UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA TÉCNICA SUPERIOR DE INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

GRUPO DE ARQUITECTURAS PARALELAS

PROYECTO FINAL DE CARRERA

Adaptación a PCI-Express de un diseño de memoria compartida distribuida basado en FPGAS

Autor: Santiago Mislata Valero

Directores: Federico Silla Jiménez
Yana Esteves Krasteva

Septiembre de 2011

PALABRAS CLAVE

- **Memoria Compartida Distribuida**
- **PCI-Express**
- **FPGA**

INDICE GENERAL

LISTA DE FIGURAS

LISTA DE TABLAS

1. Introducción

1.1 Motivación y objetivos

Con la aparición de los sistemas multiprocesadores, necesarios por el avance tecnológico y por la resolución de todo tipo de tareas informáticas y del mundo de la computación de manera más rápida y eficiente, surgieron también nuevos problemas no existentes en sistemas con un único procesador. Uno de los principales problemas es el acceso a los datos con los que se quiere operar. Por ello, actualmente, uno de los principales aspectos a tener en cuenta en los sitemas multiprocesador es la organización de la memoria. En términos generales la memoria puede estar organizada de forma *compartida, distribuida* o *compartida distribuida*. Las distintas ventajas de cada tipo serán objeto de discusión en el siguiente punto del documento. Aquí únicamente cabe destacar que el proyecto se centrará en la solución de **memoria compartida distribuida**.

Un segundo aspecto a tener en cuenta es el estándar de comunicación entre los distintos componentes de un sistema multiprocesador, incluyendo bancos de memoria, y la unidad de procesamiento, que puede contener uno o varios cores. Actualmente, existen numerosos protocolos de comunicación en la industria, entre los cuales cabe destacar HyperTransport, PCI-Express (*Peripheral Component Interconnect Express*) e Infiniband, entre otros.

El Grupo de Arquitecturas Paralelas (GAP) de la Universidad Politécnica de Valencia ha creado un prototipo de un cluster de supercomputación compuesto por 64 nodos Quad Core Opteron a 2 GHz con 16 GB de RAM y 250 GB de disco duro interconectados por fibra óptica, que consiguió en el 2010 una capacidad de cómputo de 11200 Gflops [1].

Una de las principales utilidades del cluster-prototipo dentro del grupo es la de estudio e investigación de nuevas soluciones, tanto en el ámbito de las arquitecturas de computadores, como en el de la comunicación y computación. Con respecto a los recursos de memoria del supercomputador, la solución que se ha seleccionado es la del uso de memoria compartida distribuida no

coherente. La llave para hacer realidad la idea de memoria compartida y distribuida ha sido el uso de hardware programable, FPGAs (*Field Programmable Gate Arrays*), que, además, ha permitido alcanzar latencias muy bajas (menos de 3 μs).

Cada nodo consta de una tarjeta FPGA de Xilinx donde se ha implementado una **Unidad Funcional de Memoria Compartida (SMFU, Shared-Memory Functional Unit)** que permite el acceso a la memoria de cada nodo por parte de cualquier otro nodo de la red. El diseño del hardware específico se ha creado y realizado en la Universidad de Heidelberg (Alemania). En la versión actual, las tarjetas FPGA (propietarias de la Universidad de Heidelberg) se comunican con las placas base y con los Opteron por medio del protocolo HyperTransport. La FPGA lleva un diseño que incluye un core de HyperTransport y la unidad SMFU procesa paquetes que son similares a los de HyperTransport.

Por otro lado, se ha notado una desventaja en el uso de HyperTransport como tecnología de comunicación. HyperTransport restringe la portabilidad de la solución de memoria compartida distribuida propuesta por el GAP a otras tecnologías como las de Intel. Por ello, es de interés para el futuro desarrollo del GAP que existan otras opciones de comunicación que sean más extendidas, como la de PCI-Express. De aquí la motivación y objetivo principal del proyecto: **realizar la adaptación del diseño de la Unidad de Memoria compartida realizado por la Universidad de Heidelberg de la tecnología HyperTransport a PCI-Express**.

Debido al volumen de cambios que, previo estudio, fueron identificados como necesarios, el documento se referirá a rediseño y no adaptación, en lo que se refiere a la unidad de memoria compartida (SMFU).

Más en concreto, los objetivos desglosados del proyecto fin de carrera son:

– Aprendizaje y perfeccionamiento de los entornos de diseño hardware para dispositivos lógicos programables, incluyendo el lenguaje de descripción hardware Verilog. De forma específica, serán objeto de estudio las FPGAs de Xilinx de la familia Virtex6, al igual que las tarjetas de desarrollo de las que dispone el GAP.

– Obtención de una visión general acerca del funcionamiento de los diferentes protocolos de comunicaciones existentes: HyperTransport y PCI-Express.

– Familiarización del proceso de desarrollo de un proyecto hardware, desde el planteamiento y especificación inicial hasta su funcionamiento en hardware real.

– Planteamiento y definición de la especificación de la SMFU-PCIe.

– Diseño, implementación y prueba del módulo SMFU-PCIe.

– Integración del módulo diseñado (SMFU-PCIe) dentro del sistema completo y realización de pruebas en hardware usando las tarjetas de desarrollo Virtex6 de las que dispone el laboratorio.

1.2 Organización del documento

Durante el desarrollo del presente capítulo, se introducirán unos conceptos básicos acerca de la organización de la memoria en un sistema y del protocolo PCI-Express, así como el contexto en el que está situado este proyecto. Además, se presentarán las herramientas hardware y software utilizadas

En el capítulo 2, *Shared-Memory Functional Unit (SMFU)*, se describirá la estructura básica y la funcionalidad completa de la SMFU, explicando detalladamente los módulos de los que consta y todas las tareas que realiza.

El capítulo 3, *SMFU para PCI-Express*, describe detalladamente la implementación de la SMFU, mostrando y describiendo todas las señales que se han utilizado en su diseño, así como las máquinas finitas de estados (FSM, *Finite States Machine*) y estructuras de datos presentes en el mismo.

El capítulo 4, *Validación y Resultados*, recogerá todas las pruebas realizadas a nivel de simulación, así como los vectores de tests utilizados para ello. Además, se mostrarán y comentarán los resultados de síntesis del diseño completo. El producto final obtenido se programará en la FPGA, y será probado con unos drivers con la intención de mostrar su funcionamiento real.

El último capítulo, *Conclusiones y Líneas Futuras*, mostrará las conclusiones obtenidas en la realización de este proyecto, así como las posibilidades del mismo de ser ampliado o mejorado en futuras versiones.

1.3 Organización de la memoria

Como ya se comentó brevemente en el punto 1.1, la forma en que esté organizada la memoria dentro de un computador multiprocesador es un factor importante a considerar. En este apartado se explicarán los 3 modelos de distribución de la memoria conocidos: *compartida, distribuida* y *compartida distribuida*.

- En el modelo de *memoria compartida*, la memoria se ve como un único bloque de almacenamiento de acceso aleatorio. Esto hace que pueda ser accedido directamente por cualquiera de los CPUs que conforman el multiprocesador. La ventaja de este modelo radica en su facilidad para ser programado. Todos los procesadores comparten una visión global de los datos, y esto hace que la comunicación entre ellos sea muy rápida. Sin embargo, este modelo tiene dos principales inconvenientes que no sea el modelo más idóneo para un sistema multiprocesador:

1) El cuello de botella en el que se convierte la conexión entre los CPUs y la memoria, limitando el número máximo de procesadores que podría tener el sistema.

2) Falta de coherencia, es decir, dos núcleos que accedan a una misma dirección podrían obtener diferentes datos.

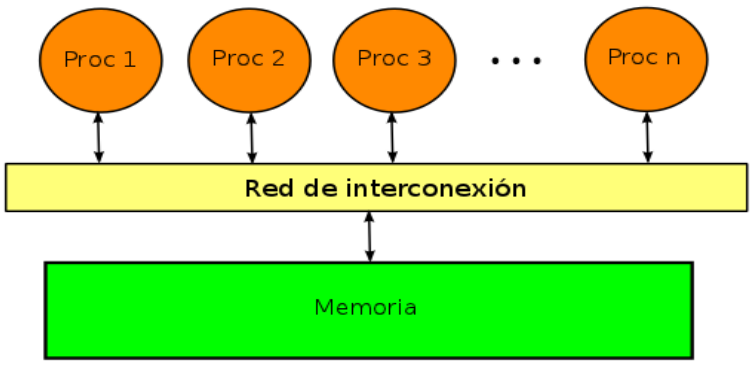

Figura 1.1: Esquema de un modelo de memoria compartida.

- El modelo de *memoria distribuida* es aquel en que cada procesador tiene su propia zona privada de memoria. En este modelo, las ventajas e inconvenientes son las opuestas a las expuestas en el modelo de memoria compartida: el acceso a memoria es rápido y no existe el problema de la coherencia, pero un acceso a una zona de memoria ajena a un procesador (memoria remota) conlleva un tiempo elevado.

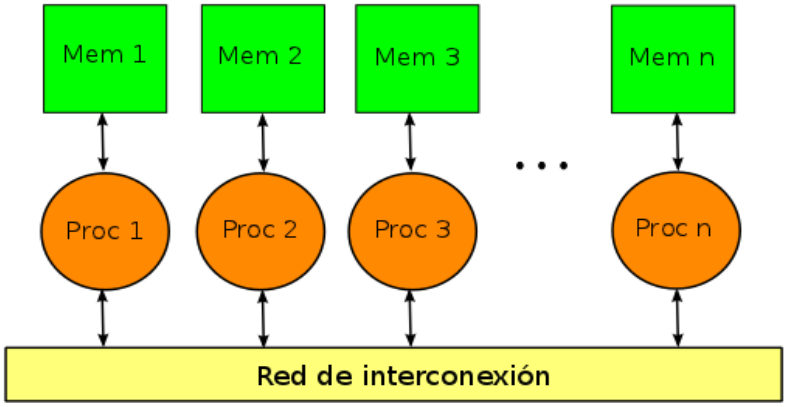

Figura 1.2: Esquema de un modelo de memoria distribuida.

– La *memoria compartida distribuida* (conocido por sus siglas en inglés como DSM, *Distributed Shared Memory*) aúna las virtudes y evita los inconvenientes de los dos modelos anteriores. En este modelo, los espacios de memoria se encuentran *distribuidos* entre los diferentes procesadores (físicamente separados), pero pueden ser accedidos por un espacio de direccionamiento *compartido* (lógicamente unido). Es decir, una misma dirección física en dos procesadores diferentes hace referencia a la misma ubicación en memoria.

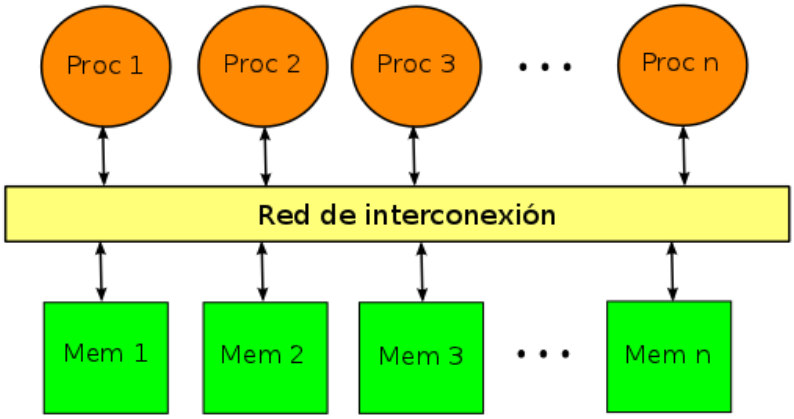

Figura 1.3: Esquema de un modelo de memoria compartida distribuida.

El modelo adoptado actualmente en el prototipo es el de memoria compartida distribuida, organización que se mantendrá para el nuevo diseño.

1.4 PCI-Express

PCI-Express (anteriormente conocido como 3GIO, *3rd Generation In/Out*), también abreviado como PCIe, es, como su nombre indica, una interconexión de entrada/salida de propósito general de alto rendimiento, definida para una amplia variedad de futuras plataformas de computación y comunicación. Es un nuevo desarrollo del bus PCI, que usa los conceptos de programación y estándares de comunicación existente, pero basado en un sistema de comunicación serie más rápido.

Un link PCI-Express representa un canal de comunicaciones full-duplex entre dos componentes. En la siguiente imagen se muestra un ejemplo de link PCI-Express fundamental, que consta de dos pares de bajo voltaje y diferente sentido: uno para Transmisión y otro para Recepción de datos.

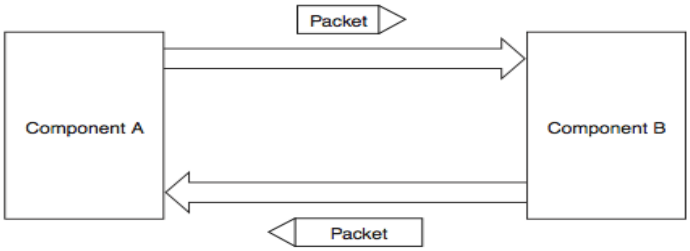

Figura 1.4: Link PCI-Express [3].

La versión de PCI-Express usada para este proyecto es la 2.2, en la que puede proporcionar un ancho de banda de hasta 8GB/s (16 links a 500MB/s) [3].

1.5 Plataforma de desarrollo

La plataforma de desarrollo en la que se ha llevado a cabo el proyecto está basado en el kit de desarrollo PCI-Express Gen 2 de HitechGlobal. Dicho kit consta de una placa de desarrollo, sobre la que se realizarán todas las pruebas de testing, cuyo núcleo es una FPGA Virtex 6 LX240T FF. En las Figuras 1.5 y 1.6 pueden verse el diagrama de bloques y la ubicación de sus componentes, extraídos de [4]. Para más información, ver dicha referencia.

Además, también se ha usado un conjunto de herramientas software, con las cuales se llevan a cabo las fases de diseño y desarrollo del módulo. Entre dichas herramientas, está el ISE Development Suit 13.2 de Xllinx y el ModelSim 6.6, ambos con la licencia EUROPRACTICE.

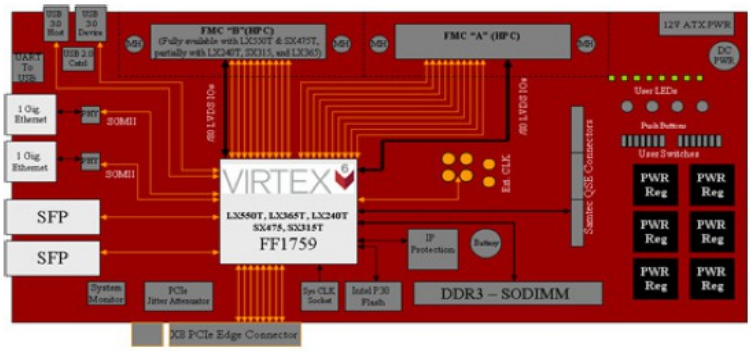

Figura 1.5: Diagrama de bloques de la HTG-V6-PCIe-XXXX [4].

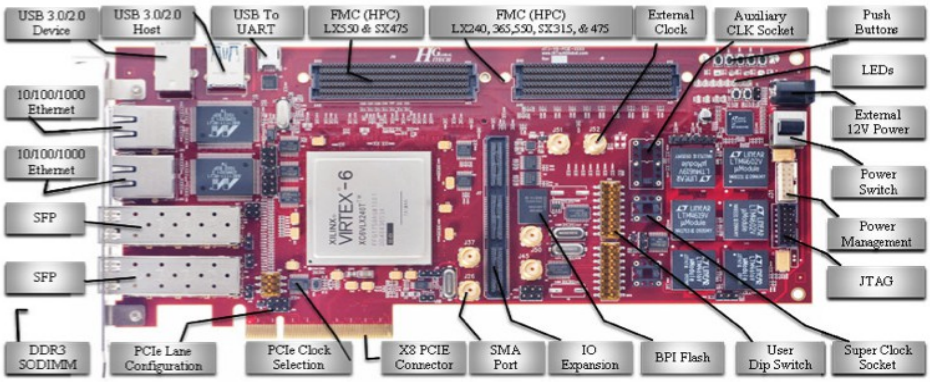

Figura 1.6: Ubicación de componentes de la HTG-V6-PCIe-XXXX [4].

Debido a que el prototipo de memoria compartida distribuida sobre el que se va a integrar este nuevo diseño es un Supermicro H8QM8E, que consta de dos servidores con 4 Quad-Core AMD Opteron 8350 con 16GB de RAM DDR2, toda esta plataforma de desarrollo se ha integrado en un Asus M4A78T-E, con un procesador Quad-Core AMD Phenom II x4 945 y 4 módulos de 2GB de RAM DDR3 bajo el sistema operativo openSUSE 11.3 de Linux, para que las condiciones de ambas máquinas sean lo más parecidas posible. Este ordenador, además, dispone de una ranura libre para PCIe Gen2x8, que será donde la placa de desarrollo se conecte a la hora de probar el funcionamiento del módulo completo.

1.6 Sistema de memoria compartida

El entorno en el cual la unidad SMFU va a estar integrada está representado en el siguiente diagrama de bloques:

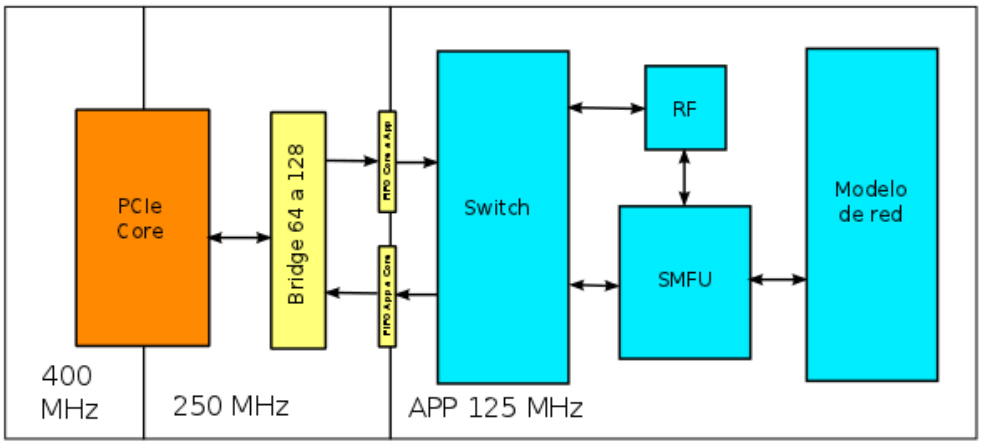

Figura 1.7: Diagrama de bloques del sistema completo.

Dentro de este entorno, se diferenciarán 3 partes:

1) **Core PCI-Express**: Es el módulo principal de todo el sistema. Implementa los 3 niveles lógicos de los que está formada la arquitectura PCIe (*Transacción*, *Enlace* y *Físico*), de acuerdo al estándar.

Tiene un doble rol. Por una parte, se comporta como un *emisor* de paquetes, que se encarga de realizar peticiones de acceso a memoria remota, siendo en este caso el **nodo origen** de las peticiones. Por otra parte, actúa como *receptor* de estos paquetes, siendo el **nodo destino** de la transmisión, responsable de su zona de memoria local, remota para el emisor del paquete.

El ISE Development Suite de Xilinx provee la opción de generar automáticamente un envoltorio para el core, el Xilinx PCIe Core Wrapper versión 2.2, usando la herramienta ISE CoreGenerator. El core resultante, que funciona a una frecuencia de 250 MHz, recibe y procesa paquetes posted, non-posted y completions, con acceso a direcciones de 64 bits, e incluye todo el código necesario para:

- Control de los transceptores Gigabit de la FPGA (GTXs), usados como entrada/salida de datos de la FPGA, y que son parte de la capa física.
- Almacenamiento de paquetes.
- Configuración lógica del core.

- Generación de la interfaz de salida.

2) **Bridge 64 a 128 / FIFOS Asíncronas:** Estos módulos se sitúan en la zona intermedia entre el Core y la aplicación, y se encargan de la formación de paquetes de 128 bits a partir de paquetes de 64 (y viceversa) y su almacenamiento de forma asíncrona para ser procesados por la parte correspondiente (aplicación o core) en su debido momento. Las FIFOS asíncronas son empleadas para el paso de datos y señales de control entre los dos dominios de reloj del diseño: el de App 125 y el de 250-Core (ver figura 1.7).

3) **Aplicación**: Es el módulo encargado del tratamiento de los paquetes. Funciona a una frecuencia de 125 MHz, la mitad que a la que funciona el Core. Como parte emisora, se encarga de procesarlo y encaminarlos en la red, para, una vez en el destino (parte receptora), se encargue de hacerlos llegar bien al core. Es el módulo en el que se encuentra integrada la SMFU, pero además, consta de otros submódulos:

– El *Switch* está conectado con el *Bridge*, el *Banco de Registros* y la *SMFU*, y se encarga de canalizar los paquetes entrantes en la aplicación para distribuirlos entre otros submódulos.

– El *Banco de Registros* (RF, *Register File*), que está conectado con el *Switch* y la *SMFU*, se encarga del almacenamiento de estadísticas y valores necesarios para la configuración de todo el sistema que incluye, como son las direcciones base de zonas de memoria.

– El *Modelo de Red*, que está conectado a la *SMFU*, es un submódulo que simula una red formada por varios nodos.

– La *SMFU*, módulo central de todo el diseño, y cuyo funcionamiento e interfaz se explicará en los siguientes capítulos.

Toda la interfaz interna del sistema, la que se encarga de comunicar estas 3 partes, será una versión modificada de la interfaz de transmisión de flujos de 128 bits AMBA AXI. Esta es la interfaz de transmisión usada por el Xilinx PCIe Core Wrapper versión 2.2 y siguientes. Para más información acerca de este protocolo, ver [6].

2. Shared-Memory Functional Unit (SMFU)

A lo largo de este capítulo, la funcionalidad de la SMFU se presentará de manera detallada. La tarea principal de este módulo es la de facilitar el intercambio de datos entre los nodos de un sistema de memoria compartida distribuida. Para ello, la SMFU consta de un sistema de traducción de direcciones, necesario para la obtención del destino de los datos. Además, también esta dotada de un mecanismo que permite el encaminamiento de los datos a través de la red, así como de la recepción de esta información para ser procesada.

La SMFU diseñada para este proyecto consiste en un rediseño del mismo módulo desarrollado por la Universidad de Heidelberg (Alemania). Los cambios realizados sobre el módulo original se basan en el cambio de protocolo de comunicaciones en el que están diseñados. Mientras que la SMFU original soporta la tecnología HyperTransport, el módulo rediseñado está basado en PCI-Express. Las diferencias entre ambos módulos afectan a la estructura de los paquetes en que se almacena la información, y como esto influye en el flujo de datos producido entre el origen y el destino.

La información a enviar desde un origen a un destino se almacena en paquetes. De entre todos los tipos de paquetes que se pueden formar, la SMFU soporta los siguientes tipos:

- *Peticiones de escritura de memoria*: Son aquellos paquetes enviados por el procesador con una determinada información con la intención de que esta sea escrita en una zona de memoria, remota o local. A este tipo de paquetes también se les conoce como *posted* (*publicados*).

- *Peticiones de lectura de memoria*: También conocidas como *non-posted* (*no publicadas*), estos paquetes sin datos adjuntos son enviados a un procesador con la intención de leer una zona de su memoria local, que es remota a efectos del nodo que solicita su acceso.

- *Completions (finalizaciones)*: Paquetes de respuesta que el

destinatario de una petición de cualquier tipo envía al emisor que las generó. Para la unidad SMFU diseñada en este proyecto, estas finalizaciones harán referencia a las peticiones de lectura de memoria (*non-posted*) que previamente han pasado por ella.

La comunicación entre dos nodos funciona de la siguiente manera: la parte origen envía un paquete que pasa por la SMFU, y en la parte destino, dicho paquete es recibido también por este módulo. En caso de que el paquete sea *non-posted*, el lado origen se mantiene a la espera de recibir una respuesta para dar por finalizada la transacción, en la que este nuevo paquete realizará el mismo recorrido pero a la inversa. Para poder implementar toda esta funcionalidad, se requieren de tres diferentes entidades:

- Un requester (solicitante), que se encarga de preparar las peticiones de lectura y escritura en memoria del nodo origen.

- Un completer (completador), que recibe los paquetes desde el origen, y que se encarga de dirigirlos hacia su destino.

- Un responder (respondedor), que se ocupa de recibir las peticiones *non-posted* y de enviar las *completions*.

Para reducir la complejidad del módulo, se han diseñado dos submódulos que implementan las funciones de estas tres entidades: **Egress** (unidad de egreso) e **Ingress** (unidad de ingreso), y que serán explicadas en los puntos 3.1 y 3.2.

2.1 Formato de paquetes en PCI-Express

Como se ha dicho en el punto anterior, la SMFU acepta 3 tipos de paquetes (escrituras en memoria, lecturas de memoria y finalizaciones). Existen más tipos de paquetes, pero en caso de que estos pasen por la SMFU, serán descartados.

Cada paquete PCIe se compone de 3 partes: cabecera, datos y verificación (esta última parte, opcional). En la Figura 2.1 se puede ver el formato genérico. De ellas, la que más interés tiene es la cabecera, que define el tipo de paquete que se está transmitiendo por la SMFU. Además,

dependiendo del paquete, la estructura de la cabecera puede variar.

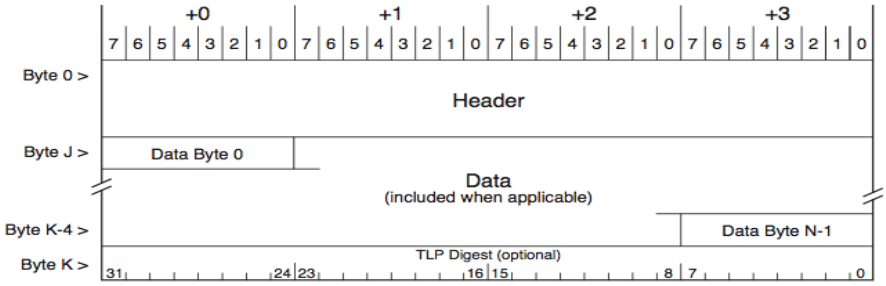

Figura 2.1: Formato de un TLP PCI-Express [3].

