UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

PROYECTO FIN DE CARRERA

Desarrollo mediante lenguaje de alto nivel de un sistema basado en FPGA para aplicaciones de red en 40 Gbps Ethernet

Hugo Peire García
JULIO 2014

Desarrollo mediante lenguaje de alto nivel de un sistema basado en FPGA para aplicaciones de red en 40 Gbps Ethernet

AUTOR: Hugo Peire García

TUTOR: Gustavo Sutter

High Performance and Computing Network Group

Ingeniería de Telecomunicaciones

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Julio 2014

Agradecimientos

Querría agradecer la realización de este proyecto a diversas personas. Primero empezaré por mi tutor Gustavo Sutter y mis compañeros del laboratorio HPCN pues son ellos los que más me han ayudado en aquellos momentos de dudas y de estar atascado y no saber avanzar.

También dando la vista atrás a todos mis compañeros de carrera que me han acompañado durante estos siete años de mi vida y han sido mis compañeros y después mis amigos. Con ellos compartí la experiencia de ser universitario y me ayudaron a superar este proceso. De ellos querría destacar a Alejandro Hidalgo, Álvaro García, Carlos Sánchez, Carlos Moreno, Sergio de la Cruz y Sergio Zurtia.

Fuera de la universidad recibí el apoyo de mis amigos los cuales me ayudaban a desconectar los fines de semana y a sudar el estrés con algún que otro partido de baloncesto en los cuales, a pesar de tener un resultado nefasto, me hacían pasar un momento memorable. Gracias Nacho Rodriguez-Auñon, Hugo Cisneros, Ricardo Grañon, Brueno Marenco y Nehuen Blanco.

Finalmente querría agradecer a mi familia y a mi novia por todo lo que me han soportado. Por un lado mi familia la cual me crió y me dio todo, a mi madre por estar ahí siempre y a mi hermana Lara que a pesar de la distancia de estos dos últimos años siempre estuvo ahí para guiarme y hacerme ver el camino correcto. Y por otro a Ángela la cual gracias a su paciencia y comprensión me escucho durante horas asintiendo meramente con la cabeza pues no entendía lo que le decían del proyecto pero sé que sin ella no habría sido capaz.

¡GRACIAS A TODOS!

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

ABSTRACT

Solutions based on Software (SW) executed on hardware (HW) of general purpose that are used to solve network issues are impracticable on high speed networks of up to 10 Gbps or more. For example, the filtering of networks packets that a firewall uses is very difficult to implement on SW at rates of tens of gigabits per second.

The solution for this type of issues is to develop the HW on reprogrammable logic (FPGA – Field Programable Gate Array). Traditionally developments on FPGA were used on HW descriptive languages (HDLs – Hardware Description Languages) such as VHDL or Verilog. None the less it is known of the poor efficiency of this type of languages. In this project a solution based on the use of a synthesis tool of high level languages (HLL), in particular Xilinx's Vivado-HLS, is proposed. This tool allows to develop HW solutions directly from C/C++ code.

KEY WORDS: NetFPGA, High Level Sysntheis, Xilinx, Network Routing

RESUMEN

Las soluciones basadas en software (SW) ejecutando en hardware (HW) de propósito general que se emplean tradicionalmente para resolver problemas de red son inviables en redes de muy alta velocidad, a 10 Gbps o superiores. Por ejemplo, el filtrado de paquetes que utiliza un cortafuego es muy difícil de implementar en SW a tasas de decenas de gigabits por segundo.

La solución para este tipo de problemas es el desarrollo de HW en lógica reconfigurable (FPGA – Field Programable Gate Array). Tradicionalmente los desarrollos FPGA se realizan utilizando lenguajes de descripción de Hw (HDLs – Hardware Description Languages) como VHDL o Verilog. Sin embargo, es conocido que la productividad de estos lenguajes es pobre. En este trabajo se propone validar el uso de una herramienta de síntesis desde lenguajes de alto nivel (HLL – High Level Language), en particular Vivado-HLS de Xilinx. Esta herramienta permite desarrollar HW directamente desde código C/C++.

PALABRAS CLAVES: NetFPGA, High Level Sysnthesis, enroutamiento de red

1 Introducción

1.1 Motivación

El análisis de tráfico de red es vital en las comunicaciones. Esta información es utilizada para encaminar el tráfico de red en routers y switches así como para administración de alto nivel en antivirus y firewalls. Las redes de alta velocidad (10 Gbps y superiores) comienzan a ser cada vez más frecuentes y sin embargo la capacidad de procesamiento en tiempo real del software no siempre es adecuada.

La solución de desarrollo de hardware específico para el análisis y administración de la red es muy frecuente, aunque muy onerosa en términos de tiempo y coste de desarrollo.

Las FPGAs son una alternativa de coste moderado para resolver con prestaciones HW problemas que con tecnologías de circuitos específicos (ASICs – Aplication Specific Integrated Circuits) serían inviables. No obstante el desarrollo HW basado en FPGAs es menos oneroso, la metodología tradicional de diseño basado en lenguajes HDL es lenta y difícil de validar.

Para el desarrollo de este trabajo se utilizará la plataforma NetFPGA-10G desarrollada por la empresa High Tech Global que utiliza un dispositivo Virtex 5 de Xilinx. El proyecto NetFPGA, liderado por la Universidad de Stanford, Xilinx y la Universidad de Cambridge es un proyecto de código libre y que se desarrolla en comunidad.

1.2 Objetivos

El objetivo principal de este proyecto es validar el uso de la síntesis de alto nivel (HLS-*High Level Synthesis*) para FPGAs aplicado al procesamiento de información de redes. Los hitos del proyecto son los siguientes:

1. Instalación y puesta en marcha de un equipo con una placa NetFPGA 10G. Se estudiará el entorno de la plataforma y se realizará la instalación y configuración de la maquina que alberga la placa.
2. Validar con ejemplos simples el uso de Lenguajes de algo nivel (HLL) para describir módulos de procesamiento de red. Para ello se tendrá que comprender y saber utilizar las herramientas de diseño proporcionadas por la familia Xilinx.
3. Analizar la facilidad de uso y el rendimiento de cores desarrollados en HLL respecto de HDL.
4. Valorar la posibilidad de establecer una metodología para el uso de lenguajes HLL en el contexto de procesamiento de redes.

1.3 Organización de la memoria

La memoria ha sido organizada en los siguientes capítulos:

- **Introducción**. En esta se explica las motivación sobre porque se decidió trabajar con esta tecnología y también se presenta el objetivo principal de este proyecto. Además contiene esta página donde se ve la organización que se sigue en esta memoria.

- **Estado del Arte,** en esta la segunda sección se describe la tecnología utilizada explicando la filosofía de la plataforma NetFPGA y una descripción física de la placa. Además se describe las herramientas de diseño que se utilizan para la generación de los archivos.

- **Diseño** es la tercera sección de esta memoria. Aquí se presentan los conocimientos necesarios que se han de tener sobre la tecnología, los componentes que se adhieren a la placa y se explica con detalles la plataforma NetFPGA. Esta sección finaliza con un primer proyecto demostrando el uso de las herramientas de diseño.

- **Desarrollo,** en esta cuarta sección se plantea detalladamente el objetivo a seguir, se plantea la solución y se da una demostración detallada sobre la solución creada.

- **Pruebas, resultados y mejoras** es la quinta y penúltima sección en donde una vez mostrada la solución al objetivo se presentan las distintas pruebas que efectivamente demuestran que lo realizado cumple con las expectativas y además presenta los resultados de manera simple. Se finaliza con dos casos prácticos en donde se aplican dos distintas mejoras para cada caso.

- **Conclusiones y Trabajo Futuro**, la memoria finaliza con esta sección en donde se resumen las mejoras que se han introducido con esta tecnología, como se han cumplido con los objetivos y finaliza con las posibilidades futuras que este proyecto plantea.

2 Estado del arte

2.1 NetFPGA

NetFPGA [14] es una plataforma de software y hardware libre destinado para la investigación en sistemas de aceleración de hardware con aplicaciones de red y prototipos de alta velocidad. Este proyecto nace en la Universidad de Stanford debido a la motivación de crear una herramienta de enseñanza para sistemas de redes. Los profesores se dieron cuenta de que los estudiantes solo ganaban conocimientos de las capas tres y superiores del modelo OSI [19] teniendo carencias de las capas uno y dos, las correspondientes al nivel físico y al nivel de enlace de datos. La NetFPGA inicial solo fue accesible por descargas de simulaciones en donde pudieron ver el funcionamiento de estas capas con la placa [36]. Actualmente existen dos plataformas NetFPGA 1G y NetFPGA 10G.

La versión de 1G es una tarjeta PCI que contiene un Virtex-II Pro 50 FPGA de Xilinx, 4 puertos Ethernet de 1 Gigabit, una RAM estática (SRAM), una DDR2 y una DRAM [16].

Figura 1: Placa NetFPGA 1G [16]

La versión 10G es la placa con la que se ha realizado el proyecto. Esta placa se distribuye gracias a HitechGlobal [11] y fue diseñada por NetFPGA. Las características son las siguientes:

- *Xilinx Virtex-5 XC5VTX240T-2FFG1759C FPGA*

- *Four SFP+ interface (using 16 RocketIO GTX transceivers and 4 Broadcom EDC devices). Supports both 10Gbps and 1Gbps modes*
- *X8 PCI Express Gen 2 (5Gbps/lane)*
- *Twenty Configurable GTX Serial Transceivers (available through two high-speed Samtec connectors)*
- *Three x36 Cypress QDR II (CY7C1515JV18)*
- *x36 Micron RLDRAM II (MT49H16M36HT-25)*
- *Mictor Connector for debugging*
- *Two Xilinx Platform XL Flash (128mb each) devices*
- *One Xilinx XC2C256 CPLD*
- *One DB9 (RS232) port*
- *User LEDs & Push Buttons*
- *9.5" x 4.25*"

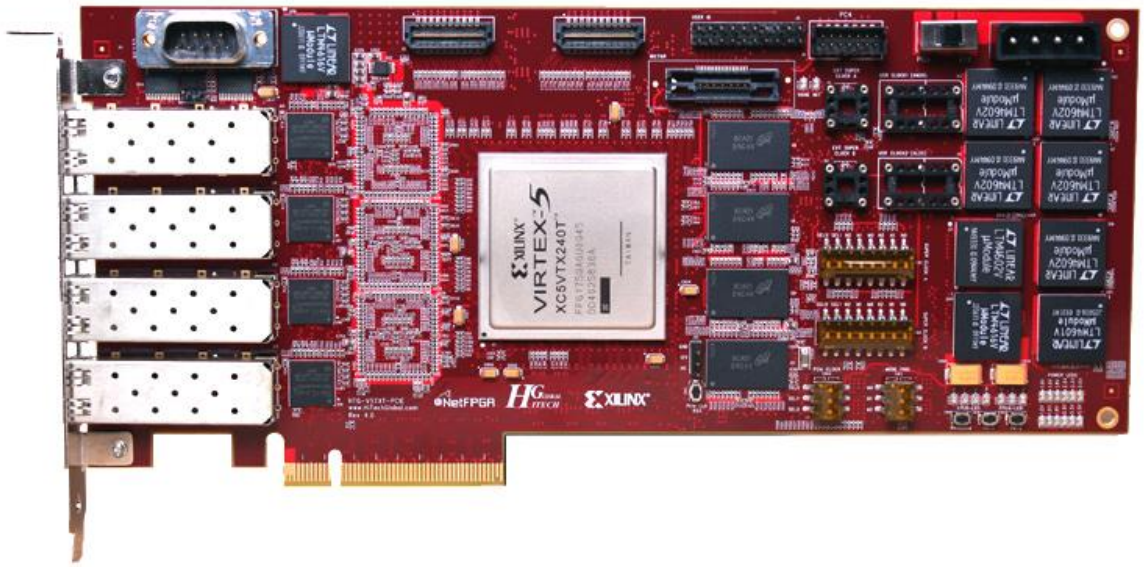


Figura 2: Placa NetFPGA 10G [11]

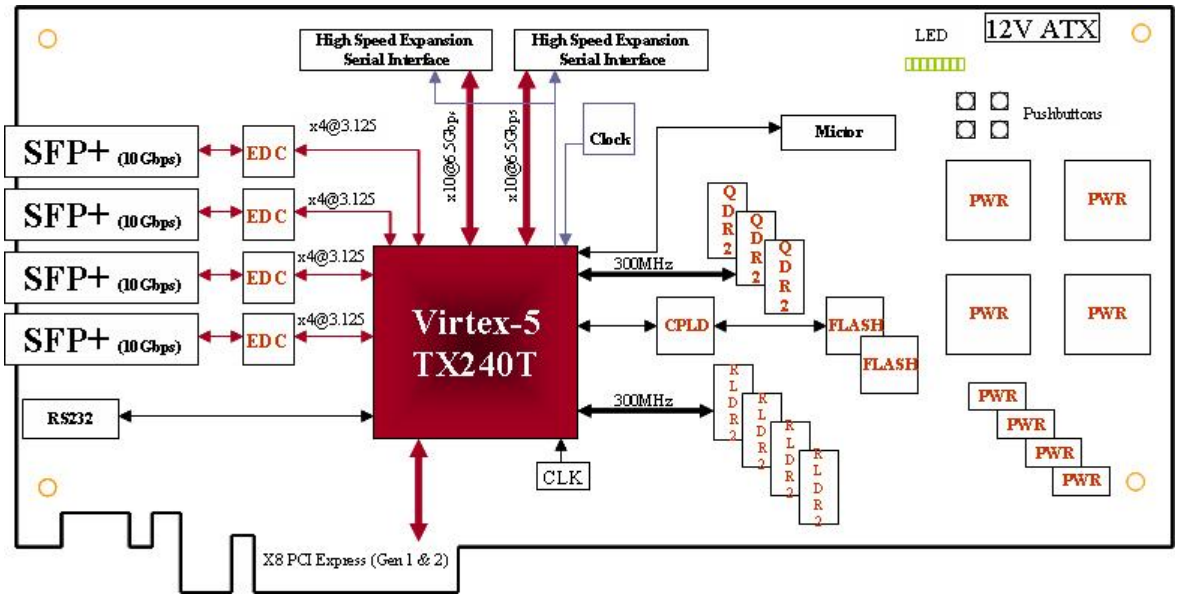


Figura 3: Plano de NetFPGA 10G [11]

Son inmensas las utilidades que se pueden sacar al uso de las placas NetFPGA, pero entre ellas conviene destacar unas pocas. Esta útil placa, gracias a su potente procesador, permite analizar paquetes a alta velocidades. Esto se puede utilizar para monitorizar los paquetes y controlar la seguridad. Por ello una de las utilidades de esta placa es utilizarla a modo de firewall pues puede inspeccionar paquete a paquete a pesar de la alta velocidad de entrada, y desechar aquellos paquetes peligrosos u ofensivos. Por otro lado gracias a uno de los proyectos ya existentes que más tarde se explicará se puede utilizar a modo de congestionador de la red. Es decir, cuando la red esté saturada por algún camino, se enrutarán los paquetes hacia otros destinos y así crear un tráfico más fluido.

2.2 Herramientas de diseño

Xilinx es una empresa Americana líder en el sector del desarrollo de *programable logic devices* (FPGA). Fue fundada por Ross Freeman, Bernie Vondescmitt y Jim Barnett en 1984 con principal sede en Silicon Valley en California, USA [30]. Para la realización de este proyecto se ha de tener conocimiento de esta empresa debido a que gran parte de las herramientas y software utilizado pertenecen a ella y el dispositivo que contiene la placa NetFPGA, un Virtex 5, pertenece a la familia tipo Virtex de Xilinx [27].

Dentro de las herramientas de Xilinx de tipo software se ha utilizado el paquete ISE (*Integrated Software Environment*) el cual incluye dos de las herramientas más utilizadas para la realización de este proyecto, *Xilinix Platform Studio (XPS)* y el *Software Development Kit (SDK)* [29]. Esta primera se ha necesitado para facilitar la construcción del proyecto en versión hardware. Por construcción también se da a entender que esta herramienta se ha utilizado para conectar los distintos módulos o IP cores (más adelante se hablarán de ellos), para configurar dichos módulos y añadirles distintas señales como puedan ser el reloj y el *reset*. XPS utiliza un interfaz gráfico muy útil para el programador en hardware que le facilitará las conexiones mediante los bloques simplemente conectando dichos terminales en vez de tener que programar señales que conecten ambos puertos [34].

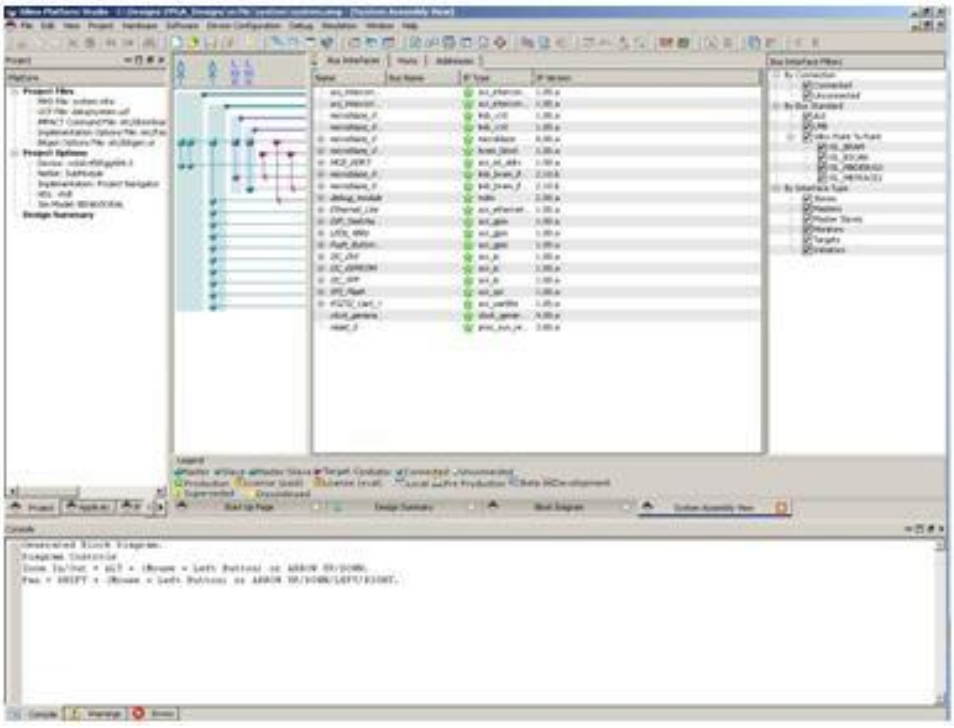


Figura 4: XPS interfaz gráfico [34]

Por otro lado la herramienta SDK está enfocada como su propio nombre indica a un uso a nivel de software. Esta herramienta que utiliza un soporte basado en Eclipse [8] realiza la tarea de dar un sentido a nivel software a los módulos conectados a nivel hardware mediante XPS. Simplemente manda ordenes al núcleo MicroBlaze para que de las ordenes convenientes y que todo funcione con correcta harmonía [32].

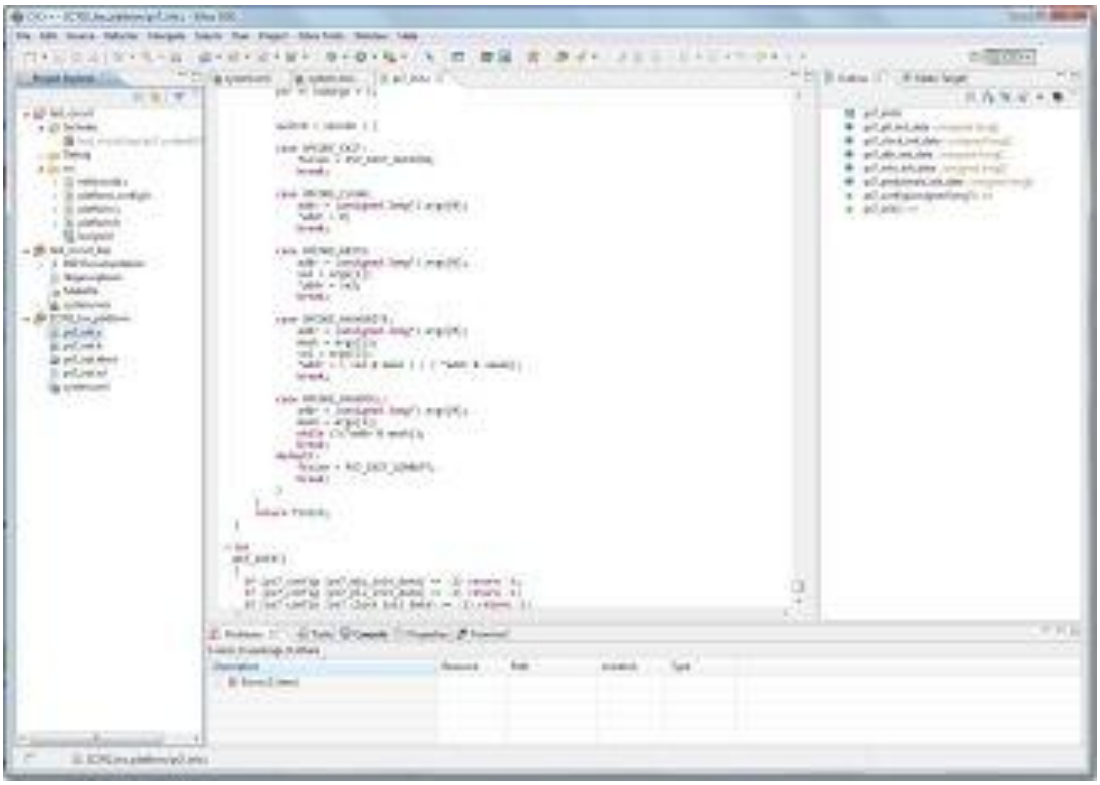


Figura 5: SDK Interfaz gráfico [32]

Finalmente nos queda por introducir la herramienta principal en la realización de este proyecto, Vivado HLS [33]. Vivado *High-Level Synthesis* es una vez más una herramienta de la familia Xilinx que permite sintetizar lenguaje de alto nivel para distintas aplicaciones. El *High-Level Synthesis* [10] es también conocido como *C synthesis* permite sintetizar un código escrito en C a lenguaje de más bajo nivel para aplicaciones de tipo hardware con lenguajes HDL. Hoy en día las complejidades de ciertos circuitos electrónicos están llevando a la inviabilidad de programar mediante hardware el total del programa. Por ello gracias a esta herramienta mediante un simple código en C se pueden crear circuitos mucho más complejos en hardware.

3 Diseño

3.1 Puesta en marcha

Para poder trabajar de manera eficiente con NetFPGA fue necesario realizar unos pasos previos antes de poder empezar. Lo primero que se hizo fue formatear los equipos para instalar el sistema operativo compatible con NetFPGA, Fedora [9] 14 (x86_64). A continuación se instaló la última versión de las herramientas de Xilinx previamente descritas y necesarias para la realización de este proyecto [23]. La licencia para estos programas se obtuvo gracias al repositorio de licencias que tiene la Universidad Autónoma de Madrid.

