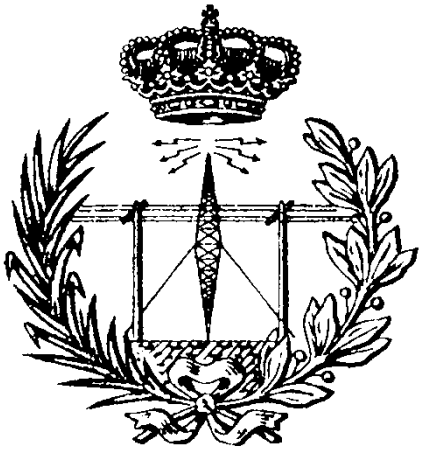
ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE GRADO

Optimización de una Plataforma de Inspección Profunda de Paquetes basada en SoC FPGA para Gigabit Ethernet

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: Dª. Sonia Raquel León Martín

Tutores: Dr. Pedro Pérez Carballo

D. Yúbal Barrios Alfaro

D. Adrián Domínguez Hernández

Fecha: Julio de 2017

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Optimización de una Plataforma de Inspección Profunda de Paquetes basada en SoC FPGA para Gigabit Ethernet

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario/a**

Fdo.: Fdo.:

Fecha: Julio de 2017

Agradecimientos

A Pedro Pérez Carballo, por todo su tiempo, su ayuda y sus palabras de ánimo para que este proyecto saliera adelante.

A mi familia, por todo el apoyo y la confianza que depositaron en mí durante estos años.

A los que estuvieron aquí, a los que no, también.

Resumen

En este trabajo se realiza el diseño, implementación y validación de un sistema de captura y filtrado de paquetes TCP/IP, que incluye un bloque destinado a eliminar las cabeceras de dichos paquetes Ethernet, con objeto de procesar únicamente su carga útil o *payload*. El sistema ha sido diseñado para ser implementado sobre un dispositivo *System on Chip* FPGA de la serie Xilinx Zynq-7000. El trabajo realizado consiste en estudiar la funcionalidad de la plataforma y de sus bloques principales, realizar la migración de los bloques necesarios, adaptándolo al flujo de diseño propuesto y realizar su prototipado con objeto de validar el funcionamiento del sistema de Inspección Profunda de Paquetes (DPI). El sistema DPI debe recibir tramas Ethernet, extraer su cabecera y determinar si se realiza su filtrado o no. En caso positivo, el paquete será enviado al motor de búsqueda para el análisis de la carga útil del paquete o reenviado por la interfaz de red Ethernet.

Una vez estudiada la funcionalidad del sistema, se pasa a estudiar las principales características de los dispositivos SoC FPGA Xilinx Zynq 7000 y de la metodología de diseño a utilizar durante el proyecto. En concreto se profundiza en la síntesis de alto nivel para la implementación de diferentes bloques desde su modelo SystemC, su verificación y su implementación para obtener sus características de tiempo, uso de recursos y potencia. A continuación, se procede a realizar el diseño de la plataforma y el desarrollo de la aplicación empotrada que se ejecuta en el procesador ARM Cortex A9 disponible en el SoC.

Para realizar esta integración se han estudiado diferentes arquitecturas para la plataforma, teniendo en cuenta los flujos de datos y tratando de evitar la creación de cuellos de botella en la arquitectura, a la vez de disminuir la necesidad de almacenamiento

de los paquetes de datos. La arquitectura resultante utiliza los recursos de comunicación para dar soporte al flujo de datos, apoyándose sobre los bloques disponibles para crear una arquitectura de comunicación compleja. En concreto se utilizan bloques para comunicación AMBA AXI4 en sus tres variantes (Stream, Memory Mapped y Lite).

Por último, se realizan las distintas validaciones del sistema para asegurar su correcto funcionamiento y realizar distintas medidas de consumo de potencia, recursos y latencia. Se ha conseguido diseñar un sistema de Inspección Profunda de Paquetes que funciona a una frecuencia de 200 MHz con una ocupación reducida del dispositivo, dando cabida para la integración de nuevas funciones.

Abstract

This work presents the design, implementation and validation of a TCP / IP packet capture and filtering system, including a block destined to eliminate the headers of Ethernet packets, in order to process only its payload. The system has been designed to be implemented on a Xilinx Zynq-7000 Series System on Chip FPGA device. The work carried out consists of studying the functionality of the platform and its main blocks, performing the migration of the necessary blocks, adapting it to the proposed design flow and performing its prototyping in order to validate the operation of the Deep Packet Inspection (DPI). The DPI system must receive Ethernet frames, extract its header, and determine whether it is filtered or not. If so, the packet will be sent to the search engine for payload analysis or forwarded to the network by the Ethernet network interface.

Once the functionality of the system is studied, the main characteristics of the SoC FPGA Xilinx Zynq 7000 devices and the design methodology to be used during the project are studied. In particular, the work put emphasis in the high-level synthesis design methodology for the implementation of different blocks from its SystemC model. Also, the verification of the design (high-level and RTL) are studied. After, it is carried out its implementation to obtain its characteristics of time, use of resources and power. Then proceed to implement the platform and the development of the embedded application that runs on the ARM Cortex A9 processor available in the SoC.

In order to perform this integration, different architectures have been studied for the platform, taking into account the data flows and trying to avoid the creation of

bottlenecks in the architecture, while reducing the need to store the data packets. The resulting architecture uses communication resources to support data flow, relying on available blocks to create a complex communication architecture. In particular, AMBA AXI4 communication blocks are used in its three variants (Stream, Memory Mapped and Lite).

Finally, the different validations of the system are made to ensure its correct operation and to realize different measures of consumption of power, resources, and latency. It has been possible to design a Deep Packet Inspection system that operates at a frequency of 200 MHz with a reduced occupation of the device, allowing for the integration of new functions.

Índice

Índice de figuras

Índice de tablas

Índice de código

Acrónimos

ACP	Accelerator Coherency Port
ADC	Analog-to-Digital Conversion
AMBA	Advanced High-Performance Bus
AMP	Asymmetrical Multiprocessing
APU	Application Processor Unit
AXI	Advanced eXtensible Interface
BRAM	Block RAM
BSP	Board Support Package
CAN	Controller Area Network
CLB	Configurable Logic Block
DDR	Double Data Rate
DFA	Deterministic Finite Automaton
DMA	Direct Memory Access
DPI	Deep Packet Inspection
DSP	Digital Signal Processor

EDIF	Electronic Design Interchange Format
EMIO	Extended MIO Interfaces
FF	Flip-Flop
FIFO	First In-Firs Out
FPGA	Field Programmable Gate Array
GIC	General Interrupt Controller
GMII	Gigabit Media-Independent Interface
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HDF	Hierarchical Data Format
HDL	*Hardware* Description Level
HLS	High-Level Synthesis
I2C	Inter-Integrated Circuit
IDS	Intrusion Detection System
IEE	Institute of Electrical and Electronics Engineers
ILA	Integrated Logic Analyzer
IoT	Internet of Things
IP	Intellectual Property
IP	Internet Protocol
IPS	Intrusion Prevention System
IUMA	Instituto Universitario de Microelectrónica Aplicada
JTAG	Joint Test Action Group Interface
LUT	Look-Up Tables

lwIP	Lightweight IP
MAC	Media Access Control
MII	Media-Independent Interface
MIO	Multiplexed Input/Output
MM	Memory Mapped
NFA	Non-Deterministic Finite Automaton
NMS	Network Monitoring System
OCM	On-Chip Memory
PCI	Peripheral Component Interconnect
PHY	Physical Layer
PL	Programmable Logic
PPI	Private Peripheral Interrupt
PS	Programmable System
QoS	Quality of Service
RAM	Random Access Memory
RGMII	Reduced Gigabit Media-Independent Interface
ROM	Read-Only Memory
RTL	Register Transfer Level
SCU	Snoop Control Unit
SDIO	Secure Digital Input Output
SDK	*Software* Development Kit
SGI	*Software* Generated Interrupt
SGMII	Serial Gigabit Media-Independent Interface

SLCR	System-Level Control Registers
SMP	Symmetrical Multiprocessing
SoC	System on Chip
SPI	Serial Peripheral Interface
SPI	Shared Peripheral Interrupt
SRAM	Static Random Access Memory
TCP/IP	Transmission Control Protocol/Internet Protocol
TEMAC	Tri-mode Ethernet MAC
TFTP	Trivial File Transfer Protocol
TLM	Transaction-Level Modeling
TTM	Time To Market
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
VCD	Value Change Dump
VHDL	VHSIC *Hardware* Description Language

Capítulo 1. Introducción

En este capítulo se exponen las razones por las que surge la necesidad de desarrollar este proyecto, incluyendo sus objetivos y la estructura del documento.

1.1 Introducción

El tráfico de datos en Internet está sufriendo un incremento constante en los últimos años, con una tasa de crecimiento anual superior al 20% (Tabla 1) [1]. Este incremento viene producido por varios factores, entre los que se puede citar el incremento del tráfico de vídeo en la red (Figura 1). Asimismo, la comunicación máquina-máquina introducida bajo el paradigma de Internet de las cosas (IoT) está conduciendo a un incremento significativo del tráfico y la conectividad en la red [2].

Asociado a este crecimiento del volumen de datos se presentan varios problemas importantes que deben ser tratados de forma consistente. Por una parte, se pueden identificar los problemas asociados a la calidad del servicio (QoS), tratando de dar prioridad a aquellos servicios que precisan una comunicación síncrona. Por otra parte, se presentan problemas de seguridad que afectan a la red y a los usuarios de la misma [3].

Tabla 1: Contexto histórico del tráfico de Internet [1]

Año	1992	1997	2002	2007	2015	2020
Tráfico Global de Internet	100 GB por día	100 GB por hora	100 GBps	2.000 GBps	20.235 GBps	61.386 GBps

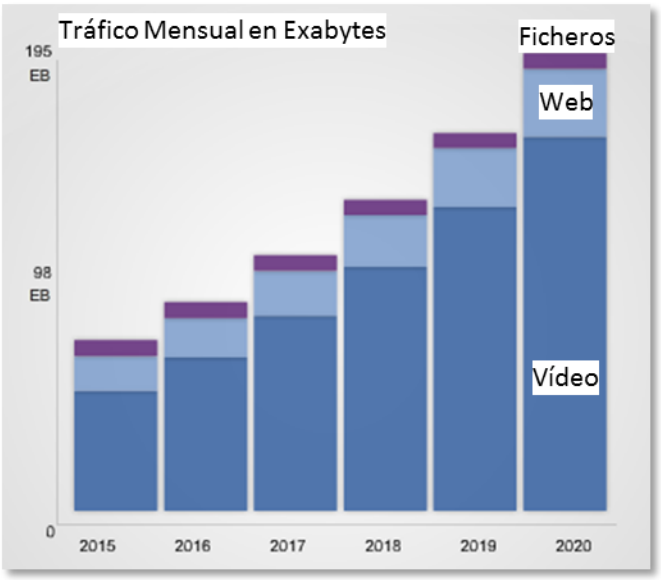


Figura 1: Perfil de tráfico mensual en Internet [1]

En relación a los problemas de seguridad se pueden seguir diferentes estrategias de análisis del tráfico, disponiendo de sistemas equipados con cortafuegos o *firewalls* que analizan diferentes parámetros orientados a la conexión, tratando de evitar que los equipos conectados detrás del cortafuegos reciban conexiones de equipos no permitidos. Para ello se analizan las cabeceras del paquete TCP/IP y se actúa (dejar pasar o desechar) sobre el paquete en función de un conjunto de reglas pre-configuradas [4]. Existen diferentes soluciones para problemas concretos tales como detección de intrusiones (IDS), prevención de intrusiones (IPS), monitorización de seguridad en la red (NSM) o procesamiento *offline* de los datos capturados para realizar un análisis forense de lo sucedido en la red [5][6].

Otras soluciones también analizan el contenido de la carga útil, o *payload,* del paquete de datos para identificar patrones que pudieran ser indicio de un contenido malicioso del mismo o con objeto de realizar una clasificación del tráfico para determinar la calidad del servicio. Estas técnicas se conocen como Inspección Profunda de Paquetes o DPI (*Deep Packet Inspection*) [7].

La implementación de sistemas de DPI precisa una alta capacidad de cómputo para evitar incurrir en un incremento en la latencia de las comunicaciones en tiempo real [8]. Una alternativa para ello es utilizar soluciones basadas en sistema *multicore*, con altas prestaciones y un alto ancho de banda. Ejemplos de este tipo de sistemas se pueden encontrar en [8] y en [9]. Otra alternativa es incrementar el paralelismo y facilitar la

reconfiguración del acelerador dedicado. En este caso, la utilización de FPGAs permite crear sistemas con requerimientos de tiempo real [10].

1.2 Antecedentes

En el IUMA se ha desarrollado una plataforma basada en FPGA para aplicaciones DPI. Incluye un acelerador *hardware* basado en Zynq que contiene un bloque que implementa el algoritmo de Boyer-Moore para búsqueda de patrones [11] [12].

Existen múltiples herramientas que pueden analizar el contenido de un paquete de manera estática, es decir, una vez que se encuentra en los servidores o en el propio ordenador. Por lo tanto, lo que hace peculiar a los DPI es la capacidad que tienen de inspeccionar el *payload* del paquete mientras éste todavía no ha llegado al destino, mejorando así la seguridad de la red pudiendo detectar en el *payload* protocolos, aplicaciones y contenido multimedia, incluso virus, *malware* e intrusiones [7]. El sistema toma decisiones sobre si procesar o rechazar el paquete en función del resultado de la inspección [13].

Los sistemas DPI están conformados generalmente por un bloque de captura de paquetes, encargado de analizar la cabecera del paquete recibido y, dependiendo de la política de análisis de la cabecera, es enviado a un bloque de análisis del *payload* en busca de un patrón determinado. Un DPI puede poseer múltiples bloques de análisis, realizando el despacho del *payload* a un bloque u otro, dependiendo del resultado generado al analizar la cabecera.

El análisis del *payload* a través de los sistemas DPI está basado en buscar en su contenido patrones predefinido, actuando como lo que se denomina una como máquina de patrones que representa uno de los bloques más importantes en estos sistemas. Es por ello que diseñar un algoritmo eficiente en cuanto a tiempo y consumo se convierte en una prioridad, dado que la mayor parte del tiempo de ejecución del sistema son empleados por estos algoritmos [14]. Para realizar esta búsqueda se implementan arquitecturas basadas en sistemas deterministas (DFA) o no deterministas (NFA) para búsqueda de expresiones regulares o con algoritmos de búsquedas de patrones fijos, como es el caso del algoritmo de Boyer-Moore [11] o de Wu-Manber [15].

Todas estas máquinas de patrones no presentan las prestaciones necesarias en las implementaciones *software*, creando cuellos de botella en el flujo de datos del sistema, por lo que es natural plantear la implementación de un sistema DPI sobre *hardware*, siendo esta una opción viable.

Es cada vez más frecuente el uso de FPGA como aceleradores *hardware*, ya que esta tecnología permite realizar procesamiento paralelo y además cuenta con la capacidad de reconfiguración, consiguiendo mejor rendimiento, reduciendo la latencia y el consumo de potencia del sistema [12].

La plataforma desarrollada en el IUMA [16] captura los paquetes provenientes de la red a través de un bloque Ethernet TEMAC y son enviados a un núcleo microprocesador de la plataforma, responsable de almacenar los paquetes en memoria, extraer el *payload* y enviar su contenido a un bloque acelerador. Este último realiza la inspección del *payload* a través del algoritmo de búsqueda de patrones de Boyer-Moore.

Con objeto de incrementar las prestaciones del sistema se ha desarrollado un bloque específico para capturar, filtrar y clasificar paquetes TCP/IP. Este bloque tiene acceso tanto al flujo de entrada como de salida de datos, conectándose directamente al bloque TEMAC [17]. Este bloque aporta un incremento significativo a las prestaciones del sistema.

Teniendo en cuenta este marco de referencia, en este proyecto se pretende incrementar las prestaciones de la plataforma integrando el bloque de captura en la plataforma de referencia y optimizando sus prestaciones para soportar tráfico multi-Gigabit Ethernet. Durante este proceso será preciso modificar la arquitectura de la misma, tratando de encontrar una solución al flujo de datos desde el bloque de captura, su almacenamiento en memoria y su flujo de datos hacia el bloque acelerador. Igualmente es necesario resolver los problemas asociados a la integración final sobre la plataforma.

1.3 Análisis del problema

El principal problema que tiene la plataforma DPI de referencia [16] es la partición *hardware/software* de su arquitectura. En esta partición, la parte *software* se encarga de realizar el filtrado del paquete a través de su cabecera, mientras que en la parte *hardware*

se localizan los aceleradores, con el fin de aumentar sus prestaciones. Estos aceleradores son los encargados de realizar el análisis de la carga útil del paquete.

Los paquetes son capturados a través de controladores de red y son enviados a la memoria SRAM del sistema de procesamiento. Para realizar esta comunicación es necesario incorporar bloques DMA (*Direct Memory Access)*. De la misma manera, para comunicar la memoria SRAM del sistema de procesamiento con los aceleradores *hardware* es necesaria la incorporación del DMA. Las plataformas que cuentan con este tipo de bloques en su arquitectura presentan cuellos de botella y limitaciones en la velocidad de transmisión. Por lo tanto, la latencia mínima que tiene el sistema DPI viene determinada por el tiempo de transmisión de un paquete desde el controlador de red hasta la memoria SRAM, el tiempo de filtrado de la cabecera del paquete y el tiempo de transmisión del paquete desde la memoria SRAM hasta el controlador de red.

Otro problema que presenta este tipo de arquitecturas es el tiempo de toma de decisión. El filtrado de la cabecera del paquete se realiza únicamente cuando el paquete ha llegado por completo a la memoria SRAM del sistema de procesamiento, pudiendo extraer la cabecera del paquete una vez que se encuentra almacenado.

Tras plantear los problemas que se presentan en las arquitecturas actuales, en este Trabajo Fin de Grado se proponen alternativas a esta plataforma con el fin de optimizarla.

Para solucionar las limitaciones de velocidad y los posibles cuellos de botella de las otras arquitecturas, se propone implementar todo el sistema DPI en el dominio *hardware,* prescindiendo así del uso del procesador y del almacenamiento en memoria de los paquetes de datos accesibles por el acelerador *hardware* mediante el uso de DMA. Para ello, el filtrado de la cabecera del paquete se implementa en *hardware* mediante el desarrollo de un bloque IP que realice esta función, con una aproximación de tipo flujo de datos (*dataflow*).

La consecuencia de ello es la reducción de la latencia de la toma de decisión debido a que, al eliminar los bloques intermediarios, el paquete es capturado a través del controlador de red y es enviado directamente al bloque de filtrado de cabecera. Mientras se está recibiendo el paquete, usando un determinado ancho de palabra, se puede realizar el análisis de la cabecera al mismo tiempo que se termina de recibir el paquete.

Con estos cambios en la arquitectura del sistema DPI se consigue obtener una plataforma optimizada, reduciendo el uso de la CPU del sistema e implementando una arquitectura basada en flujo de datos.

Otro de los problemas que se presentan en este tipo de sistemas, donde existen diferentes bloques optimizados para la aplicación, es migrar su funcionalidad entre entornos de diseño, ya sea porque se alcanza el final de vida del entorno donde se ha desarrollado, se implementa en un nuevo dispositivo para mejorar las prestaciones o por requerimientos del proyecto. La utilización de estándares mitiga este problema, pero la experiencia indica que no es un problema trivial debido a la diferente interpretación que realizan las diferentes herramientas de cara a su implementación en la arquitectura final. En este proyecto ha sido necesario abordar este tipo de problemática para diferentes bloques del diseño final.

1.4 Objetivos

El objetivo principal de este Trabajo Fin de Grado es la integración del bloque IP de captura de paquetes en la plataforma existente, con el objeto de incrementar las prestaciones globales del sistema en el contexto indicado anteriormente.

Los objetivos operativos que se proponen se pueden resumir en los siguientes puntos:

O1. Conocer la plataforma en su implementación actual.

O2. Conocer la funcionalidad del bloque de captura y sus interfaces de E/S.

O3. Diseñar la nueva arquitectura del sistema, incluyendo las comunicaciones internas del sistema.

O4. Integrar el bloque de captura de paquetes TCP/IP en la plataforma modificada.

O5. Realizar la validación funcional del sistema.

1.5 Peticionario

Actúa como peticionario de este proyecto la División Sistemas Industriales y CAD (SICAD) del Instituto Universitario de Microelectrónica Aplicada (IUMA) de la Universidad de Las Palmas de Gran Canaria.

De igual manera, es solicitante de este proyecto la Escuela de Ingeniería de Telecomunicación y Electrónica para cubrir los requerimientos de la asignatura Trabajo Fin de Título dentro de la materia Trabajo Fin de Grado del plan de estudios en vigor.

1.6 Estructura del documento

Este documento se divide en nueve capítulos donde se describe el trabajo realizado durante el proceso del proyecto. En el primer capítulo se presentan los antecedentes del proyecto, se analiza el problema de los sistemas DPI y se muestran los objetivos. En el segundo capítulo se muestra la metodología de diseño usada en este proyecto, explicando las herramientas que se han utilizado. A continuación, en el tercer capítulo, se habla del dispositivo utilizado, el SoC Zynq ZC706. Se explicarán sus características, así como su arquitectura.

Es en el cuarto capítulo donde se describe la arquitectura propuesta para implementar un sistema DPI, mostrándose distintas variaciones y justificándose el uso de una u otra. En el capítulo siguiente se procede al diseño de los bloques IP teniendo en cuenta la arquitectura elegida, se muestran los resultados de la simulación y la co-simulación para comprobar que se ha realizado el modelado y la síntesis en alto nivel de manera correcta. Y se finaliza por el empaquetado del bloque para realizar su integración en el sistema completo.

Con los bloques IP diseñados, en el capítulo seis se procede a la integración de la plataforma al completo, viendo los distintos bloques que la componen y su resultado final en forma de diagrama de bloques. Se realizará la síntesis e implementación de la plataforma. El capítulo siguiente trata la integración *hardware-software* para comunicar el PS con el PL, y muestra también los pasos seguidos para la realización de la aplicación empotrada.

En el capítulo ocho se describe el proceso de validación realizado en la plataforma para comprobar su funcionalidad, apoyándose en todo momento de capturas que lo demuestran. Por último, el capítulo final es el de las conclusiones del proyecto exponiendo los resultados a los que se han llegado y presentando las líneas futuras que pueden mejorar el proyecto. Finalmente, se incluye la bibliografía en la que se apoya el documento, el presupuesto y el pliego de condiciones del proyecto.

Capítulo 2. Metodología de diseño

En este capítulo se explica la metodología de diseño que se ha utilizado en el proyecto para el modelado, la síntesis y la implementación del sistema. Se justifica la elección de esta metodología, así como el flujo de diseño y las diversas herramientas que lo conforman.

2.1 Introducción

La complejidad de los circuitos integrados se ha ido incrementado exponencialmente, tal y como se formuló en la Ley de Moore, integrando millones de componentes con geometrías cada vez más reducidas. Debido a esto, los últimos *System on Chip* (SoC) alcanzan niveles de integración equivalentes a millones de puertas lógicas. El método de diseño dominante en la actualidad, el del nivel de transferencia de registros (RTL), ya no es capaz de escalar a la complejidad de los diseños que están surgiendo hoy en día. Por ello, la implementación de algoritmos complejos en sistemas electrónicos se ha convertido en una ardua tarea en la que se requiere planificar las operaciones. Todo esto se resume en que continuar usando métodos de diseño RTL ya no resulta económicamente viable. Esto hace necesario crear nuevos métodos y técnicas de diseño con un nivel de abstracción mayor, ya sea a nivel algorítmico o de sistema [18].

En este nuevo nivel de abstracción, superior al RTL, los detalles temporales y estructurales se abstraen y la comunicación entre los módulos se denomina transacción, siendo por lo tanto este nivel de abstracción el de modelado a nivel de transacciones (TLM).

Este modelado se basa en lenguajes de alto nivel para la descripción de sistemas, como C/C++/SystemC. Los componentes se modelan como módulos con un conjunto de procesos que calculan y representan su comportamiento. Estos módulos intercambian información a través de un canal abstracto modelado en alto nivel. Se implementan interfaces en los canales para encapsular los protocolos de comunicación, por los cuales los procesos pueden acceder para establecer comunicación. La principal característica por tanto es separar las comunicaciones del procesamiento, con lo que cada parte del sistema se puede evolucionar por separado.

2.2 Diseño en alto nivel

Existen muchas razones para utilizar la metodología de diseño de alto nivel. La primera, es la importancia de acortar los tiempos en los que salen al mercado los productos (TTM – *Time To Market*). En segundo lugar, tiene un alto potencial en realizar análisis y verificación funcional de la arquitectura, que son temas cruciales en el desarrollo de un diseño. Y la última razón es la verificación del diseño *hardware*, que representa una parte notable (entre el 50 – 70 % del esfuerzo de un proyecto), aportando el rendimiento final de la integración del sistema. Una verificación eficiente reduce drásticamente el tiempo de desarrollo [19].

Como se ha comentado anteriormente, la descripción del sistema electrónico se basa en lenguajes de alto nivel, tales como C/C++ o SystemC. Este último, se ha convertido en estándar IEEE [20] para modelado de sistemas basados en transacciones debido a que describe el comportamiento funcional, la estructura y las especificaciones temporales, en diferencia de C/C++, que carece de posibilidades de planificación temporal.

Una vez que la descripción del circuito en alto nivel se ha realizado a nivel de transacciones en C/C++/SystemC, se realiza la síntesis de alto nivel que generará la microarquitectura del sistema a nivel RTL. Por tanto, la síntesis de alto nivel (HLS) aporta el nexo necesario entre la descripción algorítmica de un diseño y su implementación estructural en el nivel de transferencia de registros (RTL).

Esta metodología de diseño, basada en la síntesis de alto nivel, transforma el algoritmo de un código C/C++ o SystemC a una microarquitectura a nivel RTL, formada

básicamente por una ruta de datos y una unidad de control y expresada en lenguajes de descripción *hardware* como VHDL o Verilog [21].

Para obtener este código basado en descripción *hardware* existen diferentes tipos de herramientas que siguen estrategias diversas. Estas herramientas de síntesis de alto nivel crean automáticamente diseños RTL a partir de modelos SystemC, C o C++.

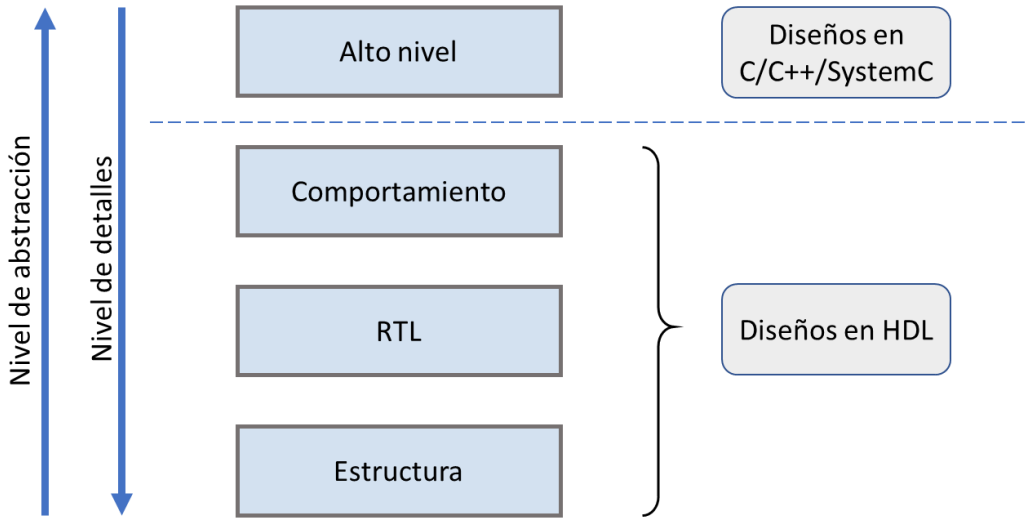


Figura 2: Nivel de abstracción. Adaptada de [22]

2.3 Modelado en alto nivel

Los lenguajes de descripción *hardware* han tenido que migrar a un nivel de abstracción más alto para satisfacer las necesidades de diseño de los sistemas electrónicos donde aparecen nuevos métodos de modelado y verificación del diseño. Lenguajes como SystemC o SystemVerilog surgen para cubrir esta necesidad de describir los sistemas electrónicos a un nivel de sistema.

El uso del lenguaje C para el diseño en alto nivel proporciona un alto nivel de abstracción, consiguiendo realizar un diseño de la funcionalidad del sistema. Con este lenguaje se consigue simplificar el diseño, no sólo con la reducción de líneas de código, también minimizando errores y permitiendo realizar depurados y validaciones de manera más rápida.

C++ posee un nivel de abstracción mayor al de C debido a que es una extensión de C pero orientado a objetos, permitiendo mayor flexibilidad en el desarrollo del código. Por otro lado, los lenguajes C y C++ no poseen nociones temporales del diseño.

SystemC es un lenguaje orientado al diseño *hardware* que pretende unificar el modelo arquitectural con la implementación. SystemC es una librería de clases de C++ la cual permite diseñar y verificar sistemas *hardware*/*software* para diferentes niveles de abstracción.

Facilita el modelado y la verificación a un alto nivel, el refinamiento de módulos iterativos desde un nivel superior a uno inferior separando las especificaciones de comunicación desde su implementación y define un modelo de computación que puede ser personalizado (tiempos discretos, continuos; activación de procesos, etc.) [20].

Además, SystemC proporciona características que no están soportadas por el lenguaje C++, como puede ser la noción temporal de los eventos, la concurrencia o tipos de datos necesarios para el *hardware*.

En este Trabajo de Fin de Grado se ha utilizado SystemC como lenguaje debido a las ventajas que ofrece en comparación con C/C++, como es el control temporal que se tiene sobre el diseño y su facilidad de gestionar la modularidad inherente en el diseño.

2.4 Herramienta de síntesis de alto nivel

En la metodología de diseño en alto nivel es necesario una herramienta de síntesis de alto nivel que transforme la especificación algorítmica en descripción RTL. Entre las diferentes herramientas disponibles para realizar este tipo de síntesis se puede citar a Cadence Stratus High-Level Synthesis y Xilinx Vivado HLS. Otro ejemplo es Mentor Graphics Catapult.

En este apartado se describen las dos primeras herramientas y se justifica su elección para este trabajo en función de sus características. Para este trabajo no se ha tenido en cuenta Mentor Catapult.

2.4.1 Cadence Stratus High-Level Synthesis

Cadence Stratus HLS es una herramienta de síntesis en alto nivel que, a partir de un algoritmo en SystemC, genera automáticamente su implementación *hardware* a nivel RTL. Al tratarse de modelado con un alto nivel de abstracción, se consigue realizar diseños mucho más rápido que otras herramientas de síntesis RTL. Además, Stratus HLS permite la exploración de múltiples versiones del algoritmo y diferentes arquitecturas para lograr los requisitos de área, potencia y latencia, facilitando la exploración del espacio de diseño.

Cadence Stratus HLS integra la verificación del diseño mediante el uso de *testbenches*. Esta verificación se puede realizar tanto en el modelo de comportamiento en SystemC, así como en el modelo RTL después de la síntesis y en la implementación en Verilog, reutilizando el *testbench*. Cadence Stratus proporciona soporte de múltiples simuladores como Cadence Incisive, Synopsys VCS o Mentor ModelSim. En este trabajo se ha utilizado Cadence Incisive como entorno de simulación de referencia.

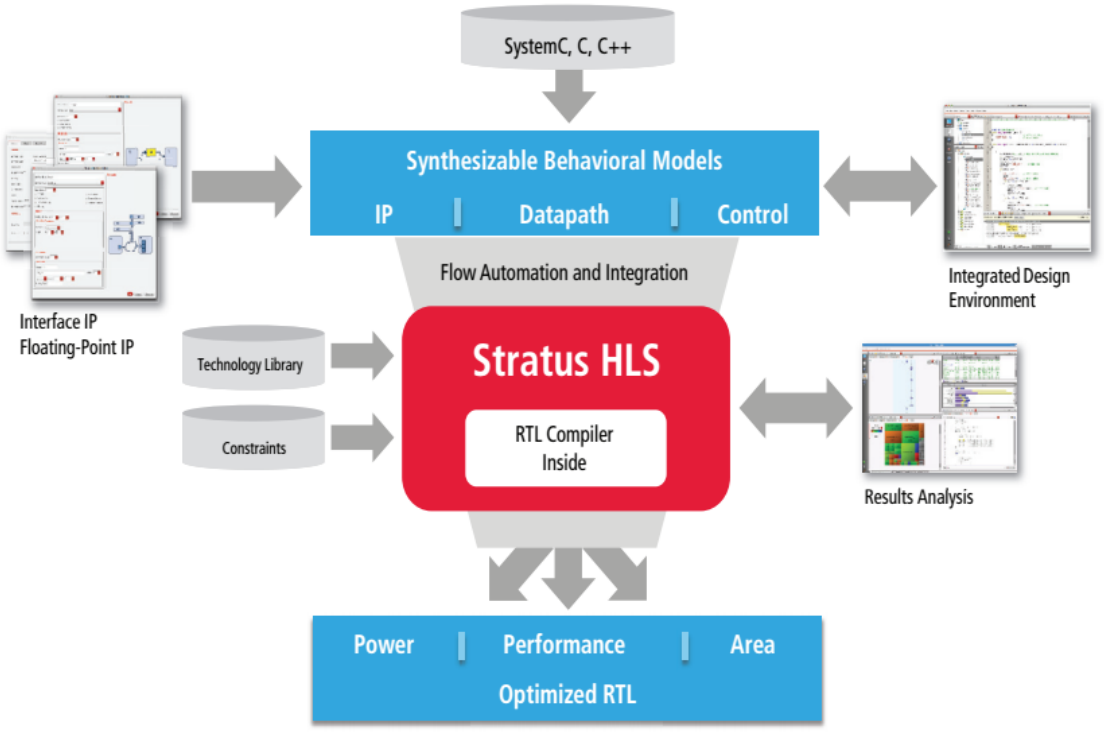


Figura 3: Flujo de diseño de Stratus [23]

Por otro lado, Stratus HLS también da soporte a herramientas de síntesis lógica como Cadence RTL Compiler, Synopsys Design Compiler, Xilinx Vivado o Altera Quartus,

pudiendo producir así la implementación a nivel lógico de cualquier implementación RTL generada por esta herramienta.

Otro de los aspectos claves de Cadence Stratus es su capacidad de depurado del modelo original ya sea en C++ o SystemC, soportando el depurado interactivo del diseño. Además, proporciona vistas de esquemáticos y de formas de onda del comportamiento en C++ o RTL y en muchos diseños se pueden obtener resultados del consumo de potencia [23].

2.4.2 Xilinx Vivado HLS

Vivado HLS permite la síntesis de la arquitectura del sistema a partir de especificaciones C, C++, SystemC u OpenCL. Permite un alto nivel de abstracción de la descripción algorítmica, de los tipos de datos y de sus interfaces. Facilita al diseñador la toma de decisiones gracias a las librerías que posee y las múltiples directivas de síntesis permiten conseguir los mejores resultados.

Vivado HLS permite continuar el flujo de diseño (Figura 4) de manera sencilla y crear bloques IP de la especificación en C, C++ o SystemC y ser volcada directamente en el dispositivo.

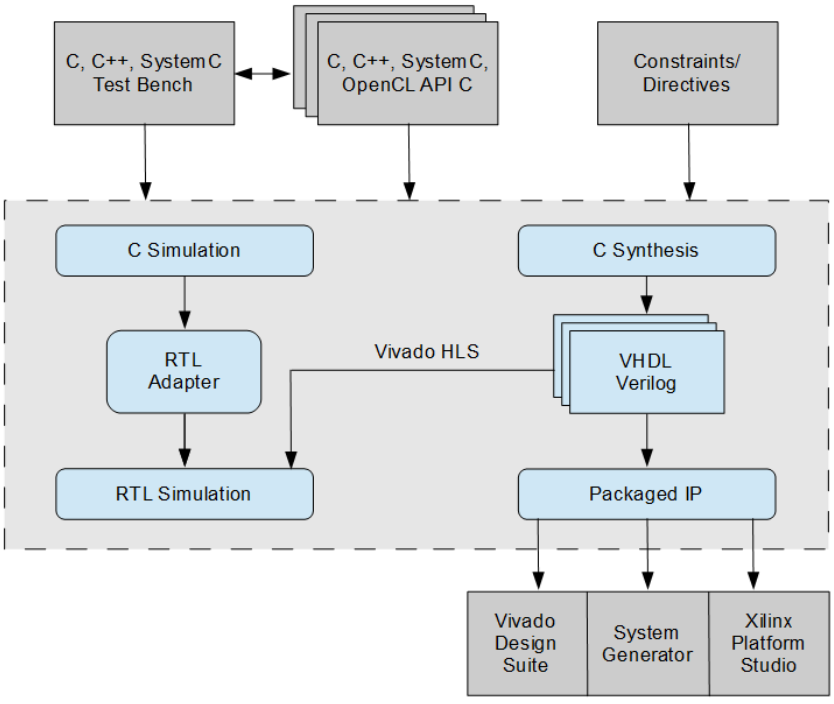


Figura 4: Flujo de diseño Vivado HLS [24]

2.4.3 Elección de la herramienta de síntesis en alto nivel

Existen varias ventajas y desventajas en ambas herramientas que se han considerado para su uso en este proyecto. Se ha decidido usar Cadence Stratus como herramienta de síntesis en alto nivel en lugar de Vivado HLS por las razones que a continuación se expresan.

Como restricción de partida el diseño se va a realizar en SystemC. Ello condiciona sobremanera la elección de la herramienta de síntesis de alto nivel. Si bien Vivado HLS soporta SystemC, la herramienta Stratus incluye todo un ecosistema de utilidades de desarrollo con soporte nativo de SystemC. Ello hace que las facilidades de depurado, la calidad final de los resultados y su flexibilidad sean más apropiados al proyecto.

Por otra parte, Xilinx Vivado HLS consigue frecuencias de trabajo superiores debido a que todos los procesos y la heurística están orientados a los dispositivos del fabricante.

Igualmente, indicar que en el grupo se tiene experiencias en ambas herramientas. Este diseño incluye bloques que han sido diseñados en SystemC usando el anterior entorno de Cadence, C-to-Silicon. La utilización de Cadence Stratus requiere la migración de los diseños desde C-to-Silicon a la herramienta Cadence Stratus.

2.5 Herramienta de síntesis lógica

Una vez que se ha generado la descripción RTL en la síntesis de alto nivel, se procede a realizar la síntesis lógica para continuar con el flujo de diseño. Las herramientas de las que se dispone para realizar esta síntesis son Synopsys Synplify Premier Design Planner y Xilinx Vivado. Ambas herramientas son complementarias en el flujo de diseño.

2.5.1 Synopsys Synplify Premier Design Planner

Synplify Premier de Synopsys es una herramienta de diseño sobre FPGA que cuenta con un entorno de implementación y depurado de diseños basados en FPGA. Esta herramienta permite localizar de manera sencilla errores en el diseño y optimizar el resultado de la implementación lógica del diseño.

Además, realiza la síntesis lógica del bloque IP partiendo de la descripción RTL generada por la herramienta de síntesis de alto nivel. Como resultado, genera un *netlist* optimizado en formato EDIF del diseño con el que se puede exportar el bloque IP a herramientas de terceros.

2.5.2 Xilinx Vivado

Xilinx Vivado es un entorno integrado en el que se pueden realizar diferentes tareas de diseño. Una de las tareas es la creación de la plataforma, donde se integran los diferentes bloques IP. Otra tarea clave es su síntesis e implementación, con objeto de obtener el *bitstream* que configura el dispositivo FPGA. Este objetivo se realiza en varias fases, incluyendo la síntesis lógica, análisis e implementación (*placement*, *routing*, *optimization*), ofreciendo tareas de optimización tanto de recursos como temporal o de consumo de potencia.

2.5.3 Elección de la herramienta de síntesis lógica

Ambas herramientas tienen la capacidad de optimizar las prestaciones temporales y de recursos. Se ha optado por utilizar la herramienta Synopsys Synplify debido a los mejores resultados obtenidos en cuanto al comportamiento temporal y uso de recursos y la facilidad de analizar temporalmente los bloques IP desarrollados. Synopsys Synplify permite analizar las rutas críticas de un bloque antes de ser implementadas en la FPGA y posibilita la aplicación de distintas estrategias de optimización a cada bloque IP. Xilinx Vivado ha sido utilizado para la síntesis de la plataforma teniendo en cuenta los recursos que usa cada bloque.

2.6 Integración de la plataforma: Vivado Design Suite

Para realizar el diseño de la plataforma final se utiliza la herramienta Vivado Design Suite, que cuenta con un entorno de desarrollo integrado (IDE) tanto para la parte *hardware* de la plataforma como para su parte *software*, aspecto que comentaremos más adelante. Esta herramienta permite la inclusión de distintos bloques prediseñados en la plataforma que se está diseñando. Por otra parte, también cuenta con la opción de crear

bloques IP propios realizados en Vivado HLS o incluso a partir de su *netlist* en formato EDIF realizados por herramientas externas.

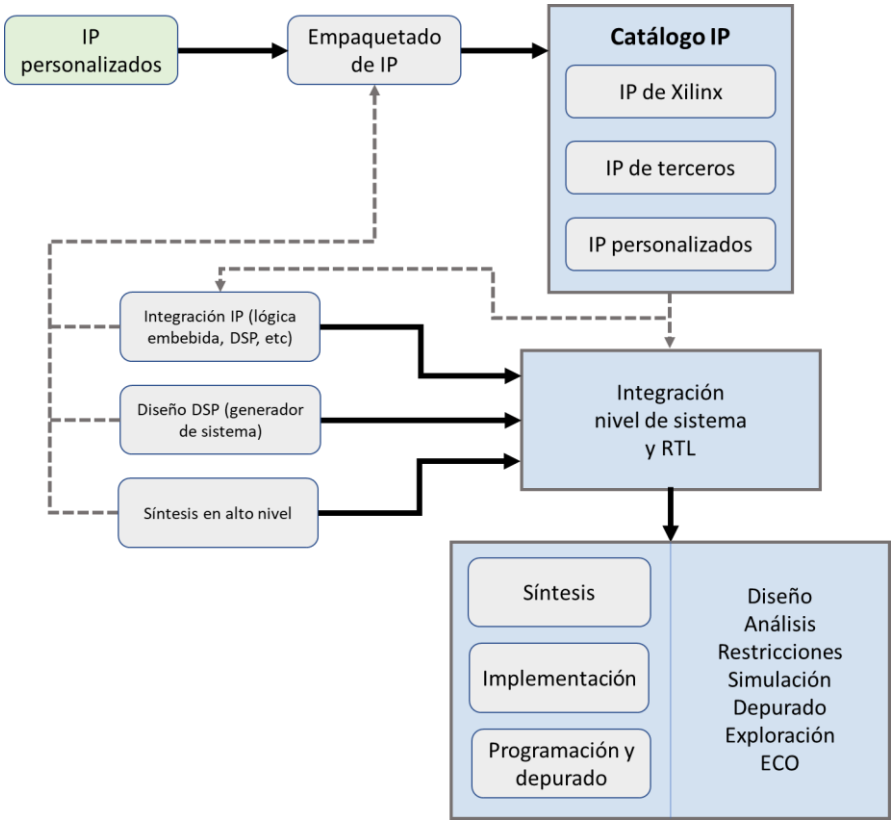


Figura 5: Flujo de diseño de Vivado Design Suite. Adaptada de [22]

Además, Vivado Design Suite proporciona un entorno que permite sintetizar, implementar, programar y depurar el proyecto sobre la FPGA de manera sencilla y obtener análisis temporales, de recursos o de potencia de la plataforma final.