Los paquetes de acceso a memoria (*posted* y *non-posted*) tienen una cabecera con un tamaño de 3DW o 4DW (96 o 128 bits, siendo 1DW igual a 32 bits), dependiendo de si el procesador tiene capacidad de direccionamiento de 32 o 64 bits, respectivamente.

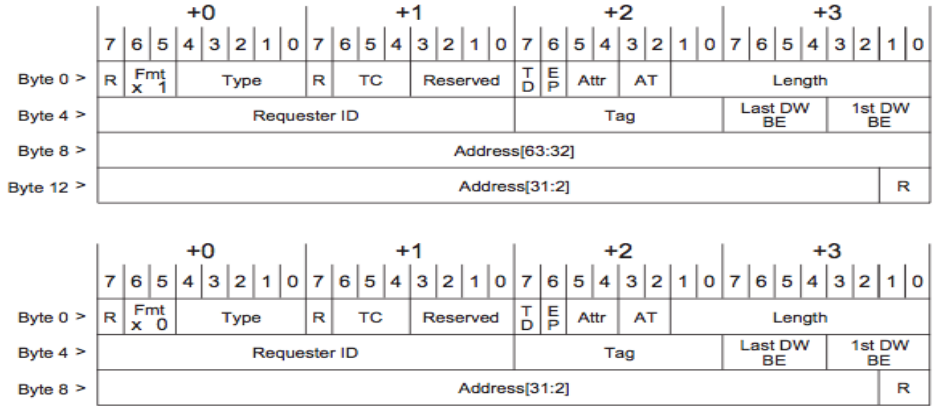

Figura 2.2: Formato de cabecera para peticiones de memoria de 64 y 32 bits respectivamente [3].

De todos los campos que conforman las cabeceras de los paquetes PCIe, hay un conjunto detallado a continuación, que resulta indispensable para permitir el acceso a memoria remota:

– *Fmt[1:0]* (bits 6 y 5 del primer byte) y *Type[4:0]* (bits 4 a 0 del primer byte): Indican el formato y tipo del paquete entrante en la SMFU. Con el valor de estos campos, se podrá averiguar si la cabecera es de un *posted* de 96 bits de cabecera o de un *non-posted* de 128 bits, entre otros.

– *RequesterID[15:0]* y *Tag[7:0]* (bytes 5º al 7º del TLP): A este par de valores se le conoce como **identificador de transacción**, y es la información mediante la cual el procesador destino de un *non-posted* sabe dónde tiene que enviar el paquete de *finalización* (ver punto 2.4 para más información).

– *Address*: Contiene la dirección destino de memoria a la cual se envía el paquete. Con este valor, y mediante unas operaciones, la SMFU se encargará del cálculo del nodo destino y de hacer una traducción de direcciones. Este proceso se explicará con detalle en el punto 2.2.

En lo que respecta a las *finalizaciones*, la cabecera de estas tiene un tamaño de 3DW (96 bits). En ellas, 3 de sus campos son los que tienen algún tipo de utilidad en la SMFU: *Fmt* y *Type*, los cuales tienen el mismo significado que en las cabeceras de los *posted* y *non-posted*; y *Tag[7:0]* (11º byte del paquete), que es el campo mediante el que se asocia la *finalización* con el *non-posted* al que está relacionado.

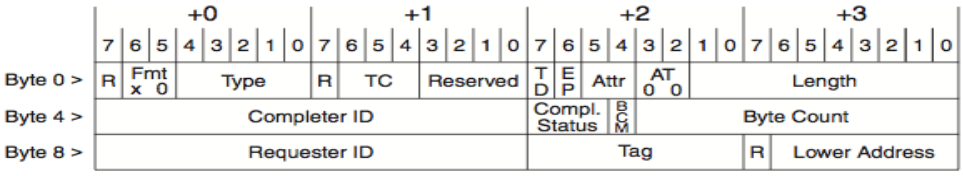

Figura 2.3: Formato de cabecera para respuestas [3].

2.2 Acceso a memoria en sistemas distribuidos y compartidos

Para usar la SMFU en un sistema multiprocesador hace falta un sistema de traducción de direcciones. Al ser un sistema de memoria compartida distribuida, el rango de direcciones de memoria usado para cada uno de los procesadores puede ser diferente. En la figura 2.4 se puede ver más claramente esto:

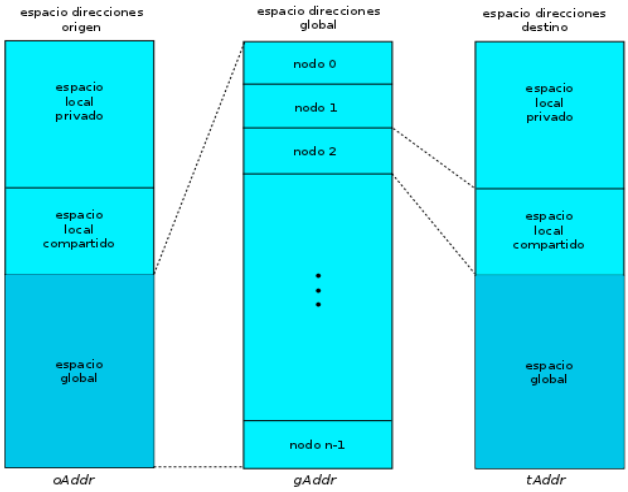

Figura 2.4: Visión general del espacio de direcciones en memoria.

Cada espacio de direcciones de un nodo consta de 2 regiones diferentes: una zona local (que consta de dos espacios: uno privado y otro compartido) y una zona global. El espacio global recoge la información de las zonas privadas compartidas de todos los nodos que forman el sistema.

Uno de los objetivos de la SMFU es el de proveer un mecanismo de traducción de direcciones, en el caso en que una petición de lectura o de escritura acceda al módulo. En la cabecera de dichas peticiones está almacenada la dirección destino del paquete, pero vista desde el nodo origen. Partiendo de este valor, el mecanismo implementado se encarga de averiguar el nodo hacia el cual debe ser mandada la petición. Conocido el nodo destino, se traduce la dirección de forma que se acceda a la dirección de memoria correspondiente.

Para realizar todo el proceso de traducción de direcciones, son necesarios un conjunto de valores. Estos valores se encuentran almacenados en el banco de registros incluidos en el entorno de desarrollo, y son los siguientes:

– **oStartAddress**: Dirección en la cual empieza la zona compartida de memoria del procesador origen. Este valor está almacenado en un registro, y es accesible por todos los nodos del sistema de manera local.

– **tStartAddress**: Dirección en la cual empieza la zona compartida de

memoria del procesador destino. Este valor aparece almacenado dentro de un bloque de memoria en el banco de registros. Dicho bloque es una tabla de correspondencias que contiene las direcciones de inicio de la zona compartida de memoria de todos los nodos del sistema. Una vez se haya calculado el nodo destino de la petición, la SMFU accede a esta tabla y extrae este valor.

– **mask**: Valor de máscara que indica los bits de la dirección que hacen referencia al nodo destino. Este valor está almacenado en un registro, y es el mismo para todos los nodos del sistema.

Los registros **oStartAddress** y **mask** son accesibles en todo momento por la SMFU y de manera inmediata. En cambio, para obtener **tStartAddress**, se necesita al menos un ciclo desde el momento en el que el nodo destino ha sido calculado.

Con estos valores conocidos, el proceso de traducción de direcciones ya puede llevarse a cabo. El proceso está basado en el implementado por S. Scott [7], y descrito por H.Fröning en [8], ya que añade una mínima latencia a la SMFU.

En primer lugar, y antes de calcular el nodo destino, se debe obtener el desplazamiento general (*gAddr*). Este valor se calcula como la diferencia entre la dirección indicada en la cabecera de la petición (*oAddr*) y la dirección de inicio de la zona de memoria compartida del origen (*oStartAddress*). El desplazamiento calculado será el mismo para cualquier zona de memoria de cualquier nodo del sistema.

$$gAddr = oAddr - oStartAddress$$

El valor calculado servirá tanto para calcular el nodo destino como la dirección final. El valor del nodo destino dentro de la dirección es el indicado por los bits puestos a 1 en la máscara (*mask*):

$$tNodeID = (gAddr \,\&\, mask) >> shift\_count$$

El valor de *shift_count* sirve para alinear el valor de *tNodeID* a la derecha.

Por último, con el nodo destino calculado, se accederá al banco de registros para obtener *tStartAddress*, y finalmente, obtener el valor de la

dirección destino (*tAddr*):

$$tAddr = (gAddr \ \& \sim mask) + tStartAddress$$

La latencia extra que implica la traducción de la dirección destino supone 4 ciclos de retardo (y que se corresponde con la latencia del diseño original, la cual ha sido mantenida) desde que empiezan a salir por la red los paquetes desde la SMFU. Este tiempo extra solo es aplicable a las peticiones de lectura y escritura en el nodo origen, por lo que este mecanismo va a situarse dentro de la unidad Egress.

2.3 Lecturas y escrituras a memoria remota

Los tres tipos de paquetes que acepta la SMFU se traducen en dos operaciones básicas de acceso a memoria: lectura y escritura [2].

La escritura implica una operación sencilla: un procesador (A) quiere escribir una determinada información en una zona de memoria remota, accesible de forma directa por otro procesador (B). Por ello, A realiza una petición de escritura en memoria, enviando un paquete *posted*. B recibe este paquete, y escribe el contenido de los datos en la zona de memoria indicada. Las figuras 2.5 y 2.6 muestran el camino recorrido por el mensaje:

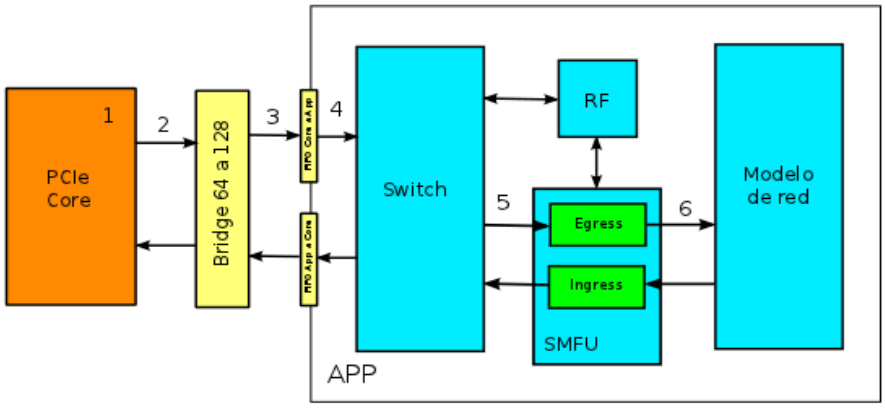

Figura 2.5: Parte origen en una operación posted.

1. El Core (A) genera una petición de escritura (*posted*) en memoria

2. El paquete completo se transmite de A al Bridge en tramas de 64 bits.

3. El Bridge forma tramas de 128 bits a partir de las tramas recibidas, y las envía a la FIFO asíncrona Core2App.

4. La FIFO Core2App envía las tramas al Switch.

5. El Switch manda las tramas a la SMFU.

6. La SMFU (unidad Egress, actuando como solicitante) calcula el nodo destino y traduce la dirección, y envía esta nueva información junto con el paquete al modelo de red.

7. El modelo de red simula el camino que el paquete sigue por la red, hasta llegar a la SMFU del destino (B).

8. La SMFU (unidad Ingress, actuando como completador) recibe el paquete y lo manda al Switch.

9. El Switch manda el paquete a la FIFO asíncrona App2Core.

10. La FIFO suministra las tramas al Bridge.

11. El Bridge forma tramas de 64 bits con las tramas de 128 bits recibidas.

12. El Bridge envía el paquete al procesador destino (B).

13. B almacena la información en memoria.

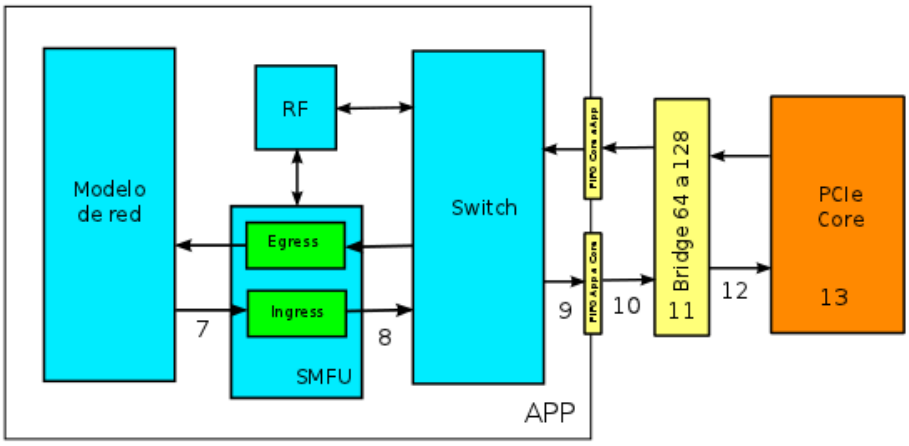

Figura 2.6: Parte destino en una operación posted.

La lectura, en cambio, conlleva una operación más compleja. En esta situación, un procesador (A) quiere leer una información de memoria que se encuentra en una ubicación remota, accesible por otro procesador (B). A realiza una petición de lectura de memoria, enviando un paquete *non-posted* a B. Cuando B recibe este paquete, se dispone a generar un nuevo TLP, en este caso, una *finalización*, con destino hacia A. Este paquete podrá contener los datos de la dirección de memoria que A quería leer, o simplemente, una confirmación de que el *non-posted* ha llegado correctamente a su destino. El trayecto que sigue la *finalización* para llegar de B a A es similar al efectuado con los otros paquetes, pero de sentido inverso. En las figuras 2.7 y 2.8 se puede ver el recorrido realizado.

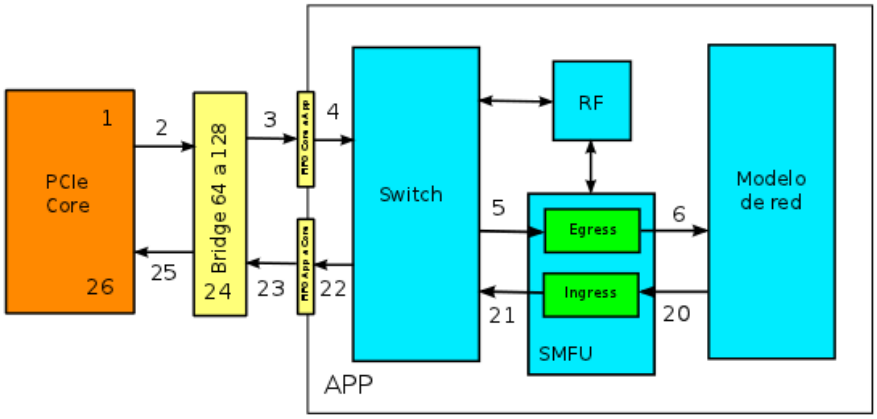

Figura 2.7: Parte origen en una operación non-posted.

1. El Core (A) genera una petición de lectura (*non-posted*) en memoria
2. El paquete completo se transmite de A al Bridge en tramas de 64 bits.
3. El Bridge forma tramas de 128 bits a partir de las tramas recibidas, y las envía a la FIFO asíncrona Core2App.
4. La FIFO Core2App envía las tramas al Switch.
5. El Switch manda las tramas a la SMFU.
6. La SMFU (unidad Egress, actuando como solicitante) calcula el nodo

destino y traduce la dirección, y envía esta nueva información junto con el paquete al modelo de red. Además, envía una trama extra junto con el paquete que informa que el nodo A es el emisor del paquete.

7. El modelo de red simula el camino que el paquete sigue por la red, hasta llegar a la SMFU del destino (B).

8. La SMFU (unidad Ingress, actuando como la primera parte del respondedor) recibe el paquete, guarda la información relativa al identificador de transacción y al nodo origen (A), y genera una etiqueta que le pasa al paquete.

9. La SMFU envía el paquete al Switch.

10. El Switch envía el paquete a la FIFO asíncrona App2Core.

11. La FIFO suministra las tramas al Bridge.

12. El Bridge forma tramas de 64 bits con las tramas de 128 bits recibidas.

13. El Bridge manda el paquete a su destino (B). B genera un paquete de finalización (*completion*) y lo envía con destino a A. Dicho paquete tendrá en el campo *Tag* de su cabecera el valor que la unidad Ingress de la SMFU de B generó.

14. El Core (B) envía el paquete al Bridge.

15. El Bridge forma tramas de 128 bits a partir de las tramas recibidas, y las envía a la FIFO asíncrona Core2App.

16. La FIFO Core2App envía las tramas al Switch.

17. El Switch manda las tramas a la SMFU.

18. La SMFU (unidad Egress, actuando como la segunda parte del respondedor) toma la etiqueta que generó la primera parte del respondedor, y le devuelve la información relativa su destino (identificador de transacción + nodo origen).

19. La SMFU envía el paquete al modelo de red.

20. El modelo de red simula el camino que el paquete sigue por la red, hasta

llegar a la SMFU del origen (A).

21. La SMFU (unidad Ingress, actuando como completador) recibe el paquete y lo manda al Switch.

22. El Switch manda el paquete a la FIFO asíncrona App2Core.

23. La FIFO suministra las tramas al Bridge.

24. El Bridge forma tramas de 64 bits con las tramas de 128 bits recibidas.

25. El Bridge envía el paquete al procesador origen (A).

26. A recibe el paquete de *finalización*.

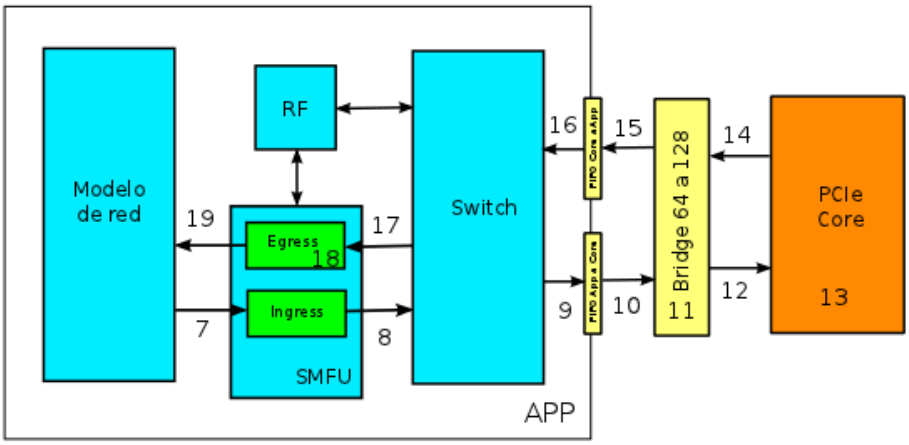

Figura 2.8: Parte destino en una operación non-posted.

2.4 Identificador de transacciones en sistemas de memoria compartida distribuida

Como se ha podido ver en el punto anterior, las operaciones de lectura de memoria son algo más complejas que las de escritura. Ya que los datos leídos deben ser enviados al nodo origen, en un sistema distribuido se presentan los siguientes casos:

- Si la comunicación fuera únicamente entre dos procesadores (un origen, un destino), no existiría ningún problema. El único emisor de la petición

de lectura sería el destinatario único de la respuesta generada, y no habría conflictos de ningún tipo.

- Si la comunicación fuera entre un emisor y varios destinos, seguirían sin existir problemas, por las mismas razones expuestas en el supuesto anterior.

- Sin embargo, en caso de que existieran varios emisores que envían peticiones de lectura a un único destinatario, se produciría un conflicto. El núcleo receptor debería enviar tantas respuestas como peticiones haya atendido, pero no podría enviarlas a un emisor en concreto, ya que no se dispone de la información suficiente para poder hacerlo.

- De la misma manera, el problema existiría también en caso de que varios emisores enviaran peticiones a varios receptores diferentes.

Para resolver este problema, la SMFU consta de una tabla de correspondencias, en la cual se almacene la información relativa al origen de la petición de lectura. Así, cuando la respuesta a la petición de lectura se genere, tomará esta información guardada para poder llegar al origen correctamente. Por ello, la SMFU consta de un módulo llamado **Matching-Store** (*Almacén de Correspondencias*), definido originalmente en [2]. A continuación se explicará en detalle de cara a la comprensión del funcionamiento de la SMFU.

La Matching-Store (MS) se ubica en el destino de una petición de lectura. Esta tabla es accedida únicamente por las unidades Egress e Ingress, cuando estas actúen como respondedoras. Es decir, la unidad **Ingress** accede a la MS cuando un paquete *non-posted* es procesado; y la unidad **Egress** lo hará con las paquetes correspondientes a respuestas (*completions*).

El funcionamiento de la Matching-Store se explica a continuación, tal y se puede ver gráficamente en la Figura 2.9:

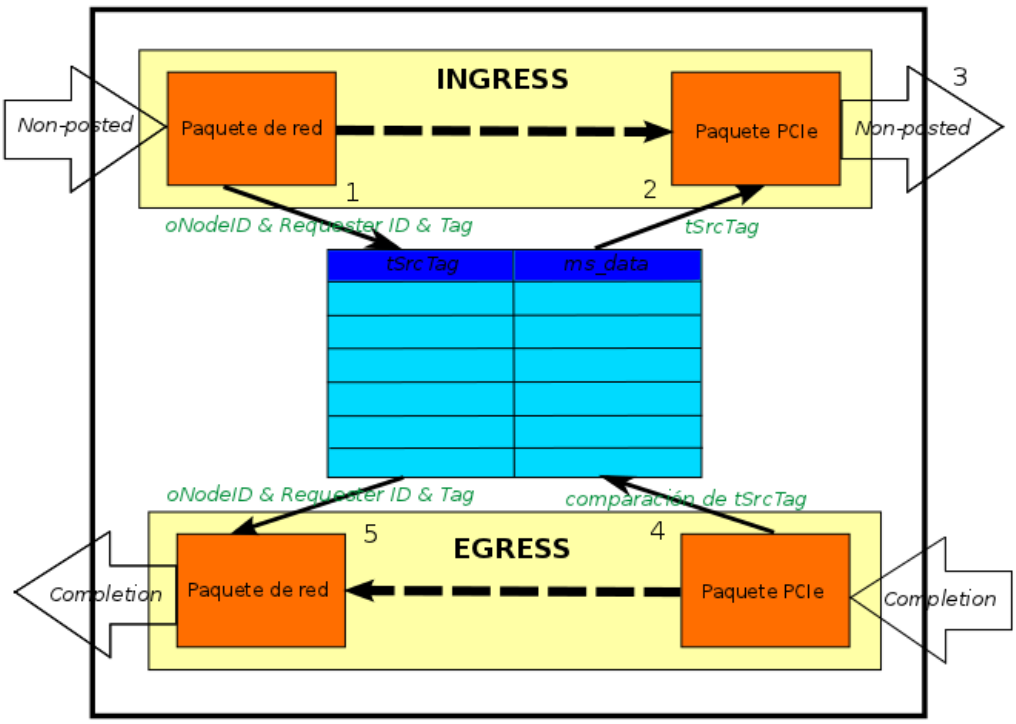

Figura 2.9: SMFU con Matching-Store en el nodo destino.

1. Cuando una petición de lectura (paquete *non-posted*) es procesada por la unidad Ingress, esta accede a la Matching-Store y almacena información acerca del origen de la petición.

2. La Matching-Store devuelve a la unidad Ingress una etiqueta, que se inserta en la cabecera del paquete en el campo *Tag* (ver figura 2.2).

3. La petición de lectura sale de la unidad Ingress.

Los puntos 1 al 3 se repetirán mientras peticiones de lectura, procedentes de un único o varios origenes, vayan llegando al destino. Por ello, la Matching-Store cuenta con un total de 32 entradas para evitar el problema de las múltiples peticiones de lectura.

En el momento que una *completion* accede a la unidad Egress, la Matching-Store vuelve a entrar en juego:

4. La *completion* generada por el destino lleva en su cabecera la etiqueta que la Matching-Store devolvió a la petición de lectura al almacenar

información en ella. Esta se busca en la tabla de correspondencias, y el valor que tenga asociado se devuelve a la respuesta.

5. La información obtenida pasa a formar parte del paquete, y este sale de la unidad Egress conociendo el procesador al cual tiene que llegar.

Este conjunto de valores que se almacena en la Matching-Store en la unidad Ingress, y que se vuelven a leer en la unidad Egress, son los siguientes:

- **oNodeID**: Identificador del nodo que ha generado la petición de lectura.

- **Requester ID + Tag**: Identificador de la transacción (en este caso, de la petición de lectura.

Estos valores son asociados a un ínidice que identifca cada una de las entradas en la Matching-Store. Este índice es el **tSrcTag**, que a su vez sirve de identificador de cada una de las 32 entradas que, como máximo, puede haber almacenadas simultáneamente en la Matching-Store. Su valor se almacena en el paquete dentro del campo *Tag* de la cabecera (ver 2.1).

La razón por la que se ha determinado que el número de entradas de la Matching-Store sea 32 es debido al número máximo de transacciones que pueden realizarse, identificado por el campo *Tag* de la cabecera. Existe un bit en PCI-Express llamado *Extended Tag Field Enable* que, cuando está habilitado, permite un máximo de 256 transacciones simultáneas (ver punto 2.2.6.2 en [3]), por las 32 que como máximo permite HyperTransport. Por defecto, este bit está deshabilitado, lo que limita el número máximo de transacciones a 32 (poniendo los 3 bits de mayor peso de *Tag* a cero). Por el momento, esta funcionalidad no está implementada en el diseño realizado en el proyecto, por lo que podría desarrollarse para futuras versiones, que supondrían una mejora sobre la actual.

2.5 Flujo de datos

La SMFU, como ya se ha visto, está integrada dentro de un sistema y se comunica por un lado con el switch y por el otro con la red (ver figura 2.5). Entre ambos lados se establece el flujo de datos que permite el envío (o

recepción) de paquetes a través de la red. Cabe destacar este aspecto, ya que la definición del flujo de datos así como su repercusión en la SMFU son específicos para este proyecto fin de carrera.

