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

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Plataforma FPGA PCI Express para aplicaciones Big Data

Autor: Irene González Crespo
Tutor(es): Antonio Núñez Ordóñez
Pedro Pérez Carballo
Fecha: Julio 2018

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

Versión: 3.4 – 27 julio 2018

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

Máster en Tecnologías de Telecomunicación

Trabajo Fin de Máster

Plataforma FPGA PCI Express para aplicaciones Big Data

HOJA DE FIRMAS

Alumna:	Irene González Crespo	Fdo.:	GONZALEZ CRESPO IRENE - 44739793D Firmado digitalmente por GONZALEZ CRESPO IRENE - 44739793D Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-44739793D, givenName=IRENE, sn=GONZALEZ CRESPO, cn=GONZALEZ CRESPO IRENE - 44739793D Fecha: 2018.07.30 09:53:50 +01'00'
Tutor:	Antonio Núñez Ordóñez	Fdo.:	NUÑEZ ORDOÑEZ ANTONIO - 50278395G Firmado digitalmente por NUÑEZ ORDOÑEZ ANTONIO - 50278395G Fecha: 2018.07.29 11:42:55 +01'00'
Tutor:	Pedro Pérez Carballo	Fdo.:	PEREZ CARBALLO PEDRO FRANCISCO - 42163642L Firmado digitalmente por PEREZ CARBALLO PEDRO FRANCISCO - 42163642L Fecha: 2018.07.30 09:27:41 +01'00'
Fecha:	**Julio 2018**		

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

Versión: 3.4 – 24 julio 2018

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

Máster en Tecnologías de Telecomunicación

Trabajo Fin de Máster

Plataforma FPGA PCI Express para aplicaciones Big Data

HOJA DE EVALUACIÓN

Calificación: .....

Presidente	Fdo.:
Secretario	Fdo.:
Vocal	Fdo.:

Fecha: Julio 2018

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

AGRADECIMIENTOS

Gracias a mis tutores, Pedro Pérez Carballo y Antonio Núñez Ordóñez, por su tiempo y dedicación. Sin su apoyo y guía no hubiera podido finalizar este Trabajo Fin de Máster. Gracias por todos los conocimientos y consejos que me han brindado a lo largo de este Máster y este proyecto.

No obstante, también he de agradecer a mi familia y amigos haberme apoyado a pesar de las complicaciones. Gracias por siempre estar a mi lado y demostrarme que con dedicación y esfuerzo todo se puede conseguir.

RESUMEN

Este Trabajo Fin de Máster está formado por dos partes fundamentales. La primera se centra en la implementación del bus de comunicaciones PCI Express para facilitar una comunicación rápida, eficaz y eficiente entre la placa de prototipado FPGA y el *host* controlador del sistema. Obteniendo así una plataforma FPGA que utiliza PCI Express como interfaz de datos. La segunda se centra en el desarrollo de un clasificador de datos apto para aplicaciones de Big Data, permitiendo trabajar con datos en tiempo real. Por ello, se procede a la combinación de ambas partes en un sistema de procesamiento para Big Data, aunque se permite que el diseño pueda ser reutilizado para otros tipos de aplicaciones (DPI, CEP, …). La implementación del sistema se ha realizado sobre la placa de desarrollo de Xilinx ZC706.

Inicialmente, se describe el bus de comunicaciones PCI Express, con todas las particularidades que lo caracterizan y su funcionamiento. De esta manera, se tiene una visión clara de las amplias bases que cubre y los beneficios que aporta a un sistema. A continuación, se explica el bloque utilizado para su implementación, RIFFA. Se tratan sus características principales y su arquitectura, para posteriormente proceder a su implementación y los pasos necesarios para adaptarlo a la FPGA utilizada. También, se realiza un breve estudio de los diferentes clasificadores estudiados, los cuales pueden ser implementados en diferentes aplicaciones Big Data, haciendo especial hincapié en el seleccionado para el desarrollo de este proyecto. El clasificador escogido se basa en el algoritmo de *bitonic sorting*, del cual se muestra su arquitectura y los cambios realizados para su posterior implementación sobre la FPGA.

Finalmente, se realiza la implementación del sistema completo sobre la FPGA, incluyendo el bus de comunicaciones PCI Express y el clasificador de *bitonic sorting*, incluyendo las fases de verificación y validación del sistema. Como conclusión, se puede observar que se consiguen elevadas tasas de transmisión de datos gracias a la implementación del bus de comunicaciones PCI Express a través de RIFFA y se obtiene un sistema completo apto para aplicaciones de Big Data. El sistema es capaz de trabajar en tiempo real y con elevadas tasas de transmisión, obteniendo un sistema funcional, fiable y efectivo.

ABSTRACT

This end-of-master project has two key parts. The first one focus on the implementation of the PCI Express communication bus. That allows a fast and efficient communication between the FPGA and the system host. By doing so the FPGA allows the use of PCI Express as a data interface. The second one focus on the development of a data sorter for Big Data applications. This allows the use of streamed data. For this reason, the combination of both parts allows the consolidation of a processing system for Big Data, but it also allows its re-utilization for other applications (DPI, CEP, …). The system implementation has been done on an ZC706 Xilinx development board.

First of all, the PCI Express communication bus is described with its characteristics and its behaviour. This way, we obtain a perspective of its fundamentals and its advantages. Hereafter, RIFFA is explained as the selected block for its implementation, describing its architecture and characteristics. Then the implementation and the changes for the selected FPGA are explained. Likewise, a brief study of different sorters is made, where each one of them can be implemented on different Big Data applications. The bitonic sort is the one selected for this project and its architecture and accommodation for the FPGA are exposed.

Finally, the implementation of the entire system over the FPGA is done, including the PCI Express communication bus and the bitonic sorter. The verification and validation phases are exposed in order to obtain the execution parameters. The entire system is able to work in real time with high transmission rates. In conclusion, the designed system on an embedded platform is functional, reliable and effective.

Tabla de contenido

Índice de Figuras

Índice de Tablas

Acrónimos

AC	Alternating Current
ACK	Acknowledgement
AMBA	Advanced High-Performance Bus
API	Application Programming Interface
AXI	Advanced Extensible Interface
BAR	Base Address Register
BRAM	Block Random Access Memory
BSP	Board Support Package
BUFG	Global Buffers
CEP	Complex Event Processing
CRC	Cyclic Redundancy Check
DC	Direct Current
DFCL	Distributed Crossproducting of Field Labels
DFT	Design for Test
DMA	Direct Memory Access
DLLP	Data Link Layer Packet
ECRC	End-to-end CRC
FF	Flip Flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FWFT	First Word Fall Through
GUI	Graphical User Interface
HLS	High Level Synthesis
IDE	Integrated Design Environment
IDSEL	Initialization Device Select
IP	Internet Protocol
IPIF	IP Interface
IUMA	Instituto Universitario de Microelectrónica Aplicada
LCRC	Link CRC

LTSSM	Link Training and Status State Machine
LUT	Look-Up Tables
LUTRAM	Look-Up Tables RAM
MSI	Message Signaled Interrupts
NAK	Negative Acknowledgement
OpenCL	Open Computing Language
PCI	Peripheral Component Interconnect
PL	Physical Layer
PLB	Processor Local Bus
PLL	Phase-Locked Loop
PLP	Physical Layer Packet
PSD	Packet Size Distribution
PSV	Packet Size Variation
QoS	Quality of Service
RAS	Reliable, Available, Serviceable
RCRB	Root Complex Register Block
RIFFA	Reusable Integration Framework for FPGA Accelerators
RTL	Register Transfer Level
SDK	Software Development Kit
SICAD	Sistemas Industriales y CAD
SoC	System on Chip
TFM	Trabajo Fin de Máster
TLP	Transaction Layer Packet
ULPGC	Universidad de Las Palmas de Gran Canaria
VC	Virtual Channel
VLAN	Virtual Local Area Network

Capítulo 1. Introducción

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

1.1. Antecedentes

Con el paso de los años nos encontramos en un mundo interconectado con un crecimiento significativo del tráfico de datos. Por ello, cada vez son necesarios nuevos métodos capaces de procesar en tiempo real una gran cantidad de información contenida en los datos que se capturan y se transmiten a través de la red. Esto implica realizar mejoras en los sistemas, permitiendo aumentar el ancho de banda y la latencia de los mismos [1], [2]. El paradigma Big Data [3] aparece como solución a estos problemas permitiendo el almacenamiento de grandes cantidades de datos, así como de su procesamiento. Una de las principales características que ofrece Big Data es la inclusión de diferentes procedimientos que permiten la búsqueda de patrones repetitivos para facilitar el almacenaje de grandes cantidades de datos. Sin embargo, aún no se ha conseguido optimizar al máximo Big Data y hay muchos sistemas que aún no lo integran.

1.1.1. Big Data

Se pretende desarrollar un sistema capaz de mejorar tanto el procesamiento, distribución y almacenaje de datos en *streaming*, como de datos estáticos almacenados previamente en memoria para una placa de desarrollo tipo FPGA. Dentro de la estructura básica de Big Data se encuentran dos partes fundamentales la etapa *map* y la etapa *reduce*, como se puede observar en la Figura 1. MapReduce es un modelo de programación paralelo para grandes cantidades de datos y es utilizado en Big Data [3], [4] .

No obstante, una de las limitaciones con las que nos encontramos es la necesidad de realizar una clasificación de datos antes de proceder a su procesado y distribución, permitiendo facilitar dichas tareas con el conocimiento previo de los tipos de datos recibidos. Hasta ahora, únicamente es

posible realizar la clasificación de datos entre las etapas *map* y *reduce*. Por ello, se propone desarrollar un clasificador que se encargará de organizar los datos durante su captura u obtención de memoria, facilitando así el proceso del resto de bloques del sistema. Posteriormente, ya será posible pasar al particionado de los datos y su procesado y distribución [1], [5]. Respecto al tipo de clasificador a desarrollar, se ha optado por un diseño abierto, de tal forma que se obtenga un clasificador genérico que permita su posterior adaptación a la aplicación del bloque de procesamiento. Sin embargo, cabe destacar que se hará uso de los metadatos incluidos dentro del *payload* de los paquetes recibidos para proceder a la clasificación de los datos.

Como consecuencia, se puede decir que la primera tarea principal de este proyecto se centra en la captura, clasificación y distribución de los datos para su posterior procesamiento en una arquitectura que incluye Big Data.

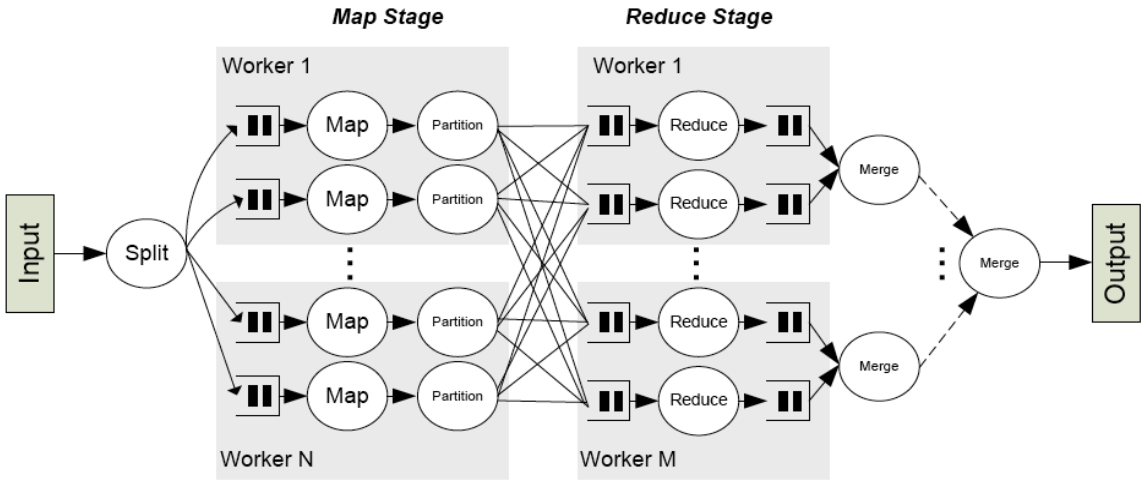

Figura 1: Ejemplo de flujo de datos en una aplicación Big Data [4]

1.1.2. Uso del bus PCI Express

Sin embargo, también se considera importante el ancho de banda de las interfaces entre la placa de prototipado FPGA y el *host* controlador del sistema. Por ello, para poder contar con una comunicación más rápida, eficaz y eficiente se ha decidido hacer uso del bus PCI Express para aprovechar el ancho de banda disponible [6], [7]. Esto se debe a que nuestro sistema requiere de una conexión dedicada punto a punto, como alternativa al uso de una unidad TEMAC para realizar la conexión entre el *host* y la FPGA. Al hacer uso del bus PCI Express, el bloque TEMAC no será utilizado como bloque principal para la transferencia de datos.

El uso de PCI Express ofrece ventajas a la plataforma final diseñada. Cabe indicar que se ha seleccionado la placa de prototipado Xilinx ZC706 (Figura 2) que incluye el SoC FPGA Zynq XC7Z045 FFG900 de la serie 7 de Xilinx. Esta elección se debe principalmente a la integración de un bloque *hard* IP PCI Express en el SoC, que facilita la implementación de sistemas usando el bus

estándar industrial PCI Express [8]. Dicho IP es compatible con la especificación 2.1 de PCI Express y soporta funcionalidad de punto final (*Endpoint*) y de puerto raíz (*Root*). Permite ser configurado para soportar tanto las especificaciones Gen1 (2,5 GT/s) como Gen2 (5 GT/s) y puede operar con 1, 2, 4, u 8 *lanes* o canales. El tamaño máximo del *payload* es de 1.024 Bytes [9]. El bloque integrado PCI Express de Xilinx posee una interfaz de datos AXI4-Stream hacia la lógica interna de la FPGA, tal como se muestra en la Figura 3. También, se puede observar en dicha Figura 3 los principales módulos e interfaces del bloque PCI Express de Xilinx.

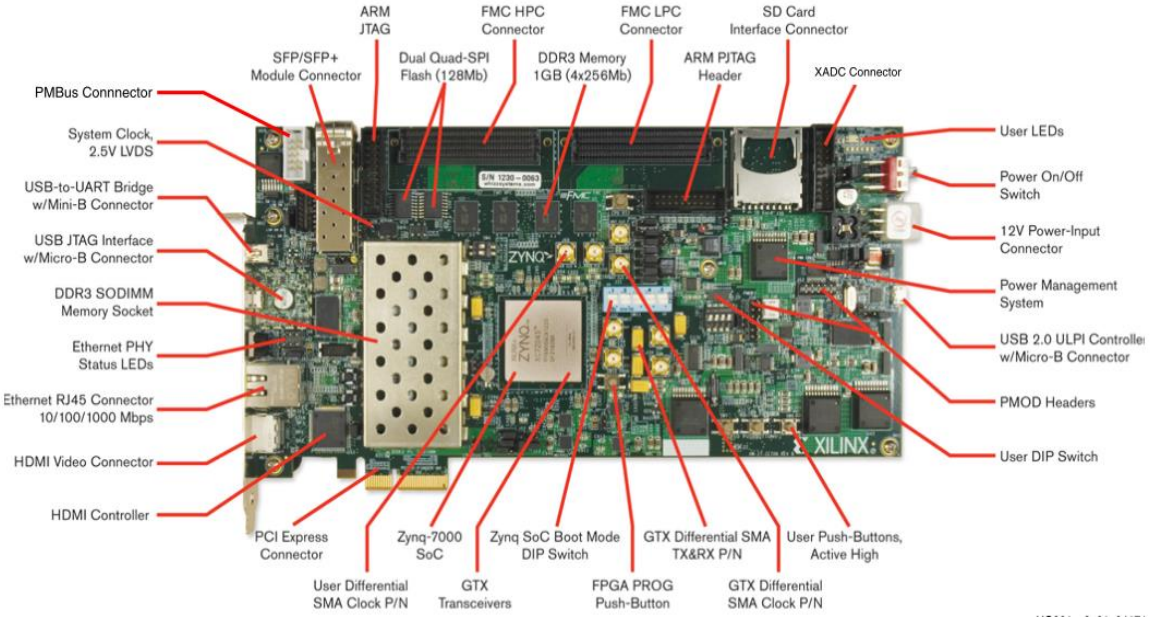

Figura 2: Tarjeta de prototipado Xilinx ZC706 [10]

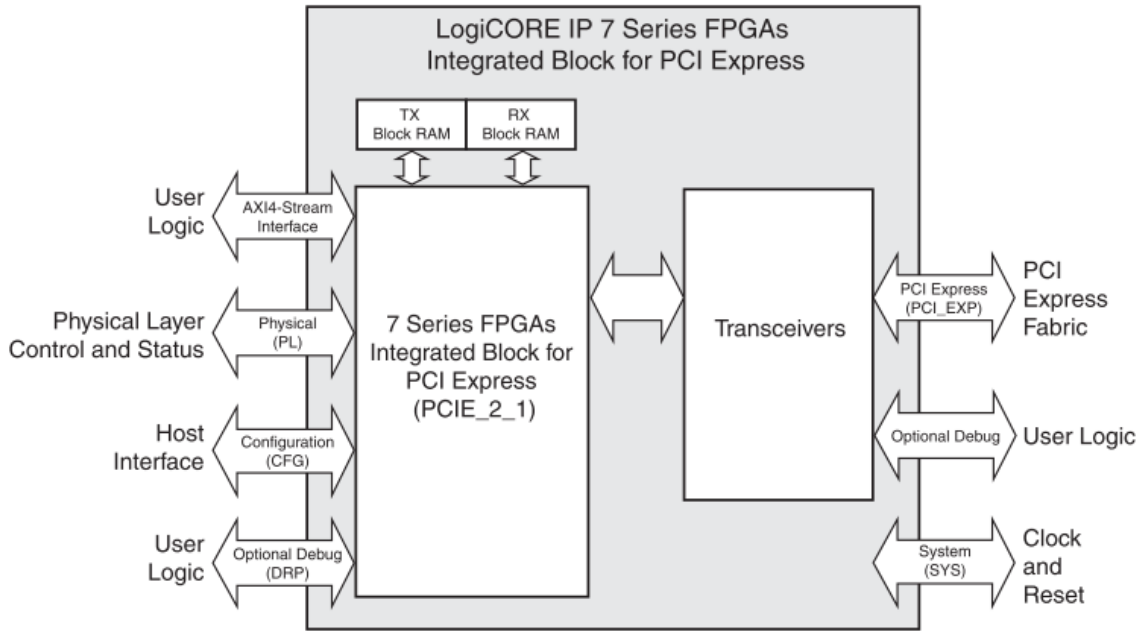

Figura 3: Principales módulos e interfaces del Bloque PCI Express de Xilinx [9]

Una vez conocida la plataforma de desarrollo sobre la que se realiza el Trabajo de Fin de Máster, se pueden destacar ciertos aspectos de interés que ayudan a entender su elección [11]. La tarjeta de prototipado Xilinx ZC706 [10] dispone de cuatro canales (4-*lane*) PCI Express, permitiendo cada uno velocidades Gen1 o Gen2. Esto supone un ancho de banda máximo efectivo de 8 Gbps (Gen1) o 16 Gbps (Gen2) teniendo en cuenta una reducción teórica del 20% debido a los bits adicionales necesitados por el protocolo [7], [12].

Otra característica soportada por el bloque PCI Express es la posibilidad de reconfigurar la FPGA utilizando la interfaz DRP (*Dynamic Reconfiguration Port*). Este puerto tiene acceso de lectura y escritura a la memoria de configuración de la FPGA. Se pretende utilizar esta característica para realizar la reconfiguración del clasificador, por lo que el uso de PCI Express permite mantener el diseño abierto del clasificador mencionado anteriormente.

En la Figura 4 se muestra la arquitectura básica escogida para la transmisión de datos entre el *host* y la FPGA, haciendo uso del bus de comunicaciones PCI Express. La idea inicial es hacer uso de los bloques propios incluidos dentro de la FPGA para realizar las comunicaciones mediante el bus PCI Express. Para ello, la conexión entre el *host* y la FPGA, se realizará haciendo uso del propio bus PCI Express. Posteriormente, se realiza la comunicación con el resto de bloques de la FPGA mediante conexiones AXI4-Stream. Primero se procederá a conectar el bloque PCI Express de la FPGA con el bloque AXI de la misma. A continuación, se encuentra el clasificador explicado anteriormente y el bloque de procesamiento del sistema. No obstante, también es necesario tener en cuenta la lógica de interrupciones para el sistema y la conexión existente entre el bloque de procesamiento y el bloque PCI Express, para poder comunicarse con el *host* y devolver la información necesaria requerida [12].

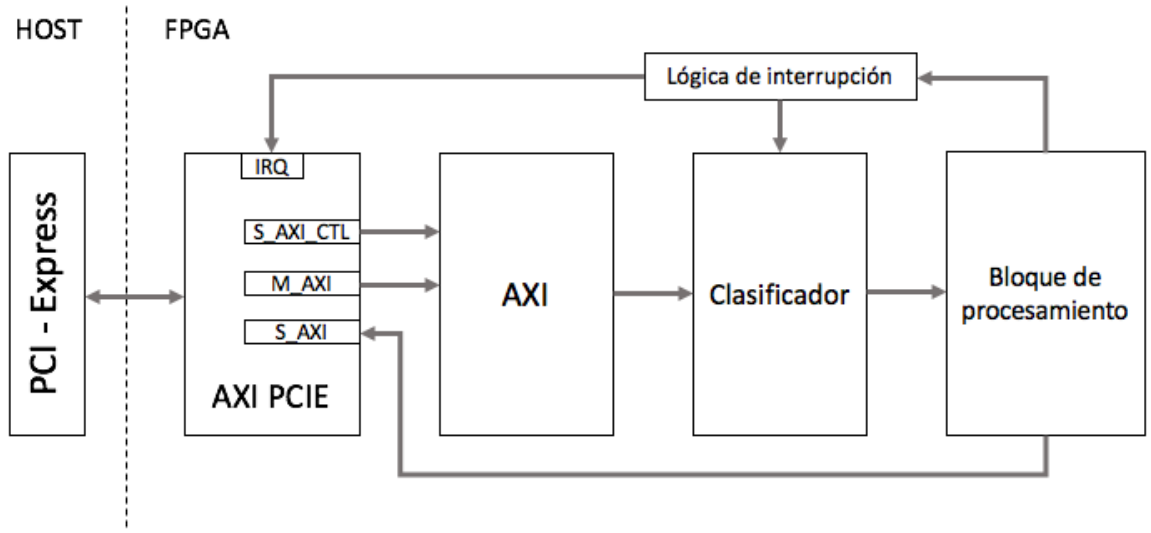

Figura 4: Arquitectura básica para la transmisión de datos entre el host y la FPGA a través de PCI Express

1.1.3. Flujo de diseño

El flujo de diseño del proyecto, al hacer uso de dos líneas de trabajos independientes que se integran en este trabajo, se basa en el desarrollo individual de cada una de ellas y su posterior unificación para obtener un sistema completo y funcional. Para ello, se ha hecho uso del entorno de herramientas de diseño Xilinx Vivado Design Suite, que permite la implementación de sistemas descritos en lenguaje de alto nivel en plataformas hardware tipo FPGA.

Se hace uso del entorno de diseño integrado (IDE), incluyendo la interfaz de usuario gráfica (GUI) disponible dentro del entorno de Vivado Design Suite. Su principal función es proporcionar al usuario la posibilidad de realizar el ensamblaje, la implementación y la validación del sistema, incluyendo el uso de bloques IP. En la Figura 5 se muestra el flujo de diseño para Vivado Design Suite [13].

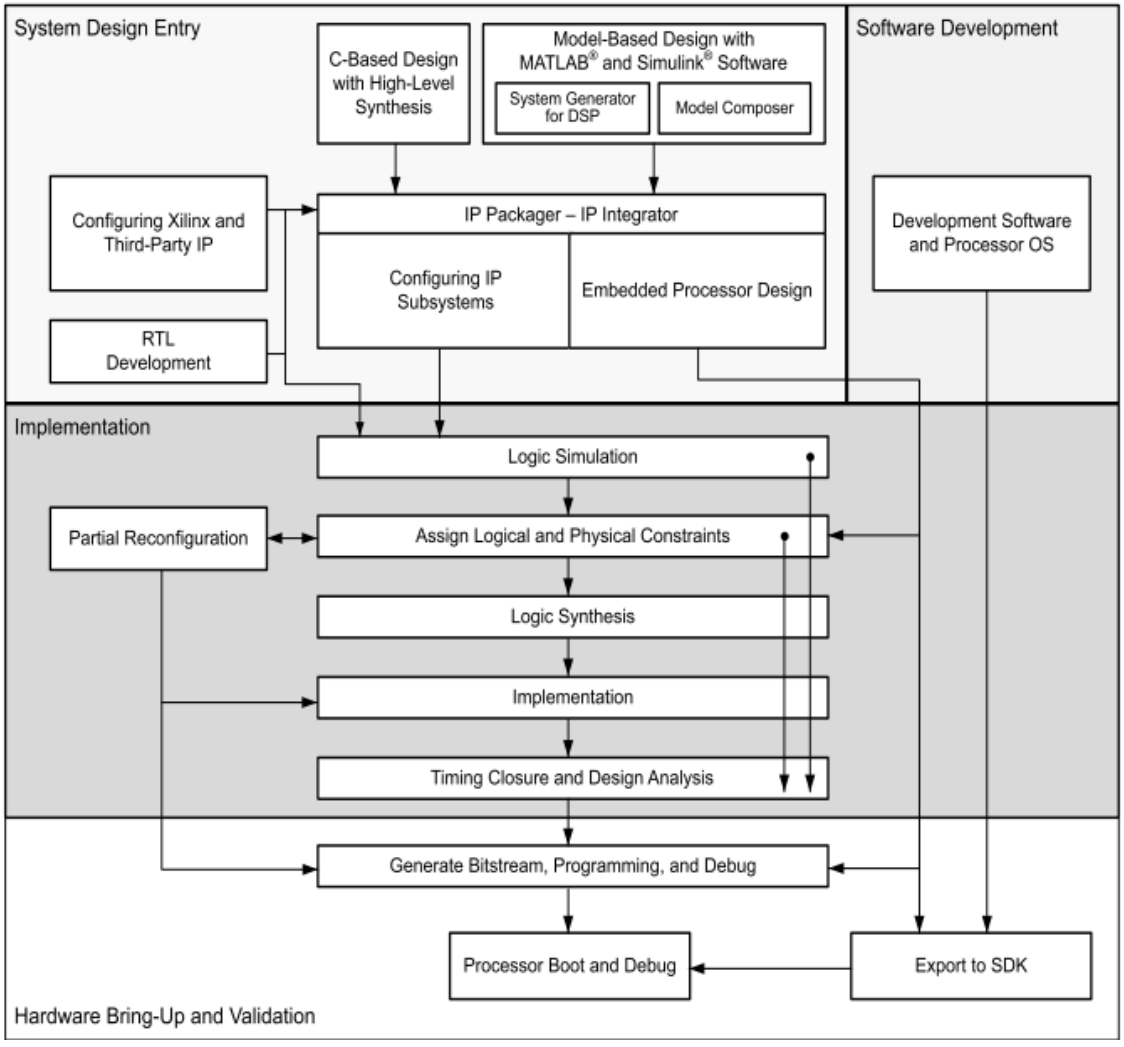

Figura 5: Flujo de diseño de Vivado Design Suite [13]

Los pasos que conforman este flujo son la síntesis, la implementación, el análisis temporal y de potencia y la generación del *bitstream*. La fase de implementación comienza con la simulación lógica, la asignación de recursos, tanto físicos como lógicos, y la síntesis lógica. Finalmente, se obtiene el *bitstream* que será exportado a la herramienta SDK para la programación y el depurado de la plataforma. SDK es un entorno de desarrollo de software empotrado que permite la programación de la plataforma FPGA y su depuración, permitiendo crear aplicaciones que posteriormente pueden ser integradas en la FPGA [14]. El flujo de trabajo de SDK consiste en:

1. Invocar SDK.
2. Crear un nuevo espacio de trabajo.
3. Crear un archivo BSP con los drivers y librerías necesarias para la aplicación.
4. Programar la FPGA con el *bitstream* y el BSP.
5. Comprobación de funcionamiento y depurado.

Vivado Design Suite proporciona dos tipos diferentes de flujos de diseño: flujo RTL (para códigos de tipo Verilog, VHDL, SystemVerilog, XDC o basados en C) o flujo de síntesis con herramientas externas (para archivos EDIF o Verilog estructural). No obstante, para este proyecto se ha hecho uso del flujo de diseño RTL, ya que este está basado en el uso de bloques IP y en la estandarización de la arquitectura de comunicaciones. El entorno de desarrollo ya proporciona una amplia gama de bloques IP estandarizados que se encuentran en el catálogo de Xilinx.

Tras la adición de los bloques IP al sistema, se pasa al empaquetado de los bloques IP y a su integración haciendo uso de la herramienta IP Packager, encargada de realizar la conversión de los diferentes módulos o modelos incorporados en bloques IP aptos para su uso en el entorno de desarrollo.

A continuación se hace uso de la herramienta IP Integrator, que proporciona un entorno de trabajo para la generación del diagrama de bloques del sistema. Se incluyen los bloques IP deseados y se realizan las interconexiones correspondientes para obtener el sistema completo que posteriormente será implementado en la plataforma seleccionada. En este momento es posible realizar la generación de los productos de salida, los cuales incluyen la configuración del bloque IP, incluyendo las restricciones del sistema, el HDL y los objetivos concretos para la simulación. Por último, tras la obtención de los bloques IP configurados también es posible elaborar el diseño RTL [13], [15].

También, se hace uso de Vivado High Level Synthesis (HLS). Vivado HLS es una herramienta de Xilinx que permite la transformación de una función C en un bloque IP que pueda

ser integrado en un sistema hardware mediante técnicas de síntesis de alto nivel. Es capaz de extraer la arquitectura de control y de flujo de datos a partir del código fuente, permitiendo implementar el diseño en función de las características, las restricciones y las directivas utilizadas [16]. El flujo de diseño de la herramienta Vivado HLS para la obtención del bloque IP se muestra en la Figura 6 y requiere de los siguientes pasos:

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

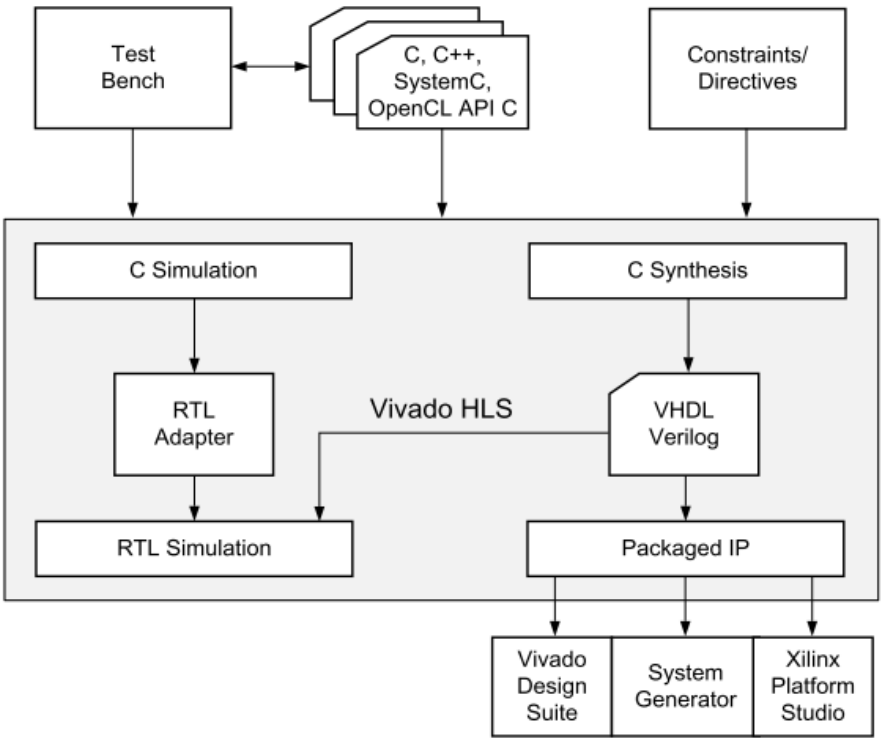

Figura 6: Flujo de diseño de Vivado HLS [16]

Respecto a la línea de trabajo de la implementación y verificación del bus de comunicaciones PCI Express sobre la FPGA, se hace uso del IDE de Vivado. Se realizan las modificaciones necesarias para la plataforma y se procede a su programación sobre esta. No obstante, se hace uso del driver de RIFFA[11] instalado en el ordenador donde se encuentra el *host* controlador del sistema, para realizar la comprobación de la comunicación entre este y la FPGA a través del bus de comunicaciones PCI Express. En el capítulo 5 se explica en detalle.