Tras tener los programas necesarios se ha de verificar que en la máquina en la que se esté trabajando tenga una ranura libre de PCI-express (PCIe). Un PCIe es simplemente un bus de expansión diseñado para sustituir las previas tecnologías PCIx [21]. Por otro lado se ha de obtener transceptores SFP+ y un set de cables de fibra óptica como los siguientes de la imagen:

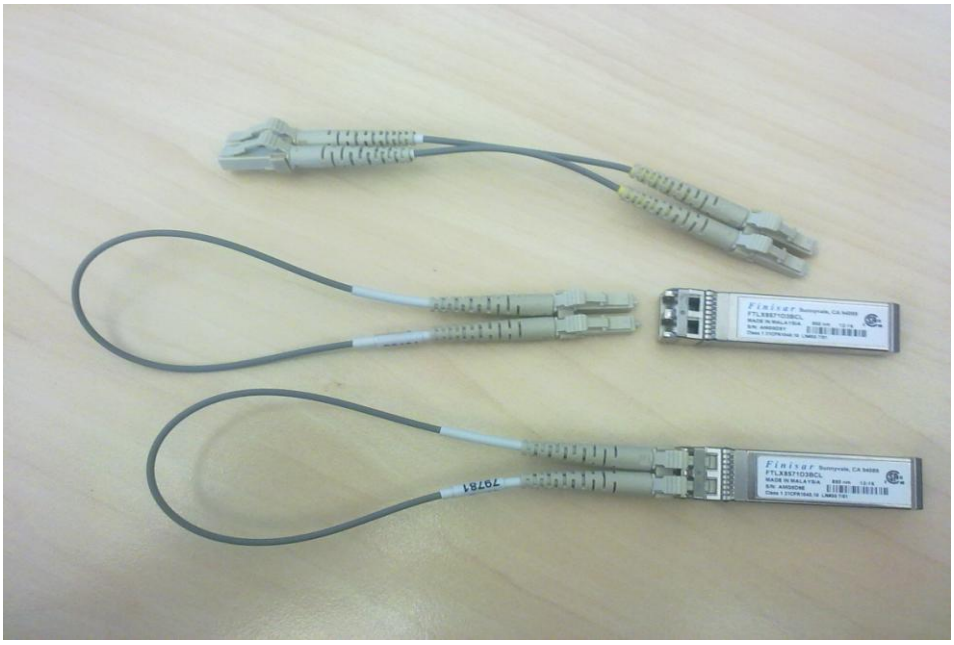


Figura 6: SFP+ transceptores y fibras ópticas

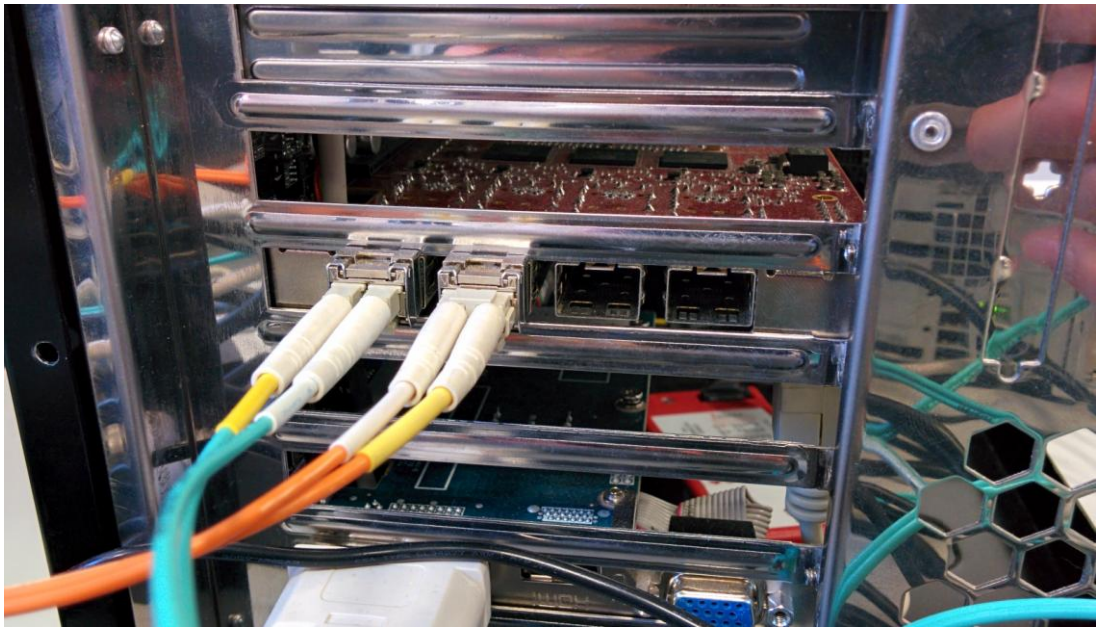


Figura 7: Conectores SFP+ y fibras ópticas

Finalmente y como en la preparación de cualquier FPGA se ha de tener un cable de configuración de la placa o JTAG. En este caso se utilizará el cable de la familia Xilinx [35]. Para poder configurar la placa una vez el cable y la placa están bien conectados se ha de cargar el bit file, archivo con el programa que se desea cargar en la placa, mediante el programa Impact de Xilinx. Este programa directamente detectará la placa FPGA conectada y mediante unos simples pasos pedirá el archivo de configuración. Tras haberse cargado el programa habrá que reiniciar la máquina para que la carga se haga efectiva y ya se podrá utilizar la placa.

Figura 8: Cable de configuración JTAG de XILINX [35]

Por lo tanto tras tener todo el equipo conectado y configurado la placa quedará de la siguiente manera:

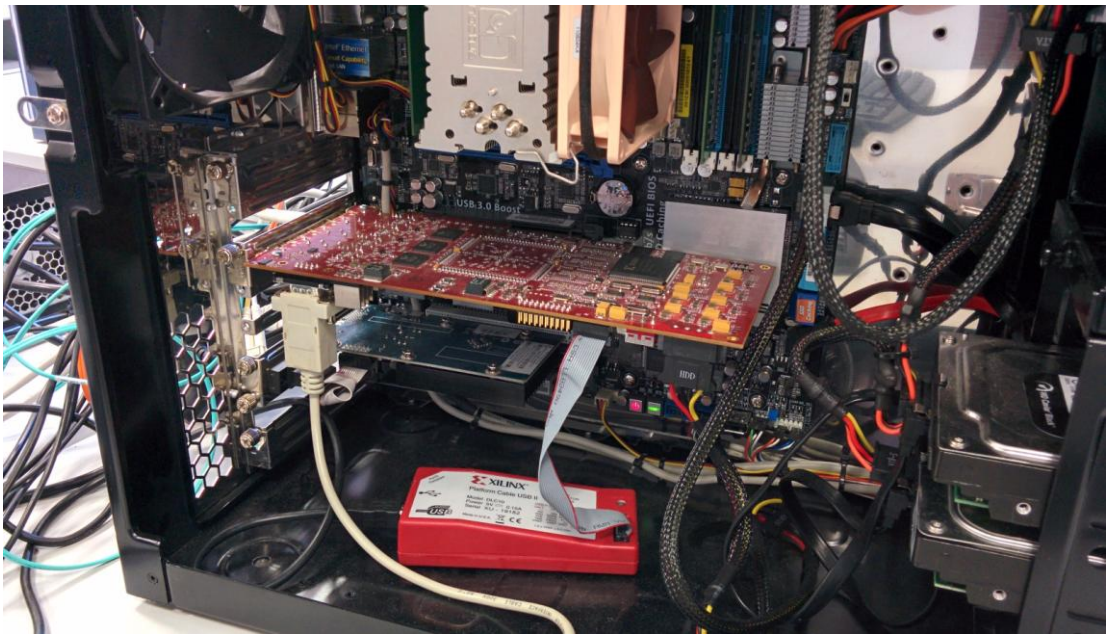


Figura 9: Foto de la placa con sus conexiones

En la imagen se puede observar la conexión de un cable RS232. Este es opcional puesto que para el desarrollo de este proyecto no es estrictamente necesario pero si puede servir. Simplemente este cable dará información sobre el MicroBlaze, e imprimirá por pantalla diversa información. Estos se pueden configurar modificando el código fuente mediante el programa SDK.

3.2 Proyecto NetFPGA

Para un mejor entendimiento de que es un proyecto NetFPGA fue necesario darse de alta en el repositorio de código www.github.com. Mediante este usuario creado y tras pedir petición se puede acceder tanto a la wiki pública: https://github.com/NetFPGA/NetFPGA-public/wiki como a la privada. Tras haber realizado estos pasos se procedió a descargarse el código con proyectos ya creados o proyectos de referencia.

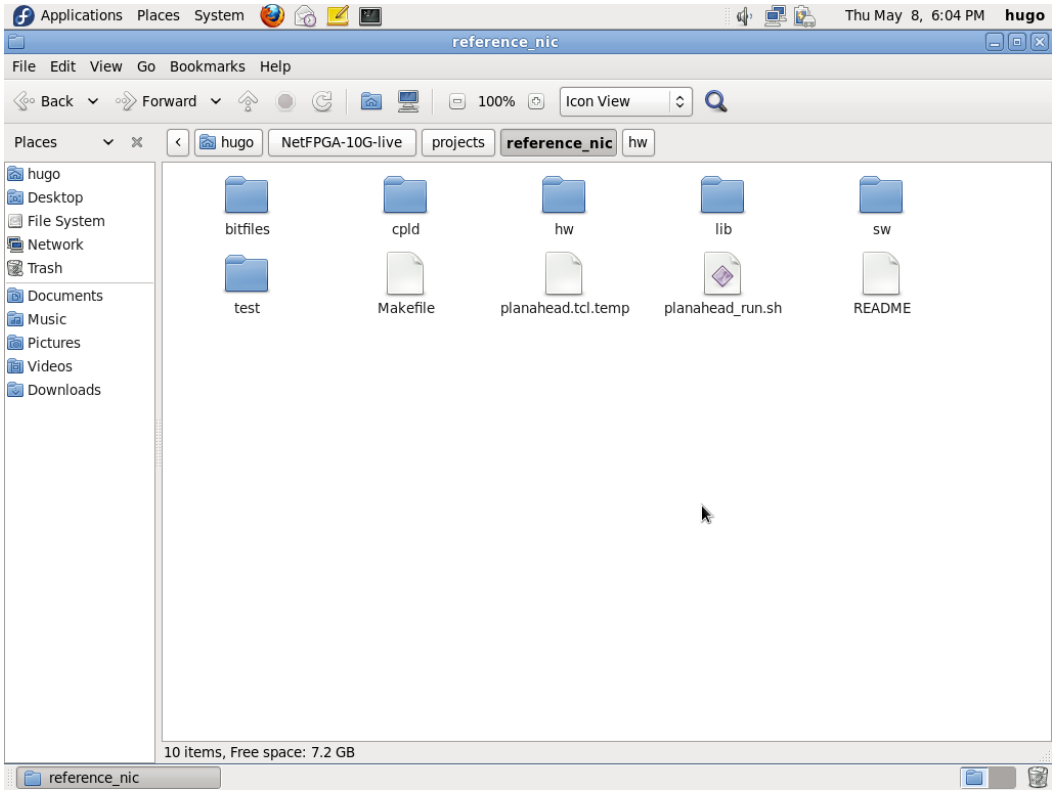


Figura 10: Estructura de un proyecto NetFPGA

En la imagen podemos observar un pantallazo del fichero de un proyecto de referencia perteneciente al código base de NetFPGA. Todos los proyectos NetFPGA han de contener dos ficheros principales, el de SW y el de HW. En el de HW es donde está el archivo system.xmp que es el proyecto principal que se abre con el programa de Xilinx EDK. Abriendo este archivo se verá el contenido de tipo hardware de cada proyecto. Es decir los distintos módulos o Pcores que contiene cada proyecto. Además si se desea añadir nuevos módulos dentro de la carpeta hardware existe una llamada Pcores en la cual se pueden añadir nuevos módulos para después conectarlos con el EDK. Por otro lado en la carpeta SW están los archivos que se abren con la

herramienta de Xilinx SDK y son las funciones a nivel software que controlan el MicroBlaze. Está estructura es la definida debido a que se utiliza el programa Xilinx EDK aunque el usuario es capaz de estructurárselo a su gusto.

3.3 Simulación

Todo proyecto de NetFPGA viene con una simple simulación que permite simular la conexión de la placa con el PCI express y los puertos de 10G. Esta simulación lo que crea son unos módulos de simulación que remplazan las conexiones con el PCIe y los puertos 10G. Los puertos 10G pertenecen a la capa física y siguen un protocolo AXI4-Stream, del cual se hablará en una sección venidera.

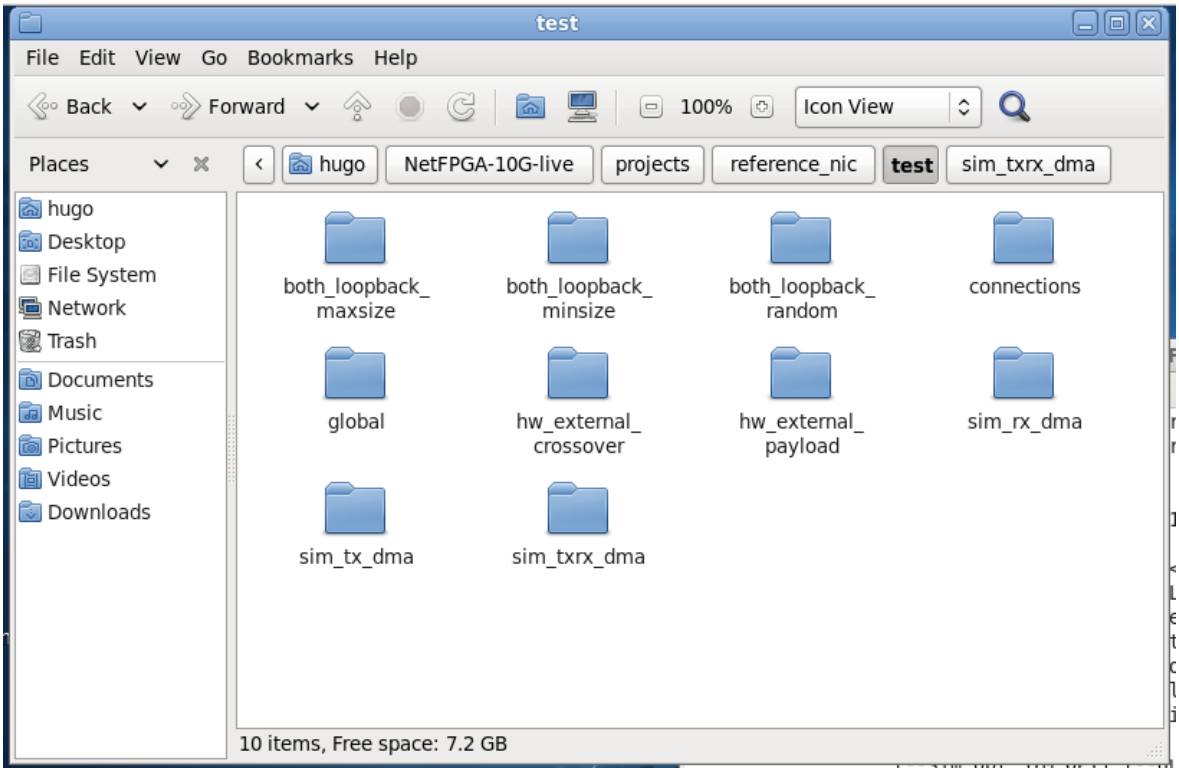


Figura 11: Archivos python

Como se puede observar en la figura todos los proyectos de tipo NetFPGA contienen una carpeta de test. En ella existen varios archivos python, .py, los cuales son ejecutados para generar la simulación.

También se pueden observar diferentes ficheros en los cuales cada uno tiene un archivo python distinto siendo el nombre de su fichero una descripción suficiente. Para ejecutar cada simulación bastará con tener un proyecto tipo NetFPGA con las variables de entorno cargadas y ejecutar lo siguiente:

```
>> cd ~/NetFPGA-10G-live/tools/bin
>> ./nf_test.py sim --major tx --minor dma --isim --tx --packet_length {size} --packet_no {number}
```

Estas dos líneas de comando lo que harán será cambiar la carpeta destino desde consola y ejecutar el archivo python que existen en ella. Este es un archivo general el cual luego llama al test sim_tx_dma. Este junto con el test de recepción y el de recepción y transmisión será uno de los más utilizados durante la realización de este proyecto. Estos paquetes de simulación lo que simulan es la generación de paquetes desde el DMA para probar su correcta transmisión y recepción.

3.4 Reference nic

Para la realización de este proyecto se partió desde uno de los proyectos ya creados de NetFPGA, uno de los de referencia, en concreto el reference nic.

Como se anticipó en la sección anterior antes de hablar de este proyecto de NetFPGA se debería de explicar el significado de un IP core. IP o *"Intelectual Property"* core es una unidad de lógica o una célula reutilizable que pertenece a una partida que contiene la patente de dicho módulo. Este concepto es vital para entender el verdadero funcionamiento del proyecto NetFPGA pues los proyectos se han creado en su totalidad juntando distintos módulos de la propia librería de NetFPGA como de la familia de Xilinx.

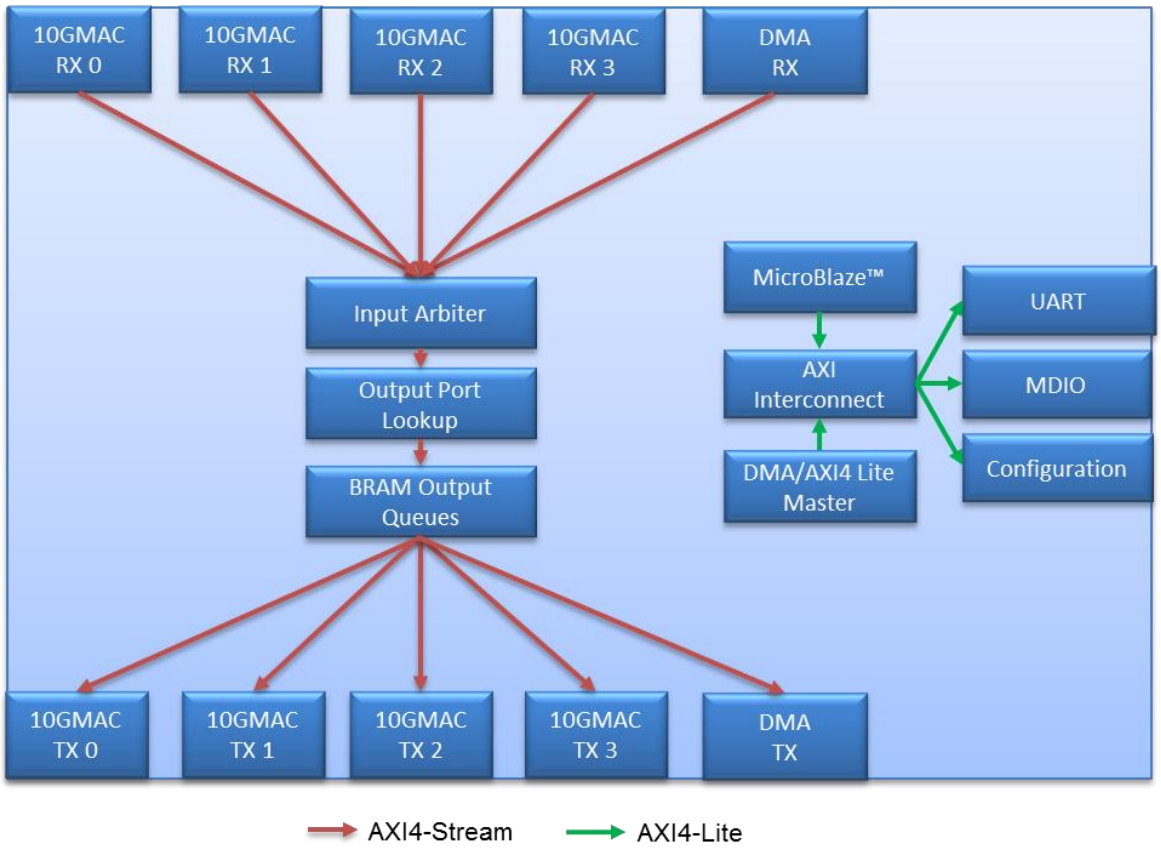


Figura 12: Diseño de bloques de Reference nic [15]

Este diseño es de bloques con distribución tipo pipeline como se puede observar en la imagen de arriba con distintos módulos. Este proyecto junto con un operador NIC simularían las condiciones de un router con puertos de 10GHz y un procesado tipo Virtex 5 expresamente para el enrutamiento de los paquetes.

A estos módulos superiores de la imagen, los 10G MAC RX, le entran los paquetes a través del interfaz 'nf10_10g_interface' [2]. Este es un módulo IP que combina cores XILINX XAUI y 10G MAC en adición a un adaptador AXI4-Stream. Los paquetes que llegan por el interfaz XAUI desde la capa física del modelo OSI, son transformados en una etapa inicial a señales XGMII y después leídos por el core de Xilinx 10G MAC. Finalmente los paquetes son transformados al formato AXI4-Stream. El proceso de transmisión sigue el exacto mismo proceso pero en dirección contraria.

El siguiente módulo que nos topamos, el DMA se explicará más adelante, es el *Input Arbiter* [12]. Éste tiene cinco interfaces de entrada, cuatro de los 10G MAC y uno para el módulo DMA. Cada entrada del arbitrario se conecta con una cola de entrada tipo FIFO, y después mediante un decisor tipo *round-robin* cogerá una cola no vacía y escribirá el paquete entero hacia el siguiente módulo. Los decisores *round-robin* son de lo más sencillos y simplemente mediante un movimiento circular va cola a cola viendo si existe paquete y si no pasa a la cola siguiente. Este decisor no entiende de prioridades y por ello es de los más simples que existen [18].

El módulo *Output Port Lookup* [20] es responsable de la decisión de a qué puerto se enviará cada paquete. Esto sería el equivalente a una tabla de rutas. Una vez tomada la decisión se enviará al módulo de *Output queues*. Este módulo implementa un esquema muy básico de búsqueda y decisión enviando todos los paquetes de los puertos 10G a la CPU y viceversa a través del PCIe, basándose en la cabecera de cada paquete. Nótese que a pesar de solo tener un módulo DMA en Verilog existen cuatro puertos virtuales de tipo DMA. Cada puerto DMA virtual se distingue del otro mediante su campo de origen y destino, 'SRC_PORT/DST_PORT".

Una vez que el paquete llega al nf10_bram_bram_output_queues [5] ya ha sido marcado con su destino, el cual viene en un canal aparte. Según el destino de cada paquete entrará en una cola específica. Existen para ello cinco colas de salida, cuatro para cada puerto de 10G y una para el bloque DMA. Hágase notar que el paquete se desechará en caso de que la cola se llene. Una vez que el paquete este entero en la cola se enviará a la salida. El orden de salida es una vez para un interfaz Ethernet y otra para el DMA, por ello las colas impares son asignadas para los puertos 10G y las pares para el módulo DMA.

El módulo DMA [7] sirve como motor DMA para el diseño NIC de referencia. Incluye un core PCIe de Xilinx y un módulo maestro de AXI4-LITE. A los otros módulos de NetFPGA se les muestra como AXIS maestro más esclavo para enviar y recibir paquetes además de un interfaz AXI4-LITE tipo maestro por el cual todos los registros de tipo AXI pueden ser consultados a través del host sobre el PCIe.

El diseño de NIC de referencia implementa a un subsistema de Xilinx MicroBlaze, incluyendo una memoria de bloque BRAM y su controlador. Adicionalmente del interfaz PCIe del bloque DMA existen dos interfaces de comunicación extras implementados en el diseño: un interfaz UART para procesos de depuración y un bloque MDIO. El bloque MDIO es una versión simplificada del core MDIO de Xilinx. Es usado principalmente para acceder a configurar los componentes de la capa física (AEL2005) usada en la placa.

Para probar el funcionamiento de este proyecto es necesario configurar otra máquina con una tarjeta NIC de referencia, teniendo en la máquina inicial una placa NetFPGA 10G. Esta tarjeta se puede observar en la imagen que sigue, es una tarjeta de red de la familia Oracle con dos puertos Ethernet similares a los de la NetFPGA con transceptores ópticos [25]:

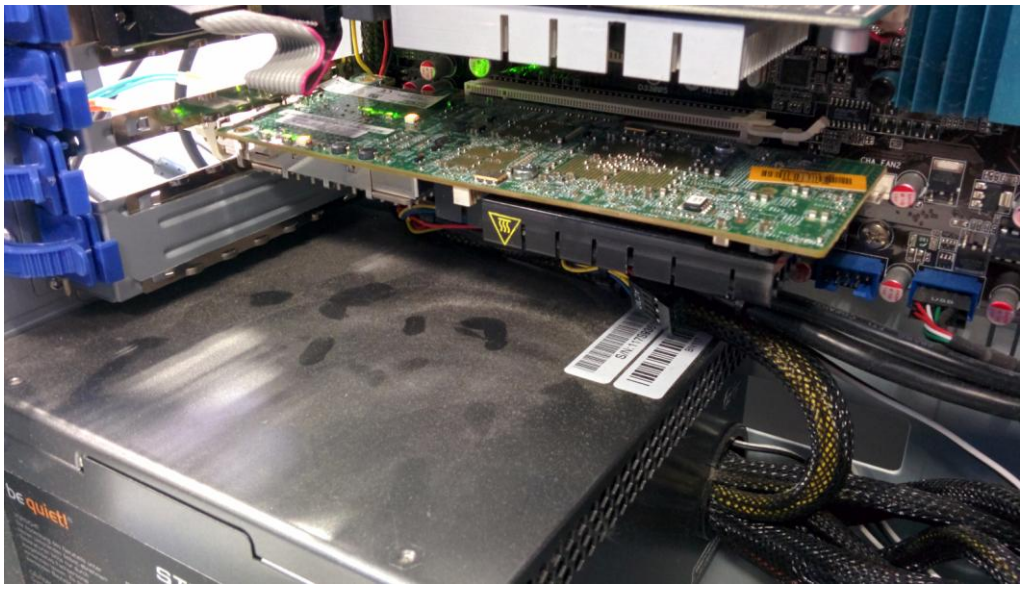


Figura 13: Tarjeta NIC de Oracle

3.5 AXI 4

La plataforma NetFPGA 10G utiliza como interfaz con los IP una serie de interfaces creados por ARM [3] llamados AXI. Más tarde estos interfaces fueron redefinidos por Xilinx llegando a utilizar la versión AXI4. Por ello y ya que la plataforma NetFPGA utiliza procesadores de Xilinx este fue el sistema de comunicación que se adoptó para las comunicaciones entre los IP cores.