Para que el flujo de datos se realice correctamente y de manera más eficiente, los paquetes se dividen en pequeños conjuntos de datos, llamados **tramas**. El formato que tiene una sucesión de tramas es diferente dependiendo del lado en que se encuentre (en la parte del switch o en la parte de la red), y la SMFU debe ser capaz de que esta conversión de formatos se realice correctamente.

El formato de las tramas sigue una modificación del estándar de transmisión AMBA AXI 2.2, con un tamaño de 128 bits [6]. En la figura 2.10 se puede ver el significado del flujo de datos en un ejemplo:

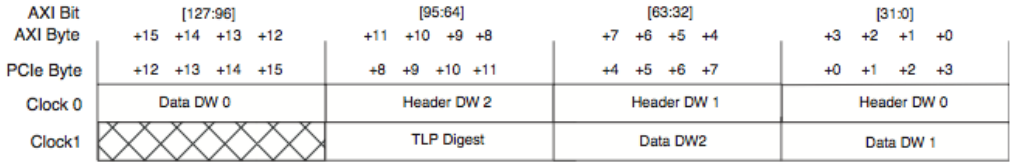

Figura 2.10: Transmisión de paquete con inicio al principio de la trama [5].

El paquete transmitido consta de una cabecera de 3 DW (96 bits) y una carga de datos y verificación de 4 DW (128 bits), por lo que se necesitan de al menos dos tramas para que pueda ser enviado. Dentro de cada trama, cada conjunto de 32 bits (DW) por el que está formado el paquete sigue una organización *Little Endian*: el DW más significativo estará en la posición menos significativa de la trama, el 2º DW más significativo estará en la segunda posición menos significativa, y así hasta llenar la trama, y procediendo de igual manera en las siguientes.

Esta forma de organización es común para ambas partes de la SMFU. Los aspectos destacados a cada parte son los siguientes:

1. Parte Switch – SMFU: A este lado de la SMFU (entrada de la unidad Egress y salida de la unidad Ingress) el contenido de los paquetes se transmite de forma consecutiva y sin espacios entre el primer DW del paquete y el último. Además, el inicio de la transmisión del paquete puede situarse en dos posibles

partes: al principio de la trama (figura 2.10), o a mitad de la misma (Figura 2.11).

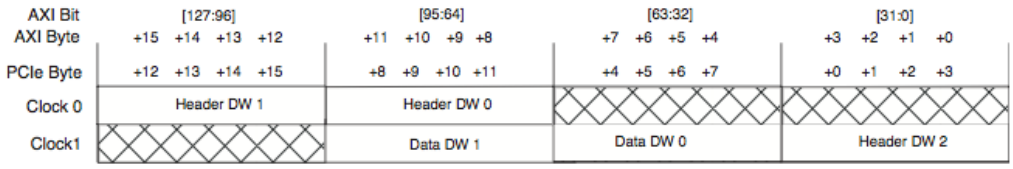

Figura 2.11: Transmisión de paquete con inicio a mitad de la trama.

2. Parte SMFU – Red: En este otro lado (salida de la unidad Egress, entrada de la unidad Ingress), se hace diferencia entre trama de cabecera y trama de datos. Eso quiere decir que, en el caso de la figura 2.10, este tipo de transmisión no sería válido a este lado. Además, el inicio de todos los paquetes estará al principio de la trama. En la figura 2.12 se puede ver un ejemplo de transmisión de este tipo, donde se puede ver que cabecera y datos se envían en tramas separadas.

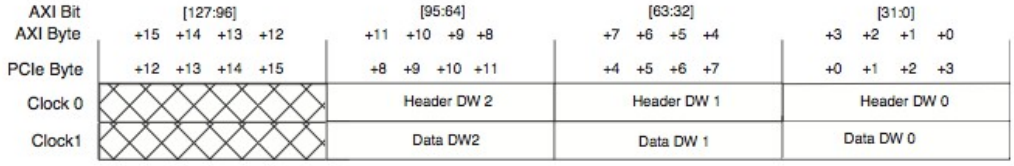

Figura 2.12: Transmisión de paquete en el lado SMFU – Red.

Además, a la salida de la Egress (parte SMFU – Red), y antes de que empiece a transmitirse el paquete por la red, la SMFU envía una trama previa a la cabecera, en la cual está contenido el identificador del nodo destino de la petición generada. Este valor debe enviarse lo antes posible a siguientes módulos, encargados de decidir el encaminamiento del paquete por la red.

En el caso de las peticiones de lectura (*non-posted*), tendrán además una trama extra, que contendrá el identificador del nodo origen (el que generó la petición). Este valor es indispensable, ya que se corresponde con el nodo destino de la respuesta generada a dicha petición. Una vez alcance la unidad Ingress, consumirá este valor, almacenándolo en la Matching-Store, y no dándoselo al nodo destino.

Las figuras 2.13 y 2.14 muestran la estructura de las peticiones a la salida de la Egress y a la entrada de la Ingress, respectivamente.

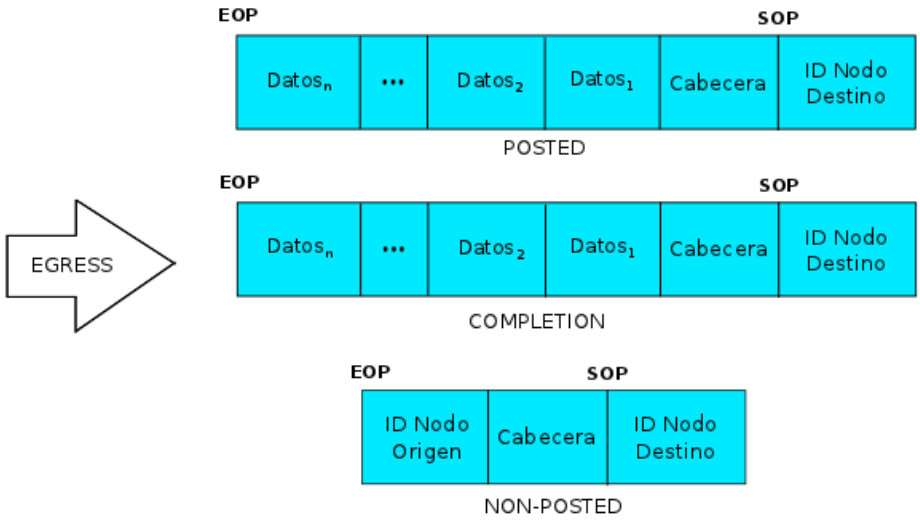

Figura 2.13: Flujo de datos a la salida de la unidad Egress.

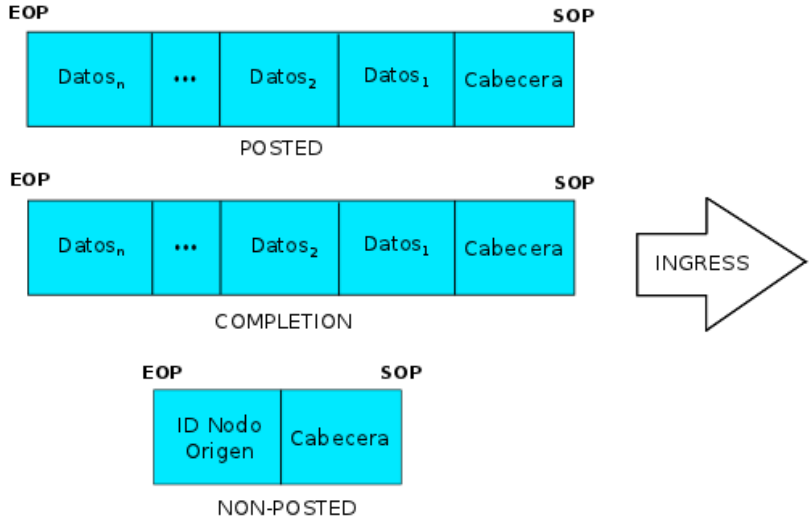

Figura 2.14: Flujo de datos a la entrada de la unidad Ingress.

A la salida de la Egress, las peticiones de lectura (*posted*) y respuestas (*completion)* inician la transferencia del paquete después de enviar el identificador del nodo destino. En la primera trama se transmite la cabecera, y en las tramas posteriores, la carga de datos, hasta llegar al final del paquete. Las peticiones de lectura se transmiten de manera similar, aunque estas no tienen carga de datos. Después de la cabecera, se enviará la última trama, con

el valor del nodo origen, tras la que terminará la transmisión.

El formato de las tramas a la entrada de la Ingress es el mismo que a la salida de la Egress, con la única diferencia que no reciben la trama que contiene el identificador del nodo destino.

2.6 Arquitectura de la SMFU

Tras lo expuesto en los puntos anteriores de este capítulo, la SMFU está formada por 3 submódulos, que son los siguientes:

- **Unidad Egress (de egreso)**: Módulo que se encarga de recibir los paquetes procedentes desde el Core. Su tarea principal es traducir la dirección de memoria a la que tienen que acceder, de forma que el nodo objetivo pueda entenderla. Además, añade la ruta de encaminamiento de los paquetes dentro de la red. En el caso de las respuestas, la unidad Egress actúa como un *respondedor*, indicándoles el núcleo que solicitó la petición de lectura por la que fueron generadas.

- **Unidad Ingress (de ingreso)**: Módulo que se encarga de recibir los paquetes procedentes de la red. Su tarea principal es la de encaminar los paquetes hacia el nodo destino, actuándo como un *completador*. En el caso de las peticiones de lectura (*non-posted*), también se ocupa de la función de *respondedor*, almacenando en la Matching-Store la información referente al origen que las generó.

- **Matching-Store**: Módulo que se encarga de hacer corresponder las peticiones de lectura con sus respuestas.

En las figuras 2.15 y 2.16 se puede ver gráficamente como es la SMFU, tanto en la parte del origen como en la del destino.

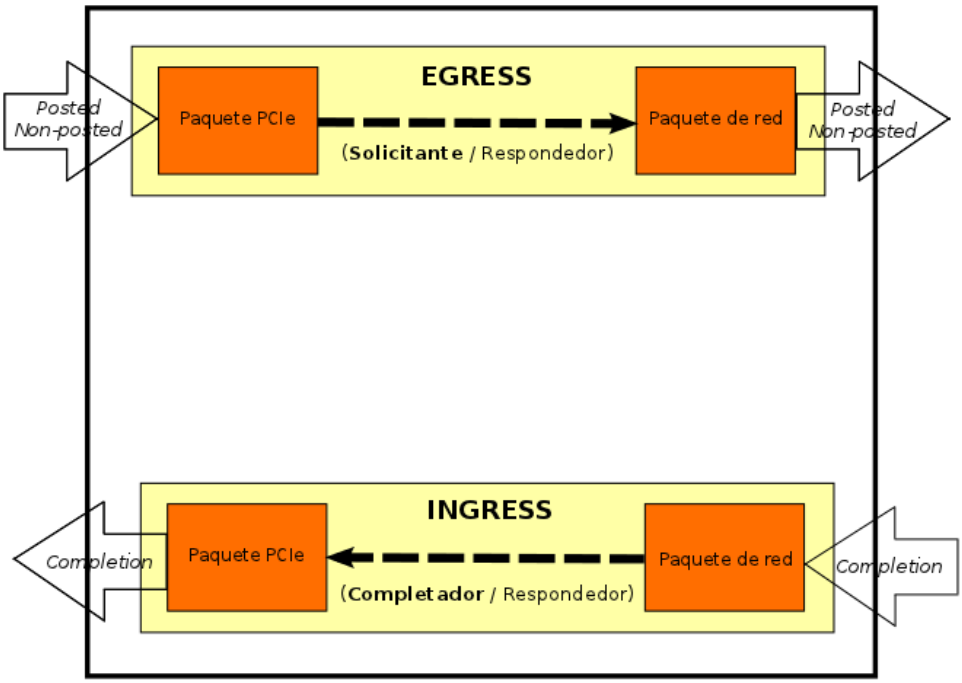

Figura 2.15: SMFU en la parte origen.

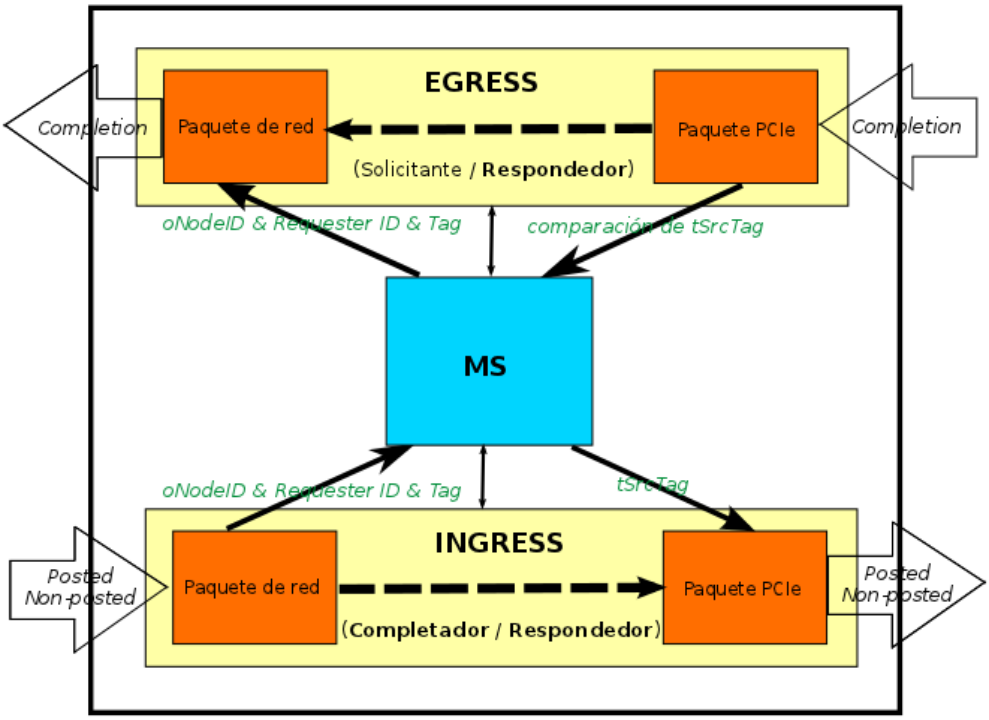

Figura 2.16: SMFU en la parte destino.

3. SMFU para PCI-Express

En este capítulo se explica el funcionamiento de la SMFU para PCIe y como ha sido diseñada. Además, se detallará la funcionalidad de cada uno de los módulos de los que se compone.

La SMFU está formada por 3 módulos (unidad Egress, unidad Ingress y Matching-Store). La figura 3.1 muestra un esquema general de como están conectados los módulos.

Además, cuenta también con una serie de conexiones a otros módulos que forman parte del sistema de acceso de memoria remoto completo (*Modelo de Red*, *Switch* y *Banco de Registros*), y que son imprescindibles para el correcto funcionamiento de la SMFU.

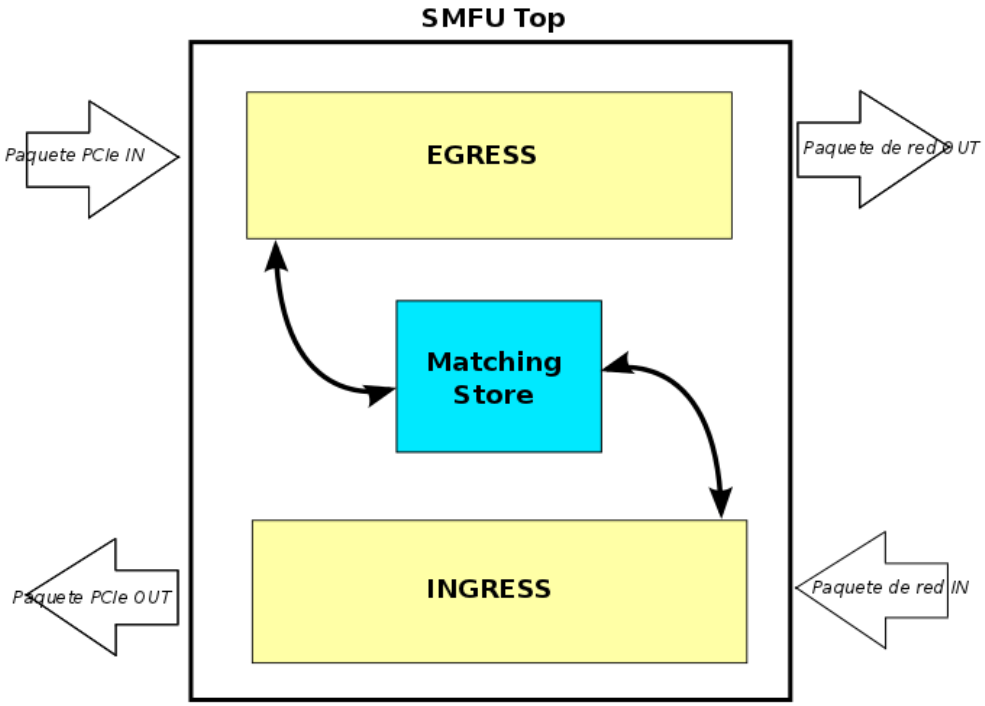

Figura 3.1: Esquema general SMFU con submódulos

3.1 Unidad Egress

El objetivo de la Unidad Egress es el de enviar paquetes procedentes

del procesador a la red. Principalmente, es responsable de dos tareas: en primer lugar, actúa como un solicitante, que se encarga de retransmitir las peticiones procedentes del procesador a la red; y en segundo lugar, actúa como un respondedor, enviando respuestas (*completions*) a los nodos que han enviado una petición de lectura (*non-posted*).

3.1.1 Interfaces

La interfaz de la unidad Egress tiene diferentes señales que pueden ser organizadas atendiendo a su origen/destino. Hay 6 señales conectadas al *AXI Switch*, que provee a la SMFU de paquetes entrantes; 6 señales conectadas al *Modelo de Red*, por donde los paquetes serán mandados; 3 señales conectadas a la *Matching Store*, responsable de facilitar a las respuestas información de su destino; y 9 señales conectadas al *Banco de Registros*. Además, existe un conjunto de señales de propósito general, utilizadas para controlar el reloj y el reset del sistema, y para averiguar el estado dentro de su máquina de estados en el que se encuentra el módulo. En la siguiente tabla aparecen listadas dichas señales, indicando su polaridad (entrada: *in*, salida: *out*) y una breve descripción de las mismas.

Nombre	I/O	Descripción
clk	in	Señal general de reloj.
rst_n	in	Señal general de reset activa a nivel bajo.
smfu_egress_state[4:0]	out	Estado actual de la unidad Egress
Señales al AXI Switch		
pcie2smfu_axis_tx_tlast	in	Indica si la trama entrante es la última del paquete.
pcie2smfu_axis_tx_tdata[127:0]	in	Trama de datos entrante en la unidad Egress
pcie2smfu_axis_tx_tuser[1] (terr_fwd)	in	Indica si el paquete entrante es erróneo.
pcie2smfu_axis_tx_tuser[14:10] (is_sof[4:0])	in	Indica el comienzo de una nueva cabecera en *pcie2smfu_axis_tx_tdata.* Además, *is_sof[4]* indica que un nuevo paquete esta entrando, con su comienzo a la parte derecha del bus, y *is_sof[3]* indica que su comienzo es en la mitad. El resto de bits hacen referencia a la posición del byte en que empieza el nuevo paquete, codificado en binario.
pcie2smfu_axis_tx_tuser[21:17] (is_eof[4:0])	in	Indica el final de un paquete en *pcie2smfu_axis_tx_tdata*. El bit de mayor peso tiene el mismo funcionamiento que *pcie2smfu_axis_tx_tlast* y el resto, hacen referencia a la posición del byte en el cual termina el paquete, codificado en binario.

pcie2smfu_axis_tx_tvalid	in	Indica la existencia de datos válidos en *pcie2smfu_axis_tx_tdata*.
pcie2smfu_axis_tx_tready	out	Señal que indica que el core está listo para transmitir datos por *pcie2smfu_axis_tx_tdata[127:0]*.
Señales al Modelo de Red		
smfu2network_axis_tx_tlast	out	Indica si la trama saliente es la última del paquete.
smfu2network_axis_tx_tdata[127:0]	out	Trama de datos saliendo de la unidad Egress.
smfu2network_axis_tx_tuser[21:17] (is_eof[4:0])	out	Indica el final de un paquete en *smfu2network_axis_tx_tdata*. El bit de mayor peso tiene el mismo funcionamiento que *smfu2network_axis_tx_tlast* y el resto, hacen referencia a la posición del byte en el cual termina el paquete, codificado en binario.
smfu2network_axis_tx_tuser[14:10] (is_sof[4:0])	out	Indica el comienzo de una nueva cabecera en *pcie2smfu_axis_tx_tdata.* Además, *is_sof[4]* indica que un nuevo paquete esta entrando, con su comienzo a la parte derecha del bus, y *is_sof[3]* indica que su comienzo es en la mitad. El resto de bits hacen referencia a la posición del byte en que empieza el nuevo paquete, codificado en binario.
smfu2network_axis_tx_tuser[15] (frame)	out	Indica el tipo de trama saliente (0 = Paquete de red, 1 = tNodeID).
smfu2bridge_axis_rx_tvalid	out	Indica que el core está presentando datos válidos en *smfu2network_axis_tx_tdata.*
smfu2bridge_axis_rx_tready	in	Indica que el core está listo para transmitir datos por *smfu2network_axis_tx_tdata.*
Señales al Banco de Registros		
smfu_mask[63:0]	in	Máscara.
smfu_sent_count_completion_incr	out	Indica la existencia de una completion entrante en la Egress.
smfu_sent_count_nonposted_incr	out	Indica la existencia de una petición de lectura entrante en la Egress.
smfu_sent_count_posted_incr	out	Indica la existencia de una petición de escritura entrante en la Egress
smfu_sent_count_error_incr	out	Indica la existencia de un paquete erróneo entrante en la Egress.
smfu_sent_count_others_incr	out	Indica la existencia de otro tipo de paquete no contemplado entrante en la Egress.
smfu_balt_hw_address[5:0]	out	Nodo destino del paquete entrante.
smfu_balt_hw_read_data[51:0]	in	Dirección de inicio del nodo destino.
smfu_general_offset[51:0]	in	Dirección de inicio del nodo origen.
oNodeID	in	Identificador del nodo origen.
Señales a la Matching-Store		
ms_transID[n:0]	in	Conjunto de datos que la Matching-Store da a las completions (oNodeID, requesterID y Tag), necesarios para conocer su destino.
rel_oSrcTag	out	Indica si el paquete actual en la Egress es una

		completion o no.
ms_sourceTag[4:0]	out	Identificador único para peticiones de lectura (las que requieren una respuesta).

Tabla 3.1: Señales de la unidad Egress

La conexión al *AXI Switch* consiste en 4 entradas y 1 salida. La señal *pcie2smfu_axis_tx_tvalid* indica el momento en que están entrando datos válidos a la unidad Egress. La señal de datos es un bus de 128 bits, *pcie2smfu_axis_tx_tdata*. Con la señal is_sof[4:0] (insertada en *pcie2smfu_axis_tx_tuser[14:10]*), averiguamos si la trama entrante es la primera de un paquete, y en qué lugar empieza: al principio del bus de datos o en su posición media. De la misma manera, la señal is_eof[4:0] (*pcie2smfu_axis_tx_tuser[14:10]*), indica si la trama actual es la última del paquete y, en caso de que lo sea, en que posición dentro del bus de datos finaliza. La única señal de salida de este conjunto, *pcie2smfu_axis_tx_tready*, indica si la SMFU está lista para empezar a recibir los paquetes entrantes. Finalmente, la señal *terr_fwd* (*pcie2smfu_axis_tx_tuser[1]*) indica si el paquete entrante es erróneo.

En las figuras 3.2 y 3.3 se pueden ver dos ejemplos de paquetes entrantes en la unidad Egress, y cual es el comportamiento de dichas señales. En la figura 3.2, el paquete empieza en la parte menos significativa del bus de datos, y en la 3.3, comienza a mitad del bus. Dentro de la señal *pcie2smfu_axis_tx_tdata*, los datos aparecen representados con H y D, haciendo referencia a si forman parte de la cabecera (H) o de la parte de datos (D).

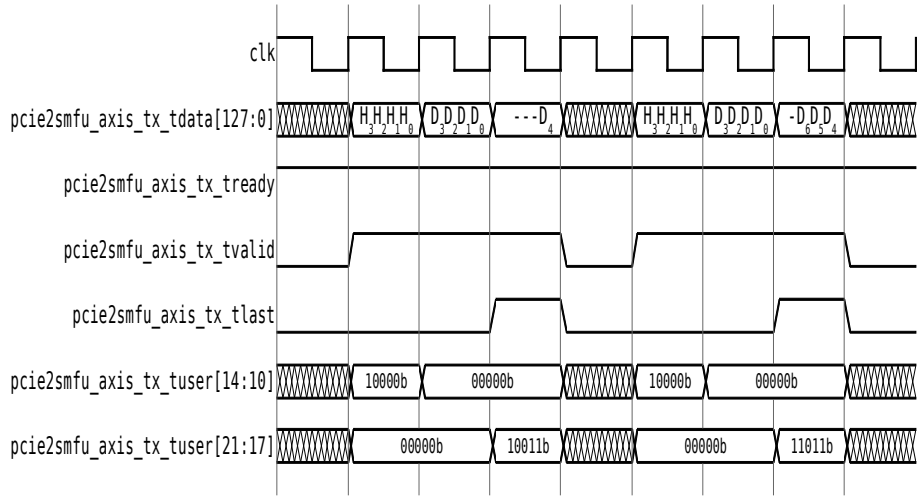

Figura 3.2: Paquete entrante en la unidad Egress.