Respecto a la línea de trabajo de la implementación y verificación del clasificador, se hace uso inicialmente de Vivado HLS y posteriormente del IDE Vivado Design Suite. El flujo de diseño en este caso se centra en el diseño del bloque IP y su implementación e integración, incluyendo su configuración y verificación, haciendo uso de la herramienta SDK para su programación y depurado sobre la FPGA.

Finalmente, se procede a la unificación de ambas partes gracias al uso de Vivado Design Suite y la generación del diseño a partir de bloques IP. Para la comprobación del correcto funcionamiento del sistema se hará uso del *driver* de RIFFA, así como de la herramienta SDK para comprobar que la unificación del bus de comunicaciones PCI Express y el clasificador funciona de forma correcta.

1.2. Objetivos

El objetivo principal del proyecto es crear una plataforma FPGA que utiliza PCI Express como interfaz de datos. Esta plataforma se aplicará en un sistema de procesamiento para Big Data, aunque se tratará que el diseño pueda ser reutilizado para otros tipos de aplicaciones (Deep Packet Inspection, Complex Event Processing, …). El bus de comunicaciones PCI Express facilita las comunicaciones entre el *host* y la FPGA. La plataforma se completa con un clasificador capaz de trabajar con datos en tiempo real, permitiendo su organización en función del tipo de datos de origen para facilitar el procesamiento y distribución realizados por el resto del sistema de Big Data.

Los objetivos operativos en los que se organiza el trabajo son:

O1. Estudiar los procedimientos de diseño necesarios para utilizar los bloques que componen el bus del sistema PCI Express y su integración en la plataforma.

O2. Diseñar la arquitectura del clasificador, realizar su modelado y simulación y analizar sus prestaciones.

O3. Integrar el bus PCI Express en la plataforma FPGA hacia su implementación y validar su funcionalidad y prestaciones.

O4. Integrar el clasificador en la plataforma FPGA hacia su implementación y validar su funcionalidad y sus prestaciones.

O5. Documentar el trabajo realizado.

1.3. Peticionario

Actúa como peticionario la División de Sistemas Industriales y CAD (SICAD) del Instituto Universitario de Microelectrónica Aplicada de la ULPGC. Igualmente actúa como peticionario de

este Trabajo Fin de Máster el Instituto de Microelectrónica Aplicada (IUMA), para cumplir con los requisitos de la asignatura Trabajo Fin de Máster, dentro del Máster en Tecnologías de Telecomunicación.

1.4. Estructura del documento

El documento está estructurado como sigue. En este primer capítulo se introducen los antecedentes del proyecto, los objetivos del mismo y los aspectos formales necesarios. También, se introduce la placa de desarrollo seleccionada para este proyecto, ZC706 de Xilinx, y se mencionan las características que permiten su utilización para aplicaciones que hagan uso del bus de comunicaciones PCI Express. El segundo capítulo introduce el bus de comunicaciones PCI Express, pues es una de las claves fundamentales del proyecto. Se tratan los diferentes detalles, incluyendo sus características generales, su arquitectura y sus particularidades.

El tercer capítulo introduce RIFFA, ya que este es el marco de integración reusable escogido para la integración del bus de comunicaciones PCI Express en FPGAs. Por ello, se muestran sus diferentes características, incluyendo sus ventajas y desventajas y explicando su elección para este proyecto. También se trata su arquitectura, haciendo particular énfasis en las interfaces software y hardware, y las diferentes versiones del mismo. Finalmente, se explica la implementación realizada del mismo sobre la placa de desarrollo ZC706 utilizada en este proyecto.

El cuarto capítulo hace referencia a los clasificadores y los algoritmos de *sorting*. En él se trata la importancia de los clasificadores en las aplicaciones actuales mediante un breve estado del arte, para poder pasar a una breve explicación de los clasificadores estudiados, los cuales se basan en algoritmos de *sorting*. Posteriormente, se razona el porqué del clasificador escogido y se muestra su diseño y desarrollo.

El quinto capítulo muestra la integración de los dos capítulos previos, pues se incluye tanto el bus de comunicaciones PCI Express, haciendo uso de RIFFA, como el clasificador seleccionado, obteniendo una plataforma conjunta. Se muestra la validación y verificación ambos sistemas de forma independiente, obteniendo valores reales de su funcionamiento sobre la placa de desarrollo.

Finalmente, el sexto capítulo presenta las conclusiones del trabajo para dar finalización al mismo. Incluye un resumen de los resultados obtenidos y una serie de posibles líneas futuras de trabajo. Al final del documento se encuentran todas las referencias utilizadas en la elaboración del trabajo.

Capítulo 2. PCI Express

PCI Express es un bus de interconexión de dispositivos de entrada/salida que permite su implementación en un alto número de aplicaciones y sobre diversas plataformas. En 2002 fueron publicadas por primera vez las especificaciones del bus. PCI Express parte de los buses previos, manteniendo las ventajas y soluciones adecuadas que aportaban, como son PCI o PCI-X. Sin embargo, se añaden diferentes mejoras.

Inicialmente PCI Express se llamó *Third Generation Input and Output* o 3GIO, cuando se comenzó a desarrollar su plataforma, y se creó como necesidad de conseguir un bus capaz de soportar la creciente demanda en lo que rendimiento y velocidad se refería. PCI Express se define como un protocolo de interconexión serial punto a punto, el cual permite un elevado ancho de banda, introduciendo escalabilidad y versatilidad, ya que puede ser implementado para una gran variedad de aplicaciones (móviles, servidores, plataformas de comunicación, sistemas embedidos, etc.).

2.1. Características generales

PCI Express es un bus de comunicaciones compatible con sus buses predecesores PCI y PCI-X. Por ello, existen una serie de características que comparte con ellos, como pueden ser el modelo de uso, el modelo de comunicación *load-store* y el modelo de espacio de memoria de configuración. También soporta transacciones de la familia como son las lecturas y escrituras de memoria, de configuración y de dispositivos de entrada/salida [17].

PCI Express es un bus punto a punto serial que permite la comunicación entre dos dispositivos utilizando un protocolo de comunicación basado en paquetes. Esta es la principal diferencia respecto a los buses previos, los cuales disponían de una arquitectura paralela multipunto, por lo que múltiples dispositivos hacían uso de un único bus. Para conseguir la interconexión de múltiples dispositivos mediante PCI Express, son necesarios el uso de *switches*. Estos *switches* pueden tener desde 2 hasta n puertos para dispositivos con enlaces PCI Express, pues no existe limitación en las especificaciones.

Dentro de las ventajas de PCI Express se encuentra la posibilidad de aumentar las frecuencias de recepción y transmisión, ya que la interconexión requiere de limitada carga eléctrica debido a la conexión punto a punto. El rendimiento también se ve ligadamente afectado, al implementarse escalabilidad en los *pines* y los canales necesarios para realizar la interconexión. De esta manera se reducen los costes de diseño de placa y se reduce su complejidad [6], [18].

A su vez, se incluye la característica de gestión de errores RAS (*Reliability, Availability, Serviceability*), la cual asegura la confiabilidad, disponibilidad y utilidad del sistema, proporcionando un entorno robusto para aplicaciones de tipo servidor. También se hace uso de campos CRC en todas las transacciones de datos, para asegurar la integridad del sistema (ECRC a nivel de Transacción de datos y LCRC a nivel de enlace – Link CRC [19], [20]). Si existen errores a nivel de enlace es el propio receptor el que reconoce estos errores y manda un mensaje al transmisor para informar del mismo y este procede al reenvío del paquete. De esta manera se realizan autocorrecciones de los errores a nivel de enlace.

Para la gestión de interrupciones se hace uso del protocolo MSI, al igual que en PCI-X. MSI son simples transacciones de petición de escritura en memoria, o transacciones de petición de interrupción de mensaje, que solo pueden ser soportadas por *legacy endpoints* o por *bridges*. Cuando se requiere realizar una interrupción, el dispositivo que lo requiera debe realizar una escritura en memoria al *bridge host* con un vector de interrupción. El *bridge* es el encargado de realizar la interrupción del CPU.

Existen diferentes versiones dentro de PCI Express, a las cuales se les denomina Gen. La primera generación, o Gen1, dispone de una tasa de transferencia de 2,5 Gb/s, correspondiente a cada *lane* en una sola dirección. Mientras que Gen2 puede llegar hasta 5 Gb/s y Gen3 es capaz de obtener tasas de transferencia de 8 Gb/s. Sin embargo, ha de tenerse en cuenta que la tasa de transferencia de datos efectiva siempre será menor a esta, debido a las diferentes pérdidas por sobrecarga y por intercambios en el sistema [17].

Una de las características más importantes en PCI Express es el uso de señales diferenciales, por lo que se requieren de *drivers* y receptores diferenciales en cada uno de los dispositivos que lo implementen. Debe existir aislamiento DC entre el *driver* de un dispositivo y el receptor diferencial del dispositivo opuesto, haciendo uso de un condensador en el *driver*.

2.2. Arquitectura

PCI Express forma parte de la tercera generación de buses de entrada/salida de alto rendimiento. Destaca por hacer uso de un modelo de capas, donde se distinguen tres diferentes: la capa física, la capa de enlace de datos y la capa de transacción. Implementa, por tanto, un protocolo de paquetes, que permite a las capas comunicarse y compartir información mediante el intercambio

de paquetes. Es posible realizar una división de cada una de las capas en dos partes diferentes, la encargada de la transmisión de datos (para tráfico de salida) y la encargada de la recepción de datos (para el tráfico de entrada).

Para poder realizar el transporte de información del transmisor al receptor es necesario crear dichos paquetes en la capa de transacción, gracias a la información del núcleo del dispositivo y de la aplicación implementada. Es necesaria la existencia de *buffers* que almacenen dichos paquetes o TLPs hasta su transmisión. También se requiere el uso de la capa de enlace de datos para completar los paquetes, ya que se requiere añadir información adicional al paquete para que cada una de las capas sea capaz de gestionarlo de forma correcta. Por ello, la capa de enlace de datos incluye la información necesaria para el chequeo de errores, obteniendo así un DLLP o paquete de la capa de enlace de datos. La capa física se encarga de realizar la codificación del paquete, generando un PLP o paquete de la capa física, y procede a su envío al dispositivo correspondiente, haciendo uso de los enlaces disponibles PCI Express.

Para poder recibir la información enviada por el transmisor, el receptor debe capturar los paquetes a través de la capa física y proceder a su decodificación. De esta manera es posible realizar el envío del paquete a la capa de enlace de datos que es la encargada del chequeo de errores. Tras finalizar, la capa de transacción recibe el paquete a través de los *buffers* disponibles para ello y adapta la información para que el dispositivo pueda trabajar con ella.

Al realizarse la comunicación entre las capas, cada una de ellas extrae la información requerida y continúa con el envío del paquete al resto de capas sucesivas. En la Figura 7 se muestra un ejemplo claro de la subdivisión de las capas en cada uno de los dispositivos PCI Express y la interconexión existente entre ellos.

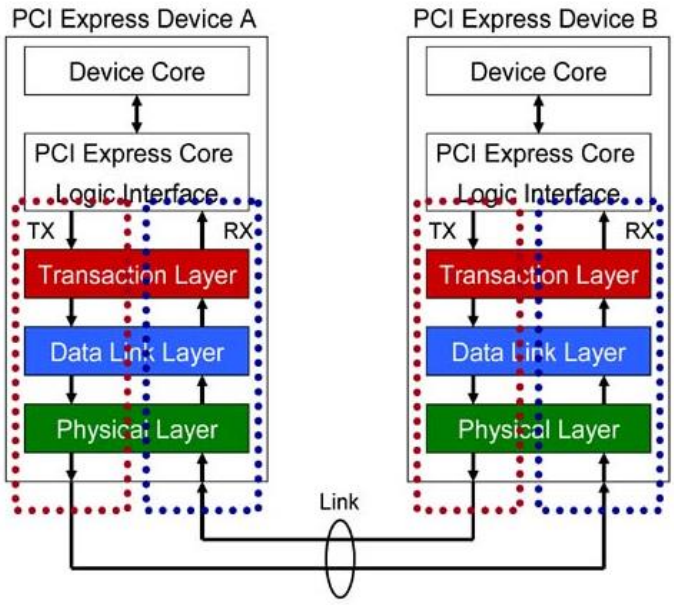

Figura 7: Capas de dispositivos para PCI Express [18]

En la Figura 8 se muestra un diagrama del funcionamiento detallado del protocolo PCI Express incluyendo cada una de las capas en las etapas de transmisión y recepción, su comunicación y todas las operaciones necesarias para su correcto funcionamiento.

La capa software, también conocida como núcleo del dispositivo, está compuesta por un núcleo lógico *root complex* o un núcleo lógico *endpoint* (controlador Ethernet, USB, etc.), pudiendo readaptarlo para generar un *endpoint* PCI Express a través de un bloque lógico de PCI o PCI-X. Esta capa es importante, pues incluye la información necesaria para generar los TLPs: dirección destino, tipo de operación a realizar, datos utilizados, etc. No obstante, debe ser capaz de obtener información de vuelta al recibir un TLP, pudiendo obtener incluso información sobre errores que hayan podido ocasionarse durante la transmisión.

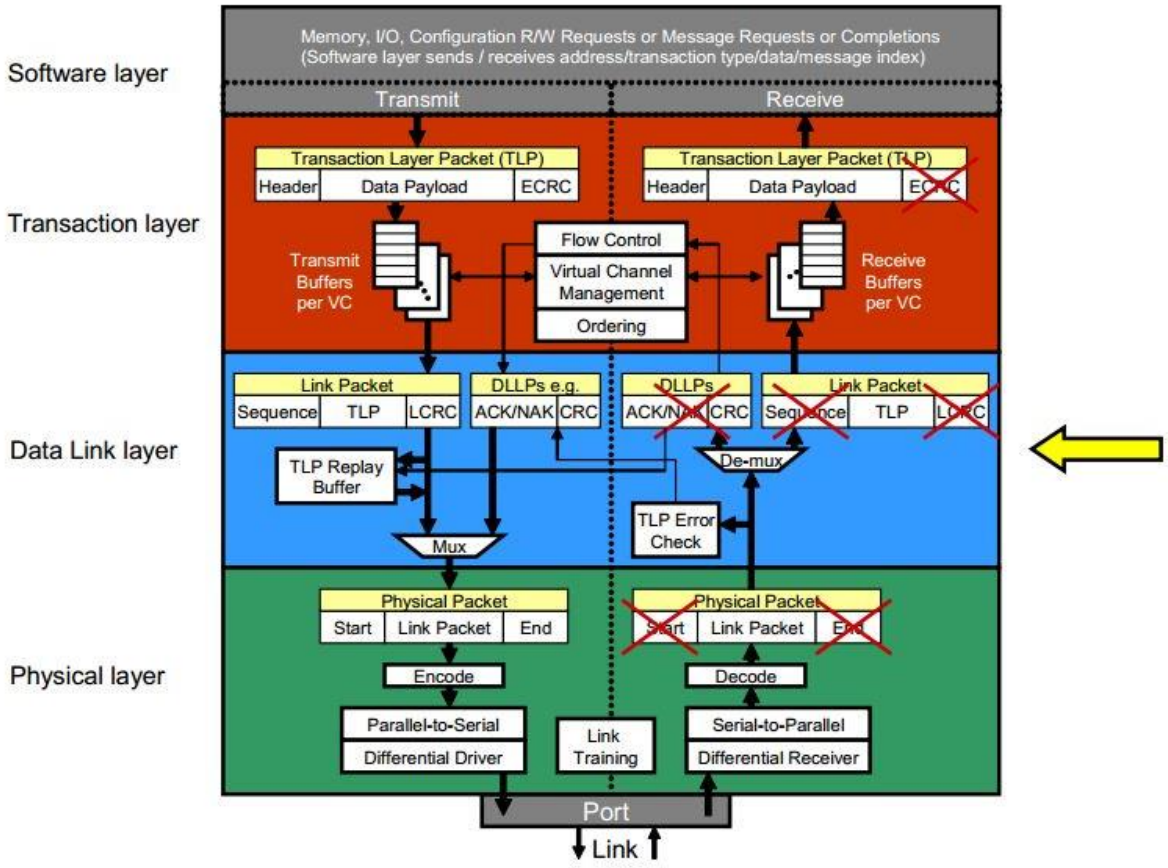

Figura 8: Diagrama de bloques detallado de las capas de dispositivo de PCI Express [18]

La capa de transacción, como vimos anteriormente, accede a la capa software para obtener la información necesaria y generar el TLP para la transferencia o recibe un TLP con la información de la transferencia a través de la capa de enlace de datos. Para la generación de la cabecera del TLP, se debe incluir la dirección destino, el tipo de TLP, el tamaño del paquete a transferir, la ID del peticionario o del ejecutor, la clase de tráfico, los bytes disponibles, los códigos de finalización y los atributos, en caso de que los tenga. En esta capa se implementa el protocolo de transacción de división para transacciones no contabilizadas y se hace uso de *buffers* de canal virtual (VC *buffers*), los cuales

incluyen un protocolo de control de flujo para asegurar que no exista desbordamiento en ninguno de los *buffers*, tanto de transmisión como de recepción. También existe la posibilidad de añadir QoS a los paquetes si se considera necesario. En esta capa se almacena la información respecto a la capacidad de enlace negociada, así como información de configuración, control y estado de la gestión de potencia. Se incluyen reglas de ordenación de TLPs y se gestiona la integridad de los mismos al realizar la generación y el posterior chequeo de ECRC [21].

La capa de enlace de datos proporciona integridad a los datos, ya que añade el chequeo de errores y realiza la generación de TLP CRC y añade al TLP una secuencia ID de 12-bit y un campo LCRC de 32-bit. Además, hace uso del protocolo ACK/NAK para añadir seguridad y fiabilidad al sistema. Incluye la generación y posterior deconstrucción de los DLLPs. El receptor es el encargado de enviar los paquetes DLLPs con ACK/NAK al transmisor para asegurar si estos han llegado o no de forma correcta. Se hace uso de DLLPs ACK para confirmar que los TLPs han sido recibidos tal y como se esperaban. En caso de que exista algún tipo de error en la recepción de los TLPs, los DLLPs NAK son los encargados de gestionar el aviso al transmisor. También incluye la inicialización y la gestión de potencia, la gestión de integridad de los DLLPs, la generación y el posterior chequeo de LCRC y CRC, y el intento de reenvío de paquetes de mensajes en caso de error [22].

Finalmente, la capa física es la encargada de recibir el paquete TLP con las modificaciones realizadas en la capa de enlace de datos, al igual que el DLLP. También se encarga de añadir caracteres de comienzo y finalización a los paquetes antes de enviarlos, y realiza la lógica de codificación necesaria para su transmisión. De esta manera se encarga de la generación de los paquetes físicos. El receptor de la capa física hace uso de un convertidor serie-a-paralelo para generar un *stream* paralelo de 10b e incluye un *buffer* de tamaño adaptable para permitir la adaptación de la variación de frecuencia de reloj entre el transmisor y el receptor. La capa física viene definida por dos partes: la capa física lógica y la capa física eléctrica. La primera incluye toda la lógica digital necesaria para el procesado y la transmisión de datos, mientras que la segunda representa la interfaz analógica que permite el conexionado con el enlace PCI Express, incluyendo *drivers* diferenciales y receptores para los *lanes*, teniendo en cuenta la configuración de estos y el previo establecimiento del ancho del enlace [23]. En la Figura 9 se muestra un ejemplo de la estructura de un TLP completo en PCI Express, mostrando las diferentes adiciones por cada una de las capas.

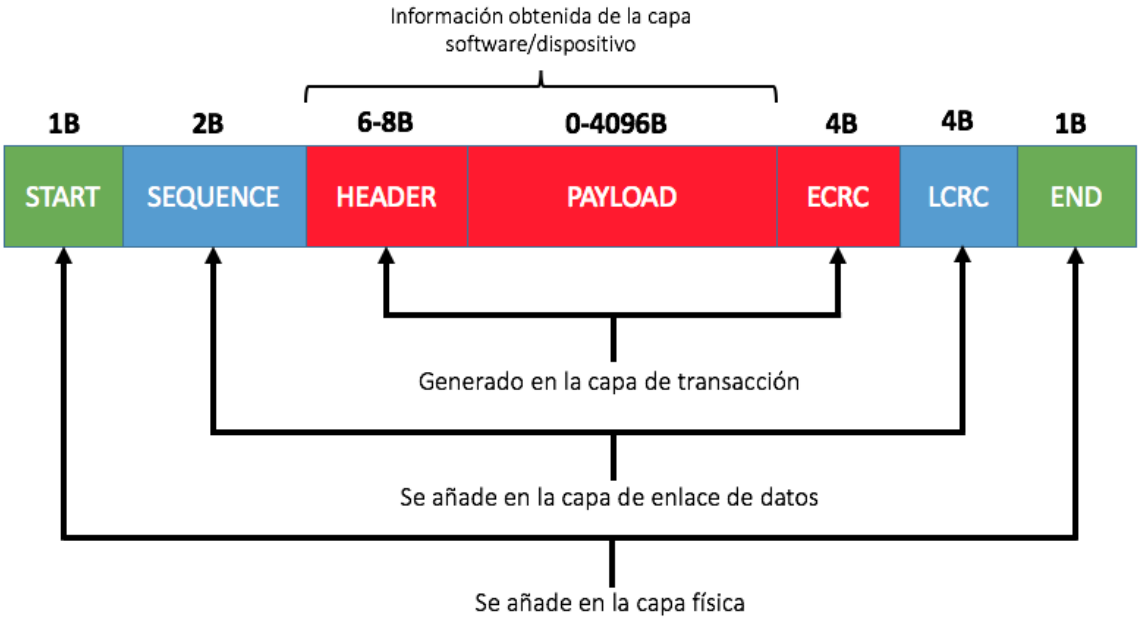

Figura 9: TLP completo de PCI Express [18]

La capa física se encarga de la configuración, control y estado del enlace haciendo uso de transacciones de estado de enlace, tanto en la etapa de inicio y entrenamiento, como en la implementación de la máquina de estado de *status* (LTSSM). También incluye diseño para test (DFT) y características de cumplimiento/conformidad. Respecto a la capa física eléctrica cabe destacar una serie de detalles. El transmisor se encuentra acoplado en AC al receptor, siendo la capacitancia requerida de entre 75 nF y 200 nF. En lo que se refiere al voltaje DC establecido en el transmisor, se obtiene en las etapas de inicialización y puesta en marcha del sistema, siendo la impedancia de DC en torno a 50 Ω. También hace uso de una impedancia diferencial que suele rondar los 100 Ω.

2.2.1. Enlace PCI Express

Cuando se hace mención de un enlace, se refiere a una interconexión PCI Express de dos dispositivos, pudiendo tener x1, x2, x4, x8, x12, x16 o x32 pares de señales (*lanes*) para cada dirección. Debido a que la transmisión y recepción de datos se realiza de forma simultánea el enlace es simple-dual, al disponer de *lanes* diferenciados para cada sentido de la comunicación. Cada enlace (x1) está compuesto por un *lane*, que corresponde con un par de señales diferenciales en cada dirección, lo que supone un total de 4 señales por enlace. No obstante, se debe tener en cuenta que al aumentar la cantidad de *lanes* en la conexión PCI Express, aumentará el ancho de banda final. En la Tabla 1 se muestra cuál sería el ancho de banda agregado en función del número de *lanes* del enlace PCI Express que se utilicen, habiendo tenido en cuenta para su cálculo la restricción de la codificación de 8-a-10 bit. Cabe destacar que no existe señal de reloj en el enlace PCI Express, pues se hace uso de un PLL para recuperar el reloj de las transiciones en función de la decodificación utilizada.

Tabla 1: Ancho de banda de PCI Express para múltiples lanes de enlace [24][6]

***Lanes* del enlace PCI Express**	**x1**	**x2**	**x4**	**x8**	**x12**	**x16**	**x32**
Número de líneas por cada dirección	2	4	8	16	24	32	64
Ancho de banda extra (Gb/s)	0,5	1	2	4	6	8	16
***Raw bit stream* por segundo (Gb/s)**	2,5	5,0	10,0	20,0	30,0	40,0	80,0
***Bandwidth bytes* por segundo (MB/s)**	250	500	1000	2000	3000	4000	8000

Para realizar la inicialización hardware no se requiere de ningún sistema operativo o firmware a nivel de enlace, pues se ajusta en cada uno de los puntos finales de la conexión la frecuencia de operación y el ancho de enlace. En la Figura 10 se puede observar un ejemplo claro de el enlace PCI Express y su composición.

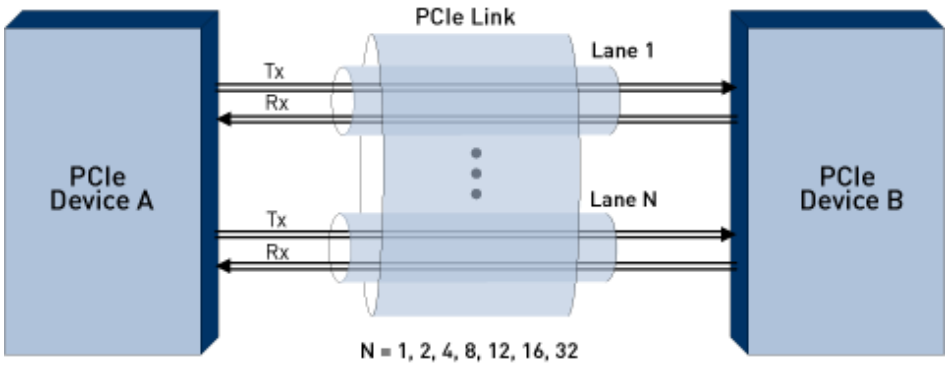

Figura 10: Enlace PCI Express

2.2.2. Topología PCI Express

En la Figura 11 se observa una topología PCI Express básica con todos los elementos necesarios para su correcto funcionamiento. A continuación explicaremos cada uno de ellos y sus características principales [18], [24]:

- ***Root complex***. Es el dispositivo que permite la conexión entre la CPU y la memoria, así como con el resto de dispositivos PCI Express. También incluye los controladores necesarios para la inicialización, la gestión de potencia, la detección de errores, las interrupciones y *hot-plug*, así como la lógica de reportes. Una estructura PCI Express solo puede estar compuesta por un *root complex*. Aunque en

la Figura 11 se muestren tres puertos conectados a *root complex*, puede soportar uno solo o múltiples más. Estos puertos permiten la conexión con los *endpoints* PCI Express, *switches* o *bridges*. Cada *switch* solo dispone de un puerto superior, pero puede tener tantos puertos inferiores como se requiere, hasta 256. Por ello, es el encargado de generar las transacciones provenientes de la CPU hacia el resto de dispositivos (transacciones tanto de memoria como de entrada/salida). Además, es posible realizar transacciones contabilizadas, pero no es capaz de recibirlas. Para determinar el ID de *root complex* se inicializa con valor 0 el número del bus, el número de dispositivo y el número de función. No obstante, el *root complex* también está compuesto por diferentes elementos primordiales para su correcto funcionamiento como son [25]:

  - *Bridge* virtual *host*/PCI. Es la interfaz central para PCI, la memoria y los dispositivos PCI Express de la plataforma.
  - Bloque de registro *root complex* (RCRB). Conjunto de registros de configuración necesarios para el *bridge*.
  - Segmento de bus virtual PCI. Permite la interconexión entre el *bridge host*/PCI y múltiples *bridges* virtuales PCI/PCI, teniendo en cuenta las señales IDSEL.
  - *Bridges* virtuales PCI/PCI. Únicamente para los puertos inferiores, permitiendo distribuir las transacciones de la CPU a los dispositivos PCI Express y desde estos dispositivos recibir las transacciones a memoria.

- *Switch.* Permite la interconexión de un enlace conectado a su puerto superior con múltiples enlaces conectados a los puertos inferiores. Es el conjunto de dos o más *bridges* PCI a PCI con puertos *switch* asociados. Cada *bridge* debe incluir registros con configuración de cabecera tipo 1, siendo estos inicializados en el arranque. Realmente, se trata de *bridges* virtuales que se conectan a través de un bus. Hacen uso de tres mecanismos de ruteado diferentes: direccionamiento, ID o implícito. También incluyen dos mecanismos de arbitraje para gestionar la prioridad de los paquetes y su ruteado: arbitraje de puerto y arbitraje VC.
- *Endpoint.* Son los dispositivos periféricos que incluyen el bus de comunicaciones PCI Express, pudiendo realizar transacciones o recibirlas. Incluyen cabeceras PCI Express de tipo 0 y no pueden generar transacciones de configuración, pero si responder a ellas. Existen dos tipos diferentes:

- PCI Express *endpoint.* Incluye la generación MSI y soporta direccionamiento de memoria de 64 bit. Sin embargo, no permite transacciones de entrada/salida o transacciones contabilizadas.
- *Legacy endpoint.* Permite transacciones de entrada/salida, transacciones contabilizadas sólo como receptor y generación de interrupciones de tipo *legacy* (teniendo que incluir en este caso generación MSI). No obstante, no necesita soportar direccionamiento de memoria de 64 bit.

- PCI Express *bridge* a PCI/PCI-X. Permite la conexión con una jerarquía de tipo PCI o PCI-X, de manera que se puedan incluir ambos buses de comunicación en una única arquitectura. Por tanto, realiza la conversión del protocolo PCI Express al protocolo PCI o PCI-X, según corresponda, permitiendo la adición de elementos de estos buses a la plataforma PCI Express existente.
- Puerto. Es la interfaz que conecta el enlace al dispositivo PCI Express. Por ello requiere de transmisores y receptores diferenciales.

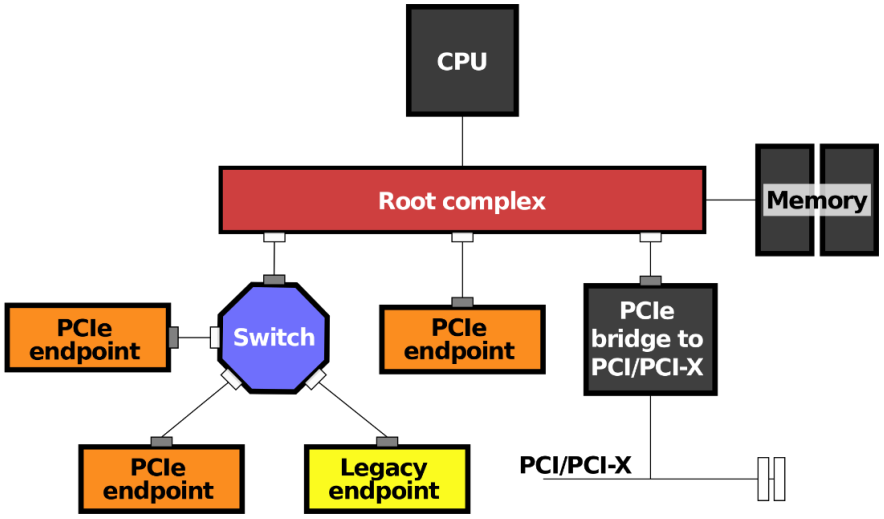

Figura 11: Topología PCI Express

No obstante, en un sistema pueden coexistir múltiples estructuras PCI Express interconectadas entre sí. Para ello se requiere de un enlace *peer-to-peer* avanzado entre dos puertos inferiores de dos *switches*, siendo cada *switch* parte de una estructura PCI Express diferente. Las únicas transacciones que pueden hacer uso de este enlace son las transacciones de petición de conmutación avanzada de mensaje (*message advanced switching requester transacctions*).

Las transacciones *peer-to-peer* son transacciones que se definen únicamente entre dos *endpoints*, dos *switches* o una combinación de estos. Estas transacciones pueden ser de tipo entrada/salida, mensaje o de memoria, pero nunca pueden ser de configuración. Al realizarse la ejecución de una transacción esta se mueve en sentido ascendente y sentido descendente a través de

los puertos. Generalmente el *root complex* no soporta este tipo de transacciones, pero si dispone de un *switch* virtual sí puede procesar este tipo de transacciones. En la Figura 12 se muestra un ejemplo de cómo se realizaría la interconexión de diferentes estructuras PCI Express a través de un enlace *peer-to-peer* avanzado.