No fue hasta 2003 cuando salió la primera versión de AXI incluida en AMBA 3.0 de ARM. Años más tarde en 2010 salió la segunda versión de AXI, AXI4. A continuación se mostraran los conocimientos mínimos y necesarios para la realización de este proyecto pero si se desea tener un conocimiento más profundo lo puede encontrar en [4].

Existen tres tipos de interfaces de AXI4. AXI4 se utiliza para requerimientos de mapeo de memoria para alto rendimiento que permite llegar hasta transferencias de 256 datos por ciclo de reloj en una única fase de dirección. Luego está AXI4-Lite, un interfaz más ligero con una única transferencia por transacción que se utiliza para señales de control y estados de registros. Finalmente está el AXI4-Stream que elimina la necesidad de una fase de dirección y con ello permite sacar todo el rendimiento y máxima transferencia. Al no tener fase de dirección este tipo de AXI4 no se considera como mapeo de memoria y por lo tanto lleva a este tipo de interfaz a ser el más rápido.

En la presente plataforma la especificación que se ha decidido utilizar es la AXI-Lite y por lo tanto es la que se ha utilizado durante este proyecto enviando esta manera un bus de dato por ciclo de reloj. Debido a su menor complejidad con esta especificación se han utilizado direcciones de 32 bits para la memoria y 32 bits para los datos.

3.5.1 ¿Cómo funciona?

La comunicación es una comunicación a ráfagas en la cual está implementada a través de 5 canales tipo FIFO: leer dato, leer dirección, escribir dato, escribir dirección y respuesta a escritura. Las especificaciones AXI describen un interfaz entre un único AXI maestro y un AXI esclavo representando cores IP que intercambian información el uno con el otro. La información se puede mover en ambas direcciones entre el

maestro y el esclavo simultáneamente. Dentro de la NetFPGA tanto el PCIe como el MicroBlaze actúan como maestros y por lo tanto son capaces de leer y de escribir en todos los periféricos. A continuación se podrá ver una breve descripción sobre cada tipo de canal, señales de los interfaces de cada canal y dos ejemplos gráficos de dos tipos de canales [1]:

- En un primer grupo se pueden incluir los canales de lectura y escritura de direcciones. Cada transacción tanto de escritura como de lectura contienen sus propios canales para la dirección en los cuales se incluye toda la información permitente. En los cuales están las direcciones requeridas ya sea para escritura como para lectura como las señales de control de información.
- Canal de lectura de dato es en el que están tanto el canal para la lectura del dato como cualquier tipo de información como respuesta de lectura por parte del esclavo al maestro.
- Canal de escritura es en el que se transmite cualquier orden de escritura por parte del maestro al esclavo. En este canal existe tanto un bus con datos como un *strobe* indicando que bits son válidos.
- El canal de respuesta de escritura permite al esclavo una forma de comunicación con el maestro para transmitir señales de confirmación tras cada escritura.

Tabla 1: Señales de Interfaz del AXI lite [24]

Channel	Signal	Status	Description
write address	AWADDR	required	write address - see AXI spec.
	AWVALID	required	write address valid - see AXI spec.
	AWREADY	required	write address flow control - see AXI spec
write data	WDATA	required	write data - see AXI spec
	WSTRB	required	write data byte strobe - see AXI spec
	WVALID	required	write data valid - see AXI spec

	WREADY	required	write data flow control - see AXI spec
write response	BRESP	required	write response - see AXI spec
	BVALID	required	write response valid - see AXI spec
	BREADY	required	write response flow control - see AXI spec
read address	ARADDR	required	read address - see AXI spec
	ARVALID	required	read address valid - see AXI spec
	ARREADY	required	read address flow control - see AXI spec
read data	RDATA	required	read data - see AXI spec
	RRESP	required	read response - see AXI spec
	RVALID	required	read data valid - see AXI spec
	RREADY	required	read data flow control - see AXI spec
clock/reset	ACLK	required	clock - see AXI spec., can be shared between multiple interfaces on the same components
	ARESETN	required	low active reset - see AXI spec, can be shared between multiple interfaces on the same components

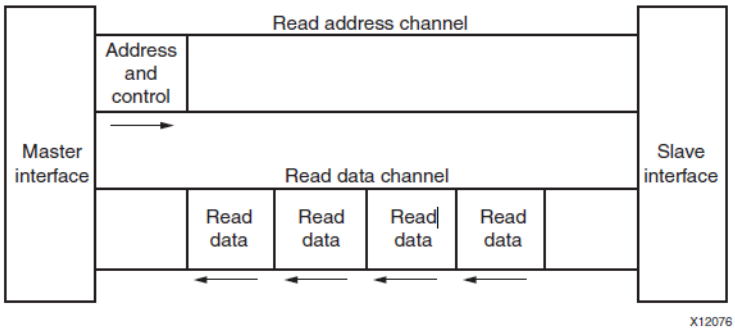


Figura 15: Canal de lectura AXI [4]

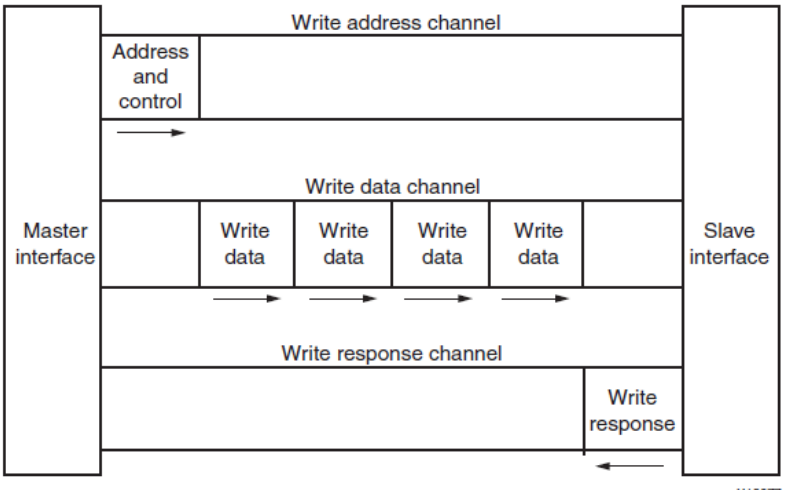


Figura 14: Canal de Escritura AXI [4]

Como se puede observar en la Tabla 1 han aparecido unas nuevas señales de un único bit de las ya descritas en la sección previa. Cada canal contiene tres señales extras, VALID, READY y LAST. El método de comunicación que utiliza el protocolo se rige bajo el concepto denominado "*handshaking*" que en castellano se traduce por dar la mano, entre el maestro y el esclavo. Esto quiere decir que tan solo habrá comunicación en el momento en que ambas señales tanto el VALID como el READY estén a alto. El extremo que va a enviar los datos una vez que haya preparados los mismo pasará a activar la señal de READY. Por otro lado, el otro extremo una vez que esté preparado a recibir los datos pasará a activar la señal de VALID, y solamente así será efectiva la transacción.

Por lo tanto esto nos lleva a tres casos posibles, primero que se active antes la señal de READY, segundo que se active antes la señal de VALID y finalmente que se activen simultáneamente ambas señales, esto será totalmente indiferente y el resultado solamente pasará cuando ambas estén alto. En la siguiente figura se pueden observar dos ejemplos de lectura y de escritura respectivamente. Por último está la señal de LAST la cual se activará cuando el último bloque de datos haya llegado indicando así que se finaliza el paquete.

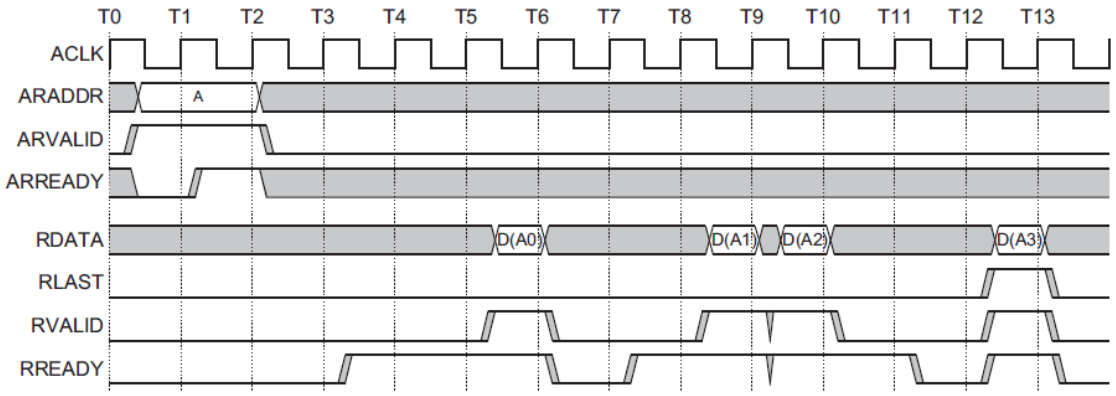


Figura 17: Ejemplo de lectura [4]

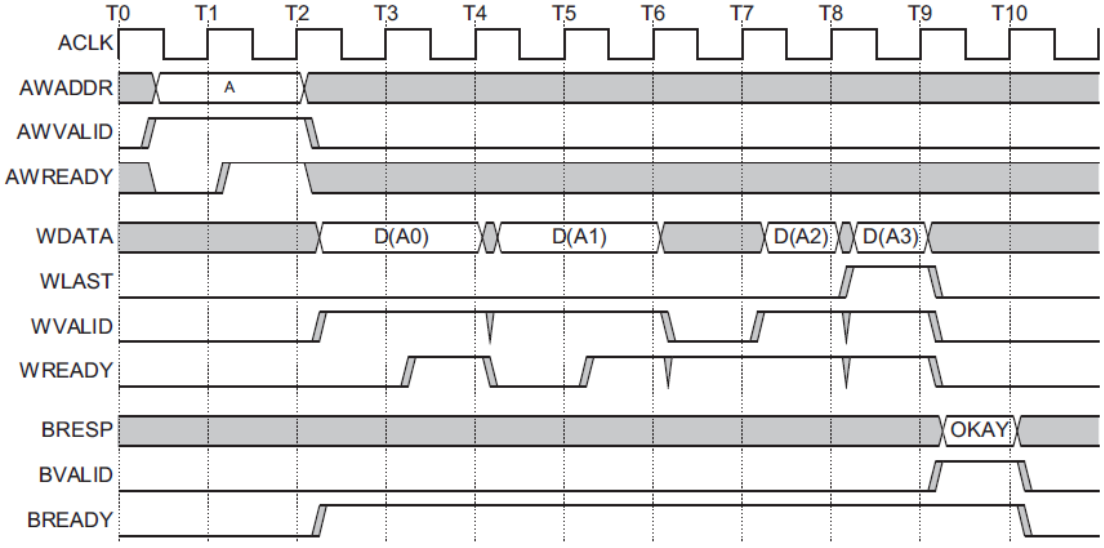


Figura 16: Ejemplo de escritura [4]

Finalmente queda describir como sería un bus de datos en un protocolo de interfaz tipo AXI. Dentro del bus existirán 6 señales además del *clock* y el *reset* común en cualquier señal dentro de la placa. 3 de ellas ya se sabe cuáles son, VALID, READY y LAST. Así que las 3 sobrante son el DATA de 256 bits aunque se puede modificar este valor, el STROBE de 32 bits y el USER fijo a 128 bits:

Tabla 2: Señales del bus de datos [24]

Signal	Bit Width	Description
TVALID	1	master to slave flow control signal - see AXI spec.
TDATA	C_DAT_DATA_WIDTH	data signal - see AXI spec.
TREADY	1	slave to master flow control signal - see AXI spec.
TSTRB	C_DAT_DATA_WIDTH/8	byte enable signal - see AXI spec.
TLAST	1	indicates end of packet - see AXI spec.
TUSER	fixed to 128	carries meta data in the first data beat of the packet

3.6 Primer Proyecto

Tras toda esta información sobre los conocimientos necesarios para entender un proyecto NetFPGA y sus conexiones se van a presentar en el primer proyecto que se creó a modo introductorio. La idea era crear un módulo que se conectaría dentro del proyecto ya existente de reference nic. Este módulo sería lo más sencillo posible dedicándose únicamente a leer una transacción de un paquete y dejarla pasar al otro lado. Lo que se va a hacer a continuación es mostrar paso a paso como se creó el proyecto y explicar las variables que se utilizaron dentro de Vivado HLS y luego ver todos los pasos hasta bajar el proyecto con el módulo integrado hasta la placa. Debido a la extensa librería de funciones y variables que existen dentro de Vivado HLS solo se explicarán aquellas que vayan saliendo en el código utilizado.

Tal y como se mencionó en el párrafo anterior el primer Pcore que se creó con la herramienta Vivado HLS fue el 'pass_through'. Este simplemente será una caja vacía en la cual le entrarán transacciones de un paquete y saldrán tal cual entran. Con esta caja vacía lo se pretendía hacer era pelearse por primera vez con la herramienta y conectarlo de manera correcta. Esto de fácil comprensión fue una de las tareas más duras de este proyecto pues fue la primera vez que se atajó un problema y se tuvieron que hacer todas las conexiones entre los módulos y las conexiones de este mismo módulo con las variables de entorno del programa. En la siguiente figura se puede ver gráficamente donde se colocaría el pcore y se vería que no afecta para nada al funcionamiento del proyecto reference nic.

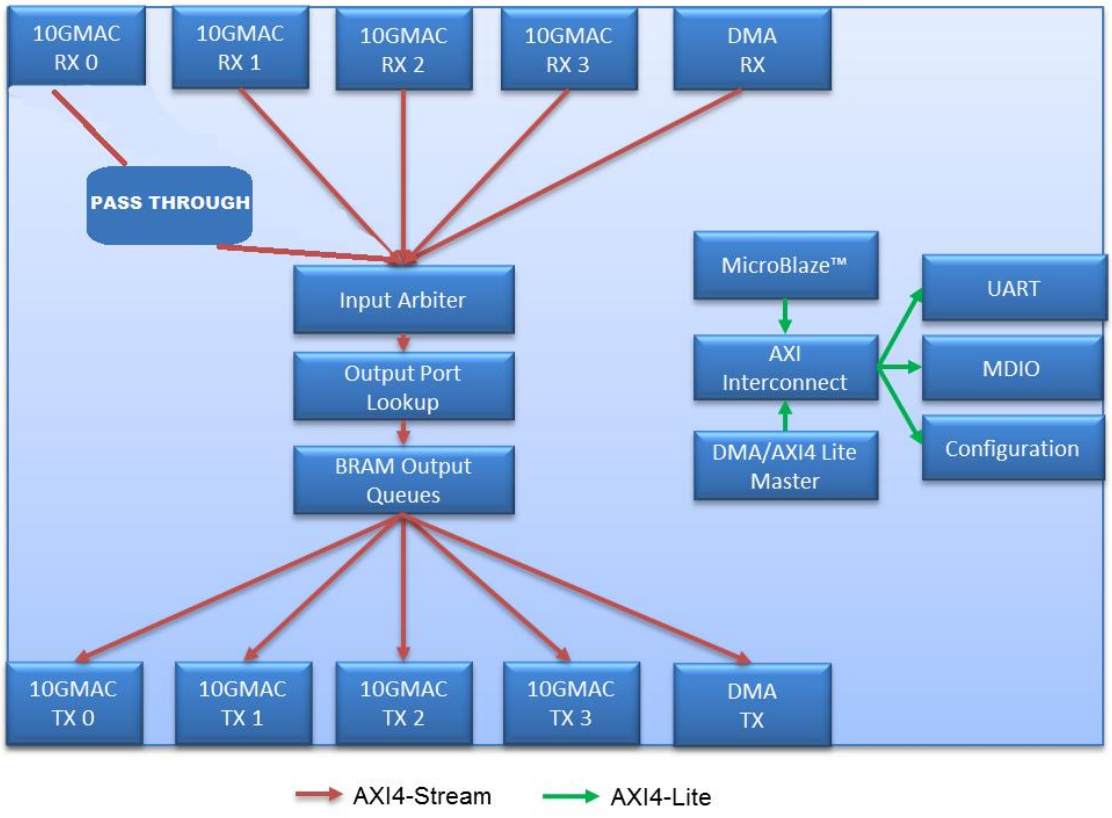


Figura 18: Reference nic con pass through

Como se aprecia se decidió insertar el Pcore en el camino del primer interfaz 10G. Para desarrollar este Pcrore se creó un proyecto nuevo en Vivado HLS donde se escribiría un código en C donde simplemente hubiese una asignación de punteros a una variable ya previamente definida que tuviese forma de un AXI. Por ello el primer paso fue crear una estructura tipo AXI:

```
template<int D,int U>
struct my_axis{
    ap_uint<D>   data;
    ap_uint<D/8> strb;
    ap_uint<U>   user;
    ap_uint<1>   last;
  };
typedef my_axis<ETH_INTERFACE_WIDTH,USER_ETH_INTERFACE> axi_interface_type;
```

Como se puede observar lo primero que llama la atención es que faltan las señales READY y VALID. Estás se crean automáticamente debido a la aplicación de *pragmas* tipo AXI a esta estructura. El tamaño del bus de data, strb y user son los definidos por la plataforma NetFPGA 10G:

```
#define USER_ETH_INTERFACE 128
#define ETH_INTERFACE_WIDTH 256
```

Una vez definida la estructura correctamente se pasó a escribir el código principal. En sí el código será de una sencillez máxima pues tan solo será una asignación de punteros como se podrá ver más adelante. A la función principal se le pasará como argumento dos punteros a la estructura AXI siendo uno la entrada de las transacciones de cada paquete y otro para la salida. Al hacer simples asignaciones no habrá que preocuparse de cada señal dentro de cada AXI pues al ser dos estructuras iguales se igualarán una a una.

```
void pass_through (axi_interface_type *data_input, axi_interface_type *data_output)
```

Tras esta declaración se pasarán a definir los *pragmas*. Los *pragmas* son directivas que se utilizan en el lenguaje C de programación cuya función es dar información extra al compilador más allá de lo que implica en el lenguaje en sí [22]. Para este primer acercamiento se utilizarán dos directivas distintas la primera para indicar al compilador que las variables que se van a utilizar son colas de tipo FIFO y por otro lado que su mascará exterior o interfaz será del tipo AXI *Stream*. Con esta directiva es donde se crean automáticamente las señales de READY y VALID según la disponibilidad de las colas FIFO.

```
//Tell the compiler this interfaces talk to FIFOs
#pragma HLS INTERFACE ap_fifo port=data_input
#pragma HLS INTERFACE ap_fifo port=data_output

//Map HLS ports to AXI interfaces. Tell the compiler this interfaces are AXI4-Stream
#pragma HLS RESOURCE variable=data_input core=AXIS metadata="-bus_bundle
STREAM_A"
#pragma HLS RESOURCE variable=data_output core=AXIS metadata="-bus_bundle
STREAM_B"
```

Finalmente solo quedará escribir el código principal del módulo que como ya se vaticinó será muy sencillo, una asignación de punteros.

```
*data_output = *data_input;
```

En este paso tanto *data_input* como *data_output* ya están declarados como colas FIFO con un interfaz compatible a los buses AXI y su conexión con el resto de módulos será plausible. Tras este sencillo código con la herramienta de Vivado HLS, en un primer paso se compilará el programa y se exportará su diseño a un módulo

RTL, el cual se podrá utilizar con el programa de Xilinx EDK en donde se cargará el proyecto reference nic. Por ello el siguiente paso a seguir será el de abrir el programa de Xilinx EDK y cargar el proyecto de reference nic. A continuación habrá que conectar correctamente nuestro módulo dentro del diseño a través del interfaz gráfico. Esto se explicará con detalles más adelante paso a paso. Por ahora bastará con decir que el módulo quedará conectado, se le añadirán las señales del reloj y del *reset* y se procederá a la construcción del archivo de carga para el MicroBlaze. Una vez se termine de generar el archivo se abrirá el SDK de Xilinx en el cual no hará falta cambiar nada. Esto se debe a que las soluciones que aquí se presentan no interactúan con el MicroBlaze y no se le pide sacar nada por pantalla. Por lo tanto se cargará junto a nuestra solución hardware la solución que la plataforma NetFPGA proporciona y se genera el archivo download.bit que se bajará finalmente a la placa. Una vez más en la siguiente sección se procederá a una descripción más detallada.

Ya generado el download.bit habrá que cargar la placa mediante la herramienta de Xilinx Impact, conectar las fibras con sus conectores correspondientes tanto a la placa a los puertos Ethernet como a los puertos de la tarjeta NIC de la otra máquina. Se reinicia el ordenador y se observará como al enviar paquetes a través del camino donde se introdujo el módulo no hay diferencia alguna ni se aprecian retardos.

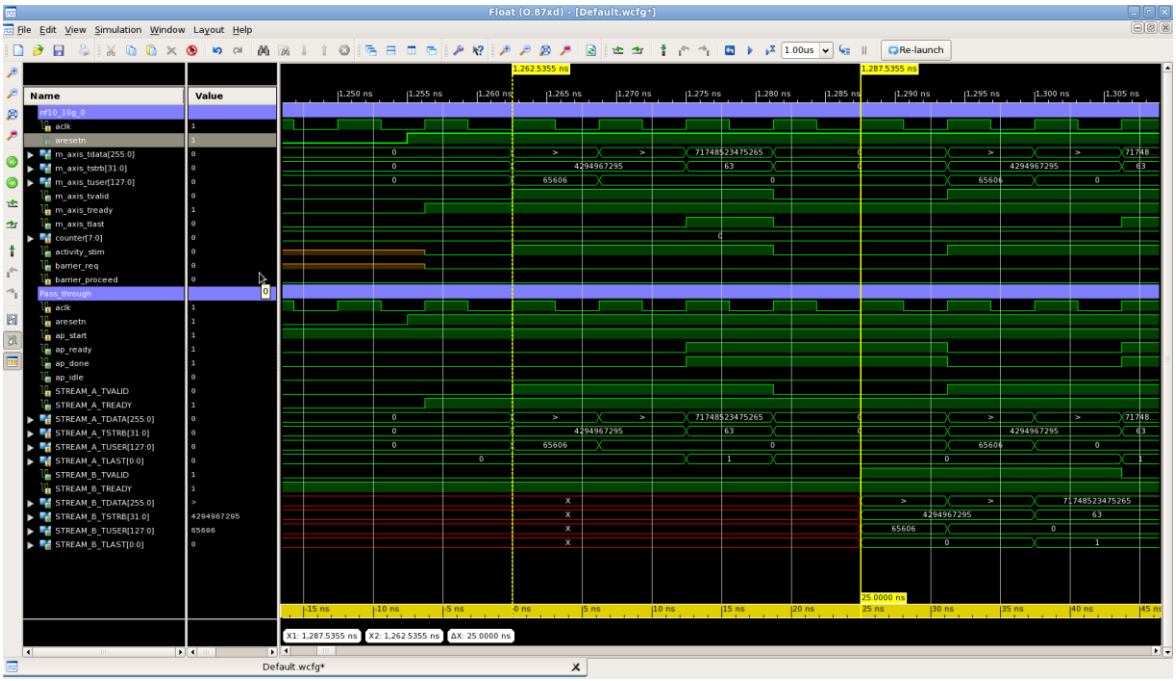


Figura 19: Simulación del módulo Pass_through usando Modelsim

De la figura anterior se puede observar como el módulo hace exactamente lo que deseamos, simplemente lo que le entra lo saca por la salida, sin modificar ningún valor ni introducir ninguna lógica. Los paquetes al pasar por el módulo creado se les introducen un retardo insignificante de unos 25ns. También por primera vez se puede observar el funcionamiento de las señales de control READY, VALID y LAST y un paquete entero con las características que tiene viendo como se transmite en un AXI STREAM.