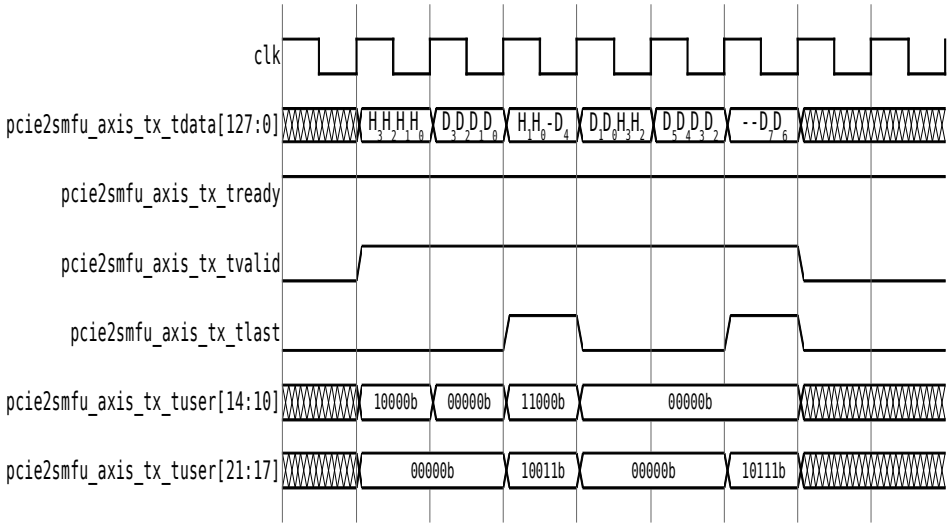

Figura 3.3: Paquete entrante en la unidad Egress, inicio a mitad de ciclo.

La conexión entre la SMFU y el *Modelo de Red* consiste también en 5 señales: 4 salidas y una entrada. El comportamiento de este bus es similar al explicado a la entrada de la Egress, pero con la polaridad cambiada (las que antes eran entradas ahora son salidas y viceversa). Además, en la señal *smfu2network_axis_tx_tuser*, en su bit 15 se ha añadido una nueva funcionalidad. Este bit indica el tipo de trama que se está enviando hacia la

red: si en la trama saliente está almacenado el valor del identificador del nodo destino, este bit valdrá '1'; en caso contrario, valdrá '0'.

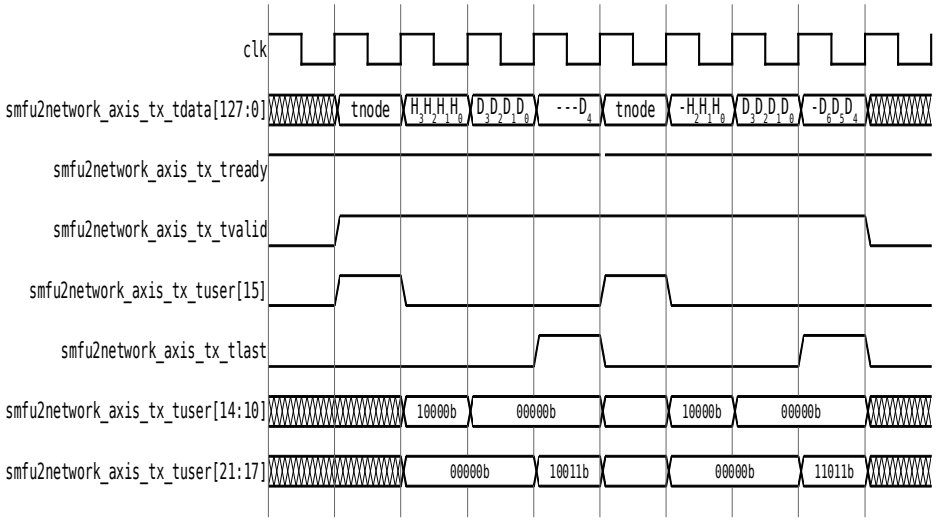

Figura 3.4: Paquete saliendo de la unidad Egress.

En la figura 3.4 se puede ver un ejemplo en el que se muestra el correcto comportamiento de los paquetes salientes de la unidad Egress. En este caso, una situación con paquetes empezando a mitad de un ciclo no se va a dar, ya que la unidad Egress se encarga de ello. Además, separa la cabecera y datos en diferentes tramas, como ya se explicó anteriormente en el punto 2.5.

La conexión entre la SMFU y la *Matching-Store* consta únicamente de tres señales: una entrada y dos salidas. La entrada es el conjunto de datos que se leen de la MS, procedentes de una petición de lectura. Este conjunto de datos son el identificador del nodo origen (*oNodeID*) y el identificador de transacción (*requesterID* y *Tag*). Estos datos se transmiten cuando el paquete entrante en la Egress es una respuesta (la señal *rel_oSrcTag* lo indica), y se activa el bus *ms_sourceTag*.

Finalmente, al *Banco de Registros*, hay conectadas 9 señales, que se pueden clasificar en varios grupos:

– Seis de ellas son un grupo de salidas binarias del tipo *smfu_sent_count_PACK_incr*, donde PACK = {*posted, nonposted, completion,*

error, others}. El hecho de que estas señales se encuentren a 1 indican la existencia de un paquete de ese tipo en la unidad Egress.

– Un segundo grupo engloba a un conjunto de señales, necesarias para el cálculo del nodo destino y la traducción de la dirección destino de la petición. En el bus *smfu_general_offset*, se almacena la dirección de inicio de la zona de memoria compartida en el nodo origen.

– El último grupo consta de una señal, *oNodeID*, que indica el identificador del nodo origen. Este valor es el que se pasa a las peticiones de lectura, para que posteriormente lo puedan almacenar en la Matching-Store.

3.1.2 Submódulos

Además de todas las señales mencionadas, y de la lógica de control necesaria para su funcionamiento, la unidad Egress está formada por 5 submódulos: una FIFO síncrona y cuatro módulos de desplazamiento de bits. Estos submódulos son los mismos que los usados en [2], y su funcionamiento viene explicado con detalle a continuacion.

FIFO Síncrona

Este submódulo consiste en una estructura de datos, cuya principal tarea consiste en almacenar las tramas que forman los paquetes que llegan a la unidad Egress. Como bien dice su nombre (FIFO son las siglas en inglés de *First In First Out*, 'El primero que entra es el primero que sale'), los flujos de datos que pasen por este módulo saldrán siempre en el mismo orden en que habrán entrado.

La FIFO usada en la unidad Egress está formada por 16 entradas de 135 bits cada una. El contenido de datos almacenado en cada una de las entradas es el siguiente:

– *pcie2smfu_axis_tx_tuser[21:17]*: Indica si la trama almacenada se corresponde con el final de paquete, y en caso de que así sea, la posición en la que finaliza.

– *pcie2smfu_axis_tx_tuser[14:13]*: Indica si la trama almacenada se corresponde con el inicio del paquete, así como la parte de la trama en la cual comienza.

– *pcie2smfu_axis_tx_tdata[127:0]*: Contenido de la trama de datos.

Esta información será la empleada por la unidad Egress para procesar paquetes PCIe.

La interfaz del submódulo está formada por las señales que se muestran en la tabla 3.2 junto con una breve descripción de las mismas.

Nombre	In/Out	Descripción
clk	in	Señal general de reloj.
rst_n	in	Señal general de reset a nivel bajo.
din[135:0]	in	Contenido de datos que se va a almacenar en una entrada de la FIFO.
shiftin	in	Señal que indica que va a almacenarse información en una entrada de la FIFO.
shiftout	in	Señal que indica que se va a liberar una de las entradas de la FIFO.
dout[134:0]	out	Próximo contenido de datos que va a liberarse de una de las entradas de la FIFO.
full	out	Señal que indica que la FIFO no tiene entradas libres.
empty	out	Señal que indica que la FIFO está vacía.
almost_full	out	Señal que indica que la FIFO se encuentra casi llena.
almost_empty	out	Señal que indica que la FIFO se encuentra casi vacía.
prog_empty	out	Señal que indica que la FIFO se encuentra de cerca de estar casi vacía.

Tabla 3.2: Señales de la FIFO de entrada de la unidad Egress.

El funcionamiento de la FIFO es el siguiente: en el momento en que una trama de datos válida se presente a la entrada de la unidad Egress (señal *pcie2smfu_axis_tx_tvalid* activa), se activará la señal *shiftin*, que habilita la entrada de datos en la FIFO si no está llena.

La FIFO es *writethrough*, con lo cual, el siguiente dato a salir aparece en *dout*. En el momento en que la señal *shiftout* se encuentre activa, la FIFO liberará su entrada más antigua, y, un ciclo después, en *dout* ocupará su lugar la siguiente entrada más antigua. En la figura 3.5 se muestra un ejemplo de los flujos de datos entrantes y salientes de la FIFO.

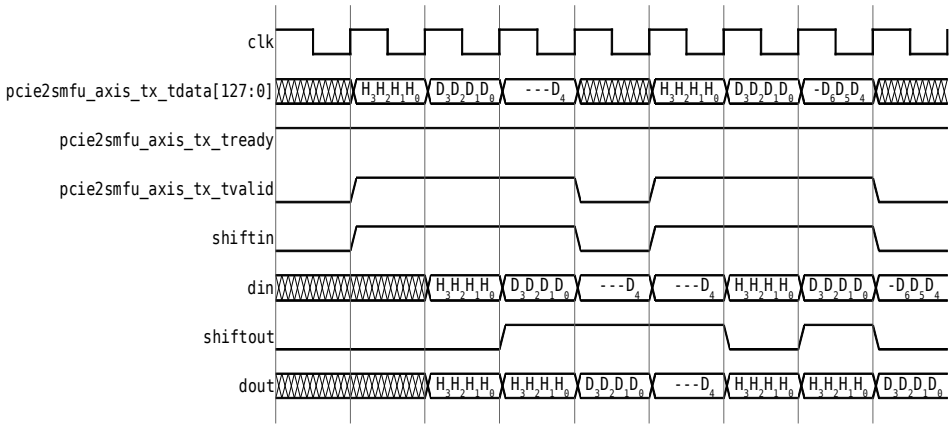

Figura 3.5: Flujo de datos entrante y saliente de la FIFO.

Con respecto al valor de las señales *almost_full*, *almost_empty* y *prog_empty*, estos vienen determinados por unos umbrales establecidos y sirven de aviso a la Egress tal y como se detalla a continuación:

– En caso de que la FIFO esté próxima a vaciarse (señales *almost_empty* y *prog_empty* activas), la unidad Egress sigue funcionando sin problemas.

– En caso de que la FIFO se quede completamente vacía (señal *empty* activa), la unidad Egress seguirá funcionando hasta que el procesamiento del paquete en curso finalice.

– Si la FIFO está próxima a llenarse (señal *almost_full* activa), la unidad Egress avisa al Core PCIe para que deje de enviarle tramas. Esto provocará que el flujo de datos se paralice hasta que la FIFO se libere. El valor que se debe establecer como umbral del “casi lleno” es el número de ciclos que pasan desde el momento en que se manda el aviso al Core PCIe hasta que la señal *pcie2smfu_axis_tx_ready* se desactiva.

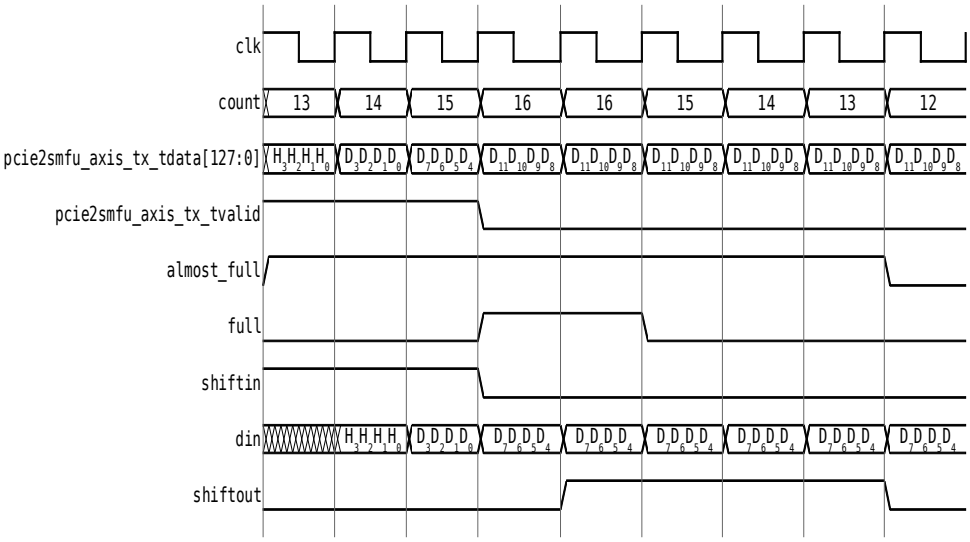

Figura 3.6: Llenado de la FIFO.

La figura 3.6 muestra un ejemplo de parada del flujo de datos. El valor definido para el umbral en este caso es 3. Es decir, cuando la FIFO alcanza las 13 entradas ocupadas (cantidad controlada por la señal *count*), la señal *almost_empty* se activa. En un máximo de 3 ciclos después (momento en el que, si se siguen recibiendo tramas a la entrada y además, no sale ninguna de la FIFO, esta se llenaría) la señal *pcie2smfu_axis_tx_ready* se desactiva. La unidad Egress habilitará la recepción de datos en el momento en que la señal *almost_empty* se desactive (es decir, cuando haya menos de 13 entradas ocupadas).

Módulos de desplazamiento

Además de la FIFO, la unidad Egress cuenta con cuatro submódulos encargados de realizar desplazamientos. Todos los módulos son idénticos, y su tarea consiste en facilitar los cálculos para obtener el identificador del nodo destino de una petición de lectura o escritura de memoria. En este caso, el valor que se quiere simplificar es la máscara.

El funcionamiento de cada módulo por separado es sencillo: a partir de un valor de entrada al módulo, proporcionado por la señal *mask[31:0]*, este devuelve por dos señales de salida si este valor tiene algún '1' (señal

onesAvail), y cuantas posiciones, empezando desde el LSB, se han recorrido hasta llegar hasta este primer '1' (señal *first_one*).

El objetivo de los dos primeros módulos es el de encontrar el primer '1' en la máscara (uno se encarga de los 32 bits de menor peso de la máscara, y el otro, de los 32 bits de mayor peso); el objetivo de los otros dos, encontrar el último '1'.

Los módulos de desplazamiento no forman parte del presente proyecto, y por ello, no serán descritos en detalle. Para más información acerca del diseño de estos módulos y de como funciona el algoritmo, ver [2] y [10].

3.1.3 Máquina de estados

Para la implementación del control de la unidad Egress específico para PCI-Express, se ha usado una **FSM** (máquina finita de estados). Esta máquina de estados, cuyo diagrama se muestra en la figura 3.5, consta de 30 estados, y su vector de salida tiene un ancho de 5 bits, el cual se refiere al estado actual de la misma. El control del estado actual es llevado a cabo por un conjunto de señales, encargadas de la transición entre unos y otros.

Después de que se active la señal de reset del sistema (a nivel bajo), la máquina de estados se sitúa en **IDLE**, donde permanecerá hasta que el AXI Switch mande datos a la unidad Egress y estos empiecen a ser almacenados en la FIFO de entrada. En caso que el paquete no sea erróneo, la FSM pasará al estado **REQ** donde el paquete entrante será procesado. Dependiendo de la posición en la que se sitúe el principio del paquete dentro de *pcie2smfu_axis_tx_tdata*, se selecciona el siguiente estado. En caso que el principio del paquete se sitúe en la parte menos significativa del bus, los estados alcanzables dependerán del tipo de paquete entrante: **P_SOT_R** (petición de escritura), **NP_SOT_R** (petición de lectura), **C_SOT_R** (respuesta) o **OTH_SOT** (otro tipo). Por otra parte, si el principio del paquete se sitúa a mitad del bus, se alcanzará el estado **SOT_M**. Puesto que en esta primera trama solo se recibe parcialmente la cabecera (llegan los primeros 64 bits de la cabecera de 96 o 128 bits), la máquina espera un ciclo en este estado para que todos los datos estén disponibles. Con toda la información del paquete, el

siguiente estado que se alcanzará será **P_SOT_M**, **NP_SOT_M** o **C_SOT_M**, en función del tipo del paquete (petición de escritura, petición de lectura o respuesta, respectivamente).

Con la cabecera completamente procesada, si el paquete recibido tiene el formato y tipo soportado (si no, el paquete será descartado en los estados **OTH_SOT** y **WAIT_EOT**, tras los cuales se volverá a **IDLE**), el siguiente estado que se alcanzará será alguno de los **SEND_NODE**. En estos estados, la unidad Egress envía un paquete a la red con el valor del identificador del nodo destino del paquete actual. Tras esto, en caso que el paquete sea una petición de lectura o escritura, los siguientes estados serán **WAIT1** y **WAIT2**, antes de que el paquete sea enviado a la red, ya que la unidad Egress necesita dos ciclos extra para calcular la dirección destino. Cuando la dirección está finalmente calculada, la máquina alcanzará el estado **CMD**. Para aquellos paquetes que sean respuestas, la transición desde el estado **SEND_NODE** a **CMD** es directa, ya que en estos paquetes no se requiere el calculo de la dirección destino (la información relativa al destino de una respuesta se obtiene de la Matching-Store).

Desde este estado, la cabecera, sin datos, se enviará a la red en una trama. En caso que el paquete no tenga datos adjuntos, la máquina de estados regresará al estado inicial (**IDLE**), donde permanecerá hasta que haya nuevos paquetes entrantes. Si, en cambio, el paquete tiene datos adjuntos, se alcanzarán los diferentes estados **DATA** (dependiendo del lugar en que empezaba la primera trama del paquete cuando entró en la unidad Egress, del tamaño de su cabecera y de su tamaño total), donde se transmitirán a la red en tramas de 128 bits. En caso que el paquete a enviar a la red sea una petición de lectura, tras el estado **CMD** se accederá a **DATA_NP**. La función de este estado es la de transmitir, en una nueva trama adjunta al paquete, el identificador del nodo origen (desde el que se envía la petición de lectura), dato necesario para las respuestas generadas debido a esta petición. Todas las tramas de datos se enviarán en orden, y empezando al principio del bus de datos *smfu2network_axis_tx_tdata*. Cuando el paquete completo ha sido enviado a la red, el último estado alcanzado será **EOT**. En este estado se mantiene activa la señal *smfu2network_axis_tx_tvalid* para garantizar la transmisión de la última trama del paquete. Con todo el paquete transmitido

correctamente a la red, la máquina de estados regresará a **IDLE**, en caso de que no haya ninguna trama almacenada en la FIFO, o a **REQ**, donde se procesará inmediatamente el siguiente paquete, en caso de que haya algo almacenado en la FIFO.

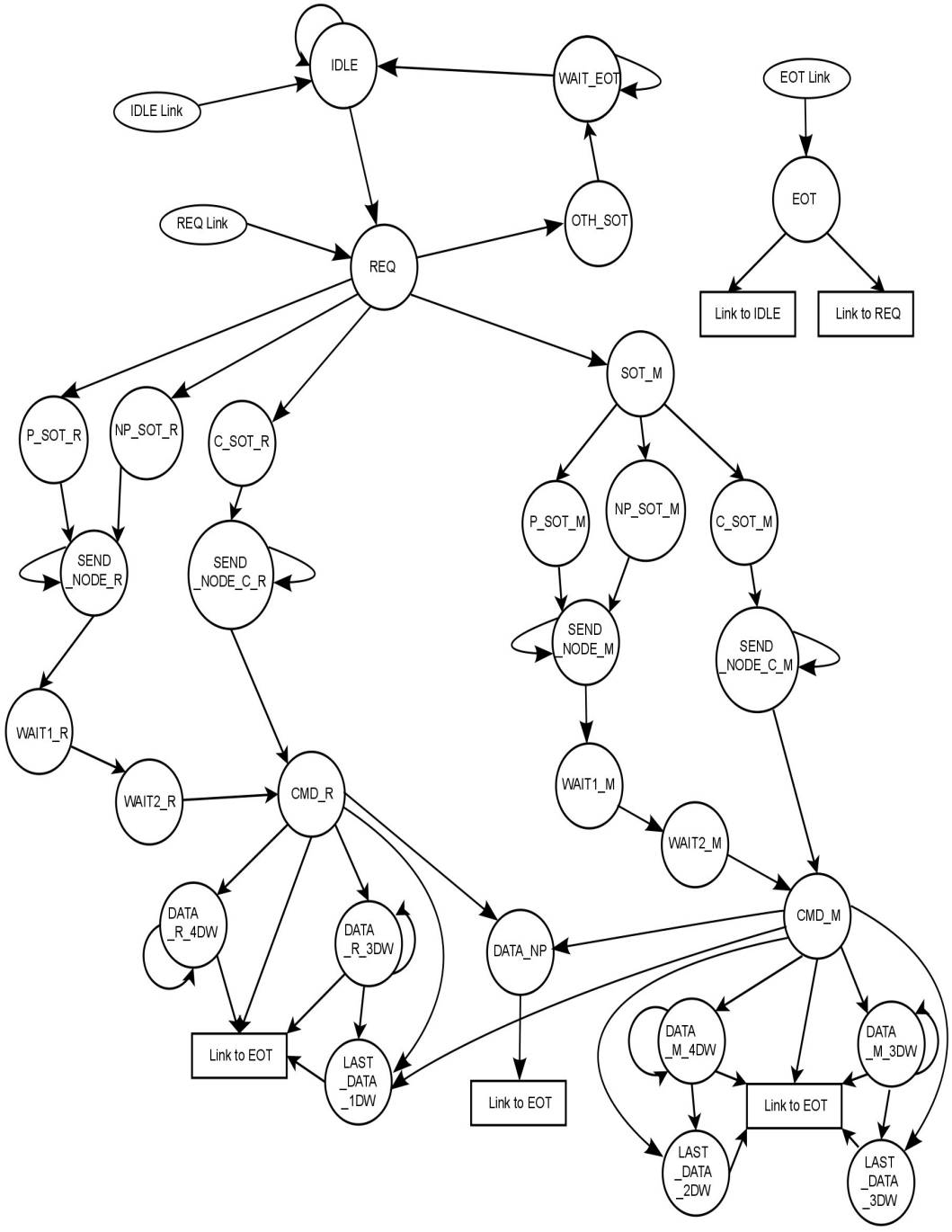

Figura 3.7: Máquina de estados de la unidad Egress.

3.2 Unidad Ingress

El principal objetivo de la unidad Ingress es el de recibir paquetes desde la red y transformarlos en paquetes PCI-Express. Principalmente, se encarga de dos tareas: en primer lugar, actúa como un completador, ocupándose de recibir respuestas y peticiones de escritura, y preparándolas para dárselas al procesador; y en segundo lugar, actúa como un respondedor, procesando las peticiones de lectura, y extrayendo de ellas la información que requerirán sus respuestas para llegar a su destino (es decir, el identificador de la petición, así como el nodo que la generó).

3.2.1 Interfaces

Al igual que en la unidad Egress, la unidad Ingress tiene diferentes señales que pueden ser organizadas atendiendo a su origen/destino. Existen 6 señales que están conectadas al *Modelo de red*, lugar por el que entran los paquetes a esta unidad; otras 5 señales se encuentran conectadas al *AXI Switch*, lugar por el que los paquetes son enviados al procesador; 4 señales conectadas a la *Matching Store*, y otras 4 señales conectadas al *Banco de Registros*. Además, existen otras señales de propósito general, necesarias para controlar el reloj del sistema, la señal de reset y el estado del módulo. En la tabla 3.3 aparecen listadas estás señales, incluyendo su polaridad y una breve descripción de las mismas.

Nombre	In/out	Descripción
clk	in	Señal general de reloj.
res_n	in	Señal general de reset activa a nivel bajo.
smfu_ingress_state[3:0]	out	Estado actual de la unidad Ingress.
cfg_completer_id[15:0]	in	Indica el identificador del completador (el mismo que el del solicitante) del nodo donde la SMFU está localizada.
Señales al Modelo de Red		
network2smfu_axis_rx_tlast	in	Indica el final de un paquete entrante en la unidad Ingress.
network2smfu_axis_rx_tdata[127:0]	in	Trama de datos entrante.
network2smfu_axi	in	Indica la presencia de datos válidos en

s_rx_tvalid		*network2smfu_axis_rx_tdata.*
network2smfu_axis_rx_tready	out	Indica que el nodo está listo para recibir datos por *network2smfu_axis_rx_tdata*.
network2smfu_axis_rx_tuser[21:17] (is_eof[4:0)]	in	Indica el final de un paquete en *network2smfu_axis_rx_tdata*. El bit de mayor peso tiene el mismo funcionamiento que *network2smfu_axis_rx_tlast* y el resto, hacen referencia a la posición del byte en el cual termina el paquete, codificado en binario.
network2smfu_axis_rx_tuser[14] (is_sof)	in	Indica el comienzo de un paquete en *network2smfu_axis_rx_tdata*.
network2smfu_axis_rx_tuser[1] (rerr_fwd)	in	Indica que el actual paquete en la unidad Ingress es erróneo.
next_vc[1:0]	in	Señala el siguiente canal virtual en el que hay información disponible, codificado en binario
next_vc_valid	in	Indica la existencia de un canal virtual válido para tener datos disponibles.
get_vc[1:0]	out	Indicates qué canal virtual es el siguiente que debe ser leído.
get_vc_valid	out	Indica la existencia de un canal virtual listo para ser leído.
Señales al AXI Switch		
smfu2pcie_axis_rx_tlast	out	Indica el final de un paquete saliente de la unidad Ingress.
smfu2pcie_axis_rx_tdata[127:0]	out	Trama de datos saliente.
smfu2pcie_axis_rx_tuser[21:17] (is_eof[4:0])	out	Indica el final de un paquete en *smfu2pcie_axis_rx_tdata*. El bit de mayor peso tiene el mismo funcionamiento que *smfu2pcie_axis_rx_tlast* y el resto, hacen referencia a la posición del byte en el cual termina el paquete, codificado en binario.
smfu2pcie_axis_rx_tuser[14] (is_sof)	out	Indica el comienzo de un paquete en *smfu2pcie_axis_rx_tdata*.
smfu2pcie_axis_rx_tuser[29:22] (port_request[7:0])	out	Indica el puerto por el cual el paquete debe abandonar la SMFU, codificado en one-hot.
smfu2pcie_axis_rx_tready[4:0]	in	Indica que el correspondiente canal está listo para empezar a mandar datos a través de *smfu2pcie_axis_tx_tdata*.
Señales a la Matching-Store		
ms_requesterID[15:0]	out	Indica el valor del identificador del solicitante en el origen de la petición de lectura entrante en la unidad Ingress.
ms_tag[7:0]	out	Indica el valor del campo *Tag* en una petición de lectura entrante en la unidad Ingress.
ms_oNodeID[15:0]	out	Indica el valor del identificador del nodo que generó la petición de lectura entrante en la unidad Ingress.
ms_ready	out	Indica al paquete entrante en la unidad Ingress como válido para escribir en la Matching-Store; es decir, que sea una petición de lectura, y que tenga todos sus datos disponibles para ser almacenados.
ms_tSrcTag[4:0]	in	Etiqueta que la MS da a las peticiones de lectura para conocer la transacción que las identifica.