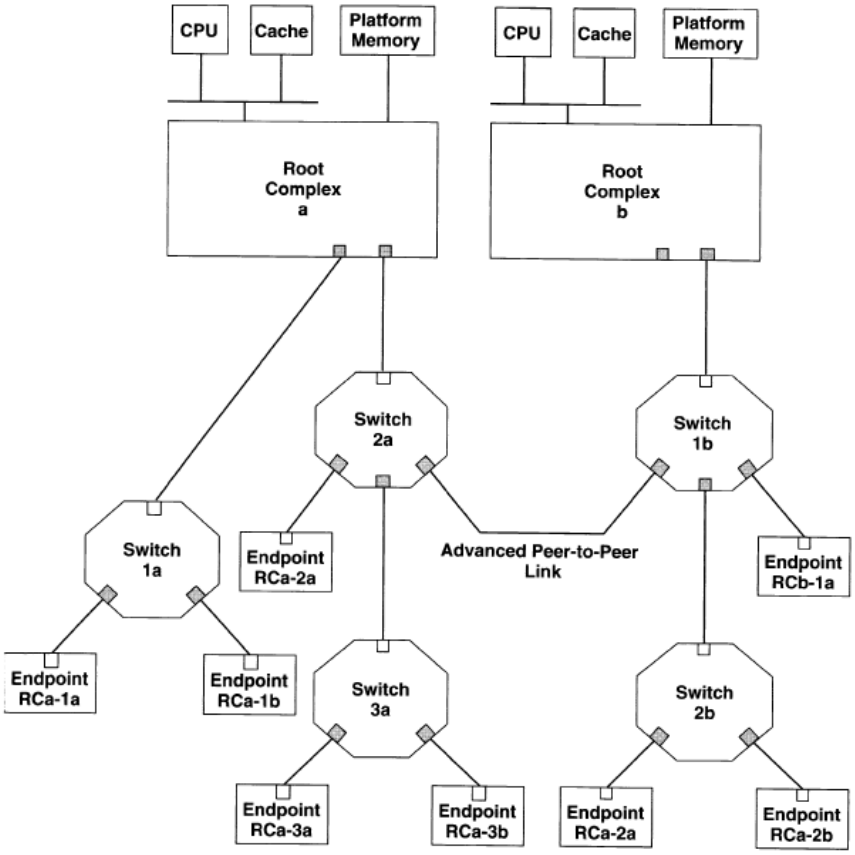

Figura 12: Enlace peer-to-peer avanzado [25]

2.2.3. Protocolo de transacciones

Es necesario el uso de TLPs o transacciones de datos para la comunicación de los dispositivos a través del bus de comunicaciones PCI Express. Gracias al uso de un protocolo basado en paquetes se añade integridad de paquetes al bus [26]. Es importante definir la estructura de un TLP, entendiendo cada uno de sus componentes:

1. **Cabecera o *header***. Está compuesta por 3 o 4 *double words*, pudiendo variar su formato, pero siempre incluyendo el tipo de transacción y los atributos asociados a esta. Los diferentes campos que puede incluir son:

    - **Tipo**. Identifica el tipo de transacción: memoria, entrada/salida, configuración, mensaje o *completer*.

    - **Clase de tráfico (TC).** Incluye el valor de prioridad de la transacción. El número de TC asignado a un TLP determina el canal virtual asociado, siendo el arbitraje

de los canales virtuales quien determina la prioridad de transmisión. El protocolo de ordenación es el encargado de aplicar el número TC a los diferentes enlaces.

- **Atributos**. Define la clasificación de transacción a usar y si se requiere de *snooping*.
- **Dirección.** Determina la dirección a la que va destinado el TLP, sea esta de memoria, de entrada/salida o de configuración.
- **Longitud**. Indica el tamaño de *double words* que se deben escribir o leer.
- **ID del peticionario**. Incluye el número de bus, el número de dispositivo y el número de función que permiten identificar la fuente de la transacción.
- **Error**. Permite asegurar la integridad del paquete y observar si ha habido algún tipo de cambio o ataque sobre el paquete.

2. **Datos.** Solamente son necesarios en las transacciones que tiene un *payload* de datos, como son las escrituras, pudiendo variar su tamaño en función de la operación.
3. ***Digest fields***. Son opcionales, permitiendo el soporte de la generación de ECRC y su correspondiente chequeo.

En la Figura 13 se muestran los tres tipos básicos de transacciones: escritura, solicitud de lectura y confirmación de la lectura.

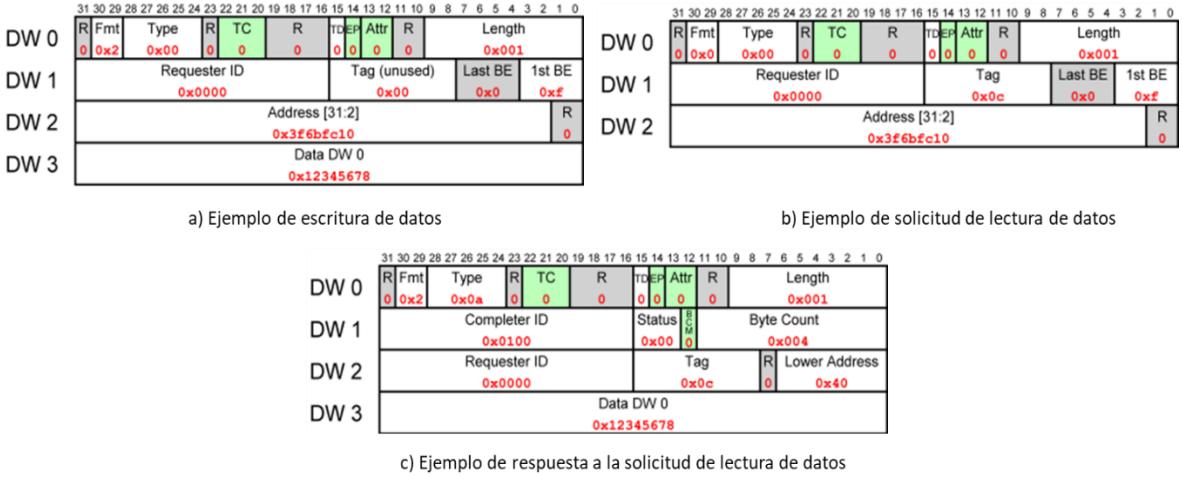

Figura 13. Ejemplos de paquetes de datos en PCI Express [27]

Las transacciones permitidas se dividen en cuatro categorías: memoria (lectura/escritura), entrada/salida (lectura/salida), configuración (lectura/escritura) y mensajes [18]. Los mensajes son *bursts* de información de control o de datos enviados de *buffer* a *buffer*. Es posible realizar la implementación de los mensajes en tres niveles diferentes:

1. Mensajes en forma de transacciones de bus de mensajes permiten ser usados como interrupciones, permitiendo la reducción del ancho de banda.
2. Mensajes para enviar tipos de información y datos que no se encontraban en las versiones previas a PCI Express, como pueden ser informaciones de errores.
3. Mensajes que hacen uso de los enlaces de pares para transferir información y datos entre múltiples PCI Express *fabrics*. Este tipo de mensajes permiten la encapsulación de otros paquetes, aunque la tecnología no esté relacionada de forma directa con PCI Express o sus predecesores.

Respecto al modelo de transacciones utilizado cabe destacar la diferencia existente entre transacciones contabilizadas y no contabilizadas. Las transacciones contabilizadas requieren de un paquete de escritura que es enviado de manera unidireccional desde el solicitante al ejecutor, sin que exista ningún paquete de vuelta para confirmar la transacción. Las transacciones contabilizadas son la escritura en memoria y las transacciones de mensajes, el resto de transacciones son no contabilizadas. Las transacciones no contabilizadas hacen uso de transacciones partidas, pues tras enviar desde el solicitante el paquete de lectura o escritura, se requiere de una respuesta por parte del ejecutor al solicitante. En la

Tabla 2 se muestran los diferentes tipos de paquetes TLP que pueden utilizarse en el bus PCI Express.

Gen1 y Gen2, hacen uso de un esquema de codificación 8B/10B en la línea de transmisión para poder mantener el balance de DC. Se realiza el envío de 10 bit, por cada byte que se desea enviar y es necesario realizar tanto una codificación de 10 bit con disparidad positiva como una codificación de 10 bit de disparidad negativa. En cada transacción es necesario escoger si se empleará la codificación positiva o negativa del byte. Esto nos lleva a un ancho de banda teórico, el cual incluye un 20% de pérdidas de *throughput*, ya que por cada byte que se quiere enviar es necesario añadir 2 bit extra para realizar la codificación. No obstante, gracias a esta codificación es posible que el receptor pueda realizar la recuperación del reloj mediante los datos obtenidos de la transacción [17].

$$Ancho\ de\ banda\ teórico\ (bytes) = \frac{tasa\ de\ transferencia\ \cdot\ 2\ direcciones\ \cdot\ ancho\ del\ lane}{10\ bits/byte}$$

Tabla 2: Tipos de paquetes TLP de PCI Express [18]

Tipos de paquetes TLP	Código	Tipo de transacción	
Petición de lectura en memoria	MRd	Transacción de lectura no contabilizada	
Petición de lectura en memoria (acceso bloqueado)	MRdLk	Transacción de lectura no contabilizada para peticiones bloqueadas	
Petición de escritura en memoria	MWr	Transacción de escritura contabilizada	
Escritura entrada/salida	IORd	Transacción de lectura no contabilizada	
Lectura entrada/salida	IOWr	Transacción de escritura no contabilizada	
Lectura de configuración (tipo 0 y tipo 1)	CfgRd0 CfdRd1	Transacción de lectura no contabilizada	
Escritura de configuración (tipo 0 y tipo 1)	CfgWr0 CfgWr1	Transacción de escritura no contabilizada	
Petición de mensaje sin datos	Msg	Transacción contabilizada	
Petición de mensaje con datos	MsgD	Transacción contabilizada	
Finalización sin datos	Cpl	Respuesta a MRd, IORd, CfgRd0 o CfgRd1 con finalización con error	Respuesta a IOWr, CfgWr0 o CfgWr1 con finalización normal o con error
Finalización con datos	CplD	Respuesta a MRd, IORd, CfgRd0 o CfgRd1 con finalización normal/correcta	
Finalización sin datos (para peticiones de escritura en memoria bloqueadas)	CplLk	Respuesta a MRdLk con finalización normal/correcta	
Finalización con datos (para peticiones de lectura de memoria bloqueadas)	CplDLk	Respuesta a MRdLk con finalización con error	

2.3. Flujo de control

El flujo de control implementado es un protocolo que permite asegurar que existe espacio suficiente en el *buffer* receptor, para que el transmisor pueda realizar el envío de información. Para ello, es necesario que el receptor envíe de forma periódica al transmisor la información sobre dicho *buffer*. Gracias a ello, se evitan posibles errores y no se requiere de soporte para reenvío de paquetes, a excepción de alguna condición de error o interrupción. El flujo de control permite que el software asigne prioridades a las transacciones. El balance del ancho de banda necesita de los diferentes

enlaces y el ancho de banda requerido para las diferentes transacciones en cada uno de los *buffers* y *switches* [24]. Por ello, el flujo de control proporciona un mecanismo para cada transacción al asignarle información de prioridad por la fuente, haciendo uso de los *switches* para comprobar su prioridad frente a transacciones de otras fuentes. Para el correcto funcionamiento del protocolo del flujo de control son necesarios los siguientes mecanismos:

1. **Calidad de servicio o QoS.** La calidad de servicio se centra en buscar un equilibrio entre la latencia, el ancho de banda y el tiempo de ejecución, pudiendo incluir también otros elementos relacionados con el rendimiento, como son la tasa de errores, la tasa de transmisión, etc. Es posible gestionar la prioridad de las transacciones y definir una QoS diferente para cada transacción, en función de la aplicación que la requiera. Cabe destacar el uso de transferencias isócronas, que al partir de una latencia y un ancho de banda delimitado permiten delimitar la QoS de la transacción.

2. **Clase de tráfico.** Ayuda a determinar la prioridad de cada una de las transacciones. El número TC se añade en la cabecera del paquete TLP, por lo que es importante que los puertos dispongan de *buffers* que permitan realizar el arbitraje de los paquetes. El número TC posteriormente se convierte en un número VC para determinar en los puertos cual es la prioridad del mismo.

3. **Canal virtual.** El canal virtual o VC es utilizado por los *switches* para determinar la prioridad de transmisión de los puertos. Se hace uso de arbitraje de puertos a través de los números VC para determinar la prioridad de transmisión y, por tanto, la asignación del ancho de banda de enlace. El arbitraje VC permite definir la prioridad de transmisión de un puerto de salida específico para diferentes números VC. Tras definir la prioridad de transmisión en el *switch*, se hace uso de los *buffers* para la reordenación de los TLPs y su posterior envío.

4. **Control de flujo de enlace.** Antes de proceder al envío de un TLP es necesario comprobar si el puerto receptor es capaz de recibir el paquete, es decir, si dispone de espacio en el *buffer* para aceptar el paquete. Gracias a ello es posible evitar pérdidas de ancho de banda al evitar el reenvío de paquetes.

2.4. Modelo Software

El espacio de direcciones es el mismo que se utiliza para el bus de comunicaciones PCI y PCI-X, como se comentó previamente. Se realiza una división entre la memoria, las entradas/salidas y la configuración, teniendo cada uno un espacio de direcciones propio. El tamaño del espacio de direcciones de memoria y de entrada/salida es igual, siendo este de 256 Bytes. Sin embargo, el espacio de memoria de configuración es de 4 KBytes, manteniéndose igual los primeros 256 Bytes

que en PCI y PCI-X. Por ello, es necesario incluir dos mecanismos para el modelo de configuración. El primero no requiere de cambios y es compatible con los sistemas operativos previos sin necesidad de variar la enumeración del bus o la configuración del software. El segundo hace uso del espacio de direcciones adicional, teniendo que actualizar el sistema operativo utilizado y los *drivers* de los dispositivos.

2.5. ZC706

ZC706 dispone de un bloque integrado para PCI Express de la serie 7 de FPGAs de Xilinx. Este bloque permite implementar PCI Express sobre la plataforma, permitiendo su escalabilidad, un elevado ancho de banda, tal y como corresponde a PCI Express, y un bloque eficiente y fiable para realizar interconexión serial. Todas las características proporcionadas por este bloque están basadas en la especificación base de PCI Express, cumpliendo en todo momento con esta. Dispone de diferentes configuraciones de canales, teniendo como consecuencia 1, 2, 4 u 8 *lanes*. Permite implementar tanto PCI Express de primera generación (Gen1) como de segunda (Gen2) con configuraciones *endpoint* o *root port*. También soporta interfaces AMBA AXI-4 Stream para la interfaz de usuario [9]. En la Tabla 3 se puede ver un resumen de las diferentes soluciones PCI Express que pueden implementarse gracias a este bloque, pudiendo ser la tasa de transferencia de 2,5 Gb/s (Gen1) o 5 Gb/s (Gen2).

Tabla 3: Resumen de las soluciones PCI Express para el bloque integrado de Xilinx [9]

Número de *lanes*	Ancho de la interfaz de usuario	Anchos de *lane* soportados
1 *lane*	64	x1
2 *lanes*	64	x1 o x2
4 *lanes*	64 o 128	x1, x2 o x4
8 *lanes*	64 o 128	x1, x2, x4 o x8

Los principales módulos e interfaces del bloque PCI Express de Xilinx se encuentran representado en el esquema de la Figura 14. Las interfaces que se incluyen dentro de este bloque son:

- Interfaz del sistema (SYS). Está compuesta por dos señales de entrada: la señal de *reset* del sistema y la señal de reloj del sistema. La señal de *reset* es asíncrona y la señal de reloj es la señal de referencia de reloj, que permite escoger entre frecuencias de 100 MHz, 125 MHz y 250 MHz.
- Interfaz PCI Express (PCI_EXP). Dispone de múltiples *lanes*, conformado cada uno por dos pares diferenciales; uno de transmisión y otro de recepción.

- Interfaz de configuración (CFG). Permite obtener información sobre el estado del *endpoint* para el espacio de configuración.
- Interfaz de transacciones (AXI4-Stream). Implementa un mecanismo que permite la generación y utilización de TLPs. Esta interfaz está compuesta a su vez por tres interfaces: la interfaz común, la interfaz de transmisión y la interfaz de recepción.
- Interfaz de estado y de control de la capa física (PL). Permite obtener el estado actual del enlace y efectuar control sobre este.

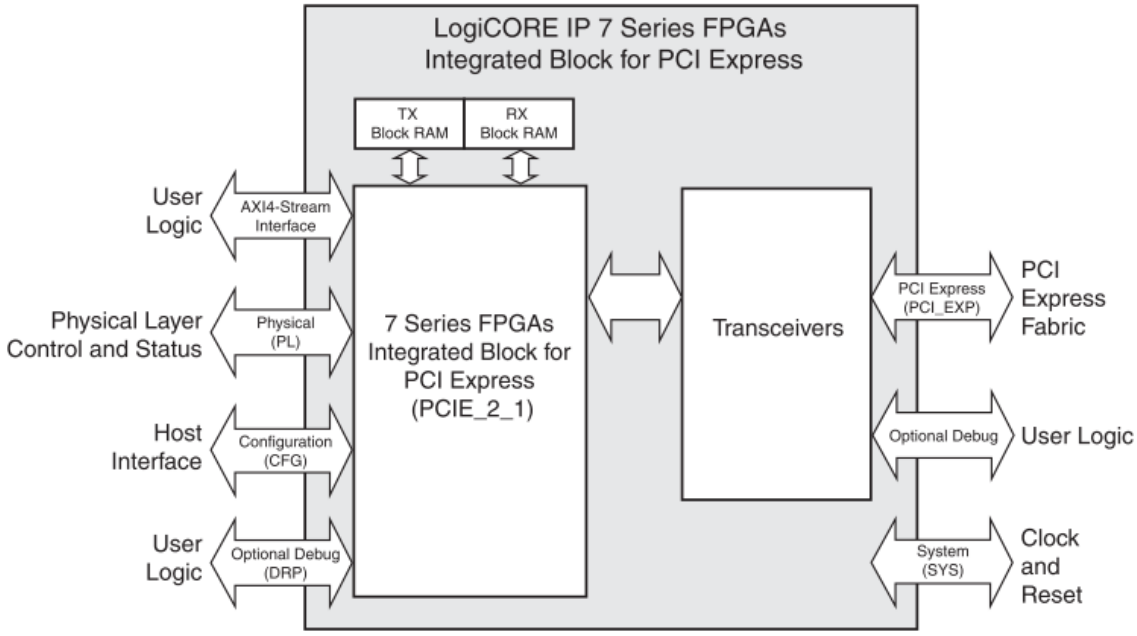

Figura 14: Principales módulos e interfaces del Bloque PCI Express de Xilinx [9]

2.6. Conclusiones

PCI Express es un protocolo de comunicaciones con el que se consiguen conexiones punto a punto de alta velocidad y alto rendimiento, haciendo uso de un enlace simple dual y de señalización diferencial. Además, gracias al uso de un bus serial, con conexiones dedicadas para cada dispositivo, es posible obtener baja latencia y baja sobrecarga en la conexión. Esto implica que los dispositivos PCI Express no tiene que arbitrar por el enlace. Todo ello, se debe al uso de un protocolo de transmisión basado en paquetes y la división por capas.

Gracias al uso de menos señales que los protocolos de comunicación anteriores y menores variaciones en las señales, PCI Express es un **bus de bajo consumo**, pudiendo gestionar de forma individual cada uno de los dispositivos integrados en el bus para gestionar la potencia utilizada. Por ello, ofrece un sistema con manejo de potencia con la posibilidad de readaptar el sistema en tiempo de ejecución al activar o desactivar dispositivos que puedan acceder al propio bus.

También se incluye la gestión de errores y se permite incluir múltiples estructuras de datos, pudiendo llegar a realizar transmisiones isócronas. PCI Express permite la compatibilidad software con PCI y PCI-X, teniendo especial importancia la configuración y el control de potencia.

Capítulo 3. RIFFA

RIFFA (*Reusable Integration Framework for FPGA Accelerators)* se define como un marco de integración reusable dirigido a aceleradores FPGA, lo que permite integrar de forma sencilla y eficiente el bus de comunicaciones PCI Express entre un *host* CPU y una FPGA apta para ello [28]. El objetivo principal de RIFFA es facilitar la implementación de aplicaciones en FPGAs y, de esta manera, expandir su uso. Por ello, facilita la comunicación y sincronización entre la FPGA deseada y el *host* CPU, mediante el uso de PCI Express. RIFFA proporciona comunicación y sincronización para el software y hardware, haciendo uso de una interfaz estándar, aportando flexibilidad y reusabilidad a los diseños en los que se implementa [29].

3.1. Características generales

RIFFA no requiere de hardware especializado o de IPs con licencia, ya que únicamente son necesarias una estación de trabajo que tenga habilitado el bus de comunicaciones PCI Express y una FPGA con un periférico PCI Express. No obstante, RIFFA ofrece soporte para ser implementado en sistemas operativos de Windows y Linux. Por ello, las diferentes APIs software necesarias para su implementación han sido desarrolladas en C/C++, Python, MATLAB y Java. Principalmente, se centran en el desarrollo de dos funciones encargadas de realizar el envío y la recepción de datos de forma correcta y acorde a las especificaciones del bus PCI Express [28].

RIFFA está soportado en una amplia gama de dispositivos FPGA, tales como son las FPGAs de Xilinx (ZC706, VC706, VC707, VC709) y de Altera (DE4, DE5-Net, Stratix IV, Stratix V, Cyclone IV, Cyclone V, Arria II, Arria V).

Inicialmente, se planteó hacer uso de RIFFA para realizar la interconexión de un único *host* CPU y una FPGA. No obstante, es posible hacer uso de más de una FPGA en un mismo sistema, pudiendo llegar a implementar hasta un máximo de 5 FPGAs. Esto aporta versatilidad y adaptabilidad al sistema que se desee implementar, así como incluye la posibilidad de expansión para futuras

situaciones. RIFFA se basa en el concepto de comunicación de canales entre hilos software en la CPU y núcleos de usuario en la FPGA [30].

Cabe destacar la implementación hardware de RIFFA, pues contiene una interfaz con señalización de transmisión y recepción independientes, incluyendo *handshaking* y *first word fall* mediante una interfaz FIFO que permite la lectura y escritura de datos. Diferentes análisis realizados muestran que es posible conseguir hasta un 97% del ancho de banda permitido por el propio bus PCI Express [30].

3.2. Arquitectura

La arquitectura utilizada para la implementación de RIFFA 2 varía notablemente frente a la arquitectura inicial diseñada para la versión inicial o RIFFA 1. Esto se debe principalmente a las mejoras que se implementaron en la versión de RIFFA 2, basándose en incluir una arquitectura de alto nivel. En la Figura 15 se puede observar dicha arquitectura.

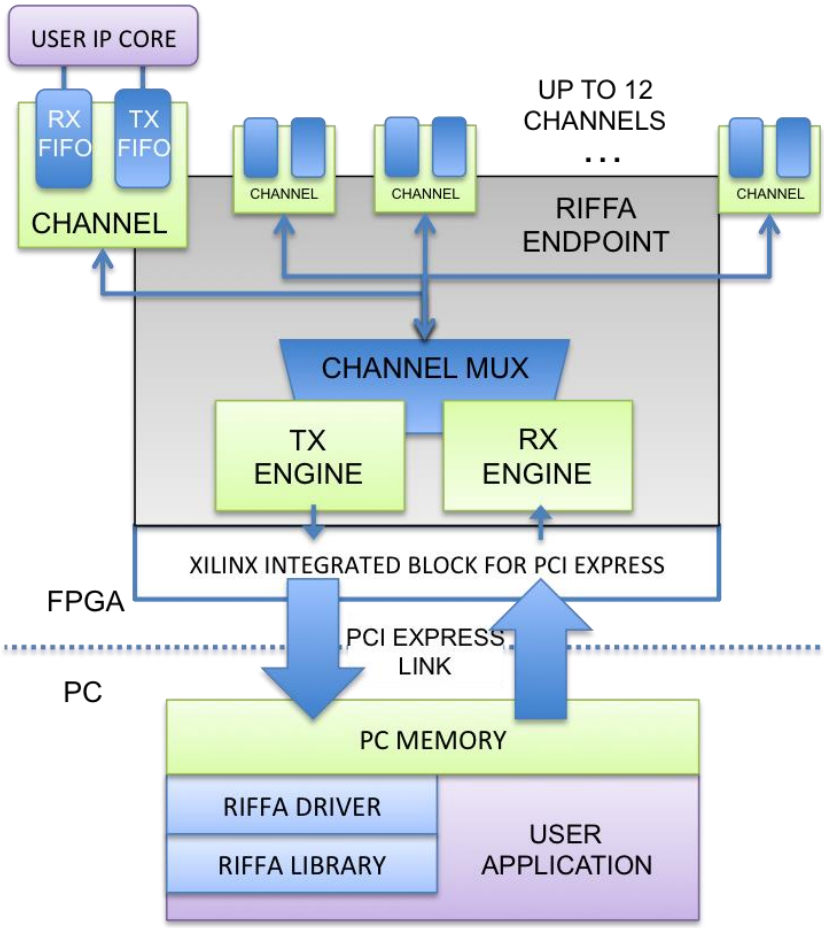

Figura 15: Esquema de la arquitectura de RIFFA 2 [28]

Se realiza la conexión de la estación de trabajo y la FPGA mediante el bus PCI Express. RIFFA consta de núcleos IP realizados en Verilog y VHDL (en la FPGA), así como de librerías en

C y un *driver* realizado en Linux (en la estación de trabajo). Se busca un *throughput* elevado con una baja latencia.

3.2.1. Interfaz software

El *driver* implementado en el sistema operativo del *host* con soporte tanto para Linux como para Windows) se encarga de probar la FPGA durante su arranque y realizar la asignación de las direcciones en la estación de trabajo para el *endpoint* PCI Express de la FPGA, teniendo en cuenta el espacio de direcciones de PCI Express válido. Durante este proceso se reserva el espacio de direcciones del *kernel* para la comunicación con la FPGA. Cabe destacar, que el espacio de direccionamiento seleccionado, puede convertirse en una limitación, si no se incluye uno lo suficientemente amplio para soportar los datos de respuesta contiguos de cualquier núcleo IP. Para la versión de RIFFA 1 se fijó un espacio de direccionamiento de 8 MB, valor aceptable para los casos prácticos llevados a cabo. En este punto, ya se encuentra disponible la opción de intercambio de información entre el *host* CPU y la FPGA.

Para acceder fuera del *kernel*, es necesario crear un archivo de dispositivo virtual en el sistema de ficheros denominado “dev”. Dicha tarea es realizada por el *driver*. Este archivo puede ser accedido y modificado por las aplicaciones dentro del espacio de usuario, y para ello se hace uso de las transacciones de lectura y escritura del bus PCI Express. Una vez llegan estas transacciones al *endpoint* PCI Express de la FPGA, se realiza su conversión mediante traducción de direcciones a peticiones del bus. Sin embargo, cabe destacar que se debe tener cuidado con esta funcionalidad, pues puede convertirse en una brecha de seguridad al exponer la FPGA al espacio del usuario, en caso de que este se viese comprometido [29].

Una parte importante de la interfaz software, es la inclusión de señalización de eventos mediante un canal de interrupciones entre la estación de trabajo y el *endpoint* PCI Express de la FPGA. Para ello, el *driver* implementa interrupciones del dispositivo PCI Express gracias al uso de hilos. No obstante, existe la posibilidad de generar hasta 16 canales de interrupciones simultáneos, siendo posible hacer uso de un esquema multi-hilo, que permite que no exista interferencia entre canales si varios se encuentran a la espera.

Por tanto, es el *driver* el encargado de establecer y controlar su normal funcionamiento, generando una serie de archivos virtuales numerados con las interrupciones sucedidas en el sistema de ficheros denominado “proc”. De esta manera, cuando un archivo quiera acceder a estos ficheros, el *driver* devolverá el número de interrupciones del IP específico consultado, o en caso de no haber sucedido ninguna interrupción, el hilo que realiza la consulta se queda suspendido hasta que suceda alguna interrupción, en cuyo caso responde a la aplicación que haya realizado la consulta. Sin embargo, existe una limitación en el diseño, pues no se aceptan nuevos vectores de interrupción,

hasta que el vector de interrupción en curso haya sido aceptado y gestionado por el *driver*. Para poder gestionarlo de mejor manera, se incluye una lista para peticiones de interrupción que se encuentren pendientes en la FPGA. De esta manera, una sola interrupción PCI Express recibida puede permitir generar múltiples canales de interrupción lógicos [29], [30].

Se añade soporte de transferencia DMA y *doorbells*, que son interrupciones en los *switches* para comunicaciones no transparentes, mediante transacciones de escritura PCI Express. De esta manera es posible realizar la petición de una transferencia DMA o una interrupción a un núcleo IP, mediante la escritura en una señal de un núcleo IP controlador por parte del propio software. Los *doorbells* se manifiestan como pulsos de línea a los núcleos IP receptores.g

Para facilitar el uso de este marco de trabajo, se realiza una API de alto nivel para aplicación de usuario, haciendo uso de librerías proporcionadas por RIFFA. De esta manera, se ocultan los detalles de la comunicación y las estructuras del *kernel* al usuario, para facilitar su utilización y evitar posibles errores o complicaciones [29]. Las funciones software proporcionadas por RIFFA para hacer uso de la API son las siguientes:

- fpgaMapMemory. Permite inicializar la conexión con la FPGA, realizando el mapeado del rango de direcciones de PCI Express en la memoria y abriendo el archivo de dispositivo virtual.
- fpgaUnmapMemory. Finaliza la conexión de la FPGA, revirtiendo el mapeado en memoria de las direcciones de PCI Express y cerrando el archivo de dispositivo virtual.
- fpgaInterruptOpen. Espera a una interrupción específica, accediendo al descriptor del archivo de interrupción del núcleo IP especificado.
- fpgaInterruptClose. Finaliza las notificaciones de las interrupciones al liberar el descriptor del archivo de interrupción del núcleo IP especificado.
- fpgaInterruptWait. Bloquea el hilo especificado hasta que se recibe la interrupción específica del núcleo IP.
- fpgaFireInterrupt. Envía una interrupción al núcleo IP especificado.
- fpgaRequestDma. Desencadena una transferencia DMA entre la FPGA y la memoria de la estación de trabajo.
- fpgaReadWord. Realiza la lectura de una palabra de 32 bits ubicada en la memoria FPGA.
- fpgaWriteWord. Realiza la escritura de una palabra de 32 bits en la memoria FPGA.

En la Figura 16 se observa un diagrama con las relaciones entre las llamadas de funciones software, explicadas anteriormente, y la señalización hardware de la FPGA.

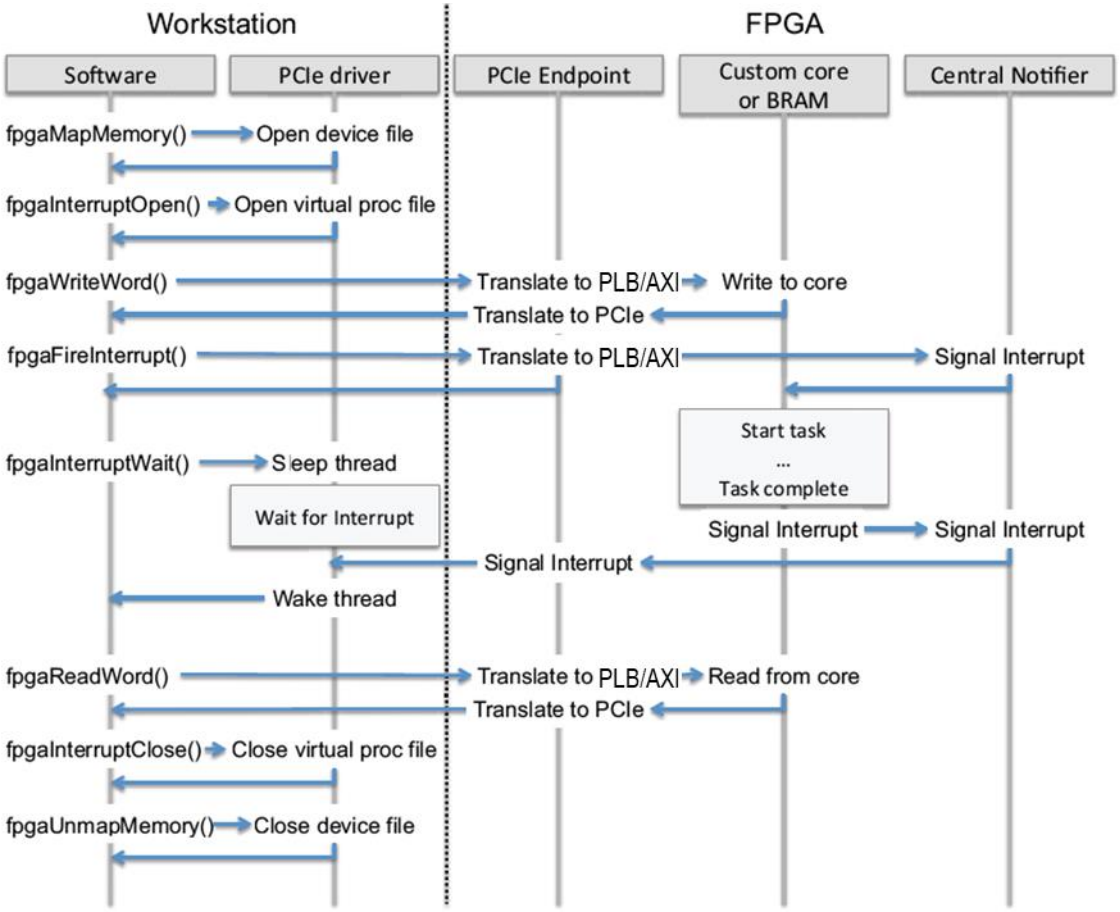

Figura 16: Diagrama de funciones software y señalización hardware de RIFFA en la FPGA [29]

3.2.2. Interfaz hardware

Los componentes de mayor importancia dentro de la interfaz hardware de RIFFA son el *endpoint* PCI Express, el controlador DMA, la central de notificación y los núcleos de peticiones DMA. A continuación, procederemos a explicar cada uno de ellos en detalle [29], [30].

