4-Stream.

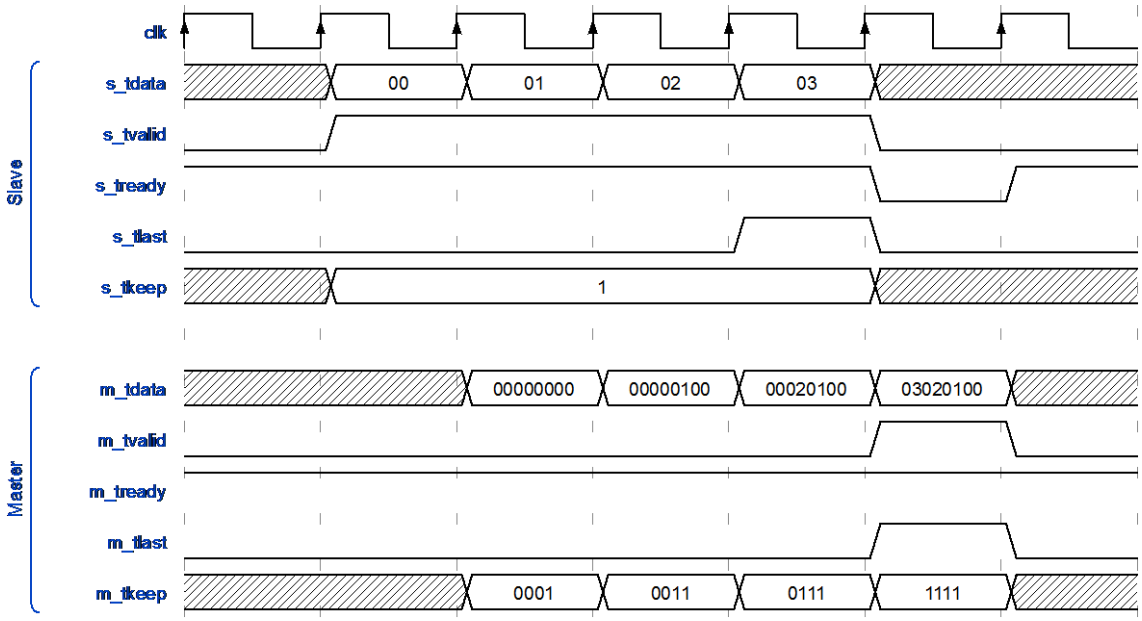

Figura 3-3: Cronograma del funcionamiento del Upsizer

En el diseño a realizar, el downsizer se configurará para adaptar los datos de entrada de 64 bits a 16 bits, mientras que el upsizer realizará el proceso inverso, convertir de 16 bits a 64 bits. Además, el bloque encargado de las comunicaciones PLC se sustituirá por un registro, que tan solo llevara a cabo la operación de transferir a la salida los datos que le llegan a la entrada simulando la latencia introducida por dicho bloque y simplificando el diseño. Este registro se comportará como una FIFO de tan solo una posición.

La Figura 3-4 muestra el interior del cliente para el DMA incluyendo el upsizer y downsizer:

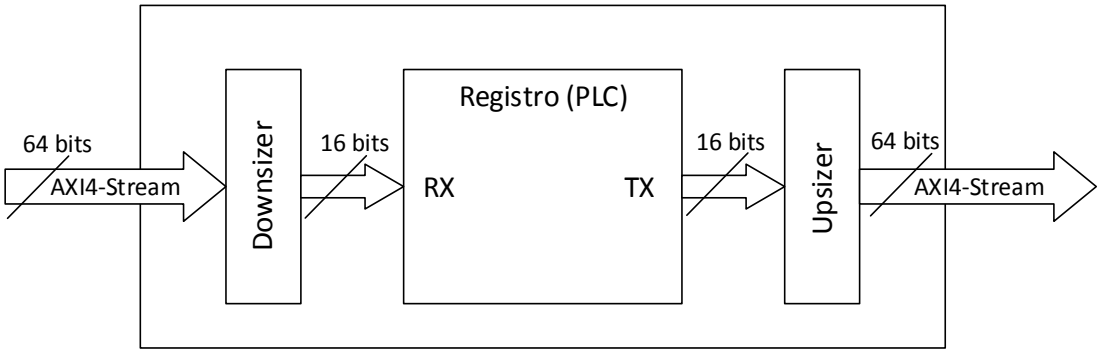

Figura 3-4: Esquema del cliente DMA

Para comprobar el funcionamiento de cada bloque por separado y finalmente todo el conjunto, se realizarán simulaciones con ModelSim generando los estímulos necesarios para enviar un stream de datos y observando los resultados a la salida.

En la simulación se debe verificar que todos los bloques funcionan ante todas las condiciones posibles, ya que entre dos transferencias de datos pueden existir ciclos de espera debidos a que en la entrada no se dispone de datos válidos, o el cliente a la salida no está disponible para recibir un dato. Para probar esto se realizarán simulaciones ante las dos situaciones extremas, el caso donde no exixten ciclos de espera y se obtiene la máxima velocidad, y el caso donde el cliente añade ciclos de espera entre cada par de transferencias.

- **Sin ciclos de espera:** La señal de m_tready estará activa siempre, luego en la interfaz de salida siempre se leerán los datos cuando m_tvalid esté activa, y no habrá que esperar ciclos de reloj adicionales para poder transferir un nuevo dato a la salida.
- **Con ciclos de espera:** La señal m_tready no siempre estará activa, por lo tanto habrá que esperar a que lo esté para poder transferir un nuevo dato a la salida.

3.1.1 Downsizer

El primer bloque del diseño del cliente para DMA es el downsizer, cuya función será convertir un bus de datos de cierto número de bits en otro de tamaño menor. Adicionalmente, las señales de control del protocolo AXI4-Stream deben ser tratadas para cumplir con las especificaciones, por ejemplo la señal TKEEP debe adaptar su tamaño al nuevo número de bytes de salida, y la señal TLAST solo debe estar activa en el último dato del último paquete recibido.

La conversión de un tamaño a otro es configurable mediante los siguientes genéricos:

- **DIN_WITDH:** Número de bits del bus de datos de entrada.
- **DOUT_WITDH:** Número de bits del bus de datos de salida.

Asimismo el tamaño de las señales TID, TDEST y TUSER es configurable mediante los genéricos:

- **TID_LENGHT:** Número de bits del bus TID.
- **TDEST_LENGHT:** Número de bits del bus TDEST.
- **TUSER_LENGHT:** Número de bits del bus TUSER.

Su interfaz tanto de entrada como de salida será del tipo AXI4-Stream para recibir los datos del DMA (64 bits) y transmitirlos al bloque central de comunicaciones PLC con el número de bits del bus de datos adecuado (16 bits). La Figura 3-5 muestra las distintas señales de las interfaces maestro y esclavo del downsizer.

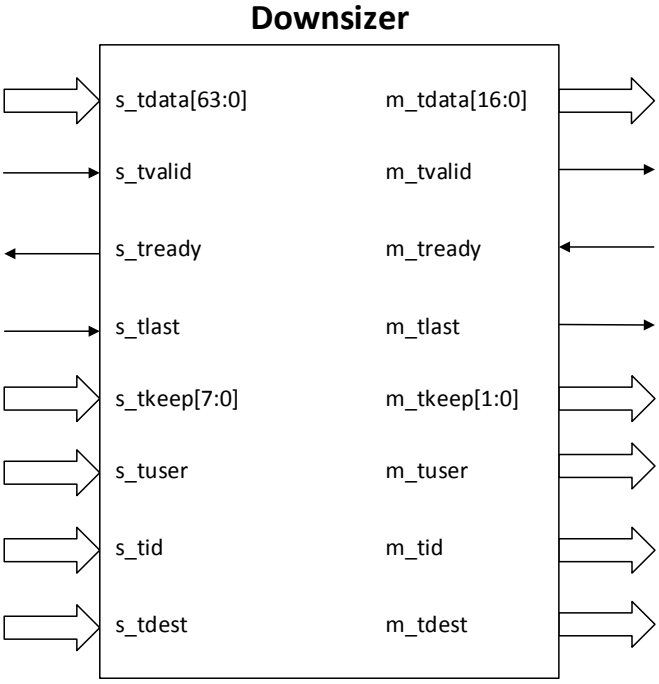

Figura 3-5: Interfaces del Downsizer

El downsizer recibirá un dato de una anchura **DIN_WITDH** y lo dividirá en varios paquetes (Subwords) de tamaño **DOUT_WIDTH** para ir enviándolos uno a uno por la interfaz de salida. Al tratarse de interfaces AXI4-Stream, también habrá que generar e interpretar las señales del proceso de negociación y señales de control para cumplir con las especificaciones del protocolo.

El diagrama de la Figura 7-1, localizada en el capítulo 7, muestra los diferentes bloques que componen el diseño del downsizer.

A continuación se comentarán las partes del código VHDL más relevantes para el modelado de los bloques ilustrados en la Figura 7-1. En primer lugar se definirán las siguientes constantes que se utilizarán a lo largo del código:

```
constant D_WIDTH_RATIO  : integer := DIN_WIDTH/DOUT_WIDTH;
constant BYTES_PER_PACK : integer := DOUT_WIDTH/8;
constant ZEROS          : std_logic_vector(BYTES_PER_PACK-1 downto 0) :=
(others => '0');
constant D_WIDTH_N_BITS : integer := integer(ceil(log2(real(D_WIDTH_RATIO))));
```

Observar la señal **s_tkeep** es muy importante para determinar qué subwords son válidas con la finalidad de indicar si todos los datos son válidos o no, para ello se calcula cuál es el byte válido más significativo (**Data_MSB**). Este valor será útil más adelante para señalar el límite de lectura en un dato entrante e indicar que se puede recibir el siguiente.

```
-- MSB del s_tkeep
  process (s_tkeep)
  begin
    for i in D_WIDTH_RATIO downto 1 loop
      if (s_tkeep((BYTES_PER_PACK*i)-1 downto BYTES_PER_PACK*(i-1)) /= ZEROS)
then
        Data_MSB <= to_unsigned(i-1,D_WIDTH_N_BITS+1);
        exit;
      else
        Data_MSB <= (others => '0');
      end if;
    end loop;
  end process;
```

Existen dos señales de habilitación internas. La señal **dout_ce** indica cuando se puede validar un dato en la interfaz de salida. Cuando esta señal esté activa, la señal de validación de datos de salida (**m_tvalid**) tomará el valor que tiene a la entrada (**s_tvalid**), y además se evaluará si el multiplexor ha seleccionado el último dato de la transferencia para generar la señal **s_tready** y **m_tlast**.

```
-- Señal de validacion de datos
  dout_ce <= not(m_tvalid_i) or (m_tvalid_i and m_tready);

-- Generacion de la señal m_tvalid
  process(clk,rst)
  begin
    if (rst = '0') then
      m_tvalid_i <= '0';
      m_tlast_i <= '0';
    elsif (clk'event and clk = '1') then
```

```
      if(dout_ce = '1') then
        m_tvalid_i <= s_tvalid;
        if (pack_sel = Data_MSB) then
          m_tlast_i <= s_tlast;
        else
          m_tlast_i <= '0';
        end if;
      end if;
    end if;
  end process;

-- Generacion del s_tready. Activo cuando se acaba de desempaquetar un dato
  s_tready_i <= dout_ce when (pack_sel = Data_MSB) else '0';
```

La segunda señal interna, **pack_sel_ce,** habilita el contador que se encarga de seleccionar las distintas subwords que el multiplexor pondrá a la salida.

```
-- Activacion del contador de seleccion de paquete
  pack_sel_ce <= s_tvalid and dout_ce;


-- Generar la señal de selección de paquete de entrada
  process (clk, rst)
  begin
    if (rst = '0') then
      pack_sel <= (others => '0');
    elsif clk'event and clk = '1' then
      if(pack_sel_ce = '1') then
        if (pack_sel = Data_MSB) then
          pack_sel <= (others => '0');
        else
          pack_sel <= pack_sel + to_unsigned(1,D_WIDTH_N_BITS+1);
        end if;
      end if;
    end if;
  end process;
```

Por último, la generación de la señal **m_tkeep**, que indica qué bytes son válidos en el bus de datos de salida, se realizará asignando a esta señal los bits de la señal **s_tkeep** correspondientes a la subword actual. Las señales **m_tid, m_tdest** y **m_tuser** tomarán su valor al registrar las señales correspondientes a la entrada del downsizer.

```
-- Asignacion de señales m_tid, m_tuser, m_tdest y m_tkeep
  process(clk,rst)
  begin
    if(rst = '0') then
      m_tdest <= (others => '0');
      m_tuser <= (others => '0');
      m_tid <= (others => '0');
      m_tkeep <= (others => '0');
    elsif(clk'event and clk = '1') then
      if(dout_ce = '1') then
        m_tdest <= s_tdest;
        m_tuser <= s_tuser;
        m_tid <= s_tid;
        m_tkeep <= s_tkeep((BYTES_PER_PACK*(to_integer(pack_sel)+1))-1 downto
BYTES_PER_PACK*to_integer(pack_sel));
      end if;
    end if;
  end process;
```

La verificación del diseño se realizará mediante simulación, generando los estímulos necesarios en un testbench para enviar un stream de datos. Para verificar su funcionamiento se realizarán varias transferencias en las que se probarán distintas situaciones, como por ejemplo el caso del último paquete de una transferencia, o el caso donde hay un dato incompleto en la entrada (marcado por la señal **s_tkeep**).

En el resultado de la simulación se han señalado estos dos casos para observar el comportamiento del downsizer. La Tabla 3-1 muestra los datos de las transferencias señaladas en el testbench.

Tabla 3-1: Datos de prueba para el Testbench del Downsizer

Color	TDATA	TKEEP	TLAST
Rojo	0x0F0E0D0C0B0A0908	11111111	0
Amarillo	0x0000000000121110	00000111	1

El resultado de la simulación para el caso donde no existen ciclos de espera es mostrado en la Figura 3-6.

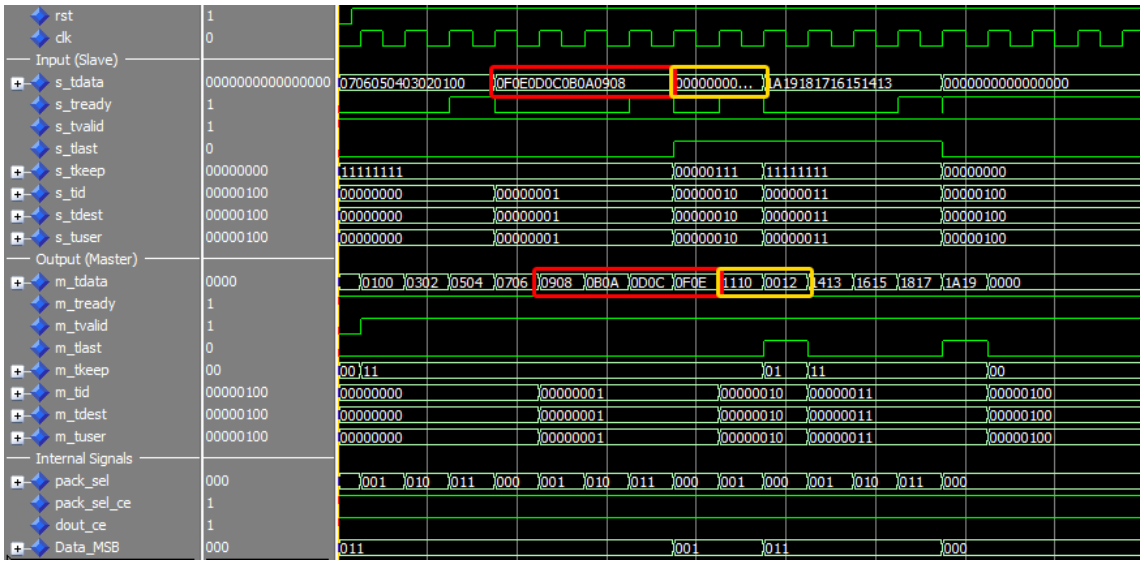

Figura 3-6: Simulación del Downsizer (Sin ciclos de espera)

El resultado de la simulación para el caso donde existen ciclos de espera es mostrado en la Figura 3-7.

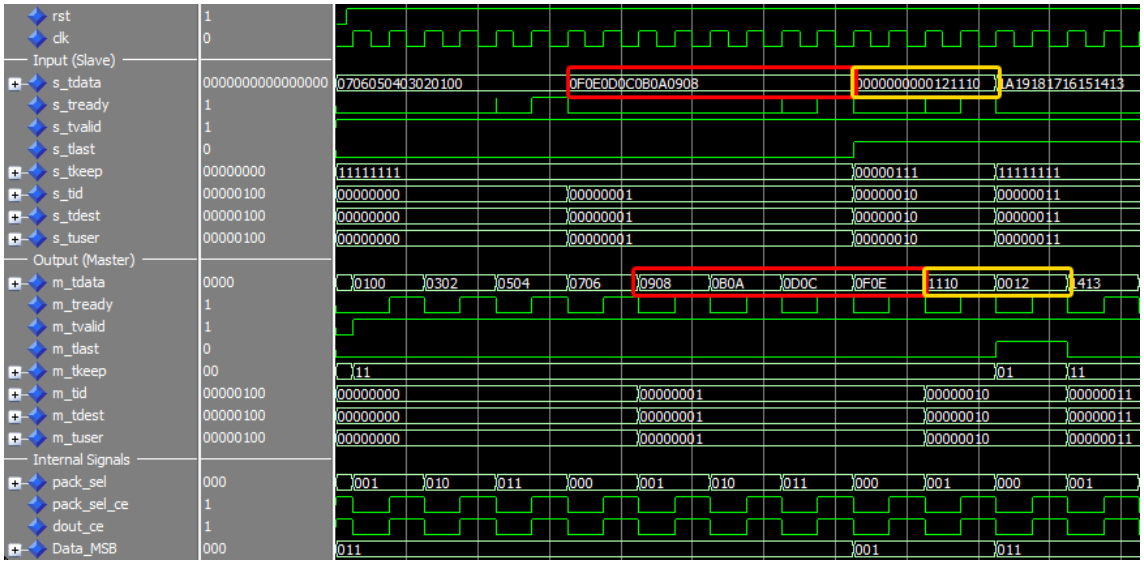

Figura 3-7: Simulación del Downsizer (Con ciclos de espera)

3.1.2 Registro intermedio

El bloque intermedio encargado de las comunicaciones por la red eléctrica será sustituido en este diseño por un registro que se comportará como una FIFO de una única posición, y simulará la latencia introducida por el Core de PLC pero simplificando el diseño de este módulo.

Las interfaces de dicho registro serán del tipo Axi4-Stream con una anchura de 16 bits para sustituir al módulo de PLC, sin embargo esta anchura es configurable para permitir reutilizar el diseño en un futuro.

El registro dispone de varios genéricos para configurar el número de bits de los distintos buses AXI4-Stream:

- **D_WIDTH:** Número de bits de los buses de datos de entrada y salida.
- **TID_LENGHT:** Número de bits del bus TID.
- **TDEST_LENGHT:** Número de bits del bus TDEST.
- **TUSER_LENGHT:** Número de bits del bus TUSER.

La Figura 3-8 muestra las distintas señales de las interfaces maestro y esclavo del registro intermedio.

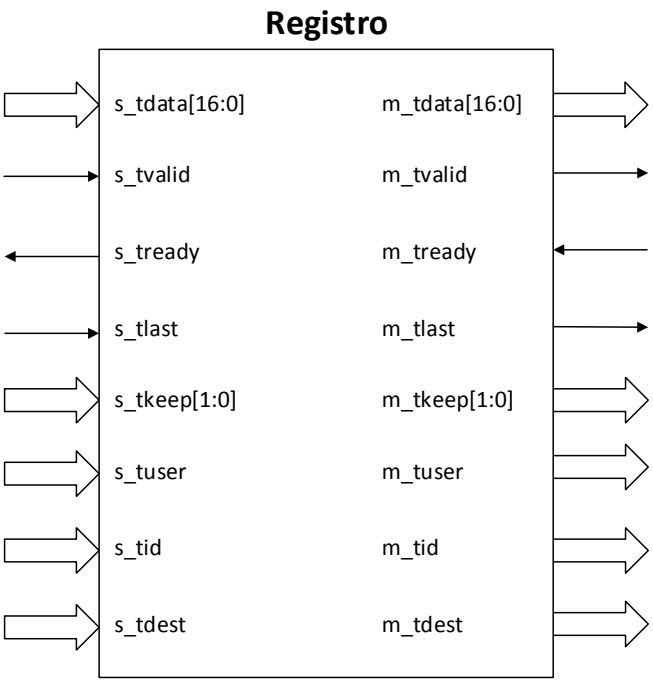

Figura 3-8: Interfaces del Registro Intermedio

El funcionamiento del registro es muy simple. Se comporta como una FIFO de una única posición cuya tarea es registrar los datos de la entrada a la salida, cumpliendo con el proceso de negociación del protocolo AXI4-Stream, es decir, comprobando cuando hay un dato válido a la entrada y cuando se puede poner un nuevo dato a la salida según las señales TVALID y TREADY.

En la Figura 3-9 se muestra el cronograma del funcionamiento básico de este bloque en lo que respecta a las señales de entrada y salida de las interfaces AXI4-Stream:

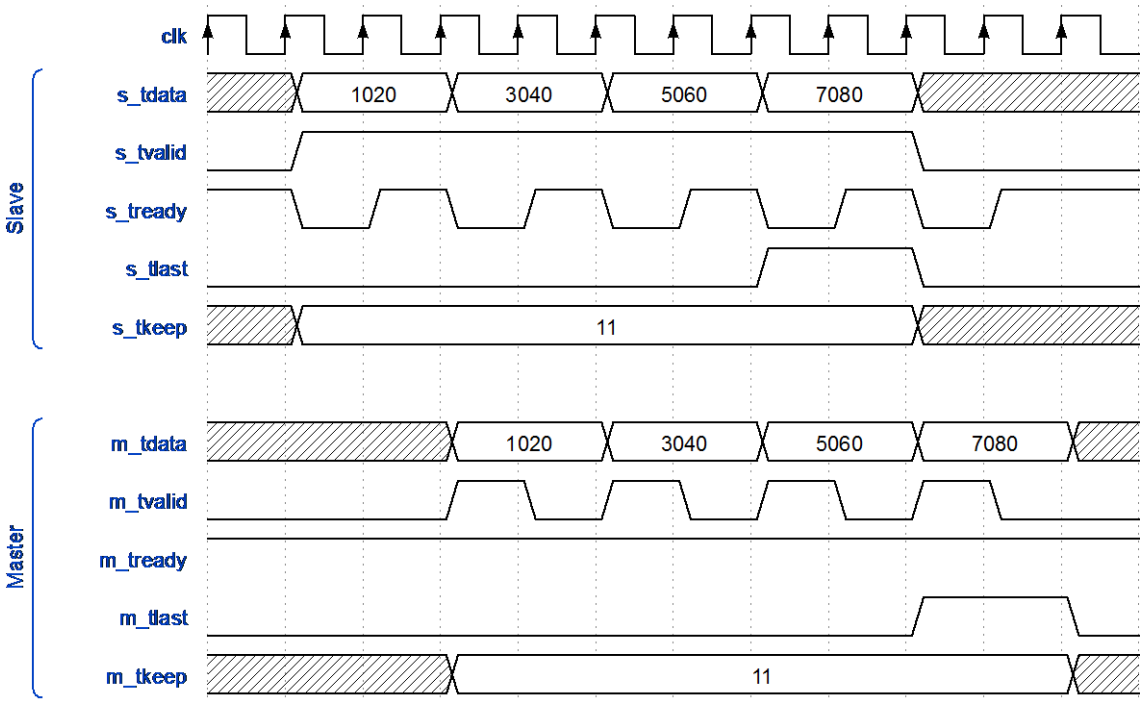

Figura 3-9: Diagrama de tiempos del funcionamiento del Registro Intermedio

El diagrama de la Figura 7-2, localizada en el capítulo 7, muestra los diferentes bloques que componen el diseño del registro intermedio. A continuación se comentarán las partes del código VHDL más relevantes para el modelado de los bloques ilustrados en la Figura 7-2.

Cuando se cumple que **s_tready = 1** y **s_tvalid = 1** entonces se dispone de un nuevo dato a la entrada que puede ser escrito a la salida. Esto se indicará con la señal **new_data**. En este momento se registra el valor de las señales de entrada y se espera a que se validen dichos datos.

```
-- Nuevo dato a la entrada
  new_data <= s_tready_i and s_tvalid;

-- Empaquetado del registro de salida, señales TID, TDEST, TUSER y TKEEP
  process(clk,rst)
  begin
    if(rst = '0') then
      dout <= (others => '0');
      m_tkeep <= (others => '0');
      m_tid <= (others => '0');
      m_tdest <= (others => '0');
      m_tuser <= (others => '0');
    elsif (clk'event and clk = '1') then
      if(new_data = '1') then
        dout <= s_tdata;
        m_tkeep <= s_tkeep;
        m_tid <= s_tid;
        m_tdest <= s_tdest;
        m_tuser <= s_tuser;
      end if;
    end if;
  end process;

m_tdata <= dout;
```

La señal interna **dout_ce** habilita la actualización de la señal de validación **m_tvalid** cuando ésta está inactiva, o cuando se ha producido una lectura en la interfaz de salida (**m_tready = 1** y **m_tvalid = 1**). **M_tvalid** validará un dato cuando **dout_ce** esté activa y se disponga de un nuevo dato a la entrada (**new_data =1**). En este momento también se generará la señal **m_tlast** que indica el último dato de la transferencia.

```
-- Señal de validacion de escritura
  dout_ce <= not(m_tvalid_i) or (m_tvalid_i and m_tready);

-- Generacion de la señal m_tvalid y m_tlast
  process(clk,rst)
  begin
    if (rst = '0') then
      m_tvalid_i <= '0';
      m_tlast <= '0';
    elsif (clk'event and clk = '1') then
      if(dout_ce = '1') then
        m_tvalid_i <= new_data;
        if(s_tready_i = '1') then
          m_tlast <= s_tlast;
        end if;
      end if;
    end if;
  end process;
```

Por último, la señal **s_tready** será la inversa de **m_tvalid**, ya que al producirse una lectura de un dato válido a la salida, el registro está listo para recibir un nuevo dato (**s_tready = 1**) y no se dispone de dato que validar a la salida (**m_tvalid = 1**).

```
s_tready_i <= not(m_tvalid_i);
```

Una vez diseñado el registro, se procederá a verificar su funcionamiento bajo simulación. En ella se comprobará que las distintas señales de control se generan correctamente y por tanto las transferencias de datos son correctas. Dicha simulación se ha llevado a cabo sobre un registro con interfaces de entrada y salida de 16 bits, pero ya que el diseño es genérico y permite cambiar el número de bits de dichas interfaces, su funcionamiento no dependerá de la anchura de los buses de entrada y salida.

Se realizarán varias transferencias de datos para comprobar que el registro funciona correctamente y se cumple el protocolo AXI4-Stream. En el resultado de la simulación se han señalado dos transferencias para observar que el cronograma obtenido cumple con el funcionamiento mostrado en la Figura 3-9.

El resultado de la simulación para el caso donde no existen ciclos de espera se muestra en la Figura 3-10:

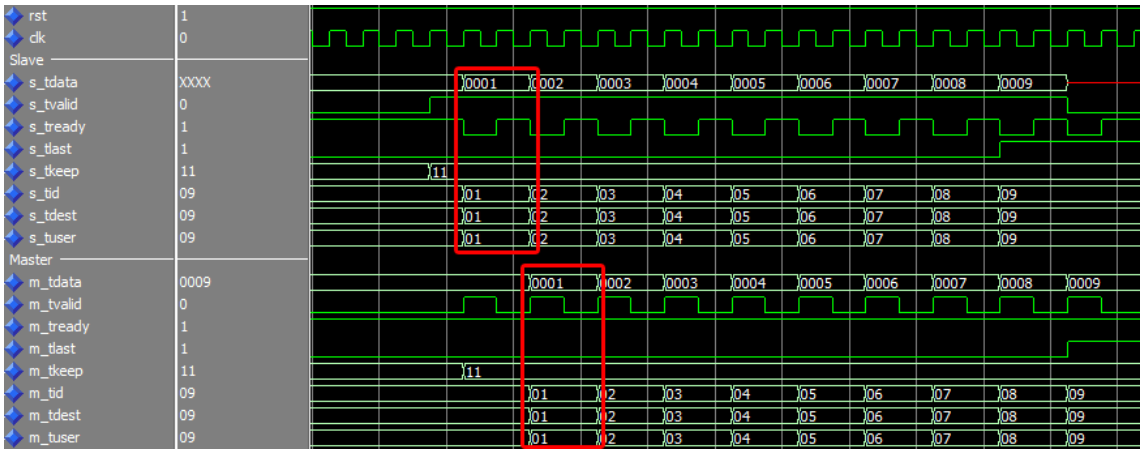

Figura 3-10: Simulación del Registro (Sin ciclos de espera)

El resultado de la simulación para el caso donde existen ciclos de espera se muestra en la Figura 3-11:

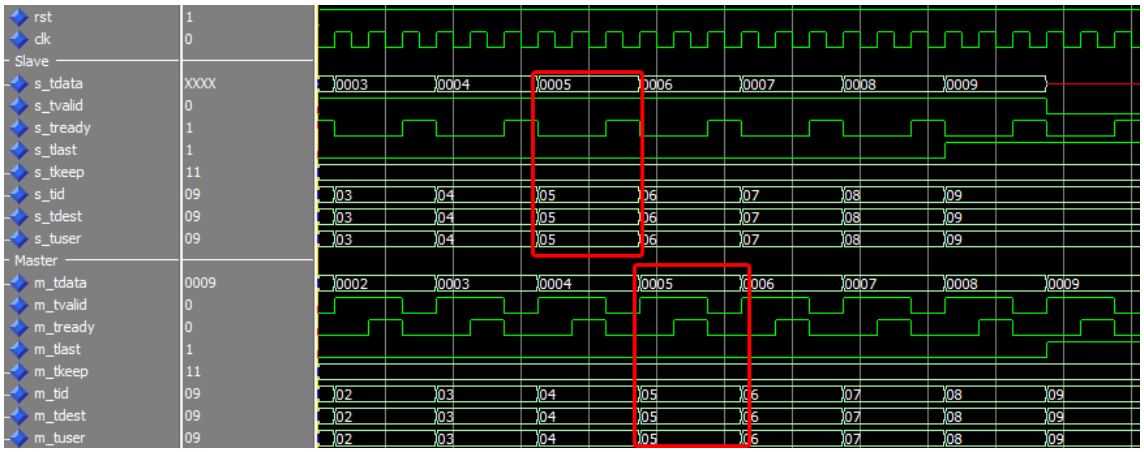

Figura 3-11: Simulación del Registro (Con ciclos de espera)

3.1.3 Upsizer

El último bloque del cliente para el DMA es el upsizer, que será el encargado de adaptar un bus de datos de cierto número de bits en otro de tamaño mayor. Para ello empaquetará los datos de entrada provenientes del registro intermedio en un registro de salida de mayor número de bits. Al igual que en el downsizer, las señales de control y negociación del protocolo AXI4-Stream deben ser tratadas para que cumplan las especificaciones. Por ejemplo la señal TKEEP debe aumentar su tamaño acorde al nuevo tamaño del bus TDATA.