3.6.1 Conexión con el Micorblaze

Tras haber creado el primer módulo y antes de ponerse con la solución del problema se desarrolló el módulo ya creado y se conectó con el MicroBlaze. Básicamente el código es el mismo salvo unas pequeñas adaptaciones para permitir la comunicación. Donde estará la mayor diferencia será a la hora de modificar el código del proyecto reference nic tras haber creado las instrucciones a nivel hardware y se crean las instrucciones del MicroBlaze.

Los cambios pertinentes se realizaron para que el MicroBlaze comunicase por pantalla, gracias a la conexión de un cable RS-232, la cantidad de transacciones de un paquete que pasan a través del módulo en cuestión e imprimir un dato fijo. Como ya se indicó se partía desde el código anterior por lo tanto la esencia no varía. La primera modificación que se aprecia aparece en la cabecera donde obviamente habrá dos variables nuevas, el contador de paquetes y el dato que se imprimirá por pantalla:

```
void pass_through_ublaze (axi_interface_type *data_input, axi_interface_type
*data_output, uint32_t *numPackets, uint32_t *unNum){
```

Estas nuevas variables obviamente no tendrán formato AXI si no que serán simples variables tipo int. La mayor diferencia viene a continuación donde se le indica al compilador mediante un nuevo *pragma* que las siguientes variables deberán de estar conectadas con el MicroBlaze. Gracias a ello será posible tanto enviar información como recibirla:

```
#pragma HLS RESOURCE variable=numPackets core=AXI4LiteS
#pragma HLS RESOURCE variable=unNum core=AXI4LiteS
```

La palabra clave en esta línea de código es core, es decir, que ambas variables estarán conectadas con el core el cual será el que se comunique con el MicroBlaze.

Una vez creadas las variables con el formato correcto solo habrá que poner el simple código que realice lo ya indicado:

```
*numPackets = (_pross_pkts++);
*unNum = 0xFAFAF0F0;
```

Siendo *pross_pkts* una variable local de la función inicializada previamente. Los siguientes pasos de conexión hardware serán exactamente los mismos que en el caso anterior y que se explicará en detalle en las siguientes secciones. La novedad aparece a la hora de realizar el último paso cuando se han de generar las instrucciones software.

El siguiente paso será conectar este módulo con el resto de módulos. El sitio indicado será el mismo que en el caso anterior en el primer interfaz antes del *input arbiter*. Pero esta vez mediante el interfaz gráfico de Xilinx XPS no solo hay que hacer las conexiones con los AXI *Stream* para el módulo del interfaz 0 y el *input arbiter* sino que también habrá que conectar un AXI4-LITE con el bus AXI general que se comunica con el MicroBlaze. Esta es la diferencia que introduce el generar variables que se comuniquen con el core, que obviamente al hacer la conexión hardware habrá que crear una conexión más. Por lo tanto una vez generado las instrucciones hardware donde viene indicado las conexiones mediante la herramienta XPS, se tendrá que crear un nuevo proyecto en Xilinx SDK. Este proyecto tendrá que ser el más sencillo, pues lo que se pretende hacer es coger el que viene con la plataforma y modificarlo. Por lo tanto una vez creado el proyecto se carga el original de la plataforma NetFPGA. Como se puede apreciar ya vendrán definidas distintas tipos de funciones correspondientes a aquellas variables que se declararon con el *pragma* tipo core y funciones correspondientes al módulo que se acaba de generar. Al final tras todas las declaraciones e instrucciones por defecto lo que se hizo fue primero declarar una variable que hablaría con el MicroBlaze y darle valores con la dirección base del módulo al que conectarse con, en este caso el 'pass_through_ublaze'.

```
XPass_through_ublaze Instance_XPass;
 XPass_through_ublaze *InstancePtr = &Instance_XPass;

InstancePtr->Axi4lites_BaseAddress=
XPAR_PASS_THROUGH_UBLAZE_TOP_0_S_AXI_AXI4LITES_BASEADDR;
InstancePtr->IsReady = XIL_COMPONENT_IS_READY;
```

Después el paso siguiente será el de conectar el módulo y dejarlo en funcionamiento, para ello inicialmente habrá que decirle que una vez que esté activado se mantenga activado. Eso se consigue con la siguiente función:

```
XPass_through_ublaze_EnableAutoRestart(InstancePtr);
```

Una vez activado ese bit habrá que inicializar el módulo:

```
XPass_through_ublaze_Start(InstancePtr);
```

Tras estas dos líneas de código ya se pasaría imprimir por pantalla el resultado deseado, esto se hizo de la siguiente manera:

```
while (1){

    int NumPack = XPass_through_ublaze_GetNumpackets(InstancePtr);
    int UnNum = XPass_through_ublaze_GetUnnum(InstancePtr);
    xil_printf("NumPack: %d; UnMun %d\r", NumPack, UnNum);
}
```

Finalmente se compilará el código para asegurarse de que no haya alguna errata y se creará el download.bit y se bajará a la placa con la herramienta Impact de Xilinx. Gracias a la conexión del cable RS-232 el cual con un conversor a USB nos permite conectarlo a la máquina y se es capaz de monitorizar lo que pasa en el MicroBlaze. Para ello se debe utilizar alguno de los programas disponibles ya sea el *minicom* o el *Putty*. Para este ejercicio se utilizo el *Putty* y a continuación se puede observar el resultado en dos imágenes. En la segunda figura se ve como el conteo de paquetes aumenta de 4 en 4 en vez de en uno en uno. Esto se debe a que cada paquete son 4 transacciones y la información llegue al MicroBlaze cada vez que se transmite un paquete entero y por ello suma de 4 en 4.

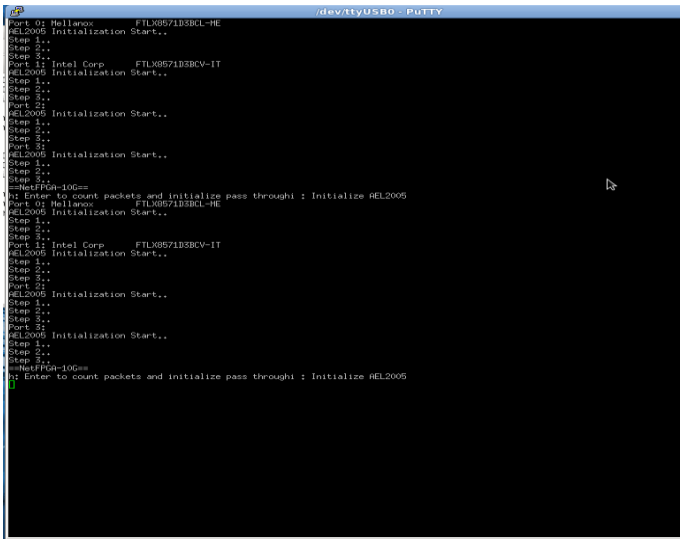


Figura 21: Inicialización del módulo pass through

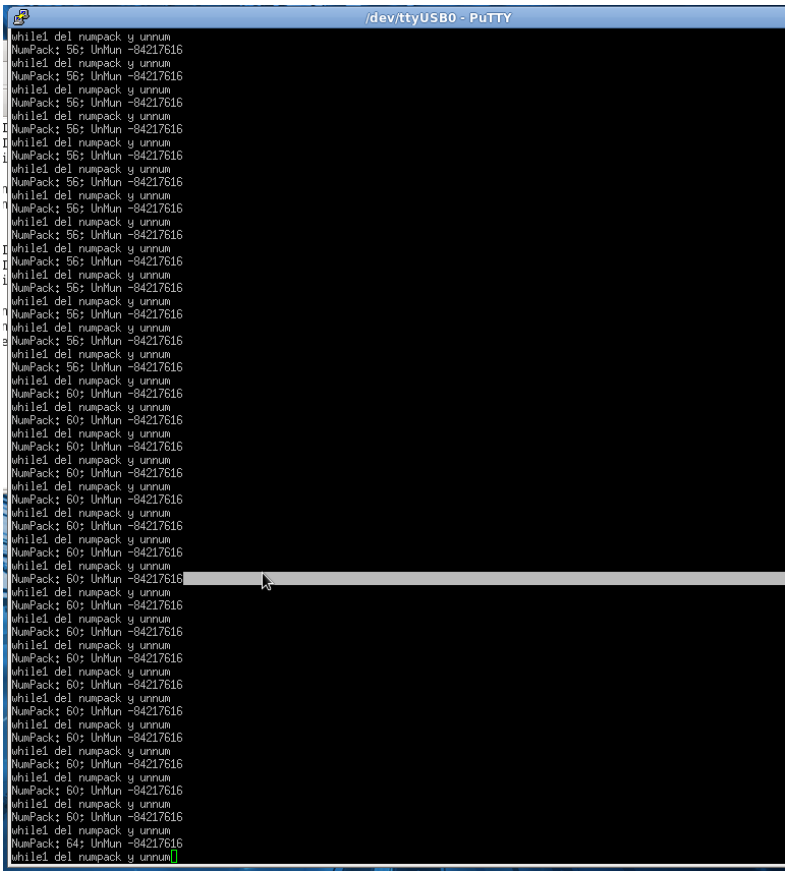


Figura 20: Contabilización de paquetes

4 Desarrollo

4.1 Objetivo

Una vez ya se han asimilado todos los conocimientos necesarios para entender un proyecto NetFPGA, crear módulos, conectarlos y bajar el diseño a la placa se va a proceder a explicar el objetivo de este proyecto.

Para este proyecto se propuso implementar mediante lenguaje de alto nivel uno de los módulos ya existentes dentro del proyecto reference nic. El módulo elegido fue el del *input arbiter*. Se llegó a esta conclusión ya que este módulo contiene conexiones con otros módulos, cuatro conexiones con los puertos Ethernet, uno con el interfaz DMA y luego se conecta con el *Output Port Lookup* o tabla de rutas y además contiene una lógica interna bastante sencilla. A pesar de ser sencilla la lógica interna se eligió este módulo pues una vez implantado en lenguaje de alto nivel permite desarrollar un poco la lógica haciéndolo más complejo y así hacer ver las ventajas que existe al realizar un diseño en alto nivel.

4.1.1 Input Arbiter

El *input arbiter* tiene cinco interfaces de entrada, cuatro de los 10G MAC y uno para el módulo DMA. Cada entrada del arbitrario se conecta con una cola de entrada tipo FIFO, y después mediante un decisor tipo Round-Robin cogerá una cola no vacía y escribirá el paquete entero hacia el siguiente módulo.

A la hora de implementar este código en lo que se habrá de tener especial cuidado es en principalmente conectar cada AXI correctamente y luego generar una lógica equivalente que no afecte el resto de funcionamiento del proyecto. Permitiendo así tanto enviar como recibir paquetes.

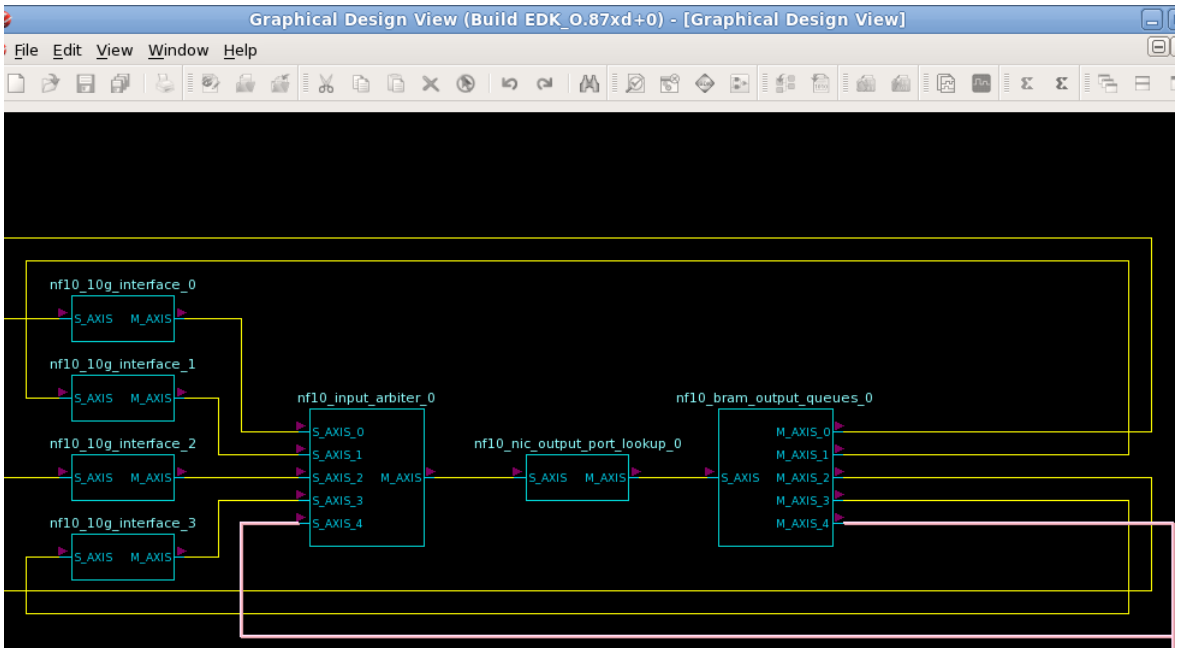


Figura 22: Input Arbiter

En la figura los *streams* que están subrayados en color rosita son aquellos que van y provienen del DMA. Aquí se puede ver una vista gráfica del proyecto viendo la diferencia entre conexiones tipo *master* y *slaves*. Hágase notar que falta toda la conexión del DMA y los controladores generales del MicroBlaze.

4.2 Planteamiento de la solución

Para encontrar una solución se tuvo que ir a los códigos Verliog del proyecto reference nic en donde se explicaba el funcionamiento. Al ser estos bastante complicados y algo indescifrables para hallar la solución se optó por atender a la descripción del módulo y en base a ella hallar una solución en alto nivel.

La solución que se ha propuesto consiste en dividir el proceso del *input arbiter* en varios módulos. Esto se ha planteado intentando seguir el concepto de sencillez que se ha ido abstrayendo de las lecturas de la plataforma NetFPGA. Teniendo claro el concepto del módulo en si mismo se propuso para cada camino, ya sea el de los cuatro interfaces Ethernet como el del DMA un módulo que guarde todas las transacciones de un mismo paquete hasta que llegue la última transición y encole en un cola el paquete entero poniendo a uno la *flag* de READY. De esta manera se está creando un módulo cuya función sea la de leer paquetes enteros y encolarlos en una cola tipo FIFO. Este módulo se insertará tras cada interfaz Ethernet y el DMA. Por lo tanto al tener todos los paquetes encolados simplemente habrá que crear un módulo que haga de decisor siguiendo el planificador Round-Robin. Esté módulo consistirá en un decisor preguntando a cada cola si está vacía o no. En caso de que lo esté irá a la siguiente cola y en caso de que no lo esté recibirá cada transición de paquete desde la cola hasta la transición con la *flag* de *last* activada a uno. Con el paquete entero lo pondrá en su propia cola de salida indicando al interfaz con el siguiente módulo que tiene un paquete listo para la transmisión. Esto como se sabrá se obtiene mediante el juego de las banderas.

A continuación se verán dos figuras las cuales intentarán explicar de manera gráfica esta solución que se acaba de plantear. En la primera imagen se observa la conexión final de los módulos creados con el resto de módulos del proyecto reference nic. Este proceso de conexión se explicará más detalladamente en las secciones venideras. Por otro lado en la siguiente figura, se ve un dibujo esquemático de la solución a implementar. Por un lado las cinco colas FIFO que albergan los paquetes enteros (en la imagen cada icono representa un paquete entero) los cuales han sido generados gracias a la lectura de todas las transacciones. Y luego se ve la cola final con el decisor representado con las flechas. Como se puede observar con los dos iconos en la primera cola, el decisor al ir a la primera cola y ver que es no vació cogerá el primer paquete, tras su traspaso a pesar de haber otro paquete en la cola esperando acudirá

a la siguiente cola en donde cogerá el siguiente paquete, y solo tras haber realizado un barrido por todas las colas acudirá a esta primera una vez más para recoger el paquete pendiente. Lo que se quiere dar a entender con este dibujo esquemático es tal y como se ha explicado anteriormente el funcionamiento sin prioridades del método Round-Robin.

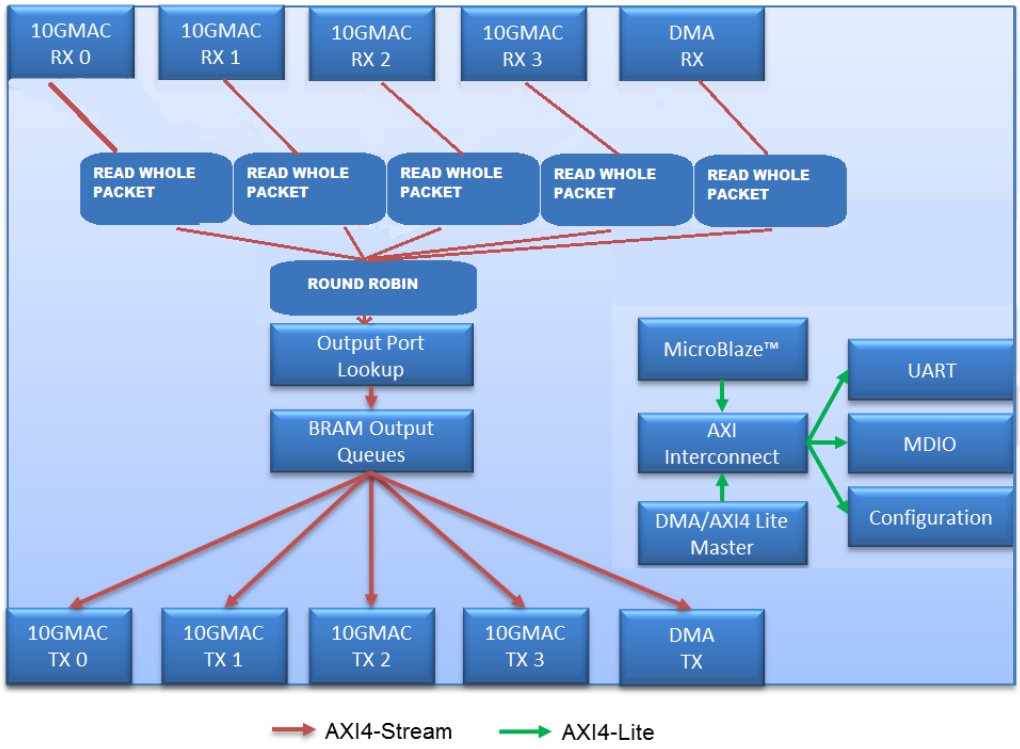


Figura 24: Reference NIC con módulos creados en Alto nivel

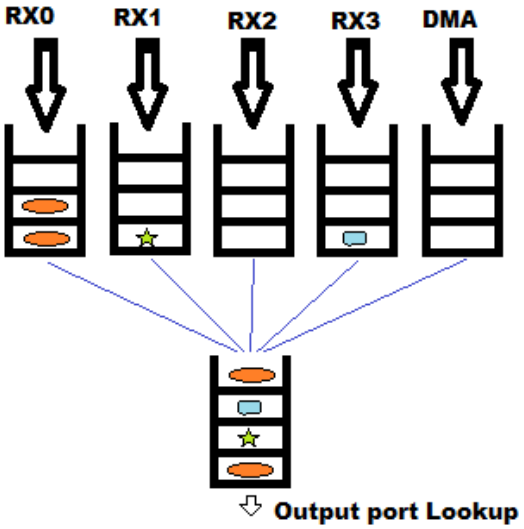


Figura 23: Método grafico de Round Robin

4.3 Solución

En esta sección se procederá a dar una explicación detallada de todos los pasos realizados para la solución del objetivo presentado. A diferencia de la sección 3.6 donde se prestó más atención en introducir el concepto del Pcore y su integración en un proyecto NetFPGA ahora se ahora un análisis exhaustivo indicando todos los pasos que se siguieron. Se introducirá el código creado para hacer énfasis en el ahorro de líneas de código lo cual se ampliará en la sección de resultados.

El primer paso fue crear un proyecto en Vivado HLS para crear el código que generaría el primer módulo. Este módulo tendrá semejanzas con el primero que se explicó en esta memoria pues no tendrá comunicaciones con el MicroBlaze y por lo tanto no habrá que modificar el código en C de las instrucciones software. Este primer módulo como ya se explicó en la sección anterior consistirá en dos bucles, uno de lectura y otro de salida. Ambos bucles se irán recorriendo hasta que se llegue al final de un paquete y por ello el paquete completo se haya albergado o transmitido. La cabecera de este módulo será bien sencilla incluyendo dos variables de la estructura creada de tipo AXI una de entrada y otra de salida.

```
void read_packet(axi_interface_type data_input[SIZE], axi_interface_type
data_output[SIZE]){
```

En este caso en vez de ser punteros a la estructura son tablas de tamaño fijo las cuales se pueden redefinir según la variable SIZE. Debido a que para albergar el paquete entero se han de guardar todas las transacciones que lo componen se decidió crear una tabla de tipo estructura axi_interface_type. El siguiente paso es el mismo que en el primer ejemplo y es el de definir los *pragmas* para el compilador indicando que se van a utilizar colas tipo FIFO y que estas colas tendrán un interfaz tipo AXI-Stream para así permitir futuras conexiones.

```
//Tell the compiler this interfaces talk to FIFOs
#pragma HLS INTERFACE ap_fifo port=data_input
#pragma HLS INTERFACE ap_fifo port=data_output

//Map HLS ports to AXI interfaces. Tell the compiler this interfaces are AXI4-Stream
#pragma HLS RESOURCE variable=data_input core=AXIS metadata="-bus_bundle
STREAM_A"
#pragma HLS RESOURCE variable=data_output core=AXIS metadata="-bus_bundle
STREAM_B"
```

En este caso hará falta el uso de una variable interna también del mismo tipo, axi_interface_type para que se vaya guardan las transacciones intermedias.

```
    axi_interface_type data_read[200];
    int j, i=0;

                    loop_in: do{
                            data_read[i] = *data_input;
                            data_input++;
                    }while(!(data_read[i++].last));


                    if(i<MAX_SIZE-1)
                            loop_out: for(j=0;j<i;j++){
#pragma HLS PIPELINE
                                    *data_output = data_read[j];
                                     data_output++;
                    }
```

Como se mencionó esa variable intermedia, en este caso se llama data_read, se está guardando el paquete entero y luego se envía a la salida. De esta manera el paquete se envía entero en vez de enviar transacción a transacción y sobre todo y más importante hace paso intermedio para el decisor. Durante este código aparece un nuevo *pragma*:

```
#pragma HLS PIPELINE
```

Con este *pragma* se permite la opción de ejecutar varias operaciones de manera paralela. En Vivado HLS la lectura tarda dos ciclos de reloj debido a que la RAM en un primer ciclo leerá el dato y en el segundo lo escribirá por ello insertando este *pragma* el dato ya estará preparado y habrá un ahorro en ciclos de reloj. A continuación se pueden observar las mejoras debidas al introducir el *pragma* tipo pipeline en cuanto a ciclos de reloj de los bucles *loop_in y loop_out*:

Tabla 3: Mejoras tras el uso del *Pragma* Pipeline

Loop Name	Latency Min	Latency Max	Iteration Latency	Initiation achieved	Interval target	Trip Count	Pipelined
loop_in	?	?	1	-	-	?	no
loop_out	?/ ?	?/ ?	2/2	-/1	-/1	?/ ?	no/sí

Como se puede ver en la tabla hay una mejora de uno. El objetivo era mejorar un ciclo y como se observar en el objetivo conseguido es de un ciclo de reloj. Además de reducir en un ciclo de reloj por ciclo cabe destacar la reducción de Flip Flops utilizados

en el diseño, pasando de 193 sin el pipeline a 101 con él y se consiguió rebajar el número de multiplexores utilizados de 159 a 128.

