Universidad Autónoma de Madrid
Escuela politécnica superior

Grado en Ingenería Informática

Trabajo Fin de Grado

GESTIÓN DE ANCHO DE BANDA EN TRÁFICO DE RED EN HARDWARE RECONFIGURABLE

Autor: Sergio Sanz Cebollero
Tutor: Gustavo Sutter Capristo

Mayo 2017

GESTIÓN DE ANCHO DE BANDA EN TRÁFICO DE RED EN HARDWARE RECONFIGURABLE

Autor: Sergio Sanz Cebolllero
Tutor: Gustavo Sutter Capristo

High Performance Computing and Networking research group
Dpto. de Ingenería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo 2017

Resumen

En los últimos años, el mundo de las transacciones en los mercados financieros ha sufrido una gran transformación con la llegada de los sistemas de transacciones de alta frecuencia HFT ejecutados mediante computadores de altas prestaciones.

Este tipo de actividad, requiere de una calidad de servicio en cuanto a tiempo de respuesta que los enlaces gestionados mediante *software* no pueden proporcionar. En este proyecto, pretendemos desarrollar un prototipo de diseño *hardware* que pueda proporcionar dicho servicio.

Este diseño, busca acelerar un algoritmo ya existente de gestión de ancho de banda como el *token bucket* haciendo uso de un dispositivo hardware reconfigurable. Este tipo de dispositivo hardware, al ser programable nos va a permitir desarrollar el prototipo en un tiempo y coste asumibles.

Palabras Clave

Hardware, Conformado de tráfico, Redes, Mercados Financieros, Transacciones de Alta Frecuencia.

Abstract

In the last few years, the world's stock markets have evolved a lot thanks to introduction of high frequency trading systems that are usually run of High Performance Computers.

This kind of systems require a quality of service in terms of latency that current software solutions cannot provide. This project approaches this problem from the hardware perspective, developing a hardware design solution that actually gives an optimal performance.

This design focuses on an acceleration of the token bucket algorithm taking advantage of a hardware programmable device. This kind of device allows us to achieve a faster and cheaper implementation.

Key words

Hardware, Traffic Policing, Network, Stock Markets, High Frequency Trading.

Agradecimientos

Me gustaría dar las gracias a las siguientes personas vinculadas al HPCN. A mi tutor Gustavo, por haberme dado la oportunidad de embarcarme en este proyecto y por sus valiosos consejos. A Mario y a Jose Fernando, agradecerle su disponibilidad para ayudarme en el laboratorio durante los momentos en los que he estado perdido. A Dixon por su generosidad desde el primer día que nos conocimos, por compartir sus recursos y conocimientos conmigo, por esas conversaciones hablando de nuestros proyectos y por enseñarme la belleza de Colombia, sé que serás todo un PhD. ¡Patrón!.

A Hao por ser un gran compañero de prácticas durante toda la carrera en tantas asignaturas desde aquellas prácticas de Programación I que hacíamos en la biblioteca en nuestro primer mes de carrera. Creo que hemos conseguido sacar lo mejor del otro en los momentos más difíciles de cada uno. ¡Espero que vuelvas a dar señales de vida!.

Por último, mencionar a los mejores y los más brillantes compañeros que he tenido nunca: Guillermo, Alex y Pablo. Gracias por esos chistes, debates y tardes de cañas tan enriquecedoras que espero que no se terminen aunque cada uno tomemos un camino en este inmenso mundo de la informática. El tiempo que paso con vosotros me hace ser cada vez mejor y en consecuencia me acerca a paso a paso a esas metas que quiero conseguir.

Índice general

Índice de figuras

Glosario

- **ATM**: Asynchronous Transfer Mode
- **AXI**: Advanced eXtensible Interface
- **BRAM**: Block Ram
- **CAM**: Content Address Memory
- **DMA**: Direct Access Memory
- **FDDI**: Fiber Distributed Data Interface
- **FIFO**: First In First Out
- **FPGA**: Field Programmable Gate Array
- **GNU**: Gnu's Not Unix
- **HCU**: Hazard Control Unit
- **HFT**: High Frequency Trading
- **IEEE**: Institute of Electrical and Electronic Engineers
- **IP**: Internet Protocol
- **NASDAQ**: National Association of Securities Dealers Automated Quotation
- **NIC**: Network Interface Controllero
- **MAC**: Media Access Control
- **MBS**: Maximum Burst Size
- **MTU**: Maximum Transfer Unit
- **PAR**: Place and Route
- **PCAP**: Packet Capture
- **RTL**: Register Transfer Language

- **SSH**: Secure Shell
- **UDP**: User Datagram Protocol
- **WAW**: Write After Write
- **XNGD**: Xilinx Native Generic Database
- **XMD**: Xilinx Microprocessor Debugger

1 Introducción

1.1. Introducción al mercado electrónico

La aparición del NASDAQ en 1971 [24] (el primer mercado electrónico) supuso un hito en la historia del mercado bursátil [10]. Este hecho dió comienzo a una revolución dentro de los mercados financieros donde los ordenadores e internet iban a ser los principales protagonistas.

En la siguiente década, la empresa Bloomberg L.P [18] lanzó en 1983 el primer ordenador financiero. Era una máquina con un sistema que ofrecía datos de los mercados en tiempo real y servía también para realizar tanto cálculo como análisis financiero [19].

En los años 90, tras cambios regulatorios [14] y gracias al avance tantos en los sistemas de análisis y monitorización como en el campo de la inteligencia artificial, surgió el HFT [23]. Es una modalidad de inversión donde la negociación de los activos es llevada a cabo por ordenadores. Dichos ordenadores, ejecutan algoritmos diseñados por compañías del sector financiero y son capaces de abrir y cerrar múltiples posiciones de inversión en un intervalo de tiempo muy pequeño.

Los ordenadores de aquella época ya eran capaces de realizar operaciones 1000 veces más rápido que el ser humano [9]. Desde entonces, los tiempos de las transacciones HFT se han reducido de forma dramática. Actualmente, con los algoritmos actuales existen chips capaces de ejecutar transacciones en orden de nanosegundos [22].

1.2. Ámbito de este proyecto

El factor principal que hace a los sistemas HFT tan potentes, es el tiempo que tardan en cerrar una operación. Cuánto más rápido sea el sistema, mayores oportunidades de negocio podrá

hacer efectivas. Este tiempo depende de principalmente de tres aspectos. Dos de ellos los hemos introducido anteriormente: **la potencia de la máquina** y **la calidad del algoritmo**.

El tercero y último y en el cual entra este proyecto, es **la calidad del enlace de red** por el circula el flujo de operaciones. Si tenemos un enlace que sufre de alta latencia, de poco nos sirve contar con un sistema HFT de última generación, ya que sus órdenes tardaran demasiado en llegar al mercado y en ese tiempo las condiciones del mercado han podido cambiar, afectando así al impacto de las decisiones que se toman.

Existen numerosas formas de mejorar los tiempos de latencia en un enlace. La posibilidad que planteamos con este proyecto es acelerar un algoritmo de **conformado de tráfico** conocido como *token bucket* [1, Cap. 5, Sec. 4.2], utilizando hardware FPGA [29] (Field Programmable Gate Array). Esta práctica lleva realizándose algunos años con el objetivo de mejorar el rendimiento de algoritmos HFT [16].

Nuestro objetivo con este proyecto es comprobar si esta técnica de aceleración que ha tenido tan buenos resultados en el rendimiento de algoritmos de otras disciplinas, lo podemos aplicar para poder proporcionar una latencia a la altura del rendimiento de los sistemas HFT de los clientes.

1.3. Estructura del documento

El presente documento se compone a continuación de cinco apartados de contenido. El primero a tratar será Estado del Arte, donde vamos a tratar algoritmos, herramientas y dispositivos disponibles que existen en la actualidad para ayudarnos a resolver el problema a tratar y el uso potencial que tienen.

A continuación daremos una descripción técnica del problema , presentaremos la solución elaborada y por último daremos una descripción de la construcción del prototipo y los resultados de las pruebas a nivel funcional.

Vistos ya los resultados de la simulación, el siguiente capítulo tratará del despliegue del diseño sobre la plataforma de hardware reconfigurable elegida.

Por último, reservamos un apartado de conclusiones para realizar un balance sobre los resultados obtenidos, posibles aportes derivados del desarrollo del proyecto y si éste da pie a la realización de trabajos futuros.

2

Estado del Arte

2.1. Enlaces Gigabit Ethernet

Desde que Xerox publicó su primera especificación en 1973 [12], el protocolo Ethernet ha sufrido diversas evoluciones conforme ha ido aumentando la demanda de tráfico, tal y como podemos apreciar en la siguiente línea temporal publicada en un artículo de la asociación IEEE[13].

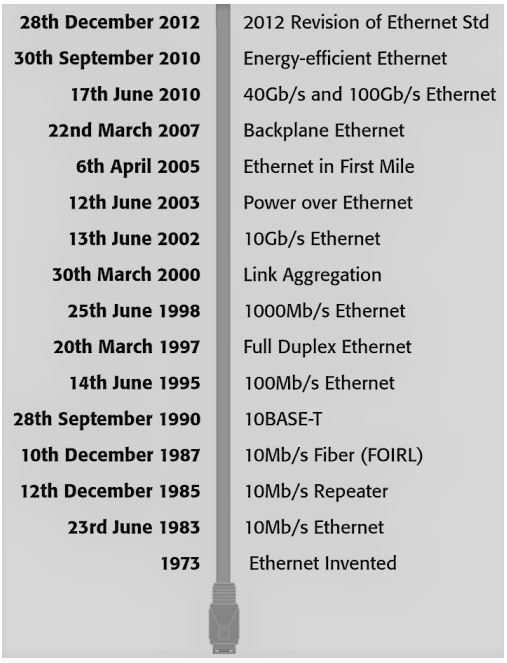

Figura 2.1: Linea histórica del protocolo Ethernet.

Pese a que existen protocolos alternativos a Ethernet (tales como ATM [21], FDDI [4]), este protocolo se ha mantenido como estándar de facto debido principalmente a cuatro razones:

- Es un protocolo simple.
- Bajo coste tanto de mejora como de mantenimiento.
- Ha sido capaz de evolucionar y mejorar sus prestaciones cuando le ha sido requerido.
- Opera muy bien con el protocolo IP, ya que ambos no son orientados a conexión.

Como ya vimos anteriormente, el último estándar de Ethernet definido, nos permite operar a tasas de 40 y 100 Gbps. Cada unidad del modelo FPGA con el que vamos a trabajar (cuya especificación detallaremos más adelante), puede trabajar sobre enlaces Gigabit Ethernet de hasta 40 Gbps sin incluirle ningún tipo de expansión.

Uniendo ambos elementos, lo que se pretende conseguir es construir una red de altas prestaciones con tasas de latencia muy bajas. Para consultar de forma más detallada aspectos sobre Ethernet, se puede consultar [1, Cap. 4, Sec.3].

2.2. Tráfico sintético

Un método muy extendido en el desarrollo de sistemas que analizan o gestionan tráfico y por tanto, se han de conectar a una interfaz de red física, es la emulación de dicha interfaz mediante el uso de ficheros que contienen la información que recibiríamos de la misma.