El *endpoint* PCI Express es el encargado de gestionar el *slot* PCI Express, y funciona como PLB para el *bridge* PCI Express, para que el espacio de direcciones pueda ser mapeado entre los buses. Xilinx proporciona este IP, y se configura con mapeado BAR de 4 MB de IPIF a PCI Express. Este IP es el encargado de traducir los accesos de IPs con direcciones PLB o AXI a accesos de memoria PCI Express de la estación de trabajo usando direccionamiento PCI Express. También funciona de forma contraria, al realizar mapeado BAR de 8 KB de PCI Express a IPIF. IPIF es la interfaz IP, lo que implica que es la parte PLB/AXI del *bridge* PCI Express.

El controlador DMA también es un núcleo IP de Xilinx, el cual permite realizar las transferencias DMA haciendo uso del bus PLB o AXI. Cada vez que se realiza una transferencia

DMA, tras ser completada correctamente se envía una interrupción haciendo uso de una FIFO de 48 posiciones y un tamaño de *burst* de 16 para lectura y escritura PLB.

No obstante, la parte central de RIFFA es la central de notificación y sin ella, no se podría operar. Se encarga de inicializar el *endpoint* PCI Express, al realizar la escritura de los registros del mismo a través del bus PLB o AXI. Incluye las peticiones de interrupción desde los núcleos IP y los envía al *endpoint* PCI Express. Tras cada reinicio el *driver* envía la información referente a las direcciones asignadas del *kernel* a la central de notificación. Esta información es necesaria previa a la configuración del BAR de IPIF a PCI Express.

La central de notificación también gestiona todas las peticiones DMA y de interrupciones mediante los registros de bus mapeados. La gestión del espacio de registros se realiza mediante la partición en bloques que soportan transferencia DMA e interrupciones para los 16 posibles canales de los núcleos IP. Cada vez que se inicia una transferencia DMA se debe escribir en los registros de cada bloque, la dirección fuente, la dirección destino, el tamaño de la transferencia y el *flag* de interrupción. Una vez es recibido el valor de la longitud, la central de notificación incluye una petición de transferencia DMA a la cola de la FIFO, para que sea atendida por el controlador DMA en cuanto sea posible. Gracias a ello, solo es necesario hacer uso de un controlador DMA y es posible que el arbitraje de bus sea equilibrado y se eviten colisiones. Cuando la transacción DMA se finaliza, si se encontraba activo el *flag* de interrupción, se envía una interrupción o *doorbell* a la estación de trabajo o al núcleo IP, según corresponda. Si, en caso contrario, únicamente se quiere iniciar la interrupción o el *doorbell*, simplemente se envía una petición de transacción con longitud de transferencia igual a cero y con el *flag* de interrupción activo.

Por último, los núcleos de peticiones DMA se encargan de la señalización DMA al exportar un conjunto simplificado de señales para realizar las transferencias DMA. De esta manera se permite obtener la información de finalización de la transferencia gracias a una simple confirmación y una interfaz de pulso. Se puede acceder a la interfaz mediante lecturas y escrituras desde el bus de la FPGA [29], [30].

3.3. Versiones

Existen diferentes versiones de RIFFA. En este trabajo nos centraremos en la versión 2, ya que la versión 1 presenta errores e ineficiencias.

En RIFFA 2 se parte de un diagrama arquitectural de alto nivel y existen diferentes versiones, que han ido mejorando o añadiendo funcionalidad al sistema, o simplemente han solucionado pequeños errores encontrados en la versión inicial [31]. A continuación se expondrá brevemente las diferencias principales entre dichas versiones:

- RIFFA 2.0.1. Es una versión general de RIFFA 2 y no incluye compatibilidad con componentes de versiones previas (RIFFA 1). Las principales mejores son la adición de *scatter gather* DMA, un mayor ancho de banda y registro de señales adicional (lo que permite cumplir con las restricciones temporales). Sin embargo, no existe un archivo de registro del sistema en Windows, lo que dificulta el depurado. Se aconseja el uso de Windows Development Kit Debugger (WinDbg) para ello.
- RIFFA 2.0.2. Se realizan cambios en "rx_engine" para evitar bloqueos de TLPs. Para ello, se implementa una FIFO para manejar su gestión. A su vez, para evitar datos corruptos de TLPs, se incluye un reseteo de las FIFOs pertenecientes a "rx_port" antes de la recepción de una transacción.
- RIFFA 2.1.0. Se añade "reorder_queue" y diferentes parámetros para el número de *tags* y para la longitud de *payload* máxima para el tamaño de la RAM utilizada en "reorder_queue". También hay adición de control de flujo para evitar transacciones *upstream* desbordantes, únicamente en FPGAs de Altera. Se eliminan señales sin usar de "rx_port_requester_mux.v". Finalmente, se implementa FWFT FIFO en "rx_port_channel_gate.v".
- RIFFA 2.2.0. Incluye soporte para Gen3 Integrated Block para PCI Express y para VC709. Se incluyen ejemplos de ZC706. En lo respectivo a Xilinx, se varía el empaquetado de los proyectos de ejemplo, teniendo ahora IP instanciada todos los proyectos Virtex 7. También, se realiza la reescritura y rediseñado de diferentes bloques de transmisión y recepción, permitiendo así mayor reutilización de código.
- RIFFA 2.2.1. Añade lógica de *reset* en la capa de motor para el manejo de *resets* inducidos. Se incluye aviso de errores en el *driver* Linux de estabilidad y concurrencia multi-hilo.
- RIFFA 2.2.2. Incluye soporte para la API "get_user_pages" en el *kernel* de Linux [28].

3.4. Implementación para ZC706

Como se ha comentado anteriormente, la placa de desarrollo seleccionada para realizar este proyecto ha sido la FPGA Xilinx ZC706. Dentro de los ejemplos proporcionados por RIFFA se encuentra uno que ya ha sido adaptado para esta placa de desarrollo. En nuestro caso hemos decidido implementar la última versión de RIFFA disponible, la 2.2.2, pues es la que incluye todas las correcciones de errores encontradas en las versiones previas y las mejoras pertinentes, sobre todo respecto a la versión inicial de RIFFA.

Para la implementación de RIFFA en la FPGA, se ha hecho uso de la herramienta de Xilinx Vivado. En la carpeta de ejemplos proporcionada por RIFFA, se encuentra un proyecto que fue realizado en la versión de Vivado 2014.4.

El proyecto se ha sintetizado en la versión 2018.1 de Vivado. Ello ha supuesto hacer ajustes en el proyecto, y adaptar el código de referencia proporcionado a las exigencias de la nueva versión de las herramientas.

El principal error tiene lugar al volver a realizar la síntesis y la implementación del proyecto. Esto se debe a una función de Verilog que no tiene soporte para las versiones de Vivado posteriores a 2016.4, pues se comprobó que hasta esta versión no era necesario realizar cambios para obtener el *bitstream* y que funciona de forma correcta RIFFA sobre la FPGA.

La función de Verilog es "clog2s". Sin embargo, sí existe soporte para esta función en SystemVerilog en la versión de Vivado 2018.1. Por tanto, es necesario cambiar el lenguaje usado para la codificación del archivo "riffa_wrapper_zc706.v".

Otro de los problemas encontrados es la redeclaración de funciones, soportado sin problemas en Verilog pero que SystemVerilog no permite. Sin embargo, esta problemática no influye para la placa utilizada en este proyecto, y será necesario tenerla en cuenta para placas con soporte a dispositivos UltraScale.

Una vez portado el proyecto a Vivado 2018.1 se ha realizado la síntesis y generado el *bitstream*, pudiendo incluir RIFFA en otros proyectos o efectuar cambios respecto al número de canales o alguna otra modificación que se crea necesaria. El diagrama de bloques utilizado para ello es el que se presenta en la Figura 17. Como se puede observar se encuentra el bloque de RIFFA conectado al bloque PCIeGen2x4lf128_i, el cual incluye el PCIe de Xilinx. Se muestran las entradas y salidas del sistema, las cuales permiten la sincronización de este y el envío y la recepción de datos con el *host* CPU. En la Figura 18 se muestra el resultado obtenido en Vivado 2018.1 de la implementación sobre la FPGA de RIFFA con ancho de interfaz AXI de 128 bit, observando los recursos utilizados para ello. En rojo se muestran los recursos utilizados por el *wrapper* de RIFFA.

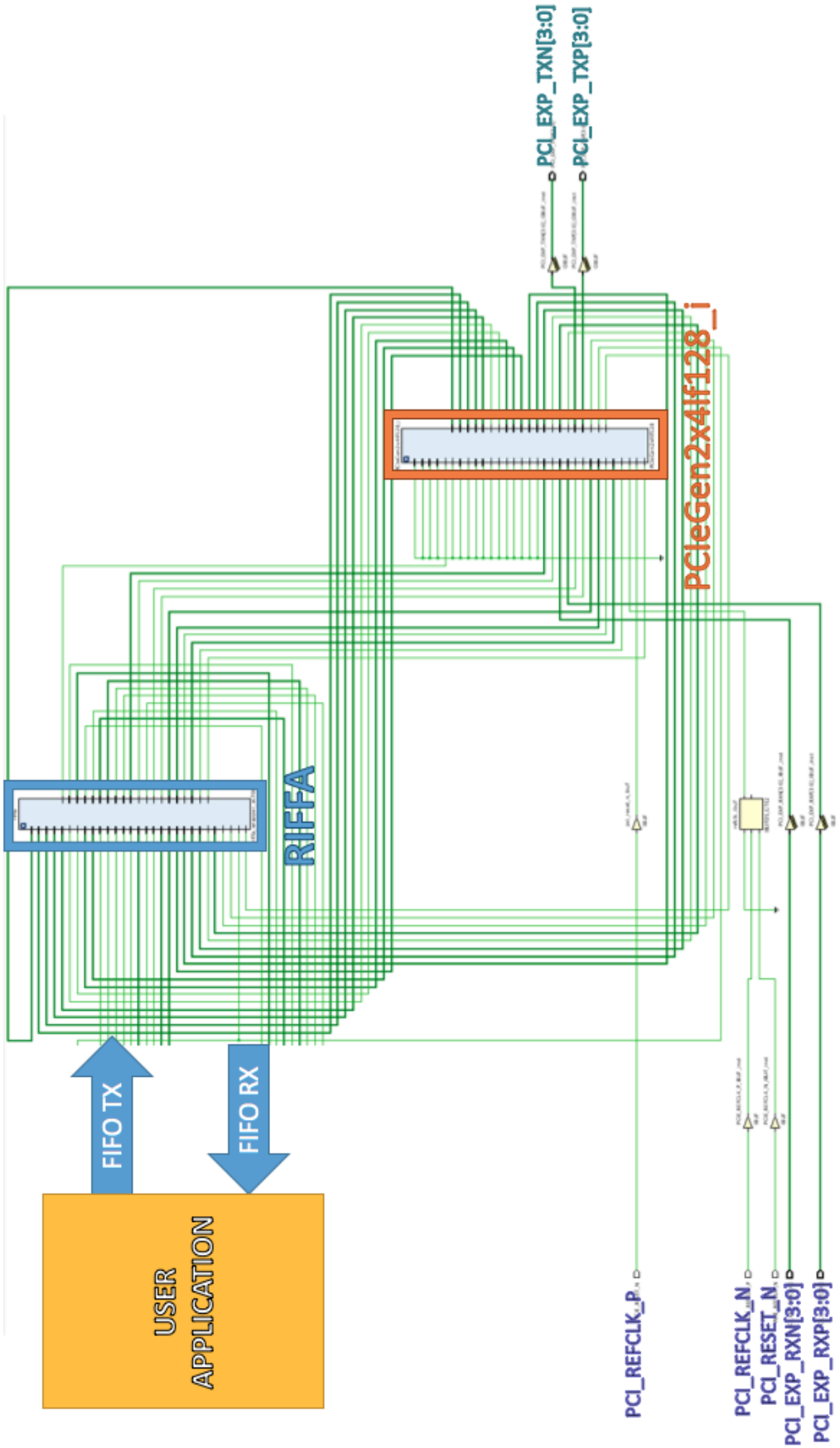

Figura 17: Diagrama de la plataforma con RIFFA

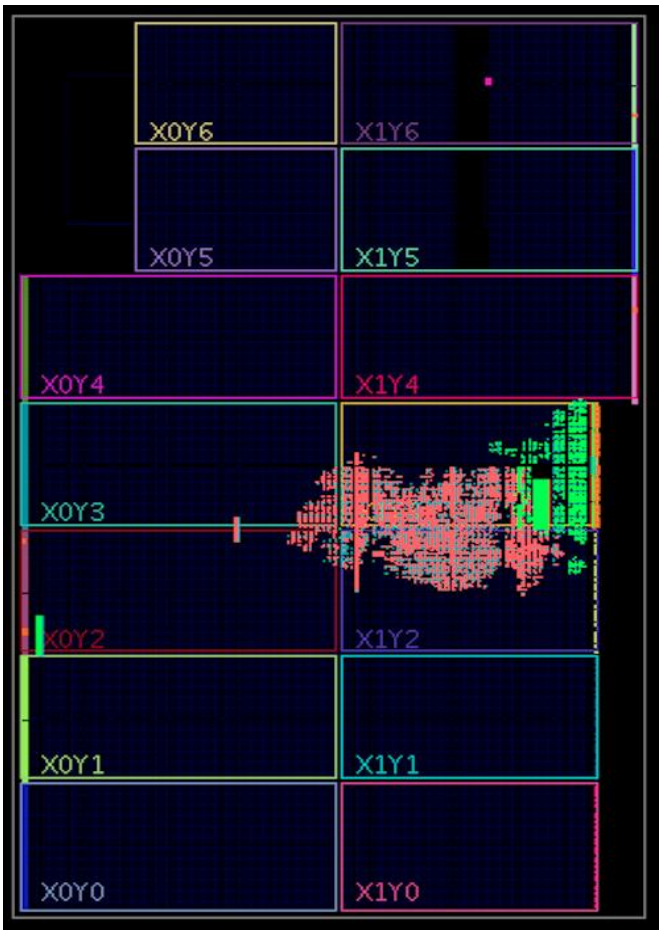

Figura 18: Layout de la implementación de RIFFA (AXI 128 bit) en ZC706

En la Figura 19 se presentan los resultados del análisis temporal del diseño, obteniendo un *slack* positivo de 0,89 ns. De esta manera se comprueba que las limitaciones temporales se cumplen, permitiendo el correcto uso del bus de comunicaciones PCI Express con las restricciones impuestas. También se pueden observar en la Figura 20 los porcentajes de recursos utilizados en la implementación del sistema, tras haber realizado la implementación.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.890 ns	Worst Hold Slack (WHS):	0.034 ns	Worst Pulse Width Slack (WPWS):	0.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	25352	Total Number of Endpoints:	25352	Total Number of Endpoints:	12544

All user specified timing constraints are met.

Figura 19: Resultados de temporización del diseño con RIFFA (AXI 128 bit)

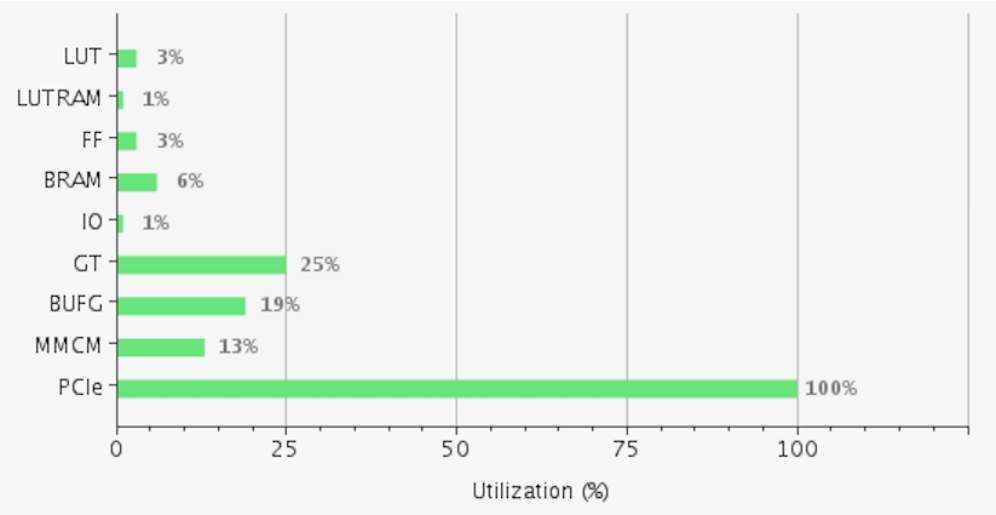

Figura 20: Gráfico de utilización de recursos tras la implementación de RIFFA (AXI 128 bit)

Las LUTs utilizadas ascienden a 7.364, siendo un 3,37% de las disponibles en la placa de desarrollo ZC706, así como del bloque PCIe de Xilinx completo. En la Tabla 4 se muestran todos los elementos utilizados, incluyendo la cantidad de cada uno de ellos y el porcentaje respecto a los disponibles en la placa de desarrollo. Como se puede observar, la gran mayoría de LUTs, LUTRAMs, FFs y BRAMs pertenecen a la implementación del *wrapper* de RIFFA, mientras que no requiere del resto de elementos.

Tabla 4: Utilización de recursos del sistema con RIFFA (AXI 128 bit)

	Sistema completo		**RIFFA *wrapper***	
LUT	7.364	3,37 %	5.585	2,55 %
LUTRAM	225	0,32 %	193	0,27 %
FF	12.061	2,76 %	9.668	2,21 %
BRAM	31	5,69 %	27	4,95 %
IO	5	1,38 %	0	0,00 %
GT	4	25,00 %	0	0,00 %
BUFG	6	18,75 %	0	0,00 %
MMCM	1	12,50 %	0	0,00 %
PCIe	1	100%	0	0,00 %

A su vez, cabe resaltar los resultados obtenidos al realizar un análisis de la potencia consumida por el sistema implementado. En la Figura 21 se muestra el consumo de potencia de la plataforma, siendo igual a 1,878 W, correspondiendo una parte importante a los *transceivers* GTX. Sin embargo, el consumo del bloque PCIe de Xilinx no supera los 58 mW.

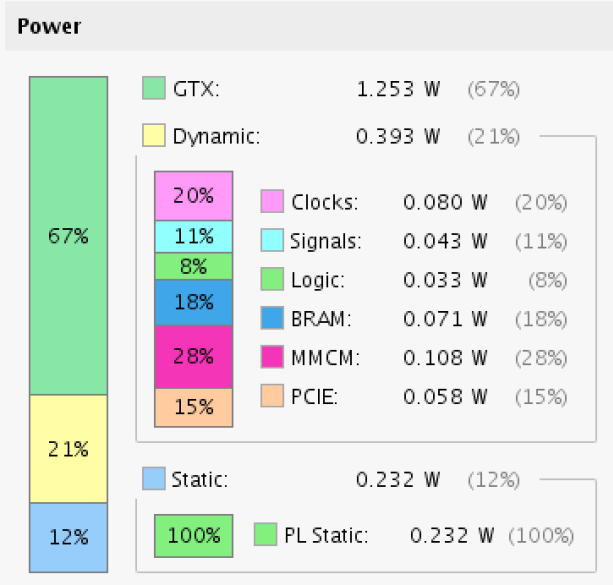

Figura 21: Consumo de potencia de la plataforma con RIFFA (AXI 128 bit)

No obstante, para la implementación sobre la FPGA de RIFFA con ancho de interfaz AXI de 64 bit se producen pequeños cambios respecto a los recursos utilizados. En la Figura 22 se observan los recursos utilizados al implementarlo en Vivado 2018.1. En rojo se muestran los recursos utilizados por RIFFA y en verde los recursos que utiliza el bloque de Xilinx para PCI Express.

Figura 22: Layout de la implementación de RIFFA (AXI 64 bit) en ZC706

En la Figura 23 se presentan los resultados del análisis temporal del diseño, obteniendo un *slack* positivo de 2,576 ns. De esta manera se comprueba que las limitaciones temporales se cumplen, permitiendo el correcto uso del bus de comunicaciones PCI Express con las restricciones impuestas, pero en este caso para una interfaz AXI de 64 bit. En la Figura 24 se muestra un gráfico con los porcentajes de recursos utilizados en la implementación del sistema una vez realizada la implementación.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.576 ns	Worst Hold Slack (WHS):	0.034 ns	Worst Pulse Width Slack (WPWS):	0.545 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	19637	Total Number of Endpoints:	19637	Total Number of Endpoints:	9539

All user specified timing constraints are met.

Figura 23: Resultados de temporización del diseño con RIFFA (AXI 64 bit)

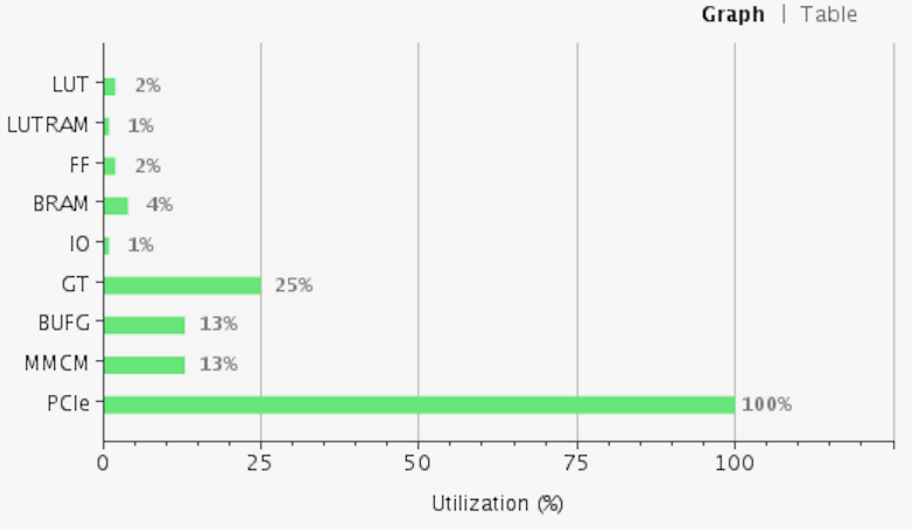

Figura 24: Gráfico de utilización de recursos tras la implementación de RIFFA (AXI 64 bit)

Para este caso, las LUTs utilizadas son 5.343, siendo únicamente un 2,44% de las disponibles en la placa de desarrollo ZC706 pero se requiere del bloque PCIe de Xilinx completo, tal y como era de esperar. En la Tabla 5 se muestra la cantidad de elementos utilizados y el porcentaje respecto a los disponibles en la placa de desarrollo. Como se puede observar, la gran mayoría de LUTs, LUTRAMs, FFs y BRAMs pertenecen a la implementación del *wrapper* de RIFFA, mientras que no requiere del resto de elementos. Las características son prácticamente iguales a la versión de RIFFA con la interfaz AXI de 128 bit, simplemente se reduce el uso de elementos a utilizar.

Tabla 5: Utilización de recursos del sistema con RIFFA (AXI 64 bit)

	Sistema completo		RIFFA *wrapper*	
LUT	5.343	2,44 %	3.649	1,67 %
LUTRAM	157	0,22 %	125	0,18 %
FF	9.186	2,10 %	7.189	1,64 %
BRAM	24	4,40 %	20	3,67 %
IO	5	1,38 %	0	0,00 %
GT	4	25,00 %	0	0,00 %
BUFG	4	12,50 %	0	0,00 %
MMCM	1	12,50 %	0	0,00 %
PCIe	1	100%	0	0,00 %

Finalmente, en la Figura 25 se muestran los valores obtenidos al realizar el análisis de potencia consumida para el sistema que implementa RIFFA con la interfaz AXI de 64 bit. Se muestra la distribución del consumo de potencia de la plataforma y su subdivisión entre los diferentes elementos que conforman el sistema, siendo el total de la potencia consumida de 1,58 W. Sin embargo, el consumo del bloque PCIe de Xilinx no supera los 35 mW. Se vuelve a apreciar el impacto de los *tranceivers* GTX en el consumo total de potencia.

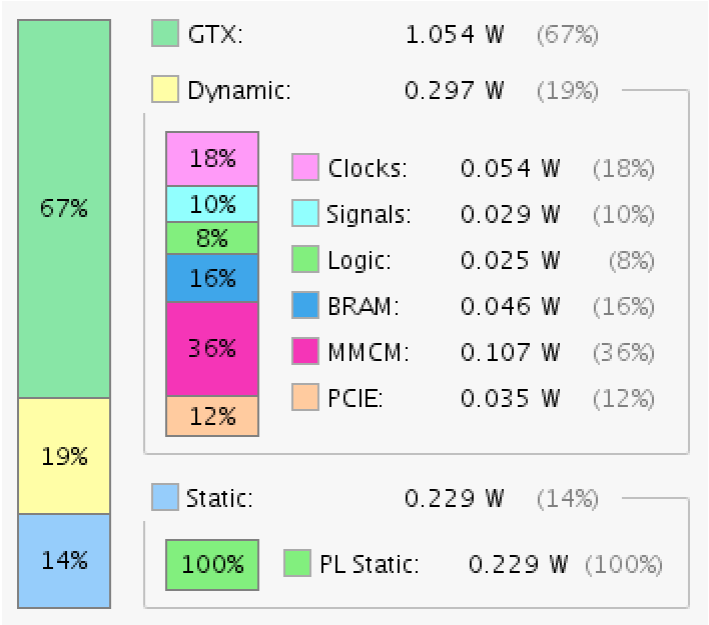

Figura 25: Consumo de potencia de la plataforma con RIFFA (AXI 64 bit)

3.5. Conclusiones

PCI Express es común en los sistemas empotrados, pues ofrece un alto ancho de banda en sus conexiones, junto a una baja latencia. Gracias al uso de RIFFA, es posible alcanzar un 97% del ancho de banda disponible en una conexión PCI Express.

Dentro de sus características más destacables se encuentra la inclusión de una API software que permite la transferencia de datos de manera eficaz y sencilla y una interfaz hardware basada en FIFO que permite realizar comunicaciones en *streaming*.

Cabe concluir que RIFFA sin necesidad de realizar grandes cambios ni esfuerzos permite la implementación del bus de comunicaciones PCI Express. Por lo que es una solución básica, eficiente, con alto ancho de banda, sin costes y multiplataforma. Además, no incluye ningún tipo de restricciones en lo que se refiere al *framework* o al hardware. Gracias a RIFFA, no es necesario entrar en los detalles específicos del protocolo PCI Express, permitiendo direccionar la tarea del programador a la implementación lógica de aplicaciones u otro cometido.

Capítulo 4. Clasificador

La importancia de los clasificadores viene determinada por la necesidad creciente de facilitar el procesado y distribución de los datos. Si se parte de un conocimiento previo de los datos, se consigue un sistema más eficiente cuyo procesado es más específico. Por ello, los clasificadores no son únicamente necesarios en proyectos relacionados con Big Data, sino, prácticamente, en casi cualquier ámbito en el que se requiera de procesado y/o distribución de datos.

Como se comentó con anterioridad, este proyecto se centra en obtener un procesamiento, distribución y almacenaje de datos en *streaming* y de datos estáticos almacenados previamente en memoria para una placa basada en dispositivos FPGA. Se parte de la base de que el clasificador desarrollado debe ser apto para implementarse en una estructura básica tipo Big Data, la cual incluye un modelo de programación paralelo MapReduce. El objetivo principal es desarrollar un clasificador que pueda incluirse entre estas etapas, permitiendo organizar los datos en la etapa de *Sort* de Big Data. De esta manera, se podrá obtener un sistema eficiente que permitirá aligerar la carga del sistema y mejorar los resultados del procesado y/o distribución de datos [1], [5].

Por ello, se ha procedido a realizar un estudio de diferentes clasificadores, para decidir el tipo de clasificador óptimo para las circunstancias previamente expuestas. Se requiere de un clasificador genérico, que incluya adaptabilidad a posibles cambios, y que sin incluir demasiada latencia al sistema, permita aligerar la carga de las etapas posteriores. Como consecuencia, en los siguientes apartados se mostrará un breve resumen de los diferentes clasificadores estudiados, una comparativa de sus funcionalidades y la explicación del clasificador escogido.

4.1. Estado del arte

Existe una amplia gama de clasificadores en función de la aplicación requerida. Sin embargo, es posible dividirlos dependiendo de los atributos utilizados para la clasificación y de los diferentes métodos utilizados para ello. No obstante, cuando se habla de Big Data se suelen considerar los algoritmos de *sorting* como los clasificadores del sistema.

Los algoritmos de *sorting* han sido diseñados para diferentes problemas y/o diferentes aplicaciones, y por ello existe una gran variedad. Sus características, funcionalidades y limitaciones han sido analizadas y optimizadas en numerosas ocasiones, a la vez que surgen nuevos algoritmos de *sorting*, que en muchas ocasiones son simples modificaciones de algoritmos previos.

No existe una única opción válida para todas las aplicaciones o sistemas donde quiera implementarse un algoritmo de *sorting*, ya que depende y varía en función de las características propias del algoritmo, como son la velocidad, el rendimiento o el espacio de memoria utilizado. Todo ello se debe aplicar teniendo en cuenta la función del sistema sobre el que se quiera implementar y sus características intrínsecas.

4.2. Clasificadores estudiados

En este apartado se mostrará un breve resumen de los clasificadores estudiados, mostrando tanto sus ventajas como sus defectos, para posteriormente proceder a realizar una comparativa de ellos. Los clasificadores que han sido estudiados se basan en los diferentes algoritmos de *sorting*, ya que para incluirlos dentro de un sistema Big Data aportan soluciones más eficientes y prácticas. Los diferentes algoritmos de *sorting* que se han escogido como posibles candidatos son los siguientes:

1. Insertion Sort
2. Quick Sort
3. Heap Sort
4. Merge Sort
5. Shell Sort
6. Buble Sort
7. Sorting Networks
8. Bitonic Sort

La idea general es encontrar un algoritmo eficiente en términos de implementación hardware para lo cual se utilizarán metodologías de diseño basadas en síntesis de alto nivel a partir de su especificación C/C++, OpenCL o SystemC.

4.2.1. Insertion Sort

Uno de los algoritmos de *sorting* más conocidos es el *insertion sort* o clasificación por inserción. Destaca por utilizar *keys* o claves de búsqueda para realizar la clasificación de los datos. Para su implementación hardware se requiere de un registro de desplazamiento para almacenar estas claves y un comparador lógico asociado a cada registro individual. Cada vez que se introduce un

nuevo dato, este valor de entrada se compara con el contenido del registro actual. En función del resultado de dicha comparación se genera una posición en el registro en el valor adecuado. Debido a ello, se requiere que se intercambien las claves mayores. En la Figura 26 se muestra el funcionamiento del algoritmo de *insertion sort*.

No obstante, la arquitectura básica necesaria no permite implementaciones a elevada velocidad. Esto se debe a la cantidad de comparadores necesarios para ello. En lo que se refiere a los recursos necesarios para la correcta implementación de *insertion sort* en caso de querer clasificar N valores (o claves), es de N comparadores y N registros, teniendo lugar la clasificación en N ciclos de operación. Debido a ello, al tener que transmitir la señal de entrada a todas las instancias de los comparadores, se crea una limitación en la cantidad de datos a clasificar y su tamaño. Mientras mayor sea la arquitectura, los ciclos de operación aumentarán, provocando una disminución notoria de la velocidad del sistema. No obstante, gracias a estos parámetros se obtiene un clasificador que ofrece unos valores lineales cuando se tratan de pequeñas cantidades de datos, pudiendo ser efectivo en una amplia variedad de aplicaciones [32], [33].