Una vez finalizado este módulo se pasó a realizar el decisor tipo Round-Robin. Como en el paso anterior se volvió a crear un proyecto en Vivado HLS y se declaró la cabecera de la función a realizar.

```
void round_robin (hls::stream<axi_interface_type> &data_input_nf0,
hls::stream<axi_interface_type> &data_input_nf1, hls::stream<axi_interface_type>
&data_input_nf2, hls::stream<axi_interface_type> &data_input_nf3,
hls::stream<axi_interface_type> &data_input_nf4, hls::stream<axi_interface_type>
&data_output){
```

Como cabía de esperar habrá cinco entradas y una salida denominadas de la manera que se puede ver. En cuanto a diferencias se encuentran varias, se puede observar que se sigue utilizando la misma estructura del axi_interface_type pero esta vez está envuelto en una nueva clase. Esta clase es particular de Vivado HLS y se denomina *Streams*. Son unas clases de C++ ideales para moldear hardware y llevan en sí mismo una implementación de una cola tipo FIFO. Por lo tanto utilizando este interfaz se ahorraría el uso de declarar el *pragma* para crear FIFOs y los únicos *pragmas* que se tendrían que declarar son los anteriores que indican al compilador que se utilizan interfaces tipo AXI *Stream*.

```
//Map HLS ports to AXI interfaces. Tell the compiler this interfaces are AXI4-Stream
#pragma HLS RESOURCE variable=data_input_nf0 core=AXIS metadata="-bus_bundle
STREAM_IN_0"
#pragma HLS RESOURCE variable=data_input_nf1 core=AXIS metadata="-bus_bundle
STREAM_IN_1"
#pragma HLS RESOURCE variable=data_input_nf2 core=AXIS metadata="-bus_bundle
STREAM_IN_2"
#pragma HLS RESOURCE variable=data_input_nf3 core=AXIS metadata="-bus_bundle
STREAM_IN_3"
#pragma HLS RESOURCE variable=data_input_nf4 core=AXIS metadata="-bus_bundle
STREAM_IN_4"
#pragma HLS RESOURCE variable=data_output core=AXIS metadata="-bus_bundle
STREAM_OUT"
```

Se introdujo esta nueva clase de interfaz por la necesidad de poder hacer consultas al estado de una cola. Si se hubiese utilizado el tipo de interfaz anterior no había una manera directa de preguntar a la cola si tenía o no tenía dato. La primera propuesta que se usó fue utilizando el mismo interfaz que en el caso anterior y viendo a ver si en la cola a la cual se iba a consultar el *Stream* que venía en su sección de datos era o no distinta de NULL. Esto presentaba el problema de que al acceder a consultar esta variable dentro de la estructura introducía un retardo el cual a la hora de asignar el

paquete entero a la cola de salida como no estaba sincronizado el dato siempre se cogía el dato de la transacción siguiente. La conclusión a la que se llegó fue de utilizar esta estructura de *Stream* particular de Vivado HLS. Una de las peculiaridades que viene es que permite consultar si la cola FIFO está vacía o no. Para más información por favor diríjase al Anexo A donde encontrará una información más detallada del *Stream*.

La manera de realizar el método Round-Robin mediante el código en alto nivel fue la de realizarlo en dos pasos separados siguiendo así el concepto de sencillez. En un primer paso se iría preguntando cola a cola o interfaz a interfaz si estaba vacía o no, en caso de que no estuviese vacía si añadiría un valor determinado a una variable de control e inmediatamente se saldría del bucle para ir al paso siguiente. En caso de que la cola sí estuviese vacía se procedería a preguntar a la siguiente cola y así hasta encontrar la cola no vacía.

```
loop_queues: for(i=0;i<5;i++){
                    queue=0;
                            switch (i){
                                    case 0: if(!data_input_nf0.empty()) queue=1; break;
                                    case 1: if(!data_input_nf1.empty()) queue=2; break;
                                    case 2: if(!data_input_nf2.empty()) queue=3; break;
                                    case 3: if(!data_input_nf3.empty()) queue=4; break;
                                    case 4: if(!data_input_nf4.empty()) queue=5; break;
                            }
```

En este trozo de código se puede ver como con unas simples líneas de código se ha podido crear un método que vaya cola a cola preguntando si es o no vacía. Una vez ya sabiendo cual es la cola de la cual se va a extraer el paquete se procederá al paso siguiente. En este siguiente proceso dentro de la misma función se extraerá el paquete entero albergado en el módulo anterior y se encolará en una cola de salida.

```
if (queue!=0){
            loop_read_packet: do{
                    switch (queue){
                            case 1: read[j++] = data_input_nf0.read(); break;
                            case 2: read[j++] = data_input_nf1.read(); break;
                            case 3: read[j++] = data_input_nf2.read(); break;
                            case 4: read[j++] = data_input_nf3.read(); break;
                            case 5: read[j++] = data_input_nf4.read(); break;
                            }
                            data_output.write(read[k]);
                    } while(!(read[k++].last));
                            }
            }
```

Como se puede observar según el valor de la variable de control la cola de salida se conectará con una cola de entrada u otra. Una vez más como en el modelo anterior el

paquete se enviará al completo y no se volverá al decisor hasta que se haya enviado la última transición, es decir hasta que la variable de control *LAST* este a uno. Se utilizará un *Stream* intermedio tipo AXI para albergar de manera temporal una única transición y esta enviará a la salida. Esto se tuvo que hacer al presentar varios problemas el asignar la salida a varias entradas distintas. Al programar en un módulo HLS tiene que haber una única declaración de la salida para que no haya confusiones.

Una vez sintetizados ambos módulos en alto nivel será el turno de crear el módulo de hardware o también conocido como el Pcore. Gracias a esta herramienta con solo hacer *click* en el botón "Export RTL":

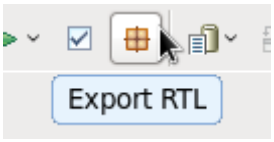


Figura 25: Exportar RTL, Vivado HLS

Dentro de la carpeta del proyecto de Vivado HLS en la parte de implementación se habrá creado el Pcore que se podrá manejar en el XPS. Estos dos pcores que se habrán generado se tendrán que llevar a la carpeta del proyecto de referenc nic dentro de NetFPGA 10G live en la sección de HW y luego pcore. De esta manera con el XPS aparecerán estos módulos para poder integrarlos en el sistema. Así abriendo el XPS y apretando al botón "*Rescan User IP repositories*" aparecerán los módulos creados en la lista de módulos disponibles. Como ya se habrá cargado el archivo system.xmp en el Xilinx XPS se podrá empezar a modificar el proyecto. El primer paso será remover el módulo *input arbiter* del proyecto, tras aceptar los mensajes que aparecen con los que se acepta que se desea borrar tanto la instancia del módulo como todas sus conexiones se insertaran los módulos creados. Habrá que insertar cinco módulos de read_packet y uno del decisor round_robin con ello se habrán insertado al proyecto los módulos pero quedará conectarlos entre sí y con el resto de módulos. En la siguiente imagen se puede observar la ventana que aparece a la hora de insertar un Pcore en el proyecto:

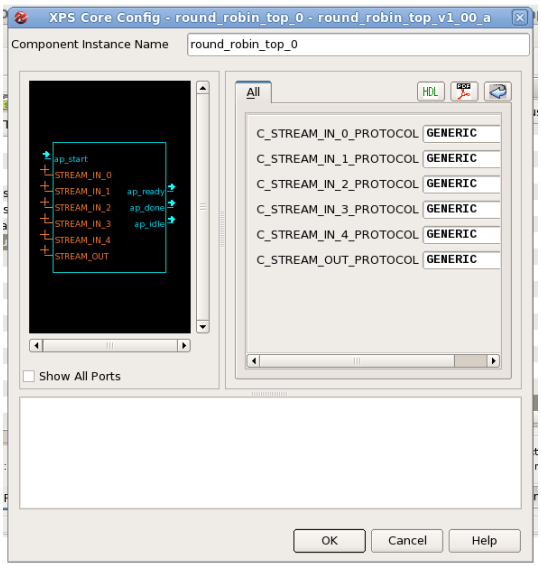


Figura 26: Inserción del Pcore Round Robin

Para la conexión de los módulos del tipo read_paccket tan solo tendrán una entrada libre “Stream_A” y luego por otro lado la salida “Stream_B” la cual se conecta automáticamente a un stream general al ser un bus tipo Master, por ello para establecer una conexión correcta habrá que conectar al “Stream_A” a el bus general correspondiente tipo Master de algún módulo superior y por otro lado el módulo siguiente que en este caso será el “Round_Robin” a una de sus entradas habrá que conectarle el bus general de la salida del Pcore “Read_Packet”. De esta manera se crea el ya explicado ‘*Handshaking’* entre un interfaz tipo Master y otro tipo Slave. Por lo tanto el primer módulo se conectará con el *Master Stream* de salida del interfaz Eth0 el segundo con el Eth1, el tercero con el Eth2, el cuarto con el Eth3 y finalmente el quinto con el DMA. El paso siguiente será hacer lo correspondiente con módulo Round-Robin, sus cinco entradas se conectaran a cada una de los *Streams* de salida de los módulos ya insertados. Como en el paso anterior la salida estará a un Master *Stream* a la cual se le ‘enganchará’ la entrada del siguiente módulo del diseño que ya se mostró, el *Output port lookup*. Esto mismo se puede observar en la siguiente figura:

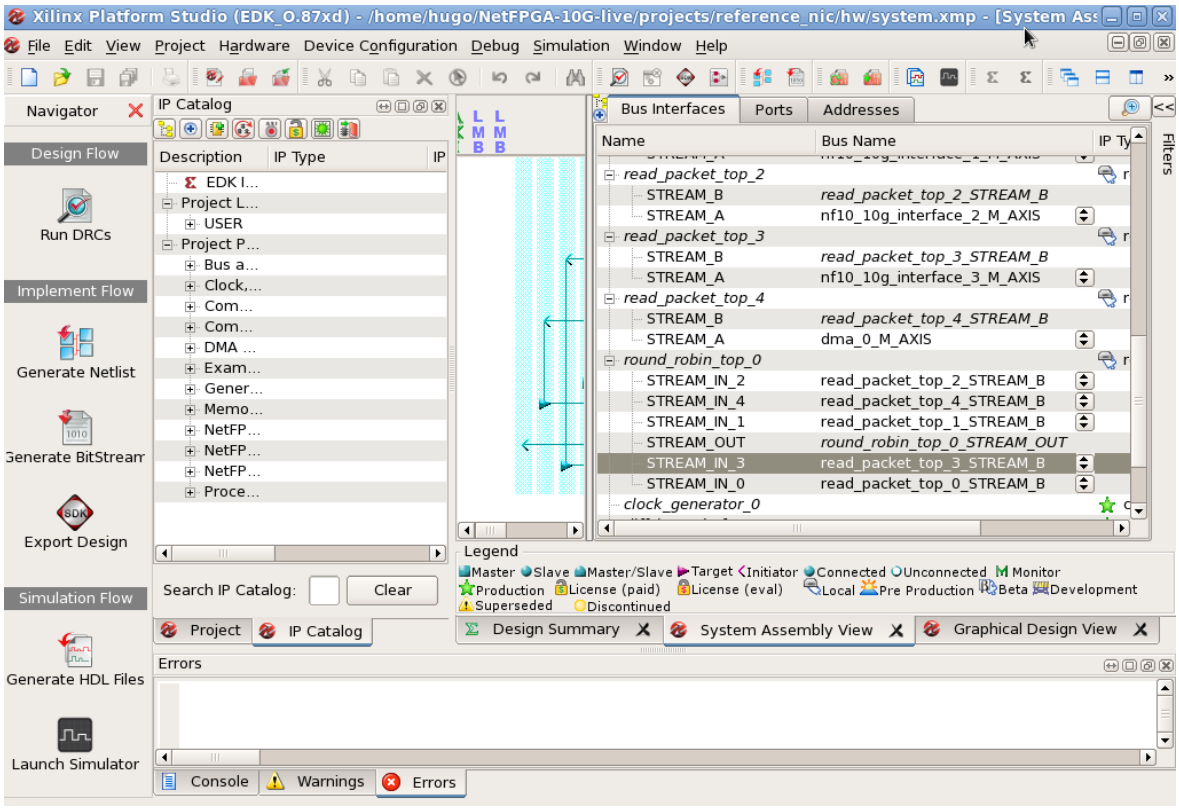


Figura 27: Captura de pantalla XPS, conexión de módulos

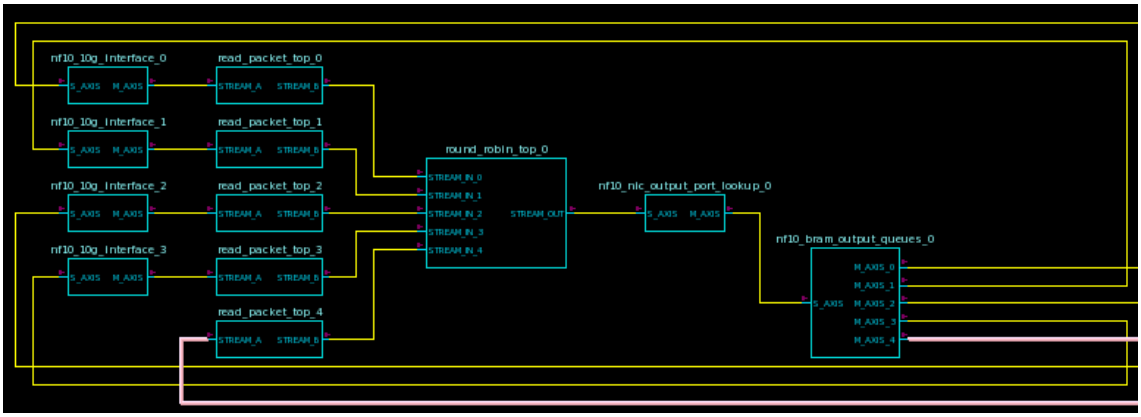


Figura 28: Vista Gráfica del proyecto

En la imagen se puede observar como quedaría la conexión final teniendo en cuenta solo las conexiones AXI y estando en rosa los *Streams* del DMA. Ahora ya estarán conectados entre sí todos los módulos y se procederá a conectar las variables globales del proyecto, el reloj y el *reset*.

Existen dos maneras de conectar los módulos a las variables globales. La primera es mediante el interfaz gráfico proporcionado por Xilinx XPS en la pestaña de '*Ports'* dentro de la ventana de '*System Assembly View*' o bien modificando el archivo MHS del proyecto reference nic. MHS que significa '*Microprocessor Hardware Specification*' define al componente hardware y la configuración del sistema del procesador embebido e incluye los periféricos, el procesador el espacio para las direcciones la arquitectura del bus y la conectividad del sistema [31]. Al abrirlo se verán las instancias de cada Pcore con sus respectivas conexiones y al bajar del todo estarán los módulos recién creados todo ello en lenguaje HDL. Dentro de cada uno de los módulos simplemente habrá que poner el siguiente código:

```
PORT aclk = core_clk
PORT aresetn = Peripheral_aresetn
PORT ap_start = net_vcc
```

Con esta primera línea se conecta el módulo a la señal de reloj al igual que con la segunda el puerto del reset queda conectado. La última línea de código alude al puerto de start que tienen todos los módulos creados. Si este puerto no está activo siempre a uno el módulo nunca funcionará.

Por último se tuvo que hacer una última modificación. Debido a la introducción de nuevos módulos es necesario ampliar la memoria asignada al MicroBlaze. Inicialmente la plataforma le reserva 16K pero tras bajar el diseño a la placa se vio que era insuficiente y por lo tanto se aumento a 32K. Finalmente cuando todo el diseño hardware haya sido completado se exportará el diseño a Xilinx SDK. Existe una forma directa usando el XPS el cual generará un archivo ELF que es un '*Executable and Linkable Format'* es decir un archivo ejecutable el cual contiene una representación binaria del procesador [31].

Como ya se indicó para esta solución no habría comunicación con el MicroBlaze y por ello no se tendría que modificar el programa que viene con las instrucciones software. No obstante habría que crear un nuevo proyecto en blanco debido que a pesar de que las instrucciones software no se han de modificar el hardware si se ha hecho y por ello se debe de crear un nuevo proyecto con los archivos generados con Xilinx XPS. Y finalmente ya tan solo faltará bajar el diseño a la placa.

5 Pruebas, resultados y mejoras

5.1 Pruebas

5.1.1 Envío de paquetes

Con el diseño programado en la placa se procederá a configurar las dos máquinas tanto la de la NetFPGA como la de la reference nic para comprobar su correcto funcionamiento. En primer lugar para realizar las pruebas se deberá de cargar el driver correspondiente al reference nic para que mediante el MicroBlaze se reconozca el proyecto que se ha cargado. Que a pesar de haberse introducido modificaciones en sí mismo el proyecto es el mismo. Este driver, nf10.ko, se encuentra dentro del proyecto reference nic en *sw/host/drivers* tras haberse hecho un *make* en ese mismo destino y se insertará mediante el comando *insmod*. El siguiente paso será el de configurar la tabla de rutas en ambas máquinas. Debido a la conexión de ambas placas mediante el PCIe se utilizará la misma tabla de rutas que la del sistema operativo que soporta la placa. Se procederá a configurar en un primer lugar la máquina que alberga la placa NetFPGA. Para ello se tendrán que habilitar o levantar los interfaces de la placa es decir los puertos nf [17]. Una vez habilitados se les deberá de asignar una IP a cada puerto. En la siguiente captura de pantalla se puede observar por consola los comandos que se utilizaron para dejar la placa puesta a punto.

```
[root@berilio ~]# ifconfig nf0 up
[root@berilio ~]# ifconfig nf1 up
[root@berilio ~]# ifconfig nf0 192.168.0.1
[root@berilio ~]# ifconfig nf1 192.168.1.1
[root@berilio ~]# ifc
ifcfg     ifconfig
[root@berilio ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 30:85:A9:A4:4C:7D
          inet addr:150.244.58.44  Bcast:150.244.59.255  Mask:255.255.252.0
          inet6 addr: fe80::3285:a9ff:fea4:4c7d/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:17425 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2785336 (2.6 MiB)  TX bytes:2239 (2.1 KiB)
          Interrupt:19 Memory:fb600000-fb620000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:480 (480.0 b)  TX bytes:480 (480.0 b)

nf0       Link encap:Ethernet  HWaddr 00:4E:46:31:30:00
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::24e:46ff:fe31:3000/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:468 (468.0 b)
          Interrupt:81

nf1       Link encap:Ethernet  HWaddr 00:4E:46:31:30:01
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::24e:46ff:fe31:3001/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:468 (468.0 b)
          Interrupt:81

[root@berilio ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 nf1
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0        0 nf0
150.244.56.0    0.0.0.0         255.255.252.0   U     0      0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     1002   0        0 eth0
0.0.0.0         150.244.56.1    0.0.0.0         UG    0      0        0 eth0
```

Figura 29: Configuración por comando las interfaces de salida en la NetFPGA

La máquina que alberga la tarjeta de red NIC también deberá ser configurada. Igualmente habrá que levantar los puertos Ethernet y asignarles una dirección IP. Al comprobar en la tabla de rutas habrá que tener en cuenta y que todo esté en orden. Para ello habrá que prestar especial interés en que la dirección por defecto o default este la última de todas pues si está la primera todos los paquetes utilizarán ese método de salida o entrada. Teniendo las fibras bien conectadas a sus puertos correspondientes se procederá a realizar *PINGs* de una máquina a la otra y a través de los distintos interfaces. Por consola se puede observar una pequeña descripción de lo que está ocurriendo pero si se desea una mayor descripción se puede abrir el

interfaz gráfico de Wireshark [28] el cual es un analizador de paquetes según el puerto que se elija y observar lo que está sucediendo.

```
root@angus:~# ifconfig eth1 up
root@angus:~# ifconfig eth2 up
root@angus:~# ifconfig eth1 192.168.0.3
root@angus:~# ifconfig eth2 192.168.1.3
root@angus:~# route del default
root@angus:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
150.244.56.0    0.0.0.0         255.255.252.0   U     1      0        0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 eth0
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth2
root@angus:~# ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_req=1 ttl=64 time=0.241 ms
64 bytes from 192.168.0.1: icmp_req=2 ttl=64 time=0.266 ms
64 bytes from 192.168.0.1: icmp_req=3 ttl=64 time=0.267 ms
64 bytes from 192.168.0.1: icmp_req=4 ttl=64 time=0.207 ms
64 bytes from 192.168.0.1: icmp_req=5 ttl=64 time=0.144 ms
64 bytes from 192.168.0.1: icmp_req=6 ttl=64 time=0.271 ms
64 bytes from 192.168.0.1: icmp_req=7 ttl=64 time=0.161 ms
64 bytes from 192.168.0.1: icmp_req=8 ttl=64 time=0.263 ms
64 bytes from 192.168.0.1: icmp_req=9 ttl=64 time=0.148 ms
64 bytes from 192.168.0.1: icmp_req=10 ttl=64 time=0.247 ms
^C
--- 192.168.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9000ms
rtt min/avg/max/mdev = 0.144/0.221/0.271/0.051 ms
```

Figura 30: Configuración por comando de los puertos Ethernet de la reference nic

5.1.2 Comparación con Simulación

En una sección anterior se explicó detalladamente la forma de ejecutar una simulación utilizando los archivos python que suministra la plataforma NetFPGA. A continuación utilizando estas simulaciones se va a demostrar que efectivamente ambos módulos ya sea el inicial input_arbiter como la conexión de los dos módulos el read_packet y el round_robin hacen lo mismo, tras haber ya demostrado a nivel de consola que los paquetes efectivamente se transmiten y se reciben sin aparente diferencia.

Puesto que es imposible plasmar sobre papel el total de la simulación se va a prestar la atención tan solo en la zona deseada. En las siguientes figuras aparecerán las simulaciónes del input arbiter y del módulo Round-Robin. Para conseguir esta primera se tuvo que partir del trabajo original y simularlo sin introducir ninguna modificación, y para la siguiente se simuló el proyecto entero y se obvió el otro módulo creado, el read packet.