El uso de dichos ficheros, nos va a permitir trabajar con tráfico de Internet sin tener que implantar el diseño en una FPGA que se encuentre conectada a la red. Esto supone un gran ahorro en términos de tiempo de desarrollo, ya que conseguiremos detectar los errores sin tener que invertir tiempo en desplegar el sistema dentro de la red.

En la fase de puesta en marcha del sistema, estos ficheros también son muy útiles, puesto que a la hora de realizar pruebas, son una herramienta que nos permite transmitir tráfico específico que cumpla ciertas características, lo que ayuda enormemente a dar más utilidad y validez a los experimentos.

Estos ficheros, se denominan comúnmente **trazas** o archivos *pcap* [11]. Es el formato estándar que utilizan herramientas de sistemas tipo UNIX como *tcpdump*, *tcpreplay*, *tcprewrite* o *wireshark*. Son herramientas de red de uso extendido para modificar, transmitir y generar trazas

Para generar dichos ficheros vamos a hacer uso de *wireshark* [5]. Es una herramienta utilizada para capturar y analizar tráfico que circula por una red (en el argot, se conoce comúnmente como *sniffer*).

Una vez obtenido el tráfico, haremos uso de la herramienta *tcprewrite* [15] para editar la cabecera de los paquetes y así poder obtener trazas que se ajusten más a lo que requieran las simulaciones de comportamiento del sistema.

Para la fase de implantación, la herramienta *tcpreplay* [2] es muy utilizada para enviar tráfico que se encuentra en una traza a través de la red física. Esto permite por ejemplo, poder enviar tráfico sintético desde un ordenador a una FPGA que se encuentre conectada a una interfaz Ethernet.

Para obtener el tráfico que transmite la FPGA, podemos utilizar tanto *tcpdump* [28] como *Wireshark*. Para leer el tráfico de la FPGA, vamos a utilizar *tcpdump*, puesto que no requiere del uso de interfaz gráfica y puede ser manejada remotamente a través de un intérprete SSH, herramienta que nos va a permitir controlar un servidor desde cualquier otro ordenador.

2.3. Hardware reconfigurable orientado a procesamiento de redes

Para ejecutar algoritmos de conformado de tráfico, necesitamos un dispositivo que posea una interfaz de red. Además de al menos poseer una interfaz de red, debe ser un hardware con la potencia suficiente como para poder ser conectado a redes de altas prestaciones como la que hemos descrito anteriormente.

La plataforma que hemos elegido para implementar este proyecto ha sido la plataforma de **hardware** abierto NetFPGA SUME [25]. Es una FPGA de la serie Virtex 7 de Xilinx, concretamente el modelo 690T.

Según los propios desarrolladores [33], es una plataforma pensada para probar prototipos de forma ágil a 10 o 40 Gbps y está especialmente dirigida a trabajos académicos y de investigación, por lo que cumple con las necesidades de hardware que requiere este proyecto.

El proyecto que presentamos en este trabajo, lo vamos a encajar dentro el proyecto **Reference NIC** [20] el cual ya nos permite de base leer y transmitir paquetes a utilizando interfaces Gigabit Ethernet. Daremos más detalles sobre el Reference NIC y la modificación que hemos realizado en el mismo en capítulos posteriores

2.4. Metodologías y herramientas de desarrollo

Antes de describir el proyecto en profundidad, vamos a describir como ha sido el desarrollo y las herramientas que nos han ayudado a hacerlo posible.

En primer lugar, realizamos un diseño esquemático superficial de la arquitectura, conectando los módulos de forma lógica sin excesivo detalle. En la siguiente figura podemos ver un ejemplo de esquemático preliminar producto de esta primera etapa.

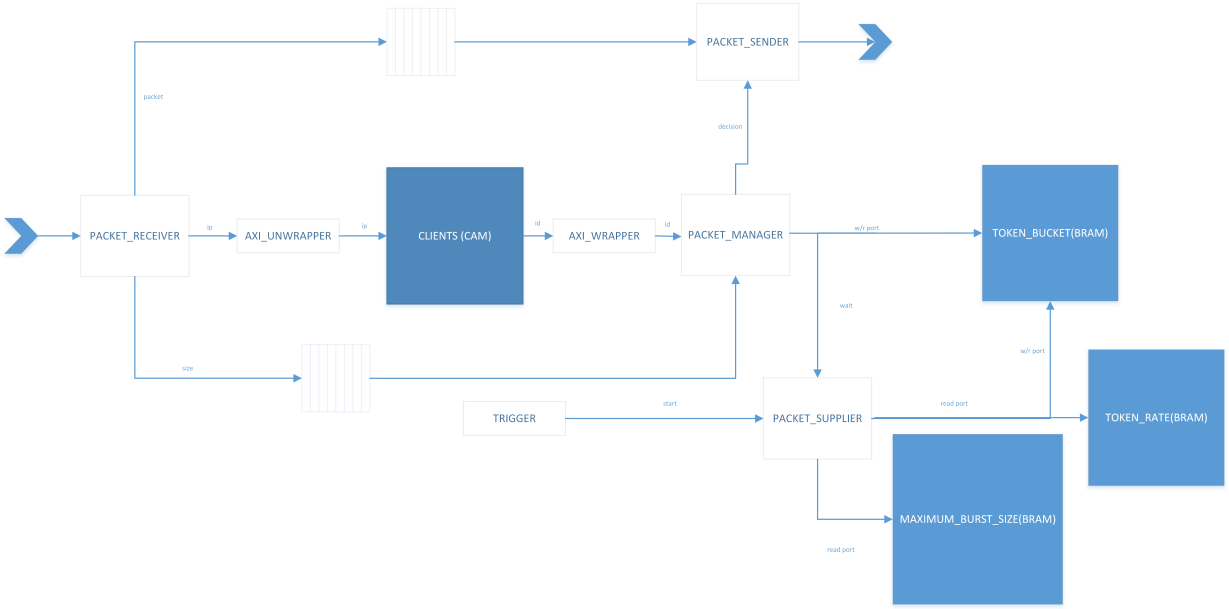

Figura 2.2: Diseño preliminar del sistema de *policing*

El siguiente paso, es desacoplar el proyecto en varios sistemas buscando la máxima cohesión y el mínimo acoplamiento. Una vez tenemos definidos los sistemas resultantes, comenzaría la etapa de desarrollo de cada sistema. Para cada uno de ellos, se sigue la misma metodología. Esta metodología se divide en los siguientes pasos:

1. Creación de un diseño esquemático del sistema, con la correspondiente definición de módulos.
2. Por cada módulo definido
   a) Se especifica su interfaz de entrada y salida.
   b) Se especifica su régimen de funcionamiento
   c) Se escribe tanto el código fuente como el código de prueba (conocido en el argot como *testbench*)
   d) Se realiza la simulación y se contrastan los resultados obtenidos con los resultados esperados. Si estos no coinciden, se ha de volver sobre alguno de los pasos anteriores.
   e) Se realiza una síntesis para comprobar que el código desarrollado es convertible a *hardware*.
3. Se escribe el código fuente donde se conectan todos los módulos que pertenecen al sistema.
4. Al igual que con los módulos, se escribe un código de prueba para comprobar su funcionamiento.
5. Se realiza la simulación del funcionamiento de todos los sistemas ya integrados y se contrastan los resultados obtenidos con los resultados esperados.

Una vez terminado el desarrollo de los sistemas, seguimos de forma recurrente el mismo enfoque. Creamos un código de prueba con todos los sistemas conectados entre sí y comprobamos si el funcionamiento se corresponde con la especificación definida previamente.

El desarrollo de este proyecto se ha realizado utilizando el entorno de desarrollo integrado (también conocido como IDE) Vivado en su versión 2014.4. Es la plataforma de desarrollo de referencia para todas las FPGA de Xilinx, ya que es utilizada en todas las fases del proyecto (tanto desarrollo, simulación, síntesis como en la implantación dentro de la FPGA) [32].

2.5. Algoritmos de gestión de ancho de banda

En este apartado, vamos a explicar el concepto del algoritmo *token bucket* y el de algunas de sus versiones. Si se quiere ampliar la información descrita en esta sección, se recomienda consultar [1, Cap. 5, Sec. 4].

Imaginemos que tenemos un cubo, el cual se rellena mediante un grifo. Este artefacto, presenta tres características a tener en cuenta:

- El cubo se rellena a una velocidad constante (R).
- El cubo se vacía a una velocidad variable.
- El volumen del cubo (B) es finito, por lo que todo rebase de su capacidad máxima, se desperdicia.
- Como el volumen del cubo es finito, podemos extraer como máximo un volumen equivalente a la capacidad del cubo dentro de una unidad de tiempo.

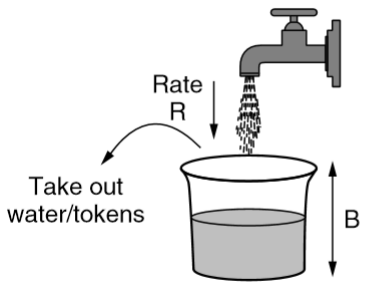

Figura 2.3: Figura extraída de [1, p. 408].

Ahora bien, ¿de qué manera se utiliza este cubo para que ayude a gestionar un enlace de tráfico? Antes de comenzar a explicar el funcionamiento del algoritmo, debemos establecer la siguiente analogía:

- La tasa de llenado del cubo se corresponde al ancho de banda disponible.
- La tasa de vaciado del cubo equivale al ancho de banda efectivo que se consume.
- La capacidad del cubo se puede entender como la ráfaga máxima que se puede transmitir a través de un enlace.
- En vez de trabajar con volumen de agua, vamos a trabajar con volumen de información, por lo que la unidad de extracción con la que vamos a trabajar va a ser el *byte*, ficha o *token*. Estos tres términos se utilizarán de forma indistinta de ahora en adelante.

También aclarar, que el identificador de un cubo es la dirección IP. Dicho esto y definida nuestra estructura de datos, vamos a pasar a definir el algoritmo *token bucket*. Su funcionamiento queda especificado a continuación:

Datos: Tamaño e IP origen del paquete
Resultado: Se transmite el paquete o no se transmite
Obtener el número de fichas disponibles en el cubo a partir de la IP;
si *el tamaño del paquete no supera el número de fichas del cubo* **entonces**
    devolver Transmitir el paquete;
en otro caso
    devolver No transmitir el paquete;
fin

De forma simultánea el cubo es rellenado por el sistema en función del ancho de banda que le corresponde al cliente que transmite.

2.5.1. Metodologías para calcular el volumen del cubo

El volumen del cubo (también denominado tamaño máximo de ráfaga) es un parámetro cuyo valor óptimo se ha de hallar de forma empírica. Es importante establecer un valor adecuado, puesto de que el dependerá en gran medida el buen aprovechamiento del enlace con el que estamos trabajando.

Este valor, depende principalmente del uso que el cliente haga del enlace. Sin embargo, existen ciertos métodos y prácticas generales que son llevados a cabo en la industria que nos pueden ayudar a establecer rápidamente un valor de referencia [26].

Para validar los resultados que vayamos a obtener con dichas metodologías, vamos a enunciar recomendaciones para el tamaño máximo de ráfaga que queramos aplicar.