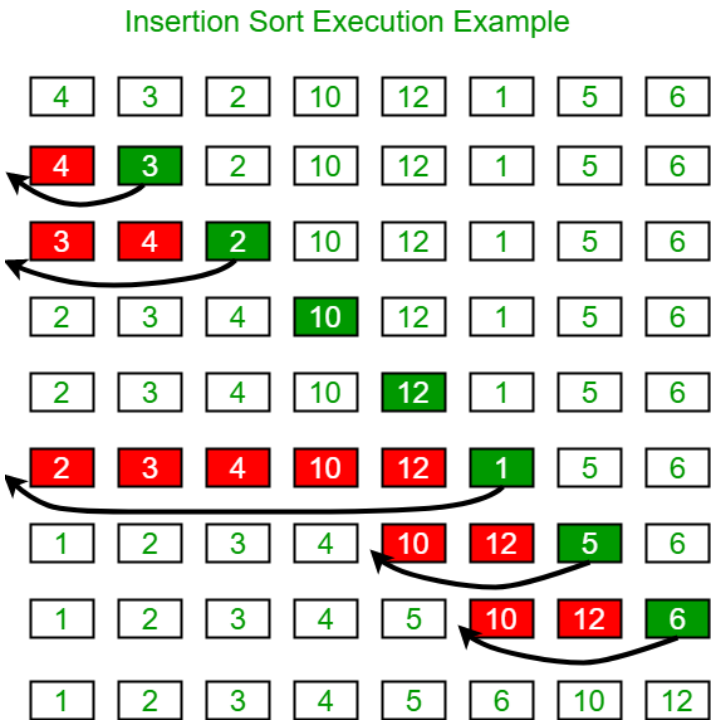

Figura 26: Esquema de funcionamiento del algoritmo de insertion sort [34]

También cabe destacar que para FPGAs es necesario hacer uso de registros para almacenar las claves de búsqueda, en lugar de bloques de memoria *on-chip* dedicados. Esto implica un alto coste de implementación relativo y limita este tipo de algoritmos para problemas de limitadas dimensiones. Desde el punto de vista del tiempo de cómputo, el caso mejor es cuando el conjunto de valores de entrada ya están ordenados y en este caso tiene un comportamiento lineal O(n), donde n

es la cardinalidad del conjunto a ordenar. En este caso únicamente es necesario realizar una comparación con la parte más a la derecha del *array* ya ordenado. Sin embargo el peor caso se presenta cuando el conjunto de valores está ordenado en orden inverso. En este caso el comportamiento es cuadrático en cuanto a tiempo $O(n^2)$. En el caso promedio presenta un comportamiento cuadrático desde el punto de vista temporal. Este algoritmo no presenta buenas propiedades para su implementación hardware por su naturaleza secuencial y su comportamiento cuadrático desde el punto de vista temporal.

4.2.2. Quick Sort

El algoritmo *quick sort* utiliza un esquema de partición. Partiendo de un *array* de datos escoge un elemento pivote que representa el punto de partición del conjunto de datos. A continuación, se colocan todos los elementos menores que el elemento pivote previos a este y los elementos mayores después del mismo, realizándose la ejecución en tiempo lineal. El siguiente paso a realizar es volver a escoger un elemento pivote, pero en esta ocasión se efectuará a ambos lados del elemento pivote inicial. Es decir, por un lado los elementos menores del pivote inicial y por otro lado los mayores. Se vuelve a realizar la recolocación de los datos, teniendo en cuenta los menores y mayores en cada uno de los grupos independientes, en función de los nuevos pivotes. Este proceso se repetirá hasta que solamente quede un elemento en cada uno de los grupos, obteniendo de esta manera el *array* de datos ordenado [32], [35].

Existen cuatro maneras diferentes para elegir el elemento pivote, en función del tipo de algoritmo de *quick sort* que se desee implementar. Se puede elegir el elemento pivote de forma aleatoria, escogiendo la mediana, el primer elemento del *array* o el último. En la Figura 27 se muestra un ejemplo del funcionamiento de un algoritmo *quick sort* que siempre elige como elemento pivote el último elemento del *array*.

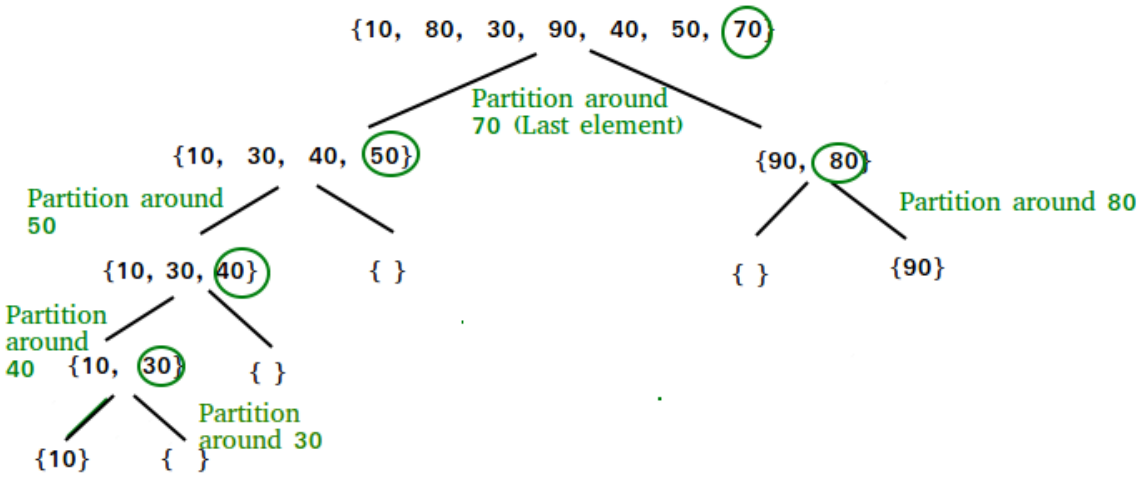

Figura 27: Esquema de funcionamiento del algoritmo quick sort [35]

No obstante, existen limitaciones respecto a este algoritmo, pues puede resultar poco práctico, ya que tiene una dificultad ascendente en función del número de elementos a ordenar y el tiempo de ejecución requerido para ello. En caso de que se encuentren los números ya ordenados en alguna de las etapas, no se realizan comprobaciones, por lo que se debe esperar hasta que finalice el algoritmo en todas las etapas. Esto implica que el algoritmo no es eficiente en cuanto al tiempo de ejecución. Sin embargo se trata de un algoritmo muy sencillo de implementar y ejecutar, ya que no requiere demasiados recursos, es recomendable para aplicaciones que requieran de limitadas cantidades de datos a clasificar.

4.2.3. Heap Sort

El algoritmo de *heap sort* se basa en el uso de la estructura de datos *binary heap*. Esta estructura de datos está formada por un árbol binario completo, el cual dispone de una condición de ordenación especial. En este caso cada nodo tiene un valor mayor o menor, según se determine el tipo de *binary heap*, que sus dos nodos inferiores. Para el algoritmo de *heap sort* se puede decidir que el *binary heap* se encuentre ordenado de forma ascendente o descendente, según sea más conveniente para la aplicación.

Inicialmente, se requiere realizar comparaciones entre los valores hasta obtener el dato de mayor o menor valor, según se haya escogido el tipo de *heap sort*. Una vez obtenido este valor se coloca en la posición raíz del árbol binario completo, colocando el resto de valores en las posiciones inferiores, y realizando comparaciones entre estos hasta obtener el árbol binario completo ordenado [32], [36]. Gracias al uso de *binary heap*, se pueden aplicar *arrays* para facilitar su implementación, reduciendo el uso de espacio en memoria necesario. En la Figura 28 se muestra un ejemplo de funcionamiento de un algoritmo de *heap sort* descendente.

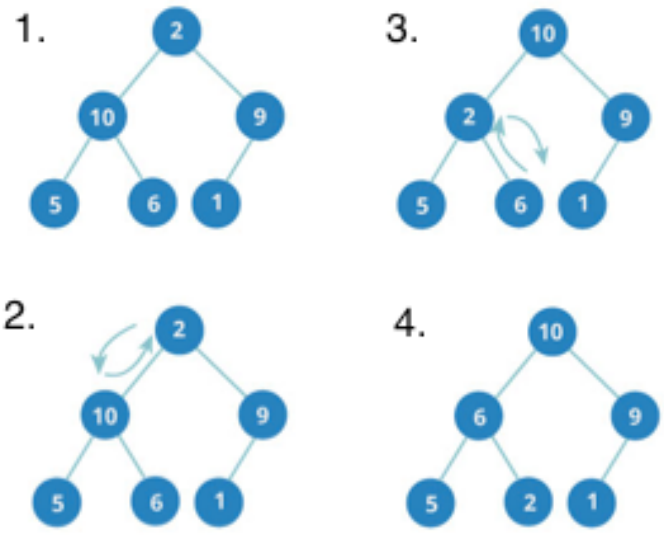

Figura 28: Esquema de funcionamiento del algoritmo heap sort descendente [36]

4.2.4. Merge Sort

Los clasificadores de combinación o *merge sort* se basan principalmente en la división de los datos para facilitar su ordenación. Se realiza una división de los datos en dos, obteniendo dos *arrays* independientes, y sobre cada uno de ellos se realiza la clasificación de datos. Una vez se hayan ordenado individualmente ambos *arrays*, se procede a la combinación de ambos de forma ordenada. Realmente, se procede a realizar la subdivisión de cada uno de los *arrays* hasta llegar a la unidad, para posteriormente proceder a la reconstrucción de los *arrays* de forma ordenada. Gracias a este algoritmo se pueden realizar tareas en paralelo de comparación de datos, reduciendo el tiempo de ejecución del mismo [32], [37].

Se trata de un algoritmo recursivo, siendo bastante estable su ejecución, ya que, a pesar de realizar la división del *array* en dos, el tiempo requerido para clasificar cada uno de estos *arrays* de forma paralela es lineal. En la Figura 29 se muestra un ejemplo de funcionamiento del algoritmo de *merge sort*, en el cual primero tienen lugar las etapas de división para posteriormente proceder a la combinación y ordenación de los datos.

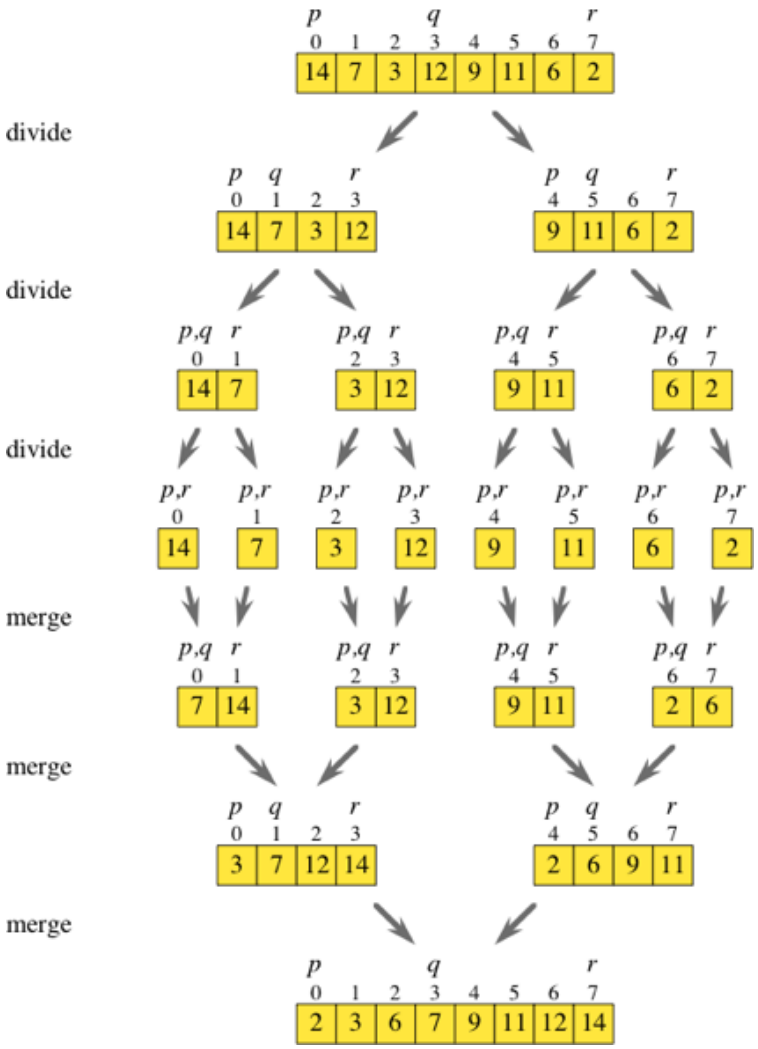

Figura 29: Esquema de funcionamiento del algoritmo merge sort [37]

No obstante, es posible que los clasificadores de combinación estén basados en FIFOs. Su característica principal es que se selecciona un elemento de valor-selección y se rodea por FIFOs en todos sus puertos de entrada y salida. De esta manera, es posible obtener a la salida el valor conjunto de la combinación de las dos FIFOs de entrada.

Sin embargo, también existen limitaciones cuando se requieren de grandes cantidades de datos a clasificar. Es posible hacer uso de múltiples FIFOs en cascada con clasificadores de combinación, lo que implica un aumento de la cantidad de memoria *on-chip* local y el tamaño de la secuencia clasificada. Este tipo de sistemas son muy prácticos para aplicaciones *off-stream*. No se puede garantizar que haya suficiente espacio en las FIFOs de entrada tras obtener los primeros paquetes, pues dicho espacio estará delimitado por el procesamiento de los datos actuales mediante el algoritmo de clasificación de combinación, que es dependiente de los datos y consecutivamente del espacio libre de las FIFOs [33].

4.2.5. Shell Sort

El algoritmo de *shell sort* se trata de un algoritmo de inserción, pero permitiendo que los cambios puedan producirse a pesar de encontrarse los datos a distancias alejadas. Por ello, las comparaciones de datos en lugar de realizarse entre datos adyacentes, como era el caso de *insertion sort*, se realizan entre datos distantes. Para comenzar a realizar la clasificación se requiere de la elección de un *gap* o intervalo de datos, que normalmente al inicio es la mitad del *array* completo con todos los datos a clasificar. A continuación, se procede a realizar la comparativa de los datos que se encuentran a este intervalo seleccionado, realizando intercambios entre las posiciones siempre que al comparar los valores estos no se encuentren ordenados de mayor a menor o viceversa, según se haya determinado. Tras completar todas las comparativas con los datos que se encuentran a la distancia del *gap* inicial, se procede a reducir este intervalo a la mitad para volver a aplicarlo sobre el *array* y proceder a la clasificación de los datos y su correspondiente intercambio en el *array* o no, según corresponda. Se continuará realizando la disminución del *gap* tras ser aplicado, hasta que este llegue a la unidad, obteniendo el *array* con todos los datos ordenados [32], [38]. En la Figura 30 se muestra un ejemplo del funcionamiento del algoritmo de *shell sort*.

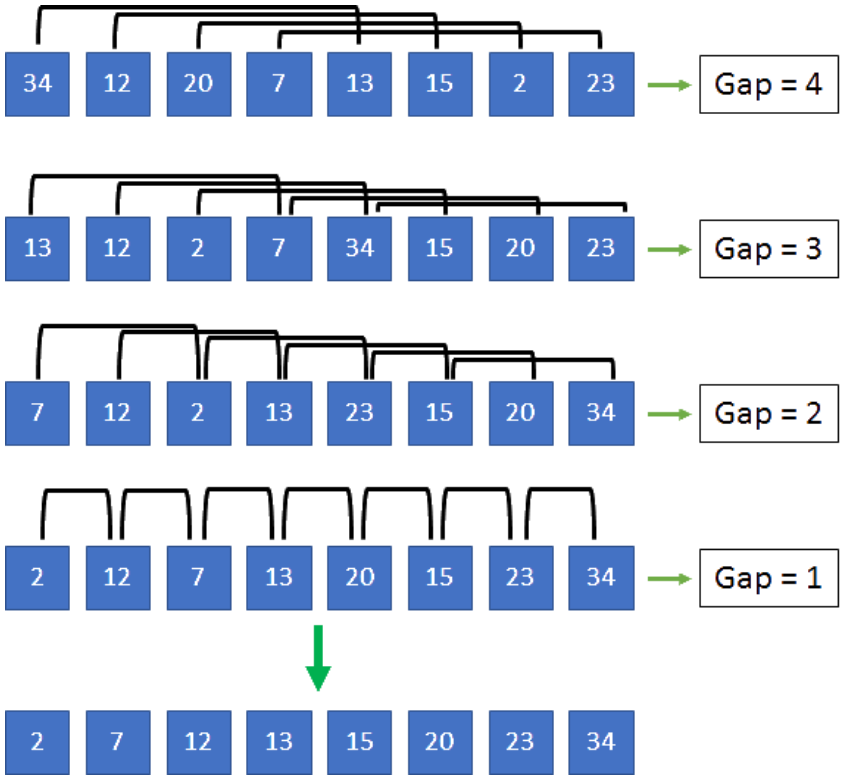

Figura 30: Esquema de funcionamiento del algoritmo de shell sort [38]

La principal diferencia entre este algoritmo y el algoritmo de *insertion sort*, es que al realizar comparativas entre datos distantes, se obtiene una colocación temprana de los datos en valores próximos a los finales. También, al permitir cambios entre posiciones distantes, no se requiere cambiar posición a posición en el *array*, como sucede en muchos algoritmos de *insertion sort*. No obstante, su implementación es bastante más compleja, pero se reduce el tiempo de ejecución.

4.2.6. Bubble Sort

El algoritmo de *bubble sort* es un algoritmo muy simple, pues realiza la comparativa entre datos adyacentes, empezando por el comienzo y procediendo a su intercambio en caso de que estos no se encuentren ordenados de forma correcta. No obstante, es bastante poco recomendable en la práctica, porque a pesar de ser muy fácil de implementar, requiere de múltiples repeticiones hasta que no se produzca ningún intercambio entre los datos, para asegurarse de que estos se encuentran clasificados correctamente. Esto supone un alto coste en tiempo de ejecución que para grandes cantidades de datos o aplicaciones en tiempo real no resulta nada práctico [32], [39]. En la Figura 31 se muestra un ejemplo claro del funcionamiento del algoritmo, y como la tercera repetición se realiza, pero es completamente innecesaria al ya encontrarse clasificado el *array*.

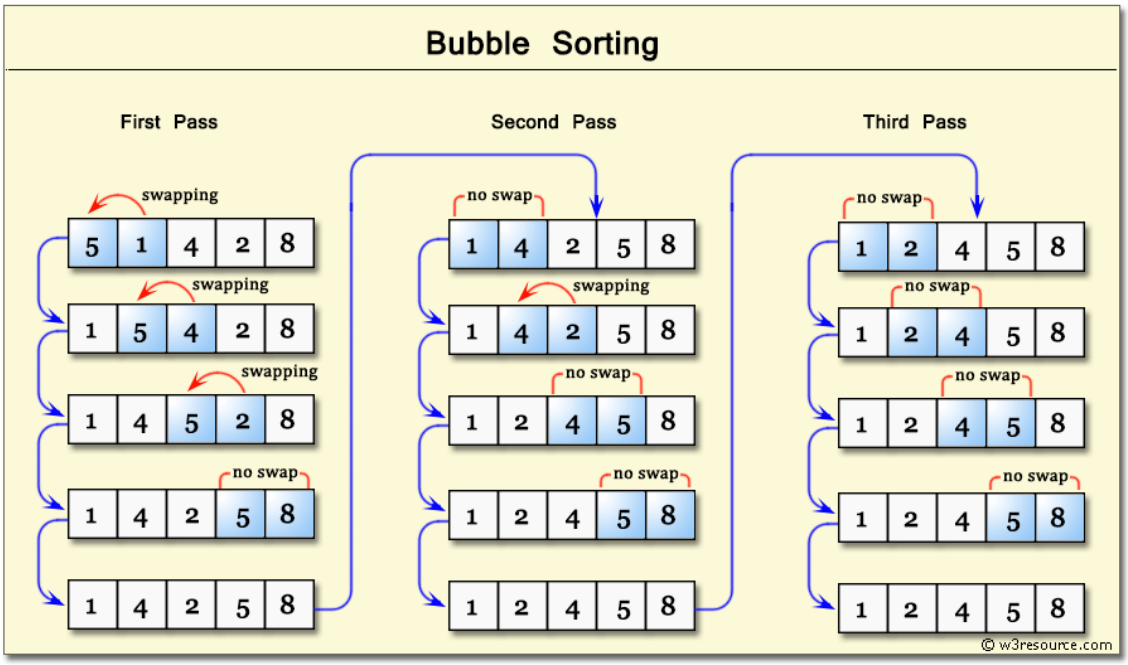

Figura 31: Esquema de funcionamiento del algoritmo de bubble sort [39]

4.2.7. Sorting Networks

Las redes de clasificación o *sorting networks* se basan en conjuntos de modelos matemáticos de redes que hacen uso de un alto grado de paralelismo para conseguir una clasificación rápida de grandes cantidades de datos. Para ello, se necesitan elementos de comparación e intercambio (*compare-swap*) que trabajen de forma paralela [32]. A pesar de poder trabajar de manera combinacional es posible añadir un registro de *pipeline* tras cada elemento *compare-swap*, siendo particularmente efectivo cuando se trata con grandes cantidades de datos y claves.

Existen una gran variedad de redes de clasificación, no obstante como valor genérico se puede decir que para un conjunto de N claves de búsqueda se requiere de una media de $\frac{log_2 N}{2}(log_2 N + 1)$ ciclos de reloj, suponiendo un coste hardware de $\frac{N}{4}(log_2 N)^2$ elementos *compare-swap* [33].

Algunas de las arquitecturas que suelen utilizarse como redes de clasificación incluyen los algoritmos *bitonic sort* y *merge sort*. No obstante, cuando se implementa en FPGAs no es posible abarcar grandes problemas debido a la creciente necesidad de capacidad, tanto de entrada como de salida, al aumentar las claves de búsqueda y generarse grandes cantidades de datos al mismo tiempo. Esto converge en una limitación de las interfaces de entrada/salida y de la lógica de la FPGA, pues se requiere alternar las operaciones de *burst* de entrada/salida y de procesamiento. De esta manera se pierde rendimiento de la propia FPGA, pudiendo subsanarse si se trabaja con la división de problemas de gran escala, realizándose múltiples redes de clasificación de menor tamaño. En una red de *sorting* la secuencia de comparaciones realizadas no es dependiente de los datos.

4.2.8. Bitonic Sort

Bitonic sorting es un tipo de red de *sorting*, que se usa en aplicaciones que requieran de elevadas velocidades de clasificación. Hace uso de vectores de *sorting* para facilitar la clasificación, permitiendo un alto rendimiento, especialmente para aplicaciones con datos centralizados. Se trata de un algoritmo paralelo que realiza la comparación de los elementos mediante una secuencia predefinida.

El funcionamiento de este algoritmo parte de una secuencia que primero se incrementa y después decrementa, esta secuencia es conocida como secuencia *bitonic*. Si tenemos un *array* con n valores, para los valores hasta x_i todos irán incrementándose ($x_0 < x_1 < \ldots \leq x_i$), mientras que a partir de x_i todos los valores irán decrementándose ($x_i > x_{i+1} > \ldots \geq x_n$). No obstante, se considera una secuencia *bitonic* ascendente si la parte decreciente se encuentra vacía y una secuencia *bitonic* descendente si la parte creciente se encuentra vacía [40].

Para comenzar a realizar el algoritmo se requiere adaptar el *array* con los valores de datos como una secuencia *bitonic*. Se realiza la clasificación en orden creciente para la primera mitad de la secuencia y en orden decreciente para la segunda mitad. Para ello, cada dos elementos consecutivos se realiza una comparación, intercambiando entre ascendente y descendente. De esta manera se obtienen secuencias *bitonic* de cuatro elementos. El siguiente paso consiste en realizar la misma comparación para cuatro elementos consecutivos, realizando la comparación a una distancia de dos elementos, alternando igualmente entre ascendente y descendente para cada conjunto de elementos. La última comparativa a realizar será cuando el número de elementos consecutivos sea igual a la mitad del *array*. En este punto ya se obtendrá la secuencia *bitonic*.

A continuación se procede a realizar la comparación entre el primer elemento de la primera mitad y el primer elemento de la segunda mitad. En caso de que el segundo sea menor que el primero se realiza un intercambio, en caso contrario permanecen los valores en sus posiciones. Se procede a realizar la misma comparación entre el segundo elemento de la primera mitad y el segundo elemento de la segunda mitad, y así sucesivamente. En este paso se obtienen dos secuencias de *bitonic* en el mismo *array*. Se procede a realizar las mismas comparaciones para cada uno de las secuencias *bitonic* por separado. Este paso se repetirá hasta que el tamaño de la secuencia *bitonic* sea la unidad, obteniendo de esta manera el *array* ordenado [32], [40]. En la Figura 32 se muestra un ejemplo del funcionamiento del algoritmo de *bitonic sort* para un *array* de ocho elementos.

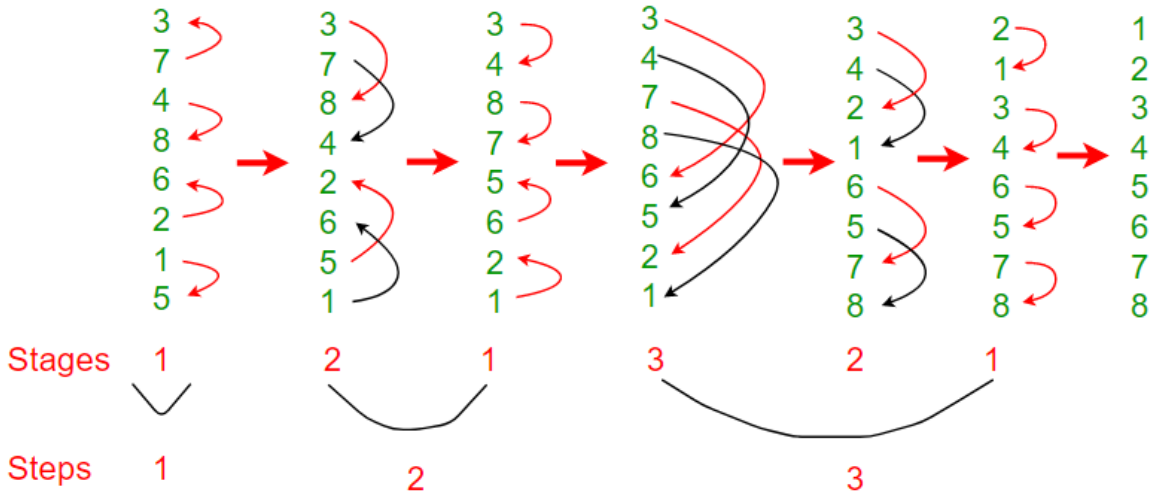

Figura 32: Esquema de funcionamiento del algoritmo de bitonic sort [40]

Gracias a que no existe dependencia entre los datos y la secuencia de comparaciones para la mayoría de las redes de *sorting*, hacen óptimo su uso para implementaciones hardware, como pueden ser las FPGAs [41].

4.2.9. Comparativa de algoritmos de *sorting*

Tras estudiar el funcionamiento de cada uno de los algoritmos mencionados anteriormente de forma individual, resulta importante realizar una comparativa entre ellos. Cada uno dispone de ventajas y desventajas en función de la aplicación sobre la que se desee implementar [42], [43]. A continuación, veremos brevemente las aplicaciones más utilizadas para cada uno de los algoritmos y una tabla con las características más destacadas.

El algoritmo de *insertion sort* es especialmente eficaz en aplicaciones con conjuntos de datos pequeños o para insertar elementos nuevos a una secuencia ya clasificada. En cambio *quick sort* es un algoritmo que se utiliza para aplicaciones que requieren de garantías probabilísticas, ya que depende de la distribución de los valores *key* de entrada. *Heap sort*, gracias al uso de la estructura de datos especial *heap* permite implementarse para aplicaciones en tiempo real, siempre y cuando se limite la velocidad de transmisión. *Merge sort* es un algoritmo que es altamente paralelizable, lo que permite su implementación en una amplia variedad de aplicaciones. También cabe destacar, que para obtener algoritmos más complejos se suele partir del algoritmo *merge sort* realizando combinaciones con otros algoritmos, debido a su alta versatilidad. *Shell sort* se usa en aplicaciones similares a *insertion sort* donde los datos a clasificar son limitados y se requiere de una implementación sencilla; siendo en muchas ocasiones preferible al propio algoritmo de *insertion sort*. No obstante, *bubble sort* a pesar de ser uno de los algoritmos más simples y con menor velocidad es útil para aplicaciones sencillas, ya que requiere de pocos recursos y tiene una fácil implementación. Finalmente, *bitonic sort* es un algoritmo especialmente práctico para implementaciones hardware o para *arrays* de

procesadores paralelos. Por ello, es especialmente interesante su utilización en FPGAs, pues permiten mejorar se rendimiento.

En la Tabla 6 y la Tabla 7 se muestra la comparativa de los algoritmos estudiados, mostrando la complejidad temporal, la velocidad, el método de cada algoritmo, la media de memoria necesaria, el tipo de complejidad, la estabilidad y la posibilidad de implementación paralela.

Tabla 6: Comparativa de los algoritmos de sorting I

	Complejidad temporal (media)	Complejidad temporal (pero caso)	Complejidad temporal (mejor caso)	Velocidad
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$	Lento
Quick sort	$O(n\ log(n))$	$O(n^2)$	$O(n\ log(n))$	Rápido
Heap sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	Medio
Merge sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	Medio
Shell sort	$O(n^2)$	$O(n^2)$	$O(n)$	Medio
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n)$	Lento
Bitonic sort	$O(\log^2(n))$	$O(\log^2(n))$	$O(\log^2(n))$	Rápido

Tabla 7: Comparativa de los algoritmos de sorting II

	Método	Memoria (media)	Tipo	Estabilidad	Implementación paralela
Insertion sort	*Insertion*	*O(1)*	Polinómico	Estable	No
Quick sort	*Partitioning*	*O(log(n))*	Logarítmico	Inestable	Sí
Heap sort	*Selection*	*O(1)*	Logarítmico	Inestable	No
Merge sort	*Merging*	*O(n)*	Logarítmico	Estable	Sí
Shell sort	*Insertion*	*O(1)*	Polinómico	Inestable	Parcialmente
Bubble sort	*Exchanging*	*O(1)*	Polinómico	Estable	No
Bitonic sort	*Bitonic*	-	Logarítmico	Inestable	Sí

4.3. Clasificador escogido

Se ha decidido hacer uso de un algoritmo de *bitonic sorting* debido a su facilidad de implementación hardware sobre FPGA, presentando buena capacidad de paralelismo y soportando una buena escalabilidad en los datos. El bloque obtenido es capaz de ser integrado con el resto de la plataforma, incluyendo la interfaz de comunicaciones PCI Express y el resto de los bloques de la arquitectura MapReduce para Big Data.

Se ha partido del algoritmo de *bitonic sorting* desarrollado en OpenCL por Mehdi Roozmeh [41]. Este algoritmo parte de un código fuente para GPUs desarrollado por NVidia, que ha sido modificado para permitir su implementación sobre FPGAs y generar un RTL de alto rendimiento. Gracias al uso de directivas específicas de síntesis de alto nivel, como son *unrolling*, *pipelining* y particionado de memoria, OpenCL permite obtener un RTL optimizado.

Hace uso de la red de *sorting bitonic merge*, siendo esta una de las redes de *sorting* más rápidas debido a que las comparaciones en cada nivel se hacen en paralelo [44]. La profundidad de la red de *sorting* y el número de comparadores requerido en función del tamaño de datos a clasificar N vienen definidas por:

$$Profundidad\ de\ la\ red\ de\ sorting \rightarrow D(N) = \frac{(log_2 N(log_2 N + 1))}{2}$$

$$Número\ de\ comparadores \rightarrow C(N) = \frac{(N \cdot log_2 N(log_2 N + 1))}{4}$$

No obstante, el algoritmo escogido al estar desarrollado en OpenCL debe seguir el modelo de arquitectura de memoria definida para OpenCL. Esto implica ciertas restricciones en las interfaces de entrada y salida del clasificador, suponiendo una limitación para proceder a su adaptación para el sistema MapReduce sobre el que se quiere implementar. Por ello, se ha optado por realizar una implementación del algoritmo de *bitonic sorting* en SystemC, partiendo de este algoritmo como ejemplo y duplicando su funcionamiento.