La conversión de número de bits a un valor mayor es configurable mediante los siguientes genéricos:

- **DIN_WITDH:** Número de bits del bus de datos de entrada.
- **DOUT_WITDH:** Número de bits del bus de datos de salida.

Asimismo el tamaño de las señales TID, TDEST y TUSER es configurable mediante los genéricos:

- **TID_LENGHT:** Número de bits del bus TID.
- **TDEST_LENGHT:** Número de bits del bus TDEST.
- **TUSER_LENGHT:** Número de bits del bus TUSER.

Su interfaz de entrada y salida, al igual que en los bloques anteriores, es del tipo AXI4-Stream, por lo tanto habrá que cumplir con dicho protocolo. En este caso, el bus de datos de entrada será de 16 bits para recibir los datos del bloque central del cliente DMA y el bus de datos de salida de 64 bits para transferir los datos al DMA. La Figura 3-12 muestra las interfaces maestro y esclavo del upsizer.

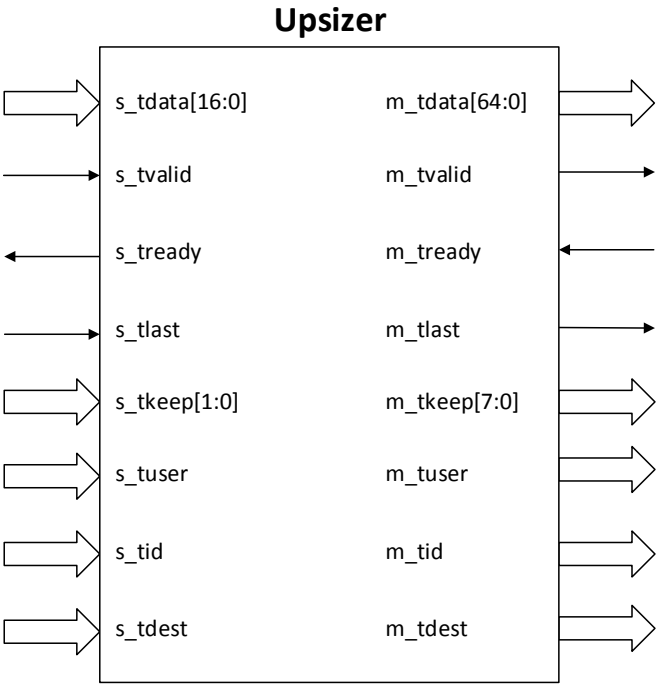

Figura 3-12: Interfaces del Upsizer

El upsizer recibirá datos de anchura **DIN_WITDH** y los empaquetará hasta completar un registro de salida que tendrá un número de bits igual a **DOUT_WIDTH**. El dato contenido en este registro se enviará por la interfaz de salida cuando esté completo, o bien se trate del último dato de la transferencia (marcado con TLAST=1). Hasta que no sea leído, no se podrá recibir un nuevo dato para ser empaquetado.

El diagrama de la Figura 7-3, localizada en el capítulo 7, muestra los diferentes bloques que componen el diseño del registro intermedio.

A continuación se comentarán las partes del código VHDL más relevantes para el modelado de los bloques ilustrados en la Figura 7-3. En primer lugar se definirán unas constantes que se utilizarán a lo largo del código:

```
constant D_WIDTH_RATIO  : integer := DOUT_WIDTH/DIN_WIDTH;
constant BYTES_PER_PACK : integer := DIN_WIDTH/8;
```

Al igual que en el downsizer, existen varias señales internas de control. La señal **new_data** indica que existe un dato válido a la entrada, y estará activa cuando **s_tvalid** y **s_tready** lo estén simultáneamente. Otra de las señales internas, **last_pack,** indica que se ha recibido la última subword del registro de empaquetado, esto ocurre cuando se recibe el último dato de la transferencia (**s_tlast=1**), o el registro de empaquetado está completo.

```
-- Nuevo dato de entrada
new_data <= s_tvalid and s_tready_i;

-- Generacion de la señal last_pack
last_pack <= pack_sel(D_WIDTH_RATIO-1) or s_tlast;
```

La señal interna **dout_ce** indica cuando se puede poner un dato en la interfaz de salida, y servirá para habilitar la generación de las señales de validación de datos (**m_tvalid**) y último dato de un paquete (**m_tlast**).

```
-- Señal de validacion de datos
dout_ce <= not(m_tvalid_i) or (m_tvalid_i and m_tready);

-- Generacion de la señal m_tvalid y m_tlast
  process(clk,rst)
  begin
    if (rst = '0') then
      m_tvalid_i <= '0';
      m_tlast <= '0';
    elsif (clk'event and clk = '1') then
      if(dout_ce = '1') then
        m_tvalid_i <= last_pack and new_data;
        m_tlast <= s_tlast;
      end if;
    end if;
  end process;

s_tready_i <= not(m_tvalid_i);
```

En el diseño del upsizer se dispone de un registro de empaquetado, donde se almacenan los datos de entrada, y un contador one-hot que selecciona la posición del registro donde se almacenará el dato recibido. El contador one-hot aumentará su valor cuando el upsizer disponga de un nuevo dato, y se reiniciará cuando se reciba la última subword.

```
-- Generar la señal de selección de paquete para registro de salida
  process (clk, rst)
  begin
    if (rst = '0') then
      pack_sel <= (others=>'0');
      pack_sel(0) <= '1';
    elsif clk'event and clk = '1' then
      if ((last_pack and new_data) = '1') then
        pack_sel <= (others=>'0');
        pack_sel(0) <= '1';
      elsif(new_data = '1') then
        pack_sel <= pack_sel(D_WIDTH_RATIO-2 downto 0) &
                    pack_sel(D_WIDTH_RATIO-1);
      end if;
    end if;
  end process;

  -- Empaquetado del registro de salida
  process(clk,rst)
  begin
    if(rst = '0') then
      dout <= (others => '0');
    elsif (clk'event and clk = '1') then
      if(new_data = '1') then
        if (pack_sel(0) = '1') then
          dout <= (others => '0');
        end if;

        for i in 0 to D_WIDTH_RATIO-1 loop
          if(pack_sel(i) = '1') then
            dout(DIN_WIDTH*(i+1)-1 downto DIN_WIDTH*i) <= s_tdata;
          end if;
        end loop;
      end if;
    end if;
  end process;
```

```
m_tdata <= dout;
```

La señal **m_tkeep** se generará a medida que se van empaquetando datos en el registro de salida, de forma que los bits de la señal **s_tkeep** se asignarán a **m_tkeep** en la posición del paquete actual indicada por el contador.

```
-- Generar la señal m_tkeep
  process (clk, rst)
  begin
    if (rst = '0') then
      m_tkeep_i <= (others=>'0');
    elsif clk'event and clk = '1' then
      if(new_data = '1') then
        if (pack_sel(0) = '1') then
          m_tkeep_i <= (others=>'0');
          m_tkeep_i(BYTES_PER_PACK-1 downto 0) <= s_tkeep;
        else
          for i in 0 to D_WIDTH_RATIO-1 loop
            if(pack_sel(i) = '1') then
              m_tkeep_i((BYTES_PER_PACK*(i+1)-1) downto (BYTES_PER_PACK*i)) <=
s_tkeep;
            end if;
          end loop;
        end if;
      end if;
    end if;
  end process;
```

Las señales **m_tid, m_tdest** y **m_tuser** tomarán el valor de las señales correspondientes a la entrada del downsizer.

```
-- Asignacion de señales TID, TUSER y TDEST
  process(clk,rst)
  begin
    if(rst = '0') then
      m_tdest <= (others => '0');
      m_tuser <= (others => '0');
      m_tid <= (others => '0');
    elsif(clk'event and clk = '1') then
      if((dout_ce and last_pack and new_data) = '1') then
        m_tdest <= s_tdest;
        m_tuser <= s_tuser;
        m_tid <= s_tid;
      end if;
    end if;
  end process;
```

Tras realizar el diseño del upsizer, se verificará su funcionamiento mediante simulación, para ello se generarán las distintas señales de control en el testbench y se comprobará que las transferencias se realizan correctamente. Dicha simulación del diseño se ha realizado con una interfaz de entrada de 8 bits y una interfaz de salida de 32 bits, pero al tratarse de un upsizer genérico, también funcionará para otras combinaciones de número de bits de entrada y salida como la que se empleará en el diseño final 16-64.

Se realizarán varias transferencias para comprobar que funciona correctamente ante varias situaciones, como el caso del último paquete (TLAST = 1), o el caso donde el paquete está incompleto.

En el resultado de la simulación se han señalado estos dos casos para observar el comportamiento del upsizer. La Tabla 3-2 muestra los datos señalados en dichas transferencias:

Tabla 3-2: Datos de prueba para el Testbench del Upsizer

Color	TDATA	TKEEP	TLAST
Rojo	0x07060504	1111	0
Amarillo	0x00000908	0011	1

El resultado de la simulación para el caso donde no existen ciclos de espera se muestra en la Figura 3-13.

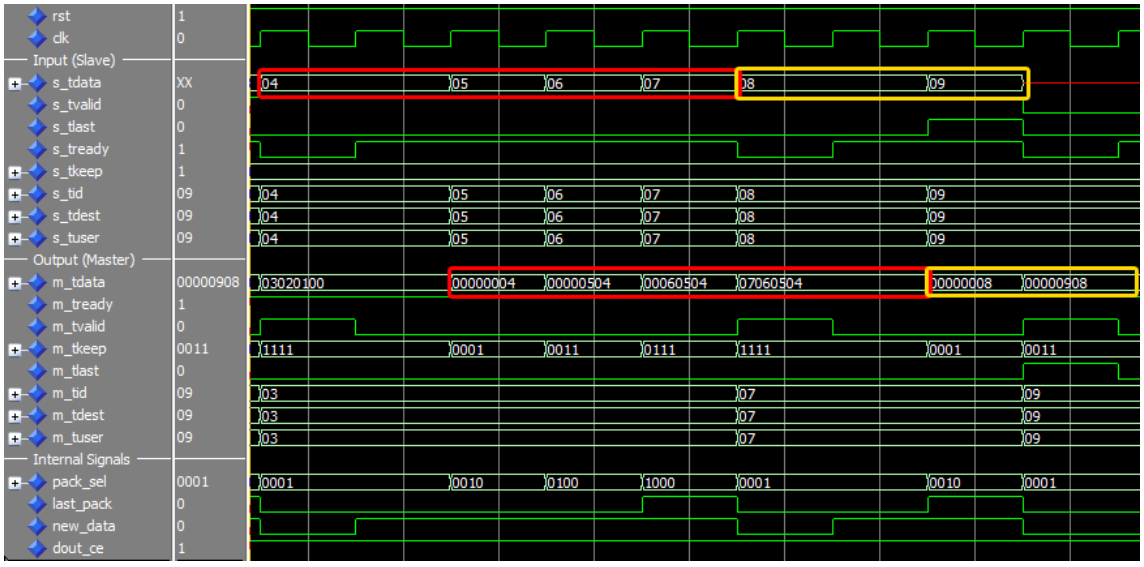

Figura 3-13: Simulación del Upsizer (Sin ciclos de espera)

El resultado de la simulación para el caso donde existen ciclos de espera se muestra en la Figura 3-14.

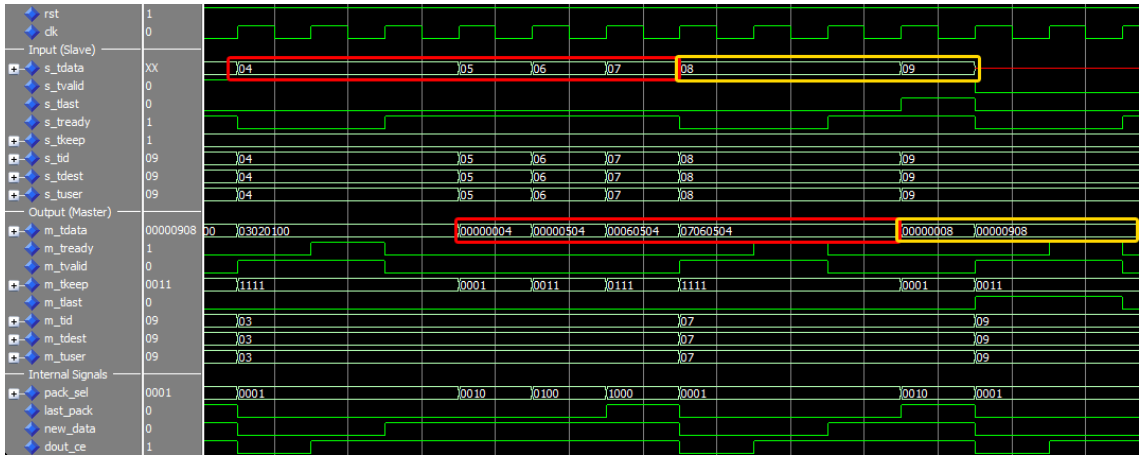

Figura 3-14: Simulación del Upsizer (Con ciclos de espera)

3.1.4 Diseño Completo (AXI PLC)

Una vez verificado cada bloque por separado, se ha creado el diseño completo, llamado AXI PLC, incluyendo downsizer, registro y upsizer como muestra la Figura 3-15, y se ha realizado una simulación funcional con Modelsim para probar el funcionamiento de todo el conjunto.

En dicha figura se ha omitido la interfaz AX4-Lite vista en la Figura 3-1, ya que hasta el momento solo se han tenido en cuenta las interfaces AXI4-Stream. Más adelante en la creación del IP para Vivado se incluirá dicha interfaz del tipo AXI4-Lite.

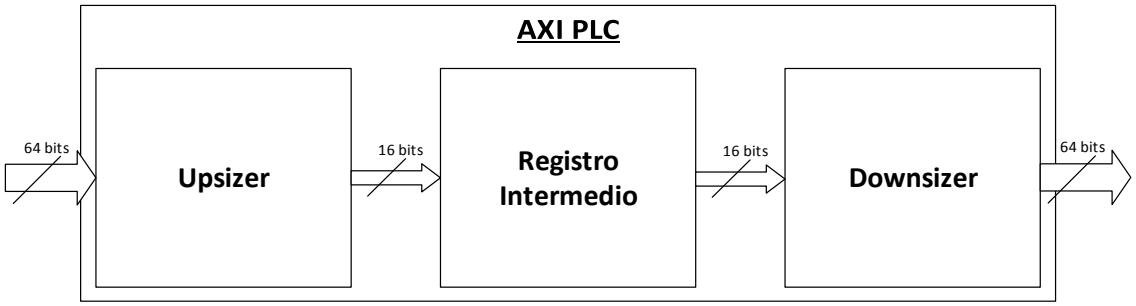

Figura 3-15: Diseño del AXI PLC. Diseño completo del sistema

En la simulación se realizará el envío de varias transferencias de datos, con la finalidad de verificar que éstas se llevan a cabo correctamente y las especificaciones del protocolo AXI4-Stream se cumplen. Para ello también se probará el diseño frente a situaciones especiales como por ejemplo los casos donde los datos están incompletos (marcado por la señal **s_tkeep**), o el caso del último dato de la transferencia (**s_tlast = 1**).

En el resultado de la simulación se han señalado estos dos casos para observar el comportamiento del sistema ante dos situaciones especiales. La Tabla 3-3 muestra los datos de las transferencias marcados:

Tabla 3-3: Datos de prueba para el Testbench del AXI PLC

Color	TDATA	TKEEP	TLAST
Rojo	0x0000000000121110	00000111	0
Amarillo	0x0A19181716151413	11111111	1

El resultado de la simulación para el caso donde no existen ciclos de espera se muestra en la Figura 3-16.

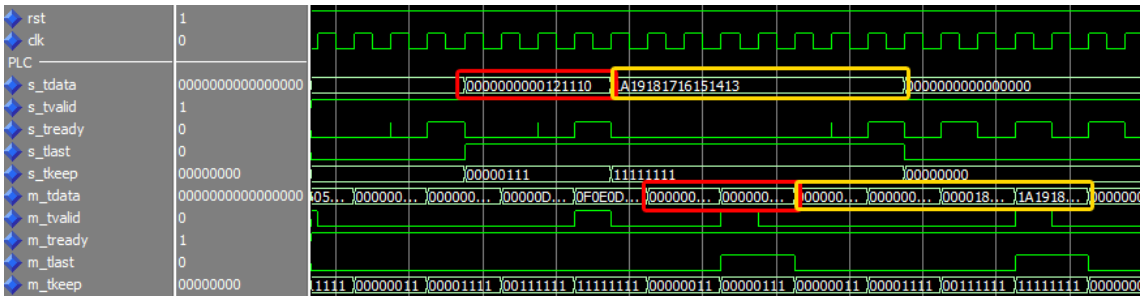

Figura 3-16: Simulación del AXI PLC (Sin ciclos de espera)

El resultado de la simulación para el caso donde existen ciclos de espera se muestra en la Figura 3-17.

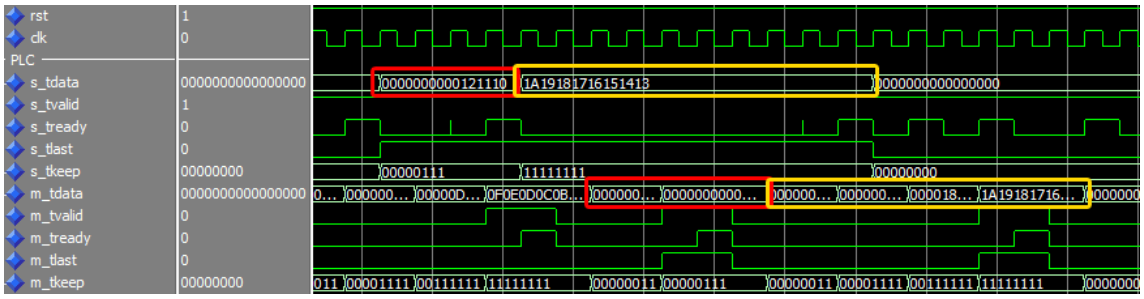

Figura 3-17: Simulación del AXI PLC (Con ciclos de espera)

3.2 Empaquetado del IP Core con IP Packager

Para permitir la reutilización del periférico en otros diseños, el código VHDL debe ir acompañado de una serie de ficheros de control y documentación. Todos esto debe encapsularse en un módulo llamado IP Core para integrarlo en Vivado. Este proceso se denomina empaquetado y se realiza con la herramienta IP Packager.

Una vez completado el diseño del bloque completo (AXI PLC) y verificado mediante simulación, el siguiente paso será empaquetar el periférico para poder integrarlo en Vivado como un IP Core, y poder reutilizarlo en los diseños posteriores.

En la Figura 3-18 se muestra el esquema interno del IP a empaquetar.

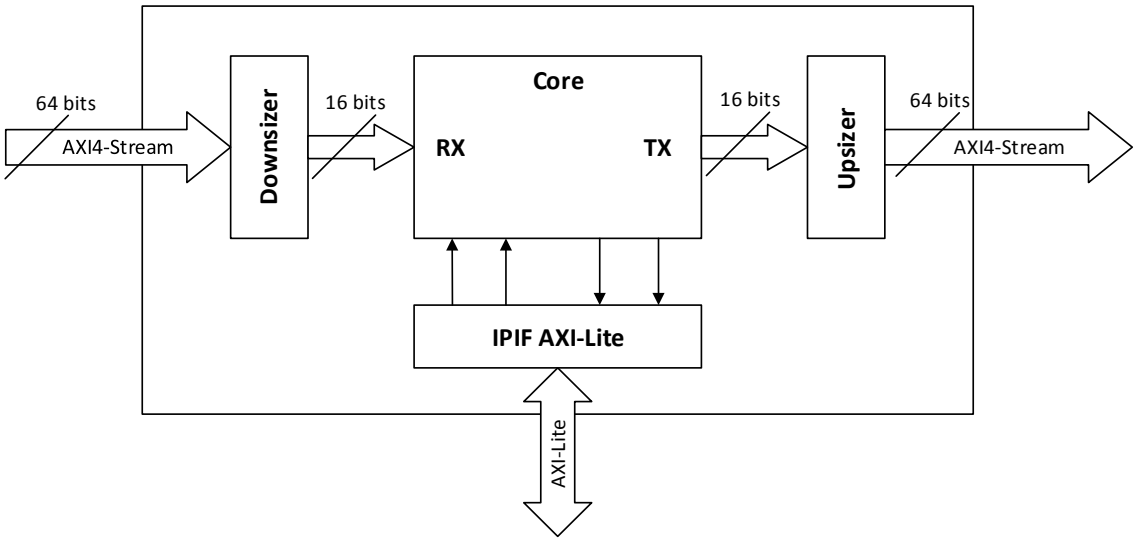

Figura 3-18: Esquema interno del IP

IP Packager es la herramienta incluida en la suite Vivado que se encargará de realizar la tarea de empaquetar el periférico en un IP Core. Dentro de éste se pueden incluir distintos tipos de ficheros, tanto los que implementan la funcionalidad, como otros destinados a documentación o simulación:

- Ficheros HDL de síntesis, simulación y testbench.
- Documentación.
- Ejemplos de diseño.
- Archivos de implementación.
- Drivers.
- Interfaz gráfica de configuración.

En este apartado se explicarán los pasos llevados a cabo para el empaquetado del periférico del cliente DMA creado anteriormente, permitiendo su posterior implementación en la arquitectura del SoC. Para más información sobre el empaquetado con IP Packager consúltese [Xilinx, 2014c].

3.2.1 Pasos para la creación del IP

Para la creación del IP core es necesario crear un proyecto de Vivado y abrir el IP Packager pinchando en **Tools – Create and Package IP**. El asistente de Vivado proporcionará las instrucciones necesarias a lo largo del proceso, facilitando la creación de una plantilla para el IP, que posteriormente se modificará para obtener el periférico deseado.

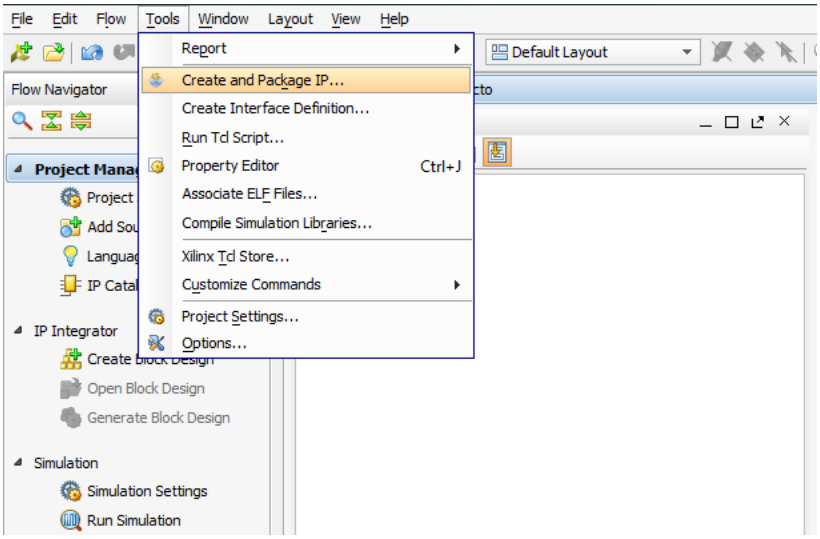

Figura 3-19: Abrir IP Packager

En el primer menú, hay que seleccionar la opción de crear un periférico AXI4, ya que el IP Core dispondrá de interfaces del este tipo, en concreto AXI4-Stream y AXI4-Lite.

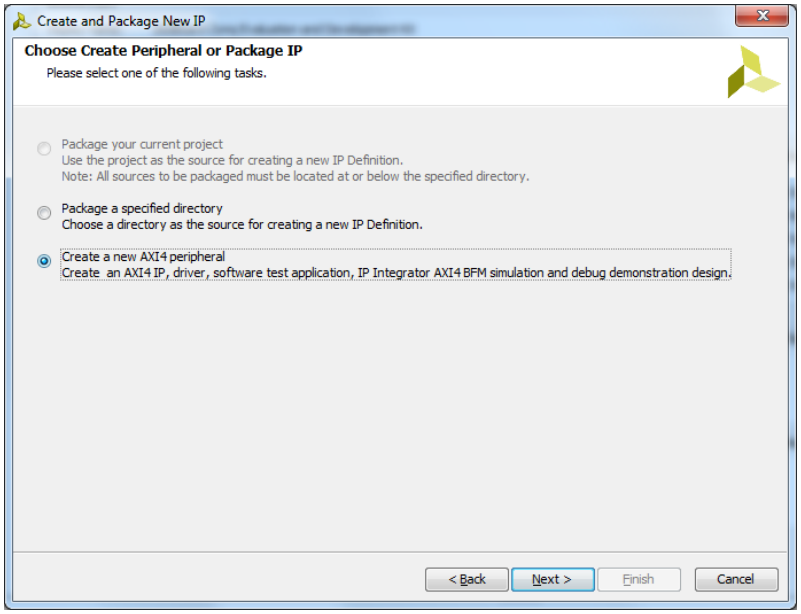

Figura 3-20: Crear periférico con interfaces AXI4

A continuación se introducirán los datos que identificarán al periférico en el repositorio de IP Cores: nombre, versión, nombre a mostrar, descripción y dirección del repositorio del proyecto donde se guardará el IP.

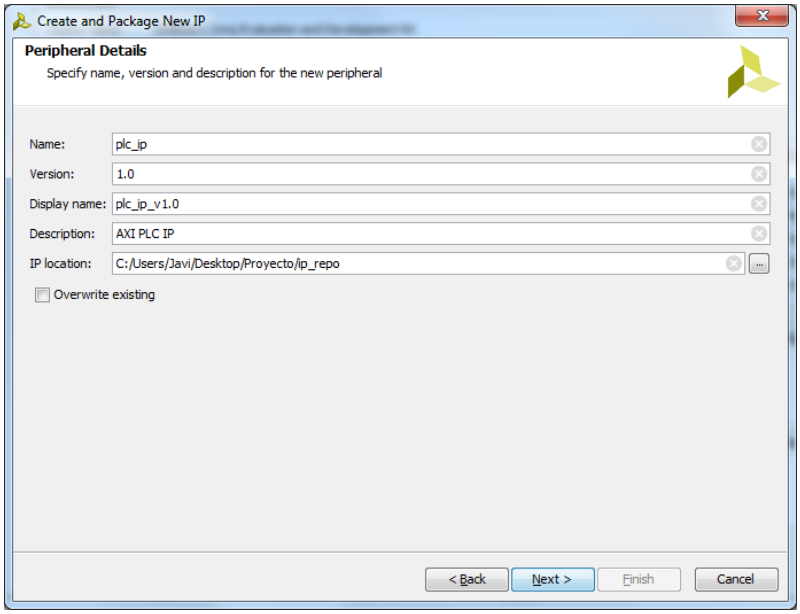

Figura 3-21: Detalles del IP Core

En la siguiente pantalla se configurarán las interfaces AXI4 de las que dispondrá el IP, indicando tipo, nombre, número de bits del bus de datos y número de registros del IPIF, entre otros parámetros.

El IPIF es una interfaz bidireccional que hace el papel de intermediario entre el bus AXI4 y el IP Core, de forma que a vista del usuario, para realizar una transferencia lo único que hay que hacer es escribir y leer registros del IPIF. El asistente de Vivado creará un IPIF para cada uno de los buses AXI4.

La Figura 3-22 muestra la estructura de un IP con interfaz AXI4 utilizando el IPIF.

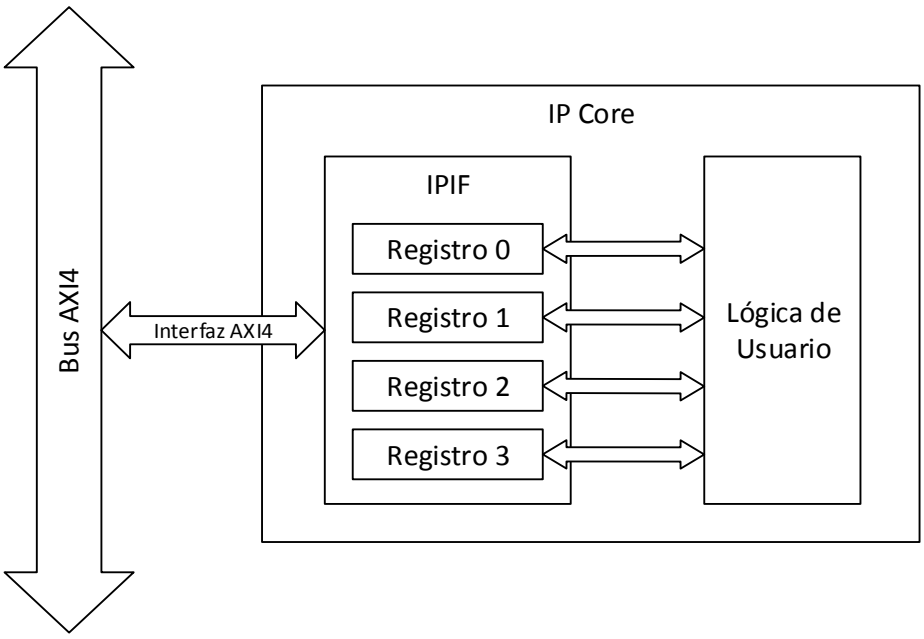

Figura 3-22: Estructura del IPIF

Para llevar a cabo una escritura en un registro habrá que realizar una transferencia cumpliendo con el protocolo AXI4-Lite, indicando la dirección de memoria de dicho registro y los datos a escribir en él, el IPIF se encargara de transferir ese valor al registro. Para una operación de lectura se indica la dirección de memoria del registro que se desea leer y el IPIF se encargará de devolver el dato contenido en éste.

Para el IP que contendrá el diseño del cliente DMA (AXI PLC) se han añadido tres interfaces:

- **AXI4-Stream Slave:** Interfaz de entrada de datos procedentes del DMA, con un bus de datos de 64 bits.
- **AXI4-Stream Master:** Interfaz de salida del IP con los datos procesados, que serán enviados de nuevo al DMA por un bus de datos de 64 bits.
- **AXI4-Lite Slave:** Interfaz de control y configuración del IP mediante la lectura y escritura de registros del IPIF. Se añadirán 4 registros, aunque en este proyecto no se implementará ninguna funcionalidad.