- El tamaño máximo de ráfaga no se recomienda que sea inferior a 10 veces la MTU y bajo ningún concepto puede ser inferior a dicha MTU. Entendemos como MTU el tamaño

máximo de transacción con el que trabaja el protocolo de capa de enlace que la red este utilizando.

- El tamaño del cubo debería ser suficiente como para permitir que el cliente pueda transmitir al máximo de su tasa durante 5ms.

Como en nuestro caso vamos a trabajar con redes Gigabit Ethernet (cuya capa de enlace utiliza una MTU de 1500 bytes), tenemos que el tamaño máximo de ráfaga no debería de ser inferior a 15000 *bytes*. En este proyecto, vamos a trabajar con dos métodos para calcular un tamaño máximo de ráfaga inicial.

El primer método, está basado en el tamaño de la MTU y consiste simplemente en establecer un tamaño máximo de ráfaga equivalente a 10 veces la MTU.

$$B_s = 10 \cdot M_{TU} \tag{2.1}$$

El segundo método está basado en el periodo de transmisión máxima y se expresa con la siguiente ecuación:

$$B_s = \frac{B_c \cdot B_p}{8} \tag{2.2}$$

Donde B_c es el ancho de banda del cliente en bits por segundo y B_p es el periodo transmisión máxima en segundos.

Cuando el algoritmo de *policing* trabaja en una red que es utilizada por varios clientes, una métrica muy útil para ayudarnos a encontrar el tamaño máximo de ráfaga óptimo, es relacionar B_s con el ancho de banda que el cliente tiene disponible. De esta manera, podremos evaluar si la proporción de la tasa pico es demasiado alta respecto a la tasa que tiene contratada.

2.5.2. Otros algoritmos existentes

Existe otra variante del algoritmo donde en lugar de descartar el paquete (en inglés esta acción se conoce como *policing*), se mantiene en la cola de espera hasta que el cubo este suficientemente lleno y el paquete pueda ser transmitido (esta técnica se conoce como *shaping*). Teniendo estas dos variantes, ¿qué criterio utilizar para elegir entre una de las dos políticas?.

Si el enlace tiene requerimientos de baja latencia, probablemente lo más adecuado sea aplicar *policing*, ya que simplemente descartando el paquete no incurrimos en un aumento del tiempo de retardo.

Si por el contrario se considera prioritario que todos los paquetes lleguen íntegros a su destino, deberíamos considerar aplicar *shaping*, ya que aunque introduzcamos latencia, no se producirán pérdidas en el enlace, (siempre y cuando el nivel de congestión no haga saturar la cola de espera)

Nosotros en este proyecto hemos optado por aplicar *policing* ya que al orientar nuestro servicio al HFT, proporcionar enlaces de **mínima latencia** resulta crítico.

3

Arquitectura hardware propuesta

En este capítulo se presenta la arquitectura propuesta para implementar el algoritmo *token bucket* sobre la plataforma NetFPGA, realizando un análisis previo de las características del problema que este algoritmo viene a resolver. La descripción de la arquitectura comenzará por una descripción de la interfaz del sistema y posteriormente con un análisis de los componentes internos.

3.1. Análisis del problema

Antes de pasar a hablar de la arquitectura propuesta para la resolución del problema, vamos a repasar las características del mismo.

Al ser una red orientada al HFT, se prioriza la latencia sobre la integridad del envío, por lo que la política de conformado de tráfico que se requiere es *policing*. Esto significa, que ante un exceso en el uso del ancho de banda por parte del cliente, el procedimiento a seguir será descartar el paquete.

El algoritmo de conformado a implementar será *token bucket* del cual hemos hablado ya anteriormente. Esto implica que cada cliente tendrá tres atributos asociados los cuales deben mantenerse registrados: el número de fichas, el tamaño máximo de ráfaga y la tasa de llenado.

Estos tres atributos, han de ser calculados de forma previa a la puesta en marcha del sistema, en función del ancho de banda que quiera solicitar el cliente. El número de fichas será actualizado en tiempo real conforme el cliente vaya haciendo uso de la red. Los otros dos, serán actualizados bajo petición del cliente.

3.2. Parámetros configurables

El sistema que se presenta a continuación tiene la tarea de controlar el ancho de banda que se utiliza dentro un enlace Gigabit Ethernet. El comportamiento de este servicio, viene definido por numerosos parámetros. Algunos son inherentes al algoritmo *token bucket* y otros son propios de las características del diseño. Dichos parámetros son los que enumeramos a continuación:

- **Número de clientes**: El sistema está preparado para soportar un número variable de clientes usuarios del enlace.
- **Direccion IP clase C**: Para identificar a cada cliente, el sistema requiere de una IP de clase C para poder identificarlo correctamente. Esta dirección se corresponde con los tres primeros octetos de una dirección IP.
- **Tasa de *tokens***: A cada cliente le corresponde un determinado ancho de banda, que está directamente relacionado con el número de tokens que se van adicionando periódicamente
- **MBS**: Este parámetro define el tamaño máximo de ráfaga que le corresponde a un cliente, o lo que es lo mismo, el número de *tokens* que pueden quedar almacenados en su cubo.

Habrá que definir tantas direcciones IP clase C, tasas de tokens y tamaños de ráfaga como clientes hagan uso del sistema.

3.3. Interfaces

La figura que se muestra a continuación, se corresponde con la especificación de las interfaces de entrada y salida del componente

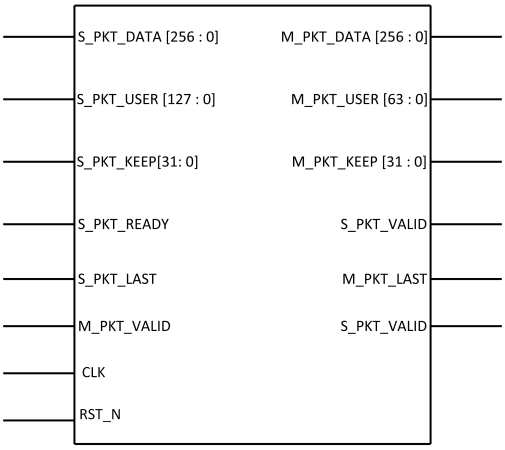

Figura 3.1: Esquemático del sistema

Este componente utiliza el protocolo **AXI Stream** en su versión 4.0 para recibir y transmitir datos entre dos entidades utilizando el paradigma maestro / esclavo . La empresa ARM lanzó

este protocolo en 1996 y la especificación utilizada en este proyecto se lanzó en 2010. Presenta un conjunto de señales imprescindibles y otras opcionales [30].

A continuación vamos a describir los buses AXI definidos para nuestro sistema:

- **READY**: Es una señal imprescindible. La utiliza el sistema esclavo para indicarle al maestro que se encuentra preparado para recibir una nueva transacción.
- **VALID**: señal imprescindible que utiliza el maestro para indicarle al esclavo que la transacción que está recibiendo es valida. Se mantendrá a uno mientras la señal READY se encuentre a cero.
- **LAST**: señal opcional que sólo se ha de incorporar cuando se requiere fragmentar la información en numerosas transacciones. Tiene valor uno cuando la transacción que transmite el maestro es la última de la ráfaga.
- **KEEP**: como en el caso de la señal LAST, sólo se ha de definir cuando el dato se envía en dos o más transacciones. Cuando nos encontramos en la última transacción y la señal LAST se encuentra activa, este vector nos indica los bits de la transacción que son validos. Si por ejemplo, de los 256 bits de la señal DATA, únicamente son validos los que se encuentran dentro del rango [0,2], la señal KEEP tendrá a uno únicamente los bits del rango [0,2].
- **DATA**: señal que contiene la transacción que se quiere transmitir siempre y cuando la señal VALID se encuentre activa. Todos los bits son válidos siempre y cuando las señales LAST y KEEP no nos indiquen lo contrario.
- **USER**: estos 128 bits, se definen de forma opcional y generalmente se utilizan para transmitir información extra acerca de la transmisión.

Como se puede apreciar en la figura 3.1, el componente presenta dos buses AXI Stream, uno de entrada que se utiliza para leer los paquetes fragmentados en forma de transacciones que provienen de la interfaz y otro de salida para transmitir las transacciones de los paquetes que el algoritmo de *policing* permita. Cuenta además con una señal de *reset* de **nivel bajo** y la correspondiente señal del reloj.

3.4. Descripción de la arquitectura

El sistema se compone de cinco IP independientes conectados entre sí y que pasamos a describir a continuación.

- **Packet_Receiver**: tiene asignadas dos tareas. La primera es extraer de la cabecera la **dirección IP** y el **tamaño** de cada paquete que llega y la segunda es transmitir al módulo *packet_cutter* los paquetes que recibe del exterior.

- **CAM** es un componente desarrollado por Xilinx del cual hemos incluido una instancia en nuestro proyecto. Nos permite obtener en un ciclo una posición de memoria en función de la dirección IP del cliente. Con la dirección de memoria resultante, el sistema puede trabajar con la información existente asociada al cliente emisor del paquete. Su señal MATCH, nos indica si se ha encontrado posición para el dato de entrada que ha recibido [17].

- **Data Assembler**: módulo cuya única tarea es unir los datos MATCH, la dirección del cliente y el tamaño del paquete que el cliente quiere transmitir en una sola conexión AXI Stream. De esta manera, nos aseguramos la integridad de los datos y podemos utilizar una única cola FIFO para almacenarlos.

- **Token Bucket**: aquí se encuentra implementada toda la lógica del algoritmo *token bucket*. Se encarga de dos tareas fundamentales. La primera es mantener un control correcto del ancho de banda que consumen los clientes y la segunda es decidir si los paquetes que se reciben se pueden transmitir o no. Como este módulo presenta una arquitectura interna más compleja, hablaremos de ella con detalle más adelante.

- **Packet Cutter**: tiene la tarea de transmitir los paquetes admitidos por la interfaz de salida. Para saber que paquetes debe transmitir se basa en los bits activos de la señal *decision*. El autor de este componente es Mario Ruiz, el cual habla con más detalle del módulo en [27, Sec 4.3].

3.4.1. Arquitectura Token Bucket

Como hemos adelantado anteriormente, este módulo se compone de un sistema que consume fichas (el sistema **decisor**), un sistema que añade fichas a los cubos (el sistema **proveedor**) y tres memorias BRAM. En primer lugar, vamos a describir el contenido y las características de las memorias BRAM, de modo que sepamos qué cometido tienen a la hora de interactuar con el resto de componentes:

- **Token Bucket**: En esta memoria, se almacena el número de fichas que tiene disponible cada cliente en función del ancho de banda que se le suministra, el consumo que haga del mismo y del tamaño máximo de ráfaga (MBS) que tenga asignado. Es por tanto un registro que se actualiza con cierta frecuencia.

- **Maximum Burst Size:** se utiliza para almacenar el tamaño máximo de ráfaga que tiene disponible cada cliente. Los registros de este componente son de sólo lectura. Aunque este parámetro es configurable, este valor no debe ser inferior al de la tasa de llenado del cubo.