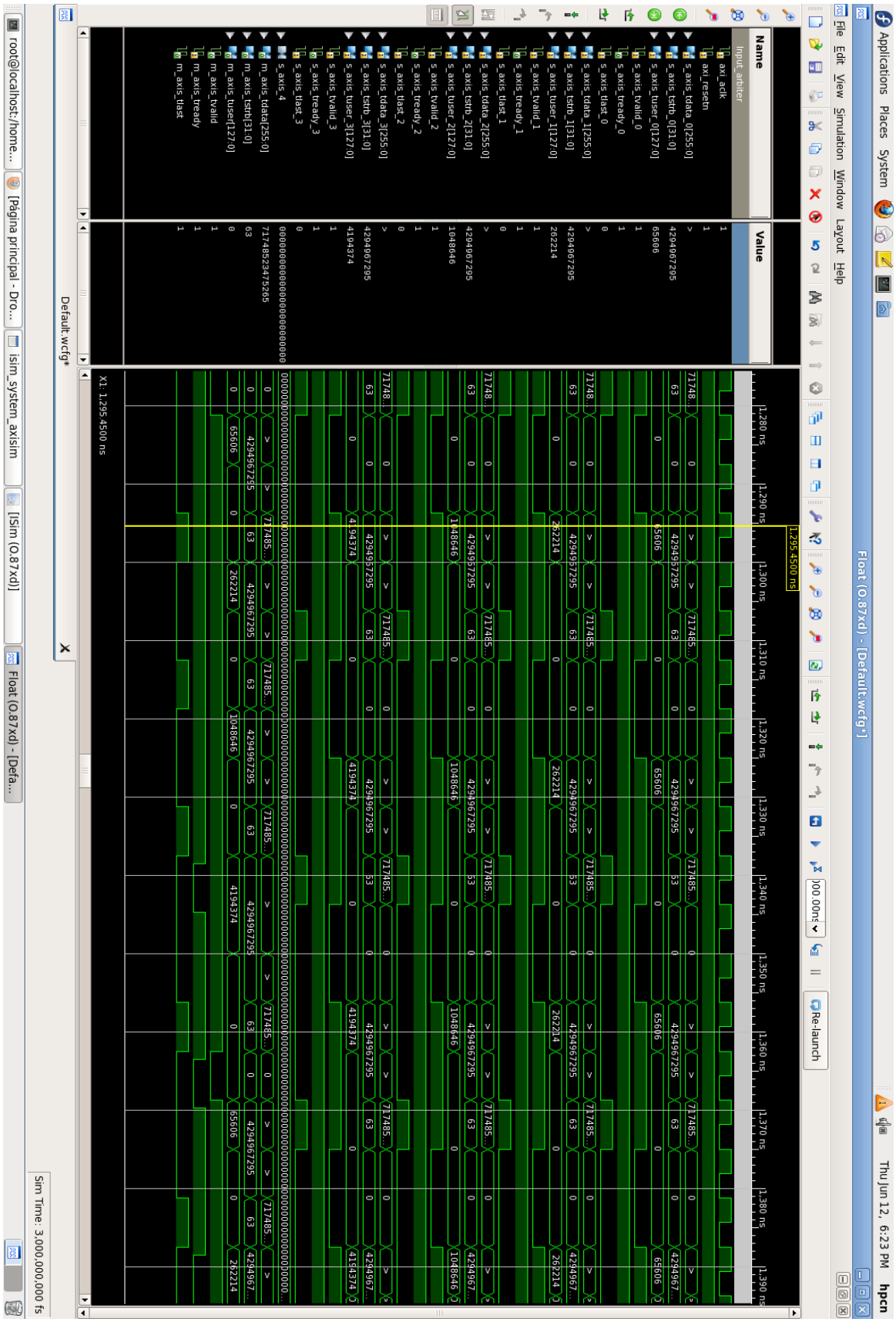


Figura 31: Simulación del input arbiter

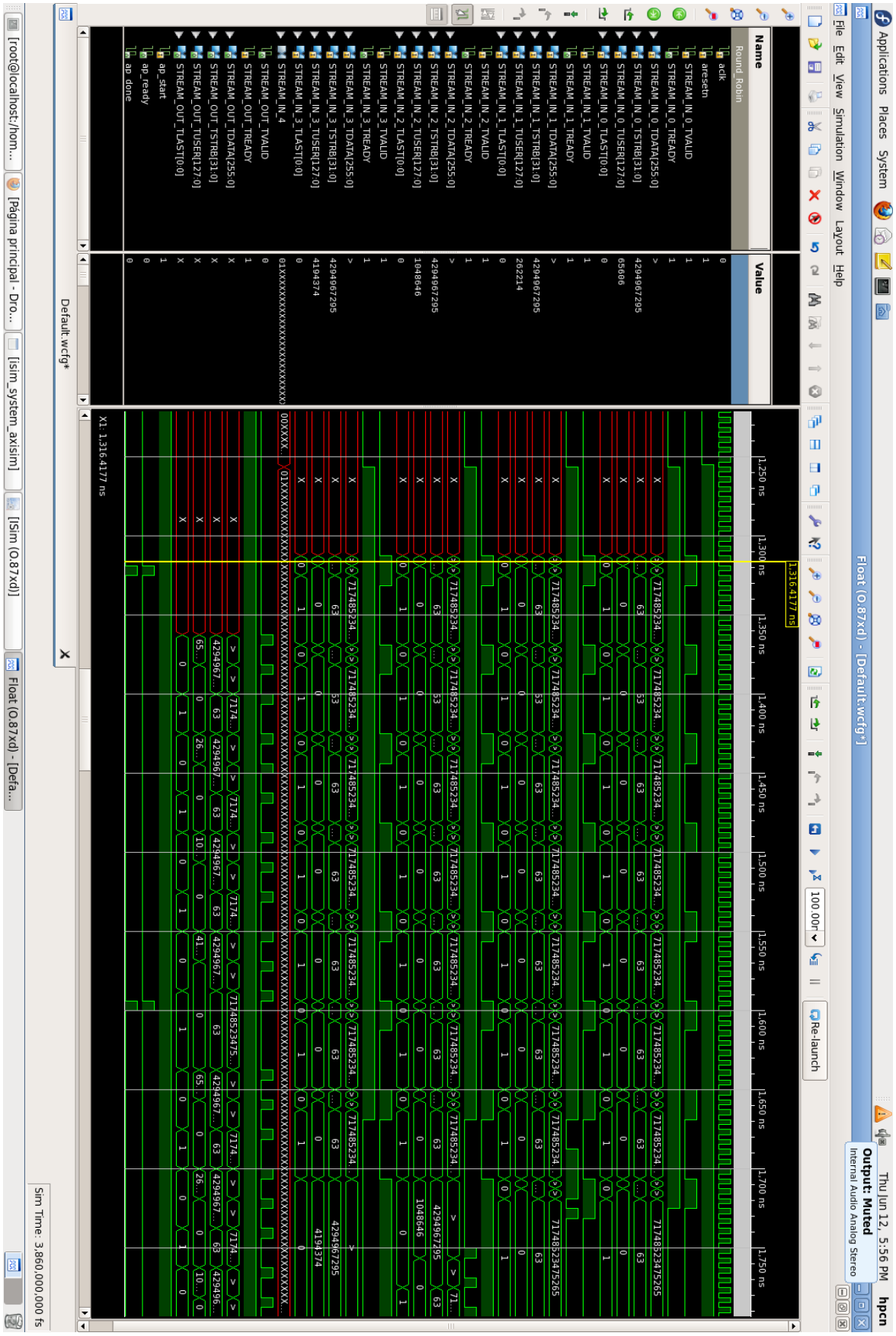


Figura 32: Simulación Round Robin

Se puede observar que se han juntado en un único bus todas aquellas señales pertenecientes al DMA ya que durante esta simulación van a ser nulas y así se podrá observar con mayor facilidad el resto de valores, en la primera figura ese bus es el "s_axis_4" y en la segunda es el "STREAM_IN_4". Para verificar que ambas simulaciones realizan lo mismo en un primer lugar se va a proceder a describir la primera figura. Lo que uno se ha de fijar para la comparación es en el valor que tiene el parámetro user dentro de cada axi. Este será el indicativo del paquete se está procesando pues como se puede observar los datos son idénticos. El *user* para el axi_0 es 65606, para el axi_1 es el 262214, para el axi_2 es el 1048646 y para el axi_3 es el 4194374. Ahora bien si prestamos atención al parámetro *user* del axi de salida se puede ver como va cogiendo un paquete de cada usuario, *user*, siguiendo la filosofía del decisor Round-Robin. Ahora bien si se procede con la siguiente imagen se ve que el valor *user* es igual para cada axi a pesar de que ahora en vez de llamarse axi se denomine *Stream*. Pues al centrarse en el bus de salida se ve como efectivamente va cogiendo un paquete de cada usuario teniendo una salida idéntica a la del ejemplo anterior. Primero coge un paquete con el usuario 65606, después otro del 262214, luego del 1048646, finalmente del 4194374 y seguirá así hasta llegar a coger el último paquete del 4194374.

5.2 Resultados

A la hora de analizar los resultados se tomarán dos perspectivas distintas. Por un lado se prestará atención a la latencia introducida utilizando la solución presentada frente a la plataforma inicial. Por otro lado se explicará detalladamente la gran ventaja que supone utilizar este tipo de solución que implica un gran ahorro en líneas de código y por lo tanto en tiempo. Teniendo en cuenta que una programación a un alto nivel es simple mucho más simple y comprensible que una programación a bajo nivel.

5.2.1 Latencia

El primer resultado el cual a uno se le puede venir a la cabeza es el que al modificar un módulo ya optimizado e incluso introducir módulos nuevos esto creará una latencia o retraso dentro del proyecto. Lo que queda por determinar es si este retraso es significativo o no para la viabilidad de la solución.

Analizando los tiempos del *input arbiter* se puede observar que tarda en procesar el primer paquete y enviarlo por la cola de salida un total de 20ns. Además de este tiempo para esta simulación y el número de paquetes enviado determinado por el archivo *python* de simulación el módulo del *input arbiter* tardó en ejecutar todos los paquetes y procesarlos un tiempo de unos 800ns. Este último valor obviamente podrá ser mayor o menor dependiendo del número de paquetes transmitidos. Pero para esta prueba debido a que ambas simulaciones se ejecutan con el mismo número de paquetes sirve para comparar ambos resultados. En las siguientes dos imágenes se puede cotejar lo anteriormente explicado.

En la primera figura abajo en amarillo se aprecia la diferencia de los dos cursores y es como se indicó de 800ns y en la segunda figura los cursores están al inicio de la simulación para mostrar el tiempo de procesamiento. Por lo tanto se puede concluir que el módulo *input arbiter* introduce una latencia total de unos 820ns para esta simulación.

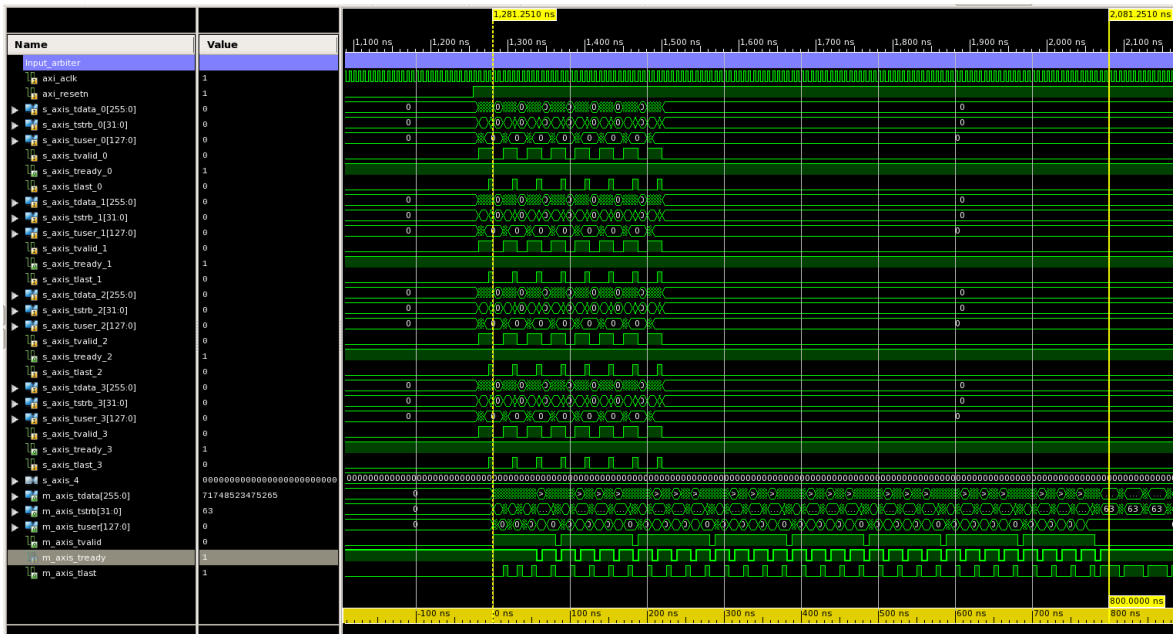


Figura 33: Tiempo total del input arbiter

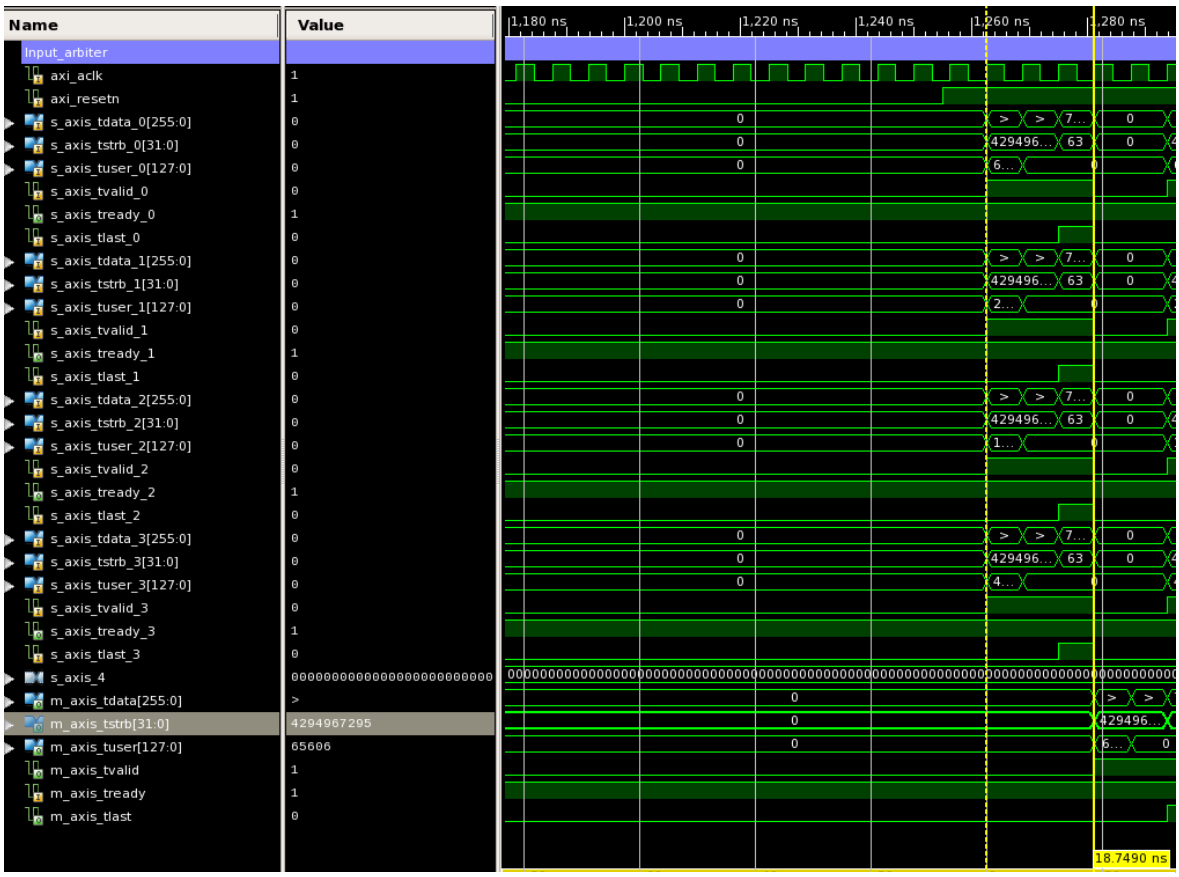


Figura 34: Tiempo inicial de procesado dentro del input arbiter

Si ahora en cambio se presta atención a la solución propuesta (en el Anexo B) se han de sumar las latencias introducidas por el Pcore read packet y por el Pcore Round-Robin para hallar el retraso total. El módulo read packet introduce un retraso de unos 50ns para el procesado del primer paquete entero. Como ya se dijo anteriormente este módulo trabaja con paquetes enteros y tan solo sacará transacciones por la salida cuando tenga el paquete entero albergado. Por lo tanto en total para esta simulación el módulo read packet tarda aproximadamente unos 410ns.

Por otro lado el siguiente módulo el Round-Robin tarda en procesar todos los paquetes un total de 2150ns y un tiempo de 50ns para una vez recibido el primer paquete. Esto hace un total de unos 2200ns que sumados a los 460ns del read packet hacen un total de unos 4660ns. En cambio la salida del Round-Robin está más optimizada pues hace eliminación de ceros los cuales transmite la salida del *input arbiter*. Esto es una ventaja pues es una transacción menos al hacer una selección previa en el principio del módulo Round-Robin, preguntando a la cola FIFO si es o no vacía.

5.2.2 Líneas de código

En esta sección se va a analizar y presentar pruebas sobre la gran ventaja que introduce el realizar módulos hardware a través de lenguajes de alto nivel. Para ello se va a utilizar el *Universal Code Lines Counter* [26]. Este programa se utiliza para contar las líneas de código que se han utilizado para crear un programa. No solo tiene en cuenta las propias líneas de código sino también las líneas en blanco y las líneas de comentarios. Permite analizar códigos de distintas fuentes por lo tanto se puede utilizar esta herramienta para lenguaje de alto nivel y para lenguaje de bajo nivel tras hacer una pequeña modificación de la extensión del archivo.

Por eso se comenzará analizando el total de líneas de código del lenguaje de alto nivel de la solución. Se presentará por separado ambos proyecto de Vivado HLS y luego se sacarán conclusión conjuntas.

Tabla 4: Líneas de código de la solución en Vivado HLS

	File	Total Lines	Source Code lines	Blank lines	Comment lines
Read Packet	read_packet.cpp	47	22	14	11
	read_packet.h	30	18	8	4
Round-Robin	round_robin.cpp	69	42	17	10
	round_robin.h	37	22	10	5
Total	-	183	104	49	30

En total se escribieron un total de 104 líneas de código de las cuales 40 pertenecen al archivo *header* o cabecera en donde se encuentran las definiciones de las estructuras y las funciones utilizadas. Por lo tanto se puede incluso decir que líneas propias de código serían solamente las correspondientes a los .cpp que son un total de 64 líneas. Ahora se mostrará una tabla con la misma estructura pero para los archivos Verilog que componen el Pcores del *Input Arbiter.*

Tabla 5: Líneas de código del Input Arbiter en Verilog

	File	Total Lines	Source Code lines	Blank lines	Comment lines
Input Arbiter	fallthrough_small_fifo_v2.v	101	68	14	15
	input_arbiter.v	281	177	38	60
	nf10_input_arbiter.v	308	219	35	54
	small_fifo_v3.v	100	58	15	23
Total	-	790	522	102	152

Los datos salen a relucir al instante y se puede concluir que gracias a sintetizar un lenguaje de alto nivel a bajo nivel se puede ahorrar muchas líneas de código. Los datos a tener en cuenta y comparar son las 104 líneas de código de la solución en alto nivel frente a las 522 es decir tan solo un quinto con todo el gran ahorro en tiempo que ello conlleva. Además anidando con la sección anterior y teniendo en cuenta que la latencia introducida es mínima se puede concluir que es un gran avance el realizar este método.

En cambio una apreciación importante es si se cotejan los resultados originales con los resultados creados tras la sintetizar en lenguaje de alto nivel. A continuación se pueden observar dos tablas, una para el conteo de líneas de lenguaje Verilog para el Pcore Round-Robin y otra para el Pcore de Read Packet.

Tabla 6: Lineas de Código en el módulo de la solución: Round-Robin

File Name	Total Lines	Source Code lines	Blank lines	Comment lines
\round_robin_ap_rst_if_c.c	21	10	5	6
\round_robin_c.c	868	774	52	42
\round_robin_read_data_V_c.c	78	53	19	6
\round_robin_read_last_V_c.c	337	259	35	43
\round_robin_read_strb_V_c.c	78	53	19	6
\round_robin_read_user_V_c.c	78	53	19	6
\round_robin_STREAM_IN_0_if_c.c	337	259	35	43
\round_robin_STREAM_IN_1_if_c.c	337	259	35	43
\round_robin_STREAM_IN_2_if_c.c	337	259	35	43
\round_robin_STREAM_IN_3_if_c.c	337	259	35	43
\round_robin_STREAM_IN_4_if_c.c	337	259	35	43
\round_robin_STREAM_OUT_if_c.c	337	259	35	43
\round_robin_top_c.c	464	410	48	6
TOTAL COUNT FOR ALL 13 FILES	3946	3166	407	373

Tabla 7: Lineas de Código en el módulo de la solución: Read_packet

File Name	Total Lines	Source Code lines	Blank lines	Comment lines
\read_packet_ap_rst_if_c.c	21	10	5	6
\read_packet_c.c	573	488	47	38
\read_packet_data_read_data_V_c.c	78	53	19	6
\read_packet_data_read_last_V_c.c	78	53	19	6
\read_packet_data_read_strb_V_c.c	78	53	19	6
\read_packet_data_read_user_V_c.c	78	53	19	6
\read_packet_STREAM_A_if_c.c	337	259	35	43
\read_packet_STREAM_B_if_c.c	337	259	35	43
\read_packet_top_c.c	188	158	24	6
TOTAL COUNT FOR ALL 9 FILES	1.768	1.386	222	160

Se puede observar como se ha aumentado drásticamente el número de líneas de código con respecto a los archivos generales que proporciona la plataforma. Como cabe esperar el módulo round robin al ser más complejo tiene un mayor número de líneas.

Por otro lado estos últimos datos se han de comparar con las líneas de código originales escritas en C. Se ve como de un total de 40 líneas de código C se pasan a unas 1.386 para el caso del módulo read packet. Y por otro lado de unas 64 a 3.166 para el módulo round robin. Este dato si es significativo pues si se tuviesen que escribir todas estas líneas en Verilog sería inviable pero no hay que olvidar que el trabajo realizado a mano fueron solo las líneas escritas en C y luego fue el programa quien creó el resto en lenguaje de bajo nivel.

5.3 Mejoras

Para explicar las mejoras posibles partiendo desde la solución propuesta se expondrán dos casos de estudio como ejemplos para poder ver la placa NetFPGA en práctica con aplicaciones reales y así aplicar mejoras para cada caso.

5.3.1 Prioridad de colas

Sea hasta un máximo de cuatro entidades financieras cuyas centralitas de salida y de entrada se transmiten a través de una NetFPGA como *router* de salida. A cada entidad financiera se le tarifará de forma diferente creando así tratos especiales con cada cliente. Obviamente el cliente que tenga una cuota más alta sus paquetes tendrán mayor prioridad que aquellas que deseen adherirse a una tarifa menor. Para ello se denominarán Entidad A, Entidad B, Entidad C y Entidad D siendo A el cliente más especial y D el menor. A cada entidad se le asignará un interfaz Ethernet 10G conectado mediante cables de fibra provenientes de cada centralita. Gracias a ello se pueden asignar distintas prioridades pues se sabe de ante mano a que puerto irá conectado cada entidad.

Por lo tanto para este caso práctico se tendrá que hacer una modificación en el código de Vivado HLS creando así un trato más especial al cliente A. La mejora que se propone para este caso de estudio es a la llegada de cada nuevo paquete en vez de seguir método Round-Robin, modificar el organizador para cada vez preguntar primero al A, luego al B, luego al C y por último al D. Por lo tanto cada vez que llegue un paquete al puerto A no tendrá que esperar y tan solo se tendrá en cuenta al resto de entidades en caso de que el A no quiera transmitir. Esto supone una gran desventaja para la Entidad D pero al ser la entidad que menor aportación económica emite será el último a tener en cuenta.

A nivel de código para plasmar la idea anterior simplemente se partirá de la solución anterior del Round-Robin y se eliminará el bucle y el *switch* inicial creando mediante una serie de *if* y *else* la correspondiente prioridad:

```
queue=0;
if(!data_input_nf0.empty()) queue=1;
else if(!data_input_nf1.empty()) queue=2;
else if(!data_input_nf2.empty()) queue=3;
```

```
else if(!data_input_nf3.empty()) queue=4;
else if(!data_input_nf4.empty()) queue=5;
```

Según esta disposición se puede ver que la entrada nf0 es la asignada a la Entidad A, la nf1 al Entidad B, la nf2 a la Entidad a C y la nf3 a la Entidad D recordando que la nf4 es para el DMA pero al no usarse para este ejemplo no importa que sea la última.

5.3.2 Prioridad según destino

Para esta segunda mejora se ha de aportar un poco de información extra para poder comprender lo que se está haciendo. A diferencia del caso anterior donde la prioridad iba predeterminada según la entrada hardware a la cual estuviese conectado cada cliente ahora se pasará a crear un tipo de prioridad según el destino final del paquete el cual se está transmitiendo. Para ello se guardarán ciertas transiciones para poder obtener el destino y según aplicarle una prioridad o no. Para este caso de estudio la IP de destino con prioridad máxima serán aquellos paquetes que vayan hacia la Escuela Politécnica de la Universidad Autónoma de Madrid con IP 150.244.56.x tipo B con máscara 22. Esto significa que cualquier paquete con IP de destino perteneciente al rango entre 150.244.56.0 hasta 150.244.59.255 irá hacia la Escuela Politécnica de Universidad Autónoma de Madrid.

Figura 35: Cabecera IPv4 [6]

La NetFPGA utiliza paquetes que siguen la versión cuatro del protocolo IP. Por lo tanto todos los paquetes que entren dentro de la placa tendrán una cabecera IPv4. Esta cabecera tiene la siguiente distribución:

Como se sabe un paquete NetFPGA que sigue el protocolo AXI tiene transacciones de 256 bits. La cabecera vendrá dentro de la primera transacción pues la cabecera son

como mínimo 160 bits y de ahí se albergará esta transacción para después hacer comprobaciones. La versión de esta cabecera será igual a 4, el IHL o Internet *Header Length* es un número de 4 bits que determina la longitud de la cabecera. Esto es necesario pues como se ha indicado como mínimo medirá 160 bits pero tras el campo de la dirección de destino se le pueden añadir una serie de opciones a la cabecera aumentando así su tamaño. Este campo se diferencia del de *Total Length* pues este para calcular el tamaño total incluye el campo data que iría después de la dirección de destino o de las opciones si las hubiese. Los siguientes 4 bytes son los relacionados a la fragmentación siendo el identificador un número único que lo diferencia de los restos de fragmentos, las banderas para aplicar nuevos fragmentos o dejar sin fragmentar el paquete y por último el offset que sirve para saber colocar en orden cada fragmento a la hora de reconstruir el paquete final. Por último vienen los campos que son de interés para esta mejora [13] la dirección de origen y la dirección de destino. Pues esta mejora se va a introducir al inicio del diseño tipo pipeline del *router* para aplicar prioridades cobra mayor interés el campo de dirección de destino. Como ya se comento si la dirección de destino es igual a alguna de las comprendidas dentro de la zona reservada para la EPS de la Universidad Autónoma de Madrid este paquete cobrará mayor prioridad.