2.7 Desarrollo de *software* empotrado, Vivado SDK

Xilinx cuenta con la herramienta Vivado SDK para el desarrollo *software* de aplicaciones empotradas para los microprocesadores que están integrados en su ecosistema. En este proyecto se ha usado la familia Zynq. Algunas de las características con las que cuenta SDK son un editor de código de C y C++ basado en Eclipse IDE, entorno de compilación, depurador de código y control de versiones, entre otros. Otra de las características que posee SDK es la depuración por JTAG sobre la placa de prototipado, programación de memorias flash o, *drivers* para los distintos bloques IP del catálogo.

Desde esta herramienta se puede realizar la programación de la FPGA a través del archivo *bitstream* que se ha generado en Vivado.

2.8 Flujo de diseño

El flujo de diseño que se ha seguido durante el desarrollo de este trabajo se muestra en la Figura 6. A partir de una descripción algorítmica en alto nivel escrita en SystemC, se ha realizado su síntesis en alto nivel con la herramienta Cadence Stratus siguiendo diversas estrategias para optimizar el diseño. Se han seguido varias iteraciones de modelado, síntesis y verificación en alto nivel hasta alcanzar la arquitectura deseada. Las restricciones impuestas al diseño tienen como objetivo minimizar la latencia del sistema con objeto de conseguir altas tasas de tráfico deseadas. La simulación incluye tanto la simulación funcional como la simulación RTL, una vez sintetizado el diseño.

La microarquitectura obtenida a nivel RTL sirve de entrada para la herramienta Synopsys Synplify con objeto de realizar la síntesis lógica, siguiendo una serie de estrategias para conseguir mejores resultados en ciclo de reloj y recursos. Synopsys Synplify genera un archivo EDIF que contiene el *netlist* del bloque para ser integrado en la plataforma FPGA.

Para la creación de la plataforma, se han importado los bloques diseñados en el catálogo de Xilinx Vivado. La plataforma se crea haciendo uso del bloque de procesamiento y sus bloques asociados, los bloques IP que conforman la arquitectura de comunicaciones basados en AMBA AXI4 y otra serie de bloques auxiliares. El entorno de creación basado en bloques posee procedimientos automáticos para la conexión de buses, verificación de reglas de diseño y otras facilidades. El entono soporta jerarquía y la creación de bloques complejos. La plataforma final se implementa mediante las herramientas de síntesis y de implementación. Todas estas tareas se realizan en el entorno Vivado IDE.

Por último, se realiza el desarrollo de la aplicación empotrada a partir de la API generada por el sistema (*Board Support Package*) en la herramienta SDK. Indicar que igualmente se realiza el depurado de la integración hardware/software mediante la utilización de puntos de ruptura y ejecución paso a paso en software y la visualización del estado de las señales usando un analizador lógico integrado (ILA).

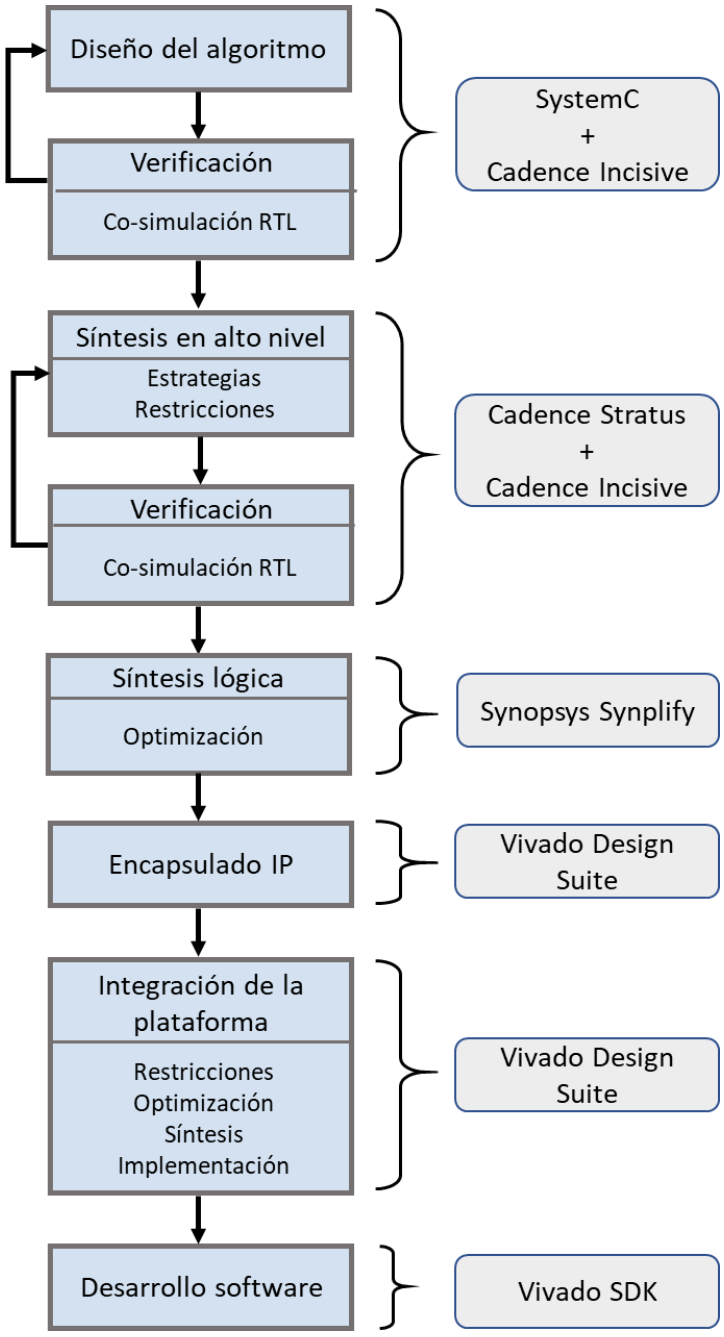


Figura 6: Diagrama del flujo de diseño

2.9 Conclusiones

Se ha visto con detalle el flujo de diseño que se utiliza en este proyecto, empezando con el modelado de los bloques en desarrollo que se realiza en SystemC al permitir un mayor control a la hora de planificar tareas e implementar interfaces.

Una vez que se tiene el bloque modelado y verificado, se realiza la síntesis de alto nivel con la herramienta Stratus de Cadence, obteniendo el diseño en RTL. El principal

condicionante al usar esta herramienta ha sido el lenguaje usado al modelar, así como las opciones que ofrece durante la síntesis.

La síntesis lógica de los bloques se ha realizado en Synopsys Synplify debido a que esta herramienta aporta unos resultados favorables en cuanto a optimización y frecuencia.

Por último, la integración de los distintos bloques IP para conformar la plataforma final se ha realizado en Vivado Design Suite y el desarrollo del *software* empotrado sobre Vivado SDK.

Capítulo 3. Zynq All Programable SoC

3.1 Introducción

En este capítulo se explica en detalle la plataforma SoC FPGA utilizada en el proyecto. Se presenta la parte del sistema de procesamiento y la parte de lógica programable, explicando las interfaces AXI, necesarias para la comunicación entre ambas partes.

3.2 SoC Programable Xilinx Zynq 7000

Los sistemas de la familia Zynq 7000 están compuestos por un *System on Chip* en el que en un único circuito integrado incorpora una FPGA y dos núcleos microprocesadores ARM Cortex-A9 con sus correspondientes extensiones NEON y otros bloques de control. Ambas partes están conectadas a través de interfaces AMBA AXI4, permitiendo la comunicación *hardware*/*software* con coprocesadores integrados en un mismo espacio de memoria. Ello permite manejar los elementos del sistema.

La arquitectura de los dispositivos Zynq está organizada en dos grandes bloques: Sistema de Procesamiento (PS) y Lógica Programable (PL). Ambos bloques pueden usarse de manera conjunta o de forma independiente, pudiendo realizar diseños complejos en un paradigma de integración *hardware*/*software*. Estas plataformas cuentan, por tanto, con la flexibilidad que ofrecen los núcleos procesadores y la escalabilidad y prestaciones que ofrece el paralelismo de la lógica programable. Una ventaja adicional y que facilita su

adopción en términos de TTM, es la ventaja en comparación con los sistemas basados en ASIC, ya que reduce el tiempo de desarrollo y el coste.

La arquitectura de este dispositivo se muestra en la Figura 7, donde se diferencian los dos bloques que conforman la arquitectura y se interconectan con las interfaces AMBA AXI. La parte de procesamiento cuenta con dos núcleos ARM Cortex A9, además de distintos periféricos.

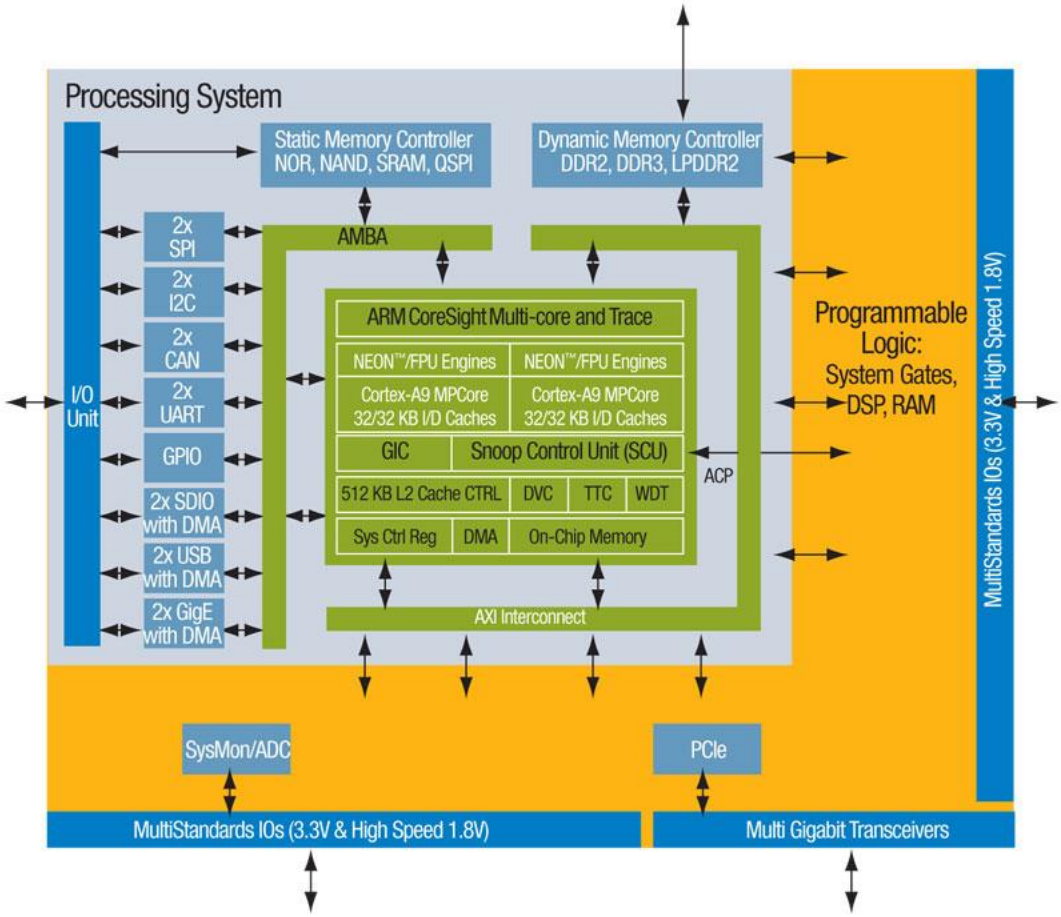


Figura 7: Arquitectura de Zynq 7000 [24]

3.2.1 Sistema de Procesamiento (PS)

En los dispositivos Zynq se distingue la parte de procesamiento de la lógica programable debido a que posee una arquitectura *hard processor*, contando con dos núcleos de procesamiento dedicados, comparándose con la gama de MicroBlaze que son *soft processor*, la cual cuenta con un procesador implementado en la parte de la lógica programable del dispositivo.

Los núcleos ARM Cortex-A9 pueden operar a una frecuencia de hasta 1 GHz. Una particularidad de estos dispositivos es que da la posibilidad de configurar el sistema para ejecutar un único procesador, elegir cuál de los dos seleccionar o también proporcionar multiprocesamiento simétrico (SMP) o asimétrico (AMP).

A través de la herramienta Xilinx *Software Development Kit* (SDK) se puede desarrollar el *software* para los procesadores ARM, ya que el compilador soporta el desarrollo de aplicaciones empotradas para diferentes procesadores integrados en el ecosistema de Xilinx. Además, es capaz de generar el lincador a medida del sistema.

EL PS se organiza en diferentes bloques, tal como se indica a continuación:

- Unidad de Procesamiento de Aplicación (APU)
    - ARM Cortex-A9 MPCore, que implementa la arquitectura ARN v7. Incluye el coprocesador NEON, memorias Cache L1 y L2 y *timers.*
    - Elementos de gestión del sistema:
        - Registros de Control de Nivel de Sistema (SLCRs) que permiten controlar el comportamiento completo del PS.
        - La Unidad de Control de *Snoop* (SCU) permite la transparencia de datos y la coherencia entre los procesadores.
        - El Puerto de Coherencia del Acelerador (ACP) que, a través de una arquitectura basada en maestro/esclavo, realiza la conexión entre los bloques PL y PS, donde el maestro es la parte lógica y el esclavo el sistema de procesamiento.
        - El Controlador de Interrupciones Generales (GIC) que se encarga de manejar las interrupciones de tres tipos distintos: interrupciones de periféricos privados (PPI), interrupciones generadas por *software* (SGI) e interrupciones de periféricos compartidos (SPI).
        - Temporizadores y *watchdog.*
        - Controlador DMA compartido para PS y PL.
        - Memoria SRAM integrada de 256 KB con paridad y doble puerto.
    - Interfaces de memoria: DDR Controller, Quad-SPI controller, Static Memory Controller (SMC).

- Periféricos de E/S: El PS es el encargado de manejar las interfaces con los componentes externos. Esta comunicación entre el PS y las interfaces externas se consigue a través de entradas y salidas multiplexadas (MIO), que se componen de 54 pines que permite una conexión flexible debido a que es el usuario quien define el mapeado entre el periférico y el pin. También cuenta con entradas y salidas de propósito general (GPIO) como son botones, *switchs* y LEDs que pueden ser usadas de diferentes maneras. Los periféricos que ofrece el dispositivo son los siguientes:
  - 2x SPI (*Serial Peripheral Interface Bus*)
  - 2x I2C (*Inter-Integrated Circuit*)
  - 2x Bus CAN (*Controller Area Network*)
  - 2x UART (*Universal Asynchronous Receiver-Transmitter*)
  - GPIO (*General Purpose I/O*): 54 canales MIO, 64 canales EMIO
  - 2x SDIO (*Secure Digital Input Output*)
  - 2x USB
  - 2x Gigabit Ethernet

3.2.2 Lógica programable (PL)

La parte de lógica programable está formada por una estructura FPGA, que se compone de distintos bloques lógicos configurables (CLB) unidos entre sí por una matriz de conexiones programables, multiplexores y bloques de entrada/salida como se muestra en la Figura 8. Un CLB está compuesto por dos *slices* que alberga cada una 4 LUTs, 8 Flip-Flops y sumadores para funciones aritméticas. Las LUTs son recursos flexibles que permiten implementar memorias ROM, RAM, registros de desplazamiento y, de manera combinada, memorias de mayor tamaño.

Aparte de estos bloques, existen dos componentes de propósito especial: *Block* RAM para uso extensivo de memoria y bloques DSP para aritmética de alta velocidad. Las *Block* RAM pueden implementar memorias RAM, ROM o FIFOs, cada una de ellas pueden almacenar 36Kb de información, pudiéndose combinar para alcanzar mayores tamaños. El uso de *Block* RAM es una alternativa al uso de RAM distribuida implementada con LUTs.

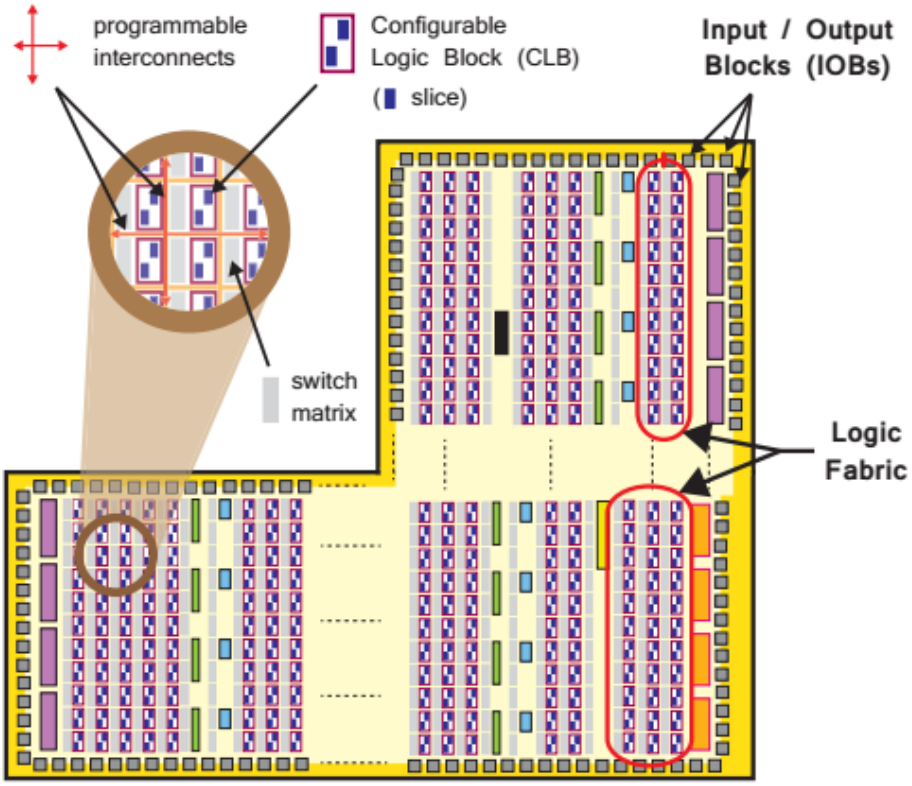


Figura 8: Parte lógica en detalle [22]

Dependiendo del dispositivo Zynq, se utilizan diferentes bloques FPGA, ya sea de la familia Artix-7 (bajas prestaciones) o Kintex-7 (compromiso consumo/prestaciones), y las prestaciones de la lógica programable varía.

3.3 Arquitectura de Interconexión entre PS y PL

La parte lógica y la de procesamiento del sistema pueden comunicarse a través de una arquitectura de interconexión basadas en interfaces AMBA AXI4, así como un puerto ACP, como se muestra en la Figura 9. Existen distintos tipos de interfaces entre el PS y el PL que se muestran continuación [25]:

- *General Purpose AXI.* Interfaces de propósito general con un tamaño de 32 bits para comunicaciones directas entre el PS y el PL. Existen cuatro interfaces de este tipo, de las cuales dos de ellas son maestras y otras dos son esclavas (considerando el PS como bloque maestro).
- *High Performance Ports.* Estas interfaces para alto rendimiento permiten que la lógica programable pueda acceder a las memorias DDR y OCM que se

encuentra dentro del PS a través de esta interfaz de alto rendimiento, ya que dispone de cuatro interfaces designadas del PL hasta el PS configurables a 32 o 64 bits y conectadas a través de controladores FIFO con una latencia reducida.

- *Accelerator Coherency Port:* Permiten conectar el PL a la SCU (*Snoop Control Unit*) de la APU, permitiendo dar coherencia entre la caché de la APU y los elementos que componen la parte del PL, a través de un bus de 64 bits asíncrono.

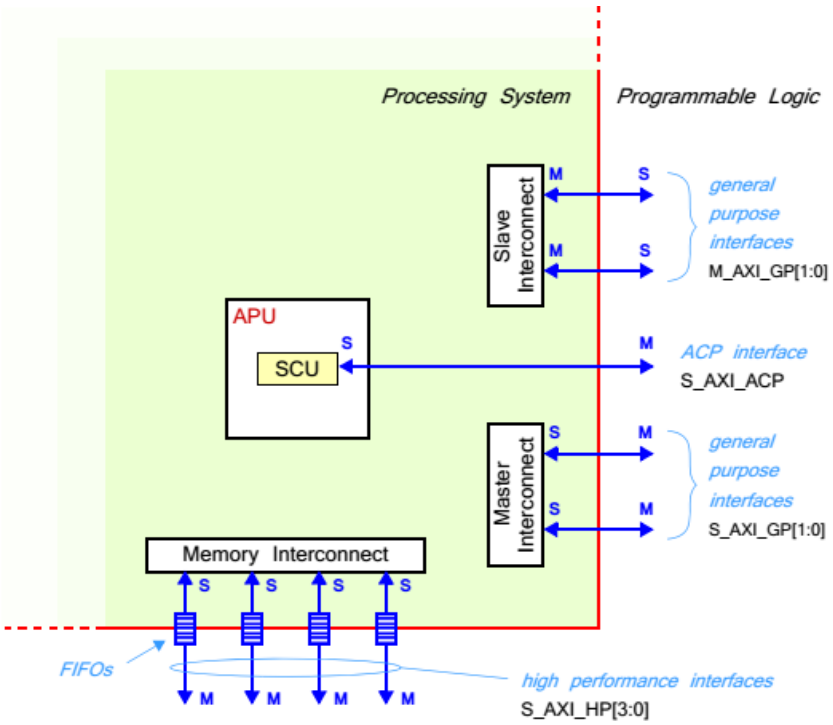


Figura 9: Conexión entre PS y PL [22]

3.4 Interfaces AMBA AXI

La interfaz AMBA AXI es un protocolo de bus que pertenece a la especificación AMBA (*Advanced Microcontroller Bus Arquitecture)* de ARM. Soporta transacciones de datos en alta velocidad con un menor consumo de silicio. Este protocolo se emplea en la comunicación entre la parte de procesamiento de sistema y la parte de lógica programable, además de la comunicación entre los distintos bloques o IP de la FPGA, facilitando la reutilización de los bloques IP.

Esta familia de buses para microcontroladores apareció por primera vez en 1996. En 2003 surge la primera versión de AXI en AMBA 3.0 y en el 2010 aparece la segunda versión de AXI, AXI4 [26].

El bus AMBA AXI4 cuenta con tres tipos de interfaces:

- AXI4MM. Destinado a comunicaciones de alto rendimiento, y mapeado en memoria, múltiples maestros y esclavos, con arbitraje.
- AXI4-Lite. Es una versión más simple del bus AXI4MM, con un único maestro y múltiples esclavos.
- AXI4-S. Es una versión dedicada a comunicaciones unidireccionales punto a punto, entre el productor (*Initiator*) y el consumidor (*Target*). Está orientado para aplicaciones en flujo de datos de alta velocidad.

El uso de este protocolo se traduce en mejoras de productividad, flexibilidad y disponibilidad. Es decir, al estar las interfaces AXI estandarizadas permite a los desarrolladores manejar un único protocolo para los IP. Por otro lado, al poseer distintas interfaces la elección de una u otra da la posibilidad de utilizar la que mejor se ajusta a la aplicación, maximizando el rendimiento del sistema. Xilinx proporciona un conjunto de bloques IP para dar soporte a este bus en la plataforma.

3.4.1 Interfaces mapeadas en memoria

Los protocolos AXI4MM y AXI4-Lite son interfaces mapeadas en memoria que necesitan un canal de direcciones en el que se indique la posición de memoria a la que se realizará la lectura o la escritura. Este tipo de interfaz permite compartir canales entre varios sistemas.

El protocolo AXI4MM es utilizado para requerimientos de altas prestaciones y permite comunicación bidireccional y mapeadas en memoria. Es capaz de realizar ráfagas de hasta 256 transferencias de datos con un tamaño de datos desde 32 bits a 1024 bits. En cambio, AXI4-Lite es una interfaz simplificada de la interfaz AXI4 que también es mapeada a memoria, pero a diferencia de la primera, solo permite la interconexión entre un maestro y varios esclavos, con transferencias de un único dato por conexión, es decir, sin posibilidad de realizar ráfagas. En este protocolo el tamaño del bus de datos se reduce a 32 o 64 bits.

Tanto la interfaz AXI4MM como AXI4-Lite consisten en 5 canales distintos:

- Canal de direccionamiento para la lectura: en este canal se incluye el bus de direcciones para la lectura de datos y las señales de control asociadas.
- Canal de dirección de escritura: en este canal se incluye el bus de direcciones de escritura de datos y las señales de control asociadas.
- Canal de escritura de datos: en este canal se realiza la transferencia de datos del maestro al esclavo.
- Canal de lectura de datos: en este canal se realiza la transferencia de datos del esclavo al maestro.
- Canal de respuesta de escritura: en este canal se indica si la escritura se ha realizado exitosamente o se ha producido algún error.

En la Tabla 2 se listan las señales que compone el protocolo AXI4-Lite.

Tabla 2: Señales AXI4-Lite

NOMBRE DE LA SEÑAL	DESCRIPCIÓN
SEÑALES GLOBALES	
S_AXI_ACLK	Reloj AXI
S_AXI_ARESET	*Reset*, activo a nivel bajo
SEÑALES DEL CANAL DE DIRECCIÓN DE ESCRITURA	
S_AXI_AWADDR	Dirección de escritura
S_AXI_AWVALID	Señal que indica que la dirección de escritura es válida y la información de control está disponible
S_AXI_AWREADY	Señal que indica que el esclavo está preparado para aceptar una dirección y una señal de control asociada
SEÑALES DEL CANAL DE ESCRITURA DE DATOS	
S_AXI_WDATA	Dato de escritura
S_AXI_WSTRB	Señal de *strobe*
S_AXI_WVALID	Señal que indica que el dato de escritura es válido y el *strobe* está disponible
S_AXI_WREADY	Señal que indica que el esclavo puede aceptar el dato de escritura
SEÑALES DEL CANAL DE DIRECCIÓN DE LECTURA	
S_AXI_ARADDR	Dirección de lectura
S_AXI_ARVALID	Señal que indica que la dirección de lectura es válida y la información de control está disponible
S_AXI_ARREADY	Señal que indica que el esclavo está preparado para aceptar una dirección y una señal de control asociada
SEÑALES DEL CANAL DE LECTURA DE DATOS	

NOMBRE DE LA SEÑAL	DESCRIPCIÓN
S_AXI_RDATA	Dato de lectura
S_AXI_RRESP	Respuesta de lectura. Señal que indica el estado de la transferencia de lectura
S_AXI_RVALID	Señal que indica que el dato de lectura es válido y la transferencia puede completarse
S_AXI_RREADY	Señal que indica al maestro que puede aceptar el dato de lectura y la información de respuesta
SEÑALES DEL CANAL DE RESPUESTA DE ESCRITURA	
S_AXI_BRESP	Respuesta de escritura. Indica el estado de la transferencia de escritura
S_AXI_BVALID	Señal que indica que la respuesta de escritura es válida
S_AXI_BREADY	Señal que indica al maestro que puede aceptar la información de respuesta

En ambas interfaces, AXI4MM y AXI4-Lite, la transferencia de datos ocurre de manera simultánea entre maestro y esclavo, poseyendo incluso distintas longitudes de datos. En la Figura 10 se muestra un proceso de lectura, donde se emplean el canal de dirección de lectura y el canal de lectura de datos.

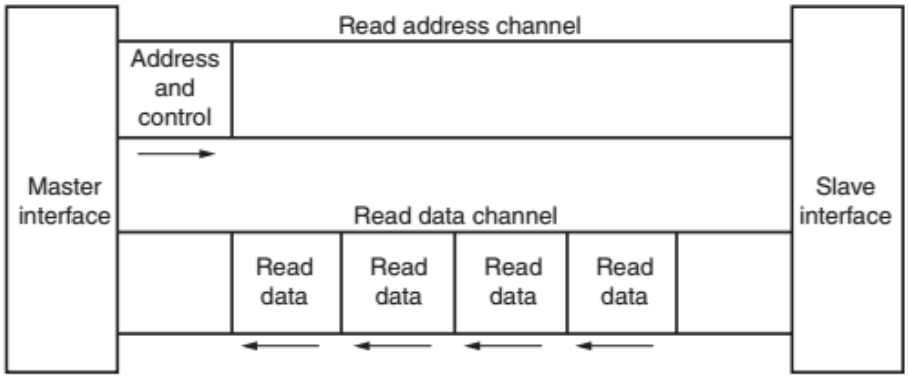


Figura 10: Proceso de lectura para AXI4 [26]

. En la Figura 11 queda reflejada la operación de escritura. En esta operación se utilizan el canal de dirección de escritura, el canal de escritura de datos y el canal de respuesta de escritura.

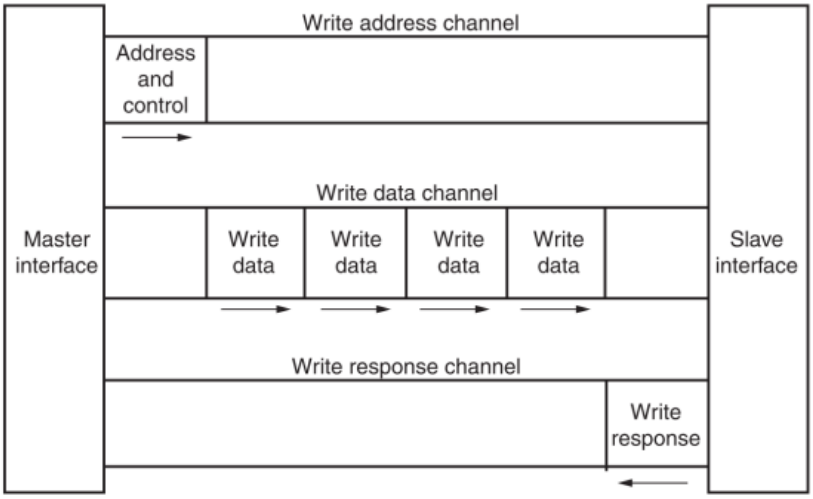


Figura 11: Proceso de escritura para AXI4 [26]

3.4.2 AXI4-Stream

Esta interfaz se usa en aquellos casos en que se precise transferencias de datos a alta velocidad, ya que soporta la realización de ráfagas sin restricciones de tamaño. La interfaz AXI4-Stream no está mapeada en memoria, por lo que se utiliza para flujos de datos directos, punto a punto, entre un iniciador (*Initiator*) y un destino (*Target*), sin requerir direccionamiento. Este protocolo de comunicación es unidireccional y utiliza señalización de control tipo *handshake* basado en las señales *ready/valid.*

Con el protocolo AXI4-Stream se permite transmitir cadenas de paquetes y especificar cuál es su final a través de la señal *TLAST*. Además, con esta señal se puede conocer el tamaño total del paquete.

Las señales que utiliza este protocolo se muestran en la Tabla 3.

Tabla 3: Señales del protocolo AXI4-Stream

SEÑAL	DESCRIPCIÓN
ACLK	Señal del reloj global
ARESETN	Señal de reset
TDATA	Bus de datos
TVALID	Indica que la fuente está preparada para enviar datos
TREADY	Indica que el destino está listo para recibir datos
TLAST	Indica el final de la trama
TSTRB	Indica qué bytes de los recibidos son bytes de datos o de posición

SEÑAL	DESCRIPCIÓN
TKEEP	Indica si los bytes recibidos en TDATA son válidos y deben ser transferidos
TID	Indica el número del canal o la ID de destino
TDEST	Indica información de la ruta del *stream*
TUSER	Indica información de usuario sobre la transacción

En la Figura 12 se muestra el diagrama temporal de una transferencia realizada usando el protocolo AXI4-Stream. Las transacciones de datos se realizan cada ciclo de reloj. Sin embargo, los *TDATA* P0 y P3 tienen más de un ciclo de duración debido a que, hasta que la señal que indica que el receptor está listo (*TREADY*) no se activa, no se realiza la transacción, siendo necesario mantener el dato en el canal. También se puede apreciar que si la señal *TVALID* no está activa, la fuente no tiene datos válidos para enviar al destinatario. Cuando no quedan más tramas que enviar, la señal *TLAST* se activa, como se muestra con P5.

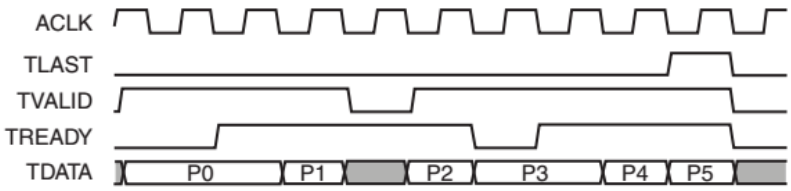


Figura 12: Transferencia AXI4-Stream [26]

3.5 Zynq ZC706

Se ha elegido como placa de desarrollo para este proyecto la Xilinx Zynq SoC ZC706 que incluye un dispositivo XC7Z045 FFG900 -2 (Figura 13). Integra en un único dispositivo físico una FPGA de la serie 7, familia Kintex y dos núcleos de procesamiento ARM Cortex A9 con una arquitectura ARMv7-A [27] de 32-bits.

Este dispositivo cuenta en la parte de lógica programable una Kintex-7, con las siguientes características principales:

- 350K celdas lógicas programables
- 218.600 LUTs
- 437.200 Flip-Flops

- 545 BRAMs de 19.1 Mb
- 900 DSP
- Bloque PCI Express Gen2 x8
- 2 conversores ADC de 12 bits, con 17 entradas analógicas configurables

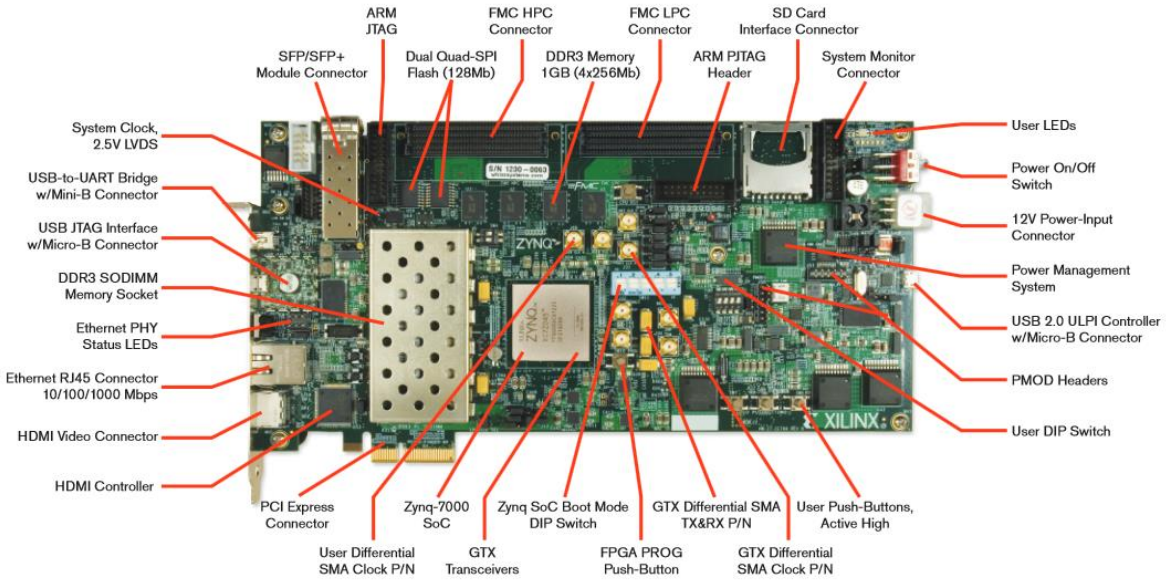


Figura 13: Tarjeta de prototipado ZC706 [28]

La arquitectura de los dispositivos Zynq permite realizar diseños basados en soluciones *hardware*/*software*. Mientras que en el PS los dos microprocesadores tienen la capacidad de ejecutar sistemas operativos completos como Linux, al mismo tiempo en el PL se puede implementar lógica de alta velocidad.

La elección de esta placa, en contraposición a las que ofrecen otras compañías, es debido a que el grupo de investigación posee un amplio recorrido de conocimiento y experiencia en las herramientas y dispositivos de Xilinx. En concreto, esta placa de prototipado ofrece los recursos necesarios y la velocidad para poder implementar el sistema a velocidades aceptables.

3.6 Conclusiones

En este capítulo se ha presentado el dispositivo SoC FPGA Zynq 7000 que ofrece un doble núcleo ARM Cortex-A9 con extensiones NEON. Además, permite la conectividad con diversos periféricos (UART, CAN, USB, etc), así como interfaces de red (PS). El dispositivo incluye una FPGA reconfigurable (PL). Además, se ha presentado la arquitectura de buses

del SoC para la comunicación entre el PS y el PL basado en el estándar AMBA AXI4. Se han explicado tanto las interfaces mapeadas en memoria como las basadas en flujos de datos para comunicación unidireccional (*Stream*). Por último, se presenta la tarjeta de prototipado ZC706 basada en el dispositivo XC7Z045 FFG900 -2.

Capítulo 4. Descripción de la arquitectura propuesta

4.1 Introducción

En este capítulo se presenta la arquitectura optimizada del sistema propuesta en el TFG, así como algunas variantes que se han descartado y la justificación de la elección de la arquitectura definitiva. También se explica el flujo de datos presente en la plataforma, desde la captura del paquete hasta que se devuelve a la red.

4.2 Análisis de arquitecturas DPI actuales

Los sistemas DPI implementan en *hardware* diferentes bloques de aceleración para mejorar el rendimiento del proceso de inspección de paquetes. Los paquetes son capturados por un controlador de red Gigabit Ethernet y del análisis de la cabecera se encarga el sistema de procesamiento. Para realizar la comunicación entre el controlador y el sistema de procesamiento es necesario disponer de bloques DMA para enviar el paquete capturado a memoria SRAM. La incorporación de estos bloques limita la frecuencia a la que puede operar la plataforma, a 200 MHz.

En la Figura 14 se muestra la arquitectura de un sistema DPI de referencia [16]. A través de un bloque TEMAC (*Tri-Mode Ethernet Media Access Controller*) se capturan los paquetes de la red y se almacenan en una memoria SRAM a través de un bloque DMA que se encarga de las transferencias desde el TEMAC. El sistema de procesamiento es el encargado de analizar la cabecera del paquete que se encuentra almacenado en memoria y dependiendo del resultado de este análisis puede requerir que el *payload* sea

inspeccionado por el bloque acelerador. Para comunicar el paquete almacenado en memoria con el bloque acelerador se utiliza otro bloque DMA.

Una vez que la carga útil del paquete ha sido examinada, el resultado puede ser que el paquete sea desechado o que deba ser devuelto a la red. Ahora el procedimiento es similar y el paquete almacenado en memoria se devuelva desde la memoria SRAM al controlador de red a través del bloque DMA. Este tipo de arquitectura está limitado a la latencia que supone almacenar el paquete en la memoria SRAM, la velocidad a la que trabajan los DMA y la velocidad del procesador. Además, hay que tener en cuenta que el sistema de procesamiento solamente podrá analizar la cabecera del paquete una vez que el paquete entero se encuentre almacenado en la memoria.

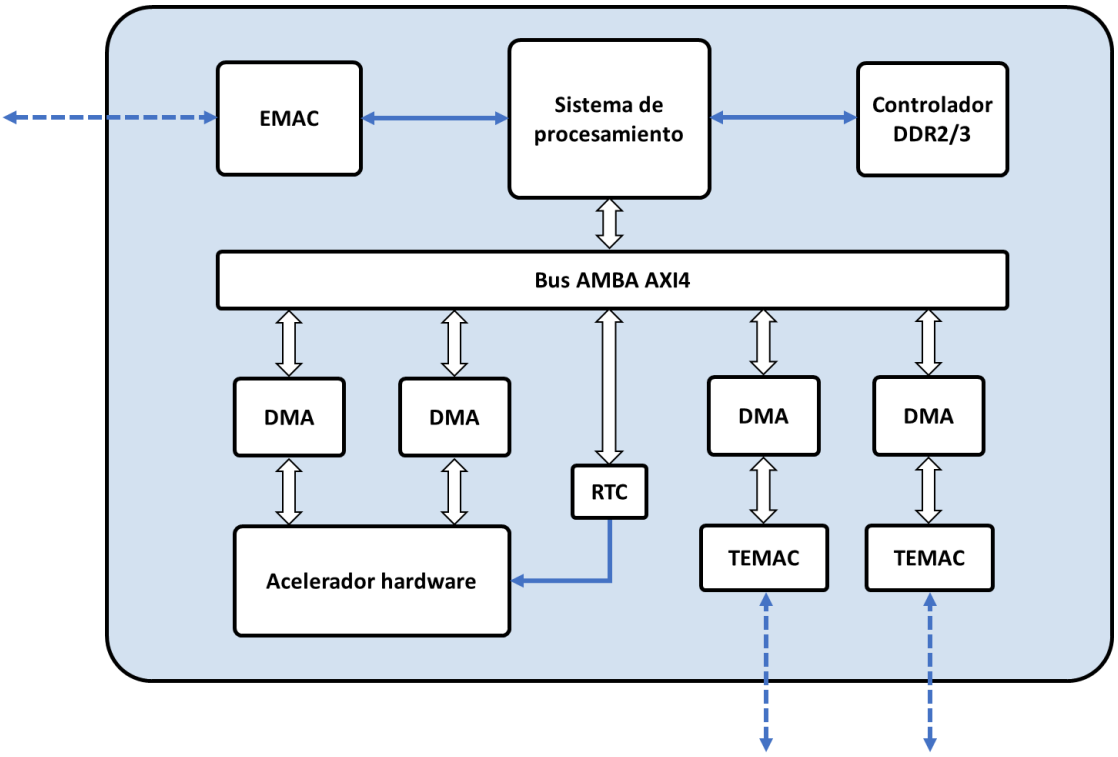


Figura 14: Arquitectura DPI de referencia

El objetivo de este TFG es optimizar la arquitectura de referencia presentada para DPI. Para ello, se sustituirá la implementación del análisis de la cabecera de *software* a *hardware*, con el fin de dejar de depender de los bloques DMA, suponiendo esto un aumento de las prestaciones de la plataforma. Es por ello que la arquitectura que se propone en los puntos siguientes de este capítulo cuenta con un bloque implementado en *hardware* denominado *Header Analyzer* que captura y analiza los paquetes recibidos por la

red, quedando el sistema de procesamiento relegado únicamente para la configuración de los distintos bloques de la arquitectura.

4.3 Flujo de datos de la plataforma propuesta

La arquitectura que se propone tiene como flujo de datos principal el que se muestra en Figura 15. A través del bloque TEMAC se recoge el paquete procedente de la red y mediante el bus AXI4 *Stream* se transmite el paquete al bloque *Header Analyzer.* Este bloque será el encargado de analizar la cabecera del paquete según unas restricciones preestablecidas. Si no existe coincidencia entre la cabecera del paquete y las restricciones, se enviará el paquete a la red a través de un bloque TEMAC de salida. En caso contrario, si existe coincidencia el paquete será enviado a un *Motor de Búsqueda* para realizar el análisis del *payload.* Para que realice el análisis de la carga útil del paquete es necesario eliminar la cabecera del paquete. Para ello, se incorpora un bloque denominado *Eliminar Cabecera* que eliminará la parte correspondiente a la cabecera del paquete.