- **Token Rate**: memoria de sólo lectura utilizada para almacenar el número de fichas que se le ha de sumar a cada cliente. Su valor depende del ancho de banda de cada cliente y

del periodo de actualización que se haya establecido. Por tanto, el número de *bytes* que se le ha de añadir a un cliente en cada actualización vendrá dada por la siguiente expresión:

$$R_c = \lceil \frac{TT_u B_c}{8} \rceil \tag{3.1}$$

Donde B_c es el ancho de banda del cliente en Gbps, T es el periodo del ciclo de reloj del sistema expresado en nanosegundos y T_u es el número de ciclos en los que el sistema proveedor ha de estar parado.

Una vez visto el contenido y la utilidad de las memorias, vamos a pasar a describir el resto de componentes. En primer lugar, el sistema decisor está compuesto por los siguientes módulos:

- **HCU**: es el componente encargado de evitar el riesgo WAW. Este riesgo se produce si en el periodo transcurrido entre la lectura del número de fichas y la actualización del mismo por parte del sistema decisor, el sistema proveedor incrementa dicho número. Como a la hora de realizar el cálculo de las fichas consumidas, no se ha tenido en cuenta el incremento que se ha producido, éste se pierde. Su otra tarea, es gestionar el procesado del paquete mediante la ejecución de la máquina de estados representada en la figura 3.2.

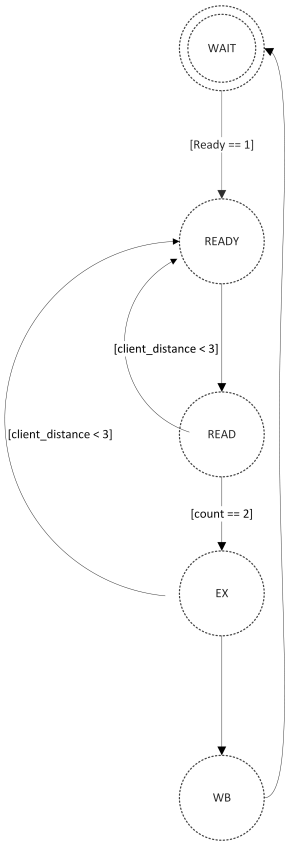

Figura 3.2: Máquina de estados del sistema decisor

- **Decision Unit**: tiene la tarea de sustraer tantas fichas del cubo como bytes de tamaño tenga el paquete. En caso de que haya suficientes fichas y el número de fichas resultantes

sea mayor o igual que cero, el bit de decisión se activará. Si el resultado es menor a cero, es decir, el número de bytes que pesa el paquete excede al número de fichas que hay en el cubo, el bit de decisión no se activará.

Los dos componentes descritos anteriormente se encuentran conectados a la memoria BRAM *token bucket*. El total de fichas de un cliente se puede actualizar siempre y cuando:

- El total de *bytes* que ocupa el paquete sea más pequeño que el total de fichas existente. En ese caso, la señal DECISION se encontrará activa
- La dirección origen del paquete sea válida. Siempre que esto sea así, el registro MATCH se encontrará activo.
- La unidad HCU no haya detectado un riesgo WAW que implique tener que repetir el cálculo. El registro VALID se encontrará activo en el ciclo de escritura.

Por otro lado, tenemos el sistema proveedor. Está formado por tres componentes y tres memorias. A continuación describiremos estos tres componentes y su interacción con las memorias

- **Client Counter**: este módulo funciona de forma similar al clásico contador de programa (PC) de un microprocesador. De este módulo se obtiene el cliente que debe ser actualizado del mismo modo que del PC se obtiene la instrucción que ha de ser ejecutada.
- **Trigger Write**: este contador, se encarga de que el registro del cliente se actualice en el ciclo en el que el nuevo dato se encuentra preparado y además esta señal le indica también al CC que obtenga el siguiente cliente que ha de ser actualizado.
- **Bucket Adder**: las entradas de este componente, se corresponden con las salidas de las tres memorias con las que el sistema trabaja y que contienen el MBS, la tasa de fichas denominada *token_rate* y el número de fichas existentes en el cubo *token bucket*. Con estos datos registrados, la única tarea del componente es realizar el siguiente cálculo:

$$N_c = max(mbs_c, N_c + tr_c) \tag{3.2}$$

  donde mbs_c representa el tamaño máximo de ráfaga, N_c el número de *bytes* del cliente y tr_c el número de fichas que se le ha de sumar al cliente por periodo de actualización.

Cabe destacar que el proceso de actualización está segmentado en dos etapas. La primera es la obtención del cliente y dura un ciclo. La segunda etapa dura cuatro ciclos y se realiza la lectura de los datos, el cálculo y la actualización de los mismos. Esto significa, que el primer cliente tardará en ser actualizado 6 ciclos (5 ciclos de instrucción más un ciclo de activación del proceso) y el siguiente cliente quedará actualizado dentro de cuatro ciclos. Si queremos obtener el número de ciclos requeridos para actualizar los clientes, podemos utilizar la siguiente fórmula

$$N = 6 + 4(C - 1) \tag{3.3}$$

Donde C es el número de clientes a los que el sistema ha de proporcionar servicio. Dada esta ecuación podemos concluir que el tiempo de actualización es linealmente dependiente del número de clientes.

3.4.2. Desarrollo del prototipo funcional

Una vez analizada la arquitectura del sistema, lo siguiente será someterlo a determinados estímulos mediante el desarrollo de un banco de pruebas o *testbench*.

La herramienta Vivado, somete a los puertos de entrada del diseño a probar a los estímulos definidos en el banco de pruebas. Nuestro objetivo en esta etapa es comprobar de forma ágil que nuestro sistema es capaz de realizar las tareas para las que ha sido construido.

Antes de crear el banco de pruebas, debemos en primer lugar definir las condiciones de del experimento de manera que nos permitan de una forma sencilla, probar las principales funcionalidades del sistema. Dichas condiciones son:

- El enlace constará de 4 clientes legítimos y 4 direcciones IP no reconocidas.
- La velocidad a la que el banco de pruebas transmitirá los paquetes, será de 1 Gbps.
- El primer cliente, tendrá una velocidad asignada de 200 Mbps, el segundo y el tercero tendrán 50 Mbps y el cuarto un enlace más reducido de 2 Mbps.
- Al primer cliente le corresponderá la IP de clase 3 0.0.1, al segundo la 0.0.2, al tercero la 0.0.3, al cuarto la 0.0.4
- El tamaño máximo de ráfaga (MBS) de cada cliente será calculado utilizando el método "5 ms", del cual hemos ya hablado en la sección 2.5.1 del Estado del Arte.

Establecidas las restricciones, vamos a calcular los parámetros requeridos para que el sistema pueda dar servicio. El primer dato del que hay que disponer para realizar los cálculos es el de la frecuencia o periodo de la FPGA con la que vamos a trabajar. Como nuestro sistema ha sido diseñado para trabajar sobre una NetFPGA SUME, debemos establecer un periodo 6,4 ns (156,25 Mhz).

Definido ya el periodo de reloj con el que vamos a trabajar, lo siguiente que debemos de hallar es el valor T_u (número de ciclos en los que el sistema de llenado se encuentra parado) para poder resolver la ecuación 3.1. Para ello, lo primero que haremos será convertir la conexión con menor ancho de banda a bytes por ciclo (utilizando un periodo de 6.4 ns) para así comprobar si es menor a la unidad. Para simplificar los cálculos lo pasamos previamente a Gbps.

$$T_u = \frac{6,4 \cdot 2 \cdot 10^{-3}}{8} = \frac{1}{625} B/Ciclo \tag{3.4}$$

Como podemos ver, tenemos una tasa inferior a uno. Como nuestro sistema de llenado sólo soporta números enteros, para cumplir con la tasa establecida deberíamos añadir un *byte* cada 625 ciclos, que se va a convertir en el periodo de actualización de nuestro sistema.

Una vez hemos obtenido el periodo de actualización del sistema de llenado, el siguiente paso es definir una posición de memoria para cada uno de los clientes. Teniendo en mente que hemos calculado T_u y suponiendo que el cliente con menos ancho de banda disponible se ubica en el primer registro, podemos asignar una posición arbitraria al resto de clientes, de manera que:

- El primer cliente, se ubicará en la posición 3
- El segundo cliente, se ubicará en la posición 2
- El tercer cliente, se ubicará en la posición 1
- El cuarto cliente se ubicará en la posición 0

Con todos los datos obtenidos, ahora podemos resolver 3.1 para cada uno de los clientes. A continuación mostramos el número de *bytes* que el sistema deberá de adicionar al cliente 1:

$$RC1 = \lceil \frac{625 \cdot 6,4 \cdot (200 \cdot 10^{-3})}{8} \rceil = 100B \tag{3.5}$$

Resolviendo las expresiones resultantes, nos quedan las siguientes cifras:

- El primer cliente, 100 *bytes*.
- El segundo cliente, 25 *bytes*.
- El tercer cliente, 25 *bytes*.
- El cuarto cliente, 1 *byte*.

Por último, debemos calcular el tamaño máximo de ráfaga. Como vamos a aplicar el método "5ms", se obtiene a partir del periodo de transmisión máxima que nosotros definamos (concepto conocido como *burst period*). Dicho periodo y el tamaño máximo de ráfaga se relacionan de la siguiente manera:

$$B_s = \frac{10^6 \cdot B_c.B_p}{8} \tag{3.6}$$

Donde B_s es el tamaño máximo de ráfaga en *bytes*, B_c es el ancho de banda del cliente en Mbps y B_p es el periodo de transmisión máxima en segundos.

Tras aplicar la fórmula utilizando $B_p = 0{,}005s$, los tamaños de ráfaga obtenidos para cada cliente son:

- Para el primer cliente, 125000 *bytes*.

- Para el segundo y tercer cliente, el tamaño máximo de ráfaga será de 31250 *bytes*.
- El tamaño máximo de ráfaga correspondiente al cuarto cliente será de 1250 *bytes*.

Interpretando los datos, nos damos cuenta que el tamaño máximo del cuarto cliente, se encuentra por debajo de la MTU de nuestro enlace Ethernet. Para este experimento, vamos a seguir las recomendaciones establecidas en la sección 2.5.1 y vamos a establecer un tamaño ráfaga suelo de 15000 *bytes*.

Estos tamaños de ráfaga son bajos en comparación a la tasa media de cada cliente y más teniendo en cuenta que los clientes retransmiten a través de un enlace Gigabit Ethernet. Además, como el volumen de tráfico que se va a transmitir en esta prueba no es excesivamente alto (puesto que lo primordial es validar que la lógica del algoritmo es correcta), son tamaños más que suficientes.

Llegados a este punto, tenemos todos los parámetros configurables definidos. El paso siguiente es configurar las memorias del sistema.