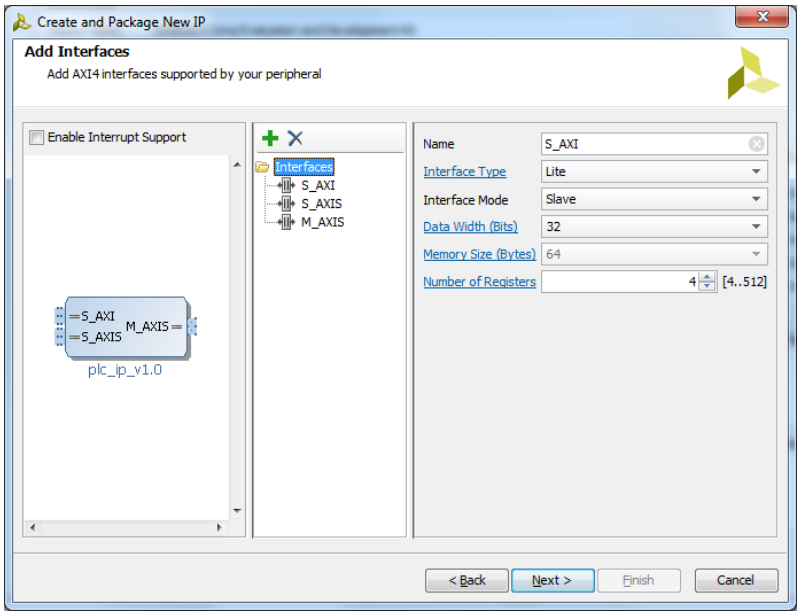

Figura 3-23: Creación de las interfaces AXI4 necesarias

Una vez configuradas las interfaces, se puede continuar hacia la siguiente ventana del asistente, donde se seleccionará “Edit IP” para proceder a la configuración del periférico personalizado.

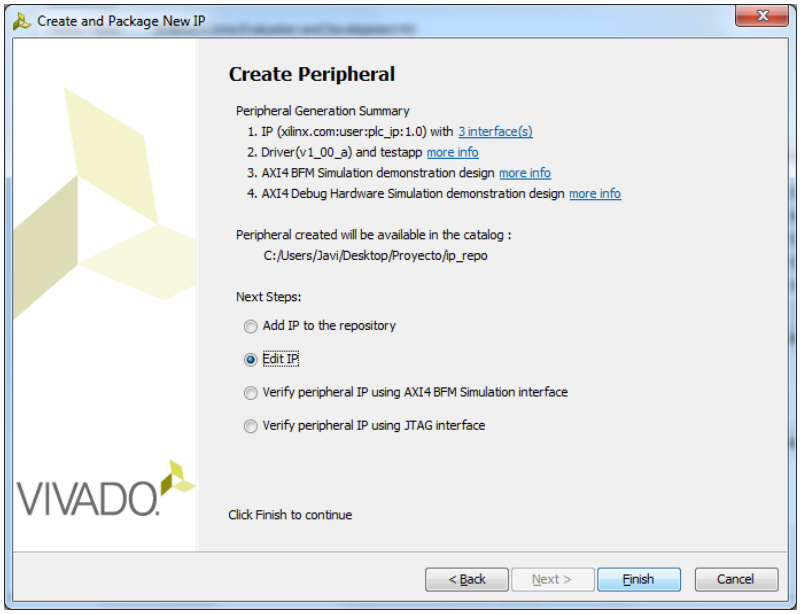

Figura 3-24: Finalización del asistente del IP Packager

3.2.2 Configuración del IP

Finalizada la creación del periférico base con las interfaces necesarias mediante el asistente, se abrirá un nuevo proyecto de Vivado destinado a editar el IP y dotar a éste de una funcionalidad específica. En primer lugar se modificará la identificación de dicho IP en la ventana "Package IP" como aparece en la Figura 3-25.

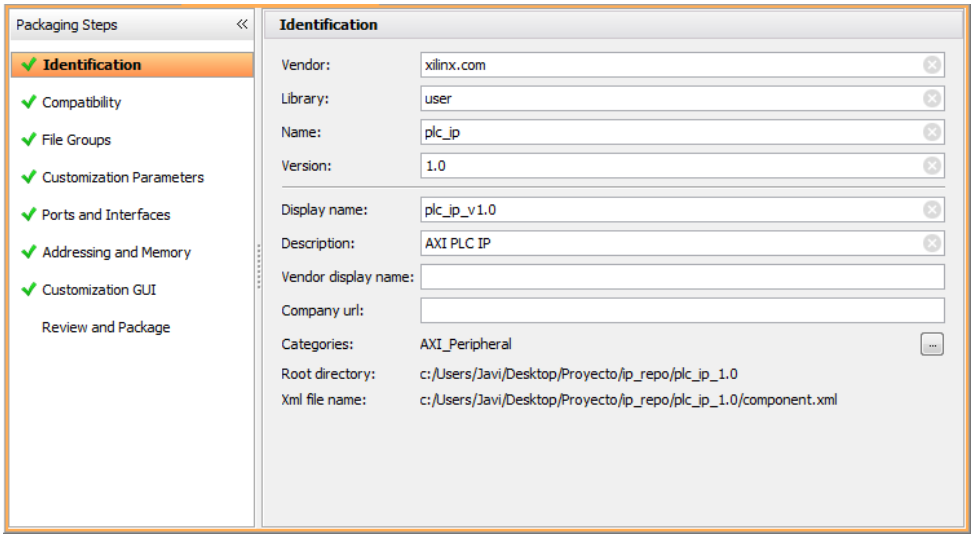

Figura 3-25: Identificación del IP Core

A continuación se añadirán los archivos VHDL del downsizer, upsizer, registro, y axi_plc al proyecto. Debido a que el diseño del periférico realizado define el protocolo de comunicaciones de las interfaces AXI4-Stream, no es necesario incluir los IPIF específicos para éstas. Por tanto, se eliminarán los ficheros VHDL que implementan los IPIF de este tipo de interfaces.

Además se eliminará la instanciación de dichos componentes del fichero top del sistema, y se instanciará el componente AXI PLC en su lugar. El árbol de archivos quedará como se muestra en la Figura 3-26.

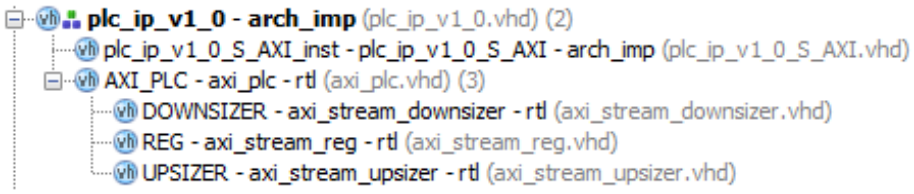

Figura 3-26: Archivos del IP tras añadir AXI PLC

El fichero top del sistema (plc_ip_v1_0.vhd) ha de ser modificado además para definir las señales y genéricos necesarios.

```
-- Users to add parameters here
TID_LENGTH      : integer := 8;
TDEST_LENGTH    : integer := 8;
TUSER_LENGTH    : integer := 8;
D_WIDTH_EXT     : integer := 64;  -- N Bits de entrada y salida
D_WIDTH_INT     : integer := 16;  -- N Bits de entrada al core
-- User parameters ends


-- Users to add ports here
  -- Ports of Axi Stream Interface
  rstn        : in std_logic;
  clk         : in std_logic;

  s_tdata     : in std_logic_vector(D_WIDTH_EXT-1 downto 0);
  s_tvalid    : in std_logic;
  s_tready    : out std_logic;
  s_tlast     : in std_logic;
  s_tkeep     : in std_logic_vector((D_WIDTH_EXT/8)-1 downto 0);
  s_tid       : in std_logic_vector(TID_LENGTH-1 downto 0);
  s_tdest     : in std_logic_vector(TDEST_LENGTH-1 downto 0);
  s_tuser     : in std_logic_vector(TUSER_LENGTH-1 downto 0);

  m_tdata     : out std_logic_vector(D_WIDTH_EXT-1 downto 0);
  m_tvalid    : out std_logic;
  m_tready    : in std_logic;
  m_tlast     : out std_logic;
  m_tkeep     : out std_logic_vector((D_WIDTH_EXT/8)-1 downto 0);
  m_tid       : out std_logic_vector(TID_LENGTH-1 downto 0);
  m_tdest     : out std_logic_vector(TDEST_LENGTH-1 downto 0);
  m_tuser     : out std_logic_vector(TUSER_LENGTH-1 downto 0);
-- User ports ends
```

De vuelta a la ventana de IP Packager, en la pestaña “Compatibility” se podrán seleccionar los dispositivos y familias de Xilinx compatibles, así como el estado de desarrollo en el que se encuentra el IP Core como se muestra en la Figura 3-27. Es este caso se seleccionarán los dispositivos Zynq.

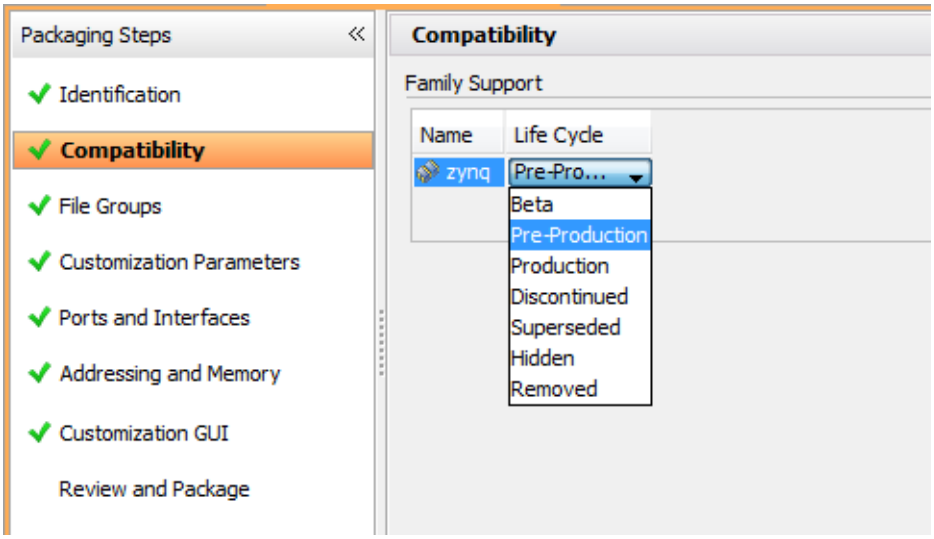

Figura 3-27: Pestaña Compatibilidad del IP

En la pestaña "Configuration Parameters" se definirán los distintos parámetros del IP Core, que servirán para configurar los genéricos del IP. Se definirá también su valor por defecto, aunque éste se podrá modificar en la interfaz gráfica de configuración del IP cuando sea añadido al diseño en Vivado.

Habrá que añadir los parámetros que aparecen en el diseño del AXI PLC, que son aquellos que sirven para configurar el número de bits de entrada y salida del periférico, el número de bits del bus de datos interno del periférico, y de las señales TID, TDEST y TUSER. Por otra parte también se añadirán los parámetros de configuración de la interfaz AXI4-Lite. El resultado se muestra en la Figura 3-28.

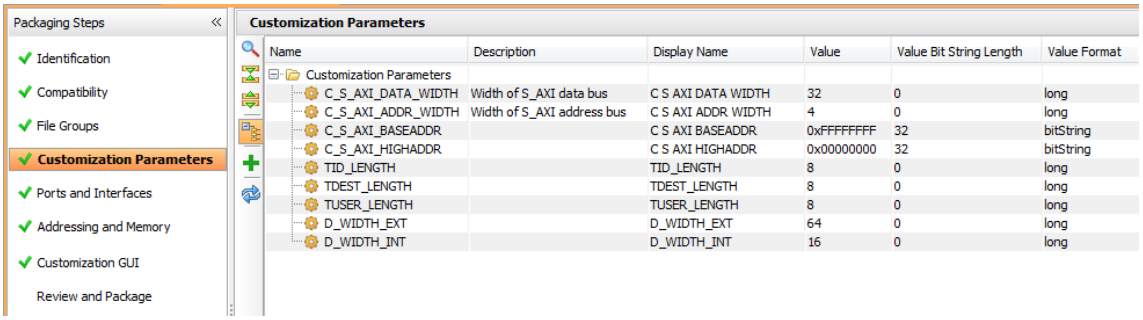

Figura 3-28: Parámetros de configuración del IP

En la pestaña "Ports and Interfaces" se configurarán los puertos de entrada y salida del IP. Será necesario mapear las señales de las distintas interfaces (S_AXI, S_AXIS y M_AXIS) con las señales del diseño, incluyendo las señales de Clock y Reset. Para ello se hace click derecho en una interfaz y se pincha en "Edit Interface". En la pestaña "Port Mapping" de la ventana emergente se podrá realizar el mapeo de puertos relacionando las señales de la izquierda (Puertos de la interfaz) con las de la derecha (Puertos del IP) como se muestra en la Figura 3-29.

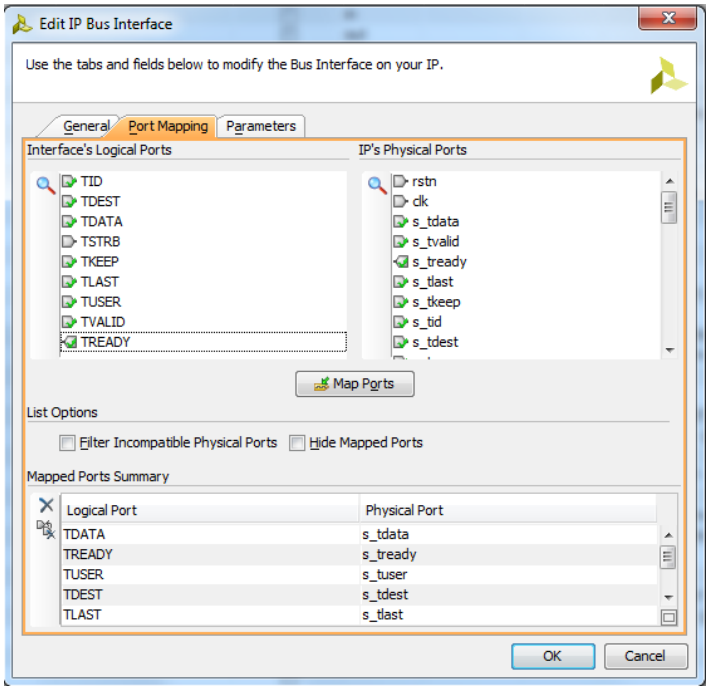

Figura 3-29: Mapeo de puertos de las interfaces

Las señales de Reloj y Reset, además de mapearlas a las señales del diseño, es muy importante configurar los siguientes parámetros para que no aparezcan errores más adelante.

- En las señales de Reset habrá que configurar el parámetro **POLARITY** como ACTIVE_LOW para definir la polaridad de esta señal y hacerla activa a nivel bajo como indica la Figura 3-30

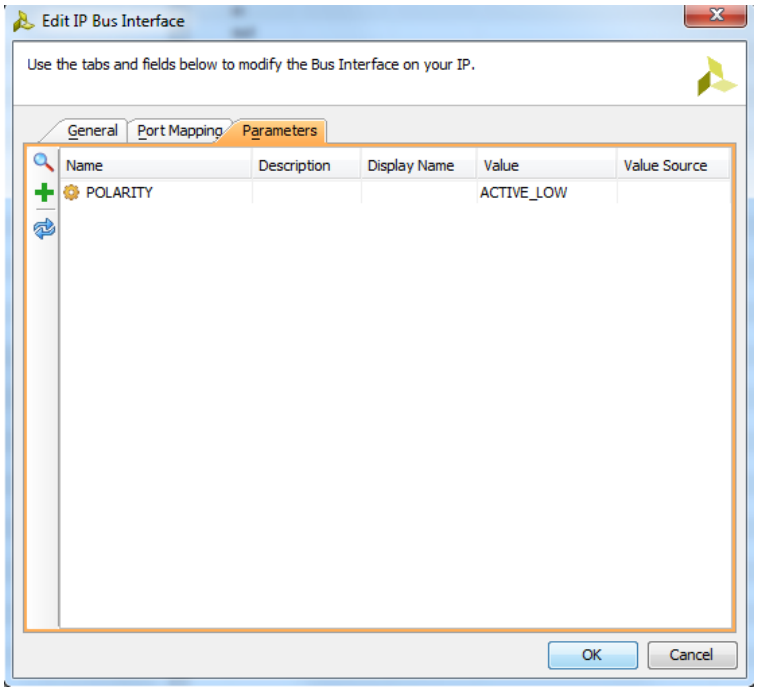

Figura 3-30: Parámetro POLARITY en las señales de Reset

- En las señales de Reloj habrá que configurar los siguientes dos parámetros:
    - **ASSOCIATED_RESET:** Indica la lista de señales de Reset asociadas a dicha señal de reloj.
    - **ASSOCIATED_BUSIF:** Nombre de las interfaces, separadas por dos puntos, que funcionan con esta señal de reloj. En las señales de reloj de la interfaz AXI4-Stream el valor de este parámetro será S_AXIS:M_AXIS, mientras que en la interfaz del tipo AXI4-Lite solo tendrá el valor S_AXI.

La Figura 3-31 muestra la configuración de los parámetros anteriores para las interfaces de tipo AXI4-Stream.

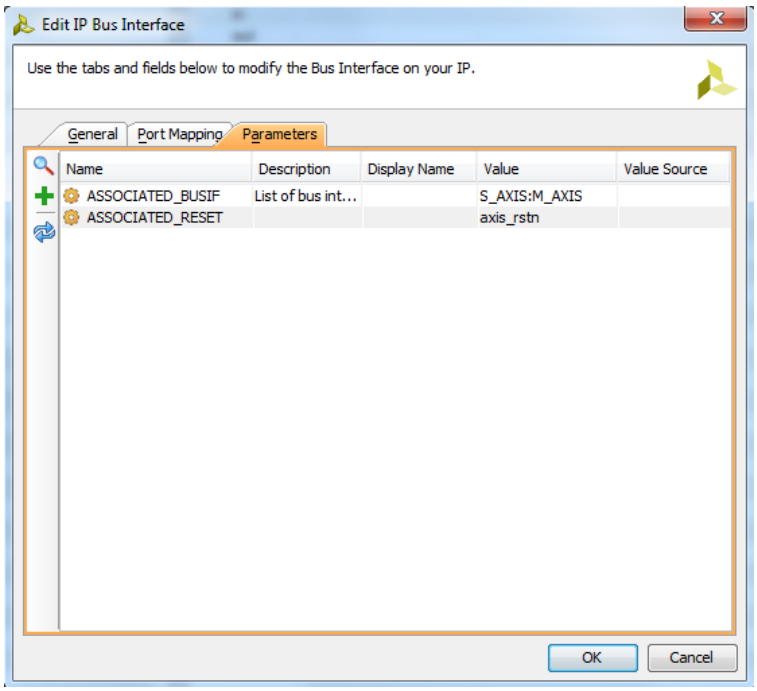

Figura 3-31: Parámetros de las señales de Reloj

La lista de interfaces con sus señales asociadas tras la configuración quedará finalmente como se muestra en la Figura 3-32.

Name	Interface Mode	Is Declaration	Direct...	Size Left	Size Left Dependency	Size Right	Type Name
S_AXI	slave						
s_axi_awaddr			in	3	(C_S_AXI_ADDR_WIDTH - 1)	0	std_logic_vector
s_axi_awprot			in	2		0	std_logic_vector
s_axi_awvalid			in				std_logic
s_axi_awready			out				std_logic
s_axi_wdata			in	31	(C_S_AXI_DATA_WIDTH - 1)	0	std_logic_vector
s_axi_wstrb			in	3	((C_S_AXI_DATA_WIDTH / 8) - 1)	0	std_logic_vector
s_axi_wvalid			in				std_logic
s_axi_wready			out				std_logic
s_axi_bresp			out	1		0	std_logic_vector
s_axi_bvalid			out				std_logic
s_axi_bready			in				std_logic
s_axi_araddr			in	3	(C_S_AXI_ADDR_WIDTH - 1)	0	std_logic_vector
s_axi_arprot			in	2		0	std_logic_vector
s_axi_arvalid			in				std_logic
s_axi_arready			out				std_logic
s_axi_rdata			out	31	(C_S_AXI_DATA_WIDTH - 1)	0	std_logic_vector
s_axi_rresp			out	1		0	std_logic_vector
s_axi_rvalid			out				std_logic
s_axi_rready			in				std_logic
S_AXIS	slave						
s_axis_tdata			in	63	(D_WIDTH_EXT - 1)	0	std_logic_vector
s_axis_tready			out				std_logic
s_axis_tuser			in	7	(TUSER_LENGTH - 1)	0	std_logic_vector
s_axis_tdest			in	7	(TDEST_LENGTH - 1)	0	std_logic_vector
s_axis_tlast			in				std_logic
s_axis_tvalid			in				std_logic
s_axis_tkeep			in	7	((D_WIDTH_EXT / 8) - 1)	0	std_logic_vector
s_axis_tid			in	7	(TID_LENGTH - 1)	0	std_logic_vector
M_AXIS	master						
m_axis_tdata			out	63	(D_WIDTH_EXT - 1)	0	std_logic_vector
m_axis_tready			in				std_logic
m_axis_tuser			out	7	(TUSER_LENGTH - 1)	0	std_logic_vector
m_axis_tdest			out	7	(TDEST_LENGTH - 1)	0	std_logic_vector
m_axis_tlast			out				std_logic
m_axis_tvalid			out				std_logic
m_axis_tkeep			out	7	((D_WIDTH_EXT / 8) - 1)	0	std_logic_vector
m_axis_tid			out	7	(TID_LENGTH - 1)	0	std_logic_vector
Clock and Reset Signals							
S_AXIS_RST	slave						
axis_rstn			in				std_logic
S_AXIS_CLK	slave						
M_AXIS_RST	slave						
axis_rstn			in				std_logic
M_AXIS_CLK	slave						
axis_clk			in				std_logic
S_AXI_RST	slave						
s_axi_aresetn			in				std_logic
S_AXI_CLK	slave						
s_axi_aclk			in				std_logic

Figura 3-32: Resultado final tras mapear las interfaces con las señales del sistema

Por último, en la pestaña “Customization GUI” se creará la interfaz gráfica de usuario (GUI) del IP, de forma que se puedan definir los distintos parámetros del periférico una vez esté instanciado en el diseño en Vivado. Se crearán varios apartados en la GUI, en este caso se han dividido los parámetros según el tipo de interfaz (AXI4-Stream y AXI4-Lite). El resultado se muestra en la Figura 3-33.

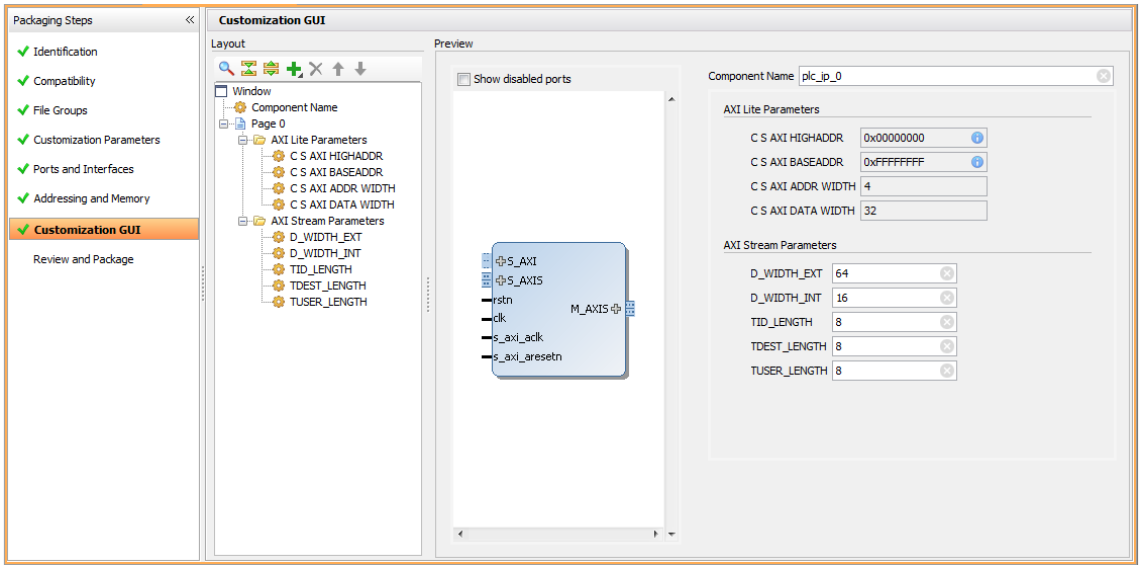

Figura 3-33: Creación de la GUI de configuración de parámetros del IP

Para finalizar la creación del periférico, se procederá al empaquetado del IP en la última pestaña: "Review and Package". Hecho esto, ya se dispone del IP Core en la carpeta del repositorio del proyecto (**ip_repo**), que contendrá todos los IP creados. Para reutilizarlos en un proyecto de Vivado será necesario añadir la ruta de dicho repositorio en las opciones de proyecto, tal y como se muestra en la Figura 3-34.

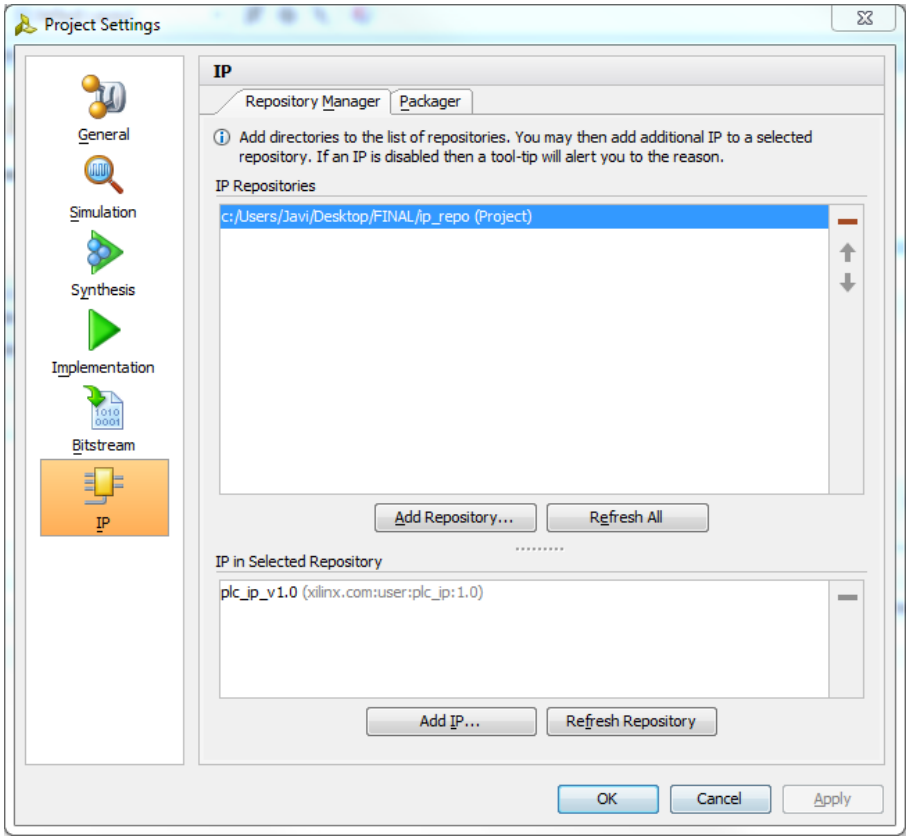

Figura 3-34: Repositorios de IPs del proyecto

3.3 Simulación con el BFM

Para verificar que el IP funciona correctamente, se realizará una simulación empleando el BFM para modelar el bus de comunicaciones AXI4-Stream y poder enviar y recibir transacciones que cumplan con las especificaciones del protocolo.

El escenario de prueba para verificar el IP Core del cliente DMA empaquetado con Vivado será el siguiente:

A la entrada del dispositivo bajo test (DUT) se situará un componente BFM actuando como maestro. Este BFM se encargará de realizar transferencias de datos y paquetes hacia el DUT. A la salida se instanciará otro componente BFM actuando como esclavo, que se encargará de la recepción de los datos de la interfaz de salida.

La simulación tendrá como base el ejemplo de la interfaz AXI4-Stream proporcionado por Xilinx con el manual del IP core AXI BFM [Xilinx, 2014a], donde el escenario es similar al que se quiere simular para probar el periférico creado. En la Figura 3-35 se muestra el escenario de prueba para verificar el IP:

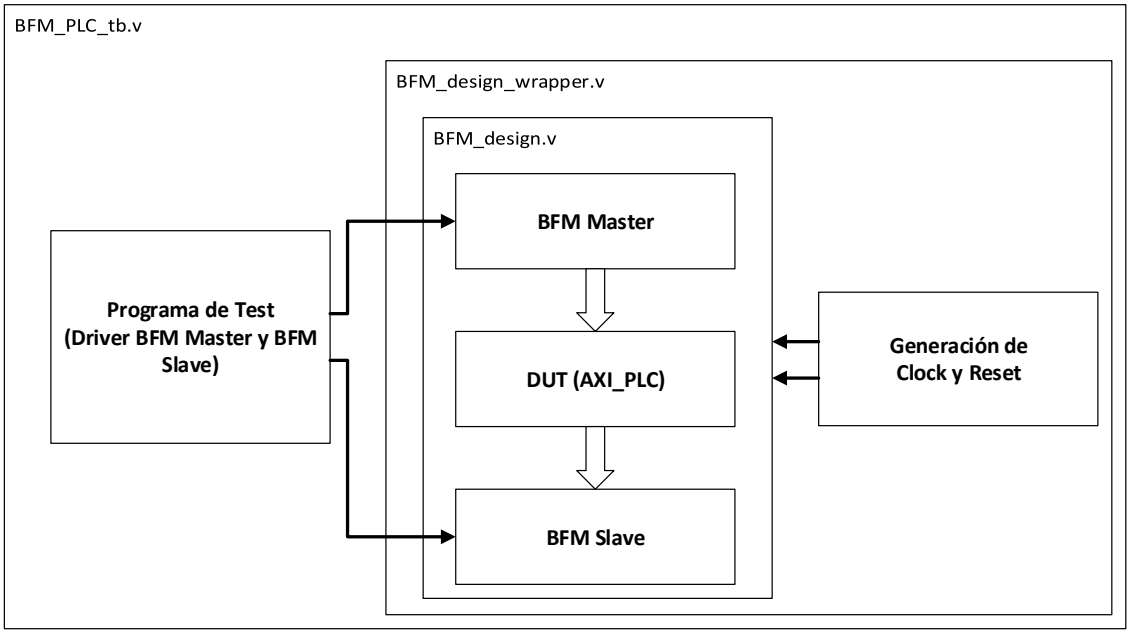

Figura 3-35: Escenario de prueba con los BFMs

En la prueba a realizar se comprobará que los datos obtenidos en la interfaz AXI4-Stream de salida coinciden con los datos enviados por el BFM. De esta manera, se puede verificar que el funcionamiento del IP creado es correcto, y por tanto, que el downsizer, registro intermedio y upsizer trabajan de la manera esperada.