Una vez abstraída esta idea se procedió a plasmarla en código. Lo que el programa hará será recorrer inicialmente en una primera ronda todas las colas tipo FIFO correspondientes a los cuatro interfaces Ethernet de 10Gbps y el del DMA. Tan solo accederá a la cola si existe algún paquete sea cual sea el destino. En caso de haber un paquete en alguna de las colas se accederá a ella y se albergará la primera transacción del paquete. Sobre esta transacción se extraerán los bits correspondientes al campo de la dirección de destino y estos se cotejarán con el valor mínimo y máximo de las posibles direcciones que se pueden tener dentro de la Escuela Politécnica de la Universidad Autónoma de Madrid. Puesto que el rango va entre 150.244.56.0 hasta 150.244.59.255 tan solo se comprobará que si los 3 primeros bytes de la dirección de destino están entre 150.244.56.X y 150.244.59.X. De esta manera el comparador a nivel de hardware será menor y el retraso que pueda llegar a introducir es menor también. Estas direcciones se tradujeron a hexadecimal para evitar problemas con el signo si se hacen comparaciones a nivel de bit o valor decimal. En caso de que haya un acierto se procederá a enviar este paquete entero. Para enviar el paquete entero primero habrá que enviar a la salida la primera transacción leído y después el resto de paquete que seguirá en la cola. Esto se debe a que cada vez que se hace una lectura de una cola FIFO de un *Stream* se saca la transacción de la cola y el siguiente valor

pasará a ser el primero. Para poder cumplir el objetivo de mantener la prioridad de los paquetes con dirección a la EPS se seguirá preguntando al siguiente paquete en esa misma cola, es decir, a diferencia del método anterior, el Round-Robin, no se pasará a la siguiente cola. A este nuevo paquete se le procederá de manera idéntica en caso de ser un paquete con dirección de la EPS, en caso contrario se guardará la primera transacción pues ya ha sido leída y se pasará a la siguiente cola. Lo que se consigue procediendo de esta manera es de al final de la primera vuelta el haber sacado todos los paquetes que hayan de la EPS y se sabe que existen paquetes esperando cuyos destinos son distintos a los elegidos, pues solamente se pasa a la siguiente cola en caso de que venga un paquete que no sea de los elegidos. Esto obviamente produce un problema en caso de que en la primera cola consultada tan solo vengan paquetes hacia la EPS de la UAM pues los otros paquetes esperando seguramente mueran y hayan de ser desechados.

Una vez finalizada la primera vuelta y cumpliendo unas condiciones que se pueden observar en el código (Anexo C) se procederá con la siguiente vuelta. En este caso el procedimiento a seguir es el mismo que en casos anteriores el ir cola a cola e ir sacando paquete a paquete de una sola vez siguiendo el método Round-Robin. Es importante recalcar que todo esto tan solo ocurre si existe un paquete esperando dentro de cualquiera de las colas. Esta lógica se siguió para evitar el caso de estar escuchando constantemente de un único camino esperando a que vengan paquetes sin saber lo que pueda estar pasando en los otros. Este método de dos vueltas se seguirá siempre y cuando hayan paquetes sabiendo que en la primera vuelta se guardarán las primeras transacciones y se comprobará si son las elegidas y en tal caso se procederá a actuar y en caso contrario se esperará a que finalice el primer barrido y se actuará de manera normal.

6 Conclusiones y trabajo futuro

6.1 *Conclusiones*

En esta sección se mostrará un resumen de los resultados obtenidos y se valorarán las implicaciones de las modificaciones que se han realizado y como ello ha afectado al rendimiento del trabajo ya existente. Por lo tanto se expondrán los casos desde dos puntos de vista distintos primero uno crítico y negativo y luego los aspectos positivos que son mayores para así llegar a una solución final.

Como aspecto negativo se ha de hacer notar que se introduce una gran latencia a los paquetes generando módulos mediante lenguaje de alto nivel. Si bien a este nivel no es relevante pues aun así al ser importantes los retrasos en general siguen siendo insignificantes en próximas evoluciones si puede llegar a ser algo notable. Hay que tener en cuenta que el gran objetivo de esta placa es el procesamiento de paquetes a alta velocidad, a velocidades de 10Ghz, y por ello si se le introduce algún retraso por procesamiento se puede llegar a congestión e incluso a pérdida de paquetes por inactividad. Un ejemplo se puede ver mediante la diferencia del módulo más simple que se ha creado en la primera aproximación utilizando los recursos con el pass through. En este módulo existe un retraso de unos 25ns de procesamiento pero en cambio si se complica un poco más la logística como en el Round-Robin el retardo inicial por procesamiento es del doble unos 50ns. Este retardo luego se va ampliando pues el tiempo del procesado de un paquete y por lo tanto al enviar X paquetes el retardo aumentará por un factor de X. No obstante se ha de tener en cuenta que se están barajando cifras en magnitudes de nano segundos es decir algo totalmente insignificante pero que si no se tiene en cuenta puede causar algún problema.

En cuanto a las ventajas se han de abordar desde dos perspectivas distintas por un lado las empíricas y por otro las posibilidades que esto aporta para el futuro. Las empíricas como ya se han visto en la sección anterior son las relacionados con el ahorro de líneas de código. Esto conlleva directamente a un gran ahorro en tiempo lo cual es un ahorro en dinero para la realización de cualquier proyecto. Debido a usar un lenguaje de alto nivel y ser mucho más intuitivo para el ser humano la programación es más sencilla y mediante el uso de funciones que pueden ya existir o se crearse se puede simplificar cualquier programa.

Este último aspecto destacado el relacionado con la simpleza del lenguaje de programación lleva a la siguiente ventaja. Las posibilidades que entabla esta nueva tecnología. Como se sabe la productividad de lenguajes de bajo nivel son bastante pobres y son hasta ahora los utilizados para programar placas FPGAs, debido a no solo tener que introducir toda la lógica que conlleve el programa sino también las conexiones entre los distintos interfaces. De esta manera utilizando un lenguaje de alto nivel se ha visto que mediante unas simples líneas de código se puede implementar el mismo sistema y con un trabajo mucho menor una vez se adquiere los conocimientos necesarios. Además se ha demostrado en la sección de mejoras como haciendo unas pequeñas modificaciones se pueden crear algoritmos más complejos y a su vez aplicaciones para la vida real que permitan procesar paquetes a niveles de gigabytes.

En conclusión durante este proyecto se ha partido desde cero y se han asimilado los conceptos necesarios para comprender, utilizar y desarrollar con una tecnología puntera utilizando herramientas complejas. Esto conllevo un gran sacrificio inicial a la hora de la puesta en marcha e incluso crear un módulo de lo más simple, pero una vez se obtuvieron estos conocimientos, el crear mejoras fue muy simple. Con esto se quiere demostrar que si se llega al nivel de conocer la plataforma las posibilidades que contrae esta tecnología son infinitas.

6.2 *Trabajo futuro*

Una vez validado la posibilidad del uso de las herramientas de síntesis de algo nivel (HLS- High Level Synthesis), los pasos siguientes van en la dirección de validar con problemas de mayor envergadura en el ámbito de las redes, así como la adecuación de la metodología para diseñadores sin conocimientos del flujo de diseño hardware.
En este sentido, algunos de los problemas a tratar en el futuro incluyen:

- Implementar el proyecto reference_nic (*Network Interface Card*) usando lenguajes de alto nivel. Validar tiempo de desarrollo y performance de la solución respecto del proyecto original propuesto por la comunidad NetFPGA.
- Validar en uso de esta técnica en otras plataformas FPGA diferentes de NetFPGA. Comprobar la portabilidad en los diseños HLL (*High Level Languages*) entre plataformas.
- Comprobar la viabilidad del uso de Vivado-HLS para el procesador de paquetes utilizando técnicas de inspección profunda de paquetes (DPI-*deep-packet inspections*).
- Desarrollo de un *framework* de desarrollo de aplicaciones de red basado en Vivado-HLS que permita a desarrolladores de Software implementar funciones en Hardware ocultando los detalles de implementación que hoy son necesarios tener en cuenta.

Referencias

[1] *AMBA AXI Protocol, Specifications*. http://esca.korea.ac.kr/teaching/comp427_ES/AMBA/AMBA_v30_AXI_v10.pdf.

[2] *10G MAC Interface with registers*. https://github.com/NetFPGA/NetFPGA-public/wiki/10G-MAC-Interface-with-registers, Mayo 2014.

[3] *ARM*. http://www.arm.com/, Mayo 2014.

[4] *AXI Reference Guide*. http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v14_1/ug761_axi_reference_guide.pdf, Abril 2014.

[5] *Bram Output Queues*. https://github.com/NetFPGA/NetFPGA-public/wiki/BRAM-Output-Queues-with-registers, Mayo 2014.

[6] *Cabecera de un paquete IPv4*. http://mattiasgeniar.be/2010/04/09/how-ipv6-headers-are-formed-compared-to-ipv4/, Julio 2014.

[7] *DMA Engine*. https://github.com/NetFPGA/NetFPGA-public/wiki/Dma%20Engine, Mayo 2014.

[8] *Eclipse*. https://www.eclipse.org/, Mayo 2014.

[9] *Fedora*. https://fedoraproject.org/, Mayo 2014.

[10] *High Level Synthesis*. http://en.wikipedia.org/wiki/High-level_synthesis, Mayo 2014.

[11] *Hitechglobal Board specifications*. http://www.hitechglobal.com/Boards/PCIExpress_SFP+.htm, Julio 2014.

[12] *Input Arbiter*. https://github.com/NetFPGA/NetFPGA-public/wiki/Input-arbiter-with-registers, Mayo 2014.

[13] *IP v4 Specifications*. http://en.wikipedia.org/wiki/IPv4, Julio 2014.

[14] *NetFPGA*. http://netfpga.org/index.html, Julio 2014.

[15] *NetFPGA 10G Reference NIC*. https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-Reference-NIC, Mayo 2014.

[16] *NetFPGA 1G Specifications*. http://netfpga.org/1G_specs.html, Julio 2014.

[17] *NIC Driver Quick Start*. https://github.com/NetFPGA/NetFPGA-public/wiki/NIC%20Driver%20Quick%20Start, Mayo 2014.

[18] *Organizador Round-Robin*. http://en.wikipedia.org/wiki/Round-robin_scheduling, Mayo 2014.

[19] *OSI Model*. http://en.wikipedia.org/wiki/OSI_model, Julio 2014.

[20] *Output port Lookup*. https://github.com/NetFPGA/NetFPGA-public/wiki/NIC%20Output%20Port%20Lookup, Mayo 2014.

[21] *PCIe Specifications*. http://en.wikipedia.org/wiki/PCI_Express, Mayo 2014.

[22] *Pragma Definition*. http://gcc.gnu.org/onlinedocs/cpp/*Pragmas*.html, Mayo 2014.

[23] *Reference Opertaing System*. https://github.com/NetFPGA/NetFPGA-public/wiki/Reference%20Operating%20System, Junio 2014.

[24] *Standards IP Interfaces*. https://github.com/NetFPGA/NetFPGA-public/wiki/Standard-IP-Interfaces, Mayo 2014.

[25] *SUN STORAGE 10GbE FCoE PCIe CONVERGED NETWORK ADAPTER*. http://www.oracle.com/us/products/servers-storage/storage/storage-networking/10gbe-fcoe-pcie-cna-ds-081129.pdf, Mayo 2014.

[26] *Universal Code Lines Counter*. http://download.cnet.com/Universal-Code-Lines-Counter/3000-2352_4-10969491.html, Junio 2014.

[27] *Virtex-5 Family Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, Julio 2014.

[28] *Wireshark*. http://www.wireshark.org/, Mayo 2014.

[29] *Xilinx Design Tools*. http://www.xilinx.com/products/design-tools/ise-design-suite/, Julio 2014.

[30] *Xilinx History*. http://en.wikipedia.org/wiki/Xilinx, Julio 2014.

[31] *Xilinx Platform Studio project Files*. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/platform_studio/ps_r_gst_project_files.htm, Mayo 2014.

[32] *Xilinx SDK*. http://www.xilinx.com/tools/sdk.htm, Mayo 2014.

[33] *Xilinx Vivado HLS Design Tool*. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/, Mayo 2014.

[34] *Xilinx XPS*. http://www.xilinx.com/tools/xps.htm, Junio 2014.

[35] *Xilinx's JTAG*. http://www.xilinx.com/products/boards-and-kits/HW-USB-II-G.htm, Mayo 2014.

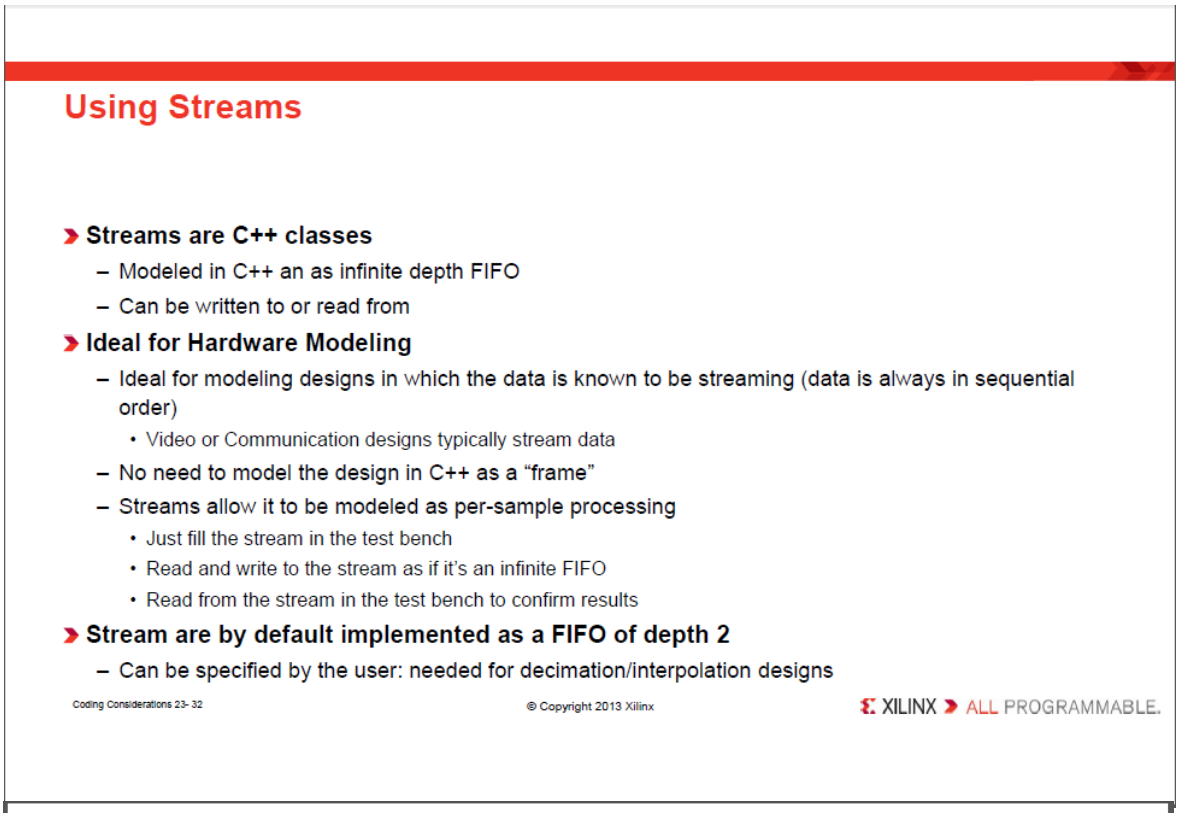
[36] Nick McKeown Greg Watson and Martin Casado. Netfpga: A tool for network research and education. *Department of Electrical Engineering Stanford University*, 2006.

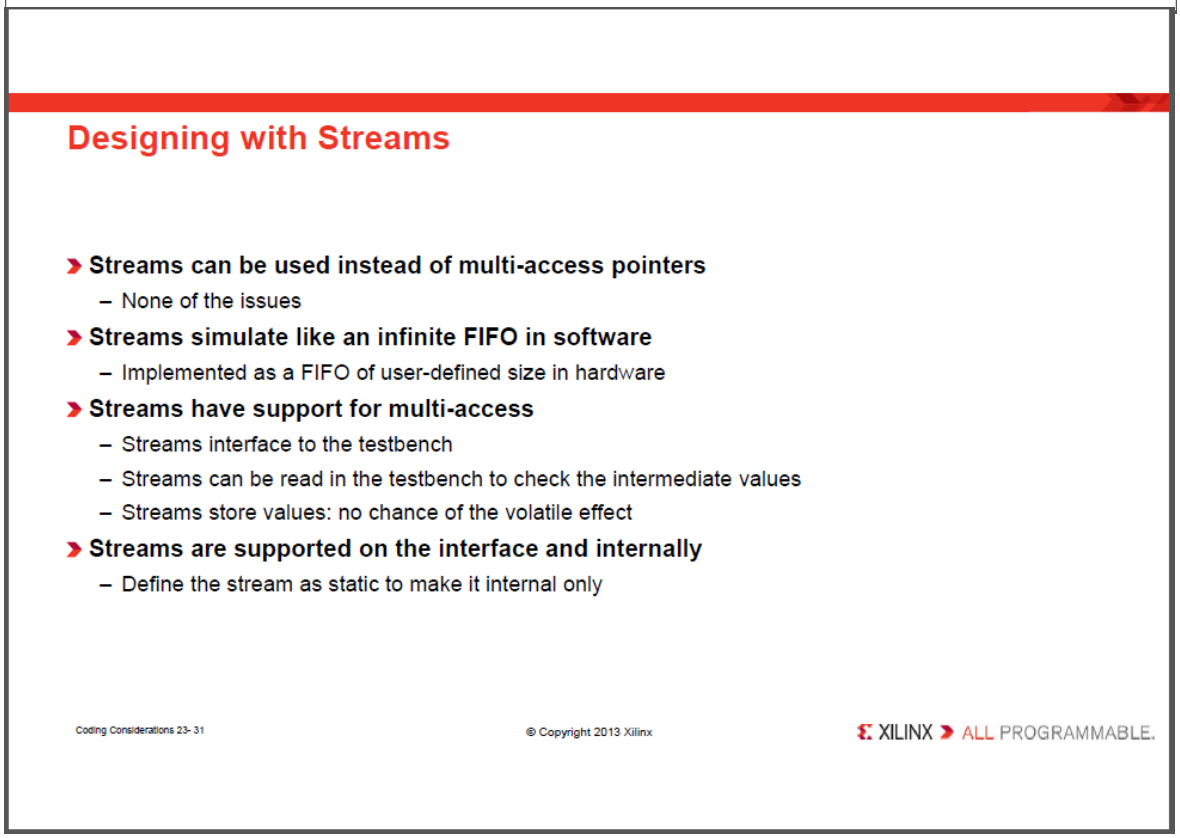
Acrónimos

FPGA:	Field Programmable Gate Array
HLS:	High Level Synthesis
HDL:	High Description Languange
HLL:	High Level Language
Gbps:	Gigabits per second
ASICs:	Application-Specific Integrated Circuit
HW:	Hardware
SW:	Software
ISE:	Integrated Software Environment
XPS:	Xilinx Platform Studio
EDK:	Embedded Development Kit
SDK:	Software Development Kit
IP:	Intelectual Property (as in IP core)
IP:	Internet Protocol
UAM:	Universidad Autónoma de Madrid
EPS:	Escuela Politécnica Superior
PCIe:	Peripheral Component Interconnect Express
DMA:	Direct Memory Access
RTL:	Register Transfer Level
MHS:	Microprocessor Hardware Specification
ELF:	Executable and Linkable Format

Anexos

A Stream Vivado HLS





Stream Example

- Create using hls::stream or define the hls namespace

```
#include <ap_int.h>
#include <hls_stream.h>

typedef ap_uint<128> uint128_t;                 // 128-bit user defined type

hls::stream<uint128_t> my_wide_stream;    // A stream declaration
```

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;                        // Use hls namespace

typedef ap_uint<128> uint128_t;          // 128-bit user defined type

stream<uint128_t> my_wide_stream;    // hls:: no longer required
```

- Blocking and Non-Block accesses supported

```
// Blocking Write
hls::stream<int> my_stream;

int src_var = 42;
my_stream.write(src_var);
// OR use:  my_stream << src_var;
```

```
// Blocking Read
hls::stream<int> my_stream;

int dst_var;
my_stream.read(dst_var);
// OR use: dst_var = my_stream.read();
```

```
hls::stream<int> my_stream;

int src_var = 42;
bool stream_full;

// Non-Blocking Write
if (my_stream.write_nb(src_var)) {
    // Perform standard operations
} else {
    // Write did not happen
}

// Full test
stream_full = my_stream.full();
```

```
hls::stream<int> my_stream;

int dst_var;
bool stream_empty;

// Non-Blocking Read
if (my_stream.read_nb(dst_var)) {
    // Perform standard operations
} else {
    // Read did not happen
}