4.4. Desarrollo del clasificador

Para poder incluir el clasificador de *bitonic sorting* dentro de la estructura MapReduce, se requiere de una entrada y una salida AXI4-Stream de 8 bit. En la Figura 33 se muestran las interfaces de entrada y salida del clasificador. En lo que se refiere a la transferencia de la clave y el valor para la clasificación, cada uno de ellos será de 8 bit y se irán enviando de forma consecutiva.

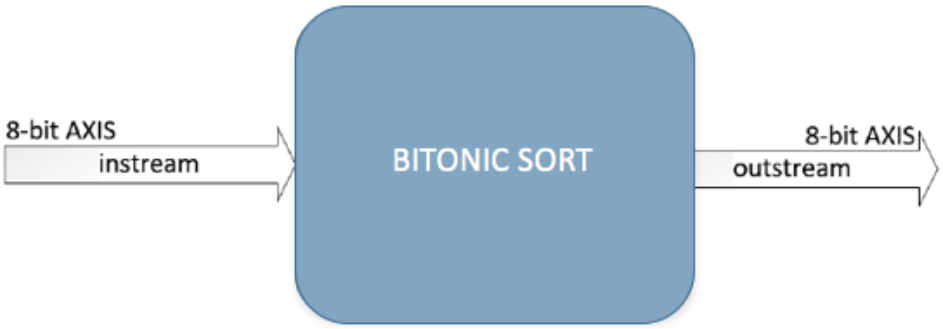

Figura 33: Diseño del clasificador

Para el desarrollo del clasificador se ha hecho uso de señales internas para la interconexión de las funciones que componen el diseño del clasificador. La funcionalidad de estas señales es

comunicar las funciones, sincronizarlas y producir el intercambio de información entre ellas. Las señales implementadas son las siguientes:

- `start_bitonic`: señal de tipo *boolean* que indica cuando un paquete completo ha sido recibido para poder proceder a su clasificación.
- `start_merge`: señal de tipo *boolean* que indica que los datos han sido ordenados según una secuencia bitónica y se puede proceder a la combinación de los datos.
- `end_bitonic`: señal de tipo *boolean* que indica que todos los datos han sido clasificados y, por tanto, el algoritmo de *bitonic sort* ha finalizado.
- `end_send`: señal de tipo *boolean* que indica que todos los datos clasificados han sido enviados correctamente.

También, se ha hecho uso de dos variables para el almacenamiento de los *key-values* a clasificar. Se trata de dos *arrays* (*key*[] y *data*[]) que almacenan 8 *unsigned* bit. El número de posiciones de los *arrays* viene definida en el archivo "param.h", donde mediante la definición de SIZE_LIMIT se define el número de pares *key-value* a clasificar. Inicialmente, para las pruebas y la implementación inicial del clasificador se ha seleccionado como 32.

Para el desarrollo del clasificador se han desarrollado cuatro funciones en SystemC, dos de ellas encargadas de la recepción y el envío de datos, respectivamente, y las otros dos encargadas de la generación de la secuencia bitónica, y la posterior combinación para la clasificación completa de los datos. Estas funciones son las siguientes:

- `void data_in()`. Es la función principal del clasificador, pues es la que recibe todos los datos según el protocolo de comunicaciones AXI4-Stream y da inicio a la clasificación, haciendo uso de la señal *start_bitonic*. Su objetivo es almacenar en variables de *arrays* las claves y los valores a clasificar, manteniendo la relación entre estos. Siempre que sea posible, se encontrará activo para la recepción de datos. En esta implementación únicamente se recibirá un paquete y hasta que este no haya sido enviado una vez clasificado no procederá a recibir más paquetes, cuando la señal *end_send* se encuentre activa a nivel alto. Sin embargo, esta limitación es únicamente temporal.
- `void bitonic_sort()`. Esta función es la principal del algoritmo, pues se encarga de realizar la comparación entre las claves y proceder a la reordenación de los *key-values* en función de una secuencia bitónica. Para ello, se requiere de múltiples bucles que realicen las comparaciones en orden ascendente y descendente y variando la distancia de comparación, tal y como se ha explicado en el apartado 4.2.8. El primer bucle se encarga de determinar el tamaño de la secuencia ascendente y descendente, siendo el mismo valor para ambas, las cuales se intercambian hasta llegar al número de elementos del conjunto

de datos a clasificar. Para determinar el tamaño de esta secuencia se ha usado la variable `size`. El segundo bucle permite determinar los saltos que se deben realizar para pasar a la siguiente secuencia que debe ser clasificada de forma ascendente, una vez se ha realizado una clasificación de una secuencia ascendente y una secuencia descendente. Después se calcula el valor del salto, al cual se ha llamado `stride` y su valor será la mitad de la variable `size`. El tercer bucle hace uso de esta variable para repetir las comparaciones, ascendentes y descendentes, hasta que el tamaño del salto sea 0, disminuyendo a la mitad cada vez que se repite el bucle. El cuarto y quinto bucle permiten realizar las comparaciones de los datos para una secuencia ascendente y una descendente, teniendo en cuenta la separación existente entre ellas (la variable "size") y haciendo uso del valor del salto (la variable "stride"). Una vez se complete la secuencia bitónica se hace uso de la señal *start_merge* para proceder a finalizar la ordenación de los datos. En la Figura 34 se puede ver el código utilizado para implementar esta función.

- `void bitonic_merge()`. Esta función es la encargada de realizar la combinación de los datos tras obtenerse la secuencia bitónica. Realiza la comparación entre las dos partes de la secuencia de datos, tal y como se explicó en el apartado 4.2.8. y finalmente se obtienen todos los *key-values* ordenados de forma correcta. Mediante la señal *end_bitonic* se avisa de que los datos ya pueden ser enviados.

- `void data_out()`. Esta función se encarga del envío de los pares *key-value*, de forma ordenada al receptor. Hace uso del protocolo de comunicaciones AXI4-Stream y tras asegurarse de que todos los datos han sido enviados de forma correcta, procede a poner a nivel alto la señal *end_send*, para avisar de que un nuevo conjunto de datos puede ser recibido para clasificar.

```
void bitonicSort::bitonic_sort() {
  ... // Reset Phase: Signal values
  wait();

  while (true) {
     if (start_bitonic == true) {
        for (size = 2; size < SIZE_LIMIT; size <<= 1) {
           for (k = 0; k < SIZE_LIMIT; k += size*2) {
              stride = size/2;
              while (stride > 0) {
                 for (j = 0; j < size; j += stride*2) {
                    for (i = j + k, m = 0; m < stride; m++, i++){
                       if ((key[i] > key[i+stride]) == true) {
                          temp1 = key[i];
                          key[i] = key[i+stride];
                          key[i+stride] = temp1;
                          temp2 = data[i];
                          data[i] = data[i+stride];
                          data[i+stride] = temp2;
                       }
                       l = i + size;
                       if ((key[l] > key[l+stride]) == false) {
                          temp1 = key[l];
                          key[l] = key[l+stride];
                          key[l+stride] = temp1;
                          temp2 = data[l];
                          data[l] = data[l+stride];
                          data[l+stride] = temp2;
                       }
                    }
                 }
                 wait();
                 stride >>= 1;
              }
           }
        }
        start_merge = true;
        wait();
     } else {
        start_merge = false;
        wait();
     }
  }
```

Figura 34: Función bitonic_sort() del algoritmo bitonic sort en SystemC

En la Figura 35 se muestra el diseño completo del clasificador, incluyendo las interfaces de entrada y salida, las funciones y las señales internas que permiten la comunicación entre estas.

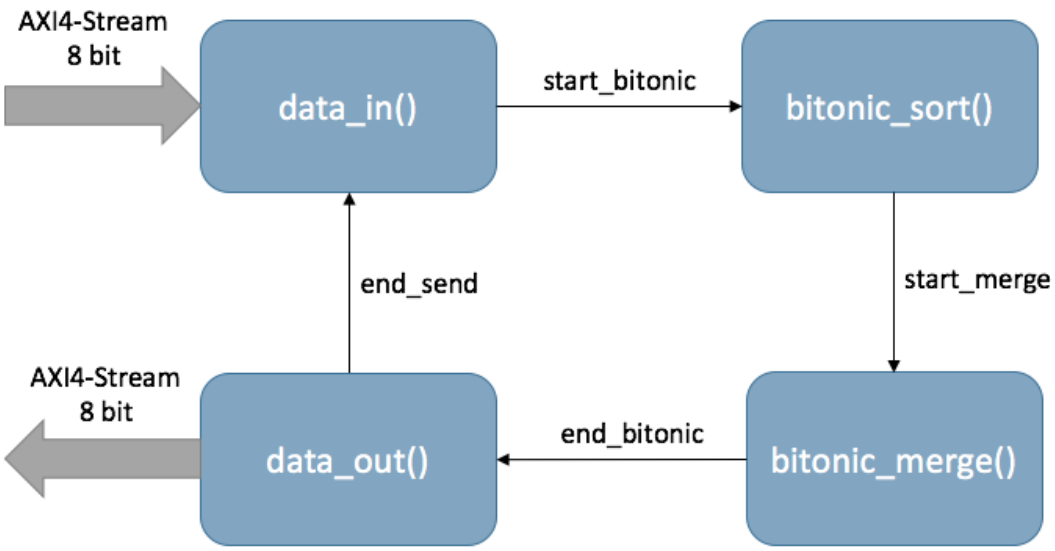

Figura 35: Esquemático del clasificador

4.4.1. Simulación en SystemC

Para la simulación del clasificador desarrollado en SystemC es necesaria la elaboración de un *testbench* que permita verificar diferentes casos. Se parte de un archivo denominado `sc_main_bitonicSort` para realizar las interconexiones entre los puertos de entrada y salida necesarios entre el clasificador y el *testbench*. El *testbench* desarrollado se encarga de simular el envío de datos y la recepción de estos una vez han sido clasificados. También se encarga de comprobar si los datos recibidos se encuentran ordenados correctamente, y en caso contrario muestra un mensaje de error. En la Figura 36 se muestra un ejemplo del envío de un paquete de 64 palabras de 8 bit, siendo un total de 32 pares *key-value*. Tras realizarse la clasificación se muestra la comparación entre los datos recibidos y los datos que fueron enviados inicialmente, comprobando que estos datos no han modificado su valor, pero si su posición, manteniendo siempre el par *key-value*.

```
Test start.
Sending message to be forwarded.
Message from - key  =  116
Message from - value  =  7
Last from = 0
Message from - key  =  63
Message from - value  =  0
Last from = 0
…
Message from - key  =  76
Message from - value  =  90
Last from = 0
Message from - key  =  77
Message from - value  =  239
Last from = 1

Original message - key 1 = 116
Message from Classifier - key 1 = 19
Original message - data 1 = 7
Message from Classifier - data 1 = 154
Original message - key 2 = 63
Message from Classifier - key 2 = 27
Original message - data 2 = 0
Message from Classifier - data 2 = 20
Original message - key 3 = 60
Message from Classifier - key 3 = 32
Original message - data 3 = 6
Message from Classifier - data 3 = 25
…
Original message - key 31 = 76
Message from Classifier - key 31 = 230
Original message - data 31 = 90
Message from Classifier - data 31 = 16
Original message - key 32 = 77
Message from Classifier - key 32 = 239
Original message - data 32 = 239
Message from Classifier - data 32 = 219
OK
Test 1 Finished OK
```

Figura 36: Ejemplo del envío, clasificación y recepción de un paquete

4.4.2. Obtención del bloque IP

A continuación, se procede a la obtención del bloque IP del clasificador *bitonic sort* para su integración en la FPGA. Para ello, se hace uso de Vivado HLS, donde se realiza la síntesis del clasificador y la exportación RTL. Los resultados de utilización estimados tras la realización de la síntesis son un periodo de reloj estimado de 6,25 ns y el uso de 2 BRAMs, 869 FFs y 1779 LUTs. De esta manera, se obtiene directamente un bloque IP válido para ser utilizado en el IDE de Vivado Design Suite. Se procede a la inclusión del bloque IP dentro del "IP Repository" en Vivado, tal y como se muestra en la Figura 37. También es necesario realizar la customización del IP para adaptarlo al sistema, haciendo uso de la generación de productos de salida que se puede observar en la Figura 38. De esta manera se obtendrá el bloque IP completamente adaptado para poder ser introducirlo en el diseño de bloques e implementado sobre la FPGA. Finalmente, en la Figura 39 se muestra el bloque IP del clasificador generado, con las entradas y salidas correspondientes, AXI4-Stream de 8 bit.

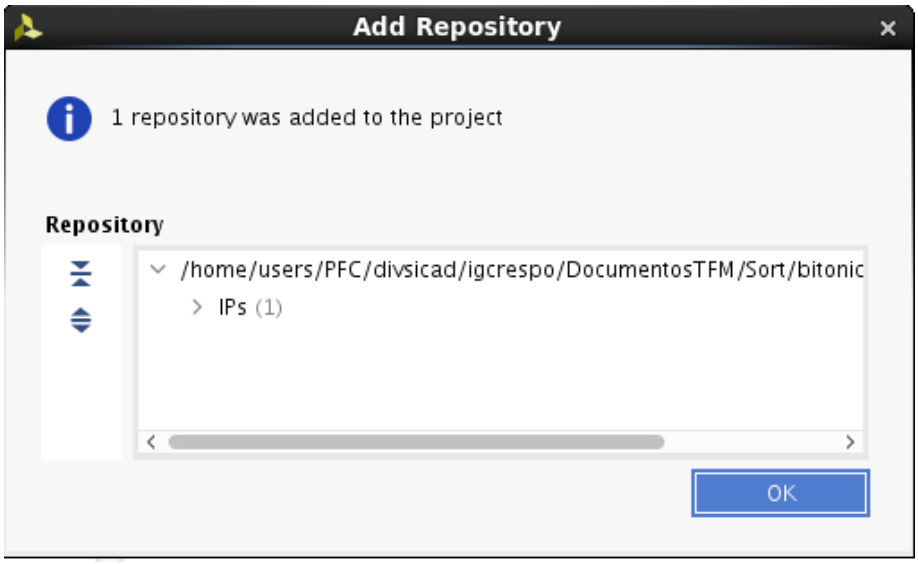

Figura 37: Inclusión del bloque IP clasificador en Vivado IP Repository

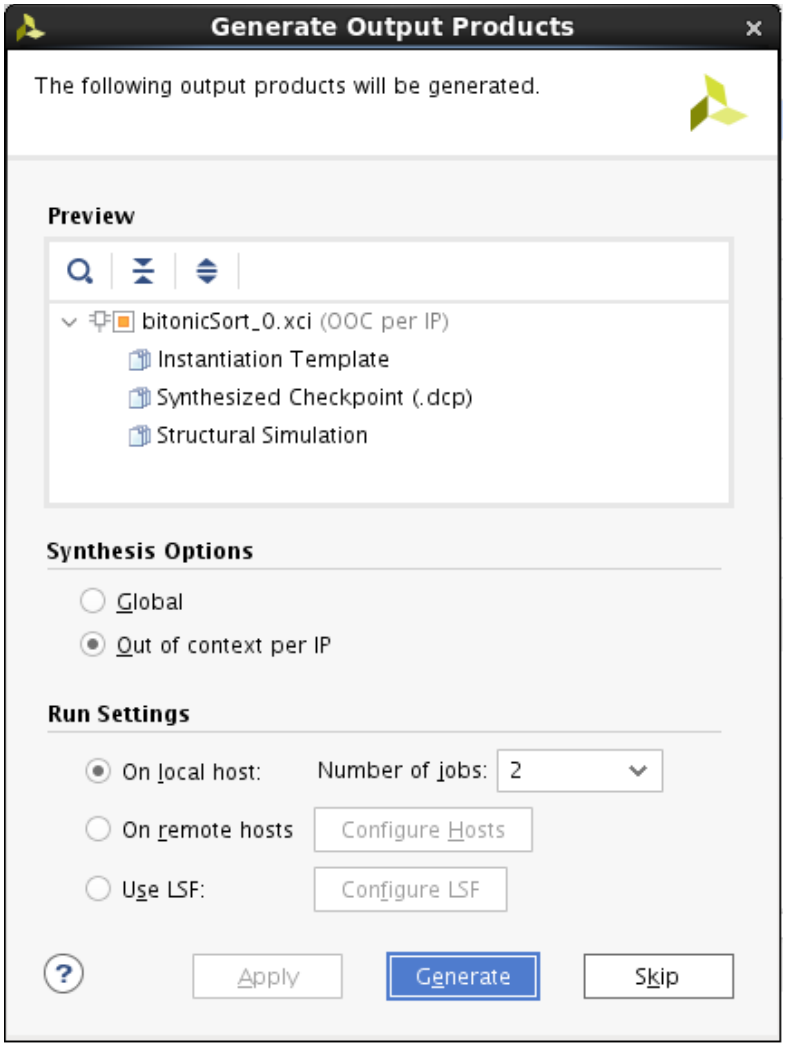

Figura 38: Customización del bloque IP clasificador

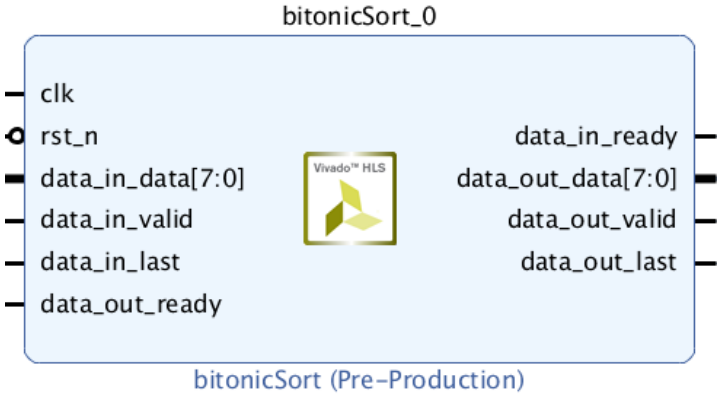

Figura 39: Bloque IP del clasificador

4.4.3. Análisis temporal y de recursos del clasificador bitonic sort

En este apartado se realizará un análisis temporal y de recursos del clasificador *bitonic sort*, de esta manera se podrá obtener el periodo de funcionamiento del mismo y el número de comparadores necesarios para su implementación. Para ello es necesario tener en cuenta ciertos valores iniciales. Antes que nada, se debe calcular el número de comparadores utilizados y la profundidad de la red de *sorting* haciendo uso de las ecuaciones mostradas en el apartado 4.3. Como valor inicial de N, que es el número de elementos del clasificador, se ha seleccionado 32 pares *key-value* al igual que en la simulación SystemC.

$$Profundidad\ de\ la\ red\ de\ sorting \rightarrow D(32) = \frac{\left(log_2 32(log_2 32 + 1)\right)}{2} = 3{,}667 \cong 4$$

$$Número\ de\ comparadores \rightarrow C(2) = \frac{\left(32 \cdot log_2 32(log_2 32 + 1)\right)}{4} = 58{,}677 \cong 59$$

Se obtiene como resultado un valor de 4 para la profundidad de la red de *sorting* y se requiere de 59 comparadores para la implementación. No obstante, se necesitan 5 etapas para la realización de la comparación completa, siendo la primera etapa de un ciclo de reloj y cada etapa posterior se le añade un ciclo de reloj extra. Como consecuencia, se obtiene que el total de ciclos de reloj necesario para la clasificación de 32 pares *key-value* es de 15 ciclos de reloj. Si por ejemplo, la frecuencia de funcionamiento del sistema es de 100 MHz, se requerirá de 150 ns para la finalización de la clasificación del conjunto de datos. No obstante, se debe sumar a este tiempo la recepción y el envío de los datos, pues una limitación de *bitonic sort* es la necesidad de disponer del *array* de datos completo antes de proceder a su ordenación. Con el sistema diseñado, se requiere de 64 ciclos de reloj, tanto en la recepción como en el envío de los datos, pues las *keys* y los *values* son enviados de forma independiente. Por ello, se puede decir que se requiere de un total de 1,43 µs para recibir el *array* de datos, realizar la clasificación y devolver el *array* ordenado. En la Figura 40 se muestra un cronograma de la transmisión de datos del bloque *bitonic sort*.

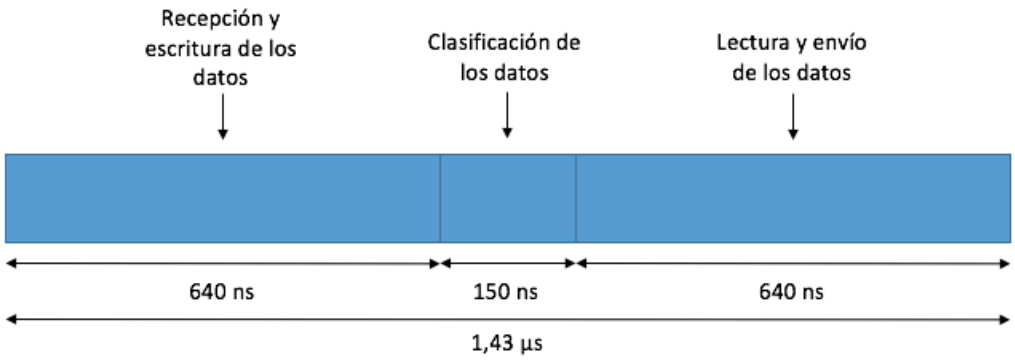

Figura 40: Cronograma de transmisión de datos del clasificador bitonic sort (f=100 MHz)

4.5. Conclusiones

Existe una gran cantidad de clasificadores que pueden ser implementados y utilizados en aplicaciones Big Data y sobre FPGAs. No obstante, la selección del clasificador de *bitonic sort*, permite potenciar las ventajas que ofrecen las FPGAs, suponiendo una mejora para la implementación en este tipo de sistemas, debido a la paralelización que ofrece. Se trata de un algoritmo que requiere de recursos bastante reducidos, lo que supone que puede ser implementado en diferentes aplicaciones sin mayores preocupaciones, o puede permitir ser ampliado para poder gestionar mayores cantidades de datos.

Capítulo 5. Implementación y verificación

Tras la obtención de forma independiente de dos sistemas funcionales, uno que implementa el protocolo de comunicaciones PCI Express y otro que implementa el algoritmo de clasificación *bitonic sort*, es necesario proceder a la integración de ambos en un único sistema, para cumplir con los objetivos de este proyecto. Por ello, se mostrarán a continuación los pasos seguidos para la integración e implementación del sistema sobre la plataforma seleccionado, que es este trabajo se trata de la placa de desarrollo ZC706 de Xilinx. Inicialmente, se describirá el diagrama de bloques diseñado y la obtención del *bitstream* para la verificación del sistema en el dispositivo.

Para la validación del sistema sobre la plataforma se requerirá del uso de la herramienta Vivado Design Suite, que permite la comprobación de los flujos de datos del sistema a través de los ILAs que se integren en el diseño gracias al configurador de hardware. También se hará uso del *driver* de RIFFA para permitir el envío y la recepción de datos por parte del *host* CPU.

5.1. Integración e implementación del sistema final

Inicialmente, es necesario realizar la obtención de diferentes bloque IP respecto al proyecto utilizado en la implementación de RIFFA. En este caso, haremos uso de la versión que soporta un ancho de interfaz AXI de 128 bit. En la Figura 41 se puede observar el bloque IP de RIFFA obtenido a partir del código proporcionado por "riffa_wrapper_zc706.v" y los ficheros *include* necesarios para su correcto funcionamiento. Como se puede observar, dispone de todas las entradas y salidas necesarias para realizar la conexión con el bloque PCIe de Xilinx, permitiendo el funcionamiento del bus de comunicaciones PCI Express. No obstante, también se ha realizado la creación de un bloque IP partiendo del archivo "PCIeGen2x4lf128.v" proporcionado por RIFFA. De esta manera, como se observa en la Figura 42 se obtiene el bloque PCIe de Xilinx con las modificaciones necesarias para poder ser integrado con RIFFA sin dificultades. También cabe destacar la posibilidad de realizar ciertas variaciones en la configuración de RIFFA a partir de su IP, como son el número de canales, el ancho de datos, el tamaño máximo de bytes del *payload* y la ID de la FPGA (Figura 43).

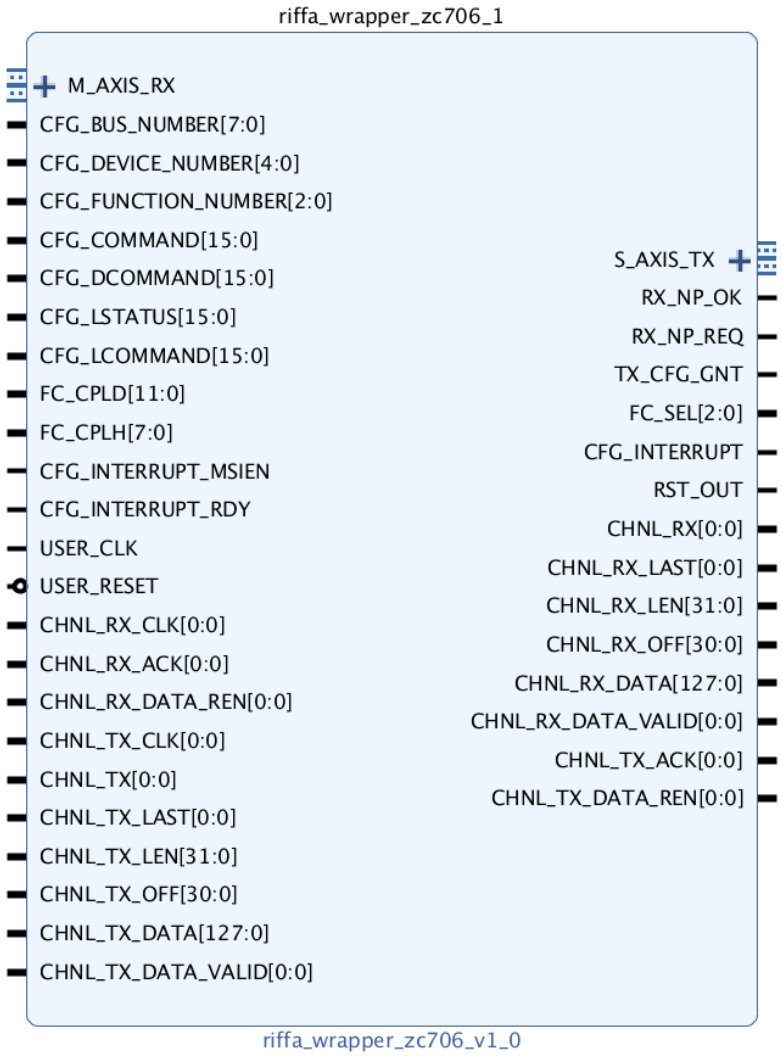

Figura 41: Bloque IP de RIFFA

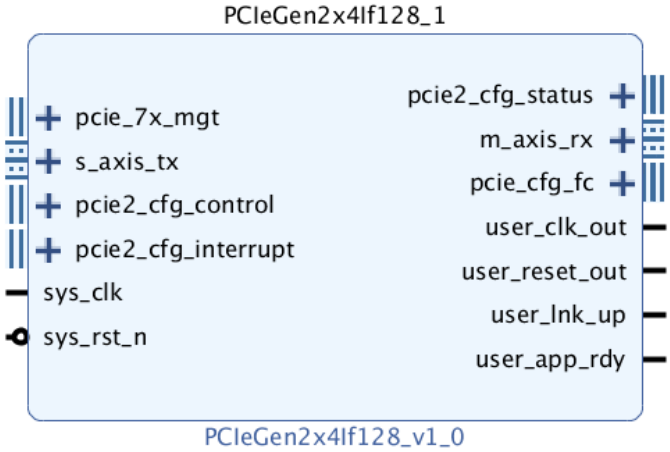

Figura 42: Bloque IP para RIFFA del bloque PCIe de Xilinx

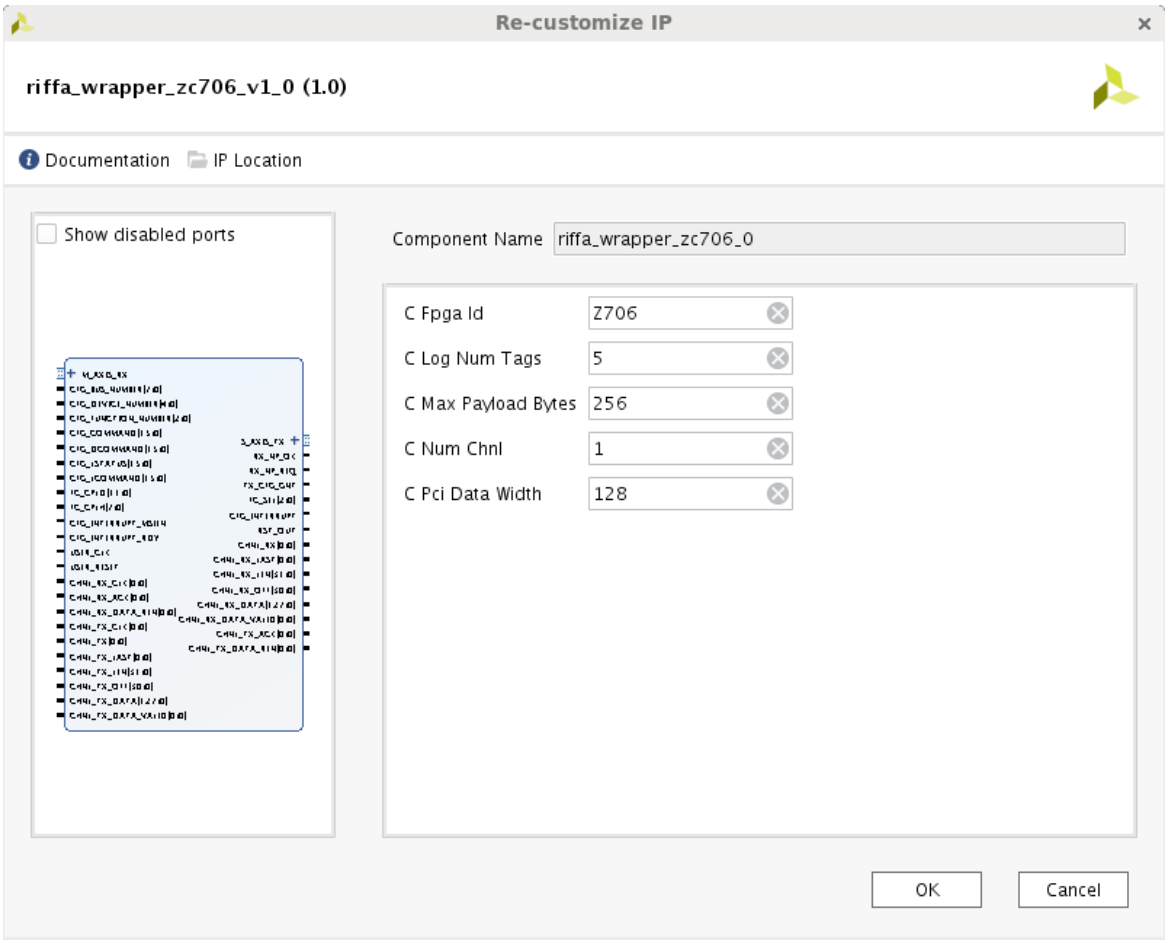

Figura 43: Configuración del bloque IP de RIFFA

Una vez obtenido los bloque IP correspondientes a RIFFA se debe proceder a la creación de un nuevo proyecto en Vivado Design Suite para realizar un nuevo diagrama de bloques, donde no solo se incluirá este nuevo bloque, sino también el bloque del clasificador de *bitonic sort* desarrollado en el Capítulo 4. Para ello, es necesaria la adición de los bloques IP en el "IP Repository" de Vivado.

A su vez, se ha realizado la adición de ILAs en el diseño, para poder monitorizar los datos a la entrada y salida del bloque IP *bitonic sort*. De esta manera aseguraremos su correcto funcionamiento. ILA es un analizador lógico integrado que permite observar el flujo de datos del diseño, pudiendo seleccionar el tipo de protocolo utilizado, que en nuestro caso es AXI4-Stream. Una vez finalizado el diagrama de bloques diseñado para la implementación del sistema, se puede observar en la Figura 44 el resultado final del diseño, habiendo realizado la validación del diseño.