3.3.1 Diseño de Bloques en Vivado

Antes de comenzar con la simulación será necesario definir el escenario de prueba en un diseño de bloques con Vivado, donde se instanciarán los los BFM necesarios para la verificación y el DUT. Se creará un nuevo diseño de bloques en un proyecto nuevo de Vivado y se añadirán los IPs, realizando las interconexiones necesarias entre ellos. La Figura 3-36 muestra el diseño de bloques del escenario de prueba diseñado en Vivado.

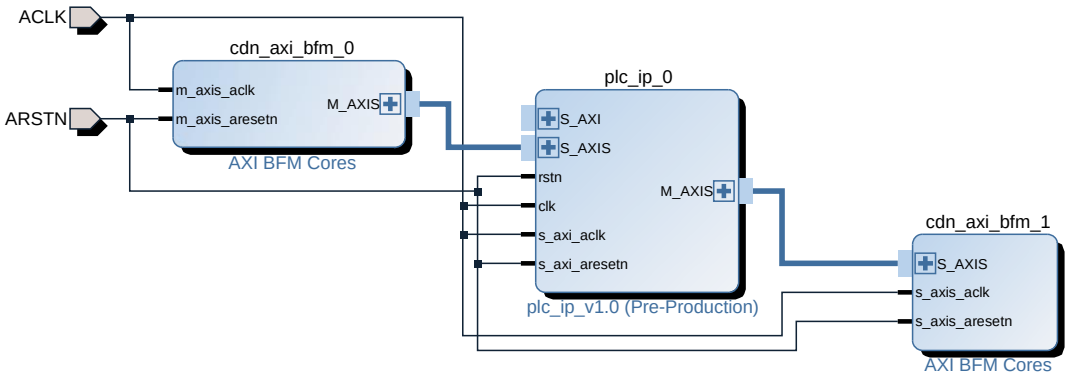

Figura 3-36: Diseño de bloques en Vivado para la simulación con los BFMs

Habrá que definir los distintos parámetros de los BFM para configurarlos de forma que se cumplan las especificaciones de la simulación que se desea realizar. Éstos se configuran en la interfaz gráfica “Customize IP” del componente, aunque también podrán ser modificados durante el testbench mediante funciones de la Utility API.

Para más información de los parámetros de los BFM del tipo AXI4-Stream consúltese el apartado 2.7.1.

La configuración de los parámetros del BFM Maestro y su interfaz AXI4-Stream se muestra en la Figura 3-37 y la Figura 3-38.

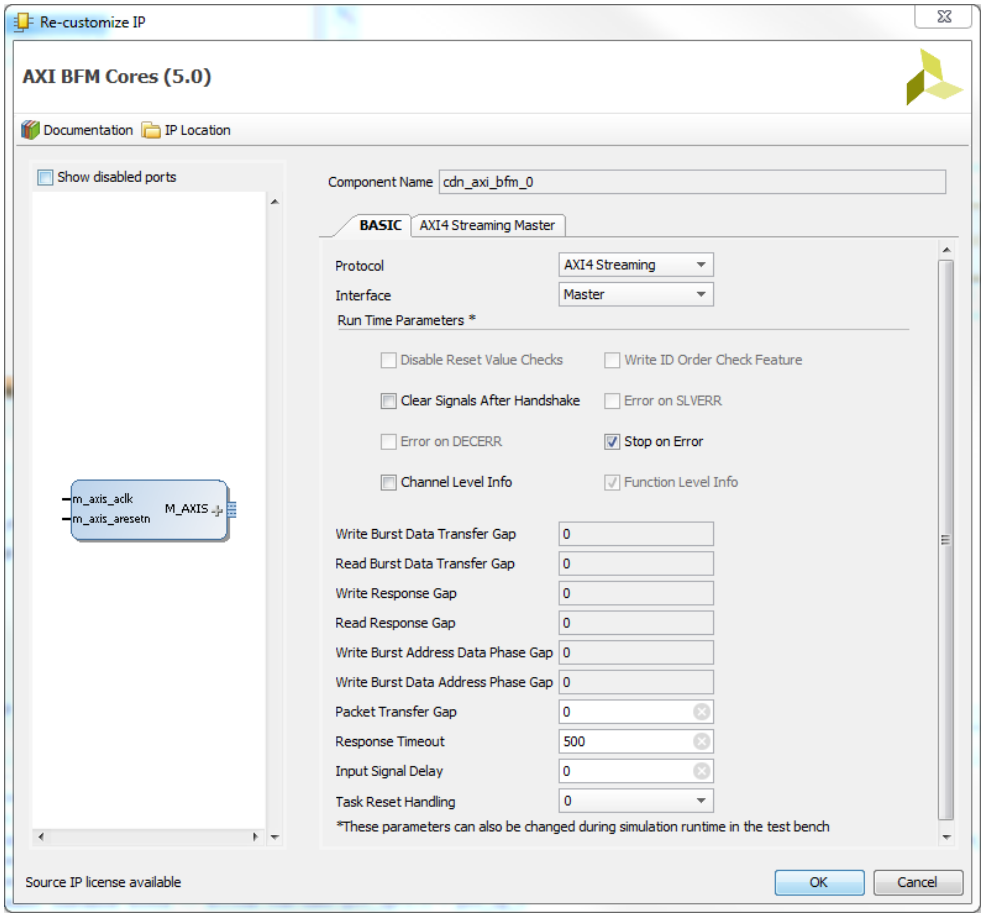

Figura 3-37: Configuración básica del BFM Maestro

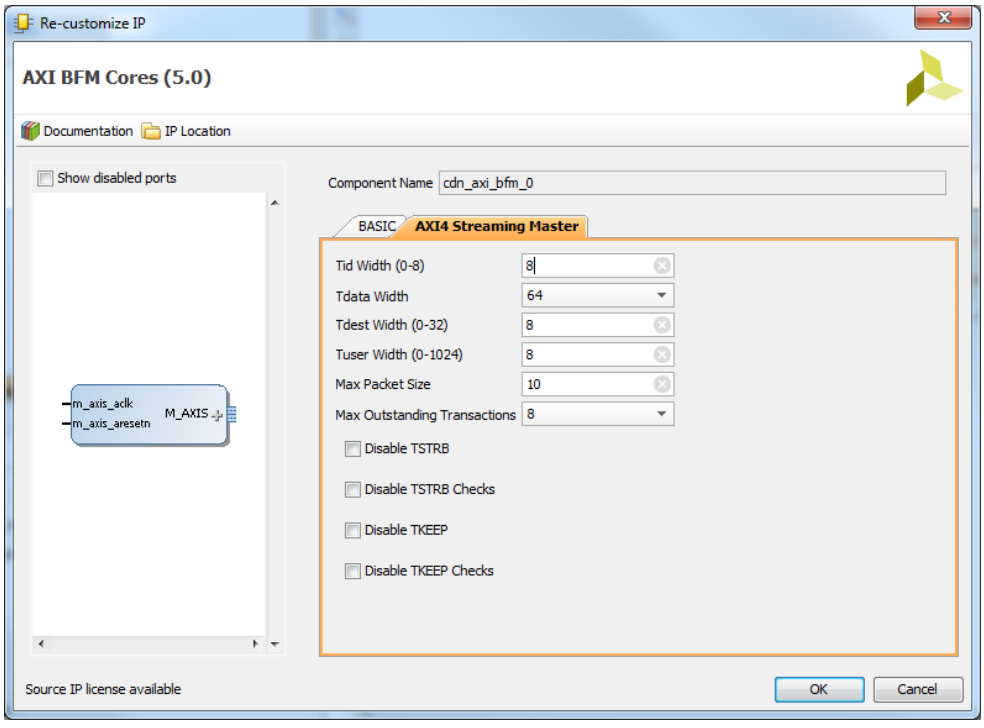

Figura 3-38: Configuración de la interfaz AXI4-Stream del BFM Maestro

La configuración de los parámetros del BFM Esclavo y su interfaz AXI4-Stream se muestra en la Figura 3-39 y la Figura 3-40.

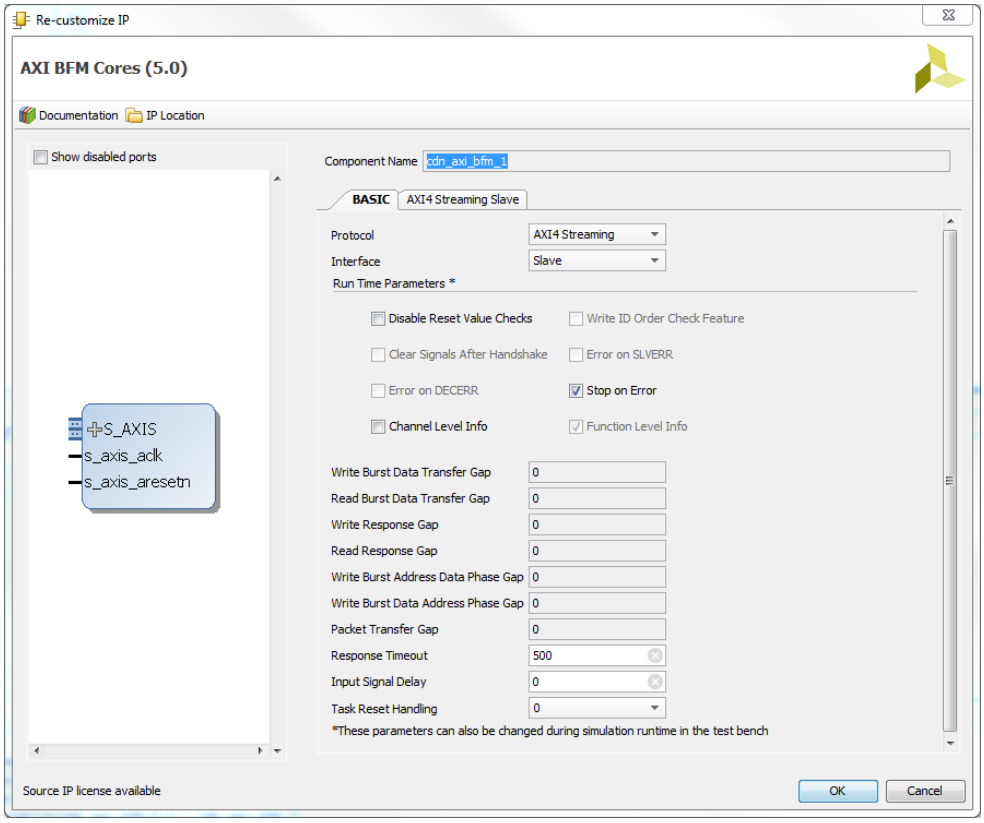

Figura 3-39: Configuración básica del BFM Esclavo

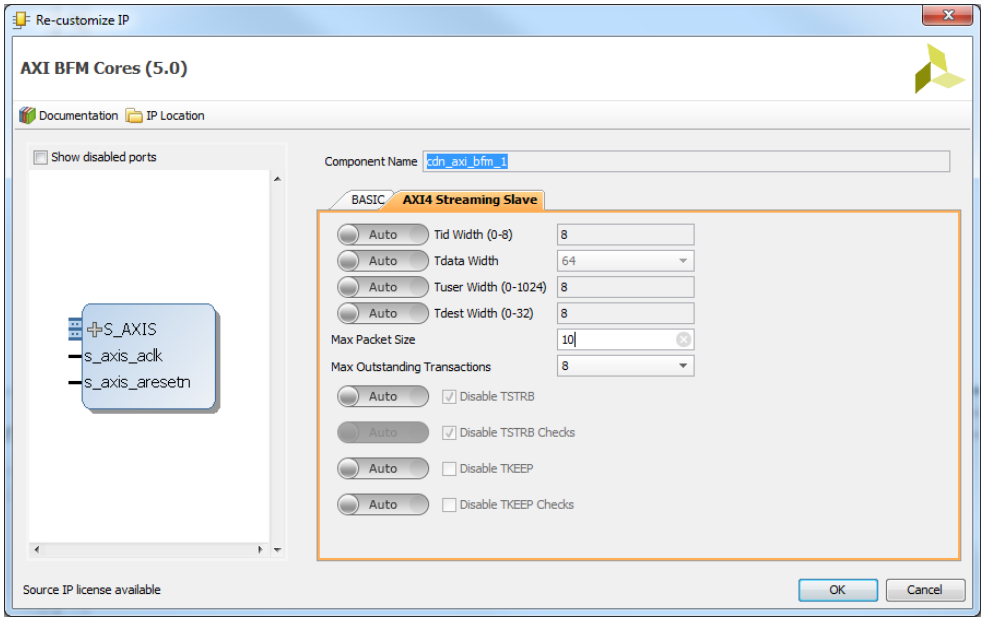

Figura 3-40: Configuración de la interfaz AXI4-Stream del BFM Esclavo

También será necesario configurar el IP del cliente DMA con los parámetros adecuados para definir el tamaño de los buses de sus interfaces. Se utilizará un bus de datos de entrada de 64 bits, y un bus intermedio de 16 bits. El tamaño de las señales TID, TDEST y TUSER en esta simulación será el tamaño por defecto (8bits). La Figura 3-41 muestra la interfaz de configuración de los parámetros del PLC IP.

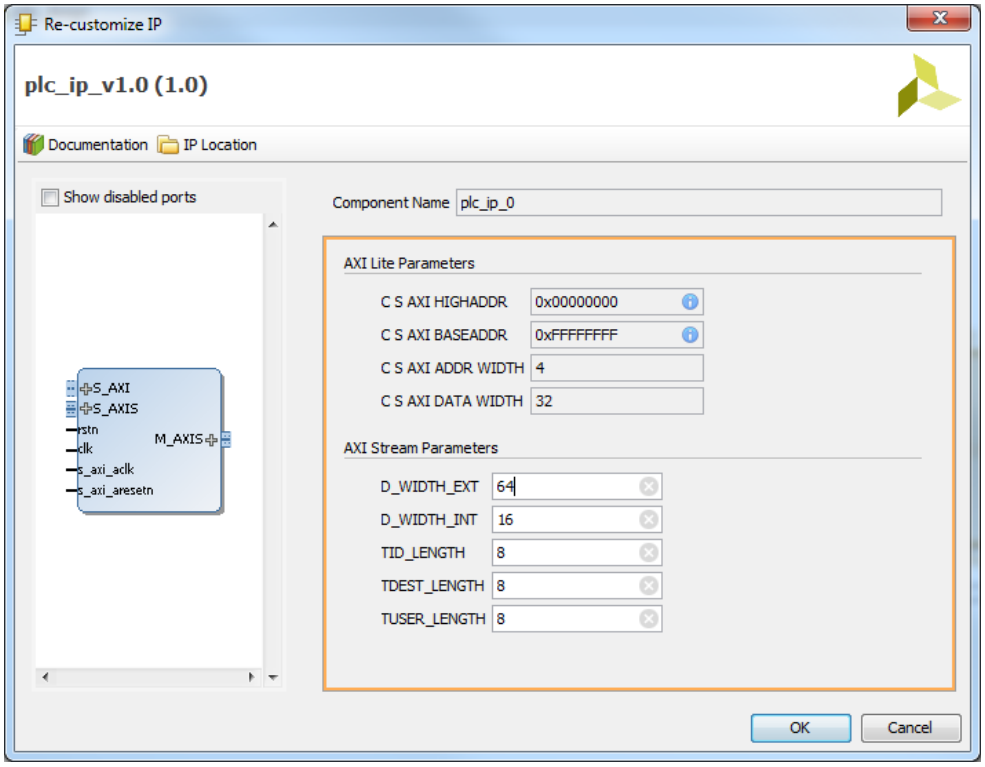

Figura 3-41: Configuración del PLC IP

3.3.2 Control de los BFM en simulación

A continuación se explicará mediante el uso de extractos de código, cómo realizar la configuración y llevar a cabo el control de los BFM para poder simular transferencias del BFM maestro al DUT y realizar las lecturas en el BFM esclavo.

Una de las características de este BFM es que el control se debe realizar mediante funciones escritas en SystemVerilog, ya que el BFM, provisto por Xilinx, fue codificado en este lenguaje, por lo que el control también debe realizarse utilizando SystemVerilog. Para más información acerca de SystemVerilog, consúltese [Sutherland et al., 2006].

En el testbench debe tenerse en cuenta la jerarquía de archivos de la simulación en la llamada a las funciones de control para que no se produzcan errores. En la Figura 3-42 se muestra la jerarquía de ficheros del diseño:

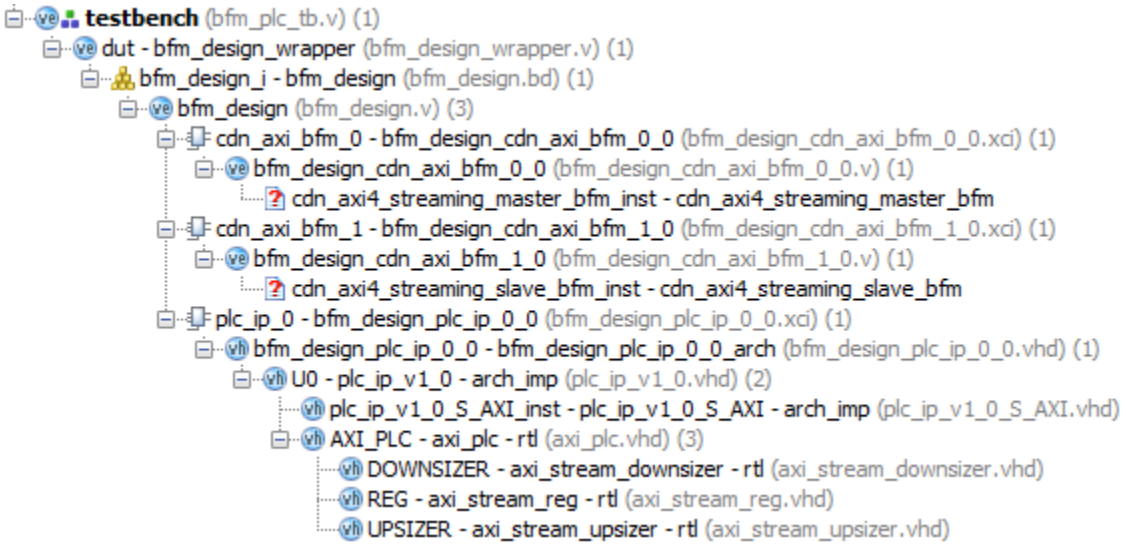

Figura 3-42: Jerarquía de ficheros en simulación BFM

Las funciones de las API deben ser llamadas desde el fichero testbench con la siguiente sintaxis:

Para el BFM Slave:
dut.bfm_design_i.cdn_axi_bfm_1.cdn_axi4_streaming_slave_bfm_inst.FUNCION().

Para el BFM Master:
dut.bfm_design_i.cdn_axi_bfm_0.cdn_axi4_streaming_master_bfm_inst.FUNCION().

3.3.2.1 Fichero Wrapper (dut)

Para realizar el testbench se seguirá la estructura adoptada por Xilinx en los ejemplos que proporciona. Como se observa en la Figura 3-35, se dispone de un fichero llamado *bfm_design_wrapper.v,* donde se instanciará el diseño de bloques hecho en Vivado, y en él se generarán las señales de reloj y reset. A continuación se explicará el contenido de este fichero.

En primer lugar se debe indicar la base de tiempos de la simulación. Esa es la finalidad de la siguiente línea de código, donde el primer valor indica la referencia de tiempos, y el segundo valor la precisión.

```
`timescale 1 ns / 1 ps
```

En este fichero se instanciará el módulo del diseño de bloques realizado con Vivado, donde se encuentran los BFM y el DUT. Para ello se emplea el siguiente código, el cual indica que dispone de dos puertos: CLK y RSTn.

```
bfm_design bfm_design_i
       (.ACLK(ACLK),
        .ARSTN(ARSTN));
```

Se generará un reloj cuyo periodo sea 20 ns (50 MHz) y un reset activo a nivel bajo que se activará durante los primeros 50ns de la simulación.

```
// Generador de Reset
initial begin
   ARSTN = 1'b0;
   #50;
   @(posedge ACLK);
      ARSTN = 1'b1;
   @(posedge ACLK);
end

// Generador de Clk
initial ACLK = 1'b0;
always #10 ACLK = !ACLK;
```

Dentro de este fichero también se incluyen algunas tareas escritas en SystemVerilog, que se emplearán durante la simulación para comprobar que los valores de las señales en las distintas transferencias de datos son correctos.

Estas tareas permiten comparar los valores de los estímulos recibidos en el BFM esclavo con los valores enviados por el BFM maestro de: TDATA, TUSER, TID, TDEST, TSTRB, TKEEP, TLAST, TDATA_VECTOR (Vector con los valores TDATA de un paquete), TUSER_VECTOR (Vector con los valores TUSER de un paquete) y DATASIZE. También se incluyen dos tareas de reporte de datos, las cuales imprimen un mensaje indicando si el test se ha realizado correctamente o no.

3.3.2.2 Fichero de Simulación (testbench)

En el fichero de testbench (*bfm_plc_tb.v)* se realizan todas las operaciones necesarias para la simulación, como pueden ser las transferencias y la configuración de parámetros de los BFM. Este fichero constituye el nivel más alto de la jerarquía vista anteriormente y en él debe estar instanciado el fichero Wrapper de la siguiente manera.

```
bfm_design_wrapper dut();
```

Se necesitará crear e inicializar un conjunto de variables que serán utilizadas para manejar las señales de la interfaz AXI4-Stream, con la finalidad de realizar transferencias de datos. Se puede ver la inicialización de algunos de los vectores a continuación:

```
initial begin

  // Vectores de datos para transferencias simples
  for (i=0; i < (DATA_BUS_WIDTH/8); i=i+1) begin
    mtestDATA[0][i*8 +: 8] = 8'h00;
    mtestDATA[1][i*8 +: 8] = 8'hAA;
    mtestDATA[2][i*8 +: 8] = 8'h55;
    mtestDATA[3][i*8 +: 8] = 8'hFF;
  end

  // Vectores TKEEP y TSTRB
  for (j=0; j < (DATA_BUS_WIDTH/8); j=j+1) begin
    all_valid_strobe[j] = 1'b1;
    all_valid_keep[j] = 1'b1;
  end

  // Vector TUSER
  for (j=0; j < USER_BUS_WIDTH; j=j+1) begin
```

```
    mtestUSER = 1'b1;
  end

  // Vectores de datos para paquetes
  for (j=0; j < ((DATA_BUS_WIDTH*(MAX_PACKET_SIZE))/8); j=j+1) begin
    v_mtestDATA[j*8 +: 8] = j;
  end

// Vector TUSER para paquetes
  for (j=0; j < ((USER_BUS_WIDTH*(MAX_PACKET_SIZE))/8); j=j+1) begin
    v_mtestUSER[j*8 +: 8] = j;
  end

end
```

Antes de comenzar con las transferencias de datos, también sería recomendable configurar los parámetros necesarios de los BFM, en el caso en el que no hubieran sido configurados en el diseño de Vivado. Por ejemplo, la función *set_channel_level_info(),* configura el parámetro CHANNEL_LEVEL_INFO para habilitar el reporte de mensajes de los BFM.

```
// Habilitamos durante el testbench los mensajes de información
dut.bfm_design_i.cdn_axi_bfm_0.cdn_axi4_streaming_master_bfm_inst.set_channel_
level_info(1);

dut.bfm_design_i.cdn_axi_bfm_1.cdn_axi4_streaming_slave_bfm_inst.set_channel_l
evel_info(1);
```

Una vez realizada toda la configuración, se procede a comandar las transferencias de datos en el BFM maestro y a leer los resultados recibidos en el BFM esclavo. Se dispondrá de dos procesos concurrentes, uno para controlar cada BFM. A continuación se explican en detalle los pasos a seguir para realizar una transferencia simple y una transferencia de un paquete.

Transferencia Simple

Para realizar una transferencia simple con los datos contenidos en el vector mtestDATA[0], todos los bytes válidos indicados por mtestKEEP, TID=4, TDEST=2 y TLAST=1 se utilizará el siguiente código, donde se inicializarán las variables a los valores anteriores y posteriormente se realizará la transferencia mediante la función SEND_TRANSFER.

```
mtestID = 4;
mtestDEST = 2;
mtestSTRB = all_valid_strobe;
mtestKEEP = all_valid_keep;
mtestLAST = 1;

dut.bfm_design_i.cdn_axi_bfm_0.cdn_axi4_streaming_master_bfm_inst.SEND_TRANSFE
R(mtestID, mtestDEST, mtestDATA[0], mtestSTRB, mtestKEEP, mtestLAST,
mtestUSER);

wait(slave_finished_example == 1'b1);
```

En el BFM esclavo se recibirá esta transferencia y los valores de las distintas señales recibidas son comparados con los valores esperados. En la recepción no se tendrán en cuenta los valores de TDEST y TID, ya que las transferencias realizadas van dirigidas hacia el mismo destinatario, no se intercalan con otras transferencias, por lo que los parámetros de entrada TDESTValid y TIDValid de la función RECEIVE_TRANSFER serán 0.

Una vez comparados los vectores recibidos con los esperados, se marcará un flag (slave_finished_example) para indicar al maestro que puede enviar otro dato.

```
dut.bfm_design_i.cdn_axi_bfm_1.cdn_axi4_streaming_slave_bfm_inst.RECEIVE_TRANS
FER(0, 0, 0, 0, stestID, stestDEST, stestDATA[0], stestSTRB, stestKEEP,
stestLAST, stestUSER);

// Comparamos los datos recibidos con los enviados
dut.COMPARE_TID(4,stestID);
dut.COMPARE_TDEST(2,stestDEST);
dut.COMPARE_TDATA(mtestDATA[0],stestDATA[0]);
dut.COMPARE_TKEEP(all_valid_keep,stestKEEP);
dut.COMPARE_TLAST(1,stestLAST);
dut.COMPARE_TUSER(mtestUSER,stestUSER);

// Slave Test completado.
slave_finished_example = 1'b1;
```

Transferencia de un paquete

La transferencia de un paquete de datos se realizará de forma similar a una transferencia simple, ya que un paquete de datos está formado por varias transferencias simples, y además, la API proporciona una función específica para realizar esta tarea. En el siguiente ejemplo se transmitirá un paquete con los datos v_mtestDATA que fueron inicializados al principio.

```
mtestID = 7;
mtestDEST = 8;
mtestDATASIZE = MAX_PACKET_SIZE*(DATA_BUS_WIDTH/8);

dut.bfm_design_i.cdn_axi_bfm_0.cdn_axi4_streaming_master_bfm_inst.SEND_PACKET(
mtestID, mtestDEST, v_mtestDATA, mtestDATASIZE, v_mtestUSER);

wait(slave_finished_example == 1'b1);
```

En el BFM esclavo se recibirá esta transferencia, y al igual que en el caso de la transferencia simple, se compararán los valores de las señales recibidas con los valores esperados. En la recepción tampoco se tendrán en cuenta los valores de TDEST y TID por lo que los parámetros TDESTValid y TIDValid de la función RECEIVE_PACKET serán 0.

Una vez comparados los vectores recibidos con los esperados, se marcará un flag (slave_finished_example) para indicar en la simulación que el maestro puede enviar otro dato.

```
dut.bfm_design_i.cdn_axi_bfm_1.cdn_axi4_streaming_slave_bfm_inst.RECEIVE_PACKE
T(0, 0, 0, 0, stestID, stestDEST, v_stestDATA[0], stestDATASIZE,
v_stestUSER[0]);

// Comparamos los datos recibidos con los enviados
dut.COMPARE_TID(7,stestID);
dut.COMPARE_TDEST(8,stestDEST);
dut.COMPARE_DATASIZE(mtestDATASIZE,stestDATASIZE);
dut.COMPARE_TDATA_VECTOR(v_mtestDATA,v_stestDATA[0]);
dut.COMPARE_TUSER_VECTOR(v_mtestUSER,v_stestUSER[0]);

// Slave Test completado.
slave_finished_example = 1'b1;
```

3.3.3 Test AXI4-Stream

Explicado el procedimiento para emplear el BFM en la verificación de un periférico, se procede a probar el funcionamiento del cliente DMA. Como se comprobó en la simulación realizada con el diseño completo, los datos a la entrada deben ser los mismos que se obtienen a la salida del IP, por lo que con la simulación con los BFM, también debe ocurrir esto.

Para comprobarlo, se llevarán a cabo 5 test diferentes para observar que el sistema responde correctamente ante distintas transferencias de datos y paquetes. Se utilizarán las funciones comentadas en el apartado anterior para controlar las transferencias y comparar los datos recibidos con los enviados.

Test 1: Transferencia de un mensaje simple. Tan solo se enviará un dato de 64 bits con todos sus bytes válidos y con TLAST=1 como se indica en la Tabla 3-4.

Tabla 3-4: Datos de la transferencia del Test 1: Transferencia Simple

TDATA	TKEEP	TLAST	TUSER	TID	TDEST
0x0000000000000000	11111111	1	0x01	0x04	0x02

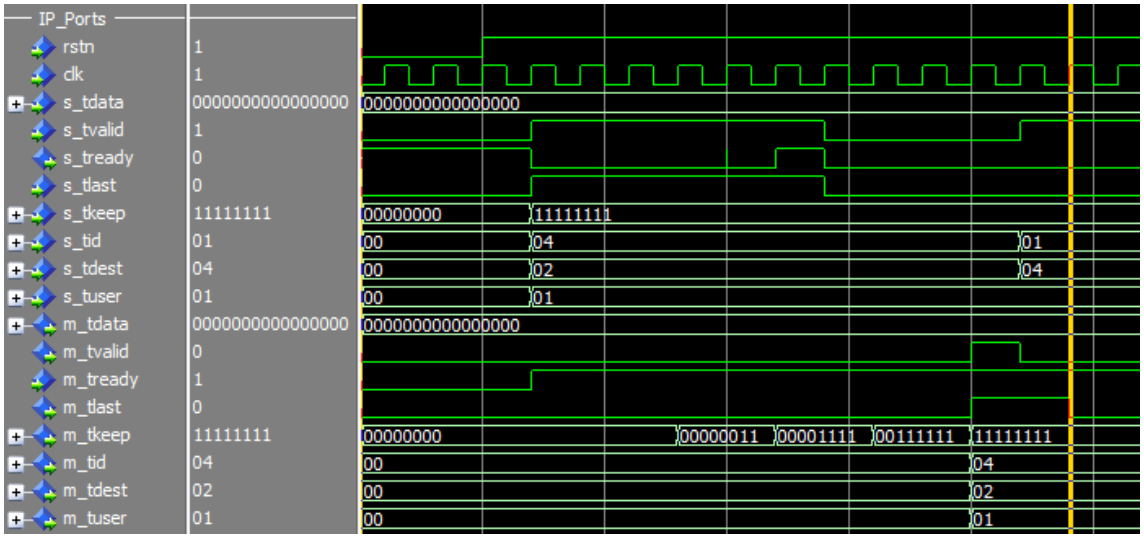

Figura 3-43: Resultado de simulación de Test 1: Transferencia Simple

Test 2: Envío de 4 transferencias seguidas. Se enviarán 4 datos, uno detrás de otro, con todos sus bytes válidos, y en el último envío se marcará TLAST = 1 como se indica en la Tabla 3-5.