Señales al Banco de Registros		
smfu_rcvd_count_ posted_incr	out	Indica la existencia de una petición de escritura entrante en la unidad Ingress.
smfu_rcvd_count_ nonposted_incr	out	Indica la existencia de una petición de lectura entrante en la unidad Ingress.
smfu_rcvd_count_ completion_incr	out	Indica la existencia de una respuesta a una petición de lectura entrante en la unidad Ingress.
smfu_rcvd_count_ others_incr	out	Indica la existencia de un paquete de otro tipo entrante en la unidad Ingress.
smfu_rcvd_count_ error_incr	out	Indica la existencia de un paquete erróneo entrante en la unidad Ingress.

Tabla 3.3: Señales de la unidad Ingress

La conexión al Modelo de Red consiste en 4 señales de entrada y una de salida. El comportamiento de estas señales es similar al explicado en el punto 3.1.1 con los paquetes entrantes en la unidad Egress (conexión AXI Switch - SMFU). La única diferencia en este caso viene en que el inicio de todos los paquetes que entren en la unidad Ingress estará al principio del bus de datos, y en ningún caso a mitad del mismo. En la figura 3.8 se ve un ejemplo de un paquete entrante a la unidad Ingress. Las señales relacionadas con el funcionamiento de los canales virtuales están incluidas en el diseño, pero no se describe su comportamiento, ya que por el momento no se usan.

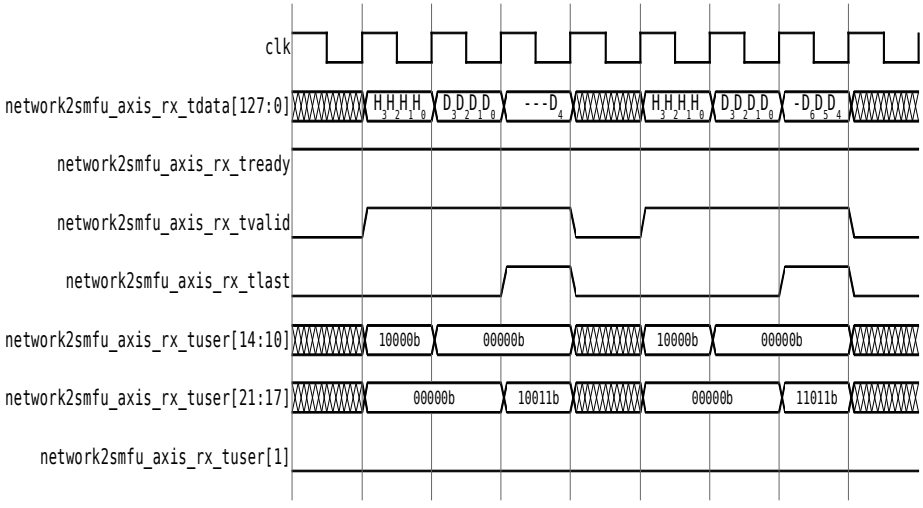

Figura 3.8: Paquete entrante en la unidad Ingress.

La conexión al AXI Switch consiste en 4 salidas y una entrada. La forma en que estas señales funcionan es análoga a las vistas en el punto 3.1.1, a la

salida de la unidad Egress (conexión SMFU con el Modelo de Red). Además, dentro de *smfu2pcie_axis_rx_tuser*, hay una señal integrada en los bits 29 al 22 llamada *port_request*[7:0], que indica a la unidad Ingress el puerto por el que el paquete saliente de esta unidad debe tomar para alcanzar el procesador (por defecto, el puerto 0). En la figura 3.9 se puede ver un ejemplo de un paquete saliente de la Ingress.

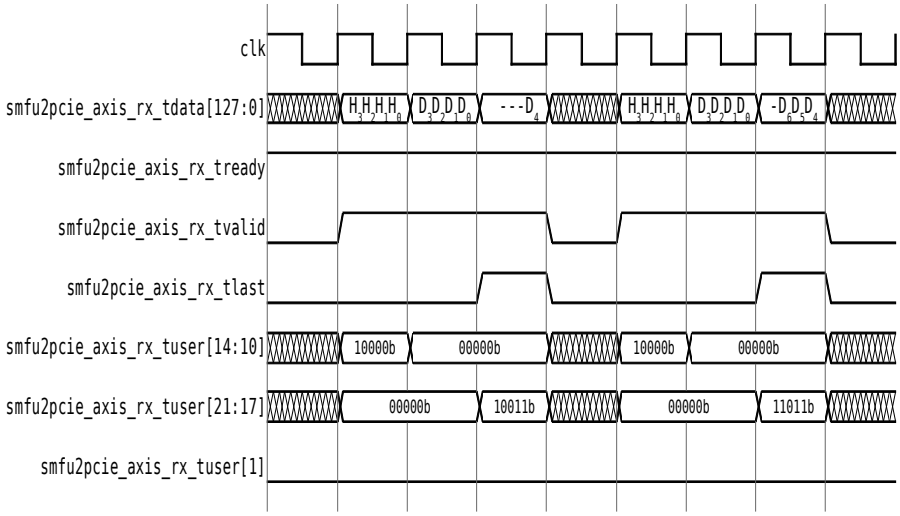

Figura 3.9: Paquete saliente de la unidad Ingress.

La conexión a la Matching-Store consiste en 4 señales de salida y una de entrada. Tres de ellas, *ms_oNodeID[15:0], ms_reqID[15:0]* y *ms_srcTag[7:0]*, contienen información acerca del origen del paquete entrante. Además la señal *ms_ready* indica si el paquete entrante en la unidad Ingress es una petición de lectura, y si toda la información necesaria para escribir en la Matching-Store esta disponible. Cuando esto ocurra, la MS enviará por la señal *ms_tSrcTag[4:0]* un valor, correspondiente al índice en que se ha almacenado toda esta información. Esta etiqueta servirá a las respuestas para obtener la información de su destino.

Por último, y conectadas al Banco de Registros, exixte un conjunto de señales llamadas *rcvd_count*_PACK_*rcvd_incr* (donde PACK = {*posted, nonposted, completion, error*}), cuyo valor será igual a '1' en caso de que el paquete entrante en la unidad Ingress sea del tipo correspondiente.

3.2.2 Submódulos

Además de las señales mencionadas y de la lógica de control, encargada del correcto funcionamiento del módulo, la unidad Ingress cuenta con un submódulo extra a su entrada, una FIFO síncrona. Este módulo es el mismo que el usado en la unidad Egress, por lo que, para más información sobre su comportamiento, ver el punto 3.1.2.

3.2.3 Máquina de estados

Para el control de la unidad Ingress, y al igual que con la unidad Egress, se ha usado una FSM. Esta máquina de estados, cuyo diagrama se muestra en la figura 3.10, consta de 10 estados, y su vector de salida tiene un ancho de 5 bits, el cual se refiere al estado actual de la misma. El control del estado actual es llevado a cabo por un conjunto de señales, encargadas de la transición entre unos y otros.

Después de que se active la señal de reset del sistema (a nivel bajo), la unidad Ingress se mantiene en el estado **IDLE**, y permanece allí hasta que se presenten datos válidos en *network2smfu_axis_rx_tdata* y estos se almacenen en la FIFO, momento en el cual se pasará a **REQ**. Este estado sirve para obtener la información del paquete entrante, para poder encaminarlo dentro de la unidad Ingress. Dependiendo del tipo de paquete entrante, se alcanzarán diferentes estados, y la máquina toma un camino diferente:

– En caso que el paquete entrante sea una petición de lectura, con una cabecera de 128 bits de tamaño, se pasará al estado **MS_DATA**. Este estado tiene una doble finalidad: preparar la cabecera de la petición y esperar a que se reciba la siguiente trama, que será la que contenga el identificador del nodo origen, es decir, el que generó la petición. Tras un ciclo, la máquina de estados pasa a **NPOSTED**. En este punto, ya esta disponible toda la información necesaria que se almacenará en la Matching-Store, además en paralelo se envía la trama que contiene la cabecera de la petición de lectura. Puesto que este tipo de peticiones no tiene datos adjuntos, tras el envío de la trama, la unidad Ingress volverá al estado inicial (**IDLE**).

– Si el paquete entrante está marcado como erróneo, se pasará al estado **ERROR**, donde permanecerá hasta que se descarte por completo. Cuando esto ocurra, la unidad Ingress regresará al estado **IDLE**.

– Si el paquete entrante es una petición de lectura, pero con un tamaño de cabecera de 96 bits, la FSM pasará directamente al estado **NPOSTED**, ya que toda la información necesaria está disponible en el primer ciclo (la trama es de 128 bits, y quedan 32 bits libres para poder almacenar otra información, si así se desea).

– Si el paquete entrante es una petición de escritura, el estado al que se llegará será **POSTED**. Dentro de este estado, pueden pasar dos cosas, dependiendo del tamaño que tenga la cabecera del paquete:

– Si su tamaño es de 128 bits, esta se reenviará en una trama al procesador (a través del switch y el Core PCIe), y se pasará al estado **DATA**, donde se irán enviando tramas de datos de 128 bits, hasta que se envíe el paquete por completo (tras lo cual, la FSM volverá al estado **IDLE**).

– En cambio, si el tamaño de cabecera es de 96 bits, se prepara una trama, pero que no se envíara todavía, y la FSM pasará al estado **DATA_FIX**. La función de este estado es la de adaptar las tramas que forman los paquetes que proceden de la red a un formato PCI-Express. En caso de que la SMFU enviase al procesador una trama con una cabecera de 96 bits, y los primeros bits de datos se enviasen en el ciclo siguiente, no se estaría respetando el formato de paquetes PCIe, explicado en la especificación del protocolo ([2]). Por ello, este estado se encargará de enviar y formar correctamente las tramas que componen un paquete hacia el procesador. Cuando todo el paquete ha sido enviado, la transacción termina, y la FSM alcanza el estado **IDLE**.

– Si el paquete entrante es una respuesta, se llegará al estado **CMPL**. En caso de que este paquete contenga datos, se llegará posteriormente al estado **DATA_FIX**, procediendo de la misma manera que con las peticiones de escritura. Si el paquete no tiene datos o ya se ha transmitido por completo, la FSM volverá a **IDLE**.

– Por último, si el paquete recibido no es de ningún tipo soportado por la SMFU, se llegará al estado **OTHERS**, en el que permanecerá hasta que quede descartado completamente. Tras esto, se volverá a **IDLE**.

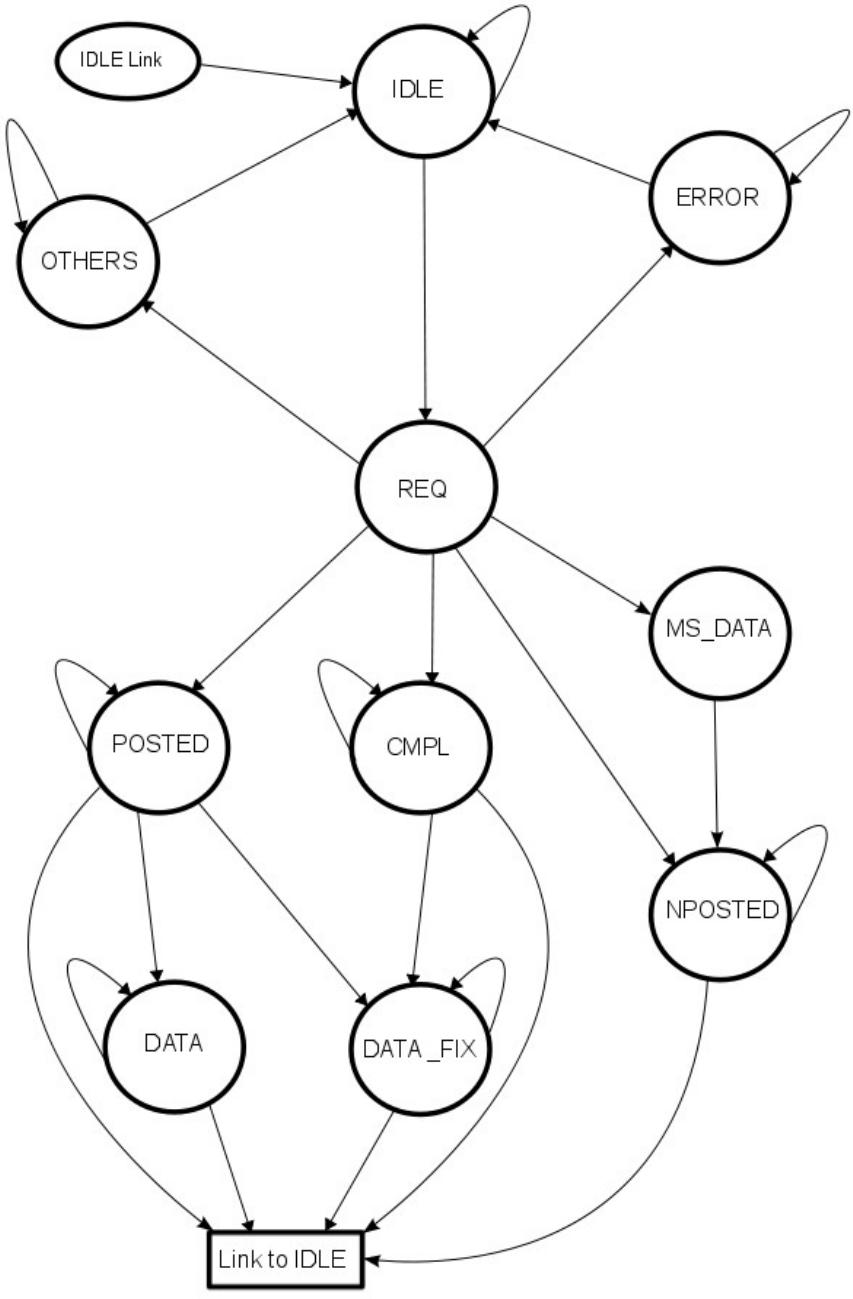

Figura 3.10: Máquina de estados de la unidad Ingress.

3.3 Matching-Store

El principal problema de usar un sistema de memoria compartida viene dado por la unicidad de las peticiones, especialmente las de lectura, ya que este tipo de peticiones necesitan respuestas. Normalmente, esto queda solucionado con etiquetas que identifican las peticiones (*srcTags*), pero, en este caso, no queda garantizado del todo. Una posible situación sería la que varios nodos envíen una petición al mismo destino simultáneamente, algo que, unido al hecho de que, como máximo, PCI-Express puede manejar 256 etiquetas diferentes (aunque por defecto, el sistema está configurado para que maneje tan solo 32 [3]), podría provocar que una respuesta no llegase a su destino.

Por ello, la existencia de la Matching-Store es necesaria. Este módulo se encarga de almacenar la información relativa al origen de las peticiones de lectura. Esta información se corresponderá con una etiqueta, con la cual la respuesta correspondiente a esta petición de lectura podrá acceder ella, sin problemas de que puedan obtenerse datos erróneos. Por ello, el problema debido a la existencia de un reducido número de etiquetas queda resuelto.

Las señales que forman la interfaz de la Matching-Store van conectadas a los otros dos módulos que tiene la SMFU. A la *unidad Egress* van conectadas tres señales (dos entradas y una salida), y a la *unidad Ingress*, cinco señales (cuatro entradas y una salida). En la tabla 3.3 se muestra el nombre y una pequeña descripción de dichas señales.

Nombre	In/Out	Descripcion
Señales a la unidad Egress		
ms_data_w[39:0]	out	Conjunto de datos que la Matching-Store da a las completions (oNodeID, requesterID y Tag), necesarios para conocer su destino.
ms_completion	in	Indica si el paquete actual en la Egress es una respuesta o no.
ms_oSrcTag[4:0]	in	Identificador único para peticiones de lectura (las que requieren una respuesta).
Señales a la unidad Ingress		
ms_requesterID[15:0]	in	Indica el valor del identificador del solicitante en el origen de la petición de lectura entrante en la unidad

		Ingress.
ms_tag[7:0]	in	Indica el valor del campo *Tag* en una petición de lectura entrante en la unidad Ingress.
ms_oNodeID[15:0]	in	Indica el valor del identificador del nodo que generó la petición de lectura entrante en la unidad Ingress.
ms_ready	in	Indica al paquete entrante en la unidad Ingress como válido para escribir en la Matching-Store; es decir, que sea una petición de lectura, y que tenga todos sus datos disponibles para ser almacenados.
ms_tSrcTag[4:0]	out	Etiqueta que la MS da a las peticiones de lectura para conocer la transacción que las identifica.

Tabla 3.4: Señales de la Matching-Store

El diseño de este módulo está basado en uno ya existente (ver [9]). Consiste en dos submódulos: una FIFO preinicializada y un bloque de memoria SRAM de doble puerto. El tamaño de la FIFO es de 256 entradas de 5 bits de ancho, y el tamaño de los datos almacenados en el módulo de memoria es de 40 bits (correspondientes al identificador del nodo origen, *Requester ID* y *Tag*). En la figura 3.11 se puede ver un esquema del módulo.

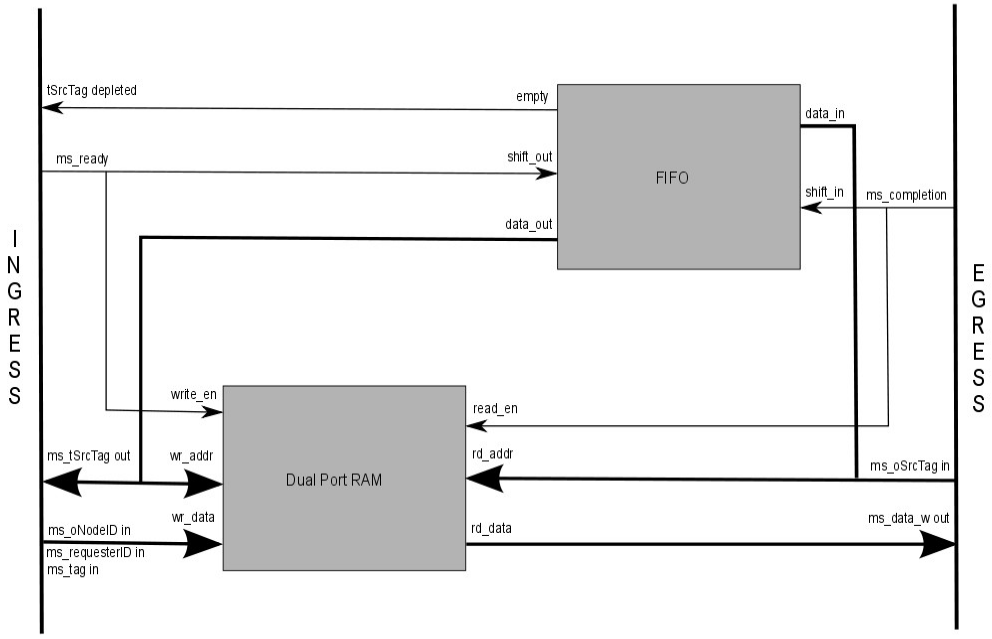

Figura 3.11: Vista esquemática de la Matching-Store.

La Matching-Store se usa inicialmente por la unidad Ingress, cuando se recibe una petición de lectura. En el momento en que todos los datos de esta petición han sido recibidos en la unidad (señal *ms_ready* activa), la Matching-Store recibirá el identificador de la transacción (*Requester ID* + *Tag*), junto con el identificador del nodo origen a través de las señales *ms_reqID[15:0]*, *ms_srcTag[7:0] y ms_oNodeID[15:0]*. Tras almacenar los datos, la MS enviará a la unidad Ingress una etiqueta para la transacción PCI-Express con la señal *ms_tSrcTag[4:0]*, que se almacenará en el campo *Tag* de la cabecera de la petición de lectura, manteniendo el resto de campos intactos.

Posteriormente, cuando una respuesta entra en la unidad Egress, se vuelve a consultar la Matching-Store activando la señal *ms_completion*. En este caso, el campo *Tag* de la cabecera de la respuesta (el cual se corresponde con el valor que la Matching-Store proporcionó a la petición de lectura), activa la MS (señal *ms_oSrcTag[4:0]*). La unidad responde enviando por la señal *ms_data_w[39:0]* la información que se almacenó al pasar la petición de lectura por la unidad Egress.

4. Validación y Resultados

El objeto del capítulo es la validación del módulo SMFU desarrollado a lo largo de este proyecto, no solo de manera individual, si no también integrado en el sistema completo, presentado en el punto 1.6.

En primer lugar, se llevarán a cabo una serie de pruebas a nivel de simulación. Estos tests consistirán en la inyección de paquetes de diferente tipo (peticiones de lectura, peticiones de escritura o respuestas, entre otros) en el entorno de simulación. Se comprobará que los paquetes entran y salen correctamente de la unidad, así como el correcto funcionamiento de las máquinas de estados de cada módulo (unidades Egress e Ingress).

Una vez los resultados de simulación han sido satisfactorios, se realizará la síntesis e implementación del proyecto completo, en la que el diseño en su totalidad se traducirá, de manera automática, en un conjunto de bits de configuración de la lógica de la FPGA. Además, los resultados de síntesis también serán presentados en este capítulo.

Por último, se realizarán una serie de pruebas reales con la tarjeta FPGA HiTech Global. La tarjeta se instala en un slot PCIe de un PC. Se controla por una serie de drivers y se ejecutan aplicaciones software de test.

4.1 Simulación

Para la realización de las pruebas de simulación del hardware desarrollado, se ha usado la herramienta ModelSim 6.6. Para lanzar las simulaciones del diseño se ha empleado el script que se ve a continuación:

```
vlib work
vmap work
vmap unisim /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/unisims_ver
vmap simprim /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/simprims_ver
vmap simprims_ver /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/simprims_ver
vmap unisims_ver /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/unisims_ver
```

```
vmap xilinxcorelib_ver /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/xilinxcorelib_ver

vmap xilinxcorelib /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/xilinxcorelib_ver

vmap secureip /opt/Xilinx/12.4/ISE_DS/ISE/verilog/mti_se/6.6d_1/lin64/secureip

vlog -work work +incdir+../.+../../example_design +define+SIMULATION +incdir+.
+../dsport+../tests /opt/Xilinx/12.4/ISE_DS/ISE/verilog/src/glbl.v -f board_ok.f

vsim -voptargs="+acc" +notimingchecks -L work -L secureip -L unisim -L simprim -L secureip -L
xilinxcorelib -t "1ps" work.board glbl
```

El script se encarga de compilar ficheros que contienen el código fuente del diseño, así como las librerías de Xilinx. La presencia de estas librerías es necesaria ya que, aunque la SMFU no incluye directamente módulos o elementos propietarios de Xilinx, si que los incluyen otros módulos del diseño.

4.1.1 Vectores de test

Los vectores de test son ficheros que controlan el entorno de depuración. Se encargan principalmente de proporcionar el flujo de paquetes que circularán por el sistema, además de inicializarlo.

De entre todas las funciones o tareas existentes, las que son de interés para los propósitos de la SMFU son aquellas que se encargan de generar paquetes con un formato reconocidos por esta unidad. Estas tareas son las siguientes:

– **TSK_TX_MEMORY_READ_32, TSK_TX_MEMORY_READ_64**: Genera una petición de lectura de memoria para direcciones de 32/64 bits.

– **TSK_TX_MEMORY_WRITE_32, TSK_TX_MEMORY_WRITE_64**: Genera una petición de escritura de memoria para direcciones de 32/64 bits.

– **TSK_TX_COMPLETION_DATA, TSK_TX_COMPLETION**: Genera un paquete de respuesta con/sin datos.

En el caso de esta última tarea, no es necesario ponerla dentro del vector de test, ya que el Core PCIe del diseño es capaz de generarlas por sí mismo cuando recibe una petición de lectura. En la cabecera de la respuesta aparecerá como *Tag* el valor que la Matching-Store dio a la petición de lectura

cuando se almacenaron los datos relativos al destino de la respuesta (o lo que es lo mismo, el origen de la petición).

Estas tareas tienen una serie de parámetros, totalmente personalizables por el usuario, que se encargan de establecer las características del paquete. De entre ellos, solo tendremos en cuenta el referido a *Length*, el cual, en el caso de las peticiones de escritura, indica el tamaño de la carga de datos que va a tener el paquete, expresado en DW (conjuntos de 32 bits).

4.2 Resultados de simulación

Con todas las consideraciones definidas en el apartado anterior, se ha procedido a realizar las siguientes simulaciones:

– Operación de lectura en memoria.

– Operación de escritura en memoria.

– Envío de varias peticiones de lectura y escritura en memoria.

Las dos primeras simulaciones están centradas en el recorrido que realiza un paquete de cada tipo (petición de lectura o de escritura en memoria) en la SMFU, tanto en la unidad Egress como en la unidad Ingress. Para el caso de la petición de lectura, se seguirá además el recorrido seguido por la respuesta generada a partir de dicha petición. Para el último caso, se va a simular un envío masivo de paquetes de cualquier tipo al sistema, forzando a la SMFU a situaciones límite tales como el llenado de las FIFOS de entrada o la inhabilitación de la señal del Core PCIe que permite la recepción de paquetes.