Para realizar dicha configuración, hay que elegir en primer lugar el tamaño de palabra para cada una de las memorias y en segundo lugar hay que definir el valor de cada uno de los registros mediante la utilización de ficheros. Como tenemos 4 memorias en el sistema (La memoria CAM y las tres memorias BRAM), debemos rellenar los siguientes ficheros:

Init.mif: Este fichero se utiliza para introducir cada contenido que queramos direccionar en una línea diferente. Este contenido deberemos introducirlo en código binario, quedando en nuestro caso 4 registros de 24 bits puesto que trabajamos con IPs de clase 3.

```
000000000000000000000100
000000000000000000000011
000000000000000000000010
000000000000000000000001
```

Figura 3.3: Salida del comando cat para el fichero init.mif

Mbs.coe: se utiliza para configurar la memoria que contiene el tamaño máximo de ráfaga de cada uno de los clientes. A diferencia del fichero init.mif, este se rellena en código hexadecimal. El tamaño de palabra utilizado es de 17 bits el cual lo hemos obtenido realizando el techo del logaritmo base 2 del registro con valor más alto ($\lceil log_2(125000) \rceil = 17$)

```
03A98
07A12
07A12
1E848
```

Figura 3.4: Salida del comando cat para el fichero mbs.coe

Las memorias ***token rate*** y ***token bucket*** se configuran de la misma manera que la memoria MBS. Esto es, utilizando los ficheros token_rate.coe y token_bucket.coe. Dichas memorias

quedarían configuradas de la siguiente manera:

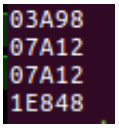

Figura 3.5: Salida del comando cat para el fichero token bucket.coe

Los valores de la memoria *token_bucket* están acotados por los valores de los registros homólogos de la memoria MBS, por lo que el tamaño de palabra también será de 17 bits.

Para la memoria *token_rate*, tenemos una palabra resultante de 7 bits ($\lceil log_2(97) \rceil = 7$)

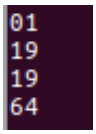

Figura 3.6: Salida del comando cat para el fichero token rate.coe

Con las memorias ya configuradas, solo nos queda definir correctamente la instancia verilog del componente 3.1 en nuestro banco de pruebas y ya estaríamos listos para ejecutar la simulación.

3.5. Validación de los resultados

El proceso de simulación cuyos resultados vamos a describir a continuación, pretendemos comprobar funcionalidades de comportamiento básicas. Estas funcionalidades se describen en los siguientes apartados. Dichas descripciones contienen tanto el motivo y la importancia de la prueba, como la respuesta del sistema al mismo y si ésta es esperada o no.

3.5.1. Lectura de un paquete

La comprobación de la lectura de un paquete es la más básica de todas. Consiste en extraer correctamente de la cabecera IP el campo *total length* y los tres primeros octetos de la dirección IP de origen.

Para comprobar esta tarea, vamos a utilizar una traza pcap de un único paquete creado de forma sintética y vamos a transmitir dicho paquete al sistema. Abriremos dicho paquete con el programa *Wireshark*, el cual nos muestra el contenido del paquete de manera que sea legible de una forma más cómoda. Los valores que visualicemos con la herramienta y los que capture el sistema deben de ser idénticos.

La traza que vamos a utilizar cumple con las características descritas anteriormente. Contiene un único paquete UDP, el cual podemos ver en la siguiente figura.

Dicho paquete contiene la IP 150.244.57.163. También podemos comprobar que el tamaño indicado son 72 *bytes*.

```
▼ Internet Protocol Version 4, Src: 150.244.57.163, Dst: 150.244.255.255
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 72
```

Figura 3.7: Cabecera IP de un paquete UDP

Para superar esta prueba el sistema debe capturar en un registro de 24 bits los octetos 150, 244 y 57 y en un registro de 16 bits el valor 72.

Para poder realizar la verificación de forma más sencilla , leeremos cada registro en formato hexadecimal. Los octetos 150, 244 y 57, se traducen a 96, F4 y 39 en formato hexadecimal. El registro 72 equivale al valor 48.

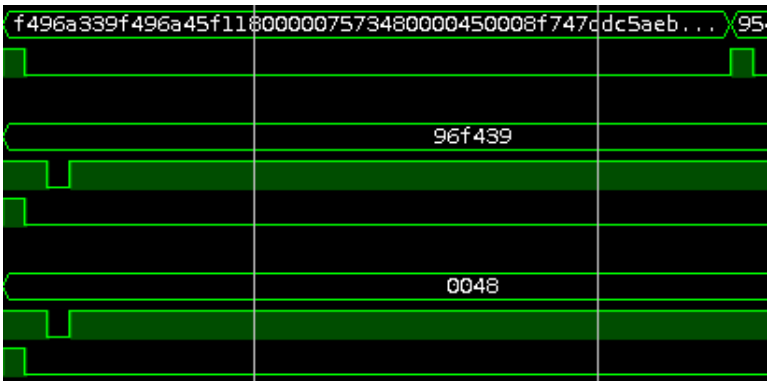

Figura 3.8: Simulación de lectura de un paquete

Las tres señales activas conforman las señales *valid* del protocolo AXI stream. Eso significa que el sistema ha leído la primera transacción del paquete en el ciclo anterior. Como podemos ver, los registros se actualizan con los valores esperados.

La transacciones del paquete se actualizan cada 32 ciclos (como también se puede apreciar en la figura 3.8), puesto que estamos trabajando con un enlace de 1 Gbps.

Por otro lado, las transacciones tanto de la IP como las del tamaño se actualizan cada vez que se lee la primera transacción de un nuevo paquete.

3.5.2. Identificación de la dirección IP

Comprobada la lectura correcta de la IP, ahora debemos verificar si el sistema es capaz de distinguir las direcciones IP de clase 3 que utiliza cada cliente para transmitir de las que no lo son.

La dirección IP que hemos capturado en la prueba anterior no pertenece a ninguno de los clientes definidos en el apartado 3.4.2, por lo que la señal MATCH de la memoria CAM debería de permanecer inactiva indicando que no ha habido coincidencia.

Figura 3.9: Dirección no reconocida

A continuación vamos a comprobar la respuesta del sistema ante una dirección legítima. Siempre que se mantenga en el puerto de entrada una IP legítima, el módulo CAM debería de mostrar a la salida la señal MATCH activa y la identificación correcta del cliente.

Para realizar la verificación, vamos a transmitir un paquete que contenga la dirección de origen 0.0.1.0. Atendiendo a la figura 3.3, nos debería de devolver el identificador número 3.

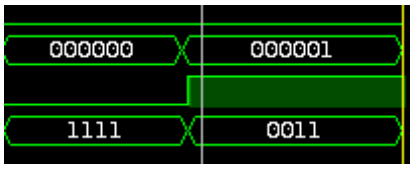

Figura 3.10: Dirección reconocida

Podemos apreciar la activación de la señal MATCH al recibir una dirección legítima. La identificación esperada coincide en este caso con la recibida.

Procesamiento de un paquete reconocido

Para que el procesamiento de un paquete pueda considerarse correcto, el sistema debe ser capaz de reconocer cuando un paquete puede ser transmitido y de ser así, actualizar correctamente el número de fichas resultantes del cliente propietario del paquete.

Utilizando el mismo paquete que en la prueba anterior, tenemos que pertenece al cliente número 3 y tiene un tamaño de 52 *bytes*.

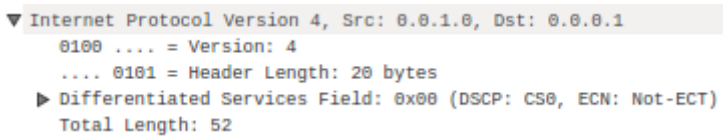

Figura 3.11: Paquete legítimo

Al ser el primer paquete de la traza y teniendo en cuenta que en esta prueba los cubos se encuentran inicialmente llenos, se espera que el sistema permita que el paquete sea enviado.

Inicialmente, el cliente tiene disponibles para consumir 15000 *bytes* atendiendo a los cifras definidas en la figura 3.5. En la siguiente figura, podemos ver la respuesta del sistema al procesar el paquete antes mencionado.

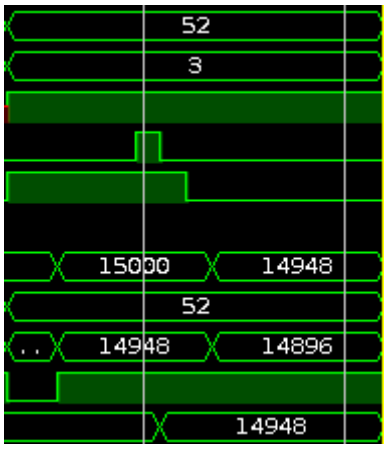

Figura 3.12: Procesamiento del paquete

Las tres primeras señales, se corresponden con los datos de entrada. Esto son el tamaño el paquete, el identificador del cliente y la señal MATCH que nos indica si el identificador es válido o no. Podemos ver que la información que se recibe coincide con los datos esperados.

Las dos siguientes señales constituyen señales de control. Estas señales son VALID, que se corresponde con la salida del módulo HCU que indica que si la decisión es válida, y la señal de habilitación del puerto de la memoria, el cual se inhabilita cuando el paquete ha sido procesado.

Las cuatro señales siguientes son los puertos de la unidad de decisión. La primera señal, se corresponde con el número de fichas que actualmente tiene el cubo, la segunda señal se corresponde con el tamaño del paquete que está siendo procesado, la tercera con el número de fichas restantes tras descontar el número de bytes del paquete y la última se corresponde con la decisión acerca del destino del paquete. Si ésta se encuentra activa, el paquete puede ser transmitido.

La última señal se corresponde con el registro asociado al cliente en la memoria *token_bucket*. Como podemos apreciar en la imagen, este registro se actualiza y tras el procesamiento el cubo alberga 52 fichas menos. Dicha actualización se realiza cuando las señales de control se encuentran activas y la unidad de decisión indica que el paquete se puede transmitir.

Para comprobar que el paquete ha sido transmitido de forma efectiva tenemos que observar las señales del módulo *Packet Cutter* (del cual hablamos en la sección 3.4). Si la señal de decisión se activa, el módulo transmitirá.

En la figura 3.13, se puede apreciar como el módulo recibe una señal positiva de 32 bits. La última señal, se ha activado varias veces. Esto nos indica que el paquete se ha divido en múltiples transacciones. La penúltima señal indica que la transacción efectuada es la última, lo que significa que el paquete ha sido transmitido.

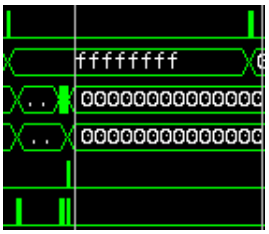

Figura 3.13: Transmisión del paquete

3.5.3. Procesamiento de un paquete no reconocido

En la siguiente prueba, queremos comprobar el funcionamiento del sistema ante la llegada de un paquete cuyo identificador de cliente es desconocido. La respuesta esperada sería no permitir que el paquete sea transmitido.

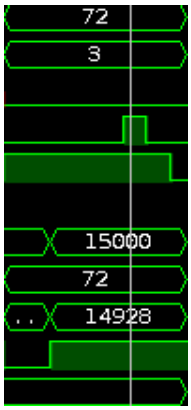

Figura 3.14: Procesamiento de un paquete no reconocido

Para realizar dicha comprobación, nos vamos a servir del paquete UDP presentado en la figura 3.8, cuya IP no se encuentra registrada en la CAM.