Tabla 3-5: Datos de la transferencia del Test 2: Múltiples transferencias

TDATA	TKEEP	TLAST	TUSER	TID	TDEST
0x0000000000000000	11111111	0	0x01	0x01	0x04
0xAAAAAAAAAAAAAAAA	11111111	0	0x01	0x02	0x05
0x5555555555555555	11111111	0	0x01	0x03	0x06
0xFFFFFFFFFFFFFFFF	11111111	1	0x01	0x04	0x07

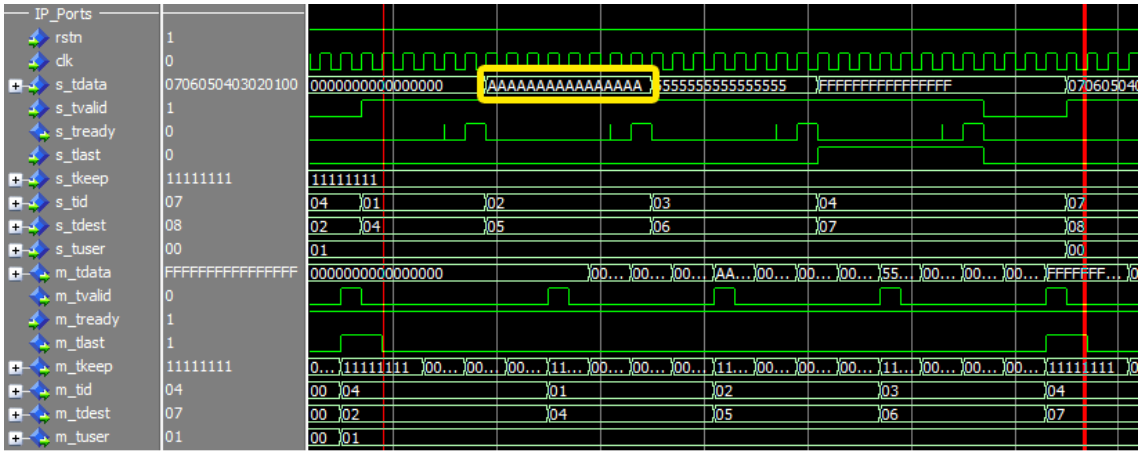

Figura 3-44: Resultado de simulación de Test 2: Múltiples transferencias

Si se amplía la Figura 3-44 en el dato marcado en amarillo se observa que el upsizer va empaquetando los datos de salida, pero hasta que el dato no está completo, no lo valida con la señal m_tvalid como se indica en la Figura 3-45.

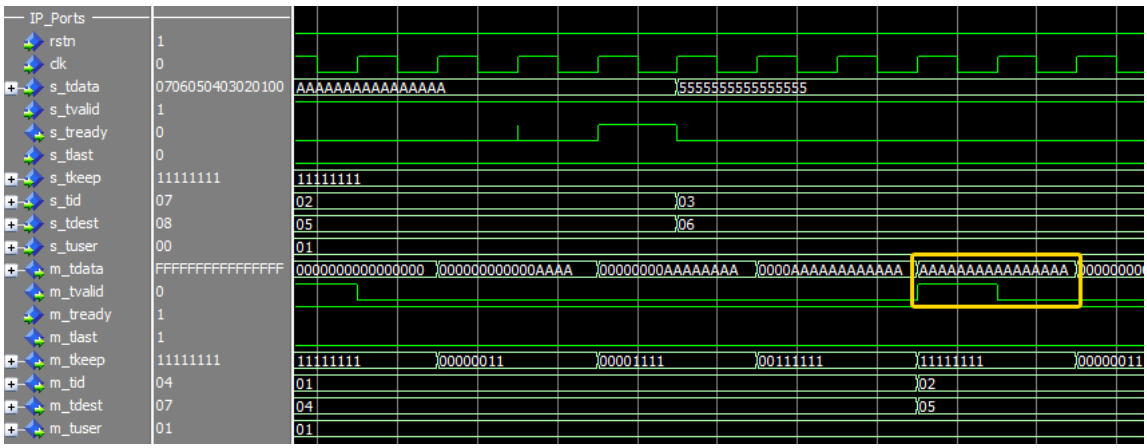

Figura 3-45: Test 2: Detalle de una transferencia del test 2

Test 3: Envío de un paquete de datos en una ráfaga. Se enviará un vector de datos que forma un paquete, en varias transferencias simples indicadas en la Tabla 3-6.

Tabla 3-6: Datos de la transferencia del Test 3: Envío de un paquete de datos

TDATA	TKEEP	TLAST	TUSER	TID	TDEST
0x0706050403020100	11111111	0	0x00	0x07	0x08
0x0F0E0D0C0B0A0908	11111111	0	0x01	0x07	0x08
…	…	…	…	…	…
0x4F4E4D4C4B4A4948	11111111	1	0x09	0x07	0x08

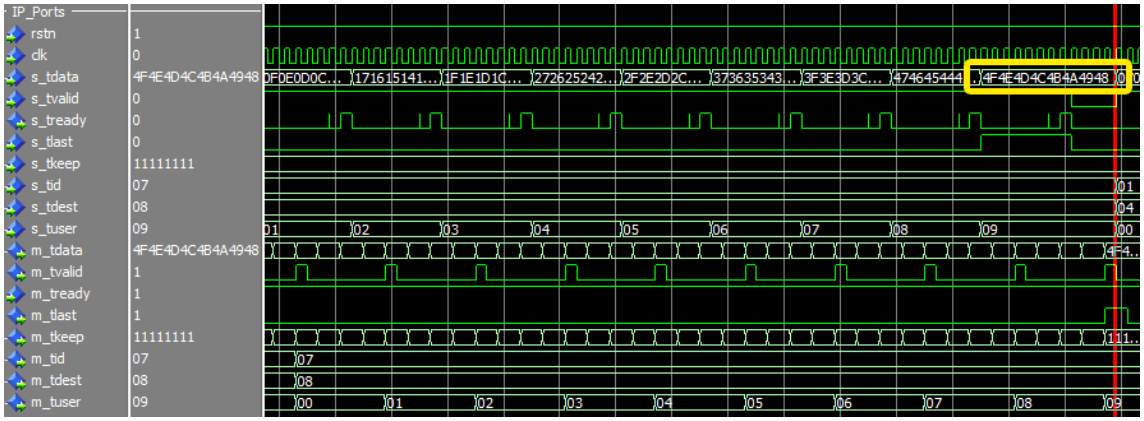

Figura 3-46: Resultado de simulación de Test 3: Envío de un paquete de datos

Si se amplía la Figura 3-46 en el último dato del paquete (marcado en amarillo), se observa al igual que antes, como el upsizer empaqueta los datos de salida, pero hasta que no tiene todos no los valida. Este último dato se marca en amarillo en la Figura 3-47.

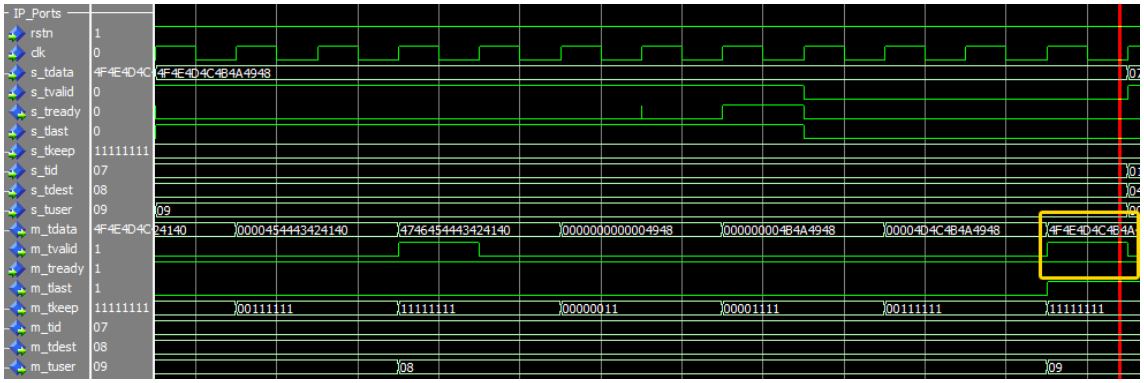

Figura 3-47: Test 3: Detalle del último dato del paquete del test 3

Test 4: Envío de 4 paquetes de datos seguidos. Se enviarán 4 paquetes de datos, uno detrás de otro, con todos sus datos válidos como se detalla en la Tabla 3-7.

Tabla 3-7: Datos de la transferencia del Test 4: Envío de múltiples paquetes

TDATA	TKEEP	TLAST	TUSER	TID	TDEST
Paquete 1					
0x0706050403020100	11111111	0	0x00	0x01	0x04
…	…	…	…	…	…
0x4F4E4D4C4B4A4948	11111111	1	0x09	0x01	0x04
Paquete 2					
0x0706050403020100	11111111	0	0x00	0x02	0x05
…	…	…	…	…	…
0x4F4E4D4C4B4A4948	11111111	1	0x09	0x02	0x05
Paquete 3					
0x0706050403020100	11111111	0	0x00	0x03	0x06
…	…	…	…	…	…
0x4F4E4D4C4B4A4948	11111111	1	0x09	0x03	0x06
Paquete 4					
0x0706050403020100	11111111	0	0x00	0x04	0x07
…	…	…	…	…	…
0x4F4E4D4C4B4A4948	11111111	1	0x09	0x04	0x07

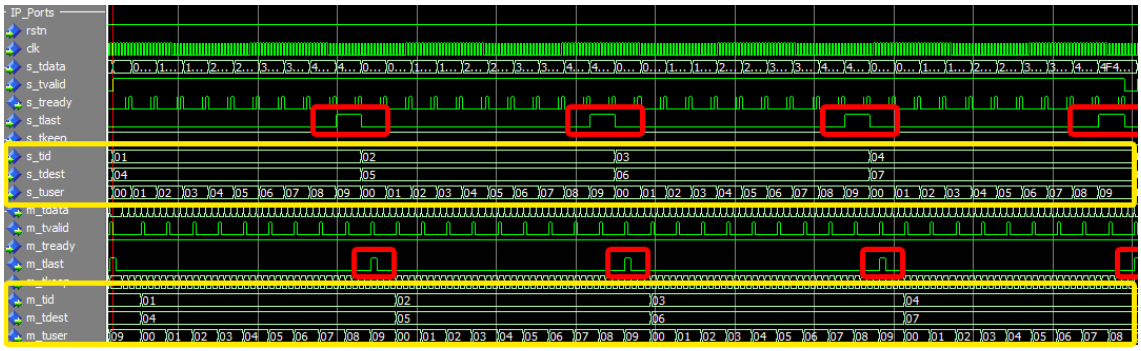

Figura 3-48: Resultado de simulación de Test 4: Envío de múltiples paquetes

En este test se envían 4 paquetes iguales al del Test 3. Se puede observar la generación de las señales TID, TDEST y TUSER marcada en amarillo en la Figura 3-48, así como los 4 pulsos marcados en rojo en la señal TLAST que indican el fin de cada paquete.

Test 5: Envío de un paquete de datos donde el último dato está incompleto o no todos sus bytes son válidos, es decir, la señal TKEEP no tiene todos los bits a 1. En este caso solo serán válidos los dos bytes menos significativos del bus TDATA. En la Tabla 3-8 se indican las distintas transferencias a realizar.

Tabla 3-8: Datos de la transferencia del Test 5: Envío de un paquete incompleto

TDATA	TKEEP	TLAST	TUSER	TID	TDEST
0x0706050403020100	11111111	0	0x00	0x07	0x08
0x0F0E0D0C0B0A0908	11111111	0	0x01	0x07	0x08
…	…	…	…	…	…
0x4F4E4D4C4B4A4948	00000011	1	0x09	0x07	0x08

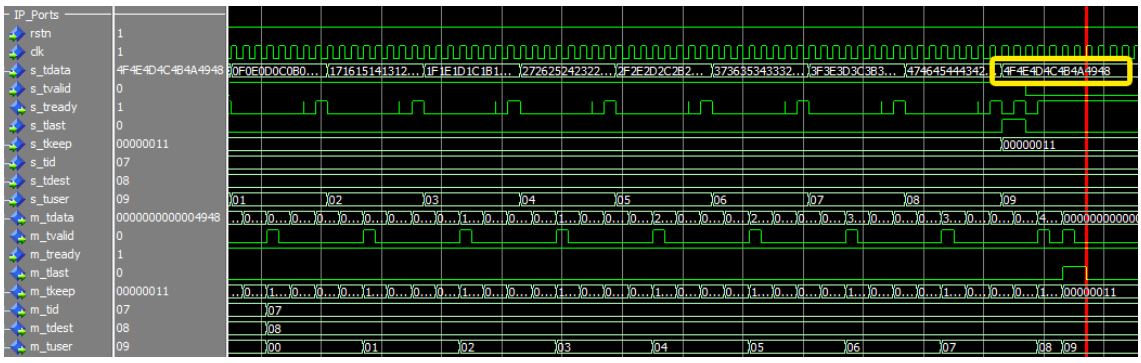

Figura 3-49: Resultado de simulación de Test 5: Envío de un paquete incompleto

Si se amplía la Figura 3-49 en el dato marcado en amarillo se puede ver como el último dato no está completo. Esto se indica en la señal TKEEP, marcando a 1 solo los bytes que son válidos, como se señala en la Figura 3-50.

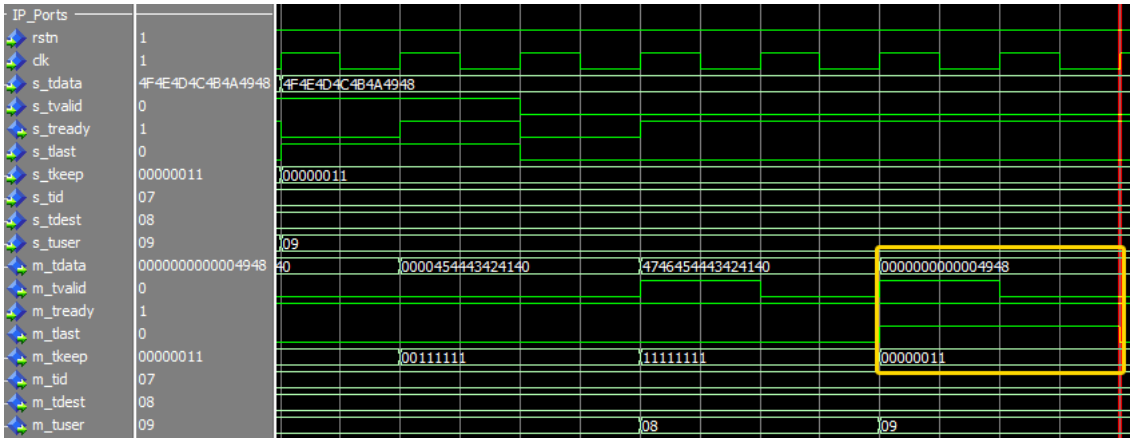

Figura 3-50: Detalle del último dato incompleto del test 5

3.3.4 Resultados Testbench BFM

Después de haber realizado los 5 tests, se podría dar por válido el diseño, ya que realiza las transferencias de datos sin detectar error en ninguno de los casos probados y los datos a la salida son los mismos que a la entrada. Sin embargo, el diseño no ha sido probado para todas las situaciones posibles que podrían surgir en la aplicación final, por lo que no se puede asegurar que el diseño funciona ante la totalidad de los casos.

Una situación que sería interesante simular sería el caso donde hay ciclos de espera entre dos transferencias, pero el API utilizado del BFM no permite introducir dichos ciclos de espera. Sin embargo, esta situación ya fue simulada con ModelSim sobre el diseño completo del periférico en el apartado 3.1.4.

3.4 Definición de la arquitectura del SoC

Habiendo verificado el periférico creado mediante simulación directa con Modelsim y mediante el empleo de componentes BFM en la generación de los estímulos para realizar distintas transferencias, se puede asegurar casi con total seguridad que el funcionamiento del cliente DMA es correcto.

Ha llegado el momento de crear un proyecto nuevo en Vivado e incorporar todos los periféricos y componentes necesarios en el diseño para realizar la transferencia con el DMA, y realizar las interconexiones necesarias para crear la arquitectura hardware del SoC.

Se debe crear un nuevo diseño por bloques e ir añadiendo los distintos IPs. Para más información acerca del diseño por bloques en Vivado con el IP Integrator, consúltese [Xilinx, 2013].

En primer lugar se añadirá el procesador (Zynq-7000 Processing System) al diseño para poder configurar las interfaces necesarias entre PS y PL, y los periféricos necesarios de Zynq.

En las opciones de configuración de la PS habrá que activar el puerto del ACP para comunicar el cliente DMA con la memoria teniendo coherencia de datos en las transferencias, activar los puertos de entrada a la PS para las interrupciones de la PL, y por último, asegurarse que la interfaz ENET 0 está activada para poder utilizar el puerto Ethernet con la finalidad de recibir y transmitir el streaming de datos.

En la pestaña “PS-PL Configuration” se activará la interfaz del ACP marcando las casillas que aparecen en la Figura 3-51. El motivo de marcar la casilla “Tie off AxUSER” es el ofrecer siempre coherencia en las lecturas y escrituras a través de dicho puerto, ya que como se explicó en el apartado 2.6, estas señales tienen que tener un valor determinado para disponer de coherencia de caché en lecturas y escrituras a memoria.

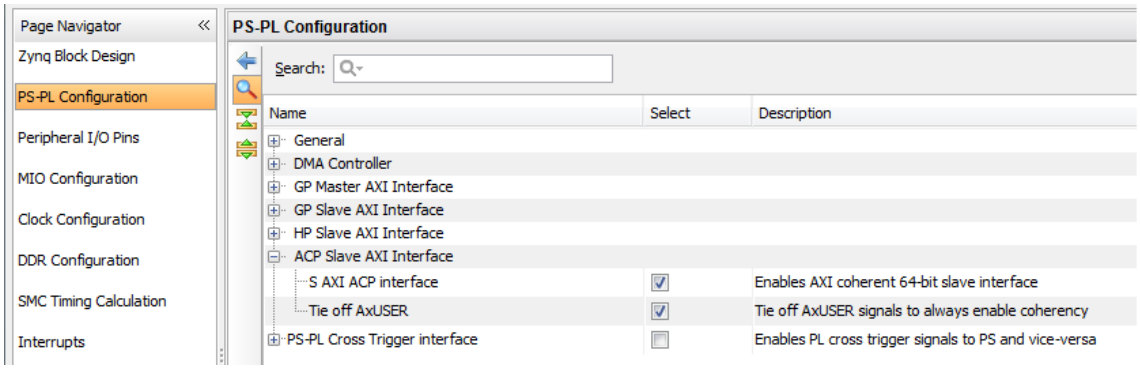

Figura 3-51: Configuración de las interfaces PS-PL

En la pestaña "Interrupts" se debe activar la opción de "Fabric Interrupts" para habilitar las interrupciones entre PS y PL, y después activar el puerto IRQ_F2P, como se indica en la Figura 3-52, con la finalidad de poder rutear señales de interrupción desde la PL hasta el PS. Se necesitarán dos señales de interrupción del DMA: una interrupción que marca el fin de transmisión por el canal MM2S, y otra que marca el fin de transmisión de los datos recibidos por el canal S2MM.

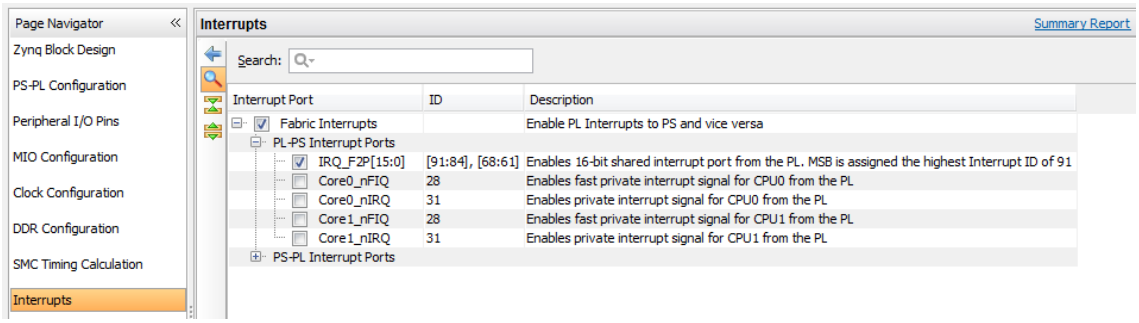

Figura 3-52: Configuración de los puertos de interrupción PS-PL

Por último, se comprobará la pestaña "Zynq Block Design" para asegurar que las interfaces ENET0 y UART1 están activas. Las interfaces activas se marcan con un tick junto a su nombre como se observa en la Figura 3-53.

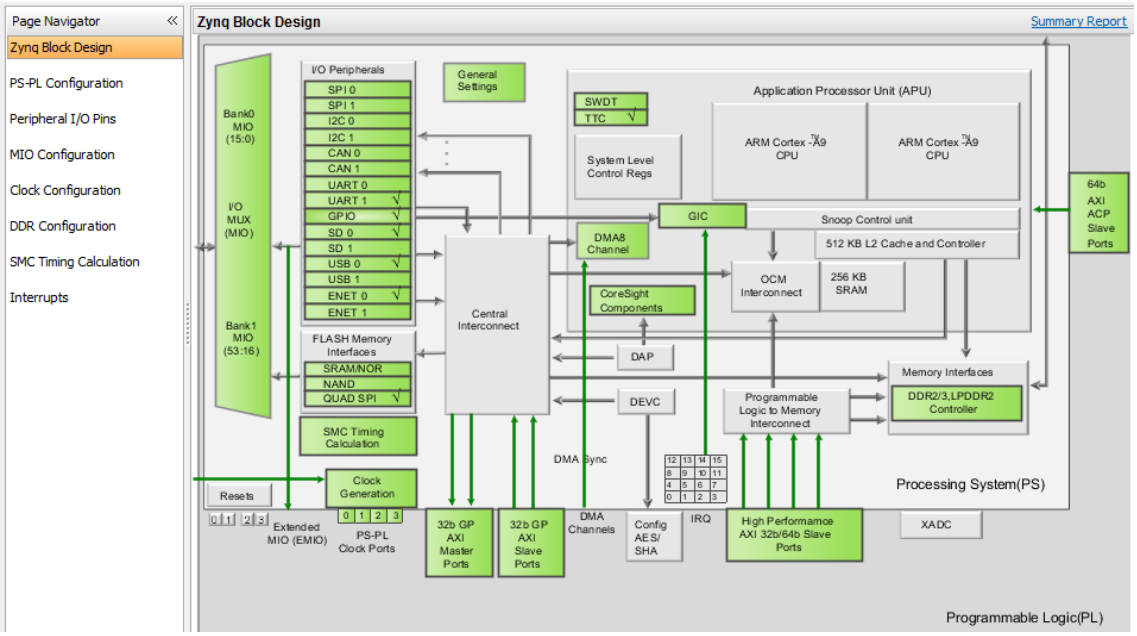

Figura 3-53: Diseño de bloques de la PS

Vivado permitirá automatizar las interconexiones entre bloques, en ciertos momentos, para facilitar el proceso de diseño. Para ello se hará click en la pestaña de información emergente “Run Block Automation” para que Vivado realice las conexiones acorde a la configuración que se indique en la configuración

Tras instanciar la PS, se podrán realizar las conexiones por defecto de la tarjeta, como por ejemplo las entradas/salidas y la conexión a la memoria DDR con los puertos entrada/salida del SoC. La ventana de configuración de la automatización se muestra en la Figura 3-54.

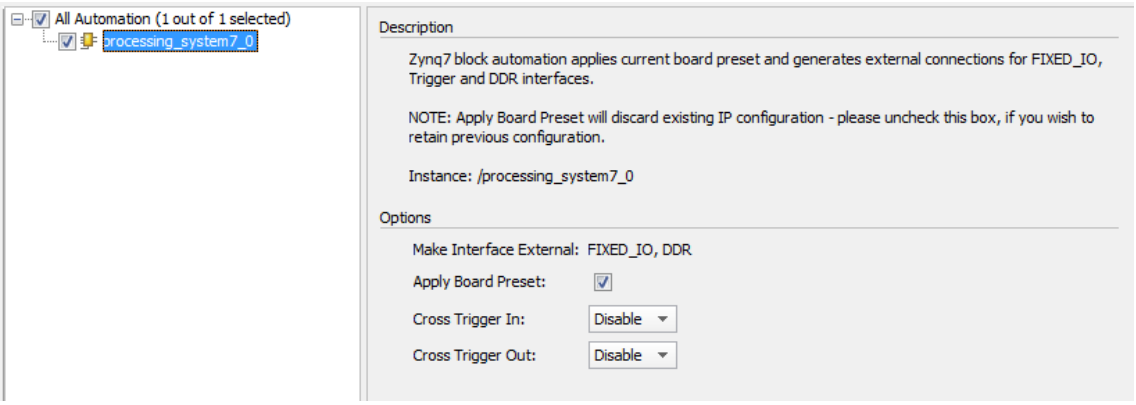

Figura 3-54: Asistente de automatización de conexiones para la PS

El resultado tras la automatización de las conexiones será el mostrado en la Figura 3-55.

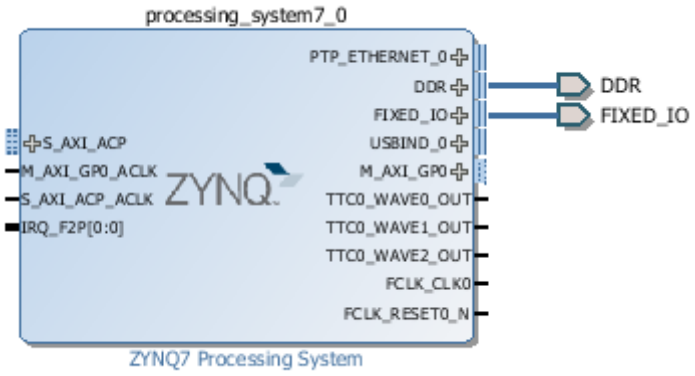

Figura 3-55: Sistema de Procesamiento Zynq

A continuación se añadirá el DMA de la misma manera que se hizo con la PS. Aparecerá de nuevo el mensaje de automatizar la interconexión para realizar las siguientes conexiones entre el DMA y la PS:

- Interfaz S_AXI-LITE con el puerto de propósito general GP0.
- Interfaz M_AXI_SG con el puerto ACP.
- Interfaz M_AXI_MM2S con el puerto ACP.
- Interfaz M_AXI_S2MM con el puerto ACP.

Vivado realizará las interconexiones indicadas entre los IP existentes en el diseño y adicionalmente añadirá tres nuevos bloques. Un bloque de generación del reset y dos AXI Interconnect, uno para las interconexiones entre DMA y el puerto GP0 para la configuración de los registros del DMA, y otro para conexiones con la memoria y configuración del modo Scatter/Gather a través del puerto ACP.

Una vez agregado el DMA al diseño, se asignarán las interrupciones de los canales MM2S y S2MM a los dos bits de menor peso del puerto IRQ_F2P del PS habilitado anteriormente. Para ello es necesario concatenarlas en un vector mediante un IP llamado “concat”. Las señales de interrupción serán las siguientes:

- **IRQ_F2P[0]**: ID de interrupción = 61, se asignará a la señal mm2s_introut.
- **IRQ_F2P[1]**: ID de interrupción = 62, se asignará la señal s2mm_introut.

El resultado de las interconexiones realizadas se muestra en la Figura 3-56.

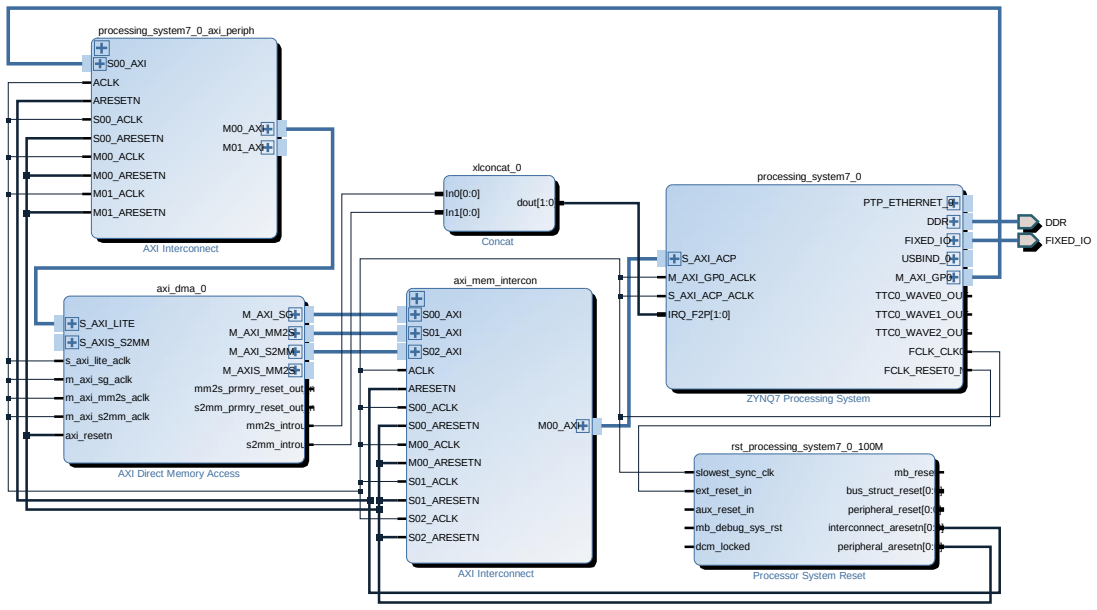

Figura 3-56: Diseño de bloques tras añadir el DMA

La configuración del DMA para que éste se adapte a las necesidades del diseño se realizará a través de la GUI de configuración de éste. Se deberán habilitar ambos canales, activar la opción Scatter/Gather para habilitar dicho modo, activar la opción de permitir transferencias desalineadas para evitar errores si los datos no están alineados y configurar los tamaños del bus de datos a 64 bits tanto en las interfaces mapeadas en memoria como en AXI4-Stream.

La Figura 3-57 muestra el estado de la GUI de configuración del DMA tras aplicar los cambios anteriores.