Respecto a los estados que forman las FSM de las unidades Egress e Ingress, estos han sido declarados como parámetros y codificados con un valor en hexadecimal (ver equivalencia en Anexo).

4.2.1 Cálculo del nodo destino y traducción de direcciones

Una de las principales funciones de las que se encarga la unidad Egress es la de hallar el destino de las peticiones de lectura y escritura que

acceden a ella. Para ello, debe de realizar dos tareas (calcular el nodo destino y traducir la dirección de memoria a la que se dirige esta petición), las cuales ya han sido explicadas en el apartado 2.2.

En primer lugar, para el cálculo del nodo destino se necesitan 4 valores:

– *Dirección de origen* (*oAddr*).

– *Dirección de inicio del origen (oStartAddr):* Dentro del espacio de direcciones del origen, es la dirección a partir de la cual empieza el espacio local compartido de memoria del nodo origen.

– *Máscara (smfu_mask):* Valor en binario que nos indica los bits de la dirección almacenada en la cabecera que se refieren al nodo destino.

– *Desplazamiento inicial (shift_count_lsb):* Valor que indica la posición del bit en la que aparece el primer '1' dentro de la máscara. El valor se obtiene a partir de dos de los cuatro módulos de desplazamiento que tiene la unidad Egress, mencionados en el apartado 3.2.2 y se calcula durante el proceso de inicialización del sistema. En la figura 4.1 se puede ver un ejemplo descrito a continuación.

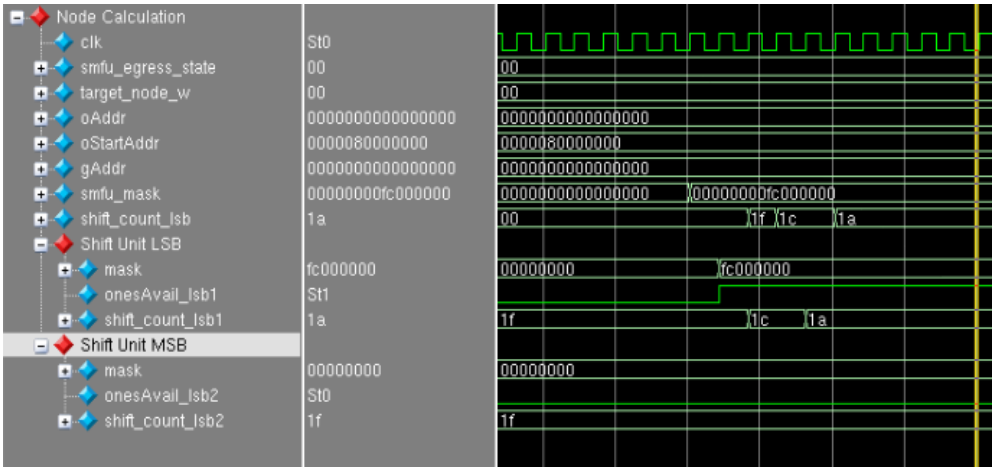

Figura 4.1: Obtencion de *shift_count_lsb* para el cálculo del nodo destino.

Partiendo del valor de *smfu_mask* 0x00000000FC000000, este se divide en dos partes, de forma que cada uno de los módulos de desplazamiento toma una: los 32 bits de menor peso para uno (0xFC000000),

y los 32 de mayor peso para otro (0x00000000). En este caso, el primer '1' aparece en la parte baja de la máscara, por lo que el valor obtenido en *shift_count_lsb1* será el valor definitivo de *shift_count_lsb* (para la máscara dada, este valor será 26). Si el primer '1' hubiese estado en el segundo módulo, el valor de *shift_count_lsb* sería el obtenido en *shift_count_lsb2* más 32 (el número de posiciones que se han recorrido adicionalmente, correspondientes a los 32 bits de menor peso de la máscara).

Una vez estos 4 valores están disponibles, se calcula el nodo destino de la petición. La fórmula $tNodeID = (gAddr \ \& \ mask) >> shift\_count$ (donde $gAddr = oAddr - oStartAddress$), es la que obtiene este valor.

Con este valor calculado, ya tan solo falta realizar la traducción de la dirección origen, para que esta se corresponda con una dirección del nodo destino. En la figura 4.2 puede verse como se resuelve a partir de un ejemplo en simulación.

El ejemplo toma los valores para *oAddr = 0x4000000020, oStartAddress = 0x80000000* y *mask = 0xFC000000;* el valor de *shift_count_lsb* será 26, y el del nodo destino, 0x20 (en decimal, 32).

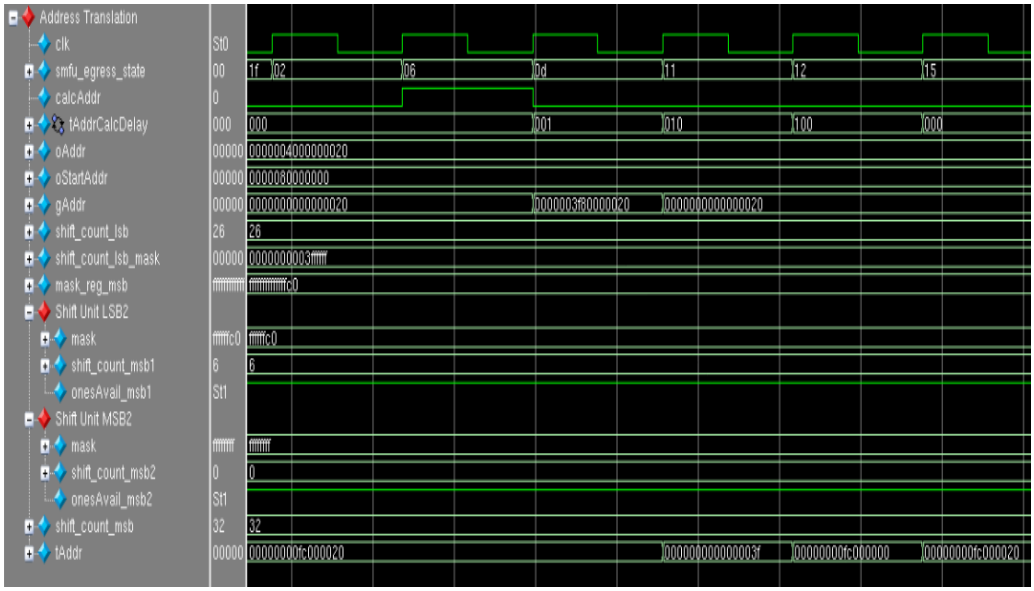

Figura 4.2: Traducción de direcciones.

El tiempo que tarda la unidad Egress en traducir la dirección es de 4

ciclos. Este proceso de cálculo se inicia cuando la unidad dispone de la dirección de memoria a la que se dirige la petición. El valor se obtiene en el momento en que la unidad se encuentra en el estado REQ (codificado en hexadecimal con el valor 0x2 en la señal *smfu_egress_state*), tras lo que activa la señal *calcAddr*, que será la que indique el comienzo de los cálculos.

Una máquina de estados controlada por la señal *tAddrCalcDelay* es la responsable de controlar el orden de las operaciones:

1. En el primer ciclo, se calcula el valor de *gAddr*, obtenido al restar el valor de la dirección origen (*oAddr)* la dirección base (*oStartAddress*).

2. En el segundo ciclo, se realizan dos operaciones. En una de ellas, se guardan los *shift_count_lsb* bits de menor peso de *gAddr*; y en la otra, son los 64 – *shift_count_msb* bits de mayor peso los almacenados.

3. En el tercer ciclo, al resultado de la segunda operación del ciclo anterior, se desplaza hacia la izquierda *shift_count_lsb* posiciones.

4. Por último, se hace una comparación entre el resultado obtenido en el ciclo anterior con el de la primera operación del segundo ciclo.

Las operaciones realizadas en los pasos 2 al 4 equivalen a realizar el cálculo entre paréntesis de la expresión $tAddr = (gAddr \ \& \sim mask) + tStartAddress$, mencionada en el punto 2.2. La suma de la dirección base del nodo destino (*tStartAddress*, en la señal *smfu_balt_hw_read_data*) se realiza en el mismo ciclo en que este dato es transmitido por la red dentro de la cabecera de la petición. La obtención de este último valor implica un retardo de un ciclo desde que se ha calculado el nodo destino.

4.2.2 Operación de lectura en memoria

Para la realización de esta simulación, se enviará una petición de lectura en memoria. El Core PCIe genera una respuesta sin datos en cuanto recibe y procesa esta petición. Para ver el camino completo que sigue la petición, se puede apreciar y está explicado en las figuras 4.3 a la 4.6.

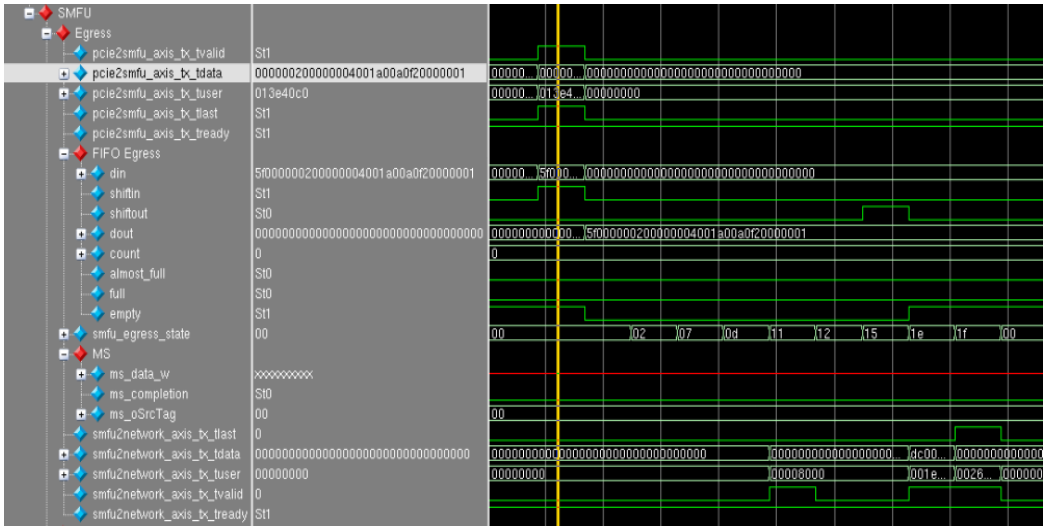

Figura 4.3: Operación de lectura: petición de lectura en unidad Egress del nodo origen.

En el momento en que las señales *pcie2smfu_axis_tx_tvalid* y *pcie2smfu_axis_tx_ready* están activas, quiere decir que hay una trama de un paquete listo para entrar en la SMFU. En este caso, la trama entrante pertenece a una petición de lectura en memoria, para direcciones de 64 bits. Su contenido, mostrado a través de la señal *pcie2smfu_axis_tx_tdata*, es el valor 000000200000004001A00A0F20000001 (en hexadecimal), y el valor de *pcie2smfu_axis_tx_tuser* es 013E40C0. Sabiendo estos datos, podemos extraer las siguientes conclusiones:

– El valor de *is_sof* (bits 14 a 10 de *pcie2smfu_axis_tx_tuser*) es 10000 (en binario), lo cual nos indica que esta trama es la primera del paquete, por lo que será la cabecera del mismo. Además, como el valor de *is_eof* (bits 21 a 17 de la misma señal) es 11111 (también binario), esto quiere decir que el paquete esta formado por esta única trama.

– Sabiendo que la trama almacenada en *pcie2smfu_axis_tx_tdata* pertenece a una cabecera, se comprueban los bits 30 al 25, los cuales hacen referencia al formato y tipo del paquete. El valor de estos es 0100000. De acuerdo a lo indicado en la especificación de PCI-Express ([3]), este valor indica que el paquete consiste en una petición de lectura, con un tamaño de cabecera de 128 bits.

– El valor de la dirección origen, indicado en los bits 127 a 64 de la

cabecera, es 4000000020. Este valor será a partir del cual se realicen los cálculos para la traducción de la dirección y la obtención del nodo destino.

Al aparecer la trama, la señal *shiftin* de la FIFO se activa, almacenándola en la primera entrada, junto con *is_eof* y los dos primeros bits de *is_sof*. En el momento en que la unidad Egress detecta que la FIFO no está vacía, es cuando la máquina de estados pasa del estado **IDLE** (codificado como 0x0) al estado **REQ** (codificado como 0x2).

A partir de este momento, es cuando la unidad Egress realiza su recorrido por la máquina de estados, de acuerdo a lo explicado en el punto 3.1.3. Los estados que recorre hasta que el paquete sale por la red son **REQ** (0x2), **NP_SOT_R** (0x7), **SEND_NODE_R** (0xD), **WAIT1_R** (0x11), **WAIT2_R** (0x12), **CMD_R** (0x15), **DATA_NP** (0x1E) y **EOT** (0x1F), para volver finalmente a **IDLE**. El estado **CMD_R** es aquel en que se compone la trama correspondiente a la cabecera, para ser enviada a la red. En el estado **DATA_NP**, se manda una trama extra, formando parte del paquete, en la cual aparece almacenado el identificador del nodo origen (el que ha generado la petición). Y por último, el estado **EOT** es el que espera a que se transmita la última trama del paquete, manteniendo activa la señal que la da como válida (*smfu2network_axis_tx_tvalid*). Desde aquí, se accederá al estado **IDLE**, ya que la FIFO está vacía y no hay más tramas para que sean retransmitidas a la red.

Con respecto a la salida de las tramas de la Egress que son generadas en los estados correspondientes, esta se puede ver en la parte baja de la figura 4.3 (bus *smfu2network_axis_tx_tdata*).

Una vez el paquete ha hecho su recorrido por la red y ha llegado al nodo destino, vuelve a pasar por la SMFU, entrando en esta ocasión por la unidad Ingress. El procedimiento seguido por las tramas que forman el paquete a la hora de almacenarse en la FIFO es el mismo que el llevado a cabo en la unidad Egress. Se explica a continuación y se ve en la figura 4.4.

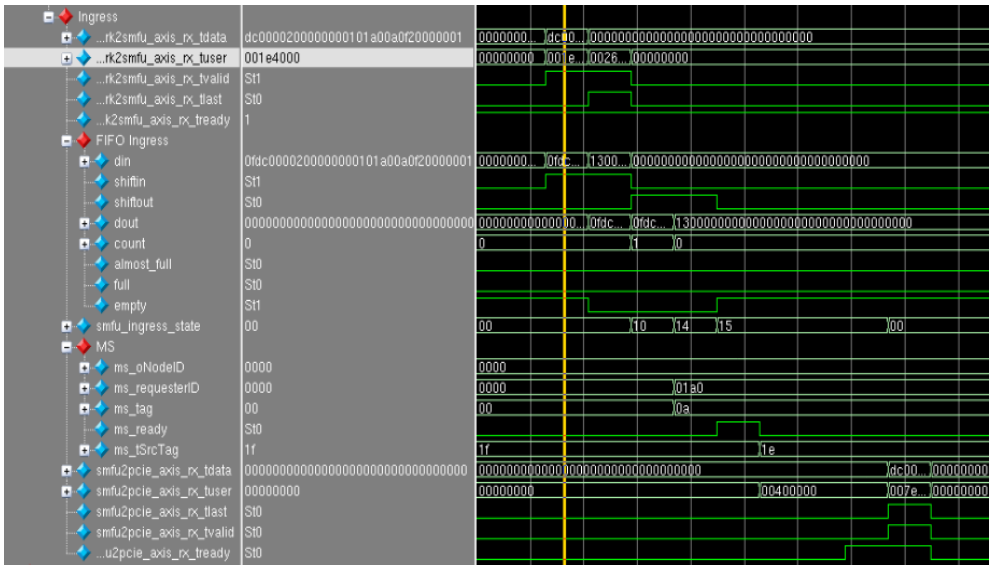

Figura 4.4: Operación de lectura: petición de lectura en unidad Ingress del nodo destino

Los estados por los que pasa la unidad Ingress en una petición de lectura son **IDLE** (0x0), **REQ** (0x10), **MS_DATA** (0x14) y **NPOSTED** (0x15), para terminar volviendo al estado inicial (**IDLE**).

Además, al tratarse de una petición de lectura en memoria, esta accederá a la Matching-Store. En el momento en que la señal *ms_ready* se encuentra activa (esto es, cuando identifica al paquete entrante como una lectura y cuando tiene todos los datos disponibles para almacenar), la unidad Ingress envía el nodo origen de la petición (en este caso, el nodo 0), junto con el *Requester ID* (0x01A0) y su correspondiente *Tag* (0x0A) a través de las señales *ms_oNodeID*, *ms_requesterID* y *ms_tag*, respectivamente. La Matching-Store, a cambio, le devuelve una etiqueta por *ms_tSrcTag*, en este caso, el valor 0x1F, como se ve en la figura 4.5, que se almacenará dentro de la cabecera.

Pasado un tiempo, y tras haber recibido y procesado la petición de lectura, el nodo destino se encarga de generar una respuesta a esta petición. Esta respuesta va dentro de un paquete sin datos, y sirve como mensaje de confirmación al nodo origen de que la petición de lectura que solicitó ha llegado a su destino. El procesamiento de un paquete de respuesta por la Egress en el nodo destino se puede ver en la figura 4.5 y se explica en detalle a continuación.

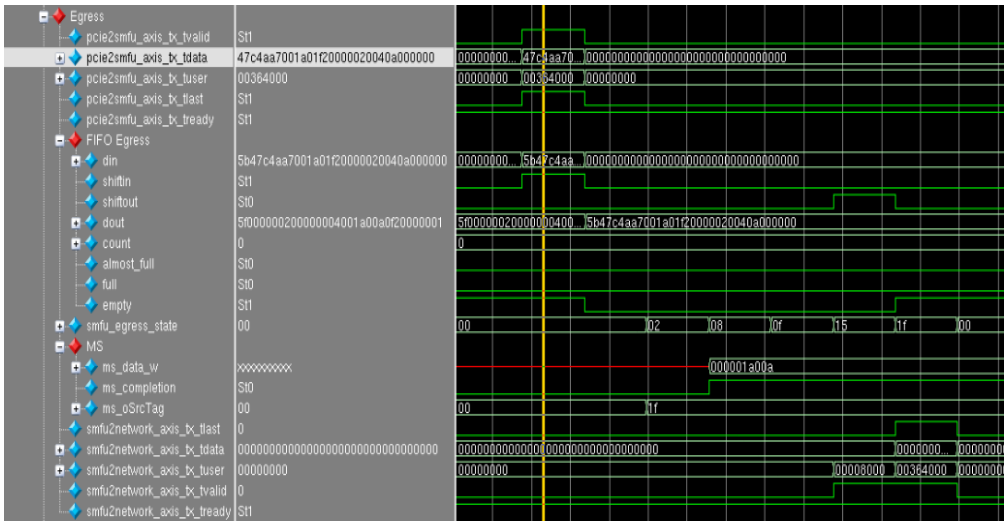

Figura 4.5: Operación de lectura: respuesta en unidad Egress del nodo destino.

Analizando el contenido de la trama entrante, vemos que el valor del bus *pcie2smfu_axis_tx_tdata* es 47C4AA7001A01F20000020040A000000, y el de *pcie2smfu_axis_tx_tuser* es 00364000, ambos en hexadecimal. Con estos datos, se puede averiguar lo siguiente:

– El valor de *is_eof* y *is_sof* es 11011 y 10000 respectivamente. Esto quiere decir que la trama es la primera y última del paquete, y que tiene un tamaño de 96 bits (los 4 bits de menor peso de *is_eof* así lo indican). Es por ello que los últimos 32 bits de la trama (que tienen el valor 0x47C4AA70) no forman parte del paquete, tratándose de información residual.

– El tipo y formato del paquete entrante, indicado en los bits 30 a 24 de la cabecera, es 0001010, el cual, según [3], identifican al paquete como una respuesta sin datos a una petición de lectura.

– La etiqueta, en los bits 79 a 72 de la cabecera, es 0x1F, y se corresponde con el valor que la Matching-Store proporcionó a la petición de lectura en la unidad Ingress.

Para obtener la información relativa a su destino, la respuesta envía este valor a la Matching-Store mediante la señal *ms_sourceTag*. Esta le devolverá por el bus *ms_data_w* toda la información (nodo origen de la petición de lectura, *requesterID* y *Tag*) cuando la señal *ms_completion* se encuentre activa (es decir, cuando reconozca una respuesta en la unidad Egress). En el

ejemplo que se ve en la figura 4.5, el valor de *ms_data_w* es 0x0001A00A, en los que los 8 primeros bits identifican el nodo origen (0x00); los 16 siguientes, al *Requester ID* (0x01A0); y los 8 de menor peso, al *Tag* (0x0A). Como se puede comprobar, estos valores son los que se corresponden con los que facilitó la petición de lectura en la unidad Ingress.

Los estados por los que pasa la máquina de estados ante una respuesta de lectura, tras el estado inicial, son **REQ** (0x2), **C_SOT_R** (0x8), **SEND_NODE_C_R** (0xE), **CMD_R** (0x15) y **EOT** (0x1F). La información extraída de la Matching-Store es obtenida durante el estado **C_SOT_R**, un ciclo antes de que parte de esta sea utilizada. En **SEND_NODE_C_R** se envía a la red la información del nodo destino de la respuesta, y en **CMD_R** y **EOT** se procede de la misma forma que con la petición de lectura. Tras **EOT**, y como no quedan más tramas almacenadas en la FIFO de entrada, se vuelve al estado **IDLE**.

Después de que la respuesta viaje por la red, esta llega a su destino, y entra en la unidad Ingress de la SMFU del nodo destino. La forma en que el paquete es procesado en esta unidad es similar al descrito con la petición de lectura, pero sin necesidad de acceder a la Matching-Store.

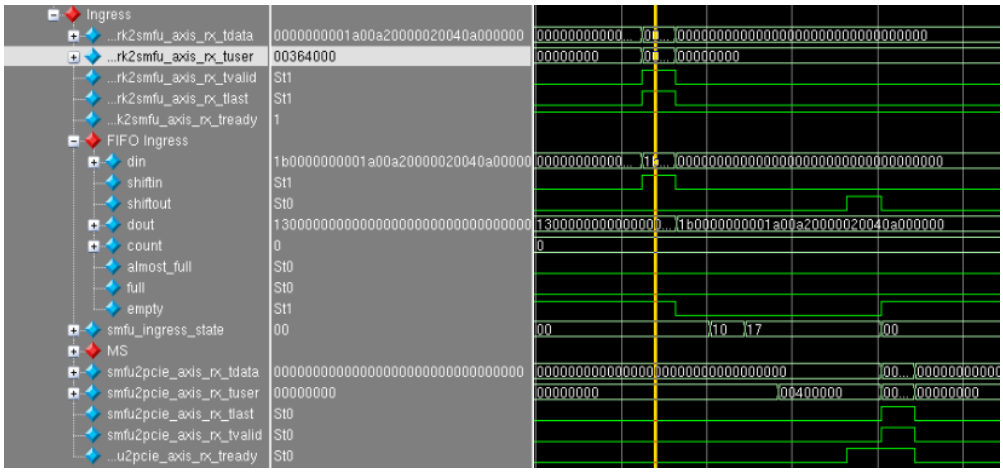

Figura 4.6: Operación de lectura: respuesta en unidad Ingress del nodo origen.

Los estados por los que pasa, tras el inicial, son **REQ** (0x10) y **CMPL** (0x17). Al tratarse de una respuesta sin datos, tras preparar la cabecera para ser enviada con dirección al nodo destino, la máquina de estados vuelve a **IDLE**.

4.2.3 Operación de escritura en memoria

Para la realización de esta simulación, se enviará una petición de escritura en memoria almacenada en un paquete con un tamaño total de 768 bits (128 bits de cabecera + 640 bits de carga de datos, ocupando un total de 6 tramas), usando direcciones de memoria de 64 bits. El camino completo que sigue esta petición está explicado resumidamente en el punto 2.3.

En el momento en que las señales *pcie2smfu_axis_tx_tvalid* y *pcie2smfu_axis_tx_tready* están activas, quiere decir que hay una trama de un paquete listo para entrar en la unidad Egress de la SMFU. Ambas señales se mantienen activas durante 6 ciclos para este tipo de paquetes. Durante este sexto ciclo, la señal *pcie2smfu_axis_tx_tlast* está activa, lo que sugiere que en el bus de datos *pcie2smfu_axis_tx_tdata* se encuentra la última trama del paquete entrante. Estas 6 tramas se han almacenado en la FIFO de entrada, junto con el indicador de inicio y final de paquete del bus *pcie2smfu_axis_tx_tuser* (bits 14 a 13 y 21 a 17, respectivamente), que serán los bits más significativos de cada entrada. El contenido de las 6 entradas de la FIFO de la simulación que han almacenado estas tramas es el mostrado en la figura 4.7.

0	4F	00000020	00000040	01A00A0F	60000014
1	0F	0C0D0E0F	08090A0B	68676665	00636261
2	0F	1C1D1E1F	18191A1B	14151617	10111213
3	0F	2C2D2E2F	28292A2B	24252627	20212223
4	0F	3C3D3E3F	38393A3B	34353637	30313233
5	1F	4E4D4C4F	48494A4B	44454647	40414243

Figura 4.7: Tramas entrantes a la entrada de la FIFO de la unidad Egress (en hexadecimal).

El valor 4F (1001111 en binario, donde los bits de mayor peso se refieren a *is_sop*, y los restantes, a *is_sof*), en la primera entrada de la FIFO, hace referencia a la primera trama del paquete (es decir, se corresponde con la cabecera). Al comprobar el formato y tipo del paquete (bits 30 a 24) de la cabecera, se puede observar que su valor (0x60, 1100000 en hexadecimal), según lo especificado en [3], identifica al paquete como una petición de escritura con una cabecera de 128 bits. Además, el valor de los 10 bits menos significativos de la cabecera identifica el tamaño de datos del paquete,

expresado en *double words* (conjuntos de 32 bits). El valor hexadecimal 0x14 equivale a 20 en decimal, dará un total de 640 bits de carga de datos.