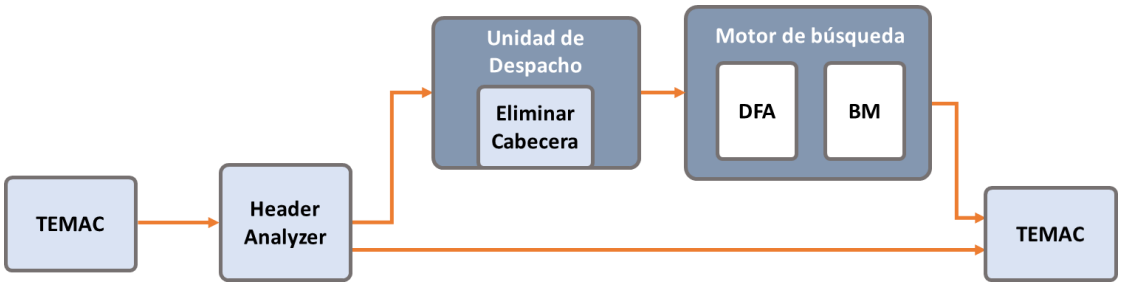


Figura 15: Diagrama de bloques de la plataforma

El *Motor de Búsqueda* cuenta con un número determinado de bloques de análisis y, para acelerar el proceso, los bloques implementan distintos tipos de algoritmos de búsqueda. Para realizar la selección del bloque de análisis al que el paquete debe ser enviado, es necesario poseer una *Unidad de Despacho*. Una *Unidad de Despacho* es la encargada de, a partir de una entrada, seleccionar una salida dependiendo de los criterios de análisis según la estrategia establecida. Estos criterios se reciben mediante la interfaz AXI-Lite disponible en la *Unidad de Despacho*.

Una vez se elija el bloque correspondiente, el paquete será enviado para su inspección y éste bloque enviará una respuesta sobre el análisis a la *Unidad de Despacho*. Como se ha indicado, el bloque de inspección profunda puede proporcionar dos tipos de

respuesta: desechar el paquete, terminando en la *Unidad de Despacho* el flujo de este paquete; o devolver el paquete a la red.

La *Unidad de Despacho* está diseñada de tal manera que se puede entender como dos bloques, uno que maneja la entrada de los datos y otro que maneja la salida. Estos bloques están conectados mediante dos memorias FIFOs que almacenan el mismo paquete en ambas memorias. Una memoria almacena el paquete en palabras de 32 bits que corresponde con las interfaces de entrada y salida de datos; la otra almacena los datos en palabra de 64 bits con el fin de adaptarlo con la señal que va hacia el *Motor de Búsqueda*. Aprovechando que el dato que es enviado a su inspección es el de la FIFO de 64 bits, se inserta el bloque *Eliminar Cabecera* entre la *Unidad de Despacho* y la memoria FIFO.

Cabe mencionar que, aunque la plataforma DPI cuenta con los diversos bloques mencionados, sólo los bloques TEMAC, *Header Analyzer* y *Eliminar Cabecera* son objeto del proyecto. El *Motor de Búsqueda* queda fuera del ámbito del proyecto mientras que la *Unidad de Despacho* se trata como una caja gris, debido a que ha sido necesario conocer su funcionamiento y se ha trabajado con ella con el fin de integrar el bloque *Eliminar Cabecera*, aunque sin realizar ningún tipo de modificación sobre la misma.

4.4 Estudio de alternativas en la arquitectura

Se han estudiado dos alternativas sobre cómo actuar en el caso de que el paquete supere la inspección profunda por el *Motor de Búsqueda* y deba ser enviado a la red. A continuación, se mostrarán y se justificará la elegida.

4.4.1 Arquitectura HA

En esta arquitectura propuesta (Figura 16), el bloque IP *Header Analyzer* es el único que tiene comunicación con los bloques TEMAC, siendo el encargado de sincronizar la salida de los paquetes hacia la red tanto de los que no necesitan una inspección profunda por parte de *Motor de Búsqueda* como los que sí, teniendo por tanto una única salida a la red que facilita la interconexión de los bloques.

Por lo tanto, una vez que la *Unidad de Despacho* reciba la respuesta por parte del *Motor de Búsqueda*, se debe enviar el paquete de nuevo al *Header Analyzer* para ser enviado por la interfaz de salida hacia el bloque TEMAC.

Pero esta arquitectura presenta una desventaja en cuanto a latencia se refiere, debido a que el paquete tiene que volver al bloque *Header Analyzer* antes de ser enviado a la red. Es simple en cuanto al flujo de control, pero incrementa la latencia en el flujo de datos en el caso de que el paquete sea analizado y al final deba ser devuelto a la red.

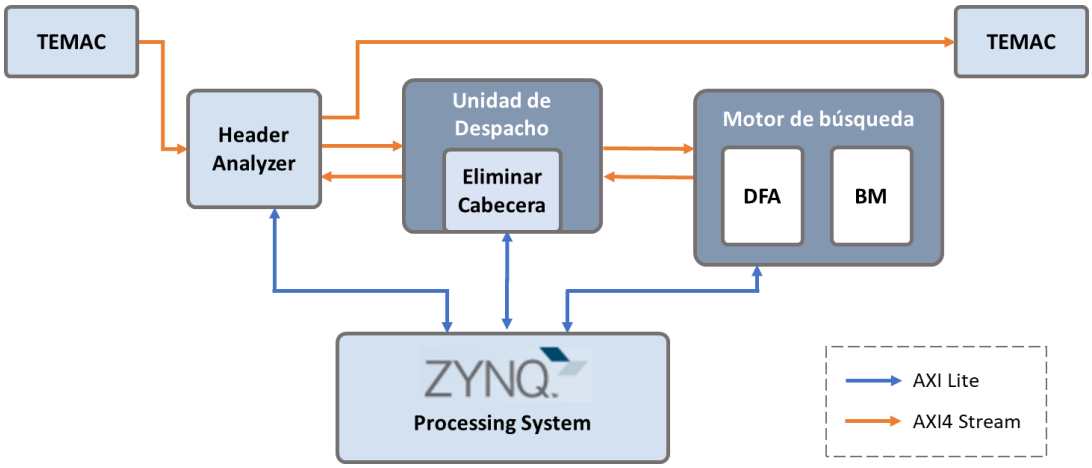


Figura 16: Arquitectura HA

4.4.2 Arquitectura DU

En esta propuesta (Figura 17), el paquete es enviado a la red desde la *Unidad de Despacho* una vez que el *Motor de Búsqueda* ha enviado la respuesta oportuna. Esto disminuye la latencia introducida por la arquitectura HA, ya que el paquete no debe volver a la unidad de análisis de cabecera, mejorando el rendimiento de la plataforma.

Sin embargo, esta arquitectura es más compleja en cuanto a esquemas de control, ya que existen dos bloques que requieren enviar datos hacia la red. Por lo tanto, se tienen dos flujos de datos hacia el TEMAC cuando este bloque cuenta con una única entrada. Para solucionar esta situación se utilizará un bloque IP denominado *AXI4-Stream Switch* [29]. Este bloque permite conmutar dos maestros a un esclavo que usan el protocolo AXI4-Stream para ser conectados al TEMAC. Para realizar la sincronización de lectura de las señales se pueden seleccionar diversos modos de arbitrajes. Se utilizará el protocolo de arbitraje Round-Robin que arbitra mediante la señal TVALID. Este método es el idóneo para

esta arquitectura debido a que verifica ambos maestros de manera cíclica en vez de seleccionar un sistema de arbitraje por prioridades en el que se pudiera colapsar el bloque con menor prioridad. El sistema está abierto a otros esquemas de arbitraje.

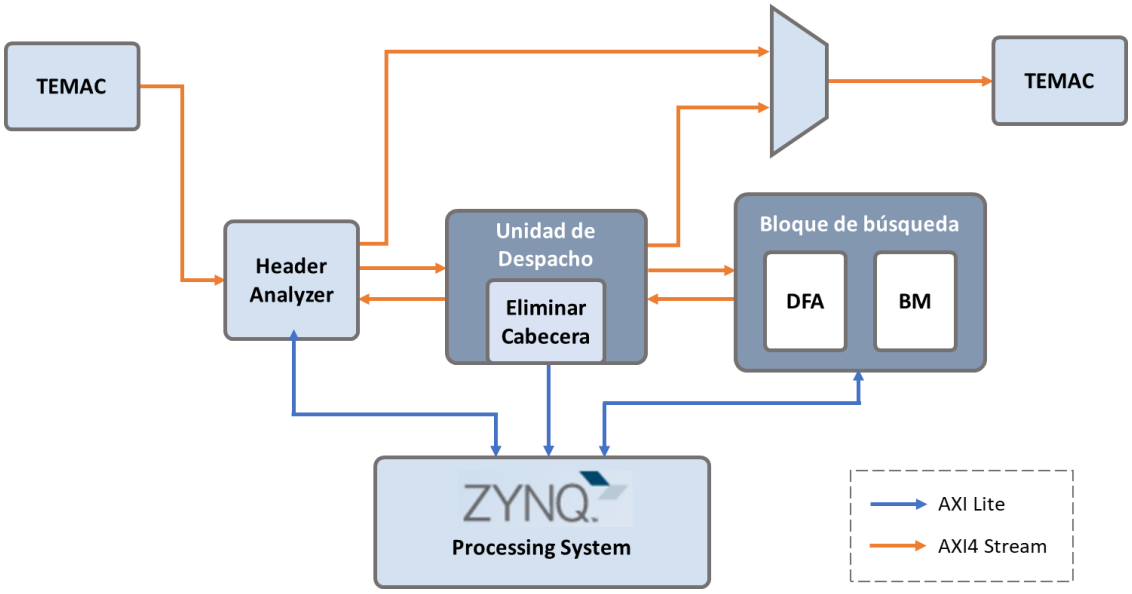


Figura 17: Arquitectura DU

Por lo expresado anteriormente y teniendo como ventaja principal la reducción de latencia de la plataforma, esta arquitectura ha sido la elegida para ser implementada.

4.5 Conclusiones

La arquitectura que se propone tiene como fin optimizar la latencia y evitar los cuellos de botella que se producen en otras arquitecturas. Los paquetes son capturados a través de un controlador de red y se envían al bloque *Header Analyzer* para realizar el filtrado de la cabecera. Si el paquete requiere una inspección profunda es redireccionado a la *Unidad de Despacho*, encargada de distribuirlo hacia el *Motor de Búsqueda*. Si el paquete no requiere un análisis en profundidad se devuelve a la red. Es en esta *Unidad de Despacho* donde el paquete es tratado con el fin de eliminar su cabecera, debido a que el *Motor de Búsqueda* sólo analiza el *payload.* Para realizar esta acción es necesario el bloque *Eliminar Cabecera.* La *Unidad de Despacho* puede desechar el paquete o enviarlo a la red dependiendo del resultado de la inspección. Para unificar las salidas a la red del bloque *Header Analyzer* y de la *Unidad de Despacho* se incorpora un *switch* que, a través de técnicas de arbitraje, habilita las interfaces.

Capítulo 5. Diseño de los bloques IP

5.1 Introducción

A lo largo de este capítulo se explica el funcionamiento del bloque *Header Analyzer,* responsable de capturar los datos de la red y analizar la cabecera del paquete entrante. Además, se ha desprendido de este bloque la acción que permite suprimir la cabecera del paquete Ethernet y se ha creado otro bloque denominado *Eliminar Cabecera*.

5.2 Descripción del bloque *Header Analyzer*

El bloque *Header Analyzer* se ha tomado de [17] con el fin de optimizar su funcionamiento para la implementación en un sistema DPI. El objetivo de este bloque es el de capturar los paquetes recibidos a través de la red, mediante bloques controladores MAC, obteniendo un flujo constante de datos entrantes y salientes. Una vez capturados se realiza un filtrado de paquetes al analizar la cabecera Ethernet y TCP/IP. Después de realizar distintas comparaciones con los campos de la cabecera se toma una decisión. Dependiendo de este resultado el paquete puede ser devuelto a la red o enviado a un bloque de inspección profunda.

En la Figura 18 se muestran los distintos sub-bloques que componen el bloque IP. A nivel de interfaces el bloque posee una interfaz AXI4-Lite necesaria para configuración del bloque IP, una interfaz de entrada AXI4-Stream de la cual recibirá el flujo de datos desde la red y dos interfaces de salida AXI4-Stream, para volcar los datos a la red o al analizador de paquetes.

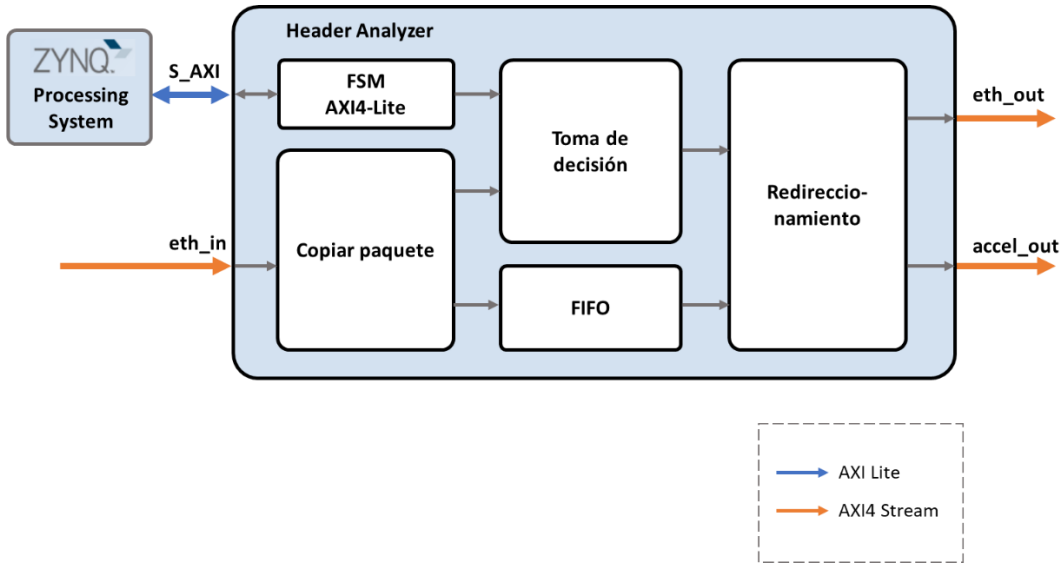


Figura 18: Arquitectura del bloque

Para realizar el filtrado, el paquete es dividido y tratado en palabras de 32 bits, en lugar de manejar el paquete completo. Esto mejora las prestaciones del sistema debido a que reduce las latencias totales al no tener que esperar por el paquete entero para empezar a realizar el filtrado. La información para analizar se encuentra en las primeras trece palabras; por lo tanto, mientras se realiza el filtrado a través de condiciones específicas, como el filtrado por protocolo, dirección de destino o incluso conjunto de distintos parámetros, el resto del paquete se va copiando para después ser reenviado.

En este proyecto se tratarán paquetes de tipo TCP/IP encapsulados en tramas Ethernet. Es por eso por lo que existen diversos campos a tratar: de la cabecera Ethernet se puede extraer tanto las direcciones MAC de origen o destino o su protocolo; de la cabecera IP se puede extraer campos importantes como la calidad de servicio, direcciones IP, la longitud del paquete o el tiempo de vida de éste. Por último, de la cabecera del paquete TCP también se pueden obtener las direcciones TCP de origen y destino.

En la Figura 19 se muestra cómo es el formato de la trama Ethernet que encapsula dentro de su campo de datos el paquete IP. Este paquete a su vez encapsula en sus datos el paquete TCP.

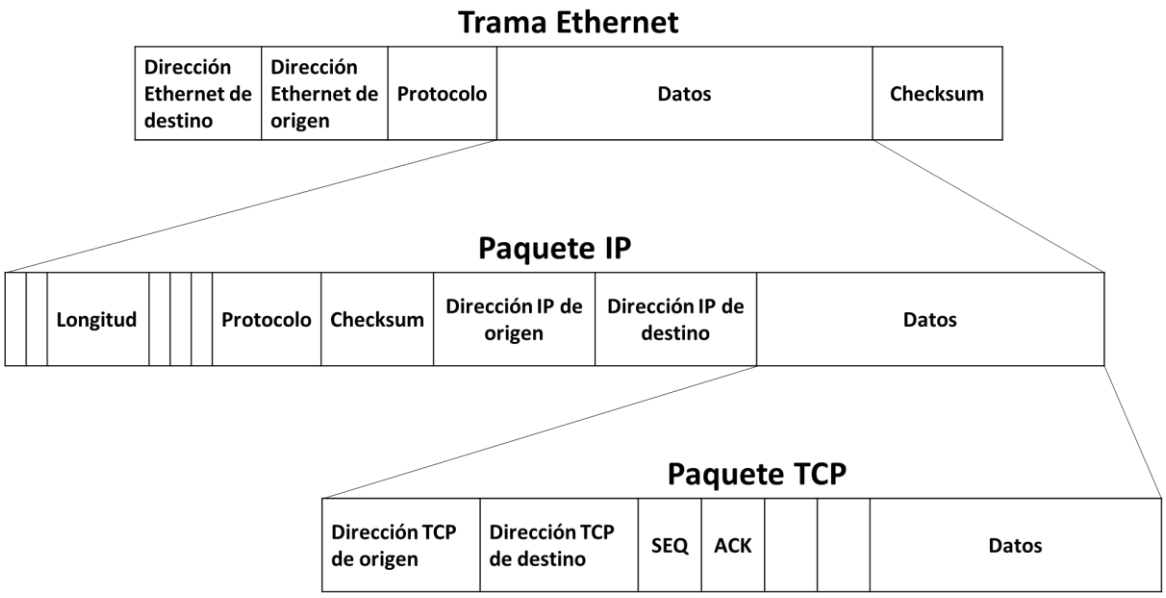


Figura 19: Encapsulado TCP/IP en trama Ethernet

5.3 Arquitectura del bloque *Header Analyzer*

El bloque de captura cuenta con un submódulo y tres procesos (Figura 20) que, en conjunto, define el funcionamiento del bloque. El submódulo proporciona un proceso asociado a la máquina de estados de la interfaz AXI4-Lite que será la encargada de configurar todo el bloque y está relacionado directamente con el proceso de toma de decisión, que realiza el filtrado del paquete. Los otros dos procesos son los encargados de copiar el paquete entrante y de redireccionarlo por la interfaz correcta dependiendo del resultado del análisis.

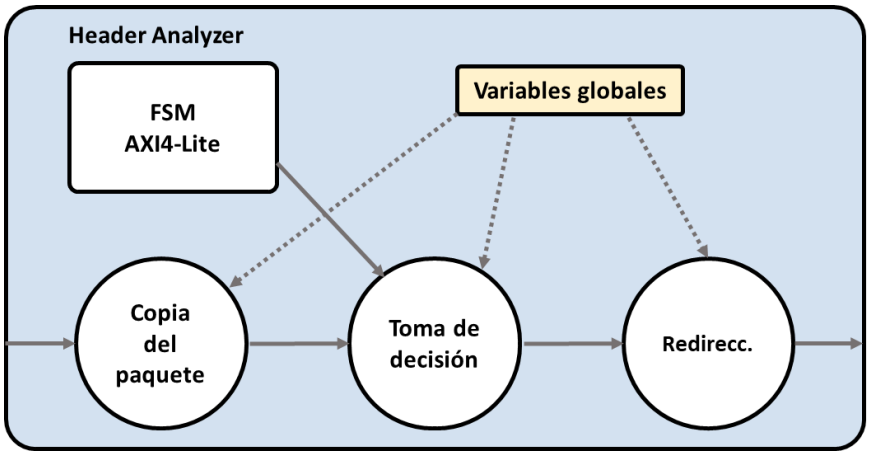


Figura 20: Diagrama de procesos del *Header Analyzer*

5.3.1 Módulo FSM AXI4-Lite

La configuración del bloque IP sobre los valores de los distintos campos de la cabecera del paquete se realiza a través de la interfaz AXI4-Lite. Esta interfaz tiene sus registros mapeados en memoria y a través del microprocesador se pueden modificar los valores de éstos, siendo el bloque IP quién lee este valor y lo compara con el campo correspondiente de la cabecera.

Los registros de configuración de los distintos campos de la cabecera que se han implementado en el bloque IP se muestran en Tabla 4.

Tabla 4: Registros de configuración

NOMBRE	DESCRIPCIÓN	TAMAÑO	DIRECCIÓN
CHECK_ETHER	Registro que determina si se realiza el análisis de la cabecera Ethernet	1 byte	0x00000000
DEST_MAC	Registro con la dirección MAC de destino	6 bytes	0x00000010
ORG_DEST	Registro con la dirección MAC origen	6 bytes	0x00000040
ETHERNET_TYPE	Registro con el tipo Ethernet	2 bytes	0x00000070
CHECK_NETWORK	Registro que determina si se realiza el análisis de la cabecera de red	1 byte	0x00001000
QOS	Registro con el valor de la calidad de servicio	1 byte	0x00001010
PROTOCOL	Registro donde se indica el tipo de protocolo se encapsula en la cabecera red	1 byte	0x00001018
IP_SRC	Registro con la dirección IP origen	4 bytes	0x00001040
IP_DEST	Registro con la dirección IP de destino	4 bytes	0x00001060

5.3.2 Proceso de copia del paquete

Este proceso es el encargado de copiar los datos entrantes y controlar las señales del bus para permitir la comunicación entre el bloque TEMAC y el bloque *Header Analyzer*, permitiendo un flujo de entrada de datos constante. El paquete entrante se divide en palabras de 32 bits y se almacena en una FIFO. Mediante la señal *TLAST* que proporciona el protocolo de comunicación se puede identificar el final de un paquete. Además de esto, también se realiza una copia local de las trece primeras palabras de cada paquete, denominada *shadow copy*, debido a que en la estructura de los paquetes TCP/IP estas trece

palabras contienen la información necesaria para realizar el análisis, encontrándose información de cabeceras del nivel de enlace y red.

Este proceso y el proceso de análisis de la cabecera del paquete están sincronizados a través de una señal de *start* que, una vez que las trece primeras palabras han sido copiadas, se activa permitiendo que el proceso de toma de decisión comience a verificar los distintos campos de la cabecera. Mientras se está analizando esta información, se puede ir almacenando en una FIFO el resto del paquete y una vez se tome una decisión y dependiendo de ésta, se redireccionará el paquete hacia la interfaz de red o al *Motor de Búsqueda* a través de la *Unidad de Despacho*.

5.3.3 Proceso de toma de decisión

En este proceso es donde se realiza el análisis y la toma de decisión. Como se comentó en el punto anterior, el proceso de copia y este proceso están sincronizados de tal forma que cuando se ha copiado la cabecera se comienza a extraer las trece primeras palabras del paquete donde se encuentra su información de cabecera. Posteriormente, compara el valor de cada uno de los campos con los valores predeterminados por el usuario que están almacenados en los registros de configuración del bloque, recibidos por el bus AXI4-Lite.

El resultado del análisis realizado se guarda en un registro interno que se comunica con el proceso de redireccionamiento. Al ir comparando campo por campo de la cabecera el valor se guarda en una misma variable, realizando una suma lógica de cada resultado. Por lo tanto, si uno de los campos coincide con el almacenado en el registro AXI4-Lite, el paquete debe ser enviado para su inspección al motor DPI. En caso contrario, si ninguno de los campos coincide, el paquete será reenviado a la red.

5.3.4 Proceso de redireccionamiento

Este proceso es el encargado de redireccionar el paquete hacia el bloque de análisis o a la interfaz de red, dependiendo del resultado del proceso anterior. A través de una señal de *start*, que controla el proceso de análisis, se sincronizan ambos procesos y se toma el valor resultante del proceso del filtrado de la cabecera guardado en un registro interno. Si

algún valor coincide en el análisis con los valores de la configuración, el proceso de redirección debe enviar el paquete al *Motor de Búsqueda* para un análisis más detallado. Si no coincide, se envía a la interfaz de red.

5.3.5 Interfaces entrada/salida

Como se comentó brevemente con anterioridad, el bloque IP cuenta con diversas interfaces de entrada y salida para su interconexionado con el resto de la plataforma. El bloque IP recibe el flujo de datos a través de un bloque TEMAC. Como son flujos de datos directos desde un maestro a un esclavo, se utiliza una interfaz de entrada AXI4-Stream para esta recepción de datos:

```
1    sc_in<sc_uint<32>>    data_in_ethernet_data;
2    sc_in<bool>           data_in_ethernet_valid;
3    sc_in<bool>           data_in_ethernet_last;
4    sc_out<bool>          data_in_ethernet_ready;
```

Código 1: Interfaz de entrada AXI4-Stream, *data_in_ethernet*

Si el paquete no requiere ser analizado en profundidad, es volcado a la red y para ello se utiliza una interfaz de salida AXI4-Stream, para realizar una conexión directa con el bloque TEMAC de salida:

```
1    sc_out<sc_uint<32>>    data_out_ethernet_data;
2    sc_out<bool>           data_out_ethernet_valid;
3    sc_out<bool>           data_out_ethernet_last;
4    sc_in<bool>            data_out_ethernet_ready;
```

Código 2: Interfaz de salida AXI4-Stream, *data_out_ethernet*

En el caso de que sea necesario analizar la carga útil del paquete, éste es enviado al bloque acelerador a través de una interfaz AXI4-Stream debido a que se requiere una transmisión de alta velocidad:

```
1    sc_out<sc_uint<32>>    data_out_accel_data;
2    sc_out<bool>           data_out_accel_valid;
3    sc_out<bool>           data_out_accel_last;
4    sc_in<bool>            data_out_accel_ready;
```

Código 3: Interfaz de salida AXI4-Stream, *data_out_accel*

Con el fin de realizar un bloque IP que no utilice una gran cantidad de recursos, se ha añadido la FIFO encargada de almacenar el paquete completo de manera externa. Se utilizan las siguientes interfaces para comunicarse con la FIFO:

```
1  //Interfaz de salida
2  sc_out<DATA>      din;
3  sc_out<bool>      we;
4  sc_in<bool>       almostfull_n;
5
6  //Interfaz de entrada
7  sc_in<DATA>       dout;
8  sc_out<bool>      re;
9  sc_in<bool>       empty_n;
```

Código 4:Interfaces entrada y salida de la memoria FIFO

Para realizar la configuración del bloque IP se ha implementado una interfaz esclava de AXI4-Lite con las siguientes interfaces:

```
1  //Write Address Channel
2  sc_in<sc_uint<32> >     S_AXI_AWADDR;
3  sc_in<bool>             S_AXI_AWVALID;
4  sc_out<bool>            S_AXI_AWREADY;
5
6  //Write Data Channel
7  sc_in<sc_uint<32> >     S_AXI_WDATA;
8  sc_in<sc_uint<32/8> >   S_AXI_WSTRB;
9  sc_in<bool>             S_AXI_WVALID;
10 sc_out<bool>            S_AXI_WREADY;
11
12 //Read Address Channel
13 sc_in<sc_uint<32> >     S_AXI_ARADDR;
14 sc_in<bool>             S_AXI_ARVALID;
15 sc_out<bool>            S_AXI_ARREADY;
16
17 //Read Data Channel
18 sc_out<sc_uint<32> >    S_AXI_RDATA;
19 sc_out<sc_uint<2> >     S_AXI_RRESP;
20 sc_out<bool>            S_AXI_RVALID;
21 sc_in<bool>             S_AXI_RREADY;
22
23 //Write Response Channel
```

```
24  sc_out<sc_uint<2> >     S_AXI_BRESP;
25  sc_out<bool>            S_AXI_BVALID;
26  sc_in<bool>             S_AXI_BREADY;
```

Código 5: Interfaces AXI4-Lite

En la Figura 21 se muestra a modo de resumen de lo comentado anteriormente las distintas interfaces entrada/salida que compone el bloque *Header Analyzer.*

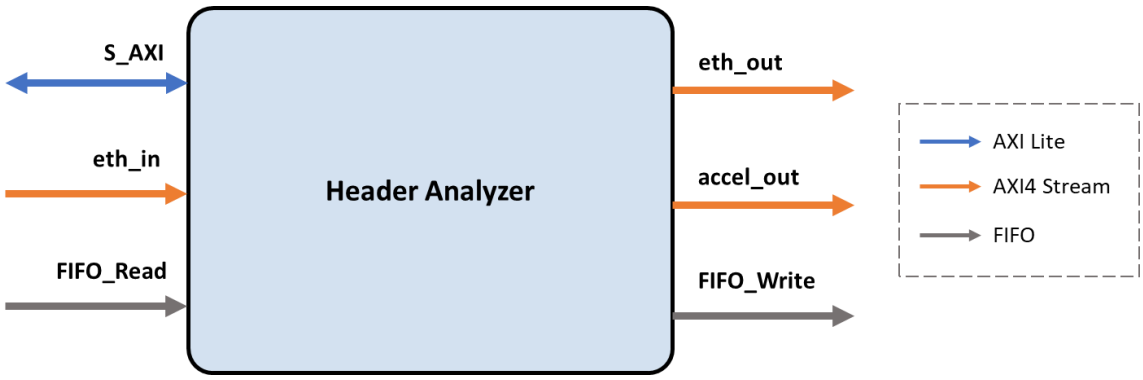


Figura 21: Interfaces *Header Analyzer*

5.4 Modelado del bloque *Eliminar Cabecera*

Si se da el caso de que el paquete debe ser analizado por el *Motor de Búsqueda*, este paquete debe ser tratado; es decir, se debe eliminar la cabecera del paquete debido a que los analizadores solo realizan la inspección al *payload* o carga útil del paquete.

Esta acción se puede realizar en el proceso de redireccionamiento del bloque IP de captura de paquetes, pero presenta distintas desventajas. Al eliminar la cabecera para que el *Motor de Búsqueda* analice el *payload*, el bloque *Header Analyzer* recibe de nuevo el paquete una vez se ha terminado el análisis. Si el resultado es el de continuar el paquete por la red, es necesario incorporar la cabecera de nuevo al paquete. Para realizar esta incorporación de la cabecera, el paquete debe encontrarse almacenado en una memoria en el bloque IP y debe esperar la respuesta del *Motor de Búsqueda* para tomar una decisión sobre el redireccionamiento. Esto se traduce en posibles cuellos de botella en este bloque debido a la limitación del tamaño de la memoria que guardaría el paquete, congestionando por tanto todo el sistema al esperar por la finalización del análisis.

Otro inconveniente que se presenta es el hecho de que el envío del paquete hacia la red se realiza desde el proceso de redireccionamiento del bloque *Header Analyzer*. Este

proceso recibe paquetes desde dos fuentes distintas: desde el proceso de análisis del *Header Analyzer* y desde el bloque de *Unidad de Despacho*. Ésto provoca de nuevo cuellos de botella en el proceso de redireccionamiento. Por ejemplo, la *Unidad de Despacho* envía un paquete al *Header Analyzer* para su envío a la red y desde el proceso de toma de decisión existe un paquete que debe ser analizado en profundidad, el paquete que llegue el último debe esperar a que el primero sea redireccionado, aunque no sean enviados por la misma interfaz.

Por último, la incorporación de una memoria para almacenar el paquete completo en el bloque *Header Analyzer* supone un aumento considerable en el uso de los recursos del sistema, debido que para evitar congestión de las comunicaciones se debe implementar una memoria lo suficientemente grande para que puede almacenar varios paquetes.

Por estos inconvenientes, se ha tomado la decisión de crear un bloque aparte para realizar esta acción. Con el fin de ahorrar recursos, este bloque se acopla junto al bloque de *Unidad de Despacho* que posee este sistema. Este bloque contiene dos memorias FIFO implementadas de manera externa, en la que en una se almacena el paquete completo, debido a que el analizador de paquetes solo devuelve una respuesta, y en la otra se almacena el paquete sin la cabecera. Por lo tanto, una vez que se obtenga una respuesta por parte del analizador, se recupera el paquete de la memoria que contiene el paquete con la cabecera. Al estar implementada de manera externa la FIFO se ha creado el bloque con interfaces de entrada y salida FIFO con el fin de evitar modificar la *Unidad de Despacho*.

Por lo tanto, el objetivo de este bloque es desechar la cabecera del paquete Ethernet. La cabecera está comprendida en los 54 primeros bytes, como se muestra en la Figura 22.

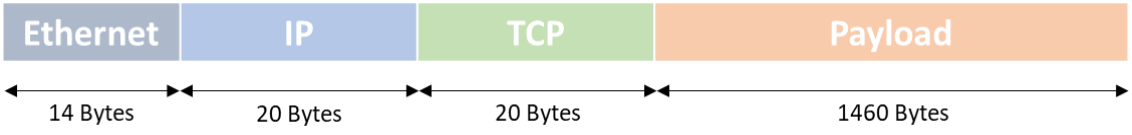


Figura 22: Tamaño de las cabeceras

La funcionalidad del bloque posee un único proceso en donde al comenzar la recepción de paquetes se activa un contador que se incrementa con cada palabra recibida. Las trece primeras palabras son desechadas y cuando el contador sea mayor que el tamaño

de la cabecera se activa la interfaz de salida para almacenar el paquete sin cabecera en la FIFO externa de la *Unidad de Despacho*.

Al contrario que el bloque de captura, este bloque trata palabras de 65 bits debido a que la *Unidad de Despacho* maneja este tamaño en sus interfaces. De ellos, 64 bits son los datos del paquete y el último bit corresponde a la señal *TLAST* que se implementa con el protocolo AXI4-Stream. Por lo tanto, el valor de este último bit se comprueba en cada recepción de palabra y si está a '1' entonces el contador se resetea para comenzar a eliminar la cabecera del nuevo paquete.

5.4.1 Interfaz entrada/salida

Este bloque cuenta con dos únicas interfaces ya que se encuentra situado entre la salida hacia la FIFO en el bloque de *Unidad de Despacho* y la entrada de dicha FIFO. Es por eso por lo que se han diseñado estas interfaces como si fuera una FIFO.

```
1  //Interfaz de entrada
2  sc_in<DATA>       dout;
3  sc_in<bool>       re;
4  sc_out<bool>      empty;
5
6  //Interfaz de salida
7  sc_out<DATA>      din;
8  sc_out<bool>      we;
9  sc_in<bool>       almostfull;
```

Código 6: Interfaces entrada/salida de *Eliminar Cabecera*

En la Figura 23 se muestra las distintas señales que conforman las interfaces de entrada y salida del bloque IP.

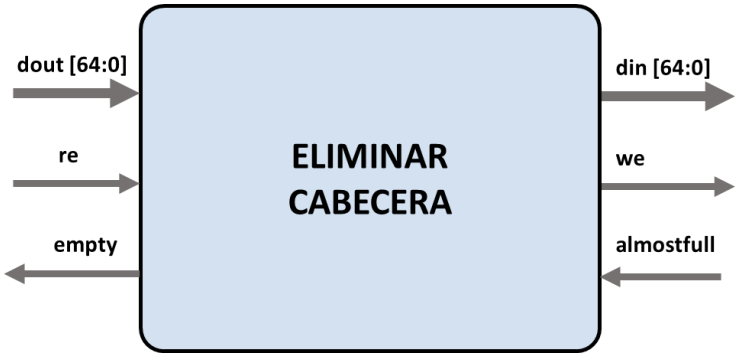


Figura 23: Bloque IP de *Eliminar Cabecera*

5.5 Simulación SystemC

Para realizar la simulación del módulo que se ha desarrollado en SystemC es necesario realizar una serie de *testbenches* en los que se contemplen todos los casos posibles para comprobar que se está realizando el filtrado de manera correcta.

Cadence Stratus integra los procesos de síntesis y simulación en SystemC del módulo en desarrollo. Para realizar esta simulación es necesario tener los siguientes archivos además del *testbench* [30]:

- `main.cpp`: en este fichero se encuentra las funcionalidades para comenzar la simulación, inicializar señales y objetos en la simulación, crear nuevas instancias del sistema para la simulación en SystemC y eliminar el sistema de la memoria una vez que la simulación haya finalizado.
- `system.cpp`: este fichero define el módulo *top-level* que instancia las señales y conexiones entre el diseño y el *testbench*. Contiene las definiciones del reloj y del *reset* del diseño.
- `system.h`: archivo de cabecera de la declaración del sistema, diseño y *testbench*.
- `tb.cpp`: este archivo es el *testbench* del módulo que envía y recibe datos del diseño. Contiene la función SystemC *sc_stop()* para finalizar la simulación.
- `tb.h`: archivo de cabecera del *testbench*.

Además, para visualizar los resultados de la simulación del sistema, Cadence Stratus genera de manera automática un fichero VCD (*Value Change Dump*) con las formas de onda de las señales del módulo una vez que la simulación ha finalizado. Por defecto, el fichero incluye todas las señales internas y las interfaces, pero se puede añadir señales adicionales a través de la función *sc_trace().*

Las formas de onda de este fichero se visualizan usando el programa GTKWave [31] que permite abrir ficheros VCD.

5.5.1 Simulación del bloque *Header Analyzer*

Para comprobar el sistema, se ha utilizado como archivos *testbench* un fichero de estímulos que comprueba el filtrado del paquete analizando todos los campos de la cabecera y otro que simula el comportamiento de la FIFO externa que almacena el paquete recibido. En el fichero de estímulos, a partir de un archivo de texto se ha almacenado un paquete Ethernet, el cual se leerá y se enviará al diseño en palabras de 32 bits. Sabiendo el contenido de este paquete, se indicará en el test que analice la dirección de destino MAC del paquete para verificar que realiza el filtrado y envía el paquete por la interfaz de salida al analizador. Para realizar esta tarea, es necesario indicar a través de la interfaz AXI4-Lite que se analice la capa de enlace y pasarle la dirección MAC de destino sabiendo que la dirección es la siguiente: FC:AA:14:74:61:E0.

```
1  //Habilitar análisis de la capa de enlace
2  address.write(CHECK_ETHER_REG_ADDR);
3  write_data.write(0x1);
4  RNW.write(false);
5  go.write(true);
6  wait();
7  wait();
8  while(!done)wait();
9  go.write(false);
10
11 //Asignar dirección MAC
12 address.write(DEST_MAC_REG_ADDR_LSB);
13 write_data.write(343171552);//0x147461E0
14 RNW.write(false);
15 go.write(true);
16 wait();
17 wait();
18 while(!done)wait();
19 go.write(false);
20
21 wait(4);
22
23 address.write(DEST_MAC_REG_ADDR_MSB);
24 write_data.write(64682); //0xFCAA
25 RNW.write(false);
```

```
26  go.write(true);
27  wait();
28  wait();
29  while(!done)wait();
30  go.write(false);
```

Código 7: Asignación de análisis y dirección MAC

A partir del archivo VCD generado en esta simulación se puede comprobar el funcionamiento del sistema. Se han hecho diversas pruebas, comprobando que funciona el filtrado para cada uno de los campos de la cabecera. En la Figura 24 se muestra cómo el bloque IP recibe a través del registro S_AXI_WDATA de la interfaz AXI4-Lite la habilitación del análisis de la capa de enlace y como dos ciclos después queda registrado en la señal *check_ether*, tras esperar los *wait()* de las líneas 6 y 7 del Código 7.

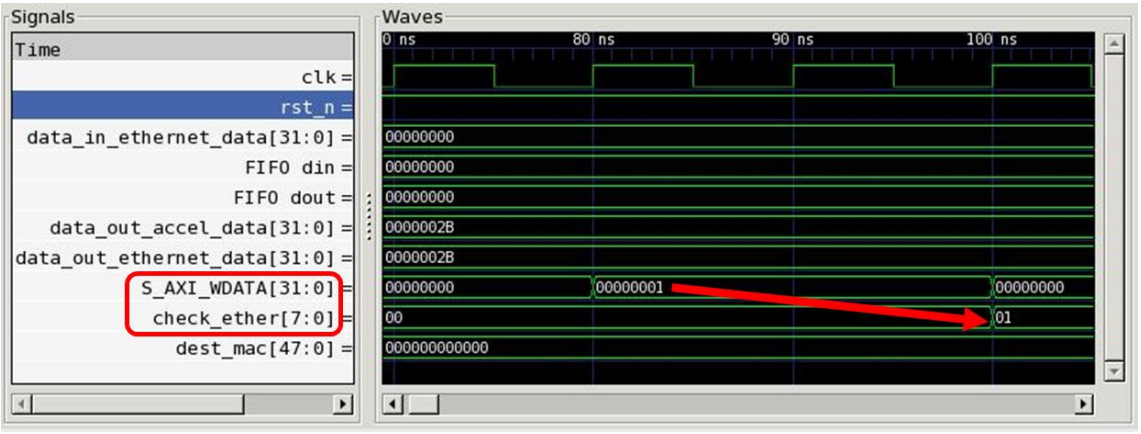


Figura 24: Simulación *Header Analyzer*. Activación *check_ether*

También se puede apreciar como al indicar la dirección MAC de destino a filtrar el bloque lo recibe de nuevo a través de la señal S_AXI_WDATA de la interfaz AXI4-Lite y se guarda en la señal *dest_mac*. Cabe destacar que la dirección MAC es enviada en dos transacciones y así se muestra en Figura 25.

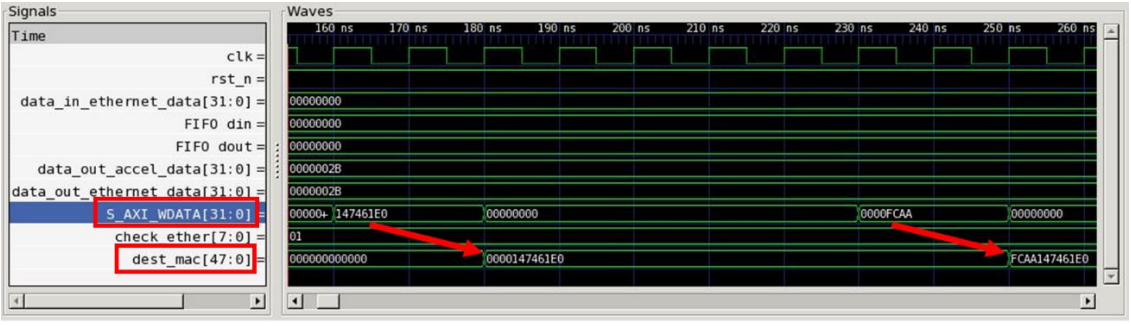


Figura 25: Simulación *Header Analyzer*. Asignación dirección MAC de destino

Sabiendo el contenido de la trama Ethernet se puede comprobar que se está recibiendo a través de la interfaz AXI4-Stream de entrada los datos sin perder palabras. Otro punto importante a comprobar es si la salida hacia la FIFO externa está transmitiendo cada palabra sin pérdidas. Esto se puede verificar con la Figura 26.