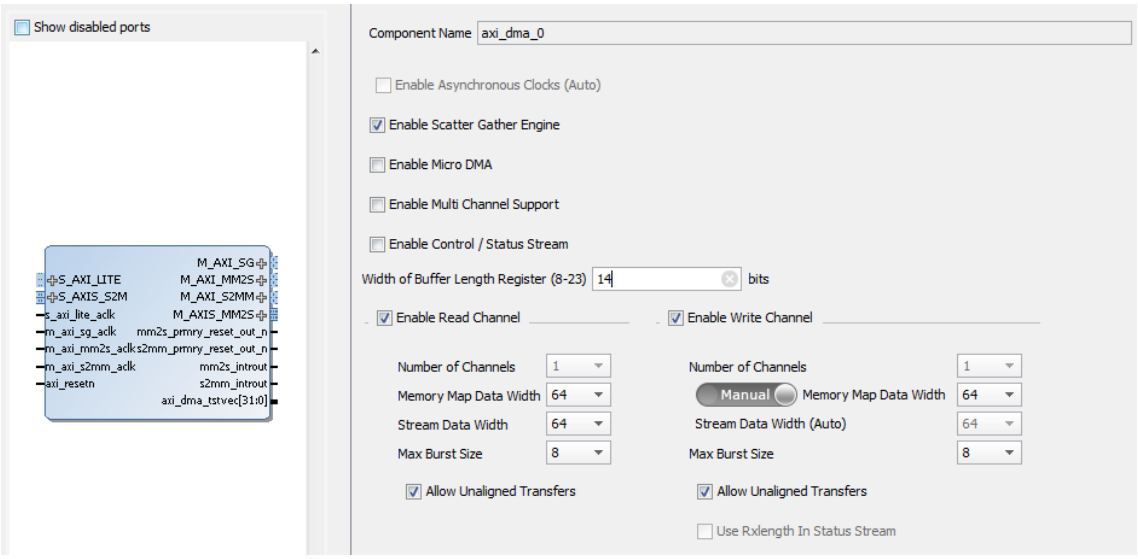

Figura 3-57: Configuración del DMA

Por último se añadirá el cliente DMA (PLC IP) y será configurado como se muestra en la Figura 3-58, mediante la GUI creada para dicho propósito, para definir el tamaño de los buses de las interfaces AXI4-Stream (TDATA) a 64 bits, el tamaño interno del bus a 16 bits, y las señales TID, TDEST y TUSER con los valores 5, 5 y 4 respectivamente. Aunque como se explicará más adelante, estas señales no serán usadas en este diseño.

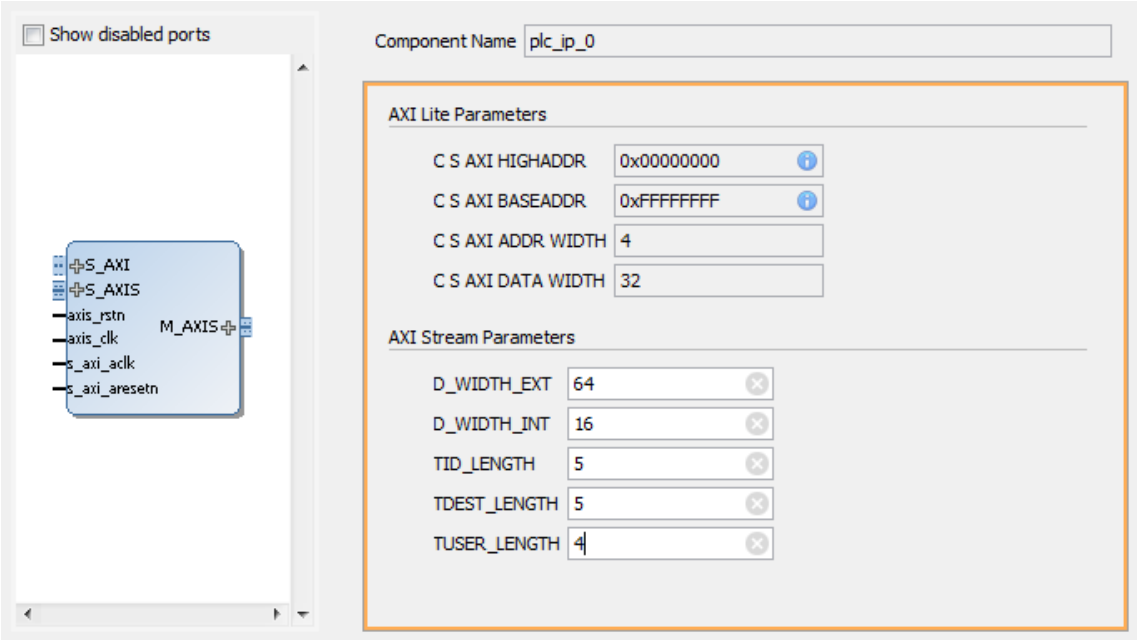

Figura 3-58: Configuración del PLC IP

La conexión de la interfaz AXI4-Lite del PLC IP con el AXI Interconnect se puede automatizar de nuevo con Vivado, sin embargo, las conexiones restantes del PLC IP se realizarán manualmente. Faltan por conectar las interfaces AXI4-Stream del cliente con el DMA, y las señales de reloj y reset.

La interfaz S_AXIS del PLC_IP debe conectarse al puerto MM2S del DMA para recibir un stream de datos desde memoria, mientras que la interfaz M_AXIS del PLC_IP debe conectarse al puerto S2MM del DMA para subir a memoria el stream de datos de salida del cliente DMA. La Figura 3-59 ilustra dichas conexiones.

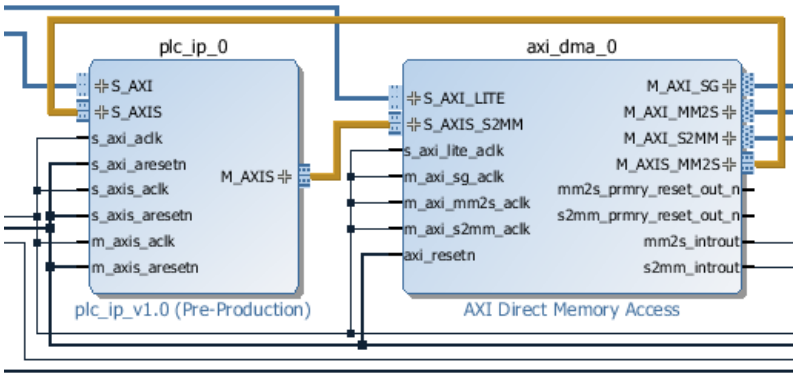

Figura 3-59: Conexiones AXI4-Stream del PLC IP con el DMA

El diagrama completo del diseño se puede consultar en la Figura 7-4 del capítulo 7.

Para terminar el diseño de la arquitectura hardware tan solo queda generar los archivos de salida haciendo click en "Generate Output Products" del menú contextual que aparece al hacer click derecho en el diseño como se indica en la Figura 3-60. Esta opción generará los archivos fuente en código HDL y ficheros de restricciones de cada IP.

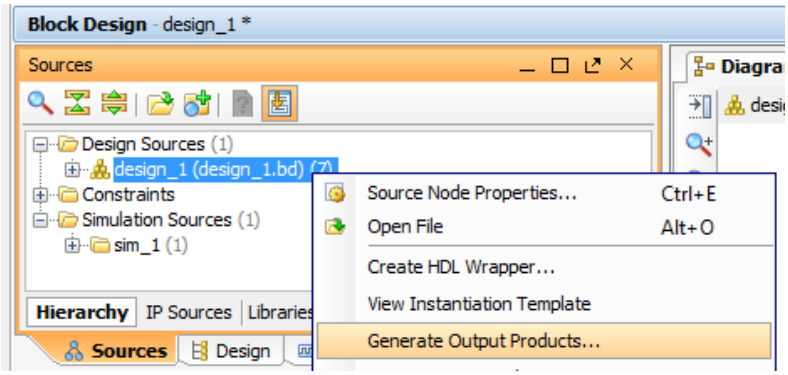

Figura 3-60: Generar ficheros de salida

Al terminar el proceso se mostrará la advertencia "crítica" de la Figura 3-61, aunque es este caso no es importante. Se debe a que las señales TID, TDEST y TUSER del puerto de entrada del PLC IP no están conectadas al DMA, ya que al estar el DMA configurado como monocanal, no las utiliza.

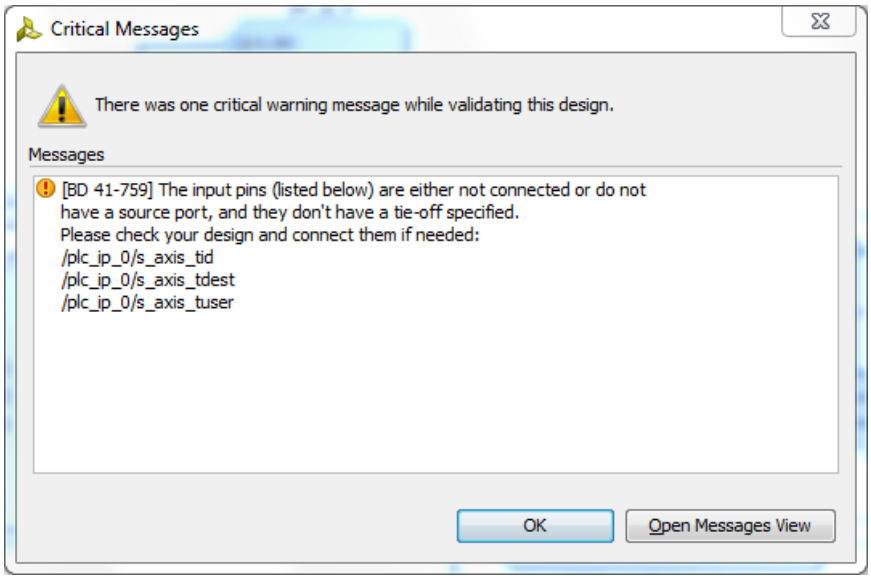

Figura 3-61: Advertencias tras generar los ficheros de salida

Por último, se creará un fichero HDL wrapper que envuelva a todo el diseño y sea el fichero top del sistema, para ello se hará click en "Create HDL Wrapper" como muestra la Figura 3-62. Finalmente se sintetizará, implementará y generará el bitstream para proceder al desarrollo software.

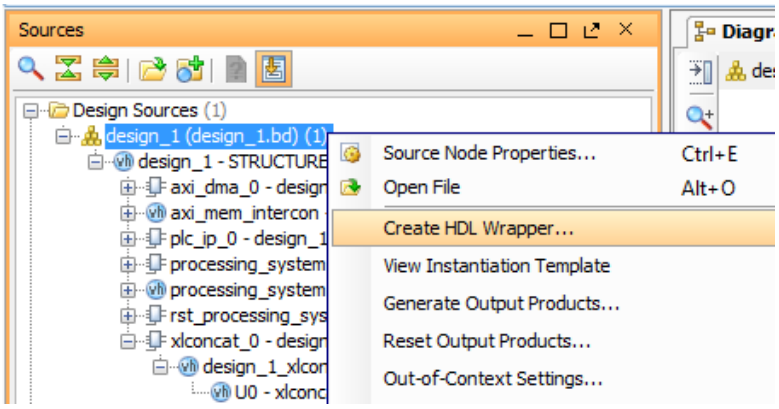

Figura 3-62: Creación del envoltorio HDL (wrapper)

4 DESARROLLO SOFTWARE

Tras finalizar el diseño de la arquitectura del SoC, es necesario desarrollar una aplicación software que se encargue de configurar y controlar el DMA. Dicha aplicación debe generar los anillos de descriptores utilizados en el modo de funcionamiento Scatter/Gather del DMA y manejarlos para poder realizar transferencias de memoria entre la PS y PL y viceversa.

Para hacer una prueba realista del sistema diseñado donde se moverá una gran carga de datos, se recibirá un stream de video por el puerto Ethernet y tras realizar la transferencia con el DMA se volverán a enviar para visualizar el video y comprobar que la transferencia se ha realizado correctamente. Para la recepción y trasmisión del stream de video se empleará la pila de protocolos TCP/IP llamada LwIP.

4.1 Software DMA

Junto con el IP del DMA, Xilinx proporciona los drivers necesarios para controlar el hardware y poder llevar a cabo el proceso de configuración del DMA para realizar transferencias de memoria. Este driver permitirá además, la generación de los anillos de descriptores necesarios para la operación Scatter/Gather, configuración y manejo de los BDs, etc.

La aplicación software del DMA se encargará de configurarlo para operar en modo Scatter/Gather, de forma que se cree un anillo de descriptores para cada canal, donde cada BD contiene un puntero al siguiente de la cadena, la dirección de memoria del buffer de transmisión o recepción y el tamaño de este buffer, entre otros datos.

Los descriptores están compartidos entre el software y el hardware y pueden encontrarse en los siguientes estados:

- **Libre:** El BD está libre y no está siendo usado en dicho momento. Es necesario asignarlo mediante la función XAxiDma_BdRingAlloc() para su uso.
- **Pre-procesado:** Tras ser asignados se encuentran en este estado, donde la aplicación software tiene el control de dicho descriptor y se encarga de configurarlo para realizar la transacción deseada.
- **Hardware:** El BD ha sido enviado al hardware mediante la función XAxiDma_BdRingToHw() y están bajo su control, esperando a ser procesados.
- **Post-procesado:** El BD ha sido procesado por el hardware y el control es devuelto al software mediante la función XAxiDma_BdRingFromHw(). El último paso será liberarlo mediante la función XAxiDma_BdRingFree() para volver al primer estado.

La Figura 4-1 resume las diferentes transiciones que pueden ocurrir en un BD del anillo de descriptores y los estados asociados a dichas transiciones.

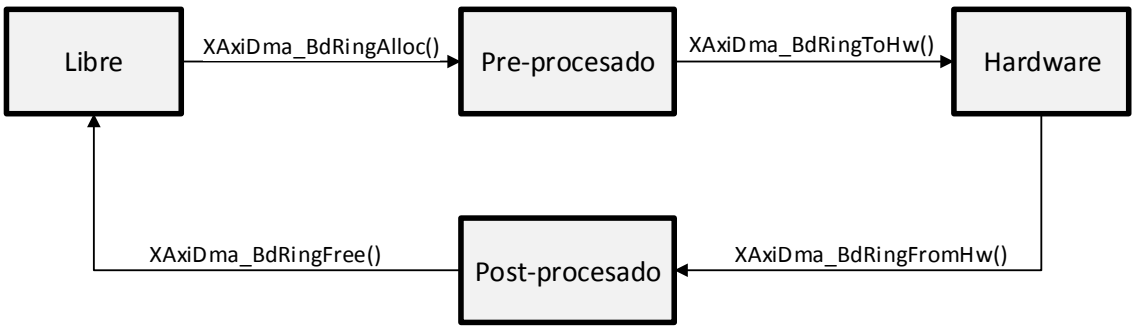

Figura 4-1: Transiciones y estados de los descriptores del DMA

La operación del controlador DMA se gestionará mediante interrupciones, en lugar de por sondeo, con el objetivo de mejorar el rendimiento del procesador y poder realizar otras tareas mientras se llevan a cabo las transferencias con el DMA. Cada canal (MM2S y S2MM) tendrá una interrupción asociada, que será tratada por la aplicación mediante una función de atención a la interrupción. Estas interrupciones se producirán cuando:

- El canal MM2S termine de procesar el descriptor TX, es decir, cuando se termine realizar la transferencia de datos de PS a PL.
- El canal S2MM termine de procesar el descriptor RX, es decir, cuando acabe de transferir los datos de PL a PS

Ante una gran carga de datos, el número de interrupciones se incrementa, produciendo penalizaciones en el procesador debido al cambio de contexto provocado por las funciones de atención a la interrupción y por tanto reduciendo el rendimiento del procesador. Para reducir dicha penalización por el cambio de contexto, el DMA ofrece la opción de fusionar las interrupciones mediante la configuración del "Interrupt Coalescing".

El DMA permite configurar el número de interrupciones a fusionar en un valor comprendido entre 0 y 255, útil cuando existe un gran flujo de descriptores como en el caso de paquetes Ethernet.

Sin embargo, cuando la carga fluctúa, no interesa esperar a completar un grupo de descriptores devueltos por el hardware para que se produzca la interrupción, por lo que se implementa un timeout cuyo valor está comprendido entre 0 y 255 unidades de C_DLYTMR_RESOLUTION (constante que indica el número de ciclos de reloj del temporizador y cuyo valor por defecto es 125 ciclos de reloj).

El proceso de configuración del DMA se puede resumir en las 4 etapas indicadas en la Figura 4-2: Inicialización del DMA, configuración del canal de transmisión (MM2S), configuración del canal de recepción (S2MM) y configuración de las interrupciones.

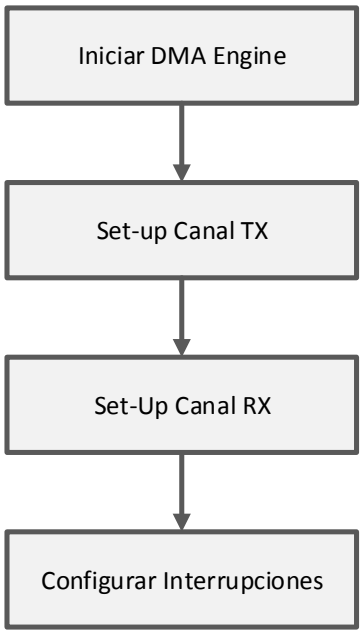

Figura 4-2: Pasos en el proceso de configuración del DMA

4.1.1 Configuración del DMA

A continuación se explicará el proceso de configuración del DMA ilustrado en el esquema anterior. Para el desarrollo del software capaz de controlar el DMA se partirá de la aplicación de ejemplo proporcionada por Xilinx.

4.1.1.1 Inicialización del DMA

En primer lugar se obtiene la configuración del hardware del DMA para después poder inicializarlo:

```
// Getting DMA Config
Config = XAxiDma_LookupConfig(DMA_DEV_ID);
// Initialize DMA engine
Status = XAxiDma_CfgInitialize(&AxiDma, Config);
```

4.1.1.2 Configuración del canal Tx (MM2S)

Para configurar el canal de transmisión (MM2S) por el que se llevarán a cabo las transferencias de memoria entre PS y PL habrá que crear un anillo de descriptores de transmisión que inicialmente tendrán sus campos sin inicializar. El siguiente código forma parte de la función Tx_Setup(), que es la encargada de realizar la tarea de configuración de este canal. El procedimiento es el siguiente:

1. Deshabilitar la interrupción del canal antes del proceso de configuración.

```
TxRingPtr = XAxiDma_GetTxRing(&AxiDma);
// Disable all TX interrupts
XAxiDma_BdRingIntDisable(TxRingPtr, XAXIDMA_IRQ_ALL_MASK);
```

2. Crear un anillo de descriptores en la dirección de memoria especificada por TX_BD_SPACE_BASE y con un número de descriptores igual a TX_N_BD.

```
// Create BD Ring
Status = XAxiDma_BdRingCreate(TxRingPtr, TX_BD_SPACE_BASE, TX_BD_SPACE_BASE,
XAXIDMA_BD_MINIMUM_ALIGNMENT, TX_N_BD);
```

3. Crear un descriptor con todos sus campos de información sin inicializar, excepto los que contienen información del anillo o del hardware para después clonar este descriptor en el resto de descriptores del anillo.

```
// Create an all-zero BD as the template
XAxiDma_BdClear(&BdTemplate);
// Copy previous template to all BDs
Status = XAxiDma_BdRingClone(TxRingPtr, &BdTemplate);
```

4. Configurar la fusión de interrupciones (Interrupt Coalescing) con los valores especificados por las constantes DMA_TX_COALESCE y DMA_TX_DELAY.

```
Status = XAxiDma_BdRingSetCoalesce(TxRingPtr, DMA_TX_COALESCE, DMA_TX_DELAY);
```

5. Habilitar la interrupción del canal de transmisión (MM2S).

```
// Enable all TX interrupts
XAxiDma_BdRingIntEnable(TxRingPtr, XAXIDMA_IRQ_ALL_MASK);
```

6. Inicializar el canal.

```
// Start TX channel
Status = XAxiDma_BdRingStart(TxRingPtr);
```

4.1.1.3 Configuración canal Rx

Para configurar el canal de recepción (S2MM) por el que se recibirán las transferencias de memoria de PL a PS, es necesario crear otro anillo de descriptores, y en este caso, a diferencia del anillo de descriptores de transmisión, se deben asignar dichos BDs a unos buffers de recepción para comenzar a recibir paquetes y guardar los datos en ellos.

El siguiente código forma parte de la función Rx_Setup(), que es la encargada de realizar la tarea de configuración de este canal. El procedimiento es el siguiente:

1. Deshabilitar la interrupción del canal antes del proceso de configuración.

```
RxRingPtr = XAxiDma_GetRxRing(&AxiDma);
// Disable all RX interrupts
XAxiDma_BdRingIntDisable(RxRingPtr, XAXIDMA_IRQ_ALL_MASK);
```

2. Crear un anillo de descriptores en la dirección de memoria especificada por RX_BD_SPACE_BASE y con un número de descriptores igual a RX_N_BD.

```
// Create BD Ring
Status = XAxiDma_BdRingCreate(RxRingPtr, RX_BD_SPACE_BASE, RX_BD_SPACE_BASE,
XAXIDMA_BD_MINIMUM_ALIGNMENT, RX_N_BD);
```

3. Crear un descriptor con todos sus campos de información sin inicializar, excepto los que contienen información del anillo o del hardware para después clonar este descriptor en el resto de descriptores del anillo.

```
// Create an all-zero BD as the template
XAxiDma_BdClear(&BdTemplate);
// Copy previous template to all BDs
Status = XAxiDma_BdRingClone(RxRingPtr, &BdTemplate);
```

4. Asignar los descriptores y configurar sus registros con la dirección del buffer de recepción y el tamaño de dicho buffer.

```
// Attach buffers to RX BDring
FreeBdCount = XAxiDma_BdRingGetFreeCnt(RxRingPtr);
Status = XAxiDma_BdRingAlloc(RxRingPtr, FreeBdCount, &BdPtr);

BdCurPtr = BdPtr;
RxBufferPtr = RX_BUFFER_BASE;

for (Index = 0; Index < FreeBdCount; Index++) {

    Status = XAxiDma_BdSetBufAddr(BdCurPtr, RxBufferPtr);
    Status = XAxiDma_BdSetLength(BdCurPtr, PACKET_SIZE,
    RxRingPtr->MaxTransferLen);

    XAxiDma_BdSetCtrl(BdCurPtr, 0);
    XAxiDma_BdSetId(BdCurPtr, RxBufferPtr);

    RxBufferPtr += PACKET_SIZE;
    BdCurPtr = XAxiDma_BdRingNext(RxRingPtr, BdCurPtr);
}
```

5. Configurar la fusión de interrupciones (Interrupt Coalescing) con los valores especificados por las constantes DMA_RX_COALESCE y DMA_RX_DELAY.

```
// Set the coalescing threshold
Status = XAxiDma_BdRingSetCoalesce(RxRingPtr, DMA_RX_COALESCE, DMA_RX_DELAY);
```

6. Enviar el descriptor al hardware.

```
// Give the BD to hardware
Status = XAxiDma_BdRingToHw(RxRingPtr, FreeBdCount, BdPtr);
```

7. Habilitar la interrupción del canal de recepción (S2MM).

```
// Enable all RX interrupts
XAxiDma_BdRingIntEnable(RxRingPtr, XAXIDMA_IRQ_ALL_MASK);
```

8. Inicializar el canal.

```
// Start RX DMA channel
Status = XAxiDma_BdRingStart(RxRingPtr);
```

4.1.2 Configuración de las interrupciones

Para tener el DMA completamente operativo tan solo falta habilitar y configurar el controlador de interrupciones del procesador, llamado Generic Interrupt Controller (GIC). El GIC además de controlar las interrupciones del procesador y periféricos, permite controlar los 16 puertos de interrupciones generadas en la PL.

El siguiente código forma parte de la función Setup_Intr_System(), que es la encargada del proceso de configuración del GIC. El procedimiento es el siguiente:

1. Inicializar el controlador de interrupciones (GIC).

```
// Initialize the interrupt controller driver
IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig, IntcConfig-
>CpuBaseAddress);
```

2. Configurar la prioridad así como el tipo de disparo de las interrupciones, cuyas IDs son las mostradas a continuación. Se ha configurado una prioridad 0xA0 y el disparo con el flanco de subida de la señal de interrupción procedente de la PL.
   - **IRQ_F2P[0]** = mm2s_introut, ID de interrupción = 61.
   - **IRQ_F2P[1]** = s2mm_introut, ID de interrupción = 62.

```
XScuGic_SetPriorityTriggerType(IntcInstancePtr, TxIntrId, 0xA0, 0x3);
XScuGic_SetPriorityTriggerType(IntcInstancePtr, RxIntrId, 0xA0, 0x3);
```

3. Indicar la función de atención a la interrupción (IRQ Handler) que corresponde con cada una.

```
Status = XScuGic_Connect(IntcInstancePtr, TxIntrId,
(Xil_InterruptHandler)Tx_Intr_Handler, TxRingPtr);

Status = XScuGic_Connect(IntcInstancePtr, RxIntrId,
(Xil_InterruptHandler)Rx_Intr_Handler, RxRingPtr);
```

4. Habilitar ambas interrupciones.

```
// Enable Tx and Rx Interrupts
XScuGic_Enable(IntcInstancePtr, TxIntrId);
XScuGic_Enable(IntcInstancePtr, RxIntrId);
```

5. Habilitar las excepciones en Zynq y conectar el controlador de interrupciones al hardware.

```
// Enable interrupts from the hardware
Xil_ExceptionInit();

Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(Xil_ExceptionHandler)INTC_HANDLER, (void *)IntcInstancePtr);

Xil_ExceptionEnable();
```

4.1.3 Funciones de Atención a la interrupción

Una vez configurado el DMA, ya está preparado para transferir datos entre PS y PL, pero habrá que crear las funciones de atención a las interrupciones para manejar los descriptores cuando hayan sido procesados por el hardware. En ellas se comprobará si ha habido algún error, y si no lo hay, realizará el tratamiento necesario del BD para que pueda ser utilizado en el futuro.

Existen dos funciones de atención a la interrupción:

- **Tx_Intr_Handler()** para el canal de transmisión (MM2S). Si el procesado ha sido correcto, se llama a la función Tx_CallBack() para realizar el tratamiento de descriptores.
- **Rx_Intr_Handler()** para el canal de recepción (S2MM). Si el procesado ha sido correcto, se llama a la función Rx_CallBack() para realizar el tratamiento de descriptores, además de los datos recibidos.

4.1.3.1 Tx_Intr_Handler()

Al terminar la transferencia de memoria desde PS hasta PL se genera esta interrupción, cuya finalidad es tratar los descriptores que han sido procesados por el hardware para volver a liberarlos en el anillo y puedan ser utilizados de nuevo. A continuación se explicarán las tareas que se llevan a cabo en esta función:

- Comprobar que se ha producido una interrupción en el canal de transmisión y no se ha producido un error en el anillo de descriptores, en caso contrario se reporta el error, se reinicia el DMA y no se realiza ninguna otra tarea. Si no se produce ningún error se continúa con los siguientes pasos.

```
// Read pending interrupts
IrqStatus = XAxiDma_BdRingGetIrq(TxRingPtr);

// Acknowledge pending interrupts
XAxiDma_BdRingAckIrq(TxRingPtr, IrqStatus);

// If no interrupt is asserted, we do not do anything
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    return;
}

// Raise error flag and reset in case of ERROR
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    XAxiDma_BdRingDumpRegs(TxRingPtr);
    Error = 1;

    XAxiDma_Reset(&AxiDma);
    return;
}

// If Interrupt on delay or Interrupt on Complete detected...
if ((IrqStatus & (XAXIDMA_IRQ_DELAY_MASK | XAXIDMA_IRQ_IOC_MASK)))
    Tx_CallBack(TxRingPtr);
```

- En la función Tx_Callback se obtienen los BDs procesados del hardware y se comprueba su estado para ver si reporta algún error.

```
// Get all processed BDs from hardware
BdCount = XAxiDma_BdRingFromHw(TxRingPtr, XAXIDMA_ALL_BDS, &BdPtr);

// Handle the BDs
BdCurPtr = BdPtr;
for (Index = 0; Index < BdCount; Index++) {

    // Check Errors
    BdSts = XAxiDma_BdGetSts(BdCurPtr);
    if ((BdSts & XAXIDMA_BD_STS_ALL_ERR_MASK) || (!(BdSts &
XAXIDMA_BD_STS_COMPLETE_MASK))) {
            Error = 1;
            break;
        }
    // Find the next processed BD
    BdCurPtr = XAxiDma_BdRingNext(TxRingPtr, BdCurPtr)
}
```

- A continuación se libera el BD para que pase a estado Free.

```
// Free all processed BDs for future transmission
Status = XAxiDma_BdRingFree(TxRingPtr, BdCount, BdPtr);
```

4.1.3.2 Rx_Intr_Handler()

Al terminar la transferencia de memoria desde PL hacia PS se genera esta interrupción, cuya finalidad es tratar los descriptores que han sido procesados por el hardware para volver a asignarlos en el anillo, y puedan ser utilizados de nuevo para recibir un nuevo paquete. A continuación se explicarán las tareas que se llevan a cabo en esta función:

- Comprobar que se ha producido una interrupción en el canal de recepción y no se ha producido un error en el anillo de descriptores, en caso contrario se reporta el error, se reinicia el DMA y no se realiza ninguna otra tarea. Si no se produce ningún error se continúa con los siguientes pasos.

```
// Read pending interrupts
IrqStatus = XAxiDma_BdRingGetIrq(RxRingPtr);

// Acknowledge pending interrupts
XAxiDma_BdRingAckIrq(RxRingPtr, IrqStatus);

// If no interrupt is asserted, we do not do anything
if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
    return;
}

// Raise error flag and reset in case of ERROR
if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
    XAxiDma_BdRingDumpRegs(RxRingPtr);
    Error = 1;
    XAxiDma_Reset(&AxiDma);
    return;
}

// If Interrupt on delay or Interrupt on Complete detected...
if ((IrqStatus & (XAXIDMA_IRQ_DELAY_MASK | XAXIDMA_IRQ_IOC_MASK))) {
    Rx_CallBack(RxRingPtr);
}
```

- En la función Rx_Callback se obtienen los BDs procesados del hardware y se comprueba su estado para ver si se reporta algún error. Se procesan los datos recibidos, que en el caso de la aplicación final, se mandan en un paquete UDP.

```
// Get finished BDs from hardware
BdCount = XAxiDma_BdRingFromHw(RxRingPtr, XAXIDMA_ALL_BDS, &BdPtr);

BdCurPtr = BdPtr;
for (Index = 0; Index < BdCount; Index++) {

    // Check status flags. In case of error, stop processing.
    BdSts = XAxiDma_BdGetSts(BdCurPtr);
    if ((BdSts & XAXIDMA_BD_STS_ALL_ERR_MASK) || (!(BdSts &
XAXIDMA_BD_STS_COMPLETE_MASK))) {
        Error = 1;
        break;
    }

    // Get address of current rx buffer
    dst_buffer = (void *) XAxiDma_BdGetId(BdCurPtr);

    // Find the next processed BD
    BdCurPtr = XAxiDma_BdRingNext(RxRingPtr, BdCurPtr);
    RxDone += 1;

    //Send UDP Packet
    send_udp((int*)dst_buffer);
}
```

- A continuación se libera el BD para que pase a estado Free.

```
// Free all processed RX BDs for future transmission
XAxiDma_BdRingFree(RxRingPtr, BdCount, BdPtr);
```

- Se asignan los BD libres y son mandados al hardware para poder volver a utilizarlos en un futuro.

```
// Return processed BDs to RX channel
FreeBdCount = XAxiDma_BdRingGetFreeCnt(RxRingPtr);
XAxiDma_BdRingAlloc(RxRingPtr, FreeBdCount, &BdPtr);
XAxiDma_BdRingToHw(RxRingPtr, FreeBdCount, BdPtr);
```

4.1.4 Envío de datos

Para comenzar una transferencia DMA se empleará la función Send_Packet(). Esta función se encarga de asignar los descriptores del canal de transmisión y modificar sus campos para establecer la dirección de memoria y el tamaño de los datos a transmitir. Además configurará los bits SOF y EOF, característicos del funcionamiento Scatter/Gather del registro de control del descriptor como se explicó en el apartado 2.5.1.