Mientras las tramas del paquete se almacenan en la FIFO, la máquina de estados de la unidad Egress está en funcionamiento, realizando un recorrido similar al hecho por la petición de lectura: **IDLE** (0x0), **REQ** (0x2), **P_SOT_R** (0x6), **SEND_NODE_R** (0xD), **WAIT1_R** (0x11), **WAIT2_R** (0x12), **CMD_R** (0x15), **DATA_R_4DW** (0x17) y **EOT** (0x1F).

En la figura 4.8 puede verse la simulación en la que se observa el paso de la petición de escritura por la unidad Egress. En el estado **DATA_R_4DW**, la máquina permanecerá tantos ciclos como sean necesarios para transmitir a la red la carga de datos del paquete, que en este caso, son 6 ciclos. Las tramas de datos salientes solo son válidas si la señal *smfu2network_axis_tx_tready*, controlada por el sistema, está activa. En caso contrario, las tramas no podrán ser enviadas, y la máquina de estados permanecerá en el estado de transmisión (**CMD_R** o **DATA_R_4DW**) sin enviar nada a la red, hasta que esta se habilite.

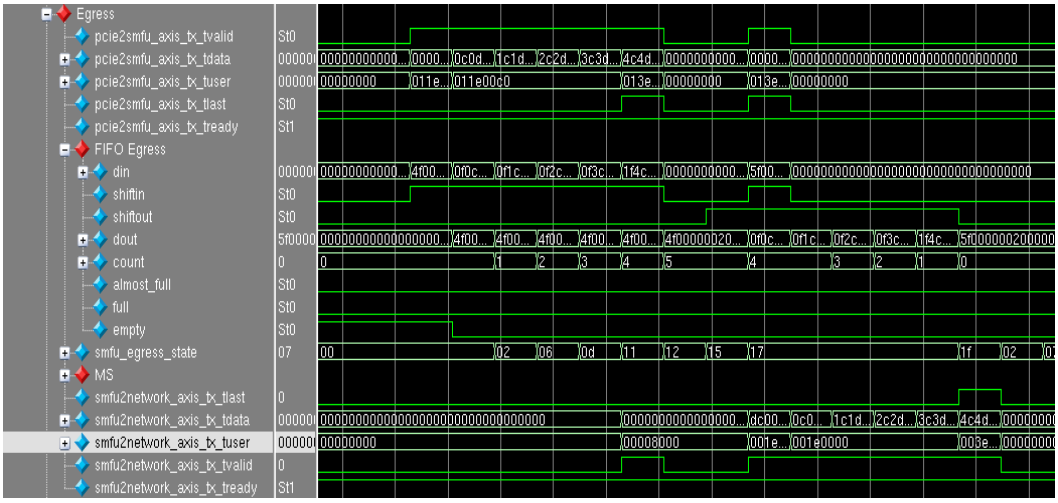

Figura 4.8: Operación de escritura: petición de escritura en unidad Egress del nodo origen.

Por último, al transmitirse la última trama, la máquina se sitúa a continuación en el estado **REQ** y no en **IDLE** (ya que la FIFO no está vacía), donde se procesará el siguiente paquete.

Después de realizar su recorrido por la red, la petición de escritura entra en la unidad Ingress del nodo destino. El almacenamiento de las tramas

entrantes en la FIFO se realiza de la misma manera que en la unidad Egress.

Tras el estado inicial, los estados por los que pasa la FSM de esta unidad son **REQ** (0x10), **POSTED** (0x13) y **DATA** (0x1F), en el que permanece hasta que todo el paquete ha sido retransmitido al switch. En la figura 4.9 se puede ver el recorrido del paquete por la unidad Ingress.

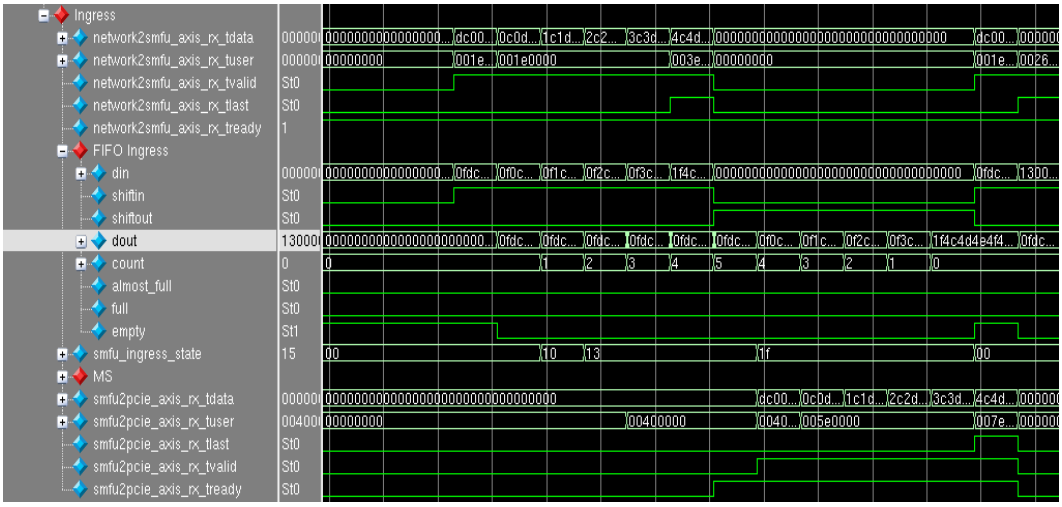

Figura 4.9: Operación de escritura: petición de escritura en unidad Ingress del nodo destino.

Además, se ha hecho una segunda simulación para una operación de escritura, enviándose en este caso una petición con una cabecera de 96 bits y una carga de datos de 544 bits. Este paquete se presenta en la unidad Egress en 5 tramas, tal y como se muestra en la figura 4.10.

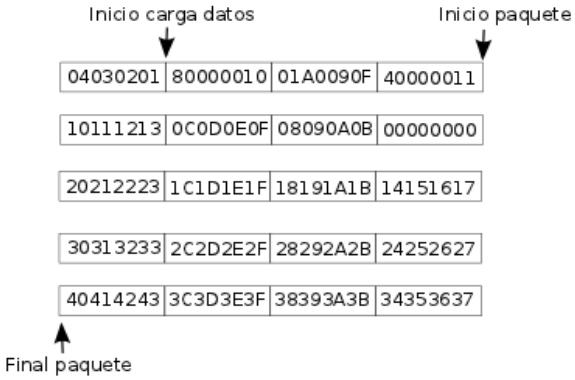

Figura 4.10: Paquete con cabecera de 96 bits entrante en la unidad Egress.

Los estados por los que pasa la unidad Egress para un paquete de estas características, son los mismos que en el caso anterior (paquete con

cabecera de 128 bits), salvo en el caso de transferencia de datos a la red. Puesto que en los paquetes de red cabecera y datos se envían en tramas diferentes, la Egress debe modificar el formato estándar de las tramas PCIe. Así, en el estado **CMD_R** (0x15), se envía por el bus de datos de salida tan solo la cabecera. Tras este estado, se llega a **DATA_R_3DW** (0x19), donde se forman y envían las tramas de datos. Por último, en caso de que no queden más tramas de datos en la FIFO pertenecientes a este paquete y, sin embargo, todavía queden datos por enviar, la Egress accede al estado **LAST_DATA_1DW** (0x1B), donde prepara la última trama de datos. En la figura 4.11 se puede ver la simulación de este supuesto, y en la figura 4.12, la forma en que salen las tramas de la Egress.

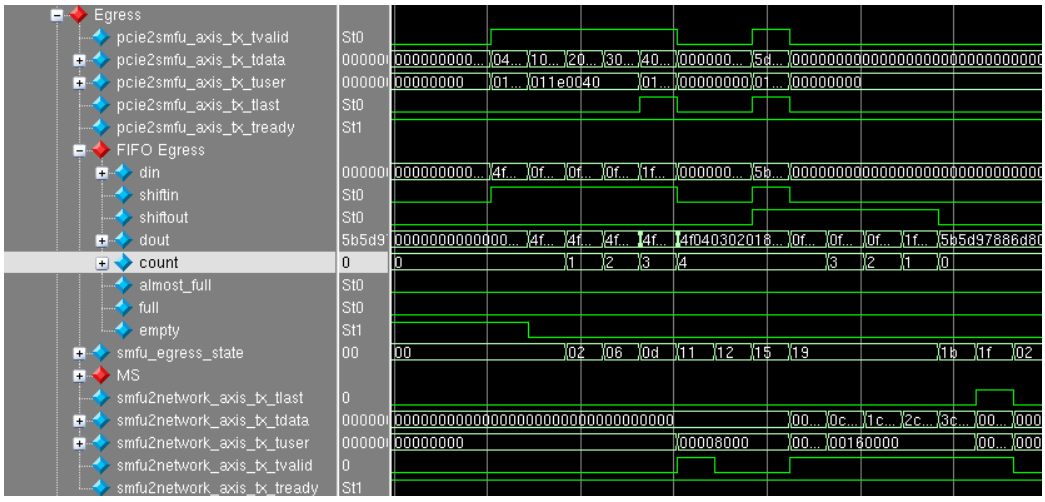

Figura 4.11: Operación de escritura: petición de escritura con cabecera de 96 bits en unidad Egress.

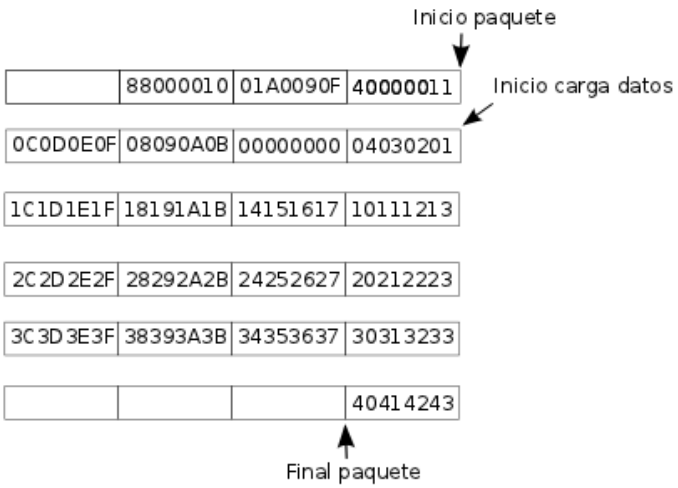

Figura 4.12: Paquete con cabecera de 96 bits saliente de la unidad Egress.

El procesado del paquete cuando alcanza la unidad Ingress es a la inversa: la figura 4.12 se correspondería con las tramas del paquete entrante en esta unidad, y la figura 4.10, con las que salen con al switch.

De igual manera, el único cambio que esto implica en la máquina de estados de la Ingress conscierne al estado en que se forman las tramas con destino al switch. Tras el estado **POSTED** (0x13), se accede al **DATA_FIX** (0x1E), donde se preparan las tramas de datos. La figura 4.13 muestra el paso de este paquete por la Ingress.

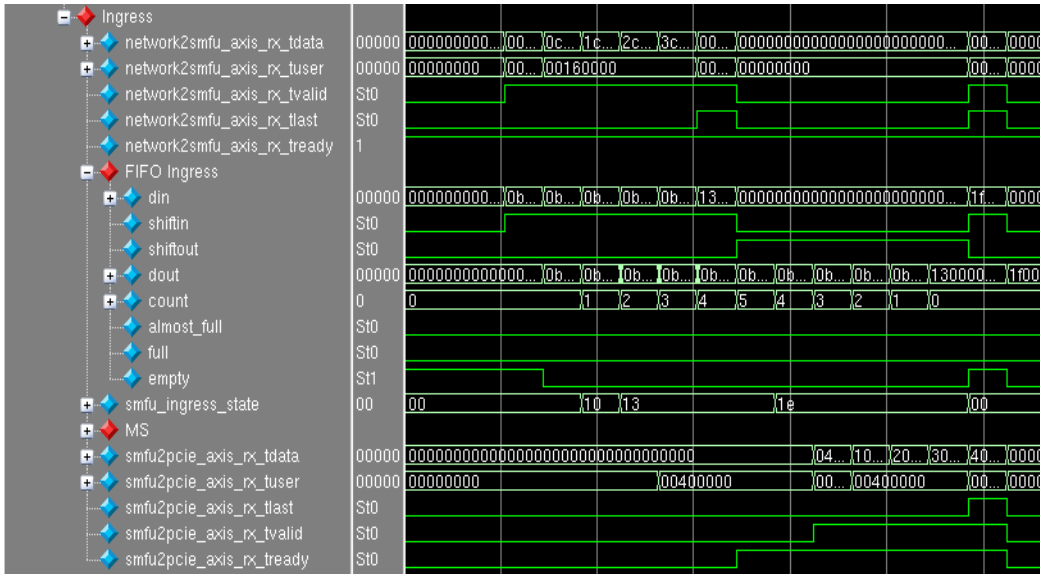

Figura 4.13: Operación de escritura: petición de escritura con cabecera de 96 bits en unidad Ingress.

4.2.4 Envío de varias peticiones de lectura y escritura en memoria

Para esta última simulación, se han tenido en cuenta los siguientes aspectos:

– Se ha realizado un envío masivo de paquetes desde el Core, entre los que hay peticiones de lectura y de escritura.

– En un momento dado, se ha forzado la deshabilitación de la señal *c_s_axis_tx_tready*, que controla la recepción de paquetes en el nodo destino.

La deshabilitación de *c_s_axis_tx_tready* provoca que el Core PCIe no

pueda recibir más peticiones. Cuando el bridge envía esta notificación al Core, también avisa a los módulos desde los que reciben las peticiones que la recepción de más paquetes no es posible. Así, en primer lugar, el switch deshabilita la señal que controla la recepción, y después, este avisa a la unidad Ingress que haga lo mismo, quedando la señal *smfu2pcie_axis_rx_tready* desactivada.

Las consecuencias de no poder mandar paquetes al switch hacen que estos no salgan de la unidad Ingress, y por tanto, que la FIFO se llene con tramas procedentes de la red. En la figura 4.14 se puede observar desde el instante en que *smfu2pcie_axis_rx_tready* se desactiva (se pone a 0) hasta que la FIFO de entrada se llena.

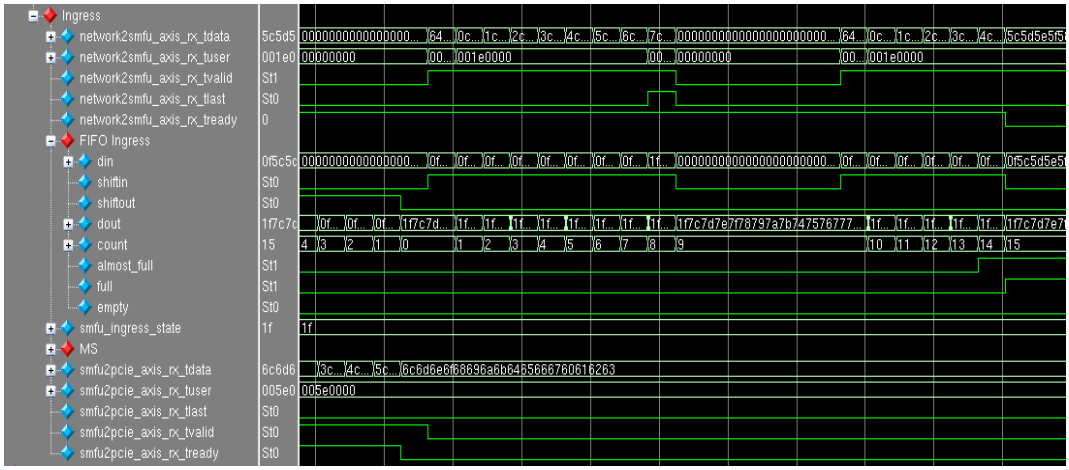

Figura 4.14: Deshabilitación de *smfu2pcie_axis_rx_tready* y llenado de la FIFO.

Al no poder la Ingress mandar ningún paquete al switch, la máquina de estados se bloquea, quedándose parada, en este caso, en el estado **DATA** (0x1F). Cuando la FIFO se encuentre próxima a llenarse (indicado por la señal *almost_full*), la Ingress envía un aviso a quien le proporciona los paquetes, en este caso, el Modelo de Red, para que pare el envío, desactivando *network2smfu_axis_rx_tready*. Esto se produce justo en el momento en que la FIFO se llena.

En esta situación, la unidad Ingress estará bloqueada hasta que el Core vuelva a aceptar la recepción de paquetes. En la figura 4.15 se puede observar el momento en que se desbloquea.

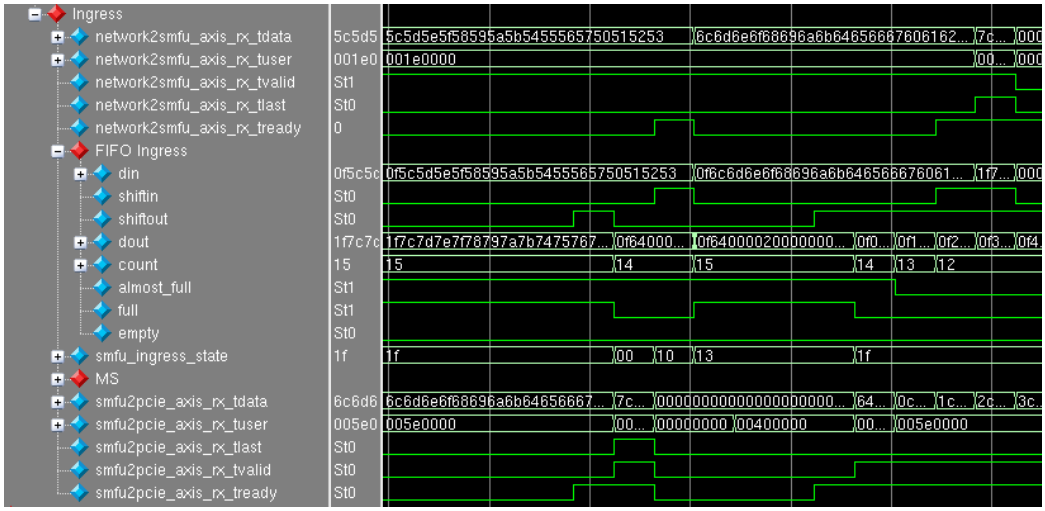

Figura 4.15: Habilitación de *smfu2pcie_axis_rx_tready.*

Tras la habilitación de la señal *smfu2network_axis_rx_tready*, el sistema vuelve a funcionar según lo esperado: la FIFO vuelve a tener la señal *shiftout* activa, permitiendo que esta libere entradas para permitir el almacenamiento de nuevas tramas entrantes; la Ingress manda al Switch las tramas restantes del paquete enviado previamente, y las señales de entrada indican que nuevas tramas están llegando a la Ingress.

Con respecto a la unidad Egress, su comportamiento ante una situación de bloqueo es similar al de la Ingress, por lo que no es necesaria su explicación.

4.3 Síntesis e implementación

Antes de probar el sistema completo en la tarjeta de HTG, con la FPGA V6 (descrita en 1.5), el diseño se debe sintetizar e implementar usando el entorno de desarrollo (ISE Development Suit 13.2 de Xilinx).

La fase de síntesis e implementación consiste en la adaptación de un diseño a un hardware concreto, en este caso, la FPGA Virtex6. Para ello, se tiene en cuenta la estructura de la FPGA.

Respecto a la lógica programable de la que se dispone en la Virtex6, esta consta de varios recursos, de entre los que se destacan dos:

– 37680 slices, formados cada uno de ellos por 4 LUTs (Look-Up Tables) de 6 entradas y 8 registros flip-flop.

– 416 bloques de memoria de 36Kbits cada uno.

La tabla 4.1 muestra cuántos de estos recursos (y el porcentaje de utilización de los mismos respecto al total) han sido usados a la hora de sintetizar la SMFU para la FPGA.

Utilización Lógica	Usados	Disponibles	Utilización
Registros	876	301440	0,29%
LUTs de 6 entradas	1817	150720	1,21%
Slices Ocupados	2006	37680	5,32%
Slices con LUT + Reg	687	2006	34,25%
Bloques de memoria	1	416	0,24%

Tabla 4.1: Utilización de recursos de la SMFU en la FPGA.

En total, la SMFU utiliza un 5,32% de los slices que tiene la FPGA, y tan solo 1 bloque de memoria de los 416 totales. La cuarta fila de la tabla hace referencia a aquellos slices en los que se está usando al menos una LUT y un registro. De los 2006 slices usados para la SMFU, el 34,25% de ellos (687) usan un elemento de cada tipo. El 65,75% restante (764 slices) solo usan uno de los dos elementos. Es por ello que el porcentaje de slices totales usados es tan grande con respecto a los registros o LUTs.

Sintetizando cada uno de los módulos por los que está formada la SMFU (unidad Egress, unidad Ingress y Matching-Store), se han obtenido los siguientes resultados:

	Egress		Ingress		Matching-Store	
Utilización Lógica	Usados	Utilización	Usados	Utilización	Usados	Utilización
Registros	977	0,32%	415	0,14%	216	0,07%
LUTs de 6 entradas	2572	1,71%	829	0,55%	273	0,18%
Slices Ocupados	2770	7,35%	872	2,31%	489	1,30%
Slices con LUT + Reg	779	28,12%	372	42,66%	0	0,00%
Bloques de memoria	0	0,00%	0	0,00%	1	0,24%

Tabla 4.2: Utilización de recursos de los submódulos de la SMFU en la FPGA.

Con los datos obtenidos, cabe destacar que la unidad Egress utiliza más registros y LUTs (y en consecuencia, más slices) que toda la SMFU al completo. Esto es debido a que el entorno de desarrollo aplica optimizaciones al diseño global.

Por último, la tabla 4.3 muestra una comparativa acerca del número de LUTs utilizadas entre el diseño actual y el diseño original para HyperTransport. El diseño original fue realizado para una Virtex4, con LUTs de 4 entradas, mientras que el diseño modificado (el realizado en este proyecto) es para una Virtex6, con LUTs de 6 entradas.

Módulo	HT Virtex4	PCIe Virtex6
Egress	890	977
Ingress	380	415
Matching-Store	285	216
Total	1555	1608
Utilización	1,84%	1,07%

Tabla 4.3: LUTs en SMFU original [1] y SMFU PCIe.

Lo único a destacar de esta comparación es el porcentaje de recursos (en este caso, LUTs) utilizadas en el diseño completo, donde el diseño actual usa un 1,07% de las LUTs de la Virtex6, por el 1,84% de la SMFU de HyperTransport en la Virtex4.

Una vez el diseño está sintetizado, se pasa a la fase de implementación. El objetivo final de esta fase consiste en obtener un diseño integrado en la FPGA de forma óptima (situando los bloques utilizados próximos e interconectándolos de forma que impliquen un mínimo retardo al sistema).

El informe resultante al realizar la ubicación y enrutamiento de los componentes en la FPGA, y considerando las restricciones de tiempo, da lugar a un período de 6,627 ns, lo que se traduce en una frecuencia de 150,9 MHz. Por ello, la frecuencia objetivo de 125 MHz ha sido alcanzada.

4.4 Prueba en entorno real

Tras realizar la síntesis e implementación del diseño, se ha generado el archivo de configuración de la FPGA con el diseño completo. Este archivo, con ayuda de la herramienta iMPACT (integrada en el ISE Development Suit 13.2 de Xilinx), ha sido programado en la FPGA.

Desde este momento, y previa instalación de unos drivers, el ordenador

en que está integrada la tarjeta, conectada a uno de sus puertos PCIe, la reconocerá como un hardware que tiene integrado un sistema de memoria compartida distribuida funcionando bajo el protocolo PCI-Express.

4.4.1 Drivers y software

Los drivers usados para el reconocimiento de este nuevo hardware en el ordenador son una adaptación de la versión usada para el diseño en HyperTransport (desarrollados por la Universidad de Heidelberg). Esta nueva versión está adaptada para PCI-Express, y se han desarrollado en paralelo junto con este proyecto [11].

Los drivers se ocupan de inicializar el sistema, establecer el valor de algunos registros (como las direcciones base y máscara) necesarios para la traducción de direcciones y de hacer llegar las peticiones (tanto de lectura como de escritura) a la SMFU.

Para probar la SMFU se ha empleado una pequeña aplicación software de test (*smfu_dump*) que escribe y/o lee en bucle desde una dirección.

```
// N_OPS = 48;
for(i=0; i<N_OPS; i=i+3)              //Peticiones de escritura
        ptr[i]=0xFAFEF1F0F8000000 + i;
for(i=0; i<N_OPS;i=i+2)               //Peticiones de lectura
        data[i]=ptr[i];
```

En este ejemplo, el valor de la dirección de memoria inicial, desde que donde empieza ptr[i], es 0x00007F0D17F9A000. Para las peticiones de lectura, data[i] almacenará el dato de la dirección leída.

4.4.2 Resultados

Para verificar que la SMFU procesa correctamente las peticiones transmitidas/recibidas, se ha usado la herramienta ChipScope (integrada en la plataforma de desarrollo de Xilinx). Con esta herramienta, se pueden tomar muestras del comportamiento de las señales dentro de la FPGA mientras que las aplicaciones de test están ejecutándose. Las señales que se monitorizan se

muestran con la señal de reloj de la SMFU. Las muestras son capturadas cuando un *trigger* predefinido se activa.

En este caso, el *trigger* que se ha definido es el paso de la señal *m_axis_rx_tvalid* a 1. La activación a nivel alto de esta señal indica que la trama de una petición sale del Core PCIe con destino a la SMFU (pasando esta previamente por el Bridge y el Switch).

En la figura 4.16 se puede ver el recorrido de las primeras peticiones (en este caso, escrituras) entrando en la unidad Egress. Todas estas peticiones están formadas por dos tramas: una cabecera de 128 bits y una trama de datos. El contenido de los datos ocupa 64 bits, y su valor es 0xF0F1FEFA000000F8. Según lo indicado en el punto 4.3.1, el contenido de datos de esta petición debería ser 0xFAFEF1F0F8000000. Esto es debido a que los bytes en tramas PCIe, dentro de cada *double word* (conjunto de 32 bits) se escriben de forma inversa a su orden normal.