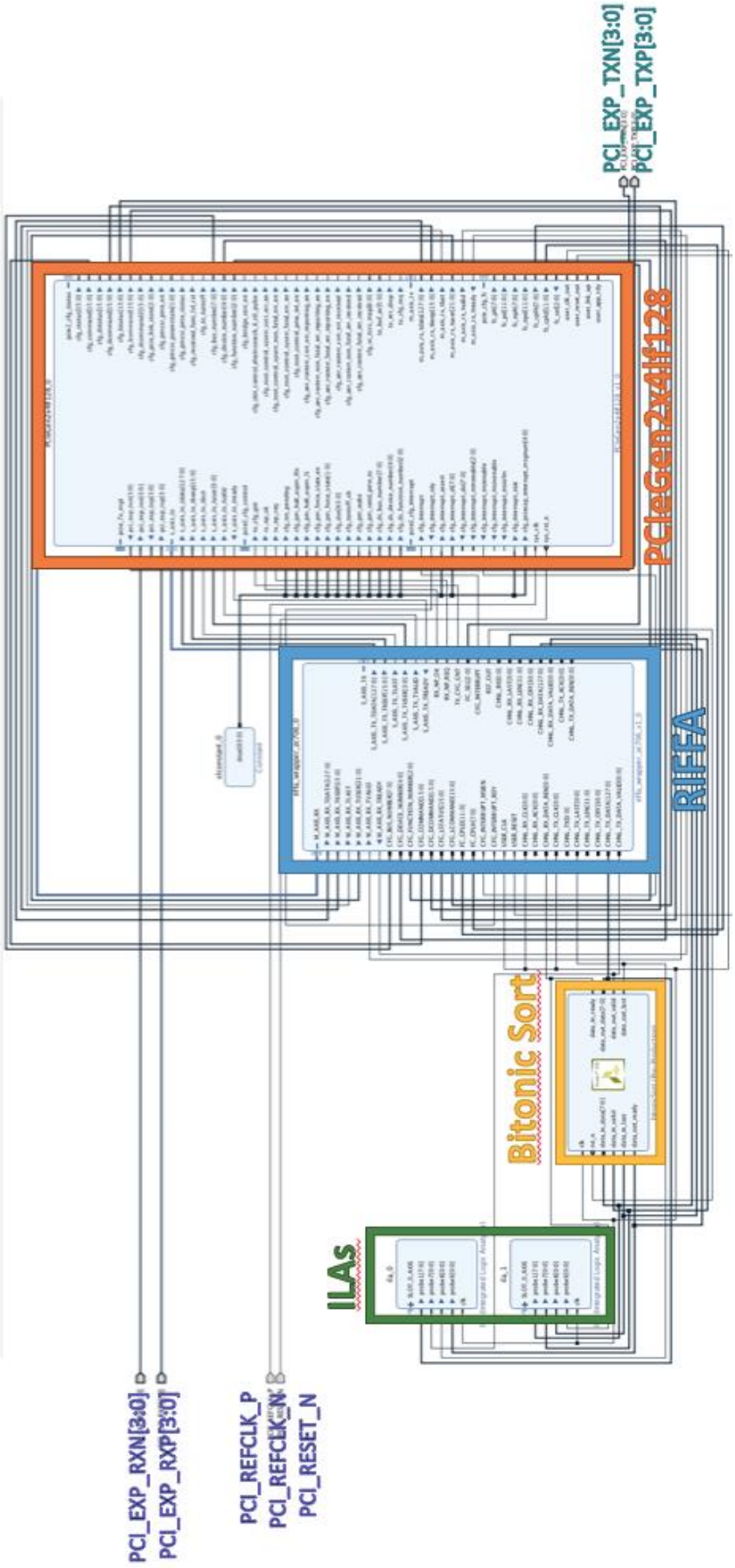

Figura 44: Diagrama de bloques del sistema completo

Para la implementación en la plataforma del sistema realizado será necesaria la obtención del *bitstream* para la programación del dispositivo, pero previamente se deben realizar la síntesis y la implementación y la exportación del hardware. Los pasos seguidos para la obtención del *bitstream* son los mismos que fueron explicados en el Capítulo 4 para la obtención del sistema con el clasificador *bitonic sort* para su verificación sobre la plataforma. Se deben generar los productos de salida del diagrama, crear el *wrapper* HDL del diseño, ejecutar la síntesis y realizar la implementación. Una vez obtenidos los resultados, es posible observar en la Figura 45 el *layout* de la plataforma con los recursos utilizados. Se observan remarcados en amarillo los recursos utilizados por RIFFA, en verde los recursos utilizados por el clasificador y en naranja los recursos utilizados por el bloque PCIe de Xilinx.

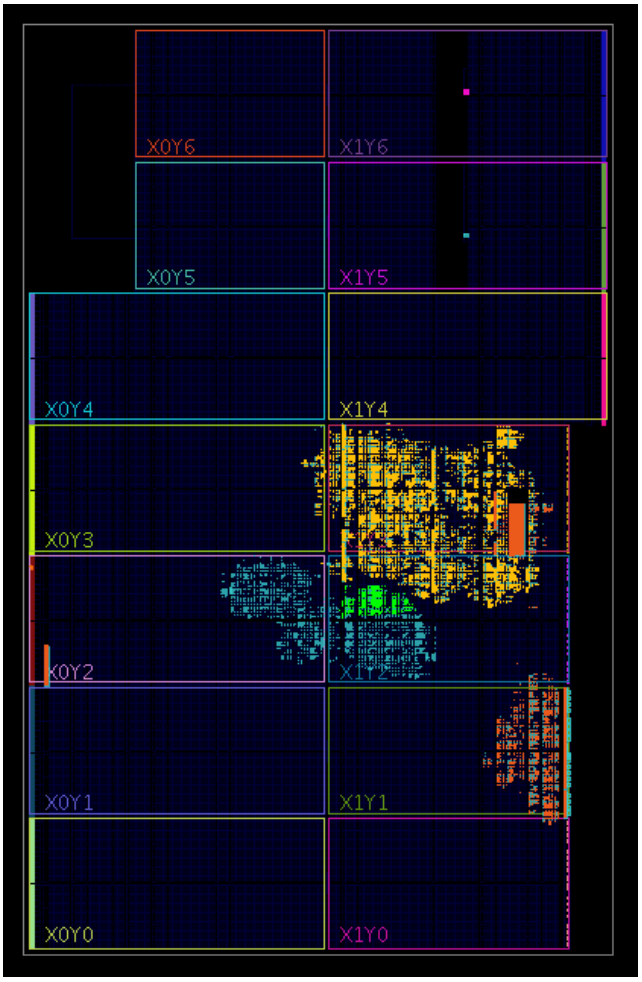

Figura 45: Layout del sistema completo sobre la plataforma

En la Figura 46 se presentan los resultados del análisis temporal del diseño, el cual muestra un *slack* positivo de 9,029 ns, teniendo en cuenta que se ha hecho uso de una frecuencia de funcionamiento de 100 MHz. Gracias a este análisis es posible comprobar el correcto funcionamiento temporal según las restricciones temporales impuestas. Sería posible aumentar la frecuencia de

funcionamiento, pero se debe tener en cuenta las limitaciones que impone el bloque PCIe de Xilinx, el cual puede funcionar a 100 MHz, 125 MHz o 250 MHz.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	9.029 ns	Worst Hold Slack (WHS):	0.044 ns	Worst Pulse Width Slack (WPWS):	4.358 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1059	Total Number of Endpoints:	1059	Total Number of Endpoints:	527

All user specified timing constraints are met.

Figura 46: Resultados de la temporización del sistema completo

También se puede observar en la Figura 47 un gráfico que indica los porcentajes de recursos utilizados en la implementación del sistema. Las LUTs utilizadas ascienden a 9.559 (4,37 %) En la Tabla 8 se muestran los resultados de recursos utilizados, realizando una comparativa entre los diferentes bloques que integran el sistema, haciendo referencia a aquellos que son de especial interés: RIFFA, el clasificador *bitonic sort* y la implementación del bloque PCIe de Xilinx.

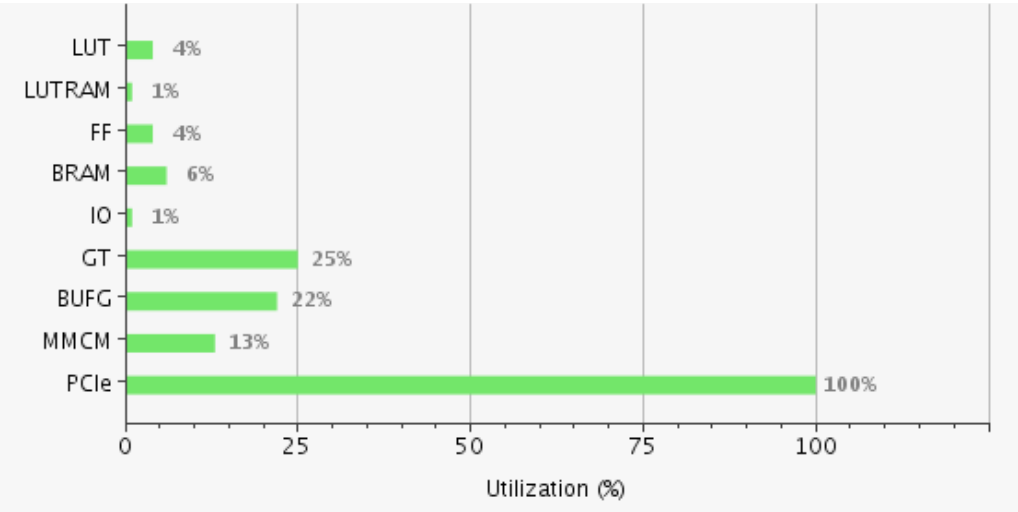

Figura 47: Gráfico de utilización de recursos del sistema completo

Tabla 8: Utilización de recursos del sistema y comparativa

	Sistema completo	RIFFA *wrapper*	*Bitonic sort*	PCIeGen2x4lf128
LUT	9.559 (4,37 %)	5.172 (2,37 %)	522 (0,24 %)	1.542 (0,71 %)
LUTRAM	467 (0,66 %)	187 (0,27 %)	0 (0,00 %)	32 (0,05 %)
FF	15.547 (3,56 %)	9.139 (2,09 %)	566 (0,13 %)	2.203 (0,50 %)
BRAM	30 (5,50 %)	24 (4,40 %)	1 (0,18 %)	4 (0,73 %)
IO	5 (1,38 %)	0 (0,00 %)	0 (0,00 %)	0 (0,00 %)
GT	4 (25,00 %)	0 (0,00 %)	0 (0,00 %)	4 (25,00 %)
BUFG	7 (21,88 %)	0 (0,00 %)	0 (0,00 %)	6 (18,75 %)
MMCM	1 (12,50 %)	0 (0,00 %)	0 (0,00 %)	1 (12,50 %)
PCIe	1 (100 %)	0 (0,00 %)	0 (0,00 %)	1 (100 %)

Para finalizar el análisis del diseño del sistema realizado, es importante destacar el análisis de potencia consumida. En la Figura 48 se muestra un gráfico con el consumo de potencia de la plataforma, el cual es igual a 1,672 W. Esto implica un bajo consumo de potencia permitiendo altas tasas de transmisión de datos, incluyendo la clasificación de estos. El consumo de RIFFA es de 55 mW, el consumo del bloque PCIe de Xilinx adaptado es de 1,378 W y el consumo del clasificador *bitonic sort* es de 6 mW. Tanto la implementación de RIFFA como del clasificador, suman un consumo prácticamente despreciable, siendo el elemento característico en la gestión de potencia del sistema el bloque PCIe de Xilinx.

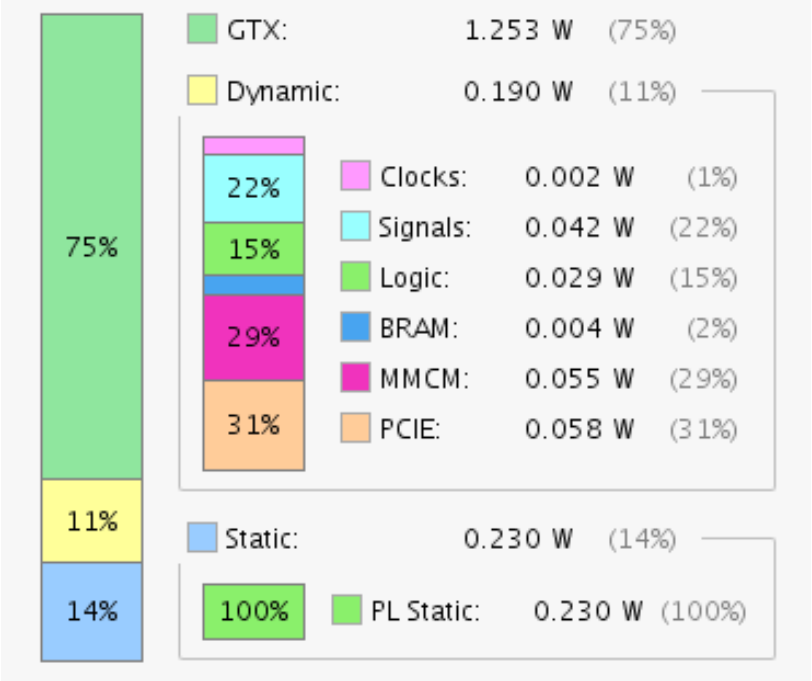

Figura 48: Consumo de potencia del sistema completo

5.2. Verificación y validación

Tras la obtención de la implementación del sistema ya es posible realizar la validación. Por ello, inicialmente se procede a realizar la validación de forma independiente, primero realizando la validación del bus de comunicaciones PCI Express haciendo uso del *driver* de RIFFA. A continuación, se procederá a la validación del clasificador *bitonic sort* mediante la creación de un diagrama de bloques en Vivado Design Suite, para comprobar su funcionamiento de forma independiente. Para ello, se hará uso de un DMA para gestionar el envío y la recepción de datos, y de la herramienta SDK para la programación y gestión del mismo.

5.2.1. Verificación y validación de RIFFA

Antes de proceder a la verificación y validación de RIFFA sobre la placa de desarrollo es necesario realizar la instalación del *driver* de RIFFA en el sistema operativo que utiliza el *host*

controlador. En este caso, se ha hecho uso del sistema operativo Red Hat Enterprise Linux 6, por lo que se ha utilizado el *driver* de Linux proporcionado por RIFFA. Gracias al uso de un *makefile* es posible realizar la instalación de las cabeceras *kernel* de Linux necesarias, así como del propio *driver* y las librerías nativas de C/C++ [31]. Para ello, se han seguido los siguientes pasos:

1. Abrir un terminal y acceder al directorio "…/riffa_2.2.2/source/driver/linux".
2. Instalar las cabeceras del *kernel* haciendo uso de la línea de comando "sudo make setup".
3. Compilar el *driver* y la librería de C/C++ con la línea de comando "make debug", pudiendo observar mientras se ejecuta mensajes que muestran si se produce algún error o se realizan las acciones de forma correcta. También es posible realizar únicamente "make" y evitar sobrecargar de mensajes el sistema de *log* de RedHat.
4. Instalar el *driver* y la librería previamente compiladas. Para ello, se hace uso de la línea de comando "sudo make install".

Una vez se ha realizado la instalación del *driver* de RIFFA correctamente, es necesario hacer uso de un diseño de bloques para la comprobación de la plataforma. Para ello, se ha partido del diseño de bloques obtenido en el apartado 3.4 y se ha procedido a añadir el bloque "Chnl_Tester", obteniendo el diagrama mostrado en la Figura 49. "Chnl_Tester" es un simple bloque IP que recibe los datos y procede a su envío sin realizar ninguna modificación, en función de los canales de PCI Express. Una vez obtenido el *bitstream* de este diseño se realiza la programación de la FPGA y se procede a la comprobación del funcionamiento del protocolo de comunicaciones PCI Express. Para ello, es necesario con la FPGA programada comprobar que la conexión mediante el cable PCI Express es correcta. A continuación, se procede al reinicio del sistema que incluye el *host* controlador del sistema. Este reinicio es necesario, ya que sin él la FPGA no es capaz de detectar el *driver* de RIFFA.

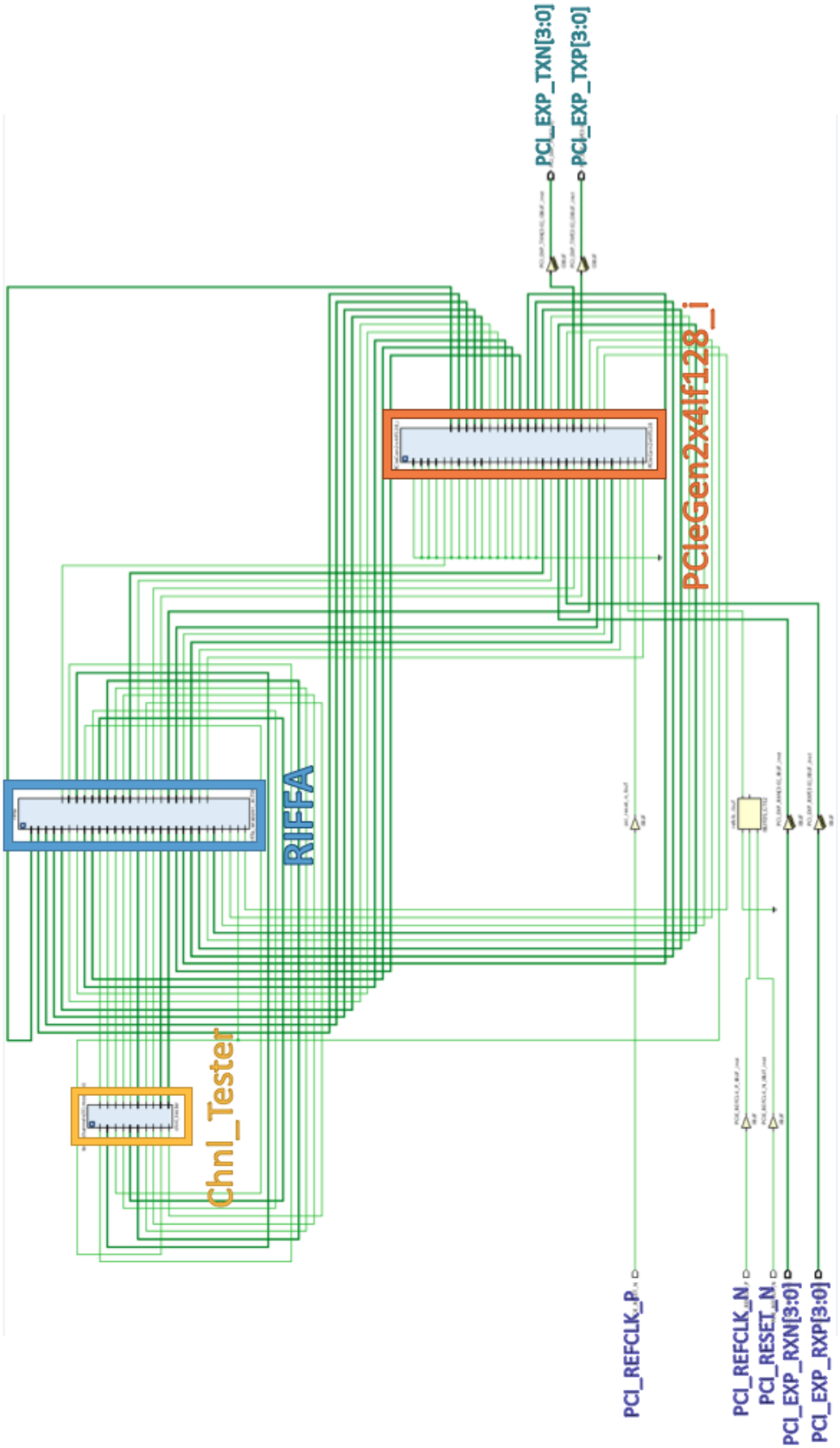

Figura 49: Diagrama de la plataforma con RIFFA para la validación

Tras reiniciar el sistema ya es posible acceder a las aplicaciones de prueba para comprobar el correcto funcionamiento de RIFFA, y por tanto, el bus de comunicaciones PCI Express. Para ello, a través de un terminal se accede al directorio "…/riffa_2.2.2/source/c_c++/Linux/x64/test_apps". Se hace uso del *testutil* que se encuentra dentro del archivo "testutil.c" para proceder a la comprobación de la conexión y la transferencia de información. Inicialmente, se hace uso del comando básico para mostrar los valores por los que está definida la FPGA y la conexión "testutil 0". De esta manera, se obtiene que solo existe un dispositivo conectado al bus de comunicaciones, la FPGA con un ID de 0. También se observa que existe un único canal y los valores de ID del vendedor y el dispositivo, tal y como se muestra en la Figura 50.

```
igcrespo@vlsiws20.iuma.ulpgc.es% cd test_apps/
/home/users/PFC/divsicad/igcrespo/DocumentosTFM/RIFFA/riffa_2.2.2-vivado2018.1/s
ource/c_c++/linux/x64/test_apps
igcrespo@vlsiws20.iuma.ulpgc.es% testutil 0
Number of devices: 1
0: id:0
0: num_chnls:1
0: name:0000:03:00.00
0: vendor id:10EE
0: device id:7024
```

Figura 50: Configuración de la FPGA respecto al bus de comunicaciones PCI Express

Para la ejecución de *testutil* se requiere de mínimo 4 palabras para la realización de los diferentes tests incluidos en él. Inicialmente se procede a realizar el *test* más simple, la opción 2, que se encarga de enviar y recibir datos de forma reiterativa, haciendo un total de 100 tests, pues es el valor definido al inicio, pudiendo variarse en cualquier momento. En la Figura 51 se muestra un ejemplo de este test, donde para cada una de las reiteraciones del mismo se realiza el envío de cuatro palabras, tal y como se ha indicado al introducir la línea de comando "testutil 2 0 0 4", siendo 2 la opción de test escogida, el primer 0 la ID de la FPGA, el segundo 0 el canal y el 4 el número de palabras a transferir. Tras cada test es posible comprobar el ancho de banda enviado y recibido, habiéndose comprobado que la información es correcta y siendo siempre algo mayor el recibido, debido a las adiciones en la transferencia.

La opción 3 de *testutil* realiza el envío y la recepción de datos, realizando reiteraciones del mismo hasta 1023. Al igual que en la opción 2, el mínimo de palabras a enviar es cuatro para poder funcionar de forma correcta. La particularidad de este test es que muestra la dirección desde la que se envían los datos a la FPGA. En la Figura 52 se muestra el funcionamiento de este test como resultado de la línea de comando introducida "testutil 3 0 0 4", realizando el envío de cuatro palabras por el primer canal, el único existente. Al igual que en la opción anterior, el valor 3 determinar la opción de test, el primer 0 la ID de la FPGA, el segundo 0 el canal y el 4 el número de palabras a transferir.

```
igcrespo@vlsiws20.iuma.ulpgc.es% testutil 2 0 0 4
Running bandwidth test from 4 up to 4 words
Test 0: words sent: 4
Test 0: words recv: 4
Test 1: words sent: 4
Test 1: words recv: 4
send bw: 0.458294
recv bw: 0.612486
Test 2: words sent: 4
Test 2: words recv: 4
send bw: 0.550723
recv bw: 0.668735
Test 3: words sent: 4
Test 3: words recv: 4
send bw: 0.590414
recv bw: 0.697191
Test 4: words sent: 4
Test 4: words recv: 4
send bw: 0.618264
recv bw: 0.661980
...
Test 96: words sent: 4
Test 96: words recv: 4
send bw: 0.728178
recv bw: 0.936229
Test 97: words sent: 4
Test 97: words recv: 4
send bw: 0.728178
recv bw: 0.936229
Test 98: words sent: 4
Test 98: words recv: 4
send bw: 0.728178
recv bw: 0.885622
Test 99: words sent: 4
Test 99: words recv: 4
send bw: 0.728178
recv bw: 0.885622
Test 100: words sent: 4
Test 100: words recv: 4
send bw: 0.728178
recv bw: 0.885622
```

Figura 51: Ejemplo de funcionamiento del testutil con la opción 2 de RIFFA

```
...
Test 1019: words sent from address 0x19a5fec: 4
Test 1019: words recv: 4
send bw: 0.885622
recv bw: 1.456356
Test 1020: words sent from address 0x19a5ff0: 4
Test 1020: words recv: 4
send bw: 0.799220
recv bw: 1.456356
Test 1021: words sent from address 0x19a5ff4: 4
Test 1021: words recv: 4
send bw: 0.840205
recv bw: 1.337469
Test 1022: words sent from address 0x19a5ff8: 4
Test 1022: words recv: 4
send bw: 0.840205
recv bw: 1.337469
Test 1023: words sent from address 0x19a5ffc: 4
Test 1023: words recv: 4
send bw: 0.840205
recv bw: 1.337469
```

Figura 52: Ejemplo de funcionamiento del testutil con la opción 3 de RIFFA

La opción 4 de *testutil* realiza el envío y la recepción de datos, realizando interaciones hasta 1023, pudiendo variar este número en cualquier momento. Si implementación es similar a las opciones 2 y 3, sin embargo muestra tanto la dirección de envío como la de recepción de datos. El ejemplo mostrado en la Figura 53 es el resultado de la línea de comando “testutil 4 0 0 4”, siendo 4 la opción de test, el primer 0 el ID de la FPGA, el segundo 0 el canal y el 4 el número de palabras a enviar y recibir en cada test.

```
...
Test 1019: words sent: 4 (Address 0xae5000)
Test 1019: words recv: 4 (Address 0xae8fec)
send bw: 0.885622
recv bw: 1.236528
Test 1020: words sent: 4 (Address 0xae5000)
Test 1020: words recv: 4 (Address 0xae8ff0)
send bw: 0.885622
recv bw: 1.236528
Test 1021: words sent: 4 (Address 0xae5000)
Test 1021: words recv: 4 (Address 0xae8ff4)
send bw: 0.840205
recv bw: 1.149754
Test 1022: words sent: 4 (Address 0xae5000)
Test 1022: words recv: 4 (Address 0xae8ff8)
send bw: 0.949797
recv bw: 1.236528
Test 1023: words sent: 4 (Address 0xae5000)
Test 1023: words recv: 4 (Address 0xae8ffc)
send bw: 0.936229
recv bw: 1.149754
```

Figura 53: Ejemplo de funcionamiento del testutil con la opción 4 de RIFFA

Para la opción 5 de *testutil* se procede a realizar el envío y recepción de datos. En este caso, se permite seleccionar sobre el propio terminal el número de iteraciones del test. En la Figura 54 se ha realizado un ejemplo de la opción 5 al introducir en el terminal la línea de comando "testutil 5 0 0 0 6 6", siendo 5 la opción de test, el primer 0 la ID del FPGA, el segundo 0 el canal, el tercer 0 el *offset*, el primer 6 el número de palabras a transferir y el segundo 6 el número de iteraciones.

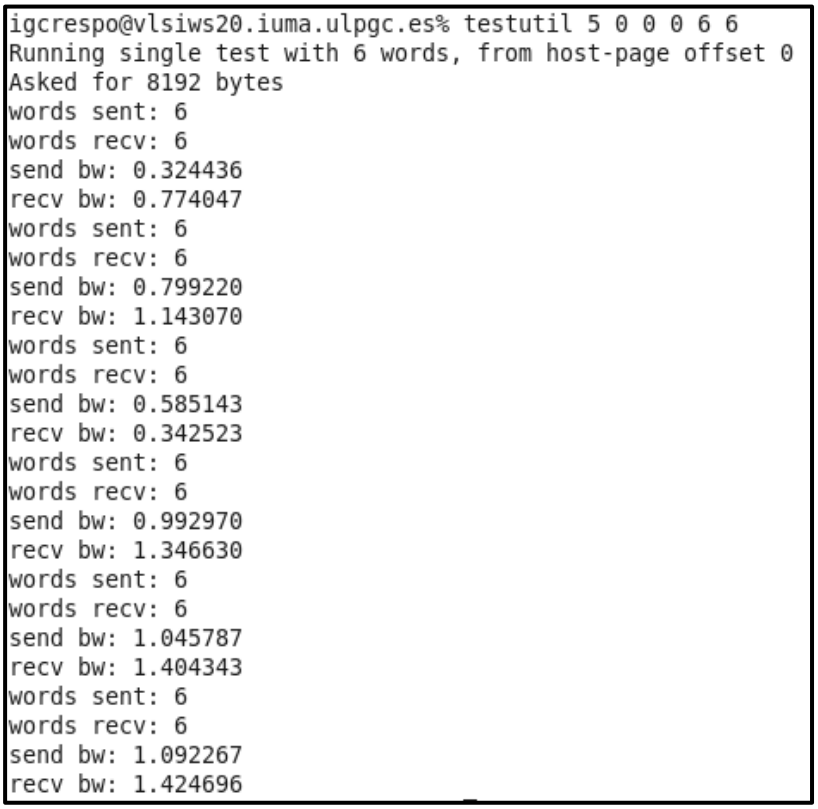

Figura 54: Ejemplo de funcionamiento del testutil con la opción 5 de RIFFA

Finalmente, la última opción de *testutil*, la opción 6, permite realizar el envío y recepción de datos de forma similar a la opción 5, pero mostrando las direcciones de envío y recepción de cada uno de los paquetes. La Figura 55 muestra un ejemplo de esta opción al introducir la línea de comando "testutil 6 0 0 0 6 3", funcionando las declaraciones al igual que en la opción 5. El 6 indica el tipo de test, el primer 0 la ID de la FPGA, el segundo 0 el canal, el tercer 0 el *offset*, el 6 el número de palabras a transmitir y 3 el número de iteraciones. Se requiere de al menos un mínimo de cuatro palabras para poder realizar el test.

```
igcrespo@vlsiws20.iuma.ulpgc.es% testutil 6 0 0 0 6 3
Running single test with 6 words, to host-page offset 0
Asked for 8192 bytes
test 0: words sent: 6
test 0: words recv: 6 (Address 0x15d8000 0x15d8018)
send bw: 0.599415
recv bw: 0.509347
test 1: words sent: 6
test 1: words recv: 6 (Address 0x15d8000 0x15d8018)
send bw: 0.862316
recv bw: 1.260308
test 2: words sent: 6
test 2: words recv: 6 (Address 0x15d8000 0x15d8018)
send bw: 0.854817
recv bw: 1.003102
```

Figura 55: Ejemplo de funcionamiento del testutil con la opción 6 de RIFFA

5.2.2. Implementación y verificación del clasificador

A continuación, se procede a la verificación del clasificador *bitonic sort*. Para poder realizar la implementación del bloque IP sobre la FPGA es necesaria la realización del diagrama de bloques. Para ello, es necesario realizar la interconexión entre los diferentes bloques IP que formarán nuestro sistema. Inicialmente, como elemento principal se añade al diagrama de bloques el bloque IP del PS de Zynq, "ZYNQ7 Processing System", realizando ciertos cambios en su configuración. La frecuencia de reloj de FCLK_CLK0 se selecciona a 150 MHz, ya que el DMA de Xilinx no puede funcionar a frecuencias superiores. Esta es una limitación únicamente temporal, para poder realizar la verificación del clasificador sobre la FPGA. También se activa la interfaz "S AXI HPO Interface", que permite la interconexión con el DMA. A continuación, se procede a realizar la inclusión del bloque IP correspondiente al DMA, "AXI Direct Memory Access". El uso del DMA nos permite realizar acceso a memoria directo, sin necesidad de transitar la CPU del sistema. No obstante, se deben realizan ciertos cambios en su configuración. Se seleccionan 32 bits para las direcciones, 14 bits para los registros del *buffer* y se deshabilitan las opciones de *Scather Gather* y de Micro DMA. También es importante asegurar que tanto el canal de lectura como el de escritura se encuentren habilitados.

Finalmente, se realiza la adición del bloque IP del clasificador, al que hemos denominado "bitonicsort" y se procede a su interconexión con el DMA para realizar el envío y la recepción de datos a través de este. Así es posible obtener de forma sencilla un sistema completo donde se puede comprobar la funcionalidad del clasificador sin mayores complicaciones. Sin embargo, se requiere de la adición de ciertos bloques intermedios, como son los AXI Interconnect, para poder permitir la interconexión entre el PS de Zynq, el DMA y el clasificador. Tras realizar todas las conexiones entre los bloques, teniendo en cuenta las señales de reloj y reset, se obtiene el diagrama de bloques completo que se puede observar en la Figura 56.