En este caso y a diferencia del anterior, la señal MATCH no se encuentra activa ya que la IP origen del paquete no se corresponde con la de ningún cliente. Aunque el identificador que aparezca sea el 3 y se hagan los cálculos con los datos del cliente 3, su registro no se actualiza, por lo que el paquete ha sido descartado. Si nos queremos cerciorar de que el paquete ha sido descartado, debemos comprobar nuevamente las señales del módulo *Packet Cutter*

En la figura 3.15, se puede apreciar como cuando el modulo *Packet Cutter* recibe una decisión negativa, las salidas no sufren cambios, lo que confirma que el paquete no ha sido transmitido

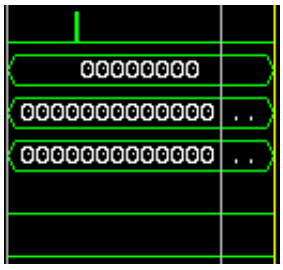

Figura 3.15: Descarte del paquete

3.5.4. Restricción del ancho de banda

Otra situación en la que el sistema debe descartar un paquete, es cuando un cliente ha transmitido un tiempo determinado por encima del ancho de banda que tiene establecido y llega un paquete cuyo tamaño, es superior al número de fichas que alberga el cubo de dicho cliente.

Para la realización de esta prueba, hemos saturado el enlace de paquetes propiedad de un mismo cliente para provocar que el sistema tenga que empezar a descartarlos. El resultado obtenido se puede apreciar es la siguiente figura:

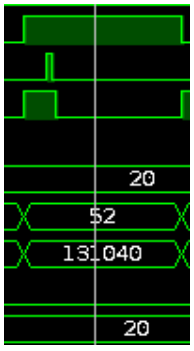

Figura 3.16: La unidad de decisión detecta que no hay fichas suficientes

El paquete en este caso es propiedad del cliente 3 y tiene un tamaño de 52 *bytes*. Como en el cubo del cliente número 3 únicamente quedan 20 *bytes*, el puerto de decisión no se activa (a diferencia de los casos 3.14 y 3.12).

La respuesta obtenida del sistema ha sido la esperada. La unidad de decisión ha sido el módulo que ha impedido en este caso la transmisión del paquete.

3.5.5. Reposición de las fichas de los clientes

El objetivo de esta prueba, va a ser analizar el funcionamiento del proceso de adición de fichas a cada uno de los clientes. El correcto funcionamiento de este proceso es fundamental, puesto que depende del mismo garantizar a los clientes que se les proporciona el ancho de banda que han requerido.

Basándonos en los datos obtenidos en la sección 3.4.2, tenemos que para gestionar la conexión

más lenta, debemos otorgarle a dicha conexión un *byte* cada 625 ciclos, por lo que debemos comprobar que se realiza dicha actualización en tal periodo.

También hay que comprobar que el sistema adiciona los *bytes* que tiene asociados cada cliente en la memoria *Token Rate*.

Para realizar esta prueba, vamos a poner en marcha el sistema con los cubos de los clientes inicialmente a medio llenar para comprobar de forma más fiable, que el sistema modifica los registros teniendo en cuenta su estado anterior, ya que lo que buscamos es incrementar el valor actual de dicho registro.

Establecidas las condiciones de la prueba, vamos a analizar los resultados de la misma que podemos ver en la siguiente figura:

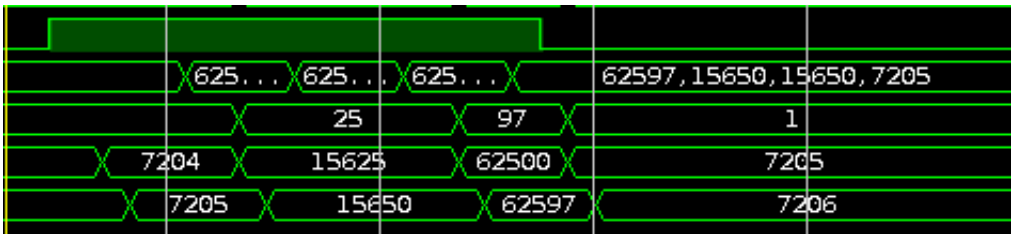

Figura 3.17: Proceso de incremento de *bytes*

La primera señal denominada BUSY, es una señal de control mediante la cual el módulo indica que está llevando a cabo actualizaciones.

En la segunda fila, tenemos el contenido de la memoria *Token Bucket* la cual vemos que se actualiza en 4 ocasiones ya que tenemos en este experimento 4 clientes.

En la tercera fila, se encuentra el número de *bytes* que el sistema ha de sumar al cubo del cliente, es decir la tasa de llenado.

En la cuarta fila tenemos el número de fichas actuales y en la última tenemos el número de fichas actualizado, el cual obtenemos añadiendo al número de fichas actuales la tasa de llenado.

Teniendo claras el significado de cada una de las señales, se puede apreciar como tanto el incremento como la escritura en la memoria se realizan según lo esperado a la hora de definir el experimento.

Si nos fijamos en los tiempos de la figure 3.18, podemos ver que el tiempo transcurrido entre dos actualizaciones es de 4000 ns, lo que dividido por los 6,4 ns de periodo se queda en 625 ciclos. Esta cifra coincide con el número de ciclos establecidos a partir de la expresión 3.4

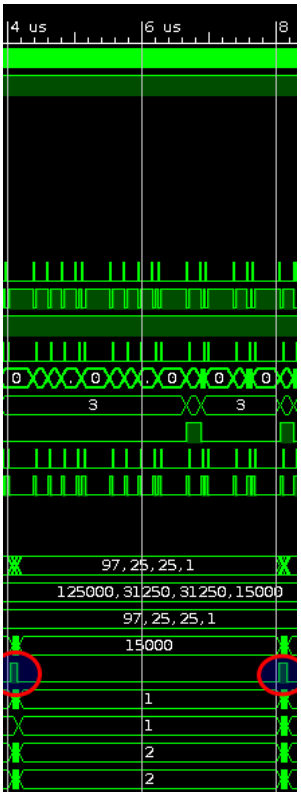

Figura 3.18: Periodo entre actualizaciones

Por otro lado, los tiempos del proceso de lectura de un paquete y del proceso de actualización de los clientes. El proceso de lectura normal de un paquete dura 13 ciclos (equivalente a 83,2 ns) se admita el paquete o no . Por otro lado, el proceso de actualización se sucede cada 625 ciclos y para 4 clientes, dura siempre 18 ciclos, lo que equivaldría a 115,2 ns.

Los bloques de entrada / salida, se encuentran reservados para las interfaces de red, por lo que no están siendo utilizados directamente por nuestro sistema.

4

Puesta en marcha del sistema

Tras realizar una simulación del comportamiento del sistema, el siguiente paso ha de ser cargar dicho sistema en la FPGA. Para realizar este proceso, hay que realizar tres etapas: Síntesis, implantación y programación del dispositivo.

Nuestro sistema por si mismo no es capaz de comunicarse con una interfaz de red física. Por esta razón, antes de poder cargar el diseño en la FPGA, debemos integrar nuestro módulo dentro de un diseño que sea capaz de comunicarse con una interfaz de red. Este proyecto es el **Reference NIC**, el cual hemos mencionado ya en alguna ocasión.

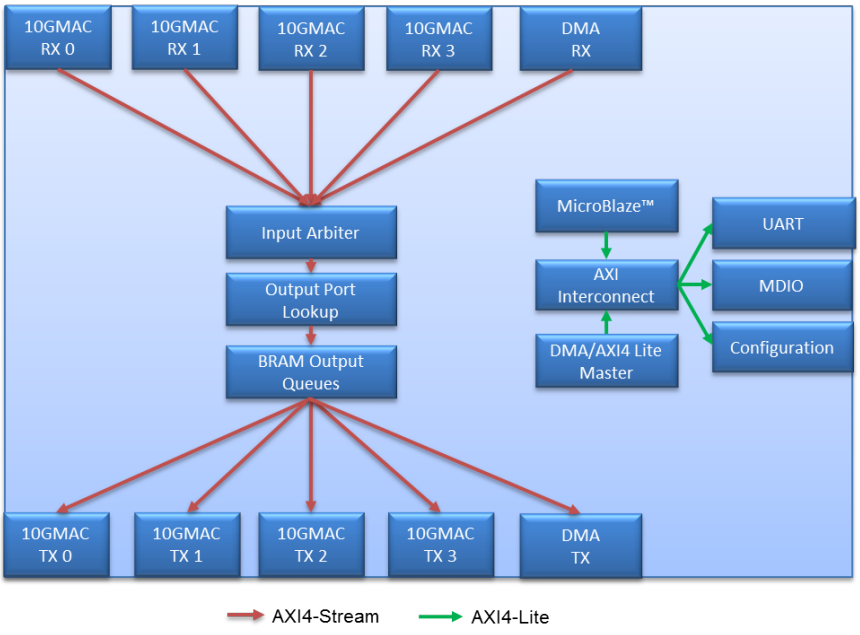

Figura 4.1: Diagrama de bloques extraído de [20]

El proyecto Reference NIC nos permite hacer uso de las cuatro interfaces Gigabit que dispone

la NetFPGA-SUME. El sistema se compone de un conjunto de módulos cohesionados. Cada interfaz de red está por un puerto de recepción y otro puerto de transmisión de paquetes. En total tenemos cuatro interfaces de red que soportan hasta 10 Gbps más la interfaz DMA que se utiliza para transmitir información a la FPGA si conectamos ésta a otro equipo.

En el diagrama 4.1, ubicados entre los puertos de entrada y salida de las interfaces, tenemos tres módulos:

- El módulo *Input Arbiter* tiene como principal cometido unificar la información de los cuatro canales en un único canal [8].
- El módulo *Output Port Lookup* es el encargado de asignar interfaz de destino a los paquetes que recibe [6].
- El módulo *BRAM Output Queues* se ocupa de transmitir los paquetes a una interfaz u otra en función de la decisión del módulo *Output Port Lookup* [7].

Para este primer despliegue, vamos a conectar el módulo a una única interfaz de red. Dicha interfaz va a estar conectada directamente a nuestro sistema, por lo que no vamos a hacer uso de los módulos descritos anteriormente. Conectaremos la entrada AXI de nuestro *policer* al módulo **10GMAC RX 0** y la salida al módulo **10GMAC TX 0**.

Para llevar a cabo este proceso, debemos realizar varios pasos que vamos a detallar a continuación. Antes aclarar que para poder poner seguir estos pasos, se ha de tener instalada y configurada la herramienta Vivado en su versión 2014.4 en un sistema operativo GNU / Linux compatible.