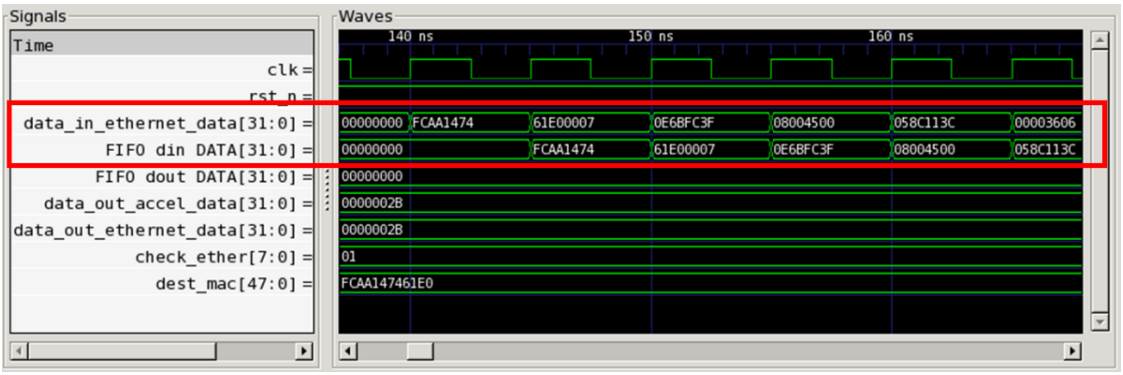


Figura 26: Simulación *Header Analyzer*. Recepción de paquetes

El modelo de comportamiento de la FIFO permite verificar que, el paquete una vez almacenado en esta memoria, es recibido de nuevo por el bloque para ser enviado a través de la interfaz correspondiente. Como se puede apreciar en la Figura 27, la dirección MAC de destino asignada para el filtrado es la FC:AA:14:74:61:E0, que corresponde con la primera palabra y la mitad de la segunda. Es por ello por lo que la trama sale a través de la interfaz *data_out_accel_data*. Con esto, y realizando las pruebas de los otros campos de la cabecera, se comprueba que el bloque en desarrollo filtra adecuadamente los paquetes recibidos.

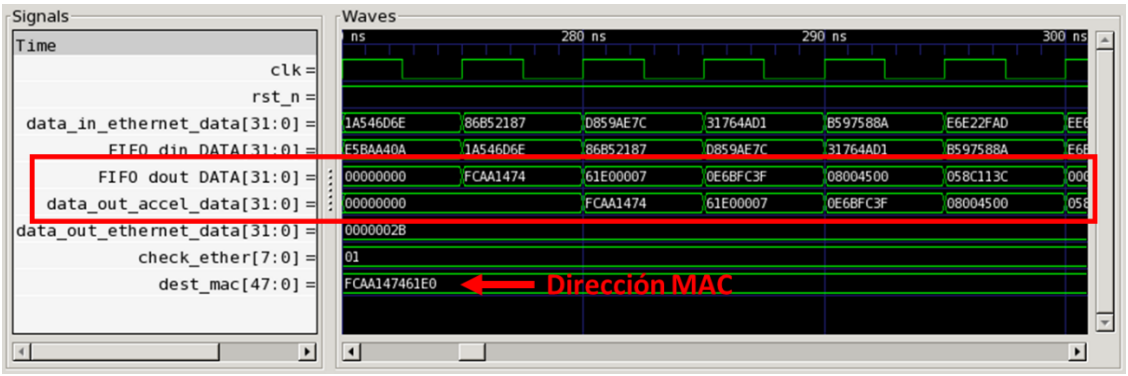


Figura 27: Simulación *Header Analyzer*. Salida de paquetes por interfaz del acelerador

A continuación, se comprobará que, si la dirección MAC de destino indicada para realizar el filtrado no corresponde con la dirección del paquete, este se enviará por la interfaz de salida hacia la red. Para ello se le indicará al bloque a través de la señal

S_AXI_WDATA del protocolo AXI4-Lite que la dirección MAC de destino es FC:BB:14:74:61:E0 como se muestra en Figura 28.

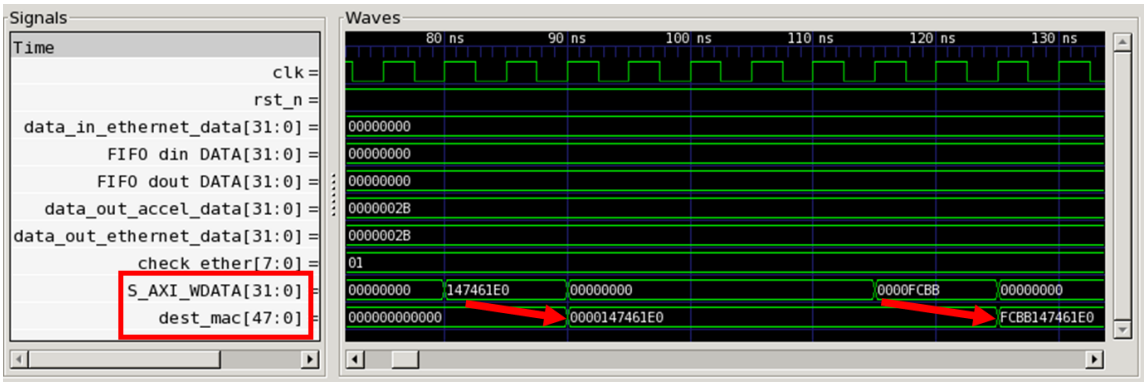


Figura 28: Simulación *Header Analyzer*. Asignación dirección MAC de destino (II)

En la Figura 29 se muestra como el paquete, después de realizar el filtrado, sale a través de la interfaz de salida AXI4-Stream en dirección al bloque TEMAC para ser devuelto a la red, debido a que las direcciones MAC de destino no coinciden.

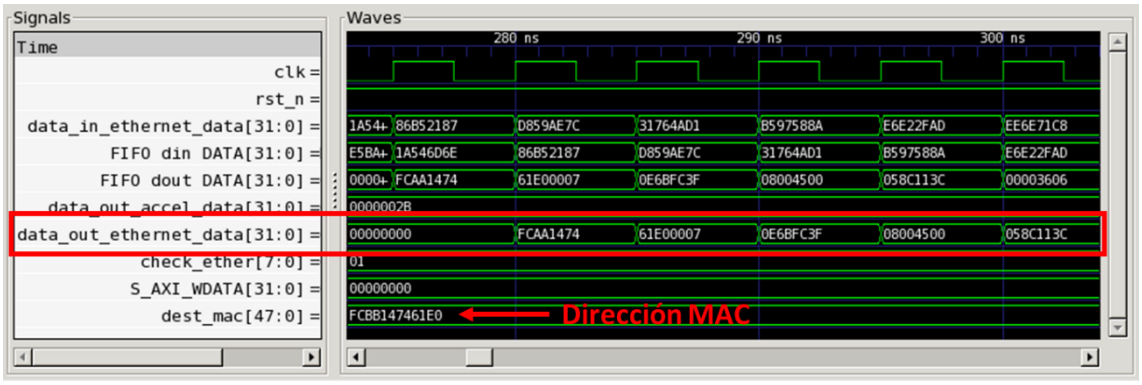


Figura 29: Simulación *Header Analyzer*. Salida de paquetes por interfaz de red

5.5.2 Simulación del bloque *Eliminar Cabecera*

Para la comprobación del funcionamiento de este bloque se creará un *testbench* que genere los estímulos correspondientes para que el bloque en desarrollo elimine la cabecera del paquete recibido. De nuevo, se tomará la trama Ethernet desde un fichero de texto almacenándolo en un *array*. A partir de este texto, se realizará una conversión a hexadecimal y se guardará en un *array* denominado *packet* del cual se enviará su contenido al bloque en desarrollo.

Como la *Unidad de Despacho* toma datos de 64 bits, se han unido dos palabras 32 bits en una única palabra de 64 bits. A continuación, se enviará cada palabra a través de la interfaz de salida del *testbench* que está conectada con la interfaz de entrada del módulo en desarrollo.

```
...
//-------------------- Preparar mensajes -----------------/
// Convertir el valor ASCII del mensaje a hexadecimal
    DATA packet0[BUFFER_SIZE_TEST];
    for(int i = 0; (i < 200) || (line[i]== '\0') ; i++){
        for(int e = 0; e<8;e++){
            word1[e] = line[16*i+e];
        }
        for(int e = 8; e<16;e++){
           word2[e-8] = line[16*i +e];
        }
        temp1= strtoul (word1, NULL, 16);
        temp2= strtoul (word2, NULL, 16);

        packet[i].data.range(31,0) =(unsigned int)temp2;
        packet[i].data.range(63,32) =(unsigned int)temp1;

        wait();
        packet[i].tlast=false;
    }
    packet[i].tlast = true;

//Comenzar transferencia
    for(int i = 0; i<BUFFER_SIZE_TEST && ready_out.read(); i++){
        data_out.write(packet0[i]);
        valid_out.write(1);
        wait();
    }
    valid_out.write(0);
    wait();
 ...
```

Código 8: Unión de palabras de 32 a 64 bits y comienzo de la transferencia

A través del archivo VCD generado por Cadence Stratus se pueden comprobar que las formas de onda producidas por la simulación con el fin de verificar que se han eliminado

los 54 primeros bytes del paquete Ethernet que corresponde con la cabecera. Como se comentó anteriormente, el paquete es recibido en palabras de 64 bits, por lo tanto, la cabecera está contenida en seis palabras. En la Figura 30 se comprueba cómo se eliminan de manera correcta las seis primeras palabras del paquete y que la séptima palabra ya es enviada a través de la interfaz de salida del bloque. También se puede comprobar como este bloque posee una latencia de ocho ciclos desde que recibe el paquete hasta que lo envía a través de la interfaz de salida.

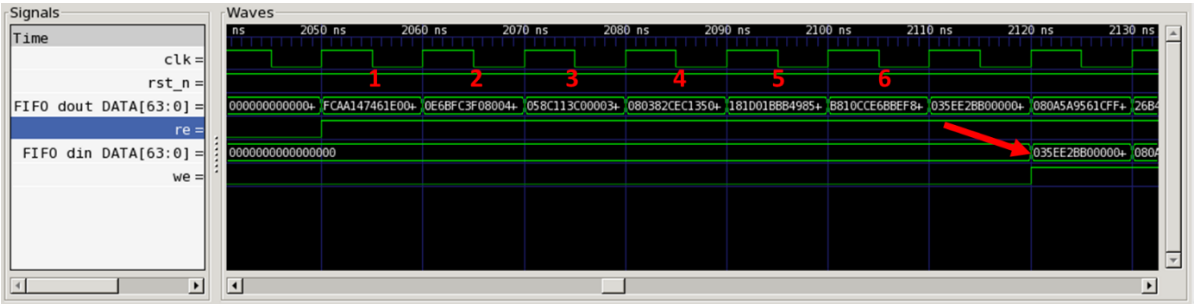


Figura 30: Simulación *Eliminar Cabecera*.

5.6 Síntesis de alto nivel

La síntesis en alto nivel realizada tanto para el bloque *Header Analyzer* como para el bloque *Eliminar Cabecera* se detalla a lo largo de este punto. Se debe tener en cuenta que en la herramienta Cadence Stratus existe un único fichero donde se define todo el proyecto, como los módulos, la configuración de simulación y las distintas estrategias que se van a usar para la síntesis de alto nivel. Este fichero es del tipo TCL (*Tool Command Lenguage*) y la información de salida será la encargada de especificar el *Makefile* del proyecto.

En el archivo del proyecto (denominado normalmente `project.tcl` en la literatura) se especificarán las opciones y la configuración del proyecto, así como los objetivos a sintetizar, pudiendo ejecutar cualquier número de éstos con diferentes opciones.

El archivo `project.tcl` agrupa su información en categorías de la siguiente manera:

1. Librería

2. Opciones de proyecto
3. Opciones de integración
4. Archivos de sistema
5. Objetivos a sintetizar
6. Simulación y síntesis lógica

En cada proyecto se deberá especificar la librería necesaria para la tecnología a utilizar (FPGA o ASIC), incluyendo información de área y *delay.* En el Código 9, se muestra como a través de comandos se especifica el dispositivo que se utilizará para implementar los bloques en desarrollo.

```
1  ######################################
2  # stratus_hls Options
3  ######################################
4  set_attr fpga_tool "vivado"
5  set_attr fpga_part "xc7z045ffg900-2"
```

Código 9: Asignación del dispositivo

Para controlar la síntesis en necesario definir una serie de atributos, lo cual sólo se puede determinar a través del *script* `project.tcl`. Los atributos de la síntesis pueden aplicarse de manera global para todo el proyecto o de manera individual para cada módulo de configuración.

Los atributos globales se aplican usando el comando *set_attr* y alguno de los atributos más clásicos son el periodo de reloj, *delay*, agresividad del planificador, resolver bucles, *arrays* etc.

Con el comando *set_attr clock_period* (Código 10) se puede fijar una frecuencia para el diseño, que será la usada en la ejecución del análisis temporal en las distintas fases del proceso de síntesis. El periodo se debe especificar como un número en punto flotante, siendo la unidad en ns.

```
1  ######################################################
2  # stratus_hls Options
3  ######################################################
4  # Clock period is required
5  set CLOCK_PERIOD "15.0"
6  set_attr clock_period $CLOCK_PERIOD
```

```
7  set_attr cc_options " -DCLOCK_PERIOD=15.0"
8  set_attr hls_cc_options " -DCLOCK_PERIOD=15.0"
```

Código 10: Restricciones del reloj para la síntesis en alto nivel

Una de las partes más importantes es la especificación de los módulos del diseño que sí serán sintetizados. Se especificarán con el comando *define_hls_module* en el archivo `project.tcl` junto con el nombre del módulo RTL o SystemC y el nombre del fichero. En este punto se añaden las distintas estrategias que se le aplicarán de manera global a todo módulo durante el proceso de síntesis.

En el Código 11 se muestran los archivos que conformará el módulo *Header Analyzer.* Como se comentó en el punto 5.3 Arquitectura del bloque *Header Analyzer*, el módulo contendrá un submódulo para la máquina de estados AXI4-Lite y los distintos procesos que hacen el funcionamiento del bloque que están incluidos en el fichero `header_analyzer.cpp`. A través del comando *define_hls_config* se puede asignar una estrategia a seguir al módulo asociado. En este caso se ha indicado con el comando *flatten_array* que no realice el aplanado de ningún *array*. Esta estrategia se ha elegido debido a que uno de los objetivos que se pretende conseguir es que los bloques a implementar no requieran un uso elevado de recursos. Al no aplanar los *arrays* se consigue ahorrar en recursos.

```
1  ############################################################
2  # Synthesis Module Configurations                          #
3  ##################### ######################################
4  define_hls_module header_analyzer "src/header_analyzer.cpp
5  src/AXI4_Lite.cpp"
6  define_hls_config header_analyzer REG --flatten_array=none
```

Código 11: Configuración del módulo a *Header Analizer*

Para el caso de *Eliminar Cabecera,* se muestra en el Código 12 como el módulo ha sido denominado como c*abecera* y se le añade el fichero correspondiente. También se le asigna como estrategia global de síntesis al módulo no aplanar los *arrays* con el objetivo de minimizar recursos.

```
1  ############################################################
2  #Synthesis Module Configurations
3  ############################################################
4  define_hls_module cabecera "src/eliminar_cabecera.cpp
```

```
5   define_hls_config cabecera REG --flatten_array=none
```

Código 12: Configuración del módulo a *Eliminar Cabecera*

5.6.1 Planificación

El planificador de Cadence Stratus proporciona un modo de dividir el comportamiento de un diseño en múltiples contextos del planificador. Es decir, puede planificarse de manera libre o de manera fija.

Para especificar que un bloque sea planificado de manera fija se añade la directiva *HLS_DEFINE_PROTOCOL* al principio del bloque. Esto permite que sea el usuario, y no la herramienta, quién especifique de manera exacta los estados, haciendo uso de los *wait()*.

Cuando no se especifica la directiva *HLS_DEFINE_PROTOCOL*, Stratus es libre de insertar cualquier número de estados necesarios de un bloque.

Es importante definir bien este bloque ya que puede fallar la etapa de planificación. Algunas de las razones más comunes son: que la latencia del bloque especificado supera al periodo de reloj fijado en la configuración, que existen dependencias en bucles *pipelined*, o que haya conflictos de recursos que acceden a éste en el mismo ciclo.

Tanto para el caso de *Header Analyzer* como para *Eliminar Cabecera* no se dejará que la herramienta realice la etapa de planificación. El bloque *Header Analyzer* cuando fue diseñado en [17] se modeló incorporando una cantidad de ciclos para obtener su diseño funcional. En este bloque la síntesis en alto nivel se realizó con la herramienta Cadence C-to-Silicon, predecesora de Cadence Stratus. Su planificador respetaba los ciclos introducidos por el usuario sin necesidad de añadir directivas ni indicar esta acción. Por lo tanto, si se dejara a la herramienta Cadence Stratus el control de planificación de este bloque insertará nuevos ciclos, traduciéndose en que el funcionamiento de este bloque no será el correcto, ya que se ha modificado su funcionalidad cuando el planificador ha incorporado los ciclos que estime necesario.

Se ha realizado la migración del bloque *Header Analyzer* desde la herramienta Cadence C-to-Silicon a Cadence Stratus. Para ello se ha realizado la planificación del bloque respetando los ciclos que poseía el código original a través de la directiva *HLS_DEFINE_PROTOCOL*. Cabe destacar que la migración entre ambas herramientas no ha

sido directa debido a que, aunque se le indique a Cadence Stratus que será el usuario quién realice la planificación a través de la directiva, la herramienta no puede planificar el código de *Header Analyzer.* Esto es debido a que el comportamiento funcional de los planificadores de ambas herramientas no es el mismo y un código que es sintetizable en Cadence C-to-Silicon no implica que sea sintetizable en Cadence Stratus, o puede ser sintetizable pero no tener el mismo comportamiento. Por ese motivo es importante realizar la co-simulación RTL para comprobar que tras la síntesis se obtiene un código funcional.

Por lo tanto, se ha tenido que hacer una serie de optimizaciones sobre el código con el objetivo de conseguir planificar este bloque y obtener el código en RTL que sea correctamente funcional con lo modelado.

El primer paso realizado en esta optimización ha sido adaptar el código para que en Cadence Stratus no tenga errores. Estas adaptaciones son, por ejemplo, inicializaciones de variables o código con directivas que usa la herramienta Cadence C-to-Silicon, entre otras modificaciones. En el Código 13 se muestra cómo se han adaptado algunas líneas del código original mediante la incorporación de macros.

```
1  #ifdef __CTOS__
2  #ifdef TOP
3  SC_MODULE_EXPORT(header_analyzer);
4  #endif
5  #endif
```

Código 13: Incorporación de macros para adaptación del código

Otro punto a tener en cuenta es que la inicialización de los puertos, señales internas y variables debe encontrarse en un bloque de *reset* que contenga la directiva *HLS_DEFINE_PROTOCOL* para evitar fallos en el funcionamiento. En el Código 14 se muestra el bloque de *reset* para el proceso de copia del paquete de *Header Analyzer.*

```
1  void header_analyzer::copy() {
2
3        DATA temp;
4        bool new_word = true;
5        sc_uint<5> e = 0;
6
7      {
8          HLS_DEFINE_PROTOCOL("rst_copy");
```

```
9           start_check_header.write(false);
10          we.write(0);
11          small_packet.write(0);
12          data_in_ethernet_ready.write(0);
13          wait();
14      }
15        ...
16  }
```

Código 14: Planificación de la inicialización de señales en *Header Analyzer*

Una vez adaptado el código, se procede a realizar la optimización para que sea sintetizable. Uno de los principales motivos por el cual el planificador no consigue realizar la síntesis es debido a la falta de ciclos que posee el bucle principal tras la incorporación de la directiva *HLS_DEFINE_PROTOCOL*.

En el Código 15 se muestra un fragmento del código original. Se trata del proceso de copia del paquete al cual se le ha añadido la directiva *HLS_DEFINE_PROTOCOL* para realizar la planificación del proceso. El motivo por el cual no se puede realizar la planificación es debido a que el bucle principal cuenta con una cantidad considerable de operaciones, cada una de ellas tiene asociado un *slack* que se va acumulando. Por lo tanto, se requiere la implementación de una sentencia *wait* para evitar sobrepasar el periodo de reloj. Como se ha comentado, la incorporación de nuevos ciclos no es viable debido a que modificaría el comportamiento del diseño.

Otro motivo por el cual no se realiza la planificación es debido a que el *array* denominado *headers* (línea 29) se ha implementado como una memoria BRAM. Aunque en el diseño en Cadence C-to-Silicon este *array* sí ha sido implementado como una memoria BRAM y se ha conseguido realizar la planificación, en Cadence Stratus no es posible esta planificación. Esto es debido a que, para acceder a un *array* implementado en memoria se toman demasiados ciclos en comparación a una memoria implementada en registros. Por lo tanto, después de la línea 37 del Código 15 es necesaria la incorporación de una sentencia *wait.* De nuevo, esta incorporación es inviable. Aun sustituyendo la implementación de este *array* en memoria por la implementación en registros seguiría existiendo la incapacidad de realizar la planificación de este proceso debido a la cantidad de *slack* que introducen las operaciones que se realizan hasta la primera aparición del primer *wait.*

```
1   void header_analyzer::copy() {
2
3         DATA temp;
4         bool new_word = true;
5         sc_uint<5> e = 0;
6   {
7         HLS_DEFINE_PROTOCOL("reset");
8         start_check_header.write(false);
9         we.write(0);
10        small_packet.write(0);
11        data_in_ethernet_ready.write(0);
12        wait();
13  }
14        data_in_ethernet_ready.write(1);
15  {
16        HLS_DEFINE_PROTOCOL("While");
17        while (true) {
18        we.write(0);
19        if(data_in_ethernet_valid.read() && almostfull_n.read()){
20                  start_check_header.write(false);
21                  small_packet.write(0);
22                  temp.data= data_in_ethernet_data.read();
23                  temp.tlast = data_in_ethernet_last.read();
24                  din.write(temp);
25                  we.write(1);
26
27        //Guardar cabecera para el proceso de toma de decisión
28                  if(new_word ==true){
29                        headers[e] = temp.data;
30                  }
31                  if((e>=HEADERS_SIZE-1) && new_word){
32                        new_word = false;
                          start_check_header.write(true);
33                  }
34                  if(data_in_ethernet_last.read() == 1 &&
35  new_word ==true){
36                        new_word = false;
37                        small_packet.write(1);
38                        start_check_header.write(true);
39                  }
40                  wait();
```

```
41                    we.write(0);
42                    small_packet.write(0);
43                    start_check_header.write(false);
44                    if(new_word){
45                          e++;
46                    }else{
47                          e = 0;
48                    }
49                    if(temp.tlast==true){
50                          new_word = true;
51                          e = 0;
52                    }
53              }else{
54              wait();
55            }
56      }
57  }
58  }
```

Código 15: Proceso de *Header Analyzer* con error en la planificación

En el Código 16 se muestra el informe detallado de los ciclos que han sido añadidos en el bloque de protocolo para que el proceso se pueda planificar. En la línea 4 se muestra el ciclo adicional incorporado, denominado ciclo 2, y a partir de la línea 17 se muestran las operaciones que están involucradas en la incorporación de este ciclo.

```
1  ####################[ Protocol Violation ]#########################
2  Protocol block "While" was relaxed to meet a schedule
3  The addition of the following cycle(s):
4    [ Cycle 2 ]
5  Violated this protocol block
6  Found 3 path(s) that caused violation
7
8  Clock period          :   10.000
9  Cycle slack           :    0.000
10 --------------------------------
11 Effective clock period:   10.000
12
13 ********************[ Violator Path 1 ]*************************
14 OP ID  Source           Resource                    Incr   Accum
15 DFFD                    DFF (Clk->Q)                0.770  0.770
16 DFFD                    DFF (Tsu)                   0.537  0.537
```

```
17 ---------------[ Cycle 2 !!! VIOLATION !!! ]--------------------
18 OP ID  Source                     Resource            Incr   Accum
19 DFFD                              DFF (Clk->Q)        0.770  0.770
20 OP201 header_analyzer.cpp:26,12   header_analyzer_Or   1.117  1.887
21 OP202 header_analyzer.cpp:26,12   header_analyzer_Not  1.117  3.004
22 OP203 header_analyzer.cpp:25,7    header_analyzer_And  1.117  4.121
23 OP204 header_analyzer.cpp:16,38   header_analyzer_And  1.117  5.238
24 OP205 header_analyzer.cpp:16,5   headers0 output port  0.537  5.775
25 ================================================================
26 ====================== Call Stack Table=========================
27 ================================================================
28 OP205 Called:
29           at header_analyzer.cpp:16,5
30           called from header_analyzer.cpp:25,4
31           called from header_analyzer.cpp:24,12
32           called from :0,0
           called from header_analyzer.h:101,17 (in headers())
33 ================================================================
34 ======================== Cycle Explanation =====================
35 ================================================================
36 ################################################################
37 # ERROR 01484: Unable to produce a valid schedule, found 1       #
38 #              Violation(s)                                      #
39 ################################################################
```

Código 16: Informe de protocolos

Haciendo un estudio exhaustivo de la cantidad de *slack* que introduce cada operación, se ha llegado a la conclusión de que la operación que más *slack* introduce es la copia de la cabecera al *array* (línea 37, Código 15). Si se prescindiera de esta copia, el proceso podría ser planificado sin ningún tipo de inconvenientes. Obviamente, prescindir de esto es ilógico ya que no se podría realizar el filtrado.

En el Código 17 se muestra como se ha modificado el código para conseguir que se pueda planificar el proceso y así poder realizar el proceso de síntesis. Esto se ha logrado gracias al análisis realizado del *slack* de las operaciones, donde la copia de la cabecera requería de un ciclo adicional. Se ha rediseñado la estructura del código y se ha aprovechado la sentencia *wait* existente en el código para estabilizar el *slack* que añade la copia de la cabecera en el *array.* El código con las modificaciones realizadas se muestra en el Código 17.

```
1  void header_analyzer::copy() {
2        DATA temp;
3        bool new_word = true;
4        sc_uint<5> e = 0;
5  {
6        HLS_DEFINE_PROTOCOL("reset");
7        start_check_header.write(false);
8        we.write(0);
9        small_packet.write(0);
10       data_in_ethernet_ready.write(0);
11       wait();
12 }
13      data_in_ethernet_ready.write(1);
14 {
15       HLS_DEFINE_PROTOCOL("While");
16       while (true) {
17       we.write(0);
18             if(data_in_ethernet_valid.read() &&
19 almostfull_n.read()){
20
21                   start_check_header.write(false);
22                   small_packet.write(0);
23                   temp.data= data_in_ethernet_data.read();
24                   temp.tlast = data_in_ethernet_last.read();
25                   din.write(temp);
26                   we.write(1);
27
28                   if((e>=HEADERS_SIZE-1) && new_word){
29                         new_word = false;
30                         start_check_header.write(true);
31                   }
32                   if(data_in_ethernet_last.read() == 1 &&
33 new_word ==true){
34                         new_word = false;
35                         small_packet.write(1);
36                         start_check_header.write(true);
37                   }
38                   wait();
39 //Guardar cabecera para el proceso de toma de decisión
40                   if(new_word ==true){
41                      headers[e] = temp.data;
```

```
42                      }
43                      we.write(0);
44                      small_packet.write(0);
45                      start_check_header.write(false);
46                      if(new_word){
47                            e++;
48                      }else{
49                            e = 0;
50                      }
51                      if(temp.tlast==true){
52                            new_word = true;
53                            e=0;
54                      }
55                }else{
56                wait();
57              }
58        }
59  }
60  }
```

Código 17: Proceso de *Header Analyzer* apto para la planificación

Para el caso de *Eliminar Cabecera*, este bloque imita las interfaces de una FIFO, por lo que la incorporación de nuevos ciclos no es viable debido a que el bloque está limitado a transferir paquetes cada ciclo. Es por ello por lo que se le asigna la directiva *HLS_DEFINE_PROTOCOL* para que la herramienta respete los distintos ciclos que contiene el código.

En el Código 18 se muestra como se le ha asignado la directiva *HLS_DEFINE_PROTOCOL* sobre el bucle principal del proceso con el fin de que la herramienta respete los ciclos que contiene este bucle.

```
1   void cabecera::copy(){
2     DATA temp;
3     sc_uint<5> cont = 0;
4     bool new_packet = true;
5   {
6         HLS_DEFINE_PROTOCOL("RSR");
7         we.write(0);
8         empty.write(0);
9         wait();
```

```
10  }
11        empty.write(1);
12  {
13      HLS_DEFINE_PROTOCOL("while");
14      while(true){
15          if(almostfull.read()){
16              empty.write(0);
17          }else{
18              empty.write(1);
19          }
20          if(re.read() && !almostfull.read()){
21                temp=dout.read();
22
23              if(new_packet == true){
24                    cont++;
25              }
26              if(cont>HEADER_SIZE){
27                    we.write(1);
28                    new_packet = false;
29                    din.write(temp);
30              }
31              wait();
32              we.write(0);
33              if(temp.tlast== true){
34                    new_packet = true;
35                    cont = 0;
36              }
37        }else{
38             wait();
39        }
40      }
41    }
42  }
```

Código 18: Planificación *Eliminar Cabecera*

5.6.2 Agresividad del planificador

El planificador de Cadence Stratus por defecto está destinado a diseños con una cantidad mínima de lógica de control. Para diseños que contengan una cantidad

significativa de lógica de control, se habilita la agresividad del planificador para mejorar la latencia o reducir área.

Existen dos directivas de control de síntesis que proporciona la agresividad del planificador:

1. *sched_aggressive_1:* Este modo crea rutas fuera de las declaraciones de control. Permiten encadenar a través de los límites del flujo de control, utilizando el paralelismo para reducir la latencia.
2. *sched_aggressive_2:* Este modo contiene procesamiento adicional que intenta reducir el área mientras el control del planificador está en paralelo. Los tiempos de síntesis aumentan.

Como el bloque *Eliminar Cabecera* no contiene una gran cantidad de lógica, no se ha aplicado ninguna agresividad al planificador debido a que esta acción no tendría ningún impacto sobre el bloque y los resultados serían los mismos. En cambio, para *Header Analyzer* se ha asignado la directiva *sched_aggressive_2* para reducir el área de ésta.

```
1  ######################################
2  # stratus_hls Options
3  ######################################
4  set_attr sched_aggressive_2 on
5  set_attr sched_aggressive_1 off
```

Código 19: Agresividad del planificador en *Header Analyzer*

5.6.3 Resultados

Los resultados obtenidos tras la síntesis en alto nivel del bloque *Header Analyzer* se muestran en la Tabla 5. Para el bloque *Eliminar Cabecera* se muestra en la Tabla 6.

La lógica secuencial está constituida por los Flip-Flops. El bloque *Header Analyzer*, su ocupación secuencial es elevada debido a que, durante el proceso de síntesis, se ha forzado a resetear todas las señales para evitar que señales no inicializadas propaguen un error al obtener el diseño en RTL. Con esto, se obliga a que todas las señales se inicialicen implementando Flip-Flops que posteriormente se resetearán.

Tabla 5: Recursos de la síntesis en alto nivel de *Header Analyzer*

MÓDULO	ÁREA TOTAL	ÁREA COMBINACIONAL	ÁREA SECUENCIAL	FLIP-FLOP
HEADER ANALYZER	3.312	2.058	1.254	1.254

Tabla 6: Recursos de la síntesis en alto nivel de *Eliminar Cabecera*

MÓDULO	ÁREA TOTAL	ÁREA COMBINACIONAL	ÁREA SECUENCIAL	FLIP-FLOP
ELIMINAR CABECERA	175	100	75	75

5.7 Co-simulación RTL

Una vez la síntesis en alto nivel se ha realizado de manera correcta, se debe continuar la verificación a través de la co-simulación. Esta co-simulación permite comprobar si el diseño RTL obtenido a través de la síntesis funciona igual que el diseño en alto nivel. Ésto asegura la funcionalidad del diseño antes de ser volcado al dispositivo.

Con la herramienta de síntesis de alto nivel Stratus se ha obtenido el diseño en RTL. Stratus permite realizar la co-simulación del sistema utilizando el mismo *testbench* que en la simulación SystemC gracias a que genera un *wrapper* que permite la reutilización del test, facilitando el proceso de verificación y ahorrando tiempo del proyecto.

5.7.1 Co-simulación del bloque Header Analyzer

De igual manera que se realizaron las comprobaciones en la simulación se realizarán en la co-simulación, debido a la reutilización del *wrapper* y así comprobar que ambos resultados son los mismos. Es por eso por lo que a través de la señal S_AXI_WDATA se pasará la habilitación del análisis de la capa de enlace y la dirección MAC de destino con el fin de comprobar que el filtrado es correcto y es enviado por la interfaz de salida hacia el *Motor de Búsqueda* si coincide los campos analizados.

En la Figura 31 se muestra como la señal S_AXI_WDATA envía al bloque un ‘1’ para la activación del análisis de la capa de enlace y es recibido por la señal *check_ether*.

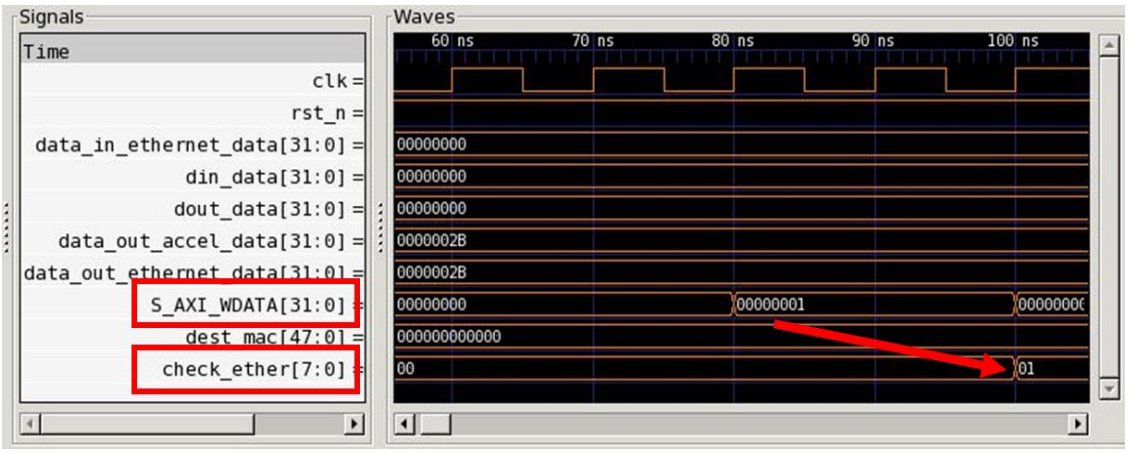


Figura 31: Co-simulación *Header Analyzer*. Activación *check_ether*

Mientras que en la Figura 32 se muestra la asignación de la dirección MAC de destino a través de la señal S_AXI_WDATA y es recibido por la señal *dest_mac.*

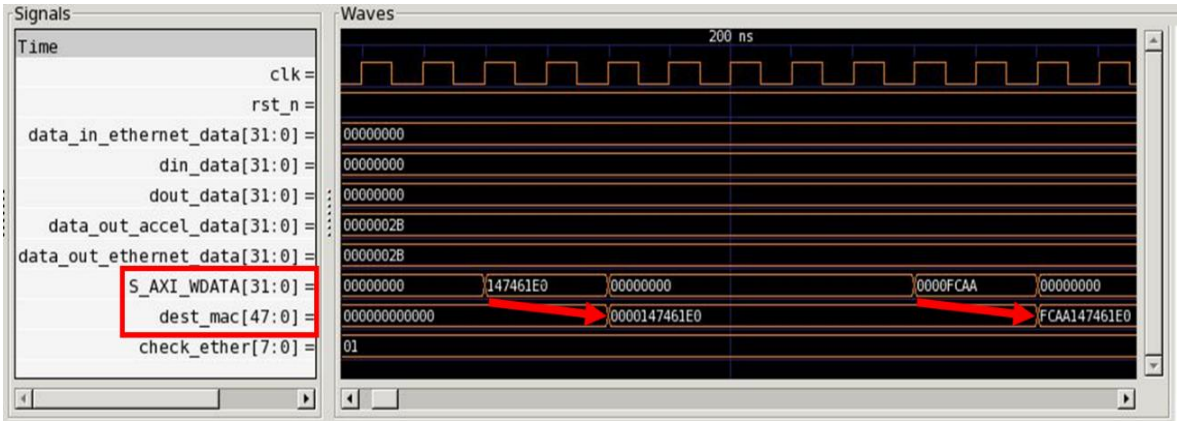


Figura 32: Co-simulación *Header Analyzer*. Asignación dirección MAC de destino

En la Figura 33 se observa como los paquetes son recibidos correctamente del *testbench* y además que son enviados a través de la interfaz FIFO para almacenar el paquete sin pérdidas.

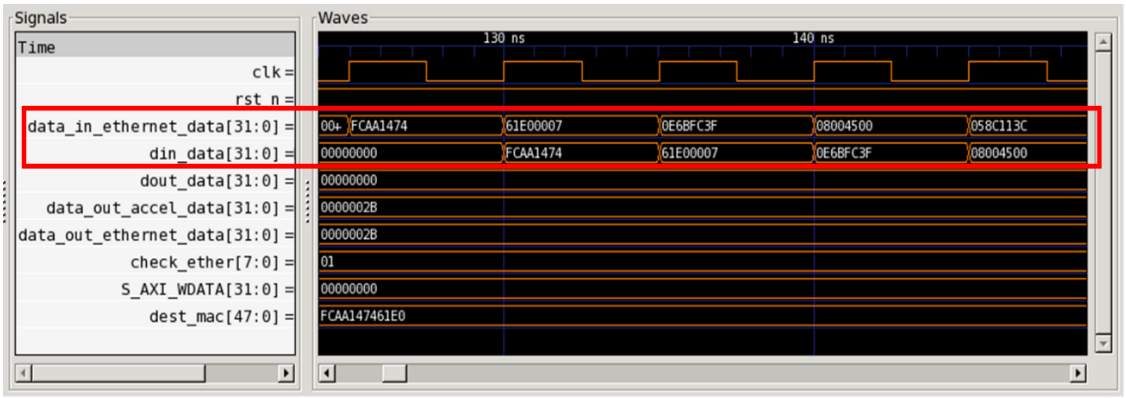


Figura 33: Co-simulación *Header Analyzer*. Recepción del paquete

Una vez realizado el filtrado y comprobado que la cabecera MAC preestablecida para realizar el filtrado y la del paquete Ethernet se corresponden, entonces el paquete es redireccionado a través de la interfaz de salida del acelerador hacia el *Motor de Búsqueda* y así se muestra en la Figura 34.

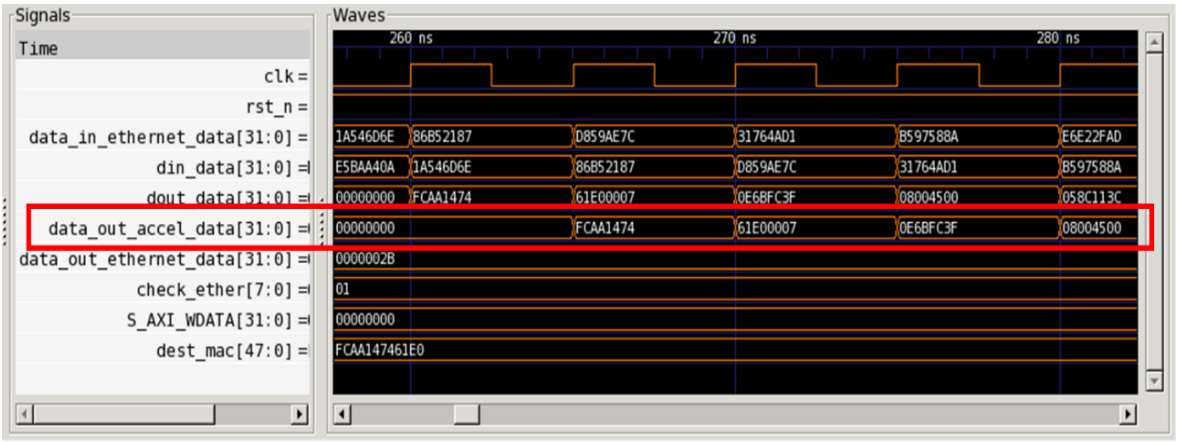


Figura 34: Co-simulación *Header Analyzer*. Redireccionamiento del paquete al acelerador

Si se asigna una dirección MAC de destino que no corresponde con la de la trama Ethernet, entonces no se realiza el filtrado y, por lo tanto, el paquete debe salir por la interfaz de salida que conecta con el TEMAC. En la Figura 35 se muestra la asignación de la dirección MAC de destino que no coincide con la del paquete.

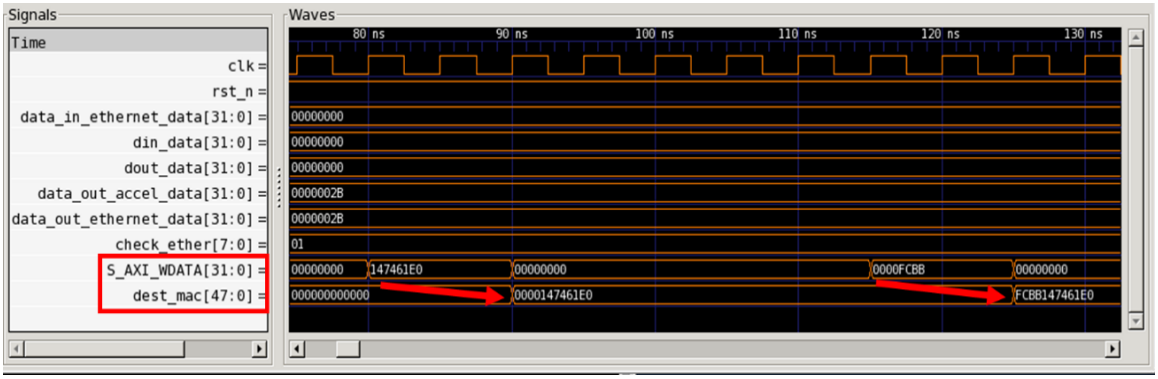


Figura 35: Co-simulación *Header Analyzer*. Dirección MAC de destino

En la Figura 36 se observa como el paquete sale por la interfaz *data_out_ethernet_data* hacia el bloque TEMAC de salida debido a que la dirección MAC de destino del filtrado no coincide con la del paquete.

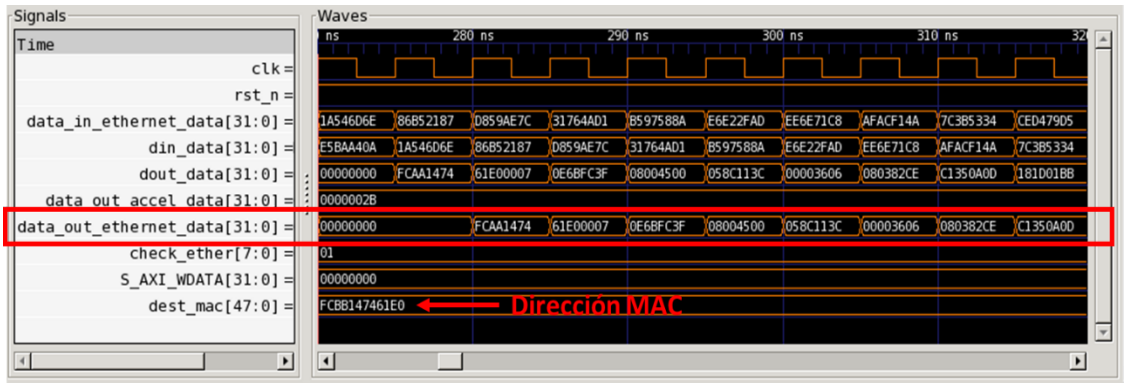


Figura 36: Co-simulación *Header Analyzer*. Redireccionamiento del paquete a la red

Por último, en la Figura 37, se observa la latencia del bloque al realizar el filtrado; es decir, el tiempo que tarda desde que el primer dato del paquete es recibido hasta que el mismo dato es redireccionado a través de la interfaz correspondiente. Esta latencia corresponde a 28 ciclos de reloj, suponiendo una frecuencia de 200 MHz supondría una latencia de 0.140 µs.

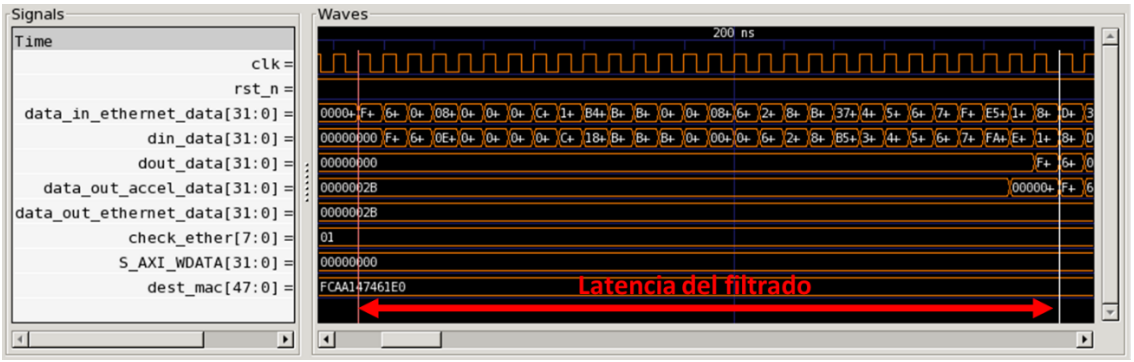


Figura 37: Co-simulación *Header Analyzer*. Latencia del filtrado

5.7.2 Co-simulación del bloque Eliminar Cabecera

Para la co-simulación de este bloque, y de igual manera que en la simulación del mismo, se le pasará un paquete Ethernet y se comprobará que en su salida se han eliminado las seis primeras palabras correspondientes con la cabecera (Figura 38).

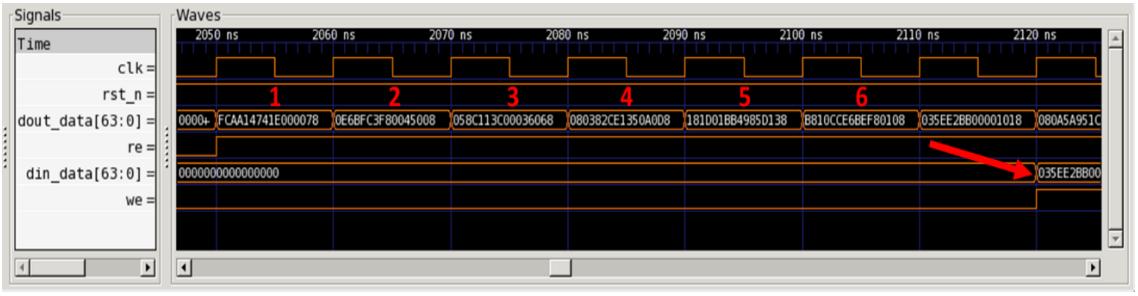


Figura 38: Co-simulación *Eliminar Cabecera.*

5.8 Síntesis lógica de los bloques IP

Una vez que la simulación y la co-simulación sean correctas se puede proceder a realizar la síntesis lógica del diseño en RTL. Para realizar esta síntesis se utilizará la herramienta Synopsys Synplify que, a partir de un fichero Verilog o VHDL del diseño se realiza este proceso. Se tomará el fichero Verilog que genera Cadence Stratus al realizar la síntesis en alto nivel.

Tanto para la síntesis lógica del *Header Analyzer* como del bloque de *Eliminar Cabecera* se usarán las mismas opciones para realizar la síntesis en Synplify y son las que se muestran en la Figura 39. Es importante activar la opción *Disable I/O Insertion* debido a que se va a sintetizar un bloque IP que posteriormente será integrado en una plataforma; por tanto, estas señales irán conectadas a otras señales que contiene la plataforma.

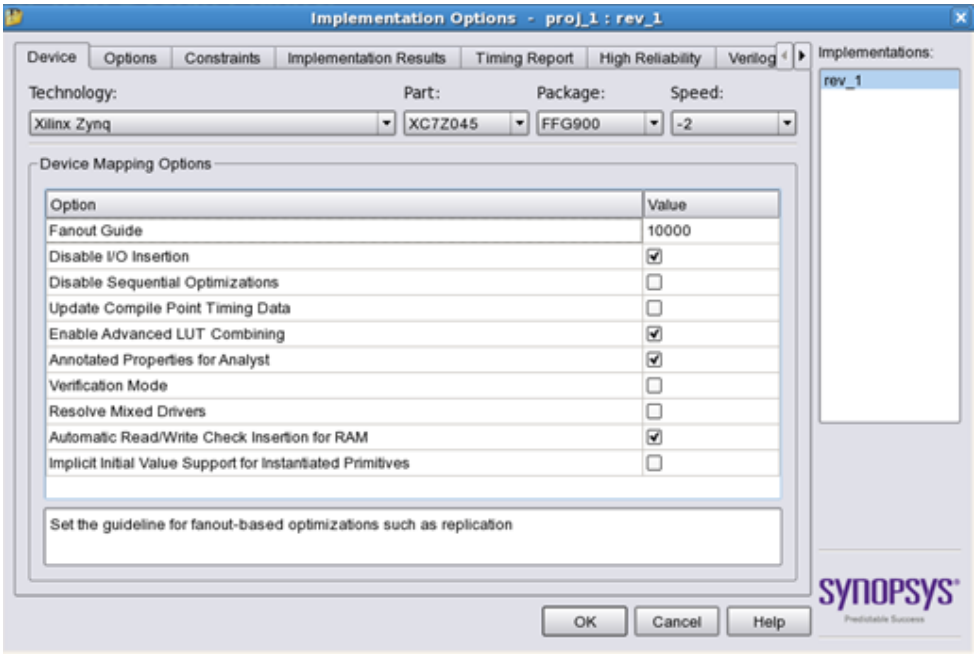
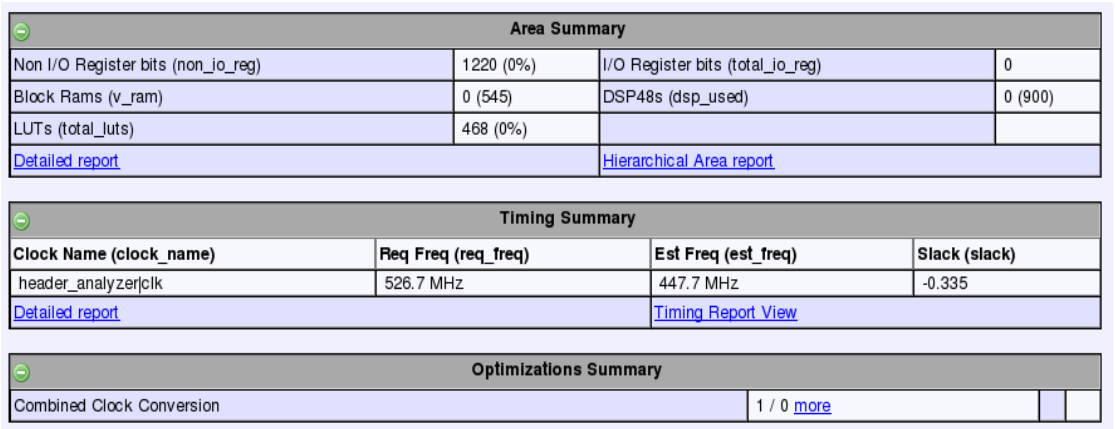


Figura 39: Opciones de síntesis en Synplify

5.8.1 Resultados del bloque *Header Analyzer*

En la Figura 40 se muestra un resumen de los resultados de la síntesis lógica, del área que se ha utilizado en el diseño de este bloque que conforman las distintas LUTs, registros y BRAM, así como la frecuencia de trabajo alcanzada. El bloque que se ha diseñado funciona a una frecuencia de 447.7 MHz y no alcanza el 1% de ocupación de recursos del dispositivo.



Area Summary

Non I/O Register bits (non_io_reg)	1220 (0%)	I/O Register bits (total_io_reg)	0
Block Rams (v_ram)	0 (545)	DSP48s (dsp_used)	0 (900)
LUTs (total_luts)	468 (0%)		
Detailed report		Hierarchical Area report	

Timing Summary

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
header_analyzer|clk	526.7 MHz	447.7 MHz	-0.335
Detailed report		Timing Report View	

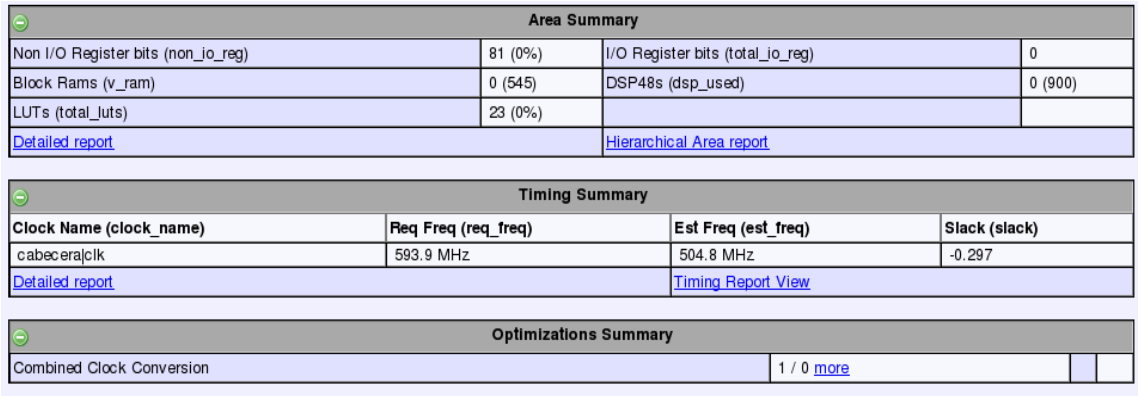
Optimizations Summary

Combined Clock Conversion	1 / 0 more

Figura 40: Resultados síntesis lógica de *Header Analyzer*

5.8.2 Resultados del bloque *Eliminar Cabecera*

Para el bloque de *Eliminar Cabecera* los resultados se muestran en la Figura 41. Este bloque utiliza un número reducido de recursos, cuya ocupación no llega al 1% del dispositivo y puede funcionar a una frecuencia de 504.8 MHz, lo que permite realizar una transferencia rápida de los datos para su análisis en bloques posteriores.



Area Summary

Non I/O Register bits (non_io_reg)	81 (0%)	I/O Register bits (total_io_reg)	0
Block Rams (v_ram)	0 (545)	DSP48s (dsp_used)	0 (900)
LUTs (total_luts)	23 (0%)		
Detailed report		Hierarchical Area report	

Timing Summary

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
cabecera|clk	593.9 MHz	504.8 MHz	-0.297
Detailed report		Timing Report View	

Optimizations Summary

Combined Clock Conversion	1 / 0 more

Figura 41: Resultados síntesis lógica de *Eliminar Cabecera*

5.9 Encapsulado del bloque IP

Para realizar el encapsulado del diseño en un bloque IP se utiliza la herramienta Vivado Design Suite. A partir de un fichero Verilog, VHDL o EDIF se puede generar un IP. La herramienta Synopsys Synplify después de la síntesis lógica genera un fichero EDIF, el cual se ha tomado para realizar el encapsulado del bloque.

Se creará un nuevo proyecto en Vivado al que se le incluirá este fichero EDIF como código fuente. El acceso a Vivado IP Integrator se realiza desde el menú *Tools*.

Este empaquetador permite modificar la identificación del bloque que se está creando, personalizar parámetros, asociar puertos e interfaces, etc., como se puede observar a la izquierda de la Figura 42

Además, en la misma figura, se puede apreciar los distintos campos que se pueden modificar para tener una identificación del bloque IP en desarrollo de manera detallada y precisa, añadiendo nombre, descripción, versión, documentación, etc

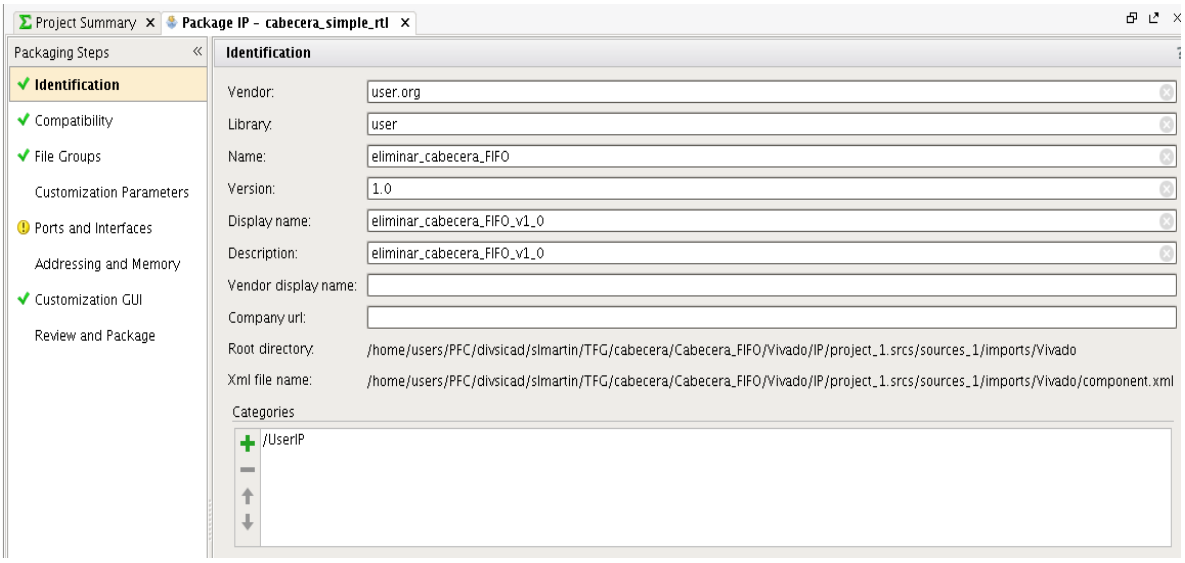


Figura 42: Modificación de la identificación del IP

En el paso de *Ports and Interfaces* se pueden agrupar distintos puertos en interfaces. Ésto es ideal debido a que, en ambos diseños de desarrollo, se han utilizado tanto protocolos AXI4-Stream, AXI4-Lite como FIFOs, dando pie a poder agrupar estas señales en interfaces para facilitar el manejo de la plataforma. En la Figura 43 se muestran los distintos puertos que tiene el bloque *Eliminar Cabecera*.

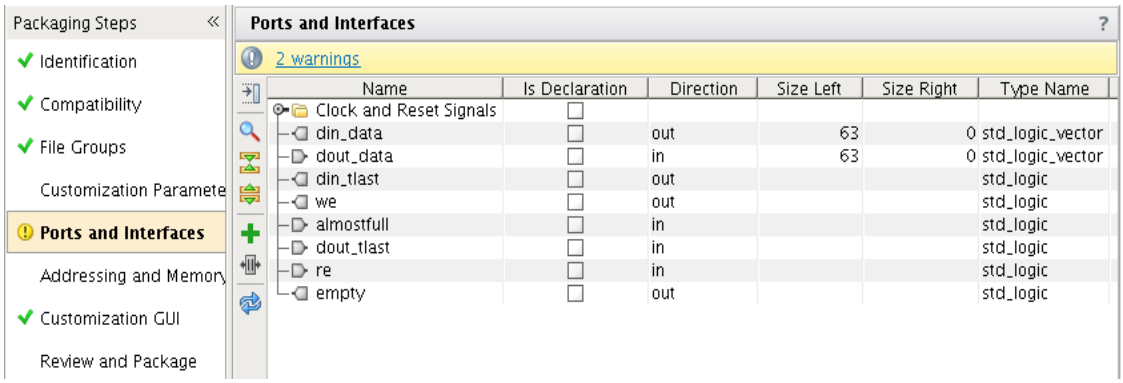


Figura 43: Puertos e interfaces del IP

En la Figura 44 se muestra cómo se crea una interfaz nueva. Se debe indicar que tipo de interfaz se desea implementar, en este caso se trata de una FIFO de escritura y se le asigna un nombre.

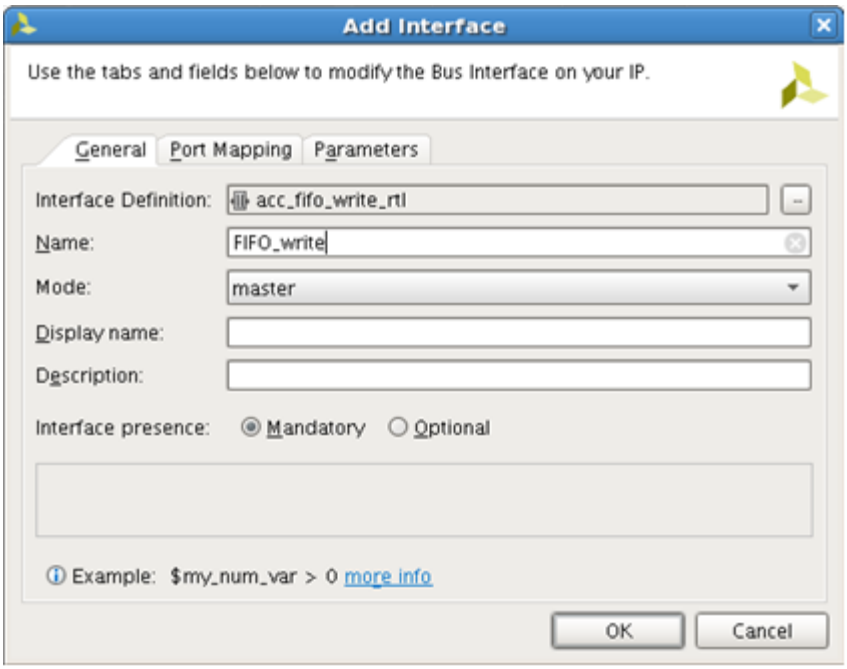


Figura 44: Añadir interfaz

Posteriormente, en la Figura 45 se realiza el proceso de asociar las señales de la interfaz FIFO con los correspondientes puertos del bloque diseñado. Se puede activar las opciones de filtrado para evitar errores durante este proceso.

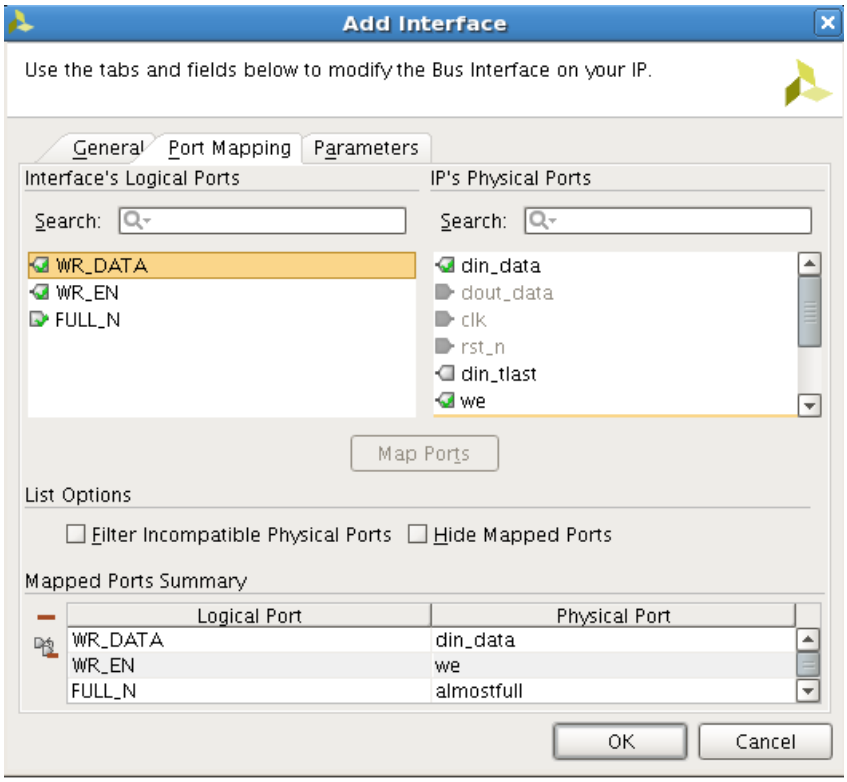


Figura 45: Asociación de interfaces y puertos

Una vez se ha terminado de crear las interfaces y asociar los puertos, se procede a realizar el empaquetado del bloque IP. En la Figura 46, se muestra el bloque IP de *Header Analyzer* una vez se ha empaquetado a través de Vivado IP Integrator. Se puede observar cómo se han agrupado distintos puertos para crear interfaces AXI4-Lite para la configuración del bloque denominado S_AXI, o las interfaces de entrada y salida de datos que se implementan en AXI4-Stream como son *eth_in, eth_out* y *accel_out*. Además, se ha generado las interfaces para conectar con la FIFO externa que se implementará en la plataforma final.

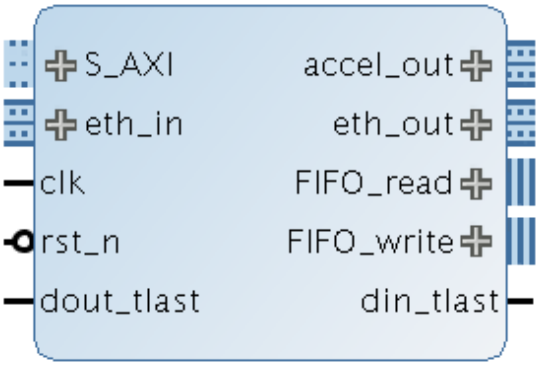


Figura 46: Bloque IP *Header Analyzer*

En la Figura 47 se muestra el bloque IP de *Eliminar Cabecera*. En este bloque sólo se ha creado la interfaz FIFO de escritura debido a que los datos que le entran a este bloque no se implementan como una FIFO de lectura. Es por ello por lo que se han dejado los distintos puertos sin agrupar.

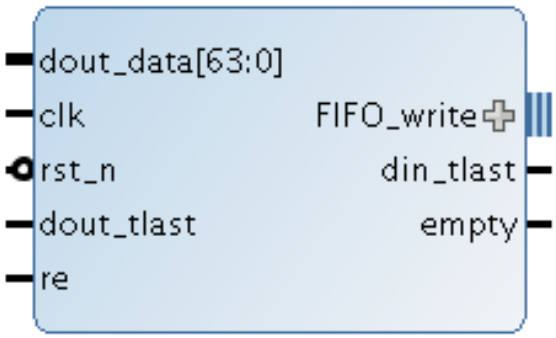


Figura 47: Bloque IP *Eliminar Cabecera*

En ambos bloques se puede apreciar como existen señales denominadas *din_tlast* o *dout_tlast*. Estas señales corresponden con las señales de las FIFOs, que poseen 32 o 64 bits de datos (dependiendo del bloque) más un bit de señalización *last*. La herramienta de síntesis de alto nivel Cadence Stratus no agrupa las estructuras, sino que divide sus campos, por lo que separa el dato del *last* de las FIFOs y no se puede implementar esta última señal como una interfaz FIFO. Cuando se realice el montaje de la plataforma final se presentará la solución a este problema.

Una vez se han implementado los bloques en el dispositivo se obtienen los distintos datos de utilización de cada bloque, recopilados en la Tabla 7:

Tabla 7: Utilización después de la implementación

BLOQUE IP	SLICE LUTS	SLICE REGISTERS	F7 MUXES	SLICE	LUT AS LOGIC	LUT FF PAIRS
HEADER ANALYZER	478	1.260	2	317	478	174
ELIMINAR CABECERA	24	81	0	26	24	13

5.10 Conclusiones

Durante este capítulo se ha explicado el desarrollo de los bloques IP. El bloque *Header Analyzer* cuenta con múltiples procesos y se ha decidido no añadir otro proceso a

este bloque para realizar la eliminación de la cabecera. Por tanto, se ha modelado un bloque llamado *Eliminar Cabecera* con el objetivo de que realice esta acción.

Se ha realizado el empaquetado de ambos bloques IP con el fin de integrarlo en la plataforma final. El bloque *Header Analyzer* trabaja con un ancho de palabra de 32 bits mientras que el bloque *Eliminar Cabecera* trabaja a 64 bits debido a que se incorporará a la *Unidad de Despacho*.

Capítulo 6. Integración de la plataforma final

6.1 Introducción

En este capítulo se explica cómo se ha montado la plataforma final sobre el dispositivo Zynq, así como los distintos bloques IP que son necesarios para conformarla. Una vez que la plataforma se encuentra disponible, se realizará la síntesis e implementación de la misma, finalizando con la generación del *bitstream* necesario para programar el dispositivo.

6.2 Construcción de la plataforma DPI

Para realizar la integración de la plataforma, se empleará la herramienta Vivado Design Suite, donde se incluirán diversos bloques necesarios para el correcto funcionamiento. Una ventaja que posee esta herramienta es que incorpora bloques IP de manera automática si son necesarios como los relacionados con el PS y la comunicación AXI.

A continuación, se muestran los distintos bloques que se han utilizado en la creación de la plataforma.

6.2.1 Zynq-7 Processing System

Como se ha venido explicando durante esta memoria, esta plataforma será implementada sobre un dispositivo SoC del cual se hará uso tanto de la parte de lógica

programable como de la parte de procesamiento. Para implementar esta última parte es necesario instanciar el PS a través del bloque Zynq [32].

En la Figura 48 se pueden apreciar las interfaces AXI necesarias para realizar la comunicación entre PS y PL. Además, cuenta con el puerto FCLK_CLK0 que es el que dará a la plataforma completa la señal de reloj.

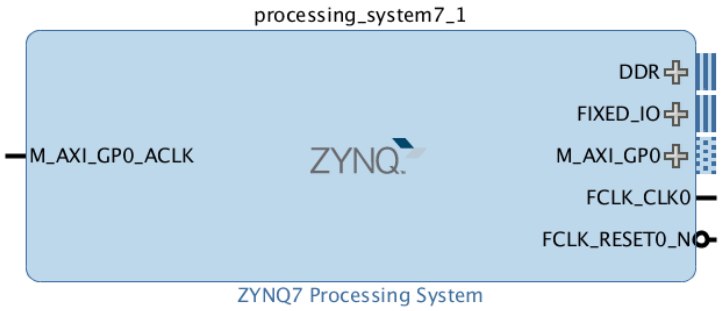


Figura 48: Bloque IP de Zynq

6.2.2 Processor System Reset

Este bloque es instanciado automáticamente por la herramienta Vivado. Proporciona señales de *reset* para el PS, los periféricos y las interconexiones [33]. A este bloque irá conectado el IP del Zynq y los *resets* de los bloques necesarios en la plataforma y que se explicarán más adelante.

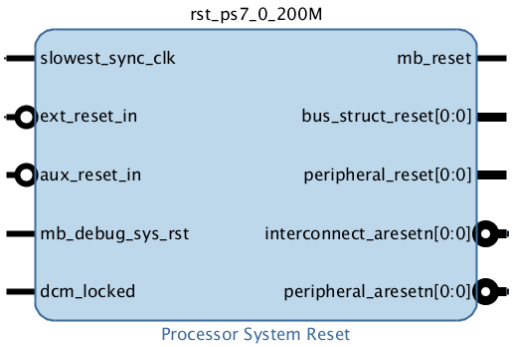


Figura 49: Bloque IP Processor System Reset

6.2.3 AXI Interconnect

El bloque AXI *Interconnect* (Figura 50) permite realizar la interconexión con los distintos periféricos del sistema a través de los protocolos AXI3, AXI4 o AXI4-Lite. Esto

facilita la conexión de distintos maestros y esclavos, pudiendo variar cada periférico su tamaño, dominio de reloj y protocolo. Este bloque es inferido automáticamente por la herramienta interconectando diversos bloques al PS a través de este bloque IP [34].

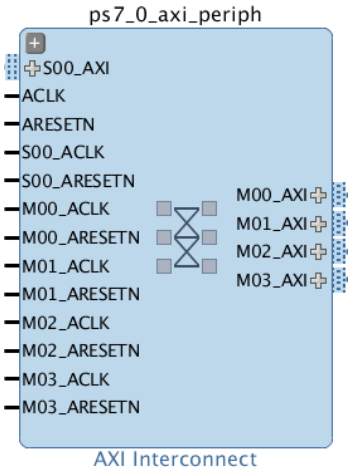


Figura 50: Bloque IP AXI Interconnect

6.2.4 AXI 1G/2.5G Ethernet Subsystem

En el diseño de la plataforma se ha utilizado el bloque IP AXI 1G/2.5G Ethernet *Subsystem* (Figura 51), que permite implementar un TEMAC funcionando a 10/100/1000 Mb/s o un Ethernet MAC de 10/100 Mb/s. Este núcleo soporta el uso de interfaces MII, GMII, SGMII, RGMII y 1000BASE-X para conectar al control de acceso al medio (MAC) a un chip de interfaz del lado físico (PHY). El MII puede conectar la MAC y el PHY externo mediante un conector o directamente al chip del PHY [35].

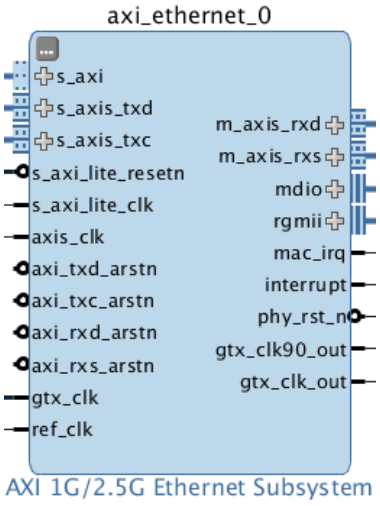


Figura 51: Bloque IP AXI 1G/2.5G Ethernet Subsystem

Para esta plataforma se empleará el estándar RGMII en el que se reduce el número de señales requeridas para conectar el PHY al MAC.

El subsistema proporciona una interfaz AXI4-Lite para conectar al núcleo procesador y permitir el acceso a los registros. La interfaz esclava aporta lectura y escritura de transferencias de datos únicas (no ráfagas). También se proporcionan buses AXI4-Stream de 32 bits para mover los datos Ethernet recibidos y transmitidos. Estos buses están diseñados para ser usados con AXI DMA, AXI-Stream Data FIFO u otro dispositivo soportado.

Al realizar la implementación de la plataforma, al bus AXI4-Stream de 32 bits se conectará el bloque IP de *Header Analyzer,* el cual cuenta con la misma interfaz, capaz de soportar esta conexión.

Los puertos del subsistema dependen del modo de funcionamiento y son agrupados en interfaces según su funcionalidad. Estas interfaces tienen asociadas puertos de *rst* y *clk*. Las interfaces I/O utilizadas en este proyecto se listan en la Tabla 8.

Tabla 8: Interfaces del bloque Ethernet [35]

NOMBRE DE LA INTERFAZ	DESCRIPCIÓN
S_AXI	Interfaz AXI4-Lite usada para configurar el subsistema AXI Ethernet. En modo procesador esta interfaz mapea al buffer AXI Ethernet que configura el TEMAC. En modos no-procesador, esta interfaz mapea y configura directamente el TEMAC.
S_AXIS_TXC	Control de transmisión AXI4-Stream
S_AXIS_TXD	Datos de transmisión AXI4-Stream
S_AXIS_RXD	Datos de recepción AXI4-Stream
S_AXIS_RXS	Estado de recepción AXI4-Stream

Los puertos del sistema se describen en la Tabla 9:

Tabla 9: Puertos del bloque Ethernet [35]

NOMBRE DEL PUERTO	DESCRIPCIÓN
PHY_RST_N	Señal de *reset* TEMAC a PHY. Se activa a nivel bajo si se mantiene activa por 10 ms después de que se le aplique potencia. Después de que el *reset* se vuelva inactivo, no se puede acceder al PHY por 5 ms.
REF_CLK	Reloj global. Frecuencia de 200 MHz para la serie 7 de FPGA.

NOMBRE DEL PUERTO	DESCRIPCIÓN
GTX_CLK	Reloj de 125 MHz usado en todas las configuraciones GMII, RGMII, y SGMII para el control del *reset* PHY.
MDIO	Interfaz MDIO para configurar el PHY. Esta interfaz es presente en modos MII, GMII, RGMII y SGMII.
RGMII	Esta interfaz está presente solo en el modo RGMII.

La configuración del bloque Ethernet a instanciar se muestra en la Figura 52, donde se seleccionará la velocidad Ethernet a 1 Gbps y también se elegirá la interfaz física que se va a utilizar, que será la RGMII.

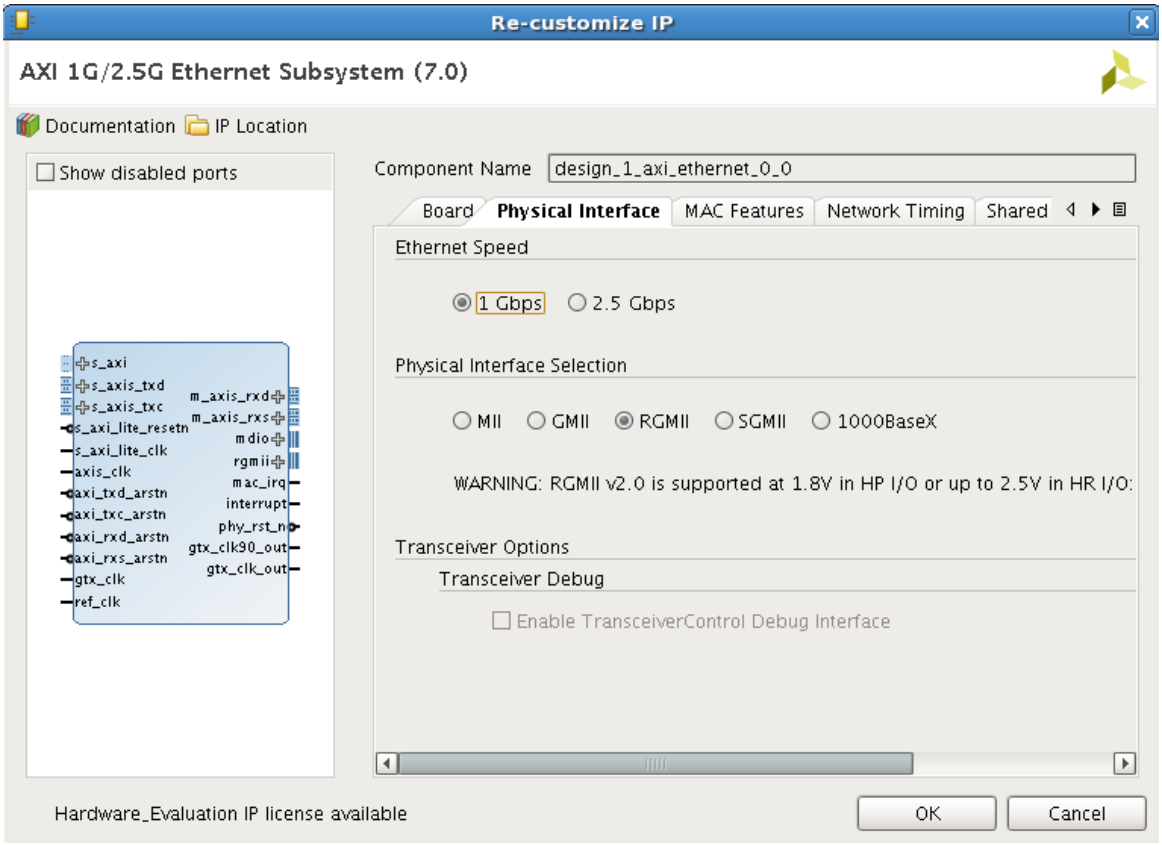


Figura 52: Configuración del bloque Ethernet (I)

La plataforma requiere dos bloques TEMAC, uno para la entrada de datos y otro para la salida después de su procesado. Cuando existe más de un bloque TEMAC en el mismo sistema, es recomendable seleccionar el modo *Include Share Logic in IP Example Design* con el fin de compartir recursos (Figura 53).

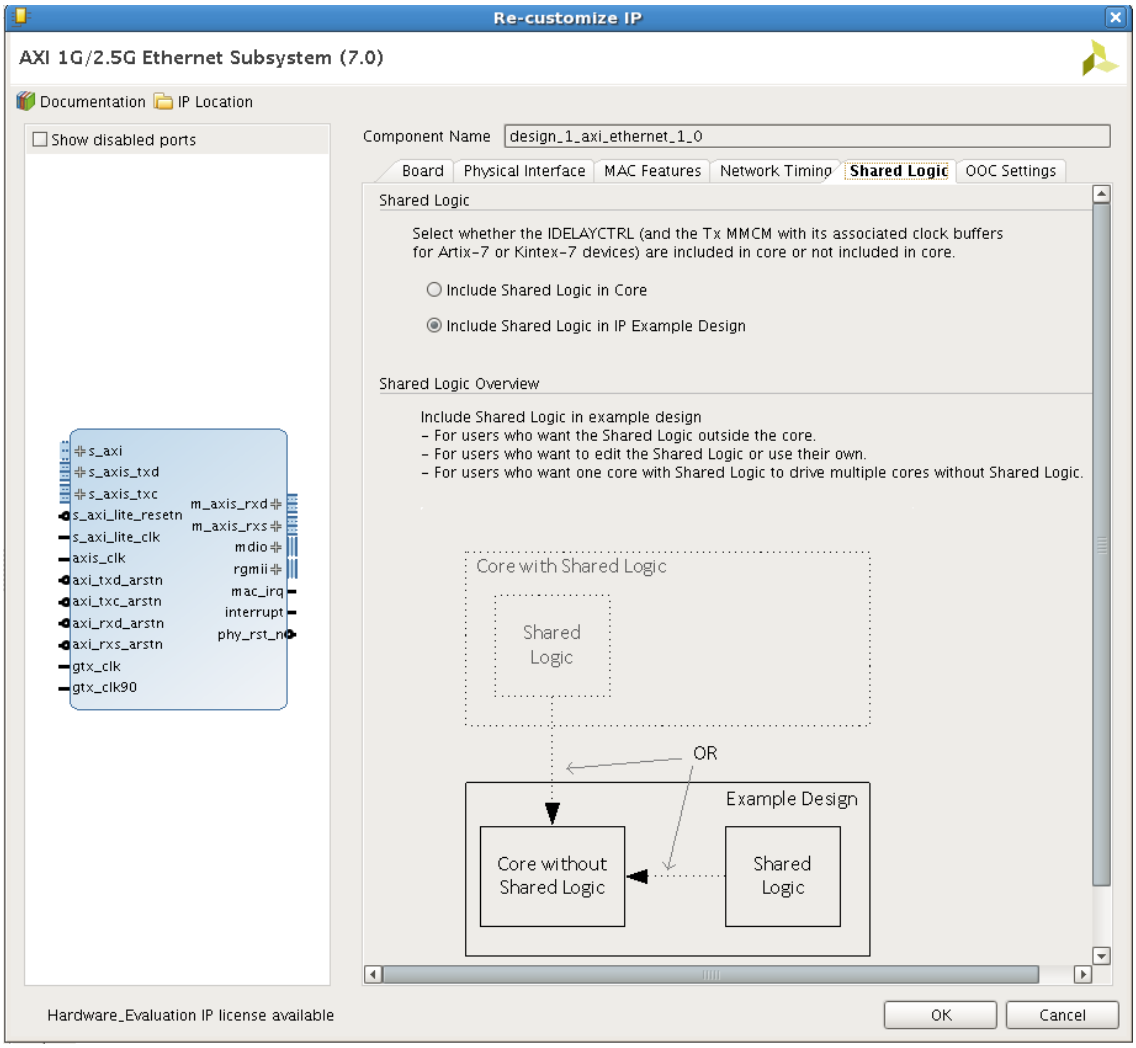


Figura 53: Configuración del bloque Ethernet (II)

Al seleccionar este modo es necesario que la interconexión entre ambos bloques sea la que se muestra en la Figura 54 para compartir la lógica.

Los bloques TEMAC proporcionan a la plataforma control de acceso al medio, capaz de conectarse con la capa física a través de una interfaz RGMII. Para implementar este medio físico se ha utilizado la placa EthernetFMC de Ospero (Figura 55), que proporciona cuatro conectores físicos que permiten tráfico Gigabit y se conecta a través de los conectores de expansión de la placa de desarrollo Zynq ZC706 [36].

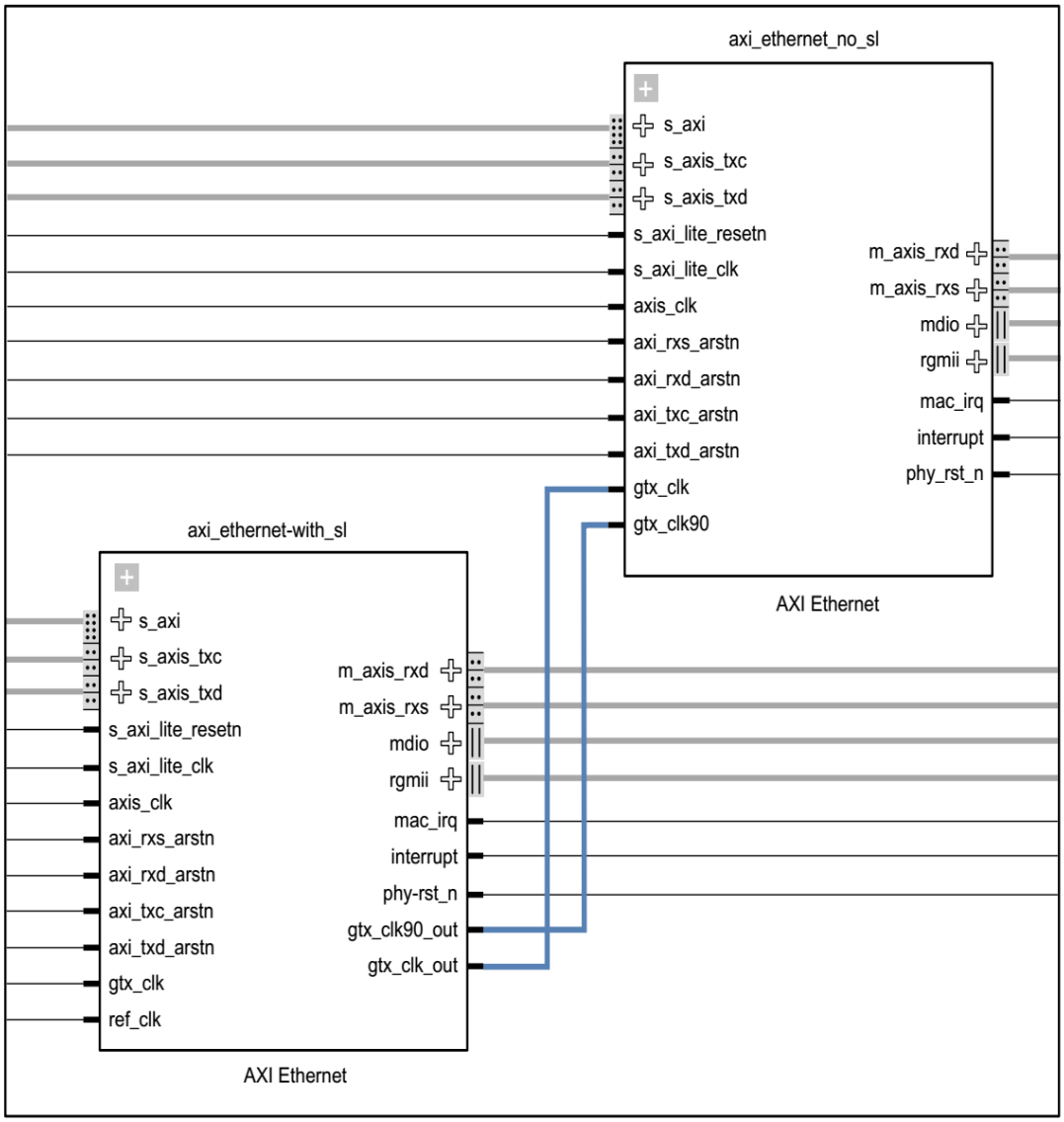


Figura 54: Conexión para habilitar el modo *Shared Logic* [35]

Para realizar la conexión entre los bloques TEMAC y los controladores PHY es necesario conocer los pines que conforman la FPGA y la placa EthernetFMC. A través del fichero de restricciones se realiza esta conexión, asignando las etiquetas usadas en el diseño a los pines y definiendo los niveles de tensión necesarios para la comunicación.

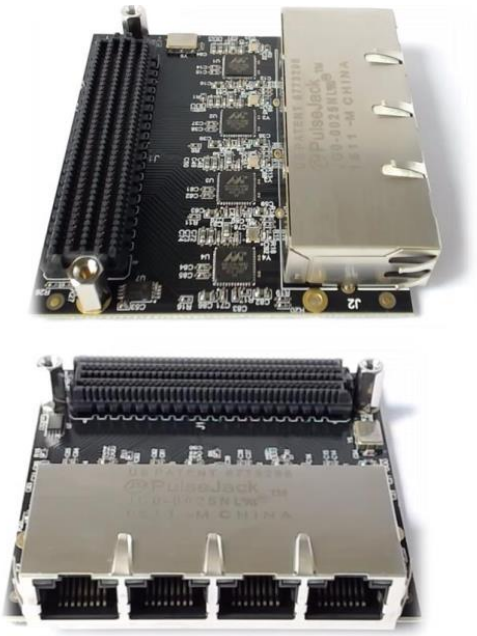


Figura 55: Placa EthernetFMC

6.2.5 Header Analyzer

Como se explicó en el Capítulo 5, el bloque en desarrollo se ha empaquetado y ahora puede ser añadido a la plataforma a través del catálogo IP (Figura 56). Este bloque posee interfaces de entrada y salida AXI4-Stream que recibe la trama Ethernet y la reenvía por una u otra interfaz dependiendo del filtrado. Además, cuenta con una interfaz AXI4-Lite de configuración al que irá conectado un bloque AXI *Interconnect*.

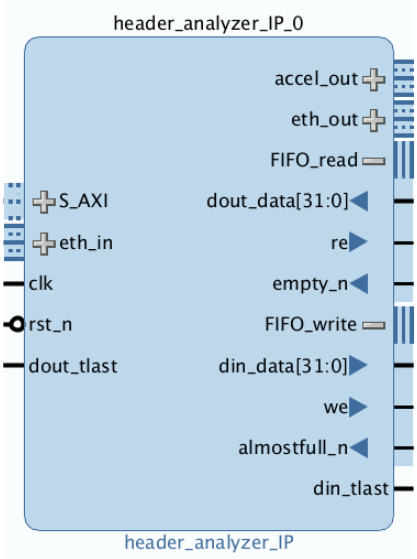


Figura 56: Bloque IP del *Header Analyzer*

Como se indicó en el apartado 5.9 Encapsulado del bloque IP, la herramienta Cadence Stratus divide la interfaz de datos FIFO implementada en alto nivel, por lo que existen dos puertos, uno de datos y otro de *last*. Ambas señales deben ir conectadas al puerto de datos de la FIFO, debido a que en ella se tiene en cuenta tanto el dato como la señalización *last.*

Los puertos *din_data* y *din_tlast* deben estar unidos para ser conectados al puerto de datos de entrada de la FIFO. Para hacer esta unión se utilizará un bloque de concatenación denominado *Concat* (Figura 57), al que a través de dos entradas genera una única salida situando el dato recibido por el puerto dos detrás del dato que se recibe a través del primer puerto. Por lo tanto, se obtiene una salida de 33 bits que coincide con la entrada de 33 bits de la FIFO.

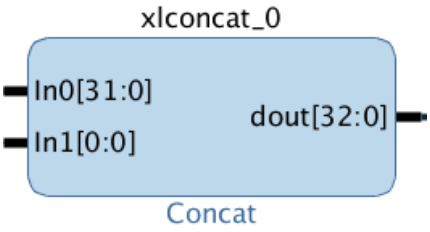


Figura 57: Bloque IP *Concat* para concatenar señales

En el caso del dato de salida de la FIFO, contiene tanto el dato como la señal *tlast* y es necesario dividir los 32 primeros bits de datos del último para que pueda ser conectado tanto al puerto *dout_data* y *dout_tlast* del bloque *Header Analyzer.* Para ello se utiliza el bloque IP *Slice* (Figura 58) donde, a partir de una entrada, se puede seleccionar un rango de este dato y obtenerlo a la salida.

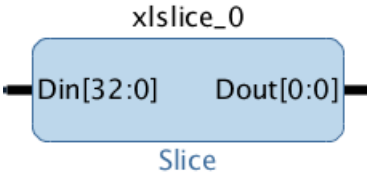


Figura 58: Bloque IP *Slice* para separar señales

6.2.6 Memorias FIFO

Mientras el bloque IP *Header Analyzer* realiza el filtrado de la cabecera del paquete, el resto del paquete se almacena en una FIFO externa hasta que el análisis de la cabecera se ha completado, donde se recupera el paquete de esta FIFO y se redirecciona por la interfaz correspondiente.

Se instanciará el bloque IP FIFO generado mediante el IP FIFO *Generator* (Figura 59), que proporciona distintos tipos de configuración para optimizar la solución al mismo tiempo que se utiliza el mínimo de recursos. Este bloque IP consigue funcionar hasta 500 MHz, maximizando el rendimiento del sistema. Este bloque puede ser personalizado por el usuario, seleccionando el ancho, profundidad, *flags*, tipos de memorias y rangos de los puertos. Además soporta distintas interfaces como la nativa, AXI Memory Mapped o AXI4-Stream [37].

Figura 59: Bloque IP FIFO

Para la implementación de esta plataforma se utilizará una interfaz FIFO nativa implementada sobre una memoria de bloque de la FPGA (BRAM), con un tamaño de ancho de palabra de 33 bits y una profundidad de 1024 palabras para que pueda albergar dos paquetes a tamaño completo.

6.2.7 Unidad de Despacho

La *Unidad de Despacho* es un bloque previamente diseñado por un tercero [38], del que se ha tomado para implementarlo en la plataforma DPI. Este bloque es el encargado

de tomar los paquetes entrantes y distribuirlos a los distintos bloques aceleradores. Una peculiaridad que tiene este bloque es que, a través de la interfaz AXI4-Lite (en la Figura 60 como S_AXI), se puede configurar el bloque con distintos modos de redireccionamiento, pudiendo elegir entre *unicast* o comunicación con un único bloque acelerador, *broadcast* o comunicación con todos los bloques aceleradores y, por último, segmentación, que divide el paquete en tantos fragmentos como bloques aceleradores hay.

Este bloque originalmente tomaba los paquetes directamente de la red a través de la interfaz *data_in_ethernet*. Como en este proyecto, el bloque *Header Analyzer* es el encargado de tomar los paquetes de la red, se conectará la interfaz de salida *accel_out* del *Header Analyzer* con esta interfaz de entrada *data_in_ethernet*, debido a que ambas interfaces utilizan el protocolo AXI4-Stream.

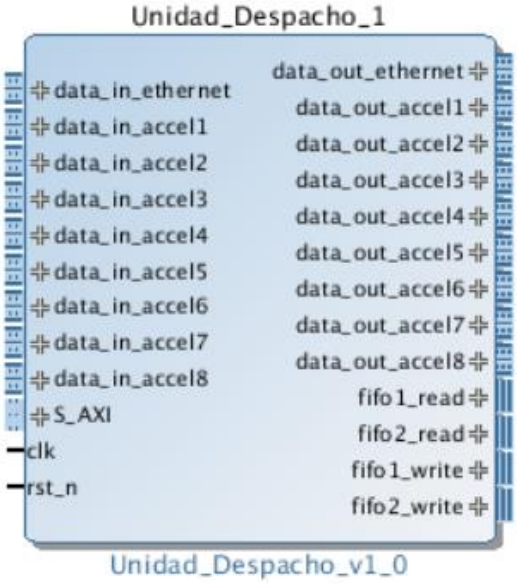


Figura 60: Bloque IP *Unidad de Despacho*

6.2.8 AXI4-Stream Switch

El bloque AXI4-Stream Switch (Figura 61) es necesario debido a que existen dos bloques que a través de sus interfaces de salida transmiten paquetes a la red: por una parte, el bloque *Header Analyzer,* si el filtrado de la cabecera no coincide y por otra, la *Unidad de Despacho,* si recibe una respuesta favorable por parte de los bloques aceleradores. Es por ello por lo que se incorpora este bloque con el fin de rutear cualquiera de las dos salidas a la entrada del bloque Ethernet. Este bloque contiene un sistema de

arbitraje Round Robin que evita colapsar el sistema debido a que comprueba ambas interfaces de manera cíclica [29].

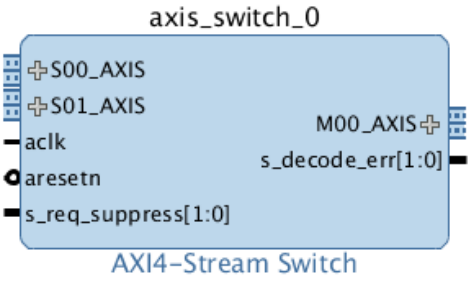


Figura 61: Bloque IP AXI4-Stream Switch

6.2.9 Eliminar Cabecera

Para que los aceleradores examinen el paquete recibido es necesario que la *Unidad de Despacho* transmita únicamente el *payload* o carga útil del paquete. Aprovechando que la *Unidad de Despacho* tiene implementada FIFOs externas al bloque que almacenan el paquete para su posterior envío a los bloques aceleradores, se implementa este bloque que eliminará la cabecera del paquete (Figura 62) y almacenará en la FIFO únicamente el *payload.*

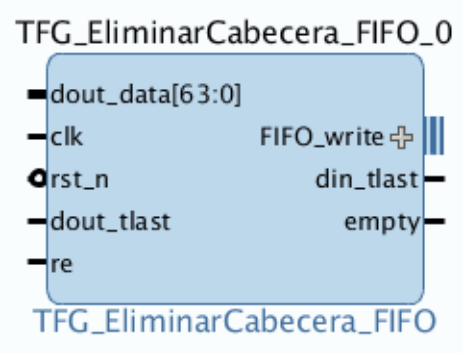


Figura 62: Bloque IP *Eliminar Cabecera*

6.2.10 Bloque acelerador

Para la implementación de los ocho bloques aceleradores que necesita la *Unidad de Despacho,* se ha utilizado un bloque de un tercero que emula el comportamiento funcional de los bloques aceleradores [38].

Este bloque recibe el *payload* o la carga útil del paquete y a través de la interfaz *answer* se le fuerza el resultado de la inspección, mostrando este resultado a través de la interfaz de salida *data_out_accel*.

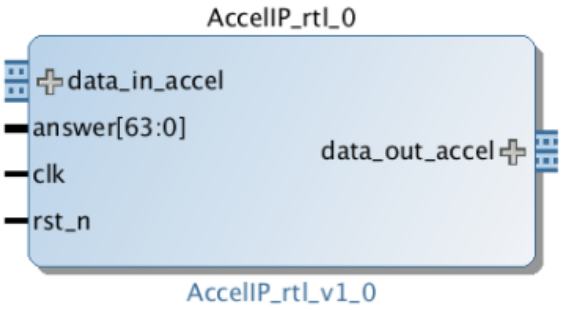


Figura 63: Bloque IP acelerador

6.2.11 ILA

ILA o *Integrated Logic Analyzer*, es un bloque necesario para observar el flujo de datos que tiene la plataforma durante el proceso de depurado y, por tanto, no estará implementado en la versión de la plataforma que se lleve a producción. Este bloque puede ser conectado a cualquier puerto con la configuración *custom,* así como a interfaces AXI debido a que soporta todos los tipos de protocolo AXI.

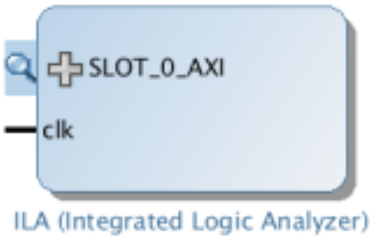


Figura 64: Bloque IP ILA

6.3 Diagrama de bloques

Una vez conocidos todos los bloques que conformar la plataforma, se realiza su integración. Para realizar esto se creará un nuevo proyecto en la herramienta Vivado Design Suite y se creará un *Block Design* para incorporar los distintos bloques del diseño. El resultado se muestra en la Figura 65, donde se pueden apreciar los distintos bloques que conforman la plataforma DPI.

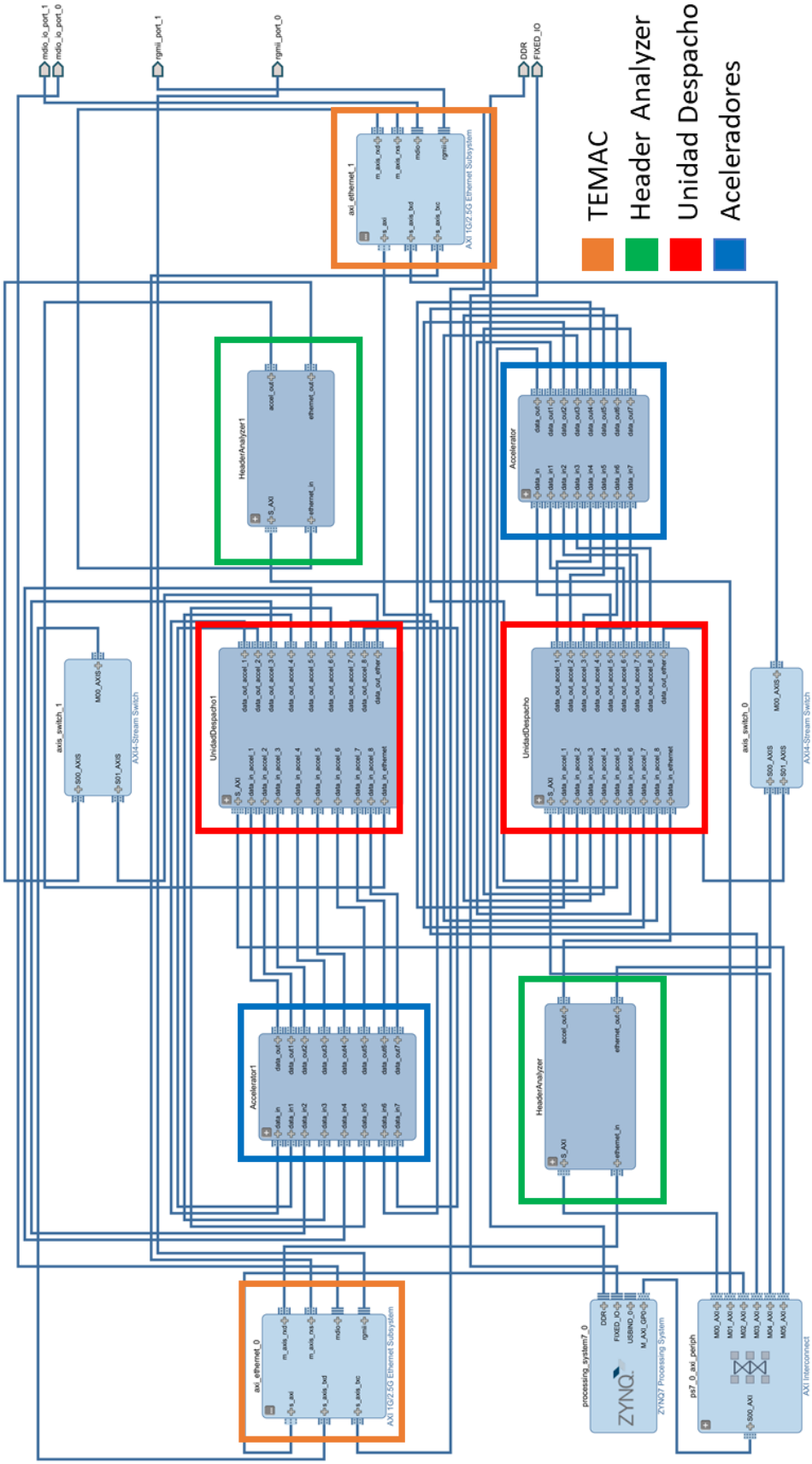


Figura 65: Diagrama de bloques de la plataforma DPI

La plataforma que se ha implementado tiene la peculiaridad de que es bidireccional. Esto se traduce en duplicar los bloques para permitir realizar el filtrado de la cabecera y el análisis de la carga útil del paquete en ambas direcciones y, por lo tanto, aumentar el tamaño y los recursos a utilizar. Una plataforma de este tamaño resulta complicada de manejar, por lo que se han creado jerarquías las cuales contienen diversos bloques.

En la Figura 66 se muestra la jerarquía de *Header Analyzer*, en la que está contenida el propio bloque y la FIFO que guardará el paquete mientras se realiza el filtrado. Además, se incluyen diversos bloques para concatenar los datos con la señalización de fin del paquete para que sea agrupado a la entrada de la FIFO, o bloques lógicos donde se implementan funciones de *glue-logic* NOT debido a que las señales de las FIFOs fueron diseñadas a nivel bajo.

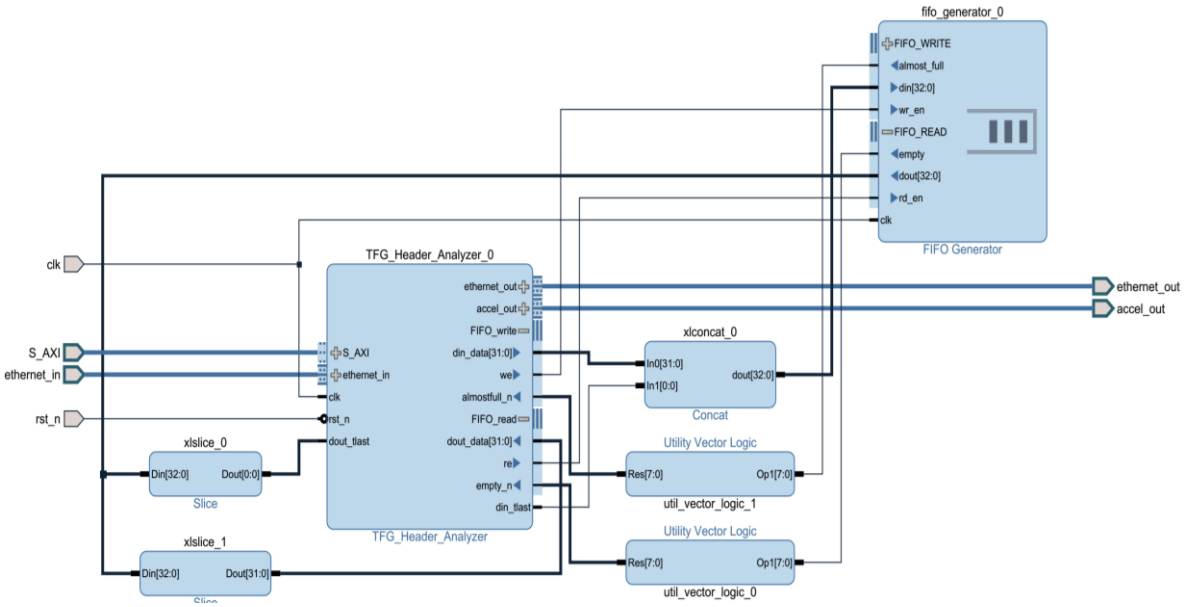


Figura 66: Jerarquía de *Header Analyzer*

La jerarquía de la *Unidad de Despacho* se muestra en la Figura 67. En ella se aprecia el bloque de *Unidad de Despacho* y las dos FIFOs que se implementan de manera externa, almacenando en una de ellas el paquete completo para devolverlo a la red si el análisis en profundidad dictamina que puede ser redireccionado por la interfaz Ethernet. La segunda FIFO almacena únicamente el *payload* para enviárselo a los bloques aceleradores. Es entre esta FIFO y el bloque de *Unidad de Despacho* donde se sitúa el bloque *Eliminar Cabecera*, con el fin de almacenar en la FIFO únicamente la carga útil del paquete.

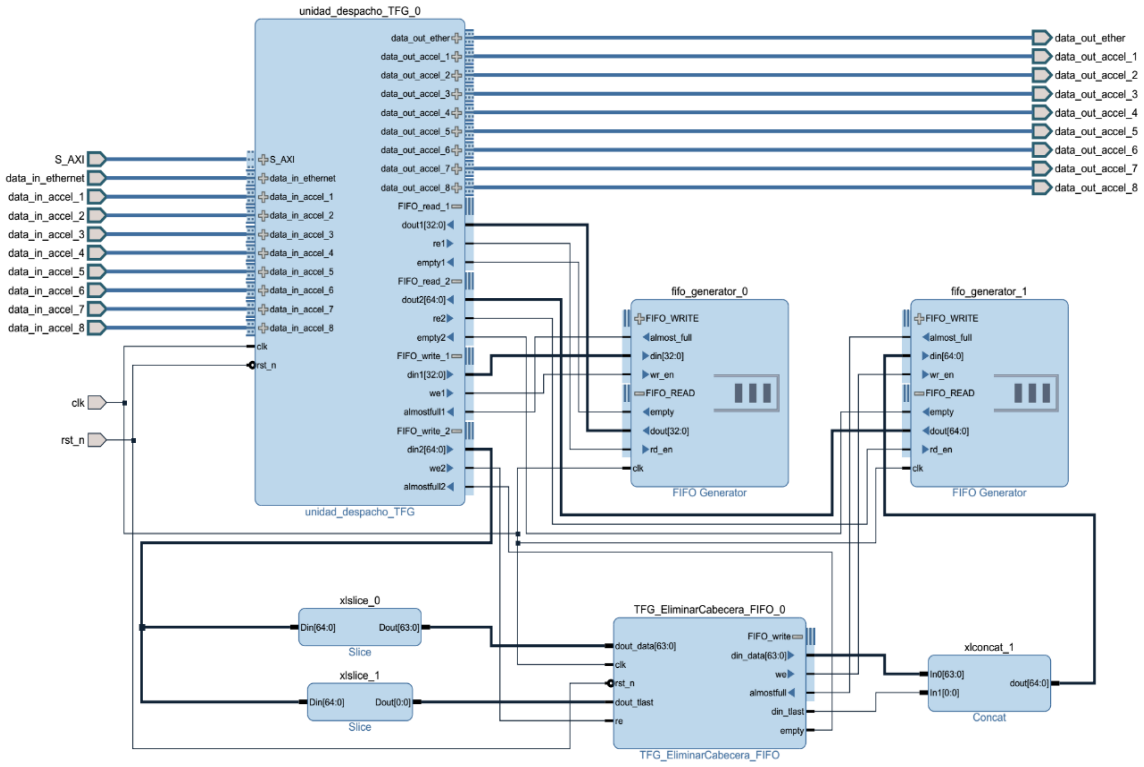


Figura 67: Jerarquía *Unidad de Despacho*

6.4 Síntesis e implementación de la plataforma

Una vez finalizado el montaje de la plataforma y de realizar una validación de la misma, se requiere realizar dos pasos adicionales antes de completar el diseño. El primero, es la generación de los *outputs products*. Esto significa generar todos los ficheros fuente y archivos de restricciones de los bloques IP para que la herramienta los tenga disponibles.

El segundo paso necesario para que el diseño esté integrado a un alto nivel crear un *wrapper* HDL (*Hardware Description Level*), generando un fichero a alto nivel para todos los IP que integra la plataforma. Una vez realizados estos dos pasos el diseño está listo para ser sintetizado e implementado.

Antes de realizar el proceso de síntesis e implementación se deben especificar las restricciones temporales que tiene la plataforma. El PS proporciona un reloj de 200 MHz, el cual está conectado a todos los bloques a excepción del bloque TEMAC. Este bloque necesita de varios dominios de reloj. Uno de ello es de 200 MHz, si bien puede estar conectado al reloj anteriormente descrito, se ha decidido crear un reloj nuevo por facilitar

la adaptación de la plataforma si se da el caso de que los bloques diseñados son sustituidos y se puede incrementar la frecuencia. El otro dominio es de 125 MHz.

Además, se puede especificar a la herramienta diversos parámetros de optimización con el fin de mejorar las prestaciones de la plataforma. La estrategia más usada es el *retiming* que mejora la frecuencia del sistema a costa de recursos. Como se ha venido comentando, el objetivo de esta plataforma no es conseguir altas velocidades, sino ahorrar en recursos.

6.4.1 Resultados de la síntesis

Una vez realizada la síntesis, la herramienta Vivado Design Suite proporciona informes sobre los resultados obtenidos en cuanto a término de ciclo de reloj, recursos utilizados y potencia consumida. En la Figura 68 se muestra el informe sobre la cantidad de recursos que se ha utilizado en la plataforma y en la Figura 69 el análisis de la potencia consumida por el sistema.

Resource	Utilization	Available	Utilization %
LUT	12117	218600	5.54
LUTRAM	818	70400	1.16
FF	22745	437200	5.20
BRAM	16	545	2.94
IO	34	362	9.39
MMCM	1	8	12.50

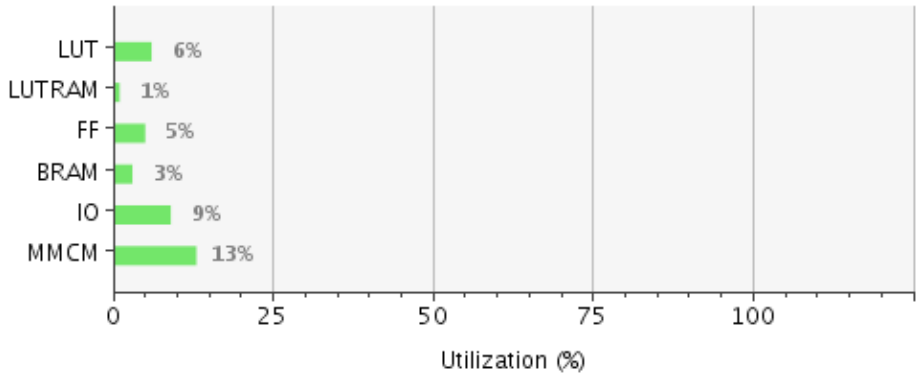


Figura 68: Utilización de recursos en la síntesis

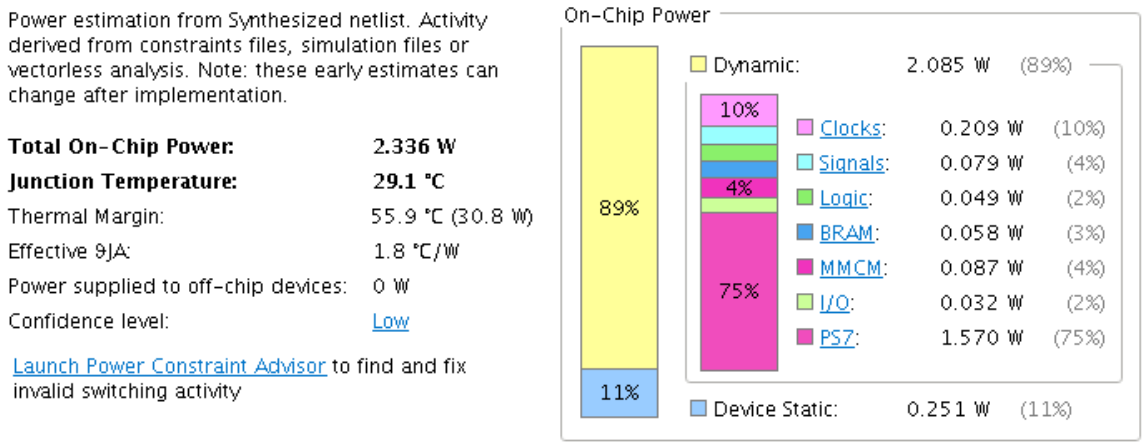


Figura 69: Análisis de potencia en la síntesis[1]

También se puede encontrar la ruta crítica del sistema. En la Figura 70 se observa que esta ruta se encuentra en el bloque TEMAC desde la interfaz RGMII de salida hasta el puerto RGMII.

Name	Slack	From	To	Total Delay
Path 201	0.450	design_1_i/axi_ethernet_0/inst...ta_out_bus[0].rgmii_txd_out/C	rgmii_port_0_td[0]	1.804
Path 202	0.505	design_1_i/axi_ethernet_0/ins...a_out_bus[2].rgmii_txd_out/C	rgmii_port_0_td[2]	1.748
Path 203	0.506	design_1_i/axi_ethernet_0/ins...a_out_bus[1].rgmii_txd_out/C	rgmii_port_0_td[1]	1.748
Path 204	0.534	design_1_i/axi_ethernet_0/ins...a_out_bus[3].rgmii_txd_out/C	rgmii_port_0_td[3]	1.720
Path 205	0.535	design_1_i/axi_ethernet_0/inst...i/rgmii_interface/ctl_output/C	rgmii_port_0_tx_ctl	1.719

Figura 70: Ruta crítica en la síntesis

6.4.2 Implementación del diseño

El siguiente paso después de la síntesis es la implementación. Una vez finalizada se pueden obtener los informes temporales, de potencia y de recursos.

Al realizar el análisis temporal del sistema y sabiendo que la frecuencia global del mismo es de 200 MHz, se obtiene que el diseño tiene un *slack* positivo y por tanto no existen problemas temporales (Figura 71).

[1] En las figuras se muestran los resultados con el punto como separador decimal

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.160 ns	Worst Hold Slack (WHS):	0.050 ns	Worst Pulse Width Slack (WPWS):	0.264 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:	49000	Total Number of Endpoints:	49000	Total Number of Endpoints:	20816

All user specified timing constraints are met.

Figura 71: Análisis temporal del diseño en la implementación

Además, se puede encontrar la ruta crítica del sistema. De manera similar a la síntesis, la ruta crítica se encuentra entre el bloque TEMAC y el PS, a través de los registros del AXI4-Lite.

Name	Slack	From	To	Total Delay	Logic Delay
Path 1	0.160	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_eth...data_int_reg[7]/D	4.299	1.127
Path 2	0.163	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_eth...data_int_reg[4]/D	4.326	1.127
Path 3	0.167	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_eth...data_int_reg[9]/D	4.260	1.127
Path 4	0.167	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...g_data_reg[26]/D	4.318	1.170
Path 5	0.167	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...g_data_reg[24]/D	4.296	1.127
Path 6	0.169	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...g_data_reg[27]/D	4.292	1.127
Path 7	0.173	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...eg_data_reg[24]/D	4.290	1.127
Path 8	0.174	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...g_data_reg[24]/D	4.283	1.127
Path 9	0.177	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...g_data_reg[27]/D	4.263	1.127
Path 10	0.179	design_1_i/processing_system7_0/inst/PS7_i/MAXIGP0ACLK	design_1_i/axi_et...g_data_reg[22]/D	4.291	1.127

Figura 72: Ruta crítica en la implementación

El informe de utilización de recursos se observa en la Figura 73, en donde se muestra el porcentaje de utilización del sistema DPI sobre el dispositivo. En comparación con la Figura 68, los resultados entre la síntesis y la implementación varían mínimamente.

Resource	Utilization	Available	Utilization %
LUT	11507	218600	5.26
LUTRAM	826	70400	1.17
FF	19837	437200	4.54
BRAM	14	545	2.57
IO	34	362	9.39
MMCM	1	8	12.50

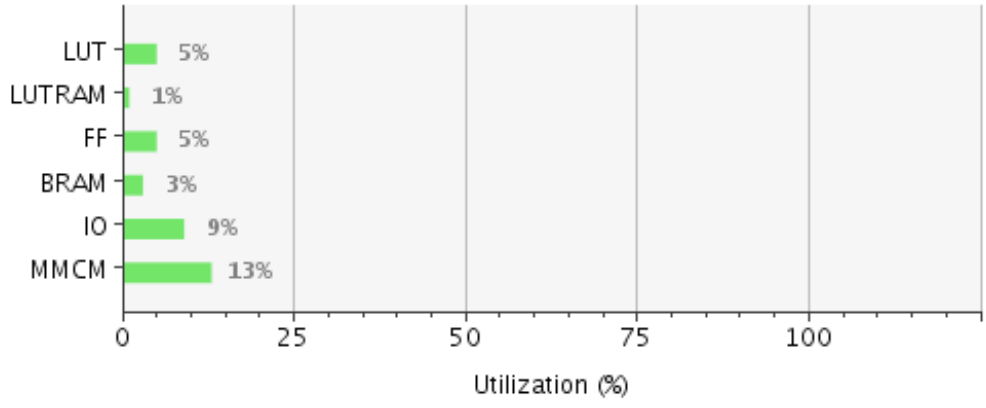


Figura 73: Análisis de recursos en la implementación

También se puede realizar una comparación entre la ocupación total del dispositivo, de la plataforma y de los bloques implementados, con el fin de conocer su tamaño (Tabla 10).

Tabla 10: Ocupación de los distintos bloques

NOMBRE	SLICE LUTS	SLICE REGISTERS	SLICE	LUT AS LOGIC	LUT FLIP FLOP PAIRS
TOTAL FPGA	218.600	437.200	54.650	218.600	218.600
DPI	**11.507 (5%)**	**19.837 (4,5%)**	**6.031 (11%)**	**10.681 (4,9%)**	**6.187 (2,8%)**
TEMAC	3.051 (1,4%)	5.258 (1,2%)	1.702 (3,11%)	2.681 (1,23%)	1.579 (0,72%)
HEADER ANALYZER	467 (0,24%)	1.220 (0,3%)	338 (0,68%)	467 (0,24%)	181 (0,09%)
ELIMINAR CABECERA	15 (0,01%)	42 (0,01%)	9 (0,02%)	15 (0,01%)	9 (0,01%)
UNIDAD DE DESPACHO	1.155 (0,53%)	1.108 (0,25%)	470 (0,86%)	1.155 (0,53%)	698 (0,32%)
ACELERADORES	333 (0,15%)	1.347 (0,31%)	320 (5,3%)	333 (0,15%)	245 (0,11%)

Los distintos bloques implementados en la plataforma no tienen un gran impacto de ocupación. Esto permite la incorporación de nuevos bloques a la plataforma para trabajos futuros, sin que existan problemas de espacio

En la Figura 74 se muestra el *layout* de la plataforma, donde se puede observar la ocupación del diseño sobre el dispositivo. Representado por colores se muestra la ocupación de los distintos bloques. Cabe mencionar la baja ocupación de los aceleradores, debido a que son bloques *dummy* que imitan su comportamiento.

Por último, se realiza el análisis de potencia consumida por la plataforma como se muestra en la Figura 75, en la que se hace distinción sobre potencia dinámica y estática, donde esta última no puede ser gestionar. Se obtiene que la potencia total del sistema en el chip es de 2,243 W. siendo la mayor parte de este consumo proveniente del microprocesador, 1,570 W.

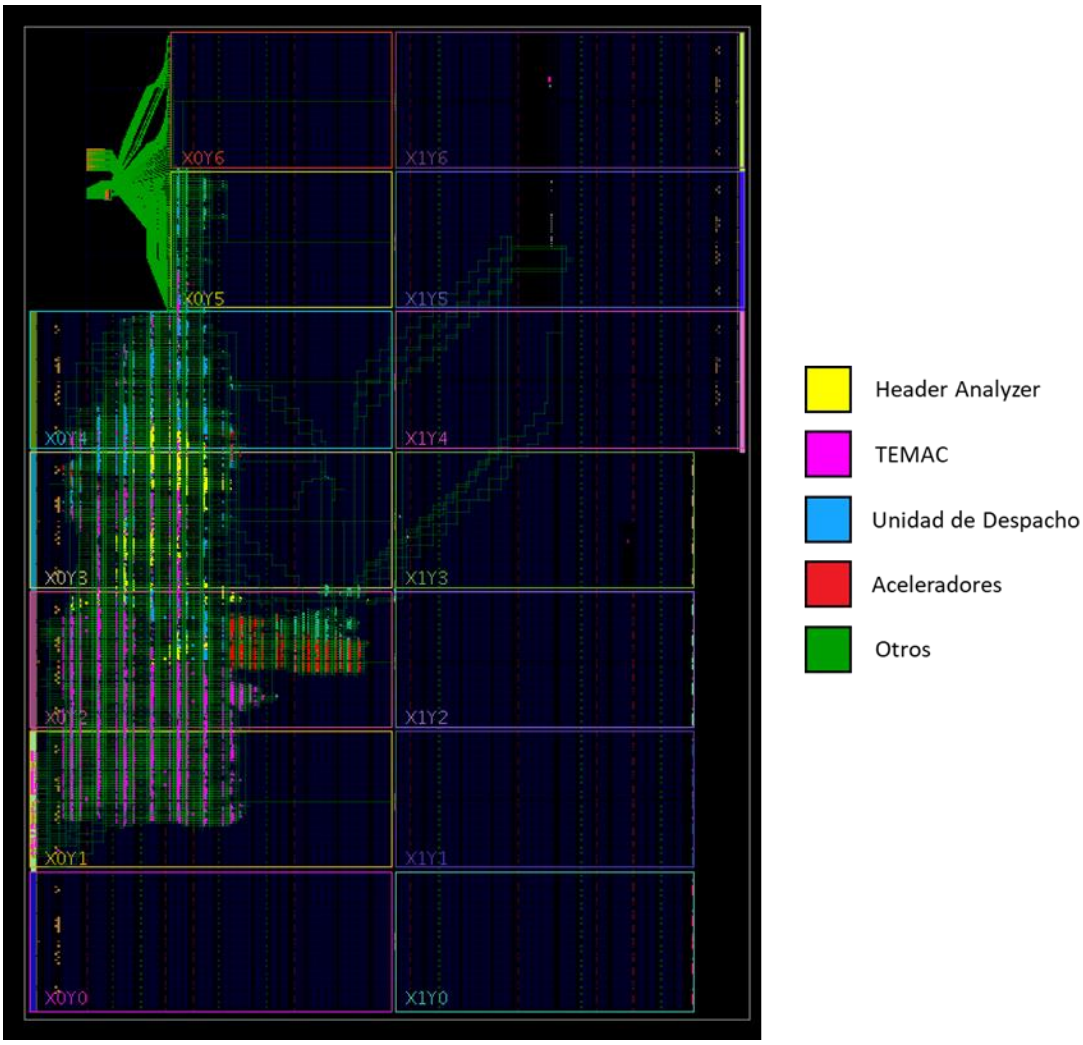


Figura 74: *Layout* del diseño

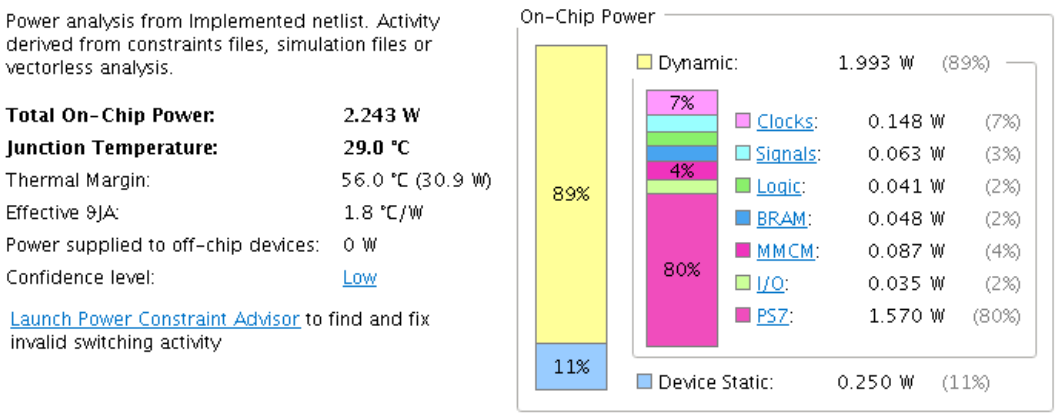


Figura 75: Análisis de potencia en la implementación[2]

[2] En la figura se muestran los resultados con el punto como separador decimal

6.5 Exportar el *hardware* de la plataforma

Una vez la plataforma completa esté sintetizada e implementada y que, a través de los diversos informes se comprueba que cumple con los requisitos, se procede a exportar el *hardware* de la plataforma para su posterior utilización en la herramienta de desarrollo del *software* empotrado SDK.

Este *hardware* contendrá información sobre los bloques que componen el sistema y también incluirá el *bitstream* para realizar la programación del dispositivo. Desde Vivado se exportará el *hardware* al entorno de desarrollo SDK para la creación de la aplicación *software*, además de realizar la creación del archivo BSP (*Board Support Package)* que contiene los distintos *drivers* y librerías que son usadas por la aplicación. Con el *bitstream* importado del Vivado y el BSP se puede realizar la programación de la FPGA y realizar la verificación del sistema.

6.6 Conclusiones

En este capítulo se muestran los distintos bloques que conforman el sistema DPI. Conocido estos bloques, se ha procedido a la creación de la plataforma la cual se ha sintetizado e integrado al dispositivo.

Una vez la plataforma esté integrada, se pueden realizar distintos informes sobre los recursos que han sido utilizados, el consumo de potencia o datos temporales de la plataforma. Con estos informes se puede concluir que la plataforma implementada tiene un bajo nivel de ocupación, que puede operar a una frecuencia de 200 MHz y que la mayor parte del consumo de potencia viene dado por el microprocesador, reafirmando el uso de FPGAs como soluciones de baja potencia.

Capítulo 7. Integración *hardware-software*

7.1 Introducción

En este capítulo se explica la aplicación *software* creada para realizar la configuración del bloque *Header Analyzer.* Además, se realiza la integración de la aplicación que dispone la *Unidad de Despacho* y, junto con la del *Header Analyzer*, se constituye la aplicación del sistema DPI.

Esta aplicación se crea a través de la herramienta Xilinx SDK que toma el *hardware* que se ha generado de la plataforma, así como los *drivers* de los bloques IP implementados, permitiendo programar el dispositivo a través del *bitstream* y lanzar la aplicación creada. Además, permite realizar el depurado de esta aplicación a través de conector JTAG.

7.2 Diseño de la aplicación

Como se concluyó en el Capítulo 6, desde la herramienta Vivado Design Suite se exporta el *hardware* de la plataforma y se incluye en él el *bitstream* generado. Una vez lanzada la herramienta SDK se debe seguir una serie de pasos para desarrollar la aplicación [39].

7.2.1 Importar el *hardware* de la plataforma

El primer paso será crear un nuevo proyecto al cual se importará este *hardware*. En él se incluye la información de todos los bloques implementados en la plataforma, además de:

1. Información sobre el procesador y los periféricos necesarios para la generación del BSP
2. Información del mapa de memoria requerido en el diseño para generar los *scripts*
3. El *bitstream* para programar el dispositivo
4. Datos de configuración del PS

En la Figura 76 se pueden apreciar los distintos ficheros que componen el *wrapper* de la plataforma *hardware*, donde se incluye el *bitstream*, los distintos ficheros para el manejo del PS y el *hardware* con el fichero de extensión ".hdf". Además, en la Figura 77 se muestra el mapa de memoria requerido por el diseño.

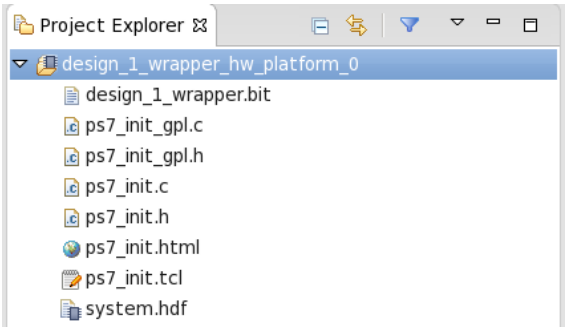


Figura 76: Contenido de la descripción *hardware* exportada desde Vivado

7.2.2 Creación del BSP

Un BSP (*Board Support Package)* es una colección de librerías y *drivers* que conforman la capa más baja de la aplicación. Para su creación (Figura 78) es necesario indicar el nombre del mismo y seleccionar la plataforma *hardware* que irá asociada a este BSP, así como la elección de uno de los dos microprocesadores que contiene el dispositivo. Además, es posible elegir un sistema operativo tipo FreeRTOS si la aplicación lo requiere o por lo contrario un *Standalone* para aplicaciones básicas que manejen excepciones o interrupciones.

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
ps7_intc_dist_0	0xf8f01000	0xf8f01fff		REGISTER
ps7_gpio_0	0xe000a000	0xe000afff		REGISTER
ps7_scutimer_0	0xf8f00600	0xf8f0061f		REGISTER
ps7_slcr_0	0xf8000000	0xf8000fff		REGISTER
ps7_scuwdt_0	0xf8f00620	0xf8f006ff		REGISTER
ps7_l2cachec_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_scuc_0	0xf8f00000	0xf8f000fc		REGISTER
ps7_qspi_linear_0	0xfc000000	0xfdffffff		MEMORY
axi_ethernet_0	0x41000000	0x4103ffff	s_axi	REGISTER
ps7_pmu_0	0xf8893000	0xf8893fff		REGISTER
ps7_afi_1	0xf8009000	0xf8009fff		REGISTER
ps7_afi_0	0xf8008000	0xf8008fff		REGISTER
ps7_qspi_0	0xe000d000	0xe000dfff		REGISTER
ps7_usb_0	0xe0002000	0xe0002fff		REGISTER
UnidadDespacho1_unidad_de	0x48000000	0x4fffffff	S_AXI	REGISTER
ps7_afi_3	0xf800b000	0xf800bfff		REGISTER
ps7_afi_2	0xf800a000	0xf800afff		REGISTER
ps7_globaltimer_0	0xf8f00200	0xf8f002ff		REGISTER
ps7_dma_s	0xf8003000	0xf8003fff		REGISTER
ps7_iop_bus_config_0	0xe0200000	0xe0200fff		REGISTER
ps7_xadc_0	0xf8007100	0xf8007120		REGISTER
ps7_ddr_0	0x00100000	0x3fffffff		MEMORY
ps7_ddrc_0	0xf8006000	0xf8006fff		REGISTER
ps7_ocmc_0	0xf800c000	0xf800cfff		REGISTER
ps7_pl310_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_uart_1	0xe0001000	0xe0001fff		REGISTER
ps7_coresight_comp_0	0xf8800000	0xf88fffff		REGISTER
ps7_i2c_0	0xe0004000	0xe0004fff		REGISTER
ps7_ttc_0	0xf8001000	0xf8001fff		REGISTER
ps7_scugic_0	0xf8f00100	0xf8f001ff		REGISTER
ps7_ethernet_0	0xe000b000	0xe000bfff		REGISTER
HeaderAnalyzer1_TFG_Heade	0x60000000	0x7fffffff	S_AXI	REGISTER
axi_ethernet_1	0x41040000	0x4107ffff	s_axi	REGISTER
ps7_dev_cfg_0	0xf8007000	0xf80070ff		REGISTER
ps7_dma_ns	0xf8004000	0xf8004fff		REGISTER
HeaderAnalyzer_TFG_Heade	0x40000000	0x4007ffff	S_AXI	REGISTER
ps7_sd_0	0xe0100000	0xe0100fff		REGISTER
UnidadDespacho_unidad_des	0x50000000	0x5fffffff	S_AXI	REGISTER
ps7_gpv_0	0xf8900000	0xf89fffff		REGISTER
ps7_ram_1	0xffff0000	0xfffffdff		MEMORY
ps7_ram_0	0x00000000	0x0002ffff		MEMORY

Figura 77: Mapa de memoria de la plataforma

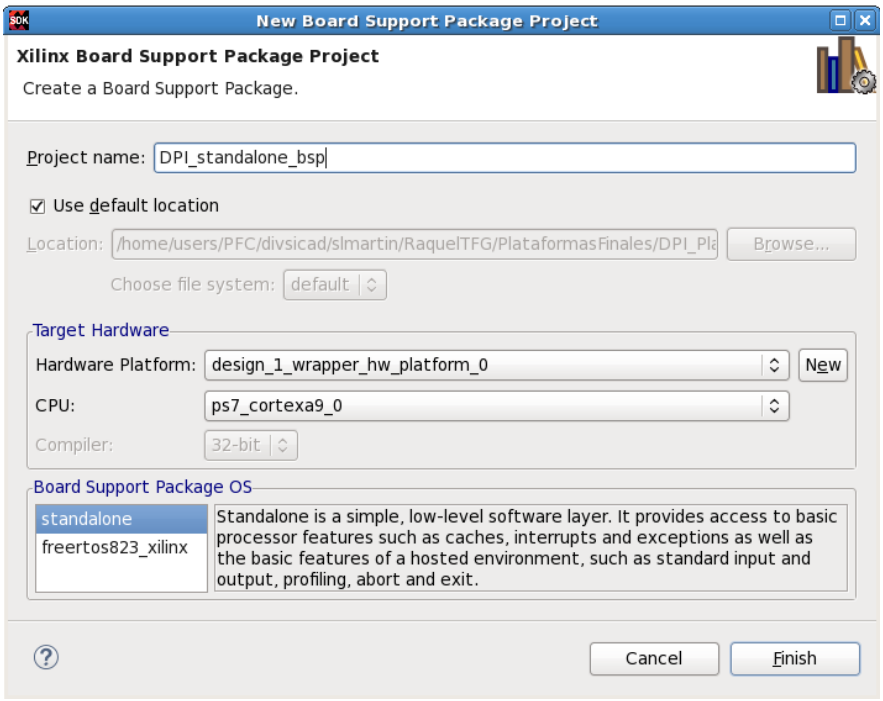


Figura 78: Creación del BSP

Una vez que se ha creado el BSP se configurará el mismo, incluyendo las librerías que sean necesarias para la aplicación. En este caso se utilizará la librería lwIP (Figura 79).

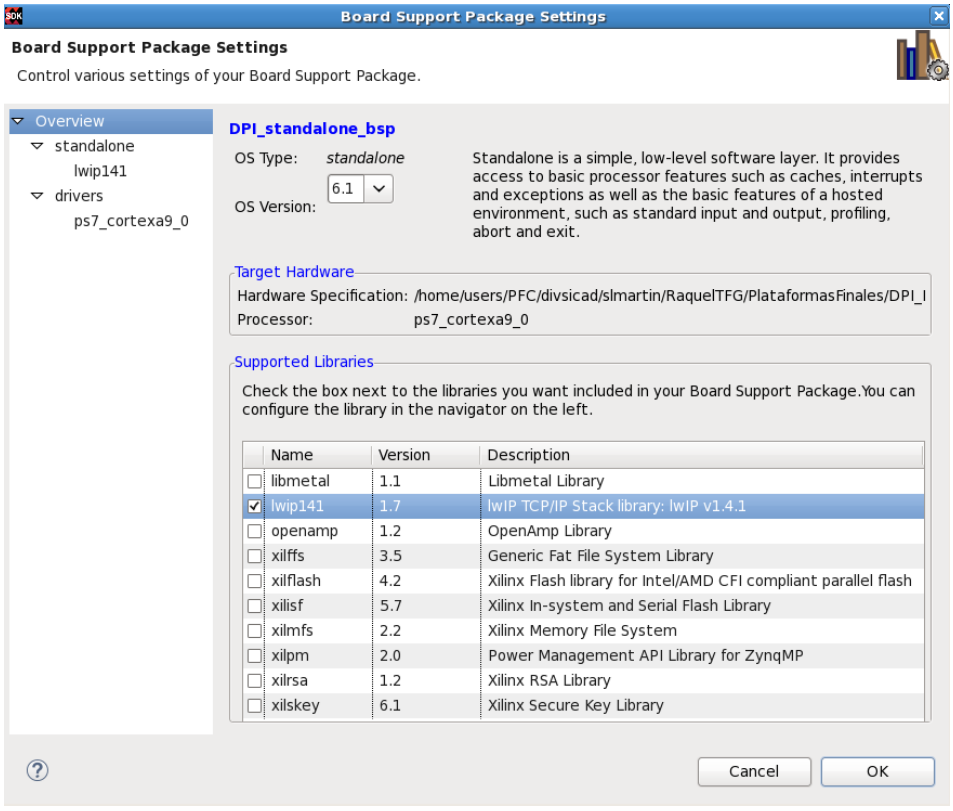


Figura 79: Elección de librerías del BSP

La librería Lightweight IP (lwIP) es un s*tack* TCP/IP de código abierto para sistemas embebidos soportado por los ARM que componen la serie Zynq-7000. Esta librería es usada para el desarrollo de aplicaciones como servidores web, *echo* o TFTP [40].

7.2.3 Creación de la aplicación

Una vez que se tiene la plataforma *hardware* importada y el BSP de la misma creada, se puede proceder a la creación de la aplicación. La herramienta SDK proporciona distintas plantillas basadas en aplicaciones con bastante uso común que puede servir de ayuda. En este caso, se creará una aplicación vacía.

Para la creación de la plataforma se le asigna un nombre, un sistema operativo pudiendo elegir entre *Standalone,* FreeRTOS o Linux. Se selecciona la plataforma *hardware* y el procesador a utilizar y por último se indica el lenguaje en el que se programará el *software* y el BSP asociado (Figura 80).

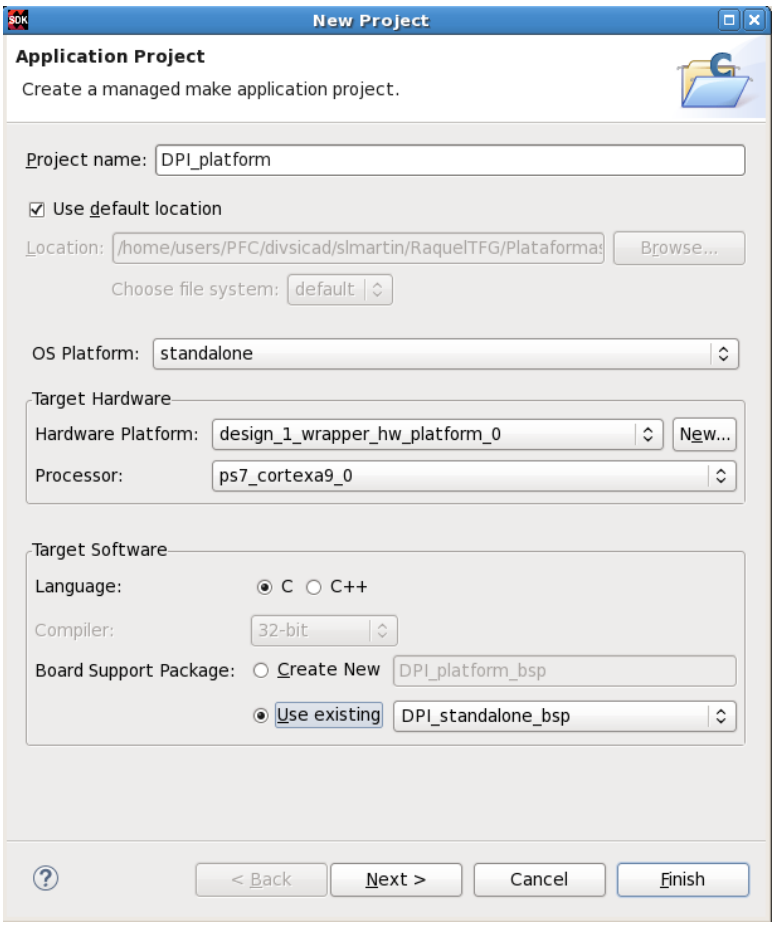


Figura 80: Creación de la aplicación

El desarrollo del *software* contará con dos partes diferenciadas:

1. la primera es la inicialización tanto de la plataforma como de los bloques TEMAC;
2. la segunda es la configuración de los bloques IP.

Cabe destacar que, aunque se ha tratado los bloques aceleradores y el de *Unidad de Despacho* como cajas negras debido a que son bloques de terceros, es necesario integrar sus funciones *software* a la aplicación junto con el *Header Analyzer.*

Esta integración se realiza en el *main* del programa (Código 20), donde se inicializa el sistema y se configuran todos los bloques que componen la plataforma. Esta configuración se realiza a través de llamadas a las librerías de los distintos bloques.

De la misma manera, para la configuración del *Header Analyzer* es necesario crear una librería la cual a base de *set* y *get* se asigna el valor o se obtiene el valor de los distintos campos de la cabecera y la dirección del bloque *Header Analyzer*. Cada función tendrá un *offset* al que se le suma a la dirección del bloque. Este *offset* es el que se mostró en la Tabla 4: Registros de configuración.

En el Código 20 se muestra la inicialización de los bloques TEMAC y la configuración del bloque *Header Analyzer* haciendo uso de la librería creada. Todo esto se encuentra contenido en el archivo *main* de la aplicación.

```
1  int main(){
2  init_platform();
3        lwip_init();
4        unsigned char mac_ethernet_address[] =
               { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };
5
6  //-------- Asignar dirección MAC a los bloques TEMAC -------//
7        EthFMC_init_axiemac(XPAR_AXIETHERNET_0_BASEADDR,
   mac_ethernet_address);
8        EthFMC_init_axiemac(XPAR_AXIETHERNET_1_BASEADDR,
   mac_ethernet_address);
9
10 //------------ Inspección capa de enlace --------------------//
11
```

```
12  // Activar comprobación de la capa de enlace para ambos bloques
13  set_check_ethernet_layer(XPAR_HEADERANALYZER_TFG_HEADER_ANALYZER
    _0_BASEADDR,1);
14
15  set_check_ethernet_layer(XPAR_HEADERANALYZER1_TFG_HEADER_ANALYZE
    R_0_BASEADDR,1);
16
17  // Asignar dirección MAC de destino FC:AA:14:74:61:E0
18  set_destination_mac(XPAR_HEADERANALYZER_TFG_HEADER_ANALYZER_0_BA
    SEADDR, (unsigned long long) 277807417811424);
19
20  set_destination_mac(XPAR_HEADERANALYZER1_TFG_HEADER_ANALYZER_0_B
    ASEADDR, (unsigned long long) 277807417811424);
21  ...
22  }
```

Código 20: Programa principal de la aplicación

Se ha realizado la inicialización de los bloques TEMAC comprobando que la aplicación *software* es correcta. Se muestra por el Minicom la existencia de negociación entre el controlador PHY y el controlador MAC (Figura 81).

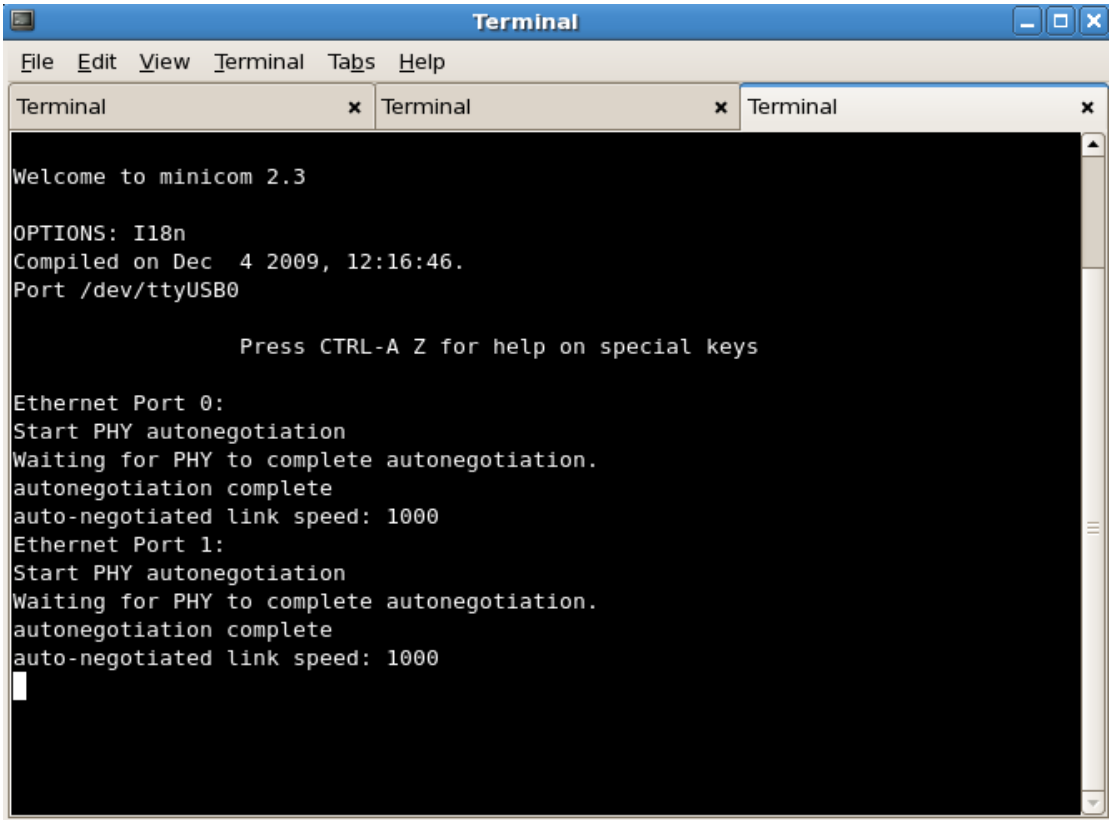


Figura 81: Inicialización de la plataforma incluyendo las interfaces TEMAC

La carpeta con los distintos ficheros que conforman la aplicación final del sistema DPI se muestra en la Figura 82 donde se aprecian las librerías del bloque *Header Analyzer, Unidad de Despacho* y del bloque Acelerador, además de los ficheros para realizar la inicialización de la plataforma y del bloque TEMAC.

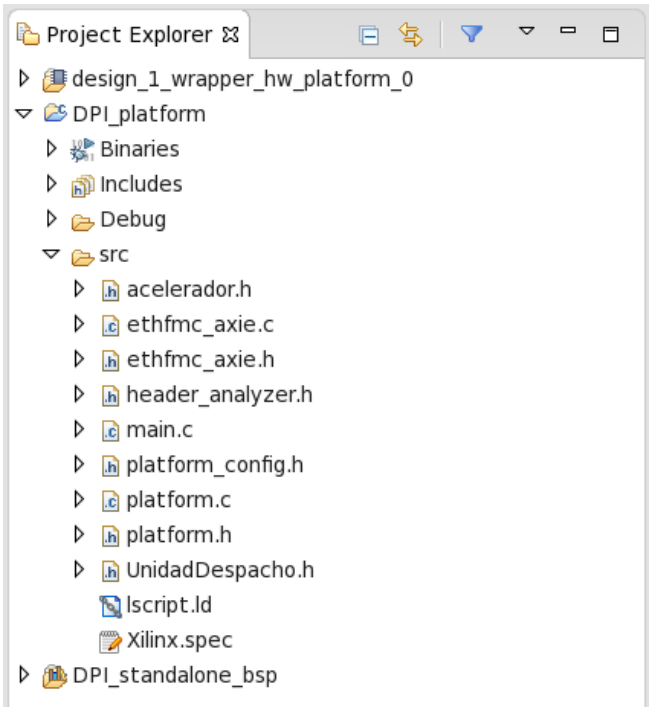


Figura 82: Ficheros de la aplicación

Una vez diseñada la aplicación, el flujo de funcionamiento de la misma es como viene reflejado en la Figura 83.

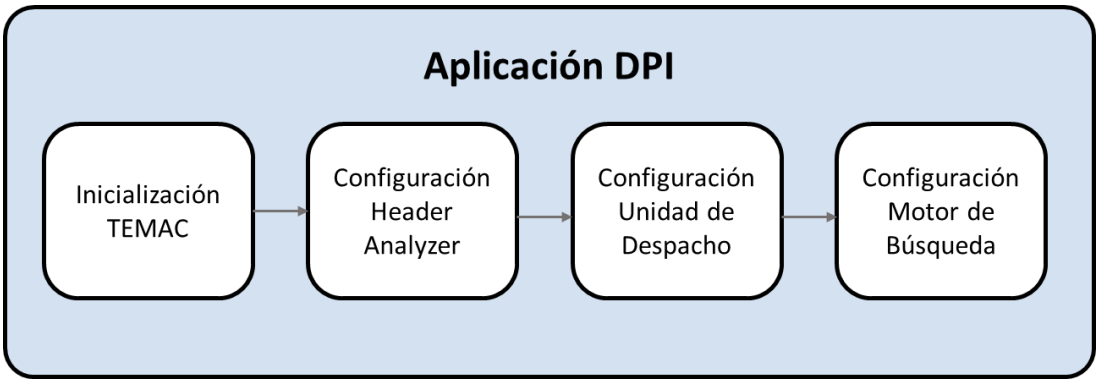


Figura 83: Flujo de funcionamiento de la aplicación

7.3 Conclusiones

Para realizar la configuración de los distintos bloques de la plataforma e inicializar los bloques TEMAC es necesario la creación de una aplicación *software.* Durante este capítulo se ha abordado la creación de este *software* empotrado, desde importar la plataforma *hardware*, pasando por la creación del BSP y terminando por el código necesario de esta aplicación para configurar los distintos campos de filtrado de la cabecera.

Capítulo 8. Validación de la plataforma

8.1 Introducción

Durante este capítulo se mostrará cómo se ha realizado el proceso de validación de la plataforma, así como los resultados que se han obtenido durante este proceso.

Una vez que se ha realizado la integración de la plataforma y se ha desarrollado la aplicación *software*, se realiza la programación del dispositivo a través del *bitstream* que se ha generado de la plataforma y se lanza la aplicación.

Gracias a los ILAs que se han incorporado en la plataforma durante el proceso de montaje se puede realizar una validación minuciosa. Además, se hará uso del Minicom que mostrará los mensajes de la aplicación permitiendo un mejor depurado y mostrando distintos resultados (Figura 84).

Se ha realizado modificaciones en la plataforma para realizar la validación enviando los paquetes directamente desde el PS con el fin de simplificar la validación de la plataforma. Para llevar a cabo esta validación, se han eliminado los bloques TEMAC de la plataforma, minimizando de esta manera el tamaño de la misma y descartando problemas derivados de la captura de datos a través de la red.

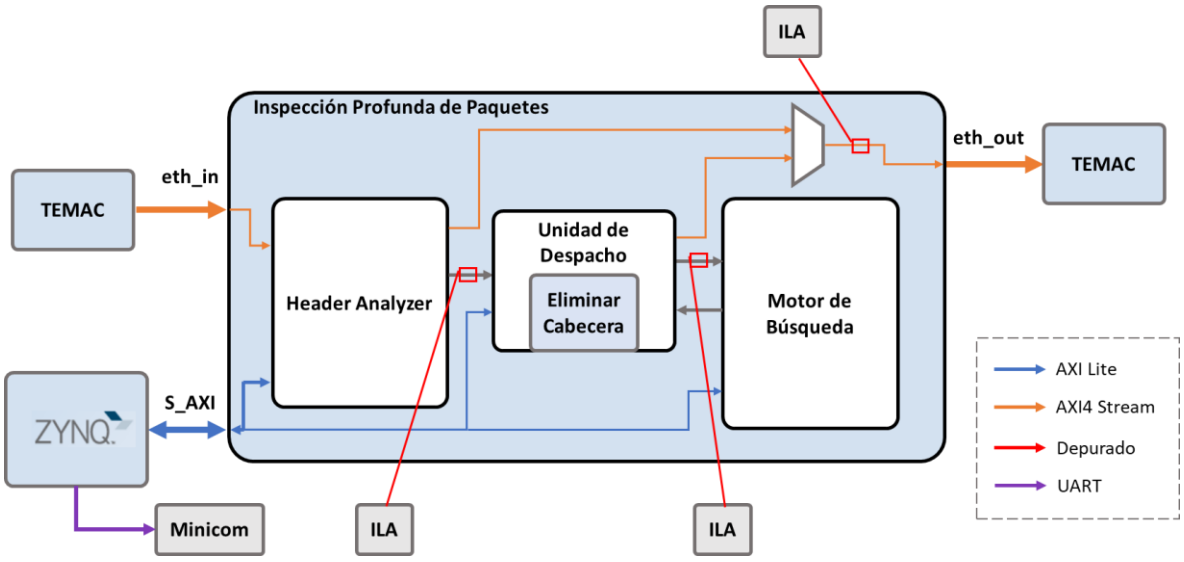


Figura 84: Diagrama de validación

8.2 Validación a través del PS

Para realizar la validación mediante el envío de paquetes desde el PS, se ha tomado la plataforma como unidireccional, eliminando la réplica de todos sus bloques que permiten la bidireccionalidad del sistema. Además, se han eliminado los bloques TEMAC, cortando toda conexión al exterior del dispositivo. Esto agiliza la validación, acotando el posible error únicamente a los bloques en desarrollo, debido a que los bloques TEMAC son bloques conocidos.

Al realizar esto, los paquetes ya no se capturarán a través de la red, por lo que es necesario disponer de una fuente que entregue paquetes a la plataforma con el fin de poder realizar la validación de ésta. Es por ello por lo que se ha optado por enviar los paquetes a través del PS. Con este fin, se guardan en la memoria del PS los distintos paquetes y se envían al PL. Para realizar esta acción es necesario la incorporación un bloque DMA en la plataforma que sea el encargado de manejar los paquetes desde el PS al PL. La plataforma quedará como se muestra en el diagrama de la Figura 85.

Con este cambio en la plataforma también es necesario realizar cambios en la aplicación *software*, tanto para realizar el control del DMA como para la incorporación de los paquetes. Estos cambios se basan en que la aplicación debe realizar las siguientes tareas:

1. Configurar los distintos bloques IP de la plataforma

2. Inicializar del DMA
3. Inicializar el *buffer* del DMA para las transmisiones
4. Enviar el paquete a través del DMA

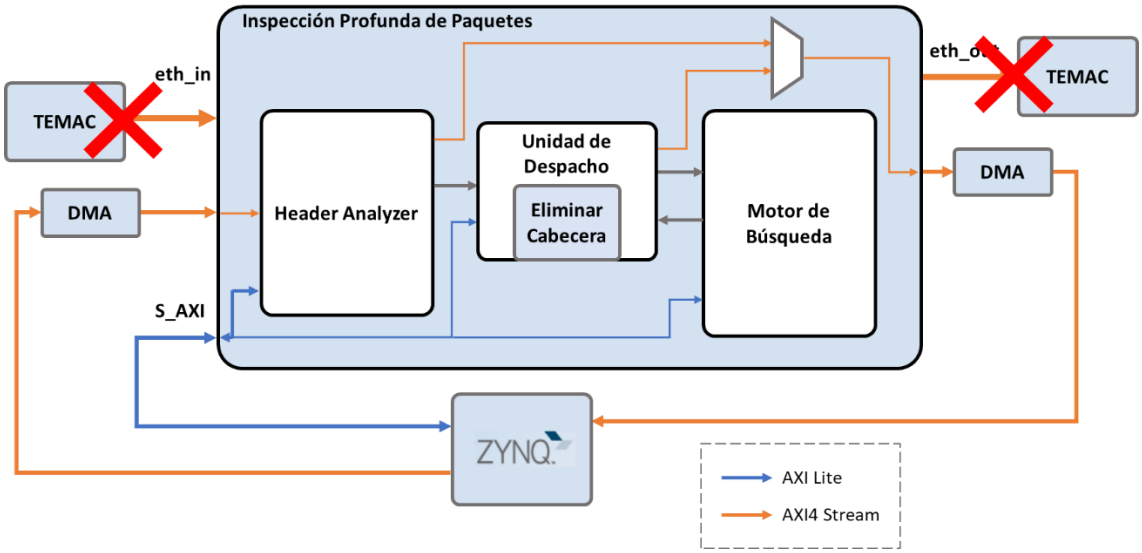


Figura 85: Diagrama de la plataforma de validación enviando paquetes desde el PS

El programa principal de la aplicación se muestra en el Código 21, donde en la línea 13 se realiza la llamada a la función que realiza la inicialización y la transferencia del paquete a través del DMA. Esta función se muestra con más detalle en el Código 22.

```
1  int main(){
2
3  //Inicializar la plataforma
4  init_platform();
5
6  //Configuración del bloque IP Header Analyzer
7  set_check_ethernet_layer(XPAR_TFG_HEADER_ANALYZER_BASEADDR,0);
8  set_check_ethernet_layer(XPAR_TFG_HEADER_ANALYZER_BASEADDR,1);
9  //Asignar dirección MAC destino FC:AA:14:74:61:E0
10 set_destination_mac(XPAR_TFG_HEADER_ANALYZER_BASEADDR, (unsigned
   long long) 277807417811424);
11
12 //Inicializar del DMA y transferencia
13 XAxiDma_polling(0);
14
15 //Limpiar la plataforma
16 cleanup_platform();
```

```
17  return 0;
18  }
```

Código 21: Programa principal de la aplicación con DMA

```
1   int XAxiDma_polling(u16 DeviceId){
2
3         XAxiDma_Config *CfgPtr;
4         int Status;
5         int Index;
6         u32 *TxBufferPtr;
7         u32 *RxPacket;
8         u32 *RxBufferPtr;
9         u32 packet[400];
10
11        TxBufferPtr = (u32 *)TX_BUFFER_BASE ;
12        RxBufferPtr = (u32 *)RX_BUFFER_BASE;
13
14        // Inicializar el DMA.
15        init_DMA();
16
17        //Deshabilitar interrupciones, se usa Polling
18        XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
    XAXIDMA_DEVICE_TO_DMA);
19        XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
    XAXIDMA_DMA_TO_DEVICE);
20
21        //Llamada a la función que copia el paquete completo
22        //en el array packet
23        copyPacket(&packet);
24
25              for(Index = 0; Index < 375; Index ++) {
26                  TxBufferPtr[Index] = packet[Index];
27              }
28        //Limpiar buffer de transmisión antes de la transferencia
29        // si la caché de datos está habilitada
30        Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN*4);
31
32        //Recepción de datos a través del DMA
33        Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR)
    RxBufferPtr, MAX_PKT_LEN*4, XAXIDMA_DEVICE_TO_DMA);
34        if (Status != XST_SUCCESS) {
```

```
35              return XST_FAILURE;
36        }
37        //Enviar datos a través del DMA
38        Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR)
    TxBufferPtr,vMAX_PKT_LEN*4, XAXIDMA_DMA_TO_DEVICE);
39
40        if (Status != XST_SUCCESS) {
41              return XST_FAILURE;
42        }
43        while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
              (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
44              wait();
45        }
46        // Invalidar el buffer de destino antes de recibir el dato,
47        //en caso de que la caché esté habilitada
48        Xil_DCacheInvalidateRange((UINTPTR)RxPacket, MAX_PKT_LEN);
49
50        return XST_SUCCESS;
51  }
```

Código 22: Inicialización del DMA y transferencia del paquete

Una vez realizada la creación de la nueva plataforma y la realización de la nueva aplicación, se procede a realizar la validación haciendo uso de los ILAs.

8.2.1 Validación de la funcionalidad *Header Analyzer*

La validación de este bloque se realiza implementando únicamente el bloque de *Header Analyzer* y el DMA para la transmisión de los datos desde el PS. El diagrama de bloques de la plataforma quedará como se muestra en la Figura 86.

Para realizar la configuración del filtrado de la cabecera del paquete es necesario que el usuario indique qué campo va a analizar y cuál es su valor. Esto se realiza a través del microprocesador y es necesario indicar tanto la dirección del registro del campo a analizar cómo su valor. La dirección del registro se obtiene a través de la dirección base del bloque IP y se le suma el *offset* del registro, como se vio en la Tabla 4: Registros de configuración. La dirección base de *Header Analyzer* se encuentra en el fichero `xparameters.h` donde está definida como:

```
#define XPAR_TFG_HEADER_ANALYZER_BASEADDR 0x60000000
```

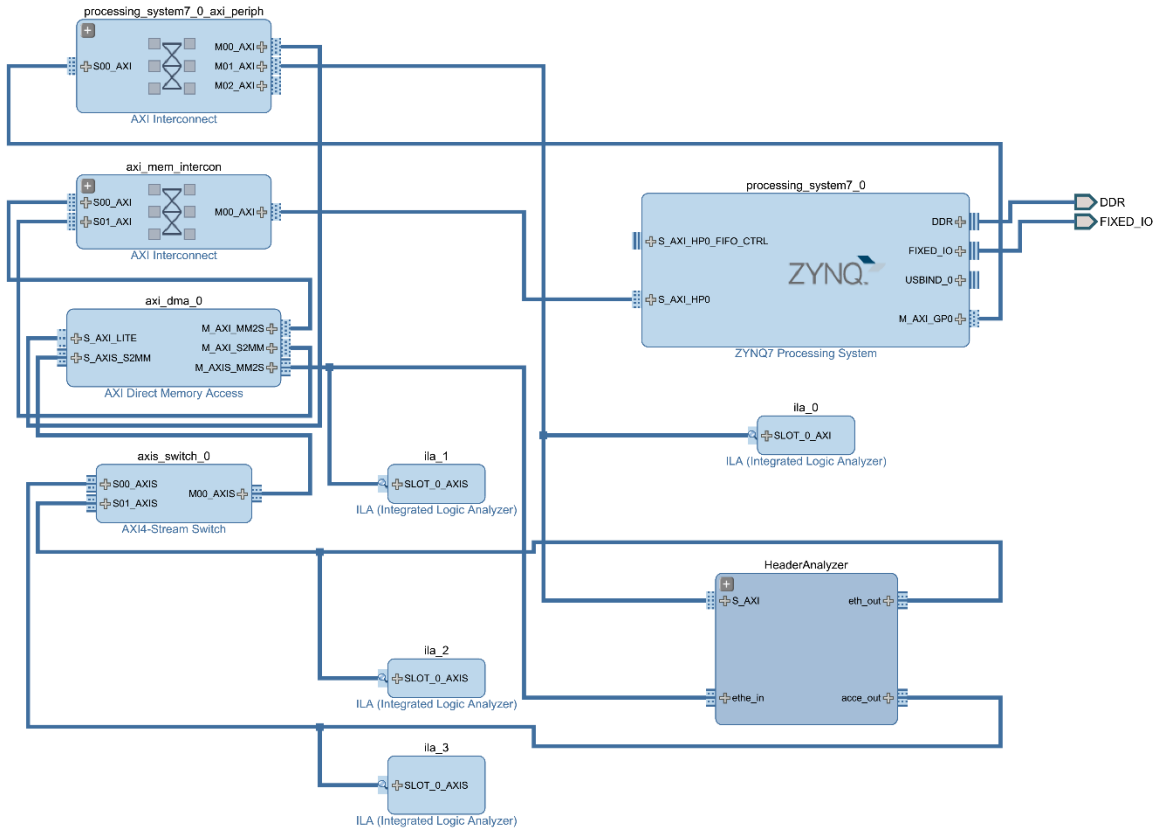


Figura 86: Diagrama de bloques de la validación del *Header Analyzer*

Una vez conocida la dirección base se definen los valores de los campos a través de las funciones contenidas en `header_analyzer.h`. Por ejemplo, en el Código 23, para la asignación de la dirección IP de origen se realizará la llamada a la correspondiente función y previamente se debe haber activado el análisis de la capa de red:

```
1  //Activar análisis de la capa de red
2  set_check_network_layer(XPAR_TFG_HEADER_ANALYZER_BASEADDR, 1);
3
4  //Asignar dirección IP de origen (0x82cec135)
5  set_source_ip(XPAR_TFG_HEADER_ANALYZER_BASEADDR, (unsigned char)
   2194587957);
```

Código 23: Validación *Header Analyzer.* Capa de red y dirección IP de origen

Una vez asignados estos parámetros se puede comprobar a través del ILA (Figura 87) como se ha asignado al canal AWADDR el valor de la dirección base del bloque más el *offset* siendo su valor final de 0x60001044 y en el canal WDATA se muestra el valor de la dirección IP.

Figura 87: Validación de la configuración del *Header Analyzer*

Una vez comprobado que existe comunicación a través de la interfaz AXI4-Lite, se comprobará que el bloque realiza el filtrado correctamente. Para ello, en el Código 24, se le asigna la dirección fuente MAC que coincida con la del paquete enviado y, por lo tanto, el paquete debe salir por la interfaz *acc_out* del bloque hacia el acelerador. Para ello se hace uso de un ILA en la interfaz AXI4-Lite para verificar la correcta asignación de la configuración y otro ILA a la salida del bloque Figura 88.

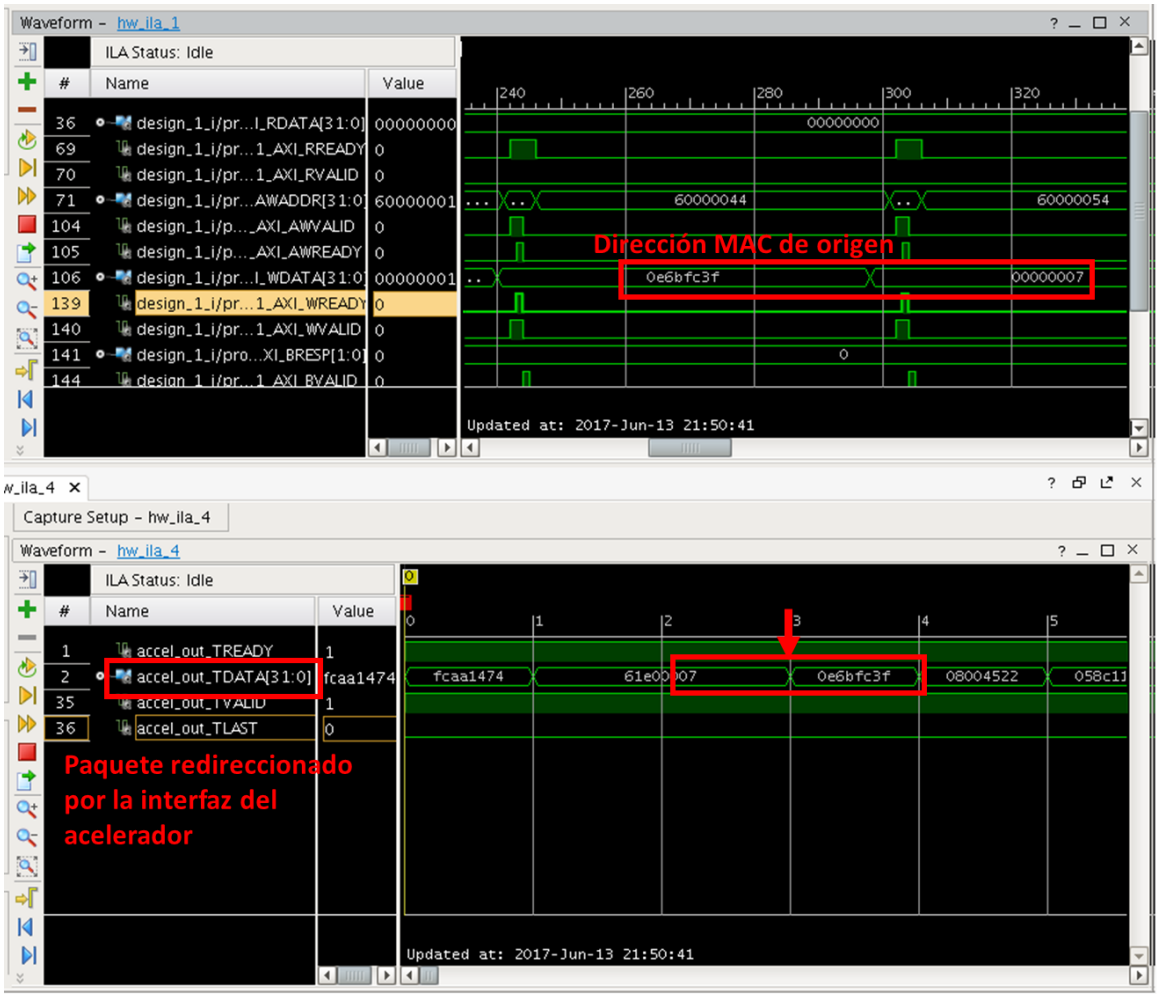


Figura 88: Validación del filtrado. Paquete redireccionado por acelerador

```
1  //Activar análisis de la capa de enlace
2  set_check_ethernet_layer(XPAR_TFG_HEADER_ANALYZER_BASEADDR, 1);
3
4  //Asignar dirección MAC de origen (00:07:0E:6B:FC:3F)
5  set_source_mac(XPAR_TFG_HEADER_ANALYZER_BASEADDR, (unsigned long
   long) 30306729023);
```

Código 24: Validación *Header Analyzer.* Capa de enlace y dirección MAC de origen

Ahora se forzará en el Código 25 a que realice el filtrado de una dirección MAC de destino que no coincide con la del paquete, dando como resultado el redireccionamiento del paquete a través de la interfaz de red de salida. De nuevo, se hace uso de un ILA sobre la interfaz AXI4-Lite y otro sobre la interfaz de salida a la red Figura 89.

```
1  //Activar análisis de la capa de enlace
2  set_check_ethernet_layer(XPAR_TFG_HEADER_ANALYZER_BASEADDR, 1);
3
4  //Asignar dirección MAC de destino (FC:BB:14:74:61:e0)
5  set_destination_mac(XPAR_TFG_HEADER_ANALYZER_BASEADDR, (unsigned
   long long) 277880432255456);
```

Código 25: Validación *Header Analyzer.* Capa de enlace y dirección MAC de destino

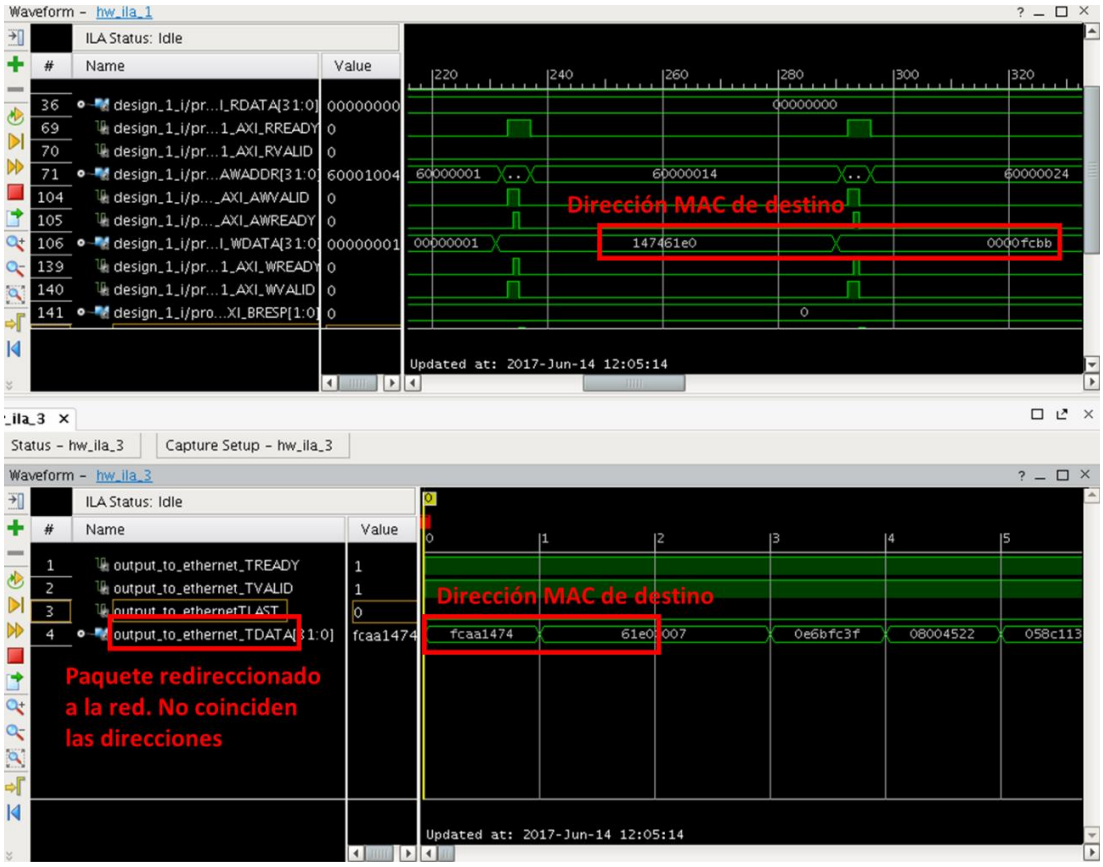


Figura 89: Validación del filtrado. Paquete redireccionado por la red

8.2.2 Validación de la funcionalidad *Eliminar Cabecera*

Una vez que se han comprobado que el bloque *Header Analyzer* funciona correctamente y que realiza el filtrado de todos los campos de la cabecera, se pasará a validar el bloque *Eliminar Cabecera*.

Para simplificar la validación se ha modificado las interfaces de entrada de este bloque, cambiando la interfaz FIFO de entrada por una AXI4-Stream y así poder conectarlo directamente. Esto simplifica el diseño de la plataforma, debido a que no es necesario, por ahora, incorporar el bloque de *Unidad de Despacho* y los bloques aceleradores.

Al diagrama de bloques que se vio en la Figura 86 se le incorpora a la interfaz de salida hacia el acelerador del bloque *Header Analyzer* el bloque *Eliminar Cabecera*, quedando el diagrama de bloques resultante de la plataforma como se muestra en la Figura 90.

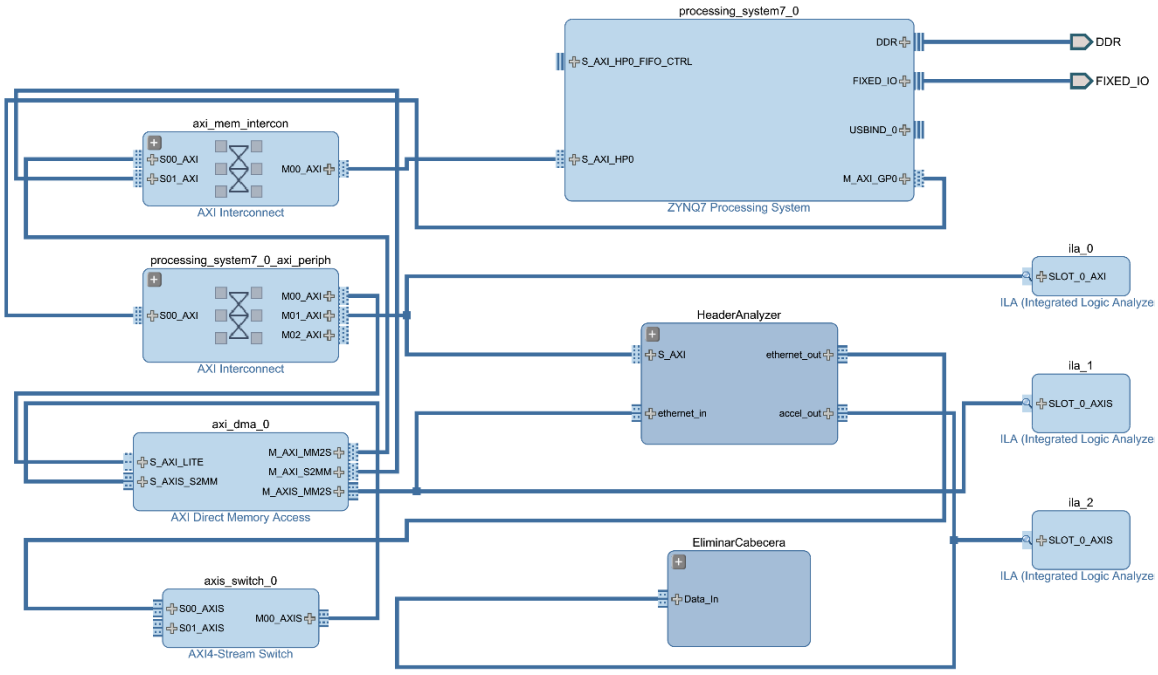


Figura 90: Diagrama de bloques para la validación de *Eliminar Cabecera*

Por tanto, incorporando el bloque de *Eliminar Cabecera* a la interfaz de salida *acc_out* de *Header Analyzer,* se puede validar que el bloque elimina correctamente las 13 primeras palabras del paquete correspondiente a la cabecera. En la Figura 91 se observa este resultado, como existen datos en la interfaz de salida del acelerador y a la salida del bloque *Eliminar cabecera* en el instante cero sale la palabra 14.

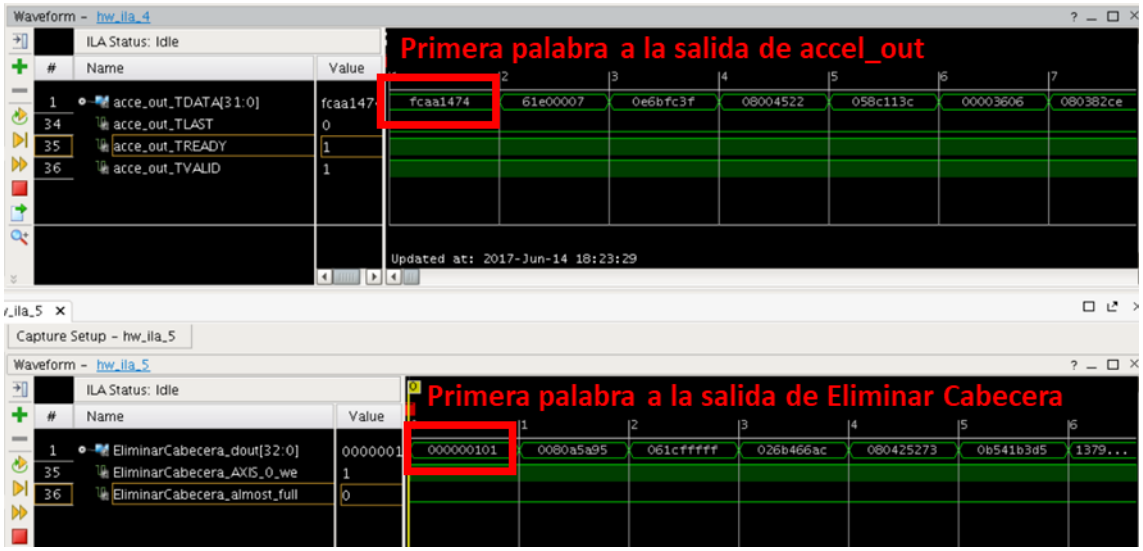


Figura 91: Validación de *Eliminar Cabecera* 32 bits

Sabiendo que el bloque de *Eliminar Cabecera* funciona correctamente se procede a incorporar los demás bloques que conforman la plataforma, *Unidad de Despacho* y bloques aceleradores. Se incorporará el bloque de *Eliminar Cabecera* con la interfaz de entrada FIFO a la *Unidad de Despacho* y se comprobará que, sabiendo que la *Unidad de Despacho* trata palabras de 64 bits, se deben eliminar las seis primeras palabras como se muestra en la Figura 92.

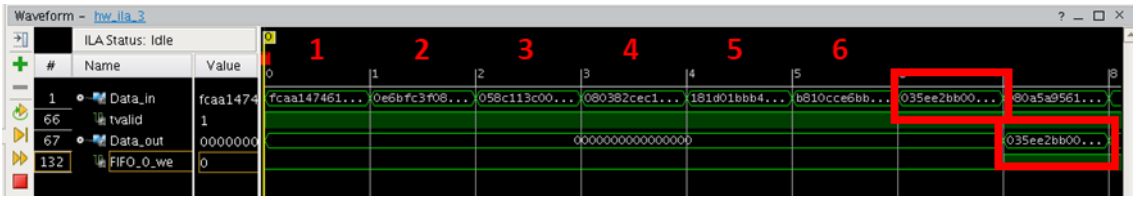


Figura 92: Validación de *Eliminar Cabecera* 64 bits

8.2.3 Análisis de latencias

Gracias a la incorporación de los ILAs a la plataforma no solo se puede observar el flujo de datos en ese punto, sino que también se pueden tomar medidas de latencias del sistema. El diagrama que muestra el ILA tiene una referencia temporal en la parte superior medida en ciclos de reloj. Conociendo la señal de reloj que se ha impuesto durante el diseño de la plataforma al incorporar el bloque PS se puede obtener el tiempo de cada ciclo; es decir, la plataforma trabaja a 200 MHz, por lo que cada ciclo corresponde a 5 ns.

Dependiendo del tipo de filtrado impuesto al bloque *Header Analyzer*, este bloque tendrá una latencia u otra. Es decir, para el filtrado de la capa de enlace se realizará en menor tiempo que el filtrado de la capa de red. En cambio, si se combina ambos tipos de filtrado tendrá otras latencias. Esto se puede observar en las distintas medidas temporales que se han tomado de la plataforma, desde que la primera palabra es capturada hasta que la misma sale a través de la interfaz correspondiente.

En la Figura 93 se muestra la latencia que posee el bloque *Header Analyzer* al realizar el filtrado únicamente de la capa de enlace. Se aprecia que desde que se recibe la primera palabra del paquete hasta que el primer dato es volcado a través de la interfaz de salida dirección al acelerador transcurren 26 ciclos de reloj, es decir, en 0,13 µs.

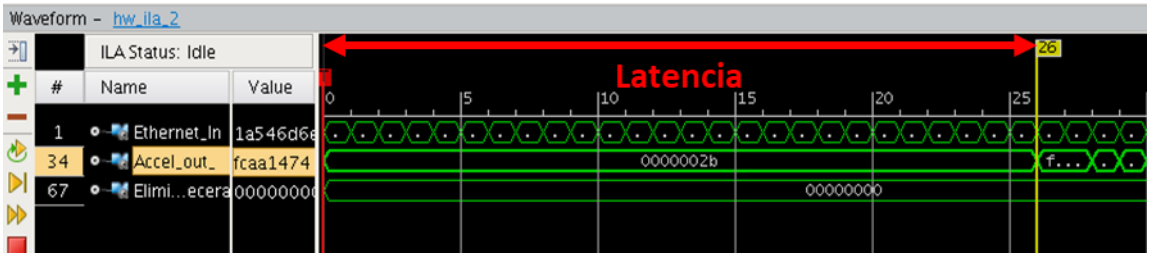


Figura 93: Validación *Header Analyzer.* Latencia filtrado capa de enlace

En cambio, al realizar el filtrado de la capa de red, el bloque *Header Analyzer* realiza esta operación en 28 ciclos, es decir, en 0,14 µs, como se muestra en Figura 94:

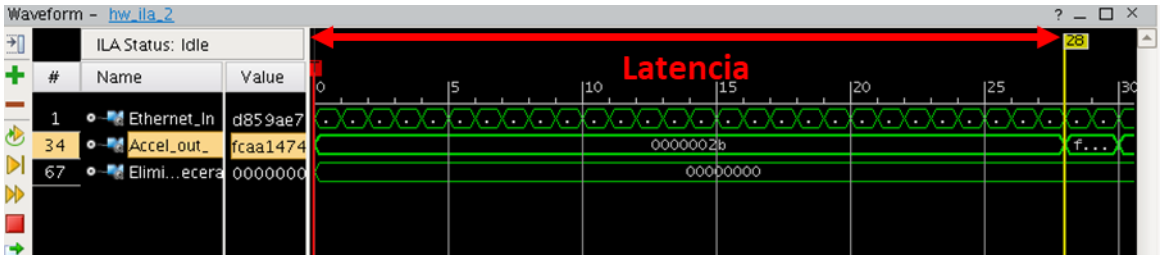


Figura 94: Validación *Header Analyzer*. Latencia filtrado capa de red

Al realizar el filtrado de ambas capas, la latencia del bloque aumenta a 33 ciclos de reloj, o lo que es lo mismo, a 0,165 µs. Esto queda reflejado en la Figura 95:

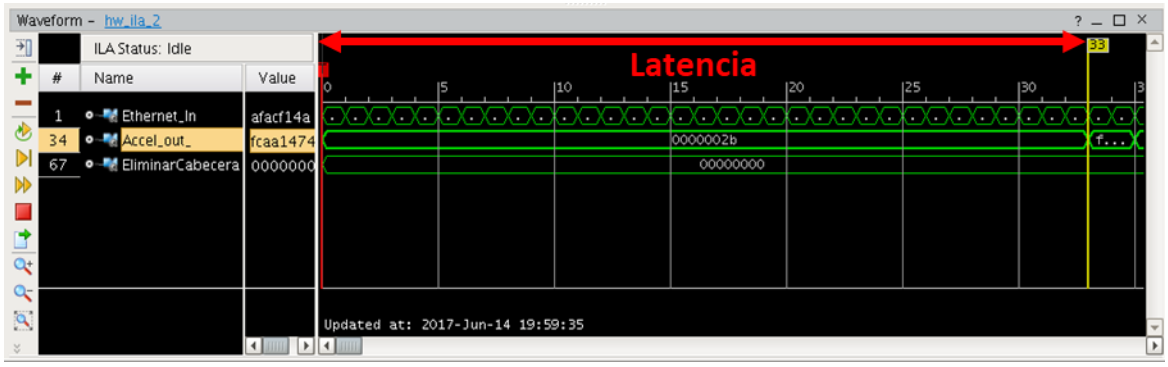


Figura 95: Validación *Header Analyzer*. Latencia filtrado capa de enlace y de red

También se ha realizado la medición temporal del bloque *Eliminar Cabecera* con el objetivo de conocer la latencia que tiene este bloque desde que se recibe el primer dato hasta que expulsa el paquete una vez ha realizado la operación de quitar la cabecera del paquete Ethernet.

Para el bloque *Eliminar Cabecera* con interfaces AXI4-Stream al que se ha conectado directamente el bloque *Header Analyzer* y trata palabras de 32 bits se muestra en la Figura 96. En ella se observa como en el ciclo 33 se recibe el primer dato y es en el ciclo 47 cuando el bloque saca el primer dato del paquete. Es decir, este bloque posee una latencia de 14 ciclos o lo que es lo mismo, 70 ns.

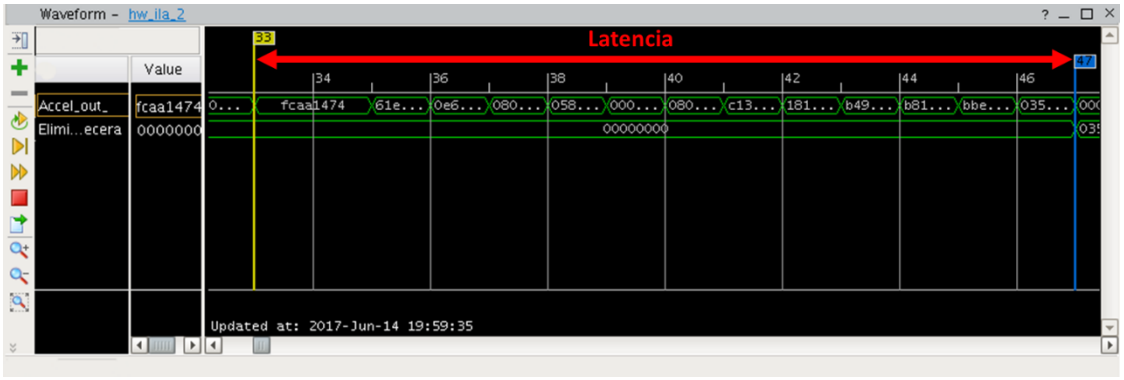


Figura 96: Validación *Eliminar Cabecera*. Latencia de *Eliminar Cabecera* 32 bits

Para el bloque *Eliminar Cabecera* que se sitúa entre la FIFO de la *Unidad de Despacho* y trata palabras de 64 bits se muestra la latencia calculada en la Figura 97. A diferencia del bloque de 32 bits, al aumentar el ancho de palabra se reduce la latencia, realizando la acción de eliminar la cabecera en 7 ciclos, es decir, 35 ns.

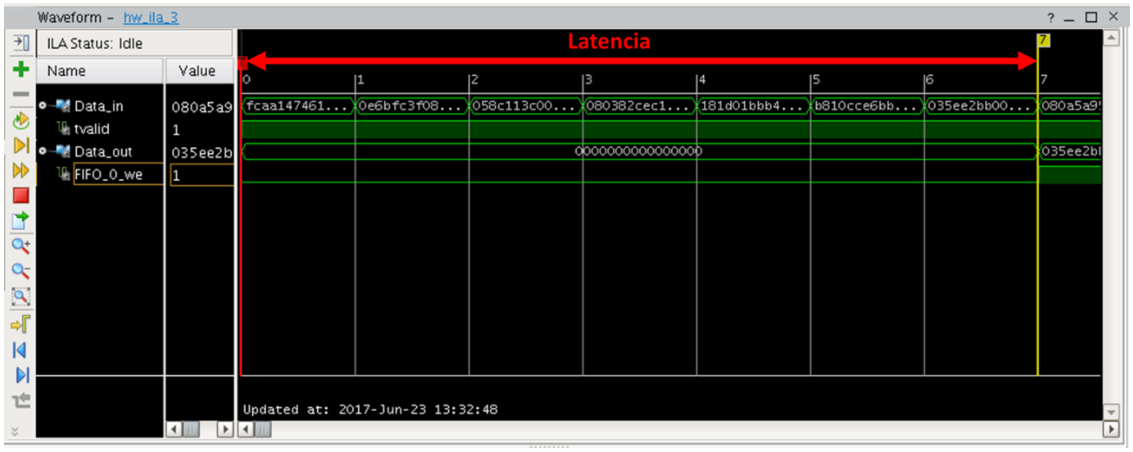


Figura 97: Validación *Eliminar Cabecera*. Latencia de *Eliminar Cabecera* 64 bits

8.2.4 Factor de utilización

Para realizar el análisis del área ocupada por el diseño y de su complejidad, se debe observar el número de *Slices* utilizados. Sabiendo que el *Slice* es la unidad básica de programabilidad de las FPGAs de la familia Zynq, en su arquitectura cada *Slice* está formado por cuatro LUTs y ocho Flip-Flops. El factor de utilización (FU) de los *Slice* se puede calcular sabiendo el número de LUTs y FF usados.

Tomando los datos de la Tabla 10, donde se muestra la cantidad de recursos que utiliza la plataforma implementada, se obtiene el factor de utilización:

$$FU_{LUTs} = \frac{N_{LUTs}}{N_{Slices}} = \frac{11.507}{6.031} = 1{,}90$$

$$FU_{FFs} = \frac{N_{FFs}}{N_{Slices}} = \frac{6.187}{6.031} = 1{,}02$$

Los factores de utilización indicados dependen de la naturaleza del diseño y del estilo de diseño utilizado. Como se desprende de los datos obtenidos, el factor de utilización de las LUTs se encuentra casi a la mitad del valor ideal (valor igual a 4). Esto implica que parte de la lógica del diseño requiere ser tratada de manera combinacional. Por otra manera, el uso de Flip-Flops muestra la lógica secuencial del sistema, en este caso se encuentra bastante alejado al valor ideal (valor igual a 8). Para poder incrementar la utilización de los S*lices* es necesario que compartan las señales de control. La utilización de *resets* globales puede aumentar del orden del 10% el factor de utilización, aunque disminuye el control sobre el diseño durante su simulación. Este proceso de empaquetado

de señales de control está influenciado por las restricciones temporales impuestas al diseño.

8.3 Conclusiones

En este capítulo se han mostrado los distintos resultados que se han obtenido durante la validación de la plataforma quedando demostrada la funcionalidad de la misma.

La validación de la plataforma se ha realizado almacenando los paquetes en la memoria del sistema. Con la incorporación de un DMA se consigue enviar estos paquetes al bloque *Header Analyzer* con el fin de comprobar el correcto funcionamiento de este bloque y del bloque *Eliminar Cabecera*.

Se ha realizado la validación funcional de ambos bloques de manera independiente y una vez comprobado su correcto funcionamiento se ha procedido a validar de manera conjunta, concluyendo que la parte de captura y filtrado de paquetes es funcional.

Capítulo 9. Conclusiones y trabajos futuros

9.1 Introducción

En este capítulo final se presentan las conclusiones a las que se ha llegado tras la realización de este Trabajo Fin de Grado y los trabajos futuros que tiene el mismo.

9.2 Conclusiones del proyecto

La motivación de este Trabajo de Fin de Grado es la realización de un sistema de Inspección Profunda de Paquetes (DPI) cuya arquitectura sea completamente implementada en *hardware,* con objeto de conseguir los objetivos de procesamiento para altas tasas de tráfico Gigabit, manteniendo la flexibilidad aportada por las soluciones *software* en cuanto a su configuración y gestión global del sistema.

Para la realización de esta arquitectura se ha integrado un bloque IP que permite capturar, filtrar y redireccionar un paquete Ethernet en función de determinadas reglas introducidas por el usuario en cuanto al tráfico a nivel de red. El filtrado se realiza comparando los distintos campos de la cabecera del paquete con valores predeterminados por el usuario. El redireccionado de este paquete se realiza a través de una interfaz de salida de red o a un sistema de inspección profunda de paquetes. En el caso de que la regla se cumpla, el paquete debe ser tratado para eliminar su cabecera antes de enviar su *payload* hacia el *Motor de Búsqueda*, ya que este se encarga de analizar únicamente la carga útil o *payload*. Para ello se ha creado un bloque IP que realiza esta acción de manera eficiente y rápida utilizando metodologías de diseño de alto nivel a partir de modelos

algorítmicos, simplificando el proceso de diseño. Por lo tanto, con estos bloques se consigue un sistema DPI que trabaja a alto rendimiento y de manera eficiente, requisitos indispensables para alcanzar las prestaciones requeridas.

Como se ha indicado, los bloques IP han sido modelados en alto nivel, permitiendo un mayor nivel de abstracción que otros lenguajes simplificando el modelado. Este modelado se ha realizado en el lenguaje SystemC, que permite realizar diseños con información temporal al permitir planificar de forma interactiva los ciclos que ocupa un proceso. Posteriormente, se ha realizado la síntesis de alto nivel del diseño a través de la herramienta Cadence Stratus, que permite tomar decisiones en cuanto a recursos e implementar distintas estrategias de síntesis para la definición de su microarquitectura. El objetivo fijado es disminuir la latencia de procesamiento, minimizando los recursos para dejar espacio disponible para implementar los *Motores de Búsqueda* que requieren mayor complejidad.

Se ha conseguido desarrollar un bloque IP que funciona a una frecuencia de 447 MHz y realiza el filtrado de las cabeceras de manera *hardware*, mejorando la frecuencia de sistemas basados en filtrados desde *software,* que tienen su frecuencia limitada por los tiempos de accesos a la memoria. Además, se ha diseñado el bloque *Eliminar Cabecera* que funciona a una frecuencia de 504 MHz. El consumo de potencia de estos sistemas DPI basado en FPGA se ha minimizado, donde la mayor parte de este consumo es realizado por el microprocesador integrado, siendo un 80% del consumo total. En cuanto a ocupación, se ha podido comprobar a través de los informes que proporciona la herramienta que los bloques implementados no alcanzan el 1% de la utilización del dispositivo, siendo la ocupación de la plataforma al completo de un 5% de las LUTs que cuenta el dispositivo. Con esto se demuestra que el uso de FPGAs es una opción idónea tanto para optimización temporal, como de recursos y de potencia.

Durante el proceso de diseño ha sido preciso abordar un flujo de desarrollo complejo que incluye tanto el diseño *hardware* como el desarrollo de aplicaciones *software* empotradas. En este sentido se ha desarrollado una aplicación en C/C++ que configura la plataforma (direcciones a analizar, direcciones IP de los Bloques TEMAC, etc.). La aplicación utiliza un conjunto de funciones que gestionan directamente el *hardware* a partir de la librería lwIP y de funciones generadas en el BSP del sistema. La aproximación utilizada

como *bare-metal* facilita la obtención de resultados en tiempo real al eliminar los *overheads* introducidos por el sistema operativo

9.3 Trabajos futuros

En base a la experiencia conseguida con este proyecto, se proponen diversos trabajos futuros que mejorarán las prestaciones y funcionalidad del sistema y que por razones de tiempo no se han podido abordar:

1. El desarrollo de un sistema con la complejidad implícita de este tipo de sistemas requiere que sea abordado por partes. La plataforma inicial realiza la captura del paquete en el PS, ocasionando problemas de prestaciones. El sistema ha evolucionado introduciendo un bloque de análisis y clasificación de paquetes en *hardware* que elimina dicho cuello de botella. Además, se le ha dotado de un conjunto de buffers y funcionalidad que permite direccionar los paquetes a los motores de análisis o directamente a la salida. En este proyecto no se ha abordado el proceso integrar diferentes bloques para gestionar el flujo de paquetes. Sin embargo, por razones de complejidad asociada, se plantea la integración del *Motor de Búsqueda* para otro proyecto, ya en marcha.

2. Como consecuencia de lo indicado anteriormente, será preciso disponer de la plataforma completa para hacer análisis completos de prestaciones de la misma obtenidos a partir de tráficos de red más realistas. Con ello se permitirá ejercitar los bloques TEMAC de la plataforma, haciendo uso de una configuración más compleja, en la que el analizador DPI se integra como un bloque en medio del flujo de datos.

3. Con objeto de realizar un análisis forense de los paquetes recibidos y desechados por la plataforma al cumplir alguna de las reglas, se propone como trabajo futuro el almacenar en memoria los paquetes que son desechados por el bloque acelerador que realiza el análisis del *payload* del paquete, para ser inspeccionado de manera *offline*. Con ello se consigue tener información adicional sobre el paquete rechazado y mejorar las reglas de configuración del sistema.

4. En el estado actual del proyecto, la configuración se realiza mediante una interfaz de líneas de comandos (CLI). Se propone crear una interfaz gráfica de usuario (GUI) en modo cliente/servidor, por ejemplo, mediante una interfaz web, que sea capaz de realizar la configuración. Otra propuesta es la realización de un formulario web en el que a través de él se pueda realizar la configuración de los campos de análisis de manera más sencilla y cómoda para el usuario, así como disponer de estadísticas de uso del sistema.

5. Por último, una vez se ha obtenido una experiencia sobre la plataforma FPGA implementadas se propone la creación de una plataforma virtual sobre la que se pueda realizar un estudio completo y detallado de tráficos en el sistema para mejorar su arquitectura de comunicaciones, simplificando donde sea posible el sistema para eliminar latencias innecesarias en el mismo.

Bibliografía

[1] Cisco, «The Zettabyte Era: Trends and Analysis», 2016. [Online]. Disponible en: www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf. [Accedido: 02-feb-2017]

[2] S. Li, L. Da Xu, y S. Zhao, «The internet of things: a survey», *Inf. Syst. Front.*, vol. 17, n.º 2, pp. 243-259, 2015 [Online]. Disponible en: http://dx.doi.org/10.1007/s10796-014-9492-7

[3] Swati Khandelwal, «World's largest 1 Tbps DDoS Attack launched from 152,000 hacked Smart Devices», *The Hacker News*, 2016. [Online]. Disponible en: http://thehackernews.com/2016/09/ddos-attack-iot.html. [Accedido: 03-feb-2017]

[4] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, y M. Rajarajan, «A survey of intrusion detection techniques in Cloud», *J. Netw. Comput. Appl.*, vol. 36, n.º 1, pp. 42-57, 2013 [Online]. Disponible en: http://dx.doi.org/10.1016/j.jnca.2012.05.003

[5] Open Information Security Foundation (OISF), «Suricata | Open Source IDS / IPS / NSM engine». [Online]. Disponible en: https://suricata-ids.org/. [Accedido: 03-feb-2017]

[6] «The Bro Network Security Monitor», 2014. [Online]. Disponible en: https://www.bro.org/index.html. [Accedido: 02-feb-2017]

[7] M. Mueller, «DPI technology from the standpoint of Internet governance studies.», Syracuse, 2011 [Online]. Disponible en: http://dpi.ischool.syr.edu/Technology_files/WhatisDPI-2.pdf

[8] Intel Corporation, «Qosmos* Deep Packet Inspection Characterization», 2016 [Online]. Disponible en: http://www.qosmos.com/wp-content/uploads/2016/10/qosmos_deep_packet_inspection_characterization.pdf

[9] Intel Corporation, «Highly-Scalable DPI (Pattern Matching) Performance across Intel® Processors using Hyperscan», 2015 [Online]. Disponible en: http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/hyperscan-scalability-solution-brief.pdf

[10] F. Yu, «High Speed Deep Packet Inspection with Hardware Support», University of California at Berkeley, 2006 [Online]. Disponible en: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-156.html

[11] R. S. Boyer y J. S. Moore, «A Fast String Searching Algorithm», *Commun. ACM*, vol. 20, n.º 10, pp. 762-772, 1977 [Online]. Disponible en: http://doi.acm.org/10.1145/359842.359859

[12] C. Xu, S. Chen, J. Su, S. M. Yiu, y L. C. K. Hui, «A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms», *IEEE Communications Surveys and Tutorials*, vol. 18, n.º 4. IEEE Press, pp. 2991-3029, 2016 [Online]. Disponible en: http://ieeexplore.ieee.org/document/7468531/. [Accedido: 22-ene-2017]

[13] Y. Lee, J. Oh, J. K. Lee, D. Kang, y B. G. Lee, «The development of deep packet inspection platform and its application», en *3rd International Conference on Intelligent Computational Systems (ICICS'2013)*, 2013 [Online]. Disponible en: http://psrcentre.org/images/extraimages/2. ICECEBE 113833.pdf

[14] Y.-S. Lin, C.-L. Lee, y Y.-C. Chen, «Length-Bounded Hybrid CPU/GPU Pattern Matching Algorithm for Deep Packet Inspection», *Algorithms*, vol. 10, n.º 1, p. 16, ene. 2017 [Online]. Disponible en: http://www.mdpi.com/1999-4893/10/1/16. [Accedido: 24-ene-2017]

[15] S. Wu, U. Manber, y others, «A fast algorithm for multi-pattern searching», University of Arizona. Department of Computer Science, Tucson, AZ (USA), may 1994 [Online]. Disponible en: http://webglimpse.net/pubs/TR94-17.pdf

[16] A. Domínguez, P. P. Carballo, B. Vega, y A. Núñez, «Dynamic-pattern string matching SoC based on Boyer-Moore algorithm», Submited for publication, 2017.

[17] B. Vega, «Diseño e Implementación mediante Síntesis de Alto Nivel de un IP para el filtrado y clasificación de paquetes TCP/IP», 2016 [Online]. Disponible en: https://en.iuma.ulpgc.es/tfm_tfg/documentos/2015/TFM_BenjaminVegadelPino_Memoria.pdf

[18] J. Zhu y N. Dutt, «Chapter 5 - Electronic system-level design and high-level synthesis», en *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, y K.-T. (Tim) Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 235-297 [Online]. Disponible en: www.sciencedirect.com/science/article/pii/B9780123743640500126

[19] F. Ghenassia y A. Clouard, «TLM: An Overview and Brief History», en *Transaction Level Modeling with SystemC*, F. Ghenassia, Ed. Berlin/Heidelberg: Springer-Verlag, 2005, pp. 1-22 [Online]. Disponible en: http://link.springer.com/10.1007/0-387-26233-4_1. [Accedido: 27-abr-2017]

[20] IEEE, «IEEE Standard for Standard SystemC Language Reference Manual», *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* -. pp. 1-638, 2012 [Online]. Disponible en: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6134619&isnumber=6134618

[21] P. Urard, J. Yi, H. Kwon, y A. Gouraud, «User Needs», en *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy; y A.Morawiec;, Eds. Dordrecht: Springer Netherlands, 2008, pp. 1-12 [Online]. Disponible en: http://dx.doi.org/10.1007/978-1-4020-8588-8_1

[22] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, y R. W. Stewart, *The Zynq Book*, 1st Editio. Glasgow: Strathclyde Academic Media, 2014 [Online]. Disponible en: http://www.zynqbook.com/. [Accedido: 21-feb-2017]

[23] Cadence Design Systems, «Stratus High-Level Synthesis», 2015 [Online]. Disponible en: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf. [Accedido: 27-abr-2017]

[24] Xilinx, «Zynq-7000 EPP», 2011 [Online]. Disponible en: http://www.uio.no/studier/emner/matnat/ifi/INF3430/h11/undervisningsmateriale/roarsk/Xilinx_ZYNQ_Product_Brief.pdf. [Accedido: 01-jul-2017]

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

[26] Xilinx Inc., «AXI Reference Guide», 2011 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[27] ARM, «Cortex-A9 – ARM Developer». [Online]. Disponible en: https://developer.arm.com/products/processors/cortex-a/cortex-a9. [Accedido: 04-jul-2017]

[28] Xilinx Inc., *Zynq-7000 All Programmable SoC ZC706 Evaluation Kit Getting Started Guide*, UG961. 2012 [Online]. Disponible en: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/2015_4/ug961-zc706-GSG.pdf

[29] Xilinx Inc., «AXI4-Stream Infrastructure IP Suite v2.2». p. 77, 2017 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf

[30] Cadence Design Systems, «Stratus High-Level Synthesis User Guide», n.º May. pp. 1-861, 2016.

[31] GTKWave, «GTKWave 3.3 Wave Analyzer User's Guide GTKWave 3.3 Wave Analyzer User's Guide 1». p. 159, 2017 [Online]. Disponible en: http://gtkwave.sourceforge.net/gtkwave.pdf. [Accedido: 01-jun-2017]

[32] Xilinx Inc., «Processing System 7 v5.5». 2017 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf. [Accedido: 07-jun-2017]

[33] Xilinx Inc., «Processor System Reset Module v5.0 LogiCORE IP Product Guide

(PG164)», 2015 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf. [Accedido: 07-jun-2017]

[34] Xilinx Inc., «AXI Interconnect v2.1 LogiCORE IP Product Guide Vivado Design Suite». Xilinx Inc., p. 170, 2017 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf. [Accedido: 08-jun-2017]

[35] Xilinx Inc, «AXI 1G/2.5G Ethernet Subsystem v7.0». p. 166, 2017 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v7_0/pg138-axi-ethernet.pdf. [Accedido: 07-jun-2017]

[36] Opsero Electronic Design, «EthernetFMC - Quad Port Gigabit Ethernet FMC». [Online]. Disponible en: https://opsero.com/product/ethernet-fmc/. [Accedido: 04-abr-2017]

[37] Xilinx Inc., «FIFO Generator v13.1». Xilinx Inc., p. 218, 2017 [Online]. Disponible en: https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_1/pg057-fifo-generator.pdf. [Accedido: 08-jun-2017]

[38] González Crespo, «Desarrollo de una Unidad de Despacho para la Plataforma SoC DPI basada en FPGA Xilinx Zynq», 2016.

[39] Xilinx Inc., «Zynq-7000 All Programmable SoC Software Developers Guide». p. 68, 2015 [Online]. Disponible en: https://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf. [Accedido: 21-jun-2017]

[40] A. Sarangi, S. Macmahon, y U. Cherukupaly, «Xilinx XAPP1026 LightWeight IP (lwIP) Application Examples, v5.1, Application Note», *XAPP1026*, n.º 1. p. 31, 2014 [Online]. Disponible en: www.xilinx.com. [Accedido: 21-jun-2017]

Presupuesto

Para la realización del presupuesto se debe tener en cuenta los recursos utilizados para la realización del Trabajo Fin de Grado. Estos recursos engloban tanto los recursos humanos como los materiales de las diversas etapas que ha tenido el proyecto: modelado, síntesis, integración, verificación y validación.

A continuación, se muestran los costes asociados al proyecto, así como los recursos materiales, que se han divido en recursos *software* y recursos *hardware.*

1. Recursos *hardware*

Se han necesitado diversas herramientas de trabajo para la realización del proyecto. En la Tabla 11 se muestra el coste de estos recursos, que han sido estimados en función a su amortización y el periodo empleado. En la Tabla 12 se muestra el coste de soporte para tener operativos y mantener los recursos *hardware*. En la Tabla 13 se muestra el coste total de los recursos *hardware.*

Tabla 11: Costes de recursos *hardware*

RECURSOS	COSTES DEL EQUIPO	TIEMPO DE UTILIZACIÓN	COSTE DE USO
Zynq-7000 ZC706 Evaluation Kit	2.234,77€	150 horas	206,03 €
***Workstation* para diseño y programación**	1.800,00€	300 horas	368,78 €
Servidores de cómputo	5.000,00 €	300 horas	921,94 €
Cables y equipo auxiliar	40,00€	150horas	40,00 €
TOTAL			1.536,75 €

Tabla 12: Coste de soporte

DESCRIPCIÓN	COSTE
Dos técnicos a tiempo completo	48.080,00 €
Proporción con 100 usuarios año	480,80 €
COSTE TOTAL (4 meses a tiempo parcial)	80,13 €

Tabla 13: Coste total asociados a los recursos *hardware*

DESCRIPCIÓN	COSTE
Coste recursos *hardware*	1.536,75 €
Coste de soporte	80,13 €
COSTE TOTAL	1.616,89 €

2. Recursos *software*

Los recursos *software* engloban las licencias de las distintas herramientas utilizadas en el proyecto (Tabla 14) y su mantenimiento (Tabla 15). Los costes totales de los recursos *software* se han calculado en la Tabla 16.

Tabla 14: Coste de licencias de los recursos *software*

RECURSOS	TIPO DE LICENCIA	COSTE DE LICENCIA	AMORTIZACIÓN DE LICENCIA
Acceso a las herramientas	Universitaria	- €	- €
Cadence Stratus	Universitaria	2.600,00 €	520,00 €
Synopsys Synplify	Universitaria	272,00 €	54,40 €
Xilinx Vivado Design Suite	Universitaria	Donación	- €
Microsoft Office 2016	Universitaria	269,00 €	269,00 €
TOTAL			843,40 €

Tabla 15: Coste de mantenimiento de los recursos *software*

RECURSOS	MANTENIMIENTO ANUAL	DOCENCIA (15%)
Acceso a las herramientas	642,00€	96,30 €
Cadence Stratus	2.628,99€	394,35 €
Synopsys Synplify	1.182,35€	177,35 €
Xilinx Vivado Design Suite	214,00€	32,10 €
Microsoft Office 2016	- €	- €
TOTAL		700,10 €

Tabla 16: Coste total recursos *software*

DESCRIPCIÓN	COSTE
Coste de licencias	843,40 €
Coste de mantenimiento	700,10 €
COSTE TOTAL	1.543,50 €

3. Recursos humanos

En Tabla 17 se ha calculado el coste de contratar un ingeniero de telecomunicaciones[3] para desempeñar las tares de investigador en proyectos según el coste por hora. Se han desglosado los distintos paquetes de trabajo del proyecto.

Tabla 17: Coste de recursos humanos

TAREA	COSTE/HORA	HORAS	DÍAS	COSTE TOTAL
WP1. Estudio de la plataforma de referencia	16,65 €/hora	4	6	399,60 €
WP2. Estudio del bloque de captura de datos	16,65 €/hora	4	6	399,60 €
WP3. Propuesta de nueva arquitectura del sistema	16,65 €/hora	4	11	732,60 €
WP4. Integración e implementación de la arquitectura final	16,65 €/hora	4	30	1.998,00 €
WP5. Validación del diseño	16,65 €/hora	4	5	333,00 €
WP6. Documentación del proyecto	16,65 €/hora	4	13	865,80 €
TOTAL				4.728,60 €

4. Material fungible

El material fungible está formado por el material empleado durante el desarrollo del proyecto. También incluye los discos necesarios para las copias del proyecto. El coste del material fungible es estimado y tiene un valor de 200,00 €.

[3] Según la resolución del BOULPGC del 4 de noviembre de 2010

5. Coste de edición del proyecto

El coste de edición de la memoria de este proyecto, incluyendo todos los gastos derivados de su impresión y encuadernación es de 100,00 €.

6. Costes indirectos

Los costes indirectos asociados al proyecto se calculan sobre un 35% de los costes directos, teniendo un valor de 2.866,15 €

7. Coste total del proyecto

Por último, se realiza un resumen de los costes totales que se han mencionado anteriormente y da como resultado el coste total del proyecto (Tabla 18).

Tabla 18: Coste total del proyecto

RECURSOS	COSTES	
1. Recursos *hardware*	1.616,89 €	
2. Recursos *software*	1.543,50 €	
3. Recursos humanos	4.728,60 €	
4. Material fungible	200,00 €	
5. Coste de edición del proyecto	100,00 €	
6. Costes directos		8.188,99 €
7. Costes indirectos (35%)	2.866,15 €	
Coste final		11.055,14 €
IGIC (7%)	773,86 €	
COSTE DEL PROYECTO		**11.829,00 €**

Dª. Sonia Raquel León Martín declara que el presupuesto del presente proyecto asciende a once mil ochocientos veintinueve euros (11.829,00 €).

Las Palmas de Gran Canaria, a 10 de julio de 2017

Fdo.: Sonia Raquel León Martín

Pliego de condiciones

1. Introducción

Durante del desarrollo de este Trabajo Fin de Grado se han utilizado equipos *hardware* y recursos *software* cuyas características y versiones se detallan a continuación. Asimismo, se hace mención a la garantía.

Los valores obtenidos y los parámetros medidos en este proyecto son válidos para las versiones de los recursos *software*, *hardware* y de edición del proyecto.

2. Recursos *hardware*

- Placa de prototipado Xilinx Zynq-7000 ZC706 *Evaluation Kit*
- Servidores de cómputo basados en arquitectura Intel x86 de 64 bits, bajo sistema operativo Linux RedHat versión 6
- *Workstation* de diseño y programación DELL con sistema operativo Red Hat versión 6

3. Recursos *software*

Herramientas de diseño, simulación, síntesis, implementación y desarrollo de *software* empotrado:

- Herramientas de diseño de alto nivel Cadence Stratus 16.22
- Simulador digital Cadence Incisive + Simvision 15.20

- Visualizador de formas de ondas *open source* GTKWave Analyzer v3.3.40
- Entorno de diseño IDE Vivado Design Suite 2016.4

Terminal serie:

- Minicom 2.3

Lenguajes, formatos y versiones:

- Modelado de sistemas: SystemC 2.3
- Lenguaje de descripción *hardware* RTL: Verilog (versión 2001)
- Formato EDIF: 2 0 0
- Lenguaje C/C++ versión 2011
- Compilador para ARM integrado en Vivado: GNU ARM Embedded Toolchain 5.2.1

4. Recursos de edición y control del proyecto

- Herramienta de edición del proyecto: Microsoft Word Office 2016 sobre Windows 10
- Herramienta de presentación del proyecto: Microsoft PowerPoint 2016
- Herramienta de gestión del proyecto: Microsoft Project 2016

5. Garantía

Este TFG se presenta con una garantía “AS IS” (tal cual), sin garantía implícita de ningún tipo. Tampoco se responsabiliza de los daños que puedan causar el código presentado o sus archivos derivados a cualquier equipo o del uso que hagan de ellos terceras personas.