Figura 4.16: Peticiones de escritura entrantes en unidad Egress.

En lo que respecta a la máquina de estados, se puede observar que los estados por los que pasa son los mismos que los vistos a nivel de simulación para las peticiones de escritura: 0x00 (**IDLE**), 0x02 (**REQ**), 0x06 (**P_SOT_R**), 0xD (**SEND_NODE_R**), 0x11 (**WAIT1_R**), 0x12 (**WAIT2_R**), 0x15 (**CMD_R**), 0x17 (**DATA_R_4DW**) y 0x1F (**EOT**), para volver al estado 0x2 (**REQ**), ya que

la FIFO no está vacía, donde se comienza a procesar el siguiente paquete.

La figura 4.17 muestra la salida de las tramas por la red, así como el proceso seguido en el cálculo del nodo destino y la traducción de direcciones.

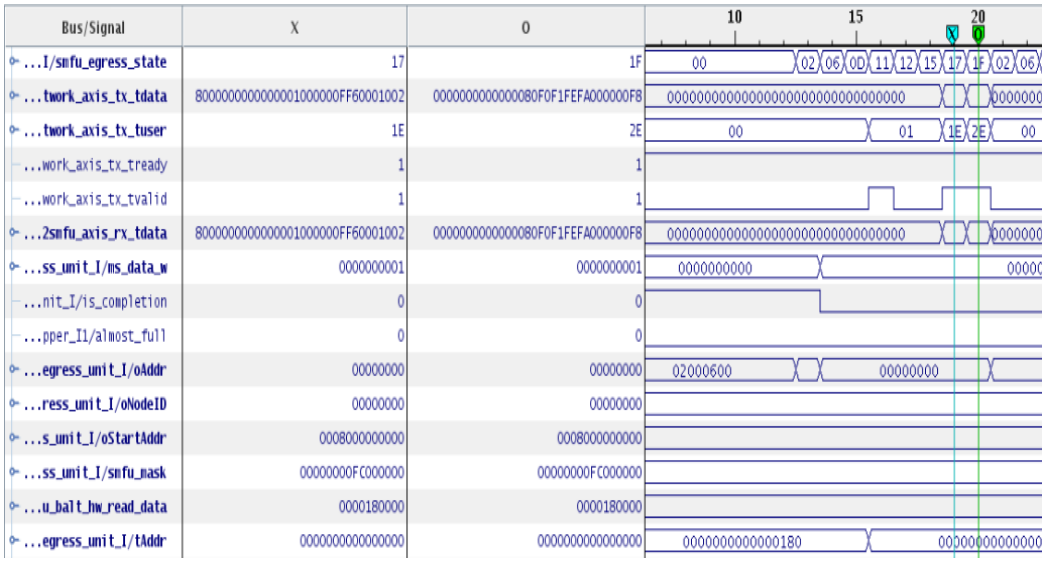

Figura 4.17: Peticiones de escritura saliendo de la unidad Egress.

Tras recorrer la red, la petición llega a la unidad Ingress, donde es procesada. En la figura 4.18 se puede ver el paso completo de la petición por la unidad, donde se puede observar que la petición sale correctamente de la Ingress con destino al Switch, los valores de la señal *smfu2pcie_axis_rx_tdata* son los correctos, y que su recorrido por la máquina de estados es el mismo que en la simulación.

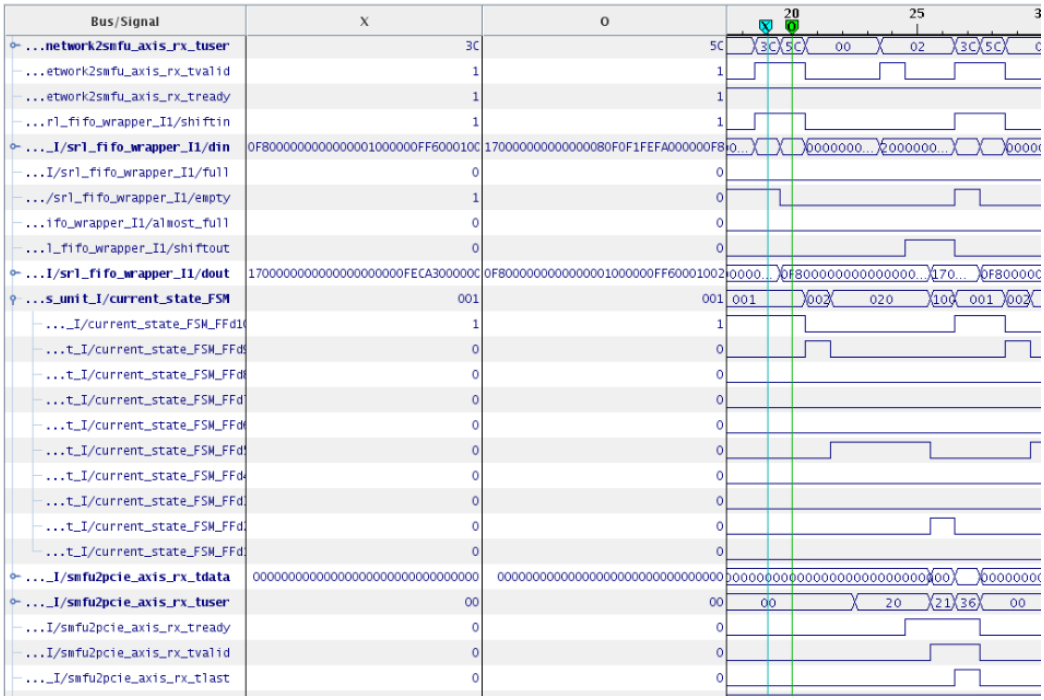

Figura 4.18: Peticiones de escritura en unidad ingress.

En las figuras anteriores se puede notar que, a diferencia de las simulaciones, la máquina de estados se encuentra codificada en *one-hot*, es decir, cada estado está representado con una señal, el cual lo identifica cuando está activa a nivel alto. Esto se debe a las opciones de síntesis seleccionadas.

Una vez finalizadas las peticiones de escritura, el contenido de la memoria ha sido leído e impreso en pantalla desde el software, comprobando así que se han realizado correctamente.

Después de que todas las peticiones de escritura han sido procesadas, llegan las peticiones de lectura. En las figura 4.19 se puede observar el paso de una de estas peticiones por la unidad Egress, desde el momento en que empieza a ser procesada hasta que sale con destino al Modelo de Red. Se puede observar que el paso por la máquina de estados es el mismo que el visto en simulación.

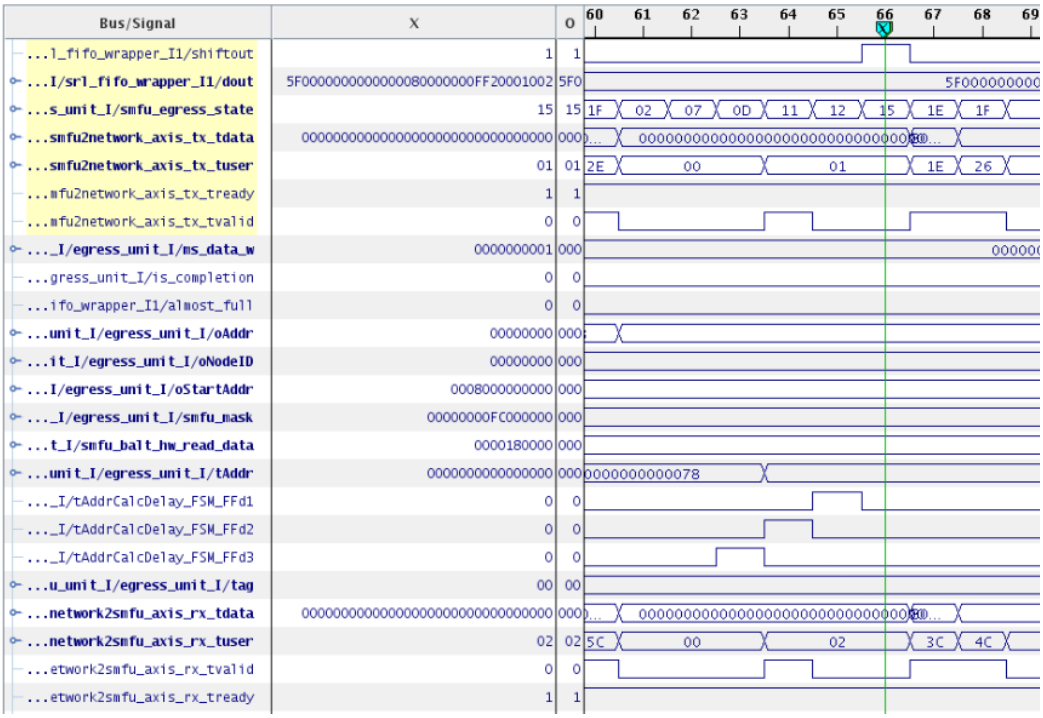

Figura 4.19: Petición de lectura en unidad Egress.

Tras su paso por la red, la petición de lectura accede a la unidad Ingress, donde es procesada. En el momento en que esta unidad detecta la petición de lectura, se activa el mecanismo que almacena información en la Matching-Store (señal *ms_ready* activa a nivel alto). Las señales *ms_oNodeID*, *ms_requesterID* y *ms_tag* toman el valor de estos campos almacenados en el paquete (*ms_nodeID*, en la segunda trama del paquete, y *ms_requesterID* y *ms_tag*, en la cabecera). Tras el almacenamiento, la MS devuelve por la señal *ms_tSrcTag* la etiqueta, cuyo destino será la respuesta generada a partir de esta petición.

En la figura 4.20, se puede observar el paso de la petición de lectura por la unidad Ingress, hasta que sale de ella con destino al Switch.

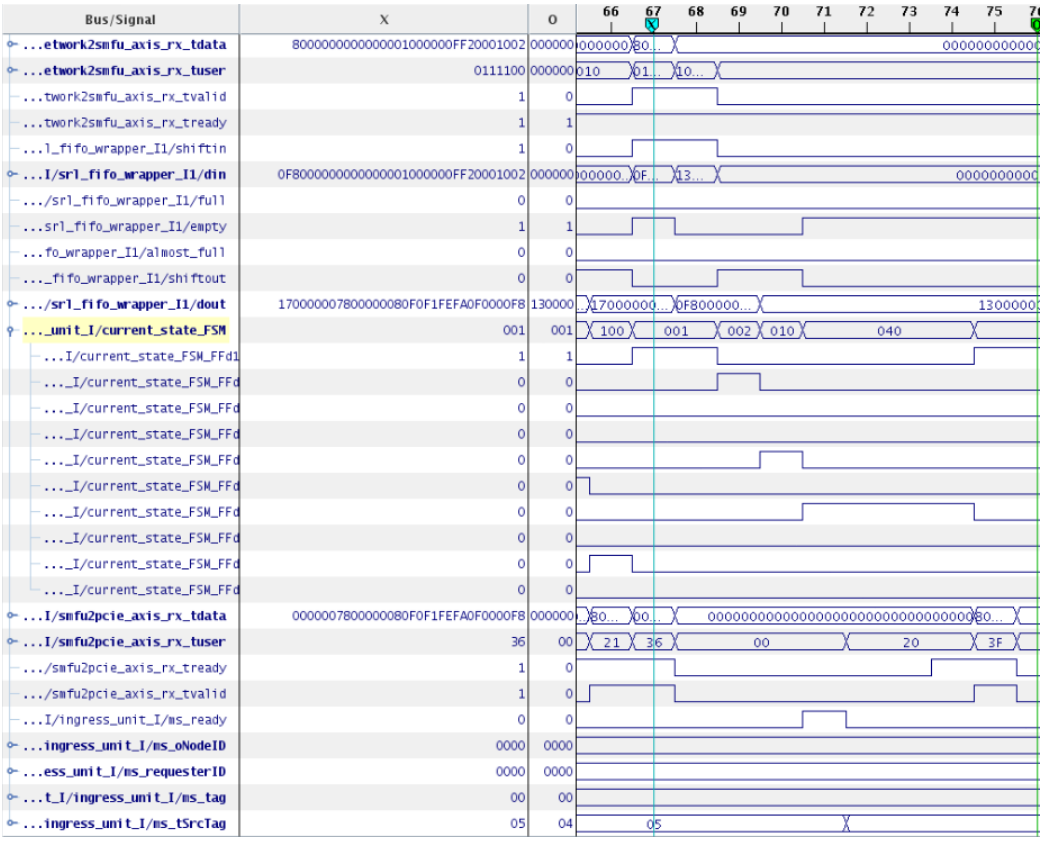

Figura 4.20: Petición de lectura en unidad Ingress.

Después de que la petición de lectura se ha procesado en su destino, este genera una respuesta. La respuesta generada en este ejemplo tiene carga de datos. En la figura 4.21 se puede ver la entrada de esta respuesta en la unidad Egress, así como el contenido de este paquete.

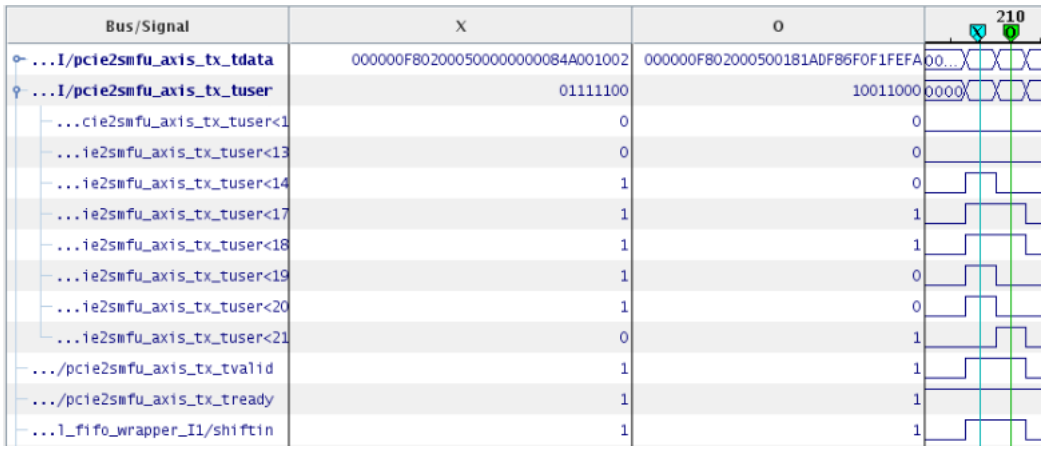

Figura 4.21: Respuesta entrante en unidad Egress.

El contenido de la respuesta se ha almacenado en dos tramas. La primera de ellas contiene la cabecera y los 32 primeros bits de la parte de datos; y en la segunda, el resto de datos, en este caso, 32 bits (el resto de la trama no interesa, ya que es información residual que no pertenece a ninguna otra trama). El contenido de estas tramas se restructurará a la salida, ya que los paquetes que salen desde la Egress a la red separan la parte de control (cabecera) de la parte de datos (ver ejemplo en imágenes 4.10 y 4.12).

Además, al tratarse de una respuesta, esta accederá a la Matching-Store a obtener los datos necesarios para que conozca su destino. El valor que la respuesta proporciona a la MS es el obtenido por la petición de lectura previa en *ms_tSrcTag* (0x05), y la MS devuelve a cambio toda la información (*oNodeID*, *requesterID* y *Tag*) en la señal *ms_data_w*.

La máquina de estados de la Egress pasa por los mismos estados que los vistos en simulación para la respuesta. Tras llegar al estado 0x15 (**CMD_R**), se llega al estado 0x19 (**DATA_R_3DW**), donde se envían las tramas de datos. En la figura 4.22 se ve el paso de la respuesta por la unidad Egress, hasta que la abandona con destino al Modelo de Red.

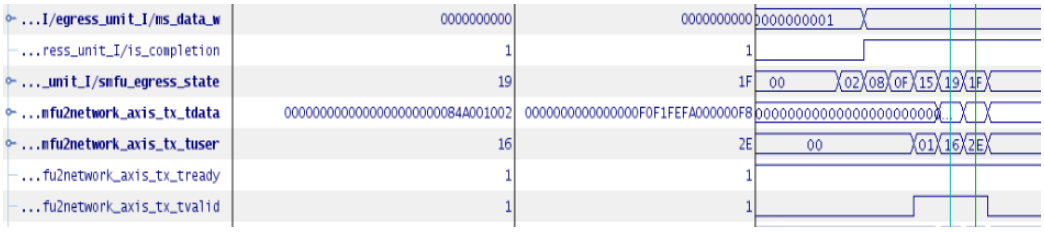

Figura 4.22: Respuesta saliendo de la unidad Egress.

Después de su paso por la red, la respuesta llega a la unidad Ingress, donde es procesada para ser enviada al Switch. El recorrido de esta, el cual ya se ha visto y ha sido explicado a nivel de simulación, puede observarse en la figura 4.23.

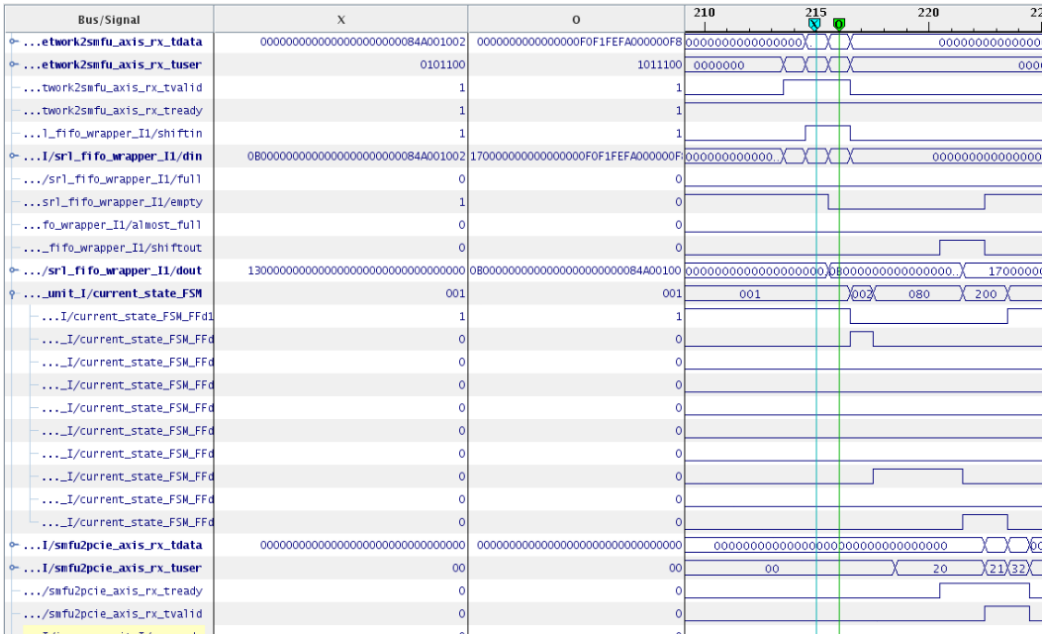

Figura 4.23: Respuesta en la unidad Ingress.

Una vez todas las peticiones (y sus correspondientes respuestas) han sido procesadas, se imprimen en pantalla para su análisis. La comprobación de los resultados de las lecturas con respecto a los datos de memoria ha demostrado que todos los paquetes han sido procesados correctamente y han llegado a su destino.

5. Conclusiones y líneas futuras

Durante este proyecto, se ha realizado la adaptación de un diseño existente de memoria compartida distribuida del protocolo de comunicaciones HyperTransport (desarrollado por la Universidad de Heidelberg, Alemania) a PCI-Express. Más concretamente, la tarea en que se ha centrado este proyecto ha consistido en el rediseño y desarrollo de su módulo principal: la SMFU.

En primer lugar, partiendo del código de la SMFU para HyperTransport, diseñado en el lenguaje de descripción hardware Verilog, se ha adaptado de forma que el módulo resultante tuviese capacidad para procesar paquetes PCI-Express de tres tipos: peticiones de lectura, peticiones de escritura y respuestas. El resultado del nuevo diseño ha supuesto una serie de cambios con respecto al diseño original (como el flujo de datos que entra y sale del módulo, descrito en detalle en el apartado 2.5), que no han mermado la principal característica de la que se partía: acceder a memoria remota con una latencia muy baja.

A su vez, la fase de simulación ha sido una de las tareas más importantes a lo largo del desarrollo de la SMFU. La continua verificación del módulo principal, así como de los submódulos que lo forman (especialmente las unidades Egress e Ingress), ha servido para descubrir todos los fallos que en la fase de diseño era imposible encontrar, sin realizar una prueba que los forzara a esta situación. Por ello, los vectores de test (descritos en el apartado 4.1) han sido un elemento necesario a lo largo de este proyecto.

Con el módulo diseñado completamente, se ha efectuado la síntesis e implementación. Como se ha podido ver en el apartado 4.3, la SMFU consume apenas un 1% de los recursos de la Virtex6. Además, todas las limitaciones de tiempo se han cumplido, consiguiendo alcanzar la frecuencia objetivo de 125 MHz.

Partiendo del módulo implementado, se ha generado un archivo de configuración para la FPGA, con el que el diseño completo ha sido probado en un entorno real. Partiendo de unos drivers y un software ya desarrollados, se han realizado una nueva serie de pruebas, las cuales han sido determinantes

para identificar fallos en el diseño que, a nivel de simulación, eran imposibles de detectar.

Con todo esto, los objetivos planteados en el punto 1.1 de este proyecto han sido alcanzados satisfactoriamente.

De cara a las líneas futuras del proyecto, podrían consistir en una serie de desarrollos y ampliaciones del sistema. Entre las cuales cabe destacar que hay una serie de funcionalidades que han sido consideradas en la fase de diseño, pero que por el momento no se han implementado, tales como la gestión de los canales virtuales por los que va a transcurrir el flujo de datos. También se podrían realizar una serie de ampliaciones, como la sugerida en el punto 2.4 (*Identificador de transacciones en sistemas de memoria compartida distribuida*), en la que se propone la ampliación del número de transacciones simultáneas en la SMFU.

6. Referencias

[1] *http://www.gap.upv.es/index.php/home/resources.html*

[2] **Janusz Schinke**; *Design and Verification of a Low Latency Functional Unit for Direct Access to Remote Memory;* Grupo de Arquitectura de Computadores en el Departamento de Ingeniería de Computadores, Universidad de Heidelberg, 2009;

[3] **PCI-SIG**; *PCI-Express® 2.0 Base Specification Revision 0.9*; 11 de Septiembre de 2006;

[4] **HiTech Global**; *HTG-V6-PCIE-xxxx User Manual Version 3.3*; Febrero 2011;

[5] **Xilinx Inc.**; *Virtex-6 FPGA Integrated Block for PCI-Express, User Guide*; 21 de Septiembre de 2010, http://www.xilinx.com/support/documentation/user_guides/v6_pcie_ug517.pdf;

[6] **ARM**; *AMBA® 4 AXI4-Stream Protocol, Specification Version 1.0;* 5 de Marzo de 2010, https://silver.arm.com/download/download.tm?pv=107410;

[7] **Steven L. Scott**; *Synchronization and Communication in the T3E Multiprocessor;* En actas de la VII Edición de la Conferencia Internacional ASPLOS (Architectural Support For Programming Languages and Operating Systems), Cambridge, Massachusetts, Estados Unidos, 1 al 4 de octubre de 1996;

[8] **Holger Fröning, Heiner Litz**; *Efficient Hardware Support for the Partitioned Global Address Space;* Grupo de Arquitectura de Computadores en el Departamento de Ingeniería de Computadores, Universidad de Heidelberg, 2010;

[9] **Timo Reubold**; *Design, Implementation and Verification of a PCI-Express to HyperTransport Protocol Bridge*; Grupo de Arquitectura de Computadores en el Departamento de Ingeniería de Computadores, Universidad de Mannheim, 2008;

[10] **Benjamin Geib**; *Improving and Extending a Crossbar Design for ASIC and FPGA Implementation*; Grupo de Arquitectura de Computadores en el Departamento de Ingeniería de Computadores, Universidad de Mannheim, 2007;

[11] **Ferrán Pérez**; *Diseño de un bloque hardware de interconexión para un supercomputador*; Grupo de Arquitecturas Paralelas del Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, 2011;

ANEXO: Codificación numérica de los estados de las FSM.

En la tabla A.1 se puede ver el nombre de cada uno de los estados de las dos FSM existentes en la SMFU, así como el módulo al que pertenecen y su codificación en binario y hexadecimal.

Nombre	Codificación binaria	Codificación hexadecimal
Estados de la unidad Egress		
IDLE	00000	00
REQ	00010	02
OTH_SOT	00100	04
WAIT_EOT	00101	05
P_SOT_R	00110	06
NP_SOT_R	00111	07
C_SOT_R	01000	08
SOT_M	01001	09
P_SOT_M	01010	0A
NP_SOT_M	01011	0B
C_SOT_M	01100	0C
SEND_NODE_R	01101	0D
SEND_NODE_M	01110	0E
SEND_NODE_C_R	01111	0F
SEND_NODE_C_M	10000	10
WAIT1_R	10001	11
WAIT2_R	10010	12
WAIT1_M	10011	13
WAIT2_M	10100	14

CMD_R	10101	15
CMD_M	10110	16
DATA_NP	11110	1E
DATA_R_4DW	10111	17
DATA_R_3DW	11001	19
DATA_M_4DW	11000	18
DATA_M_3DW	11010	1A
LAST_DATA_R_3DW	11011	1B
LAST_DATA_M_4DW	11100	1C
LAST_DATA_M_3DW	11101	1D
EOT	11111	1F
Estados de la unidad Ingress		
IDLE	00000	00
REQ	10000	10
NONPOSTED	10101	15
POSTED	10011	13
COMPLETION	10111	17
OTHERS	11001	19
MS_DATA	10100	14
DATA	11111	1F
DATA_FIX	11110	1E
ERROR	00001	01

Tabla A.1: Codificación de estados de las unidades Egress e Ingress.