4.1. Despliegue del proyecto en Vivado

Lo primero que hay que hacer es abrir una terminal y descargar el repositorio de NetFPGA SUME.

```
git clone https://github.com/NetFPGA/NetFPGA-SUME-live.git
```

Como este repositorio aloja varios proyectos, debemos cargar en una variable de entorno el nombre del proyecto que queremos desplegar, que en este caso es el Reference NIC.

```
export NF_PROJECT_NAME=reference_nic
```

Lo siguiente será cargar ficheros tanto de vivado como del proyecto que contienen variables de entorno básicas.

```
source NetFPGA-SUME-live/tools/settings.sh source /opt/Xilinx/Vivado/2014.4/settings64.sh
```

```
[sanzceb@devpc:~]$ cd $SUME_FOLDER
```

Comprobamos que las variables de entorno están correctamente cargadas moviéndonos a la raíz del repositorio

Si nos hemos desplazado al repositorio correctamente, lanzaremos el programa *make* para sintetizar la colección de módulos de NetFPGA. Esta orden puede tardar varios minutos en ejecutarse dependiendo de la potencia de la máquina.

```
[sanzceb@devpc:~/NetFPGA-SUME-live]$ make
```

Teniendo ya los módulos base, pasamos a generar el proyecto Reference NIC:

```
[sanzceb@devpc:~/NetFPGA-SUME-live]$ cd projects
[sanzceb@devpc:~/NetFPGA-SUME-live/projects]$ cd $NF_PROJECT_NAME
[sanzceb@devpc:~/NetFPGA-SUME-live/projects/reference_nic]$ make
```

Una vez abrimos el proyecto generado con la herramienta, nos encontraríamos un entorno similar a éste:

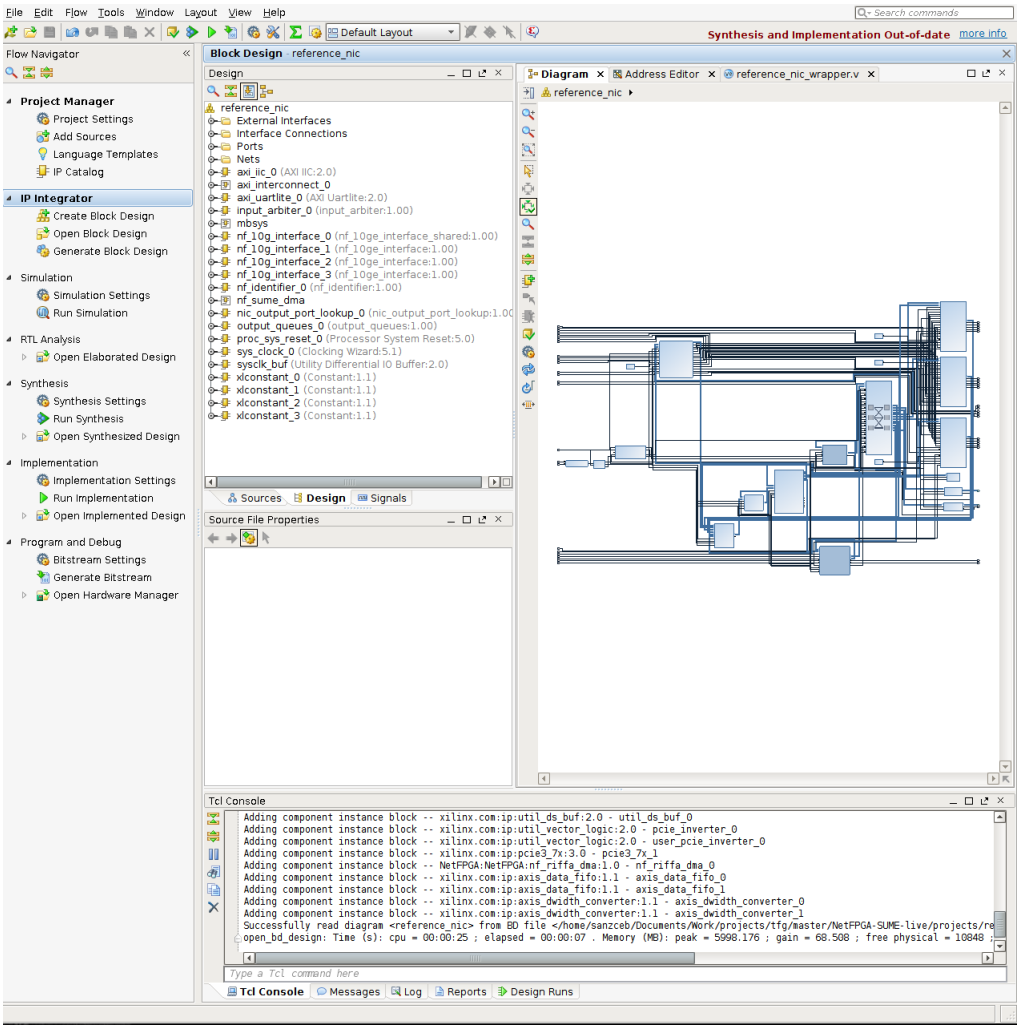

Figura 4.2: Entorno de desarrollo del proyecto

El primer paso que tenemos que dar es importar el algoritmo de *policing* que hemos generado. En este caso lo hemos llamado policer. Una vez importado debemos asignar a los parámetros los mismos valores que en el apartado 3.4.2.

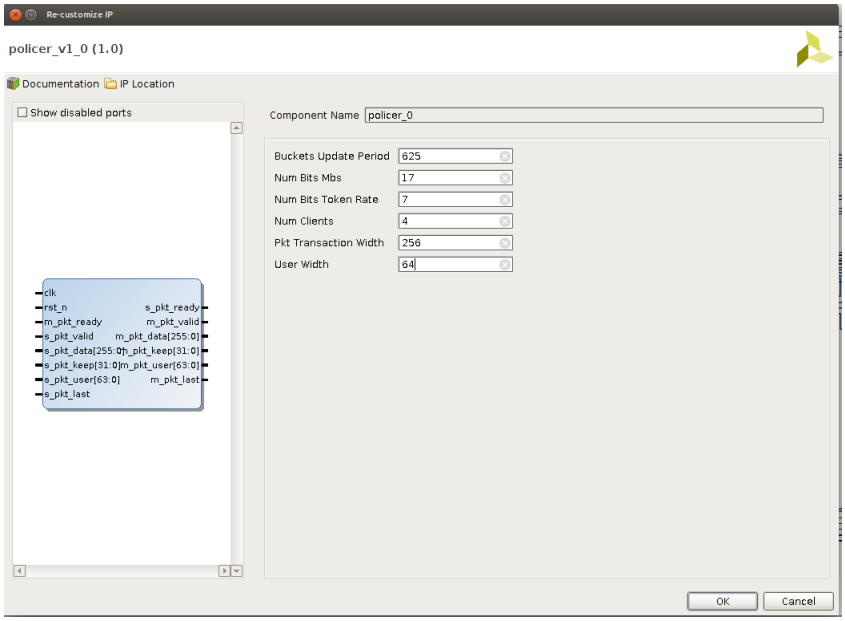

Figura 4.3: Parámetros del sistema

Por último, nuestro módulo a la interfaz 0 del NIC.

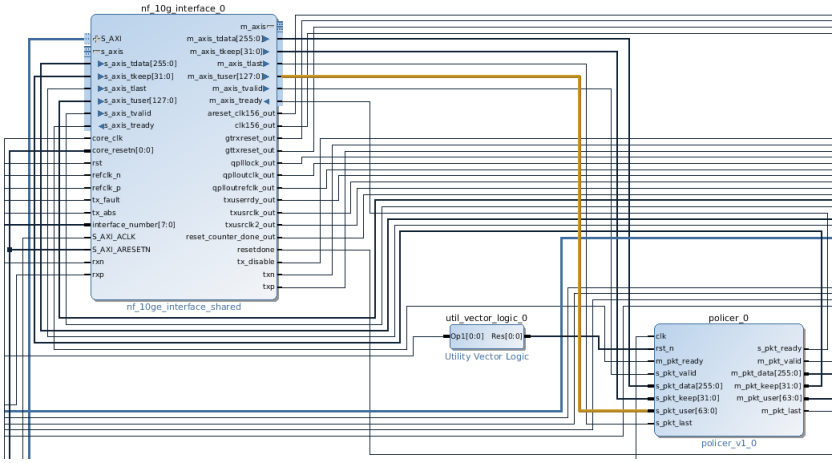

Figura 4.4: Modificación del diseño Reference NIC.

4.2. Síntesis

La siguiente etapa del proceso es la síntesis. Es un proceso de optimización que consiste en realizar la descripción de un circuito o algoritmo utilizando componentes de un nivel más bajo de abstracción. La síntesis que realiza Vivado consiste en describir un circuito a nivel de puertas lógicas a partir del código Verilog que hemos desarrollado. Como este proceso es bastante

complejo, se divide en varias etapas, como podemos apreciar en la siguiente figura:

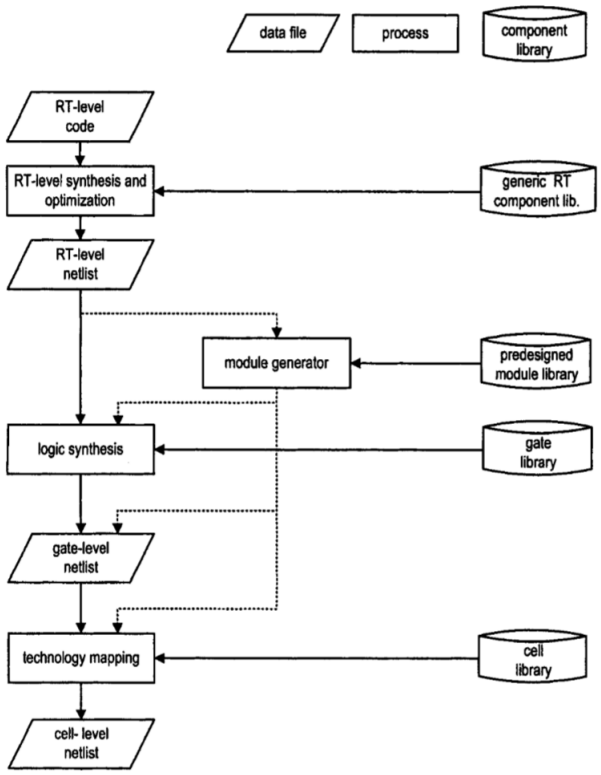

Figura 4.5: Esquema genérico del proceso de síntesis. Extraída de [3, p. 160]

Ya sea de código VHDL o código Verilog, la primera tarea consiste en sintetizar el código descrito en un código RTL intermedio que se limita a describir el sistema de forma lógica. El siguiente paso, es sintetizar el código RTL en un circuito RTL.

El circuito RTL, se sintetiza en un circuito de puertas lógicas con la ayuda de un programa (denominado en la figura como *Module Generator)* y utilizando bibliotecas donde se definen las puertas lógicas disponibles.

Los componentes RTL, pueden ser transformados de forma directa sin necesidad del programa sólo si son estructuras conocidas y de baja complejidad, ya que su descripción vendrá predefinida en una biblioteca. El resto de componentes, han de ser transformados utilizando el generador [3, Sec 6.4].

Si hacemos uso de Vivado, la opción de sintetizar el diseño la podemos encontrar en la barra izquierda, como se muestra en la figura 4.1.

4.3. Implantación

El primer paso que realiza Vivado en este proceso es simplificar el circuito obtenido en la síntesis utilizando su propio conjunto de funciones primitivas denominado XNGD.

El siguiente paso sería encajar de forma lógica este circuito en la FPGA si ésta tiene los recursos

disponibles. Este proceso sería el equivalente al proceso *Technology Mapping* de la figura 4.5. Con el producto final de esta etapa, se ha comprobado que la FPGA tiene capacidad para albergar el diseño.