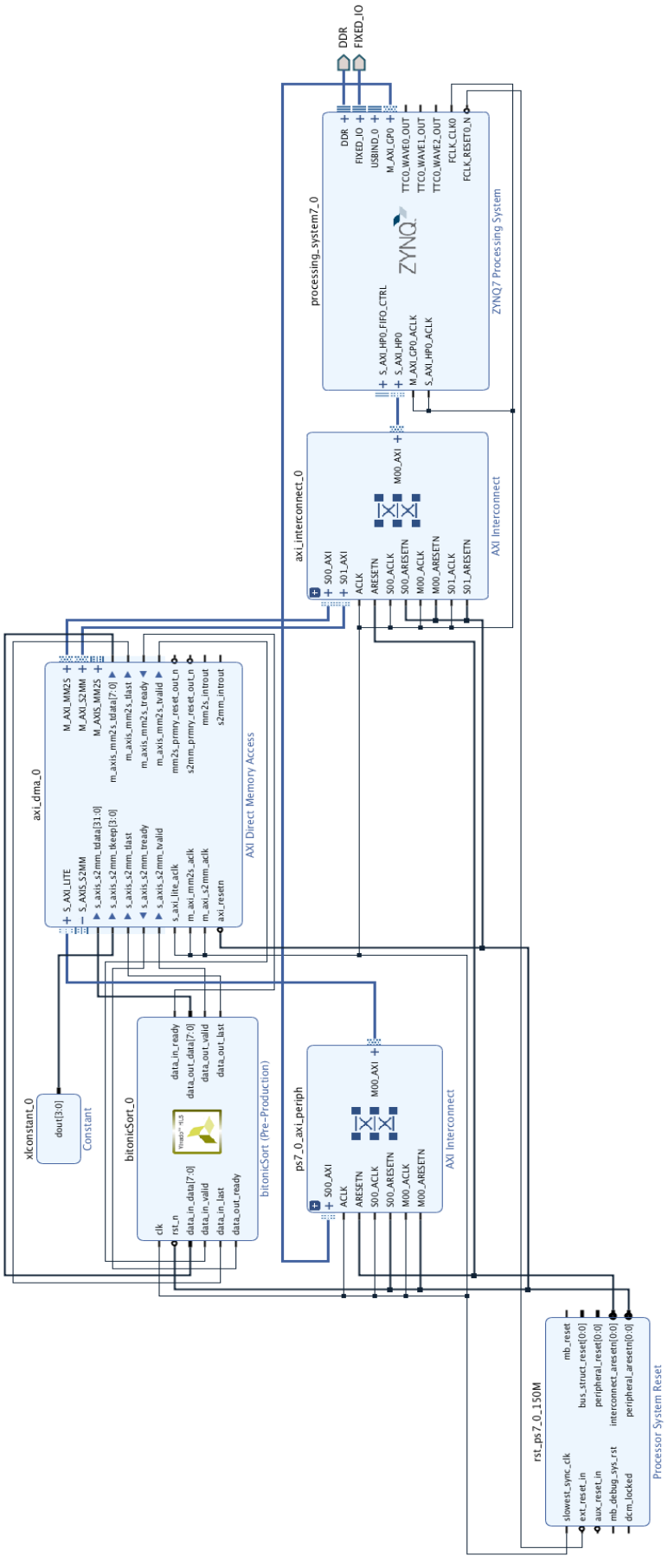

Figura 56: Diagrama de la plataforma final del clasificador

A continuación, se procede a la validación del diseño realizado. Para ello es necesario la generación del *bitstream* necesario para programar la plataforma. No obstante, se debe generar los productos de salida del diagrama realizado, mediante "Generate Output Products", tal y como se muestra en Figura 57. Para ello, se realiza una síntesis global y se ejecuta en el servidor local. Una vez finalizado, también es necesario crear el *wrapper* HDL del diseño, permitiendo a Vivado manejar el *wrapper* de manera independiente y actualizarlo. Tras ello, el diseño ya está listo para realizar la síntesis, la implementación y la generación del *bitstream*, en dicho orden.

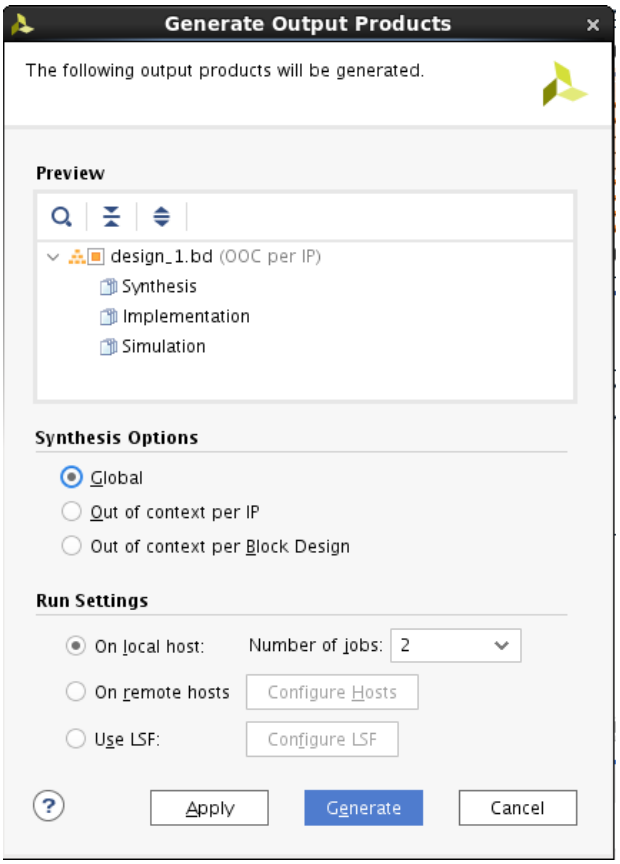

Figura 57: Generación de productos de salida en Vivado Design Suite

Una vez finalizada la generación del *bitstream*, se obtiene el *layout* de la plataforma con los recursos utilizados, como se muestra en Figura 58. En verde se muestran los recursos utilizados por el clasificador, siendo el resto de recursos los necesarios para la implementación del resto de bloques del sistema. Los resultados del análisis temporal del diseño, se muestran en la Figura 59. Como se puede observar, el *slack* positivo obtenido es de 1'037 ns, lo que implica que la frecuencia de funcionamiento utilizada para el bloque PS de 150 MHz cumple con las restricciones del diseño. También indica que la frecuencia aún puede ser aumentada, pero para realizar la validación del bloque no será necesario realizar cambios.

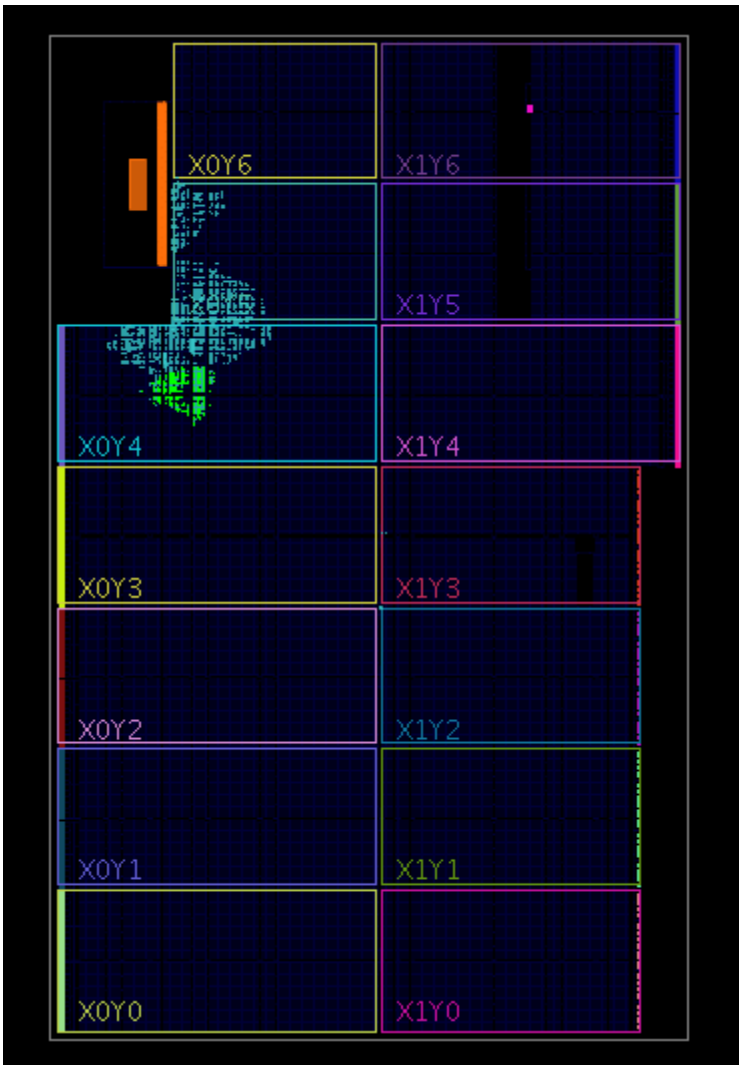

Figura 58: Layout de la plataforma

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.037 ns	Worst Hold Slack (WHS):	0.037 ns	Worst Pulse Width Slack (WPWS):	2.732 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	10391	Total Number of Endpoints:	10391	Total Number of Endpoints:	4180

All user specified timing constraints are met.

Figura 59: Resultados de la temporización del diseño

En la Figura 60 se muestra un gráfico con los porcentajes de recursos utilizados tras la implementación del sistema. Las LUTs utilizadas ascienden a 3218, siendo un 1,47% de las disponibles en la FPGA ZC706. También se hace uso de 217 LUTRAMs (0,31%), 3876 FFs (0,89%), 3 BRAMs (0,55%) y 1 BUFG (3,13%). Sin embargo, los recursos utilizados por el clasificador son mucho menores a los del sistema completo. Por ello, en la Tabla 9 se observa una comparativa de los recursos totales del sistema implementado y los recursos del clasificador.

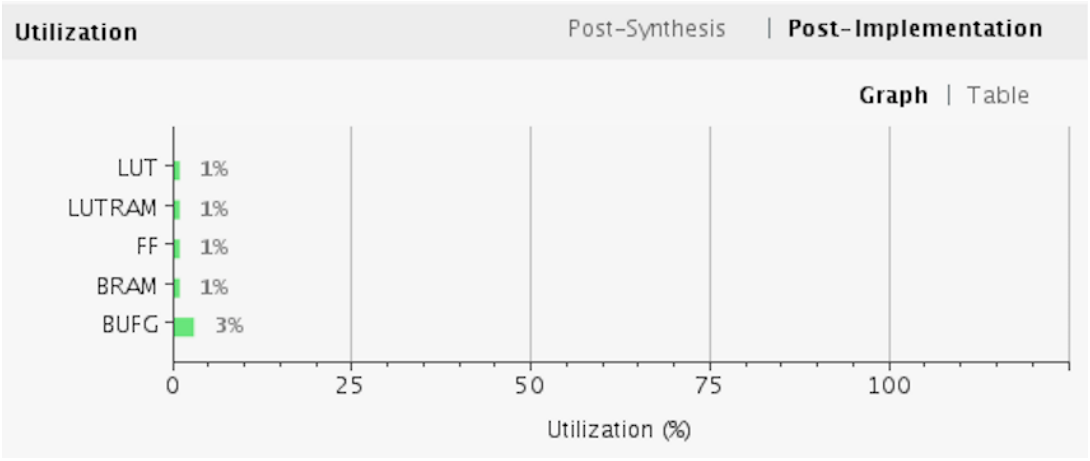

Figura 60: Gráfico de utilización de recursos tras la implementación

Tabla 9: Utilización de recursos del sistema y del clasificador

	Sistema completo		*Bitonic sort*	
LUT	3.218	1,47 %	521	0,24 %
LUTRAM	217	0,31 %	0	0,00 %
FF	3.876	0,89 %	566	0,13 %
BRAM	3	0,55 %	1	0,18 %
BUFG	1	3,13 %	0	0,00 %

Por último, cabe resaltar los resultados obtenidos al realizar un análisis de la potencia consumida por el sistema implementado. En Figura 61 se muestra el consumo de potencia de la plataforma, siendo este igual a 1,619 W.

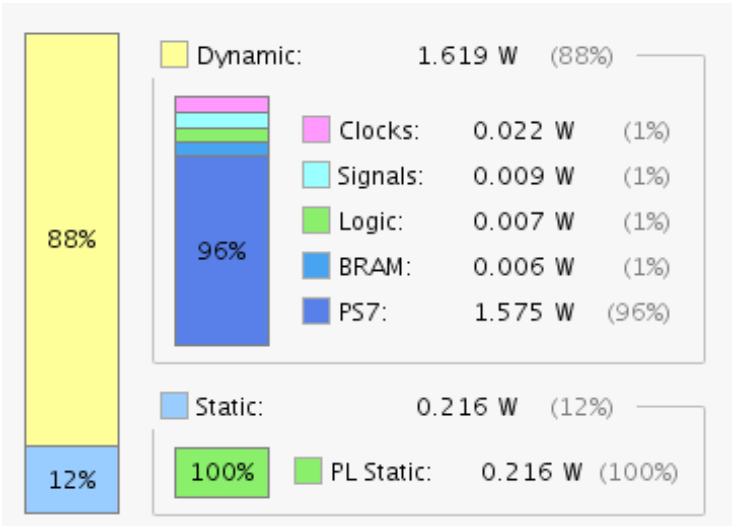

Figura 61: Consumo de potencia de la plataforma

Una vez obtenido el sistema completo, es necesaria realizar la exportación del hardware obtenido en el desarrollo del diseño de la plataforma. Una vez exportado, se hace uso de la

herramienta SDK para poder realizar la programación del software empotrado. Para ello es necesario realizar la creación de dos proyectos en SDK en el mismo entorno de trabajo que el proyecto realizado en Vivado Design Suite. El primer proyecto es un BSP que incluye el paquete de soporte del dispositivo. Para ello, se hace uso de la configuración por defecto, asegurándose de que el nombre de la plataforma utilizada sea "design_1_wrapper_hw_platform_0" y de que la CPU utilizada sea la de la plataforma, "ps7_cortexa9_0".

El segundo proyecto es un proyecto de aplicación que incluye la aplicación de la verificación del correcto funcionamiento del sistema. No obstante, se ha de tener en cuenta las mismas restricciones que para el proyecto anterior, incluyendo en este caso que se haga uso del BSP generado anteriormente.

La aplicación es la encargada de gestionar el funcionamiento del DMA y realizar la transmisión y recepción de datos del clasificador, permitiendo validar así su correcto funcionamiento. En la Figura 62 se muestra el archivo "main.c" utilizado para la gestión de la aplicación. En él se incorpora el comportamiento general del DMA, el envío de datos y la comprobación de la recepción. Para ello, se hace uso de la señal "XAxiDma_SimplePollExample" que realiza la inicialización del DMA con configuración simple y la maneja a través de *polling*. También realiza el envío de un paquete almacenado en memoria "payload[]", del cual se conoce tanto su dirección como su valor. Mediante la utilidad Minicom se pueden observar por pantalla los mensajes recibidos y si estos cumplen con la ordenación del clasificador. El uso de datos almacenados en memoria, asegura que los datos enviados y recibidos no han variado, verificando así el correcto funcionamiento del sistema.

```
// Include & define files
…
int XAxiDma_SimplePollExample(u16 DeviceId);
void payload0(u8* payload);
char a;
/************************Variable
Definitions************************/
/* Device instance definitions */
XAxiDma AxiDma;

int main() {
    printf("Init Configuration \n\r");
    XAxiDma_SimplePollExample(0);
    return 0;
}

int XAxiDma_SimplePollExample(u16 DeviceId) {
    // Variables
    …
    TxBufferPtr = (u32 *)TX_BUFFER_BASE;
```

```
/* Initialize the XAxiDma device. */
    CfgPtr = XAxiDma_LookupConfig(DeviceId);

    if (!CfgPtr) {
        xil_printf("No config found for %d\r\n", DeviceId);
        return XST_FAILURE;
    }
    Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);

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

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

    /* Disable interrupts, we use polling mode */
    XAxiDma_IntrDisable(&AxiDma,
      XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(&AxiDma,
      XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
    u8 payload[130];

    for (Index = 0; Index < 64; Index ++) {
        TxBufferPtr[Index] = payload[Index];
    }

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

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

    while ((XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) ||
         (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
        /* Wait */
    }
    u8 *RxPacket;
    RxPacket = (u8 *) RX_BUFFER_BASE;
```

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

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

Figura 62: Archivo "main.c" con el código principal para la validación del clasificador

Para la configuración de la depuración es necesario seleccionar en las características del depurado una aplicación Xilinx C/C++ (GDB), la cual se define con los siguientes parámetros. Hace uso de una aplicación independiente con conexión local, escoge como plataforma hardware "design_1_wrapper_hw_platform_0" y como procesador "ps7_cortexa9_0", al igual que en el BSP y el proyecto de aplicación. Es necesario incluir el archivo *bitstream* "design1_wrapper.bit" y el fichero de inicialización "ps7_init.tcl", ambos generados de forma automática por Vivado Design Suite. También es necesaria definir la aplicación creada para validar el sistema, que en este caso es "Verification.elf", tal y como se muestra en Figura 63.

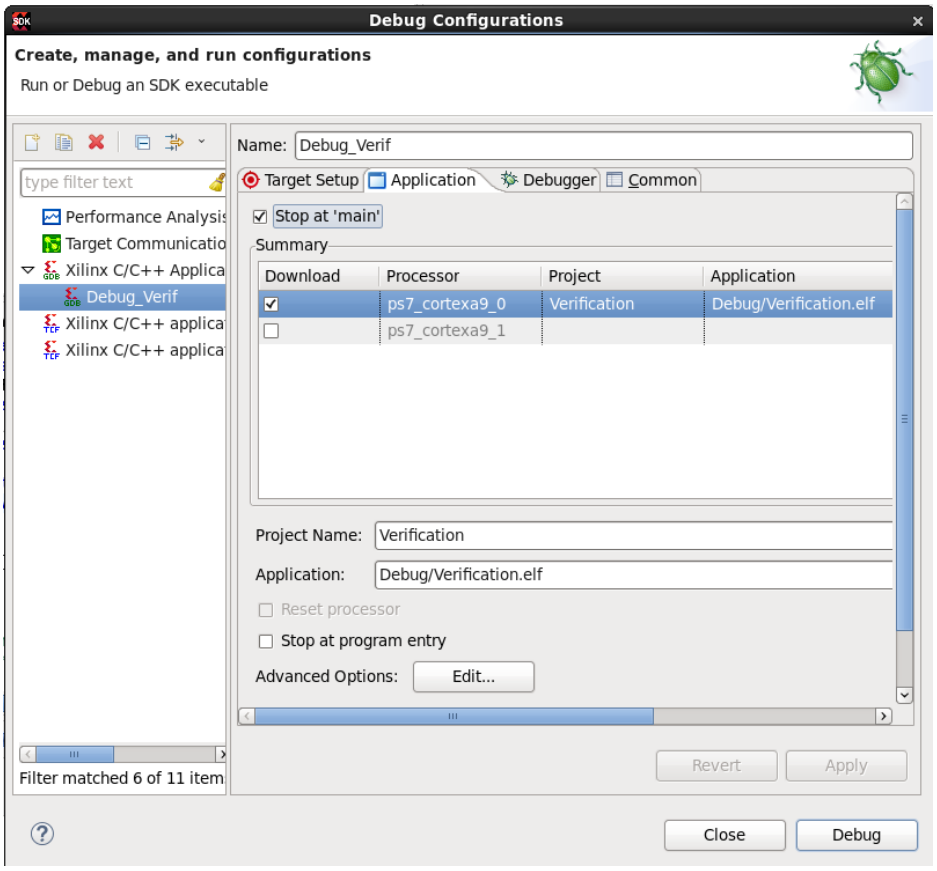

Figura 63: Configuración de la depuración del sistema

Finalmente, ya es posible proceder a la validación del sistema sobre la FPGA. Tras ejecutar la depuración creada, se hace uso de la utilidad Minicom para observar a través del terminal los mensajes recibidos, permitiendo de esta manera comprobar los datos enviados y recibidos y el correcto funcionamiento del sistema. Como se puede observar en la Figura 64 se muestra un mensaje que indica la inicialización de la aplicación y de la configuración del DMA y, a continuación, se muestran los mensajes recibidos una vez han sido clasificados. De esta manera, se comprueba el correcto funcionamiento del sistema, y como consecuencia, del clasificador implementado.

```
Init Configuration
Data[0] - key: 03
Data[1] - value: 00
Data[2] – key: 05
Data[3] – value: 28
Data[4] – key: 08
Data[5] – value: c1
…
Data[60] - key: FA
Data[61] - value: 61
Data[62] - key: FC
Data[63] – value: E5
```

Figura 64: Resultado del Minicom en la validación del clasificador bitonic sort

5.3. Conclusiones

Como conclusión, cabe resaltar la importancia de obtener resultados reales para la validación del sistema. Tras conseguir unos resultados favorables, es posible confirmar la funcionalidad del sistema. Con pequeñas modificaciones este puede ser implementado en sistemas más complejos, permitiendo el desarrollo de aplicaciones Big Data con un bus de comunicaciones de bajo consumo, elevados anchos de banda y reducidas latencias, como es PCI Express.

Con pequeñas modificaciones es posible realizar la inclusión de la arquitectura MapReduce, permitiendo así obtener el diseño final para el análisis de datos y la implementación de un sistema completo Big Data. No obstante, debido a limitaciones temporales y complejidades en la elaboración del diseño no ha sido posible su adición en este proyecto.

Capítulo 6. Conclusiones y trabajos futuros

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

6.1. Conclusiones del Trabajo

Tras haber realizado el desarrollo, implementación y validación del sistema, es posible confirmar su funcionalidad y validez sobre la placa de desarrollo utilizada. Los objetivos generales definidos para el desarrollo del trabajo han sido cumplidos, obteniendo como resultado final un sistema funcional con el protocolo de comunicaciones PCI Express implementado y un bloque IP funcional con el algoritmo de clasificación escogido.

Inicialmente, se ha llevado a cabo un estudio respecto al bus de comunicaciones PCI Express, debido a la creciente necesidad de mejorar la velocidad en la recepción y transmisión de paquetes en las FPGAs. Tras explicar las ventajas que supone la implementación de este bus de comunicaciones, se ha realizado el estudio de RIFFA, pues se trata del macro utilizado para su implementación. Por tanto, se procede a su verificación y validación sobre la FPGA ZC706 de Xilinx, haciendo uso del entorno de herramientas Vivado Design Suite. PCI Express es un bus de comunicaciones de bajo consumo de potencia, como se ha podido comprobar en este trabajo siendo el consumo de potencia de 1,672 W para la implementación final del mismo.

Previo a realizar el diseño del clasificador, se ha realizado un estudio de diferentes algoritmos que podrían ser completamente válidos para cumplir con la funcionalidad especificada. No obstante, se ha decidido hacer uso del algoritmo *bitonic sort*, el cual permite paralelización sobre la FPGA y, como consecuencia, altas velocidades de clasificación. El flujo de diseño utilizado permite volver atrás en el desarrollo de nuestro clasificador, permitiendo realizar cualquier cambio y cumplir así con los objetivos del trabajo. Tras la obtención del diseño del clasificador mediante el lenguaje de

programación de alto nivel SystemC es posible obtener un bloque IP gracias a la herramienta Vivado HLS y su posterior implementación sobre la plataforma tras la generación del *bitstream* en Vivado Design Suite, realizándose la programación y verificación del clasificador mediante la herramienta SDK.

Finalmente, cabe destacar que el algoritmo de clasificación desarrollado en SystemC permite flexibilidad, adaptabilidad y portabilidad. La reusabilidad del diseño y la maximización de la frecuencia de funcionamiento son dos objetivos de este trabajo, ya que se pueden integrar de forma conjunta el bus de comunicaciones PCI Express y el clasificador, o utilizarse de forma separada para una gran variedad de aplicaciones.

6.2. Trabajos futuros

A continuación se trataran una serie de propuestas para futuros trabajos, que buscan la mejora del funcionamiento y el rendimiento, tanto del clasificador como del protocolo de comunicaciones PCI Express. Inicialmente, debido a falta de tiempo no se ha podido realizar la integración final del sistema con el módulo MapReduce de Big Data, por lo que esta línea de trabajo sería de especial interés, ya que facilitaría en gran medida el procesado y clasificado de los datos, junto a unas altas tasas de comunicación.

No obstante, también resulta interesante proseguir con el desarrollo del clasificador, ya que existen infinidad de posibilidades para continuar esta línea de trabajo. Partiendo desde la elección de un algoritmo de clasificación diferente, como a mejoras en los tiempos de clasificación y el número de datos posibles a clasificar. También podría resultar de interés realizar comparativas entre diferentes algoritmos y sus prestaciones sobre la FPGA, pudiendo obtener un estudio más detallado sobre el tipo de algoritmos óptimos para este tipo de plataformas, o incluso en función del tipo de aplicación requerida.

También, resultaría interesante estudiar otras líneas de trabajo en lo que se refiere a la implementación del protocolo de comunicaciones PCI Express sobre la FPGA, pudiendo realizar comparativas con otras implementaciones ya existentes, como puede ser Xillybus. También es posible realizar modificaciones dentro de RIFFA en el número de canales de PCI Express y realizar estudios respecto a la gestión de grandes cantidades de datos o estructuras más extensas y complejas.

Referencias

[1] K. Neshatpour, A. Sasan, and H. Homayoun, “Big Data Analytics on Heterogeneous Accelerator Architectures,” in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2016, p. 16:1--16:3 [Online]. Available: http://doi.acm.org/10.1145/2968456.2976765

[2] K. Olukotun, “Big Data Analytics in the Age of Accelerators,” London (UK), 2015 [Online]. Available: http://reconfigurablecomputing4themasses.net/files/1.2 Kunle.pdf. [Accessed: 01-Mar-2017]

[3] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, “FPMR: MapReduce Framework on FPGA,” in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010, pp. 93–102 [Online]. Available: http://doi.acm.org/10.1145/1723112.1723129

[4] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for Multi-core and Multiprocessor Systems,” in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 13–24 [Online]. Available: https://doi.org/10.1109/HPCA.2007.346181

[5] D. Diamantopoulos and C. Kachris, “High-level synthesizable dataflow MapReduce accelerator for FPGA-coupled data centers,” in *Proceedings - 2015 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2015*, 2015, no. Samos Xv, pp. 26–33 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7363656&isnumber=7363643

[6] N. Budruk, D. Anderson, and T. Shanley, “Chapter 1: Architectural Perspective,” in *PCI Express System Architecture*, Ed. Addison Wesley, 2004, pp. 9–54.

[7] M. Sadri, C. De Schryver, and N. Wehn, “High-Bandwidth Low-Latency Interfacing with FPGA Accelerators Using PCI Express BT - FPGA Based Accelerators for Financial Applications,” in *FPGA Based Accelerators for Financial Applications*, C. De Schryver, Ed. Cham: Springer International Publishing, 2015, pp. 117–141 [Online]. Available: http://dx.doi.org/10.1007/978-3-319-15407-7_6

[8] Xilinx Inc., *Zynq-7000 All Programmable SoC. Technical Reference Manual*, UG585 ed. 2016 [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[9] X. Inc., *7 Series FPGAs Integrated Block for PCI Express v3.3*, PG054 ed. 2016 [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_3/pg054-

7series-pcie.pdf

[10] Xilinx Inc., *Zynq-7000 All Programmable SoC ZC706 Evaluation Kit Getting Started Guide*, UG961 ed. 2012.

[11] T. B. Preußer and R. G. Spallek, "Ready PCIe data streaming solutions for FPGAs," *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Munchen (Germany), pp. 1–4, 2014 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6927444&isnumber=6927322

[12] A. Rjabov, A. Sudnitson, V. Sklyarov, and I. Skliarova, "Interactions of Zynq-7000 devices with general purpose computers through PCI-express: A case study," in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, 2016, pp. 1–4 [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7495400&isnumber=7495297

[13] Xilinx Inc., "Vivado Design Suite User Guide: Design Flows Overview." pp. 1–106, 2018 [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug892-vivado-design-flows-overview.pdf

[14] Xilinx Inc., "Vivado Design Suite User Guide: Getting Started." pp. 1–22, 2018 [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug910-vivado-getting-started.pdf

[15] Xilinx Inc., "Vivado Design Suite User Guide: Designing with IP." pp. 1–114, 2018 [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug896-vivado-ip.pdf

[16] Xilinx Inc., "Vivado Design Suite User Guide: High-Level Synthesis." pp. 1–667, 2018 [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug902-vivado-high-level-synthesis.pdf

[17] J. Lawley, "Understanding Performance of PCI Express Systems," *Xilinx*. 2014 [Online]. Available: www.xilinx.com

[18] N. Budruk, D. Anderson, and T. Shanley, "Chapter 2: Architecture Overview," in *PCI Express System Architecture*, Ed. Addison Wesley, 2004, pp. 55–105.

[19] Richard Solomon, "RAS Data Protection Consideration for PCI Express Designs," 2016. [Online]. Available: https://www.synopsys.com/designware-ip/technical-bulletin/ras-data-protection.html. [Accessed: 03-Jul-2018]

[20] PLDA, "PCI Express Lexicon," 2018. [Online]. Available: https://www.plda.com/glossary. [Accessed: 03-Jul-2018]

[21] E. Solari and B. Congdon, "Chapter 6: Transaction Layer," in *The Complete PCI Express Reference. Design Insights for Hardware and Software Developers*, Intel Press, 2003, pp. 215–313.

[22] N. Budruk, D. Anderson, and T. Shanley, "Chapter 5: ACK/NAK Protocol," in *PCI Express System Architecture*, Ed. Addison Wesley, 2004, pp. 209–250.

[23] E. Solari and B. Congdon, “Chapter 8: Physical Layer and Packets,” in *The Complete PCI Express Reference. Design Insights for Hardware and Software Developers*, Intel Press, 2003, pp. 359–420.

[24] E. Solari and B. Congdon, “Chapter 1: Architecture Overview,” in *The Complete PCI Express Reference. Design Insights for Hardware and Software Developers*, Intel Press, 2003, pp. 1–39.

[25] E. Solari and B. Congdon, “Chapter 2: PCI Express Architecture Overview,” in *The Complete PCI Express Reference. Design Insights for Hardware and Software Developers*, Intel Press, 2003, pp. 41–91.

[26] N. Budruk, D. Anderson, and T. Shanley, “Chapter 4: Packet-Based Transactions,” in *PCI Express System Architecture*, Ed. Addison Wesley, 2004, pp. 154–209.

[27] Xillybus, “Down to the TLP: How PCI express devices talk (Part I) | xillybus.com,” 2018. [Online]. Available: http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1. [Accessed: 03-Jul-2018]

[28] UCSD, “RIFFA: A Reusable Integration Framework For FPGA Accelerators.” [Online]. Available: http://riffa.ucsd.edu/

[29] M. Jacobsen, Y. Freund, and R. Kastner, “RIFFA: A Reusable Integration Framework for FPGA Accelerators,” *IEEE 20th Int. Symp. Field-Programmable Cust. Comput. Mach.*, 2012 [Online]. Available: http://cseweb.ucsd.edu/~kastner/papers/fccm12-riffa.pdf

[30] M. Jacobsen, D. Richmond, and M. Hogains, “RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, 2015 [Online]. Available: http://dl.acm.org/citation.cfm?id=2815631

[31] D. Richmond and M. Jacobsen, “RIFFA 2.2.2 Documentation.” 2016 [Online]. Available: http://riffa.ucsd.edu/download

[32] D. Knuth, *The Art of Computer Programming. Sorting and Searching*, 2nd Editio. Addison-Wesley, 1973.

[33] D. Koch and J. Torresen, “FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on Fpgas for Large Problem Sorting,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011, pp. 45–54 [Online]. Available: http://doi.acm.org/10.1145/1950413.1950427

[34] “Insertion Sort,” *GeeksForGeeks*, 2018. [Online]. Available: https://www.geeksforgeeks.org/insertion-sort/

[35] “Quick Sort,” *GeeksForGeeks*, 2018. [Online]. Available: https://www.geeksforgeeks.org/quick-sort/

[36] “Heap Sort Algorithm,” *Programiz*, 2018. [Online]. Available: https://www.programiz.com/dsa/heap-sort

[37] T. Cormen and D. Balkcom, “Overview of merge sort,” *Khan Academy*, 2018. [Online]. Available: https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

[38] “Sorting Algorithms In Software Development,” *RoBa’s Worlds*, 2018. [Online]. Available:

http://www.robasworld.com/sorting-algorithms/

[39] "Bubble Sort Program in C Using Function," *CodingCompiler*, 2017. [Online]. Available: https://codingcompiler.com/bubble-sort-program-in-c-using-function/

[40] "Bitonic Sort," *GeeksForGeeks*, 2018. [Online]. Available: https://www.geeksforgeeks.org/bitonic-sort/

[41] M. Roozmeh, "High Level Synthesis of Bitonic Sorting Algorithm," *GitHub*, 2016. [Online]. Available: https://github.com/mediroozmeh/Bitonic-Sorting

[42] J. Zhang, Q. Zhao, M. Kuga, M. Amagasaki, M. Iida, and T. Sueyoshi, "A Comparison of Sorting Algorithms with FPGA Acceleration by High Level Synthesis," pp. 200–201, 2014.

[43] A. Széll, "Parallel Sorting Algorithms In FPGA," *13Th Phd Mini-Symposium*, 2006.

[44] E. Bainvile, "OpenCL Sorting," 2011. [Online]. Available: http://www.bealto.com/gpu-sorting_intro.html