El procedimiento es el siguiente:

- Asignar un BD para la transmisión de datos.

```
TxRingPtr = XAxiDma_GetTxRing(AxiDmaInstPtr);
```

```
// Allocate BD to transmit
Status = XAxiDma_BdRingAlloc(TxRingPtr, 1, &BdPtr);
```

- Configurar la dirección de los datos a transmitir y su tamaño.

```
// Set up the BD using the information of the packet to transmit
Status = XAxiDma_BdSetBufAddr(BdCurPtr, BufferAddr);

Status = XAxiDma_BdSetLength(BdCurPtr, PACKET_SIZE, TxRingPtr-
>MaxTransferLen);
```

- Configurar los bits SOF y EOF del registro de control, ya que la transmisión solo consta de un único paquete.

```
// Set SOF and EOF bits
CrBits |= XAXIDMA_BD_CTRL_TXSOF_MASK;
CrBits |= XAXIDMA_BD_CTRL_TXEOF_MASK;
XAxiDma_BdSetCtrl(BdCurPtr, CrBits);
XAxiDma_BdSetId(BdCurPtr, BufferAddr);
```

- Enviar el descriptor al hardware para que sea procesado.

```
// Give the BD to hardware
Status = XAxiDma_BdRingToHw(TxRingPtr, 1, BdPtr);
```

4.2 LwIP

Para poder poner bajo prueba al sistema en una situación real, donde la carga de datos a transmitir con el DMA sea grande, se recibirá un stream de video en datagramas UDP procedente de un ordenador, que tras realizar la transferencia con el DMA se reenviarán de nuevo al ordenador para visualizarlo en el reproductor VLC Player.

Con la finalidad de dotar al sistema de la conectividad Ethernet necesaria para la transferencia del streaming de video se empleará la pila de protocolos TCP/IP LwIP, la cual proporciona varias APIs para su control y configuración.

LwIP proporciona una API de nivel bajo y dos de alto nivel. La API de nivel bajo, llamada “Raw API”, se basa en el empleo de funciones Callback para realizar diferentes tareas (Por ejemplo, que al recibir un paquete se llame a una función callback). Esta librería destaca por su rapidez y bajo consumo de memoria a pesar de que su uso no es el más sencillo. Sin embargo al emplear el protocolo UDP se simplifica bastante.

Las API de nivel alto son Netconn y Socket, son API secuenciales que facilitan el uso de LwIP, sin embargo no pueden ser utilizadas sin sistema operativo ya que requieren el uso de hilos de ejecución.

En esta aplicación se utilizará la “Raw API” para el control y configuración de LwIP, ya que el sistema que se utilizará no ejecutará ningún sistema operativo, sino que será Standalone. La ejecución del programa se basa en la llamada a funciones Callbacks para realizar tareas específicas. Por ejemplo, en la aplicación que se está desarrollando se dispondrá de un callback que se ejecutará al recibir un nuevo paquete UDP y en dicha

función se realizarán las tareas necesarias para transferir con el DMA los datos que han sido recibidos.

Como se explicó en el apartado 2.8, se utilizará una estructura llamada PCB para determinar los parámetros de la conexión. La estructura específica para el protocolo UDP tendrá los campos indicados en la Tabla 4-1.

Tabla 4-1: Campos de la estructura de un PCB UDP

Campo	Descripción
Next	Puntero al siguiente elemento de la cadena de PCBs.
Local_IP	Dirección IP local (ZedBoard).
Dest_IP	Dirección IP destino (PC).
Local_port	Número de puerto de recepción de los paquetes.
Dest_port	Número de puerto al que se envían los paquetes.
Flags	Indican la política de chequeo utilizada.
Checksum	Suma de verificación.
Recv	Función callback a la que llamar cuando se recibe un paquete.
Recv_arg	Argumento pasar a la función callback.

Al recibir un paquete UDP se buscará en la lista de PCBs enlazadas una que coincida con los parámetros del paquete recibido, y una vez se obtenga un PCB se llamará al callback establecido.

4.2.1 Configuración LwIP

Para el desarrollo del software capaz de controlar LwIP se partirá de la aplicación de ejemplo proporcionada por Xilinx, donde se emplea el protocolo TCP para recibir datos y volver a enviarlos. Se realizarán las modificaciones necesarias para utilizar el protocolo UDP y adaptarlo a las especificaciones del proyecto.

El procedimiento de configuración de LwIP para que funcione con el protocolo UDP es el siguiente:

1. Configurar Zynq e inicializar un timer con una interrupción asociada a él, cuya función es resetear el canal de recepción en el caso en el que no responda debido a un bloqueo por gran cantidad de tráfico. En el fichero platform_zynq.c se encuentra el código necesario para la configuración de dicho timer.

```
// Initialize Zynq Platform
init_platform();
```

2. Inicializar LwIP empleando la función lwip_init.

```
// lwip Stack initialization
lwip_init();
```

3. Indicar la dirección IP y MAC que tendrá la tarjeta ZedBoard

```
unsigned char MAC_addr[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

IP4_ADDR(&ipaddr, 192, 168, 1, 10);
IP4_ADDR(&netmask, 255, 255, 255, 0);
IP4_ADDR(&gateway, 192, 168, 1, 1);
```

4. Añadir una nueva interfaz de red con esta IP y la dirección MAC de la tarjeta a la lista de interfaces de LwIP.

```
if (!xemac_add(p_netif, &ipaddr, &netmask, &gateway, MAC_addr,
PLATFORM_EMAC_BASEADDR))
{
	xil_printf("Error adding N/W interface\n\r");
	return -1;
}
```

5. Configurar esta interfaz para que sea la interfaz por defecto.

```
netif_set_default(p_netif);
```

6. Habilitar las interrupciones del timer de alto tráfico.

```
// Enable interrupts
platform_enable_interrupts();
```

7. Habilitar la interfaz para que esté activa (up).

```
netif_set_up(p_netif);
```

8. Crear un PCB que contendrá la información de la conexión y vincularlo a un puerto de entrada.

```
// Create new UDP Protocol Control Block (PCB) structure
pcb = udp_new();

// Bind to specified input port
error = udp_bind(pcb, IP_ADDR_ANY, INPUT_PORT);
```

9. Por último se especifica el callback a ejecutar cuando se reciba un nuevo paquete UDP.

```
// Specify receive_callback function
udp_recv(pcb, udp_receive_callback, NULL);
```

Tras este proceso, LwIP ya estará configurado para recibir y enviar paquetes UDP, ahora tan solo falta definir que tarea realizar al recibir un paquete. En el caso de la aplicación final lo que se hará es enviar los paquetes UDP recibidos al DMA, para realizar una transferencia hacia la PL.

4.2.2 Recepción de un paquete UDP

Al recibir un nuevo paquete UDP se llamará automáticamente al callback de recepción llamado udp_receive_callback, donde se harán las siguientes tareas:

- Guardar la dirección donde se encuentran los datos recibidos en un puntero para realizar la transferencia DMA.

```
// Pointer to DMA Tx Data
Packet = (u32*)p->payload;
```

- Marcar flag para indicar al DMA que se dispone de un dato nuevo. En la función main se comprobará este flag y se hará una llamada a Send_Packet para enviar el dato recibido con el DMA.

```
// Data received Flag
rx_data_ready = 1;
```

- Liberar PBUF para evitar llenar el buffer de recepción.

```
// Free PBUF to avoid overflow of data
pbuf_free(p);
```

4.2.3 Envío de un paquete UDP

Tras realizar la transferencia DMA y recibir los datos procedentes del cliente (PLC IP) se llamará a la función send_udp, cuya tarea es empaquetar los datos en un datagrama UDP y enviarlos. El procedimiento es el siguiente:

- Reservar un PBUF con el tamaño de los datos a enviar.

```
// Allocate pbuf structure to send
struct pbuf * p = pbuf_alloc(PBUF_TRANSPORT, PACKET_SIZE, PBUF_ROM);
```

- Indicar la dirección de memoria donde se encuentran los datos a enviar.

```
// Indicate address of data to the pbuf
p->payload = data;
```

- Enviar paquete a la dirección IP destino y por el puerto especificado.

```
// Send packet
udp_sendto(pcb, p, &ip_dest, OUTPUT_PORT);
```

- Liberar PBUF para evitar llenar el buffer de transmisión.

```
//Free pbuf
pbuf_free(p);
```

4.3 Flujo de funcionamiento del software

El funcionamiento del Sistema completo se muestra en la Figura 4-3.

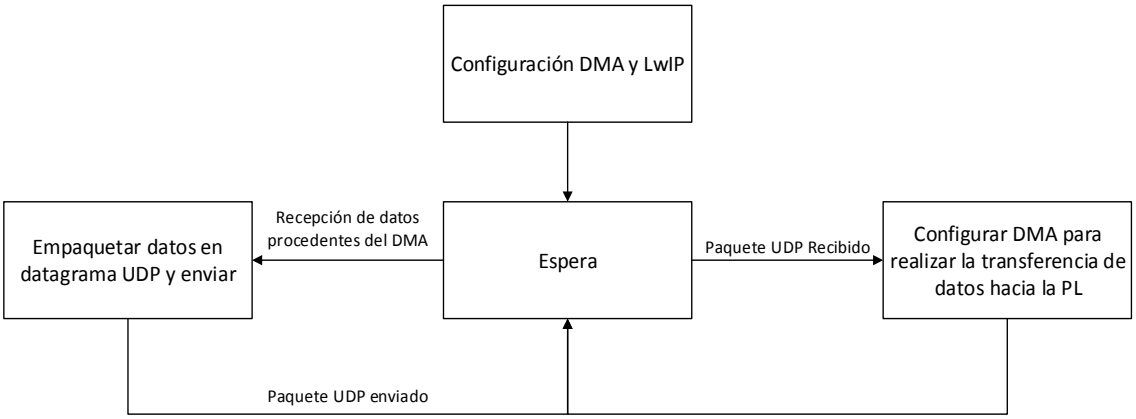

Figura 4-3: Diagrama del funcionamiento de la aplicación software

- En primer lugar se realizará la configuración del DMA y LwIp conforme a lo explicado en los apartados 4.1 y 4.2.
- El software pasará a un estado de espera, donde el procesador podría estar realizando otras tareas.
- Al recibir un paquete UDP, se configurará un BD del DMA para que realice la transferencia de los datos recibidos hacia la PL. Al enviar el BD al hardware se volverá al estado de espera.
- Al recibir datos procedentes del DMA por el canal de recepción, éstos se empaquetarán en un datagrama UDP y se enviarán. Al realizar la transmisión se volverá al estado de espera.

5 PRUEBAS EXPERIMENTALES Y RESULTADOS

5.1 Pruebas de streaming de video

Para comprobar que las transferencias del DMA se realizan satisfactoriamente, se ha diseñado un escenario de prueba donde se recibirá un stream de video desde un ordenador, formado por paquetes UDP, utilizando para ello el reproductor VLC Player. El objetivo de esta prueba es poner al sistema bajo una situación de funcionamiento real, donde hay que realizar la transferencia de un gran volumen de datos.

Tras realizar la transferencia con el DMA, los datos pasarán por el cliente PLC IP donde no sufrirán ninguna modificación, y volverán a ser transmitidos hacia la memoria empleando de nuevo el DMA. En este momento la aplicación software volverá a empaquetar los datos en datagramas UDP para volver a enviarlos al ordenador y reproducirlos en otro reproductor VLC Player. Este escenario se representa en la Figura 5-1.

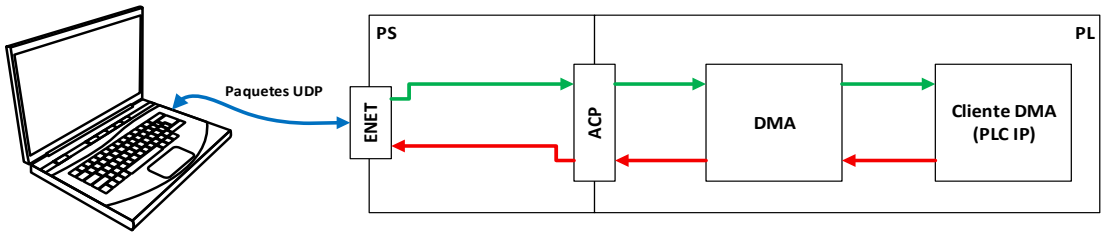

Figura 5-1: Escenario de prueba del diseño a realizar

La configuración del DMA será la siguiente:

- Las interrupciones se fusionarán para reducir las penalizaciones en el procesador debido al cambio de contexto provocado por las funciones de atención a la interrupción. Para ello se establecerán los parámetros DMA_TX_COALESCE y DMA_RX_COALESCE a un valor igual a 10, y los parámetros DMA_TX_DELAY y DMA_RX_COALESCE también 10.
- Los anillos de BDs de ambos canales tendrán 200 descriptores.
- El tamaño de los paquetes a transmitir por el DMA es de 1316 bytes, ya que el tamaño de los datos útiles recibidos en un paquete UDP enviado por VLC Player es de este tamaño.

La configuración de LwIP será la siguiente:

- Dirección IP de la tarjeta ZedBoard: 192.168.1.10.
- Dirección IP del ordenador: 192.168.1.20.
- Número de los puertos de entrada y salida: 1234.

Una vez configurada la aplicación software para que cumpla con las especificaciones del escenario de prueba propuesto, se procederá a mandar el streaming de video con VLC Player. Para ello se utilizará el asistente de emisión de video, o directamente se introducirá el comando que aparece en la Figura 5-2 en la consola del programa.

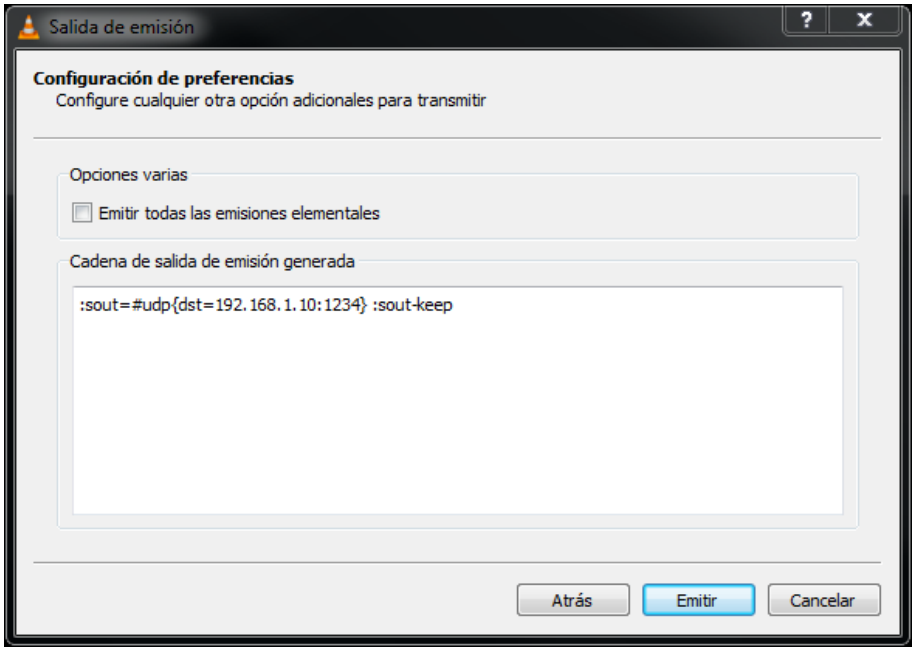

Figura 5-2: Configuración de VLC Player para el streaming de video

Para recibir el stream de datos en otro reproductor se hará click en "Abrir Medio" y se introducirá la URL: **udp://@:1234** para reproducir el stream de video que llega por el puerto 1234. Los paquetes UDP del stream de video se enviarán a la ZedBoard y serán devueltos al ordenador tras realizar todo el proceso de transferencia con el DMA.

En Wireshark se puede observar el tráfico de paquetes entre ambos. Como se muestra en el ejemplo de la Figura 5-3, los paquetes enviados por el ordenador hacia la ZedBoard aparecen con puerto de salida 61042, y puerto destino 1234. Los paquetes devueltos por la ZedBoard tendrán el mismo número de puerto de salida y entrada (1234).

No.	Time	Source	Destination	Protocol	Length	Info
1164093	956.752297			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164094	956.753248			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164095	956.753330			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164096	956.754246			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164097	956.754304			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164098	956.755245			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164099	956.755312	DTS 785.340000000	PTS 785.340000000	MPEG TS	1358	video-stream
1164100	956.756246			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164101	956.756305			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164102	956.757245			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164103	956.757294			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164104	956.758245			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164105	956.758303			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164106	956.759255			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164107	956.759367			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164108	956.759661			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164109	956.760249			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164110	956.760250			MPEG TS	1358	Source port: 1234 Destination port: 1234
1164111	956.760290			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164112	956.760534			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164113	956.760760			MPEG TS	1358	Source port: 61042 Destination port: 1234
1164114	956.761084			MPEG TS	1358	Source port: 61042 Destination port: 1234

Figura 5-3: Captura de paquetes del streaming de vídeo en Wireshark

El escenario ha sido probado realizando el streaming de videos con distintas características para probar que el sistema funciona correctamente. A continuación se muestran tres ejemplos probados donde se obtienen distintas velocidades de transferencia.

Prueba 1:

La primera prueba se ha realizado con un video en HD 1080p de pequeño tamaño y duración media. Tamaño: 35 MB y duración: 2:00. Lo que supone una velocidad de transferencia de 300 KB/s como se muestra en la Figura 5-4.

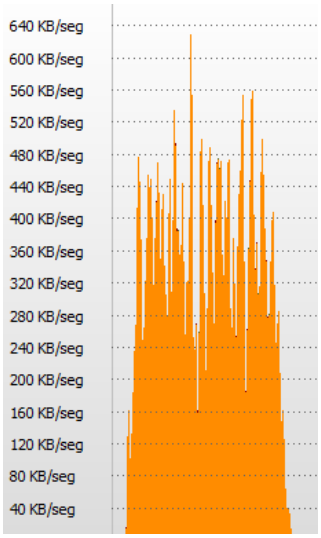

Figura 5-4: Prueba 1 (Tamaño 35 MB, Duración 2:00)

Prueba 2:

La segunda prueba se ha realizado con un video en HD 1080p de gran tamaño y duración media. Tamaño: 303 MB y duración: 2:17. Lo que supone una velocidad de transferencia de 2.26 MB/s como se muestra en la Figura 5-5.

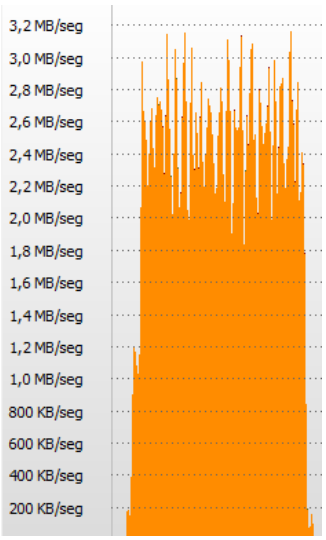

Figura 5-5: Prueba 2 (Tamaño 303 MB, Duración 2:17)

Prueba 3:

La última prueba se ha realizado con un video en HD 720p de larga duración. Tamaño: 600 MB, duración: 5:57. Lo que supone una velocidad de transferencia de 1.72 MB/s como se muestra en la Figura 5-6.

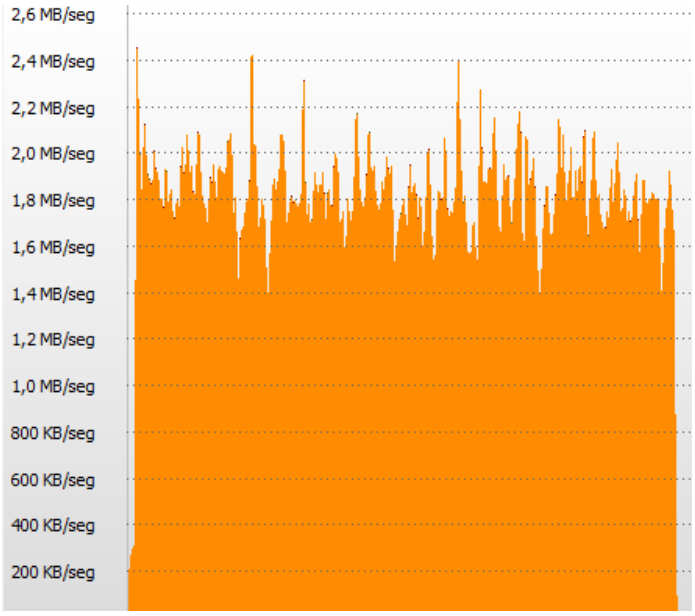

Figura 5-6: Prueba 2 (Tamaño 600 MB, Duración 5:57)

5.2 Pruebas de velocidad

Con el objetivo de medir empíricamente la velocidad alcanzable por el sistema diseñado, se han realizado pruebas en las que se ha transmitido un streaming de paquetes de datos de 1 KB a diferentes velocidades para observar el comportamiento del sistema.

Para ello se ha empleado la aplicación Jperf, la cual permite enviar un stream de datos UDP de un tamaño y a una velocidad personalizada, y se ha observado con un monitor de red la velocidad de transferencia de subida y de bajada. La interfaz de JPerf se muestra en la Figura 5-7.

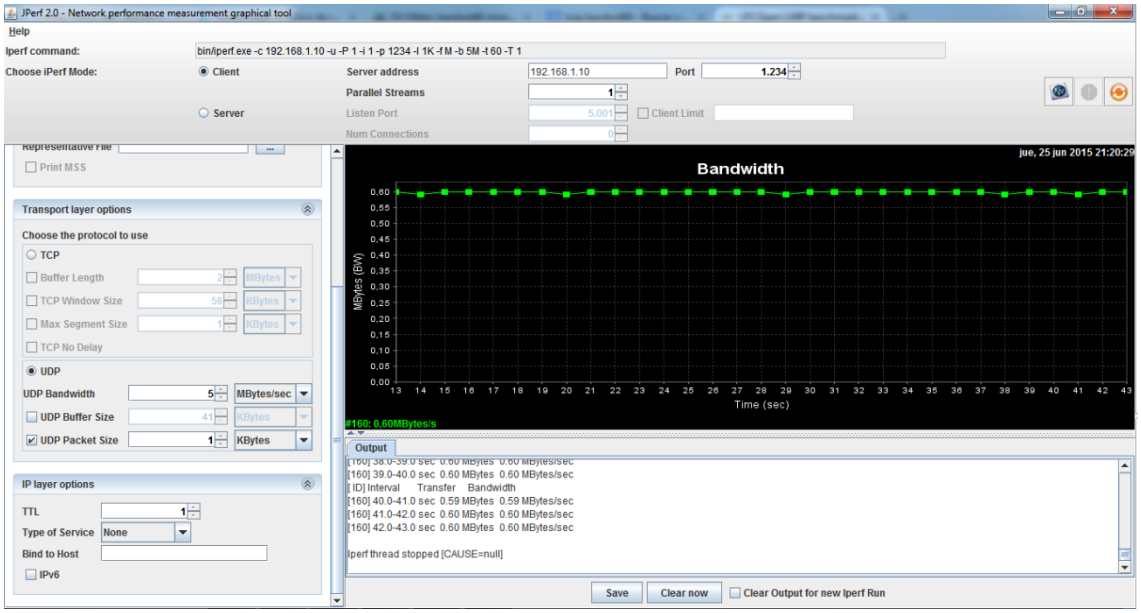

Figura 5-7: JPerf

El método para determinar la máxima velocidad estable alcanzable será el siguiente:

Se establecerá una velocidad de transferencia y se realizará una prueba con esa carga durante 2 minutos, si la velocidad de subida y bajada es estable y los datos enviados son los mismos que los recibidos se aumentará dicha velocidad para realizar otra prueba.

Si por el contrario, la velocidad no es estable durante ese tiempo o los datos recibidos no son los mismos que los enviados, se descarta la prueba.

El resultado obtenido indica que la máxima velocidad estable que se ha alcanzado es de 7.8 MB/s cuando la aplicación software se ha compilado en modo “Release”. La selección de este modo a la hora de compilar es importante, ya que permite mejorar considerablemente la velocidad del sistema. En modo “Debug” tan solo es posible alcanzar 2.9 MB/s.

5.3 Consumo de recursos de la lógica programable

El consumo de recursos de la lógica programable se muestra en la Figura 5-8 y la Figura 5-9.

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	Block RAM Tile (140)	DSPs (220)
design_1_wrapper	5175	6572	2147	3	0
design_1_i (design_1)	5175	6572	2147	3	0
axi_dma_0 (design_1_a...	2955	4119	1399	3	0
axi_mem_intercon (desi...	1317	1418	518	0	0
plc_ip_0 (design_1_plc_i...	293	290	120	0	0
U0 (plc_ip_v1_0__pa...	293	290	120	0	0
AXI_PLC (axi_plc)	115	121	61	0	0
DOWNSIZER (...	35	23	25	0	0
REG (axi_stre...	3	20	11	0	0
UPSIZER (axi_...	77	78	29	0	0
plc_ip_v1_0_S_A...	178	169	59	0	0
processing_system7_0 (...	0	0	0	0	0
processing_system7_0_...	594	714	251	0	0
rst_processing_system7...	17	31	11	0	0
xlconcat_0 (design_1_xl...	0	0	0	0	0

Figura 5-8: Consumo de recursos de la lógica programable

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	Block RAM Tile (140)	DSPs (220)
design_1_wrapper	9.72 %	6.17 %	16.14 %	2.14 %	0.00 %
design_1_i (design_1)	9.72 %	6.17 %	16.14 %	2.14 %	0.00 %
axi_dma_0 (design_1_a...	5.55 %	3.87 %	10.51 %	2.14 %	0.00 %
axi_mem_intercon (desi...	2.47 %	1.33 %	3.89 %	0.00 %	0.00 %
plc_ip_0 (design_1_plc_i...	0.55 %	0.27 %	0.90 %	0.00 %	0.00 %
U0 (plc_ip_v1_0__pa...	0.55 %	0.27 %	0.90 %	0.00 %	0.00 %
AXI_PLC (axi_plc)	0.21 %	0.11 %	0.45 %	0.00 %	0.00 %
DOWNSIZER (...	0.06 %	0.02 %	0.18 %	0.00 %	0.00 %
REG (axi_stre...	<0.01 %	0.01 %	0.08 %	0.00 %	0.00 %
UPSIZER (axi_...	0.14 %	0.07 %	0.21 %	0.00 %	0.00 %
plc_ip_v1_0_S_A...	0.33 %	0.15 %	0.44 %	0.00 %	0.00 %
processing_system7_0 (...	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
processing_system7_0_...	1.11 %	0.67 %	1.88 %	0.00 %	0.00 %
rst_processing_system7...	0.03 %	0.02 %	0.08 %	0.00 %	0.00 %
xlconcat_0 (design_1_xl...	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %

Figura 5-9: Consumo de recursos de la lógica programable (En tanto por ciento)

Como se observa en las figuras anteriores, el consumo de recursos de la infraestructura mínima para realizar un movimiento de datos entre PS y PL no es significativo en comparación con la cantidad de recursos de los que dispone el dispositivo Zynq, siendo utilizado un 16.14% de los Slices y un 2.14% de BRAMs.

El consumo de recursos del cliente de DMA (plc_ip) creado en el presente proyecto es de tan solo un 0.9% de Slices y 0% BRAMs. Sin embargo, dicho cliente no aporta ninguna funcionalidad más que la de transferir los datos de entrada a la salida tras realizar un proceso de conversión de anchura de los datos, por lo que era de esperar que el consumo de recursos fuese mínimo.

6 CONCLUSIONES

A lo largo de este proyecto se ha mostrado cómo llevar a cabo el diseño de un System on Chip basado en FPGA, en concreto para Zynq, el cual dispone de un procesador y una lógica programable en el mismo chip. La característica de Zynq de tener ambos dispositivos en un mismo circuito integrado permite utilizarlo en una gran cantidad de aplicaciones de sistemas empotrados, con la potencia que aporta el procesador y la flexibilidad de la lógica programable.

Debido a la existencia de estas dos partes, para poder obtener el máximo rendimiento es muy importante una comunicación eficiente entre ambas, siendo éste el objetivo principal del trabajo realizado.

Durante el desarrollo del proyecto ha sido necesaria una etapa inicial de aprendizaje del manejo de las distintas herramientas utilizadas, ya que algunas de ellas no habían sido utilizadas con anterioridad como por ejemplo Vivado Design Suite. Para ello han resultado de gran utilidad los manuales y guías de usuario que proporciona Xilinx junto con sus productos.

La simulación con el BFM ha supuesto una de las etapas del desarrollo más importantes del proyecto, ya que ha permitido verificar el funcionamiento del cliente DMA diseñado y localizar algunos errores que ocasionarían problemas bajo ciertas condiciones. Sin este componente la tarea de verificación hubiera sido mucho más complicada y no tan precisa.

A la hora del diseño de la arquitectura del SoC y la aplicación software se han podido aplicar los conocimientos adquiridos en una situación real, lo que ha permitido enfrentarse a los distintos problemas que han surgido. Para la resolución de los problemas ha sido necesario realizar una depuración de la aplicación software y un proceso de investigación en la documentación disponible para localizar la fuente de error.

Mediante las pruebas en la tarjeta ZedBoard y el escenario de prueba propuesto, se ha podido comprobar visualmente que el diseño realizado funciona, y el movimiento de datos entre el procesador y la lógica programable se lleva a cabo correctamente.

Un aspecto crítico a la hora de compilar el software ha sido seleccionar el modo “Release” en lugar del modo “Debug”, lo que ha permitido mejorar la velocidad de transferencia, alcanzando velocidades estables de 7.8 MB/s, en lugar de 2.9 MB/s que se conseguían en el modo “Debug”. En las pruebas de streaming de video se ha conseguido transmitir archivos de una duración considerable y gran tamaño satisfactoriamente, lo que ha permitido verificar que el sistema funciona en una situación donde la carga de datos es grande.