En último lugar, la herramienta lleva a cabo la tarea PAR donde el objetivo es elegir qué bloques se utilizan y cuales no para conseguir minimizar la frecuencia de reloj necesaria para que el circuito funcione.

Para comenzar el proceso con Vivado, seleccionamos la opción *implementation* que aparece en la barra izquierda de la figura 4.1.

4.4. Programación de la FPGA

Para hacer funcionar el diseño en la FPGA, es necesario generar un fichero de configuración *bitstream*. Este fichero se utiliza para configurar la FPGA a partir del diseño PAR obtenido. Esta acción se puede realizar con la ayuda de Vivado seleccionando la opción *Generate Bitstream* de la figura 4.2.

Antes de escribir el programa en la FPGA, debemos cerciorarnos de que el controlador del dispositivo se encuentra instalado. En caso de que no aparezca la FPGA, hay que cargar el controlador y reiniciar el dispositivo. Esta comprobación la podemos realizar con la siguiente instrucción:

```
[sanzceb@devpc: bitfiles]$ lspci -vxx | grep Xilinx
```

Para la programación, vamos a hacer uso de la herramienta XMD [31]. Ubicándonos en el directorio donde se encuentra el fichero *bitstream*, invocamos a xmd y programamos la fpga indicándole el fichero de configuración:

```
[sanzceb@devpc: bitfiles]$ xmd
[sanzceb@devpc: bitfiles]$ fpga -f reference_nic.bit
```

5

Conclusiones

El proyecto partía de una premisa, diseñar un sistema de gestión de ancho de banda de redes de Internet utilizando dispositivos FPGA, que además, fuese capaz de proporcionar calidad de servicio en cuanto a latencia, puesto que como comentábamos en la introducción se buscaba que el enlace gestionado pueda dar servicio a sistemas HFT.

Para cumplir con esta premisa, se tomó la decisión de programar en *hardware* un algoritmo de conformado de tráfico ya existente como *token bucket* y aplicar una política de *policing*. Tras analizar las distintas alternativas, se concluyó que era la mejor opción teniendo en cuenta la baja latencia requerida.

El lenguaje que hemos utilizado para el desarrollo ha sido Verilog y como entorno de desarrollo hemos utilizado la plataforma de desarrollo de Xilinx Vivado en su versión 2014.4, puesto que es la que se ha utilizado en el proyecto NetFPGA. El proyecto NetFPGA nos proporciona el diseño de un NIC en *hardware* abierto como base para poder utilizar la FPGA como lector y transmisor de paquetes.

Tras revisar todo el trabajo realizado, podemos determinar que se ha conseguido construir un prototipo funcional, que es capaz de ejecutar correctamente la lógica del algoritmo y que en promedio, apenas ocupa un 1 por ciento de los recursos de la FPGA con la que estamos trabajando.

Estamos ante un sistema altamente parametrizable, ya que se puede definir el número de clientes, el tamaño máximo de ráfaga y la tasa de rellenado de cada cliente de forma individual.

5.1. Futuros trabajos

Dada la magnitud del proyecto, aún hay ciertas aspectos sobre las que se puede trabajar en futuros proyectos.

En este prototipo, el identificador de cliente, se encuentra actualmente restringido a direcciones IP de clase C y los datos de los clientes no pueden ser modificados mientras el sistema se encuentra en funcionamiento. Son aspectos que pueden ser revisados en futuros trabajos.

Por otro lado, para poder lograr un sistema que se pueda llegar a comercializar, quedarían más tareas aún por realizar.

Entre esas tareas, estaría la construcción de una plataforma de prueba en la que habría que montar el dispositivo FPGA con el algoritmo programado. En dicha plataforma, se buscaría emular un entorno lo más parecido posible al entorno real de producción, donde el dispositivo se vería expuesto a un volumen de tráfico mucho mayor.

Otro trabajo importante que quedaría sería desplegar un generador de tráfico que transmita un flujo de paquetes de características muy similares al tráfico que generaría un sistema HFT y con una distribución predeterminada que nos permita probar como se comporta el sistema ante los riesgos WAW que ya comentamos en la sección 3.4.

5.2. Experiencia y conocimientos adquiridos

Antes de comenzar el desarrollo del proyecto, había cursado la asignatura optativa de Dispositivos Integrados Electrónicos en cuyas prácticas aprendí conceptos básicos de desarrollo con FPGA y diseño *hardware* en general. Este primer acercamiento y mi interés por el ámbito de las redes, me hicieron tomar la decisión de elegir este proyecto como final de mis estudios de grado.

Una vez comenzado el proyecto me di cuenta que se requería además de conocimientos más amplios sobre FPGA, más experiencia de desarrollo con este tipo de plataformas.

Aunque esto suponía enfrentarme a una curva de aprendizaje pronunciada, mis conocimientos previos sobre redes, hicieron que pudiera invertir tiempo extra en el diseño y desarrollo del sistema que no tenía que invertir en comprender y estudiar el algoritmo con el que íbamos a trabajar, puesto que yo ya lo conocía de antemano.

Acostumbrado a la programación de algoritmo en plataformas *software* de alto nivel, fue duro adaptarme al cambio de paradigma y a nuevas metodologías a la hora de abordar el problema.

Sin embargo, tras finalizar este trabajo, sé que tomé la decisión correcta embarcándome en este proyecto, ya que, de ahora en adelante como futuro ingeniero, puedo tener en cuenta el diseño *hardware* como herramienta para resolver problemas que el enfoque de software convencional no puede. Espero poder tener la oportunidad de trabajar en proyectos similares en un futuro.

Bibliografía

[1] S. Tanenbaum Andrew and J. Wetherall David. *Computer Networks*. Pearson, 5th edition, 2010.

[2] AppNeta. Tcpreplay - pcap editing and replaying utilities. Available at: http://tcpreplay.appneta.com/, May 2017. [Accessed 17 May 2017].

[3] Pong P Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.

[4] Cisco. Fiber Distributed Data Interface (FDDI). Available at: https://www.cisco.com/cpress/cc/td/cpress/fund/ith2nd/it2408.htm. [Accessed 17 May 2017].

[5] Gerald Combs. About wireshark. Available at: https://www.wireshark.org/. [Accessed 17 May 2017].

[6] Adam Covington and James Hongyi Zeng. NIC Output Port Lookup. Available at: https://github.com/NetFPGA/NetFPGA-public/wiki/NIC Output Port Lookup, March 2012. [Accessed 19 May 2017].

[7] Adam Covington and James Hongyi Zeng. BRAM Output Queues. Available at: https://github.com/NetFPGA/NetFPGA-public/wiki/BRAM-Output-Queues-with-registers, February 2014. [Accessed 19 May 2017].

[8] Adam Covington and James Hongyi Zeng. Input arbiter with registers. Available at: https://github.com/NetFPGA/NetFPGA-public/wiki/Input-arbiter-with-registers, February 2014. [Accessed 19 May 2017].

[9] Daniel Cross. The Rise of High-Frequency Trading: A Brief History. Available at http://traderhq.com/trading-education/the-rise-of-high-frequency-trading/, July 2015. [Accessed 18 May 2017].

[10] Business English Dictionary. *Definition of trading*. Pearson Longman, 2007.

[11] Guy Harris. Libpcap file format. Available at: https://wiki.wireshark.org/Development/LibpcapFileFormat. [Accessed 17 May 2017].

[12] IEEE-SA. History of Ethernet. Available at: http://standards.ieee.org/events/ethernet/history.html. [Accessed 17 May 2017].

[13] IEEE-SA. IEEE-SA 802.3 ™ Ethernet Standards Milestones. Available at: http://standards.ieee.org/events/ethernet/timeline.pdf. [Accessed 17 May 2017].

[14] Dror Y. Kenett, Eshel Ben-Jacob, H. Eugene Stanley, and Gitit gur Gershgoren. How High Frequency Trading Affects a Market Index. *Scientific Reports*, 3, July 2013.

[15] Fred Klassen. Man page of tcprewrite. Available at: http://tcpreplay.appneta.com/wiki/tcprewrite-man.html. [Accessed 17 May 2017].

[16] C. Leber, B. Geib, and H. Litz. High Frequency Trading Acceleration Using FPGAs. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 317–322, September 2011.

[17] Kyle Locke. Xilinx® XAPP 1151: Parameterizable Content-Addressable Memory. Available at: https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf, March 2011. [Accessed 18 May 2017].

[18] Bloomberg L.P. Facts & Spaces. *Bloomberg L.P.*, pages Available at: https://www.bloomberg.com/company/bloomberg–facts/. [Accessed 17 May 2017].

[19] Bloomberg L.P. History of Bloomberg L.P. – FundingUniverse. Available at: http://www.fundinguniverse.com/company-histories/bloomberg-l-p-history/. [Accessed 17 May 2017].

[20] Neelakandan Manihatty. NetFPGA-Reference NIC. Available at https://github.com/NetFPGA/NetFPGA-public/wiki/NetFPGA-10G-Reference-NIC, March 2014. [Accessed 18 May 2017].

[21] Microsoft. What Is ATM?: Asynchronous Transfe Mode(ATM). Available at: https://technet.microsoft.com/en-us/library/cc783455(v=ws.10).aspx, March 2003. [Accesed 17 May 2017].

[22] Michael Millar. 'Lightning fast' future traders working in nanoseconds. Available at: http://www.bbc.com/news/business-15722530, November 2011. [Accessed 21 May 2017].

[23] Nasdaq, Inc. Definition of High Frequency Trading (HFT) - NASDAQ Financial Glossary. Online at: http://www.nasdaq.com/investing/glossary/h/high-frequency-trading. [Accessed 17 May 2017].

[24] Nasdaq, Inc. The Nasdaq Story: History, Business, Awards. Available at: http://business.nasdaq.com/discover/nasdaq-story/index.html. [Accessed 17 May 2017].

[25] Netfpga. Netfpga sume public wiki. Online at: https://github.com/NetFPGA/NetFPGA-SUME-public/wiki. [Accessed 17 May 2017].

[26] Juniper Networks. Determining Proper Burst Size for Traffic Policers. Available at: https://www.juniper.net/documentation/en_US/junos/topics/concept/policer-mx-m120-m320-burstsize-determining.html, November 2016. [Accessed 18 May 2017].

[27] M. Ruiz, G. Sutter, S. López-Buedo, and J. E. L. de Vergara. FPGA-based encrypted network traffic identification at 100 Gbit/s. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, November 2016.

[28] Craig Leres Van Jacobson and Steven McCanne. Manpage of tcpdump. Available at: http://www.tcpdump.org/tcpdump_man.html. [Accessed 17 May 2017].

[29] Xilinx®. What is an FPGA? Field Programmable Gate Array. Available at: https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. [Accessed 17 May 2017].

[30] Xilinx®. AXI Reference Guide. Available at: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, March 2013. [Accessed 18 May 2017].

[31] Xilinx®. Embedded System Tools Reference Manual (UG1043). Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug1043-embedded-system-tools.pdf, May 2014. [Accessed 26 May 2017].

[32] Xilinx®. Vivado Design Suite User Guide: Using the Vivado IDE. Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug893-vivado-ide.pdf, April 2017. [Accesed 21 May 2017].

[33] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, September 2014.