// Empty test
fifo_empty = my_stream.empty();
```

Stream arrays and structs are not supported for RTL simulation at this time: must be verified manually.

e.g. hls::stream<uint8_t> chan[4])

B Simulaciones

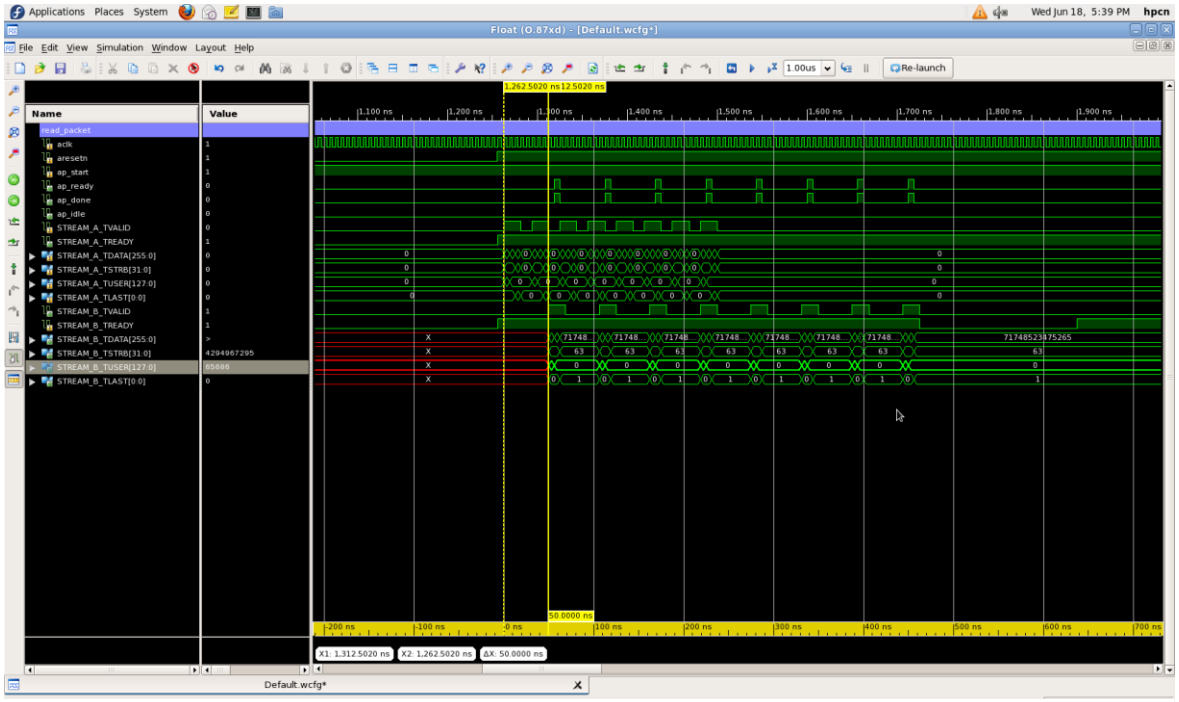


Figura 36: Read Packet Procesado inicial

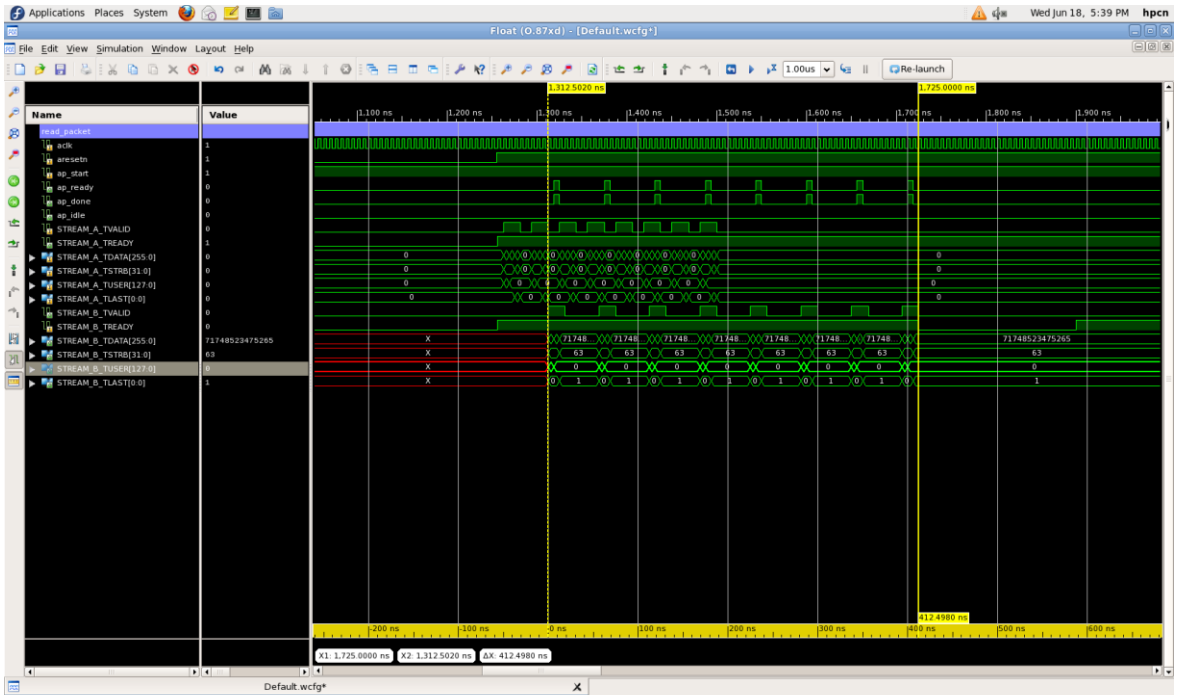


Figura 36: Read Packet Tiempo Total

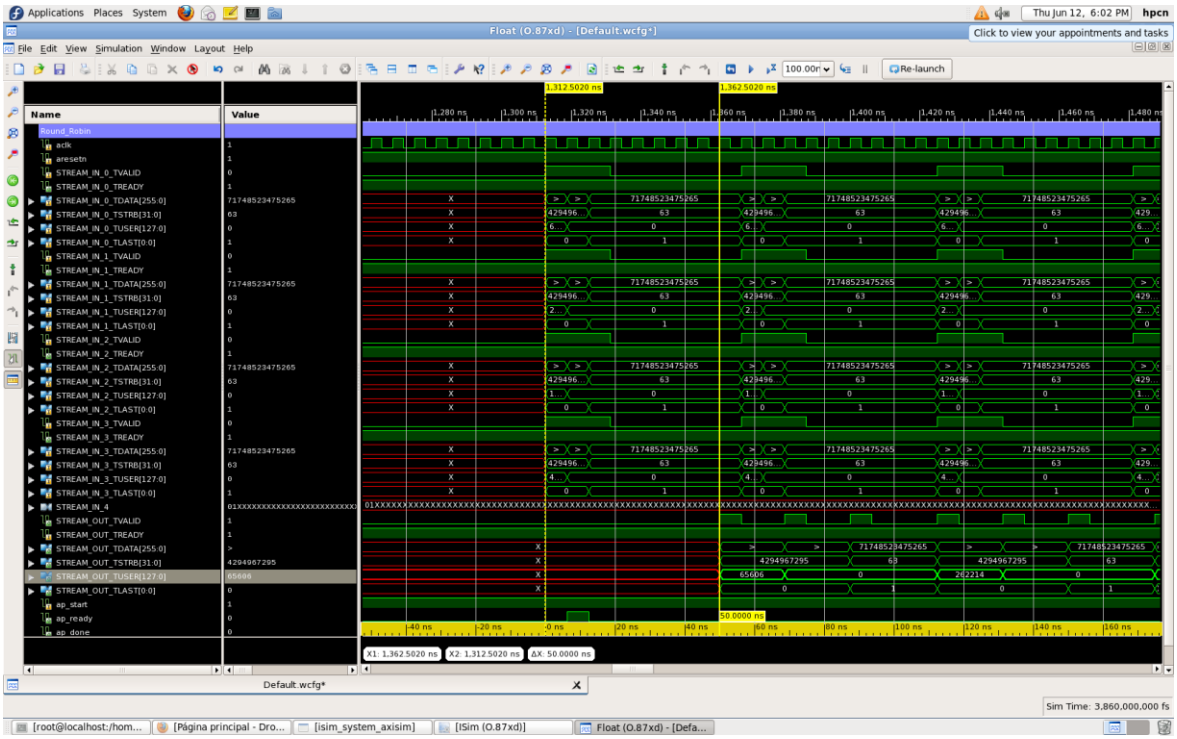


Figura 37: Round Robin Procesado inicial

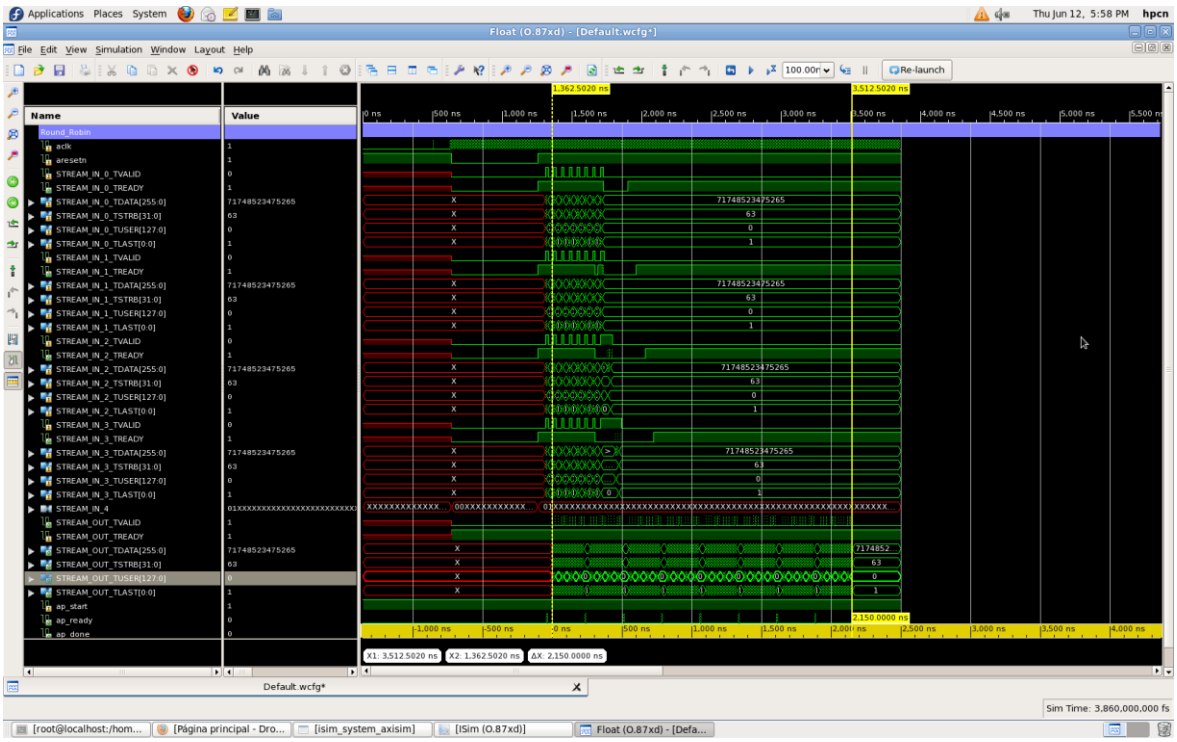


Figura 38: Round Robin Tiempo Total

C Mejora: Prioridad según destino

```
#include "round_robin.h"
#include <ap_int.h>
#include <stdint.h>
#include <stddef.h>
#include <hls_stream.h>
#include <sys/types.h>
#include <unistd.h>

//Global parameters
axi_interface_type read_nf0, read_nf1, read_nf2, read_nf3, read_nf4;
uint32_t j=0,k=1;

void round_robin_uam (hls::stream<axi_interface_type> &data_input_nf0,
hls::stream<axi_interface_type> &data_input_nf1, hls::stream<axi_interface_type>
&data_input_nf2, hls::stream<axi_interface_type> &data_input_nf3,
hls::stream<axi_interface_type> &data_input_nf4, hls::stream<axi_interface_type>
&data_output){

//Map HLS ports to AXI interfaces. Tell the compiler this interfaces are AXI4-Stream
#pragma HLS RESOURCE variable=data_input_nf0        core=AXIS metadata="-bus_bundle
STREAM_IN_0"
#pragma HLS RESOURCE variable=data_input_nf1        core=AXIS metadata="-bus_bundle
STREAM_IN_1"
#pragma HLS RESOURCE variable=data_input_nf2        core=AXIS metadata="-bus_bundle
STREAM_IN_2"
#pragma HLS RESOURCE variable=data_input_nf3        core=AXIS metadata="-bus_bundle
STREAM_IN_3"
#pragma HLS RESOURCE variable=data_input_nf4        core=AXIS metadata="-bus_bundle
STREAM_IN_4"
#pragma HLS RESOURCE variable=data_output           core=AXIS metadata="-bus_bundle
STREAM_OUT"

        int i=0, queue, second_round=0;
        axi_interface_type read[200], ip_dest;
        int ip_min, ip_max, not_empty=0;

        ip_min=0x96F438;
        ip_max=0x96F43B;


                loop_queues: for(i=1;i<6;i++){
                                        queue=0;

                                        switch (i){
                                                case 1: if(!data_input_nf0.empty()){
                                                        not_empty=1;

if(second_round==0){

//During the first round the program will read and store the first transaction of the
packet

  read_nf0 = data_input_nf0.read();

  ip_dest=read_nf0;

         if(ip_dest.data.range(128,151) > ip_min && ip_dest.data.range(127,151) <
```

```
ip_max){
                    //In case of having a packet going to EPS UAM it will start
queueing it to the output and the counter will
                    //be reduced by one in order to read the same FIFO in case anther
packet with the same destination arrives
                    queue=1;
                    read[++j]=read_nf0;
                    i--;}}
                                                                        else if
(second_round==1){
  queue = 1;
  read[++j]=read_nf0;}}
                                                        break;
                                                case 2: if(!data_input_nf1.empty()){
                                                        not_empty=1;
if(second_round==0){
read_nf1 = data_input_nf1.read();
ip_dest=read_nf1;
  if(ip_dest.data.range(128,151) > ip_min && ip_dest.data.range(127,151) < ip_max){
          queue=2;
          read[++j]=read_nf1;
          i--;}}
                                                                        else if
(second_round==1){
  queue = 2;
  read[++j]=read_nf1;}}
                                                        break;
                                                case 3: if(!data_input_nf2.empty()){
                                                        not_empty=1;
if(second_round==0){
read_nf2= data_input_nf2.read();
ip_dest=read_nf2;
  if(ip_dest.data.range(128,151) > ip_min && ip_dest.data.range(127,151) < ip_max){
          queue=3;
          read[++j]=read_nf2;
          i--;}}
                                                                        else if
(second_round==1){
```

```
  queue = 3;
  read[++j]=read_nf2;}}
                                                    break;
                                                case 4: if(!data_input_nf3.empty()){
                                                    not_empty=1;
if(second_round==0){
read_nf3 = data_input_nf3.read();
ip_dest=read_nf3;
  if(ip_dest.data.range(128,151) > ip_min && ip_dest.data.range(127,151) < ip_max){
          queue = 4;
          read[++j]=read_nf3;
          i--;}}
                                                                                  else if
(second_round==1){
          queue = 4;
          read[++j]=read_nf3;}}
                                                    break;
                                                case 5: if(!data_input_nf4.empty()){
                                                    not_empty=1;
if(second_round==0){
read_nf4 = data_input_nf4.read();
ip_dest=read_nf4;
  if(ip_dest.data.range(128,151) > ip_min && ip_dest.data.range(127,151) < ip_max){
          queue = 5;
          read[++j]=read_nf4;
          i--;}}
                                                                                  else if
(second_round==1){
  queue = 5;
  read[++j]=read_nf4;}}
                                                    break;
                                            }

                                            if (queue!=0){
                loop_read_packet: do{
                        #pragma HLS PIPELINE
                                            if((read[j].last)==0){
                                                    switch (queue){
                                                                                  case 1:
read[++j] = data_input_nf0.read(); break;
                                                                                  case 2:
read[++j] = data_input_nf1.read(); break;
```

```
                                                                    case 3:
read[++j] = data_input_nf2.read(); break;
                                                                    case 4:
read[++j] = data_input_nf3.read(); break;
                                                                    case 5:
read[++j] = data_input_nf4.read(); break;
                                                                }}
                                                        data_output.write(read[k]);
                                                } while(!(read[k++].last));
                                        }
                                        //This will initiate the loop and start the
second round
                                        if(i==5 && second_round==0 && not_empty==1){
                                                i=0;
                                                second_round=1;
                                        }
                                }
}
```

PRESUPUESTO

1) Ejecución Material

- Compra de ordenador personal (Software incluido)..... 2.000 €
- Compra de ordenador personal (Software incluido)..... 2.000 €
- Alquiler de impresora láser durante 6 meses..... 50 €
- Material de oficina ..... 150 €
- Placa NetFPGA..... 1.500€
- Placa Reference NIC..... 500€
- Fibras Ópticas ..... 150€
- Conectores SFP+..... 350€

- Total Ejecución Material..... 6.700€

2) Gastos generales

- 16 % sobre Ejecución Material ..... 1.072 €

3) Beneficio Industrial

- 6 % sobre Ejecución Material ..... 402 €

4) Honorarios Proyecto

- 850 horas a 20 € / hora..... 17.000 €

5) Material fungible

- Gastos de impresión..... 60 €
- Encuadernación..... 200 €

6) Subtotal del presupuesto

- Subtotal Presupuesto ..... 23.960 €

7) I.V.A. aplicable

- 16% Subtotal Presupuesto ..... 3.833,6 €

8) Total presupuesto

- Total Presupuesto..... 27.793,6 €

Madrid, Julio del 2014
El Ingeniero Jefe de Proyecto
Fdo.: Hugo Peire García
Ingeniero de Telecomunicación

PLIEGO DE CONDICIONES

Este documento contiene las condiciones legales que guiarán la realización, en este proyecto, Desarrollo mediante lenguaje de alto nivel de un sistema basado en FPGA para aplicaciones de red en 40 Gbps Ethernet. En lo que sigue, se supondrá que el proyecto ha sido encargado por una empresa cliente a una empresa consultora con la finalidad de realizar dicho sistema. Dicha empresa ha debido desarrollar una línea de investigación con objeto de elaborar el proyecto. Esta línea de investigación, junto con el posterior desarrollo de los programas está amparada por las condiciones particulares del siguiente pliego.

Supuesto que la utilización industrial de los métodos recogidos en el presente proyecto ha sido decidida por parte de la empresa cliente o de otras, la obra a realizar se regulará por las siguientes:

Condiciones generales

1. La modalidad de contratación será el concurso. La adjudicación se hará, por tanto, a la proposición más favorable sin atender exclusivamente al valor económico, dependiendo de las mayores garantías ofrecidas. La empresa que somete el proyecto a concurso se reserva el derecho a declararlo desierto.

2. El montaje y mecanización completa de los equipos que intervengan será realizado totalmente por la empresa licitadora.

3. En la oferta, se hará constar el precio total por el que se compromete a realizar la obra y el tanto por ciento de baja que supone este precio en relación con un importe límite si este se hubiera fijado.

4. La obra se realizará bajo la dirección técnica de un Ingeniero Superior de Telecomunicación, auxiliado por el número de Ingenieros Técnicos y Programadores que se estime preciso para el desarrollo de la misma.

5. Aparte del Ingeniero Director, el contratista tendrá derecho a contratar al resto del personal, pudiendo ceder esta prerrogativa a favor del Ingeniero Director, quien no estará obligado a aceptarla.

6. El contratista tiene derecho a sacar copias a su costa de los planos, pliego de condiciones y presupuestos. El Ingeniero autor del proyecto autorizará con su firma las copias solicitadas por el contratista después de confrontarlas.

7. Se abonará al contratista la obra que realmente ejecute con sujeción al proyecto que sirvió de base para la contratación, a las modificaciones autorizadas por la superioridad o a las órdenes que con arreglo a sus facultades le hayan comunicado por escrito al Ingeniero Director de obras siempre que dicha obra se haya ajustado a los preceptos de los pliegos de condiciones, con arreglo a los cuales, se harán las modificaciones y la valoración de las diversas unidades sin que el importe total pueda exceder de los presupuestos aprobados. Por consiguiente, el número de unidades que se consignan en el proyecto o en el presupuesto, no podrá servirle de fundamento para entablar reclamaciones de ninguna clase, salvo en los casos de rescisión.

8. Tanto en las certificaciones de obras como en la liquidación final, se abonarán los trabajos realizados por el contratista a los precios de ejecución material que figuran en el presupuesto para cada unidad de la obra.

9. Si excepcionalmente se hubiera ejecutado algún trabajo que no se ajustase a las condiciones de la contrata pero que sin embargo es admisible a juicio del Ingeniero Director de obras, se dará conocimiento a la Dirección, proponiendo a la vez la rebaja de precios que el Ingeniero estime justa y si la Dirección resolviera aceptar la obra, quedará el contratista obligado a conformarse con la rebaja acordada.

10. Cuando se juzgue necesario emplear materiales o ejecutar obras que no figuren en el presupuesto de la contrata, se evaluará su importe a los precios asignados a otras obras o materiales análogos si los hubiere y cuando no, se discutirán entre el Ingeniero Director y el contratista, sometiéndolos a la aprobación de la Dirección. Los nuevos precios convenidos por uno u otro procedimiento, se sujetarán siempre al establecido en el punto anterior.

11. Cuando el contratista, con autorización del Ingeniero Director de obras, emplee materiales de calidad más elevada o de mayores dimensiones de lo estipulado en el proyecto, o sustituya una clase de fabricación por otra que tenga asignado mayor precio o ejecute con mayores dimensiones cualquier otra parte de las obras, o en

general, introduzca en ellas cualquier modificación que sea beneficiosa a juicio del Ingeniero Director de obras, no tendrá derecho sin embargo, sino a lo que le correspondería si hubiera realizado la obra con estricta sujeción a lo proyectado y contratado.

12. Las cantidades calculadas para obras accesorias, aunque figuren por partida alzada en el presupuesto final (general), no serán abonadas sino a los precios de la contrata, según las condiciones de la misma y los proyectos particulares que para ellas se formen, o en su defecto, por lo que resulte de su medición final.

13. El contratista queda obligado a abonar al Ingeniero autor del proyecto y director de obras así como a los Ingenieros Técnicos, el importe de sus respectivos honorarios facultativos por formación del proyecto, dirección técnica y administración en su caso, con arreglo a las tarifas y honorarios vigentes.

14. Concluida la ejecución de la obra, será reconocida por el Ingeniero Director que a tal efecto designe la empresa.

15. La garantía definitiva será del 4% del presupuesto y la provisional del 2%.

16. La forma de pago será por certificaciones mensuales de la obra ejecutada, de acuerdo con los precios del presupuesto, deducida la baja si la hubiera.

17. La fecha de comienzo de las obras será a partir de los 15 días naturales del replanteo oficial de las mismas y la definitiva, al año de haber ejecutado la provisional, procediéndose si no existe reclamación alguna, a la reclamación de la fianza.

18. Si el contratista al efectuar el replanteo, observase algún error en el proyecto, deberá comunicarlo en el plazo de quince días al Ingeniero Director de obras, pues transcurrido ese plazo será responsable de la exactitud del proyecto.

19. El contratista está obligado a designar una persona responsable que se entenderá con el Ingeniero Director de obras, o con el delegado que éste designe, para todo relacionado con ella. Al ser el Ingeniero Director de obras el que interpreta el proyecto, el contratista deberá consultarle cualquier duda que surja en su realización.

20. Durante la realización de la obra, se girarán visitas de inspección por personal facultativo de la empresa cliente, para hacer las comprobaciones que se crean oportunas. Es obligación del contratista, la conservación de la obra ya ejecutada hasta la recepción de la misma, por lo que el deterioro parcial o total de ella, aunque sea por agentes atmosféricos u otras causas, deberá ser reparado o reconstruido por su cuenta.

21. El contratista, deberá realizar la obra en el plazo mencionado a partir de la fecha del contrato, incurriendo en multa, por retraso de la ejecución siempre que éste no sea debido a causas de fuerza mayor. A la terminación de la obra, se hará una recepción provisional previo reconocimiento y examen por la dirección técnica, el depositario de efectos, el interventor y el jefe de servicio o un representante, estampando su conformidad el contratista.

22. Hecha la recepción provisional, se certificará al contratista el resto de la obra, reservándose la administración el importe de los gastos de conservación de la misma hasta su recepción definitiva y la fianza durante el tiempo señalado como plazo de garantía. La recepción definitiva se hará en las mismas condiciones que la provisional, extendiéndose el acta correspondiente. El Director Técnico propondrá a la Junta Económica la devolución de la fianza al contratista de acuerdo con las condiciones económicas legales establecidas.

23. Las tarifas para la determinación de honorarios, reguladas por orden de la Presidencia del Gobierno el 19 de Octubre de 1961, se aplicarán sobre el denominado en la actualidad “Presupuesto de Ejecución de Contrata” y anteriormente llamado ”Presupuesto de Ejecución Material” que hoy designa otro concepto.

Condiciones particulares

La empresa consultora, que ha desarrollado el presente proyecto, lo entregará a la empresa cliente bajo las condiciones generales ya formuladas, debiendo añadirse las siguientes condiciones particulares:

1. La propiedad intelectual de los procesos descritos y analizados en el presente trabajo, pertenece por entero a la empresa consultora representada por el Ingeniero Director del Proyecto.

2. La empresa consultora se reserva el derecho a la utilización total o parcial de los resultados de la investigación realizada para desarrollar el siguiente proyecto, bien para su publicación o bien para su uso en trabajos o proyectos posteriores, para la misma empresa cliente o para otra.

3. Cualquier tipo de reproducción aparte de las reseñadas en las condiciones generales, bien sea para uso particular de la empresa cliente, o para cualquier otra aplicación, contará con autorización expresa y por escrito del Ingeniero Director del Proyecto, que actuará en representación de la empresa consultora.

4. En la autorización se ha de hacer constar la aplicación a que se destinan sus reproducciones así como su cantidad.

5. En todas las reproducciones se indicará su procedencia, explicitando el nombre del proyecto, nombre del Ingeniero Director y de la empresa consultora.

6. Si el proyecto pasa la etapa de desarrollo, cualquier modificación que se realice sobre él, deberá ser notificada al Ingeniero Director del Proyecto y a criterio de éste, la empresa consultora decidirá aceptar o no la modificación propuesta.

7. Si la modificación se acepta, la empresa consultora se hará responsable al mismo nivel que el proyecto inicial del que resulta el añadirla.

8. Si la modificación no es aceptada, por el contrario, la empresa consultora declinará toda responsabilidad que se derive de la aplicación o influencia de la misma.

9. Si la empresa cliente decide desarrollar industrialmente uno o varios productos en los que resulte parcial o totalmente aplicable el estudio de este proyecto, deberá comunicarlo a la empresa consultora.

10. La empresa consultora no se responsabiliza de los efectos laterales que se puedan producir en el momento en que se utilice la herramienta objeto del presente proyecto para la realización de otras aplicaciones.

11. La empresa consultora tendrá prioridad respecto a otras en la elaboración de los proyectos auxiliares que fuese necesario desarrollar para dicha aplicación industrial, siempre que no haga explícita renuncia a este hecho. En este caso, deberá autorizar expresamente los proyectos presentados por otros.

12. El Ingeniero Director del presente proyecto, será el responsable de la dirección de la aplicación industrial siempre que la empresa consultora lo estime oportuno. En caso contrario, la persona designada deberá contar con la autorización del mismo, quien delegará en él las responsabilidades que ostente.