Respecto a los recursos necesarios de la lógica programable, se ha visto que la infraestructura básica para realizar el movimiento de datos entre PS y PL no implica un

gran impacto en el consumo de recursos del dispositivo Zynq, siendo del 16% del total, lo que permite la implementación de algoritmos en esta parte.

Como trabajos futuros, a parte de la aplicación comentada para comunicaciones por la red eléctrica, se podría realizar la misma aplicación software pero como una tarea de un sistema operativo en tiempo real Linux para poder integrarla en sistemas empotrados más complejos.

7 DIAGRAMAS

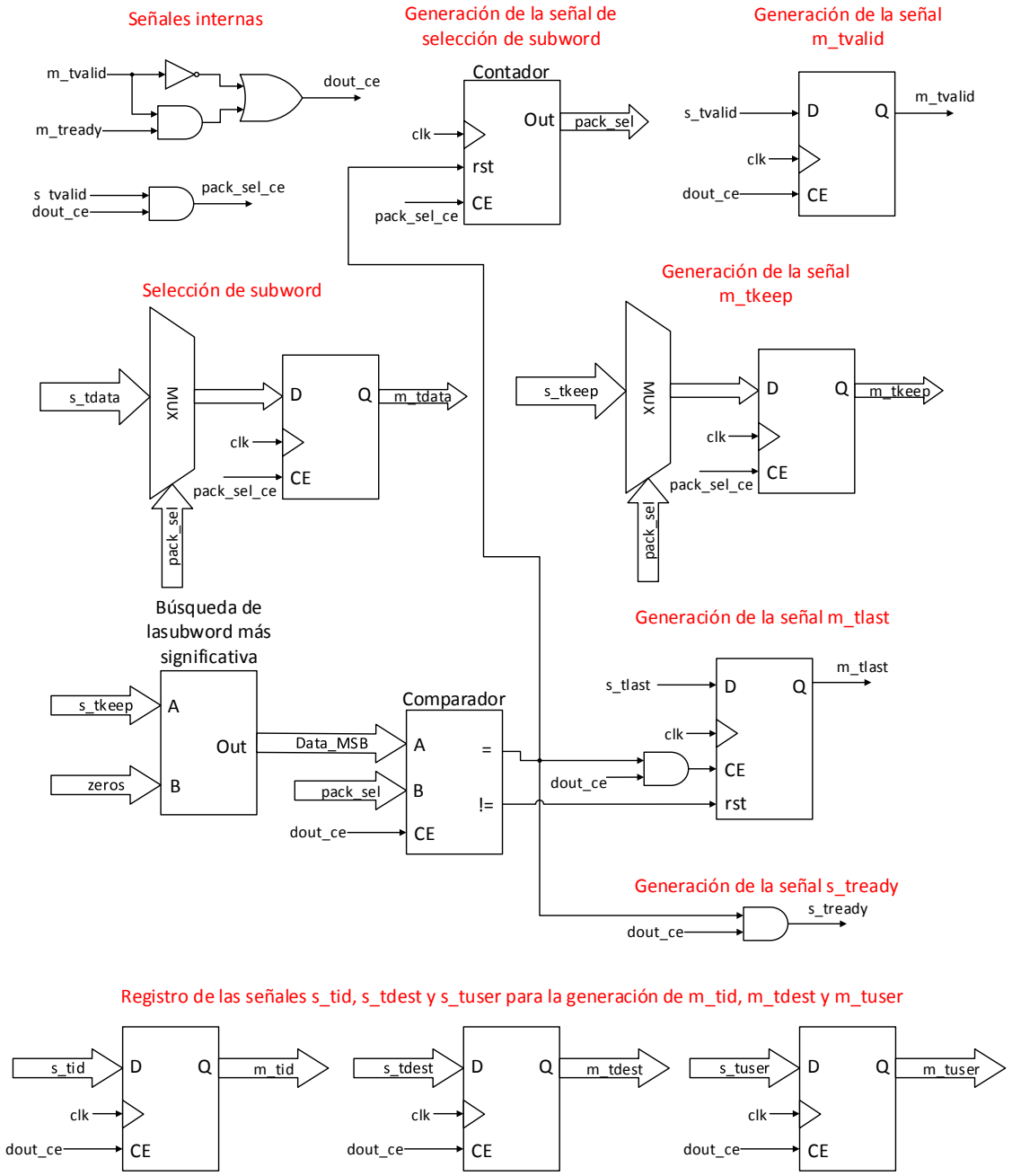

Figura 7-1: Diagrama de bloques del Downsizer

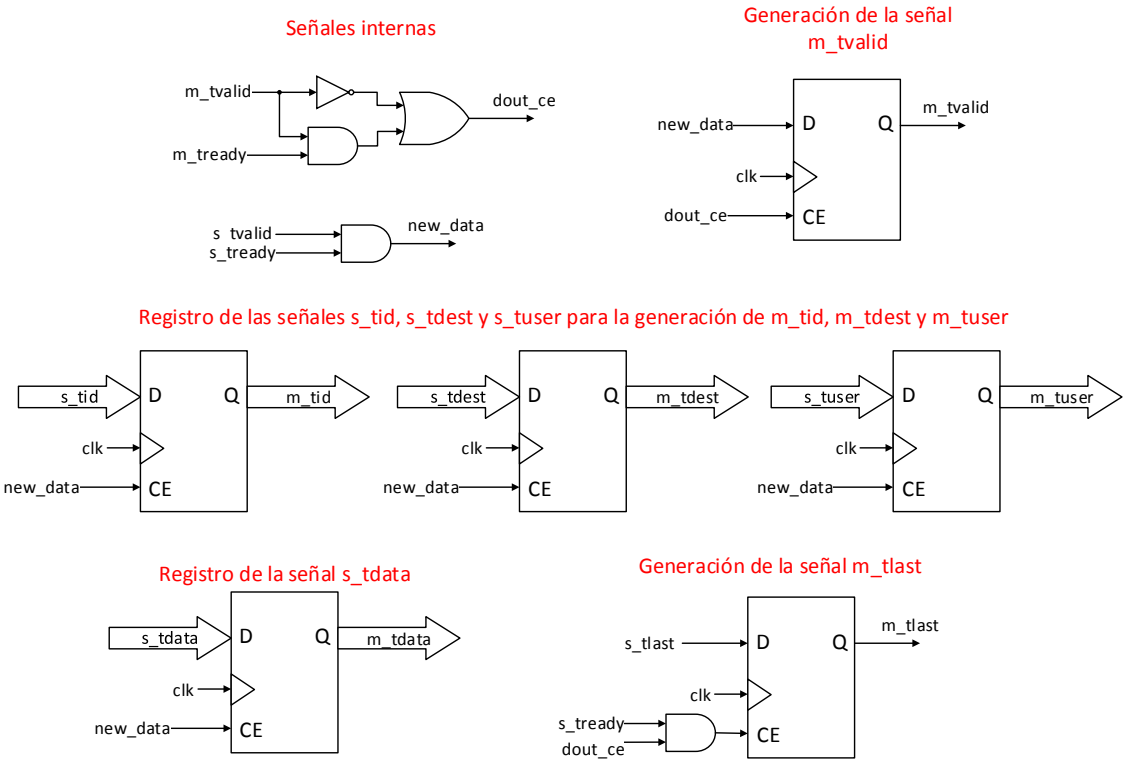

Figura 7-2: Diagrama de bloques del registro intermedio

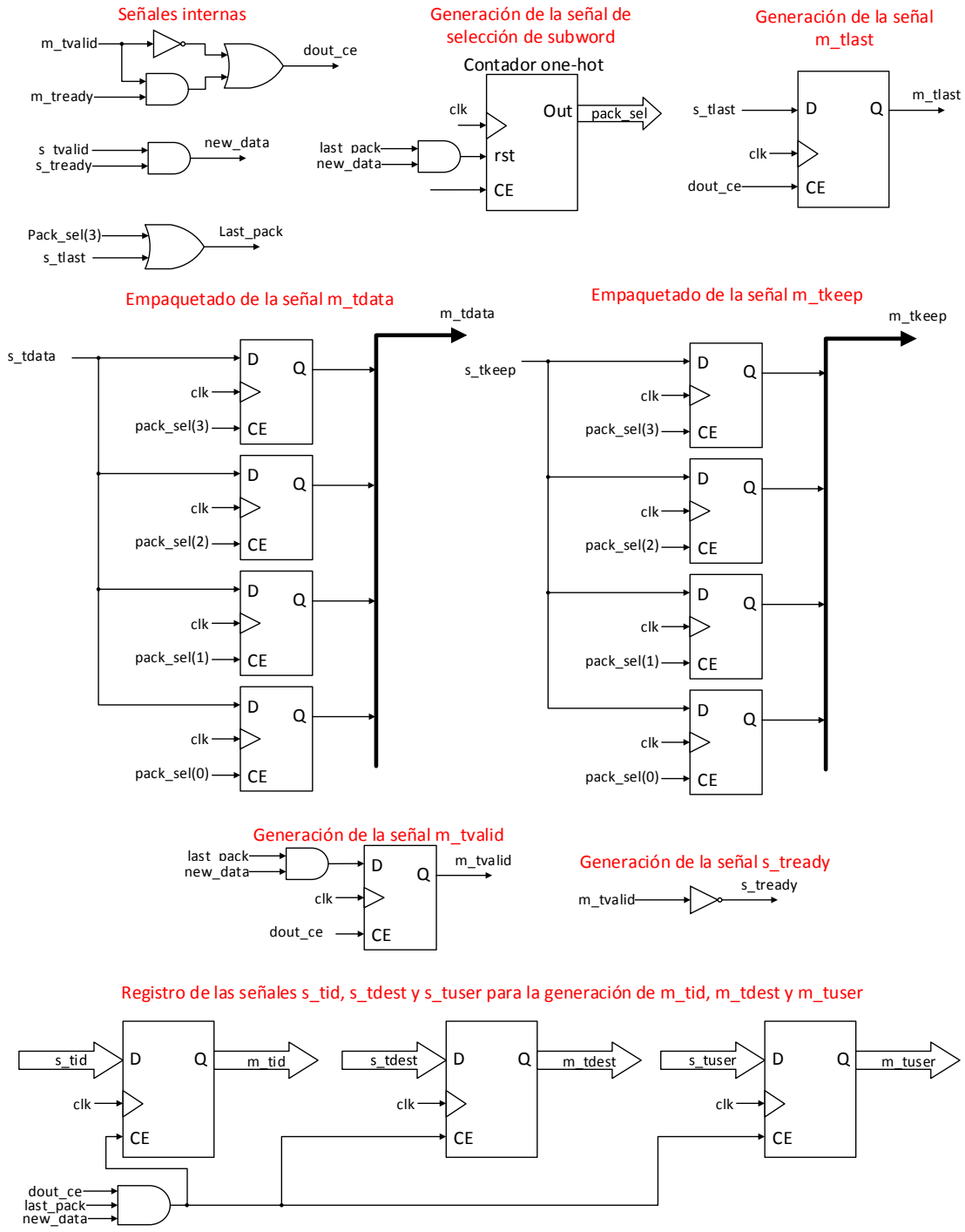

Figura 7-3: Diagrama de bloques del Upsizer

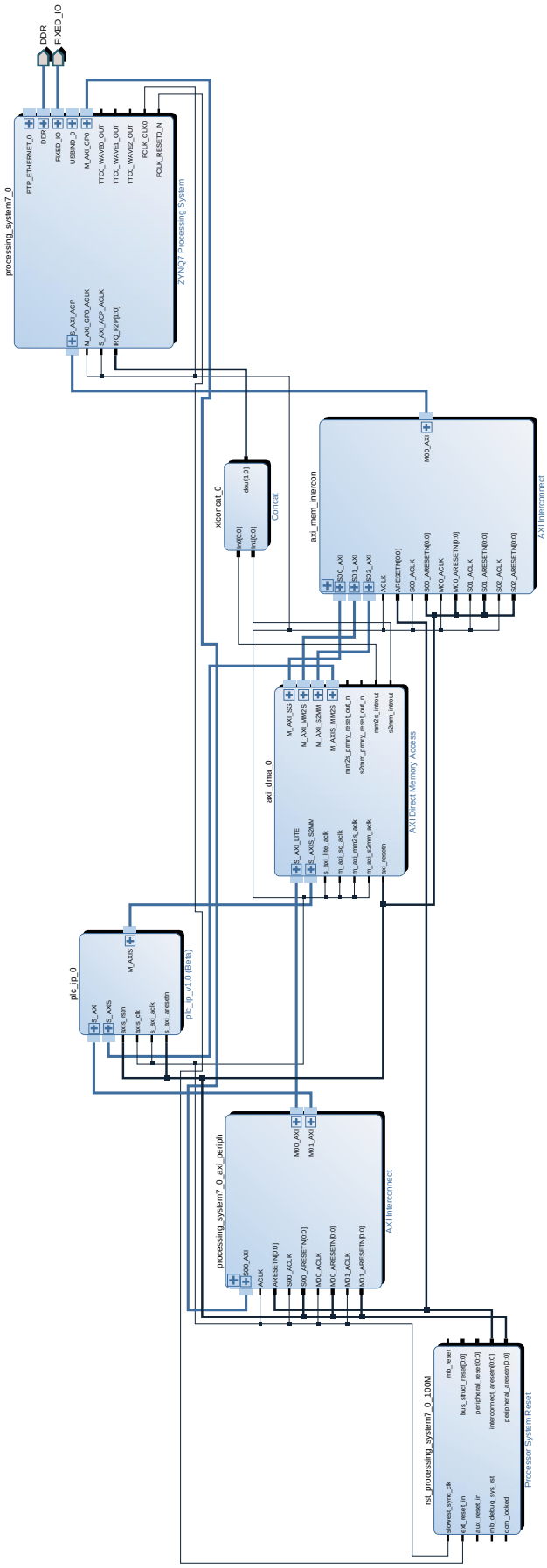

Figura 7-4: Diseño completo de la arquitectura del SoC

8 PLIEGO DE CONDICIONES

En este apartado se indican los requisitos de hardware y software necesarios para la realización del proyecto.

8.1 Requisistos Hardware

- Ordenador portátil HP Pavilion dv6
    - Procesador Intel Core i5-2410M 2.3GHz
    - 6 GB de memoria RAM
    - Windows 7
- Tarjeta de desarrollo ZedBoard
- Cable Ethernet

8.2 Requisitos Software

- Xilinx Vivado Design Suite 2014.3.1
- ModelSim 10.1a
- Microsoft Office + Visio
- Adobe Reader
- Editor de texto Notepad++
- Wireshark
- jPerf

9 PRESUPUESTO

En este apartado se realizará el desglose del presupuesto estimado para la realización de este proyecto. En él se tendrá en cuenta la mano de obra necesaria, equipos informáticos, licencias de programas y materiales, entre otros gastos.

Mano de obra

En este apartado se incluirán los gastos debidos a la mano de obra necesaria para realizar el proyecto. El tiempo necesario para el desarrollo del proyecto ha sido de 3 meses (90 días) para la parte de diseño del sistema, que a una media de 4 horas diarias hacen un total de 360 horas. Por otra parte, la redacción del libro ha supuesto una duración de 45 días, a una media de 4 horas diarias hacen un total de 180 horas.

Para los costes de mano de obra se tomará como ejemplo el caso del personal investigador de la UAH, donde el salario anual mínimo de un ingeniero recién titulado es de 25.216,77 €. Si la jornada laboral consta de un total de 1800 horas anuales, fijadas por convenio, se obtiene un salario mínimo por hora de 14€. El salario especificado para este proyecto será de 40€/hora, lo cual se ajusta a la normativa.

Tabla 9-1: Coste de la mano de obra

Tarea	Horas	Salario (Euros/Hora)	Total
Diseño del sistema	360	40	14.400 €
Redacción del libro	180	40	7.200 €
TOTAL			21.600 €

Recursos Hardware

En este apartado se incluirá el equipo utilizado para el desarrollo del proyecto, así como la tarjeta de desarrollo y materiales utilizados en las pruebas realizadas del sistema que ha sido diseñado.

Tabla 9-2: Coste de los recursos hardware necesarios

Material	Precio	Amortización	Tiempo de uso	Subtotal
Ordenador Portátil HP	799 €	3 años	5 meses	110,97 €
ZedBoard	282 €	2 años	3 meses	35,25 €
Cable Ethernet	5 €	1 años	3 meses	1,25€
TOTAL				147.47 €

(*) Precio de la edición académica para universidades

Recursos Software

Algunos de los programas utilizados requieren de licencias para el uso completo de todas las herramientas del programa, o de algunos componentes en concreto, como el caso de los BFM en Vivado. En la Tabla 9-3 se muestran los gastos asociados a las licencias de programas.

Tabla 9-3: Coste de las licencias del software

Programa	Precio	Amortización	Tiempo de uso	Subtotal
Vivado Design Suite (*)	2.647 €	2 años	5 meses	551,45 €
ModelSim	35.000 €	6 años	3 meses	1.458,33 €
Microsoft Office + Visio	668 €	3 años	3 meses	55,66 €
TOTAL				2.065,45 €

(*) La versión gratuita de Vivado (WebPack) no permite el uso del BFM ni del Logic Analyzer (ChipScope)

Conexión a Internet

Tabla 9-4: Coste de la conexión a Internet

Contrato	Duración	Precio	Total
Fibra Óptica 50Mb	5 meses	42 €/mes	210 €

Impresión y encuadernación de los libros

En este apartado se incluirán todos los materiales necesarios para la redacción, impresión y encuadernación de los libros.

Tabla 9-5: Coste de la impresión y encuadernación de los libros

Concepto	Precio
Impresión del proyecto	40 €
Encuadernación	60 €
TOTAL	100 €

Coste Total

El coste total del proyecto será la suma de cada uno de los costes calculados en el desglose anterior. En la Tabla 9-6 se muestra el presupuesto total en euros.

Tabla 9-6: Presupuesto total

Concepto	Precio
Mano de Obra	21.600 €
Recursos Hardware	147.47 €
Recursos Software	2.065,45 €
Conexión a Internet	210 €
Impresión y encuadernación	100 €
TOTAL	**23.975,45 €**

10 MANUAL DE USUARIO (PLC IP)

PLC IP es un periférico con interfaces de entrada y salida del tipo AXI4-Stream, cuyo núcleo es un core, actualmente con la funcionalidad de transferir los datos de entrada a la salida sin modificarlos (modo bypass), pero que en el futuro implementará la capacidad de modular y demodular los datos para una aplicación de comunicaciones por la red eléctrica (PLC). Este IP también incorpora una interfaz del tipo AXI4-Lite cuya función será permitir el acceso a un conjunto de registros de control y estado con los que se gestiona la operación del futuro core, por lo que actualmente sólo tiene una utilidad básica.

Debido a que el core interno tiene un bus de datos de menor tamaño que el bus de datos de entrada y salida, el IP incorpora un Downsizer a la entrada y un Upsizer a la salida para adaptar los datos al tamaño especificado por los parámetros configurables. La Figura 10-1 muestra la estructura del IP.

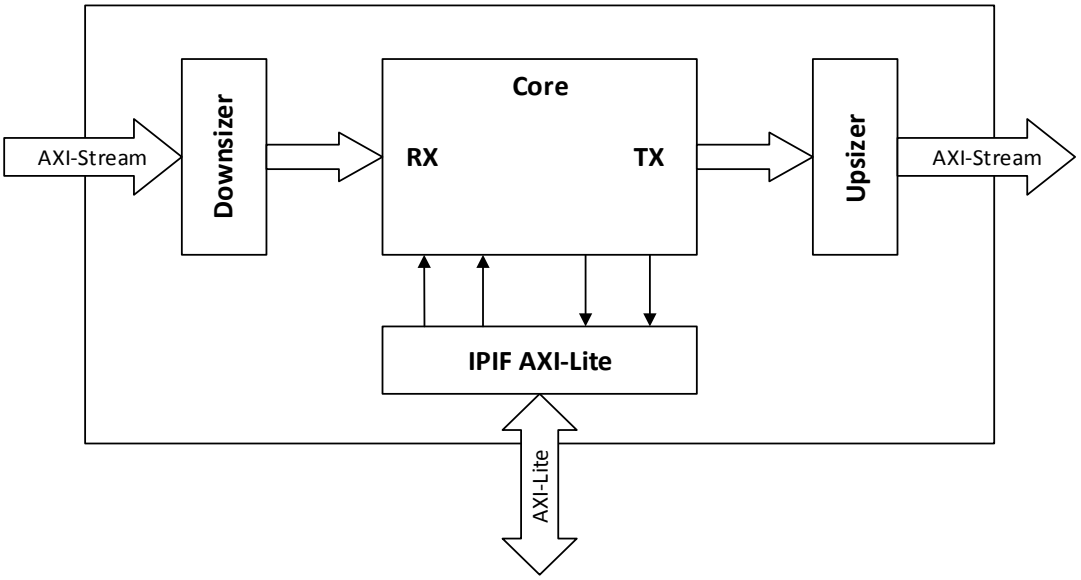

Figura 10-1: Estructura del PLC IP (Cliente de DMA)

10.1 Incorporar IP al diseño

Para reutilizar este IP en otros diseños será necesario añadirlo a un repositorio personal o al repositorio específico del proyecto. Una vez hecho esto habrá que configurar las opciones del proyecto para definir la ruta de dicho repositorio, para ello habrá que hacer click en “Add Repository” dentro de las opciones del diseño e introducir la ruta al directorio, como muestra la Figura 10-2.

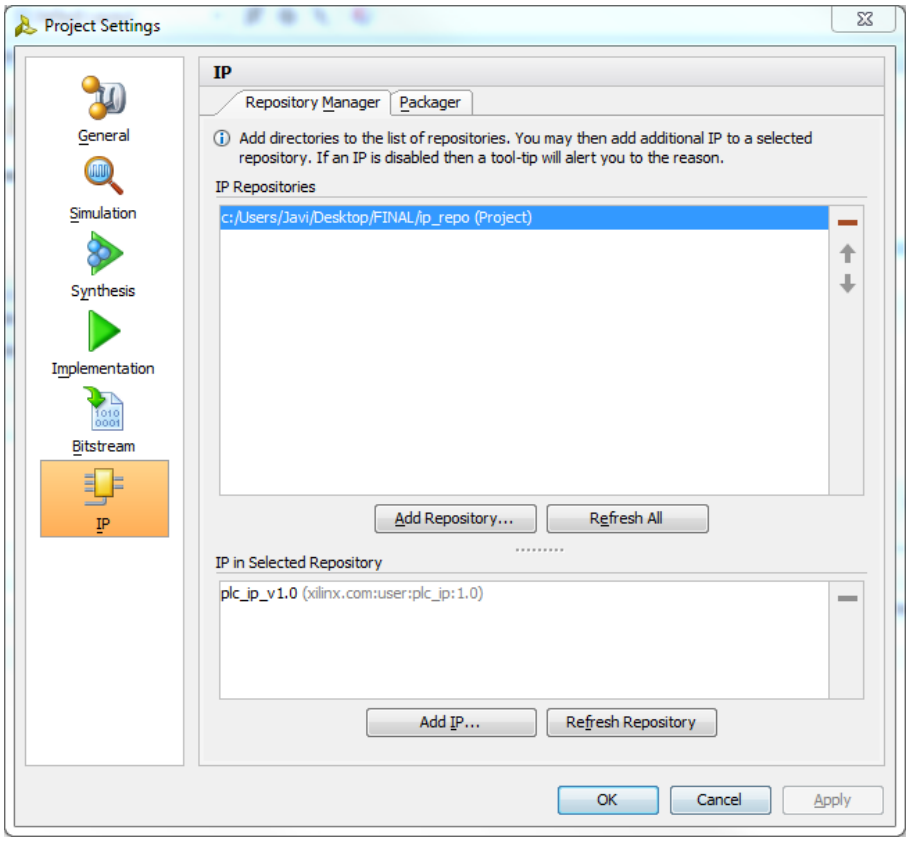

Figura 10-2: Añadir repositorio de IPs al proyecto

Una vez incorporado al repositorio, aparecerá en la lista de IPs del IP Integrator, preparado para ser utilizado en el diseño.

10.2 Señales e interfaces

El diseño del bloque PLC IP se muestra en la Figura 10-3.

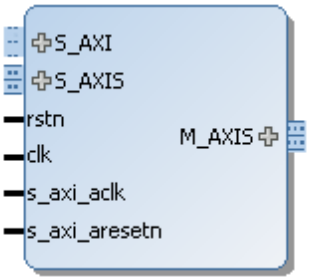

Figura 10-3: Interfaces del PLC IP

La Tabla 10-1 describe la función de las señales e interfaces de comunicaciones del PLC IP.

Tabla 10-1: Interfaces del PLC IP

Nombre	Descripción
S_AXI	Interfaz del tipo AXI4-Lite destinada a la configuración de registros internos del Core. **Funcionalidad no implementada actualmente.**
S_AXIS	Interfaz de entrada del tipo AXI4-Stream para recibir un stream de datos de un número de bits de anchura indicado por el genérico D_WIDTH_EXT.
M_AXIS	Interfaz de salida del tipo AXI4-Stream para enviar el stream de datos procesados por el Core, con una anchura de bus de datos indicada por D_WIDTH_EXT.
rstn	Señal de reset de las interfaces AXI4-Stream. Activa a nivel bajo.
clk	Señal de reloj de las interfaces AXI4-Stream.
s_axi_aclk	Señal de reloj de la interfaz AXI4-Lite.
s_axi_aresetn	Señal de reset de la interfaz AXI4-Lite. Activa a nivel bajo.

10.3 Parámetros de configuración

PLC IP contiene diversos parámetros configurables para hacer el diseño más versátil y permitir que sea reutilizable en diseños con distintas especificaciones. Los parámetros se dividen en dos grupos, los que hacen referencia a la interfaz AXI4-Lite, y los de las interfaces AXI4-Stream. Actualmente solo podrán ser configurados los de éstas últimas, ya que la interfaz AXI4-Lite no implementa ninguna funcionalidad.

Estos parámetros se configurarán en la ventana “Customize IP”, que se muestra en la Figura 10-4. La Tabla 10-2 muestra la lista de parámetros asociados a las interfaces AXI4-Stream.

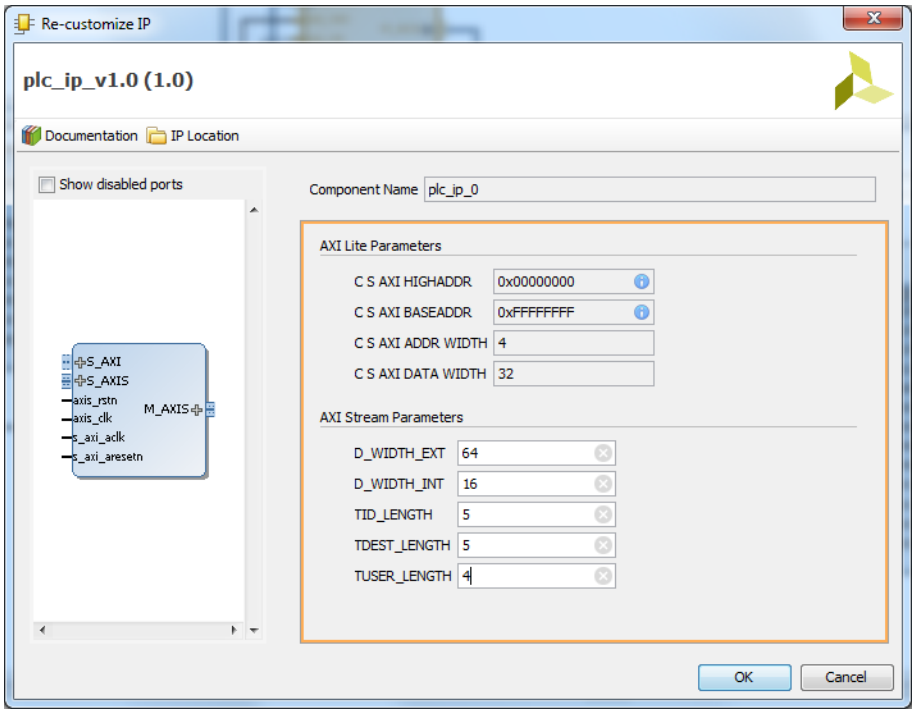

Figura 10-4: GUI de configuración de los parámetros del IP

Tabla 10-2: Parámetros del PLC IP

Nombre	Descripción
D_WIDTH_EXT	Número de bits del bus de datos AXI4-Stream a la entrada y a la salida del IP.
D_WIDTH_INT	Número de bits del bus de datos interno AXI4-Stream. Debe ser de un tamaño menor que D_WIDTH_EXT.
TID_LENGTH	Número de bits del bus TID.
TDEST_LENGHT	Número de bits del bus TDEST.
TUSER_LENGTH	Número de bits del bus TUSER.

BIBLIOGRAFÍA

[ARM, 2004] ARM. (2004). AMBA® AXI Protocol Specification v1.0.

[ARM, 2010] ARM. (2010). AMBA® 4 AXI4-Stream Protocol Specification v1.0.

[ARM, 2011] ARM. (2011). AMBA® AXI™ and ACE™ Protocol Specification. AXI3™, AXI4™, and AXI4-Lite™ ACE and ACE-Lite™.

[Bagni et al., 2013] Bagni, D., Noguera, J., & Martinez Vallina, F. (2013). Zynq-7000 All Programmable SoC Accelerator for Floating-Point Matrix Multiplication using Vivado HLS (XAPP1170).

[Crockett et al., 2014] Crockett, L. H., Elliot, R. A., Enderwitz, M. A., & Stewart, R. W. (2014). *The Zynq Book - Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC.* Glasgow: Strathclyde Academic Media. ISBN-13: 978-0992978709.

[Dunkels, 2001] Dunkels, A. (2001). Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science.

[LwIP, 2004] LwIP. (2004). *LwIP Wiki*. Obtenido de http://lwip.wikia.com/wiki/LwIP_Wiki

[Pasricha & Dutt, 2008] Pasricha, S., & Dutt, N. (2008). *On-Chip Communication Architectures: System on Chip Interconnect (Systems on Silicon).* Burlington: Morgan Kaufmann. ISBN-13: 978-0123738929.

[Sutherland et al., 2006] Sutherland, S., Davidmann, S., & Flake, P. (2006). *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Springer. ISBN-13: 978-0387333991.

[Xilinx, 2013] Xilinx. (2013). Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994).

[Xilinx, 2014a] Xilinx. (2014a). LogiCORE IP Product Guide AXI BFM Cores v5.0 (PG129).

[Xilinx, 2014b] Xilinx. (2014b). LogiCORE IP Product Guide AXI DMA v7.1 (PG021).

[Xilinx, 2014c] Xilinx. (2014c). Vivado Design Suite: Creating and Packaging Custom IP (UG1118).

[Xilinx, 2015] Xilinx. (2015). Zynq-7000 AP SoC Technical Reference Manual (UG585).

[ZedBoard, 2012] ZedBoard. (2012). ZedBoard Hardware User's Guide.

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR