UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Trabajo de Fin de Grado

Desarrollo de una sonda Ethernet activa basada en el SoC programable Xilinx Zynq

AUTOR: Alfredo Sosa Muñoz
TUTOR: Víctor Moreno Martínez
PONENTE: Dr. Sergio López Buedo

High Performance Compunting and Networking
Dpto. de Ingeniería de Telecomunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2014

Resumen

La necesidad de proveer mejores servicios a los usuarios de Internet obliga a las empresas a tener la capacidad de controlar el estado de sus servicios. Durante mucho tiempo la calidad de servicio que recibe un usuario viene determinada por el retraso en los enlaces de la red, la pérdida de datos, el ancho de banda en los enlaces... y para obtener estos datos se han implementado diversas plataformas que se encargan del estudio de la red. Sin embargo el desarrollo y el uso de estos sistemas es complejo y conlleva gastos económicos elevados.

Para facilitar y abaratar estos sistemas se ha estudiado el uso de un chip programable de bajo coste. El SoC programable de Xilinx Zynq es un dispositivo formado por una FPGA tradicional más un procesador físico. Esta estructura permite el desarrollo de un dispositivo a medida con toda la potencia de un procesador del mercado consiguiendo un precio no muy elevado. Gracias a la arquitectura de este dispositivo, se ha planteado el desarrollo de una sonda de monitorización de red con la capacidad de disponer de un sistema autónomo ejecutando su propio sistema operativo, es decir, no dependerá de un ordenador anfitrión al que conectarse.

Para desarrollar la sonda de monitorización de red se han implementado dos diseños orientados al estudio de los retardos entre paquetes. Uno de los diseños es el encargado de la generación del tráfico con el que tomar las medidas, mientras que el otro diseño es el encargado de redirigir el tráfico por la red de vuelta a su origen, pudiendo utilizar ambos diseños como una conexión punto a punto y tomar el tiempo de ida y vuelta de los paquetes de red. Estos diseños están basados en los componentes lógicos que proporciona Xilinx. Algunos de estos componentes son el Tri-Mode Ethernet MAC, AXI-DMA, Zynq Processing System, buses AXI, etc.

Otro de los aspectos abordados en este documento es el desarrollo de hardware desde interfaces y lenguajes de alto nivel, facilitando el desarrollo y abaratando los costes que supondría el esquema de desarrollo típico sobre FPGAs. Esta tarea se ha realizado mediante el software de Xilinx, Vivado(IP Integrator) y Vivado HLS.

Después de realizar el proyecto se ha conseguido establecer la base para el desarrollo de dispositivos de red basados en Xilinx Zynq, proporcionando un esquema de desarrollo que agiliza la implementación de funcionalidades específicas para el diseño propuesto. Además se ha comprobado que la sonda de monitorización desarrollada se puede integrar fácilmente para su uso en sistemas de red reales.

Palabras Clave: FPGA, CORE, Zynq, ZedBoard, QoS-Poll, AXI4, AXI-Ethernet, AXI-DMA, AXI4-Stream, AXI4-Lite, QoS, PL, PS, bitstream, devicetree, reset, FMC, LogiCORE, HLS, Vivado, Checksum.

Abstract

The need to provide better services to Internet users requires companies to have the ability to monitor the status of their services. For a long time the quality of service received by a user is determinated the delay on the links of the network, data loss, bandwidth on links... and to get these details have implemented several platforms that handle network study. However, the development and use of these systems is complex and involves high economic costs

To facilitate and cheapen these systems has been studied using a low cost programmable chip. Xilinx Zynq programmable SoC is a device consisting of a traditional FPGA more physical processor. This structure allows the development of a device with the power of a market processor not getting a very high price. Thanks to the architecture of this device has been raised the development of network monitoring probe with the ability to have an autonomous system running its own operating system that is not dependent upon a host computer.

To develop the network monitoring probe has been implemented two desing two designs aimed at studying delays between packets. One of the designs is responsible for generating traffic to take measures while the other design is responsible for redirecting traffic through the network back to its origin. This allows both designs use as a point to point connection and take the packet's time to return. These designs are based in logical components provided by Xilinx. Some of these components are the Tri-Mode Ethernet MAC, AXI-DMA, Zynq Processing System, AXI buses, etc.

Another aspect discussed in this paper is the hardware development from high level interfaces and languages, esasing the development and lowering costs involved in typical development scheme on FPGA. This task is performed using the Xilinx software, Vivado(IP Integrator) and Vivado HLS.

After this project has been achieved the basis for the development of network devices based on Xilinx Zynq, providing a development scheme that streamlines the implementation of specific features for the proposed design. Also has been verified that the network monitoring probe can be easily integrated for use in real systems network.

Index Terms: FPGA, CORE, Zynq, ZedBoard, QoS-Poll, AXI4, AXI-Ethernet, AXI-DMA, AXI4-Stream, AXI4-Lite, QoS, PL, PS, bitstream, devicetree, reset, FMC, LogiCORE, HLS, Vivado, Checksum.

Agradecimientos

En primer lugar me gustaría agradecer al grupo de investigación HPCN por facilitarme todos los recursos necesarios para realizar este proyecto, además de toda la ayuda que me han prestado todos los miembros del equipo.
También me gustaría dar las gracias a, Gustavo, Sergio y Víctor por ayudarme a realizar este trabajo en el día a día, soportando lo pesado que puedo llegar a ser y solucionando los momentos de pánico cuando algo no me funcionaba.
No menos importantes son mis compañeros de laboratorio, Daniel, José, Rafael...todos ellos me han acompañado durante todo este último año de carrera, haciendo más entretenido todos esos momentos en los que cuesta trabajar.
Por último agradecer a mi familia todos esos momentos soportando la luz encendida y el ruido de mi teclado hasta altas horas de la madrugada y por apoyarme siempre en las decisiones que he tomado.

Índice

Índice de figuras

Índice de tablas

Glosario

FPGA: Field Programmable Gate Array
ZedBoard: Zynq Evaluation and Development Board
QoS: Quality of Service
ISP: Internet Service Provider
IP: Internet Protocol
PL: Processing Logical
PS: Processing System
HLS: High Level Synthesis
SoC: System-on-Chip
PTP: Precision Time Protocol
ARM: Advanced RISC Machine
TTL: Time To Live
CPD: Centro de Procesamiento de Datos

1.-Introducción

A lo largo de las últimas décadas, las redes y la tecnología se encuentra en un proceso de constante cambio, en el que además los usuarios cada vez más formados y con más experiencias exigen un servicio que ofrezca un cierto nivel de calidad de servicio(QoS). Además aquellas empresas que se encargan de proveer a Internet a los usuarios(ISPs) necesitan estudiar el comportamiento y los servicios que ofrecen las redes que se encuentran bajo su nombre.

Por otro lado el tráfico de Internet varia continuamente y estas empresas necesitan una gran precisión en la captura o clasificación de paquetes y flujos de red y de sus tiempos de referencia. Obviamente todas estas empresas disponen de sus propios mecanismos para realizar estas tareas, sin embargo en la mayoría de los casos estas tareas se realizan a nivel software con algoritmos complejos y con máquinas muy potentes y con un fuerte coste económico.

Lo que se propone con este proyecto es la utilización de sistemas de bajo coste para obtener el mismo nivel de precisión ganando principalmente económicamente y una fiabilidad y precisión equivalente al que se puede obtener con una máquina de alto rendimiento.

A pesar de que el uso de FGPAs es cada vez más habitual para este tipo de tareas, gracias a la potencia de computo y al bajo consumo que ofrecen, suponen que el desarrollo de estos sistemas sea largo y caro, sobre todo debido a que la programación de sistemas en lenguajes descriptivos puede llegar a alcanzar grandes niveles de complejidad. En el caso de este trabajo se ofrece la alternativa del uso del Soc programable de Xilinx Zynq que junto al diseño de hardware en HLS (High Level Synthesis) se consigue ofrecer las mismas ventajas que las FPGAs, como el bajo consumo y potencia de computo, pero además se dispone de un procesador de alto rendimiento dual Corex-A9 (ARM) conectado a través de buses de alto capacidad. Además gracias a HLS se puede conseguir un desarrollo rápido y sencillo, abaratando los costes de producción.

Con esta FPGA de Xilinx se ha desarrollado una sonda de monitorización de red activa y para ello se han desarrollado dos diseños independientes. Por un lado se ha desarrollado un generador de paquetes o tráfico, encargado de crear datagramas UDP con un número de secuencia y el instante de salida hacia a la red. Por otro lado se ha creado un diseño de recepción de datagramas con el objetivo de analizarlos, clasificarlos y reenviarlos de vuelta a su origen. Aunque esta tarea parece que puede ser implementada en cualquier FPGA, cuenta con una comunicación directa con un sistema GNU/Linux implementado para sistemas empotrados, que ofrece los mismos servicios que cualquier otro sistema operativo, y en el que se pueden recoger todos los datos que se quieran de la FPGA.

1.1.-Motivación

A pesar de la cantidad de algoritmos y dispositivos dedicados a la clasificación de paquetes de red, este proyecto viene motivado por la necesidad de disponer de una herramienta de monitorización portátil para medir la calidad desde los centros de procesamiento de datos(CPD). De esta forma podemos tener un sistema en el que los técnicos no necesiten nada más que conectarlo en algún punto de las red y obtener directamente los datos deseados.

La principal pregunta es, ¿Por qué Xilinx Zynq?. El Soc programable Zynq de Xilinx ofrece la disponibilidad de hardware programable a bajo coste junto a toda la capacidad de un procesador Dual-Core y un bajo consumo de potencia eléctrica.

Otro punto a favor para la realización de este proyecto es la ausencia de la implementación de drivers para el diseño propuesto. El sistema GNU/Linux proporcionado por Xilinx para sistemas empotrados basados en ARM proporciona los drivers necesarios para trabajar con los procesadores de red y para la comunicación entre el sistema operativo y la parte programable del dispositivo.

Y por último este proyecto también viene motivado por la explotación y la investigación de las nuevas herramientas para el desarrollo de hardware desde interfaces y lenguajes de alto nivel, dejando atrás la programación de hardware sobre lenguajes descriptivos.

1.2.-Objetivos

Los objetivos que se han propuesto en este trabajo de fin grado son:

1. Desarrollo de dos módulos hardware, uno para la clasificación y análisis de paquetes de red y otro para la inyección de paquetes, a través de las herramientas de Vivado Xilinx.

2. Ampliación de interfaces de red de Gigabit en la plataforma ZedBoard.
    a. Esta ampliación se realiza sobre una prueba de concepto en la que se implementa un pequeño modulo de clasificación de paquetes de protocolo UDP.

3. Creación del kernel para sistemas empotrados Zynq.
    a. Ajustar las propiedades del kernel adecuadamente y añadir el sistema de archivos apropiado para el proyecto.

4. Interconexión de los módulos hardware desarrollados, las interfaces añadidas y el kernel a medida.

1.3.-Fases de desarrollo

La complejidad de este proyecto reside en la integración de las distintas partes, principalmente en obtener una comunicación fluida y fiable entre la parte programable del dispositivo y el procesador de sistema del mismo. Es por este motivo que el desarrollo se ha dividido en varias fases, en las que se desarrollan los distintos módulos del diseño deseado.

En la primera fase del proyecto se ha realizado una prueba de concepto en la que se ha implementado un pequeño modulo de clasificación de paquetes para obtener una aproximación del objetivo final, sin embargo, en esta prueba solo se ha trabajado con la parte programable del dispositivo como si de una FGPA pura se tratase. Otro de los objetivos de esta prueba era conectar correctamente un modulo de ampliación de la placa que contiene dos interfaces de red comunicadas a través del conector FMC.

En la segunda fase del proyecto se ha abordado en varias fases:

1. En primer lugar, la implementación de un Kernel de Linux a medida para sistemas empotrados Zynq con procesadores ARM.

2. Una segunda fase en la que la prioridad consistía en la conexión de una de las interfaces de red del modulo de ampliación, alojadas en la parte programable del dispositivo, con el kernel creado, de forma que esta interfaz sea reconocidas por el sistema automáticamente. Posteriormente se añadió la segunda interfaz del modulo pero esta tarea es repetir el proceso descrito en este mismo punto con la primer interfaz.

3. En la tercera fase se procede a desarrollar un módulo situado entre las interfaces de red y el sistema para el tratamiento de paquetes, aunque su funcionalidad en esta fase es transparente para el sistema. Además se modifica el kernel para que pueda comunicarse con este módulo.

4. En la última fase se ha desarrollado el módulo de clasificación y de inyección de paquetes de red.

1.4.-Estructura del documento

Este documento expone el desarrollo de un diseño hardware con herramientas de síntesis de alto nivel. Para ello expone en primer lugar una pequeña introducción al contexto del proyecto. El segundo capítulo se trata de una exposición de conceptos relacionados con la monitorización de red, el desarrollo de hardware y otro proyectos relacionados con estos temas. El tercer capítulo expone las características de la plataforma sobre la que se ha implementado la idea del proyecto.

El cuarto capítulo se centra en el proceso de implementación llevado a cabo para obtener el producto deseado. Este capítulo es amplio y está dividido en las diferentes fases de desarrollo del proyecto. El quinto capítulo expone las pruebas realizadas para

verificar que los diseños implementados son funcionales y se pueden integrar en diferentes entornos.
El sexto y último capítulo expone las conclusiones obtenidas durante la realización de este proyecto.

1.5.- Entorno de desarrollo

Para la realización de este proyecto por la parte de medios físicos se ha dispuesto de:

- Plataforma de desarrollo ZedBoard.
- Fuente de alimentación.
- Switch para una subred de pruebas.
- Ordenador de sobremesa con las siguientes características:

  - Intel(R) Core(TM) i7 CPU 920 2.67GHz 64bits

  -6 GB de memoria RAM.

  - Disco duro de 220 GB.

  -Tarjeta de red de 4 interfaces.

Respecto a las herramientas software utilizadas han sido:

- Sistema Operativo Centos 5.0
- Herramientas de desarrollo hardware de Xilinx:
  - ISE Design 14.1 (64bits).
  - XPS14.1(64bits).
  - SDK 2013.4 (64 bits).
  - ChipScope 14.1 (64bits).
  - Vivado 2013.4 (64 bits).
  - Vivado HLS 2013.4 (64 bits).
  - Impact 14.1 (64bits).
- Herramientas para el tráfico de red:
  - WireShark 1.8.1(64 bits).
  - tcpdump / tcpreplay

2.- Estado del arte

2.1.-Monitorización de red

Por definición la monitorización de redes consiste en un sistema encargado de controlar el estado de un conjunto de equipos a través de una infraestructura de conexión entre las máquinas[1]. Sin embargo en el caso de este proyecto la monitorización de red está orientada al control de calidad de las redes, es decir, a controlar los retardos en las transmisiones, las capacidades de los enlaces de la red, las pérdidas de datos, etc.

Las sondas hardware de monitorización de red están orientadas al control de las latencias y el ancho de banda en los enlaces y extremos de las redes. La latencia está definida como el retardo de un paquete de extremo a extremo, es decir, el tiempo que tarda un paquete en llegar de un terminal a otro a través de una red.

El ancho de banda por otro lado significa la capacidad de un enlace de proporciona servicio a un usuario. Además para medir el ancho de banda se puede enfocar de diferentes maneras, por ejemplo como capacidad de un enlace, como el ancho de banda disponible en una ruta de red durante un cantidad de tiempo definida, o como en rendimiento de una conexión TCP.

Normalmente para los usuarios de la red y para los administradores de la misma, el dato interesante de medir es el ancho de banda. Para medir el ancho de banda existen diversas técnicas[4]. Es técnicas y medidas de red se pueden clasificar bajo la siguiente clasificación:

- Activas y pasivas
- Reactivas y proactivas.

Las medidas activas se basan en la inyección de tráfico en la red para medir las características de la misma. Este tipo de medida están orientadas a conexiones punto a punto en una red o enlaces por los que no hay demasiada ocupación. Los grandes inconvenientes de las técnicas activas es la influencia del tráfico ajeno a los test de medidas y las políticas de enrutamiento y filtrado de paquetes de los routers que forman la red. Dentro de las técnicas activas podemos incluir las metodologías definidas anteriormente como la descarga de ficheros, los pares de paquetes y los trenes de paquetes.

Las medidas pasivas están basadas en la captura de tráfico en una red para la estimación de sus parámetros y análisis del rendimiento de la red. La información capturada se puede dividir en dos tipos, la información capturada del tráfico de la red y la información obtenida de los enlaces como routers y servidores. Una de las grandes ventajas de este tipo de técnicas es la ausencia de tráfico inyectado en la red, ahorrando capacidad de los enlaces para trabajar con el resto de aplicaciones. Obviamente la desventaja que ofrece es la dependencia de los datos de los que se dispone, ya que el tráfico varia constantemente.

Hay que tener en cuenta que las diferentes técnicas y medidas que se acaban de explicar deben ser usadas en función de los parámetros que se deseen obtener. Por ejemplo si se quieren analizar las pérdidas que se ocasionan en un red será más coherente recurrir a técnicas pasivas mientras que si se desea obtener el ancho de banda lo lógico es recurrir a las técnicas activas.

Las medidas reactivas y proactivas deben enfocarse desde otro punto de vista debido a que su objetivo es controlar el estado de la red basándose en la información obtenida a través de las técnicas activas o pasivas. Las técnicas reactivas se pueden definir como políticas de estudio de la red cuyo objetivo es informar de eventos cuando se encuentran medidas de red anormales. Por otro lado las medidas proactivas se encargan de analizar la red y modificar sus parámetros en función de los valores que se analizan. Un ejemplo típico de medida proactiva es cuando se encuentra un enlace saturado por exceso de tráfico se ejecuta un sistema para informar de que el tráfico debe ser dirigido por otra ruta o enlace.

Para este proyecto no es necesario profundizar en los sistemas reactivos o proactivos, ya que el objetivo es obtener información de la red y no en su gestión. Por este motivo es interesante conocer las diferentes técnicas activas más comunes, que son los métodos basados en la descarga de ficheros y los métodos basados en los pares de paquetes[2,3].

Los métodos de descarga de ficheros están basados en la descarga de ficheros con un tamaño 8 veces mayor a la velocidad del enlace. Una de las grandes desventajas de este método es la utilización del enlace por aplicaciones concurrentes que pueden a ocupar una gran parte del ancho de banda necesario para obtener unos resultados óptimos.

Los métodos basados en pares de paquetes están orientados a aprovechar la velocidad de los enlaces transmitiendo paquetes a la máxima velocidad permitida de un camino definido. Se pueden diferenciar en dos tipos, los pares de paquetes y los trenes de paquetes. Ambos métodos poseen características similares como el envió de ida y vuelta entre terminales y se diferencian en la cantidad de paquetes transmitidos. En el caso de los pares de paquetes, la ráfaga de paquetes está compuesta por grupos de dos y en los trenes de paquetes está compuesta de N. Ambos métodos están basados en el cálculo de la dispersión entre el primer y el último bit de los pares o trenes para obtener la capacidad del ancho de banda en los enlaces[2,3,4].

En este proyecto se ha considerado el método de los trenes de paquetes como un buen sistema en el que basar la sonda de monitorización desarrollada, de forma que a partir de esta se puedan obtener los datos necesarios para obtener la dispersión y la capacidad de las conexiones entre terminales. Además gracias a esta técnica se puede cumplir con el proyecto incluyendo la sonda de monitorización desarrollada dentro de la categoría de técnicas activas.

2.2.- Lenguajes para desarrollo hardware

En la década de los 90 se consolidaron los lenguajes de descripción de circuitos electrónicos como estándar para el desarrollo de componentes electrónicos basados en celdas programables. Los lenguajes considerados estándar por parte del Instituto de Ingeniería Eléctrica y Electrónica(IEEE) son el VHDL (Hardware Description Language) y Verilog [25,26].

Estos lenguajes descriptivos fueron creados buscando cierta similitud con los lenguajes para desarrollo de software clasificados como estructurados, en los que se definen códigos fácilmente legibles y estructurados en bloques de ejecución secuencial. Además con el uso de estos lenguajes aparecieron las herramientas para su desarrollo en las que se empezaron a agrupar códigos que definían una funcionalidad concretas como componentes creando diseños definidos como register-transfer level (RTL). Estos RTL supone, por parte de los programadores, el desconocimiento del funcionamiento interno de los bloques, definiendo un nivel de desarrollo por encima.

Finalmente los desarrolladores comenzaron a programar sus FPGAs mediante la instanciación de componentes y funcionalidades descritas en estos lenguajes. Sin embargo el uso de estos lenguajes en el desarrollo de diseños hardware para la programación de dispositivos programables y FPGAs implican largos periodos de tiempo de desarrollo, elevando los costes de los proyectos. Este incremento de tiempo viene motivado por los complejos diseños resultantes, con una frecuente aparición de fallos y su dificultad de depuración. Por estos motivos ha aparecido durante los últimos años el desarrollo de diseños basados en la síntesis de lenguajes de alto nivel y herramientas para la conexión de módulos desde interfaces gráficas de usuario.

La síntesis de lenguajes de alto nivel consiste en la traducción de lenguajes de programación de software como ANSI C y C++ a los lenguajes descriptivos VHDL o Verilog. Para la síntesis de estos programas Xilinx ha desarrollado su propia herramienta en la que los códigos escritos en C++ y con un conjunto de directivas propias son compilados o sintetizados a lenguajes hardware.

Esta herramienta puede generar a partir de un mismo código diferentes estructuras hardware, sin embargo la funcionalidad siempre será la deseada. Además se pueden realizar bancos de pruebas a nivel software antes de la compilación y bancos de prueba software ejecutados sobre el hardware sintetizado. De esta forma se puede agilizar el proceso de desarrollo sin necesidad de desarrollar pruebas para los esquemas RTL. Por último hay que añadir que la herramienta permite la generación de esquemas RTL a partir del hardware generado en lenguaje descriptivo.

Además de que el desarrollo de software está más extendido y es más sencillo encontrar soluciones a problemas computacionales, la herramienta de Xilinx (Vivado HLS) trata de optimizar siempre el código implementado buscando el máximo grado de paralelismo a nivel hardware. De esta forma el desarrollador simplemente tiene que encargase de indicar mediante directivas, que bloques código tienen alto coste computacional y deben sintetizarse de una forma concreta.

Una de las grandes ventajas del desarrollo sobre lenguajes de alto nivel es la portabilidad entre plataformas, ya que a partir de un mismo código obtenemos el mismo componente para las diferentes plataformas, tan solo cambiando el objetivo de la implementación del proyecto desarrollado sobre este tipo de herramientas.

Por último queda comentar una de las ultimas herramientas añadidas al entorno de desarrollo de Vivado. Xilinx a proporcionado mediante una interfaz gráfica, denominada IP Integrator , la posibilidad de la creación de diseños hardware basados en esquemas de bloques, en los que cada bloque define un componente RTL. Esto acelera todavía más el proceso de desarrollo evitando escribir más líneas de código, además de permitir incorporar a estos diseños de bloques, los componentes exportados a RTL mediante la sintientes de lenguajes de alto nivel.

2.3-Sondas basadas en software

Anteriormente se han visto diferentes técnicas para calcular el ancho de banda de los enlaces de la red, pero para poder realizar estos cálculos se debe disponer de los mecanismos y dispositivos necesarios. Durante mucho tiempo todos estos sistemas han sido implementados sobre desarrollos software, llegando a desarrollar programas que facilitan a cualquier tipo de usuario el cálculo del ancho de banda de los enlaces próximos a su conexión a Internet desde un navegador en su terminal personal.

Actualmente diversos programas se basan en el cálculo del ancho de banda para dirigir el tráfico o ofrecer diferentes servicios, como por ejemplo las aplicaciones multimedia, en las que se puede ofrecer diferentes formatos de audio o video en función del ancho de banda disponible.

A pesar de la cantidad de programas disponibles para calcular la calidad de servicio de las redes se suelen encontrar problemas en los resultados relacionados con la precisión de los tiempos de referencia tomados o con problemas de los terminales sobre los que se ejecuta el software. Cuando se ejecuta un programa de este tipo sobre un ordenador con una gran cantidad de tarea o procesos activos, se pueden observar variaciones de las medidas tomadas ya que se han visto afectadas por la ocupación de la CPU o la memoria.

Para evitar los errores de estimación en los test de velocidad de los enlaces existe el software QoS-Poll [2], que se encarga de estimar la cantidad de recursos del sistema ocupados, incluyendo el tráfico proveniente desde otras aplicaciones, y además define un margen de aceptación del ancho de banda medido en función de los parámetros anteriores.

2.4.-Sondas basadas en hardware

Una alternativa a las herramientas basadas en software para la estimación de los parámetros de la capacidad de los enlaces son los dispositivos hardware desarrollados específicamente para realizar esta tarea. Este proyecto es una de estas alternativas, sin aunque en este apartado se quieren mostrar los diversos sistemas implementados para realizar esta tarea. Principalmente se ha incidido en aquellos desarrollos que se hayan

realizado sobre la plataforma Zynq y se han tomado otros modelos sobre FPGAs como referencia.

Gracias a estos dispositivos hardware se pueden solucionar los problemas de precisión tomando tiempos de referencia en los instantes precisos durante la realización de los test de velocidad. Otra de las ventajas es que la ejecución de los dispositivos es independiente de la ocupación de la memoria o la CPU. Como se ha mencionado anteriormente una de las grandes desventajas de estos dispositivos es que tiene un alto coste y requieren grandes intervalos de tiempo para el desarrollo de los mismo.

Se pueden encontrar varios ejemplos de estas implementaciones en proyectos como ETOMIC[5] o clasificadores de tráfico basados en la tarjeta NetFPGA[6].

2.4.1.- High-accuracy network monitoring using ETOMIC testbed

ETOMIC (European Traffic Observatory Measurement In- frastructure) se trata de una red distribuida a lo largo de las universidades públicas europeas que permite a los usuarios realizar estudios de la red como medir el ancho de banda, el retraso del tráfico, la topología de la red, etc. Aprovechando esta infraestructura de ordenadores realiza conexiones punto a punto en una red de servidores junto con una tarjeta de red diseñada para la captura de tráfico para estimar los datos necesarios [5].

ETOMIC trabaja en colaboración con OneLab que se encarga de la generación de bancos de pruebas orientados al desarrollo de Internet.

Esta infraestructura proporciona medidas de alta precisión sincronizadas entre terminales del orden de 10 ns. Para lograr esto las máquinas de la infraestructura disponen de equipamiento de monitorización de red avanzado(Advance Network Monitoring Equipment, ANME) formado por una placa ARGOS para la sincronización punto a punto y CoMo Software para la monitorización pasiva de tráfico .

ARGOS es una placa, formada por una tarjeta NetFPGA y un PCB a medida., desarrollada por la Universidad Autónoma de Madrid para la sincronización punto a punto entre dos terminales de la red y la estimación aproximada de las marcas de tiempo incluidas en los flujos de paquetes.

El uso de la tarjeta ARGOS permite al proyecto ETOMIC evitar los problemas de ocupación de CPU o retrasos en la pila de red del sistema, ya que gracias a su diseño se puede evitar el timestamp a nivel de driver. Otra de las ventajas de esta tarjeta es que se puede utilizar integrada con un kernel de Linux de forma que el Linux interprete el dispositivo como un interfaz cualquiera de red.

2.4.2.- NetFPGA-based Traffic Classifier(Beta).

Se trata de un proyecto aun en desarrollo basado en la tarjeta de red NetFPGA para la clasificación de tráfico de red. Básicamente se basa en el uso de NetThreads[7] para la inspección de paquetes en busca de campos predefinidos.

La tecnología de NetThreads está diseñada para ejecutar fácilmente hilos a nivel software sobre la tarjeta.

2.5.- Zynq

Antes de explicar cualquier proyecto basado en Zynq resulta interesante realizar una primera aproximación de que es este sistema. Como se ha explicado a lo largo del documento, Zynq permite el desarrollo de hardware de alto rendimiento en poco tiempo , además de la implementación de software optimizado para este hardware y todo esto en bajo coste.

En la figura 1 se muestra la arquitectura de componentes de la placa. El dispositivo está formado por un procesador de doble núcleo ARM Cortex-A9, dos niveles de cache siendo el primer nivel una cache de 32 Kb por CPU y el segundo una cache compartida de 512 Kb. También cuenta con un controlador dinámico del protocolo de memoria con interfaces DDR3, DDR3L, DDR2 y LPDDR2[14].

La característica principal de esta es la comunicación directa entre el núcleo ARM y el área de lógica programable. Los buses que unen estas dos áreas permiten alcanzar velocidades de transmisión de hasta 100 Gbps.

Por otro lado otra una gran característica son los dos niveles de cache y el bus de aceleración ACP del área programable. En primer lugar está una cache de 512Kb compartida por ambos procesadores y después están las dos caches de CPU independientes con un tamaño de 32Kb, gestionadas a través de la SCU o unidad de control de coherencia cache .La SCU también está conectado al bus ACP, permitiendo al área programable escribir directamente en las caches de CPU.

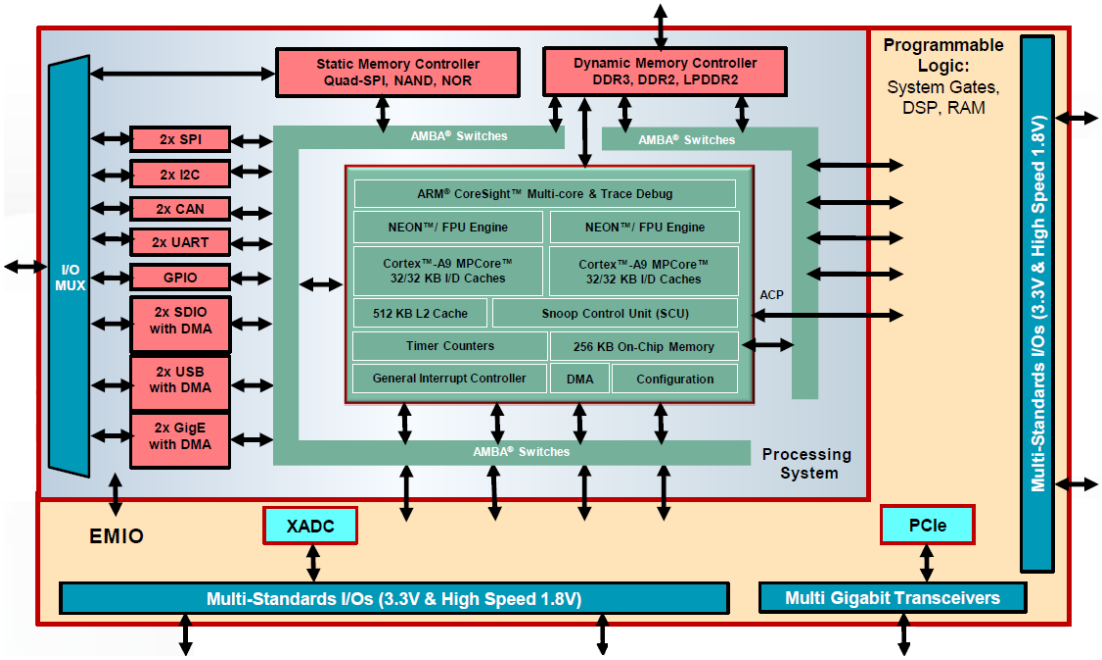


Figura 1.-Arquitectura básica Zynq [28]

Además dispone de una gran cantidad de periféricos conectados directamente a núcleo del procesador de sistema, con la capacidad de redirigir su funcionalidad hacia la parte programable.

Por último añadir que todas estas conexiones de gran ancho de banda se realizan sobre conexiones de protocolo AXI basado en ARM-AMBA [13,15].

AMBA se trata de un estándar diseñado para la interconexión de componentes en sistemas basados en Chip(SoC), como en el caso de Zynq. Su objetivo principal es la reutilización de diseños partiendo de interfaces de comunicación comunes basadas en protocolos.

2.6.-Proyectos basados en Zynq. Sondas y sistemas operativos

Hasta el momento no se han publicado sondas de monitorización de red basadas en Zynq pero sí que existen diversos proyectos orientados al tratamiento de paquetes de red. Estos proyectos buscan aprovechar el rendimiento proporcionado por el área programable de diferentes formas para operar el tráfico de la red. Por este motivo resulta interesante como base para este proyecto comprender los diferentes diseños.

2.6.1.- Zynq-7000 AP SoC Redirecting Ethernet Packet to PL for Hardware Packet Inspection Tech Tip.

El proyecto que se describe a continuación es un diseño para la inspección del contenido de paquetes basado en la implementación de bloques de memoria RAM en lógica programable(PL)[8].

Este diseño se encarga de recoger los paquetes recibidos a través de la interfaz de Gigabit Ethernet localizada en el Processing System de Zynq(PS) y son desviados hacia la parte programable(PL) para ser inspeccionados

Detalles de implementación:

- Tipo de diseño: PS & PL.
- Tipo de Software: Standlone.
- CPUs: 1 ARM Cortex-A9 666MHz
- Recursos del PS:
  - DDR3 533 MHZ.
  - Puerto AXI Maestro de propósito general.
  - OCM.
  - EMAC.
- Cores in PL:
  - Dos PL Block RAMS Used

Diagrama de bloques del diseño

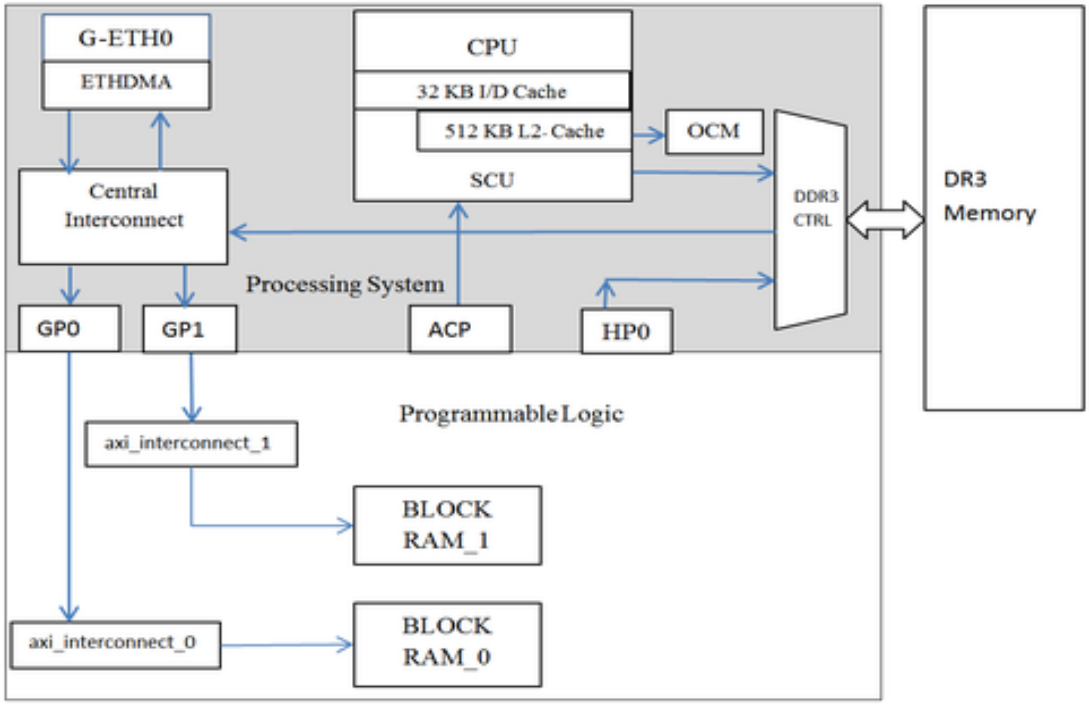


Figura 2.- Diagrama Bloques Interfaz de red conectada a Memoria [29]

En el diseño de la figura 2 se puede observar como el primer bloque de RAM situada en el PL está conectado al Maestro de propósito general AXI(MAXI_GP0) puerto 0. Esta conexión está pensada como un canal separado de los datos recogidos en el Ethernet

MAC, ideado para acceder a la información de control de los Cores implementados en el PL.

Por otro lado el segundo bloque de RAM se encuentra conectado al otro puerto maestro AXI de propósito general del PS. A través de esta conexión se reciben los datos del Gigabit Ethernet. Este segundo bloque de memoria está implementado de forma que es capaz de gestionar hasta dos paquetes consecutivos.

2.6.2.-Ethernet Performance with Jumbo Frame Support & PL Ethernet in Zynq-7000 AP SoC

En el proyecto que se describe en este apartado, se realiza la implementación de una tarjeta de red de alto rendimiento y una tarjeta de red para el tratamiento de paquetes jumbo. Estas dos ideas están separadas en proyectos distintos.

En primer lugar se utiliza el Processing System de Zynq(PS) con el Gigabit Ethernet situado en el mismo a través de la interfaz EMIO con la interfaz física 1000Base-X junto con el uso de transmisores en serie de alta velocidad en la parte programable(PL)[9].

Por otro lado se implementa un controlador Ethernet alojado completamente en el área programable de Zynq para soportar paquetes jumbo.
Hay que añadir que estos diseños están pensados para trabajar junto una distribución de micro-linux para ARM, por eso es necesario la implementación de drivers a medidas de estos diseños.

Aceleración del Gigabit Ethernet PS

Como se comenta anteriormente el primer proyecto consiste en utilizar Zynq para acelerar el rendimiento de la tarjeta de red del mismo. Para realizar esta tarea, se conecta la interfaz Ethernet del PS a través de la interfaz EMIO al área programable del chip para conectarlo a una interfaz de alta velocidad(1000Base-X). A su vez esta interfaz consiste en un Core que accede a través de transmisores de alta velocidad a la zona SFP de Zynq.

Diagrama de bloques del diseño

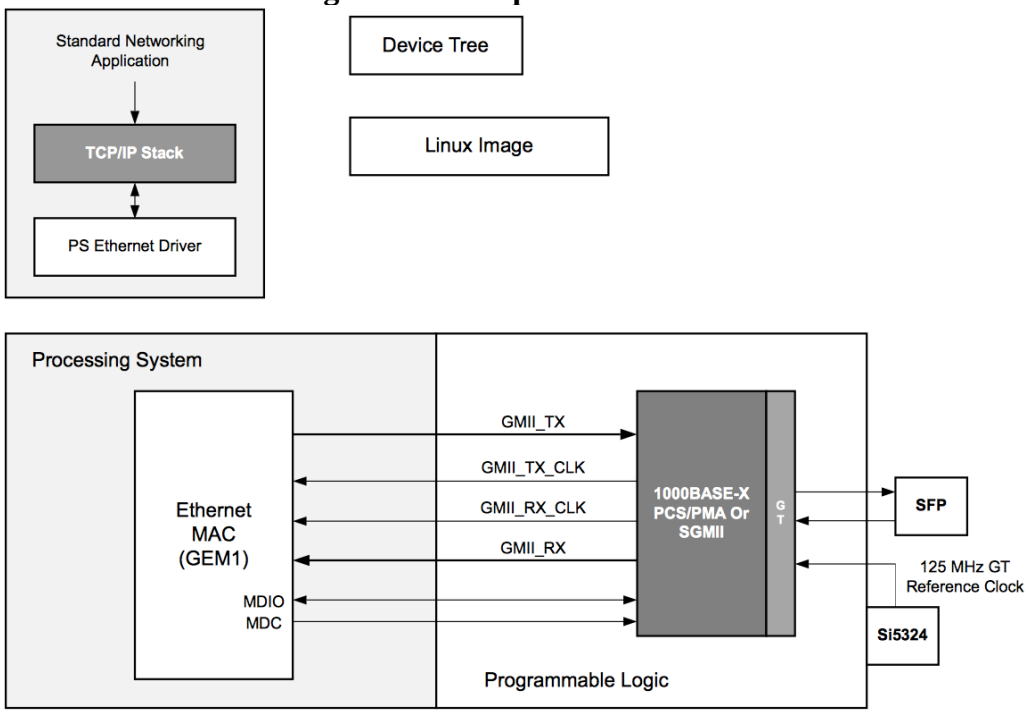


Figura 3.- Esquema Conexión PS LogiCore al medio físico [30]

Implementación de Ethernet PL para Jumbo Frames

Se trata de un diseño basado en el uso de los siguientes Cores en el área programable:

- AXI Ethernet.
- AXI DMA.
- AXI Interconnect.

El diseño utiliza el AXI Ethernet para la transmisión y recepción de paquetes conectado a la interfaz física de 1000Base-X. Una peculiaridad de este diseño es el uso de los puerto de master AXI de alto rendimiento para el acceso a la memoria de PS-DDR3[9].

Diagrama de bloques del diseño

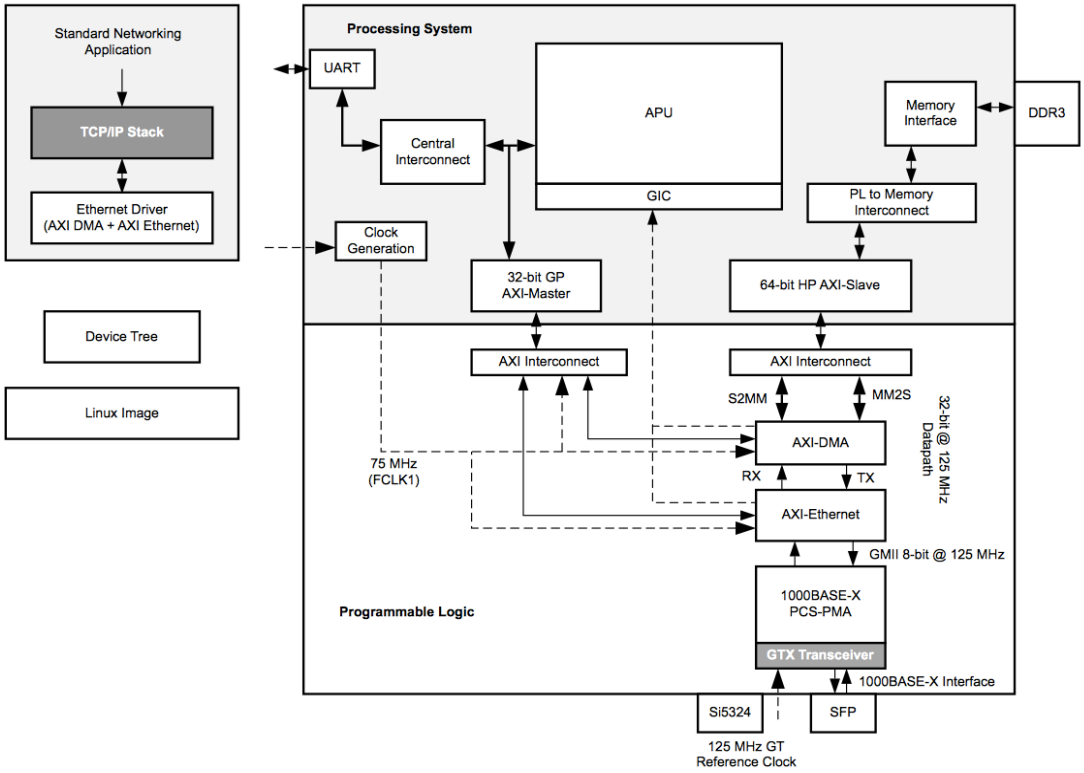


Figura 4.-Diagrama Conexión LogiCore en área programable [31]

Como se puede observar en la figura 4 el uso de los puertos de aceleración para conectar con el AXI DMA requiere el uso del uso de AXI-Interconnect, que facilitan la conversión entre los puertos de 64-bit del PS y los de 32 bit del AXI DMA.

2.6.3.- Zynq AP SoC Redirecting Ethernet Header to Cache via PL and ACP port Tech Tip

El proyecto que se describe aquí es una extensión del proyecto comentado en el apartado 2.5.1 en el que se utilizaba la plataforma Zynq para redirigir los paquetes desde el Gigabit Ethernet situado en el PS hacia el PL y almacenarlos en bloques de memoria RAM. En este caso los paquetes no son almacenados en memorias lógicas sino que son procesados en el PL y se trasladan directamente hacia la cache L2 de Zynq a través del puerto de aceleración(ACP). El rendimiento se ve incrementado en este proyecto al reducir el tiempo de procesamiento de la cabecera de los paquetes de red al evitar la lectura desde la memoria RAM DDR3 de Zynq y situarlos más cerca del procesador[10].

Diagrama de bloques del diseño

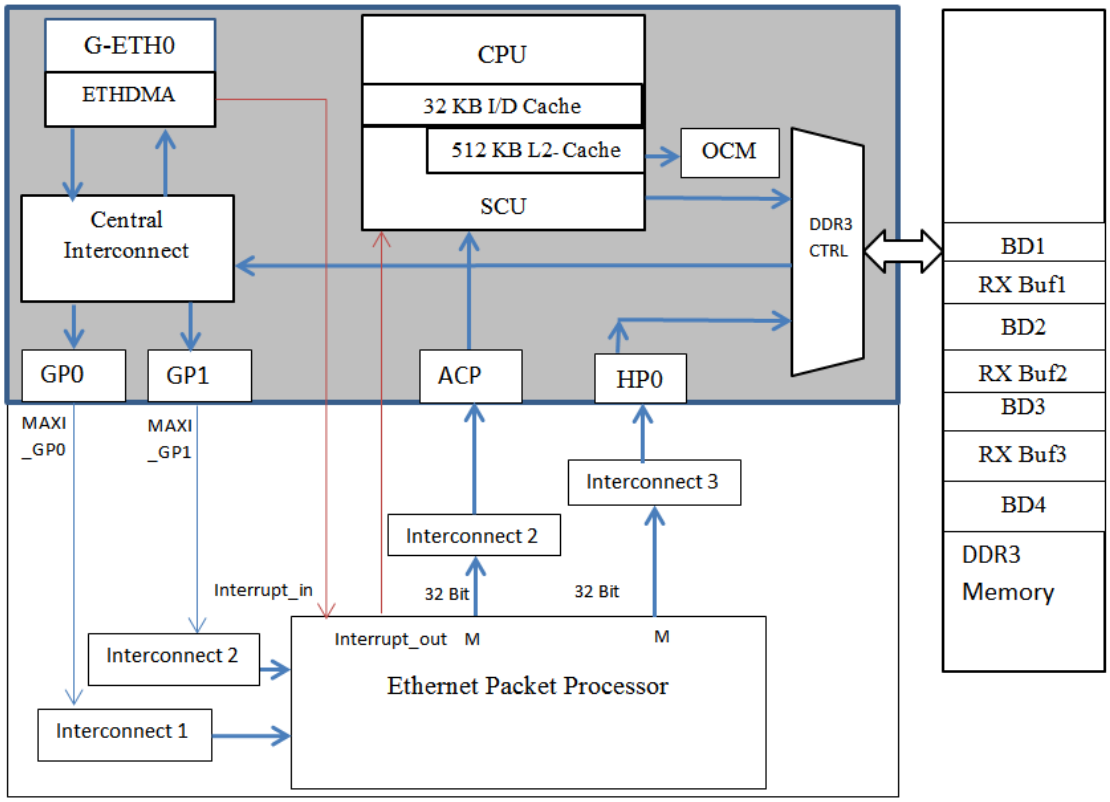


Figura 5.- Procesamiento de paquetes en área programable [32]

2.6.4.- LightWeight IP (lwIP) Application Examples

El proyecto que se comenta en este apartado es una librería de código abierto para procesamiento de red TCP/IP en sistemas empotrados como Zynq. Esta librería también puede ser aplicada a muchos otros dispositivos como Microblaze o PowePc[11]. Aunque no se trata de un desarrollo hardware propiamente dicho, no hay que olvidar que Zynq dispone de unidad de procesamiento física y no lógica como en el caso de Microblaze y puede explotar esta librería para obtener un mayor rendimiento.

En la información recogida sobre esta librería se describen ejemplos de aplicaciones que se ejecutan sobre estos sistemas empotrados, como por ejemplo, un servidor echo, un servidor web, un servidor TFTP y transmisiones TCP para ejecutar un test de rendimiento.

Esta librería está incluida en la plataforma para desarrollo de software de Xilinx, SDK(Software Development Kit). El único requisito para poder utilizar esta librería es que la plataforma hardware sobre la que desarrollar debe incluir el controlador Ethernet apropiado. En el caso de la plataforma Zynq es necesario incluir únicamente el controlador Gigabit Ethernet alojado en el PS y habilitar la opción "Checksum Offload".

La librería incluye funciones para programar transmisiones TCP o UDP con sockets y también para transmisiones sobre IP. Además la programación es muy similar a la de las

librerías de sockets en sistemas GNU/Linux. Gracias a esta librería se puede desarrollar rápidamente aplicaciones de red que se ejecutan sobre el procesador empotrado de Zynq.

2.6.5.- Xillybus

Aunque para este trabajo no se ha requerido un sistema operativo muy complejo al estar orientado en la interoperabilidad entre este sistema operativo y el hardware programado, resulta interesante observar las diferentes distribuciones que han sido desarrolladas para Zynq y otras FPGAS. Estas distribuciones muestran grandes cualidades propias de sistemas operativos diseñados para ordenadores personales y servidores de red.
Se trata de un proyecto de código abierto que consiste en la implementación de sistema simple de acceso a memoria para el traspaso de información entre la FPGA y un sistema GNU/Linux o Windows tradicional[12].

Básicamente se centra en la conexión de los componentes del sistema y la lógica programable a través de FIFO's. Al final el usuario dispone de un sistema operativo completo, en el caso de Zynq, ejecutando sobre el procesador de sistema que se comunica con el área programable a través de programas escritos en C que escriben y leen directamente de FIFO's.

El diseño que se implementa es el mostrado en la Figura 6. En el caso de Zynq el bus PCIe para la comunicación se sustituye por un bus AXI4.

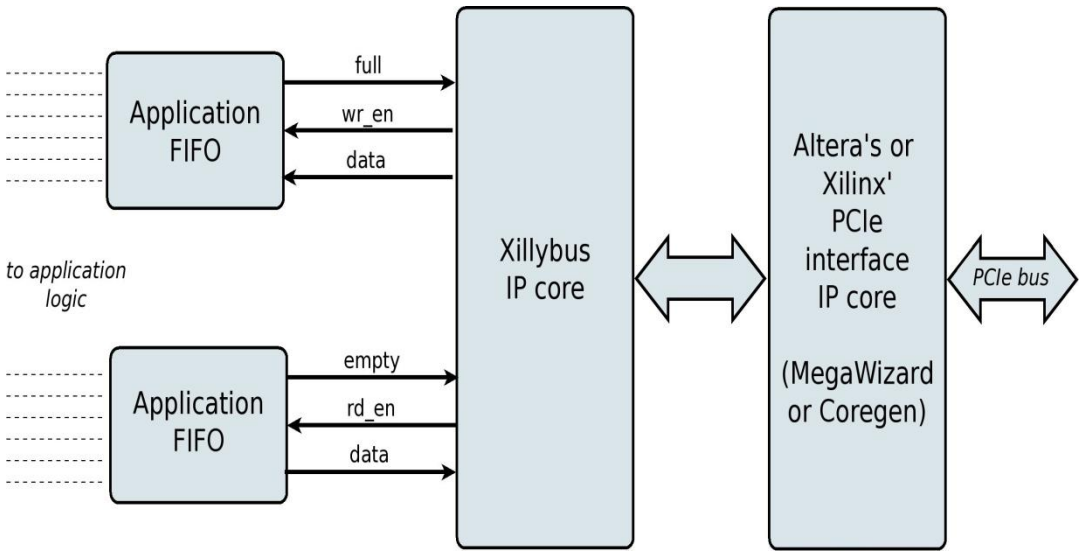


Figura 6.- Estructura del sistema Xillybus [33]

2.7.- Comparativa de los sistemas

Se han proporcionado diferentes puntos de vista con los que aplicar la técnicas para obtener el ancho de banda de los enlaces de la red. Pero lo que se quiere mostrar en este apartado son las diferencias entre los distintos sistemas desde un punto de vista práctico como puede ser el consumo de potencia o el impacto económico.

En la comparativa de las diferentes opciones propuestas se debe establecer los recursos físicos sobre los que se ejecutan los proyectos. Para los proyecto software se puede afirmar que en una misma máquina puede ejecutarse varios programas y por lo tanto el consumo vendrá definido por la cantidad de máquinas necesarias para realizar transmisiones con objetivo de medir el ancho de banda. Para aquellos proyectos hardware, incluyendo este proyecto, se deben tener en cuenta tanto máquinas terminales como FPGAs.

Para todos los proyectos contemplados se asume que los nodos intermedios de la red sobre los que circula el tráfico de red, como routers, suponen un coste nulo.

2.7.1- Potencia consumida

El consumo del dispositivo Zynq depende de múltiple factores como por ejemplo la frecuencia de reloj, la cantidad de bloques de memoria utilizados y la cantidad de CORES instanciados en el área programable. Por este motivo se ha decido realizar una estimación a través de la información proporcionada por la fuente de alimentación a la que se encuentra conectado el dispositivo utilizado para la realización de este proyecto. La tabla 1 muestra los datos observados:

Estado	**Potencia Consumida(Watios)**
Reposo	4.1 Watios
Ejecutando Linux	4.3 Watios
Linux + Interfaz de red PS	4.5 Watios
Linux + Interfaces de red en PL	7.6 Watios
Linux + Interfaces PS y PL	8 Watios

Tabla 1.- Consumo observado de zynq

Podemos observar que Zynq al disponer de un sistema operativo sobre la FPGA no necesita un consumo excesivo de potencia. Por ejemplo ejecutando un test de ancho de banda con un sistema Zynq en cada extremo de la red produciría un consumo medio de 16Watios.

Por otro lado se ha considerado dentro de los ordenadores de gama media-alta con las siguientes características:

- Procesador AMD Phenom II 955 BE (125 Watios)
- Placa base Gigabyte GA-890GPA-UD3H (36 Watios)
- Memoria RAM 4 GB DDR-1333 (30 Watios)

- Tarjeta gráfica Asus GTX460 DirectCu 1 GB (178 Watios)
- 1 disco duro mecánico (7 watios)
- Caja con 3 ventiladores de 140 mm (8 Watios x 3)

Sumando los datos que proporcionan los diferentes fabricantes tiene un consumo de potencia máxima de 400 Watios en ejecución. Si para ejecutar un software necesitamos dos máquinas para una comunicación de extremo a extremo, durante la ejecuación de ambas máquinas habría un consumo máximo de 800 Watios .

En el caso de la tarjeta programable NetFPGA debe estar conectada a un ordenador a través del puerto PCI por lo que el uso de la misma implica un consumo mínimo de 400 Watios .En el caso de aplicar la NetFGPA para el clasificador de tráfico analizado anteriormente consumiría de máximo los 400 Watios del PC.

En el caso del proyecto ETOMIC también se conecta el equipamiento ANME a una máquina [4], además este utiliza la tarjeta ARGOS que requiere de una salida a un terminal, una pantalla tiene un consumo habitual de 60 Watios, por lo que supondría un consumo máximo de 460Watios.

En conclusión podemos observar que la autonomía de Zynq proporciona una descenso notable del consumo de potencia eléctrica frente a las soluciones que se encuentran en el mercado.

2.7.2.- Impacto económico.

Otro de los factores que motivan este proyecto es el coste que supone disponer de este tipo de aplicaciones. Por este motivo resulta interesante comparar el precio de los diferentes proyectos mencionados.

En primer lugar analizamos la repercusión económica que conlleva desarrollar este proyecto. Todos los diseños se han desarrollado orientados a la plataforma ZedBoard que básicamente es un conjunto de periféricos más el SoC programable Xilinx Zynq. Esta plataforma tiene un precio para estudiantes y profesores de 319 $ (233.78 €). Además se dispone de un módulo de ampliación de interfaces de red conectado al FMC con un precio de 1000 $ (732.84 €). En total el desarrollo de este proyecto supone un coste de 1319 $ (967 €) en materiales.

El coste de una sonda de monitorización a nivel de software supone el precio de un ordenador de mesa con prestaciones relativamente altas. Se debe suponer que se quiere obtener el máximo rendimiento y por lo tanto tiene que tener una gran capacidad de computo pero se puede prescindir de computación de video por lo que no es necesario una tarjeta gráfica de alto rendimiento. Un ordenador de estas características tiene actualmente un precio medio de 800 €.

En el caso de proyectos con hardware dedicado como NetFPGA se debe disponer del ordenador de características similares al caso anterior, con un precio aproximado de 800

€ y del hardware específico. En el caso de NetFPGA con interfaces de 1Gbps supone un coste para personal universitario de 599 $ (439 €).

Por otro lado el equipamiento de monitorización de ETOMIC consta de la tarjeta ARGOS, que como se dice anteriormente está formado por una tarjeta NetFPGA y un modulo PCB a medida, elevando el precio. Aunque la máquina necesaria para dar soporte a el equipamiento avanzado de monitorización de red del proyecto ETOMIC debe tener características similares a las de un servidor de alto rendimiento[24], la tarjeta ARGOS puede conectarse a cualquier ordenador, por lo que el precio de la máquina estaría entorno a los 800 € también.

En la tabla 2 se pueden observar un resumen de los precios mínimos aproximados para la plataforma de desarrollo de un dispositivo como el especificado en este proyecto.

Dispositivo	**Precio Mínimo**
Sondas basadas en software	~ 800 €
Sonda basada en Zynq	~ 967 €
Sonda basada en NetFPGA	~ 1239 € (800 + 439 €)
Proyecto ETOMIC	~ 1539 € (800 + 439 € + 300 €)

Tabla 2.- Precios sondas de monitorización

3.- Descripción de la plataforma

3.1.-Introducción a Zedboard

La plataforma ZedBoard es una placa para la evaluación y el desarrollo sobre el chip Xilinx Zynq-7000. Esta plataforma dispone de un procesador Dual Corex-A9(ARM) para el procesamiento de sistema(PS) combinado con un bloque de 85000 celdas de área programable(PL) [17]. Además esta plataforma está diseñada con un amplio abanico de periféricos que permiten exprimir al máximo la potencia de los procesadores.

Características:

- Memoria:
  - Memoria RAM 512 MB DDR3 (módulos de 128 Mb x 2).
  - 256 Mb de memoria QSPI Flash.
- Interfaces Entrada/Salida
  - USB-JTAG.
  - Interfaz de red de 10/100/1G Ethernet.
  - USB OTG 2.0.
  - Lector de tarjetas SD.
  - USB 2.0 FS USB-UART.
  - 5 conectores PMOD (1 al PS, 4 al PL).
  - 1 conector LPC FMC.
  - 1 AMS Header.
  - 2 botones de reset(1 para PS, 1 para PL).
  - 7 botones(2 PS, 5PL).
  - 8 interruptores conectados unicamente al PL.
  - 9 LEDs para el usuario (1 PS, 8 PL).
  - 1 DONE LED (PL).
- Relojes en placa:
  - 33.33 MHz para el PS.
  - 100 MHz dirigidos al PL.
- Vídeo/Audio:
  - Salida HDMI.
  - Salida VGA de 12-bit.
  - 128x32 OLED Display.
  - Entrada/Salida de audio, salida de cascos y entrada de micrófono.

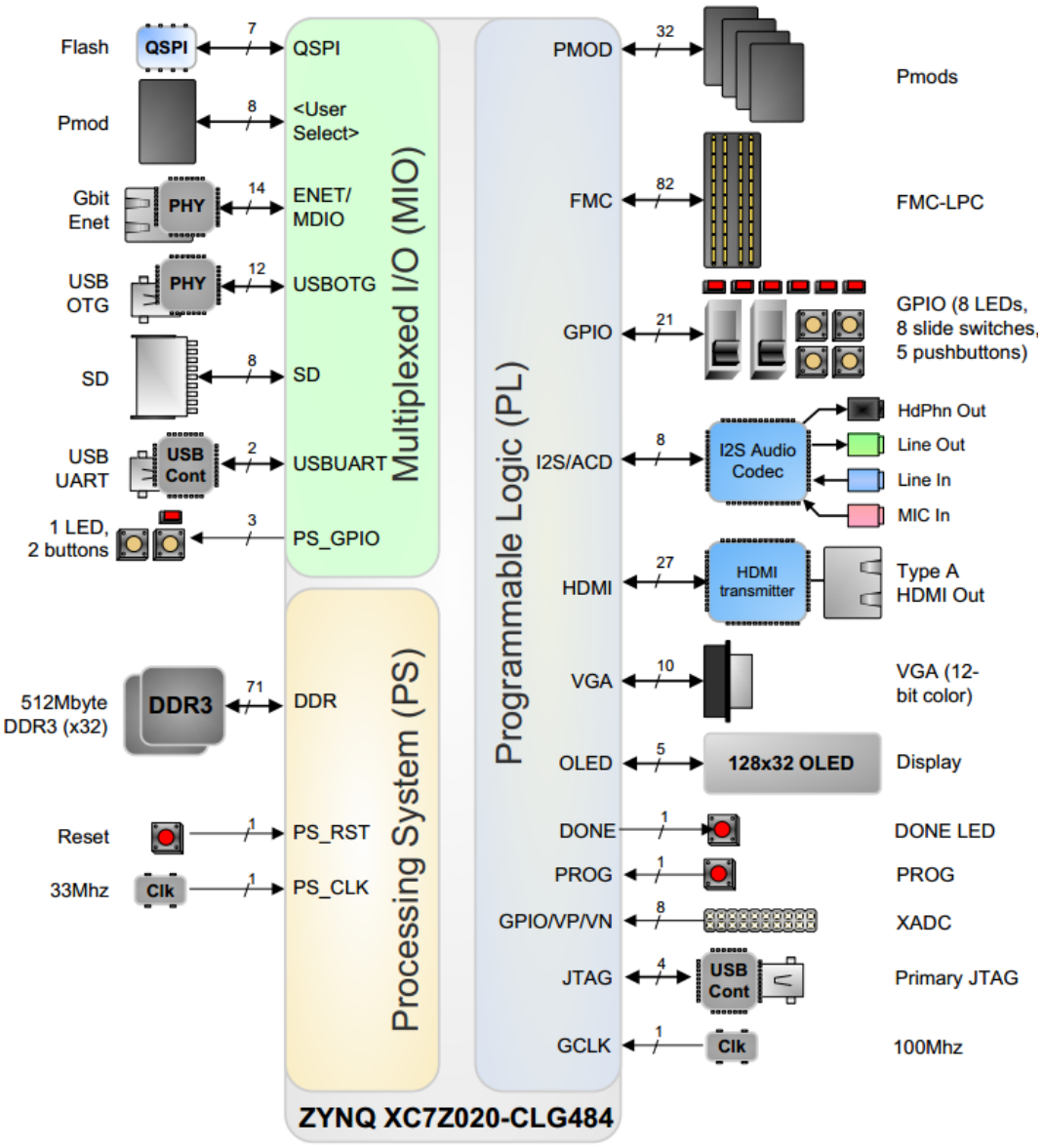


Figura 7- Diagrama básico de Zynq [34]

3.2.- Componentes de ZedBoard

La plataforma ZedBoard dispone de más periféricos que los utilizados para el proyecto y describir todos ellos es innecesario. Por ello a continuación se describe la funcionalidad de los componentes que tienen o han tenido algún aspecto relevante para el desarrollo del trabajo.

Memoria QSPI Flash

La memoria de entrada y salida múltiple SPI Flash se usa habitualmente para la configuración no volátil de códigoo. Su uso más común suele estar dedicado a la inicialización de sistemas en el PS y también en la programación del Bitstream para el PL.

Sus capacidades son:

- 256 Mbit
- Trabaja a frecuencias de reloj de 100 MHz.
- Alimentada desde 3.3 V.

Lector Tarjetas SD

La disposición de este lector de tarjetas permite el almacenamiento no volátil de datos. Este lector de tarjetas está conectado únicamente al procesador de sistema de la plataforma, es decir, no se puede trabajar con él en el área programable.

Relojes de la tarjeta

El PS esta exclusivamente alimentado por un reloj de una frecuencia de 33.33 MHz, sin embargo dispone de un generador de señales de reloj orientado a la parte programable, pudiendo especificar hasta 4 señales de reloj dedicadas siempre que cumplan las restricciones de tiempos. Además el PL dispone de una señal de reloj por defecto de 100 MHz.

Señales de Reset

La placa dispone de las siguientes señales de reset por defecto:

- Power Reset : Encargado del reset del chip completo. Esta señal es gestionada de forma que mantiene el reset activo mientras que un comparador conectado a una señal de la alimentación se vea modificado.
- Botones programables: Estos botones pueden ser configurados desde el PL para especificar una señal de reset.
- PS Reset: Esta señal de reset se encarga también de recuperar el estado original del sistema, pero al contrario que el Power reset, esta señal restaura únicamente la funcionalidad lógica del dispositivo.

10/100/1G Ethernet PHY

ZedBoard dispone de un puerto de red basado en el medio físico del fabricante Marvell, 88E1518. Marvell mantiene privada la información sobre este dispositivo, pero dispone dos interfaces, una para entrada y otra para salida de datos en el conector RJ45. Puede estar conectado a los pines de bloque MIO(conexiones al área programable), pero trabaja directamente sobre el PS. Por último se puede añadir que viene alimentado por 1.8 V.

Conector de ampliación LPC FMC

Se trata de un puerto simple pensado para soportar la conexión de módulos externos con una gran cantidad señales. Dispone de 68 puertos de entrada/salida, configurables en dos pares de 34. Está conectado directamente a los bancos de alimentación de la parte programable. Dispone de jumpers para ajustar la configuración de voltaje que alimenta al módulo que se conecte a través de este puerto, siendo posible proporcionar 1,8V, 2,5V y 3.3V.

3.3- Descripción FMC-Ethernet.

El desarrollo de la sonda propuesta en este proyecto, tanto el módulo de clasificación como el módulo de inyección de paquetes requieren de más de una interfaz física de red. Actualmente en el mercado solo se encuentra el módulo descrito en este apartado para disponer de más interfaces de red.

Por los motivos anteriores y a que este módulo de ampliación tiene un coste elevado, todo el procesamiento de red se ha realizado usando estas interfaces físicas, dejando la interfaz física alojada en el procesador de sistema de ZedBoard como una interfaz de configuración.

Este módulo de ampliación dispone de dos interfaces físicas de Ethernet Gigabit compatibles con el estándar de red IEEE 802.3, además cada interfaz de red trabaja independientemente de la otra interfaz [19]. Para conectarlo a una FPGA este módulo dispone de un conector LPC FMC.

Las dos interfaces de red de las que dispone el módulo pueden trabajar en dos modos diferentes para alcanzar los 1000 Mb de transferencia. Estos modos son GMII y RGMII.

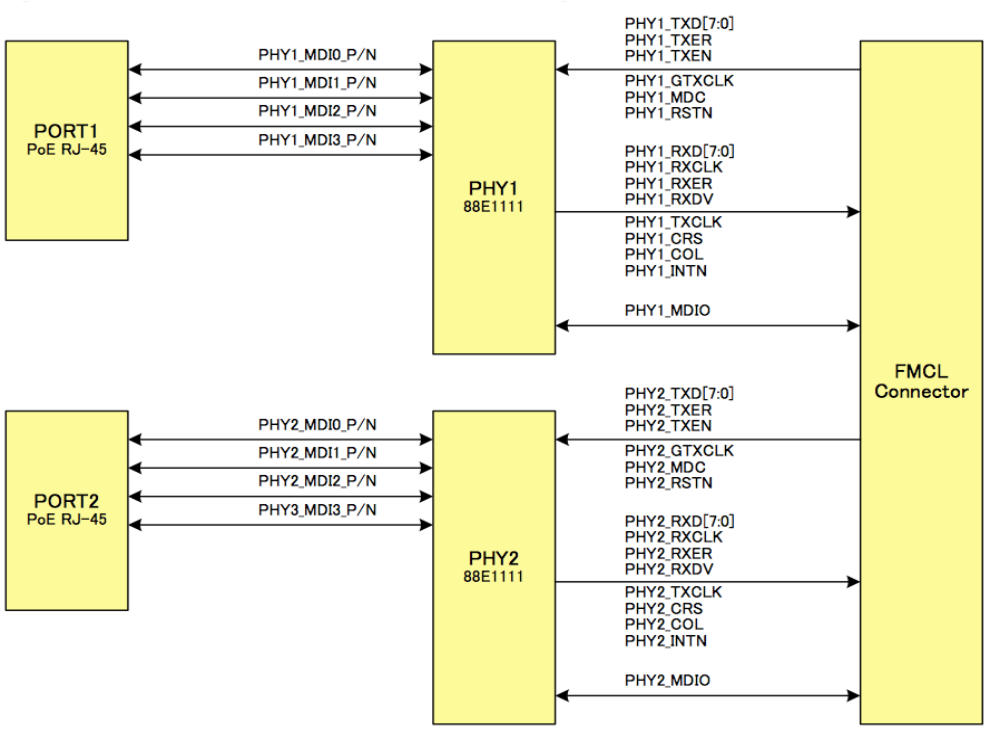


Figura 8.- Arquitectura Módulo FMC-Ethernet [35]

Es importante entender el significado de los LEDs de los que dispone el módulo, ya que a simple vista permite apreciar si el medio físico del módulo funciona como es debido. Por ejemplo permite comprobar a qué velocidad de transmisión está trabajando, si está o no enviando/recibiendo paquetes de red o si se ha producido algún fallo.

Al igual que la plataforma ZedBoard los conectores físicos RJ45 pertenecen al fabricante Marvell, modelo 88E1111-B2-BAB1C000.

4.- Desarrollo

A lo largo de este apartado se va a explicar cómo se ha desarrollado el dispositivo propuesto en el proyecto. Como se indica al principio del documento, para realizar esta tarea el proyecto se divide en las siguientes 4 fases:

- Conexión módulo FMC y prueba de concepto.
- Creación del kernel y comunicación con el PS
- Establecer plataforma y modificación del kernel.
- Desarrollo del módulo de clasificación

4.1.-Conexión FMC Ethernet y prueba de concepto.

El objetivo principal de esta fase consiste en el desarrollo de un código sencillo para programar la FPGA de la plataforma ZedBoard conectada con el módulo de ampliación FMC-Ethernet. El comportamiento de este diseño consistirá en un rebote simple de paquetes de protocolo de transporte UDP por una de las interfaces del módulo FMC.

Por otro lado, se va a explicar con detenimiento cada uno de los componentes que forman el proyecto, ya que resulta bastante interesante entender sus capacidades para futuras fases del proyecto. Hay que incidir en que una vez se conoce el comportamiento de cada uno de estos componentes la única tarea que hay que realizar es integrarlos todos dentro del mismo diseño.

4.1.1.- LogiCore Tri-Mode Ethernet MAC

Este componente se trata de un procesador o CORE lógico desarrollado por los fabricantes de Xilinx que permite el procesamiento de datos a una velocidad entre 10/100/1000 Mb/s [27].

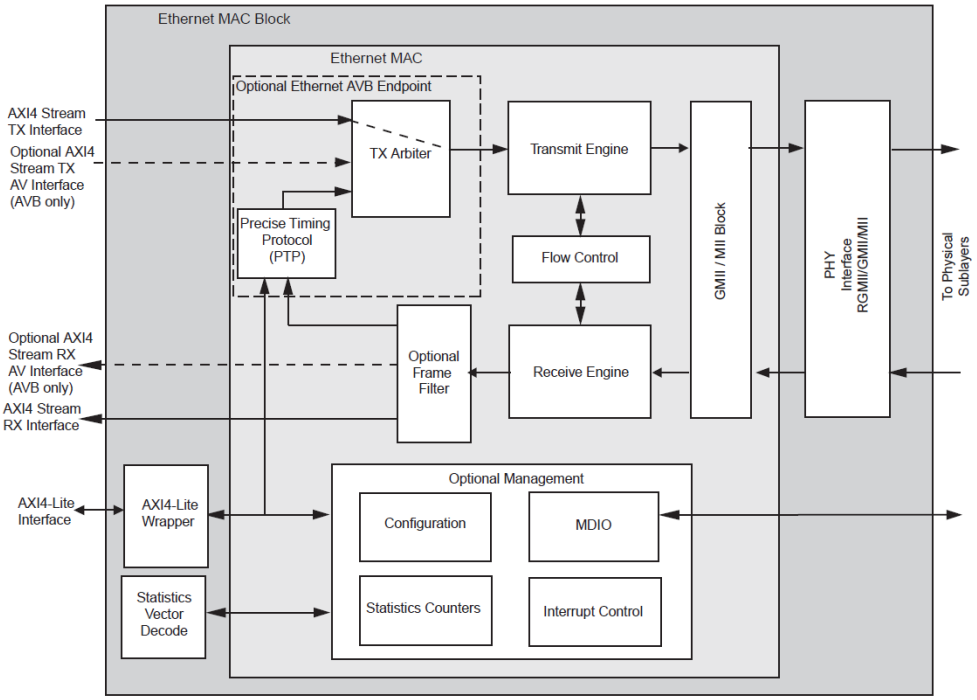


Figura 9.- Diagrama Bloques Tri-Mode Ethernet MAC [36]

En la figura 9 se pueden observar los principales módulos de funcionalidad del componente, así como las interfaces de entrada y salida del mismo.

- **Ethernet MAC Block**: Se trata del bloque principal de cualquier componente Ethernet MAC. Simplemente se encarga de encapsular las diferentes partes de un componente Ethernet lógico.

- **AXI4-Lite Wrapper**: Bloque que permite la conexión del bloque con una interfaz AXI4-Lite, a través de la cual, el componente es configurado.

- **Statics Vector Decode**: Almacena la información proporcionada por el componente durante el procesamiento de paquetes de red.

- **PHY Interface**: Medio de conexión con el dispositivo físico en cualquiera de los modos posibles (RGMII, GMII, MII).

- **PTP(Precising Time Protocol)**: Proporciona la lógica necesaria para la implementación del protocolo PTP para tomar tiempo precisos en los paquetes de red, sin embargo requiere de hardware y software adicional para un correcto funcionamiento.

- **Transmit Engine**: Es el módulo encarga de tomar los datos de la interfaz AXI4-Stream de transmisión(Tx) y convertirlos en el formato del modo de salida, como por ejemplo, GMII. Es este módulo el encargado de agregar los campos de preámbulo y "checksum" al paquete antes de salir hacia el medio físico. Por último proporciona información al módulo de "Statics Vector Decode".

- **Receive Engine**: Este módulo se encarga de tomar los datos de la interfaz GMII y transformarlos para colocarlos en la interfaz AXI4-Stream de recepción(Rx). Retira los campos de preámbulo y "checksum" del paquete y comprueba que no haya errores. Por Por último proporciona información al módulo de "Statics Vector Decode".

- **Flow Control:** Este componente permite la gestión de todo el componente de forma que puede realizar pausas en la recepción o transmisión de datos.

- **GMII/MII Block**: Se encarga de transformar los flujos a los distintos tamaños de bit. En caso de GMII opera únicamente a 1 Gb/s y con tamaño de 8-bit. Para MII trabajo a 4-bit.

4.1.2.- Clasificador HLS Básico

Para realizar una primera aproximación al diseño final durante esta prueba se ha implementado, a través de la herramienta de Vivado HLS, un módulo hardware que se encargue de procesar los paquetes recibidos y decidir porque interfaz deben transmitirse.

Anteriormente se ha explicado que es Vivado HLS y que se puede lograr con esta herramienta, pero en este apartado se quiere mostrar cómo se trabaja con esta herramienta para obtener todos los beneficios que puede aportar el desarrollo de hardware desde lenguajes de alto nivel.

Antes de empezar a desarrollar el componente deseado hay que entender cómo funciona Vivado HLS. Básicamente esta herramienta analiza cada función de un código en C en un bloque hardware RTL con una máquina de estados, con un mínimo de tres estados, inicialización, ejecución y fin. En el estado de ejecución se definirán las operaciones deseadas, pudiendo generar nuevos estados o bucles dentro de la máquina de estados, por este motivo cuando se desarrolla sobre esta herramienta es necesario recurrir a las utilidades de la misma para analizar su comportamiento y ajustar el código escrito para obtener unos resultados u otros acordes al diseño deseado.

Cuando se realizan operaciones aritméticas y bucles, el compilador de Vivado HLS realiza un análisis de dependencia y riesgos de datos que determinaran el hardware y los ciclos necesarios en los que se completará una transacción en el componente desarrollado. Es por esto que para desarrollar sobre esta herramienta no basta con conocer los lenguajes de alto nivel. Es frecuente encontrar códigos en los que no basta con modificar las operaciones para obtener el hardware deseado, si no que es necesario añadir directivas sobre el código que el compilador interpreta durante la síntesis para definir el hardware y el comportamiento. Obviamente cuenta con una gran diversidad de directivas y para este proyecto muchas son irrelevantes, por ello se explicarán aquellas que hayan sido utilizadas.

Lo último que se debe añadir sobre el compilador de Vivado HLS es que analiza el código buscando el máximo grado de paralelismo en la ejecución de operaciones, siempre que no existan riesgos de integridad de datos.

El primer paso en un proyecto de Vivado HLS es definir que la solución será para el dispositivo ZedBoard(en el caso de este proyecto). El siguiente paso es definir la función principal. En esta fase del proyeto y en general en todas, las funciones implementadas tendrán entrada y salida a través de bus del tipos AXI4-Stream. Para definir estos buses es necesario definir una estructura de datos con las señales asociadas al bus. La forma común de definirlo es la siguiente:

```
template<int D>
struct "minombre"{
        ap_uint<D> data;
        ap_uint<D / 8> strb;
        ap_uint<D / 8> keep;
        ap_uint<1> user ;
        ap_uint<1> last;
}
typedef  "minombre"<"valor de D"> "nombre-AXI";
```

Como se puede observar en la estructura anterior se han definido muchas señales del protocolo AXI. Para este proyecto en la definición podemos evitar el uso de *user* y *strb*, ya que los componentes del diseño no la utilizan. Además el valor de *D* se ha sustituido por 32.

En total los parámetros de la función principal son cinco, tres variables del tipo AXI4-Stream definido, en concreto tiene una entrada y dos salidas. Los otros dos parámetros son una entrada de 32 bits para indicar el protocolo por el que debe clasificar los paquetes y una salida de 32 bits también que cuenta la cantidad de paquetes procesados. Hay que añadir que las variables de tipo AXI4-Stream deben declararse como buses de AXI4-Stream a través de la directiva:

- ***HLS RESOURCE variable=<variable_streaming> core=AXI4Stream***

Con esta directiva el compilador de Vivado HLS traducirá las entradas y salidas deseadas como grupos de interfaces de AXI4-Stream.

Dentro de la función se realiza una lectura secuencial de la entrada de tipo Stream, comprueba que el campo correspondiente al tipo de protocolo de transporte es el indicado en la entrada de 32 bits y escribe secuencialmente en una de las dos salidas de tipo Stream.

El pseudo-código correspondiente al código implementado es el siguiente:

```
Leer cabecera paquete:
do{
        -leer entrada
        -guarda entrada en buffer
} hasta entrada .last == 1
 Decidir interfaz salida:
 Si buffer(6) contiene UDP then
        do{
                -escribir buffer en salida 1
        } hasta buffer.last == 1
else
        do{
                -escribir buffer en salida 2
        } hasta buffer.last == 1
```

La lectura de la entrada y escritura de las salidas puede incrementar notablemente la latencia y producir perdidas, por lo que será necesario aplicar la directiva **PIPELINE**. Esta directiva obliga a realizar las operaciones del bucle de forma segmentada, de forma que paraleliza siempre que sea posible las operaciones del bucle.

Por otro lado es importante que las sentencias condicionales del código se ejecuten en un solo ciclo de reloj, ya que si no se producirían pérdidas.

4.1.3.- Implementación

La implementación de esta fase del proyecto se ha basado en el código de prueba proporcionado del LogiCore de red comentado anteriormente. En este código de prueba se proporcionaba el diseño de la figura 10.

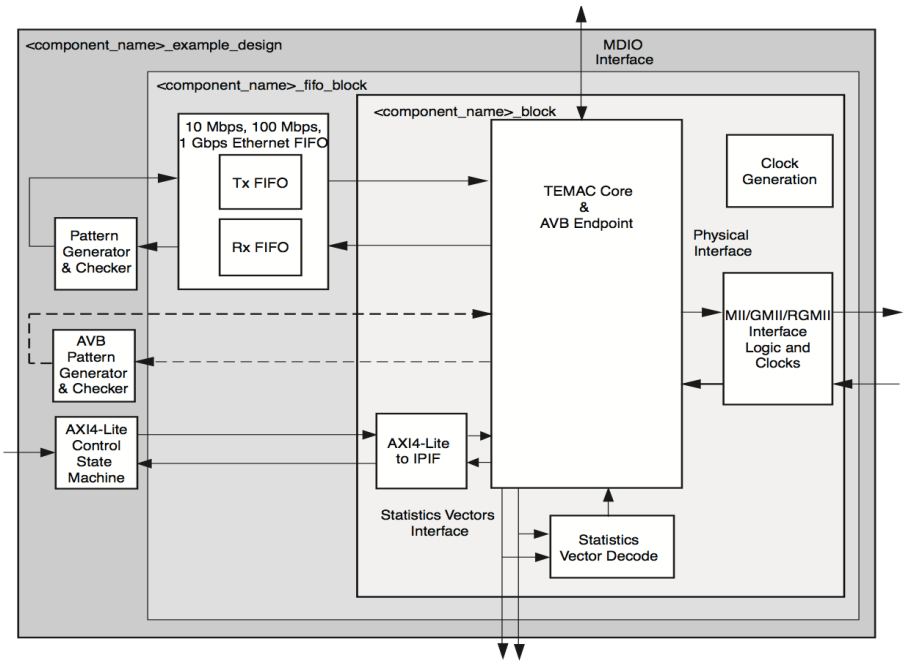


Figura 10.- Diseño EjemploTri-Mode LogiCore [37]

En el diagrama de bloques de la figura 10 hay que comentar dos aspectos fundamentales. En primer lugar simplemente se encapsula el LogiCore descrito anteriormente en un módulo de forma que la interfaz AXI4Stream queda directamente conectada con una FIFO para transmisión y otra para recepción de datos. En segundo lugar el módulo Pattern Generator se encarga de realizar un rebote de paquetes, es decir, todo paquete recibido se envía de vuelta.

Una de las primeras pruebas consiste en conectar la interfaz GMII con el medio físico del FMC-Ethernet, sintetizar y programar este diseño en la placa. Para conectar el medio físico es necesario añadir un archivo de restricciones en el proyecto antes de la síntesis, en el que se realice la relación entre las señales orientadas a las entrada y salida con los pines físicos del dispositivo.

Para todas las pruebas todos los bits del Vector de configuración permanecen con una palabra fija, de forma que se realice una transmisión sencilla, sin retraso de transmisión, ni control de flujo.

A partir de este momento el objetivo es ir ampliando el proyecto que se acaba de probar. El siguiente paso ha consistido en ampliar el módulo de rebote de paquetes, de forma que contenga otra entrada y otra salida de paquetes para poder añadir dentro del bloque de “example_desgin” otro Tri-Mode Ethernet y conectar su interfaz GMII con la segunda interfaz física del módulo FMC-Ethernet.

El siguiente paso es sencillo, solo consiste en sustituir el Pattern Generator con el doble de interfaces por el clasificador exportado desde Vivado HLS. Es recomendable realizar un test bench software antes de programar la FPGA con el diseño implementado, ya que es durante esta fase donde se pueden encontrar los problemas de integración. El resultado debe ser similar al de la figura 11 .

Como prueba de concepto se obtienen una latencia en el procesamiento de paquetes de 2 us y consumiendo una potencia máxima de 7 watios, cumpliendo una de las características principales de los chips de bajo coste. Por otro lado la latencia de procesamiento aparentemente puede parecer demasiada alta ya que estamos trabajando con interfaces de 1 Gbit que pueden tener hasta 1.48 x 10^6 paquetes por segundo, o dicho de otra forma, puede procesar un paquete en 0,676 us. Por otro lado hay que tener en cuenta que 2 us es el tiempo en procesar un paquete en recepción, clasificarlo y procesar un paquete en transmisión, suponiendo un retardo de:

$$0.676\ us(Rx) + 0.676\ us(Tx) + T.Clasificación \approx 1.2\ us + 0.8\ us$$

Además el objetivo de esta fase es conseguir un prototipo funcional por lo que el diseño implementado no ha sido optimizado.

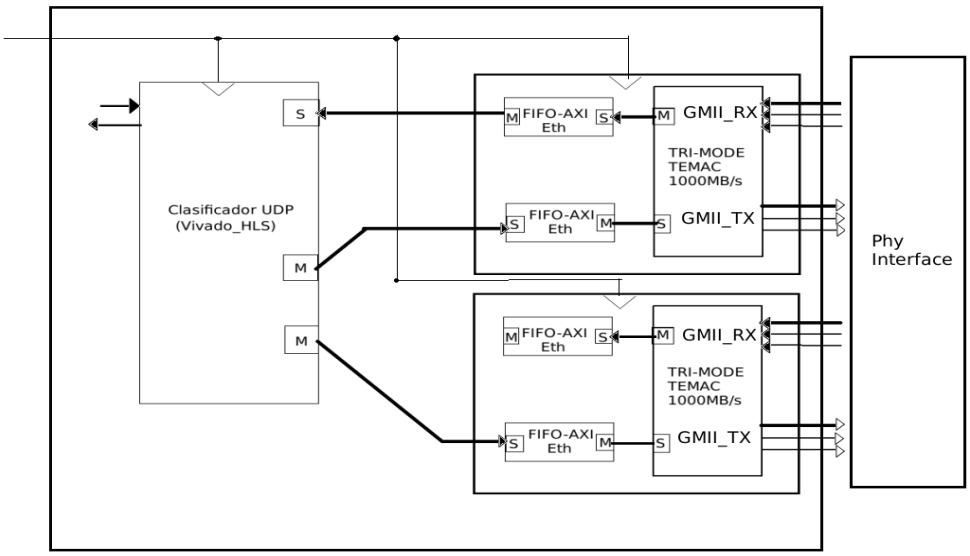


Figura 11.- Diseño hardware Clasificador Básico

4.2.- Creación del Kernel y comunicación con el PS

Hasta ahora el diseño no cumple muchos de los objetivos propuestos al principio del documento, sin embargo se han establecido las bases para el desarrollo sobre la plataforma ZedBoard. Como se comenta al principio del documento en la fase que se describe a continuación está dividida en otras fases para dividir la complejidad del problema. De esta forma la realización del proyecto se ha realizado en el siguiente orden:

- Implementación del Kernel.
- Conexión AXI-Ethernet y PS.

4.2.1.- Creación del Kernel.

El objetivo de esta fase es conseguir un sistema operativo embebido ejecutando sobre el procesador de sistema de Zynq. Para ello lo primero es definir el sistema o hardware sobre el que se va a ejecutar dicho kernel. En este proyecto se ha recurrido a la herramienta IP Integrator del programa Vivado, donde se pueden crear proyectos orientados a las especificaciones del dispositivo sobre el que se va a programar el diseño, desde una interfaz gráfica. Esta herramienta es otra de las proporcionadas por Xilinx para agilizar el proceso de desarrollo de hardware.

La funcionalidad de IP Integrator está basada en la integración de diseños RTL, representados en un diagrama de bloques . Sobre el diagrama de bloques el usuario solo debe añadir componentes, configurarlos y conectar los distintos puertos de entrada y salida de cada componente donde corresponda.

Para comenzar un proyecto sobre Vivado IP Integratos simplemente se debe crear un proyecto indicando el dispositivo en el que se va a programar el diseño. Para ello las opciones del proyecto deben configurarse de la siguiente maneras:

-Board Vendor: em.avnet.com
-Library: Zynq
-Name: zed
-Version: d

De esta forma cuando se crea un nuevo diseño de bloques se puede instanciar desde el IP Catalog (repositorio de diseños RTL de Xilinx) un nuevo procesador de sistema*(ZYNQ7 Processing System).*

Como diseño de prueba se puede configurar el procesador de sistema añadido simplemente con los periféricos de los que dispone ZedBoard como el Ethernet, la tarjeta SD, el USB y un conector GPIO(General Port Input/Output). A partir de este diseño, sintetizamos y generamos un archivo programable *"bitstream"* que reservamos para generar una aplicación de arranque más adelante.

Dejando a un lado el diseño hardware, se procede a compilar el programa de arranque del sistema operativo y el propio kernel. Aunque previamente es importante explicar que es SDK.

SDK o Xilinx Software Development Kit es una herramienta orientada a la generación de aplicaciones standalone (ejecución sobre diseños hardware). Esta herramienta ofrece diversas herramientas para desarrollar programas adecuados a las plataformas hardwares que se han desarrollado. Una de sus características principales es que proporciona los drivers de los componentes de Xilinx añadidos a los diseños. Además de programas para ejecución sobre hardware permite el desarrollo de aplicaciones software para ejecutar sobre el procesador de sistema de Zynq. Estos programas pueden estar orientados como aplicaciones linux o simplemente para cargar en memoria.

Una vez aclarado que es SDK, para compilar el programa de arranque o "u-boot" se necesita descargar del respositorio de Github de Xilinx los archivos fuentes del programa.El enlace al repositorio web es:

- git://github.com/Xilinx/u-boot-xlnx.git

Además para configurar el programa será necesario un repositorio de SDK alojado en otro repositorio de Github. El enlace es:

- git://github.com/Xilinx/u-boot.gitbsp/u-boot_v0_00_x

Una forma rápida de obtener los repositorios anteriores simplemente usar el comando de linux *"gitclone"*.

Para configurar correctamente las propiedades de los archivos fuente y de compilación de los mismos, se debe exportar hardware del diseño implementado anteriormente y abrir SDK para crear un nuevo proyecto de tipo "Board Support Package" basado en el diseño exportado y extendiendo el tipo proyecto a u-boot(habiendo cargado previamente el repositorio descargado). En la creación del paquete u-boot en SDK se deben definir como propiedades los diferentes componentes que se toman como referencia para la ejecución del linux, como por ejemplo, la entrada y salida(UART), la memoria principal (controlador DDR) y la dirección donde se aloja este programa.

En la compilación de este proyecto se generarán una serie de scripts que se deben copiar y añadir al repositorio de archivos fuente.

- xparameters.h
- config.mk

Para compilar el programa de arranque únicamente falta:

- Dirigirnos al directorio de los archivos fuente y exportar el compilador cruzado "arm-xilinx-linux-gnueabi-".
- Configurar el programa de arranque para zynq => *$ make zynq_zed_config*.
- Compilar.

Después se obtiene un archivo ejecutable "u-boot.elf" que se reservara para incluir en la tarjeta SD de carga del sistema operativo.

Únicamente falta la compilación del kernel. Para ello hay que descargar del repositorio de Github de Xilinx:

- git://github.com/Xilinx/linux-xlnx.git

Y antes de compilar se debe indicar las herramientas usadas para compilar el "u-boot":

- $ export PATH=$PATH:[DIR]/u-boot-xlnx/tools/

Para modificar más rápidamente la configuración del kernel se puede tomar la plantilla de xilinx y después modificarla de acuerdo al diseño implementado.

- $ make ARCH=arm xilinx_zynq_defconfig.

Y para modificarla visualmente:

- $ make ARCH=arm xconfig

Que abrirá una interfaz donde podemos indicar los drivers que debe incluir nuestro kernel y diferentes opciones. Es en esta fase donde se podrá indicar que compile los drivers para los módulos AXI-Ethernet, DMA y UIO.

Para terminar la compilación del kernel, simplemente se compila con el siguiente comando:

- $ make ARCH=arm UIMAGE_LOADADRR=0x8000 uImage

Es importante fijar ese valor para la dirección en memoria para la carga del archivo, ya que si se cambia, el dispositivo no arrancará el sistema operativo.

Ya solo falta crear un sistema de archivos Linux. Sin embargo el proceso de crear un sistema de archivos requiere más tiempo del necesario para este proyecto, por eso se puede tomar uno ya creado orientado hacia dispositivos ARM del siguiente enlace:

- http://www.wiki.xilinx.com/Build+and+Modify+a+Rootfs

Como el sistema de archivos es genérico para cualquier dispositivo basado en ARM se debe añadir la cabecera del u-boot con el siguiente comando:

- $ mkimage -A arm -T ramdisk -C gzip -d arm_ramdisk.image.gz uramdisk.image.gz

Para probar y ejecutar el kernel compilado se debe guardar en la tarjeta SD los siguientes archivos:

- BOOT.bin
- devicetree.dtb
- uImage
- uramdisk.image.gz

De estos archivos falta por obtener el "BOOT.bin" y para ello simplemente se abre el proyecto Vivado con el diseño hardware y se exporta hacia SDK. En este momento se debe crear una aplicación de tipo FSBL(First Stage Boot Loader) que generará un archivo ".elf".

A través de las herramientas de SDK (Xilinx Tools) se puede crear una imagen de arranque de Zynq(Zynq Boot Image) que programara la FPGA desde la tarjeta SD. En esta Zynq Boot Image se deben incluir los siguientes archivos en el orden en que se disponen:

- zynq_fsbl.elf
- bitstream del diseño hardware
- u-boot.elf

Ahora ya se obtiene el archivo BOOT.bin y se puede probar el sistema operativo sobre Zynq.

4.2.2.- Conexión AXI-Ethernet y PS.

El objetivo principal de esta fase es juntar la parte de la lógica programable (PL) orientada a la red con las dos interfaces alojadas en el módulo FMC con el diseño básico que dispone de un procesador de sistema(PS) para la ejecución del sistema operativo empotrado. Además hay que adaptar el sistema operativo para que soporte los cambios de la lógica programable.

Para obtener el diseño mostrado en la figura 12 se debe crear un proyecto de Vivado y añadir a través de IP Integrator un nuevo diseño de bloques. Este diseño deberá contener los siguientes módulos:

- Zynq7 Processing System.
- Clock_wizard.
- AXI DMA x2.
- Tri Mode Ethernet MAC x 2.
- AXI Memory Interconnection x3.
    - processing_system7_0_axi_periph
    - axi_mem_intercon
    - axi_mem_intercon_1
- AXI GPIO.
- XConcat.

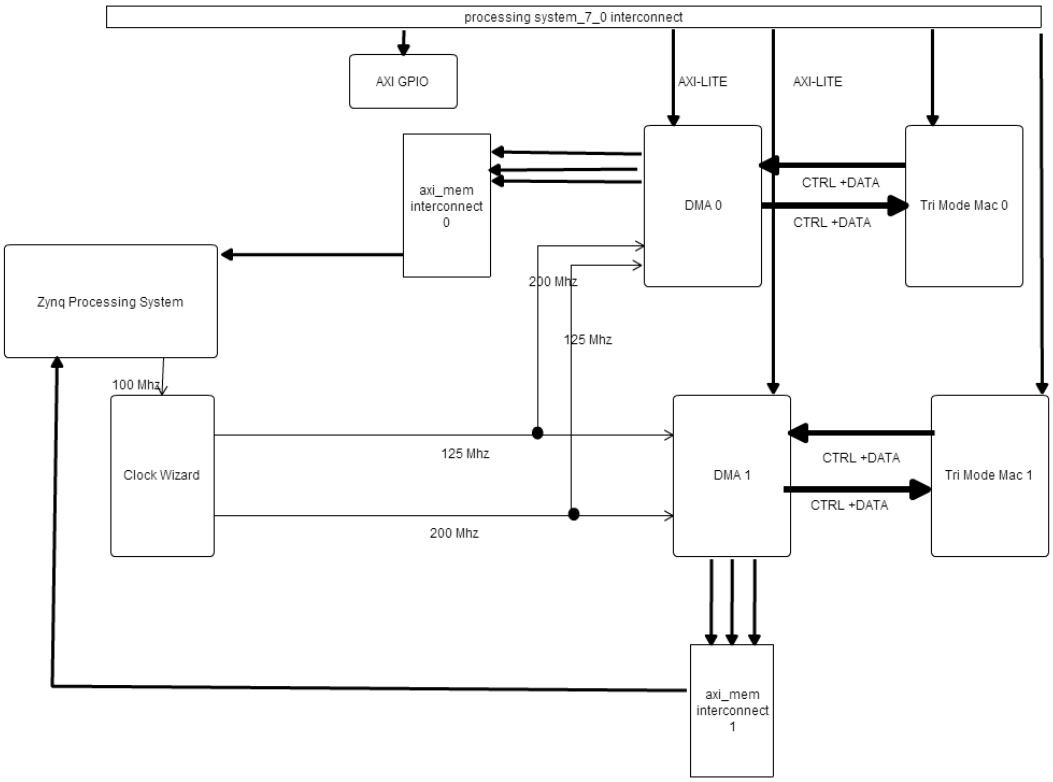


Figura 12.- Diseño Sin Lógica Añadida

En primer lugar se tiene que explicar cómo debe configurarse cada componente antes de conectarlo entre ellos.

En el caso de **Zynq7 Processing System** se trata de una configuración más extensa debido a la gran cantidad de periféricos de los que se disponen. En la configuración de PS-PL hay que destacar:

- UART0 y UART1 Baud Rate debe ser 1152000.
- Habilitar un reset de reloj: FCLK_RESET0_N.
- Habilitar interfaz M_AXI_GP0.
- Habilitar interfaces S_AXI_HP0 y S_AXI_HP1.

Los periféricos conectados al banco de conexiones 0 a 3.3V y los del banco de conexiones 1 a 1.8V. Los que se han habilitado son:

- Quad SPI Flash.
- Ethernet 0(MIO 16-27).
- USB 0(MIO 28-39).
- SD 0(MIO 40-45).
- UART 1(MIO 48-49).
- TTC0
- GPIO MIO.

La configuración de reloj debe ser:

- Processor/Memory Clocks: CPU = 666.667 MHz.
- Processor/Memory Clocks: DDR = 533.33 MHz.
- IO Peripheral Clocks:SMC = 100 MHz.
- IO Peripheral Clocks:QSPI = 200 MHz.
- IO Peripheral Clocks:ENET0 = 1000Mbps/125 Mhz.
- IO Peripheral Clocks:SDIO = 50 Mhz.
- PL Fabric Clocks: FCLK_CLK0 = 100 MHz.

Y se deben habilitar las interrupciones entre PS-PL:

- Interrupts/Fabric Interrupts(Enable)/PL-PS interrupts Ports: IRQ_F2P[15:0].

La configuración para los **DMAs** es:

- Enable Scatter Gather Engine.
- Enable Control / Status Stream.
- Width of Buffer Length Register = 14.
- Enable Read/Write Channel.
- Memoy Map Data Width = 32.
- Stream Data Width = 32.
- Max Burst Size = 256.
- Enable Unaligned Transfers.

El **Tri-Mode Ethernet** se debe configurar:

- Physical Interface: GMII (Enable Include IO Elements).
- FIFO & Checksum: -TX/RX Memory Size = 32.
  -No Checksum Offload.

En el **XConcat** se deben activar 8 puertos.

Y por último para el **Clock Wizard** se debe definir un reloj de entrada y dos de salida. Entrada de 100 Mhz y salidas a 125 y 200 MHz.

El **GPIO** no es usado durante el proyecto así que se puede optar por cualquier configuración o eliminarlo, pero en el proyecto se ha mantenido porque esta fase se ha implementado sobre la anterior. Los **AXI Memory Interconnection** dependen directamente de las conexiones del diseño.

Una vez que se dispone de todos los bloques configurados correctamente se puede proceder a conectarlos entre ellos. En primer lugar se debe definir todos los dominios de reloj de cada bloque. De una forma sencilla se conectaran todos las entradas de reloj que responda a una funcionalidad del protocolo AXI4 a la salida de reloj orientada al área programable del procesador de sistema Zynq con una frecuencia de 100 Mhz. También se conectara esta salida a la entrada del Clock Wizard para generar los otros dos dominios de reloj necesarios.

La salida del generador de reloj de 200 Mhz se utilizará únicamente para la entrada de los AXI-Ethernet llamada "ref_clk". El último dominio de reloj, que también proviene del Clock Wizard, tiene un valor de 125 Mhz y se conectara a la señal de los AXI-Ethernet denominada "gtx_clk".

Los AXI-Ethernet cuentan con otras entradas de reloj, pero todas forman parten de la interfaz externa del mismo, y por tanto esa señal de reloj provendrá desde el medio físico del dispositivo.

Los DMAs cuentan con dos interfaces de configuración AXI, una maestra y otra esclava. Los AXI-Ethernet cuentan también con una entrada de configuración AXI pero únicamente esclava. Además el bloque AXI-GPIO cuenta con otra entrada esclava de configuración. Ante la gran demanda de interfaces AXI4 el procesador de sistema requiere de los buses de memoria de interconexión mencionados anteriormente. Normalmente al incluirlos en el diseño IP Integrator da la posibilidad de conectarlos automáticamente.

Las conexiones entre entre AXI-Ethernet y DMA responden a las transmisiones de datos y información de control para su procesamiento. El bloque de red recolecta los bytes de datos que vienen desde las red y los transforma al protocolo AXI Stream para transmitirlos hacia el DMA. El DMA es el encargado de transmitirlos hacia el procesador del sistema convirtiendo los datos en formato AXI Stream a AXI4, ideado para escribir en memoria. El sistema ya puede procesar estos datos como quiera.

Entre todos los AXI-Ethernets y DMAs que se quieran incluir en el diseño deben conectarse de la siguiente forma:

- s_axi_txc => M_AXIS_CNTRL.
- s_axi_txd => M_AXIS_MM2S.
- m_axis_rxd => S_AXIS_S2MM.
- m_axis_rxs => S_AXIS_STS.

Para gestionar las interrupciones de todos los bloques, el procesador de sistema tiene habilitado un bus de interrupciones (IRQ_F2P), que por defecto debería permitir conectar directamente las señales individualmente, sin embargo la herramienta obliga a utilizar un bloque XConcat en el que conectar las interrupciones deseadas y genera un bus para el sistema.

Por último falta añadir un reset asíncrono para el sistema. Para ello se debe tomar la señal generada por el procesador de sistema para el área programable y conectarla a un generador de reset de sistema (Processor System Reset), que se utiliza para generar dos

señales de reset, una para todos los buses AXI de configuración y otra para todo lo demás.

A partir de este diseño se puede sintetizar y generar un bitstream para incluir en el fichero BOOT.bin, que programara la FPGA en el arranque del linux.

Antes de arrancar el sistema operativo en la placa, no basta solo con programarla, debemos comunicar al Linux la cantidad de interfaces de red y componentes de los que dispone. Para ello se debe modificar el devicetree de arranque y recompilar el kernel para que incluya todos los drivers necesarios.

La recompilación del kernel es sencilla, ya que como se explica anteriormente, solo se deben seguir los mismo pasos que antes, pero en la configuración se deben incluir los drivers AXI DMA y Xilinx AXI-Ethernet.

Modificando el devicetree

Para generar un devicetree a medida del diseño actual se debe exportar la plataforma hardware y arrancar la herramienta SDK. A través de esta herramienta se puede generar un fichero '.dts' para compilarlo y obtener el "devicetree.dtb" con los componentes que tiene el proyecto.

Para exportar dicho ".dts" es necesario añadir otro repositorio a la herramienta SDK. El repositorio se puede descargar del siguiente enlace:

- git://github.com/Xilinx/device-tree.gitbsp/device-tree_v0_00_x

Una vez añadido el repositorio, SDK permite crear un proyecto de tipo "Board Support Package" con extensión device-tree, que en su compilación generará un fichero, normalmente llamado "xilinx.dts", que contendrá un devicetree del hardware exportado.

Hay que asegurarse que el espacio de direcciones que ha incluido en este fichero ".dts" es el correcto. Además de comprobar que los argumentos input/output corresponden al puerto serie y que se han declarado las interfaces de red como puertos de entrada/salida:

```
aliases{
        ethernet0 = &ps7_ethernet_0;
        ethernet1 = &axi_ethernet_0_eth_buf;
        ethernet2 = &axi_ethernet_1_eth_buf;
        serial0 = &ps7_uart_1;
};
chosen{
        bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk";
        linux,stdout-path = "/amba@0/serial@e0001000";
};
```

Por último se debe comprobar que se han declarado los dos DMAs y los dos AXI-Ethernet conectados entre ellos. Además se debe definir el registro en el que se conectarán las interfaces físicas de red. Para evitar problemas definiremos todas hacia el registro de difusión 0x0, que localizara un registro disponible al que conectarlo. Un ejemplo:

```
phy2: phy@4{
        compatible = "marvell,88e1111";
        device_type = "ethernet-phy";
        reg = <0x0>;
};
```

4.3.-Estrategia Plataforma y Integración en el Kernel

El proyecto ya permite el procesamiento de paquetes de red en el área de lógica programable para su posterior transmisión hacia el sistema. Nuestro objetivo final será procesar esos paquetes antes de que lleguen hasta el procesador de sistema, reduciendo el trabajo que realiza el procesador y alcanzando el rendimiento esperado para este dispositivo.

Después de realizar esta fase se ha optado por explicar todo el proceso de desarrollo en dos fases para comprender las diferentes decisiones tomadas sobre el diseño final. Una primera en la que se opta por incluir un bloque IP exportado desde otro proyecto de Vivado, en el que se conectan a todos los streams que provienen de los DMAs y los Cores de red. En la segunda fase se opta por diseñar un modulo sobre Vivado HLS que permita el paso libre de paquetes de red en la dirección deseada.

4.3.1.Desarrollo y Integración de plataforma transparente al sistema

Para realizar el procesamiento de los paquetes de red antes de llegar al procesador de sistema(PS) es necesario integrar una plataforma entre los procesadores AXI-Ethernet y los AXI-DMA. En primer lugar la plataforma tendrá la única funcionalidad de pasar los paquetes de un lado a otro. La idea principal es que dicha plataforma resulte transparente al kernel de Linux que se ejecuta sobre el procesador de sistema, de forma que en el futuro se pueda ampliar la funcionalidad de dicha plataforma sin que el kernel tenga necesidad de conocer su estructura o funcionamiento.

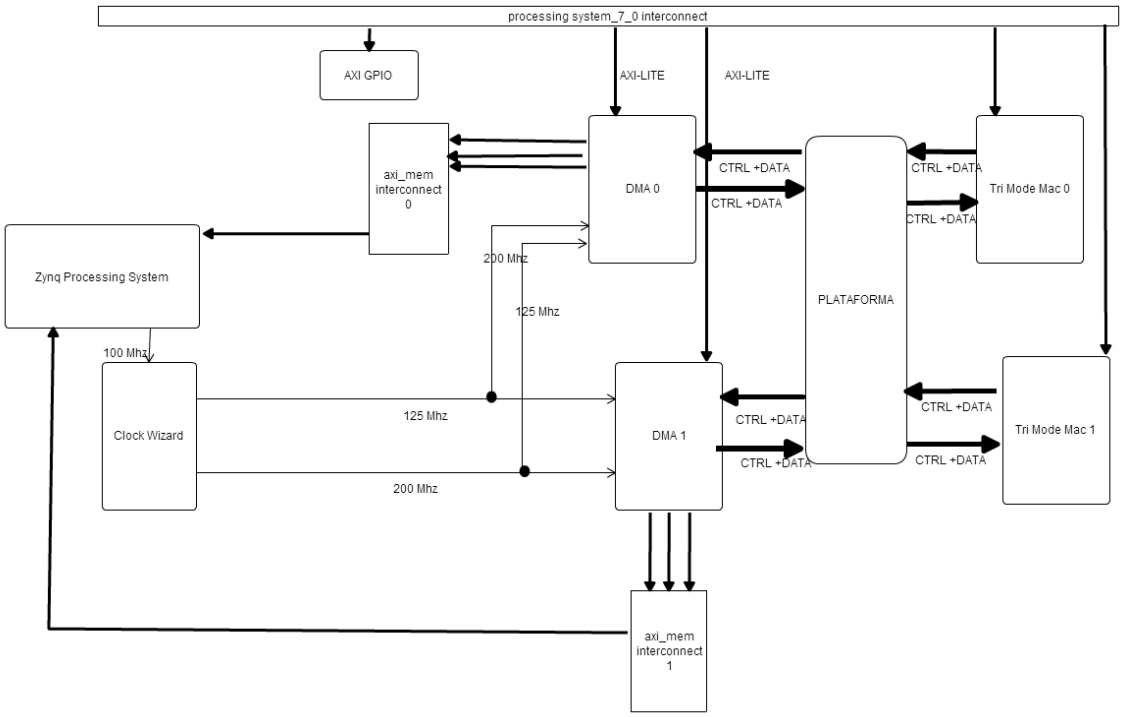


Figura 13.- Diseño Plataforma Transparente

Para realizar esto se ha implementado dicha tarea en un proyecto independiente de Vivado, de forma que una vez que el diseño ha sintetizado se puede exportar como bloque IP desde la herramienta para incluirlo dentro del diseño anterior a través de la herramienta IP Integrator.

La plataforma cuenta con:

- Entrada de reloj: aclk.
- Entrada de reset a nivel bajo: aresetn.
- Bus AXI-LITE para la configuración de la plataforma.
- 8 buses AXIStream esclavos
- 8 buses AXIStream maestros.

Se deben tener tantas buses AXI4-Stream ya que cada componente DMA o AXI-Ethernet posee una entrada de AXI4-Stream para datos junto otra entrada AXI4-Stream de control para los datos y lo mismo para la salida de datos, haciendo un total de 4 interfaces por componente.

Como la plataforma de momento no realiza ninguna funcionalidad que afecte al sistema no se han de indicar restricciones de tiempo sobre esta.

Hay que tener en cuenta que para exportar el proyecto como un IP Core es importante definir bien el control de versiones del bloque que se genere para su correcta integración en el diseño objetivo.

Integración

Una vez exportado el bloque IP de la plataforma es importante realizar tres tareas importantes:

1. Guardar una copia del proyecto con el sistema.
2. Añadir a uno de los proyectos el repositorio generado por la exportación de la plataforma a un IP Core.
3. Integrar el bloque IP de plataforma en el proyecto con el repositorio.

Un aspecto trivial pero importante para el desarrollo del diseño, es que las datos en la plataforma se reciben por los buses denominados esclavos("slaves") y se transmiten por los buses denomidos maestros ("masters"). Por regla general IP Integrator no permite conexiones maestro-maestro, ni tampoco esclavo-esclavo.

Por último se generará el bitstream del diseño y se debe exportar a una carpeta fácilmente localizable, antes de exportar nuestro diseño hardware a SDK.

El proceso normal para generar el soporte de arranque de linux sería exportar nuestro diseño a SDK, generar la aplicación FSBL y crear el fichero de arranque BOOT.bin a partir del fsbl.elf, el bistream generado anteriormente y el fichero u-boot.elf generado de la compilación del programa de arranque del kernel. Después se debería generar a través de la plataforma exportada un fichero '.dts' como se ha realizado anteriormente.

Sin embargo el diseño no cumple los requisitos del SDK ya que una interfaz AXI-Ethernet solo está permitido conectarla a una interfaz DMA o AXI-Fifo-Stream. Se puede indicar modificando los scripts “tcl" del Core AXI-Ethernet que ignore este suceso, pero el resultado lleva a que la funcionalidad del diseño no es capaz de configurar y levantar las interfaces físicas de red por falta de comunicación. Aparentemente esto debería ser abstracto al sistema, sin embargo el kernel conoce la existencia de la plataforma y los paquetes de negociación no alcanzan su objetivo, principalmente por el protocolo de transmisión del DMA.

La solución obtenida tiene que ver con la copia del proyecto que hemos conservado anteriormente. La idea es mantener el bistream del diseño con la plataforma para la programación del hardware, pero los archivos de arranque con el FSBL y el devicetree.dtb mantenemos los del anterior diseño sin plataforma entre las interfaces de red y el sistema. De esta forma el kernel se ejecuta sin conocer la existencia de la plataforma, y como consecuencia el DMA y el AXI-Ethernet creen estar en comunicación directa.

Las desventajas por las que este esquema de desarrollo no se ha considerado el apropiado para terminar el proyecto son:

1. No hay posibilidad de configurar dinámicamente los parámetros de configuración de la plataforma.
2. Difícil depuración de errores. Desde linux no se puede observar nada y desde la herramienta ChipScope es necesario ampliar las entradas y salidas de la plataforma para añadir un bloque ILA al diseño con el que comprobar el estado del componente.

4.3.2.-Desarrollo y Integración de bloque HLS reconocido por el sistema.

La siguiente estrategia propuesta y elegida para el proyecto, consiste en la implementación de bloque IP desde lenguaje de alto nivel en Vivado HLS. En la prueba de concepto realizada anteriormente se realizo una introducción al manejo de interfaces AXI4-Stream en Vivado HLS. A partir de este momento se va a recurrir a una librería de sencillas funciones orientadas al manejo de este tipo de señales. Gracias a esta librería se obtiene un código mas legible y con más potencia de computo en el hardware generado porque esta librería esta optimizada para conseguirlo. En el anexo A se puede apreciar la diferencia entre los códigos del clasificador para la prueba de concepto y los código finales que utilizan esta librería.

Como en el clasificador de la prueba de concepto sigue siendo necesario declarar en el proyecto la estructura para definir el tipo de datos AXI4-Stream. Ahora además es necesario incluir las siguientes líneas de cabecera:

- #include <hls_stream.h>
- using namespace hls;

Estas líneas permiten el uso de la librearía para streaming de datos, de forma que se manejen variable de tipo streaming como si de una FIFO se tratase. Además las variables de streaming encapsulan cualquier tipo de dato, que en este proyecto será el tipo definido por la estructura para AXI4-Stram. Para manejar las variables de streaming la interfaz dispone de las siguientes funciones:

- read / read_nb (lectura de variable stream bloqueante / no bloqueante).
- write / write_nb (escritura de variable stream bloqueante / no bloqueante)..
- empty? / full?(variable stream vacio/llena).

La función principal de este primer diseño permitirá el paso transparente de paquetes en una única dirección, por lo que solo dispondrá de dos parámetro entrada y dos parámetros de salida y todo ellos del tipo stream. No hay que olvidar que el objetivo principal de esta fase, es la integración de un módulo de procesamiento entremedias del diseño y que sea reconocido por el sistema operativo ejecutado en el procesador de sistema de Zynq. Para conseguir dicha funcionalidad el pseudo-código del módulo implementado sería:

Comprobar informacion control:
Si stream(entrada_control) is not empty then
    read(entrada_control, A)
    write(A, salida_control)
Comprobar datos:
Si stream(entrada_datos) is not empty then
    read(entrada_datos, B)
    write(B, salida_datosl)

Una vez desarrollado el componente deseado sobre la plataforma Vivado HLS, se debe sintetizar y exportar en forma de IP Catalog. De esta forma se obtiene un repositorio de archivos que contienen el propio Core del componente para trabajar desde IP Integrator,

el código Verilog del mismo, y una serie de recursos como drivers a nivel de aplicación para ejecutar aplicaciones con ese dispositivo.

Se pueden definir como drivers a nivel de aplicación, ya que se trata de un conjunto de funciones para arrancar, parar, obtener y mandar información al CORE desarrollado desde distintos escenarios, como por ejemplo Linux o ejecución en Standalone.

Para la integración del Core hay que seguir los siguientes pasos:

1. Añadir el repositorio a las propiedades del proyecto de Vivado con la plataforma de red.
2. Añadir el componente al diseño, conectarlo y sintetizarlo.
3. Exportar el bitstream y el diseño a SDK.
4. Generar un nuevo "BOOT.bin" desde otro proyecto sin el Core HLS pero con el bistream generado.
5. Modificar el devicetree para el arranque de linux.
6. Recompilar el kernel para incluir drivers de UIO(User Input/Output).

De los pasos anteriores se debe comentar lo siguiente:

- En la conexión del dispositivo se debe añadir una entrada más al XConcat para las interrupciones del mismo.

- Durante la configuración del kernel, antes de la configuración, en el menuconfig o xconfig debemos activar los siguientes atributos:
  - Device Drivers/Userspace I/O drivers/ Userspacce I/O platform driver
  - Device Drivers/Userspace I/O drivers/ Userspacce I/O platform driver with generic IRQ handling.

- La modificación del devicetree para el arranque de linux debe incluir:
  - Añadir el componente HLS a la zona del bus AMBA, declarándolo compatible con "generic-uio" para que incluya los drivers necesario durante la carga del sistema.

```
parser_top_0:parser-top@43c80000 {
        compatible = "generic-uio";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 36 4>;
        reg = <0x43c80000 0x10000>;
};
```

- Los dispositivos de red AXI-Ethernet y los AXI-DMA deben estar conectados entre sí, aunque en el diseño estén conectados al dispositivo generado de HLS, ya que en el instante de carga del sistema operativo, los drivers de red solo permiten la conexión con DMA o AXI-FIFO.

- Asegurar que el registro del componente físico ethernet este definido como la dirección *<0x0>*, para que tome el registro disponible y evitar colisiones entre las tres interfaces. Por ejemplo:

```
phy2: phy@4{
        compatible = "marvell,88e1111";
        device_type = "ethernet-phy";
```

```
                reg = <0x0>;
        };
```

Una vez realizado estos pasos desde SDK se puede generar uno o varios programas para ejecutar en el Linux antes de poner el diseño a trabajar. Para ello simplemente se deberá crear un proyecto de tipo aplicación con objetivo linux y añadir a la carpeta de recursos los ficheros con las funciones generadas por el Vivado HLS. Por último se escribe un pequeño programa principal donde inicializar el dispositivo y sobre todo configurarlo como auto reiniciable para que nunca para de ejecutarse.

Los ficheros *".elf"* generados por estos proyectos se puenden traspasar al linux en ejecución a través de una conexión SSH desde SDK. Sin embargo resulta más práctico copiarlos en el tarjeta SD que se inserta en la Zynq, y acceder desde el directorio *"/mnt"* de linux para ejecutarlos.

4.4.-Desarrollo de los módulos hardware.

A partir de los pasos anteriores se ha logrado una plataforma de desarrollo estable, en la que se va a proceder a integrar cambios de una forma progresiva hasta conseguir los objetivos del proyecto.

Como se ha descrito a lo largo del documento este proyecto se trata de una sonda de monitorización de red activa, y para que los módulos desarrollados cumplan con la funcionalidad de una herramienta de este estilo, se tienen que desarrollar dos diseños diferentes, aunque basados en la plataforma hardware desarrollada hasta ahora.

Por un lado, y tomando como precedente la prueba de concepto, uno de los diseños se encargará de la clasificación de paquetes, devolviendo los paquetes que cumplan los criterios a la red. Por el otro lado, el otro diseño se encargara de inyectar paquetes a la red para que sean recogidos en otro lado, como por ejemplo, por el diseño anterior.

4.4.1.- Desarrollo del módulo de clasificación de paquetes.

Como en el caso del último diseño de lectura de interfaces AXI-Stream en Vivado HLS, se recurrirá al uso de las librerías para streaming de datos. Aunque para desarrollar este módulo, se ha cambiado la filosofía de lectura de datos, ya que el objetivo es leer las entradas, almacenar la información para su procesamiento, y escribirla en la salida. Para realizar esta tarea se debe recurrir a la lectura en bucles.

Durante el desarrollo de este módulo se encontraron diversos problemas, de forma que para conseguir un diseño optimo para el objetivo del proyecto se realizaron diferentes pruebas. A lo largo de este apartado se explican la evolución de estos diseños de prueba para comprender la problemática del desarrollo y comprender las decisiones tomadas sobre el diseño final. Se pueden definir los tres modelos de prueba como desarrollo de un único bloque de procesamiento, bloques de lectura bloqueantes, y conversión a un solo stream.

Etapa 1: Desarrollo de un único bloque

Durante esta etapa el objetivo consistía en la implementación de único bloque HLS que se encargue de la lectura de los buses AXI4-Stream de datos y control de los dos AXI-Ethernet. Esto implica la lectura de 4 entradas de AXI4-Stream, la escritura de otras 4 salidas de AXI4-Stream, y todas ellas con sus respectivas señales de control, es decir, un total de 16 interfaces AXI4-Stream para un único bloque HLS, además de una señal AXI4-Lite para la configuración del módulo. El resultado debería ser similar a la estructura de la figura 14

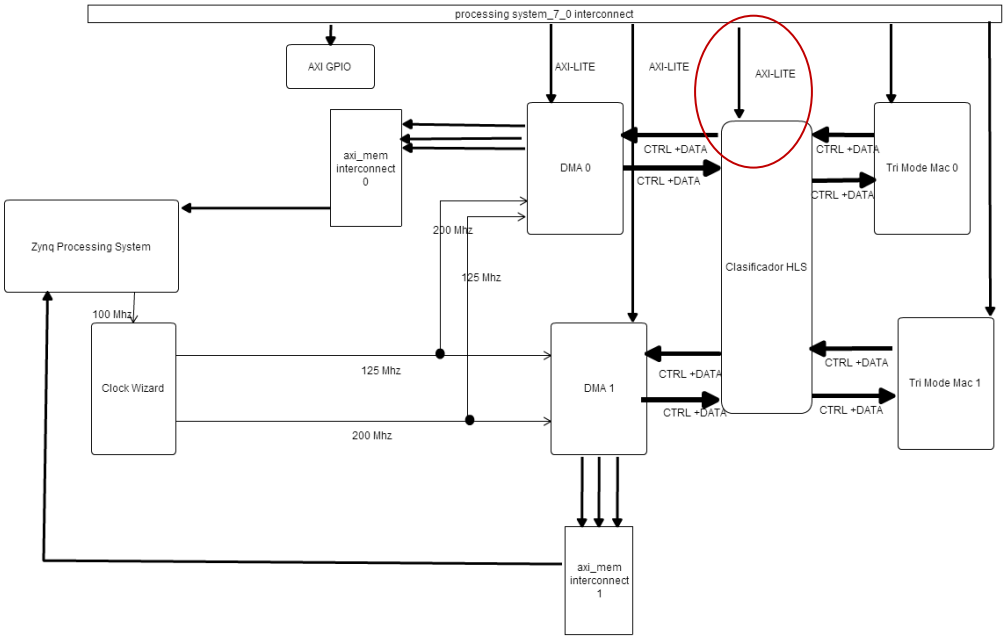


Figura 14.- Diseño 1 solo bloque HLS

Para el desarrollo de este bloque HLS se debe tener en cuenta que incluir un bucle en Vivado HLS implica que el procesamiento en hardware se bloquea hasta que la condición de parada del bucle se cumple. Si los Cores de red se encuentran conectados a este módulo, y estamos esperando leer de uno, el resto de las entradas del dispositivo quedan bloqueadas, por lo que la otra interfaz resultaría inútil y todos los datos de entrada o salida que lleguen a esta se perderían.

Obviamente si situáramos FIFO's delante de las entradas y salidas del módulo los datos quedaría almacenados en estas FIFO's para su posterior lectura, pero en el caso de recibir o mandar ráfagas de datos con un tamaño mayor al habitual produciría un retardo en los datos causando pérdidas o datos desactualizados. La solución correcta a esta situación se trata de lecturas no bloqueantes, permitidas por la librería para streaming por las funciones mencionadas antes.

En este momento el bloque HLS dispone de todas las entradas y salidas, la capacidad de leer o no una entrada hasta que llegan todos los datos, y obviamente es conveniente añadir las FIFO's por cualquier tipo de retardo en el procesamiento. Aparentemente es

un diseño estable, sin embargo la gran cantidad de entradas es difícil de manejar y sobre todo la sincronización de las entradas de control y de datos. Durante las pruebas de este diseño son constantes los fallos de recepción y transmisión por parte del AXI-DMA al encontrar palabras incoherentes de control para los datos recibidos.

Etapa2: Lecturas bloqueantes.

Ante los problemas manifestados en la etapa anterior como la gran cantidad de señales, y la mala sincronización entre datos y control. Se decide tomar la lectura en bucle desde otro punto de vista.

En primer lugar, hay que entender cómo se transmiten los datos entre el AXI-Ethernet y el AXI-DMA [20, 21]. En el caso de la recepción de datos desde la red hacia el sistema, la interfaz AXI-Stream Rx, una vez recibida toda la información de datos Ethernet, envía toda la información de control primero y posteriormente los datos. En el caso de la transmisión de sistema hacia red, la interfaz AXI-Stream Tx, puede enviar los datos y el control en cualquier orden, pero nunca al mismo tiempo, aunque sin embargo en la práctica envía siempre primero toda la información de control. Es importante añadir que la información de control está compuesta por una constante 6 palabras de 32 bits, mientras que los datos pueden variar desde 4 bytes hasta 9 Kbytes.

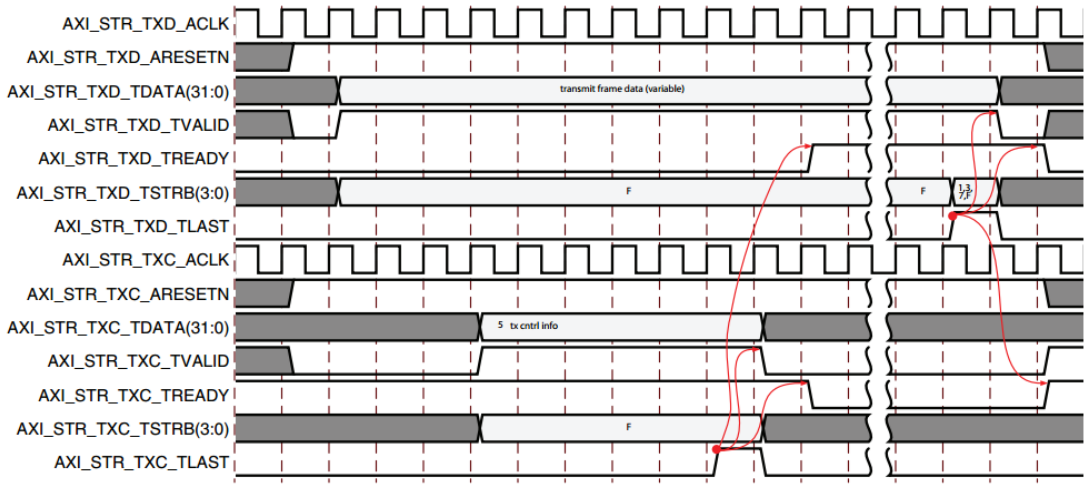


Figura 15-Transmisión desde AXI-ethernet [38]

Teniendo en cuenta esta información uno de los objetivos de esta etapa es dividir el componente HLS en bloques más pequeños. Estos bloques cumplen todos la misma funcionalidad de leer una entrada, almacenar en buffer interno y mandar. A diferencia de la etapa anterior, al dividir en bloques no es necesario realizar lecturas no bloqueantes, ya que todas las entradas se leen independientemente del estado de las demás. Esto permite asegurar la coherencia y sincronización entre datos y palabras de control, de forma que se obliga a leer 6x32 bits palabras de control y un dato por ciclo hasta llegar a la señal “last” de la ráfaga. Además simplifica la producción de código, reduciendo el número de líneas y variables que se necesitan.

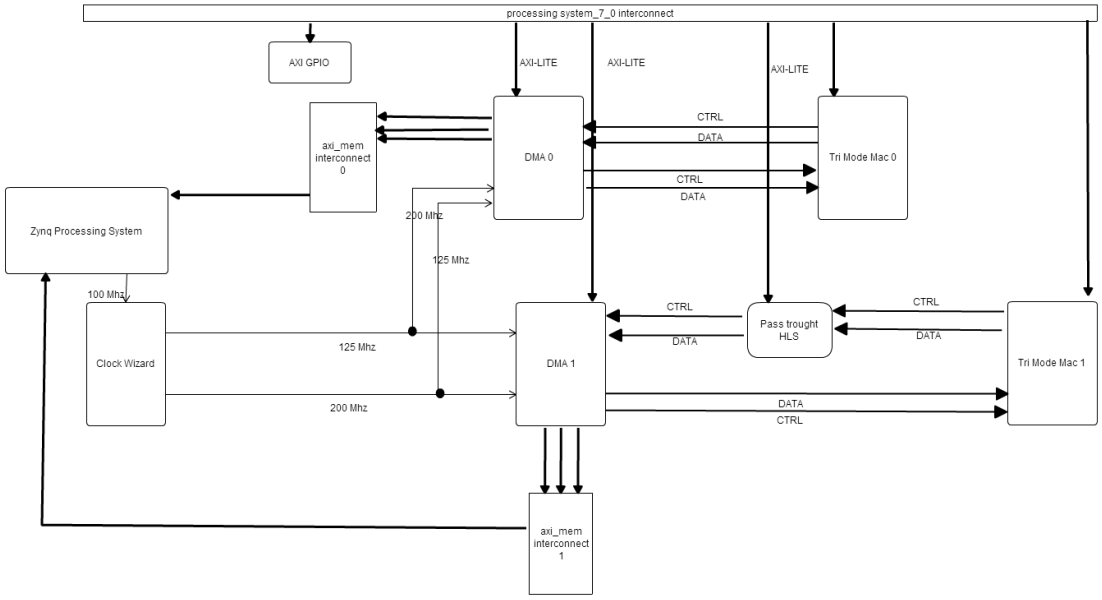


Figura 16.- Diseño PassTrought 2 AXI4-Stream

Con este diseño se consigue la transmisión y recepción de datos sin errores. Actualmente se dispone del módulo HLS que permite la transmisión transparente de datos, realizando alguna operación sobre ellos. A partir de este momento simplemente consiste en ampliar la funcionalidad.

Etapa 3: Conversión a un solo stream

A pesar de obtener un diseño estable con la etapa anterior, se ha considerado interesante, realizar dos pequeños módulos para el manejo de los dos streams de control y datos como uno solo, evitando así problemas de sincronización o perdidas de datos entre control y datos de cara a la red y el sistema.

La idea principal consiste en realizar la lectura en un módulo, que en este proyecto se ha denominado Mixer (Mezclador), encargado de realizar la misma lectura bloqueante que en la etapa anterior, a diferencia que no almacena esos bytes leídos sino que lo transmite todo directamente hacia una única salida(control y datos).

Por otro lado los componentes AXI-Ethernet y AXI DMA siguen necesitando dos señales independientes, por lo que es necesario implementar otro módulo, que en este proyecto se ha denominado Splitter(Separador), encargado de leer de una única entrada 6 palabras y transmitirlas hacia una salida y continuar leyendo de la misma entrada hasta encontrar el final del streaming y enviarlo por otra salida. De esta forma se recupera el flujo de datos esperados por los Cores del diseño.

Para el uso de estos nuevos bloques en el diseño propuesto no es necesario incluirlos en el device-tree, ni la implementación de drivers para la ejecución del kernel, ya que debido a su sencilla funcionalidad, no es necesario que dispongan de interfaz AXI4-Lite de configuración y por lo tanto solo es necesario mantener una constante activa en el diseño de bloques para la señal que generan en su exportación “ap_start”, que indica al bloque que esta “encendido”.

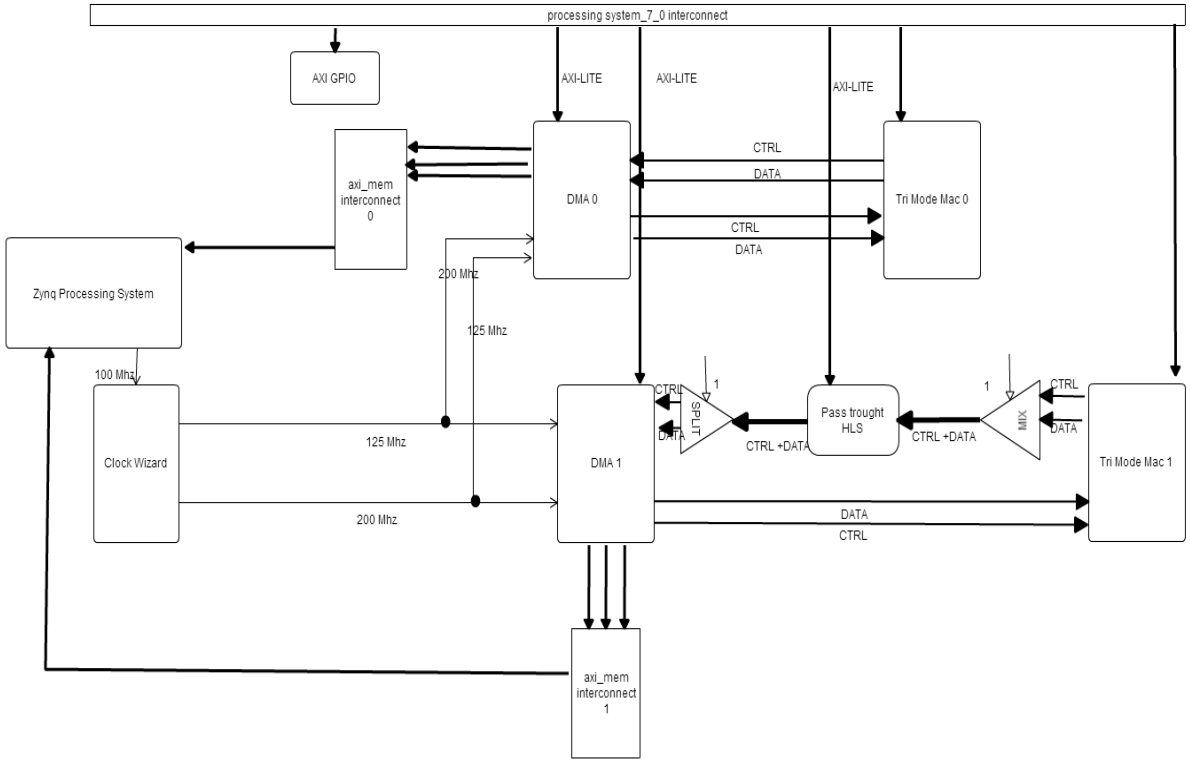


Figura 17.- Diseño PassThrougth y un único bus AXI Stream

A pesar de que estos bloques son transparentes, sigue siendo necesario un bloque de procesamiento, en el que se realicen las tareas deseadas. Este bloque sí que posee una interfaz AXI4-Lite de configuración y por tanto el kernel debe dispone de su información (mediante los procesos explicados anteriormente). En este bloque de procesamiento, se realiza la lectura bloqueante de 6 palabras de la entrada y de la misma entrada el resto de datos, se almacenan en variables internas y se procesan con la finalidad deseada. Una vez procesados los datos, simplemente se transmite por un única salida tanto las 6 primeras palabras como el resto de datos, manteniendo así la existencia de un único stream.

Diseño final del clasificador

Una vez se dispone del bloque de procesamiento con un paso transparente de paquetes, se debe incluir toda la lógica necesaria para obtener la funcionalidad de una sonda de red. Hasta ahora se ha hablado de una forma general del desarrollo de los módulos, intentado conseguir una funcionalidad sencilla en cualquier dirección del streaming y en cualquier interfaz de red. Sin embargo para el proyecto se deberá disponer del diseño mostrado en la figura 18.

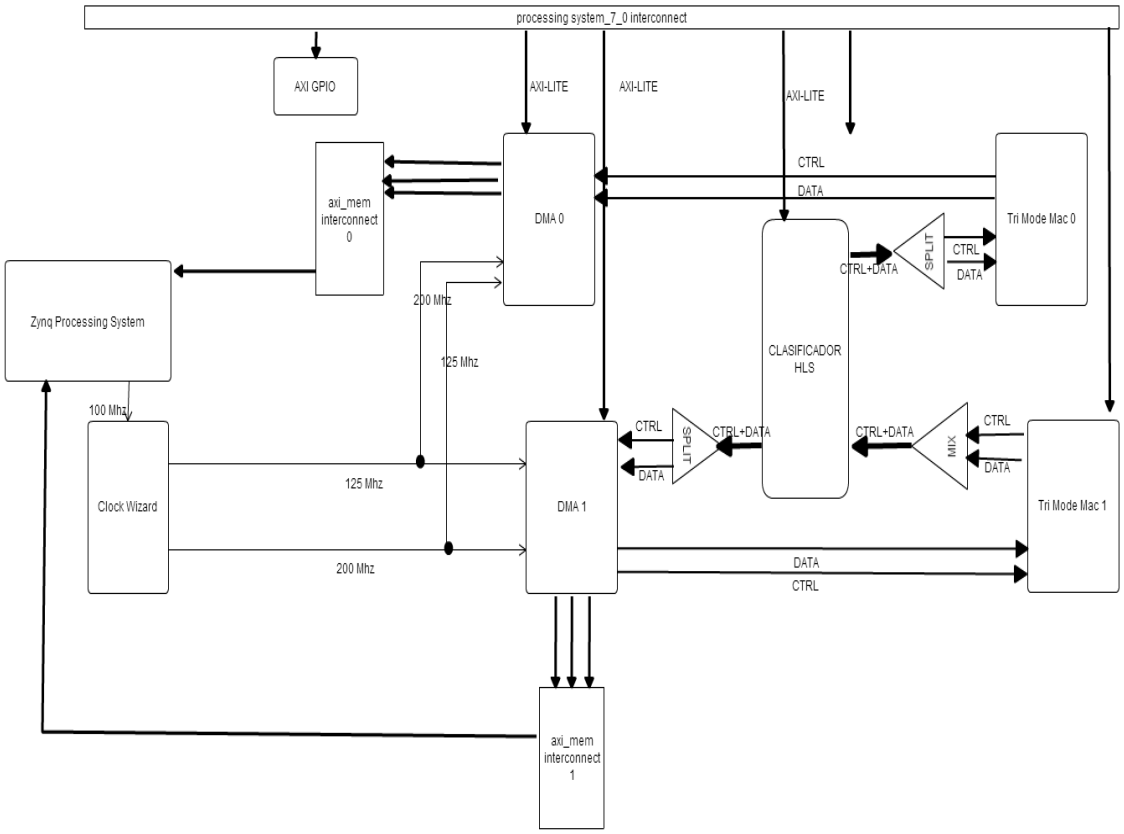


Figura 18.- Diseño Clasificador

De la figura 18 hay que destacar la eliminación de la entrada del segundo DMA para recibir datos, ya que todos los paquetes que salgan hacia la red por la segunda interfaz serán aquellos que son clasificados por el Core HLS implementado. Se debe insistir en eliminar la entrada de datos del DMA, ya que si simplemente desconectamos la señal de control y datos el resto de señales que quedan conectadas pueden interferir en el funcionamiento del sistema mediante interrupciones o resets.

Aunque se ha eliminado la señal que tenia la segunda interfaz de red para transmitir paquetes, esta no queda desconectada, ya que tendrá conectado un Spliter que proviene de una salida alternativa del Core HLS.

En la primera interfaz de red la conexiones son las esperadas de las etapas anteriores de desarrollo con un Mixer, un Core HLS, y un Splitter para transmitir hacia el procesador de sistema.

Una vez se dispone del diseño de la figura es importante comprender como se ha desarrollado el clasificador de paquetes. Este clasificador de paquetes se encarga de leer la entrada de datos hasta encontrar el final del streaming. Una vez que dispone del paquete completo separa el flujo de bytes en dos buffers, uno para el control de 6 palabras y otro para los datos de red. Sobre este segundo buffer, que contiene un paquete de red con la estructura mostrada en la figura 19 realiza la siguiente operación:

Si buffer_datos(4) contiene 0x0800(IP) then

Si buffer_datos(4) contiene IP v4 then

Si Si buffer_datos(6) contiene protocolo_AXI4_Lite then

Si buffer_datos(9) contiene puerto_AXI4_Lite then

- Intercambiar direcciones fuente y destino(MAC y IP)

- Intercambiar puertos origen y destino

- salida = 2

else

- salida = 1

Durante las pruebas del proyecto el protocolo configurado a través de la interfaz AXI4-Lite ha sido UDP (0x11).

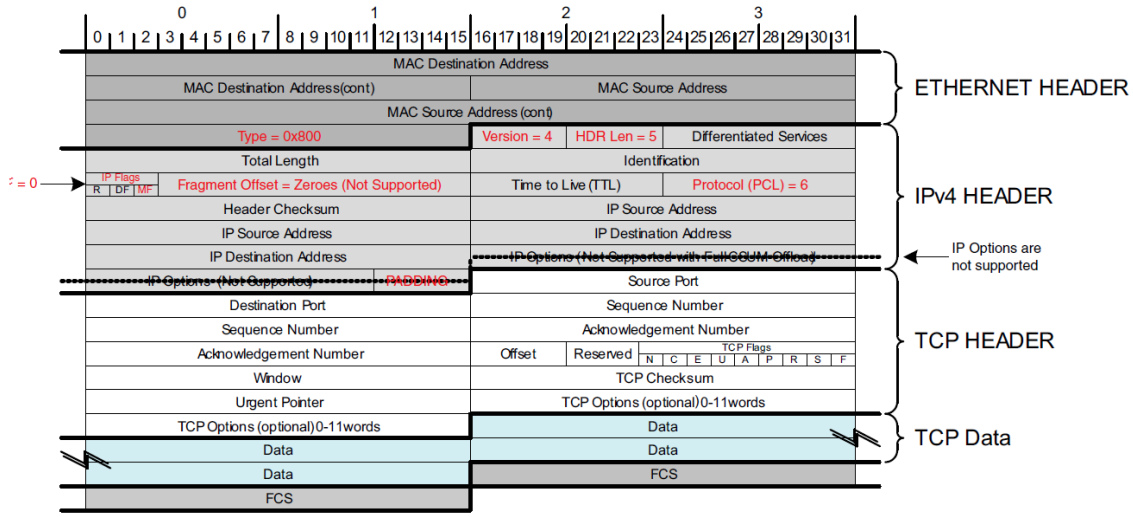


Figura 19.- Paquete Ethernet + IP + TCP [39]

Para analizar y dar la vuelta a los campos en el paso 4 es muy importante tener en cuenta que las interfaces de red están conectadas hacia un procesador ARM y por tanto su codificación de bytes se encuentra en BIG ENDIAN, por lo tanto cualquier campo va a tener los bytes invertidos.

Mientras el paquete es procesado se ha activado o no una bandera para encaminar el paquete en una dirección. Si la bandera esta activa (salida == 2) debe ser retransmitido por ambas salidas del Clasificador. Si no está activa (salida == 1) simplemente se escribe en dirección hacia el procesador sistema. Obviamente para que el sistema o la red puedan trabajar este paquete se debe retransmitir en la misma salida de datos las palabras de control y siempre antes que los datos, para que puedan ser tratados por los Splitters o Cores que pueda haber detrás.

El programa necesario el arranque del módulo en el entorno del sistema operativo debe encargarse de configurarlo como auto reiniciable, además de obligar al usuario a indicar por parámetros el protocolo y el puerto por el que se desean clasificar los paquetes.

Con este procesamiento se dispondría de una parte de la sonda de monitorización que se desea desarrollar. Obviamente falta una parte del proyecto que genere los paquetes de origen que se desean clasificar y poder calcular las latencias que se obtienen de los posibles trenes de paquetes entrantes.

4.4.2.- Desarrollo del inyector de paquetes.

Para el desarrollo del inyector de paquetes, al igual que para el clasificador, se parte del diseño estable en el que los streams de datos y control convergen en un solo bus. En este módulo solo nos interesa la transmisión por una de las interfaces del dispositivo, por lo que se puede de la otra interfaz para trabajar libremente. El diseño que debe ser similar al de la figura 20.

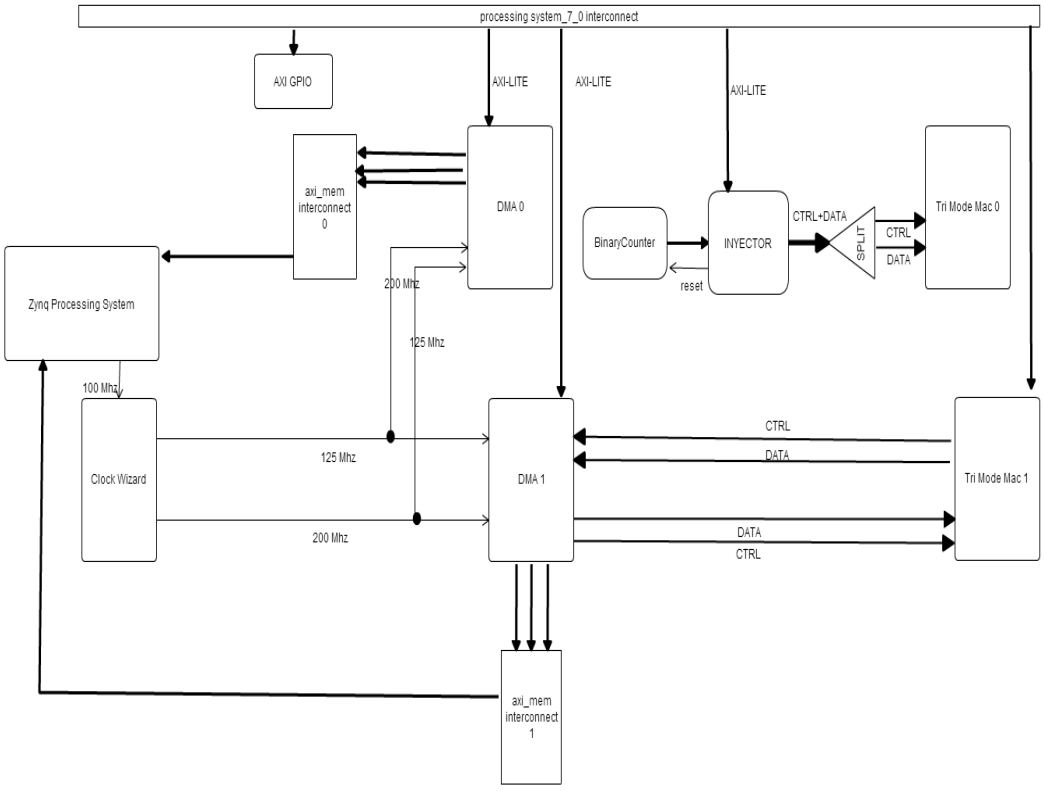


Figura 20.- Diseño Inyector Paquetes

Además de la figura 20 hay que destacar varios detalles. En primer lugar es importante desconectar la recepción de paquetes en la interfaz para emitirlos, si no se hiciera esto, el diseño provocaría errores continuos por la fase de negociación del DMA. Por otro lado, aunque se explicará más tarde, hay que añadir un componente más, un contador binario conectado a la misma señal de reloj que el inyector de paquetes. El objetivo de

este contador binario es ir contando la cantidad de ciclos de reloj transcurridos durante la emisión de los paquetes.

Partiendo del diseño explicado, para desarrollar el generador de tráfico sobre Vivado HLS se ha realizado de la siguiente forma. Primero es definir las entradas y salidas del inyector. El inyector dispondrá de las siguientes señales de 32 bits para su configuración a través de un bus AXI4-LITE:

- Source MAC
- Dest MAC
- Source IP
- Dest IP
- Source Port
- Dest Port
- Tiempo entre paquetes.
- Número de paquetes a enviar

También dispondrá de otra entrada de 32 bits no asociada a un bus AXI4-Lite por la que recibirá el número de ciclos contados por el contador binario.

En el caso de las salidas son dos, un bus AXI4-Stream para la salida de los datos y el control, y otra señal de un bit, para reiniciar el contador.

Teniendo claras las entradas y salidas del generador de paquetes hay que entender cómo funciona. Para este proyecto básicamente se necesita rellenar un buffer de datos con la información de un paquete típico de red, aunque en primer lugar se debe rellenar otro buffer con la información de control coherente para la transmisión.

En el buffer de 6 palabras de 32 bits de control deben almacenar las palabras con las estructura mostrada en la figura 21. La bandera de la primera palabra indica el tipo de transmisión, siendo habitualmente la palabra A que indica tipo de transmisión normal. El resto de palabras se usan para el cálculo de checksum TCP/UDP en hardware, sin embargo para el proyecto no se han utilizado.

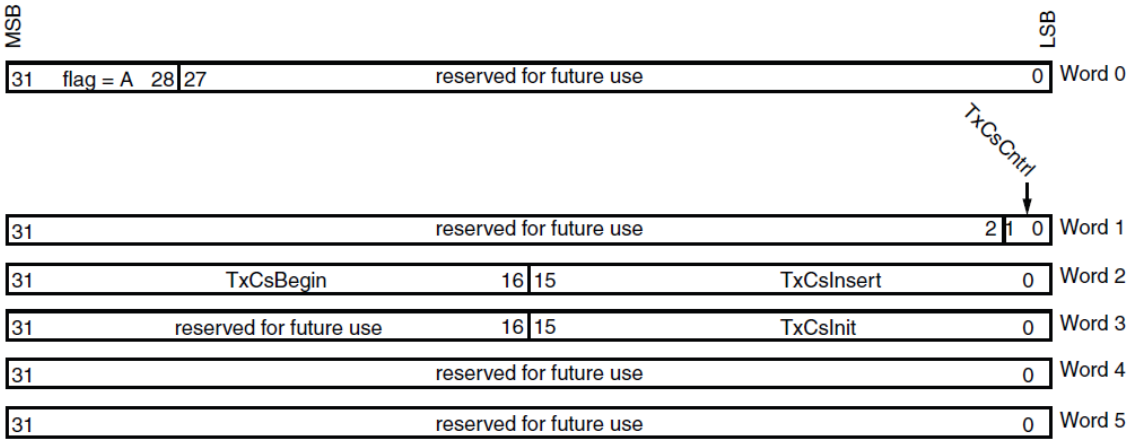


Figura 21.- Palabras de control AXI4-Stream de transmisión [40]

En el buffer de los datos se almacenará siempre la misma palabra pero los datos estarán compuestos por las entradas definidas en el AXI4-Lite explicadas anteriormente(sin el tiempo entre paquetes, ni la cantidad de paquetes). Únicamente hay que añadir algunos datos típicos de un paquete IP como la versión y el protocolo. Por último hay que crear el checksum de la cabecera IP. La operación es bastante sencilla, ya que se dispone de la información en bloques de 32 bits, de esta forma solo se deben desplazar los bits de cada palabra 16 posiciones o multiplicar por una máscara de bits con la que sean válidos los últimos 16. En el momento en que se van ajustando los datos en bloques de 16 bits se van sumando sobre una variable, y para terminar se realiza la operación complemento a 1 con el operador '~'. Con esto se tendría los paquetes listos para el envío.

Una cosa que no se ha comentado es tener en cuenta el overflow de la suma de datos de 16 bits. Para solucionar este problema la operación se realiza sobre variables de 32 bits y antes de realizar el complemento a 1 basta con sumar la parte alta con la baja de la variable sobre la que se ha ido realizando la suma.

Otro aspecto que hay que tener en cuenta durante el rellanado del buffer, tanto de control como de datos, es que los datos son transmitidos hacia el LogiCore de AXI-Ethernet, por lo que como en el caso del clasificador la información está tratada en Big Endian. El gran inconveniente de este hecho es que los datos recibidos a través del AXI4-Lite deben ser volteados, pero basta con desplazar los bits en los dos sentidos en función de la información útil que contengan las variables usadas para leer estos parámetros.

Del desarrollo del inyector de paquetes solo queda por comentar el envió de estos. Como en casos anteriores, para transmitir el paquetes se utilizan dos bucles que escriben sobre la salida AXI4-Stream, uno para transmitir las 6 palabras de control y otro para el resto de datos de red.

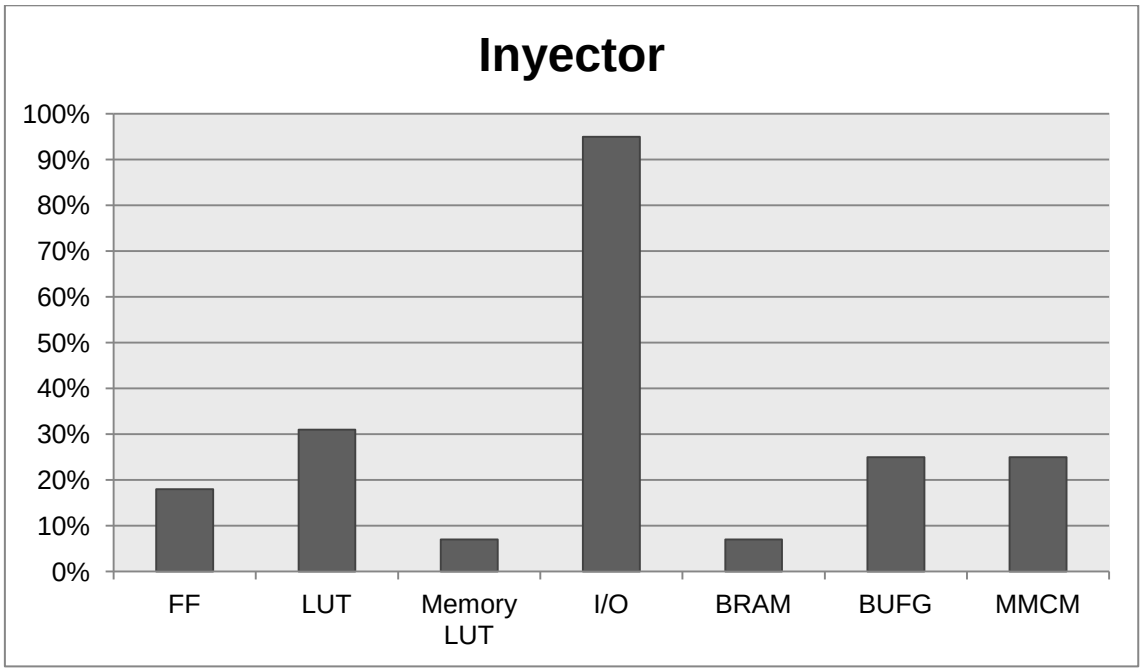
El objetivo de este componente es la generación de trenes de paquetes, por lo que se desea enviar una cantidad de paquetes determinada con un intervalo de tiempo en la transmisión entre un paquete y otro. Para realizar esta tarea el envío de un paquete debe encerrarse en un bucle con el límite definido por el parámetro del bus de configuración. Por otro lado para habilitar el envío de paquetes, el contador de ciclos externo debe haber contador una cantidad determinada de ciclos de reloj mayor al tiempo entre paquetes recibido por parámetros(su valor también son ciclos de reloj necesarios). Para controlar toda esta información, dentro del bucle se añade un bloque condicional en el que se comparan ciclos contados con el límite, y solo si supera el límite se incrementa la variable del bucle, se envía el paquete y se reinicia el contador de ciclos externos.

En el caso del programa de arranque de este módulo no se debe indicar que es auto-reiniciable porque sino el programa mandaría paquetes a la red indefinidamente y no es un comportamiento deseado. Además debe obligar al usuario a indicar por argumento del programa todos los parámetros que se deben configurar en el AXI4-Lite como datos de entrada.

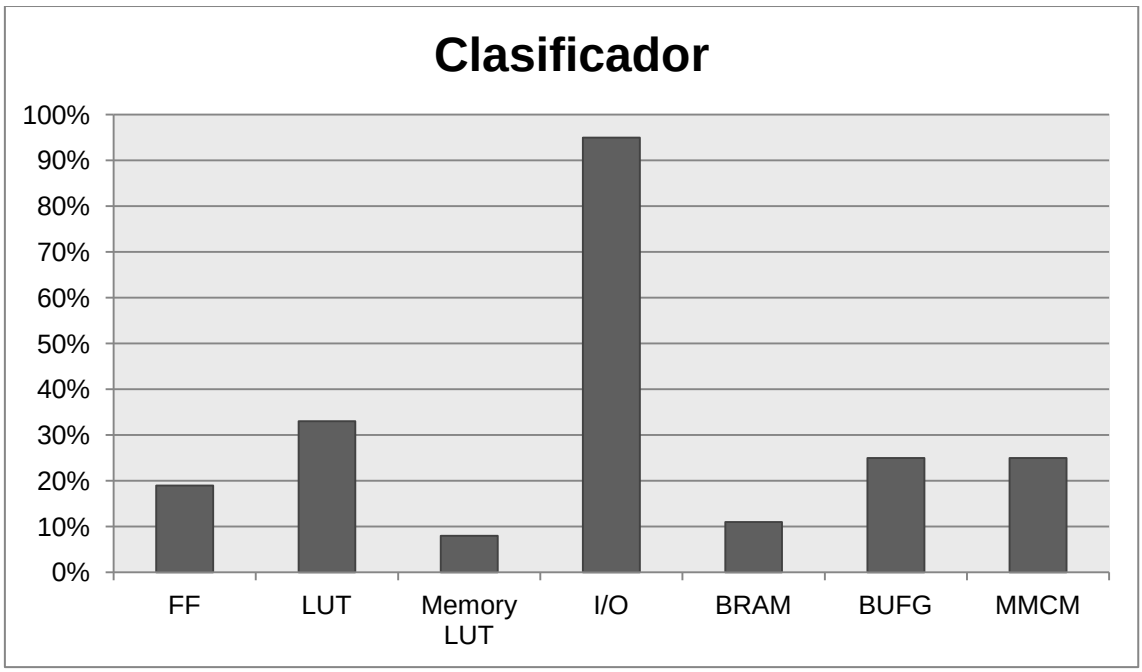
Con se dispondría de los diseños necesarios para completar el desarrollo del proyecto, y solo deberían realizarse pruebas para comprobar que su funcionamiento es correcto.

4.5.- Ocupación de los diseños.

Una vez que los diseños han sido desarrollados hay que tener en cuenta el espacio o celdas programables que han sido ocupadas en la FPGA del dispositivo Zynq. En las gráficas 1 y 2 se pueden observar los porcentajes de ocupación de ambos diseños.



Gráfica 1.- Ocupación Inyector de Paquetes en Zynq



Gráfica 2.- Ocupación Clasificador de Paquetes en Zynq.

Los datos de ocupación de estos dos diseños básicamente muestran la cantidad de componentes de bajo nivel instanciados como por ejemplo Flips-Flops, tablas Look-UP o bancos de memoria, necesarios para que la FGPA se comporte como en los diseños deseados.

Ambos diseños presentan una características similares y uno de los datos más llamativos es el alto grado ocupación de la entrada/salida del dispositivo. La entrada y salida del dispositivo es tan alta debido a la cantidad de pines físicos ocupados en el conector FMC de la placa ZedBoard y los de la interfaz de red integrada en el procesador de sistema.

A pesar del alto grado de ocupación de la entrada y salida es una característica que no debería aumentar si se modifica la lógica interna del procesamiento de paquetes, permitiendo optimizar y mejorar los diseños implementados.

5.- Banco de Pruebas

Para probar los diseños desarrollados se han realizado pruebas en dos tipos de entornos para cada diseño. Por un lado se han probado en un entorno controlado y por otro en un entorno real.

El entorno controlado esta formado por tres dispositivos, la plataforma ZedBoard, un conmutador de red y un ordenador. El ordenador y la ZedBoard están conectados a través del conmutador, encargado de dirigir el tráfico entre las interfaces de ambos dispositivos. La interfaces de red quedan aisladas en una red privada con direcciones IP del rango 10.0.0.0/32 .

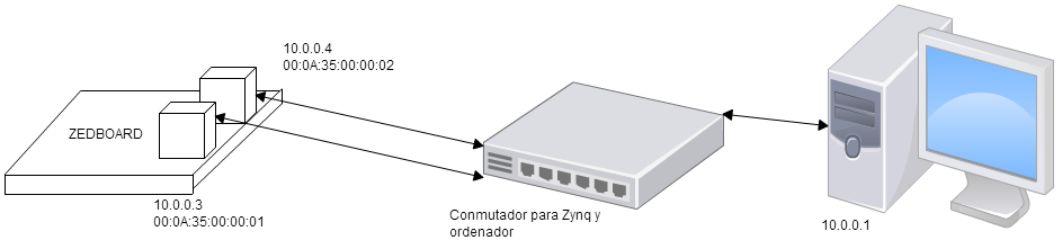


Figura 22-Entorno controlado

El entorno real es parte de la red del laboratorio del grupo de investigación HPCN. Este laboratorio de la Universidad Autónoma de Madrid dispone de un red privada y un router de enlace a la red pública de la universidad. Así de esta forma se conectará el ordenador encargado de capturar el tráfico a la red pública de la universidad y por otro lado se conectará y configurará la plataforma ZedBoard en la red privada del laboratorio.

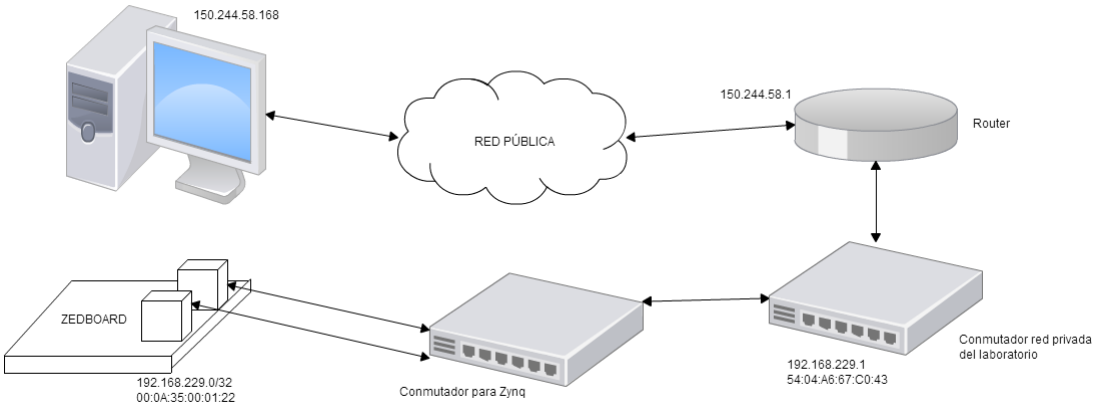


Figura 23.- Entorno Real

5.1.-Generador de paquetes

5.1.1.-Pruebas en el entorno controlado

En el entorno controlado se ha probado a ejecutar el módulo de inyección de paquetes con la finalidad de verificar que los paquetes se generan correctamente y que los trenes de paquetes se envían con el intervalo indicado en el programa de ejecución. A través de este módulo se puede medir el retardo de los paquetes enviados en un sentido.

Para verificar que los paquetes llegan hasta la máquina de desarrollo se ha utilizado el programa TCPDump en modo promiscuo sobre la interfaz de red conectada al conmutador de red del ordenador.

Al programa que arranca el generador de tráfico se le indica que genere 1000 paquetes con diferentes intervalos y con direcciones de red apropiadas. Como la captura de paquetes se realiza en modo promiscuo se utiliza el puerto 2556 como ejemplo.

Para medir el intervalo medio de llegadas de paquetes de red se han realizado dos pequeños programas en lenguaje C. El primer programa recurre a las librerías pcap para el tratamiento de tráfico o ficheros con captura de tráfico. Este programa se encarga de leer el fichero en el que se almaceno la captura de los paquetes inyectados y tomar de cada paquete el timestamp asociado a la recepción, además de contar la cantidad de paquetes procesados. Estos datos son volcados sobre un fichero.

El segundo programa se encarga de leer los timestamps recogidos por el otro programa y calcular los intervalos entre paquetes y la media aritmética de estos. Para calcular el error relativo el programa simplemente se encarga de realizar la diferencia entre el tiempo esperado y el tiempo real y se divide por el valor esperado. Para expresar el valor en forma de porcentaje simplemente se multiplica el resultado por cien.

Para permitir la recepción de paquetes en el sistema encargado de la captura se debe configurar la interfaz de recepción de forma que:

- Se deshabilite el protocolo de negociación de la interfaz Ethernet. La interfaz de red dispone de un protocolo de negociación a través del cual el receptor indica al emisor que limite o pare el envió de información al no poder soportar la recepción de más paquetes.

- Los descriptores de la interfaz soporten una cantidad razonable de paquetes. Es recomendable configurar el tamaño del buffer de los descriptores de forma que la interfaz soporte la recepción de al menos 4096 paquetes.

- Se deshabiliten el agrupamiento de paquetes de protocolo de transporte TCP o UDP. Esto es obligatorio ya que los paquetes formados por el generador desarrollado en este proyecto crea paquetes con una estructura similar y difieren muy pocos bits de las cabeceras. Esta situación puede provocar que el kernel de captura interprete los paquetes como duplicaciones o partes de un fragmento y descarte la mayoría de ellos.

Para realizar esta configuración se dispone de los siguientes comandos:

- *$ ethtool -K <interfaz> lro off gso off gro off gso off tso off ufo off*
- *$ ethtool -A <interfaz> autoneg off rx off tx off*
- *$ ethtool -G <interfaz> rx 4096 tx 4096*
- *$ ethtool -k <interfaz>*
- *$ ethtool -g <interfaz>*
- *$ ethtool -a <interfaz>*

Paquetes Capturados	Intervalo esperado	Intervalo Medio	Error Relativo
1000	10 ns	1 us	100 %
1000	100 ns	1 us	100 %
1000	1 us	1 us	0 %
1000	10 us	10 us	0 %
1000	100 us	100 us	0 %
1000	10^3 us	9.97 x 10^2 us	0.3 %
1000	10^4 us	9.97 x 10^3 us	0.3 %
1000	10^5 us	9.97 x 10^4 us	0.3 %

Tabla 3.- Estimación tiempo entre paquetes

Si se observan los datos obtenidos en la tabla 3 se puede apreciar que la medias para intervalos del orden de los micro segundos han resultado precisas. Las resultados para los intervalos del orden de los nano segundos han resultado erróneos, sin embargo se debe a un error de precisión de la estructura para el timestamp de la librería pcap, es decir, cuando el tiempo de llegada del paquete es recogido no se puede obtener más precisión que del orden de 10^{-6}. Si se obtuviera una captura más precisa es probable que se obtuvieran buenos resultados.

Por último el resto de medidas tiene un pequeño error, que observando los intervalos obtenidos de la captura de tráfico aparenta desviarse de forma que los paquetes llegan antes de tiempo. Esto puede deberse a una desviación al contar los ciclos necesarios antes de transmitir un paquetes o a un error en la precisión de las variables que almacenan las marcas de tiempo

5.1.2.-Pruebas en el entorno real

El objetivo de probar en un entorno real en el que existe más tráfico además del generado es comprobar si el diseño es funcional para integrarlo en algún tipo de sistema para medir la QoS. Además también comprobar si la existencia de tráfico ajena afecta al intervalo entre paquetes propuesto.

Para realizar las pruebas se ha propuesto el esquema mostrado en la figura 23, utilizando la máquina con IP pública como objetivo del inyector. Esta máquina escucha en la interfaz de red conectada a Internet. Por otro lado la ZedBoard conectada en la red privada se configura de forma que envíe 1000 paquetes a la dirección IP pública de la máquina, pero como el diseño no cuenta con la implementación del protocolo ARP, se debe indicar directamente la dirección física del router de enlace.

Para obtener los datos deseados se recurren a los mismos programas implementados en el apartado anterior.

Paquetes Enviados	**Paquetes Capturados**	**Intervalo esperado**	**Intervalo Medio**	**Error Relativo**
1000	0	1 us	0 us	100 %
1000	964	10 us	10 us	0 %
1000	1000	10^2us	100 us	0 %
1000	1000	10^3 us	9.97 x 10^2 us	0.3 %
1000	1000	10^4 us	9.97 x 10^3 us	0.3 %
1000	1000	10^5 us	9.97 x 10^4 us	0.3 %

Tabla 4.-Estimación tiempo entre paquetes en entorno real

De esta forma observando los datos obtenidos en la tabla 4 hay que destacar en primer lugar los paquetes enviados con un intervalo de tiempo de 1 us. Como se puede comprobar se ha producido un error de 100%, provocado por el router de enlace, ya que este rechaza los paquetes con tampoco intervalo de tiempo.

El resto de datos observamos que presenta las mismas características que en el entorno controlado, sin embargo en el caso de un intervalo de 10 us se han producido pérdidas de paquetes. Esto afectaría a la estimación de los parámetros de la red provocando errores de cálculo o precisión.

5.2.- Clasificador de tráfico

5.2.1.- Entorno Controlado

Como en el otro diseño las pruebas en el entorno controlado tienen la finalidad de comprobar el correcto funcionamiento del diseño implementado, además de comprobar que el sistema es útil para comprobar el tiempo de ida y vuelta de los paquetes(TTL). En este entorno el clasificador de tráfico tiene que ser capaz de identificar los diferentes campos de los paquetes de red y de transmitirlos de forma correcta de vuelta a la red a través de una de las interfaces del módulo de ampliación de red FMC.

Para realizar las pruebas se ha recurrido al esquema de conexión de la figura 22, y se ha realizado un pequeño programa en lenguaje C encargado de abrir un socket UDP y enviar datagramas con un identificador de paquete y un timestamp en los datos UDP. Para recoger los paquetes se ha utilizado el programa de análisis de tráfico WireShark, filtrando los paquetes por emisor(dirección MAC del diseño implementado en la ZedBoard).

A partir de esta captura se ha calculado el retardo de ida y vuelta de los paquetes con un programa en C basado en la libreria "pcap". A través de la librería se pueden extraer los datos (timestamp de salida) y el timestamp de recepción. Como la emisión y recepción es sobre la misma CPU se puede omitir la sincronización entre relojes para la captura.

En las pruebas realizadas se han enviado 1000 paquetes con un intervalo de 1 segundo para evitar sobrecargar la entrada del diseño, obteniendo un retardo de medio de 1.6 x 10^{-3} segundos (160 us).

5.2.2.- Entorno Real

Para las pruebas en el entorno real hay que volver basarse en esquema de la figura 23, sin embargo la generación de paquetes se produce en la CPU y no en la ZedBoard por lo que debe configurarse en el otro sentido, es decir, la ZedBoard tendrá la IP pública y el ordenar se alojará en la red interna.

Como en el caso del generador de tráfico, con esta prueba se puede observar como el tiempo de ida y vuelta de los paquetes se ve afectado por la existencia de tráfico externo al del sistema implementado, además de comprobar el correcto funcionamiento del módulo de clasificación recibiendo tráfico no clasificable.

Para la prueba se han utilizado los mismos programas que en el entorno controlado, a diferencia de una pequeña modificación del programa de generación de paquetes, en el que se debe abrir el puerto de recepción para que el router de la red externa no rechace los paquetes devueltos por la ZedBoard.

En este caso sobre los 1000 paquetes enviados con un intervalo de 1 segundo se obtiene un retardo medio de 5.1 x 10^{-4} segundos (510 us).

5.3.- Conclusiones de las pruebas

Las pruebas anteriores se han centrado en verificar el correcto funcionamiento de los módulos implementados. Sin embargo con estas primeras pruebas ya se pueden obtener datos con los que estimar medidas de red. Concretamente en estas pruebas cuyo objetivo no era medir la latencia de la transmisión ya se pueden obtener el retardo en un sentido con el generador de tráfico y el tiempo de ida y vuelta con el clasificador de paquetes.

Los resultados finales se han obtenido tras varios intentos, ya que durante la ejecución de las pruebas se encontraban errores de diseño. Por ejemplo, el módulo de generación de tráfico funcionaba correctamente en el entorno controlado, sin embargo al enviar paquetes hasta la red pública se descubrió que sus paquetes eran rechazados por el router porque tenían mal formado el checksum de la cabecera IP. Finalmente los resultados obtenidos de ambos diseños han sido satisfactorios, ya que se comprueba que los diseños se han integrado fácilmente en un sistema real de red.

Una de las pruebas que se debería realizar en siguiente lugar es utilizar dos plataformas ZedBoard conectadas punto a punto y cada una con uno de los diseños. Por ejemplo se podría conectar el generador de tráfico en la red privada del laboratorio HPCN y el clasificador con la IP pública, recogiendo el tráfico para obtener las estadísticas de la

red. Esta prueba no se ha podido realizar porque no se disponía de dos módulos de ampliación FMC.

6.- Conclusiones

Tras el desarrollo de este trabajo se ha conseguido una plataforma de red para el desarrollo sobre el dispositivo Zynq de Xilinx, fácilmente ampliable para nuevas funcionalidades y facilitando el desarrollo de estas a través del uso de herramientas de síntesis de alto nivel..

Desde el punto de vista didáctico el proyecto ha sido muy útil para consolidar los conocimientos sobre la arquitectura de las redes, sus protocolos de aplicación y mecanismos para trabajar con estas. También ha servido para ampliar los conocimientos de programación a lo largo de toda la carrera y para mejorar el manejo de sistemas con GNU/Linux. Sin embargo el grueso de los conocimientos aprendidos han sido la arquitectura de computadoras y la metodología de desarrollo sobre FPGA.

El uso de las nuevas aplicaciones y herramientas de Xilinx para el desarrollo de hardware han establecido una base para poder desarrollar las nuevas tecnologías que quedan por aparecer en el mercado, que se espera que estén basadas en estos esquemas de desarrollo de alto nivel.
Por último fuera desde el punto de vista de desarrollo ha sido fundamental obtener la capacidad de leer y interpretar de forma ágil y correcta los manuales de los dispositivos hardware.

Con las pruebas realizadas en el diseño se ha comprobado que los diseños que se implementen sobre esta plataforma serán fácilmente integrables en entornos reales para disponer de una herramienta de bajo coste para medir la QoS de las redes. Además en el futuro se podrían abaratar los costes de producción trasladando esta plataforma a dispositivos más baratos basados en SoC programable Xilinx Zynq como MicroZedBoard. Los costes de portar el proyecto serían mínimos gracias al uso de herramientas para el desarrollo como Vivado y Vivado HLS.

Como futuros trabajos se podría mejorar la implementación realizado incluyendo mecanismo de sincronización entre los dos diseños implementados de forma que se puedan situar cada diseño en distintos puntos de la red y realizar conexiones punto a punto para realizar los estudios necesarios con datos fiables de gran precisión. También se podría mejorar la precisión de los tiempos de capturas de timestamp realizando estas capturas a nivel hardware.

Referencias

[1] Wikipedia. Definición monitorización de red. http://es.wikipedia.org/wiki/Monitorizaci%C3%B3n_de_redes

[2] J. Ramos, P.M Santiago del Río, J. Aracil, J.E. López de Vergara (2011). *On the effect of concurrent applications in band width measurement speedometers*

[3] Javier Ramos de Santiago(2010). *Análisis e implementación de un sistema real de medida de ancho de banda*. Trabajo de Fin de Master publicado. Universidad Autónoma de Madrid.

[4] Javier Ramos de Santiago (2013). *Proactive Measurement Techniques for Networking Monitoring In Heterogeneus Enviroments*. Tesis doctoral publicada. Universidad Autónoma de Madrid.

[5] Victor Moreno, Jaime J. Garnica, Francisco J. Gomez-Arribas, Sergio Lopez-Buedo, Ivan Gonzalez, Javier Aracil, Mikel Izal, Eduardo Magaña, Daniel Morato. *High-accuracy network monitoring using ETOMIC testbed.*

[6] Soheil Hassas Yeganeh, Milad Eftekhar, Mohammad Jalali, Yashar Ganjali(2013) . GitHub.NetFPGA . https://github.com/NetFPGA/netfpga/wiki/TrafficClassification#NetFPGAbased_Traffic_Classifier

[7] Github. NetThreads. https://github.com/NetFPGA/netfpga/wiki/NetThreads

[8] E. Srikanth (2013). *Zynq-7000 AP SoC Redirecting Ethernet Packet to PL for Hardware Packet Inspection Tech Tip*. http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+Redirecting+Ethernet+Packet+to+PL+for+Hardware+Packet+Inspection+Tech+Tip

[9] Srinivasa Attili, Sunita Jain, Sumanranjan Mitra (2013). *Ethernet Performance with Jumbo Frame Support & PL Ethernet in Zynq-7000 AP SoC.* http://www.wiki.xilinx.com/Zynq+PL+Ethernet

[10] E Srikanth (2013). *Zynq AP SoC Redirecting Ethernet Header to Cache via PL and ACP port Tech Tip*. http://www.wiki.xilinx.com/Zynq+AP+SoC+Redirecting+Ethernet+Header+to+Cache+via+PL+and+ACP+port+Tech+Tip

[11] Anirudha Sarangi , Stephen MacMahon Xilinx(2012). *LightWeight IP (lwIP) Application Examples.*

[12] Guía implementación proyecto XillyBus. Proyecto XillyBus. http://xillybus.com/downloads/xillybus_product_brief.pdf

[13] ARM.*Protocolo AMBA*. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[14] Documentación Xilinx. *Guia usario Zynq*. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[15] Xilinx. *AXI*. http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf

[16] V.López, J.L añamuro, V.Moreno, J.E. López de Vergara, J. Aracil, C. García, J.P Fernández-Palacios, M. Izal. *Implementation of Multi-layer techniques using FEDERICA, PASITO and OneLab network infrasestructures.*

[17*] ZedBoard Hardware User Guide(2012).* http://www.zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[18] Xilinx (2013).*Vivado Design Suite User Guide. High Level Synthesis*.

[19] Zhan.z, Y.Amano(2012). *TB-FMCL-GLAN-B Hardware User Manual.*

[20] Xilinx. *LogiCORE IP AXI Ethernet(2014)* . http://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v6_1/pg138-axi-ethernet.pdf

[21] Xilinx(2012). *LogiCORE IP AXI DMA Product Guide.*

[22] Xilinx(2014). *LogiCORE IP Binary Counter Product Guide.*

[23] Xilinx(2012). *LogiCORE IP Clocking Wizard Product Guide.*

[24] A.Fekete, P.Hága, J.Stéger , J.Aracil , G.Iannaccone , G.Vattay (2008). http://onelab.eu/images/PDFs/Deliverables/onelab2d42.pdf

[25] Wikipedia. Lenguaje VHDL. http://es.wikipedia.org/wiki/VHDL

[26] Wikipedia. Lenguaje Verilog. http://es.wikipedia.org/wiki/Verilog

[27] Xilinx(2012). *LogiCORE IP Tri-Mode-Ethernet MAC.* http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_eth_mac/v5_5/pg051-tri-mode-eth-mac.pdf

Referencias Imágenes

[28] "Arquitectura básica Zynq *". Introduction to Zynq-7000, All Programmable SoC.* AVNET SpeedWay Slides

[29] E. Srikanth (2013)." Diagrama Bloques Interfaz de red conectada a Memoria" *.Zynq-7000 AP SoC Redirecting Ethernet Packet to PL for Hardware Packet Inspection Tech Tip.*
http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+Redirecting+Ethernet+Packet+to+PL+for+Hardware+Packet+Inspection+Tech+Tip

[30] Srinivasa Attili, Sunita Jain, Sumanranjan Mitra (2013). " Esquema Conexión PS LogiCore al medio físico ".*Ethernet Performance with Jumbo Frame Support & PL Ethernet in Zynq-7000 AP SoC.*
http://www.wiki.xilinx.com/Zynq+PL+Ethernet

[31] Srinivasa Attili, Sunita Jain, Sumanranjan Mitra (2013). " Diagrama Conexión LogiCore en área programable ".*Ethernet Performance with Jumbo Frame Support & PL Ethernet in Zynq-7000 AP SoC.*
http://www.wiki.xilinx.com/Zynq+PL+Ethernet

[32] E Srikanth (2013)."Procesamiento de paquetes en área programable ". *Zynq AP SoC Redirecting Ethernet Header to Cache via PL and ACP port Tech Tip*.
http://www.wiki.xilinx.com/Zynq+AP+SoC+Redirecting+Ethernet+Header+to+Cache+via+PL+and+ACP+port+Tech+Tip

[33] "Estructura del sistema Xillybus". Guía implementación proyecto XillyBus.
http://xillybus.com/downloads/xillybus_product_brief.pdf

[34] "Diagrama básico de Zynq". *ZedBoard Hardware User Guide(2012).*
http://www.zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[35] Zhan.z, Y.Amano(2012). "Arquitectura Módulo FMC-Ethernet". *TB-FMCL-GLAN-B Hardware User Manual.*

[36] Xilinx(2012). "Diagrama Bloques Tri-Mode Ethernet MAC". *LogiCORE IP Tri-Mode-Ethernet MAC.*
http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_eth_mac/v5_5/pg051-tri-mode-eth-mac.pdf

[37] Xilinx(2012). "Diseño Ejemplo Tri-Mode LogiCore". *LogiCORE IP Tri-Mode-Ethernet MAC.*
http://www.xilinx.com/support/documentation/ip_documentation/tri_mode_eth_mac/v5_5/pg051-tri-mode-eth-mac.pdf

[38] Xilinx."Transmisión desde AXI-Ethernet". *LogiCORE IP AXI Ethernet(2014)* . http://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v6_1/pg138-axi-ethernet.pdf

[39] Xilinx."Paquete Ethernet + IP +TCP". *LogiCORE IP AXI Ethernet(2014)* . http://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v6_1/pg138-axi-ethernet.pdf

[40] Xilinx."Palabras de control AXI4-Stream de transmisión". *LogiCORE IP AXI Ethernet(2014)* . http://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v6_1/pg138-axi-ethernet.pdf

Apéndice A: Código para Vivado HLS del proyecto

Clasificador HLS Básico

```
#include "parser.h"
#include <stdint.h>

using namespace std;

uint32_t _pross_pkts = 0;

void parser_top (axi_interface_type data_input[240], axi_interface_type data_output1[240],
axi_interface_type data_output2[240], axi_interface_type data_output1_PS[240],
                uint32_t *pross_pkts, uint32_t *protocol_selection){
//Register interfaces to be connected to the core that talks to ublaze
#pragma HLS INTERFACE ap_none port=protocol_selection
#pragma HLS INTERFACE register port=pross_pkts

//Tell the compiler this interfaces talk to FIFOs
#pragma HLS INTERFACE ap_fifo port=data_input
#pragma HLS INTERFACE ap_fifo port=data_output1
#pragma HLS INTERFACE ap_fifo port=data_output2
#pragma HLS INTERFACE ap_fifo port=data_output1_PS

//Map HLS ports to AXI interfaces. Tell the compiler this interfaces are AXI4-Stream
#pragma HLS RESOURCE variable=data_input core=AXI4Stream metadata="-bus_bundle STREAM_A"
#pragma HLS RESOURCE variable=data_output1      core=AXI4Stream metadata="-bus_bundle STREAM_B"
#pragma HLS RESOURCE variable=data_output2      core=AXI4Stream metadata="-bus_bundle STREAM_C"
#pragma HLS RESOURCE variable=data_output1_PS core=AXI4Stream metadata="-bus_bundle STREAM_D"



        axi_interface_type data_read, data_tosend;
        ap_uint<32> internal_buffer[240];
        ap_uint<4> last_strobe;
        ap_uint<32> data_out;
        ap_uint<1> lastt;


        int i;

        /*Read HEAD*/
        read_head: for(i=0;i < 6;i++){
                data_read = *data_input;
                internal_buffer[i] = data_read.data;
                data_input++;
        }


        /*Classify frames by the type of them*/
        int decision = 0;
        classify_pross: {
                uint32_t temp;
                uint32_t *byte_pointer;
                temp = internal_buffer[5];
                byte_pointer = &temp;
                byte_pointer = byte_pointer + 0x000000FF;
                if (*protocol_selection == *byte_pointer )
                        decision = 1;
                else
                        decision = 2;
        }
        /*Count number of processed pakects*/
        update_statistics: {
                _pross_pkts = _pross_pkts + 1;
                *pross_pkts = _pross_pkts;
        }

        /*Read until the frame ends*/
        int j= 6;
```

```
        read_tail: do {
                data_read = *data_input;
                internal_buffer[++j] = data_read.data;
                last_strobe = data_read.strb;
                data_input++;
        } while (!(data_read.last));

        /*Send the packet in one way*/
        if (decision == 1){
                send_out1: for(i=0;i<j;i++){
#pragma HLS PIPELINE
                        data_tosend.data = internal_buffer[i];
                        data_tosend.strb = 1;
                        data_tosend.last = 0;
                        *data_output1 = data_tosend;
                        *data_output1_PS = data_tosend;
                        data_output1++;
                        data_output1_PS++;
                }
                data_tosend.data = internal_buffer[j];
                data_tosend.strb = last_strobe;
                data_tosend.last = 1;
                *data_output1 = data_tosend;
                *data_output1_PS = data_tosend;
                data_output1++;
                data_output1_PS++;
        } else {
                send_out2: for(i=0;i<j;i++){
#pragma HLS PIPELINE
                        data_tosend.data = internal_buffer[i];
                        data_tosend.strb = 1;
                        data_tosend.last = 0;
                        *data_output2 = data_tosend;
                        data_output2++;
                }
                data_tosend.data = internal_buffer[j];
                data_tosend.strb = last_strobe;
                data_tosend.last = 1;
                *data_output2 = data_tosend;
                data_output2++;
        }
```

Splitter

```
//------------------------------------------------------------------------------
// TFG - 2014
// Packets Splitter
// Author: Alfredo Sosa Muñoz, Gustavo Sutter
// DESCRIPTION: This program read a AXI4 Stream input and split it in two flow.
// One output contains the control or status information about the Ethernet Data.
// The other output transmits the Ethernet Data. The control information is always
// 6x32 bits words.
// REV: 1.0
//------------------------------------------------------------------------------


#include "split.h"

using namespace std;

void split_top(stream<axi_interface_type> &data_input,
                stream<axi_interface_type> &data_output1,
                stream<axi_interface_type> &data_output1_ctrl) {

#pragma HLS RESOURCE variable=data_output1 core=AXIS metadata="-bus_bundle AXI4Stream_M1"
#pragma HLS RESOURCE variable=data_input core=AXIS metadata="-bus_bundle AXI4Stream_S1"
#pragma HLS RESOURCE variable=data_output1_ctrl core=AXIS metadata="-bus_bundle
AXI4Stream_M1_ctrl"

        /*Variables for reading the interfaces*/
        axi_interface_type buf_data_input;
```

```
        axi_interface_type buf_data_input_ctrl;
        int i;

        /*Always are 6x32 bits words*/
        Loop_ctrl: for (i = 0; i < 6; i++) {
#pragma HLS PIPELINE rewind
                data_input.read(buf_data_input_ctrl);
                data_output1_ctrl.write(buf_data_input_ctrl);
        }


        Loop_data: do {
#pragma HLS LOOP_TRIPCOUNT min=1 max=10
#pragma HLS PIPELINE rewind
                data_input.read(buf_data_input);
                data_output1.write(buf_data_input);
        } while (!buf_data_input.last);

}
```

Mixer

```
//------------------------------------------------------------------------------
// TFG - 2014
// Packets Mixer
// Author: Alfredo Sosa Muñoz, Gustavo Sutter
// DESCRIPTION: This program takes the control information and data about one Ethernet packet to
mix
// in one flow. Read 6x32 bits words of control or status from an AXI4Stream input and then puts
them
// in other AXI4Stream output. The same output is shared by the control information and data
information.
// REV: 1.0
//------------------------------------------------------------------------------

#include "mix.h"

using namespace std;

void mix_top(stream<axi_interface_type> &data_input,
                stream<axi_interface_type> &data_input_ctrl,
                stream<axi_interface_type> &data_output1) {

#pragma HLS RESOURCE variable=data_output1 core=AXIS metadata="-bus_bundle AXI4Stream_M1"
#pragma HLS RESOURCE variable=data_input core=AXIS metadata="-bus_bundle AXI4Stream_S1"
#pragma HLS RESOURCE variable=data_input_ctrl core=AXIS metadata="-bus_bundle
AXI4Stream_S1_ctrl"

        /*Variables for reading the interfaces*/
        axi_interface_type buf_data_input;
        axi_interface_type buf_data_input_ctrl;
        int i;


        /*Always are 6x32 bits words*/
        Loop_ctrl: for (i = 0; i < 6; i++) {
#pragma HLS PIPELINE rewind
                data_input_ctrl.read(buf_data_input_ctrl);
                data_output1.write(buf_data_input_ctrl);
        }


        Loop_data: do {
#pragma HLS LOOP_TRIPCOUNT min=1 max=10
#pragma HLS PIPELINE rewind
                data_input.read(buf_data_input);
                data_output1.write(buf_data_input);
        } while (!buf_data_input.last);

}
```

Clasificador final

```
//------------------------------------------------------------------------------
// TFG - 2014
// Packets Clasifier
// Author: Alfredo Sosa Muñoz, Gustavo Sutter
// DESCRIPTION: This program read one input stream a analyze the packet's data to
// send it in the right way. If data is UDP packet and contais the same source
// port like the user indicate, the packet is return to the network.
// REV: 1.0
//------------------------------------------------------------------------------

#include "clasifier.h"

using namespace std;

uint32_t _pross_pkts = 0;
uint32_t _ctrl_contador = 0;
uint32_t _data_contador = 0;

void clasifier_top(stream<axi_interface_type> &data_input,
                stream<axi_interface_type> &data_output1,
                stream<axi_interface_type> &data_output2, uint32_t *pross_pkts,
                uint32_t *protocol_selection, uint16_t *port) {

#pragma HLS RESOURCE variable=data_output1 core=AXIS metadata="-bus_bundle AXI4Stream_M1"
#pragma HLS RESOURCE variable=data_output2 core=AXIS metadata="-bus_bundle AXI4Stream_M2"
#pragma HLS RESOURCE variable=data_input core=AXIS metadata="-bus_bundle AXI4Stream_S1"

#pragma HLS RESOURCE variable=protocol_selection core=AXI4LiteS metadata="-bus_bundle
axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=pross_pkts core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=port core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"

        /*Variable to read AXIStream input*/
        axi_interface_type buf_data_input;
        axi_interface_type buf_data_input_ctrl;

        /*Buffer to write control information before send*/
        axi_interface_type buffer_ctrl[10];
        /*Buffer to write data before send*/
        axi_interface_type buffer_data[1024];

        /*variable to iterate with data*/
        int it_data;
        int i = 0;

        /*Always 6x32 bits words are about control information*/
        Loop_ctrl: for (i = 0; i < 6; i++) {
#pragma HLS PIPELINE
                data_input.read(buf_data_input_ctrl);
                _ctrl_contador++;
                buffer_ctrl[i] = buf_data_input_ctrl;
        }
        it_data = 0;
        Loop_data: do {
#pragma HLS LOOP_TRIPCOUNT min=1 max=10
#pragma HLS PIPELINE
                data_input.read(buf_data_input);
                _data_contador++;
                buffer_data[it_data++] = buf_data_input;
        } while (!buf_data_input.last);

        /*rebote inteligente*/
        int decision = 1;
        classify_pross: {
                //variables to read data
                uint32_t type_ip;
                uint32_t version_ip;
                uint32_t protocol;
```

```
                type_ip = buffer_data[3].data;
                version_ip = buffer_data[3].data;
                protocol = buffer_data[5].data;
                ;

                //mascarade big endian
                version_ip = version_ip >> 20;
                version_ip = version_ip & 0x0000000F;
                type_ip = type_ip & 0x0000FFFF;

                if (type_ip == 0x0008) {
                   if (version_ip == 0x4) {
                     protocol = protocol >> 24;
                     protocol = protocol & 0x000000FF;
                     if (protocol == *protocol_selection) {
                        uint16_t dest_port = (uint16_t)(buffer_data[9].data & 0x0000FFFF);
                        uint16_t port_be = 0;
                        port_be = (*port) << 8;
                        port_be = port_be | (*port >> 8);

                        if(port_be == dest_port ){
                                decision = 2;
                                /*Change direction of MAC address*/
                                uint32_t buffer_mac1, buffer_mac2, buffer_mac3,buffer_despl;

                                buffer_mac1 = buffer_data[0].data;
                                buffer_mac2 = buffer_data[1].data;
                                buffer_mac3 = buffer_data[2].data;

                                buffer_despl = buffer_mac3 << 16;
                                buffer_despl = buffer_despl | (buffer_mac2 >> 16);
                                buffer_data[0].data = buffer_despl;
                                buffer_despl = buffer_mac1 << 16;
                                buffer_despl = buffer_despl | (buffer_mac3 >> 16);
                                buffer_data[1].data = buffer_despl;
                                buffer_despl = buffer_mac1 >> 16;
                                buffer_despl = buffer_despl | (buffer_mac2 << 16);
                                buffer_data[2].data = buffer_despl;
                                /*change direction IP address*/

                                uint32_t buffer_ip1, buffer_ip2, buffer_ip3;
                                uint32_t  buffer_ipdespl,buffer_sport;

                                buffer_ip1 = buffer_data[6].data;
                                buffer_ip2 = buffer_data[7].data;
                                buffer_ip3 = buffer_data[8].data;
                                buffer_sport = buffer_data[8].data;

                                buffer_ipdespl = buffer_ip2 & 0xFFFF0000;
                                buffer_ipdespl = buffer_ipdespl | (buffer_ip1 & 0x0000FFFF);
                                buffer_data[6].data = buffer_ipdespl;

                                buffer_ipdespl = buffer_ip1 & 0xFFFF0000;
                                buffer_ipdespl = buffer_ipdespl | (buffer_ip3 & 0x0000FFFF);
                                buffer_data[7].data = buffer_ipdespl;

                                buffer_ipdespl = buffer_ip3 & 0xFFFF0000;
                                buffer_ipdespl = buffer_ipdespl | (buffer_ip2 & 0x0000FFFF);
                                buffer_data[8].data = buffer_ipdespl;

                                /*Change direction of UDP ports*/
                                uint32_t buffer_dport,desplaza_puerto = 0;
                                buffer_dport = buffer_data[9].data;

                                desplaza_puerto = buffer_dport << 16;
                                buffer_data[8].data = desplaza_puerto | (buffer_data[8].data &
0x0000FFFF);


                                desplaza_puerto = buffer_sport >> 16;
                                buffer_data[9].data = desplaza_puerto | (buffer_dport &
0xFFFF0000);
```

```
                                }
                            }
                        }
                } else {
                        decision = 1;
                }
        }
        if (decision == 2) {
                Loop_outp: for (i = 0; i < 6; i++) {
#pragma HLS PIPELINE
                        data_output1.write(buffer_ctrl[i]);
                        data_output2.write(buffer_ctrl[i]);
                }
                Loop_outp2: for (i = 0; i < it_data; i++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=10
#pragma HLS PIPELINE
                        data_output1.write(buffer_data[i]);
                        data_output2.write(buffer_data[i]);

                }
        } else {
                Loop_outp3: for (i = 0; i < 6; i++) {
#pragma HLS PIPELINE
                        data_output1.write(buffer_ctrl[i]);
                }
                Loop_outp4: for (i = 0; i < it_data; i++) {
#pragma HLS LOOP_TRIPCOUNT min=1 max=10
#pragma HLS PIPELINE
                        data_output1.write(buffer_data[i]);

                }
        }
        _pross_pkts = _pross_pkts + 1;
        *pross_pkts = _pross_pkts;

}
```

Inyector de paquetes final

```
/------------------------------------------------------------------------------
// TFG - 2014
// Packets Clasifier
// Author: Alfredo Sosa Muñoz
// DESCRIPTION: This program creates and send N packets to an AXI-Ethernet.
// REV: 1.0
//------------------------------------------------------------------------------

#include "paquetsinyector.h"

using namespace std;

uint32_t _pross_pkts = 0;
uint32_t _ctrl_contador = 0;
uint32_t _data_contador = 0;

void paquets_inyect(stream<axi_interface_type> &data_output1,
                uint32_t *source_mac1, uint32_t *source_mac2, uint32_t *dest_mac1,
                uint32_t *dest_mac2, uint32_t *source_ip, uint32_t *dest_ip,
                uint32_t *source_port, uint32_t *dest_port, uint32_t *npaquets,
                uint32_t *TBpaquets,uint32_t *external_counter,ap_uint<1> *reset_counter) {

#pragma HLS RESOURCE variable=data_output1 core=AXIS metadata="-bus_bundle AXI4Stream_M1"

#pragma HLS RESOURCE variable=source_mac1 core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=source_mac2 core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=dest_mac1 core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=dest_mac2 core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=source_ip core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=dest_ip core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=source_port core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
```

```
#pragma HLS RESOURCE variable=dest_port core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=npaquets core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=TBpaquets core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle axis_CONTROL_BUS"

        /*Variable to read AXIStream input*/
        axi_interface_type buf_data_input;
        axi_interface_type buf_data_input_ctrl;

        /*Buffer to write control information before send*/
        axi_interface_type buffer_ctrl[10];
        /*Buffer to write data before send*/
        axi_interface_type buffer_data[1024];
        /*Statics control word to inyect paquets*/
        axi_interface_type inyect_ctrl[10];
        /*Static data to inyect paquets*/
        axi_interface_type inyect_data[1024];

        *reset_counter = 0;
        inicialize_ctrl_inyect_paquet: {
                inyect_ctrl[0].data = 0x00450042;
                inyect_ctrl[0].keep = 0xFFFF;
                inyect_ctrl[0].last = 0;
                inyect_ctrl[1].data = 0x00000000;
                inyect_ctrl[1].keep = 0xFFFF;
                inyect_ctrl[1].last = 0;
                inyect_ctrl[2].data = 0x00000000;
                inyect_ctrl[2].keep = 0xFFFF;
                inyect_ctrl[2].last = 0;
                inyect_ctrl[3].data = 0x00450042;
                inyect_ctrl[3].keep = 0xFFFF;
                inyect_ctrl[3].last = 0;
                inyect_ctrl[4].data = 0x00000000;
                inyect_ctrl[4].keep = 0xFFFF;
                inyect_ctrl[4].last = 0;
                inyect_ctrl[5].data = 0x00450042;
                inyect_ctrl[5].keep = 0xFFFF;
                inyect_ctrl[5].last = 1;
        }

        inicialize_data_inyect_paquet: {
                /*variables to converto data to big endian*/
                uint32_t smac1 = 0;
                uint32_t smac2 = 0;
                smac1 = (*source_mac1 << 24) & 0xFF000000;
                smac1 = smac1 | ((*source_mac1 << 8) & 0x00FF0000);
                smac1 = smac1 | ((*source_mac1 >> 8) & 0x0000FF00);
                smac1 = smac1 | ((*source_mac1 >> 24) & 0x000000FF);

                smac2 = (*source_mac2 << 24) & 0xFF000000;
                smac2 = smac2 | ((*source_mac2 << 8) & 0x00FF0000);
                smac2 = smac2 | ((*source_mac2 >> 8) & 0x0000FF00);
                smac2 = smac2 | ((*source_mac2 >> 24) & 0x000000FF);

                /*variables to converto data to big endian*/
                uint32_t dmac1 = 0;
                uint32_t dmac2 = 0;
                dmac1 = (*dest_mac1 << 24) & 0xFF000000;
                dmac1 = dmac1 | ((*dest_mac1 << 8) & 0x00FF0000);
                dmac1 = dmac1 | ((*dest_mac1 >> 8) & 0x0000FF00);
                dmac1 = dmac1 | ((*dest_mac1 >> 24) & 0x000000FF);

                dmac2 = (*dest_mac2 << 24) & 0xFF000000;
                dmac2 = dmac2 | ((*dest_mac2 << 8) & 0x00FF0000);
                dmac2 = dmac2 | ((*dest_mac2 >> 8) & 0x0000FF00);
                dmac2 = dmac2 | ((*dest_mac2 >> 24) & 0x000000FF);

                uint32_t sip = 0;
                sip = (*source_ip << 24) & 0xFF000000;
                sip = sip | ((*source_ip << 8) & 0x00FF0000);
                sip = sip | ((*source_ip >> 8) & 0x0000FF00);
                sip = sip | ((*source_ip >> 24) & 0x000000FF);
```

```
uint32_t dip = 0;
dip = (*dest_ip << 24) & 0xFF000000;
dip = dip | ((*dest_ip << 8) & 0x00FF0000);
dip = dip | ((*dest_ip >> 8) & 0x0000FF00);
dip = dip | ((*dest_ip >> 24) & 0x000000FF);

uint32_t sport = 0;
sport = (*source_port << 24) & 0xFF000000;
sport = sport | ((*source_port << 8) & 0x00FF0000);
sport = sport | ((*source_port >> 8) & 0x0000FF00);
sport = sport | ((*source_port >> 24) & 0x000000FF);

sport = sport & 0xFFFF0000;

uint32_t dport = 0;
dport = (*dest_port << 24) & 0xFF000000;
dport = dport | ((*dest_port << 8) & 0x00FF0000);
dport = dport | ((*dest_port >> 8) & 0x0000FF00);
dport = dport | ((*dest_port >> 24) & 0x000000FF);

dport = dport >> 16;

/*IP Checksum*/
uint16_t sum = 0;
uint32_t sum_tmp = 0;
sum_tmp += 0x4500;
sum_tmp += 0x002B;
sum_tmp += 0x0811;
sum_tmp += (uint16_t) (*source_ip >> 16);
sum_tmp += (uint16_t) (*source_ip & 0x0000FFFF);
sum_tmp += (uint16_t) (*dest_ip >> 16);
sum_tmp += (uint16_t) (*dest_ip & 0x0000FFFF);

sum = (sum_tmp & 0x0000FFFF) + ((sum_tmp & 0xFFFF0000) >> 16);
/*C-1*/
sum = ~sum;

/*BIG ENDIAN*/
uint32_t sum_f = 0;
sum_f = sum >> 8;
sum_f = sum_f | ((sum & 0x00FF) << 8);

inyect_data[0].data = dmac1;
inyect_data[0].keep = 0xFFFF;
inyect_data[0].last = 0;
inyect_data[1].data = (smac1 << 16) | dmac2;
inyect_data[1].keep = 0xFFFF;
inyect_data[1].last = 0;
inyect_data[2].data = (smac2 << 16) | (smac1 >> 16);
inyect_data[2].keep = 0xFFFF;
inyect_data[2].last = 0;
inyect_data[3].data = 0x00450008;
inyect_data[3].keep = 0xFFFF;
inyect_data[3].last = 0;
inyect_data[4].data = 0x00002B00;
inyect_data[4].keep = 0xFFFF;
inyect_data[4].last = 0;
inyect_data[5].data = 0x11080000;
inyect_data[5].keep = 0xFFFF;
inyect_data[5].last = 0;
inyect_data[6].data = (sip << 16) | sum_f;
inyect_data[6].keep = 0xFFFF;
inyect_data[6].last = 0;
inyect_data[7].data = (dip << 16) | (sip >> 16);
inyect_data[7].keep = 0xFFFF;
inyect_data[7].last = 0;
inyect_data[8].data = (dip >> 16) | sport;
inyect_data[8].keep = 0xFFFF;
inyect_data[8].last = 0;
inyect_data[9].data = 0x0A000000 | dport;
inyect_data[9].keep = 0xFFFF;
```

```
                inyect_data[9].last = 0;
                inyect_data[10].data = 0x0000FFFF;
                inyect_data[10].keep = 0xFFFF;
                inyect_data[10].last = 0;
                inyect_data[11].data = 0xFFFFFFFF;
                inyect_data[11].keep = 0xFFFF;
                inyect_data[11].last = 0;
                inyect_data[12].data = 0xFFFFFFFF;
                inyect_data[12].keep = 0xFFFF;
                inyect_data[12].last = 1;
        }

        /*Send packets*/
        for (int j = 0; j < *npaquets;) {
                if (*external_counter >= *TBpaquets && *reset_counter != 1) {
                        int i = 0;
                        Loop_outp3: for (i = 0; i < 6; i++) {
#pragma HLS PIPELINE
                                data_output1.write(inyect_ctrl[i]);
                        }
                        *reset_counter = 1;
                        Loop_outp4: for (i = 0; i < 13; i++) {
#pragma HLS PIPELINE
                                data_output1.write(inyect_data[i]);
                        }
                        j++;
                }else{
                        *reset_counter = 0;
                }
        }
}
```